두 남자의 Visual Studio 2010 TDD 이야기

Visual Studio Korea Team Blog http://vsts2010.net

작성자 : 강보람 ITFlow 선임 컨설턴트(netscout82@naver.com)

박세식 유니위스 대리(xodim@naver.com)

감수자 : 강성재 부장 한국 마이크로소프트 개발자 및 플랫폼 사업 총괄 부서 (Win2kin2@hotmail.com)

Contents

프롤로그 3

세식의 기록 4

다시 현실로 돌아와서 6

TDD란? 7

TDD로 얻을 수 있는 장점 12

TDD를 위한 최적의 IDE, VISUAL STUDIO 2010 18

간단한 TDD 예제 24

프로젝트 및 테스트 프로젝트 생성 25

테스트 실패하게 만들기 26

테스트를 통과하도록 코드 수정 29

세식의 기록 2 32

12일 후 33

시나리오 1 : ASP.NET 웹 폼에서 35

요구사항 수집하기 35

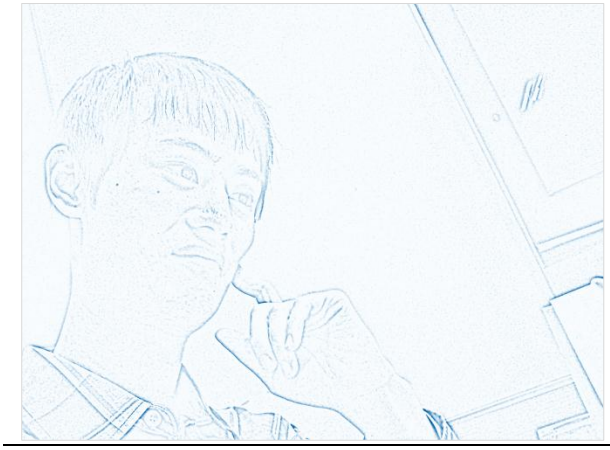
데이터 액세스 레이어 작성 38

시나리오 2 : ASP.NET MVC 40

시나리오 3 : ASP.NET MVC + TDD 47

에필로그 65

이 문서는 친구인 두 개발자의 대화 속에서 오간 내용을 담고 약간의 재구성을 통해서 완성했습니다. 그 개발자들을 소개해드릴까요?(여기에 등장하는 회사의 이니셜이나 프로젝트 명, 날짜 등은 실제 회사와 전혀 관계가 없으므로 이니셜을 추측하느라 애쓰지 마시기 바랍니다.)



Who : 강보람
Where : ITFlow
What : 선임 컨설턴트
Self Intro : Microsoft C# MVP이며, Visual Studio 공식팀(www.vsts2010.net) 멤버로 활동 중입니다. C#을 비롯한 언어 및 TFS에 관심이 많습니다. 개인블로그 '워너비의 소프트웨어 팩토리'(blog.naver.com/netscout82)를 운영 중입니다.

Who : 박세식
Where : 유니위스
What : 대리
Self Intro : 아이를 낳기 전에 승부수를 던지고 있는 유부남 청년입니다. Visual Studio 공식팀(www.vsts2010.net) 에서 ASP.NET MVC 카테고리를 담당하고 있고 개인블로그 sses's blog (http://sses.tistory.com) 를 운영하고 있습니다.

프롤로그

보람 : 칭구. 오랜만이군. 요즘 잘 지내고 있나?

세식 : 잘 지내 보이냐? 앙?

보람 : 아 뭐 안 좋은 일이라도 있으신가 봐요 고갱님.

세식 : 하~~. 말도 마라. 하~~~~~~.

보람 : 이런, 니 한숨에서 세상의 끝을 본 기분이야.

세식 : 내가 저번에 프로젝트 시작한 거 있다 그랬잖아? 기분 좋~~~~게 시작한 거.

보람 : 아~ 그 My Happiness인가 뭔가 하던 그거?

세식 : 어. 그거.

보람 : 근데..?

세식 : 망했어 이씨.

보람 : 아. 망했구나. My Happiness라고 설레발 치고 다니더니 어찌 망했대?

세식 : 야. 말로 하긴 귀찮고. 이거나 봐라.

세식의 기록

날짜 : 2010년 11월 9일
프로젝트 이름 : MHSG프로젝트
제목 : 프로젝트를 시작하기 전 날.

아, 내일은 프로젝트를 시작하는 날이다. 클라이언트는 MHSG. 이게 이 회사의 진짜 이름은 아니고, MH는 '망할'의 약자이다. 내가 붙인 이름은 역시 아니고, 업계에서 이런 쪽으로 유명한 거 같다. 뭐 그럼 어떤가. 우리나라 IT에서 어디 한 곳 안 위험한 데가 있었던가. 나는 주어진 과제가 어려우면 어려울 수록 불타 오르는 화끈한 성격의 소유자. 좋다. 어디 한 번 부딪혀서 성공적으로 프로젝트를 이끄는 데 한 몫 당당히 해보자.

날짜 : 2010년 11월 10일
프로젝트 이름 : MHSG프로젝트
제목 : 프로젝트 시작한 날.

고객과 첫 미팅을 했다. 미팅에 참석한 사람은 고객대표와 프로젝트 담당자. 우리 회사에서는 이사 몇 명과 차장, 그리고 전선에서 총알받이 할 개발자들이 모였다.
분위기는 생각 이상으로 화기애매 하다.... 아니 오타 났다. 화기애애하다. 이렇게 가축적인...아니 가족적인 분위기에서 회의를 한 게 얼마만이었던가. 서로 웃고 떠들고 농담을 하나라도 더 따 먹으려고 애쓰던 모습이 선

하다. 아무래도 이 프로젝트는 잘 될 것만 같다.
고객사가 원하는 것의 개요를 들었다. 요구사항은 매우 명확해 보였고, 이사와 차장도 흔쾌히 고개를 끄덕여 성공적인 장미 빛 미래를 속삭였다.
기분 좋은 날. 빨리 집에 들어가 와이프에게 보고하고 칭찬받아야겠다. 낄낄낄
ps.
MH는 망할이 아니고, My Happiness의 약자이다. 이렇게 바꿔 불러야겠다.

날짜 : 2010년 11월 25일
프로젝트 이름 : 망할SG프로젝트
제목 : 2주 후 밤샘

이상하다. 분명 그 동안 프로젝트의 경험으로 비춰봤을 때. 이런 격한 밤샘 근무는 보통 프로젝트 막바지에 하기 마련인데, 왜 프로젝트 시작한지 2주밖에 안된 이 시점에 나는 몇 일째 밤샘 근무를 하고 있을까. 뭔가 이상하다. 이상해.
점점 머리가 아파온다. 코딩을 해야 하는데, 아이디어는 머리 속에서 '나 잡아 봐라~'하고 도망 다닌다. 그래서 눈은 뜨고 있고, 키보드는 잡고 있지만 일이 되지 않는다.
저 뒷자리에 앉아서 아무것도 안하고 자리만 지키고 있는 차장. 저 인간 때문에 퇴근도 못한다. 몇 일 전에는 저녁 11시 반에 퇴근하다가 '뭔데 이렇게 일찍 퇴근하느냐'하는 말을 들었다. 다시 한번 말하지만 아직 2주밖에 안됐다. 2주 밖에 안됐다고.....

날짜 : 2011년 1월 13일
프로젝트 이름 : 멍게해삼SG프로젝트
제목 : 중간 보고

이 자식들은 멍게와 해삼인 게 분명하다. 내가 몇 번을 확인했건만... '이게 정말 확실합니까? 이렇게 해야 되는 거 맞지요?'라고 내가 몇 번을 확인했건만. 중간 발표에서 자기들이 원하는 건 그게 아니었다고 말한다. 이 자식들은 멍게 해삼이 분명하다.
결국 프로젝트의 끝은 가까워지고, 남은 기간에 바뀐 요구사항을 또 적용해서 완료를 해야 한다. 지금까지 이틀에 한번 꼴로 밤샘을 했다. 과연 이런 일정이 언제까지 이어지는 걸까?
ps.
오늘은 멍게와 해삼을 안주 삼아 술 한잔 해야겠다.

날짜 : 2011년 2월 28일
프로젝트 이름 : 아 몰라 말하기도 싫어
제목 : 막장(章) and 막장

분명 계약한 프로젝트 기간은 이미 끝이 났다. 그런데, 우리가 추가로 비용을 지불하면서 프로젝트를 계속 진행하는 이상한 일이 벌어지고 있다. 혹시 멍게와 해삼이라서 돈의 관념이 없는 건 아닐까.
그리고 이 프로젝트는 분명 끝이 나야 할 것 같기는 한데, 새로운 요구사항은 끊임없이 들어오고 있다. 혹시 평생 직장이 필요한 건가?
게다가 이번 달 월급이 안 나왔다. 와이프에게 할말이 없다. 직원들의 분위기는 점점 험해지고 있고, 요구사항은 하이패스처럼 걸러지지 않고 계속 날아든다. 이제.. 이제... 그만. 그만해 이 자식들아..

다시 현실로 돌아와서

보람 : 아..... 멍게와 해삼의 후예들이랑 일을 했고만?

세식 : 밤샘까지는 어떻게 참을 만 했는데... 이쒸 돈도 안주고.

보람 : 가정이 있는 사람한테는 그게 가장 가슴 아픈 일이지.

세식 : 그러게.. 요즘은 한숨만 나온다. 이런 멍게해삼 프로젝트 말고 정말 My Happiness 프로젝트는 없는 걸까?

보람 : 글세다. 현실적으로 어려움이 있긴 하지만, 도움이 될만한 방법들은 있지.

세식 : 커헉. 그게 뭔데?

보람 : 자네 TDD라고 들어봤나?

세식 : 글쎄, 여러 군데서 많이 들어보기는 했는데 자세하게 TDD 의 형식을 띈 개발을 딱히 해봤다고 하기도 좀 어렵고, 얼마 전에 끝난.... 아 이름도 말하기 싫은데, 그 멍게 해삼들이랑 했던 프로젝트는 전혀 그거랑 상관없는 거 같고, 회사에 QA 팀이 따로 있기는 한데 그 팀 자체에서 테스트 케이스를 만들어서 테스트를 해주고 있지. 개발자가 개발한 것들을 QA 팀으로 넘기면 그 팀에서 테스트를 해주고 그것에 대한 버그 리포트가 넘어오면 우리가 다시 수정하고 넘기면서...

보람 : 음.. 이야기를 길게 했는데. 전형적인 폭포수 방식이네..

세식 : 하하 얘기하고 보니 그렇네.

보람 : 클라이언트 요구사항에 의해 설계하고 구현하고 구현이 다 끝난 다음에 QA 팀에서 테스트하고, 버그나 기타 요구사항이 있으면 다시 고치고 테스트하고, 전형적인 폭포수처럼 쭉 가는.. 음.. 아마 대부분이 그렇긴 그럴거야. 내가 예전에 일했던 회사들도 대부분 그랬고. 대부분이 조직이 있다고 해도 그 뭐랄까 SI 특성상 시간이 많이 부족하고 요구사항이 막 요동치기 때문에 그렇게 체계적으로 잡아서 할만한 에너지가 없다고 할까?

세식 : 그런데, 이미 사람들이 TDD 에 대해 많이들 알고 있잖아?

보람 : 근데 생각보다 많이는 모르고 있을 거야. 단위테스트를 안 해본 사람도 꽤 되는 것으로 알고 있고, 예제는 들어봤지만 실제로 어떻게 하면 좋을지 모르는 사람도 참 많고 게다가 남들 다 안 하는데 나 혼자 하기가 좀 그래서 안 하는 사람도 있고

세식 : 응. 테스트 케이스를 만드는 시간 자체도 바로 개발하는 것과 같이 시간을 소비하는 것처럼 느껴져. 난 개발 일정에 맞춰 빨리 일을 끝내야 하는데

보람 : 그렇지 이렇게 하는 동안 다른 사람들이 볼 때는 일 안 하는 것처럼 보이고. 그래서 뭐. 그런 단점이 있어서 많이 안하고, 그리고 TDD 자체가 문화적인 측면이 많아 가지고.. 자 한번 얘기해보자.

TDD 란?

보람 : 내가 TDD를 생각을 할 때 가장 뭐랄까 와 닿았던 것 중에 하나는 하드웨어와 소프트웨어의 차이점이었거든. 차이점이 뭐냐면 하드웨어는 칩셋 같은 만들어 진 것을 기판 위에 올려서 서로 연결하고 연결하고 이런 식으로 하잖아. 각 칩셋을 만드는 제조사가 다르고 각 칩셋을 조립해서 또 하나의 기능을 하는 기계를 만드는 건데 어떻게 보면 소프트웨어가 지향하는 점이 그거거든. 소프트웨어가 궁극적으로 지향하는 점이 하드웨어처럼 조립식으로 딱딱딱 조립만 하면 완성품이 나오게. 이게 소프트웨어가 궁극적으로 지향하는 바이긴 하거든, 그래서 예전에 한창 컴포넌트 기반에서 개발하는 게 유행이기도 했지. 그래서 많은 컴포넌트 기술도 나왔고, 물론 여러 가지 한계점을 많이 안고 있었지만, 암튼 뭐 그런 식으로 소프트웨어도 조립해서 끝나면 좋겠다라는 사람도 많이 있었는데 그렇게 성공적이지는 못했지. 물론 개발 생산성을 높이는 데는 굉장히 많이 기여했다고 봐. 객체

지향 기술이 없었다면 지금 처럼의 시스템을 개발할 수 있었을 것 같진 않거든. 객체지향 언어가 뭔가 없었다면 말이지.. 이러한 노력들이 객체지향 개발 컴포넌트 기반의 개발 방법론 이라던지, 그런 것들이 상당히 기여했다고 생각을 하는데 아직 뭐 소프트웨어를 조립을 하는 데까지는 못나갔다고 생각해.

암튼, 소프트웨어도 결국은 하나의 블랙박스로서의 기능을 하는 거거든. 그래서 객체지향의 요소를 보면은.. 어... 그거 뭐라고 하지?

세식 : 캡슐?

보람 : 응 캡슐화. 캡슐화가 객체지향의 하나의 중요한 개념이잖아. 그래서 외부에서 알 필요가 없는 것들은 철저하게 감춰서 추상화하고, 필요한 것만 외부로 공개를 해서 그것을 통해서 서로 조립되듯이 인터페이스를 통해서 결합되고, 내부의 동작을 바꾸더라도 인터페이스만 바뀌지 않으면 외부의 것들이 영향을 받지 않게끔.. 그게 레이어를 나누는 거잖아. 마찬가지로 요즘 유행하고 있는 ASP.NET MVC 같은 MVC 패턴도 레이어 간의 책임과 역할을 명확하게 분리해서 서로 한쪽에서 변경이 일어났을 때 다른 쪽까지 전파되지 않게끔 분리 하는 거고 그렇게 분리했기 때문에 분리해놓은 것들이 서로 각자 테스트하기도 쉽고.

하드웨어를 보면 각 칩셋들이 철저하게 분리되어 있는데, 각 칩셋마다 그 인터페이스가 공개가 되어있거든. 몇 번 선에는 무슨 입출력이 있고 여기는 전원 꼽는 데고. 이렇게 명세가 있어. 그 다음에, 이 칩셋이 잘 동작하는 지 테스트할 수 있는 방법이 있단 말야. 왜냐면 칩셋은 블랙박스이기 때문에 얘가 잘 동작하는지 안 하는지 내부를 뜯어보고 알 수는 없어, 대신에 외부에서 이 칩셋이 잘 동작하는지 테스트하는 방법이 있다는 거지. 그래서 이렇게 하드웨어를 사서 조립을 하지만 걔가 잘 동작할거라고 알 수 있는 게, 각 칩셋이 잘 동작하는지 테스트를 하고, 잘 동작한다면 그 칩셋들을 모아서 만들어 놓으면 걔네도 기능적으로 잘 동작할거다라고 가정을 할 수 있는 거지.

그래서 하드웨어에서 그런 블랙박스로 하드웨어 같은 칩셋을 만들 때 이 칩셋을 테스트할 수 있는 방법을 제공하듯이, 소프트웨어도 하나의 블랙박스처럼 만들 때 외부에서 테스트할 수 있는 방법이 있어야 한다는 거지. 그래서 어떻게 보면 소프트웨어의 TDD.. 꼭 TDD가 아니라 코딩 후에 단위테스트를 만든다 할지라도 그런 식으로 하나의 만들어진 소프트웨어 블랙박스가 잘 동작하는지 테스트할 수 있는 방법을 제공한다 라고 하는 그런 의미로 봤을 때 굉장히 가슴에 많이 와 닿았거든.

거대한 하드웨어일수록 각 부품들을 빼서 자가테스트 할 수 있는 방법들이 다 있거든. 자동차 라던지 말야. 어떤 부품을 테스트할 때 이 부품은 이렇게 테스트하고 저 부품은

저렇게 테스트하고 다 방법이 있단 말야. 그렇게 하게 되면 장점이 또 뭐가 있냐면 어디가 고장 났는지 쉽게 알 수 있다는 거지. 각 하드웨어의 부분부분을 테스트할 수 있는 방법이 있다면 어디가 고장 났는지 쉽게 알 수 있잖아. 소프트웨어는 하드웨어보다 더 결합되기가 쉽고, 스파게티처럼 더 꼬이기 쉽기 때문에 어디서 어디부터가 문제가 생겼는지 알기가 쉽지가 않거든. 만약에 단위 기능. 가장 작은 단위를 메서드로 생각해보면, 메서드 별로 테스트할 수 있다면 어떤 메서드에 문제가 생겨서 어디까지 문제가 전파되는지 그것도 단위테스트로 철저하게 해놓으면 파악을 할 수 있는 거지 나중에 자세하게 얘기하겠지만 그러한 장점도 있는 거고.

우리가 아까 처음에 시작할 때 프로젝트 어떻게 하고 있냐고 했을 때 폭포수 개발 얘기가 나왔었는데 그런 소프트웨어 역사를 보면 테스트는 항상 뒷전에 밀려있었거든. 항상 그랬지, 테스트는 맨 뒤로. 왜냐면 만드는데도 시간이 부족하기 때문에 우리가 테스트할 수 있는 시간이 있겠냐 하는 거지. 그러다 보면 이제 테스트를 고객이 하게 되지. 이게 뭐 어쩔 수가 없는 게 소프트웨어 역사가 짧다 보니까 아직 이런 사람들의 마인드 라던지 요구사항의 관리 같은 것들이 잘 안되게 되면, 프로젝트를 종료해가는 시점에서도 계속 새로운 요구사항이 밀려오기 때문에 그걸 다 처리하게 되면 테스트고 뭐고 간에 그냥 돌아가는 코드 짜면 끝나는 거지 특별히 뭐 테스트할게 없거든

세식 : 그렇지. 정말 테스트에 시간 쓰는 것은 둘째치고, 출시일 전날부터 밤 새워가며 출시하기 바로 직전까지 개발해서 배포하는 경우도 있지.

보람 : 뭐 일단은 출시가 가장 중요하니까. 요즘 대세가 되는 스크럼 같은 방법론을 보면, 스크럼 마스터나 프로그램 책임자 같은 사람들이 제품의 요구사항을 모두 담고 있는 백로그를 책임지고 우선순위에 따라서 관리하도록 되어 있는데, 아직 스크럼이 퍼지려면 시간이 필요한 거 같고, 스크럼을 하더라도 프로그램 책임자 같은 사람들의 마인드가 제대로 정착이 되야 좀 좋은 쪽으로 문제가 해결되겠지.

스크럼 이야기가 나왔으니까 말인데, 소프트웨어 공학의 역사를 보면, 어떻게 하면 더 효율적으로 빠르게 개발을 할 수 있을까에 대해서 연구를 해 온 건데, 사람들이 연구를 하다 보니까 몇 가지 법칙을 발견한 거지. 뭐냐면, 처음에 문제를 발견했을 때 프로젝트 초기에 해결을 하게 되면 비용이 굉장히 적게 드는데 이게 나중에 프로젝트 중 후반으로 가서 고치려 하면 이게 굉장히 복잡해 지는 거지. 이유를 생각해보면, 초기에는 예를 들어서 어떤 메서드에 문제가 있다고 할 때 그 메서드의 기능을 활용하는 코드의 수가 작을 거란 말야. 초기에는 많이 작성을 안 한 상태이기 때문에 서로 엮인 코드가 작겠지. 그렇기 때문에 문제가 되는 코드를 고치면 얘 때문에 영향을 받는 메서드들이 적을 거고 그래서 쉽게 고칠 수 있겠지. 그런데 이게 프로젝트 중 후반으로 가게 되면, 인기 있는

메서드 같은 경우는 엄청나게 많이 호출되거든, 예를 들어서 약 백 번 정도 다른 곳에서 호출이 된다고 생각해보자. 그러면 얘를 고쳤을 때 과연 백 군데가 다 문제 없이 돌아갈 거냐 라고 물어봤을 때, 확실하게 보장할 수 없다는 거지. 게다가 백 군데를 다 파악할 수 있을 것이냐. 이건 조금 더 복잡한 문제가 될 수 있는데, 왜냐면 많은 사람들이 코드를 짜기 때문에 한 사람이 짰으면 그래도 자기가 뭐 기억은 하겠지만, 예를 들어서 수십 명이 짰다, 이러면 문제가 심각해 지는 거지. 그래서 초반에 발견했을 때 고치면 굉장히 비용이 절약되는데 후반으로 가서 고칠수록 문제가 굉장히 커 진다는 거지. 그래서 어떻게 하면 효율적으로 테스트할 수 있을까 하는 게 대두되기 시작했지.

