망설이기보다는 차라리 실패를 선택하라.

– 버트런드 러셀

CHAPTER 4

한계 돌파를 위한 노력, Mock을 이용한 TDD

TDD를 할 때 발생하는 어려운 상황에 대해서는 앞에서 간략히 살펴봤다. 그리고 그 한계상황을 돌파하는 방법 중 하나로 Mock에 대해 언급했었다. 이번 장에서는 Mock 객체란 무엇이고, 어떻게 사용할 것이며, TDD 적용 시 발생하는 한계상황을 극복할 수 있도록 어떻게 도와주는지 살펴본다. 또한 Mock 객체 사용의 장점뿐 아니라 단점에 대해서도 살펴보고, Mock을 사용하는 TDD란 어떤 것인지 알아보도록 하자. 이 과정에서 좀 더 자유롭게 TDD를 적용할 수 있는 방법을 익힐 수 있을 것이다. 참고로, 이 장의 내용은 다소 어려울 수도 있다. 하지만 슬쩍 훑고만 지나가더라도 끝까지 읽어보자. TDD를 하는 데 도움이 되는 강력한 무기를 얻게 될 것이다.

4장 체크리스트

- ☐ Mock 객체
- ☐ 테스트 더블
  - ■ 더미 객체
  - ■ 테스트 스텁
  - ■ 페이크 객체
  - ■ 테스트 스파이
  - ■ Mock 객체
- ☐ 상태 기반 테스트와 행위 기반 테스트
- ☐ EasyMock
- ☐ jMock
- ☐ Mockito
- ☐ Mock 사용 시 유의사항

4.1 Mock 객체

Mock 객체란 무엇인가?

예전에 자동차 CF에서 자동차 모양을 나무로 깎아 디자인 원형을 만들어놓은 것을 본 적이 있는지 모르겠다. 설계도면을 보면서, 나무를 깎아 붙여 거의 실제 크기에 맞먹는 크기로 만든 다음 다각도로 살펴보는 장면이 나오는 CF였다. 이때 나온 자동차 모형을 Mock이라고 한다. 자동차의 디자인을 검토하기 위해 실제 재료로 만들려면 비용과 시간이 많이 들기 때문에, 비교적 제작하기 쉽고 비용이 덜 드는 나무를 재료로 만들어본 것이다. 이렇듯 일반적으로 Mock이란, 조각하기 쉬운 재료(보통 나무나 점토 등)를 이용해 추후 만들어질 제품의 외양을 흉내 낸 모조품을 말한다. 마찬가지로 소프트웨어 개발에 있어서도 모듈의 겉모양이 실제 모듈과 비슷하게 보이도록 만든 가짜 객체를 Mock 객체라고 한다. 실제 객체를 만들기엔 비용과 시간이 많이 들거나 의존성이 길게 걸쳐져 있어 제대로 구현하기 어려울 경우, 이런 가짜 객체를 만들어 사용한다.

예를 들어, 사용자 암호를 변경하는 기능을 구현하기 위해 테스트 케이스를 먼저 작성한다고 가정해보자.

사용자 암호 저장 기능에 대한 테스트 케이스

```
@Test
public void testSavePassword() throws Exception {
    UserRegister register = new UserRegister();

    String userId = "sweet88";
    String password = "potato";

    register.savePassword(userId, password);
    assertEquals(password, register.getPassword(userId));
}
```

TDD의 정석대로, savePassword 기능을 구현하기 위해 위와 같은 테스트 케이스를 만들었다. 그런데 savePassword 기능 구현 시 추가 요구사항이 붙어 있는 걸 발견했

다. 사용자의 패스워드는 반드시 암호화한 다음에 저장해야 한다는 것이다. 그리고 사용하게 될 암호화 모듈의 스펙은 다음과 같이 정해져 있었다.

```
public interface Cipher {
    public String encrypt(String source);
    public String decrypt(String source);
}
```

논의 결과 우리 제품에는 MD5[1] 기반 암호화 모듈이 적정하다고 판단됐고, MD5Cipher라는 이름의 모듈로 해당 기능을 독립적으로 구현하기로 결정했다. '이제 암호학까지 공부해야 하나?'라는 걱정을 하고 있는데, 다행히도 해당 암호화 모듈은 직접 팀에서 만들지 않고 옆 팀에서 만들어 제공하기로 결정됐다. (이게 과연 다행인 건가?) 어쨌든, 옆 팀에서 받기로 했으니 암호화 부분은 제외하고 테스트 케이스를 작성해서 빨리 savePassword 기능 구현에 들어가기로 했다.

```
@Test
public void testSavePassword() throws Exception {
    UserRegister register = new UserRegister();
    Cipher cipher = ????      // 옆 팀에서 만들어주기만 하면 되는데…

    String userId = "sweet88";
    String password = "potato";

    register.savePassword( userId, cipher.encrypt(password) );
    String decryptedPassword = cipher.decrypt(register.getPassword(userId));
    assertEquals(password, decrytedPassword);
}
```

1 MD5(Message-Digest algorithm 5): 론 리베스트(Ron Rivest)라는 암호학자에 의해 1992년에 만들어진 단방향 해시함수(one way hash function)다. 기본적으로 메시지를 역으로 찾아낼 수 없도록 단방향으로만 변환해주는 함수로, 일종의 암호화(encryption)만 가능하고 복호화(decryption)는 불가능하게 만들어주는 함수다. 그런데 원칙적으로는 복호화가 확률적으로 0에 가깝지만, 생일역설(birthday paradox) 현상으로 값이 충돌하는 일이 발생해서 현재는 실용적인 수준에서만 사용되고 있다. http://blog.doortts.com/65에서 해당 내용에 대한 좀 더 자세한 설명을 살펴볼 수 있다.

문제는 MD5Cipher 클래스를 옆 팀에서 아직 제공하질 않고 있어서, 위와 같은 테스트 메소드를 만들어놓고도, 실행은커녕 컴파일조차 못하고 있다는 점이다. 사실 우리 팀 입장에선 암호화 기능, 그 자체는 관심 밖이다.

> 'savePassword 기능을 구현해야 하는데 이거야 원… TDD고 진행하지 말고 이번엔 그냥 구현에 바로 들어갈까?'

절충안으로, 앞에서 작성한 테스트 케이스를 그대로 이용해서 모듈을 작성할 수도 있지만, 그건 암호화 기능 적용 자체를 누락한 채 개발하는 셈이다. 그렇다고 암호화 모듈을 직접 구현할 수도 없고, 옆 팀에서 암호화 모듈을 만들어줄 때까지 놀 수도 없지 않은가? 그래서 찾은 방법이, MD5Cipher처럼 보이는 객체를 만들어서 개발에 사용하는 것이었다. 일단, 스펙(Cipher 인터페이스)이 있으니, 스펙을 구현하는 MockMD5Cipher라는 클래스를 만들었다. MockMD5Cipher 클래스를 만들기 전에 암호화할 대상인 문자열 'potato'를 웹툴[2]을 이용해 해시 문자열을 찾아두었다.

대상 문자열	해시 문자열
potato	8ee2027983915ec78acc45027d874316

테스트 케이스에 사용할 수 있도록 동작을 흉내 내어 구성한 MockMD5Cipher 클래스

```
public class MockMD5Cipher implements Cipher {

    @Override
    public String decrypt (String source) {
        return "potato";
    }

    @Override
    public String encrypt (String source) {
        return "8ee2027983915ec78acc45027d874316";
    }
}
```

2 md5 Hash Generator 웹페이지: http://www.miraclesalad.com/webtools/md5.php

구현된 내용 자체는 바보 같다는 생각이 들 수도 있지만, 우리가 정말 구현해야 하는 savePassword 기능을 테스트 케이스로 만들기엔 충분한 수준의 코드다. 최종적으로 만들어진 테스트 케이스는 다음과 같다.

savePassword 메소드 구현을 시작할 수 있도록 준비를 마친 테스트 케이스

```
@Test
public void testSavePassword() throws Exception {
    UserRegister register = new UserRegister();
    Cipher cipher = new MockMD5Cipher();

    String password = "potato";
    String userId = "sweet88";
    register.savePassword(userId, cipher.encrypt(password));
    String decryptedPassword =
                cipher.decryption(register.getPassword(userId));
    assertEquals(password, decryptedPassword);
}
```

Mock 객체는 이렇듯 우리가 구현을 하는 데 필요하지만 실제로 준비하기엔 여러 가지 어려움이 따르는 대상을 필요한 부분만큼만 채워넣어서 만들어진 객체를 말한다.

언제 Mock 객체를 만들 것인가?

테스트 케이스 작성이 어려운 상황과 Mock 객체가 필요한 상황은 종종 일치하곤 한다. 대부분의 경우 모듈이 가진 '의존성'이 근본적인 원인이 된다. 그게 환경적이든지, 아니면 팀이 처한 상황적이든지 간에, 모듈이 필요로 하는 '의존성'은 테스트 작성을 어렵게 만든다. 그리고 그 의존성을 단절시키기 위해 Mock 객체가 사용된다. 이미 앞에서 간단한 예를 하나 봤지만, 언제 Mock 객체가 사용되는지 좀 더 자세히 살펴보자.

01 테스트 작성을 위한 환경 구축이 어려워서

- **환경 구축을 위한 작업 시간이 많이 필요한 경우에 Mock 객체를 사용한다.** 오라클 데이터베이스를 설치해야 테스트가 가능하다든가, 웹 서버나 웹 애플리케이션 서버, FTP 서버 등을 설치해야만 테스트 케이스가 가능해지는 경우가 바로 Mock이 필요한 상황이다.
- **경우에 따라서는 특정 모듈을 아직 갖고 있지 않아서 테스트 환경을 구축하지 못할 수도 있다.** 그런 상황에 대한 이유는 다양하다. 개발이 안 돼서, 혹은 모듈을 아직 넘겨받지 못해서, 버그가 있어서 등등 처한 상황만큼이나 다양하게 존재한다.
- **타 부서와의 협의나 정책이 필요한 경우에도 Mock이 필요하다.** 연계 모듈이라서 다른 쪽에서 승인을 해줘야 테스트가 가능한 경우, 방화벽으로 막혀 있어서 통과가 어려운 경우 등이 이에 속한다.

02 테스트가 특정 경우나 순간에 의존적이라서

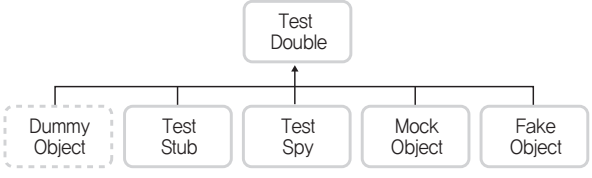
특정한 경우에 의존적으로 동작하는 기능들이 있다. 이런 경우는 특히 예외처리를 테스트할 때 많이 접하게 된다. 이를테면 FTP 클라이언트 프로그램을 만드는데, 네트워크 연결의 접속시간제한(timeout)을 구현한다고 가정해보자. 접속 시도 후 5초까지 1초마다 접속을 재시도하고 그 이후엔 연결 실패 메시지를 띄우는 기능을 테스트해야 한다. 3초까지는 접속이 안 돼서 접속을 시도하다가 4초에, 즉 네 번째 재시도에 접속이 되는 경우를 테스트 케이스로 만들려면 어떻게 해야 할까? 옆에 타이머를 놓고 네트워크 선을 뽑았다, 꼈다 할 수는 없지 않겠는가? 혹은 파일을 이용하는 프로그램을 작성한다고 가정해보자. 작업 중간에 파일에 작업 내용을 기록하고자 한다. 이때 만일 데이터가 부정확하게 저장되는 경우에 대비해, 이를 처리하는 코드를 작성해야 한다. 그런데, 그런 특정한 상황을 테스트 시점마다 발생하도록 만드는 것이 어려울 수 있다. 이럴 때 Mock을 이용하면 특정 상태를 가상으로 만들어놓을 수 있다.

03 테스트 시간이 오래 걸려서

테스트 케이스의 실행시간 단축을 위해 Mock이 사용될 때도 있다. 시간이 오래 걸리는 테스트가 있다. 개인 PC의 성능이나 서버의 성능 문제로 인해 오래 걸릴 수도 있지만, 그것보다는 특정한 모듈을 호출했을 때, 속칭 멀리 갔다 오느라 시간이 걸리는 경우가 있다. 이렇듯 기능 수행 그 자체보다 테스트 수행 시 영향을 미치는 다른 부분으로 인해 테스트 시간이 오래 걸리는 경우 Mock이 사용된다. 이럴 때는 보통 시간이 오래 걸리는 구간이나 모듈을 Mock으로 통째로 대체해서 만든다. 물론 이 경우 Mock으로 대체하는 부분은 신뢰도 높은 모듈(high reliability module)이란 가정이 선행돼야 한다. 아니면 Mock 대체 모듈에 대해서는 다른 부분에서 반드시 테스트할 수 있어야 한다.

4.2 Mock에 대한 기본적인 분류 개념, 테스트 더블

『xUnit Test Patterns』의 저자인 제라드 메스자로스(Gerard Meszaros)는 책을 쓰면서 다양한 용어를 많이 만들어냈는데, 그중 하나가 **테스트 더블(Test Double)**이라는 단어다.[3] 이는, '대역, 스턴트맨'을 나타나는 스턴트 더블(Stunt Double)이라는 단어에서 차용해온 단어로, 오리지널 객체를 사용해서 테스트를 진행하기가 어려울 경우 이를 대신해서 테스트를 진행할 수 있도록 만들어주는 객체를 지칭한다. 앞에서 설명한 Mock 객체와 의미가 겹치는데, 제라드는 Mock 객체를 좀 더 프레임워크와 밀접한 형태로 설명하면서 **테스트 더블**의 하위로 분류해놓았다.



