올리는 말

테스트 주도 개발(Test-Driven Development)을 학습하고자 하는 분들을 위해 출판물 "**테스트 주도 개발: TDD 실천법과 도구**"(2쇄본)의 일부를 공개합니다. 처음 TDD를 시작하시는 분들이 인터넷 검색 등으로 알음알음 시작하시는 것 보다는 그래도 이 편이 좀 더 낫지 않을까 하는 마음에 자료를 올리게 되었습니다. 비록 작은 나눔이고 부족한 노력의 일환이지만 보시는 분들께 조금이나마 도움이 되었으면 좋겠습니다. 그리고, 인쇄해서 보시는데 방해가 되지 않도록, 추천사, 서문, '이 책을 어떻게 읽을 것인가' 등등의 앞 부속은 제외하였습니다. 대신, 전문가 인터뷰 자료, 첨언이 들어간 '추천사'등의 추가 자료는 tddbook.com에 게재되어 있으니 자유롭게 이용해 주세요.

참고로, **본 문서의 상업적 이용이나 blog.doortts.com, tddbook.com 이외의 사이트를 통한 배포 및 무단 전재는 금지합니다.** 기타 문의사항이 있으시면 아래 메일 주소를 이용해 주세요.

그럼, 언제나 늘 건강하시고, 행복하세요.
감사합니다.



2011.02.01. 채수원 올림

Contact
doortts@Gmail.com
BLOG: blog.doortts.com
Twitter: @doortts

for non-commercial purpose only

테스트 주도 개발

고품질 쾌속개발을 위한

TDD

실천법과 도구

한빛미디어 Hanbit Media, Inc.

for non-commercial purpose only

채수원 tddbook@gmail.com

LG CNS에 시스템 엔지니어로 입사 후 10년 동안 다수의 프로젝트에서 다양한 역할을 수행하고 있다. 현재는 LG CNS 경영기술교육원 기술교육팀 전임강사로, 분석설계 관련 과목과 디자인 패턴, 리팩토링 등의 과목을 강의하고 있다.

또한 보다 나은 미래를 함께 만들어 갈 수 있다는 믿음 하에 사내에 애자일 방식을 소개하고 직접 프로젝트에 참여해 실천하며 이를 바탕으로 교육을 만들어 전파하는 등 사람을 중시하는 개발 문화를 SI 현장에 확산하기 위해 노력하고 있다.

책 관련 질문 및 자료 사이트: tddbook.com

테스트 주도 개발: 고품질 쾌속개발을 위한 TDD 실천법과 도구

지은이 | 채수원
펴낸이 | 김태헌
펴낸곳 | 한빛미디어(주)
주 소 | 서울시 마포구 서교동 480-26 한빛빌딩 2층 IT출판기획부
전 화 | IT전문서팀 02)325-0984, 영업1팀 02)336-7114
팩 스 | 02)336-7124
등 록 | 1999년 6월 24일 제10-1779호
초판 발행 | 2010년 6월 17일
2쇄 발행 | 2011년 1월 25일
정 가 | 30,000원
ISBN | 978-89-7914-726-1 93560

기 획 | 조연희
편 집 | 너구리 엔터테인먼트

Published by HANBIT Media, Inc. Printed in Korea

Copyright © 2010 by **채수원** & HANBIT Media, Inc.
이 책의 저작권은 채수원과 한빛미디어(주)에게 있습니다.
저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

이 책에 대한 의견을 주시거나 오탈자 및 잘못된 내용의 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 연락주십시오. 잘못된 책은 구입하신 서점을 통해 교환해 드립니다.

http://www.hanb.co.kr
ask@hanb.co.kr

for non-commercial purpose only

목차

추천사 3
저자서문 6
이 책을 어떻게 읽을 것인가? 8
베타 리더 소개 10

1 테스트 주도 개발

1.1 흔하디 흔한 소프트웨어 개발 방식 ····· 20
1.2 테스트 주도 개발(TDD) ····· 24
1.3 테스트 주도 개발의 목표 ····· 27
1.4 테스트 주도 개발의 기원 ····· 27
1.5 개발에 있어 테스트 주도 개발의 위치 ····· 29
1.6 테스트 주도 개발의 진행 방식 ····· 30
1.7 실습 먼저 시작해보기 ····· 32
1.8 TDD의 장점 ····· 89

2 JUnit과 Hamcrest

2.1 JUnit ····· 96
2.2 비교표현의 확장: Hamcrest ····· 134
2.3 정리 ····· 149
전문가 인터뷰 김창준 ····· 151

3 TDD 좀 더 잘하기

3.1 테스트 케이스 클래스의 위치 ····· 155
3.2 테스트 메소드 작성 방식 ····· 170
3.3 테스트 케이스 작성 접근 방식 ····· 172
3.4 TDD의 한계 ····· 175
전문가 인터뷰 변신철 ····· 179

for non-commercial purpose only

4 한계 돌파를 위한 노력, Mock을 이용한 TDD

4.1 Mock 객체 ····· 186
4.2 Mock에 대한 기본적인 분류 개념, 테스트 더블 ····· 191
4.3 Mock 프레임워크 ····· 215
4.4 Mock 프레임워크 마무리 ····· 248
전문가 인터뷰 황상철 ····· 253

5 데이터베이스 테스트: DbUnit

5.1 DbUnit의 장점 ····· 257
5.2 데이터셋 ····· 259
5.3 DbUnit 데이터셋의 종류 ····· 271
5.4 DbUnit의 DB 지원 기능 ····· 274
5.5 DbUnit과 Ant ····· 276
5.6 정리 ····· 282
전문가 인터뷰 안영회 ····· 283

6 단위 테스트 지원 라이브러리: Unitils

6.1 Unitils를 사용하기 위한 환경 준비 ····· 288
6.2 Unitils의 단위 테스트 지원 기능들 ····· 290
6.3 Unitils 모듈 ····· 299
6.4 DbUnit과 함께 사용하는 데이터베이스 지원 모듈 ····· 302
6.5 DBMaintainer: DB를 자동으로 유지보수해주는 DB 유지보수 관리자 ····· 310
6.6 기타 지원 모듈들 ····· 314
전문가 인터뷰 박재성 ····· 318

for non-commercial purpose only

7 개발 영역에 따른 TDD 작성 패턴

7.1 일반적인 애플리케이션 ····· 324
7.2 웹 애플리케이션 ····· 341
7.3 데이터베이스 ····· 377
7.4 안티패턴(anti-pattern), 전통적으로 잘못 인식되어 있는
테스트 메소드의 리팩토링 ····· 385
전문가 인터뷰 이일민 ····· 391

8 TDD에 대한 다양한 시각

8.1 TDD와 소프트웨어 디자인 ····· 395
8.2 TDD 유의사항 ····· 406
8.3 TDD와 리팩토링 ····· 409
8.4 TDD와 짝 프로그래밍 ····· 410
8.5 TDD와 심리학 ····· 411
8.6 TDD를 어렵게 만드는 요인 ····· 414
8.7 행위 주도 개발 ····· 420

9 자주 접하게 되는 질문들, FAQ

10 실습 예제

10.1 자동판매기 잔돈 계산 모듈 ····· 459
10.2 비디오 가게 ····· 477

for non-commercial purpose only

11 테스트 자동화와 커버리지

11.1 테스트 케이스 수행 자동화: Ant 491
11.2 테스트 커버리지 510

부록 A 본 책보다 더 가치 있을 수도 있는 '부록'

A.1 놓치기엔 아까운 JUnit 4의 고급 기능들 529
A.2 스프링으로 JUnit 기능 확장하기 553
A.3 코드 리뷰 556
A.4 3분 안에 Java DB 설치하고 확인까지 마치기 573
A.5 TDD에 대한 연구 보고서(테스트 주도 개발을 통한 품질 향상 실체화) 577
A.6 TDD 학습에 도움이 되는 책과 문헌 579

찾아보기 580

for non-commercial purpose only



만일 당신이 때때로 실패하지 않는다면, 그건 안이하게 살고 있다는 확실한 증거다.

– 우디 앨런

for non-commercial purpose only

CHAPTER 1

테스트 주도 개발

1장에서는 전통적인 소프트웨어 개발의 모습을 떠올려보고, 어떠한 문제점이 있는지 살펴본다. TDD는 그중 어떤 부분을 보완하고 있으며, 적용 시의 장점은 무엇인지 살펴본다. 또한 TDD의 진행 방식을 간단한 실습과 함께 따라 해나가면서, TDD 기본을 익혀본다.

안내

각 장 도입부에는 해당 장에서 학습할 주제에 대한 체크리스트를 만들어뒀다. 충분히 학습하거나 이해한 부분에는 ✓ 표시를 해놓고, 어렵거나 다시 볼 필요가 있는 부분은 남겨둬 나중에라도 다시 살펴보자.

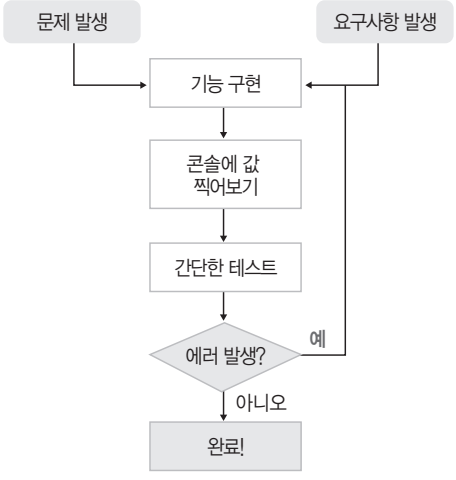
1장 체크리스트

- ☐ TDD의 정의
- ☐ TDD의 목표
- ☐ 개발에 있어 TDD의 위치
- ☐ TDD의 진행 방식
- ☐ TDD의 장점

for non-commercial purpose only

1.1 흔하디 흔한 소프트웨어 개발 방식

소프트웨어를 개발할 때 흔히 사용하는, 전통적이라 불릴 만큼 고전적인 개발 방식에 대해 살펴보자. 일반적으로 '소프트웨어 개발'이라는 작업은 특정한 무언가가 불편하거나 필요하다든가, 또는 어떤 것이 잘못되었다든가 하는 식의 **'문제영역'**이 그 시발점이 된다. 그리고 자의든 타의든 문제영역을 인지한 개발자는 그 문제를 해결해주는 '기능을 구현'한다. 그리고 일정 시점이 지나면 구현의 **'검증'**을 위한 테스트를 수행한다. 이때 '검증'을 위해 사용하는 대표적인 방법은 콘솔(console) 화면에 값을 찍어보는 방식이다. 아주 고전적이면서도, 어찌 보면 눈에 보이는 확실한 방법이기에 흔히 사용된다. 콘솔 화면에 찍힌 글자를 보고, 정상적으로 기능이 동작하는지, 혹시 문제가 있는지, 문제가 있다면 어떤 문제인지를 찾아낸다. 뭐, 나쁘지 않은 선택이다. 여태까지 쭉 그래왔고, 아직까진 이 방식을 사용하는 데 있어 충격적이라고 할 만한 문제점은 겪지 않았다. 다만 안타까운 점 한 가지는, 대개 이런 경우 작성된 코드의 문제 유무 판단을 개발자 자신의 두뇌에 상당 부분 의존하게 된다는 사실이다.



for non-commercial purpose only

하지만 사람의 머리란 얼마나 얄팍하고 간사한가? 상황마다 매우 다르게 반응하고 보여주는 효율도 그때그때 다르다. 특히 개발이란 업무는 개인의 지식기능을 최대한으로 발휘해야 하는 작업 중 하나다. 따라서 개발을 잘하기 위해서는 집중할 수 있는 일관된 환경이 필요하다. 하지만 항상 그 조건이 충족되는 환경 아래에서 개발이 이뤄지는 건 아니다. 아니, 오히려 그 반대의 상황이 더 많다. 게다가 우리 삶에서 처하게 되는 각종 상황은 소프트웨어 개발환경을 매우 다양하게 만들어준다.

자, 어떤 상황을 하나 가정해보자. 당신은 개발 업무를 위해 노트북 모니터를 대면하고 앉아 있다. 하지만 좀처럼 설계문서도 잘 읽히지가 않고, 개발하다 중단했던 코드도 전혀 눈에 들어오지가 않는다. 전날 밤 동창들과의 술자리에서, 출판물엔 쓰지 않는 수식어를 팀장 이름 앞뒤에 접두사와 접미사로 붙여가며 미친 듯이 술을 먹어댄 덕분이다. 이런! 키보드가 순간순간 마른안주 세트 접시로 보이곤 한다. 그래도 어찌어찌 출근은 했으니 다행이라고 생각한다.

아니면, 이런 상황이라고 가정해도 좋겠다. 며칠 전, 느닷없이 헤어지자는 여자친구의 전화너머 한마디가 소스코드를 한 줄 칠 때마다 머릿속에서 for 문을 돌며 예외(Exception)를 연달아 내고 있다고 말이다.

어쨌든, 머릿속이 평소보다 조금 복잡한 상태다. 하지만 그렇다고 개발을 진행하지 않을 수는 없다. 책임감이 강한 당신은 있는 힘을 다해 개발을 마친다. 이런 극한 상황에서도 임무를 완수해내는 자신이 스스로 대견하다는 생각이 든다. 이후 간단한 테스트 몇 가지를 수행한 다음, 눈으로 보니 프로그램이 잘 동작하는 것처럼 보인다. 개발이 끝났다고 판단한다.

이런 식의, (어떤 의미론 명쾌한) '작성자의 판단'에 근거한 개발은, 중간 중간의 상태를 콘솔로 출력하게 만들어놓은 부분을 제거한다든가, 아니면 로깅(logging) 라이브러리를 사용한 디버깅용 출력을 비활성(disable) 상태로 만들어놓는다든가 하는 식으로 마감된다. 뭐, 평소보다는 집중력이 조금 부족했기에 동작하는 코드가 썩 마음에 들지는 않지만, 조금 찝찝한 그 느낌 때문에 다시 소스를 보는 경우는 드물다. '문제가 있으면, 나중에 테스트할 때 나오겠지~'라는 식으로, 소스에도 마음에도 덮개를 씌워놓고

for non-commercial purpose only

는 세제 CF의 주부 같은 자세로, "개발 끝~!!"이라고 기지개를 펴며 퇴근을 한다. 전체적으로 나쁘진 않은 시나리오[1]다.

그런데 이후 어느 시점엔가에서, 새로운 요구사항으로 인해 기능이 추가된다든가, 특정 오류가 발생해서 수정을 해야 한다든가 하는 일이 발생하면 어떻게 될까? 아마 로그를 다시 찍게 만들고, 세세한 부분을 따라가며 순간순간 변하는 프로그램의 상태를 살펴봐야 하는 일련의 작업들이 또 한 번 필요해질 것이다. 코드를 작성한 지 좀 되어서 이전보다는 좀 더 집중력이 필요하겠지만, 다행히 어제는 일찍 잠이 들었던 데다, 떠난다던 여자친구와도 극적으로 화해했기 때문에 때문에 어렵지 않게 해결할 수 있었다.

그런데 세상일이 무릇 그러하듯, 한 번에 끝나고 영원히 변치 않는 게 어디 있을까? 시간이 지나고, 코드(codebase)[2]의 크기가 커지면 커질수록, 버그 수정에 필요한 부분을 찾아내기가 어려워진다. 그러다 어느 순간에는 디버깅용 코드와 비즈니스 로직이 물반, 고기 반을 넘어서서, 흡사 남녀 비율 3:7 미팅에 남자 측 선수로 나간 것처럼 몸 버리고 돈 버리며 내내 희롱당하다 집에 오는 모습이 돼버린다. 즉, 해결되지 않는 문제와 쉽사리 추가되지 않는 기능, 그리고 그만큼이나 늘어나는 버그(bug)와 야근의 콤보(combination)로 인해, 결국은 업계에 대한 회의와 짜증을 실업수당과 저울질하는 수준이 된다.

약간 무리하게 예를 들긴 했지만, 전통적인 개발 및 테스트에서는 이런 문제점들을 흔히 볼 수 있다. 이런 식의, 고전적인 개발 방식에서 쉽게 나타나며 전혀 신선하지 않은 몇 가지 문제점을 꼽아본다면 다음과 같다.

1. 특정 모듈의 개발 기간이 길어질수록 개발자의 목표의식이 흐려진다.

- 어디까지 짰더라?
- 아, 내가 지금 뭘 하는 거였지?
- 이 모듈이 무슨 기능을 해야 한댔더라?

---

1 혹자는 이런 형태의, 개발자의 주관적인 판단에 의존한 개발을 '우연에 의존한 코딩(programming by coincidence 혹은 accidentally working code)'이라고 부르기도 한다. 참고: http://www.pragprog.com/the-pragmatic-programmer/extracts/coincidence

2 codebase: 제품 개발에 사용되는 기반 소스코드들을 지칭한다. 저장소의 의미도 일부 포함되어 있다.

for non-commercial purpose only

2. 작업 분량이 늘어날수록 확인이 어려워진다.

- 로그가 어디 있더라?
- 이것도 화면으로 출력해보고…

3. 개발자의 집중력이 필요해진다.

- 앗! 화면 지나갔다!

4. 논리적인 오류를 찾기가 어렵다.

- 여기서 그러니까 이 값이 들어가면 나와야 하는 게… 아… 이게 맞던가?

5. 코드의 사용 방법과 변경 이력을 개발자의 기억력에 의존하게 되는 경우가 많다.

- 맞아! 개인고객 인증을 고치면 법인 고객인증 부분도 함께 고쳤어야 했었지!!

6. 테스트 케이스가 적혀 있는 엑셀 파일을 보며 매번 테스트를 실행하는 게 점점 귀찮아져서는 점차 간소화하는 항목들이 늘어난다.

- 날짜? 1111. 주민번호? 우선 222222-2222222. 주소? 서울 개똥이네

7. 코드 수정 시에 기존 코드의 정상 동작에 대한 보장이 어렵다.

- 휴~ 찾았다. 여길 고쳐야 하는 거였군! 아, 근데 이 금칙어 필터 모듈 혹시 다른 데서도 쓰는 거 아냐?

8. 테스트를 해보려면 소스코드에 변경을 가하는 등, 번거로운 선행 작업이 필요할 수 있다.

- 입고 처리를 테스트하려면, 주문이 완료됐다고 테이블에 직접 업데이트를 해줘야…

9. 그래서 소스 변경 시 해야 하는 회귀 테스트[3]는 곧잘 희귀 테스트(rare test)가 되기 쉽다.

- 아, 그걸 언제 다 다시 테스트해? 우선 급한 불부터 끄고 보자구. 집에 안 갈거야?