결론은 테스트를 뒤에 가서 하게 되면 문제가 생긴다는 건데, 그게 왜 문제가 생기냐면, 테스트를 제대로 안 하게 되거나 건너뛰게 된다는 거지. 왜냐면 항상 보면은 가장 중요한 일은 처음에 해야 되거든 가장 중요한 일을 처음에 해야 나중에 시간이 부족하더라도 가장 중요한 일을 끝내고 덜 중요한 일은 못 끝내는 상황이 생기더라도 그래도 이건 괜찮은데, 가장 중요한 일을 나중에 미뤄놓고 앞에서 별로 중요하지 않은 일에 시간을 다 허비하고 시간이 없어서 가장 중요한 일을 못하면 안되잖아. 프로젝트의 진행을 보면 테스트가 가장 중요한 것임에도 불구하고 항상 방법론을 보면 맨 끝에 가있어. 그러다 보니 시간상 이게 잘 처리가 안 되는 거야. 왜냐, 구현하는데 진땀 빼고 제대로 관리 안 되는 요구사항이랑 싸우느라 시간이 없는데 뒷전으로 밀려있는 테스트까지 시간이 갈까? 아니라는 거지.

세식 : 근데 솔직하게 얘기해보면 테스트라는 시간을 따로 생각하거나 빼놓지 않아. 일단 나한테 일이 주어지면 바로 그냥, 그 일을 빨리 처리 해야 하거든.

보람 : 그것도 나도 충분히 이해를 해. 그렇기 때문에 이 TDD는 테스트를 중심으로 해서 프로젝트가 진행이 되는 거란 말이지. 테스트를 최우선으로 놓고. 아까 니가 얘기했듯이 개발을 하다 보면 테스트를 마인드에 두고 개발을 하진 않지. 일단 다 구현해놓고 나서 자가 테스트를 하잖아, 클릭클릭 하면서 잘 돌아가나 확인하고 아 잘 돌아가네 하고 QA 팀에 넘기잖아. 그러면 버그리스트가 막 날아오지. 그거와는 다르게 테스트를 항상 중심에 놓고 개발하는 게 TDD 이지. 요즘에 애자일이 굉장히 유행을 하고 있고, 예전에 내가 만났던 사람 중에서 애자일을 맹신하는 사람도 있었는데, 그만큼 효율적 이라는 거겠지. 애자일이 부상하면서 또 TDD라는 것도 한 때 이슈가 되었었는데, 이제 TDD에 대해서 좀더 깊게 얘기해 보면...

테스트라는 것을 중심에 놓는다고 하는데 과연 어떻게 중심에 놓는지 하나씩 생각해보자고. 일단 TDD를 하기 위해서는 우선 요구사항을 명확하게 해야 돼. 왜냐하면 코드 작성에 앞서서 먼저 테스트를 작성해야 하잖아 그치? 테스트를 작성하려면 뭘 테스트 해야

하는지, 테스트할 때 어떤 항목들을 테스트하려는지 알아야 해. 그러기 위해서는 요구사항이 있어야 돼. 그래야 그 요구사항에 맞게 테스트를 하니까. 그렇기 때문에 요구사항을 명확하게 하는 데서 시작을 하고, 그 다음에 개발하려는 기능의 세부사항을 명확하게 해야 하지. 예를 들어서 뭐 어떤 기능을 구현하는데 그 기능에 세부적으로 필요한 기능이 몇 개다, 메서드를 가지고 생각해보면, 이런 이런 일을 하는 메서드가 몇 개 필요하다. 이런걸 명확하게 해야 돼. 그걸 명확하게 한 다음에 그 각각의 세부기능에 대해서 이제 입력 값, 출력 값이 명확해야지. 그래야 그 단위를 테스트 할 수 있으니까. 그런 것들이 명확하게 된 다음에 테스트코드를 작성 하는 거야. 그리고 테스트 코드를 작성한 다음에는 일단 컴파일이 되도록 해야 돼. 테스트 코드만 작성된 상태에서는 컴파일도 안되거든. 그래서 일단 컴파일이 되게끔 코드를 작성을 하고, 테스트가 성공할 때까지 코드를 작성해나가는 거야. 그리고 테스트가 요구사항을 만족할 때까지 구체화 시키면서, 메서드가 그 테스트를 모두 통과할 때까지 작성 및 리팩토링 해가는 거지. 테스트를 먼저 작성해놓고 그것을 통과하도록 메서드의 코드를 작성해나가기 때문에 기능 구현이 완료가 되면 이 메서드는 이런 이런 테스트를 통과했기 때문에 기능적 요구사항을 만족한다 라고 얘기할 수 있는 거지. 그리고 이런 통과 여부를 명확하게 통계로 가져갈 수 있는 거지.

세식 : 그래서 이 케이스 자체가 요구사항 명세를 의미한다 라고 들은 것 같아.

보람 : 응. 그 말도 맞을 거야.

세식 : 이 테스트 자체가 요구사항대로 다 처리를 하는 케이스이기 때문에 이것이 요구사항 명세를 대체한다 라고.

보람 : 응 그렇게 할 수 있지 그렇게 하기 위해서는 애자일한 마인드로 개발을 해야 하는데, 왜냐면 애자일한 마인드로 개발을 하게 되면 뭐랄까 특별히 무거운 문서작업에 크게 얽매이지 않는 경향이 있거든. 예를 들어서 화이트보드에 이렇게 막 한 다음에 그걸 문서 대용으로 사용한다던가, 포스트 잇을 사용한다 던지 하는 방식으로 말이야. 일반적으로 딱딱하게 굳어있는 방법론에서는 상상도 할 수 없지. 왜냐 모든걸 격식에 맞춰서 만들어야 하는데 그게 아니라 스케치하듯이 그런 식의 유연한 문화를 받아들이는 것도 애자일에 하나니까. 그런 마인드로 모두가 문화적으로 되어있다면 아까 너가 얘기했던 대로 테스트케이스. 테스트 코드 쭉 통과한 거 그걸 문서로 대체한다 그것도 가능할 수 있겠지 오히려 그런 게 애자일한 방법에 더 맞을 수도 있고 그렇지.

- 정리

그 동안의 개발 방법론들은 소프트웨어를 하드웨어처럼 예측 가능하게 개발하려는 시도로 점철되었었다. 하지만, 소프트웨어는 여러 면에서 전혀 소프트 하지 않으며, 관리하기는 더욱 더 어려워 지고 있다. 코딩보다는 테스트 및 디버깅에 더 많은 비용이 들어가는 현실에서 테스트의 중요성은 점점 더 대두되고 있지만, 기존의 테스트를 맨 뒤에 놓는 개발 프로세스로는 전혀 이런 목표를 달성할 수 없다. 아니, 테스트 조차 할 수 없는 상황이 종종 발생하기도 한다.

TDD 로 얻을 수 있는 장점

보람 : TDD를 통해서 얻을 수 있는 장점을 보게 되면 아까 얘기했듯이 테스트코드를 먼저 작성하고 그것을 만족시키기 위해서 실제 코드를 작성하기 때문에 코드작성이 끝나게 되면은 '내가 작성한 코드는 이런 테스트 코드를 통과했어요. 그리고 이 테스트 코드가 요구사항을 잘 반영하죠. 그래서 내가 작성한 코드는 기능을 잘 수행하는 겁니다'라고 보증할 수 있는 거야. 보통 우리 보면 개발하면서 메서드를 많이 작성하는데 그 메서드가 정말 제대로 작동하는 건지는 거의 보장할 수 없는 경우가 많거든. 예를 들어 보면, 내가 이 메서드를 작성했는데 이 메서드를 내가 쓸 때는 문제가 없어, 근데 이 메서드를 다른 개발자가 쓰면 문제가 생기는 그런 케이스가 많이 있거든. 왜냐면 이 메서드는 내 생각의 범

위 안에 있기 때문에 나는 이 메서드가 오류가 나지 않는 대로 자연스럽게 쓰는데 이 사람은 내가 만든 메서드를 자기 스타일대로 쓰려고 하면 '어 이거 왜 안되지? 이거 안되요'라고 하는 케이스가 많이 있거든.

세식 : 어? 그럼 그 케이스자체가 테스트를 하려고 만드는데 그것 또한 내가 생각한 케이스대로 테스트 메서드를 만들 거잖아.

보람 : 그건 좀 다른 얘기인데, 방금 내가 얘기했던 것은 테스트를 안 할 경우에, 테스트를 안 했을 경우에 에 대한 이야기 였거든.

세식 : 안하고 그냥 넘겼을 경우에..

보람 : 그렇지. 안하고 넘겼을 경우에 내가 작성한 코드에서는 잘 돌아가는데 다른 사람이 작성한 코드에서는 잘 안 돌아갈 수 있다는 거지. 그리고 물론 니가 얘기했던 대로 테스트코드 작성할 때도 엄밀하게 작성 안하고 자기 편의대로 작성한 경우에는 그런 케이스가 생길 수 있지. 왜냐하면 요구사항이 만약에 명확하게 정의가 되어있어서, 예를 들어서 '1을 넣었을 때는 0이 나오고 2를 넣었을 때는 1이 나와야 한다' 이런 식으로 명확하게 되어있으면 그거는 비껴 나갈 수가 없거든. 그런데 요구사항이 좀 애매모호하게 정의가 되어있다면..

세식 : 응 그치. 오히려 그런 게 더 많고.

보람 : 그렇게 되어있다면 좀 사람에 따라서 어떤 사람은 테스트를 잘 통과했어요 하는데, 옆에 있는 사람은 어 이거 에러 나네 뭐 그럴 수도 있는 거고. 니가 얘기한대로 그럴 가능성이 충분히 있지. 일단 이야기를 계속 이어보면, 첫 번째 장점은 내가 작성한 코드를 내가 보증할 수 있다.

또 하나의 장점을 보면 테스트 코드를 작성을 하는데 그 코드 자체가 하나의 예제가 된다는 거지. 예를 들어서 어떤 메서드를 테스트하려면 객체를 생성하고 그 객체의 어떤 적절한 속성을 준 다음에 이 메서드를 호출을 하고 그 다음에 이 값을 어떻게 변환해서 테스트하고 뭐 이러한 쭉 있을 거 아냐. 그러니까 이 테스트코드를 보게 되면 '아 이 메서드는 이렇게 테스트를 하는구나' 하고 사람들이 테스트 코드를 보면은 사용방법을 알 수 있지. 물론 우리나라 문화상 그걸 좋아하는 거 같지는 않지만. 아무튼 충분히 그런 식으로 예제 코드로 활용할 수도 있어. 그래서 그런 장점도 가지고 있지.

그 다음에 내가 생각할 때는 이게 제일 큰 장점 같은데 테스트 코드를 작성을 해놓고 메서드가 그 테스트를 통과하도록 작성을 했어. 그러면 작성을 완료한 메서드는 테스트를 모두 통과한 거 겠지? 만약에 10개의 메서드가 그렇게 되어있다고 해봐. 그러면 나중에

내가 메서드의 동작을 변경을 했을 때, 그러니까 꼭 동작변경이 아니더라도 예를 들어서 리팩토링이라고 해보자. 리팩토링은 메서드의 동작을 변경시키지 않고 그 구조를 좀더 깔끔하게 하는 작업이거든. 그래서 만약에 동작을 변경시키지 않으려고 내부 리팩토링을 했는데, 아무리 주의를 집중해서 조심스럽게 했더라도 메서드의 기능이 변경이 될 위험은 항상 있지. 그럼 그걸 파악하려면 어떻게 해야 될까?

세식 : 글쎄?

보람 : 음. 그걸 파악하기가 쉽지가 않어. 그러니까 내가 변경을 한 메서드가 어디까지 영향을 미칠지, 동작이 변경이 된 건지 안 된 건지 명확하게 알 수 가 없거든. 그것을 만약에 10개의 메서드에 대해서 테스트코드를 깔끔하게 다 작성을 해놓았어. 그리고 내가 하나의 코드를 수정을 했어. 그럼 이 메서드에 대한 테스트 코드를 쭉 돌려보고 나면 얘가 변경이 됐는지 안됐는지 바로 알 수 있지.

세식 : 아..

보람 : 그렇게 10개의 메서드에 대해서 테스트 코드를 쭉 돌렸어. 그러면 내가 변경한 메서드 때문에 다른 메서드에서 에러가 나는지 안 나는지 한번에 확인할 수 있지. 뭐 이런 걸 회귀 테스트라고 하거든. 그러니까 내가 지금 수정한 내용 때문에 이전에 작성한 코드에 이상이 없는지 쭉 테스트 할 수 있는 거야. 그러니까 테스트 코드를 잘 작성해 놓으면 내가 뭔가를 수정했을 때 이게 수정이 제대로 된 건가, 동작이 변경되지 않았을까 하는 불안함을 안 가지고 있어도 돼. 테스트 돌려보면 되거든. 테스트 코드를 작성을 하게 되면 이런 장점이 있지. 이거는 프로젝트가 뒤로 갈수록, 요구사항이 요동칠수록 좀더 유용하기는 해. 내가 작성한 기능이 요구사항을 그대로 지키는지 뭐 이런 거.

세식 : 순간 생각이 든 건데. 전에 이런 게 있었어. 회원가입을 해. 물론 회원가입 절차를 다 밟지. 뭐 실명인증을 하고 몇 개의 폼 양식을 거쳐서 회원가입 완료를 하는데 맨 마지막 처리에 회원가입완료가 되면 문자를 보내주는 기능이 추가가 되었어. 만들어 놓고 보니 이것을 테스트하기 위해서는 뷰 페이지에서 처음부터 다시 실명인증을 거쳐서 여러 폼 양식의 유효성을 통과해야 이를 테스트 할 수 있었지. 실명인증도 중복가입은 안되니 이것 때문에 다른 부분의 코드까지 수정을 해서 테스트 했어야 했고. 그런데 만약 이러한 테스트 케이스가 있었다면 몇 가지의 입력 값 만으로 몇 번이 되었든 손쉽게 테스트를 할 수 있었을 텐데.

보람 : 그렇지. 그러니까 그게 장점이지. 만약 그게 MVC 처럼 잘 분리가 되어있다. 그럼 내가 뷰를 거치지 않아도 내가 값만 잘 입력하면 뷰 없이도 얼마든지 테스트할 수 있잖아. 다른 사람이 만들어 놓은 거 없이. 그리고 뷰를 거치기 않더라도 얼마든지 뷰의 뒷 단을

가지고 변수에 값을 집어넣는 형태로 분리해서 테스트가 가능하지.

세식 : 응. 그래서 내가 이걸 테스트하려면 더 많은 작업 시간이 필요했는데..

보람 : 맞어. 앞에서 개발했던 사람이 실명인증 제대로 통과하는지 테스트 코드 작성해 놓고 뭐 작성해 놓았으면 그대로 활용하면 되는데 그게 없으니까 테스트 한번 하려면 다 거쳐야 하고.

세식 : 그렇지. 마지막 테스트를 위해 전 페이지의 유효성 검사를 모두 통과해야 하니까.

보람 : 그것도 내가 만든 코드도 아니고 그래서 뭐 그런 단점이 있지. 그래서 테스트 코드를 작성을 하면 그런 장점이 있기는 해. 물론 지금까지 얘기한 내용들이 꼭 TDD를 해야만 얻을 수 있는 장점이 아냐. TDD는 테스트를 꼭 먼저 작성하고 가자는 건데, 단위테스트. 그러니까 단위테스트라고 하지. 그러니까 메서드 단위 테스트라던가.

세식 : 그런데 그것의 명확한 차이가 뭐야?

보람 : 응 TDD는 테스트를 먼저 작성하고 가는 거고, 그러니까 단위테스트를 먼저 작성하고 코드를 개발하는 거고, 단위테스트는 그냥 메서드를 테스트 하는 거야. 메서드 같이 그냥 작은 기능단위를 테스트하는 코드를 작성하는 거야.

세식 : 단위테스트에는 그 의미밖에 없는 거야?

보람 : 응, 그렇지. 단위테스트는 그거밖에 없어. 기능이 잘 동작하는지 그냥 그걸 확인하는 거야. 근데 그것을 먼저 작성하고 코드를 작성하냐 아니면 코드를 작성한 뒤에 테스트 코드를 작성하느냐 그런 시점의 차이가 있는데.

세식 : 아. 나는 단위테스트 자체에도 그런 개념이 들어가 있는지 알았거든. 뭔가를 개발할 때 단위테스트 케이스를 먼저 만들고 여기서 발생하는 컴파일 에러까지 포함한 모두가 단위테스트에 속하는 줄 알고 있었거든. 그래서 지금 말한 TDD와 차이가 없는 건가 라고 생각했지. 내가 생각한 단위테스트 역시 먼저 테스트 코드를 작성하니 에러가 발생할 것이고 이 에러를 하나하나 수정해가면서 작업을 진행한다는..

보람 : 아냐. 단위테스트 자체에 그런 개념은 없어. 단위테스트는 그냥 메서드가 잘 동작하는지, 그냥 테스트하기 위한 거지. 그 자체에 먼저냐 나중이냐라는 개념은 없어. 다만 TDD는 그런 테스트를 먼저 작성하고 테스트 중심적으로 개발을 하자는 그런 것을 주장하는 거고, 단위테스트 자체에 그런 개념은 없어. 그래서 위에서 이야기 한 것들은 단위테스트만 작성해도 얻을 수 있는 장점이긴 한데 왜 TDD 라고 사람들이 꼭 주장하고 나서느냐. 아까 내가 했던 얘기하고 비슷해. 개발방법론을 지금까지 보면 테스트는 항상 마지막에 있

잖아. 그러니까 내가 항상 단위테스트를 작성할거라는 보장이 없어. 왜냐 예를 들어서 내가 엄청 바뻐. 엄청 바쁜데 일이 또 뭐 하나가 터졌어. 그러니까 내 업무만으로 충분히 바쁜데 내가 이것까지 수습해야 하는 상황이 생겼어. 뭐 프로젝트 하다 보면 쉽게 만날 수 있는 상황이긴 하지. 과연 이런 상황에서 내가 작성한 코드에 대해서 단위테스트를 만들까? 그렇지, 안 만들지. 그러니까 구멍이 생긴단 말야, 이런 식으로 하게 되면. 그러니까 단위테스트가 가질 수 있는 장점을 다 못 가지는 거야, 그런 식으로 프로젝트를 하게 되면. 많은 프로젝트들이 그렇고, 나도 그래 봤고, 또 그게 현실이고. 대신에 TDD를 해서 테스트를 중심에 놓게 되면 테스트를 먼저 작성하고 가게 되잖아. 물론 TDD를 한다고 해도 느슨하게 하면 위험성은 똑 같지.

세식 : 그렇네, 너가 지금 말한 것처럼 TDD를 한다고 해도, 예전과 똑같은 상황이 벌어진다면 테스트 보다는 일단 먼저 개발부터 할 것 같아.

보람 : 다만 TDD를 좋아하는 사람들은 이래. 그런 사람들은 이렇게 얘기해. 나는 테스트를 작성하지 않고 메서드를 작성한다는 것에 대해서 너무 불안하다 라는 거지. 내가 만드는 코드이지만 그 코드는 내가 보장할 수가 없으니까. 스스로 불안 하다는 거야. 그러니까 이런 부분을 보면은 TDD는 테크닉이 아니거든, 테크닉이 아니고 문화적인 측면이 굉장히 강해. 예를 들면, 우리나라 사람들은 빨리빨리 문화에 적응되어 있지만 외국사람들은 적응이 안되어있어. 근데 과연 빨리빨리 문화가 좋은 거냐 나쁜 거냐에 대해서는 좀 따지기 힘들지.

세식 : 그렇지. 그건 뭐 장단점이 있는 거니까.

보람 : 그러니까 결국에 TDD라는 것도 문화적인 측면이 굉장히 강하기 때문에 꼭 이걸 하는 게 반드시 좋으냐 글쎄.. 뭐 꼭 그렇게 생각하진 않아. 다만 장점이 굉장히 많다는 거지. 그리고 문화적인 측면이 굉장히 강하기 때문에 이런 걸 정착시키려면 사람들의 인식에 변화가 있고 공감대가 만들어 져야 하지.

세식 : 그게 힘든 건데..

보람 : 그게 힘든 거지. 그러니까 조직 차원에서.. 그래서 뭐랄까 쉽게 손을 못 대는 부분이기도 하고. 일단 TDD에는 생각보다 장점이 많다는 거지 이런 장점들을 가져갈 수 있다라는 거지. 그리고 여담이지만, TDD의 모토는 개발한 코드에서 빨리 피드백을 받자는 거거든. 그러니까 빨리빨리 문화에 어울리지 않을까 하는 생각도 드네.