테스트 더블의 분류

3 테스트 더블 외에도 제라드가 만든 단어 중에 흔히 많이 쓰이는 단어로 SUT라는 단어가 있다. SUT는 System Under Test, 즉 테스트 대상이 되는 시스템(이나 모듈)을 뜻한다. 이 책에서는 SUT라는 단어 대신에 **테스트 대상 모듈**이나 **테스트 대상 메소드** 등의 일반적인 용어를 사용했다.

그의 분류법은 일반적으로 대다수의 사람에게 받아들여지는 분위기다. 특히 『리팩토링』의 저자인 마틴 파울러(Martin Fowler)[4]가 그의 분류 방식을 자신의 사이트[5]에 게재한 이후 좀 더 많은 사람들에게 받아들여지기 시작했다. 하지만 많은 수의 개발자는 제라드의 엄밀한 구분법에 따라 이야기하기보다는 좀 더 포괄적이고 보편적인 개념으로 Mock이라는 단어를 사용하고 있다. 하지만 그럼에도, 제라드의 분류법은 알아둘 만한 가치가 있다.

테스트 더블에 대한 설명을 하기에 앞서 예제를 하나 살펴보자. 인터넷 쇼핑몰에서 유저에게 쿠폰을 발급하는 업무를 가정해보자. 고객이 쿠폰을 발급받아 저장하고, 그 내역을 확인할 수 있는 기능을 구현하려고 한다. 쿠폰은 종류가 매우 다양하고 복잡한 형태로 되어 있다. 현재 쿠폰은 인터페이스이고, 다음과 같이 정의되어 있다.

```
public interface ICoupon {
    public String getName();                // 쿠폰 이름
    public boolean isValid();               // 쿠폰 유효 여부 확인
    public int getDiscountPercent();        // 할인율
    public boolean isAppliable(Item item); // 해당 아이템에 적용 가능 여부
    public void doExpire();                 // 사용할 수 없는 쿠폰으로 만듦
}
```

쿠폰 받기 기능 구현용 테스트 시나리오를 간략히 적어보면 다음과 같다.

유저 ID를 기준으로 신규유저 객체를 생성
→ 현재 쿠폰 확인
→ 신규 쿠폰 받기
→ 유저의 보유 쿠폰 개수 확인
(우선은 보유 쿠폰 수가 증가됐는지 판단하는 수준에서 작성)

4 『UML 디스틸드(UML Distilled)』와 『리팩토링(Refactoring)』의 저자이며, Thought Works라는 회사에서 (무려) 과학자(scientist) 직함으로 일하고 있는 엔지니어다. 그의 책은 20세기 소프트웨어 개발에 많은 영향을 끼쳤다. 특히, 『리팩토링』은 리팩토링 분야를 수면 위로 떠올린 최초의 책으로 평가받으며, 종종 『GoF의 디자인 패턴』에 비견될 만한 저서로 꼽힌다. 또한 그의 사이트 martinfowler.com은 전 세계 IT 서적 및 논문에서 가장 많이 참조되는 사이트 중 하나가 됐을 정도로 알찬 내용들이 담겨 있다. 근시일 내에 BDD 관련 책을 발표할 예정인데, 많은 사람이 주목하고 있다.

5 http://martinfowler.com/bliki/TestDouble.html

이제 TDD 방식으로 테스트를 먼저 구현하다 보면 쿠폰 부분에서 작업이 멈추게 된다.

```
public class UserTest {

    @Test
    public void testAddCoupon() throws Exception {
        User user = new User("area88");
        assertEquals("쿠폰 수령 전", 0, user.getTotalCouponCount());

        ICoupon coupon = ??????  // 아! 여길 어떻게 처리해야 하지?

        user.addCoupon(coupon);
        assertEquals("쿠폰 수령 후", 1, user.getTotalCouponCount());
    }
…
```

객체 지향 원칙에 기반해서, 의존성을 최소화하는 형태로 공동 개발을 하다 보면, 빠지지 않고 계속 나오는 부분이 바로 인터페이스다. 마찬가지로 여기서도 ICoupon이라는 인터페이스가 나오는데, 테스트 케이스를 만드는 데 있어 다소 걸림돌이 된다. 현재 쿠폰은 테스트 대상은 아니고, 테스트에 사용되는 일종의 테스트 픽스처다. 우리가 집중하려는 건 User 클래스의 메소드 구현이다. 하지만 ICoupon의 구현 객체가 없으면 정상적인 테스트 케이스를 작성할 수가 없다. 어떻게 해야 할까? ICoupon을 구현하는 클래스를 만들어야 하나? 그건 우리가 개발할 부분이 아니라면 어떻게 해야 할까? 아니면, ICoupon 구현 자체가 쉽지 않은 부분이라면 어떻게 해야 할까?

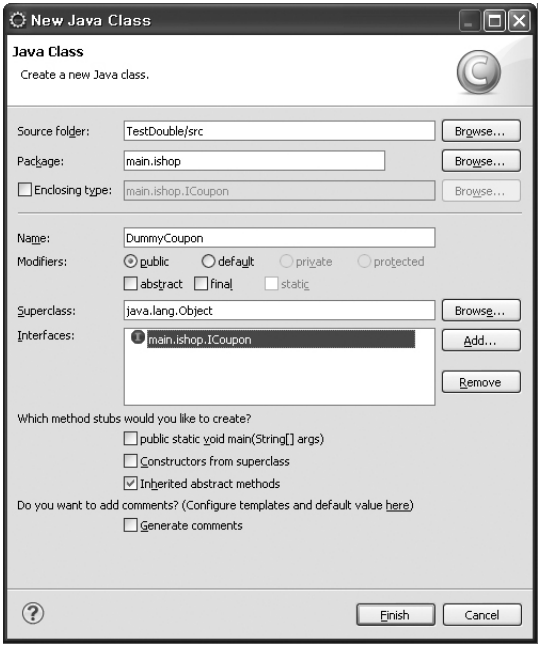
자, 이제부터 **테스트 더블**은 이 상황을 어떤 식으로 극복하는지, 그리고 **테스트 더블**의 분류에 따라 어떻게 적용되는지 하나씩 살펴보자.

더미 객체(Dummy Object)

더미 객체는 말 그대로 멍청한 모조품, 단순한 껍데기에 해당한다. 오로지 인스턴스화될 수 있는 수준으로만 ICoupon 인터페이스를 구현한 객체다.

그럼, ICoupon을 구현하는 DummyCoupon 클래스를 만들어보자.

이클립스를 이용하면 매우 간단히 만들 수 있다. 새로운 클래스를 선택하고 Interfaces에 ICoupon을 Add로 추가한 다음 Finish를 누르자.




```
package main.ishop;

public class DummyCoupon implements ICoupon {

    @Override
    public int getDiscountPercent() {
        return 0;
    }

    @Override
    public String getName() {
        return null;
    }
```

```
    @Override
    public boolean isAppliable(Item item) {
        return false;
    }

    @Override
    public boolean isValid() {
        return false;
    }

    @Override
    public void doExpire() {
    }
}
```

이클립스의 기능을 이용하면 인터페이스의 메소드가 아무리 많아도 간단히 껍데기(class outline)가 만들어진다. 이제 UserTest 클래스의 테스트 메소드 작성을 마무리 해보자.

```
@Test
public void testAddCoupon() throws Exception {
    User user = new User("area88");
    assertEquals(0, user.getTotalCouponCount());

    ICoupon coupon = new DummyCoupon();

    user.addCoupon(coupon);
    assertEquals(1, user.getTotalCouponCount());
}
```

이제 테스트 메소드에 맞춰서 addCoupon 메소드를 구현하면 된다.

더미 객체는, 단지 인스턴스화된 객체가 필요할 뿐 해당 객체의 기능까지는 필요하지 않은 경우에 사용한다. 따라서 해당 **더미 객체**의 메소드가 호출됐을 때의 정상 동작은 보장되지 않는다. 일부 개발자는 **더미 객체**의 메소드는 호출을 가정하고 만들어진 것이 아니기 때문에, 만일 호출 시엔 예외를 발생시키게 만들어놓아야 한다고 말한다. 보통

은 위에서 만들어놓은 것처럼, 타입 기본값(0, null, false 등)으로 반환값을 만들어주는 선에서 마무리한다.

```
public class DummyCoupon implements ICoupon {
…
    @Override
    public String getName() {
        throw new UnsupportedOperationException("호출되지 않을 예정임");
    }
…
```

그런데 만일 기본 타입이 아닌, 특정 상황을 가정해야 하는 경우가 생긴다면 다른 식의 접근이 필요하다. 이를테면 DummyCoupon 클래스로 생성된 coupon 인스턴스에서 coupon.getName()이나 coupon.getDiscountRate() 같은 메소드의 호출이 테스트 케이스 작성 과정에 필요한 경우다. 그리고 메소드가 호출되면 구체화된 쿠폰 이름이나 쿠폰 할인율을 리턴값으로 반환해줘야 테스트가 정상 진행 가능하다고 가정해보자. 이럴 때는 **더미**보다는 좀 더 발전된 객체가 있어야 한다. 그런 기능을 하는 객체를 **테스트 스텁**이라고 부른다.

테스트 스텁(Test Stub)

테스트 스텁(Stub)[6]은 더미 객체가 마치 실제로 동작하는 것처럼 보이게 만들어놓은 객체다. **더미 객체**로 만들어진 DummyCoupon의 경우에도, 메소드가 호출되면 동작을 하긴 한다. 다만, 리턴 타입이 있는 메소드는 타입 기본값으로 리턴이 되고, void 메소드일 경우 아무 일도 안 일어난다. 반면에 **테스트 스텁**은 객체의 특정 상태를 가정해서 만들

6 'Stub'은 여기저기서 많이 쓰이지만 제대로 해석되거나 해설된 적은 거의 없는 단어 중 하나다. 사전에는 보통 대표어로 '그루터기'나 '꼬투리' 같은 뜻으로 나오는데, IT에서는 종종 '껍데기'의 의미로 쓰인다. 하지만 Stub은 단순한 껍데기는 아니고, 오리지널을 확인하거나 유추할 수 있는 수준의 '껍데기'를 뜻한다. 따라서 Test Stub의 Stub은 실제 객체는 아니지만, 겉모양이 실제와 같고 오리지널 객체를 유추할 수 있는 무언가라고 생각하면 되겠다. 참고로 RMI(Remote Method Invovation) 통신에서도 Stub/Skeleton이라는 용어가 사용된다. 클라이언트가 마치 실제 객체인 것처럼 호출에 사용하는 부분을 Stub이라고 부른다. 마찬가지로 클라이언트가 갖고 있는 Stub은 단순히 껍데기에 지나지 않는다. 이 경우 실제 오리지널은 네트워크 너머 서버에 존재한다. 다만 클라이언트는 오리지널 객체가 이렇게 생겼을 거라고 Stub을 통해 판단(추측)할 뿐이다.

어놓은 단순 구현체다. 특정한 값을 리턴해주거나 특정 메시지를 출력하는 등의 작업을 한다. 앞에서 Mock 객체를 설명할 때 만들었던 MockMD5Cipher 클래스가 스텁에 해당한다. 다시 예제로 돌아가서, 이번엔 위에 있는 ICoupon을 다시 **테스트 스텁**으로 구현해보자. DummyCoupon을 만들 때와 동일한 방식으로 만들어도 되는데 이미 DummyCoupon이 만들어져 있으니까, 이클립스의 기능을 이용해 간단히 만들어보자. 패키지 탐색기에서 DummyCoupon을 선택한 다음 복사하여 붙여넣기(Ctrl+C → Ctrl+V)를 차례대로 눌러보자. 모양 자체는 같은 위치에서 같은 파일로 붙여넣는 셈인데, 다음과 같은 창이 뜰 것이다.

파일 이름이 중복되니 이름을 변경해서 붙여넣으라고 한다.

이름을 StubCoupon이라고 바꾼 다음 OK를 누르자. DummyCoupon과 동일하지만, 클래스 이름만 StubCoupon으로 바뀐 걸 알 수 있다.

복사된 DummyCoupon. 똑똑한 이클립스가 코드의 클래스명까지 StubCoupon으로 변경해줬다.

```
public class StubCoupon implements ICoupon {


        @Override
        public int getDiscountPercent () {
            return 0;
        }
    …
    …
```

이제 구현부를 넣어보자.

테스트 스텁으로 구현된 쿠폰 클래스

```
public class StubCoupon implements ICoupon {
    @Override
    public int getDiscountPercent() {
        return 7;
    }

    @Override
    public String getName() {
        return "VIP 고객 한가위 감사쿠폰";
    }

    @Override
    public boolean isAppliable(Item item) { return true; }

    @Override
    public boolean isValid() { return true; }

    @Override
    public void doExpire() { }
}
```

이렇게 구현한 코드를 **테스트 더블** 중 **테스트 스텁**이라고 부른다. 사실 **더미**랑 별로 다른 게 없어 보인다. **스텁**과 **더미**의 차이점을 정리해보면 다음과 같다.