10. 이래저래 테스트는 개발자의 귀중한 노동력(man-month)을 적지 않게 소모한다.

- 품질(QA) 담당자 가라사대: 소스 수정사항 생기면 엑셀에 적힌 단위 테스트 다시 수행하는 걸 절대! 빼먹지 마!(그리고 자꾸 빼먹으면 재미없어질 거라는 눈빛!)

이런 문제는 어제 오늘 일이 아니고, 이미 수십 년 전부터 소프트웨어 개발 영역에서 끊임없이 문제로 제기돼왔다. 해당 문제를 제거하기 위해, 때로는 사회 공학적이고, 때

3 회귀 테스트(regression test): 이미 개발과 테스트가 완료된 모듈에 수정을 가하게 될 경우, 기존에 동작하던 다른 부분도 정상적으로 동작하는지 확인하기 위해 수행하는 테스트. 원칙적으로 기존 모듈에 수정이 가해질 때마다, 해당 모듈뿐 아니라 그 모듈과 연관되어 있는 다른 모든 모듈도 변함없이 목표대로 동작하는지를 매번 테스트해야 한다.

for non-commercial purpose only

로는 시스템적인 다양한 방법을 찾아왔다. 하지만 여전히 은탄환은 없다.[4] 그래도 확률을 낮춰줄 수 있는 여러 가지 기법은 그 가치를 인정받았는데, 그중 하나가 테스트를 개발의 전면으로 내세운 TDD이다. XP에서 유래된 이 기법은 2000년대에 이르면서 그 참신성과 효과로 인해 본격적으로 두각을 드러내기 시작한다.

1.2 테스트 주도 개발(TDD)

> 프로그램을 작성하기 전에 테스트 먼저 하라!
> Test the program before you write it.[5]
>
> – 켄트 벡(Kent Beck)[6]

TDD란 뭘까?

XP(eXtreme Programming) 창시자 중 한 명이며, TDD를 주도한 켄트 벡은 TDD를 소개한 자신의 책에서 "프로그램을 작성하기 전에 테스트를 먼저 작성하는 것"이라고 테스트 주도 개발을 정의했다. 프로그램을 작성하지도 않았는데, 테스트를 먼저 하라는 건 무슨 뜻일까? 마치, 자동차를 만들 때 주행 테스트를 먼저 하란 말만큼이나 이상하게 들리기도 한다.

---

4 전설에 따르면 늑대인간을 한방에 물리치는 방법으로 은탄환을 사용한다고 한다. 비유적으로 '은탄환'이란 어떤 문제점을 해결하는 특효약 혹은 단일 해결책을 뜻한다. 프레더릭 브룩스라는 소프트웨어 엔지니어가 1986년에 발표한 논문에서 소프트웨어 설계의 복잡성과 어려움에 대해 이야기하며 처음으로 "No Silver Bullet"이라는 표현을 사용했다. 참고: 『The Mythical Man-Month』(Brooks, Frederick P. Addison-Wesley, 1975), 번역서 『맨먼스 미신』(케이앤피북스)

5 『TDD by Example』이라는 책 서문의 시작에 이렇게 적혀 있다. TDD를 설명하는 짧고도 명료한 문장인데, (살짝 분하지만) 아직까지 이보다 더 단순명료한 문장은 찾지 못했다.

6 켄트 벡(Kent Beck): Smalltalk 전문가로 TDD와 XP의 창시자다. 그는 디자인 패턴으로 유명한 IBM 엔지니어 에릭 감마(Erich Gamma)와 함께 JUnit을 만들었으며, '객체 지향 프로그래밍을 위한 패턴 언어의 사용(Using Pattern Languages for Object-Oriented Programs, OOPSLA-87)'이라는 논문을 통해 소프트웨어 개발에 있어 패턴의 개념을 최초로 끌어낸 프로그래머다.

for non-commercial purpose only

TDD의 정의는 곧잘 아래와 같이 이야기되곤 한다.

"업무 코드를 작성하기 전에 테스트 코드를 먼저 만드는 것!"

코드를 검증하는 테스트 코드를 먼저 만든 다음에 실제 작성해야 하는 프로그램 코드 작성에 들어가라는 뜻이다. 최초에는 테스트 우선 개발(Test First Development)이라고 불렸으나 지금은 테스트 주도 개발(Test-Driven Development, TDD)이라 불린다. 테스트 코드부터 작성한다는 TDD의 정의가 다소 과격하게 들릴 수도 있지만, 메소드나 함수 같은 프로그램 모듈을 작성할 때 '작성 종료조건을 먼저 정해놓고 코딩을 시작한다'는 의미로 받아들이면 편하다. 이를테면, 두 숫자의 합을 구해서 반환하는 sum이라는 메소드를 작성한다고 가정해보자.

작성 메소드 이름	sum
기능 구현에 필요한 재료(argument)	int a, int b
반환 값의 타입	int
정상 동작 만족 조건(작성 종료조건)	a와 b를 더한 값을 결과로 돌려줌. 즉, sum(10, 15)는 25가 돼야 함

위 표는 만들고자 하는 메소드를 일종의 설계문서(Spec)처럼 간단히 적어본 모습이다. 사실 이런 형식은 굳이 TDD라는 용어를 꺼내지 않더라도, 우리가 프로그램을 작성할 때 머릿속으로 생각하는 내용과 별반 다르지 않다. 다만 **'문서로 만들어 머리로 생각하고 눈으로 확인할 것인가?'** 아니면, **'예상 결과를 코드로 표현해놓고 해당 코드가 자동으로 판단하게 할 것인가?'**의 차이가 있다. 이를테면 위 설계문서에 따라 sum 메소드를 작성할 때, 코드를 통해 정상적으로 구현됐는지를 판단하는 방법을 선택한다면 아래와 같은 코드로 작업할 수도 있다.

```
public class Calculator {
    public int sum(int a, int b) {
        return 0;
    }

    public static void main(String[] args) {
```


for non-commercial purpose only

```
        Calculator calc = new Calculator();
        System.out.println( calc.sum(10, 20) == 30);
        System.out.println( calc.sum(1, 2) == 3);
        System.out.println( calc.sum(-10, 20) == 10);
        System.out.println( calc.sum(0, 0) == 0);
    }
}

-----실행 결과-----

false
false
false
true
```

위 예제에서는 main 메소드를 테스트 메소드처럼 사용했다. sum 메소드는 컴파일 에러만 나지 않도록 해놓고, 내부는 비어 있는 상태다. sum 메소드를 먼저 구현한 다음에 테스트를 할 수도 있지만, 그렇게 하지 않고 검증코드를 먼저 만들어놓았다. 그 검증코드에 해당하는 테스트 케이스를 모두 만족하면, 즉 main 메소드의 실행 결과가 모두 true로 나오면 sum 메소드가 정상적으로 작성됐다고 판단하기로 한 것이다. 다시 말해, 명시적인 코드로 개발 종료조건을 정해놓은 것이다. 이런 식의 개발 접근 방식이 바로, TDD이다. 무언가 특별한 테크닉이나 테스트 프레임워크를 써야 TDD일 것 같지만 그렇지 않다. 테스트 케이스 작성으로 구현을 시작하는 것, 그게 바로 TDD이다. 뭐, TDD가 간단한 듯 말은 쉽게 했지만 이런저런 상황별로 자세히 살펴보기 시작하면 테스트 코드부터 작성한다는 규칙을 정확히 지키기가 쉽진 않다는 걸 알게 된다. 사실 제품 코드부터 작성하는 것이 더 편하게 느껴질 때도 많이 있다. 그 때문에, 혹자는 "100% TDD식 개발이란 없다!"라고 말하기도 한다. 자, 문제점과 해결책은 차차 살펴보기로 하고 우선은 TDD의 목표를 살펴보자.

for non-commercial purpose only

1.3 테스트 주도 개발의 목표

> 잘 동작하는 깔끔한 코드
> Clean code that works
>
> – 론 제프리(Ron Jeffries)[7]

우리가 TDD라는 방식을 통해 얻고자 하는 최종 목적은 **'잘 동작하는 깔끔한 코드'**이다. 이는 일반적인 소프트웨어 개발의 목표와 별반 다르진 않다. 다만 TDD에선 정상적으로 동작하는 코드만을 개발의 목표로 삼지 않고, 작성된 코드도 명확한 의미를 전달할 수 있게 작성돼야 한다고 말한다. 즉, '제대로 동작함(works)'뿐 아니라 '깔끔함(clean)'까지도 동등한 수준의 개발 목표로 삼는다는 점이 일반적인 개발 방식과 다르다. 이 차이점은 소프트웨어의 품질을 비롯한 유지보수의 편의성, 가독성, 그리고 그에 따른 소프트웨어의 비용과 안정성 등 여러 가지 측면의 의미를 내포한다. 따라서 이 책의 나머지 부분 전반에 걸쳐서 TDD를 사용해 어떻게 하면 '깔끔하고 잘 동작하는 코드'를 얻을 수 있는지를 이야기할 것이다.

1.4 테스트 주도 개발의 기원

> 애자일 개발[8] 방식 중 하나인 XP의 실천 방식 중 하나

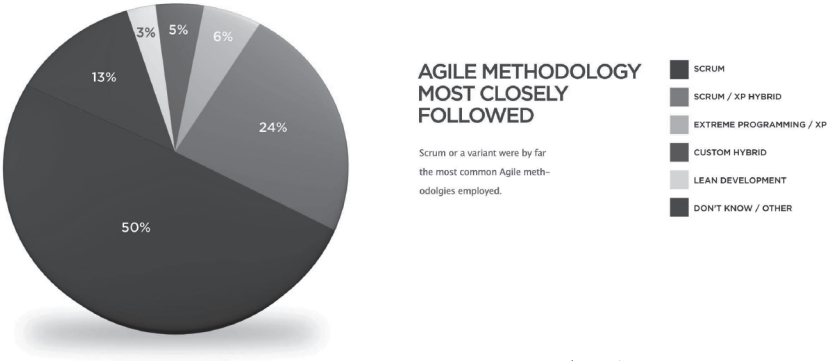
7 론 제프리는 켄트 벡, 워드 커닝엄(Ward Cunningham)과 함께 초대 XP 방법론을 이끈 이들 중 한 사람이다. 론 제프리는 켄트 벡의 친구이고, 워드 커닝엄은 켄트벡의 멘토다. 참고로 워드 커닝엄은 여럿이 함께 컨텐츠를 작성하는 위키(Wiki)의 개념을 처음으로 고안한 사람이기도 하다.

8 애자일(agile)이라는 단어는 '민첩한', '기민한' 등의 뜻을 갖는다. 애자일 개발은 말 그대로, 좀 더 민첩하고 유연하게 개발에 임하는 것을 말하며, 개발 그 자체에 집중할 수 있도록 개발환경을 조성한다. 전통적인 개발 방법론들이 소프트웨어 개발 업무 그 자체보다는 문서와 프로세스 같은 부가적인 부분들에 집중하는 모습에 대항하여 나온 방식이기도 하다. 시중에 다양한 관련서들이 나와 있으니, 만일 애자일에 관심이 있다면 참고하길 바란다. 개인적으로는 『익스트림 프로그래밍 제2판』과 『스크럼: 팀의 생산성을 극대화시키는 애자일 방법론』을 입문자를 위한 추천서로 권한다.

for non-commercial purpose only

TDD는 XP에서 등장하는 여러 가지 실천 방법 중 하나로 테스트 우선 개발과 동일한 의미를 갖는다. XP는 2000년대 초반에 급부상한 애자일 소프트웨어 개발론의 하나로 단순성, 상호소통, 피드백, 용기 등의 원칙에 기반해 '고객에게 최고의 가치를 최대한 빨리 전달하는 것'을 목표로 삼는다. TDD, 짝 프로그래밍(Pair Programming), 일일빌드(Daily Build), 지속적인 통합(Continuous Integration) 등 다양한 실천 방법을 제시하고 있다. 다만 XP는 extreme이라는 단어처럼 일부분 극단적인 실천 방법을 요구하기도 했기에 이 모든 내용을 적용하는 기업은 많지 않았다. 근래에는 XP를 단독으로 사용하기보다는 XP의 유용한 기법들을 다른 애자일 방법론[9]과 혼용해서 적용하는 형태가 많다. 또한 그중에선 XP와 스크럼(Scrum)을 혼용하는 경우가 가장 많다.

애자일 현황 및 애자일 기법 사용 순위

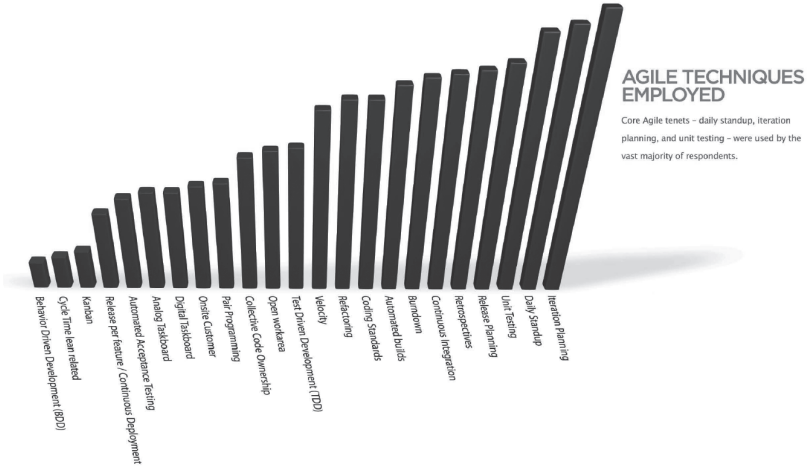


[어떤 애자일 방법론에 가까운 방식을 따르고 있습니까?]
스크럼(Scrum)이 50%, 스크럼과 XP 혼합이 24%를 차지하고 있다.

다음 페이지도 계속

9 사람에 따라 애자일 개발을 애자일 방법론이라고 칭하기도 하고, 애자일 실천 방식(agile practice)이라고 부르기도 한다. 애자일 개발에 대한 지칭에는 다소 논쟁의 소지가 있다. 가끔 '애자일'에 방법론이라는 단어를 붙여서 부르는 사람을 무시하거나 비난하는 경우가 있는데, 사실 그럴 필요는 없다. 설사 그리 불릴지라도 팀 내에 도입할 수만 있다면 무엇이 문제이겠는가? 만일 정 마음에 들지 않는다면 "저는 애자일이 방법론이라 불릴 때 사실 좀 아쉽다고 생각합니다. 왜냐하면…" 하는 식으로 대화를 취해보는 것도 한 가지 방법이다. 개인적으로는 애자일 개발 문화(agile development culture)라는 표현을 더 좋아한다.

for non-commercial purpose only



[어떤 애자일 기법을 적용하고 있습니까?]
관리 기법을 제외한 개발 테크닉으로만 순위를 뽑아보면 다음과 같다.
1위: 단위 테스트(Unit Testing)
2위: 지속적인 통합(Continuous Integration)
3위: 자동화된 빌드(Automated Builds)
4위: 테스트 주도 개발(TDD)
5위: 짝 프로그래밍(Pair Programming)

애자일 컨설팅 업체 VersionOne이 게재한 보고서에서 발췌.
2009년 7월부터 12월까지 88개국 총 2570명을 대상으로 조사한 결과다.

1.5 개발에 있어 테스트 주도 개발의 위치

개발자가 처음으로 수행하는 테스트

소프트웨어 개발에서 TDD는 '개발자가 자신을 위해 처음으로 수행하는 테스트'에 해당한다. 그래서 흔히 개발자 테스트(programmer test)라고 부르기도 한다. 때로는 그냥 단위 테스트(unit test)라고도 하는데, 이는 보통 전통적인 테스트 방법론에서 이야기하는 단위 테스트보다는 범위가 다소 협소하다. 따라서 구분해서 이야기할 필요가

for non-commercial purpose only

있는데, TDD에서 말하는 단위 테스트는 일반적으로 메소드 단위의 테스트를 뜻하고, 전통적인 테스트 방법론에서 이야기하는 단위 테스트는 사용자 측면에서 제품의 기능을 테스트[10]하는 쪽에 더 가깝다. TDD에서 개발자는 자신이 작성한 프로그램에 대해 메소드 혹은 함수 단위로 테스트를 수행하며, 이는 결과적으로 이후에 발생하는 테스트 단계(통합 테스트나 인수 테스트 등)에서의 결함발생 비용을 줄여준다. 결함을 빨리 발견하면 할수록 적은 비용으로 처리가 가능하기 때문에 규모가 큰 프로그램일수록 적용 효과가 크다.

앞으로 이 책에서, 단위 테스트 케이스(unit test case)라는 말은 TDD에서 말하는 메소드 단위의 '단위 테스트'를 의미하는 걸로 이해하면 되겠다.

그럼, 이제부터 TDD가 지향하는 개발 방식을 구체적으로 살펴보자.

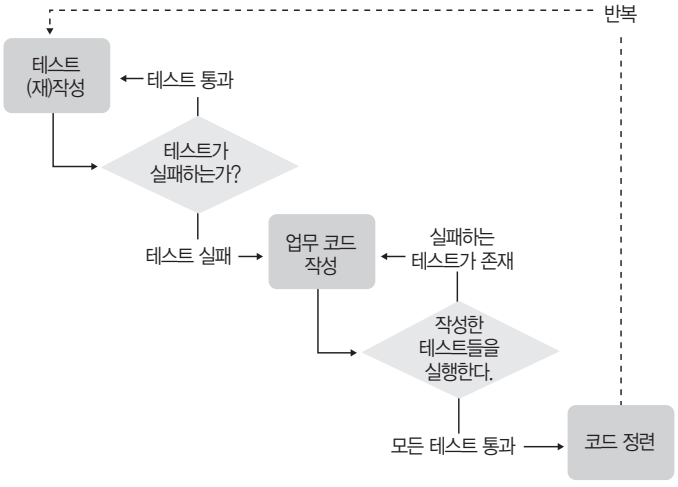
1.6 테스트 주도 개발의 진행 방식

- **질문(Ask)**: 테스트 작성을 통해 시스템에 질문한다. (테스트 수행 결과는 실패)
- **응답(Respond)**: 테스트를 통과하는 코드를 작성해서 질문에 대답한다. (테스트 성공)
- **정제(Refine)**: 아이디어를 통합하고, 불필요한 것은 제거하고, 모호한 것은 명확히 해서 대답을 정제한다. (리팩토링)
- **반복(Repeat)**: 다음 질문을 통해 대화를 계속 진행한다.