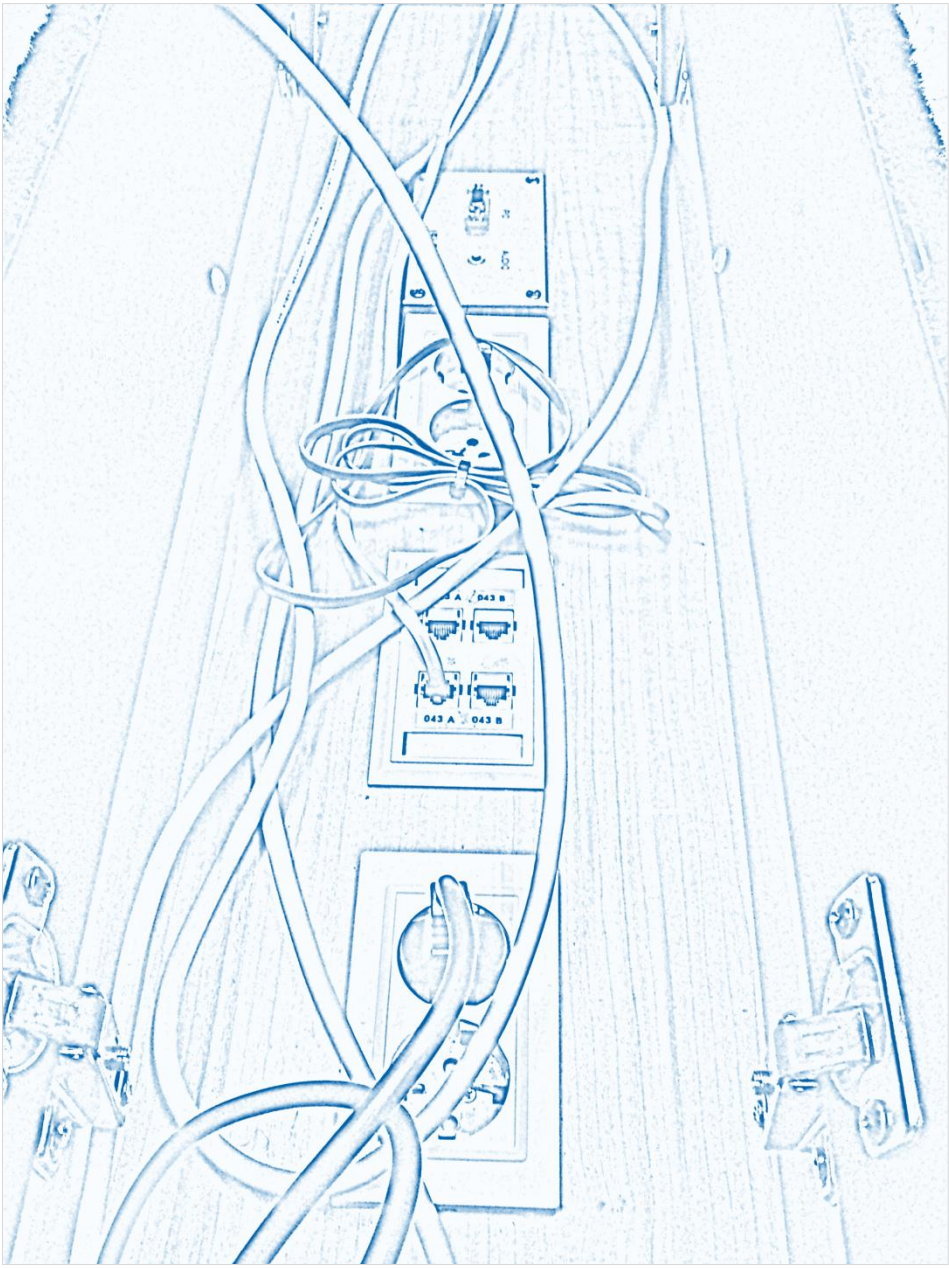
- 정리

단위테스트는 가장 작은 기능단위인 메서드가 제대로 동작하며, 요구사항을 제대로 만족하는지 확인하는 테스트를 말한다. 그리고 TDD란 단위 테스트를 기반으로 짧은 개발 주기를 반복하는 소프트웨어 개발 프로세스를 말한다. 각 개발 주기는 완료된 메서드가 만족해야 할 요구사항을 미리 테스트로 작성해 놓고, 그 테스트를 모두 통과할 때까지 코드를 작성해 나가면서 테스트를 통과하게 만든 후, 코드를 리팩토링하는 과정을 거친다. 이렇게 함으로써 얻을 수 있는 장점은,

1. 테스트를 반드시 작성해야 만 한다.
기존의 방법론에서는 테스트를 맨 뒤에 둠으로써, 테스트를 제대로 수행할 수 없었다. 요구사항이 제대로 관리되지 않으면서 프로젝트 후반 부에서도 계속 새로운 요구사항이 밀려든다면, 테스트 할 시간은 그만큼 줄어들기 때문이다.
2. 요구사항을 먼저 명확하게 하게 된다.
테스트를 작성하려면, 먼저 요구사항이 명확해야 한다. 그래야 요구사항을 만족하는지 테스트를 작성할 수 있기 때문이다. 그리고 테스트를 작성하면서 미진한 요구사항이나, 불확실한 요구사항을 파악할 수 있어서 빠른 피드백을 얻을 수 있다.
3. 기존 코드를 변경할 때, 잘못된 점을 쉽게 파악할 수 있다.
기존에 작성된 코드를 변경할 때, 가장 두려운 점은 내가 수정한 코드 때문에 다른 코드에서 버그가 생기진 않는가 하는 점이다. 그리고 이런 점을 명확하게 파악하려면 내가 수정하는 코드와 관계된 기능을 실제로 돌려서 테스트 해봐야 한다는 난점이 있다. 하지만, TDD를 통해서 단위 테스트가 제대로 작성되어 있다면, 단위 테스트를 돌려보는 것 만으로도 기존 코드의 기능이 제대로 동작하는지 확인할 수 있다.

물론 이런 TDD가 국내 현실에 잘 들어맞진 않는다. 하지만, 꼭 TDD가 아니라고 하더라도 단위 테스트는 매우 큰 도움이 되므로 꼭 작성하길 권장한다.

TDD에서 얻을 수 있는 장점은 개발자에게만 돌아가는 것이 아니라, 관리자에게도 돌아간다. 개발자가 먼저 요구사항을 점검하고, 요구사항에 대한 피드백을 제공할 수 있다는 것은, 그만큼 관리자에게는 불확실성이 줄어든다는 것이며, 코드 작성을 완료함과 동시에 테스트를 모두 통과한 다는 것은, 각 코드가 요구사항에 맞춰 잘 동작할 것이라는 최고의 보증서가 된다. 나중에 터질 문제에 고심하지 않아도 되는 것이다. 그리고 누가 무슨 일을 제대로 했는지도 쉽게 점검할 수 있다.



TDD 를 위한 최적의 IDE, VISUAL STUDIO 2010

보람 : 자 그럼 이제 TDD가 뭔지 이야기를 해봤는데, 그럼 이걸 어떻게 할 수 있을까? 이게 너무 귀찮으면 사람들이 못하겠지.

세식 : 그렇지.

보람 : 너무 귀찮으면 못하고.

세식 : 그리고 이게 일로서 다가오지 않으면 상관없는데.

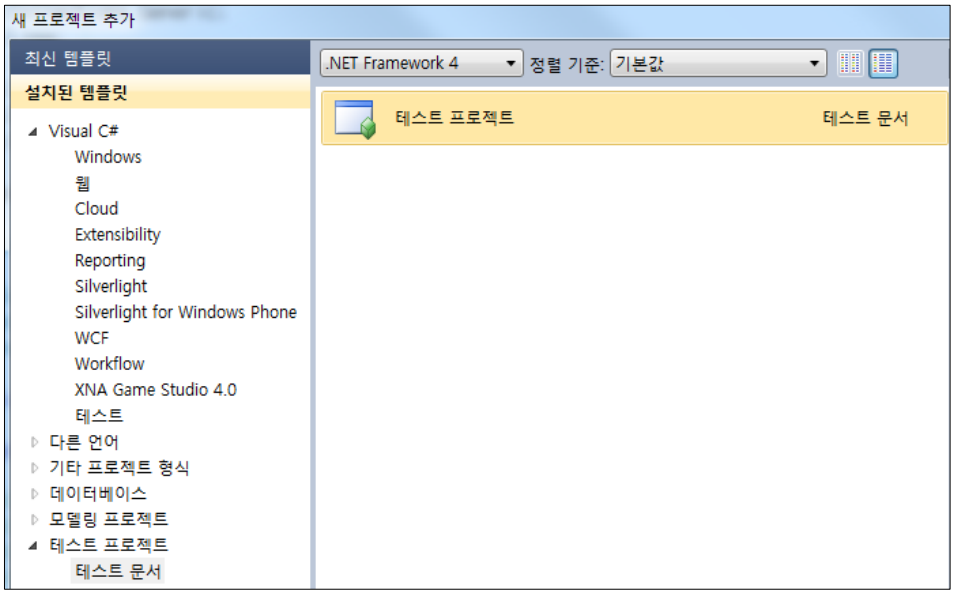
보람 : 그렇지 적당히 편해야지.

세식 : 이게 나한테 또 하나의 일이나 짐처럼 다가오면 절대 못하지.

보람 : 그렇지. 우리 닷넷 개발자들은 어차피 Visual Studio를 써야 하는데, 아니 닷넷 개발자들한테 Visual Studio만한 툴이 없지, 최고의 툴이지. 암튼 이번 2010버전을 보면 TDD를 위해서도 굉장히 신경을 많이 쓴 흔적을 볼 수 있어. 사소한 기능인데 많이 신경을 쓴 듯한 흔적이 있거든. 내가 봤을 때 TDD를 위해서 쓰기에는 굉장히 좋은 툴이 아닌가 하는 생각이 들어. 일단 한번 보면서 이야기 하자고

세식 : 그래.

보람 : 이렇게 콘솔 프로젝트가 있다고 할 때, 새 프로젝트를 추가하면 여기에 테스트 프로젝트가 따로 있어.



세식 : 이거 전엔 없었어?

보람 : 2008에도 있었어. 그 전에는 내가 별로 심각한 프로그래머가 아니었기 때문에 잘 모르겠고, 이렇게 테스트 프로젝트를 만들게 되면 자동으로 단위테스트가 생기는데, UnitTest1.cs 얘는 지우고. 그리고 단위테스트를 만드는 방법을 보자면 두 가지 방법이 있어. 첫 번째는.. 테스트 코드를 먼저 작성을 하고 테스트 코드에 사용된 코드에서 클래스를 생성해 내고, 메서드를 생성해 내는 거야. 첫 번째 방법을 먼저 확인해보면, 일단 프로젝트에 단위테스트를 하나 추가를 하자고. 새 항목 추가에 들어가서 단위테스트를 하나 추가를 해. 그리고 Fib이라는 클래스가 있다고 해보자.

```
[TestMethod]
public void FibTest()
{
    Fib fib = new F
}
```

Exception
ExecutionEngineException
ExpectedExceptionAttribute
ExpectedExceptionBaseAttribute
Fib
FieldAccessException
FileStyleUriParser
FlagsAttribute
float

보람 : 그런데 여기서 한가지 인텔리센스에서 TDD를 굉장히 신경 썼다고 생각이 되는 부분이, 현재 Fib이라는 클래스가 없지? 없는데 인텔리 센스에 Fib이 나타나.

세식 : 음~

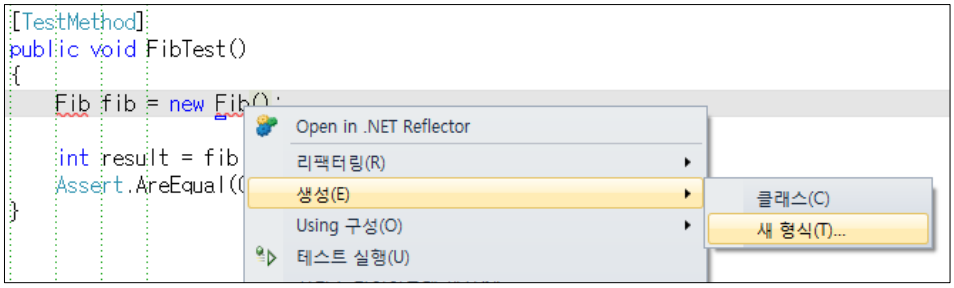
보람 : 이 부분이, 이런 사소한 부분까지도 TDD를 위해서 신경을 썼구나 하는 부분이야. 이게 사소한 거 같지만 굉장히 편리한 거거든.

세식 : 이게 이번 버전에서 추가된 거지?

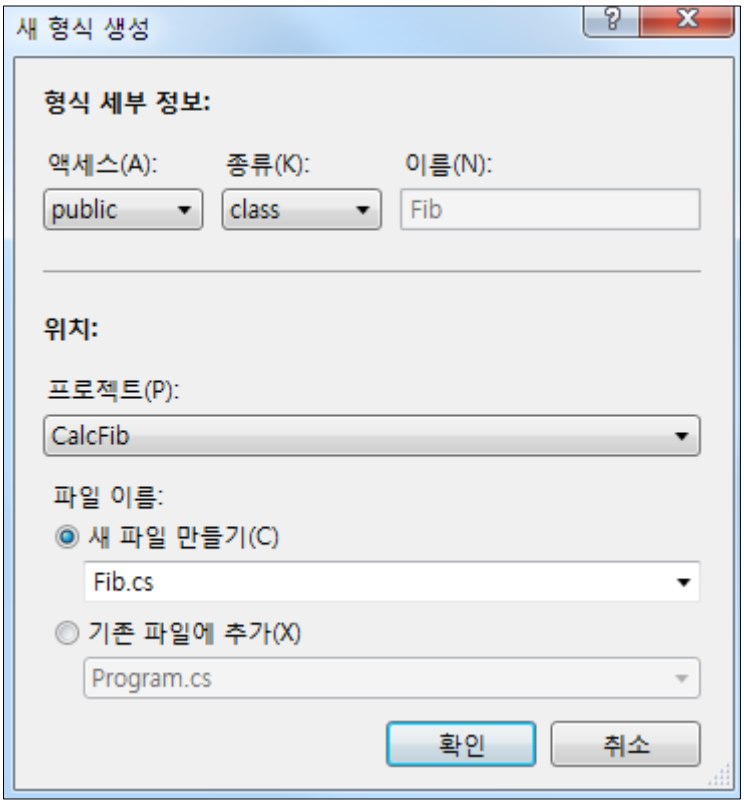
보람 : 어 그렇지. 이런 식으로 result에다가 DoFib이라는 메서드를 이용해서 값을 받아와. 그리고 Assert를 이용해서 기대 값이 0이고 결과로 받아온 값이 같은지 검사를 하지. 피보나치 수열을 보면 0일 때는 0이거든.

```
[TestMethod]
public void FibTest()
{
    Fib fib = new Fib();

    int result = fib.DoFib(0);
    Assert.AreEqual(0, result);
}
```

보람 : 이제 보면, 지금 Fib이라는 클래스 밑에 빨간 줄이 가있는데 지금 이 클래스가 없다는 거거든. 그래서 여기서 마우스 오른쪽 버튼을 누르면,



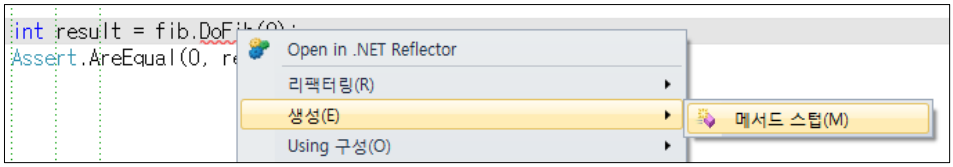
보람 : 생성 밑에 새 형식이라는 메뉴가 있어. 새 형식을 누르게 되면



보람 : 보게 되면, 클래스, 엑세스 종류, 그리고 어떤 파일에 추가할지, 어떤 프로젝트에 추가할지를 정할 수 있어. 이 Fib은 테스트 프로젝트가 아니라 콘솔 프로젝트에 생성이 되어야 하니까 프로젝트를 CalcFib이라고 정해주고 확인을 하는 거지. 그러면 콘솔 프로젝트에 Fib.cs라는 파일이 생성이 돼. 그리고 이제 클래스를 생성하고 나니 DoFib메서드에 빨간 줄이 가지. 왜냐면 클래스를 생성했지만, 빈 클래스이기 때문에 DoFib이라는 메서드의 정의가 없거든.

세식 : 그렇지

보람 : 그래서 이번에는 DoFib위에서 마우슨 오른쪽 버튼을 누르고 생성 밑에 '메서드 스텁'메뉴를 선택하면



보람 : Fib 클래스 안에 DoFib메서드가 생성이 되는 거지.

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CalcFib
{
    public class Fib
    {
        public int DoFib(int p)
        {
            throw new NotImplementedException();
        }
    }
}
```

세식 : 이게 지금 테스트 프로젝트에서만 가능한 건가?

보람 : 아니 어디에서나 가능해.

세식 : 음~

보람 : 그러니까 클래스를 먼저 만들고 나서 코드를 작성하는 걸 별로 안 좋아하는 사람들. 예를 들면 클래스를 만들기 전에 클래스가 어떤 모양으로 쓰일지 먼저 코드를 작성해본 다음에 거기서 클래스를 만들어 내는, 그런 게 유용할 때가 있거든

세식 : 음... 그렇게 쓴 적이 있나?

보람 : 음~ 난 해본 적 있긴 해. 실제로 이 클래스가 어떻게 사용될지 먼저 작성해보는 거지. 왜냐면 클래스를 만들 때 이 클래스에 어떤 메서드가 있어야 되나 하고 좀 모호하거든. 그러면 가상의 클래스가 있다고 가정하고 알고리즘을 짜는 거야. 알고리즘을 짜다 보면 이 클래스에 이런 메서드가 필요하구나 하는 걸 알게 되거든. 대충, 아 이런 메서드가 반복 호출이 되어야 하고, 이런 속성이 있어야 되는구나 하고 감이 오게 되는 거지.

세식 : 음..

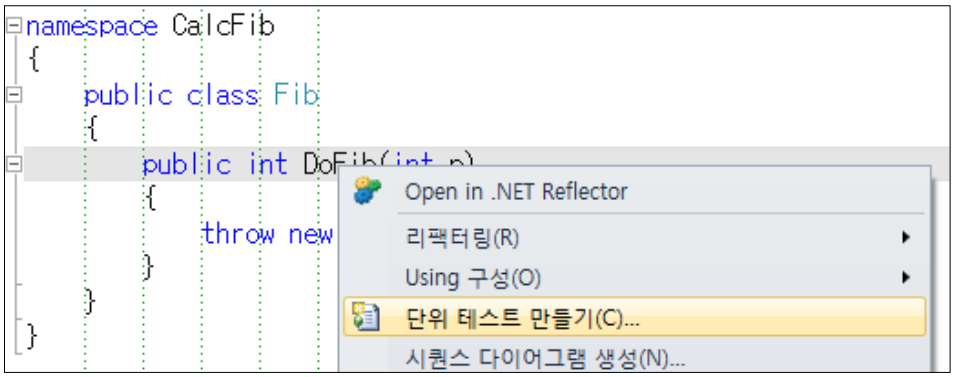
보람 : 그런 경험 많이 있지 않아? 클래스를 작성하려고 public class하고 치다 보면, 이 클래스

에 어떤 메서드가 있어야 하나? 멍~때리는 거 말이야. 상당히 모호할 때가 많거든.

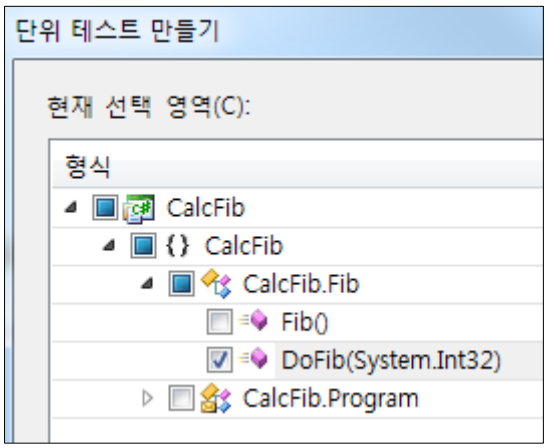
세식 : 그럼 이게 또 하나의 테스트의 어떠한 장점이라고 생각해도 되나? 내가 생각지 못 했던 것들을 이끌어갈 수 있다는..

보람 : 어 그렇지 테스트 코드를 작성하다 보면, 내가 빼놓은 부분들을 발견하기도 하거든. 나중에 테스트 코드를 작성하더라도 내가 코드 작성하면서 이걸 빼먹었네 하는 경우가 상당히 많이 있지. 그래서 단위테스트를 작성하는데 도움이 되기도 하지. 이런 방법도 있고, 다른 방법도 있어. 메서드에서 단위테스트를 생성해내는 거지.

세식 : 아 이미 만들어 놓고



보람 : 그렇지. 예를 들어서 이런 DoFib이라는 빈 껍데기 메서드가 있다고 해보자. 여기서 단위테스트를 생성해내고 싶어. 그러면 메서드 안에서 아무데서나 마우스 오른쪽 버튼을 누르고 '단위 테스트 만들기'라는 걸 선택해.



보람 : 이걸 누르면, 클래스의 어떤 메서드에 대해서 단위테스트를 생성할지, 그리고 단위테스트는 어떤 프로젝트에 만들겠다 하는 걸 선택할 수 있지. 테스트 프로젝트를 선택하고 확인을 누르면, 이렇게 DoFib이라는 메서드에 대한 단위테스트가 쭉~ 생성이 돼. 기본적인 테스트 코드를 다 생성해 주지 쭉~ 하고.