- 단지 인스턴스화될 수 있는 객체 수준이면 **더미**
- 인스턴스화된 객체가 특정 상태나 모습를 대표하면 **스텁**

위 StubCoupon은 인스턴스화됐을 때 'VIP 고객 한가위 감사쿠폰'이 된다. 테스트 케이스에서 'VIP 고객 한가위 감사쿠폰'을 테스트 픽처스로 사용할 수 있게 됐다. 하지만

StubCoupon은 하드코딩되어 있기 때문에, 아직까진 극히 한정적인 부분에서만 정상 동작한다. 그리고 위 예제에서는 테스트에 사용되지 않는 부분도 여기저기 값을 지정해 놓았지만, 실제로 **스텁**을 사용할 때는 테스트에 필요한 메소드 부분만 하드코딩해놓으면 된다. 위와 같은 **스텁** 방식은 다음과 같은 테스트 케이스 작성에 사용될 수 있다. 마지막으로 고객이 받은 쿠폰을 꺼내와서 예상 쿠폰과 일치하는지 확인하는 테스트 케이스다. 단, User 클래스는 여기서 직접 소개하진 않지만, getLastOccupiedCoupon() 메소드로 마지막에 획득한 쿠폰을 찾을 수 있다고 가정한다.

```
@Test
public void testGetLastOccupiedCoupon() throws Exception {
    User user = new User("area88");
    ICoupon eventCoupon = new StubCoupon();
    user.addCoupon(eventCoupon);
    ICoupon lastCoupon = user.getLastOccupiedCoupon();

    assertEquals("쿠폰 할인율", 7, lastCoupon.getDiscountPercent());
    assertEquals("쿠폰 이름", "VIP 고객 한가위 감사쿠폰",
                userCoupon.getName());
}
```

스텁은 특정 객체가 상태를 대신해주고 있긴 하지만, 거의 하드코딩된 형태이기 때문에 로직이 들어가는 부분은 테스트할 수 없다. 이를테면 특정 쿠폰이 구매 제품에 적용되는지 여부에 따라, 결재액이 바뀌는 걸 테스트한다고 생각해보자. 결제 예상금액 조회 같은 서비스라고 생각해도 무방하다.

```
@Test
public void testGetOrderPrice_discountableItem() throws Exception {
    PriceCalculator calculator = new PriceCalculator();
    // new Item(이름, 카테고리, 가격)
    Item item = new Item("LightSavor","부엌칼",100000);
    ICoupon coupon = new StubCoupon();

    assertEquals("쿠폰으로 인해 할인된 가격", 93000,
                calculator.getOrderPrice(item, coupon));
}
```

위와 같은 테스트 케이스를 만들어놓은 다음 PriceCalculator의 getOrderPrice(Item item, ICoupon coupon) 메소드를 구현하게 될 것이다. 그런데 쿠폰이 적용된 할인 가격을 구하려면 쿠폰 객체의 다음과 같은 메소드가 필요하다.

할인된 아이템 가격을 구하기 위해 필요한 쿠폰 인터페이스의 메소드들

isValid	쿠폰 유효 여부 확인
getDiscountRate	할인율
isAppliable	특정 아이템에 적용 가능 여부

이를 기반으로 getOrderPrice를 구현해보자.

```
package main.ishop;

public class PriceCalculator {
    public int getOrderPrice(Item item, ICoupon coupon) {
        // 쿠폰이 유효하고 적용 가능하면
        if(coupon.isValid() && coupon.isAppliable(item)) {
            return (int)(item.getPrice()*getDiscountRate(
                    coupon.getDiscountPercent()) ); // 쿠폰의 할인율을 적용한다
        }
        return item.getPrice();
    }

    private double getDiscountRate(int percent){
        return (100 - percent) / 100d;    // int 연산이 안 되도록 d를 붙임
    }
}
```

뭐, 나쁘지 않다. 그럼 이번엔 할인 적용이 안 되는 아이템에 대한 테스트 케이스를 작성해보자.

```
@Test
public void testGetOrderPrice_undiscountableItem() throws Exception {
    PriceCalculator calculator = new PriceCalculator();
```

```
    Item item = new Item("R2D2", "알람시계", 20000);
    ICoupon coupon = new StubCoupon();

    assertEquals("쿠폰 적용 안 되는 아이템", 20000, calculator.
                getOrderPrice(item, coupon));
}
```

안타깝지만 이번엔 테스트가 실패한다.

```
Java.lang.AssertionError: 쿠폰 적용 안 되는 아이템 expected:<20000> but
was:<18600>
    at org.junit.Assert.fail(Assert.Java:91)
…
…
```

getOrderPrice 구현을 잘못해서 발생한 문제인가? 안타깝지만 아니다.

```
if(coupon.isValid() && coupon.isAppliable(item)) {
    return (int)(item.getPrice()*getDiscountRate(
        coupon.getDiscountPercent()));
}
return item.getPrice();
```

이는 현재 테스트에 사용한 쿠폰이 **스텁**으로 만들어져 있고 아래와 같이 하드코딩되어 있기 때문에 발생하는 문제다.

```
public class StubCoupon implements ICoupon {
    …
    @Override
    public boolean isAppliable(Item item) {
        return true;
    }
…
```

이걸 false로 바꾸면 할인 적용이 안 되는 두 번째 테스트 케이스는 성공하지만, 첫 번째 케이스는 실패한다. 이럴 때 취하는 방법이 두 가지가 있다.

- isAppliable이 항상 false가 되는 StubCouponTwo를 만들어서 두 번째 케이스에서 이용한다.
- isAppliable에 대입되는 item에 따라 true 혹은 false가 나오도록 만든다.

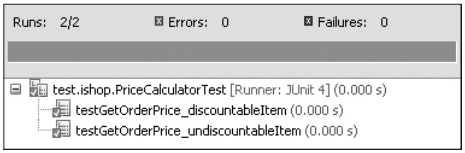
우선 첫 번째는, 잘못된 방식이다. 왜냐하면 우리가 테스트하려는 건 '아이템에 할인이 적용되는 쿠폰/적용 안 되는 쿠폰'이 아니라 '쿠폰의 적용 여부에 따른 예상 가격'인데, 첫 번째 방식은 전자가 되어버리기 때문이다. 따라서 두 번째 방법을 택할 수밖에 없다. 그런데 실제로 아이템 적용 가능 여부의 판단은, DB로부터 쿼리를 통해 조회해서 판단하게 되어 있는 상황이라면 어떻게 해야 할까? 이제와서 DB를 구성해서 DAO[7]부터 개발하고 앉아 있을 수도 없고, 더군다나 ICoupon 인터페이스가 다른 부분에서 어떻게 쓰일지도 모르는 상황에서 섣불리 우리가 집중하고 있는 작업 외의 일을 건드릴 수 없다. 우린 단지 테스트 케이스를 통해 getOrderPrice만 구현하면 된다. 어쩔 수 없다. 조금 사기 같아 보일 수 있지만, StubCoupon의 isAppliable 메소드를 다음과 같이 고쳐보자.

```
@Override
public boolean isAppliable(Item item) {
    if(item.getCategory().equals("부엌칼") ) {
        return true;
    } else if (item.getCategory().equals("알람시계")) {
        return false;
    }
    return false;
}
```

부엌칼이면 쿠폰이 적용 가능하다고 판단하고, 알람시계이면 쿠폰 적용 불가로 판단한다. 하지만, 사실 이런 식으로 쿠폰 클래스를 작성하고 보면 조금 심란하다. 이걸 구현이라고 해야 하나? 로직이긴 로직인데, '그때그때 달라요' 식으로 하드코딩되어 있는 로직이다.

7 DAO(Data Access Object): 데이터를 다루는 역할을 책임지는 객체. DAO를 사용하는 입장에서는 파일로 저장되는지 DB로 저장되는지 구분하지 않고, DAO 쪽에 저장/갱신/자료요청 등의 기능만 호출하면 된다. 층을 나누는 아키텍처(layered architecture)에서 흔히 사용된다.

어찌 됐든, 앞선 두 개의 테스트 케이스를 다시 실행해보자. 테스트를 모두 통과할 것이다. **테스트 더블**로 만들어진 객체가 이 정도까지 지원하는 수준으로 발전하면 단순 **스텁** 수준은 벗어났다고 본다. 마치, 실제 로직이 구현된 것처럼 보이는데, 그렇게 만들어진 객체를 **페이크 객체**라고 부른다.



페이크 객체를 이용한 테스트 케이스 작성과 통과

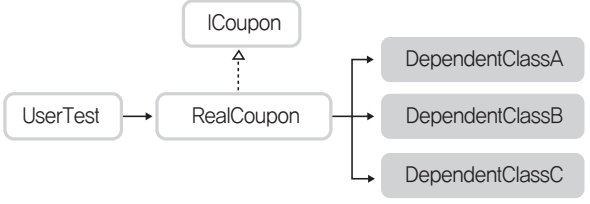
페이크 객체(Fake Object)

모양만으로는 **더미**와 **스텁**의 경계가 모호할 수 있던 것처럼, **스텁**과 **페이크**의 경계도 사실 딱 구분 짓기는 어렵다. 다만 **스텁**은 앞서 이야기했던 것처럼, 하나의 인스턴스를 대표하는 데 주로 쓰이고, **페이크는 여러 개의 인스턴스를 대표할 수 있는 경우이거나, 좀 더 복잡한 구현이 들어가 있는 객체를 지칭한다.** 이를테면 실제로는 DB를 통해 쿠폰 적용 가능 카테고리나 아이템을 확인한다고 하면, **페이크 객체**에서는 테스트에 사용할 아이템과 카테고리에 대해서만 실제로 DB에 접속해서 비교할 때와 동일한 모습처럼 보이게 만들 수도 있다. 보통 내부에 리스트(List)나 맵(Map)을 이용해서 DB 같은 외부 의존 환경을 대체한다. 다음은 DB 대신 목록을 리스트로 관리하도록 만든 코드다.

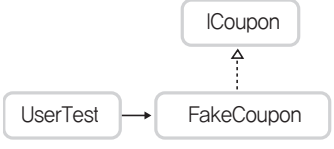
```
public class FakeCoupon implements ICoupon {

    List<String> categoryList = new ArrayList();  // 내부용으로 사용할 목록

    public FakeCoupon (){
        categoryList.add("부엌칼");
        categoryList.add("아동 장난감");
        categoryList.add("조리기구");
    }

    @Override
```

```
    public boolean isAppliable(Item item) {
        if( this.categoryList.contains( item.getCategory() )) {
            return true;
        }
        return false;
    }
…
…
```

페이크 객체는 복잡한 로직이나, 객체 내부에서 필요로 하는 다른 외부 객체들의 동작을, 비교적 단순화하여 구현한 객체다. 결과적으로는 테스트 케이스 작성을 진행하기 위해 필요한 다른 객체(혹은 클래스)들과의 의존성을 제거하기 위해 사용된다.



UserTest를 작성할 때 실제 RealCoupon이 제공됐을 때의 동작 방식



RealCoupon 대신 만들어놓은 FackCoupon

위 그림에서 보는 것처럼, FakeCoupon에서 의존성 객체를 이용해 동작하는 방식을 흉내 내어 구현해놓는다면, 사용하는 쪽인 UserTest에서는 차이를 느끼지 못한다. 보통 **더미**보다는 **테스트 스텁**이 복잡하고, **테스트 스텁**보다는 **페이크 객체**가 더 복잡하다. 그래서 **페이크 객체**를 만들 때 지나치게 집중력을 발휘해서 만들었다간, **페이크 객체** 자체를 다시 테스트해야 할 정도로 복잡해질 수도 있다. 따라서 **페이크 객체**를 만들 때는, 적

절한 수준에서 구현을 접고, 뒤에서 소개될 Mock 프레임워크 등을 사용하든가, 아니면 실제 객체를 직접 가져와서 테스트 케이스 작성에 사용할 것을 권장한다.

이제 **테스트 더블**의 종류에는 **테스트 스파이**와 **Mock 객체**가 남았다. 먼저 **테스트 스파이**부터 살펴보자.

테스트 스파이(Test Spy)

보통 일반적으로 몰래 무언가를 조사해서 정보를 넘기는 일을 하는 사람을 스파이라고 부른다.

아는 사람만 안다는, 스파이 대 스파이(Spy vs Spy)[8]

마찬가지로 테스트에 사용되는 객체에 대해서도, 특정 객체가 사용됐는지, 그리고 그 객체의 예상된 메소드가 정상적으로 호출됐는지를 확인해야 하는 상황이 발생한다. 보통은 호출 여부를 몰래 감시해서 기록했다가, 나중에 요청이 들어오면 해당 기록 정보를 전달해준다. 그런 목적으로 만들어진 **테스트 더블**을 **테스트 스파이**라고 부른다. 특정 메소드의 정상호출 여부 확인을 목적으로 구현되며, **더미**부터 시작해서 **페이크 객체**에 이르기까지 **테스트 더블**로 구현된 객체 전 범위에 걸쳐 해당 기능을 추가할 수 있다. 보통 스파이들이 다른 일도 하면서 스파이 일을 겸업으로 하듯이, **테스트 스파이** 객체도 다른 동작을 하면서 **스파이** 기능까지 하는 경우가 많다.