10 기능 테스트(functional test): 개발자 입장보다는 사용자 입장에 좀 더 가까운 형태로 진행되는 기능들의 테스트. 일반적으로 여러 개의 단위 모듈이 합쳐져서 한 개의 기능을 이루는 경우가 많다. 예를 들어 주민번호 입력란에 사용자가 기입할 때, 기능 테스트 입장에선 숫자만 들어가도록 강제화되어 있는지를 테스트한다. 하지만 그 기능을 위해 여러 개의 단위 모듈이 동작할 수도 있기 때문에 TDD에서의 단위 테스트는 그 기능 테스트 하위의 단위 단위들의 모듈을 테스트한다고 보면 되겠다.

for non-commercial purpose only

TDD를 이용한 개발은 크게 **'질문 → 응답 → 정제'**라는 세 단계가 반복적으로 이루어진다. 하나하나에 대한 설명은 차차 예제와 함께 살펴보도록 하고, 우선은 차분히 한 번 읽어 보고 다음 그림을 보자.



TDD 순서도. 참조: http://en.wikipedia.org/wiki/Test-driven_development

동일한 내용을 다이어그램으로 도식화해보면 위와 같다. TDD는 위와 같은 형태의 반복적인 흐름을 갖는다. 기존에 TDD를 본 적이 없다면 '테스트 작성'이 어떤 모습일지 사실 상상하기 어려울 수 있다. 흔히 단위 테스트 프레임워크(Unit Test Framework)를 사용한 테스트 코드 작성이 이뤄진다고 생각하면 된다. 자세한 진행 방식은 예제와 함께 살펴볼 예정이다.

for non-commercial purpose only

1.7 실습 먼저 시작해보기

앞에서 간략하게 TDD의 작업 방식에 대해 이야기해봤다. 지금부터는 어떤 식으로 TDD가 진행되는지 한번 살펴보려고 한다. 우선 TDD의 대표 예제라 불릴 만한 은행 계좌(Account Class) 클래스를 TDD 방식으로 만들어본 다음, 뒷부분에선 좀 더 생각할 요소들이 들어 있는 자판기(Vending Machine) 잔돈 예제를 살펴볼 예정이다. 그럼 은행계좌 클래스부터 시작하기로 한다. 만일 TDD에 어느 정도 익숙한 독자라면 이번 절은 가볍게 훑고 넘어가도 무방하다. 참고로 이하 예제는, Eclipse IDE for Java EE Developers(eclipse 3.5, Galileo 이상)를 기준으로 작성됐다.

실습 목표

- TDD의 기본적인 진행 방식을 익힌다.
- 시간을 절약시켜줄 이클립스 IDE의 관련 기능을 익힌다.
- TDD를 진행할 때 발생할 수 있는 몇 가지 기본적인 고려사항을 살펴본다.

실습 예제 – 은행 계좌(Account) 클래스 만들기



□ 은행계좌 클래스
- 계좌 잔고 조회
- 입금/출금
- 예상 복리 이자(추가 개발)

업무를 파악해본 결과, 은행계좌(Account) 클래스를 만들어야 한다고 판단됐으며, 기능 요구사항은 위와 같다고 가정한다. 잔고 조회가 가능해야 하고, 입금과 출금을 관리할 수 있어야 한다. 단, 예상 복리 이자를 구하는 것은 추가 개발로 나중에 들어올 것 같다고 한다. 어쨌든, 현재 계좌 클래스에서 필요한 기능은 위와 같다. 위 내용을 기초로, 앞서 배운 **'질문 → 응답 → 정제'**의 단계를 밟아가며, 실습을 진행해보자. 중간 중간 다소 힘들 수도 있는데, TDD의 가장 기본적이면서도 반드시 알아야 하는 내용들이 나오므로 조금 집중해서 봐줬으면 좋겠다. 자, 그럼 시작해보자!

for non-commercial purpose only

첫 번째 질문: 계좌 생성 테스트

- 구현해야 할 기능을 파악하고, 목록을 작성한다.
- 계좌 생성 기능을 구현하기 위한 최초의 테스트 케이스를 만들고 실패하는 모습을 확인한다.

질문
응답
정제

일반적인 소프트웨어의 개발이 기능을 구현하고 테스트를 수행하는 형태라고 한다면, TDD에서는 그와 반대로 진행된다. 본 실습에서도 테스트를 먼저 작성한 다음 실행해 볼 생각이다. 그럼 테스트 작성의 기본 단위부터 살펴보자. TDD에는 테스트의 최소 작성 단위를 최하위 모듈의 단위와 일치시킨다. Java 언어 기준으로 최하위 모듈은 '메소드'다. 다른 언어에서는 종종 함수(Function)가 최하위 모듈이 된다. 가장 처음에 해야 하는 단계인 질문(ask) 단계에서는 바로 이 메소드 수준의 단위 테스트를 작성하게 된다. 앞에서 '질문 단계에서는 시스템에 질문을 던진다'고 표현했지만 실제로 해야 하는 일은 **'작성하고자 하는 메소드나 기능이 무엇인지 선별하고 작성 완료 조건을 정해서 실패하는 테스트 케이스를 작성하는 것'**이다. 클래스 설계서 같은 산출물이 있는 경우라면, 크게 고민할 것 없이 메소드 외양부터 만들기 시작하면 된다. 이때 리턴 타입은 기본 초기값(null, 0 등) 위주로 설정해놓으면 편하다. 혹자는 이런 방식을 클래스 스켈레톤(skeleton) 구현이라고 부른다.

만약 개발을 위해 업무전문가나 설계자로부터 넘겨받은 산출물이 없다면 개발에 필요한 모든 내용을 개발자가 스스로 머릿속에서 떠올려야 한다. 이 경우, 작성해야 할 업무 클래스의 외형을 한 번에 만들기는 힘들다. 이럴 때는 우선 대략적인 설계를 먼저 진행한 다음에 질문 단계를 시작한다. 이번 예제에서는 설계문서 없이 곧바로 개발한다고 가정했다. 가능하다면 주의 환기 차원에서 음료수를 옆자리에 놓은 다음 실습을 시작하길 권한다. 별건 아니지만, 건강을 해칠 만큼 오랫동안 앉아 있게 되는 상황을 방지하는 데 도움이 될 수 있다(물론 이때, 음료수로 주류를 선택하는 사람은 없길 바란다).

for non-commercial purpose only

지금은 뚜렷한 설계서가 없다는 가정하에서 진행하고 있으니까, 구현해야 하는 기능과 유의사항을 생각나는 대로 적어보자.

■ 클래스 이름은 Account
■ 기능은 세 가지
  • 잔고 조회
  • 입금
  • 출금
* 금액은 원 단위로(예: 천 원 = 1000)

메모장에 기능 요구사항과 유의사항을 ToDo 목록처럼 적어보았다.

질문 단계에서는 메모장이나 마인드맵(mind map) 등을 활용하면 도움이 된다. 개인적으로는 사각거리는 소리가 나는 종이와 펜을 더 선호한다. 어느 정도 준비가 됐으면 테스트 케이스를 작성해보자. 보통 이때 두 가지 접근 방식을 이용할 수 있다.

Case 1 구현 대상 클래스의 외형에 해당하는 메소드들을 먼저 만들고 테스트 케이스를 일괄적으로 만드는 방식

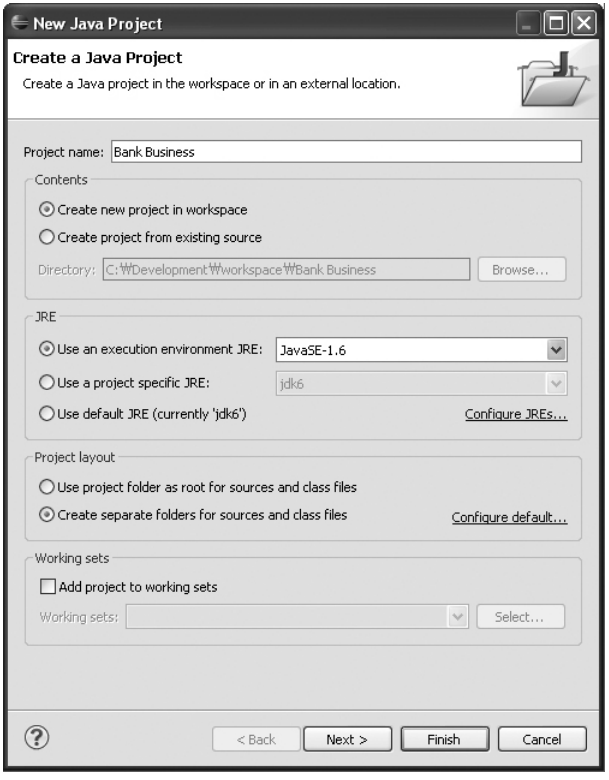
Case 2 테스트 케이스를 하나씩 추가해나가면서 구현 클래스를 점진적으로 만드는 방식

비교적 문제상황이 단순한 본 예제에서는 둘 중 어느 것이 특별히 좋다고 하긴 어려우니, 각자 해보고 편한 방식을 선택하면 된다. 참고로 대부분의 TDD 책에서는 2번을 권장한다. 첫 번째 방식을 사용했을 경우에는, 모든 테스트가 정상 통과하는 올 그린(All Green)[11]상태에 이르기까지 긴 시간이 걸릴 수 있다. 그로 인해 TDD의 몇 가지 장점을 잃을 우려가 있기 때문에 초보자에게는 권하지 않다. 이왕이면 좋은 습관을 기본으로 익히도록 우리는 두 번째 방식으로 진행할 생각이다.

우선 적당한 이름으로 Java 프로젝트를 생성한다. 그런 다음 테스트 클래스를 작성하자. 본 예제에서는 Bank Business라고 이름을 정했다.

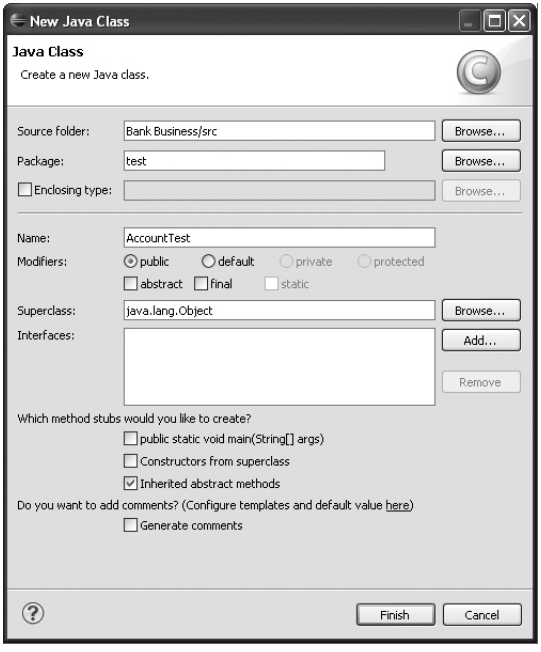
11 TDD에서는 모든 테스트 케이스가 통과하면 녹색 막대가 표시된다. 테스트 케이스를 기준으로 소스코드가 모든 테스트를 정상통과해, 현재 시스템이 건강하다고 판단하는 상태를 흔히 올 그린 상태라고 표현한다.

for non-commercial purpose only



이클립스 메뉴에서 [File] → [New] → [Class]**를 선택해서 클래스를 새로 만든다.** 물론 이때 [File] → [New] → [JUnit Test Case]를 곧바로 선택해도 되지만, 나중에 좀 더 익숙해진 다음에 사용하기로 하고 우선은 일반 클래스로 실습해보자.

for non-commercial purpose only



테스트 클래스로 사용할 AccountTest 클래스 생성

계좌 클래스(Account)에 대한 테스트 케이스 작성을 위해 만든 테스트 클래스

```
package test;

public class AccountTest {

}
```

테스트 클래스를 만들었으니 이제 테스트 케이스를 만들어보자. 테스트 케이스는 테스트하고자 하는 대상에 대해 간단한 시나리오를 만들고 그것을 코드로 표현한 모습이다. 우선 계좌 생성에 대한 테스트 시나리오는 다음과 같다.

계좌를 생성한다. → 계좌가 정상적으로 생성됐는지 확인한다.

for non-commercial purpose only

어쩌면 잔고 조회까지 한꺼번에 시나리오로 넣고 싶을 수도 있지만, 우선은 참아보자. TDD에서는 하나의 테스트 케이스가 하나의 기능을 테스트하도록 만드는 것이 기본 원칙이다. 그리고 대부분의 경우 하나의 테스트 케이스는 하나의 메소드로 표현된다. 이 메소드를 테스트 메소드라고 부른다. 그럼 이 테스트 시나리오를 기준으로 계좌 생성 테스트 메소드를 만들어보자.

*AccountTest.java

```
package test;

public class AccountTest {

    public void testAccount(){
        Account account = new Account();
    }

}
```

계좌 생성을 테스트하기 위해 만든 메소드이므로 이름을 testAccount()라 지었다. 그리고 시나리오의 첫 부분인 '계좌를 생성한다' 부분을 코드로 기술했다. 타이핑을 마치는 순간 곧바로 문제가 발생했음을 알려주는 붉은색 밑줄과 x 표시가 붙은 노란색 전구 아이콘이 코드 옆에 나타날 것이다. 작성한 소스코드를 저장(Ctrl+S)한 다음 이클립스 하단의 Problems 항목을 살펴보자.

Problems | Javadoc | Declaration | Console

2 errors, 0 warnings, 0 others

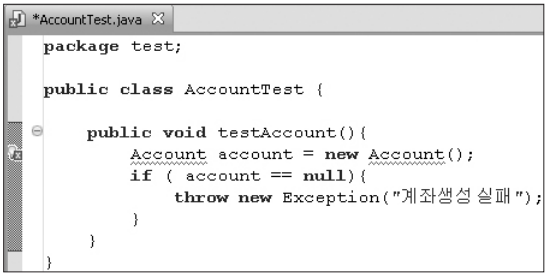
Description	Resource	Path	Locat...	Type
Errors (2 items)				
Account cannot be resolved to a type	AccountTest.java	/BankBusiness/src/test	line 6	Java Problem
Account cannot be resolved to a type	AccountTest.java	/BankBusiness/src/test	line 6	Java Problem

Account가 무슨 타입인지 못 알아듣겠다며 이클립스가 에러 항목으로 표시하고 있다.

Account 클래스를 만든 적이 없으니 에러가 나는 것이 당연하다. 당장 이 문제를 해결할 수도 있지만, 우선은 무시하고 넘어가자. 테스트 시나리오를 코드로 기술하는 작업을 먼저 마치고, 그 다음에 해결하기로 하자. **이런 식으로 진행하는 이유는 테스트 케이스 작성 시 흐름을 잃지 않기 위해서다.**

for non-commercial purpose only

현재 작성 중인 테스트 케이스 코드에는, 계좌가 정상적으로 생성됐는지 확인하는 부분이 빠져 있다. 추가해보자.


*AccountTest.java

```
package test;

public class AccountTest {

    public void testAccount(){
        Account account = new Account();
        if ( account == null){
            throw new Exception("계좌생성 실패");
        }
    }
}
```

만일 생성된 계좌가 null이라면 예외를 발생시키도록 만들었다. 여전히 Account 클래스 타입 선언부는 문제가 있다고 표시된다. 하지만 테스트 시나리오를 코드로 표현하는 일은 완료됐다. **만일 testAccount() 메소드를 실행했을 때 어떤 문제나 메시지도 발생하지 않는다면, 계좌 생성에 대한 테스트가 성공한 것으로 간주할 생각이다.** 이제, 테스트 케이스에 해당하는 testAccount() 메소드를 실행시켜 봤으면 좋겠다. testAccount() 메소드를 실행시키는 데는 여러 방법이 있지만, main 메소드를 이용하는 것이 제일 간단하다. 테스트 케이스를 실행하는 main 메소드를 작성해보자.

```
package test;

public class AccountTest {

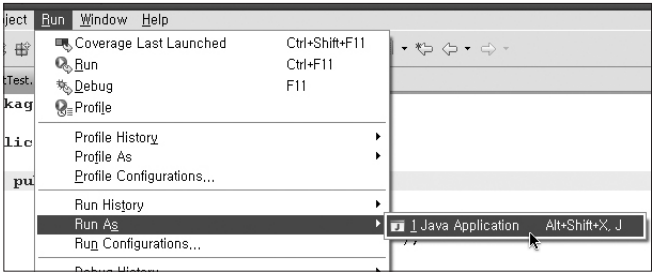
    public void testAccount(){
        Account account = new Account();
        if ( account == null){
            throw new Exception("계좌생성 실패");
        }
    }

    public static void main(String[] args) {
        AccountTest test = new AccountTest();
        test.testAccount();    // 테스트 케이스 실행
```

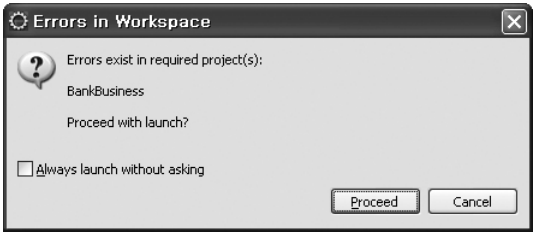

for non-commercial purpose only

```
        }
    }
```

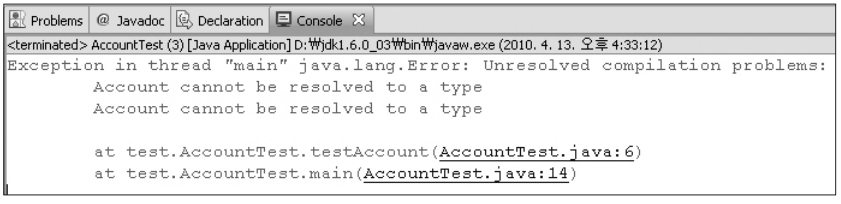
테스트를 수행해볼 수 있는 main 메소드도 완성됐다. 에러가 다소 존재하지만, 모른 척 무시하고 테스트를 실행해보자. 이클립스 메뉴에서 [Run] → [Run As] → [Java Application]을 선택한다.



이클립스의 애플리케이션 실행 메뉴



에러가 있다는 경고문이 나온다. 한글로 된 메시지가 아니니까 그냥 무시하고 넘어가자…는 아니고, 우선은 테스트를 억지로나마 실행시켜 보자(돌이키기엔 이미 늦었다). 계속진행(Proceed)을 선택한다.



실행 결과를 보여주는 콘솔 화면

for non-commercial purpose only

붉은색으로 에러 메시지가 잔뜩 나오며, 테스트가 예상대로 동작하지 않는다. 음… 자존심이 살짝 상하지만, 별 수 없다. 다시 소심한 개발자로 돌아가자. 하긴, 컴파일 에러도 처리 안 하고 실행했는데, 아무 문제 없이 실행됐다면, 오히려 더 당황했을 것도 같다.