```
/// <summary>
///DoFib 테스트
///</summary>
[TestMethod()]
public void DoFibTest()
{
    Fib target = new Fib(); // TODO: 적절한 값으로 초기화합니다.
    int p = 0; // TODO: 적절한 값으로 초기화합니다.
    int expected = 0; // TODO: 적절한 값으로 초기화합니다.
    int actual;
    actual = target.DoFib(p);
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("이 테스트 메서드의 정확성을 확인합니다.");
}
```

보람 : 기본적인 코드를 다 생성해주지. 앞에서 설명한 것처럼 TDD를 할 때 두 가지 방법이 있어. 테스트 코드를 먼저 작성해서 없는 클래스를 생성해 내던가, 아니면 빈 껍데기 메서드를 먼저 만들어 놓고 거기서 단위 테스트를 쭉 생성해 내던가 말이지.

세식 : 여기 보니까 초기값이랑 다 들어가 있네

보람 : 맞어. 기본적으로 필요한 코드를 다 만들어 주지. 이런 부분들을 보면 Visual Studio가 TDD를 위한 기능을 잘 만들어 놨다는 생각이 들어.

- 정리

Visual Studio 2010에는 TDD를 쉽게 할 수 있도록 지원하는 기능이 포함되어 있다.
1. 존재하지 않는 클래스에 대한 인텔리 센스 지원
2. 사용된 코드에서 클래스 및 메서드의 정의 생성
3. 메서드에서 단위 테스트 코드 자동 생성

간단한 TDD 예제

보람 : 이제 간단한 TDD 예제를 하나 보면서 설명을 해볼까 생각을 하는데, 지금까지 너무 말로만 설명을 해서 말이지. 실제로 어떻게 하는지 한번 보자고. 이 예제가 굉장히 유명한 예제거든. 이 예제를 이제 Visual Studio버전으로 바꿔서 해보자고

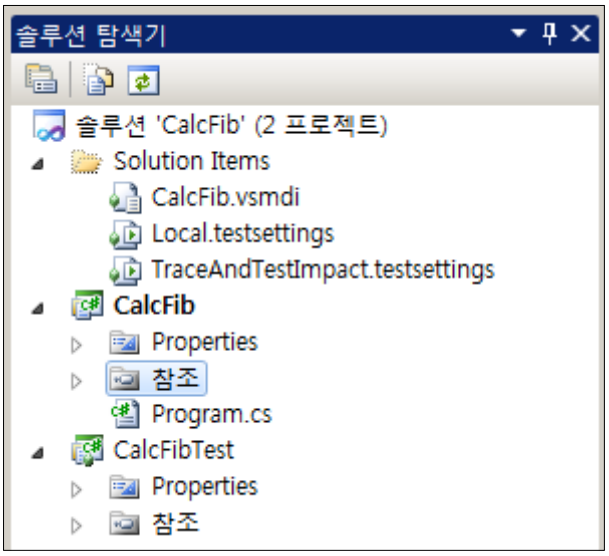
세식 : 어, 이 유명한 예제를 모르면 어떡하지? 아주 유명한 예제인데.

보람 : 아니 뭐 잘 모를 수도 있어.

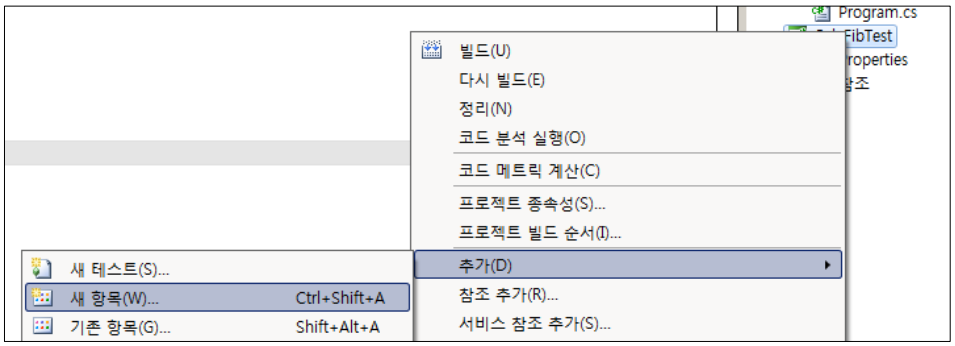
세식 : 그렇지 뭐 알아가면 되는 거니까.

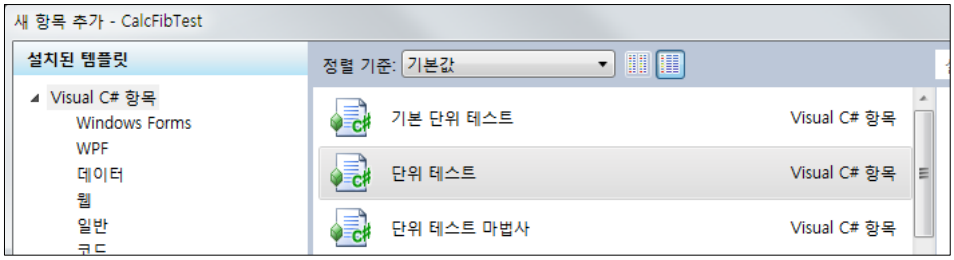
프로젝트 및 테스트 프로젝트 생성

보람 : 먼저 CalcFib이라는 콘솔 어플리케이션 프로젝트를 만들고, 저장을 한다고. 그 다음에 새 프로젝트 추가로 들어가서 테스트 프로젝트를 추가를 해야지. 테스트 프로젝트의 이름은 CalcFibTest. 이렇게 추가를 하지. 기본적으로 추가되어 있는 UnitTest1.cs는 일단 지우고.



보람 : 자 이렇게 하고 나서, 아까 TDD를 하는 두 가지 방법을 소개했지만, 여기서는 단위테스트를 먼저 작성하고 없는 클래스를 생성해 내는 방법으로 가보자고. 테스트 프로젝트에 새 항목 추가를 통해서 단위테스트를 추가를 해.





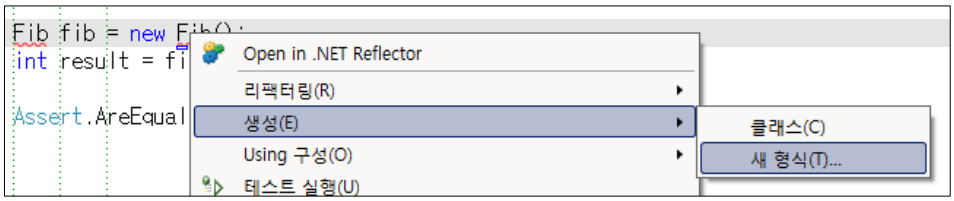
보람 : 단위 테스트 이름은 FibTest라고 입력하고 완료.

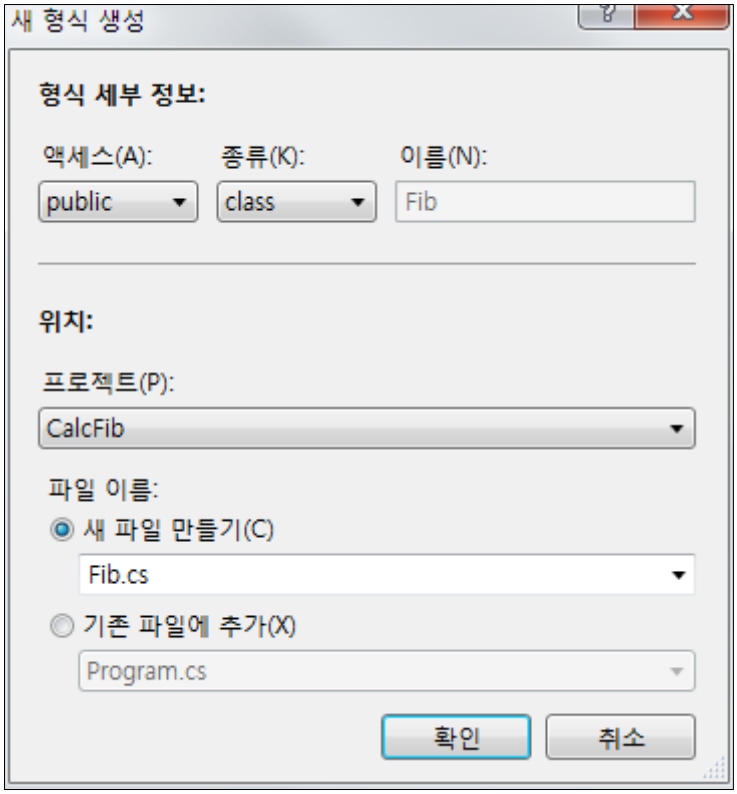
테스트 실패하게 만들기

보람 : 자 이제 생성된 코드를 보면, TestMethod라는 어트리뷰트가 붙어있지 이게 테스트 메서드인데. 우선은 테스트를 실패할 수 있도록 해야 돼. 테스트 메서드 이름을 FibTestMethod로 바꾸고 테스트 코드를 작성해. 우선 Fib클래스의 객체를 생성하고, result에 DoFib이라는 메서드에 0을 넣고 결과값을 받아. 그리고 result가 0이랑 같은 검사하는 거지.

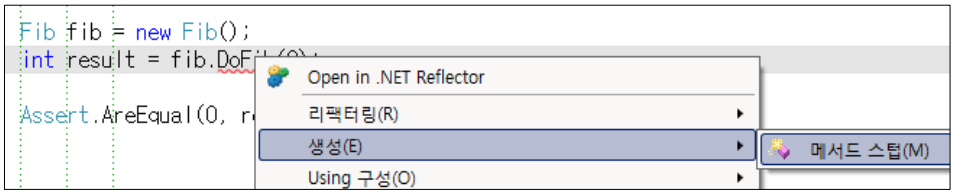
```
[TestMethod]
public void FibTestMethod()
{
    Fib fib = new Fib();
    int result = fib.DoFib(0);

    Assert.AreEqual(0, result);
}
```

보람 : 자 그럼 여기서 이제 빨간 줄을 없애야 돼. 일단 테스트를 실패하게 하려면 테스트 프로젝트가 빌드가 되어야 하거든. 우선 Fib클래스를 새 형식 생성을 통해서 CalcFib프로젝트에 생성을 하고,





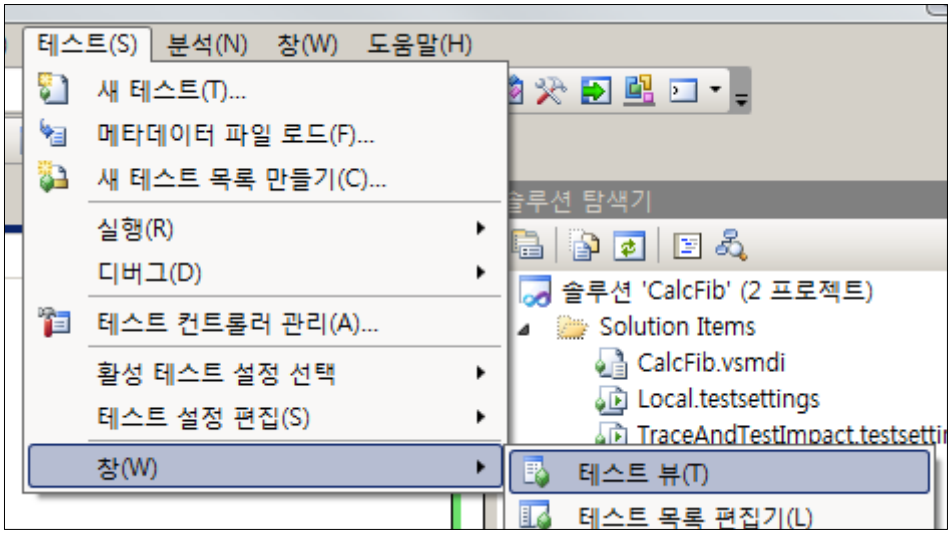
보람 : Fib클래스에 DoFib메서드를 생성을 하지.



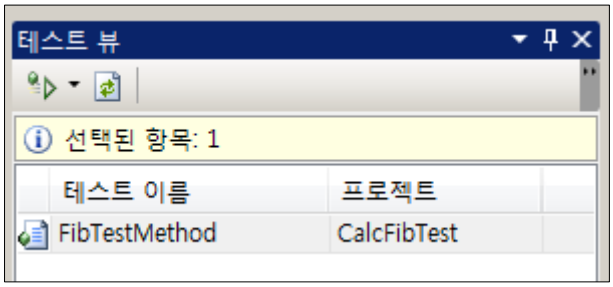
보람 : 그러면 Fib클래스에 DoFib메서드의 빈 껍데기가 생성이 돼.

```
public int DoFib(int p)
{
    throw new NotImplementedException();
}
```

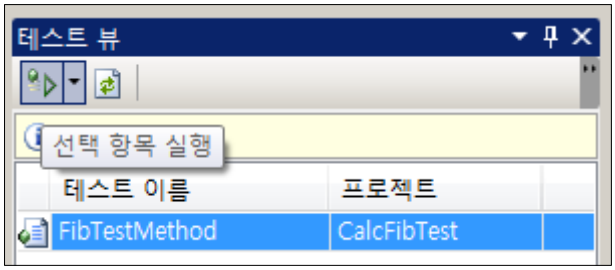
보람 : 그리고 이제 테스트 메뉴에서 창을 들어가면 '테스트 뷰'가 있어.



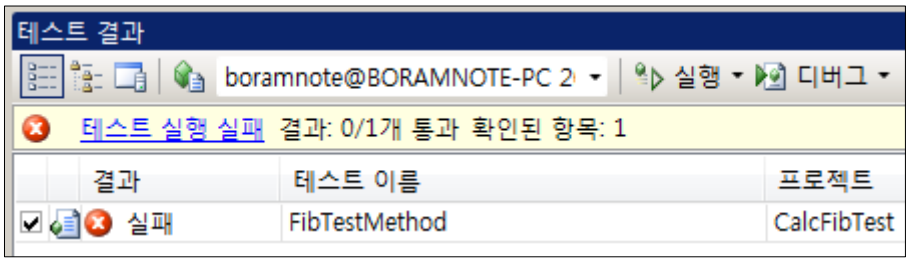
보람 : 테스트 뷰를 눌러서 테스트 뷰를 띄우고,



보람 : 새로 고침을 누르면 우리가 만든 FibTestMethod가 나타나. 그럼 이 테스트를 실행해보자고.



보람 : 선택항목 실행을 눌러서 테스트를 실행하면,



보람 : 테스트 실패가 떠. 그 이유는 DoFib메서드의 스텁만 만들었기 때문에

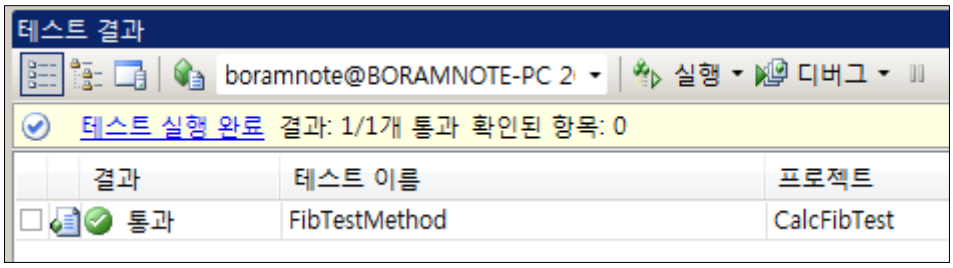
NotImplementedException이 뜬 거거든.

테스트를 통과하도록 코드 수정

보람 : 자 그럼 이제 테스트를 성공하도록 바꿔보자. 지금 테스트를 하는 게 피보나치 수열인데, 피보나치 수열은 0일 때 0이거든? 그래서 테스트를 통과하기 위해서 DoFib메서드에서 그냥 0을 리턴 하자고.

```
public int DoFib(int p)
{
    return 0;
}
```

보람 : 그리고 나서 테스트를 돌려보자.



보람 : 그럼 이제 메서드에서 0을 반환하니까 테스트를 통과했지. 이제 그 다음 단계를 테스트 해보자.

```
[TestMethod]
public void FibTestMethod()
{
    Fib fib = new Fib();
    int result = fib.DoFib(0);

    Assert.AreEqual(0, result);

    //피보나치 수열의 순서
    //0 1 1 2 3 5 8....
    result = fib.DoFib(1);
    Assert.AreEqual(1, result);
}
```

보람 : 피보나치 수열은 0일 때 0이고 1일 때 1이거든. 그리고 테스트를 돌려보면, 실패했지. 왜냐면 DoFib메서드가 0을 리턴 하게 되어 있기 때문에.

```
public int DoFib(int n)
{
    if (n == 0)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}
```

보람 : 우선 파라미터 이름을 p에서 n으로 바꾸고, n이 0이면 0을 리턴 하고, 아니면 1을 리턴 하게 코드를 수정하고 다시 테스트를 돌려보자고. 자 이제 테스트를 통과하지. 다음에 2일 때는, 피보나치 수열 순서를 보면, 0 1 1 2 3 5 8 이렇게 되어 있거든. 그러니까 2일때도 결과는 1이어야 하거든.

```
result = fib.DoFib(2);
Assert.AreEqual(1, result);
```

보람 : 그래서 그 부분을 테스트 하도록 이렇게 코드를 추가하고, 테스트를 돌려보면 테스트가 통과를 해. 지금 DoFib메서드를 보면, n이 0이 아니면 1을 리턴 하도록 되어 있는데, 공교롭게도 1일때와 2일일 때 값이 같기 때문이지. 그럼 다음으로 n이 3일때를 테스트 하는 코드를 추가해보자.

```
result = fib.DoFib(3);
Assert.AreEqual(2, result);
```

세식 : 그럼 이제 문제가 되겠네.

보람 : 그렇지 이제 테스트를 돌려보면 실패가 뜨지. 이제 이걸 통과하기 위해서 코드를 이렇게 수정하자고.

```
public int DoFib(int n)
```

```
{
    if (n < 2)
    {
        return n;
    }

    return DoFib(n-2) + DoFib(n-1);
}
```

보람 : 지금 보면, n이 0이나 1일 때는 n값을 그대로 리턴 하면 되고, 그렇지 않을 때는 앞 두 수의 합을 가지면 되니까 코드를 이렇게 수정을 하고 테스트를 돌려보자. 그럼 이제 통과 했지? 자 이제 테스트 코드를 쭉~ 이어 붙여 볼까?

```
[TestMethod]
public void FibTestMethod()
{
    Fib fib = new Fib();
    int result = fib.DoFib(0);

    Assert.AreEqual(0, result);

    //피보나치 수열의 순서
    //0 1 1 2 3 5 8....
    result = fib.DoFib(1);
    Assert.AreEqual(1, result);

    result = fib.DoFib(2);
    Assert.AreEqual(1, result);

    result = fib.DoFib(3);
    Assert.AreEqual(2, result);

    result = fib.DoFib(4);
    Assert.AreEqual(3, result);

    result = fib.DoFib(5);
    Assert.AreEqual(5, result);