8 80년대 유명했던 게임 중 하나. 두 스파이가 비밀 아이템을 찾기 위해 서로 경쟁하는 내용의 게임

```
@Test
public void testGetOrderPrice_discountableItem() throws Exception {
    PriceCalculator calculator = new PriceCalculator();
    Item item = new Item("LightSavor","부엌칼",100000);
    ICoupon coupon = new FakeCoupon();

    assertEquals("쿠폰으로 인해 할인된 가격", 93000, calculator.
                getOrderPrice(item, coupon));
}
```

앞에서 본 예제인데, assertEquals에서 calculator.getOrderPrice(item, coupon)이 호출되면 결과값이 93000이 나와야 assert 문장이 참이 된다. 그런데 단순히 참이 되는 것뿐 아니라, getOrderPrice를 구하기 위해서는 내부적으로 ICoupon 인터페이스에서 정의된 isAppliable(Item item) 메소드가 1번 호출되어서 계산에 사용돼야 한다는 가정이 있을 수 있다. 그럼 호출 횟수도 확인하고 싶다면 어떻게 해야 할까? 어렵지 않다. 스파이들이 하듯이 몰래 정보를 수집하고, 어딘가 넘겨줄 수 있는 뒷문(?)을 만들어놓으면 된다.

```
public class SpyCoupon implements ICoupon {

    List<String> categoryList = new ArrayList();
    private int isAppliableCallCount;

    @Override
    public boolean isAppliable(Item item) {
        isAppliableCallCount++;  // 호출되면 증가
        if(this.categoryList.contains(item.getCategory())) {
            return true;
        }
        return false;
    }

    public int getIsAppliableCallCount(){  // 몇 번 호출됐나?
        return this.isAppliableCallCount;
    }
…
```

앞에서 **스텁**을 **페이크**로 만들었던 코드에 **테스트 스파이** 부분을 추가한 다음 클래스 이름을 SpyCoupon으로 변경한 모습이다. isAppliable 메소드 호출 횟수를 저장할 내부 변수를 하나 잡아놓고, 그 값을 이용한다. "어, 그런데 getIsAppliableCallCount() 같은 메소드는 원래 스펙에 없던 메소드인데, 마음대로 추가해도 괜찮나요?"라고 물을 수도 있다. 어차피 가짜 객체이고, 테스트를 위해 사용하는 것뿐이니까 크게 상관없다. 예제에서는 호출 횟수 기록 같은 단순한 정보 수집만을 예로 들었지만, **테스트 스파이**의 사용법은 매우 다양하다. 이를테면 호출될 때마다 깊은 복사(deep copy)나 복제(cloning)하는 식으로 특정 객체를 컬렉션 객체에 넣도록 만든다면 어떨까? 그렇게 했다면, 호출 시점별로 객체의 상태를 담고 있는 일종의 스냅샷 같은 기능을 구현한 셈이다. 호출 시점별 좀 더 면밀한 감시가 필요한 경우 유용하게 사용할 수 있을 것이다. 이 외에도 **테스트 스파이**에서는 감시 대상이 되는 건 무엇이든 상관없이 기록한다.

자, 다시 테스트 케이스로 돌아가 보자. **스파이** 기능을 넣은 결과, 테스트 케이스는 다음과 같은 식으로 테스트할 수 있다.

```
@Test
public void testGetOrderPrice_discountableItem() throws Exception {
    PriceCalculator calculator = new PriceCalculator();
    Item item = new Item("LightSavor","Kitchen knife",100000);
    ICoupon coupon = new SpyCoupon();

    assertEquals("쿠폰으로 인해 할인된 가격", 93000, calculator.
                getOrderPrice(item, coupon));

    int methodCallCount = ((SpyCoupon)coupon).getIsAppliableCallCount();
    assertEquals("coupon.isAppliable 메소드 호출 횟수", 1, methodCallCount);
}
```

예상값의 비교뿐 아니라 호출 횟수를 따져보는 추가적인 체크도 가능해졌다. 예제가 다소 억지스러운 면이 없잖아 있긴 하지만, **테스트 스파이**가 무얼 의미하는지는 잘 알았으리라 본다.

일반적으로 **테스트 스파이**는 아주 특수한 경우를 제외하고 잘 쓰이지 않는다. 보통은 **테스트 스파이**가 필요한 경우에도 Mock 프레임워크를 이용하는 것이 더 편하기 때문이다. 대부분의 Mock 프레임워크들은 기본적으로 **테스트 스파이** 기능을 제공해준다. 잠시 뒤에 보게 될 Easymock이나 jMock, Mockito 등도, 모두 **스파이** 기능은 기본으로 제공한다.

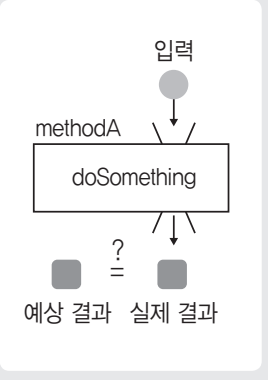
자, 이제 메스자로스의 분류 중 **Mock 객체** 하나 남았다. 그런데 **Mock 객체**를 이해하려면 상태 기반 테스트와 행위 기반 테스트에 대한 이해가 필요하다. 필자가 지금까지 미뤄왔던 숙제를 하나 할 시간이다. 이어지는 내용을 읽어보자.

저자 한마디

상태 기반 테스트 vs 행위 기반 테스트

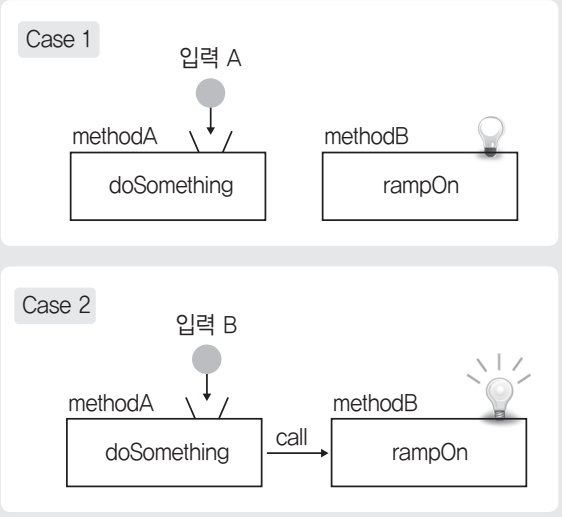
상태 기반 테스트(state base test)

객체 지향 프로그램의 특징 중 하나는 객체가 특정 시점에 자신만의 상태를 갖는다는 점이다. 상태 기반 테스트는 이런 특징에 기반한 테스트 방식이다. 동작하는 모양만으로 봤을 때, 상태 기반 테스트는 테스트 대상 클래스의 메소드를 호출하고, 그 결과값과 예상값을 비교하는 식이다. 물론, 메소드는 '두 값의 합 구하기' 같은 식의 '기능'으로만 동작할 수도 있다. 하지만 많은 경우에 있어 메소드 호출은 객체의 상태를 변경한다. setName("조연희") 같은 메소드만 보더라도, 객체의 이름 속성 값을 바꿔버린다. 특정한 메소드를 거친 후, 객체의 상태에 대해 예상값과 비교하는 방식이 상태 기반 테스트이다. setName 메소드를 호출했으면, getName() 메소드로 확인해보는 식이다.



행위 기반 테스트(behavior base test)

행위 기반 테스트는 올바른 로직 수행에 대한 판단의 근거로 특정한 동작의 수행 여부를 이용한다. 보통은 메소드의 리턴값이 없거나 리턴값을 확인하는 것만으로는 예상대로 동작했음을 보증하기 어려운 경우에 사용한다. 다음 그림은 methodA와 methodB의 관계를 보여준다. methodA에 특정 값이 들어왔을 때 발생하는 호출관계를 살펴보자.



methodA에 A가 입력되면 methodB는 호출되지 않아야 정상이다. 그리고 B가 입력되면 methodB가 호출돼야 정상이다. 하지만 테스트 대상이 되는 methodA만 놓고 봤을 때는 입력 값에 따른 차이를 methodA만으로는 알아낼 수가 없다. 즉 methodB의 호출 여부를 조사하지 않으면, methodA의 정상 동작 여부를 판단할 수 없다. 만일 methodA가 정상 동작했을 경우 methodB가 반드시 호출되는 구성이라면, 반대로 methodB의 호출 여부로 methodA의 정상 여부를 판단할 수 있다고 보는 것이다. 따라서 이럴 때는 methodB의 호출 여부를 확인하는 것이 테스트 시나리오의 종료조건이 된다. 하지만 전통적인 테스트 케이스 작성 방식, 즉 상태 기반 테스트에선, 사실상 이런 상황에 대한 테스트 케이스를 작성하기가 매우 어렵거나 불편했다. 테스트 대상인 A가 상태를 갖고 있지 않기 때문이다. (void 메소드를 생각해보라!)

이럴 때 찾아낸 방법이, 테스트 스파이 객체를 사용하거나 자체적으로 검증 기능을 제공하는 Mock 객체를 따로 만들어서 테스트 케이스를 작성하는 것이었다. 즉, 행위를 점검하는 걸로 테스트 케이스를 만드는 방식인 것이다. 이런 방식을 행위 기반 테스트라고 부른다. 따라서 행위 기반 테스트를 수행할 때는 예상하는 행위들을 미리 시나리오로 만들어놓고 해당 시나리오대로 동작이 발생했는지 여부를 확인하는 것이 핵심이 된다. 참고로, 초창기에 나온 Mock 프레임워크들은 태생 자체가 이런 행위 기반 테스트를 지원해주기 위해서인 경우가 대다수였다.

Mock 객체(Mock Object)

> 일반적인 테스트 더블은 상태(state)를 기반으로 테스트 케이스를 작성한다.
> Mock 객체는 행위(behavior)를 기반으로 테스트 케이스를 작성한다.

Mock 객체는 행위를 검증하기 위해 사용되는 객체를 지칭한다. 수동으로 만들 수도 있고, Mock 프레임워크를 이용할 수도 있다. 수동으로 만드는 건 고통이 매우 크기 때문에, 현재는 대부분 Mock 프레임워크를 사용한다. Mock 프레임워크에 대해서는 다음 장부터 자세히 살펴볼 예정이고, **Mock 객체**에 대해선 설명하지 않고 넘어갈 예정이다. 대신에, 뒤에서 보게 될 내용이지만, Mock 프레임워크를 이용해 행위를 테스트하는 소스를 미리 살짝 살펴보자.

앞의 쿠폰 예제에서 이어지는 내용인데, 이번엔 지켜야 하는 업무 규칙 하나가 추가로 존재한다고 가정하자. '유저가 쿠폰을 받을 때, 쿠폰이 유효한 상태일 경우에만 다운받을 수 있다'는 규칙이다. 이 규칙을 따라서 user.addCoupon(coupon) 메소드가 정상 작성됐다면, addCoupon 호출 시에 반드시 coupon.isValid()도 함께 호출되어, 쿠폰의 유효 여부를 확인하는 과정을 거쳐야 할 것이다. 그러고 나서 유저가 해당 쿠폰을 취득할 수 있게 해야 한다. 다음은 일반적인 상태 기반 테스트 케이스다.

```
ICoupon coupon = new FakeCoupon();

user.addCoupon(coupon);
assertEquals(1, user.getTotalCouponCount());
```

상태(1이라는 값)가 정상적으로 보이는지 확인한다. 업무 규칙을 테스트하는 부분이 addCoupon 내에 들어 있었는지, 정상적으로 호출되어 판단에 사용됐는지 확인할 방법이 없다. 그나마 getTotalCouponCount 메소드가 있어서 부분적으로나마 확인이 가능했다. 하지만 getTotalCouponCount 메소드가 존재하지 않는다면 어떻게 할 것인가?

다음은 jMock이라는 Mock 라이브러리를 사용한 예제다. 약간 옛날 문법이므로, 문법 자체엔 너무 집중하지 말고 그냥 뉘앙스를 느끼길 바란다.

```
ICoupon coupon = mock(ICoupon.class);
coupon.expects(once()).method("isValid")    // ❶
    .withAnyArguments()      // ❷
    .will(returnValue(true));    // ❸
user.addCoupon(coupon);
assertEquals(1, user.getTotalCouponCount());
```

❶ Mock으로 만들어진 coupon 객체의 isValid 메소드가 한 번 호출될 것을 예상함
❷ isValid에서 사용할 인자는 무엇이 됐든 상관 안 함
❸ 호출 시에 리턴값은 true를 돌려주게 될 것임

위 예제는, 호출 횟수 및 파라미터 값 지정 여부, 호출 시 반환할 리턴값까지 지정해서 예상대로 동작하는지를 테스트한다. 예상과 달리 coupon.isValid() 메소드가 여러 번 호출된다든가, 한 번도 호출되지 않는다든가, 아니면 예상했던 파라미터와 다른 값으로 호출된다면, 테스트가 실패하게 된다. 현재 isValid()는 메소드 호출 시 필요한 파라미터는 따로 없지만 말이다.

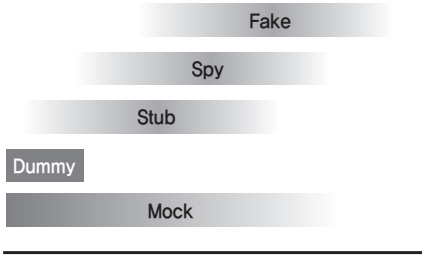
나중에도 다시 언급하겠지만, 행위 기반 테스트는 복잡미묘한 시나리오가 사용되는 경우가 많고, 모양이나 작성 등 여러 가지 측면에서 어색한 경우가 많기 때문에, 만일 상태 기반으로 테스트를 할 수 있는 상황이라면 굳이 행위 기반 테스트 케이스는 만들지 않는 것이 좋다.

제라드의 **테스트 더블** 분류에 대해서는 이 정도로 살펴보고 마치기로 한다.

Mock 객체라는 용어가 주는 혼란스러움 정리

이제 Mock **객체**에 대한 설명을 마치고 다음으로 넘어갈 예정인데, Mock **객체**를 설명하면서 조심스러웠던 부분에 대한 이야기를 조금 할까 한다. 사실 'Mock **객체**'라는 단어는 방금 설명한 제라드의 분류법에 근거한 '행위 기반 테스트를 위해 사용되는 객체'

의 의미보다는 더 넓은 일반적인 '가상 임시 구현체'의 의미로 사용되는 경우가 더 많다. 일상에서는 지금까지 본 **테스트 더블**이란 단어와 **Mock 객체**라는 단어가 거의 동등한 의미로 사용되는 경우가 더 많았다. 따라서 그 둘을 제대로 구분해서 표현할 필요가 있다. 이 책에서는 앞으로 **Mock 객체**라는 단어를, 행위 기반 테스트를 지원하는 객체에 한정된 것이 아닌, **테스트 더블**과 동일한 의미로 사용할 예정이다. 그리고 따로 고딕체로 구분지어 Mock 객체를 표시했던 것도 더 이상 하지 않을 것이다. 왜냐하면 Mock 객체를 만든다는 뜻의 Mocking이란 단어는, 일반적으로 지금 소개한 행위 기반 테스트 객체를 의미하는 것이 아닌, **테스트 더블** 객체를 만드는 것을 의미하기 때문이다. 또한 Mock 객체를 만들어주는 Mock 프레임워크 자체도 사용 영역이 굉장히 넓어서, **테스트 더블** 전체 분류에 걸쳐 여러 가지 의미로 사용된다. 무슨 말인가 하면, Mock 프레임워크를 이용해 아주 간단히 클래스만 하나 인스턴스화하면 **더미 객체**와 동일하다. 그리고 추가해서 특정 리턴값을 돌려주도록 만들면 **스텁**, 호출 확인(verify) 기능을 사용하면 **테스트 스파이** 등으로 불리게 된다.