그런데 콘솔 창에 찍힌 에러 메시지를 보다 보니 문득, '그런데 지금 이 괴상한 코드로 에러까지 무시해가며 뭘 하겠다는 거냐?'라는 의문이 들 수 있다. 지금 만들어진 코드는 Account 클래스의 생성자 메소드 Account()에 대한 테스트 코드에 해당한다. 즉, 만들고자 하는 메소드의 예상 동작을 시나리오에 기반하여 코드로 먼저 만들어놓은 모습이다. 그리고 그 예상대로 올바르게 동작하는지 확인해보기 위해 실행해봤다. 이렇게 작성된 테스트 코드가 바로, 시스템에 대해 개발자가 하는 질문이다.

'코드가 예상대로 동작하는지 판단해줄래?'

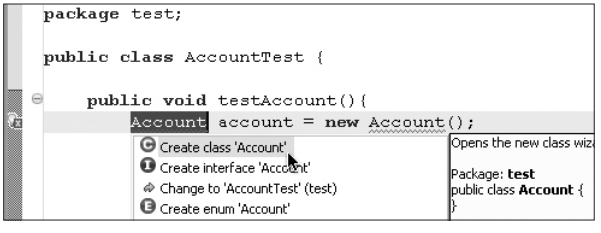
시스템은 에러 메시지를 보여주며 '아니! 실패했는데?'라고 대답했다. 이제, 시스템의 메시지에 응답할 시간이다.

첫 번째 응답: 계좌 생성 메소드 구현

- 계좌 생성 테스트 케이스를 통과하는 코드를 작성한다.

질문
응답
정제

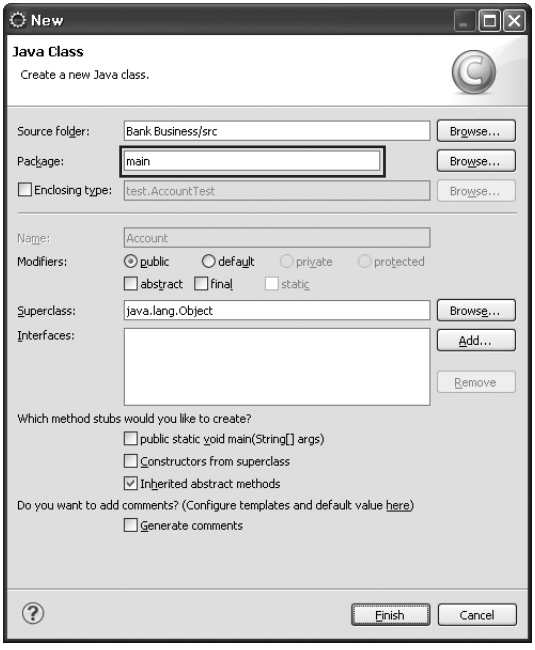
방금 전 실행했던 테스트 케이스는 에러와 함께 실패했다. 즉, 질문에 대한 시스템의 응답(Response)은 실패(Fail)다. 이번엔 테스트를 성공시켜 보자.



Quick Fix 실행 화면

for non-commercial purpose only

에디터 창 좌측에 있는 전구 아이콘을 누르면 선택지가 나온다. 에러가 나는 줄에서 Quick Fix 기능의 단축키인 Ctrl+1을 눌러도 마찬가지 메뉴가 뜬다. 필자 개인적으로는 Quick Fix 단축키 쪽을 더 선호하고 권장한다. 선택 메뉴 중에서 Create class 'Account'를 선택해 Account 클래스를 만들어보자.



Create class 'Account'를 선택하면 나오는 클래스 생성창

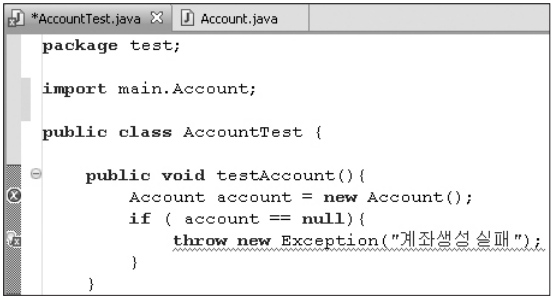
어떻게 만들든지 크게 상관은 없지만, 이번엔 패키지(package)를 main으로 만들었다. 앞으로 제품코드(Production Code)는 main 이하에, 테스트 코드는 test 패키지 이하에 만들 예정이다. 지금 단계에서는 그다지 중요한 내용은 아니지만, 어쨌든 테스트 코드와 테스트의 대상이 되는 제품 코드의 위치를 분리시켰다. 테스트 코드와 제품 코드의 위치를 어떻게 선정할 것인지는 뒤에서 다시 자세히 다룰 예정이다. 이 외에는 특별히 수정할 내용이 없으므로 완료(Finish) 버튼을 힘껏 누른다. 빈 Account 클래스가 생성된 걸 볼 수 있다.

for non-commercial purpose only

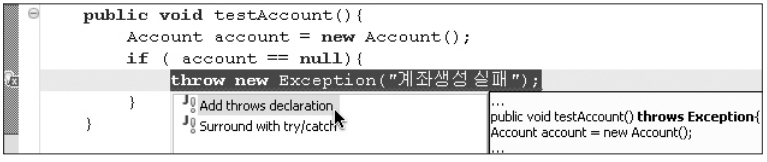
```
package main;
public class Account {

}
```

아직 별 필요는 없으니까 방금 전 작성한 테스트 코드로 돌아가 보자(Alt+←).



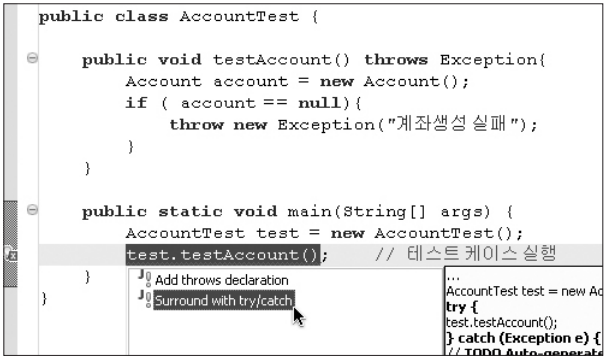
여전히 에러가 나거나 위 화면처럼 보인다면 저장(Ctrl+S)을 눌러서 새롭게 컴파일 될 수 있게 만들자. 코드 중간쯤 계좌 생성 실패 시 예외를 발생시키도록 만든 부분에 전구 표시가 뜬 걸 볼 수 있다. 마찬가지로 Quick Fix를 이용해서 예외(Exception)를 호출하는 쪽으로 던져버리도록 만들자.



testAccount() 메소드에서 발생하는 예외를 호출 쪽으로 넘기도록 만든다.

저장을 누르면 이번엔 main 메소드에서 오류 표시가 발생할 것이다. testAccount() 메소드에서 예외를 무작정 던져버렸기 때문이다. 이번에도 던져버릴 수 있는데, 그렇게 하지 않고 try ~ catch 구문으로 감싸보자. 잘 이용하면 좀 더 우아하게 테스트 성공 실패를 보여주도록 만들 수 있을 것 같다.

for non-commercial purpose only



Quick Fix를 이용해서 try/catch 문장으로 감싸기(Surround with try/catch)를 선택한다.

```
package test;

import main.Account;

public class AccountTest {

    public void testAccount() throws Exception{
        Account account = new Account();
        if ( account == null){
            throw new Exception("계좌생성 실패");
        }
    }

    public static void main(String[] args) {
        AccountTest test = new AccountTest();
        try {
            test.testAccount();  // 테스트 케이스 실행
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


for non-commercial purpose only

현재까지의 코드는 위와 같다. 자동완성 기능을 이용했기 때문에, 예외가 발생하면 예외 메시지를 출력하도록 catch 부분이 작성되어 있다. 가독성을 높이는 차원에서, 예외가 발생하면 '실패', 예외가 발생하지 않으면 '성공' 메시지가 나오도록 변경해보자.

```
package test;

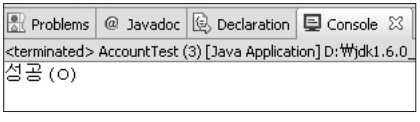
import main.Account;

public class AccountTest {

    public void testAccount() throws Exception{
        Account account = new Account();
        if ( account == null){
            throw new Exception("계좌생성 실패");
        }
    }

    public static void main(String[] args) {
        AccountTest test = new AccountTest();
        try {
            test.testAccount();  // 테스트 케이스 실행
        } catch (Exception e) {
            System.out.println("실패(X)");   // 예외가 발생하면 실패(X)
            return;
        }
        System.out.println("성공(O)");
    }
}
```

try 구문 안에서 예외가 발생하면 '실패(X)'라고 출력하고, main 메소드를 종료한다. 예외가 발생하지 않으면 '성공(O)'이라고 출력한다. main 메소드를 실행해보자.



테스트가 아무런 예외 없이 잘 실행됐다!

for non-commercial purpose only

이제 Account 클래스 생성 작업을 마쳤고, 질문(Ask)에 대답하는 응답(Respond) 단계를 마쳤다. 이제 한 주기의 마지막 단계인 정제(Refine) 단계로 들어가자.

저자 한마디

최대한 빨리 실패하기

TDD에서는 테스트 자동화를 통해서 개발이 시작된 시점부터 완료될 때까지 가능한 한 빠른 시점 내에 그리고 자주 실패를 경험하도록 유도하고 있다. 심지어는 개발이 시작되기도 전에 실패가 발생하는 상황부터 보고 시작하라고 하니 말 다했다. 그런데 이런 방식은 사실 일반적인 문제 해결 방식과는 조금 다르다. 보통은 작업의 중간 과정에서조차도 실패를 하지 않기 위해 최대한 노력하는 경우가 더 많다. 하지만 TDD는 실패를 통해 배움을 늘려가는 기법이다. OK 조건을 사전에 정해두고 빠르게 실패를 경험하며, 그 조건을 등대로 삼아 실패 상황을 최대한 빨리 극복하고자 노력한다. 성공한 항목과 실패한 항목이 명확하고, 작업해야 하는 부분이 확실하다. 성공에 필요한 조건을 만들고, 실패하는 조건 항목을 성공시킨다. 그래서 빨리 실패하면 실패할수록 좀 더 성공에 가까워지는 묘한 개발 방식이다.

최초의 정제

- 리팩토링을 적용할 부분이 있는지 찾아본다.
- ToDo 목록에서 완료된 부분을 지운다.

질문
응답
정제

한 주기가 끝나기 전에 소스를 정제해보자. 보통 이 단계에서 리팩토링(refactoring)작업을 수행한다. 리팩토링은 쉽게 말하면, 정상적으로 동작하는 코드를 수정해서, '사람'이 좀 더 '이해하기 쉽고', '변경하기 용이'한 구조로 소스코드를 개선하는 작업을 지칭한다. 정상적으로 동작하는 코드를 수정하는 일이 때때로 부담스러운 건 사실이지만, 자동화된 테스트 케이스가 이미 만들어져 있다면 두려움이 적어진다. 여차하면 되돌리기(Ctrl+Z) 기능을 이용해서 마지막으로 테스트가 성공한 시점으로 소스코드를 되돌리면 되기 때문이다. 그래서 자동화된 테스트 케이스를 만들어내는 TDD와 소스를 개선하는 리팩토링은 궁합이 잘 맞는다. TDD에서도 한 주기의 마지막 단계로 정제 단계

for non-commercial purpose only

를 만들어놓고 리팩토링을 권장하고 있다. 리팩토링을 수행하게 되는 정제 단계에서는 일반적으로 다음과 같은 질문에 대해 고민해보는 시간을 갖는다.

- 소스의 가독성이 적절한가?
- 중복된 코드는 없는가?
- 이름이 잘못 부여된 메소드나 변수명은 없는가?
- 구조의 개선이 필요한 부분은 없는가?

위와 같은 질문을 해보면서, 안전그물(=앞서 작성한 테스트 케이스)에 의존해 소스를 가다듬는 단계가 정제 단계다. 지금은 원체 소심하게 진행했고, 단계 자체가 얼마 나아가지 못했기 때문에 내용적으로는 별달리 정제할 것이 없다.

앞으로는 테스트 케이스 작성 후에 main 메소드의 try ~ catch 문 사이에 넣어놓기만 하면 자동적으로 예외 발생 여부가 검사될 것이다. 다만 현재 구조에서는 예외가 발생하면 테스트가 더 이상 진행되지 않는 불편함이 있다.

테스트 케이스가 추가됐을 때를 가정한 테스트 실행 코드의 모습

```
public static void main(String[] args) {
    AccountTest test = new AccountTest();
    try {
        test.testAccount();        // 계좌 생성 테스트
        test.testGetBalance();     // 잔고 조회 테스트
              …                    // 또 다른 테스트 케이스
    } catch (Exception e) {
        System.out.println("실패(X)");
        return;
    }
    System.out.println("성공(O)");
}
…
```

뭔가 조금 아쉽긴 하지만, 여기까지 해서 TDD의 한 주기를 끝내도록 한다. 자, 그럼 아까 적었던 메모에서 한 줄을 처리해보자.

for non-commercial purpose only

~~■ 클래스 이름은 Account~~
■ 기능은 세 가지
  • 잔고 조회
  • 입금
  • 출금
* 금액은 원 단위로(예: 천 원 = 1000)

이제 한 주기를 마쳤으니, 두 번째 질문으로 넘어갈 차례다.

하지만, 잠깐!!

사실 이 정도 코드면 자동화된 테스트 케이스가 비교적 잘 작성된 것으로 간주할 수 있다. 복잡하지 않은 경우라면 위에서 작성한 예제의 테스트 실행 방식을 지속적으로 이용할 수도 있을 것이다. 하지만 테스트 케이스가 늘어나면 지금의 구조로는 조금 부족하다. 좀 더 다양한 방식으로 테스트 케이스를 작성하려면 몇 가지 추가요소가 필요하다. 이를테면, 대량의 테스트 케이스 항목 관리, 실패 시 메시지 처리, 현재의 if 구문을 이용한 예외처리 구조 대신에 좀 더 단순화된 식으로 예상값과 결과값 비교 등이 지원된다면 좀 더 구조적인 테스트 케이스 작성이 가능해질 것이다. 사실, 충분히 나름의 가치가 있는 일이라 생각되어 TDD의 첫 두 단계를 일부러 조금 돌아왔다. 맨땅에서 시작하는 식으로 말이다. 이제, 현재 코드에 JUnit 단위 테스트 프레임워크를 적용해볼 생각이다. 단위 테스트 프레임워크를 사용하면 어떤 부분이 구조화되고, 어떤 장점을 갖게 되는지 함께 살펴보자.

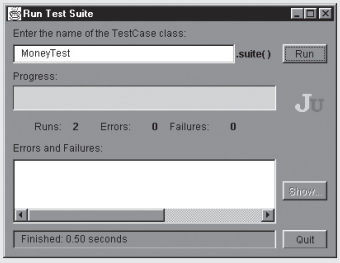
for non-commercial purpose only

저자 한마디

단위 테스트 프레임워크 엿보기

□ **JUnit[12] 테스트 프레임워크(http://www.junit.org)**

1. 단위 테스트 프레임워크 중 하나
2. 문자 혹은 GUI 기반으로 실행됨
3. xUnit이라 불리는 스타일을 따른다.
   assertEquals(예상 값, 실제 값)
4. 결과는 성공(녹색), 실패(붉은색) 중 하나로 표시

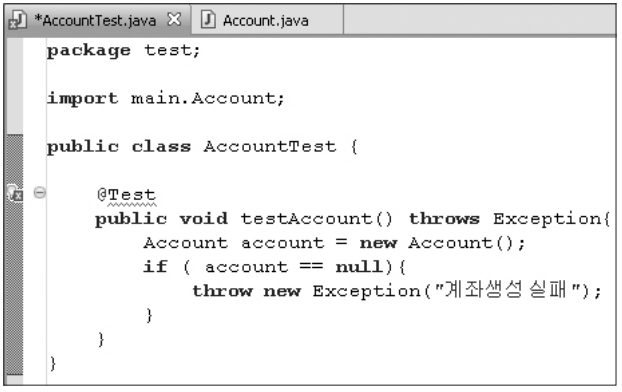


예제에 적용하기에 앞서 TDD의 부흥을 만들었고, 소프트웨어 개발 방식의 전환을 만드는 초석이 된 단위 테스트 프레임워크인 JUnit에 대해 살펴보고자 한다. 위 상자 안의 네 개 항목을 읽어보자. 여기서는 예제에서 사용할 수 있게 이 정도로만 간략히 살펴보고, 자세한 내용은 뒤에 다시 다루기로 한다.

코드 내에서 main 메소드를 지운 다음, @Test라고 testAccount() 메소드 위에 적어보자. @Test는 해당 메소드를 테스트 메소드로 지정하기 위한 애노테이션(annotation)이며, 이런 식의 애노테이션 지정은 JUnit 4 버전에서 사용하는 방식이다. 애노테이션 및 JUnit에 대해서는 이후 JUnit을 설명하는 부분에서 자세히 다룰 예정이다.

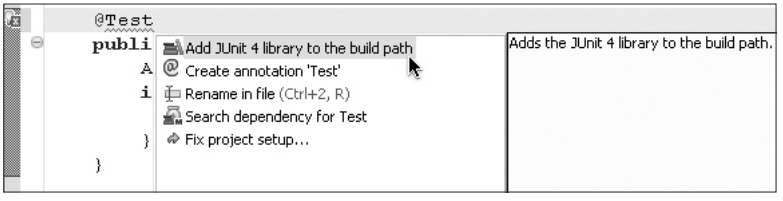
12 이번 실습에서 사용하게 될 JUnit 테스트 프레임워크(http://www.junit.org)는 앞서 말한 TDD 사이클을 진행하는 데 도움을 주는, 가장 대표적인 단위 테스트 프레임워크다. 현재 이클립스(Eclipse), 인텔리제이(IntelliJ IDEA), 넷빈즈(NetBeans) 등 대부분의 Java 통합개발환경(IDE)에 기본으로 내장되어 있다. 현재 Java 단위 테스트에 있어 사실상의 업계 표준(De Facto Standard)이 되어 있다.

for non-commercial purpose only



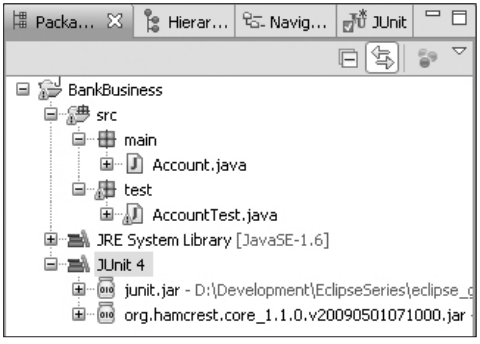
타이핑하고 보니 에러가 보인다. JUnit이라는 단위 테스트 라이브러리(unit test library)가 클래스 경로, 혹은 빌드 경로에 포함되어 있지 않아서 발생하는 에러다. Java 컴파일러가 참조할 수 있도록 클래스 경로에 JUnit 라이브러리를 추가해놓으면 된다. 직접 해도 되지만, 우선은 이클립스 IDE의 기능을 적극 이용해보자.

@Test 줄에 있는 전구 아이콘을 마우스로 클릭한다(마찬가지로 Quick Fix의 단축키 Ctrl+1을 눌러도 된다).



Add JUnit 4 libaray to the build path를 선택해서 빌드 경로 내에 JUnit 4 라이브러리가 포함될 수 있게 한다. 그 다음 패키지 탐색기(package explorer)에서 JUnit 라이브러리가 제대로 포함됐는지를 확인해보자.

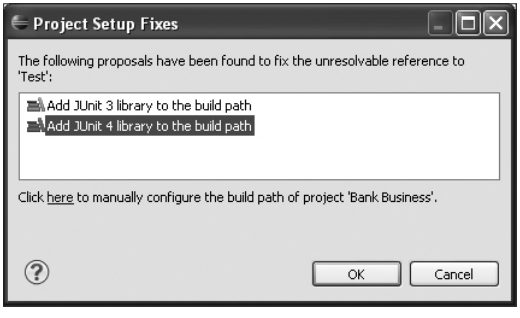
for non-commercial purpose only



JUnit 4 항목으로 JUnit 라이브러리가 빌드 경로에 포함됐다.

눈치 빠른 개발자라면 JUnit 프레임워크를 사용하기 위해서는 클래스 경로 내에 원하는 버전의 junit.jar 파일이 들어 있기만 하면 된다는 사실을 알아 챌 것이다. 물론 경우에 따라 JUnit 4 대신에 JUnit 3을 선택할 수도 있다. 우리는 JUnit 4 버전을 이용할 생각이다. JUnit 4 버전은 Java SDK 5.0 이상에서 동작한다. 관련해서는 마찬가지로 JUnit에 대해 살펴볼 때 다시 설명할 예정이다(JUnit 4 항목 꽁지에 붙어 있는 org.hamcrest.core_1.1.0은 뭘까? 이건 matcher 라이브러리인데, 관련해서 JUnit과 함께 따로 항목을 잡아서 설명할 예정이다. 우선은, 나중의 즐거움으로 남겨두자).

참고로 혹시 Quick Fix 메뉴의 제일 마지막 항목인 [Fix project setup…]을 선택했다면 아래와 같은 팝업이 뜰 것이다.



프로젝트 설정 수정(Project Setup Fixes) 화면

for non-commercial purpose only

이런 식으로 선택해도 빌드 경로 내에 JUnit 라이브러리가 포함된다. 자, 지금까지의 소스코드는 다음과 같다. 그리고 에러는 사라졌을 것이다.

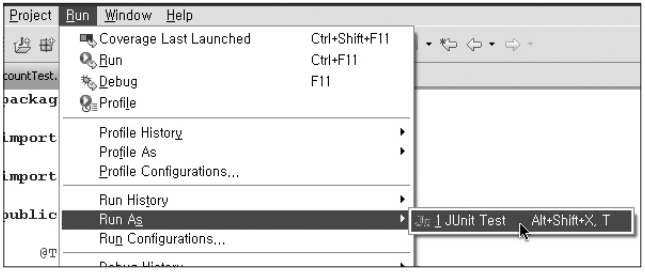
main 메소드가 사라지고, 대신 JUnit 프레임워크가 사용된 테스트 코드