    result = fib.DoFib(6);
    Assert.AreEqual(8, result);
}
```

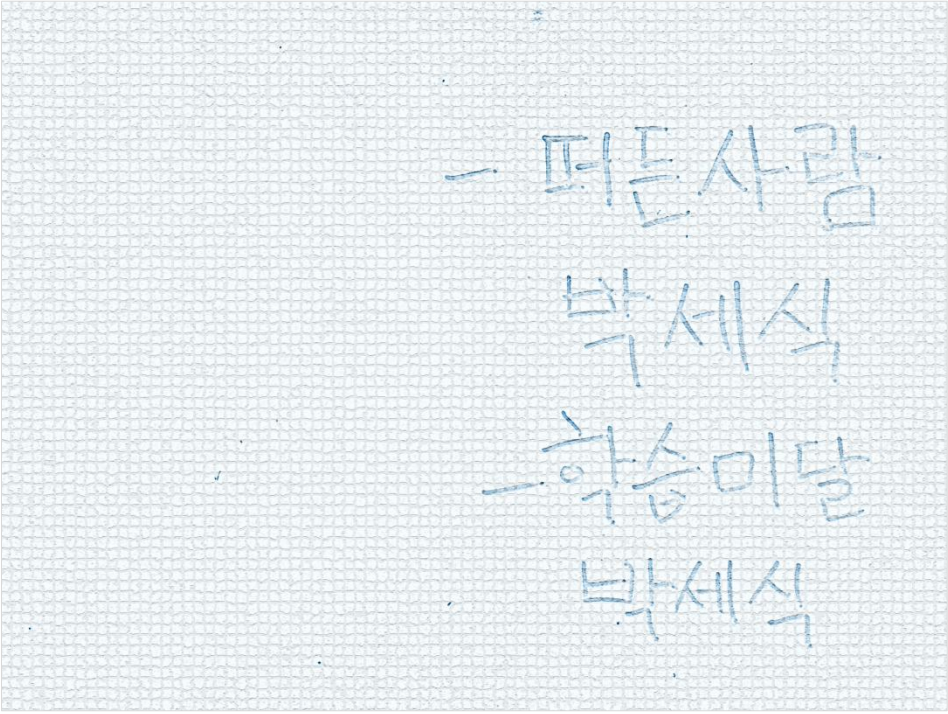
보람 : 이렇게 테스트 코드를 전부 붙여 넣고, 테스트를 돌리면, 자 이제 전부 통과했지. 이런 식으로 테스트를 작성하면서 메서드를 작성하면, 메서드 작성을 완료함과 동시에 이 테스트 코드가 작성한 메서드가 기능을 제대로 수행한다는 것을 보장할 수 있게 되는 거지.

- 정리

TDD를 진행하는 순서는,

1. 완료된 메서드가 충족해야 할 요구사항을 명확히 한다.
2. 요구사항을 만족하도록 단위 테스트를 작성한다.
3. 단위 테스트가 실패할 수 있도록 코드를 작성한다.
4. 단위 테스트가 성공할 수 있도록 코드를 작성한다.
5. 리팩토링

위 단계를 순차적, 순환적으로 통과하고 나면, 메서드가 요구사항을 모두 충족한다고 보장할 수 있다.



세식의 기록 2

날짜 : 2011년 3월 16일
프로젝트 이름 : 아까 그 막장 프로젝트
제목 : TDD 실험 중

보람이랑 TDD에 대해서 이야기 하고 나서 혼자서 많은 생각을 했다. 그리고 너무 많이 생각을 한 탓일까? 회의에서 무심결에 제안을 하고야 말았다.

"저... 저희도 앞으로 개발되는 부분에 대해서는 TDD를 적용해서 해보면 어떨까요?"

그리고 돌아온 대답은,
"뭐? TDD? 야 그게 뭔진 모르겠지만, 언제부터 회의시간에 대리가 말을 했냐? 열심히 듣고 까라면 까는 거 아니었나?"

그래서 나는 입을 다물고, 내 본연의 임무에 충실했다. 그리고 회의가 끝나고, 회사에서 이걸 적용하기 힘들더라도, 혼자서 연습을 해야겠다는 마음을 먹었다. 그리고 ASP.NET을 통해서 몇 가지 시험을 해본 결과 몇 가지 깨달음을 얻었다. 목욕탕에서 이 깨달음을 얻었다면 아르키메데스처럼 유레카라도 외쳤건만. 아쉽다.... 일단 다음 주에 보람이 만나면 열심히 내가 깨달은 거에 대해서 설명 해줘야지. 낄낄낄.

12 일 후

보람 : 와~ 나 강남역 카페베네에는 처음 와바. 맨날 지나다니면서도 한번도 안 와봤네.

세식 : 나도 여기 처음이야.

보람 : 그렇군. 근데 너 이 자식 오늘 보자고 하면서 뭔가 대단한 걸 발견한 것처럼 떠들어 대더니 무신 이야기를 하려고?

세식 : 몰라서 묻는 건 아니겠지?

보람 : 몰라서 묻는 거야.

세식 : 흣. 내 나름대로 TDD에 대해서 생각해봤지. 그리고 어떻게 적용할 수 있을지도 말이지.

보람 : 호~ 그래? 당연히 웹에서 적용해 본 거겠지?

세식 : 당연하지. 그게 전문분야니까.

보람 : 내가 최근에 웹을 좀 소홀히 하느라 거기까진 생각을 못 해봤는데, 기대가 되는 군.

세식 : 좋아. 그렇게 기대가 된다면, 천천히 이야기 해주지. 일단 내가 생각한 시나리오는 우리에게 너무나 친숙한 회원가입, 회원가입을 통해서 TDD의 장점을 알아보려고.

보람 : 오케이 좋아. 예제는 친숙한 것일수록 좋지. 그리고 짧고 굵게 이야기 하고자 하는 핵심을 표현해주는 것일수록 좋고 말이야.

세식 : 그렇지.

보람 : 니 이야기를 듣기 전에 TDD의 장점을 다시 한번 짚고 넘어가자고. TDD의 장점이라면, 완성된 소프트웨어가 가져야 될 기능적 요구사항을 테스트 코드로 먼저 작성하고, 그 테스트 코드를 모두 통과할 때까지 코드를 작성하니까, 요구사항을 만족하는지, 요구사항이 불충분한 건 아닌지에 대한 빠른 피드백을 받을 수 있다는 장점이 있지.

세식 : 정리 잘하는데

보람 : 내가 이걸로 먹고 살잖니

세식 : 좋겠다. 하하하하하하하

보람 : 자 그럼 한번 이야기를 풀어나 봐.

세식 : 그래. 풀어놔 주지.

첫 번째 예제는 웹폼에서 회원가입을 해 볼 거고, 두 번째는 MVC에서, 세 번째는 MVC에서 TDD를 써서 해보는, 이 세가지로 진행해볼까 한다.

보람 : 어 좋은데?

세식 : 바로 시작해도 되나

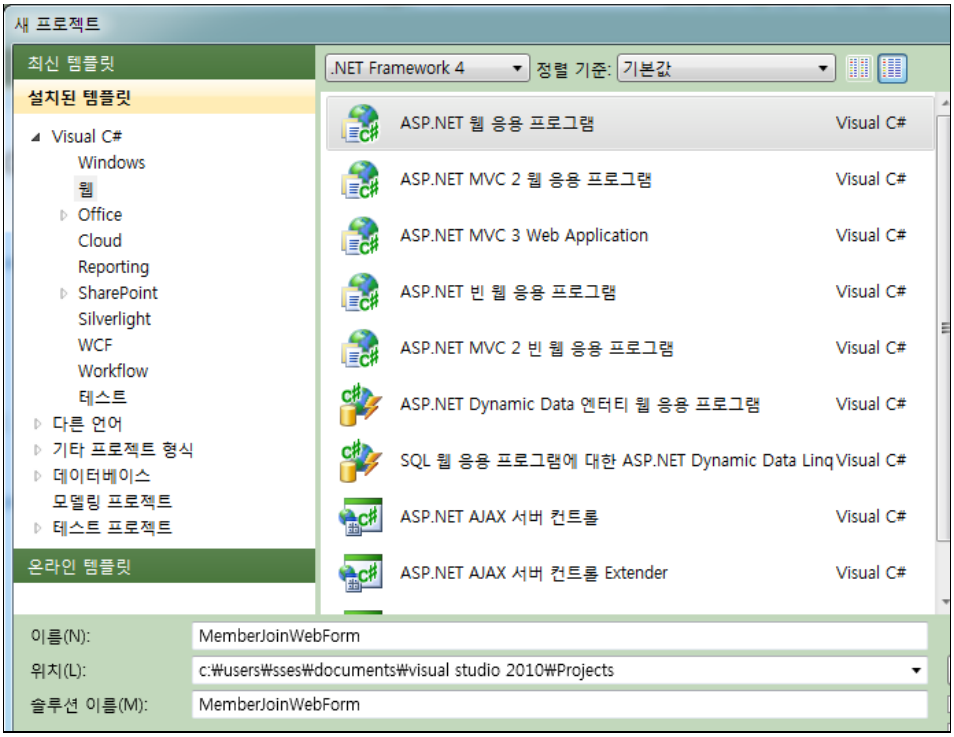
보람 : 하세요.

세식 : 연습은 해봤지만, 이 자리에서 즉흥적으로 해보지. 쉽지 않을 테지만 한번 해보자고

보람 : 오케이. 고고! 고고!

시나리오 1 : ASP.NET 웹 폼에서

세식 : 박수 짝짝) 자 이제 새 프로젝트에서 ASP.NET 웹 응용 프로그램을 선택하고 이름은, MemberJoinWebForm으로



(음악에 맞춰 해드뱅잉 중)

보람 : 오 박자 잘 맞추는데

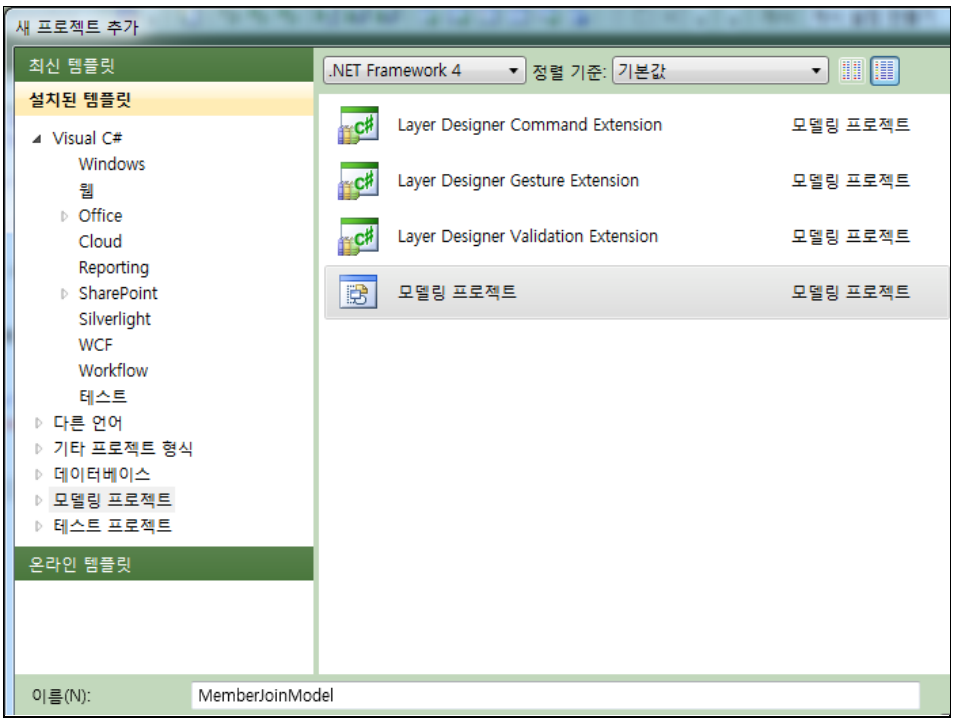
세식 : 훗. 뭐부터 해야 될까? 일단 요구사항부터 정리 해야겠지?

요구사항 수집하기

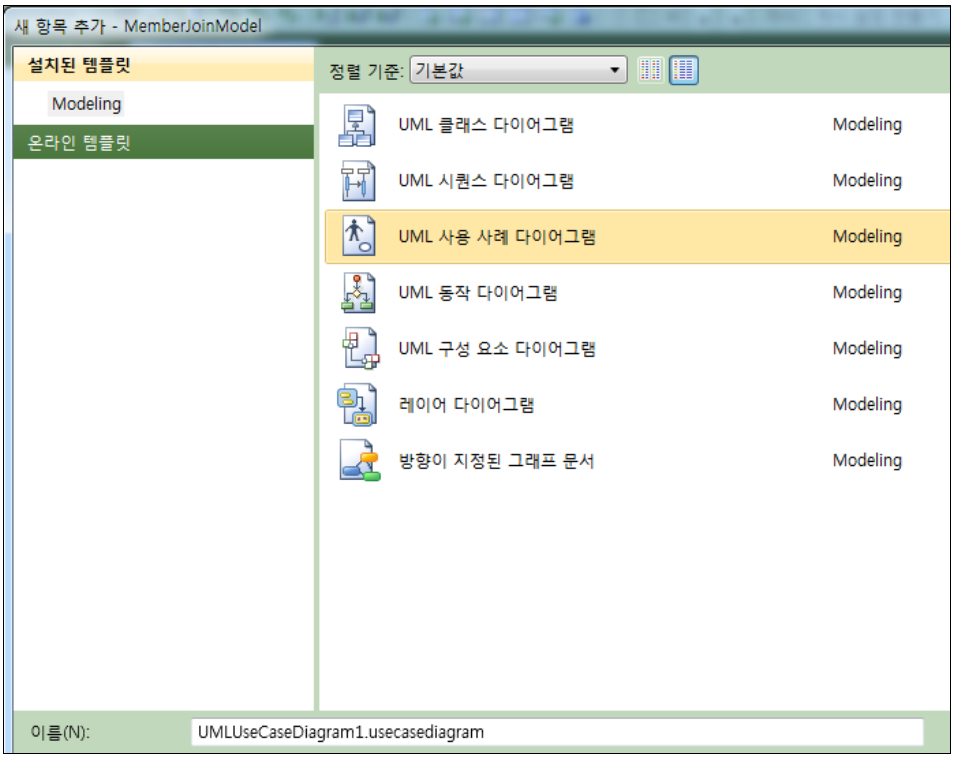
보람 : 오케이. 일단 우리는 Visual Studio Ultimate을 쓰니까. Visual Studio의 설계 기능을 이용하자.

세식 : 오케이. 일단 솔루션에 새 프로젝트 추가, 그리고 모델링 프로젝트로 가서 모델링 프로젝

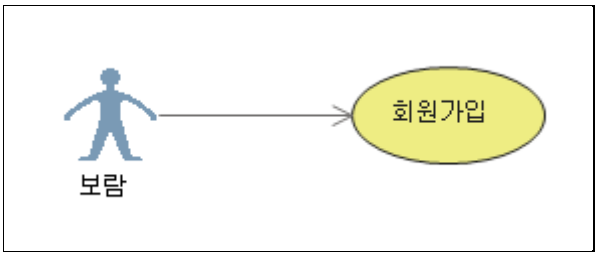
트 추가. 이름은 MemberJoinModel



세식 : 여기서 유스케이스 다이어그램을 하나 추가하고, 거기다가 요구사항을 즉흥적으로 정리해봅시다.



세식 : 도구상자에서... 행위자, 액터 아니에요, 행위자를 추가하고, 이름은.. 보람.

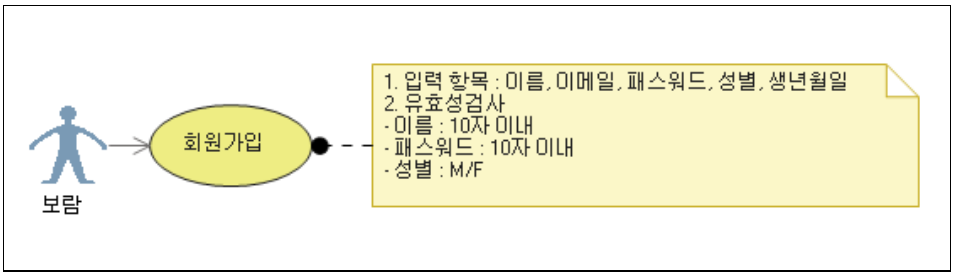


보람 : 얼시구.

세식 : 이 놈이 뭘 요구했냐 하면.. 회원가입을 요구했는데..

보람 : 야. 너 말에 감정이 실렸다.

세식 : 니 착각이겠지. 일단 회원가입에서 요구한 항목은, 이름, 이메일, 패스워드, 성별, 생년월일만 받도록 요구했어. 입력 값에 대한 요구사항은 이름과 패스워드는 10자 이내로 하고, 성별은 뭐 알다시피 둘 중 하나지, 남 앤 여.



보람 : 그렇고만 누군지 몰라도 명쾌한 요구사항 이고만

세식 : 이렇게만 요구를 한 거지

이 요구사항에 맞춰서 페이지를 구성해보자고.

보람 : 잠깐, 요즘에는 협업을 많이 하잖아?

세식 : 그렇지

보람 : 그러니까 UI쪽 개발자랑 서비스 쪽 개발자가 협업을 한다고 생각해보자. 예를 들어서 데이터 액세스 레이어 쪽에서 데이터를 다루는 걸 만들고, UI쪽에서는 자바스크립트 UI 프레임워크나 실버라이트나 뭐 이런걸 가지고 데이터 레이어를 호출해서 데이터를 전송하거나 가져오는 거지. 자 그럼 너는 일단 뒷단 개발자로 회원가입을 위해 필요한 코드를 만들어라~

세식 : 오~ 중간중간에 새로운 제안 감사요~

데이터 액세스 레이어 작성

세식 : 처음에 폴더를 하나 생성할게 모델 폴더.

보람 : 이쁜 모델로 해줘

세식 : 죽여주는 모델로 해줄게, 기대해도 좋아.

먼저 모델 폴더에 MemberRepo라는 이름으로 리파지터리 클래스를 생성할 게

이 리파지터리에서 데이터를 가져오고 저장할 거야. MemberRepo 클래스에 멤버들을 추가해보자고.

보람 : 이 멤버들은 회원가입에 필요한 필드를 이야기는 거지?

세식 : 어 그렇지

첫 번째로 회원데이터를 저장할 리스트를 만들게. 이 리스트는 private으로 선언하고 static으로 구성할게.

```
private static List<Member> _member = new List<Member>();
```

보람 : 아 리파지터리 패턴이라는 걸 써서 데이터베이스 없이도 목록을 관리할 수 있고, 나중에 데이터베이스를 연결해도 인터페이스는 동일하게 유지되니까 코드를 변경할 필요가 없겠군?

세식 : 그렇지.

그리고 이렇게 멤버를 추가를 위한 Add메서드를 하나 추가할게.

```
namespace MemberJoinWebForm.Models
{
    public class MemberRepo
    {
        private static List<Member> _member = new List<Member>();

        public void Add(Member member)
        {
            _member.Add(member);
        }
    }
}
```

보람 : 어? 그럼 이제 Member클래스 만들어야지?

세식 : 어 만들려고 했어. 했다고.

보람 : 만드세요~

세식 : 짜잔~ 이렇게 생겼지. 아까 보람이란 놈이 요구사항에서 요구한 대로 만들었어.

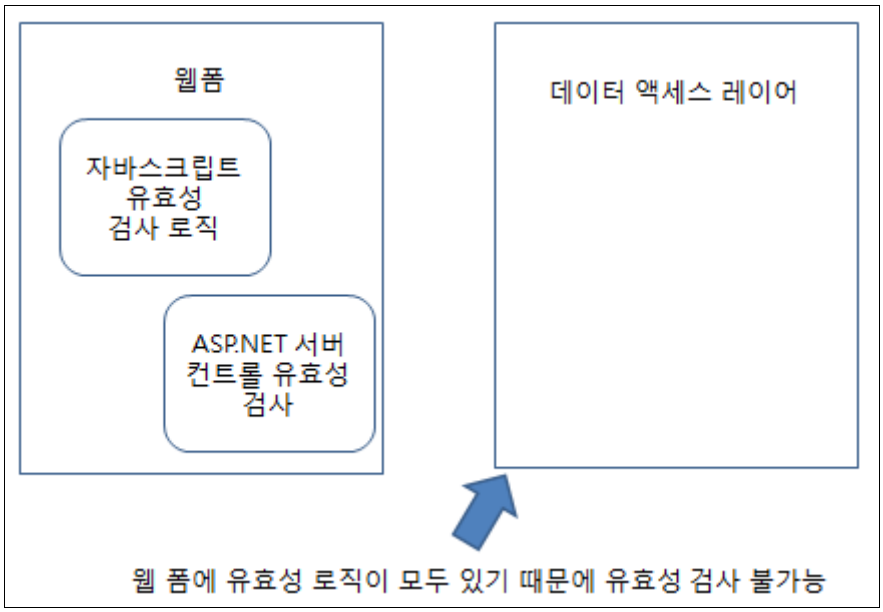
```
namespace MemberJoinWebForm.Models
{
    public class Member
    {
        public Guid Id { get; set; }
        public string Name { get; set; }
        public string Password { get; set; }
        public string Email { get; set; }
        public string Sex { get; set; }
        public DateTime Birth { get; set; }
    }
}
```

보람 : 오케이. 이제 뒷 단을 한번 테스트 해보자.

세식 : 그렇지. 테스트 하고 싶지. 근데 그러려면 웹 폼이 있어야 하지.

보람 : 그렇지! 웹폼이 있어야지. 그래야 테스트가 가능하겠지.

세식 : 당연한 이야기. 일단 아까 회원 가입할 때 입력해야 되는 데이터를 유효성 검사를 해야 되는데, 그것도 웹폼이 없으면 테스트를 할 수가 없어. 왜냐면, 유효성 검사를 하기 위해서는 웹폼에서 자바스크립트를 이용해서 유효성 검사를 해야 되는데, 이것 또한 웹폼 자체가 필요하게 돼. 한 가지 더 유효성 검사 서버 컨트롤을 이용하는 방법도 있는데, 이것 또한 웹폼에서 밖에 못해.



보람 : 그렇군. 그러면 지금 당장 니가 작성한 회원가입 코드가 요구사항을 제대로 만족하는지 알 수 있는 방법이 없군.

세식 : 응. 없는 셈이지.

보람 : 그렇군. 물론 웹폼이 장점이 많긴 하지만, 이게 단점이 아닐까? 그러니까... UI랑 강하게 연결된 부분이 많기 때문에, 웹폼과 분리해서 뭔가를 독립적으로 테스트하거나 하기가 힘들다는 거 말이지.

세식 : 그렇지. 그래서 지금 내가 개발한 회원가입 코드가 제대로 동작하는지 즉시 테스트하기도 힘들고, 그 다음에 유효성 검사가 제대로 동작하는지, 그리고 그 유효성 검사가 적절한 건지 빠른 피드백을 받기가 힘든 점이 있지.

보람 : 그렇군. 그래서 ASP.NET MVC가 나왔지.

세식 : 엉. ASP.NET MVC가 처음 소개되기 시작했을 때, 우리나라에서는 잘 모르겠지만, 외국에서는 단위테스트를 편리하게 할 수 있다는 점이 많은 호응을 받았던 거 같애. 특히 테스트가 쉽다는 점 때문에 Ruby on Rails같은 플랫폼을 억지로 선택하던 사람들도 ASP.NET 플랫폼 상에서 이런 장점을 가져갈 수 있다는 거에 대해서 좋아했던 거 같애.

보람 : 그렇군. 그러면 이제 MVC를 가지고 똑 같은 요구사항을 가지고 예제를 한번 만들어 볼까?

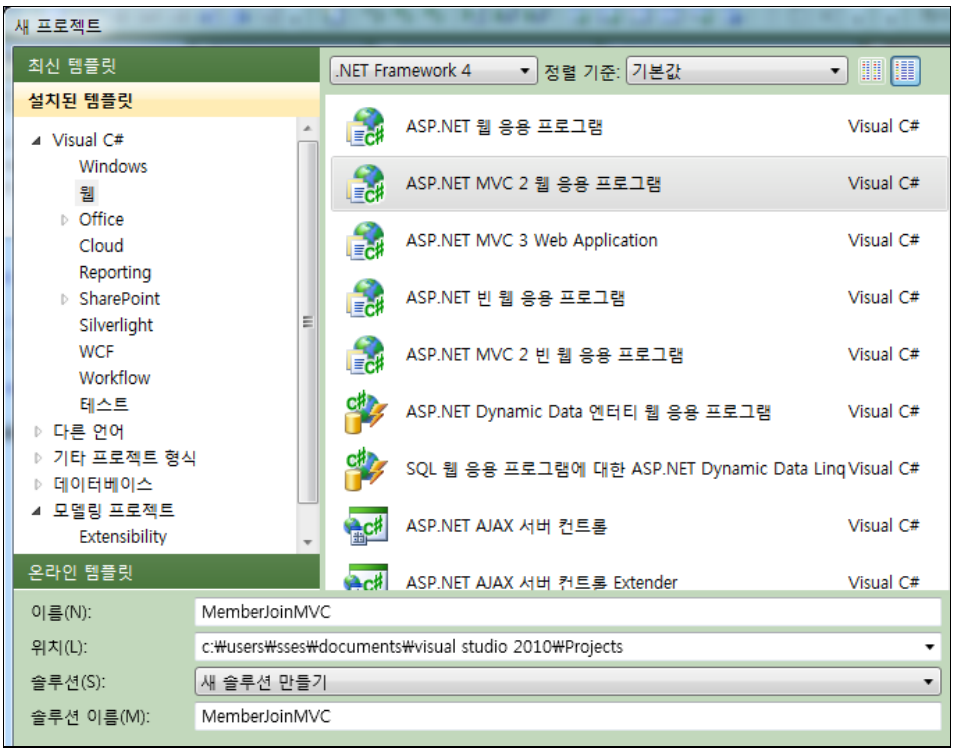
- 정리

ASP.NET 웹 폼에서 TDD를 해보려고 했지만, 요구사항을 제대로 테스트 할 수 없었다. 회원 가입 폼에서 사용자의 입력 값에 대해 유효성 검사를 해야 하는데, 그 유효성 검사 로직은 웹 폼에 존재해야 하기 때문이다. 그래서 이런 구조적 한계를 극복하기 위해서 등장한 ASP.NET MVC 프레임워크에서 같은 요구사항을 테스트 해본다.

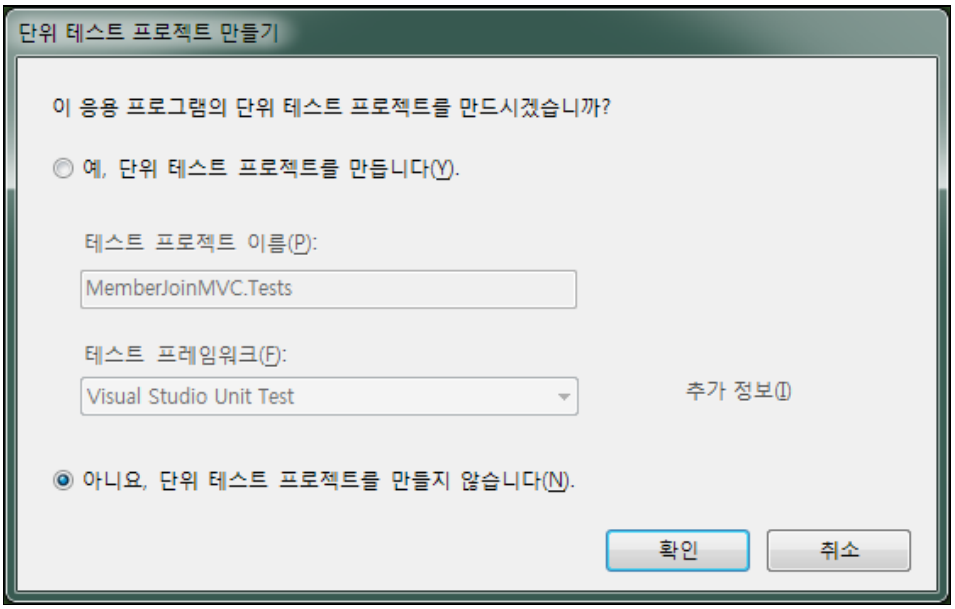
시나리오 2 : ASP.NET MVC

세식 : 오케이. 자 새 프로젝트를 생성하자. ASP.NET MVC 2 웹 응용 프로그램을 선택하고 이름

은 MemberJoinMVC로 할게.



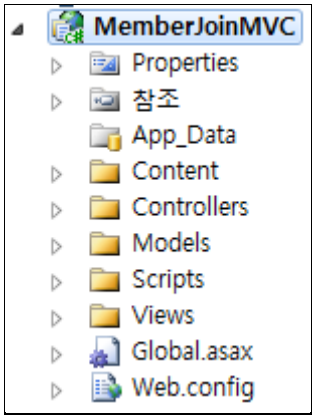
세식 : 단위 테스트 만들기 화면이 나오는데, 일단은 단위 테스트 프로젝트를 만들지 않고 진행해볼게



보람 : 그렇군.

세식 : MVC가 테스트가 용이하다고 했던 이유가, 웹폼은 화면 단하고 되게 강하게 결합되어 있는 상태였잖아. 하지만 MVC는 여기 프로젝트 구조대로 모델과 컨트롤러, 그리고 뷰가 나

눠져 있거든. 그래서 아까처럼 우리가 뷰를 안 만들어도 뒷 단의 로직에서 테스트를 할 수가 있지.



보람 : 고거 괜춘하군

세식 : 이미 모델 폴더가 생성이 되어 있는데, 여기다가 아까 웹폼에서 처럼 리파지터리 클래스를 만들게.

이름은 MemberRepo라고 해서 클래스를 추가. 그리고 내용은 똑같지.(열심히 타이핑) 똑같이 Add 메서드 하나. 멤버 리스트에 새 멤버를 추가하도록.

보람 : 자 이제 멤버 클래스 만들어 야지

세식 : 그렇지. 멤버 클래스는... 아까 작성한 코드에서 복사하자.

음. 됐으.