테스트 더블의 분포 범위. 출처: 테스트 더블의 연속성 살펴보기
http://msdn.microsoft.com/ko-kr/magazine/cc163358.aspx

저자 한마디

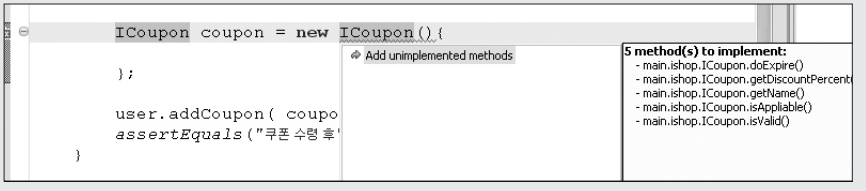
테스트 더블을 익명 클래스로 만들기

더미 객체에서 페이크 객체에 이르기까지, 앞 예제들에서는 ICoupon 인터페이스를 구현하는 클래스를 명시적으로 만들어서 사용했다. 그렇게 만들어 사용하는 데 별 문제는 없지만, 위치가 어디가 됐든 클래스의 개수가 늘어나는 건 사실이고, 그건 별로 달갑진 않은 게 또 사실이다. 그래서, 만일 가능하고 적절하다면, 굳이 명시적인 클래스로 만들기보다 익명 클래스(anonymous

class)로 만들어서 테스트 케이스 작성 시에만 사용하는 것도 괜찮다. 예를 들어 맨 처음에 작성한 더미 객체 구현의 경우, 테스트 케이스에서 다음처럼 익명 클래스로 만들어 진행할 수 있다.

```
@Test
public void testAddCoupon() throws Exception {
     User user = new User("area88");
     assertEquals("최초 쿠폰 수", 0, user.getTotalCouponCount());

     ICoupon coupon = new ICoupon(){

     };

     user.addCoupon(coupon);
     assertEquals("쿠폰 수령 후", 1, user.getTotalCouponCount());
}
```

우선 인터페이스를 마치 클래스인 것처럼 new 키워드를 붙인 다음 괄호({ })를 붙인다. 닫는 괄호(}) 뒤에 세미콜론(;)을 반드시 붙여줘야 한다는 점에 유의하자. 아마 위와 같이 작성하고 저장 버튼을 누르면, 이클립스에서 오류 전구에 불이 들어올 텐데, 전구를 클릭하거나 Quick Fix에 해당하는 Ctrl+1을 눌러서 Add unimplemented methods를 선택하자.



선택 결과는 다음과 같다.

```
@Test
public void testAddCoupon() throws Exception {
     User user = new User("area88");
     assertEquals("최초 쿠폰 수", 0, user.getTotalCouponCount());

     ICoupon coupon = new ICoupon(){
          @Override
          public void doExpire() {
```

```
        }

        @Override
        public int getDiscountPercent() {
            return 0;
        }

        @Override
        public String getName() {
            return null;
        }

        @Override
        public boolean isAppliable(Item item) {
            return false;
        }

        @Override
        public boolean isValid() {
            return false;
        }
    };

    user.addCoupon( coupon );
    assertEquals("쿠폰 수령 후", 1, user.getTotalCouponCount());
}
```

이제 익명 클래스 안에 필요한 만큼 내용을 채워넣으면 더미든 스텁이든 페이크든 스파이든, 뭐든 만들 수 있다. 그리고 필요하다면 coupon을 필드 변수로 뽑아내고, 익명 클래스로 인터페이스를 구현하고 있는 부분을 setUp 메소드로 이동시키면 좀 더 깔끔해질 것이다.

중급이상 도전!

4.3 Mock 프레임워크

Mock 프레임워크(혹은 라이브러리)는 동적으로 Mock 객체를 만들어주는 프레임워크를 지칭한다. Mock 프레임워크 사용 시의 이점은 크게 두 가지다.

- Mock 객체를 직접 작성해서 명시적인 클래스로 만들지 않아도 된다.
- Mock 객체에 대해서 행위까지도 테스트 케이스에 포함시킬 수 있다.

IDE를 사용하면 더미나 스텁 같은 단순한 Mock 객체를 만드는 작업 그 자체는 별 부담이 되지 않는다. 그렇기 때문에 Mock 프레임워크를 사용하는 것이, 오히려 학습의 부담을 줄 뿐 큰 장점이 없는 것처럼 느껴질 수도 있다. 하지만 Mock 객체를 직접 만들다 보면 클래스 숫자가 점점 늘어나서 관리에 부담이 생긴다. 결국 언제가 됐든 Mock 프레임워크가 필요한 시점이 올 수밖에 없다. 현재 세계적으로 놓고 봤을 때, Mock 프레임워크 랭킹을 뽑으라면, EasyMock과 jMock이 1, 2등을 다툰다. 특히 EasyMock의 경우 만들어진 지가 오래됐고, 많은 프레임워크에서 EasyMock을 기본으로 지원하고 있다. 스프링 프레임워크도 테스트 케이스를 만드는 데 EasyMock을 사용했었다. 그리고 하나 더 꼽자면, 비록 역사는 오래되지 않았지만 간편함으로 인해 다크호스로 떠오른 Mockito(목키토)를 꼽을 수 있다. Mockito는 간편함과 함께 기존 Mock 프레임워크들이 지향했던, 행위 기반 테스트 위주에서 상태 위주 테스트로의 회귀를 전면으로 내세우고 있다. 그리고 이 점이 Mockito의 흥행 요소 중 하나이기도 하다. 이제부터 이 세 개의 Mock 프레임워크를 각각 간략히 살펴볼 예정이다. 단, EasyMock과 jMock의 경우 나중에 해당 Mock 프레임워크로 만들어진 소스를 읽을 수 있는 수준 정도로만 설명할 예정이다. 중요한 점은, 어떤 모의 객체 프레임워크(Mock Object Framework)[9]를 사용하느냐가 아니라, 무엇을 모의 객체로 만들 것인가임을 잊지 말자. 참고로, 만일 시간이 없어서 Mock 프레임워크를 하나만 보겠다면, 마지막에 설명하게 될 Mockito를 권장한다. (필자가 보기엔) 셋 중 가장 발전된 형태의 Mock 프레임워크다. 물론 세 개

9 Mock 프레임워크를 이렇게 부르기도 한다. 참고하자.

의 Mock 프레임워크를 차분히 하나씩 비교하면서 각각의 특징과 스타일, 차이점을 함께 살펴보는 것이 제일 좋다는 건 두말하면 잔소리다.

저자 한마디

고전주의 TDD 개발자 vs Mock 주도개발자

Mock 프레임워크는 클래스를 명시적으로 만들지 않아도 된다는 장점에다, 행위 기반 테스트 같은 곤란한 상황에 대한 테스트도 지원해줬기에 많은 개발자의 눈길을 끌었다. 그리고 전통적인 상태 기반 테스트 방식을 사용하는 사람들을 고전주의 TDD 개발자(Classicist)라고 불렀고, 이런 Mock 프레임워크를 열심히 사용하는 개발자들을 Mock 주도개발자(Mockist)라고 부르기 시작했다. 초기 대부분의 Mock 프레임워크들은 행위 기반 테스트 지원에 강점을 가졌기 때문에 Mock 프레임워크를 이용한다는 건 곧 행위 기반 테스트를 즐겨한다는 것과 동일시됐고, Mock 주도개발자도 마찬가지로 행위 기반 테스트를 즐겨하는 개발자로 간주됐다. 이로 인해, 본의 아니게 묘하게 상태 기반 테스트 지향자와 행위 기반 테스트 지향자로 나뉘는 형국이 됐다. 그리고 응당 모든 일이 그러하듯, 선 긋고 넘어간 사람과 남아 있는 사람들은, 서로 자신이 맞다면서 툭탁대는 당연한 수순의 시기를 겪는다. 지리한 논쟁에 대한 마틴 파울러의 의견은 'Mocks aren't Stubs(Mock은 스텁이 아니다)'라는 글에서 볼 수 있다. 생각보다 많은 사람이, 자신이 존경하는 과학자 마틴 파울러의 대답을 듣고 싶어했다. "어느 쪽이 TDD에 더 적절한가요?" 하지만 마틴 파울러는 자신은 전통적인 TDD 개발자이지만, Mock 주도개발자의 테크닉을 쓰지 않으려니 참 힘들다는 식으로 표현하며 어느 쪽을 더 권장하는지에 대해서는 교묘하게도 대답을 회피했다. 결국 둘 중 어느 것을 사용할지는 전적으로 개발자에게 달려 있다. 개인적으론, 상태 기반 테스트가 좀 더 쉽고 깔끔하다고 생각한다.

참고

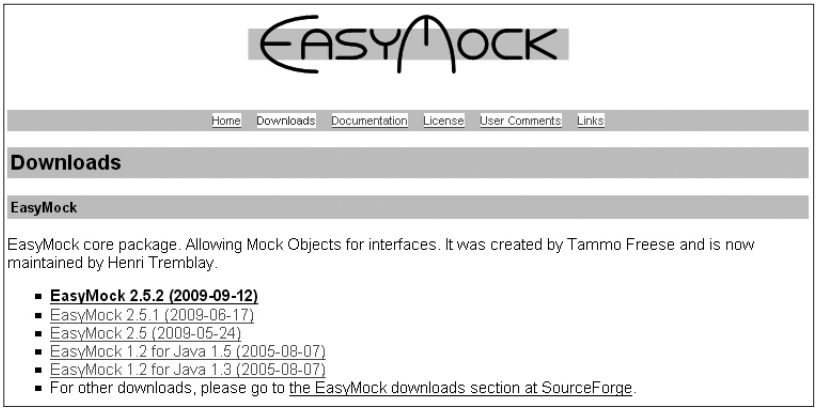
Mocks aren't Stubs(http://www.martinfowler.com/articles/mocksArentStubs.html)

4.3.1 EasyMock

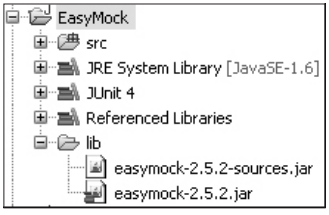
가장 오래된 Mock 프레임워크로 현재 2.5 버전까지 나와 있다. 탐모 프리스(Tammo Freese)라는 독일인이 만들었고, 현재는 앙리 트랑블레(Henri Tremblay)라는 프랑스인 개발자가 유지보수를 이어나가고 있다.[10] EasyMock은 현재 오픈소스로 공개되어 있는 프로젝트이며, EasyMock 2 버전부터는 JDK 1.5 이상에서 정상 동작한다.

10 에릭 레이먼드(Eric S. Raymond)의 '성당과 시장(The Cathedral & Bazzar)'이란 글 중에 이런 문장이 나온다. "프로그램에 흥미를 잃었다면 프로그램에 대한 당신의 마지막 의무는 능력 있는 후임자에게 프로그램을 넘겨주는 것이다." EasyMock도 그런 사례 중 하나로 판단된다.

환경 구성



EasyMock 공식 사이트(easymock.org)에서 최신 파일을 내려받는다. 참고로 내려받는 파일 항목 중에 EasyMock Class Extension이라는 파일이 있는데, 이건 구현부가 들어 있는 클래스를 Mock 객체로 만들 수 있게 해주는 확장 라이브러리다. 일반적인 Mock 프레임워크는 인터페이스를 기반으로 Mock 객체를 만들도록 되어 있다. 몇몇 Mock 프레임워크가 인터페이스뿐만 아니라 구현 클래스를 Mock 객체로 만드는 기능을 제공했는데, 생각보다 편리해 개발자 사이에서 좋은 반응을 얻자 다른 프레임워크에서도 해당 기능을 제공하고 있는 추세다.



라이브러리를 내려받아서 클래스패스에 포함시킨 모습

lib 폴더에 내려받은 다음, easymock-2.5.2.jar 파일을 클래스패스로 지정했다.

기본 사용법

EasyMock은 Record & Replay라는 메타포(metaphor)[11]를 사용하며, 기본적으로는 다음의 네 단계로 동작한다. 각 단계는 경우에 따라 생략될 수 있다.

CreateMock	인터페이스에 해당하는 Mock 객체를 만든다.
Record	Mock 객체 메소드의 예상되는 동작을 녹화한다.
Replay	예정된 상태로 재생한다.
Verify	예상했던 행위가 발생했는지 검증한다.

01 Mock 객체 만들기

```
createMock(타깃_인터페이스);
```

예

```
import static org.easymock.EasyMock.*;

...
List mockList = createMock(List.class);
```

02 예상동작 녹화 및 재생

녹화

```
expect(Mock_객체의_메소드).andReturn(예상결과값);
```

재생

```
replay(Mock_객체);
```

11 원뜻은 '은유적인 표현'인데, 소프트웨어에서는 시스템의 상징적인 개념을 종종 메타포라고 부른다.

예

```
expect(mockList.add("item")).andReturn(true);
expect(mockList.size()).andReturn(0).times(2);
mockList.clear();
replay(mockList);
```

다소 추상적인 개념으로 인해 많은 이들에게 EasyMock 사용을 좌절시킨 녹화 및 재생의 개념이다. Mock 객체가 만들어진 이후부터 재생(replay)이 호출되기 전까지, 동작이 기록된다. 행위 기록이기 때문에 횟수를 지정할 수도 있다. void 메소드일 경우엔 굳이 andReturn을 지정해주지 않아도 된다.