```
package test;
import org.junit.Test;
import main.Account;

public class AccountTest {

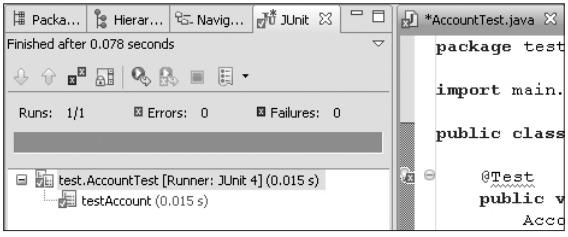
    @Test
    public void testAccount() throws Exception{
        Account account = new Account();
        if ( account == null){
            throw new Exception("계좌생성 실패");
        }
    }
}
```

이제는 main 메소드 대신 JUnit 단위 테스트 프레임워크(이하 JUnit)를 이용해 테스트 메소드를 실행해보자. [Run] → [Run As] → [JUnit Test]를 선택한다.



JUnit 테스트 실행 메뉴

for non-commercial purpose only



두둥! 에디터 창 옆에 상큼한 녹색 막대가 나타났다(인쇄가 흑백이라 구분 안 되겠지만, 녹색이다!)

축하한다! testAccount() 메소드가 실행됐고, 성공했다는 표시로 녹색 막대가 나타났다. 그리고 방금 전까지 작성했던 테스트 코드는 이제 JUnit 프레임워크를 이용하는 코드로 전환됐다. 덕분에 테스트 케이스 코드도 훨씬 간결해졌다. 이제 거의 다 왔다. 마무리로 현재의 테스트 코드를 좀 더 간결하게 만들어보자.

testAccount() 메소드의 if 문을 자세히 살펴보자.

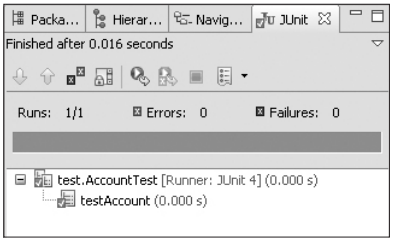
```
@Test
public void testAccount() throws Exception{
     Account account = new Account();
     if ( account == null){
          throw new Exception("계좌생성 실패");
     }
}
```

사실 위의 if 문에서 account를 null과 비교했을 때 일치되는 경우는 발생하지 않는다. 단지 시나리오의 흐름상 '검증'이라는 부분을 표현하기 위해 사용했을 뿐이다. 만일 계좌를 생성하는 부분(new Account())에서 문제가 생긴다면 throw new Exception()을 하지 않아도 자동적으로 예외가 던져질 것이다. 그리고 JUnit이 해당 상황을 처리한다. 따라서 if 문은 현재 의미가 없다. 지우자.

```
@Test
public void testAccount() throws Exception{
     Account account = new Account();
}
```


for non-commercial purpose only

테스트 케이스 항목이 극단적으로 간단해졌다. '계좌를 생성한다. 그때 특별한 에러는 없어야 한다'의 시나리오가 됐다. 다시 테스트 케이스를 실행해보자.



여전히 성공(여전히 인쇄가 흑백이라 구분은 안 되겠지만, 녹색이다!)[13]

JUnit은 테스트 케이스 실행의 성공과 실패를 글자 대신에 색깔이 있는 막대로 표시해 준다. 글자보다 훨씬 간결하고 보는 즉시 성공과 실패가 느껴진다. 실패일 경우에는 녹색 대신에 붉은색 막대로 표시한다. 예전에는 이걸 깃발이라고 표현했었다. 성공하면 녹색 깃발, 실패하면 붉은 깃발. JUnit의 기본 사상 중 하나는 테스트의 성공 여부를, 글자를 이해해서 머리로 판단하는 것이 아니라, O/X 개념의 막대로 단순하게 판단하도록 만든다는 것이다. 결과적으로 개발자의 오해와 잘못된 판단의 여지를 줄여준다. 자세히 보면, 막대 왼쪽 상단에는 Runs라는 항목이 보인다. 수행한 테스트 케이스의 숫자가 여기 적힌다. 그리고 그 옆으로 에러 항목이나 실패 항목들이 표시된다.

앞으로의 실습은 JUnit을 이용해서 테스트 케이스들을 추가하고 실행해나갈 생각이다. 그럼 머리도 살짝 환기시킬 겸, 아래 내용을 읽어보고 두 번째 질문으로 넘어가자.

저자 한마디

클래스 설계 시 속성을 먼저 고민해야 하지 않나?

클래스의 이름과 그 클래스의 동작이 정해졌다면 그 동작에 맞는 테스트를 작성하게 된다. 그런데 우리는 흔히 클래스를 설계할 때 속성에 다소 집중해서 설계를 하는 경향이 있다. Account라는 클래스를 만들기로 결정했다면, 그 다음엔 내부에 어떤 속성을 가져야 할까라는 고민을 시작한다. '계좌 이름도 넣어야 할 것 같고, 고객 이름도 넣어야 할 것 같고… 아… 또 뭘 포함하고 있어야 할까?'라는 고민을 한참 동안 하게 되기 십상이다. 클래스를 정의할 때 중요한 건 '속성'이

13 만일 앞으로도 색이 잘 안 보인다면 Errors와 Failure 항목의 숫자를 살펴보자. 둘 다 0이면 성공인 상태다.

for non-commercial purpose only

아니라 '동작'이다. '동작을 먼저 정하고, 그 동작에 필요한 속성을 고려한다'는 식으로 접근하는 편이, 불필요한 속성이 클래스 내에 섞여 들어가는 걸 줄여준다. 불필요하거나 자리를 잘못 잡은 속성이 클래스에 포함되면 그 결과 또 다른 불필요한 동작이 클래스에 추가되어 클래스의 정체성이나 모듈성을 훼손시킨다. 이를테면 위 예제에서 Account 클래스의 경우는 동작만 정했지 Account 클래스의 속성에 대해서는 고민하고 있지 않다. 진행해나가면서 필요하다면 추가할 예정이다.

두 번째 질문: 잔고 조회

- 잔고 조회(getBalance) 기능 작성을 위한 테스트 케이스를 작성한다.
- 테스트 수행 결과가 오류(error)로 표시된 항목은 실패(failure) 항목으로 만든다.

질문
응답
정제

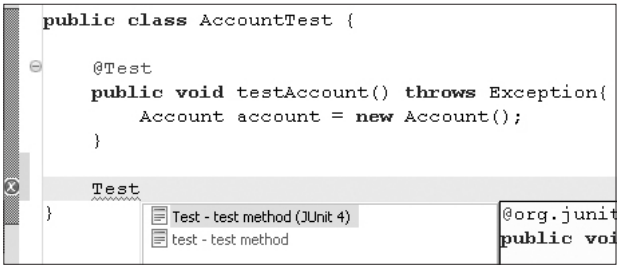
다음은 세 가지 기능 중에서 잔고 조회 부분이다. 잔고 조회 기능은 어떻게 하면 제대로 구현됐는지 확인이 가능할까? 테스트 시나리오를 조금 발전시켜 보자.

~~■ 클래스 이름은 Account~~
■ 기능은 세 가지
  • 잔고 조회
    – 10000원으로 계좌 생성
    – 잔고 조회 결과 일치
  • 입금
  • 출금
* 금액은 원 단위로(예: 천 원 = 1000)

이번엔 잔고 조회 시나리오에 맞춰 테스트 메소드를 작성해보자. 메소드를 직접 다 타이핑하려면 번거로우니까 이클립스의 템플릿 기능을 이용할 생각이다. testAccount()

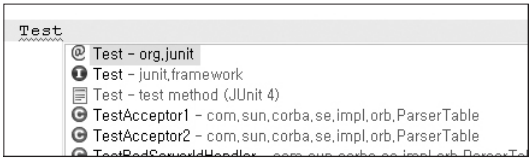
for non-commercial purpose only

메소드 아래에 커서를 놓고 Test까지 타이핑한 다음 Ctrl+Space Bar를 눌러서 [Test - test method (JUnit4)]라고 써 있는 템플릿을 선택한다.



이클립스 템플릿을 이용하기 위해 Test까지 타이핑하고 Ctrl+Space Bar를 누른 모습

만일 위 그림처럼 나오지 않고, 아래와 같이 나온다면 Ctrl+Space Bar 키를 한 번 더 눌러보자(**주의!** Test의 T를 대문자로 쳐야 템플릿이 제대로 동작한다).



어느 경우에도 해당하지 않는다면 아래와 같이 직접 타이핑하자.

잔고 조회(getBalance)를 구현하기 위한 테스트 메소드 testGetBalance()

```
@Test
public void testGetBalance() throws Exception {

}
```

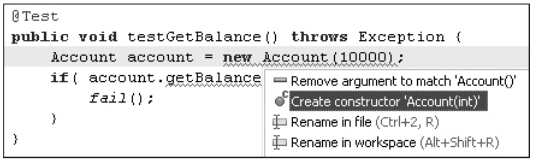
잔고 조회에 해당하는 테스트 시나리오인 **'10000원으로 계좌 생성 → 잔고 조회 결과 일치'**를 앞의 경우와 마찬가지로 코드로 구현해보자.

for non-commercial purpose only

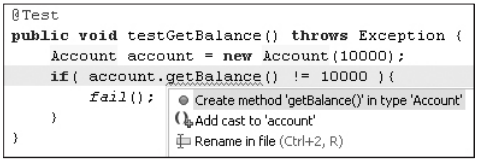
```
@Test
public void testGetBalance() throws Exception {
    Account account = new Account(10000);
    if( account.getBalance() != 10000 ){
        fail();
    }
}
```

fail()은 JUnit에서 제공하는 메소드인데, fail 메소드가 호출되면 해당 테스트 케이스는 그 순간 무조건 실패한다. fail 메소드 대신에 throw new Exception(); 형태로 작성할 수도 있다. 앞에서와 마찬가지로 Quick Fix 기능을 이용해 생성자와 getBalance() 메소드를 작성한다.

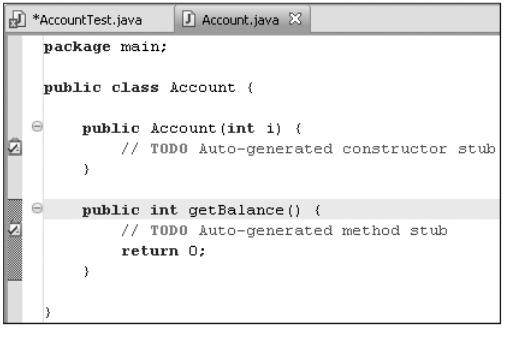
01 인자(argument)로 int를 갖는 Account 생성자를 만드는 걸 선택한다.



02 테스트 메소드로 다시 돌아가서 Quick Fix 기능을 이용해 getBalance() 메소드를 생성한다. getBalance() 메소드 생성을 선택하는 순간 Account 클래스로 화면이 이동된다.



03 Quick Fix를 이용해 변경한 Account 클래스, 완성된 다음엔 저장을 한 번 해주자.



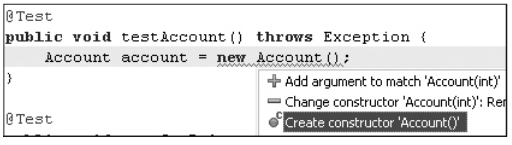
for non-commercial purpose only

04 현재 AccountTest 클래스의 모습, 만일 getBalance()가 10000원이 아니면 실패처리한다.

```
public class AccountTest {
    @Test
    public void testAccount() throws Exception {
        Account account = new Account();
    }

    @Test
    public void testGetBalance() throws Exception {
        Account account = new Account(10000);
        if( account.getBalance() != 10000) {
            fail("Error occurred!");
        }
    }
}
```

다시 AccountTest 클래스로 돌아와 보면, 이전에는 발생하지 않았던 위쪽 testAccount 클래스에서 에러가 난다. Account(int i)라는 생성자가 생겼기 때문이다. 이 경우 Java에서는 기본 생성자를 명시적으로 만들어주지 않으면 Account()를 호출할 수 없다. 마찬가지로 Quick Fix를 이용해 해결해보자.



잠깐! 그런데 이번엔 어느 걸 선택해야 하나

'아, 그럼 그까짓 거 생성자 하나 만들어놓지 뭐~' 하고 Quick Fix를 이용해 Account() 생성자를 바로 만들어줄 수도 있지만, 잠깐만 숨을 돌려보자. 이 상황에서의 선택지는 두 개다.

Case 1 **Account() 생성자를 만든다.**

Case 2 **Account() 생성자는 만들지 않고, Account(int value) 호출만 허용하는 걸로 정한다.**

이때 단순히 Java 언어의 문법적 측면으로만 바라보지 않고, 좀 더 업무적으로 생각할 수도 있다. '초기 예치금액 없이 계좌를 만들 수 있게 할 것인가, 말 것인가?'와 같은 식으로 말이다. Case 1번을 선택하면 예치 금액 없는 계좌 생성을 허용하는 셈이다. 초반

for non-commercial purpose only

에 별 생각 없이 만들었던 테스트 케이스 자체가, "Account 클래스의 구조에 대해 조금 생각해보라"고 개발자에게 말을 걸고 있다. 이런 것이, TDD의 긍정적인 부가효과 중 하나이기도 하다. 지금은 우선 Case 2번을 선택하자. 초기 입금액을 명시적으로 설정하지 않고는 계좌를 생성할 수 없도록 만들 생각이다.

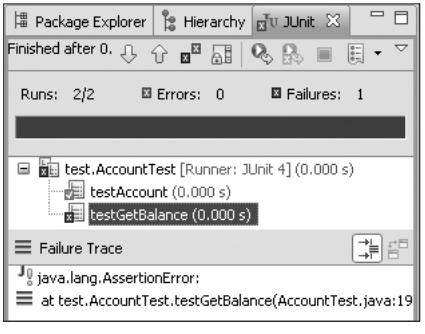
```
public class AccountTest {
    @Test
    public void testAccount() throws Exception {
        Account account = new Account(10000);
    }

    @Test
    public void testGetBalance() throws Exception {
        Account account = new Account(10000);
        if( account.getBalance() != 10000 ){
            fail();
        }
    }
}
```

계좌 생성 시 초기 입금액을 설정한 만든 모습

계좌 생성 시 반드시 초기 입금액을 설정해야 한다. 물론, 이 경우에도 초기 금액 설정이 필요하다 뿐이지, 0원으로 시작할 수 없게 만든 것은 아니다.

이제, 더 이상 AccountTest에서 컴파일 에러는 발생하지 않는다. 테스트를 다시 실행해보자('마지막 실행했던 내용을 다시 실행'에 해당하는 F11 키를 눌러도 된다).

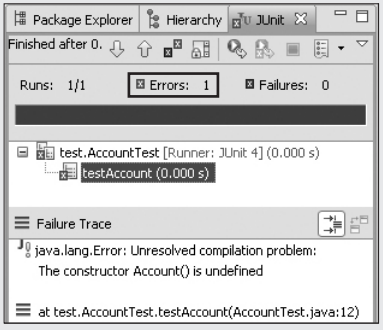


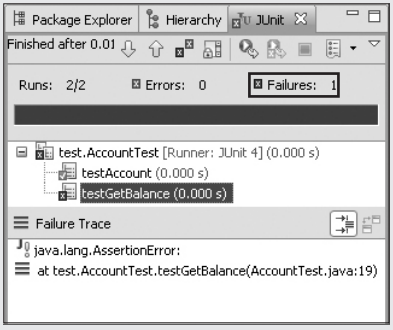
테스트를 실행한다. 이제 이렇게 시스템에 대한 질문이 또 하나 만들어졌다.

for non-commercial purpose only

저자 한마디

오류와 실패





왼쪽은 앞에서 Account() 클래스를 만들면서 실행했을 때의 테스트 결과창이고, 오른쪽은 방금 getBalance()에 대한 테스트 케이스 작성 후 실행한 화면이다. 둘 다 붉은색 막대를 보여주며 실패로 표시하고 있지만, 원인의 종류는 조금 다르다. 왼쪽은 오류(Errors) 항목이 1이고, 오른쪽은 실패(Failures) 항목이 1이다. 오류와 실패의 차이는 무엇일까? 실패는 AssertEquals 등의 테스트 조건식을 만족시키지 못했다는 것을 의미한다. 또, 그로 인해 내부적으로 fail()이 호출됐다는 의미이기도 하다. 오류는 테스트 케이스 수행 중 예상치 못한 예외(exception)가 발생해서 테스트 수행을 멈췄다는 것을 뜻한다. 만일 앞에서 작성한 testGetBalance() 메소드 내부에서 fail() 대신에 예외가 발생했다면, 실패가 아니라 오류로 간주된다. 즉, 아래의 경우는 오류가 된다.