```
namespace MemberJoinMVC.Models
{
    public class MemberRepo
    {
        private static List<Member> _member = new List<Member>();

        public void Add(Member member)
        {
            _member.Add(member);
        }
    }
}
```

보람 : 아 일단 아까 요구사항을 적었던 모델링 프로젝트를 솔루션에 추가하자.

세식 : 그래. 모델링 프로젝트를 솔루션에 추가해주고, (요구사항을 보면서)이 요구사항 대로 처리하기 위해서 MemberController를 추가할게. MVC에서는 기존의 웹폼과 다르게 직접적인 페이지 호출이 아닌 URL이 'localhost/Member/Add' 이런 식으로 라우팅 룰에 따라서

요청이 들어와. 지금은 디폴트 라우팅을 사용하게 되는데. 그러면 Membercontoller의 Add메서드로 요청이 들어오게 되지.

보람 : 그래서 Add메서드를 만드는 구만

세식 : 그렇지. 이렇게 새 멤버를 인자로 받게끔 파라미터를 주고 View를 리턴 하도록.

이제 기본적인 구조는 다 짠 거고, 이제 유효성 검사를 해야 되는데.

```
namespace MemberJoinMVC.Controllers
{
    public class MemberController : Controller
    {
        public ActionResult Add(Member member)
        {
            return View();
        }
    }
}
```

보람 : 리파지터리 안 붙여도 돼?

세식 : 아. 붙여야지. 하하하하

아까 생성한 리파지터리를 여기 컨트롤러에 우선 선언을 할게.

멤버 컨트롤러가 요청이 될 때 이 리파지터리 객체를 생성을 할 거거든. 그래서 생성자에 멤버 리파지터리를 인자로 넘기고, 여기서 리파지터리 객체를 생성하도록 하는 거지.

```
private MemberRepo _repo;

public MemberController(MemberRepo repo)
{
    _repo = repo;
}
```

세식 : 리파지터리에 Add메서드가 있었잖아. 여기 Add액션 메서드에서 리파지터리의 Add메서드에 새 멤버를 인자로 넘기도록 하는 거지.

여기까지가 1차 작업이 완료된 거지.

```
public class MemberController : Controller
{
    private MemberRepo _repo;

    public MemberController(MemberRepo repo)
    {
        _repo = repo;
    }

    public ActionResult Add(Member member)
    {
        _repo.Add(member);
        return View();
    }
}
```

보람 : 그렇군. 생각보다 간단한데?

세식 : 그래서 이제 아까 보람이 놈이 요구한 요구사항대로 유효성 검사를 추가하도록 할게. 요구사항을 잠깐 보면, 이름은 10자 이내로 하라고 했거든? 그래서 Add 액션 메서드에서 멤버를 추가하기 전에 멤버의 이름이 10자보다 작은지 체크하는 거지. MVC에서는 모델 스테이트라고 하는 객체에 에러를 저장해.

```
public ActionResult Add(Member member)
{
    if (member.Name.Length > 10)
        ModelState.AddModelError("Name", "이름은 10자 이내여야합니다.");

    _repo.Add(member);
    return View();
}
```

보람 : 그러니까 이 모델이라는 게, 결국 저장되는 데이터와 이 데이터가 어떤 형태여야 하는지 정의하는 부분을 말하는 건가?

세식 : 그렇지. 우리가 인자로 넘겨받은 데이터들의 상태를 저장하게 되는 거지.

보람 : 그러니까 모델 스테이트라고 하는 객체에다가 에러를 저장한다고 하는 거는 , 모델의 데이터 형태? 규칙? 을 만족하지 못하는 것에 대한 에러를 저장한다는 거지?

세식 : 응 그렇지.

그래서 지금 이름의 길이를 체크했는데, 이 조건을 만족하지 못하는 걸 AddModelError메서드를 통해서 모델 스테이트에 등록하면... 에러의 키는 Name으로 에러메세지는 '이름은 10자 이내여야 합니다.' 이렇게 에러가 등록이 되면 모델 스테이트의 IsValid라는 속성이 false 값을 가져. 이 속성은 모델의 상태가 유효한지 안 한지 체크해주는 거거든. 그래서 이 모델 스테이트에 에러가 하나라도 등록이 되어있으면 이 IsValid값은 false가 되

겠지. 결국 우리는 이 값이 유효할 때 즉, true일 때 리파지터리에다가 저장을 하는 거지.

보람 : 그럼 유효하지 않을 때는?

세식 : 에러메세지를 보여주는 화면을 다시 그리도록 하는 거지.

```
public ActionResult Add(Member member)
{
    if (member.Name.Length > 10)
        ModelState.AddModelError("Name", "이름은 10자 이내여야합니다.");

    if (ModelState.IsValid)
    {
        _repo.Add(member);
        return View();
    }
    else
    {
        return View();
    }
}
```

보람 : 그렇군. 이렇게 보니까 유효성 검사 로직이 뷰가 아닌 컨트롤러서 수행되니까 방금 봤던 웹폼 과는 다르게 뷰가 없어도 테스트가 가능하겠군

세식 : 원래는 이 유효성 검사를 컨트롤러가 아닌 모델에서 유효성 검사 로직이 추가되거든.

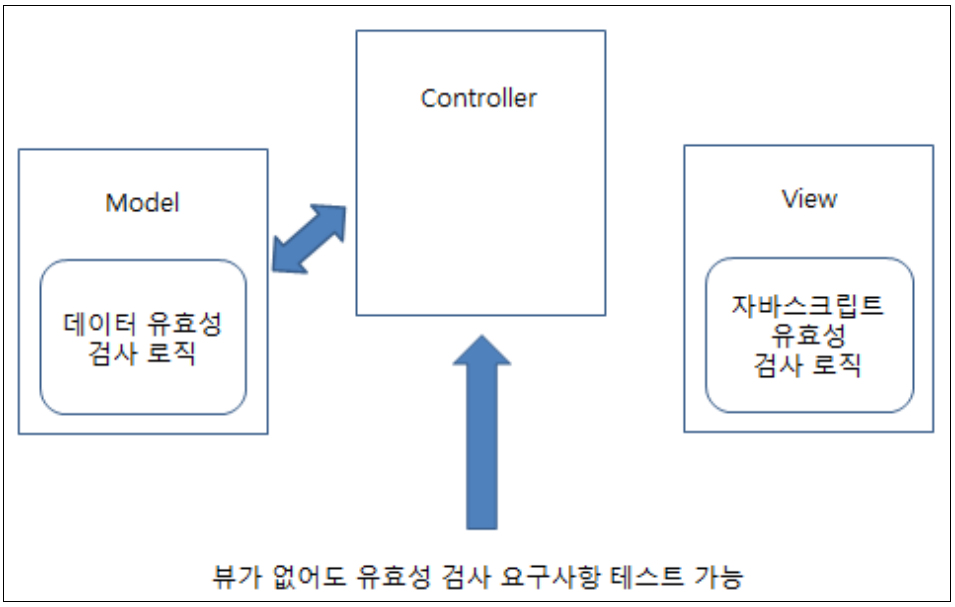
보람 : 근데 여기서는 왜 컨트롤러에서 했어?

세식 : 일단 간단한 설명을 위해서 이렇게 했는데, 그럼 모델에서 수행하도록 유효성 검사 로직을 옮겨볼까?

우리가 아까 만든 멤버 클래스에서 Name필드위에 DataAnnotation을 추가하는 거거든. 그 중에 StringLength라고 있어. 여기다가 10자이내로 받고, 에러메세지를 이렇게 등록하면 되거든.

```
[StringLength(10, ErrorMessage="이름은 10자 이내여야합니다.")]
public string Name { get; set; }
```

보람 : 그렇군 이렇게 하면 유효성 검사를 모델에서 하는 거로구만



세식 : 어. 좀 전에 아까 우리가 컨트롤러에서 하도록 했던 이 유효성 검사 로직을 빼도 되는 거지.

```
public ActionResult Add(Member member)
{
    if (ModelState.IsValid)
    {
        _repo.Add(member);
        return View();
    }
    else
    {
        return View();
    }
}
```

보람 : 그렇군. 일단 프레임워크의 구조상 뷰와 모델이 분리되면서 테스트를 하기 용이한 부분이 있는데, 만약에 프로젝트를 하면서 기존처럼 테스트를 맨 나중에 하는 형식을 따른다면, 결국 이 코드도 지금 당장 피드백을 받지는 못하는 게 아닐까.

세식 : 그렇지. 지금 여기서 테스트를 하려면, 따로 테스트 메서드를 하나 만들어서 현재 이 Add메서드에 대한 걸 테스트해야 하거든.

보람 : 일단 코드가 다 구현된 다음에 테스트를 하는 거니까 피드백은 늦겠지.

만약에 이 코드를 작성하는 도중에, 죽을 만큼 바쁜 일이 생겼다면. 그 테스트는 뷰가 만들어지고 나서야 테스트가 되던지, 아니면 고객이 테스트를 하게 되겠군.

세식 : 그렇지. 바쁘면 개발에만 집중하지 테스트는 뒷전이거든. 그러니까 테스트는 안 한다고

보면 되지.

보람 : 그렇군. 자기 만든 코드가 제대로 돌아가는지 보증할 방법이 없다는 거군.

세식 : 음. 어차피 뷰 페이지에서 다 입력해보는 방법 등을 이용해서 테스트를 하는 거지 뭐. 그런 거지 뭐.

보람 : 좋아. 그럼 이제 TDD를 통해서 동일한 요구사항을 구현해보자고.

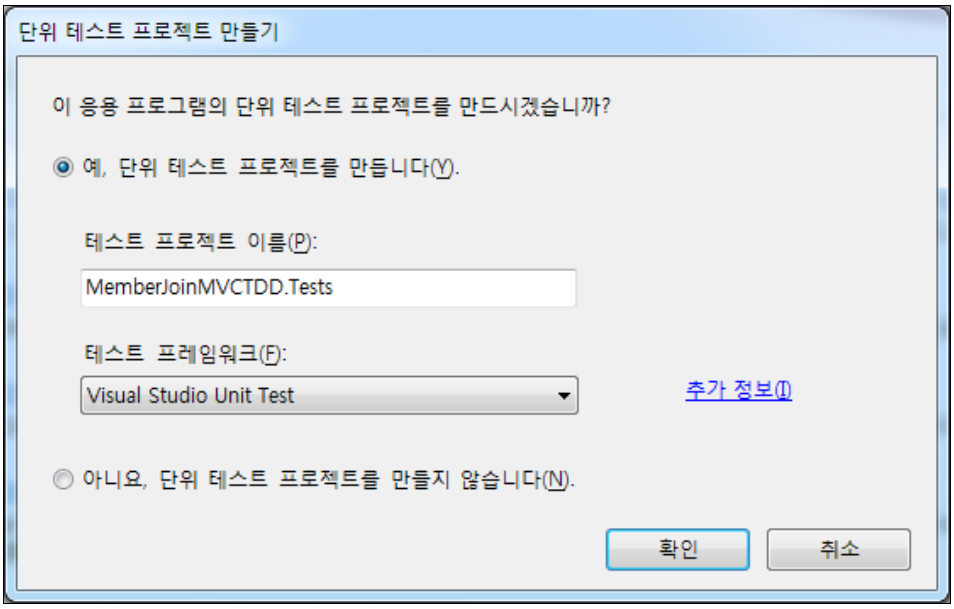
- 정리

ASP.NET MVC 프레임워크에서는 각각 데이터와 데이터의 규칙을 담고 있는 모델(M), 사용자의 URL요청을 받아 어떤 페이지에 어떤 모델을 전달해야 하는지 판단하는 컨트롤러(C), 그리고 전달받은 모델을 가지고 화면을 구성해서 표시하는 뷰(V)로 역할을 분리해 놨다. 그래서 뷰가 없어도 컨트롤러와 모델만 가지고 유효성 검사를 진행 할 수 있다. 하지만, 이런 구조적 장점이 있다고 해도, 개발 프로세스가 테스트를 맨 뒤에 수행하는 구조라면 ASP.NET 웹 폼과 별로 다를 게 없다. 그래서 다음에는 ASP.NET MVC에서 TDD를 통해서 먼저 테스트를 수행하는 것을 해본다.

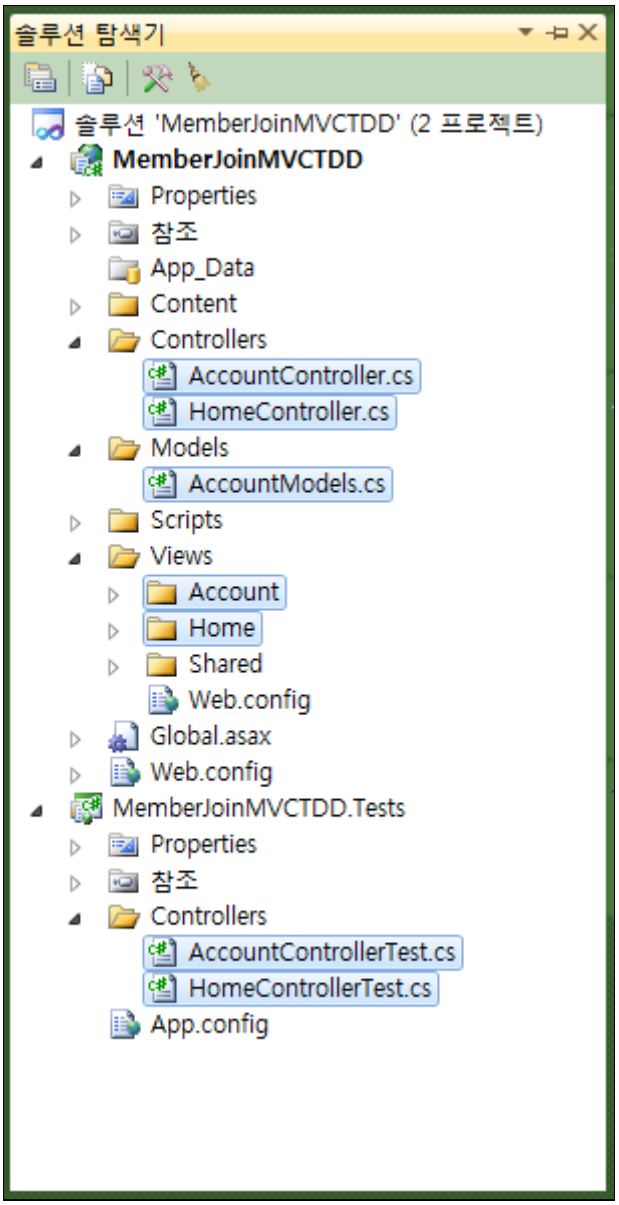
시나리오 3 : ASP.NET MVC + TDD

세식 : ASP.NET MVC2 웹 응용프로그램을 선택하고, 프로젝트 이름은 MemberJoinMVCTDD.

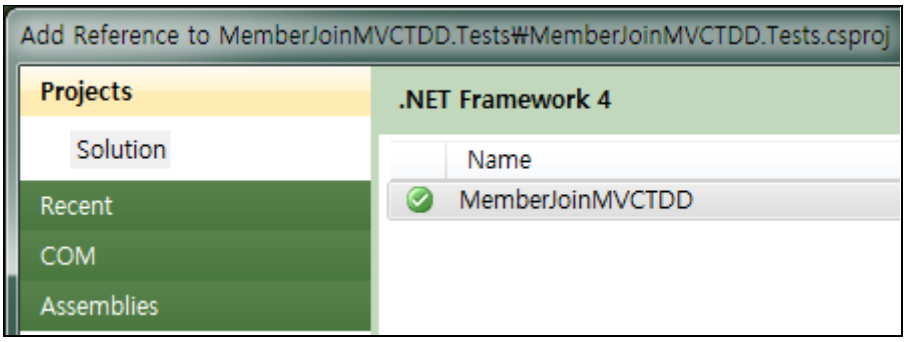
그리고 이번에는 단위 테스트 프로젝트를 추가하자 이름은 기본으로 설정되어 있는 이름을 그대로.



세식 : 프로젝트가 모두 생성이 되었는데 일단 불필요한 파일들은 제거하도록 할게.

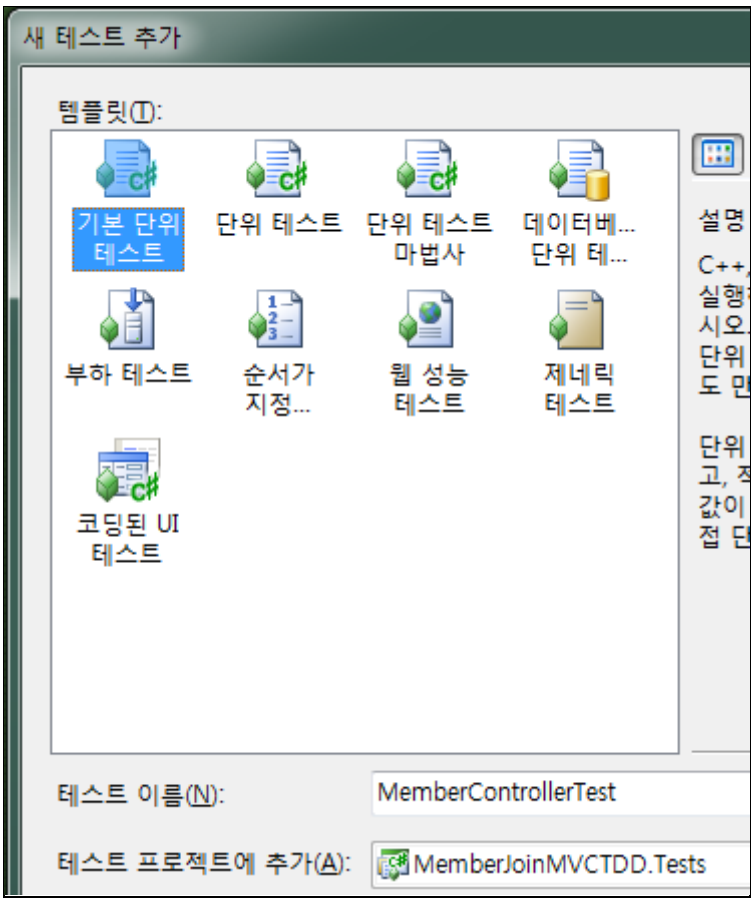


세식 : 테스트는 MemberJoinMVCTDD를 대상으로 수행할 거기 때문에, 참조 추가를 누르고 MemberJoinMVCTDD프로젝트를 참조 추가해주고.



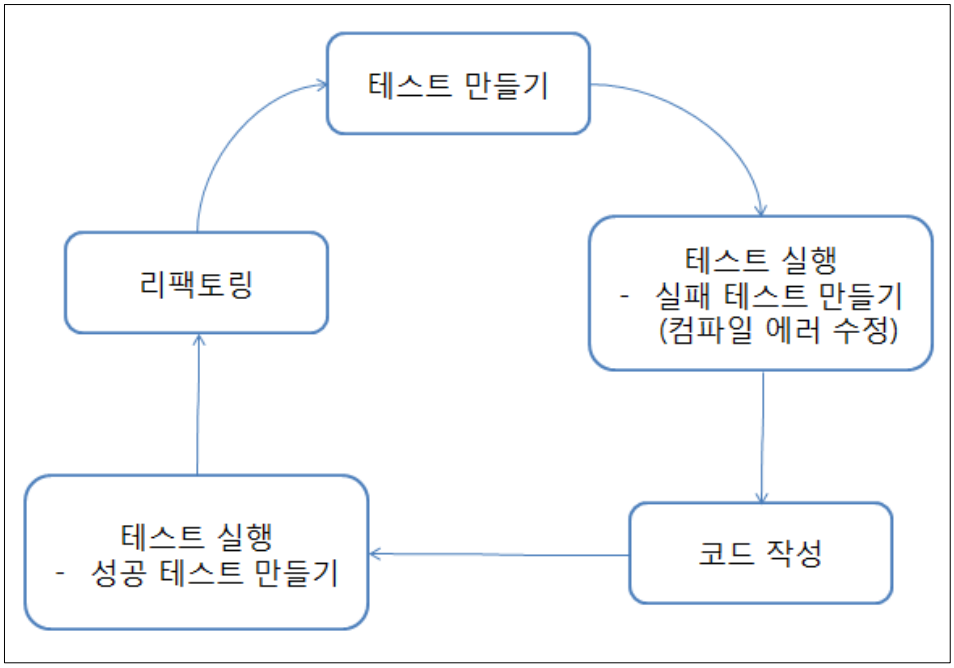
세식 : 일단 MemberController를 테스트할 거기 때문에 테스트 프로젝트의 Controllers폴더에서 새 테스트를 선택하고 기본 단위 테스트를 선택. 테스트 이름은 MemberControllerTest.cs

로 입력하고 확인.



보람 : 자 이제 먼저 테스트를 작성을 하는 거구만.

세식 : TDD개발 순서가 먼저 테스트 코드를 작성한 후에 실제적인 코드를 작성하는 거거든.



세식 : 시작을 해보면, TestMethod1대신에 AddTest로 테스트 메서드 이름을 바꿀게. Add메서드를 테스트 할거거든.