03 검증

```
verify(Mock_객체);
```

예

```
@Test
public void listMockTest() throws Exception {
    mockList = createMock(List.class);
    expect(mockList.add("item")).andReturn(true);
    expect(mockList.size()).andReturn(0).times(2);
    mockList.clear();
    replay(mockList);
    verify(mockList);
}

-----실행 결과-----

Java.lang.AssertionError:
    Expectation failure on verify:
        add("item"): expected: 1, actual: 0
        size(): expected: 2, actual: 0
        clear(): expected: 1, actual: 0
```

verify로 검증하는 시점에서 녹화된 분량만큼 실행되지 않았다면 테스트 케이스가 실패한다.

응용 예제

다음은 인터페이스를 Mock 객체로 만드는 EasyMock 프레임워크의 사용 예다. EasyMock 사이트에 있는 예제를 조금 수정해 사용했다. 문서박스(DocumentBox)와 문서변경에 따라 동작하도록 만들어놓은 IListener 인터페이스다.

```
public interface IListener {
    void documentAdded(String title);
    void documentChanged(String title);
    void documentRemoved(String title);
    byte voteForRemoval(String title);
    byte[] voteForRemovals(String[] title);
    int getDocumentSize(String title);
}

public class DocumentBox{
    // ...
    public void addListener(IListener listener) {
        // ...
    }
    public void addDocument(String title, byte[] document) {
        // ...
    }
    public boolean removeDocument(String title) {
        // ...
    }
    public boolean removeDocuments(String[] titles) {
        // ...
    }
}
```

testRemoveNonExistingDocument는 존재하지 않는 문서를 제거하려 할 경우에 대한 테스트 케이스다.

```
public class ExampleTest extends TestCase {
    private DocumentBox documentBox;
    private IListener mockListener;

    protected void setUp() {
        documentBox = new DocumentBox();
        documentBox.addListener(mockListener);
    }

    public void testRemoveNonExistingDocument() {
        documentBox.removeDocument("DocumentNameA");
    }
}
```

documentBox(문서함)에 현재 들어가 있는 문서가 없다. 하지만 테스트 케이스에서는 DocumentNameA라는 문서를 제거하는 removeDocument 메소드를 호출하고 있다. 아마, 호출 시점에 문제가 발행할 것이다. 현재 IListener 타입의 mockListener는 외부에서 인스턴스를 넘겨 받아와야 하는 인터페이스다. 현재, Mock 대상이 되는 인터페이스이기도 하다.

```
public class ExampleTest extends TestCase {

    private DocumentBox documentBox;
    private IListener mockListener;

    protected void setUp() {
        mockListener = createMock(IListener.class); // Mock 객체 생성
        documentBox = new DocumentBox();
        documentBox.addListener(mockListener);
    }

    public void testRemoveNonExistingDocument() {
        replay(mock); // 재생
        documentBox.removeDocument("DocumentNameA");
    }
    …
```

위 소스에서 mock으로 만든 인터페이스에 대해 기대하는 동작(메소드 호출)은 없다. 만일 replay 호출 이후에 documentBox에서 넘겨받은 mockListener를 이용해 IListener의 메소드를 호출한다면, 다음과 같은 에러가 발생한다.

```
Java.lang.AssertionError:
  Unexpected method call documentRemoved("DocumentNameA"):
...
```

이제 mock에 향후 기대하는 행동, 즉 메소드 호출의 모습을 기록해보자.

```
public void testAddDocument() {
    mockListener.documentAdded("New Document"); // 예상동작을 기록한다.
    replay(mockListener); // 재생한다.
    documentBox.addDocument("New Document", new byte[0]);
}
```

위 소스는 mock.documentAdded("New Document");가 호출될 것임을 녹화(record)하는 모습이다. replay 메소드 호출 이후에 mock.documentAdded("New Document")가 인자에 맞게 정상적으로 호출되지 않는다면 에러가 발생한다. 다만, 한 번도 호출되지 않을 경우에는 테스트가 통과하는 걸로 나온다. 따라서 테스트 케이스 마지막에서는 정상적으로 원하는 만큼 호출됐는지 확인할 필요가 있다. 이때 이용하는 것이 verify라는 메소드다.

```
public void testAddDocument() {
    mockListener.documentAdded("New Document"); // 예상동작을 기록한다.
    replay(mockListener); // 재생한다.
    documentBox.addDocument("New Document", new byte[0]);
    verify(mockListener); // 정상적으로 1회 호출됐는지 확인한다.
}
```

만일 정확하게 몇 회 호출됐는지까지 확실히 체크하고 싶다면, expectLastCall()이나 times(int times) 등의 메소드를 사용해 체크할 수가 있다.

```
public void testAddAndChangeDocument() {
    mockListener.documentAdded("Document");
    mockListener.documentChanged("Document");
    expectLastCall().times(3);
    replay(mockListener);
    documentBox.addDocument("Document", new byte[0]);
    documentBox.addDocument("Document", new byte[0]);
    documentBox.addDocument("Document", new byte[0]);
    documentBox.addDocument("Document", new byte[0]);
    verify(mockListener);
}
```

위와 같은 경우, 처음에는 added로 추가되고, 그 다음부터는 changed로 변경이 호출되도록 예상하는 경우다. expectLastCall().times(3);라는 메소드를 이용해 예상대로 호출되는지 확인할 수 있다. 만일 예상과 다를 경우에는 assertion fail 메시지가 나온다.

```
Java.lang.AssertionError:
  Unexpected method call documentChanged("Document"):
    documentChanged("Document"): expected: 3, actual: 4
...
```

Mock 객체를 사용하는 목적 중 하나는 위 경우처럼, 해당 메소드에 대한 호출이 정상적으로 이뤄졌느냐를 확인하기 위해서일 경우도 있지만, 대개는 특정한 결과값을 돌려주길 기대할 때가 많다. 이를테면, customer.getName()을 호출하면 "Suwon"이라는 이름이 반환되는 식이다. 이럴 때 EasyMock에서는 특정한 예상값을 돌려주도록 미리 지정해놓을 수가 있다. 이런 기법을 다이내믹 프록시(dynamic proxy) 기법이라고도 한다. 방식은 아주 간단한다. expect(메소드).andReturn(지정값)을 이용한다. 다음은 이에 대한 예다.

```
public void testVoteForRemoval() {
    mockListener.documentAdded("Document");   // 문서 추가
    expect(mockListener.getDocumentSize("Document")).andReturn(1024);
    mockListener.documentInfo("Document");    // 문서 정보 출력
    replay(mockListener);
```

```
        documentBox.addDocument("Document", new byte[0]);
        assertEquals("Document: 1024", documentBox.documentInfo("Document"));
        verify(mockListener);
    }
```

documentInfo 메소드는 문서 이름과 크기를 돌려주는 메소드다. 이때 documentInfo는 getDocumentSize(문서이름)을 호출해서 해당 문서의 크기를 체크한다. 이미 expect로 사이즈 호출 메소드 getDocumentSize에 대한 반환값을 결정해놓았기 때문에 위 테스트는 정상적으로 통과한다.

EasyMock 정리

EasyMock은 나온 지 오래된 만큼, 많은 사용자를 확보한 Mock 프레임워크이며, 한동안은 대안이 없던 프레임워크였다. 하지만 녹화(record)와 재생(play)이라는 개념은 처음 접하는 사람들에게 적지 않은 혼란을 줬다. 특히 초창기엔 문법도 복잡해서 쓰는 사람만 쓰던 프레임워크였다. 버전 2로 올라서면서 문법이 많이 간결해졌지만, 한편으론 다른 Mock 프레임워크와 큰 차별점이 없는 모습이 됐다. 이건 비단 EasyMock만의 문제는 아니지만 말이다. 만일 새로 Mock 프레임워크를 배우려 한다면, 권장하고 싶은 프레임워크는 아니다. 그래도 기존 소스들 중에 EasyMock으로 테스트 케이스가 작성되어 있는 게 꽤 많기 때문에, 읽을 수 있는 수준 정도로는 배워두자.

4.3.2 jMock

jMock은 스티브 프리먼(Steve Freeman), 냇 프라이스(Nat Pryce)[12]라는 두 명의 영국 개발자가 만든 Mock 프레임워크다. 그 자체로는 아주 오래된 프레임워크는 아니지만, DynaMock이라는 그보다 훨씬 이전에 만들어졌던 Mock 프레임워크의 개발자에 의해 새롭게 만들어진 프레임워크인지라 Mock 작성에 관련된 노하우가 많이 담겨 있

12 스티브 프리먼과 냇 프라이스는 소프트웨어 엔지니어이자 독립적인 애자일 컨설턴트로 영국에서 활동하고 있다. MockObjects.com이라는 사이트를 만들어 Mock의 발전에 많은 공헌을 하고 있다. 둘은 굉장한 절친인데, DynaMock에서 jMock, 그리고 .net 개발자들을 위한 nMock까지 함께 개발했고, 종종 함께 전 세계로 강연을 다닌다.

는 프레임워크다. jMock은 여타 프레임워크와 구별되는 몇 가지 독특한 차이점을 목표로 개발됐는데, 그중 특히 테스트의 표현(expression) 확대와 가독성(readability) 증진에 많은 노력이 들어가 있다. 그중 몇은 여타 프레임워크나 개발 스타일에도 영향을 끼쳤다. 이를테면 연쇄 호출 기법 같은 경우가 그렇다.

jMock의 특징

01 연쇄 호출(call-chain)

jMock의 연쇄 호출은 동일 객체에 여러 개의 메시지를 보낼 때 발생하는 번잡스러움을 해결하기 위해 만들어진 스타일이다. 예를 들면 발권된 열차티켓 예약 변경 작업을 하는 로직이 있다고 가정해보자. 기본적으로 사용자 이름, 출발지, 목적지, 출발시간, 운임요금 차이 갱신, 발권수량 정정 등의 작업이 '예약 변경' 작업을 위해 필요하다. 코드로 만들면 다음과 같다.

```
bookedTicket.setDeparture(locationA);
bookedTicket.setArrival(locationB);
bookedTicket.setDepartureDate(date);
bookedTicket.addAdditionalFare(fare);
```

여기에 일종의 워크플로(workflow, 업무 흐름)를 적용해서 메소드들의 연쇄 호출이 일어날 수 있도록 만들면 다음과 같이 된다.

```
bookedTicket.departure(locationA)
            .arrival(locationB)
            .departureDate(date)
            .additionalFare(fare);
```

언뜻 보면 그저 리턴된 객체를 특정 변수로 지정하지 않고 계속 그대로 사용하고 있는 것 아니냐고 물을 수도 있는데, 보면 알겠지만 원래는 void 메소드들이었다. 그런 메소드를, 사전에 특정 워크플로를 만들어놓고, 해당 순서에 해당하는 객체를 리턴하는 방식으로 변경한다. 이를테면 departure 메소드는 Arrival 클래스를 리턴하

게 만들고, Arrival 클래스는 도착 업무와 관련된 메소드들을 제공한다. 그중 arrival 메소드는 DepartureDate라는 업무모델 클래스를 리턴하도록 만들고, 또 마찬가지로 DepartureDate 클래스는 자신의 업무(work)에 맞는 메소드를 제공하는 방식이다. 물론 마지막 클래스는 최종 내용이 반영된 bookedTicket을 리턴하도록 만들거나 최초의 bookedTicket을 최종 완성된 bookedTicket으로 치환하도록 만든다. 물론 이건 연쇄 호출을 설명하기 위한 대략적인 설명이다. 실제로는 좀 더 정교하게 만든다.

그런데 jMock 2부터는 이런 방식을 많이 탈피했고, 위와 같은 방식으로 사용하진 않는다. 잘 쓰면 편리하지만, 규칙을 세워 사용하지 않으면 오히려 코드 자체가 보기 어렵게 변하기 때문이다. 특히 호출 순서가 있는 메소드라면 좀 더 정교한 설계가 필요하고, 자칫 클래스 숫자가 늘어나기 쉬워진다. 그래서 jMock의 경우도 버전 2에 들어서면서 이 방식을 버리고, 좀 더 간결한 구조로 바뀌었다. 정공법보다 기교적인 측면이 강하므로 초급 개발자에겐 이런 구조의 설계를 권하지 않는다. 그럼에도 연쇄 호출을 소개하는 이유는 이런 방식의 설계를 살펴볼 가치가 있고, 여타 Mock 프레임워크에서도 종종 이용하는 방식이기 때문이다.

jMock 1의 연쇄 호출 스타일

```
mockSubscriber.expects(once()).method("receive").with(eq(message));
```

저자 한마디

연쇄 호출과 메소드 연쇄

이전에 연쇄 호출(call-chain)이나 메소드 연쇄(method chain)라는 단어를 들어봤는지 모르겠다. 두 단어는 어감이 비슷하고, 심지어 구현된 모습도 비슷하지만 안타깝게도 따로 구별되어 불리길 원하는 단어들이다. 보통 연쇄 호출은 워크플로의 개념을 구현할 때 사용하고, 메소드 연쇄는 메소드의 리턴값으로 자기 자신(this)을 넘겨주는 기법을 지칭한다. 이를테면 Java 기본 클래스인 StringBuilder의 append 메소드 같은 경우가 바로 메소드 연쇄가 사용된 대표적인 예다.

```
StringBuilder builder = new StringBuilder();
builder.append("Flight Number is ")
       .append(flightNo)
```

```
        .append(" and seat Number is")
        .append(seatNumber);
```

실제 구현은 다음과 같은 식이다.

```
public StringBuilder append(String str) {
     super.append(str);
     return this;
}
```

연쇄 호출이나 메소드 연쇄 등은 Java SDK를 비롯하여 다양한 라이브러리나 프레임워크에서 많이 사용되고 있다. 참고로, 메소드 연쇄는 플루언트 인터페이스(Fluent Interface, 유창한 인터페이스)의 종류 중 하나로 간주된다. 플루언트 인터페이스에는 메소드 연쇄 외에도 Function Sequence(기능순차나열), Nested Function(기능 속의 기능), Hybrid(이종교합) 등이 있다.