```
@Test
public void testGetBalance() throws Exception {
    Account account = new Account(10000);
    if( account.getBalance() != 10000 ){
        throw new Exception();
    }
}
```

테스트 케이스가 가치를 지니기 위해서는, 어떠한 경우에도 테스트 케이스 그 자체는 정상적으로 끝까지 수행돼야 한다. 그래서 단정문을 실행한 결과 실제값이 예상값과 다르다는 신호인 실패가 나오도록 테스트 케이스를 작성해야 한다. 일반적으로 오류는 작성자가 의도하지 않은 예상치 못한 실패(unexpected failure)를 뜻하며, 이 경우 테스트 케이스 자체에 문제가 있음을 시사한다. 따라서 본인이 작성한 테스트 케이스에 오류(Errors)로 인한 실패가 발생하고 있다면, 빠른 시일 내에 실패(Failures)로 카운트될 수 있게 만들어야 한다.

for non-commercial purpose only

두 번째 응답: 잔고 조회 기능 구현

- 앞서 작성된 테스트 케이스를 이용해 잔고 조회 기능을 구현한다.
- 테스트 실패 시에 메시지를 보여줄 수 있는 구조를 생각해본다.

질문
응답
정제

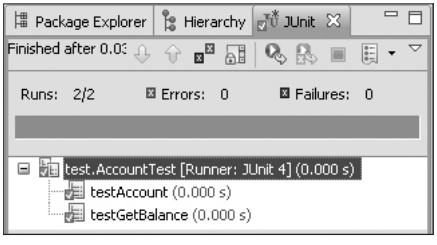
이제 실패하는 질문에 대한 응답으로 녹색 막대를 볼 수 있도록, 계좌의 잔고를 알려주는 getBalance 메소드를 구현해보자. 그런데 약간 비겁하게 앞서 작성한 테스트 케이스의 맹점을 노려보자.

```
public class Account {

    public Account(int i) {
    }

    public int getBalance() {
        return 10000;
    }
}
```

getBalance 메소드 호출 시에 10000을 돌려주도록 하드코딩으로 작성했다. 다시 테스트를 실행해보자.



테스트를 통과했다!! 녹색 막대가 나왔으니 우선 기뻐하자!(정말??)

for non-commercial purpose only

지금쯤 '에엣? 대체 지금 뭘 하자는 거야?'라는 생각이 들 것이다. 일단 저렇게 하드코딩으로 값을 넣어놓는 것도 어처구니가 없고, 이런 식이면 테스트 케이스 성공을 알리는 녹색 막대가 보여도 전혀 기쁘지가 않다. 그럼에도 필자가 이런 짓(?)을 한 데는 두 가지 이유가 있다. 우선 첫 번째 이유.

테스트 케이스를 엉성하게 만들면 테스트 자체를 신뢰할 수 없게 된다는 점을 말하고 싶었다.

그 다음으로는 **테스트 케이스를 통한 제품 코드 구현을 하드코딩으로 시작하는 것도 괜찮은 출발점**이라고 말하고 싶었다. 이런 방식을 필자는 '눈앞의 당신이 바로 범인!' 기법이라고 부른다. 수사를 시작할 때 아무 조건이나 하나만이라도 조건에 맞는 사람을 최대한 빨리 찾아서 '그 사람이 범인'이라고 단정하고 그가 정말 범인인지 이런저런 경우를 대입해보는 방식이다. 대입해볼 수 있는 방법을 모두 대입해봤는데, 여전히 화살표가 처음 찍은 사람이었다면 그 사람이 범인이다. 물론 처음 찍은 사람(처음 구현한 로직)이 범인(솔루션)일 확률은 극히 낮겠지만 어쨌든 수사(개발)를 시작할 때 방황하지 않고 진행할 수 있도록, 출발점을 만들어준다는 데 의의가 있다. 게다가 적어도 이 방식은 두 가지 이상의 테스트 케이스를 작성하도록 자연스럽게 유도해준다는 점도 장점 중 하나다. 마찬가지로 프로그램의 로직을 작성할 때도 테스트 조건을 만족하는 가장 간단한 방법을 우선적으로 선택한다. 이번 경우엔 꽤 과격한 방법으로 하드코딩된 숫자를 사용했다. 이렇듯 응답 단계에 들어가면, 최대한 간단한 로직이나 하드코딩 값을 이용해서 테스트를 최대한 빨리 통과시킨다. 그런 다음, '첫 번째 녹색 막대는 신뢰하지 않기'로 하고 두 번째 테스트 케이스 작성에 바로 들어간다. 앞서 작성한 하드코딩 값, 혹은 엉성하게 작성한 로직이 작성된 테스트 케이스를 모두 통과한다면, 찜찜하고 허탈할 수 있지만 그 로직이 현재 가장 유력한 정답이 된다. 비록 코드가 어설프고 if/else로 점철되어 있다 하더라도, 작성된 테스트 케이스를 전부 통과했다면 우선은 다음 단계로 진행하자. 아직 기회가 남아 있다. 정제 단계에서 리팩토링을 통해 지금의 찜찜한 마음을 달랠 수 있다.

좀 더 생각해볼 주제인 **'그럼, 신뢰할 수 있는 테스트는 어떻게 만들 수 있는가?'**에 대해서는 뒤에서 다루기로 하고, 지금은 로직 검증을 위한 테스트 케이스를 추가해보자.

for non-commercial purpose only

두 개의 새로운 테스트 항목이 추가된 테스트 메소드

```
@Test
public void testGetBalance() throws Exception {
    Account account = new Account(10000);
    if(account.getBalance() != 10000){
        fail();
    }

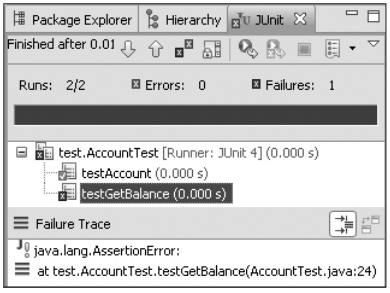
    account = new Account(1000);
    if(account.getBalance() != 1000){
        fail();
    }

    account = new Account(0);
    if(account.getBalance() != 0){
        fail();
    }
}
```

작성하다 보면 느끼겠지만, 이 단계에서도 몇 가지 논의점이 있을 수 있다. 위 예제에는 account 변수를 하나만 선언한 다음 재사용해서 테스트에 이용하고 있다. 그래도 문제 없을까? account2, account3 등으로 처음부터 새로운 변수를 사용해야 하진 않을까? Account 계좌 생성 예제를 10000, 1000, 0 이렇게 세 개만 사용했는데 이 정도면 과연 충분한가? … 이런 내용에 대해서는 인지만 해놓고 뒤에서 차차 고민하기로 하자. 우선은 계속 개발을 진행해나가기로 한다.

자, 이제 testGetBalance의 테스트 케이스를 보강했으니, 테스트를 다시 실행해보자.

for non-commercial purpose only



테스트가 실패로 바뀌었다.

나이스!! 이번엔 테스트가 실패하면서 로직에 문제가 있음을 발견해냈다. '충분한 만큼'[14]의 테스트 케이스가 작성됐다고 생각되면, 다시 Account 클래스의 getBalance 메소드를 제대로 다시 구현해보자.

AccountTest.java | Account.java

```
package main;

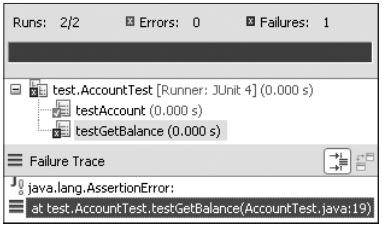
public class Account {
    private int balance;

    public Account(int i) {
    }

    public int getBalance() {
        return this.balance;
    }
}
```

다시 구현된 getBalance() 메소드

상식적인 수준에서의 작성이다. 테스트 케이스를 다시 실행해보자.



붉은색 막대가 보이며 실패

14 사실 충분한 만큼의 테스트라는 건 다소 모호하게 들릴 수 있다. 보통은 현재 집중하고 있는 요구사항, 바로 그 한 가지만을 통과하기에는 적절한 숫자의 테스트 케이스라고 본다. 경험이 축적됨에 따라 노련해지는 부분 중 하나다.

for non-commercial purpose only

엣? 예상과 달리 또 실패했다. 뭐지? 실패가 표시된 부분을 더블클릭해보자.

```
@Test
public void testGetBalance() throws Exception {
    Account account = new Account(10000);
    if( account.getBalance() != 10000 ){
        fail();
    }

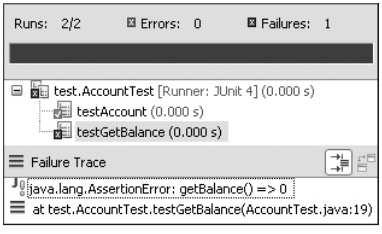
    account = new Account(1000);
    if( account.getBalance() != 1000 ){
```

10000원 아니라서 실패

이상하다. 10000원으로 계좌를 생성한 다음 getBalance()를 호출했는데 10000원이 아니라니. 뭐, 별 방법이 없다. getBalance()가 얼마를 돌려주는지 확인해보자. 디버그(Debug) 기능을 사용할 수도 있지만, 그러면 복잡해지니까 fail() 메소드를 조금 고쳐보자.

```
@Test
public void testGetBalance() throws Exception {
    Account account = new Account(10000);
    if( account.getBalance() != 10000 ){
        fail( "getBalance() => " + account.getBalance() );
    }
```

다시 테스트를 돌려보자.



아! 0이 찍히네… 왜 그럴까? Account 클래스를 다시 잘 살펴보자.

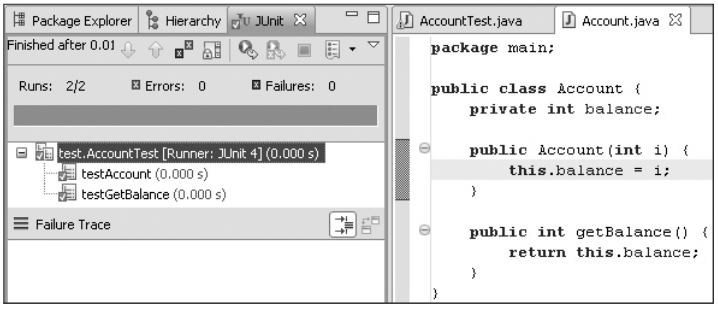
```
public class Account {
    private int balance;

    public Account(int i) {
    }

    public int getBalance() {
        return this.balance;
    }
}
```


for non-commercial purpose only

이런! 계좌를 생성할 때 입력받는 금액을 내부 필드에 저장하는 로직이 빠져 있잖아!! 해당 부분을 수정하고 다시 테스트를 실행해보자.



휴! 드디어 성공이다! 기쁜 마음으로 작업 목록에서 한 항목 더 줄을 그어주자.

~~■ 클래스 이름은 Account~~
■ 기능은 세 가지
~~• 잔고 조회~~
– 10000원, 1000원, 0원으로 계좌 생성
– 잔고 조회 결과 일치
• 입금
• 출금
* 금액은 원 단위로(예: 천 원 = 1000)

저자 한마디

부끄러운 테스트 케이스?

```
@Test
public void testAccount() throws Exception {
    Account account = new Account(1000);
}
```

그리고 이제 와서 하는 이야기지만, 앞서 작성했던 Account 클래스 생성에 대한 테스트 케이스, testAccount() 메소드는 좀 이상하다고 생각하지 않는가? 특별히 무언가를 테스트하는 비교 구문도 없고, 단순히 클래스를 생성하는 부분만 덜렁 들어 있다. 이게 무슨 테스트 케이스란 말인

for non-commercial purpose only

가? 게다가 하는 일도 단순히 new로 Account를 생성하고 있다. 이런 걸 테스트 메소드라 부르기엔 부끄럽지 않나? Java JDK가 망가지기 전에는 결코 문제가 생길 것 같지 않은 테스트 메소드인 것이다. 차라리 지워버리면 어떨까?

일반적으로 생성자 메소드에서 특별한 업무로직을 처리하지 않는다면 굳이 테스트 케이스를 작성하지 않아도 무방하다. 다만, 0원으로 계정 생성 금지라던가 마이너스통장으로 개설 같은 식의 업무로직이 생성자 메소드와 관련 있다면 그때는 생성자에 대한 테스트 케이스도 만들어야 한다. 어떻게 보면 모순처럼 들릴 수도 있겠지만, 생성자 메소드에 대한 테스트 케이스는 작성하는 데 큰 노력이 들지 않기 때문에, 가급적 만들었으면 좋겠다. 테스트 케이스의 부수적인 효과 중 '메소드 사용에 대한 설명서'적인 측면에서 해당 클래스를 사용하게 될 다른 개발자들에게 도움이 되기 때문이다.

두 번째 정제

- 구현된 잔고 조회 로직에 대한 리팩토링 작업을 한다.
- 본격적으로 JUnit 테스트 프레임워크를 사용한다.

질문
응답
정제

앞선 정제 단계에서는 별달리 한 일이 없었다. 그러나 이번엔 살펴볼 부분이 조금 있다.

```
public class Account {
    private int balance;

    public Account(int i) {
        this.balance = i;
    }

    public int getBalance() {
        return this.balance;
    }
}
```

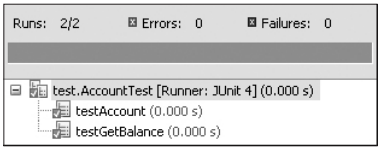

for non-commercial purpose only

Account 클래스 생성자의 파라미터 변수명 i는 의미가 명확하지 않고, 적절하지도 않다. 변수 i의 이름을 money로 변경한 다음, 다시 테스트 케이스를 실행해보자.

```
public class Account {
    private int balance;

    public Account(int money) {
        this.balance = money;
    }

    public int getBalance() {
        return this.balance;
    }
}
```



변함없이 성공! 만일 이때 녹색 막대가 나오지 않는다면, 오히려 어떤 의미론 '굉장하다!'고 할 수 있다.

사실 이런 작업은 다소 부끄럽다고 할 만큼 단순한 작업인데, 좋은 습관을 들이기 위한 연습 단계라고 생각하자. 앞에서도 말했지만, 훌륭한 프로그래머란 좋은 습관을 많이 가진 프로그래머와 동일하다고 본다. 올바른 습관과 연습은 시간이 많이 필요하고 또 그만큼 중요하다. 부디 이런 연습을 나쁘게 생각하진 말자.

정제 작업을 하나 더 해보자. testGetBalance 메소드를 보면, 다음과 같은 부분이 보인다.

```
if(account.getBalance() != 10000){
    fail("getBalance() => " + account.getBalance());
}
```

이 문장은 테스트 케이스의 기본 사상을 나타내고 있는데, **'예상값과 실제값을 비교하는 조건을 만족하지 않으면 실패!'**라는 컨셉이다. 위 경우처럼 다소 장황하게 쓸 수도 있지만,

for non-commercial purpose only

JUnit 테스트 프레임워크에서 제공하는 assertEquals()라는 메소드를 이용하면 좀 더 편리하다.

```
assertEqauals(예상값, 실제값)
assertEqauals("설명", 예상값, 실제값)
```

예상값(expectedValue)과 실제값(actualValue)은 다양한 타입으로 오버라이드(override)되어 있기 때문에 대부분의 타입에 대해 편리하게 비교 가능하다. assertEquals를 현재 예제에 적용해보면 다음과 같이 변경 가능해진다.

```
assertEquals(10000, account.getBalance());
```

만일 테스트 실패 메시지까지 정확하게 표현하고 싶다면 다음과 같이 사용한다.

```
assertEquals("10000원으로 계좌 생성 후 잔고 조회", 10000, account.getBalance());
```

혹자는 "뭣! 이걸 왜 이제 알려줘??"라고 버럭 할 수도 있는데, 워워~!! 잠시 심호흡을 하고, 내 얘기를 들어보기 바란다. TDD를 진행하는 데 있어 assertEquals라는 특정 메소드 자체가 중요한 건 아니다. assertEquals 같은 메소드를 쓰지 않았다고 해서 테스트 주도 개발이 아닌 건 아니다. 자동화된 테스트 케이스를 미리 만들면서 개발한다면 System.out.println() 등을 사용해도 TDD라 불릴 수 있다. TDD에서 중요한 건 **'목표 이미지를 미리 세운다 → 자동화된 테스트 케이스를 작성한다 → 만족시키는 로직을 작성하고 정련한다'**인 거지, 'JUnit을 사용한다'라든가 'assertEquals나 assertTrue 등의 assert 문을 사용한다'가 절대 아니란 사실을 알아줬으면 한다. 그러니 부디 너그러운 마음으로 이해해주고, 앞으로는 if 문 대신에 assertEquals 메소드를 적극 사용하기로 하자.