```
[TestClass]
public class MemberControllerTest
{
    [TestMethod]
    public void AddTest()
    {
    }
}
```

세식 : MVC에서는 모든 요청이 컨트롤러에서 제어를 하게 되거든, 그래서 MemberController를 받는 컨트롤러 변수를 하나 만들고, 이 컨트롤러에서 멤버를 추가하는 Add메서드가 있겠지?

```
var controller = new MemberController();
var member = new Member();
```

세식 : 잠깐만. 그전에 멤버 객체를 하나 생성해야 되잖아?

보람 : 그렇지.

세식 : 멤버 객체를 생성하고 이 멤버를 인자로 받는 Add메서드를 호출하게 되면,

보람 : 잠깐만, 호출하고 나서 어떤 페이지로 이동해야 되지?

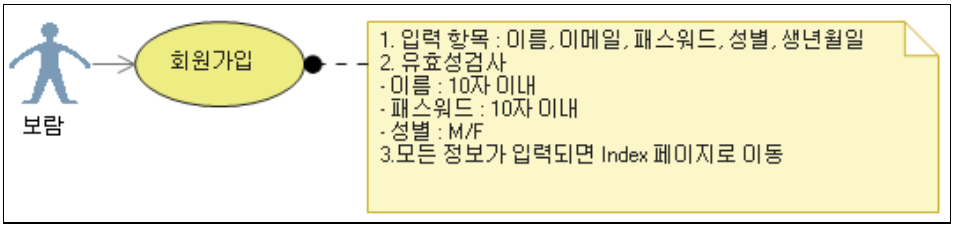
세식 : Index페이지로 이동하겠지?

보람 : 그럼 이것도 요구사항 중에 하나인데, 아까 요구사항 적을 때는 못 넣었었네. 일단 요구사항부터 다시 정리합시다.

세식 : 좋아~ 아까 만들었던 모델링 프로젝트를 다시 솔루션에 추가하고.

보람 : 오케이 요구사항을 받아 적어라~ 나는 갑이니라.

세식 : 하하. 자, 모든 정보가 완벽하게 들어갔을 때는 저장 후에 Index페이지로 이동하게끔 요구사항을 추가하지. 이제 됐나?



보람 : 오케이 됐어. 난 갑이니까.

세식 : 오예.

다시 테스트로 돌아와서 보면, Add메서드를 호출한 뒤에 받는 결과 값이 RedirectToRouteResult타입 이겠군. 왜냐면 저장이 완료된 후에 index페이지로 리다이렉트 시킬 거거든. 그래서 result의 타입이 RedirectToRouteResult인지 먼저 검사를 하고, 그리고 이 result의 이름이 Index인지 검사를 해 볼 거야.

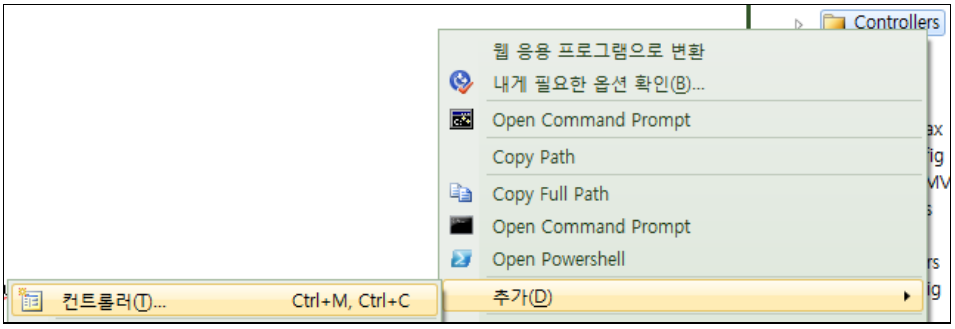
```
[TestMethod]
public void AddTest()
{
    var controller = new MemberController();
    var member = new Member();

    var result = controller.Add(member);

    Assert.IsInstanceOfType(result, typeof(RedirectToRouteResult));

    var typeResult = result as RedirectToRouteResult;

    Assert.AreEqual("Index", typeResult.RouteValues["action"]);
}
```

보람 : 오케이 그럼 먼저, 다른 요구사항 테스트하기 전에 페이지 이동이 제대로 되는지 검사해 보자.

세식 : 좋아. 그런데 TDD에서는 첫 단계가 실패한 테스트를 만드는 건데 지금 보면, 없는 타입 밑에 빨간 줄이 가있잖아? 밑줄을 제거하도록 할게. 그래야 컴파일이 되고, 테스트가 실

패하니까.

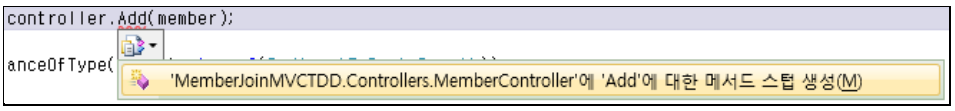
일단 MemberContoller를 추가하자. MemberJoinMVCTDD의 controllers폴더에서 마우스 오른쪽 버튼을 누르고 새 컨트롤러 추가를 선택하고,



세식 : 이름을 MemberController로 하고 추가.

그러면 컨트롤러 클래스가 추가되었고, 다시 MemberControllerTest.cs로 돌아와서 Ctrl + . 을 통해서 MemberController에 대한 참조를 추가하고,

세식 : Add메서드에서 Ctrl + .을 눌러서 스텁 생성을 선택하지.



세식 : 그러면 클래스에 메서드가 생성이 되었지.

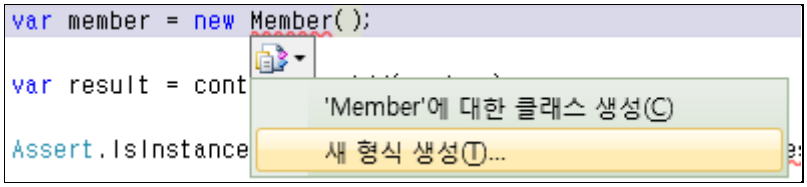
```
public class MemberController
{
    public object Add(var member)
    {
        throw new NotImplementedException();
    }
}
```

세식 : 일단 Add메서드의 파라미터 타입을 var대신에 Member로 고치자. 그리고 Add메서드의 리턴 타입을 ActionResult로 고쳐야지. 그리고 일단 null을 리턴 하게 해두지.

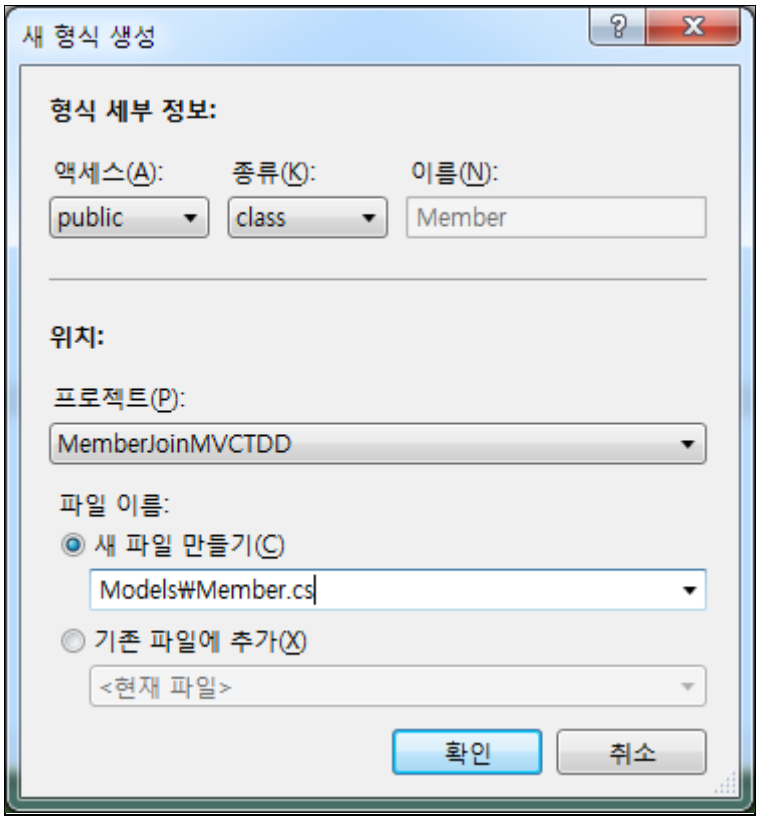
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MemberJoinMVCTDD.Controllers
{
    public class MemberController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }


        public ActionResult Add(Member member)
        {
                return null;
        }
    }
}
```

보람 : 일단 테스트를 실패해야 하니까.

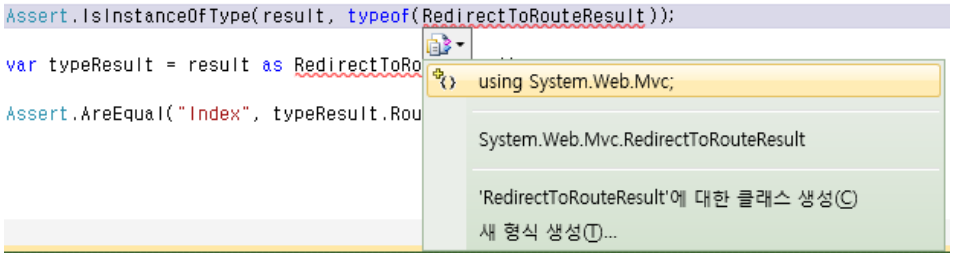
세식 : 이제 멤버 클래스를 만들지.



세식 : Model폴더에 생성되도록.



세식 : 그리고 Add메서드에서도 Member를 찾을 수 있도록 Model네임스페이스를 추가하지. RedirectToRouteResult를 사용하기 위해서 System.Web.Mvc추가하고.

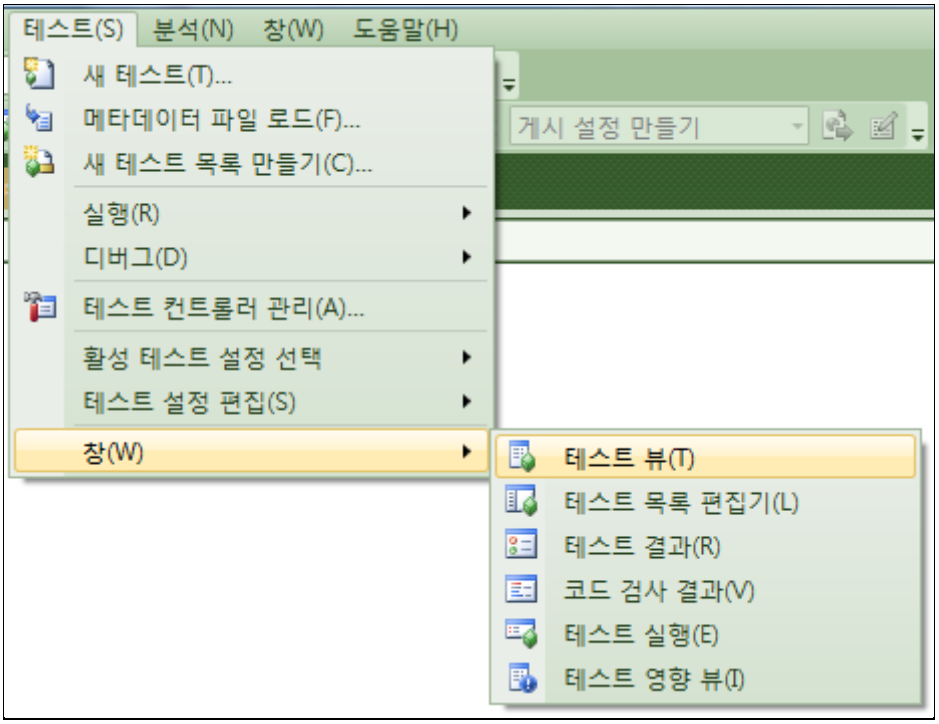


세식 : 이렇게 하면, 이제 테스트를 실행할 수 있게 빨간 줄을 다 없앤 거지.

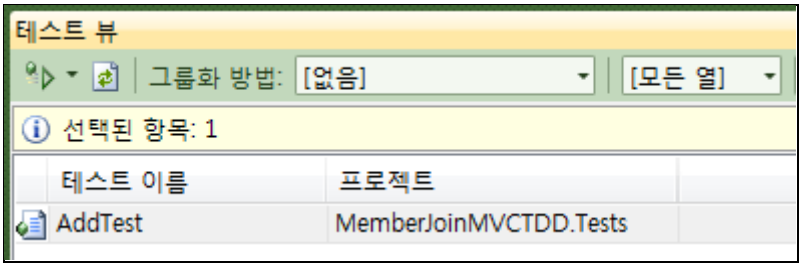
```
[TestMethod]
public void AddTest()
{
    var controller = new MemberController();
    var member = new Member();

    var result = controller.Add(member);

    Assert.IsInstanceOfType(result, typeof(RedirectToRouteResult));

    var typeResult = result as RedirectToRouteResult;

    Assert.AreEqual("Index", typeResult.RouteValues["action"]);
}
```

보람 : 오케이 테스트 발사!

세식 : 자자 진정하고. 우선 테스트를 확인하기 위해서 '테스트' 메뉴에서 '창' 항목 안의 '테스트 뷰'를 선택해서 테스트 뷰를 띄우자.

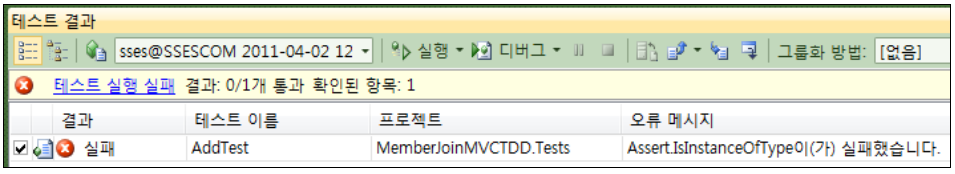


세식 : 그러면 AddTest가 보이지? 이제 테스트 발사!



보람 : 발사!

세식 : 자~ 실패했습니다.



세식 : 실패를 했으니 성공하게 만들어야지. 문제는 뭐냐면~ RedirectToRouteResult타입이어야 하는데 지금은 Add메서드에서 null을 리턴 하고 있지. 그래서 Add메서드에서

RedirectToAction을 사용해서 Index페이지로 리다이렉트 하도록 수정하고 다시 테스트 발사!

```
public class MemberController : Controller
{
    public ActionResult Add(Member member)
    {
        return RedirectToAction("Index");
    }
}
```

보람 : 오 이제 통과. 다음은 뭐지?

테스트 결과

sses@SSESCOM 2011-04-02 13 실행 디버그

테스트 실행 완료 결과: 1/1개 통과 확인된 항목: 0

결과	테스트 이름	프로젝트
통과	AddTest	MemberJoinMVCTDD.Tests

세식 : 이제 유효성 검사 제대로 되는지 테스트 해야지.

새로 테스트 메서드를 만드는데 우선 TestMethod라는 어트리뷰트를 선언을 하고 메서드 이름은 MemberValidationTest. 아직 Member클래스의 내용을 채울 때는 아니니까 일단 MemberController 객체를 생성하고, Member클래스의 Name 필드에 10자 이상의 값을 입력할게.

```
[TestMethod]
public void MemberValidationTest()
{
    var controller = new MemberController();
    var member = new Member();
    member.Name = "12345678901";
}
```

세식 : 이 멤버객체를 인자로 하여 Add 메서드를 호출하고 결과값을 받으면 다시 뷰가 그려진다고 했잖아. 그러니 결과값의 타입은 ViewResult겠고, 에러메시지는 '10자 이내여야 합니다.' 라고 리턴 받겠지. 유효성 검사가 실패하면, 즉 에러가 나면 MVC 에서는 그 에러들이 ModelState 에 등록이 된다고 했었지?