참고

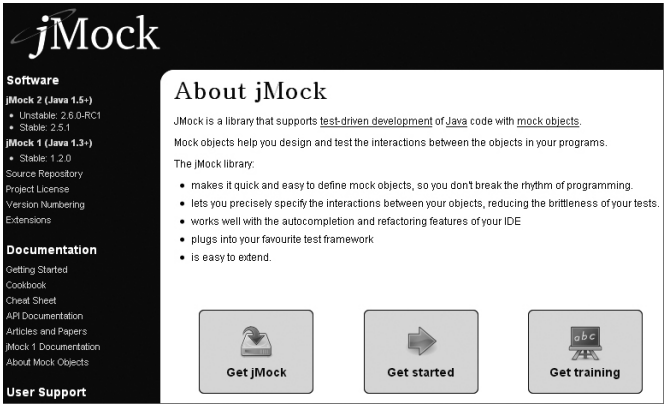
- http://blogs.atlassian.com/developer/2009/06/how_hamcrest_can_save_your_sou.html (아틀라시안 개발자)
- http://martinfowler.com/dslwip/MethodChaining.html

02 전용 Matcher 사용

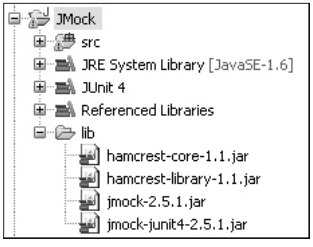
jMock은 기본적으로 Hamcrest Matcher 라이브러리를 사용하고 있다. 앞에서 Hamcrest를 설명할 때 이야기했지만, Hamcrest 자체가 jMock을 만들다 나온 라이브러리다. 이런 식의 좀 더 언어적으로 자연스러운 문체를 구현하기 위한 노력이 jMock의 특징 중 하나다.

자, 이제 실제로 jMock을 한번 사용해보자.

환경 구성



jMock을 사용하려면 우선 jmock 라이브러리가 필요하다. www.jmock.org에서 jMock2를 다운로드 받는다. 그 다음 클래스패스 내에 다음 라이브러리들을 위치시킨다.



JUnit 3 관련 라이브러리는 제외하고 클래스패스에 추가한 모습

기본 사용법

jMock은 Expect와 Verify가 중심을 이루며 다음과 같은 네 단계로 진행된다.

CreateMock	인터페이스에 해당하는 Mock 객체를 만든다.
Expect	Mock 객체의 예상되는 동작을 미리 지정한다.
Exercise	테스트 메소드 내에서 Mock 객체를 사용한다.
Verify	(프레임워크가) 예상했던 행위가 발생했는지 검증한다.

마지막 단계인 Verify는 EasyMock과 달리 작성자가 직접 지정하지 않는다. 프레임워크가 자동으로 판별한다.

01 테스트 픽스처 클래스 정의와 Mockery 생성

JUnit 4와 jMock을 함께 사용할 때는 @RunWith로 JMock.class를 설정한 다음, Mockery 클래스를 만든다. Mockery는 Mock 객체를 만들고, 예상 동작(Expectation)을 지정하는 데 사용된다. jMock에서는 이 Mockery 클래스를 컨텍스트(context)라고 부른다. 그래서 일반적으로 변수명으로 context라는 이름을 사용한다. 여기서는 JUnit 4를 이용할 예정이기 때문에, JUnit4Mockery()를 이용한다.

```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;

@RunWith(JMock.class)
class PublisherTest {
    Mockery context = new JUnit4Mockery();
    ...
}
```

02 Mock 객체 만들기

테스트 케이스 작성 시에 사용될 인터페이스를 mock 메소드를 이용해 Mock 객체로 만들어낸다. 여러 개를 만들 수 있다.

```
context.mock(타깃_인터페이스);
```

예

```
final Subscriber subscriber = context.mock(Subscriber.class);
```

03 예상동작 지정 및 테스트 수행

Mock 객체를 어떻게 사용할지를 미리 정한다. 해당 방식대로 Mock 객체가 사용되지 않으면 예외가 발생된다.

```
context.checking(new Expectations() {{
        …예상동작들을 지정한다...  // Expectation
}});
```

예

```
context.checking(new Expectations() {{
    oneOf (subscriber).receive(message);    // ❶
}});
```

❶ 이 부분을 예상값 지정 부분, 즉 Expectation이라고 부른다.

한 가지 특이한 점은 Moockery인 context로 예상값을 지정하는 부분의 모습이다. '{'가 연속적으로 두 번 사용된 것이 보인다. 이건 Anonymous 클래스로 만들어지고 있는 Expectation 클래스의 필드 부분에 멤버블록(member block)을 만든 코드다. 이 부분은 클래스의 필드에 해당하기 때문에, 클래스가 생성됨과 동시에 호출된다. 아래 예제를 보자.

```
public class Job {

    public void printYahoo(){
        System.out.println("YAHOO");
    }
    public static void main(String[] args) {
        Job job = new Job();
        job.printYahoo();

        new Job(){ {printYahoo();} };
    }
```

```
}

-----실행 결과-----

YAHOO
YAHOO
```

위 코드는 Job 클래스의 메소드인 printYahoo를 호출하는 두 가지 방식을 보여주고 있다. 변수를 지정해서 생성한 다음, 해당 변수를 통해 호출하는 방식과, anonymous 클래스의 필드블록을 이용해 호출하는 방법, 두 가지다. 위 예제에서 Job 클래스가 생성되는 순간 필드에 해당하는 printYahoo() 메소드가 호출된다. jMock에서는 이처럼 다소 트릭 같은 방식이 사용되고 있다. Expectation의 사용 예는 다음과 같다.

04 예상값 지정하기

Expectations 클래스에서 사용 가능한 메소드들은 다음과 같은 방식으로 구분되어 사용된다.

```
호출횟수지정메소드(Mock_객체).Mock객체메소드(argument_지정);   // ❶
inSequence(sequence-name);    // ❷
when(state-machine.is(state-name));    // ❸
will(action);    // ❹
then(state-machine.is(new-state-name));
```

❶ Mock 객체를 지정한 다음, Mock 객체의 어떤 메소드가 몇 번 호출될 것인지를 미리 예상해놓는다. 조건과 다르면 테스트를 실패로 간주한다.

oneOf	딱 한 번만 호출
exactly(n)	정확히 n번만큼 호출
atLeast(n).of	적어도 n번 호출
atMost(n).of	최대 n번까지 호출

between(min, max).of	min에서 max 사이로 호출
allowing	호출될 수도 있고 안 될 수도 있고
ignoring	allowing과 동일. 문장적으로 어색하지 않은 걸 둘 중에서 선택해 사용
never	호출되면 안 됨

```
oneOf(car).turnLeft(45);
allowing(car).LEDLightOn();
ignoring(turtle2);
allowing(car). klaxon(); will(returnValue("빵빵"));
atLeast(1).of(car).stop();
```

❷ Expectation은 꼭 하나의 Mock 객체 메소드만 지정 가능한 것이 아니라 여러 번 사용해서 여러 개를 지정할 수 있다. 이럴 경우 순서대로 호출돼야 한다는 조건을 붙이고 싶으면 inSequence 메소드를 사용한다.

```
final Sequence carDrive = context.sequence("carDrive");
…
context.checking(new Expectations() {{
    oneOf(car).turnLeft(45); inSequence(carDrive);
    oneOf(car).turnRight(50); inSequence(carDrive);
}});
```

❸ jMock 상태 머신이라는 개념을 따로 만들어서 해당 상태일 경우에 실행되도록 예상값을 만들 수 있다. 이때 then으로 상태를 지정하고, when으로 확인하는 식이다.

```
final States gear = context.states("gear").startsAs("down");
…
context.checking(new Expectations() {{
    oneOf(car).startup(); then(gear.is("down"));
    oneOf(car).drive(); when(gear.is("up"));
}});
```

❹ void 메소드가 아닌 리턴값이 있는 메소드일 경우 결과값을 지정한다.

05 검증

jMock은 명시적으로 행위 검증을 표시하지 않고 프레임워크가 알아서 처리해준다. 해당 테스트 메소드 내에서 예상대로 동작이 일어나지 않으면 실패로 간주한다.

응용 예제

소음을 측정하는 NoiseChecker라는 클래스를 만든다고 가정해보자. NoiseChecker는 INoise라는 인터페이스를 구현한 클래스를 변수 타입으로 받아서 소리를 측정하게 된다. 그래서 소리를 측정한 후 네 가지 타입으로 결과를 알려준다. 소음 정도는 다음의 네 가지다.

소리없음(MUTE), 조용함(SILENT), 소음있음(NOISY), 시끄러움(LOUD)

INoise 구현체를 현재 구할 수 없거나, 객체 생성이 어려울 경우 Mock 프레임워크를 사용하게 된다. jMock을 사용해 구현하면 다음과 같다.

```
public interface INoise {
    public int sound();
}

@RunWith(JMock.class)
public class INoiseTest {
    Mockery context = new JUnit4Mockery();

    @Test
    public void testSound_MUTE() {
        final INoise noise = context.mock(INoise.class);
        final int MUTE = 0;
        context.checking(new Expectations(){
            {
                allowing(noise).sound();
                will(returnValue(MUTE));
            }
        });
```

```
        NoiseChecker checker = new NoiseChecker();
        assertThat(checker.checkDecibel(noise), is(SoundType.MUTE));
    }
}
```

만일 동일 테스트 코드를 EasyMock으로 작성한다면 다음과 같다.

```
public class NoiseCheckerTest {

    @Test
    public void testSound_MUTE() {
        INoise noiseMock = createMock(INoise.class);
        final int MUTE = 0;
        expect(noiseMock.sound()).andReturn(MUTE);
        replay(noiseMock);

        NoiseChecker checker = new NoiseChecker();
        assertEquals(SoundType.MUTE, checker.checkDecibel(noiseMock));
        verify(noiseMock);
    }
}
```

이어서 배우게 될 Mockito 프레임워크의 경우에는 어떻게 테스트가 구현될지 기대해 보자.

다음은 테스트 케이스를 통해 구현된 NoiseChecker의 최종 모습이다.

```
enum SoundType {
    MUTE, SILENT, NOISY, LOUD;
}

public class NoiseChecker {
    public SoundType checkDecibel(INoise noise){
        if (noise.sound() == 0 ) {
            return SoundType.MUTE;
        } else if ( noise.sound() > 0 && noise.sound() < 10){
            return SoundType.SILENT;
```

```
            } else if ( noise.sound() >= 10 && noise.sound() < 100){
                return SoundType.NOISY;
            } else if ( noise.sound() >= 100){
                return SoundType.LOUD;
            } else {
                throw new IllegalArgumentException();
            }
        }
    }
```

jMock 정리

jMock은 jMock 자체보다, 그 안에 담겨 있는 여러 가지 참고할 만한 아이디어가 많다. 그리고 생각보다 jMock을 사용하는 사람들도 적지 않다. 하지만 현재 모습은 jMock보다 jMock에서 파생한 Hamcrest의 인기가 더 높은 아이러니를 가진 Mock 프레임워크가 되었다.

추천 4.3.3 Mockito[13]

Mockito는 최근에 급격히 떠오르고 있는 Mock 프레임워크다. 역사는 오래되지 않았지만, 간편한 사용법으로 인해 빠르게 확산되고 있다. 또한 Mockito가 인기를 얻게 된 데는, 전통적인 TDD 개발자들이 주로 사용했던 방식인 상태 기반 테스트를 지향한다는 점도 크게 작용했다. 심지어 BDD(행위 주도 개발)의 창시자인 댄 노스(Dan North)는 "JUnit 4와 Mockito가 Java 영역에서의 TDD와 목킹(mocking)[14]의 미래다"

13 '목키토'라고 읽는다.

14 mock 객체를 만드는 작업

라고까지 말할 정도로 일부 지지자에겐 절대적인 Mock 프레임워크로 자리잡아 가고 있다.

Mockito의 특징

Mockito를 개발한 수제빵 파베르(Szczepan Faber)는 자신의 사이트에서 좋은 Mock 프레임워크는 어때야 하는지를 설명하고 있다. 그 내용이 바로 Mockito라는 Mock 프레임워크를 만들게 된 동기이자 Mockito의 특징에 해당한다.

Mock 프레임워크는 어때야 하는가?

- 단순해야 한다. 결국 상호작용이란 건 따지고 보면 메소드 호출이 전부 아닌가? Mock을 만드는데 언어를 배우는 수준의 복잡함이 왜 필요한가?
- Mock 프레임워크를 DSL[15]로 만들지 말자. 복잡해진다.
- 문자열을 메소드 대신에 사용하지 말자.[16]
- 읽기 어려운 anonymous inner 클래스를 사용하지 말자.
- 리팩토링이 어려워서는 안 된다.

그가 이야기하는 문제점들의 대부분은 마치 jMock을 지칭하는 것처럼 들린다.

Mockito 프레임워크의 차별점은 무엇인가?

1. **테스트 그 자체에 집중한다.**

   테스트의 행위와 반응(interaction)에만 집중해서 테스트 메소드를 작성할 수 있게 한다.

15 DSL(Domain-Specific Language): 특정 문제영역을 해결하기 위해 만들어진 언어나 스펙을 통칭한다. 반대말로는 GPPL(General-purpose programming language)이 있는데, 범용 프로그래밍 언어 C, C++, Java 등을 지칭한다. 위에서 DSL을 언급한 건 TDD 시 발생하는 문제상황(problem domain)을 해결하기 위해 DSL 수준의 프레임워크를 만드는 경향이 있기 때문이다. 특히 jMock이 이런 DSL 스타일을 지향하는 대표적인 Mock 프레임워크다.