```
@Test
public void testGetBalance() throws Exception {
    Account account = new Account(10000);
    assertEquals(10000, account.getBalance());

    account = new Account(1000);
```


for non-commercial purpose only

```
        assertEquals(1000, account.getBalance());

        account = new Account(0);
        assertEquals(0, account.getBalance());
    }
```

assertEquals를 사용해 메소드를 위와 같이 바꾼다. 테스트를 실행해보면 여전히 동일한 녹색 막대가 보일 것이다.

이제 나머지 두 개의 기능 구현은 좀 더 스피드를 내서 진행해보자.

세 번째 질문: 입금과 출금 테스트

- 입금과 출금 기능을 구현하기 위한 테스트 케이스를 작성한다.

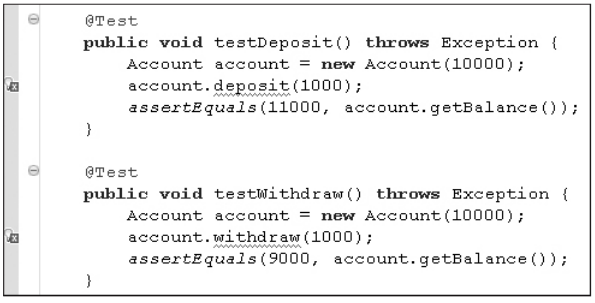
질문
응답
정제

이번에 구현할 기능은 입금하기(deposit)와 출금하기(withdraw)다. 순서대로 testDeposit()과 testWithdraw() 두 개의 테스트 케이스를 작성해보자.

- ~~■ 클래스 이름은 Account~~
- ■ 기능은 세 가지
  - ~~• 잔고 조회~~
    - – 10000원으로 계좌 생성
    - – 잔고 조회 결과 일치
  - • 입금
    - – 10000원으로 계좌 생성
    - – 1000원 입금
    - – 잔고 11000원 확인
  - • 출금
    - – 10000원으로 계좌 생성
    - – 1000원 출금
    - – 잔고 9000원 확인

for non-commercial purpose only

위와 같이 시나리오를 좀 더 구체화한 다음, 그에 대응되는 테스트 코드를 만들자.



```
@Test
public void testDeposit() throws Exception {
    Account account = new Account(10000);
    account.deposit(1000);
    assertEquals(11000, account.getBalance());
}

@Test
public void testWithdraw() throws Exception {
    Account account = new Account(10000);
    account.withdraw(1000);
    assertEquals(9000, account.getBalance());
}
```

Quick Fix를 이용해 Account 클래스에 두 개의 메소드를 자동으로 만든다(소스 자동 생성 기능을 사용할 때는 저장 버튼을 수시로 눌러주는 것을 잊지 말자).

```
public void deposit(int i) {
    // TODO Auto-generated method stub
}

public void withdraw(int i) {
    // TODO Auto-generated method stub
}
```

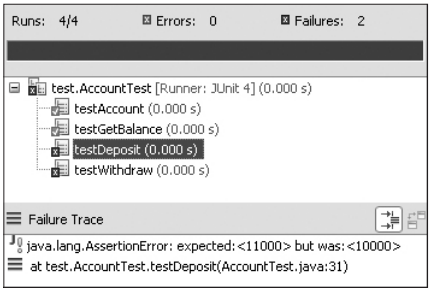
Account 클래스에 자동 생성된 메소드를 확인한다.

```
@Test
public void testDeposit() throws Exception {
    Account account = new Account(10000);
    account.deposit(1000);
    assertEquals(11000, account.getBalance());
}

@Test
public void testWithdraw() throws Exception {
    Account account = new Account(10000);
    account.withdraw(1000);
    assertEquals(9000, account.getBalance());
}
```


for non-commercial purpose only

컴파일 에러가 사라진 것을 확인하고 테스트 실행!



새로 만든 두 개의 테스트가 모두 실패! 각 항목을 클릭해 예상값과 결과값이 어떻게 나와서 실패하는가를 확인한다.

저자 한마디

잠깐만요! 실패하는 테스트 케이스를 동시에 여러 개 만들면 안 된다고 들었는데요?

순수한 TDD 지향자라면 현재 상황을 비판할 수 있다. 왜냐하면 실패하는 테스트 케이스를 두 개나 동시에 만들고 시작했기 때문이다. 일반적으로 TDD에서는 실패하는 테스트를 여러 개 만들어놓고 진행하는 걸 권장하지 않는다. 그리고 본인도 그 규칙을 준수하는 게 좋다고 여긴다. 하지만 때로는 규칙에 철저하게 얽매이기보다는, 상식적인 선에서 지켜야 하는 가이드라인이라고 보는 편이 더 나을 경우도 있다. 물론 TDD가 익숙하지 않다면 우선은 가이드를 잘 준수하는 습관을 들이자.

세 번째 응답: 입금과 출금 기능 구현

- 입금과 출금 기능을 구현한다.

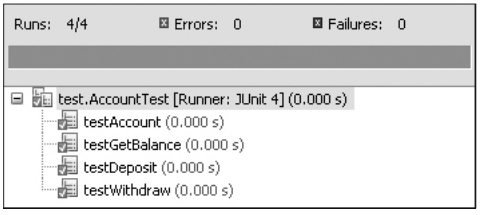
질문
응답
정제

```
public void deposit(int i) {
    this.balance += i;
}
```


for non-commercial purpose only

```
public void withdraw(int i) {
    this.balance -= i;
}
```

Account 클래스에 두 메소드를 구현하고 테스트 실행!



녹색 막대가 나오며 테스트가 성공한다.

~~■ 클래스 이름은 Account~~
■ 기능은 세 가지
~~• 잔고 조회~~
– 10000원으로 계좌 생성
– 잔고 조회 결과 일치
~~• 입금~~
– 10000원으로 계좌 생성
– 1000원 입금
– 잔고 11000원 확인
~~• 출금~~
– 10000원으로 계좌 생성
– 1000원 출금
– 잔고 9000원 확인
* 금액은 원 단위로(예: 천 원 = 1000)

구현이 된 부분은 ToDo 목록에서 지운다.

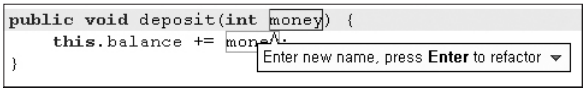
for non-commercial purpose only

세 번째 정제

- 중복해서 나타나는 계좌 클래스 생성 부분을 리팩토링한다.
- 테스트에 사용할 객체를 초기화하기 위한 setUp 메소드를 구현한다.
- ToDo 목록에서 작성된 부분을 제거한다.

질문
응답
정제

변수명을 좀 더 의미 있게 고친다. 방금 전처럼 마우스 오른쪽 버튼 메뉴를 이용해도 되지만, 자주 쓰이는 기능이니까 단축키 Alt+Shift+R (Rename)을 사용하자.



Account 클래스의 최종 모습

```
public class Account {
    private int balance;

    public Account(int money) {
        this.balance = money;
    }

    int getBalance() {
        return this.balance;
    }

    public void deposit(int money) {
        this.balance += money;
    }

    public void withdraw(int money) {
        this.balance -= money;
    }
}
```


for non-commercial purpose only

AccountTest의 현재 모습

```
package test;

import static org.junit.Assert.*;
import main.Account;
import org.junit.Test;

public class AccountTest {
    @Test
    public void testAccount() throws Exception {
        Account account = new Account(10000);
    }

    @Test
    public void testGetBalance() throws Exception {
        Account account = new Account(10000);
        assertEquals( 10000, account.getBalance() );

        account = new Account(1000);
        assertEquals( 1000, account.getBalance() );

        account = new Account(0);
        assertEquals( 0, account.getBalance() );
    }

    @Test
    public void testDeposit() throws Exception {
        Account account = new Account(10000);
        account.deposit(1000);
        assertEquals(11000, account.getBalance());
    }

    @Test
    public void testWithdraw() throws Exception {
        Account account = new Account(10000);
        account.withdraw(1000);
        assertEquals(9000, account.getBalance());
```


for non-commercial purpose only

```
        }
    }
```

이번엔 AccountTest 클래스의 소스 구조 다듬기를 진행해보자.

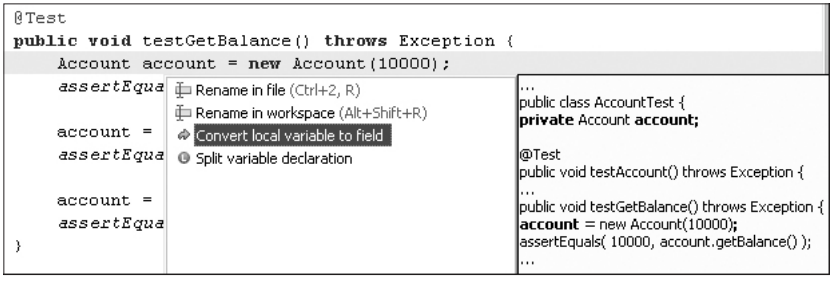
- Account 계정을 생성하는 부분이 반복적으로 나타나고 있다. 이 부분을 클래스의 Field(클래스의 멤버 변수 영역을 지칭한다)로 옮겨보자.

간단하니까 직접 복사 및 붙여넣기(Copy&Paste)로 작업해도 되지만, 이번에도 이클립스의 리팩토링 기능을 사용해보자. 로컬 변수를 필드 변수(혹은 멤버 변수)로 만드는 데는 두 가지 방식이 있는데, 하나는 Quick Fix를 이용하는 방식이고, 다른 하나는 마우스 오른쪽 버튼 메뉴의 Refactoring 메뉴를 이용하는 방식이다. 개인적으로는 Quick Fix를 이용하는 방식이 더 편리하다고 생각하는데, 이클립스 버전이 낮을 경우 지원이 안 된다는 아쉬움이 있다.

정제 계정(account)을 필드로 옮기기

01 Quick Fix를 이용한 방식

account 변수에 커서를 대고 Quick Fix(Ctrl+1)를 누른다.

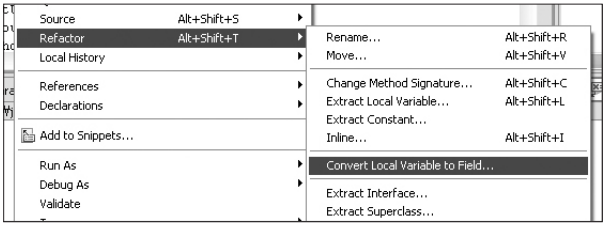


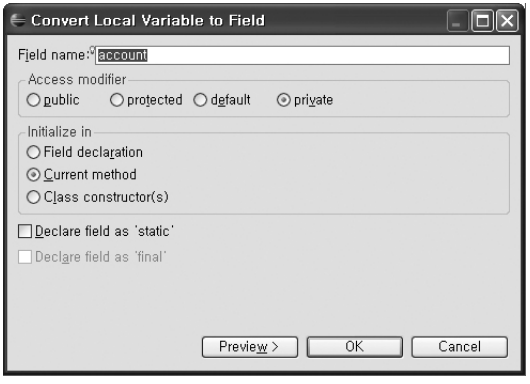
Quick Fix 메뉴에서 '지역 변수를 필드로(Convert Local Variable to Field…)'를 선택한다.

for non-commercial purpose only

02 마우스 오른쪽 버튼 메뉴를 이용한 방식

account 변수로 커서를 이동하고 마우스 오른쪽 버튼을 클릭해서 Refactor 메뉴를 선택한 다음 '지역 변수를 필드로(Convert Local Variable to Field…)'를 선택한다.





변수 이름은 그대로 유지한다.

testDeposit 메소드나 testWithdraw 메소드에서도 Field로 추출한 account를 이용하도록 소스를 수정한다.

```
package test;

import static org.junit.Assert.*;
import main.Account;
import org.junit.Test;

public class AccountTest {
    private Account account;

    @Test
```


for non-commercial purpose only

```
    public void testAccount() throws Exception {
        account = new Account(10000);
    }

    @Test
    public void testGetBalance() throws Exception {
        account = new Account(10000);
        assertEquals( 10000, account.getBalance() );

        account = new Account(1000);
        assertEquals( 1000, account.getBalance() );

        account = new Account(0);
        assertEquals( 0, account.getBalance() );
    }

    @Test
    public void testDeposit() throws Exception {
        account = new Account(10000);
        account.deposit(1000);
        assertEquals(11000, account.getBalance());
    }

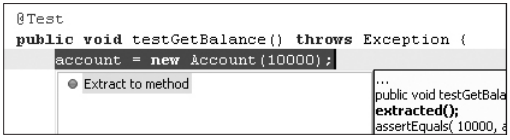
    @Test
    public void testWithdraw() throws Exception {
        account = new Account(10000);
        account.withdraw(1000);
        assertEquals(9000, account.getBalance());
    }
}
```

별건 아니지만, 그래도 고쳤으니까 테스트를 수행해보자. 여전히 녹색 막대가 보이는지 반드시 확인한 다음 진행한다. 소스를 보면 매 테스트 메소드마다 account 변수를 초기화하는 부분이 있다. 자, 이번엔 이 부분을 setup()이라는 메소드로 추출(extract)해보자. 마찬가지로 Quick Fix를 이용하는 방식과 마우스 오른쪽 버튼을 이용하는 방식 둘 다 지원된다. 어느 걸 사용해도 결과는 같으니 자신이 편한 쪽을 택하면 된다.

for non-commercial purpose only

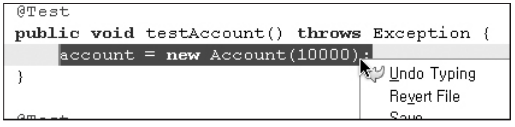
정제 setUp 메소드 추출하기

01 Quick Fix를 이용한 방식

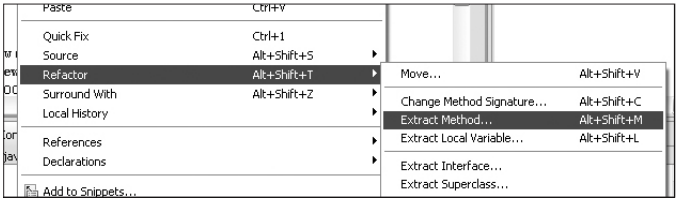


02 마우스 오른쪽 버튼 메뉴를 이용한 방식

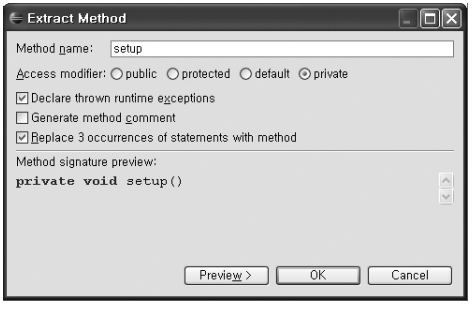
메소드로 뽑아낼 부분을 선택한다.



마우스 오른쪽 버튼의 메뉴에서 [Refactor] → [Extract Method]를 선택한다.



메소드 이름을 정한다.



for non-commercial purpose only

잘 보면 세 군데가 치환 가능하다고 표시된다. 어떻게 바뀔지 미리 보고 싶으면 Preview(미리보기) 버튼을 누른다. 그대로 OK 버튼을 눌러도 무방하다. 왜냐하면 테스트 케이스가 있으니까 실행해보면 소스에 문제가 발생하는지 여부를 알 수 있기 때문이다.

여기까지 정상진행 했다면, 현재 소스는 다음과 같은 모습이 된다.

```
package test;

import static org.junit.Assert.*;
import main.Account;
import org.junit.Test;

public class AccountTest {
    private Account account;

    @Test
    public void testAccount() throws Exception {
        setup();
    }

    @Test
    public void testGetBalance() throws Exception {
        setup();
        assertEquals(10000, account.getBalance());

        account = new Account(1000);
        assertEquals(1000, account.getBalance());

        account = new Account(0);
        assertEquals(0, account.getBalance());
    }

    private void setup() {
        account = new Account(10000);
    }

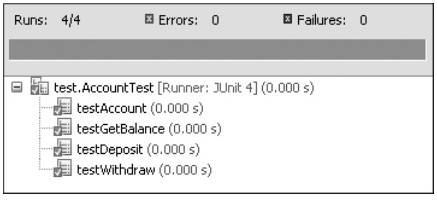
    @Test
```


for non-commercial purpose only

```
    public void testDeposit() throws Exception {
        setup();
        account.deposit(1000);
        assertEquals(11000, account.getBalance());
    }

    @Test
    public void testWithdraw() throws Exception {
        setup();
        account.withdraw(1000);
        assertEquals(9000, account.getBalance());
    }
}
```

저장(Ctrl+S)을 누르고 테스트 케이스를 실행해 보자.



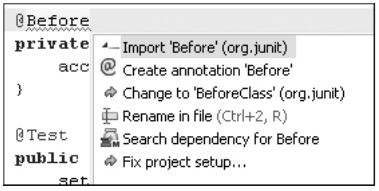
No Problem!

이 부분에서는 크게 드러나고 있진 않지만, 많은 경우 하나의 테스트 케이스는 아래와 같은 순서로 이뤄진다.

테스트 환경 준비 →	테스트 실행 →	환경 정리

그리고 보통 테스트에 사용할 자원이나 객체들을 준비해놓는 부분을 픽스처(fixture, 지지대)라고 부른다. 대부분의 테스트 프레임워크는 픽스처를 지원하고 있다. JUnit에서는 Before와 After라는 개념으로 준비와 정리 작업에 해당하는 처리 방법을 제공한다. 이전 버전인 JUnit 3에서는 준비와 정리를 각각 setUp과 tearDown이라는 메소드로 지원했다. 우리는 현재 JUnit 4 버전을 사용하고 있다. 앞에서 만든 setup 메소드 선언부 위에 @Before라고 적고 저장한 다음 Quick Fix를 이용해 import 처리를 하자.

for non-commercial purpose only



setUp 메소드를 테스트 픽스처 메소드로 지정

이제부터는 @Test로 표시된 각 테스트 케이스가 실행되기 전에 @Before라고 표시된 메소드가 먼저 실행된다(자세한 내용은 뒤에서 다시 다룰 예정이므로 지금은 이 정도로만 이해해두고 조금만 참자). 참고로 이때 setup 메소드 선언부의 접근자를 private에서 public으로 바꿔주지 않으면 테스트 실행 시에 에러가 발생하니 유의하자. 그 다음에는 모든 테스트 케이스 메소드에서 setup() 메소드를 호출하는 부분을 제거한다. 여기까지 진행한 다음 테스트 케이스를 다시 수행해보자.

테스트 클래스의 최종 모습

```
public class AccountTest {

    private Account account;

    @Before
    public void setup(){
        account = new Account(10000);
    }

    @Test
    public void testAccount() throws Exception {
    }

    @Test
    public void testGetBalance() throws Exception {
        assertEquals(10000, account.getBalance());