보람 : 어

세식 : 그래서 이 ModelState에서 Name 키 값에 해당하는 모델의 상태정보를 가져와서 에러메시지가 맞게 들어왔는지 확인할게.

```
[TestMethod]
public void MemberValidationTest()
{
    var controller = new MemberController();
    var member = new Member();
    member.Name = "12345678901";

    var result = controller.Add(member);

    Assert.IsInstanceOfType(result, typeof(ViewResult));

    var typeResult = result as ViewResult;

    var nameState = typeResult.ViewData.ModelState["Name"];

    Assert.IsTrue(HasErrorMessage(nameState, "이름은 10자 이내여야합니다."));

}
```

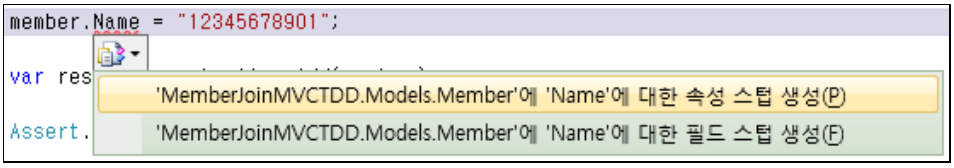
세식 : HasErrorMessage 메서드는 ModelState의 에러들에서 우리가 찾는 에러메시지가 있는지 체크해주는 메서드야.

```
private bool HasErrorMessage(ModelState modelState, string errorMessage)
{
    foreach (var error in modelState.Errors)
    {
        if (error.ErrorMessage == errorMessage)
            return true;
    }

    return false;
}
```

세식 : 여기까지 하고 테스트를 해볼게.

보람 : 오케이. 근데 멤버클래스에 필드 추가해야 되지 않나? 빌드오류 날텐데.

세식 : 오류 나겠지. 그럼 추가해보자고. 빨간 밑줄이 있는 곳에서 컨트롤 점을 입력해서 Name 속성 스텁을 생성할게.



세식 : 그리고 이렇게 된 김에 Member 클래스에 나머지 필드들도 추가해보자.

패스워드, 이메일, 성별, 생년월일..

```
namespace MemberJoinMVCTDD.Models
{
    public class Member
    {
        public string Name { get; set; }
        public string Password { get; set; }
        public string Email { get; set; }
        public string Sex { get; set; }
        public DateTime Birth { get; set; }
    }
}
```

세식 : 오케이 다 입력이 되었으니. 다시 테스트를 진행해볼까?

보람 : 좋아~ 발사!

세식 : 실패했군. 결과값을 ViewResult 로 기대했는데. 지금 MemberController의 Add 메서드는 Redirect 만 하고 있으니 문제가 되는군.

테스트 결과

sses@SSESCOM 2011-04-03 19 실행 디버그

테스트 실행 실패 결과: 1/2개 통과 확인된 항목: 1

결과	테스트 이름	프로젝트
통과	AddTest	MemberJoinMVCTDD.Tests
실패	MemberValidationTest	MemberJoinMVCTDD.Tests

보람 : 그럼 유효성 검사를 추가해서 성공이랑 실패에 따른 분기를 해야지.

세식 : 응. 바로 코드를 작성해볼게.

보람 : 그래. 난 갑이니까.

세식 : 인자로 받은 member 객체의 Name 의 길이가 10자를 넘으면 에러를 등록하고, ModelState의 IsValid 값이 true 이면 Index 액션 메서드로 리다이렉트하고, false이면 다시 뷰를 리턴 하도록 할게. 구현이 되었으니 테스트를 다시..

```
public ActionResult Add(Member member)
{
    if (member.Name.Length > 10)
        ModelState.AddModelError("Name", "이름은 10자 이내여야합니다.");

    if (ModelState.IsValid)
    {
        return RedirectToAction("Index");
    }
    else
    {
        return View(member);
    }
}
```

보람 : 발사!

세식 : 지금 테스트한 것은 통과가 되었는데 기존 테스트에서 에러가 발생했네.

테스트 결과

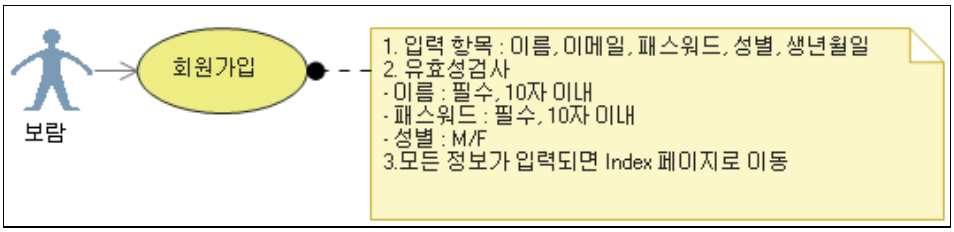
sses@SSESCOM 2011-04-03 19 실행 디버그

테스트 실행 실패 결과: 1/2개 통과 확인된 항목: 1

결과	테스트 이름	프로젝트
실패	AddTest	MemberJoinMVCTDD.Tests
통과	MemberValidationTest	MemberJoinMVCTDD.Tests

세식 : 기존 코드를 보면 그냥 멤버객체만 넘기게 되는데 우리가 수정한 Add 메서드에서는 Name값의 Length를 얻으려고 하다 보니 널 참조 에러가 난 거군. 당연한 건데 Name 값을 필수로 입력 받아야겠지.

보람 : 그렇지. 이 자식, 갑의 요구사항을 정확하게 읽어봐.



세식 : 한대 맞고 할까? 응? 일단 여길 보면, Name 값은 필수. 이 참에 Member 객체의 생성자를 추가해서 객체를 생성할 때 초기값을 입력되도록 수정할게.

```
public Member()
{
    Name     = "Tester";
    Password = "password";
    Email    = "tester@vsts2010.net";
    Sex      = "M";
    Birth    = new DateTime(1988, 1, 1);
}
```

보람 : 지금 이거 그냥 테스트를 위해서 추가 하는 거지?

세식 : 응. 테스트가 완료되면 제거해도 되니까. 그래서 다시 테스트를 실행하면..

테스트 결과

sses@SSESCOM 2011-04-03 19 실행 디버그

테스트 실행 완료 결과: 2/2개 통과 확인된 항목: 0

결과	테스트 이름	프로젝트
통과	AddTest	MemberJoinMVCTDD.Tests
통과	MemberValidationTest	MemberJoinMVCTDD.Tests

보람 : 오~ 모두 통과했군.

세식 : 굿. 이제 진행할 것은 유효성 검사 로직을 컨트롤러에서 모델로 옮겼을 때 동일하게 유효성 검사가 처리되는지 테스트할거야. 모델에는 유효성 검사 로직을 DataAnnotation을 통해서 하게 되는데, 이렇게 말이지.

```
public class Member
{
    [Required(ErrorMessage="이름은 필수입니다.")]
    [StringLength(10, ErrorMessage="이름은 10자 이내여야합니다.")]
    public string Name { get; set; }
    public string Password { get; set; }
```

보람 : 음.. 그렇구만? 테스트 코드는 안 바꿔도 되나?

세식 : 바꿔야 돼.

보람 : 왜?

세식 : 여기까지 진행하고 테스트를 실행하면 실패가나. DataAnnotation 을 추가해서 에러가 발생한다고 해도 모델스테이트에 에러가 등록이 되지 않거든.

보람 : 왜 그렇지?

세식 : DataAnnotation의 속성을 검사해주는 것이 기본 모델 바인더거든. 이 놈의 역할은 우리가 뷰에서 폼을 전송하게 되면 자동으로 관련된 모델에 속성들과 값을 매핑 시켜 주고,

에러가 나면 그 에러를 등록해주는 역할을 해. 근데 이 모델 바인더가 동작을 하려면 웹 요청이 되야 하거든. 무슨 말인지 알겠지?

보람 : 아~ 그럼 인터넷 익스플로러를 통해서 페이지를 요청할 때는 코드에 변경 없이 저 유효성 검사가 동작하는데, 지금은 웹으로 요청하는 게 아니라 단순히 컨트롤러에 요청을 넣은 거라 모델바인더가 동작을 안 했다는 거지?

세식 : 그렇지. 그런 거지. 이제 모델의 유효성 검사를 해줄 메서드를 따로 만들게.

```
private IList<ValidationResult> ValidateModel(object model)
{
    // 유효성 검사 요청 결과 컬렉션 생성
    var validationResults = new List<ValidationResult>();
    // 유효성 검사 수행 컨텍스트 생성
    var ctx = new ValidationContext(model, null, null);
    // 유효성 검사 수행
    Validator.TryValidateObject(model, ctx, validationResults, true);
    return validationResults;
}
```

세식 : 유효성 검사 환경을 만들고, Validator라고 하는 유효성 검사 헬퍼 클래스를 통해서 각 필드의 DataAnnotation 속성을 체크한 결과를 ValidationResult 형식의 컬렉션으로 리턴하는 메서드지.

보람 : 그럼 저 validationResults 라는 컬렉션에는 무슨 값이 들어가는 거지?

세식 : 유효성 검사를 실패한 필드의 에러 메시지와 필드 이름이 들어가.

보람 : 만약에 세 개의 필드가 실패했다면 그 각각의 필드 에러메시지와 이름이 들어가는 거구만.

세식 : 오~ 이해가 빠른데?!

보람 : 내가 좀 하지.

세식 : 아, 그러세요 고갱님? 다음으로 MemberValidationTest() 메서드에 모델의 유효성 검사 수행 후 결과로 받은 컬렉션을 돌면서 모델스테이트에 에러를 등록하면 끝.

```
foreach (var validationResult in ValidateModel(member))
{
    controller.ModelState.AddModelError(
        validationResult.MemberNames.First(), validationResult.ErrorMessage);
}
```

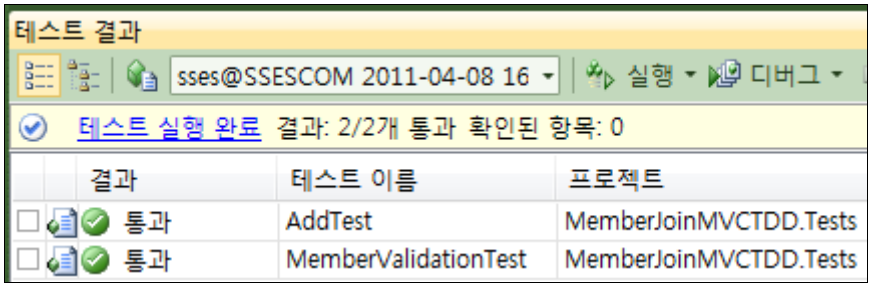
세식 : 그래서 전체 테스트 코드는 이렇게 되지.

```
[TestMethod]
```

```
public void MemberValidationTest()
{

    var controller = new MemberController();
    var member = new Member();
    member.Name = "12345678901";

    foreach (var validationResult in ValidateModel(member))
    {
        controller.ModelState.AddModelError(
            validationResult.MemberNames.First(),
            validationResult.ErrorMessage);
    }

    var result = controller.Add(member);

    Assert.IsInstanceOfType(result, typeof(ViewResult));

    var typeResult = result as ViewResult;

    var nameState = typeResult.ViewData.ModelState["Name"];

    Assert.IsTrue(HasErrorMessage(nameState,
                  "이름은 10 자 이내여야합니다."));

}

private IList<ValidationResult> ValidateModel(object model)
{
    // 유효성 검사 요청 결과 컬렉션 생성
    var validationResults = new List<ValidationResult>();
    // 유효성 검사 수행 컨텍스트 생성
    var ctx = new ValidationContext(model, null, null);
    // 유효성 검사 수행
    Validator.TryValidateObject(model, ctx, validationResults, true);
    return validationResults;
}
```

보람 : 좋아. 테스트 발사!

세식 : 발사!



보람 : 오~ 통과했군.

세식 : 굿. 이런 식으로 패스워드나 다른 필드들도 유효성 검사를 테스트해나가면 돼.

보람 : 그리고 이렇게 테스트를 진행하면서 요구사항이 부족하거나 내용자체가 빠진 부분들을 추가해 나갈 수 있군.

세식 : 그렇지. 이렇게 테스트 코드를 먼저 작성하면서 개발을 하니 요구사항도 분명해지고 개발 자체가 테스트 코드에 맞도록 하는 거니까 이 테스트코드만 모두 통과하도록 코드를 짜면 그것만으로 개발이 완성되었다고 볼 수 있지.

보람 : 그렇군.

- 정리

ASP.NET MVC와 TDD를 결합하면, MVC 패턴이 가지는 장점을 최대한 이용할 수 있을 뿐만 아니라, 요구사항을 먼저 명확하게 하게 된다. 그리고 테스트 코드를 작성하면서 요구사항의 미비한 점을 금방 알 수 있으므로, 요구사항에 대한 빠른 피드백을 받을 수 있다.

에필로그

보람 : 오~ 자네 준비 열심히 했는데? 나도 생각해보지 못한 것들을 많이 본 거 같애.

세식 : 아이 쑥스럽게 왜 그래~

보람 : 그래? 그럼 다음부턴 그냥 욕을 해줄게.

세식 : 아이쿠 감사합니다.

보람 : 그래 감사해야지. 아무튼, 이번에 니가 준비한 TDD예제를 지켜보면서 더 확실하게 느낀 거지만, 역시 TDD는 잘 수행하기만 하면, 빠른 피드백을 받는데 큰 도움이 되는 거 같애. 위 예제를 보면, 우리가 즉흥적으로 하느라 요구사항이 엉성하다가 실제로 테스트를 작성하는 와중에 요구사항이 조금씩 더 구체화 되고 있거든. 근데 이게 더 실제 개발 모델에 근접한 거 같애. 어차피 요구사항은 한번에 다 나오는 것도 아니고, 실제로 하려다 보면 더 구체화 되는 거거든.

세식 : 정말 해보니까 확실히 TDD 가 이러한 점에서 좋다고 느껴져. 원래 요구사항이란 게 뭔가 명확하지도 않고 계속 추가되고 수정되는데 TDD 기반으로 하면 바로 결과를 알 수 있으니까.

보람 : 테스트를 작성하려면 요구사항을 명확하게 알아야 하니까 더 확실하게 정의하는 면도 있고, 테스트 작성 중에 요구사항의 미비한 점을 코드 작성 전에 바로 알 수 있고, 그리고 코드 작성 후에 요구사항을 제대로 만족하는지 바로 알 수 있고.

세식 : 역시 정리하는 것에는 탁월하구먼~

보람 : 허허허. 그렇지. 내가 쫌 하지.

세식 : 아이고 그러세요 고갱님?

보람 : 훗. 일단 다시 이야기로 돌아가보면, 스크럼 같은 방법론이 많은 관심을 받고 있고, 그리고 성공사례도 많이 접할 수 있는 건 스크럼은 말 그대로 애자일한 방법론이기 때문이거든. 현재 빗발치는 요구사항 중에서 일정 주기 동안 개발 가능한 것들을 고르고, 그것들을 개발하면서 추정치를 바꿔나가고, 그리고 한 주기가 끝나면, 그 주기에서 만든 걸 보여줘서 고위 간부나 클라이언트가 빠른 판단을 할 수 있도록 도와주고 말이지. 덤으로 개발팀이 합심해서 삽질을 하더라도 한달 정도의 주기 마다 금방 알 수 있으니, 또한 빠른 방향 수정을 할 수 있도록 해주지. 그런 면에서 이런 TDD도 개발 중의 요구사항이 미비하진 않은지, 그리고 코드가 제대로 기능을 만족하는지 빠르게 알 수 있도록 해서, 의사소통을 돕고, 협업을 돕는 요소가 될 수 있을 거 같애.

세식 : 그렇지. 내 피와 땀이 서린 일기장을 봤겠지만, 갑 쪽에서 오케이 오케이 하고 넘겨가며 작업을 진행했는데 결국에는 전혀 다른 것을 원하고 있더라고.

보람 : 그렇지. 나도 그런 경험이 있는데, 결국 중간발표 때 클라이언트랑 우리 쪽 대표랑 언성을 높이면서 싸우더라고. 우리 쪽 대표는 '제안서에 다 있는 내용이었다'라는 거고, 클라이언트는 '이런 거 인줄 알았으면 투자 안 했다'라고 말이지.

세식 : 휴. 클라이언트가 요구하는 것을 명확하게 알 수 있으면 좋은데. 그렇게 하지 못하니까 이렇게 TDD로 개발을 함으로써 클라이언트와도 소통을 통해서 하나하나 발전시켜 나가는 게 좋은 듯?

보람 : 그러면 좋겠지. 하지만, 우리나라 실정에 잘 들어맞지는 않아서 말이지. 빨리 코드가 돌아가는 걸 봐야 하는데, 테스트 코드 작성하고, 테스트 코드 작성에 필요한 요구사항 분석하고 있으면 좋아할 사람은 없겠지.

세식 : 그렇지. 정말 클라이언트 옆에 끼고 개발하는 게 좋은데.

보람 : 좋긴 하지만, 너무 힘든 이야기고. 일단 뭐 꼭 TDD까지는 아니더라도, 개발자가 자기가 작성한 코드에 대해서 단위 테스트 만드는 것 정도까지 하면 정말 좋을 거 같애. Visual Studio 같은 툴에서 잘 지원을 하고 있기도 하고, 이제 웹에서도 단위 테스트 만들기 좋게 ASP.NET MVC같은 프레임워크도 나왔고 말이지.

세식 : 그렇지. 결국은 개발자 마인드 자체가 변해야겠지.

보람 : 근데 뭐, 딱히 누굴 위해서 하자는 것도 아니고. 어차피 나중에 득을 보는 사람은 개발자가 되거든.

세식 : 그렇지. 결국에는 내가 편한 거니까. 근데 정말 편하다고 생각해도 미루게 되는 건 어쩔 수 없나 봐. 훗.

보람 : 그러니까 뭐 냐면. 비유를 하나 들어보지. 이런 경우 많을 거야. 뭐 가스 불을 끄고 나왔는지 신경이 계속 쓰인다거나, 문을 잠그고 나왔는지 계속 신경이 쓰이는 거 말이지.

세식 : 응. 그런 경우 많아.

보람 : 결국 확실하게 하는 방법은 테스트를 해보는 거거든. 가스 불을 껐는지, 문을 잠궜는지. 안 그러면 계속 불안한 마음을 가지고 있게 되지. 단위 테스트가 그런 게 아닐까. 내가 만든 코드가 제대로 돌아가는지 불안한 마음을 가지는 대신에, 테스트를 통해서 확실하게 하는 거지. 각 메서드가 제대로 돌아가는지 말이야.

세식 : 그래. 근데 정말 자기 자신에게 관대한 개발자 많은 듯. 나도 포함되는 건가? 내가 작성한 코드에 불안한 마음이 잘 안 들어. 하하.

보람 : 그렇지. 그게 '설마 무슨 문제 있겠어?'하는 마인드가 아닐까. 위에서 들었던 예를 보자면, 저렇게 걱정이 되는 경우 중에 실제로 안 끄거나, 안 잠근 경우는 많지 않을 거야. 근데, 마침 문을 안 잠근 때에 도둑님께서 다녀가신 다면 대박 터지는 거지.

세식 : 하하. 그렇군. 항상 확실하게 해야 하는 거군.

보람 : 그게 모두를 위해서, 그리고 자기 자신을 위해서 좋은 거지. 근데 이야기가 너무 길어 지는 거 같은데, 이제 이쯤 정리해야 될 거 같다.

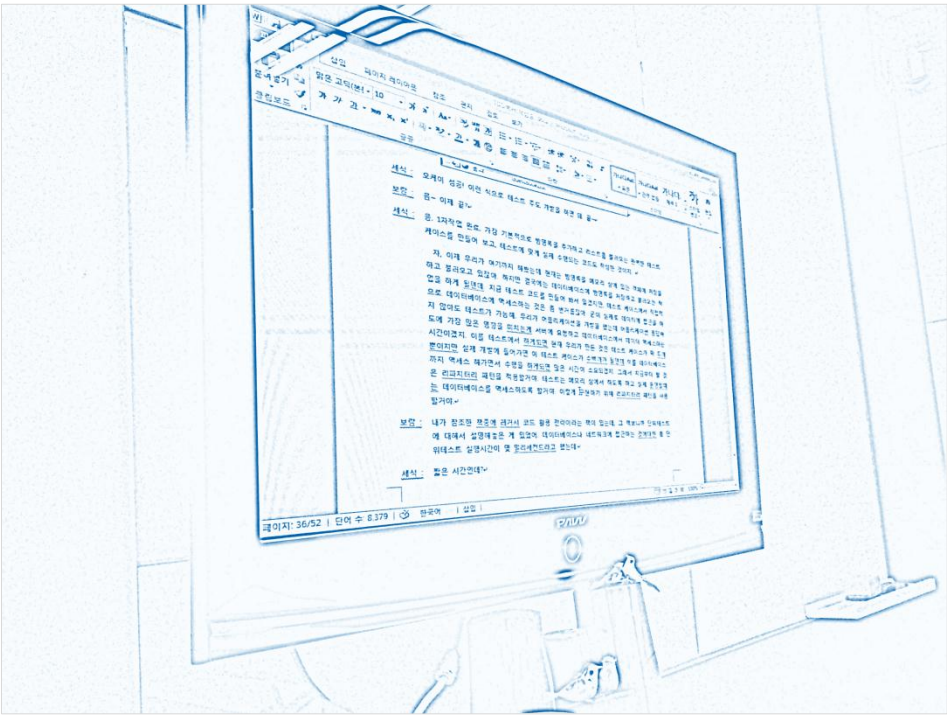
세식 : 그래. 좀 길어지는 듯 해 보인다. 너의 장점인 정리능력을 무한하게 보여주렴~

보람 : 이미 정리는 다 했으니까 또 하면 내용 채우려고 했던 얘기 또 한다고 할 거 같고. 여기까지 하자. 하하하.

세식 : 하하. 오케이. 저번에는 고생했다고 마무리 했지만. 왠지 모를 뿌듯함이 밀려오는데?

보람 : 그러냐. 그 뿌듯함을 일터에서 맘껏 발휘하셔.

세식 : 오케이!

보람 : 그래 고생했다.

세식 : 결국은 '고생했다'로 마치는군. 너도 고생했다. 하하.