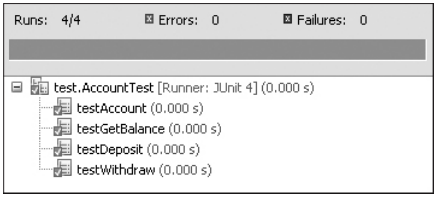
        account = new Account(1000);
        assertEquals(1000, account.getBalance());
```


for non-commercial purpose only

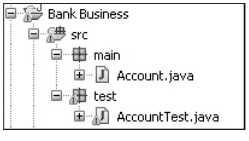
```
        account = new Account(0);
        assertEquals(0, account.getBalance());
    }

    @Test
    public void testDeposit() throws Exception {
        account.deposit(1000);
        assertEquals(11000, account.getBalance());
    }

    @Test
    public void testWithdraw() throws Exception {
        account.withdraw(1000);
        assertEquals(9000, account.getBalance());
    }
}
```



테스트는 여전히 모두 OK!



제품 코드(Account)와 테스트 코드(AccountTest)

자, 이렇게 해서 TDD를 이용한 개발을 완료했다. 작업의 결과물로 제품 코드에 해당하는 Account 클래스와 그에 대응하는 자동화된 테스트 클래스인 AccountTest 클래스가 만들어졌다. 앞으로 Account 클래스에 변경을 가할 일이 생길 때마다 기존 코드들의 안정성을 보장해줄 수 있는 유용한 보호장치가 업무 코드 작성과 동시에 만들어진

for non-commercial purpose only

셈이다. 게다가 자동화되어 있기 때문에 언제든 실행해볼 수 있으며, 테스트 수행 결과도 느낌 팍 오는 형태로 제공된다. 지금은 비록 자세한 설명 없이 이클립스에 내장되어 있는 JUnit 테스트 프레임워크를 사용했지만, 간단한 부분 위주로 사용했기 때문에 이해하는 데 큰 무리는 없었을 것이다. 혹, 어려웠다 하더라도 괴로워하진 말자. JUnit 사용법은 잠시 뒤에 자세히 다룰 예정이니까 조금만 기다리자.

실습 정리

자, 지금까지 진행한 계좌 생성 실습을 한번 정리해보자.

1단계: 계좌 생성

- 구현해야 할 기능을 파악하고, 기능 구현 ToDo 목록을 작성한다.

  ■ 클래스 이름은 Account
  ■ 기능은 세 가지
  • 잔고 조회
  • 입금
  • 출금

  질문 / 응답 / 정제

- 계좌 생성 기능을 구현하기 위한 최초의 테스트 케이스를 만들고 실패하는 모습을 확인한다.

- 계좌 생성 테스트 케이스를 통과하는 코드를 작성한다.

  질문 / **응답** / 정제

- 리팩토링을 적용할 부분이 있는지 찾아본다.
- ToDo 목록에서 완료된 부분을 지운다.

  질문 / 응답 / **정제**

for non-commercial purpose only

계좌 생성 구현 최종 코드

```
public class AccountTest {

    @Test
    public void testAccount() throws Exception {
        Account account = new Account();
    }
}
```

```
public class Account {

    public Account(int i) {
    }
}
```

2단계: 잔고 조회

- 잔고 조회(getBalance) 기능 작성을 위한 테스트 케이스를 작성한다.

질문
응답
정제

잔고 조회
- 10000원으로 계좌 생성
- 잔고 조회 결과 일치

- 테스트 수행 결과가 오류(error)로 표시된 항목은 실패(failure) 항목으로 만든다.

- 앞서 작성된 테스트 케이스를 이용해 잔고 조회 기능을 구현한다.
- 테스트 실패 시에 메시지를 보여줄 수 있는 구조를 생각해본다.

질문
응답
정제

- 구현된 잔고 조회 로직에 대한 리팩토링 작업을 한다.
- 본격적으로 JUnit 테스트 프레임워크를 사용한다.

질문
응답
정제

for non-commercial purpose only

잔고 조회 구현 최종 코드

```
public class AccountTest {

    @Test
    public void testAccount() throws Exception {
        Account account = new Account(10000);
    }

    @Test
    public void testGetBalance() throws Exception {
        Account account = new Account(10000);
        assertEquals(10000, account.getBalance());;

        account = new Account(1000);
        assertEquals(1000, account.getBalance());

        account = new Account(0);
        assertEquals(0, account.getBalance());
    }
}
```

```
public class Account {
    private int balance;

    public Account(int money) {
        this.balance = money;
    }

    public int getBalance() {
        return this.balance;
    }
}
```


for non-commercial purpose only

3단계: 입금/출금

- 입금과 출금 기능을 구현하기 위한 테스트 케이스를 작성한다.

질문
응답
정제

입금
- 10000원으로 계좌 생성
- 1000원 입금
- 잔고 11000원 확인

출금
- 10000원으로 계좌 생성
- 1000원 출금
- 잔고 9000원 확인

- 입금과 출금 기능을 구현한다.

질문
응답
정제

- 중복해서 나타나는 계좌 클래스 생성 부분을 리팩토링한다.
- 테스트에 사용할 객체를 초기화하기 위한 setUp 메소드를 구현한다.
- ToDo 리스트에서 작성된 부분을 제거한다.

질문
응답
정제

입금/출금 구현 최종 코드

```
public class AccountTest {

    private Account account;

    @Before
    public void setUp(){
        account = new Account(10000);
    }
```


for non-commercial purpose only

```
    @Test
    public void testAccount() throws Exception {
    }

    @Test
    public void testGetBalance() throws Exception {
        assertEquals(10000, account.getBalance());

        account = new Account(1000);
        assertEquals(1000, account.getBalance());

        account = new Account(0);
        assertEquals(0, account.getBalance());

    }

    @Test
    public void testDeposit() throws Exception {
        account.deposit(1000);
        assertEquals(11000, account.getBalance());
    }

    @Test
    public void testWithdraw() throws Exception {
        account.withdraw(1000);
        assertEquals(9000, account.getBalance());
    }
}
```

```
public class Account {
    private int balance;

    public Account(int money) {
        this.balance = money;
    }

    public int getBalance() {
```


for non-commercial purpose only

```
        return this.balance;
    }

    public void deposit(int money) {
        this.balance += money;
    }

    public void withdraw(int money) {
        this.balance -= money;
    }
}
```

각 단계는 설명과 그림으로 인해 다소 복잡해 보이고 읽는 시간도 오래 걸렸지만, 다시 실행해보면 불과 10여 분이면 진행할 수 있는 짧은 단계다. 실습을 직접 해보면, 각 단계가 너무 짧은 걸음으로 진행된다고 느낄 수도 있지만, 조금씩 짧은 보폭으로 빨리 걷는 습관을 초반부터 들이도록 노력할 필요가 있다. 그렇게 연습하다 보면, 한 보폭의 속도가 매우 빨라질 것이다. 또한 문제가 생겼을 때도 조금만 뒤로 가면 되기 때문에 빠르고 효율적으로 해결이 가능하다. 긴 실습, 포기하지 않고 여기까지 따라오기가 쉽진 않았을 것이다. 쉽게 배운 건 어쩌면 그만큼이나 가치가 높진 않을 수 있다. 최초의 실습은 이것으로 마칠까 한다.

저자 한마디

잠깐! 개발이 완료됐다고? 복리 이자 계산은?

기억력이 좋은 독자라면 앞서 Account 클래스의 요구사항을 정의할 때 예상 복리 이자 계산이 추가 개발 항목으로 적혀 있었음을 떠올릴 것이다. 그러면서 '이왕 달리는 김에 복리 이자까지 미리 만들어놓자구!'라며 들뜰 수 있는데, 그러지 말자. TDD는 간결함을 추구하는 경제성의 원리가 내포되어 있는 개발 방식이다. 현재 필요한 기능이 아니라면 절대 미리 만들지 말자. 책임감 강하고 욕심 있는 개발자일수록 그냥 넘어가기 어려울 수 있는 부분임을 안다. 나중에 해당 부분에 대한 요구사항이 들어왔을 때, "짜잔! 그럴 줄 알고 제가 미리 만들어놓았지요! 후훗! 바로 됩니다!"라고 말하는 것이 얼마나 짜릿한지도 잘 안다. 하지만 그러지 말자. 여유가 된다면, 현재 작성한 소스코드를 개선하는 데 좀 더 시간을 투자하자. 자신의 코드가 좀 더 이해하기 쉽고, 앞으로의 변화에 민첩하게 대응할 수 있는 구조가 될 수 있도록 고민하는 시간을 가졌으면

for non-commercial purpose only

좋겠다. 초반에 필요하리라 예상했던 기능 중 몇 % 정도를 고객이 결국 사용하게 되는지를 안다면, 집 못 가고 잠 못 자며 고생했던 시절이 떠올라 눈물이 절로 날 것이다.

1.8 TDD의 장점

지금까지 진행한 실습을 떠올리며, TDD는 어떠한 장점들을 제공하는지 살펴보자.

■ 개발의 방향을 잃지 않게 유지해준다

현재 자신이 어떤 기능을 개발하고 있고, 또 어디까지 와 있는지를 항상 살펴볼 수 있다. 그리고 남은 단계와 목표를 잊지 않게 도와준다. 어찌 생각하면 별 것 아니라고 여길 수 있지만, 굉장히 중요한 장점이다. 개발이란 작업은 앉은 자리에서 일어날 때까지 처음부터 끝까지 한 번에 이뤄지는 일도 아니고, 자기 혼자만으로 진행되리라는 법도 없다. TDD를 진행할 때 만들어지는 테스트 케이스들은, 자신이 어디까지 왔고, 앞으로 나아가야 하는 곳이 어디인지를 알려주는 나침반이 된다. 그래서 일부 개발자는 개발 도중 자리를 비우게 될 때, 작성하는 부분의 테스트 케이스를 일부러 실패하도록 만들어놓기도 한다. 다음에 자리로 돌아왔을 때 재시작 시점을 바로 알 수 있도록 말이다.

■ 품질 높은 소프트웨어 모듈 보유

TDD를 통해 만들어진 애플리케이션은 필요한 만큼 테스트를 거친 '품질이 검증된 부품'을 갖게 되는 것과 마찬가지다. 품질 좋은 부품이 꼭 품질 좋은 제품을 보장해주는 건 아니지만, 좋은 제품을 만드는 데 있어 기본 조건임에는 틀림없다. 미항공우주국(NASA)에서 우주선을 만들 때도, 부품 조립을 먼저 하고 특정 기능을 테스트하는 게 아니라, 우선 부품을 엄격히 테스트한 다음 통과한 것들만 조립에 이용한다. 그만큼 단위 모듈의 품질이 중요하다는 뜻이다. TDD와 제품 품질과의 상관관계 및 개발기간에 대한 수치화된 조사내용은 이 책의 부록(A.5절 'TDD에 대한 연구 보고서')에 첨부했다. 간략하게 요약하자면, TDD를 사용하지 않은 개발팀에 비해 TDD를 적용한 팀의 결함률이 최대 1/10 정도까지 감소됐다.

for non-commercial purpose only

■ **자동화된 단위 테스트 케이스를 갖게 된다**

TDD의 부산물로 나오는 자동화된 단위 테스트 케이스들은, 개발자가 필요한 시점에 언제든지 수행해볼 수 있다. 그리고 그 즉시 현재까지 작성된 시스템에 대한 이상 유무를 바로 확인할 수 있다. 또한 기능을 추가한다든가, 수정하게 됐을 때 수행해야 하는 회귀 테스트에 대한 부담도 줄어든다.

농담처럼 개발자는 눈이 좋아야 한다고 말하곤 한다. 왜냐하면 콘솔에 미친 듯이 찍히며 화면 스크롤되는 무수한 메시지 중 원하는 부분의 로그를 눈으로 잡아낼 수 있어야 하기 때문이다. 각종 초능력을 지닌 등장인물들이 등장하는 미국 드라마 〈히어로즈(Heroes)〉에 출연해도 전혀 어색하지 않을 만큼 뛰어난 동체 시력을 가진 개발자들을 가끔씩 만나곤 한다. 그런데 만일 여러분이 단위 테스트 케이스를 갖고 있다면, 그런 능력을 가진 개발자들을 더 이상 부러워하지 않아도 된다.

■ **사용설명서 & 의사소통의 수단**

TDD로 작성된 각 모듈에는 테스트 케이스라고 하는 테스트 코드가 개발 종료와 함께 남게 된다. 비록 테스트 케이스 코드는 고객이 돈을 지불하는 코드에 해당하진 않지만, '품질'을 고려한다면 놓쳐서는 안 되는 귀중한 코드들이다. 그리고 그 테스트 코드들의 가치는 시간이 지나면서 두고두고 빛을 발한다. 그 가치 안에는 고객만을 위한 것이 아니라, 현재의 자신과 주위의 개발자, 그리고 미래의 개발자에게 제공되는 상세화된 모듈 사용 설명서라는 부분도 포함되어 있다. 즐겨 보는 물건은 아니지만, 그렇다고 매뉴얼 없는 TV 등의 가전 제품을 받는다면, 제품에 대한 느낌이 어떨까? TDD를 통해 작성된 테스트 케이스는 사용 설명서이자, 그와 동시에 다른 개발자와 소통하는 커뮤니케이션 통로가 된다.

사실, 무한한 상상력을 필요로 하는 기존 코드(legacy codes)의 난해함과 문서화 부재가 주는 상실감이 IT 프로젝트에서 얼마나 악영향을 미치는지에 대한 연구 사례는 많다. TDD는 이 부분에서도 장점을 갖는다. 일반적으로 TDD는 리팩토링과 단짝으로 진행되며, 리팩토링의 최대 미덕은 '사람이 이해할 수 있는 코드로 만든다'이다. 또한 TDD의 산물인 테스트 케이스들은 그 자체가 API들의 사용 예가 돼준다. 대부분의 개

for non-commercial purpose only

발자는 API 참조매뉴얼을 좋아하지 않는다. 그보다는 상황에 들어맞도록 잘 만들어진 예제를 더 좋아한다. 테스트 케이스를 잘 작성하면, 개발 문서뿐 아니라 테스트 문서까지도 함께 해결할 수 있다. 관련해서는 차차 살펴보기로 한다.

■ **설계 개선**

테스트 케이스 작성 시에는 클래스나 인터페이스, 접근제어자(access modifier), 이름 짓기(naming), 인자(argument) 등에 이르는 개발에 포함된 다양한 설계 요소들에 대해 미리부터 고민하게 된다. 흔히 테스트하기 어렵다고 생각되는 코드들은 객체 설계 원리 중 기본에 해당하는 원칙들이 잘못 적용됐거나 충분히 고려되지 않았을 가능성이 높다. TDD를 진행해나가면서, 테스트가 가능하도록 설계 구조를 고민하다 보면 자연스럽게 디자인을 개선하게 된다. 즉 누가 모니터 옆에서 손가락으로 개선해야 하는 부분을 가리켜주는 것이 아니라, 스스로의 사고와 수련으로 발전해나갈 수 있게 도와준다.

또한 테스트 케이스를 작성할 때는, 작성하게 되는 모듈이 접할 상황에 대해 사전에 고민해보고 준비하게 된다. 미리 고민해보고 작성하는 것과 작성하면서 그때그때 추가하는 것은, 모듈의 응집도와 의존성 정도에 있어 적지 않은 차이를 만들어낸다.

■ **보다 자주 성공한다**

기본적으로, TDD는 매 주기(cycle)를 짧게 설정하도록 권장한다. 그렇게 하면 앞서 말한 녹색 막대를 자주 볼 수 있고, 그때마다 목표를 이뤘다는 성취감을 느낄 수 있다. 이런 성취감은 개발자에게 큰 힘이 되곤 한다. 또한 이런 성공 습관은 개발의 기초를 바꿀 수 있게 도와준다. 기존 개발 가이드 담론의 최고 덕목 중 하나가 '분할하여 정복(Divide & Conquer)'이었다면, 앞으로는 '분할하여 테스트 후 정복(Divide and Test, then Conquer)'을 문제해결의 기본 원칙으로 삼자.

정리

TDD는 개발자가 목표를 세워 개발을 진행해나가고, 설계에 대해 지속적으로 고민할 수 있게 도와준다. 또한 그 와중에 정형화된 형태의 테스트 케이스들을 작성함으로써

for non-commercial purpose only

테스트 과정을 자동화하고, 테스트의 수행 결과와 개발의 목표 달성 여부를 즉각적으로 알 수 있게 된다. 그리고 개발 완료 시점 이후로도 TDD의 부산물인 자동화된 단위 테스트 케이스를 이용한 지속적인 테스트가 가능해진다. 결과적으로 TDD는 개발에 좀 더 집중할 수 있게 도와주고, 프로그램의 안정성에 크게 기여한다.

이 외에도 장점으로 꼽을 내용은 더 많다. 그런 것에 대해서는 여기서 말로 다 적어놓기보다, 이후 다양한 예제와 함께 설명해나갈 예정이다. 이제부터는 Java TDD의 기본이 되는 JUnit 테스트 프레임워크를 비롯하여 TDD 시에 우리가 도움 받을 수 있는 다양한 기법과 도구, 사례들을 학습해보기로 하자.

저자 한마디

엉클 밥의 TDD 원칙

엉클 밥이라는 별명으로 불리는 로버트 마틴(Robert C. Martin)은 객체 지향 개발의 선구자이며, 객체 지향의 수많은 원리와 개념을 만든 사람 중 한 명이다. 그는 또한 애자일 개발 및 TDD 분야에서도 많은 활동을 하고 있는데, 반드시 따르길 권장하는 TDD의 원칙을 다음과 같이 말하고 있다.

❶ 실패하는 테스트를 작성하기 전에는 절대로 제품 코드를 작성하지 않는다.
❷ 실패하는 테스트 코드를 한 번에 하나 이상 작성하지 않는다.
❸ 현재 실패하고 있는 테스트를 통과하기에 충분한 정도를 넘어서는 제품 코드를 작성하지 않는다.

로버트 마틴은 이 세 가지 법칙이 TDD 개발의 주기를 30초에서 1분 사이로 유지시켜 준다고 말한다. TDD의 주기가 짧아진다는 건 그만큼 리듬을 타고 빠르게 진행할 수 있도록 만들어준다는 의미이기도 하다. 위 원칙은 TDD를 시작하는 사람들에게 TDD가 습관이며 개발의 한 부분으로 자리잡을 수 있도록 도와주는 방법이기도 하다. 적극적으로 따르도록 해보자.

출처: Coplien and Martin Debate TDD, CDD and Professionalism
http://www.infoq.com/interviews/coplien-martin-tdd

for non-commercial purpose only