16 call("getName") 식으로 사용하고 리플렉션으로 getName이 호출되는 경우를 지칭한다.

2. 테스트 스텁을 만드는 것과 검증을 분리시켰다.

3. Mock 만드는 방법을 단일화했다.

4. 테스트 스텁을 만들기 쉽다.

5. API가 간단하다.

6. 프레임워크가 지원해주지 않으면 안 되는 코드를 최대한 배제했다.

   record( ), replay( ) 메소드도 없고 에일리언 코드 같은 control/context 객체도 없다.

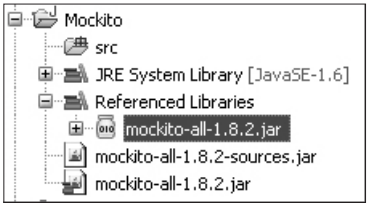
7. 실패 시에 발생하는 에러추적(stack trace)이 깔끔하다.

   특정 API나 라이브러리를 모르면 이해할 수 없는 부분을 제외했다.

쉽게 말하자면, Mockito는 EasyMock과 jMock의 단점을 보완하기 위해 나온 Mock 프레임워크라고 생각하면 되겠다.[17]

환경 구성

Mockito 사이트에서 파일을 내려받아서 mockito-all-x.x.x.jar 파일을 클래스패스에 포함시킨다. 좀 더 자세히 살펴보고 싶다면 소스까지 프로젝트 안에 넣어놓는다.



라이브러리를 내려받아서 클래스패스에 포함시킨 모습

기본 사용법

Mockito는 Stub 작성과 Verify가 중심을 이루며 다음과 같은 순서로 진행된다.

17 참조: http://monkeyisland.pl/2008/01/14/mockito/

CreateMock	인터페이스에 해당하는 Mock 객체를 만든다.
Stub	테스트에 필요한 Mock 객체의 동작을 지정한다(단, 필요 시에만).
Exercise	테스트 메소드 내에서 Mock 객체를 사용한다.
Verify	메소드가 예상대로 호출됐는지 검증한다.

여타 프레임워크는 Stub과 Expectation을 둘 다 진행하지만, Mockito는 Stub 작업만 한다.

01 Mock 객체 만들기

```
Mockito.mock(타깃 인터페이스);
```

예

```
import static org.mockito.Mockito.*;
…
List mockedList = mock(List.class);
```

Mockito 클래스의 static 메소드인 mock을 이용해 인터페이스나 클래스를 지정한다. 이후부터는 구현 클래스로 객체가 생성된 것처럼 동작한다. 물론 메소드들은 현재 정상 동작하진 않는다. Mock 메소드 외에도 여러 가지 static 메소드가 사용되기 때문에 일반적으로 위 예제처럼 Mockito 클래스를 static import 처리해서 사용한다.

```
List mockedList = mock(List.class);
System.out.println(mockedList.size());

-----실행 결과-----

0
```

리턴 타입(return type)의 기본값으로 동작한다. int는 0, boolean은 false, 클래스 타입들은 null을 돌려주는 식이다.

02 예상값 지정

다른 Mock 객체와 두드러지게 차이나는 부분 중 하나가 바로 예상값에 대한 접근 방식이다. Mockito는 예상값을 지정하는 것이 아니라, 필요 시에 테스트 스텁만 만들고 나중에 호출 여부를 확인하는 방식을 사용한다. 기존 프레임워크가 **스텁 → 예상 → 수행 → 검증**이라면, Mockito는 **스텁 → 수행 → 검증**으로 단순화되어 있다. '뭣하러 미리 예상 행동을 고민해야 하느냐? 테스트를 수행하고 결과(상태)를 보자'의 개념이다. 그래서 Mockito는 예상값 지정 부분이 없다.

03 테스트에 사용할 스텁 만들기

```
when(Mock_객체의_메소드).thenReturn(리턴값);
when(Mock_객체의_메소드).thenThrow(예외);
```

단, 스텁은 필요할 때만 만드는 것이 원칙이다. 메소드 호출 여부를 검증만 할 때는 사용하지 않는다. 다음 소스를 살펴보자.

```
List mockedList = mock(List.class);

// Mock 객체를 사용한다.
mockedList.add("item");
mockedList.clear();

// 검증
verify(mockedList).add("item");
verify(mockedList).clear();
```

위와 같은 경우에 Mock 객체를 생성만 하고, 나중에 호출을 검증하는 식으로 테스트 코드가 작성됐다. 만일 테스트를 위해 Mock 객체의 특정 메소드를 사용해야 할 필요가 생긴다면 그때 만든다.

```
// Stub 만들기
when(mockedList.get(0)).thenReturn("item");
when(mockedList.size()).thenReturn(1);
when(mockedList.get(1)).thenThrow(new RuntimeException());

System.out.println(mockedList.get(0));
System.out.println(mockedList.size());
System.out.println(mockedList.get(2));
System.out.println(mockedList.get(1));

-----실행 결과-----

Item
1
null
Java.lang.RuntimeException
    at SimpleMockTest.testMockList(SimpleMockTest.Java:25)
...
```

04 검증

```
verify(Mock_객체).Mock_객체의_메소드;
verify(Mock_객체, 호출횟수지정_메소드).Mock_객체의_메소드;
```

Mock 객체의 특정 메소드가 호출됐는지 확인한다. 호출 횟수도 지정해서 검증할 수가 있다.

호출횟수 지정 메소드 종류

times(n)	n번 호출됐는지 확인. n=0은 times를 지정하지 않았을 때와 동일하다.
never	호출되지 않았어야 함
atLeastOnce	최소 한 번은 호출됐어야 함
atLeast(n)	적어도 n번은 호출됐어야 함
atMost(n)	최대 n번 이상 호출되면 안 됨

예 1

```
verify(mockedList).add("item");
verify(mockedList, times(1)).add("item");

verify(mockedList, times(2)).add(box);

verify(mockedList, never()).add(car);

verify(mockedList, atLeastOnce()).removeAll();
verify(mockedList, atLeast(2)).size();
verify(mockedList, atMost(5)).add(box);
```

예 2

```
import static org.mockito.Mockito.*;

public class SimpleMockTest {

    @Test
    public void testMockList() throws Exception {
        List mockedList = mock(List.class);

        verify(mockedList).size();
    }
}

-----실행 결과-----

Wanted but not invoked:
list.size();
-> at SimpleMockTest.testMockList(SimpleMockTest.Java:19)
Actually, there were zero interactions with this mock.
```

verify 메소드를 이용해 mockedList.size()가 호출됐는지 확인한다. 상태 기반 테스트에 유용하다.

검증을 할 때 매번 특정한 메소드의 호출을 그대로 표현할 순 없는 경우가 있다. 메소드의 인자로 특정한 값을 지정하는 것이 아니라, 일종의 패턴 같은 방식을 사용해야 할 경우를 말한다. 이를테면 다음과 같은 검증 구문이 있다고 가정해보자.

```
verify(mockedList, times(5)).add( box );
```

이렇게 작성하면 mockList.add(box)가 5번 호출됐는지 검증하라는 뜻인데, 인자까지는 특정하지 않고 add 메소드가 5번 사용됐기만 하면 될 때는 어떻게 해야 할까? 그럴 때 사용하는 것이 Argument Matcher이다.

a. Argument Matcher

앞 예제에 Argument Matcher를 사용하면 다음과 같이 표현할 수가 있다.

```
verify(mockedList, times(5)).add( any() );
```

즉, 어떤 객체가 add 메소드의 인자로 와도 무방하고, 대신 5번 호출됐다면 성공이란 뜻이다. Mockito의 Argument Matcher는 메소드로 구현되어 있고, 검증(verification) 단계 및 Stub을 만들 때 모두 쓸 수 있다. 대표적인 종류는 다음과 같다.

any	어떤 객체가 됐든 무방
any 타입	anyInteger( ), anyBoolean 등 Java 타입에 해당하는 any 시리즈가 있다. 이런 시리즈들은 null이거나 해당 타입이면 만족한다.
anyCollection anyCollectionOf	List, Map, Set 등 Collection 객체이면 무방. anyCollectionOf는 anyCollection과 동일하다. 자연스런 문장을 위해 사용한다.
argThat(HamcrestMatcher)	Mockito도 Hamcrest Matcher를 사용하고 있다. Hamcrest에서 사용하던 Matcher를 그대로 이 부분에서 사용할 수 있다.
eq	Argument Matcher가 한번 사용된 부분에선 Java의 타입을 그대로는 더 이상 쓸 수가 없다. verify(mock).add(anyString(), “item”); 즉, 위와 같이는 쓸 수 없다. 이럴 때 아래와 같이 eq를 이용한다. verify(mock).put(anyString(9), eq(“item”));

anyVararg	여러 개의 인자를 지칭할 때 사용한다.
	
	예)
	// verification
	mock.foo(1, 2);
	mock.foo(1, 2, 3, 4);
	
	verify(mock, times(2)).foo(anyVararg( ));
	
	// Stubbing:
	when(mock.foo(anyVararg( )).thenReturn(100);

matches(String regex)	정규식 문자열로 인자(argument) 대상을 지칭한다.
startswith(String) endWith(String)	특정 문자열로 시작하거나 끝나면 OK
anyList anyMap anySet	anyCollection의 좀 더 디테일한 버전이다. 해당 타입이기만 하면 OK
isA(Class)	해당 클래스 타입이기만 하면 된다.
isNull	Null이면 OK
isNotNull	null만 아니면 OK

b. 순서 검증

만일 Stub으로 만들어진 Mock 객체 메소드의 호출 순서까지 검증하고 싶다면 InOrder 클래스를 이용한다.

```
List firstMock = mock(List.class);
List secondMock = mock(List.class);

firstMock.add("item1");
secondMock.add("item2");

InOrder inOrder = inOrder(firstMock, secondMock);
inOrder.verify(firstMock).add("item1");
inOrder.verify(secondMock).add("item2");
```

만일 add 메소드의 호출이 item1 → item2 순으로 이뤄지지 않았다면 실패로 간주한다.

Mockito의 특징적인 기능

01 void 메소드를 Stub으로 만들기

void 메소드는 특별히 리턴할 내용이 없기 때문이 Stub으로 만들 일이 거의 없다. 그런데 예외가 딱 하나 있는데, 바로 void 메소드에서 예외(exception)가 발생하는 경우를 Stub으로 구현할 때다. 그때 사용할 수 있는 메소드가 doThrow이다.

```
doThrow(예외).when(Mock_객체).voidMethod();
```

예

```
doThrow(new RuntimeException()).when(mockList).clear();
```

02 콜백으로 Stub 만들기: thenAnswer

Mock은 보통 하드코딩된 값만 돌려주도록 만들 수 있다. 그런데 특정 Mock 메소드에 대해 실제 로직을 구현하고자 할 때 콜백(CallBack) 기법을 사용한다. 그런데 TDD 자체에서 권장하는 방식은 아니라는 걸 알아두자. 테스트에 사용하는 대상의 일부에 테스트를 위한 로직을 강제로 넣는다는 건 역시 깔끔한 테스트 케이스 작성과는 거리가 멀어진다. 다음 예는 필자가 JDBC 테스트를 위해 만들었던 코드의 일부다. 내용을 볼 필요는 없고, 모양만 살짝 살펴보자.

```
when(rs.getInt("no")).thenAnswer( new Answer<Integer>(){
    public Integer answer(InvocationOnMock invocation) throws Throwable {
        if (currentRecord == null)
            throw new SQLException("access fields is empty");
        return ((Integer)currentRecord[0]).intValue();
    }
});
```

03 실제 객체를 Stub으로 만들기: SPY

지금까지 인터페이스를 Mock으로 만들어왔다. 그런데 Mockito는 실 객체도 Mock으로 만들 수 있다. Mockito의 강력한 기능 중 하나인데, 너무 강력해서 오히려 문제가 될 수도 있는 기능이다. 부분 Mocking이라고 불리는 방법으로 서드파티(3rd Party) 제품, 고칠 수 없는 라이브러리만 남아 있는 코드 등에 대해서만 한정적으로 사용하는 걸 권장한다. Mockito의 저자는 spy 기능을 쓰게 된다면, 그건 이미 뭔가 잘못된 코드를 건드리고 있다는 증거라고 이야기하고 있다.

예

```
ArrayList<String> realList = new ArrayList<String>();
realList.add("Hello");
System.out.println(realList.get(0));

List mockedList = spy( realList );

when(mockedList.get(0)).thenReturn("item");
System.out.println(mockedList.get(0));

-----실행 결과-----

Hello
item
```

이 정도면 DynamicProxy 정도의 레벨은 훌쩍 벗어난 수준이다. 참고로, final로 선언된 메소드는 Stub으로 만들 수 없다.

04 똑똑한 NULL 처리: SMART NULLS

Stub으로 만들지 않은 기본형 외의 값을 리턴하는 메소드는 null 값이 찍힌다. 이렇게 만들어진 Mock 객체들은 종종 NPE(Null Pointer Exception)를 유발한다. 필요에 따라, 좀 더 유용한 값이 기본값으로 찍히게 만드는 것이 SMART NULLS라는 방식이다.

```
List mockedList = mock(List.class);
List smartMockedList = mock(List.class, RETURNS_SMART_NULLS);

System.out.println(mockedList.toArray());
System.out.println(smartMockedList.toArray());

-----실행 결과-----

null
[LJava.lang.Object;@3eca90
```

SMART NULLS 규칙

- 기본형 래퍼(primitive wrapper) 클래스는 해당 기본형 값으로 바꾼다.
- String은 “”로 바꾼다.
- 배열은 크기 0인 기본 배열 객체로 만들어준다.
- Collection 계열은 빈 Collection 객체로 만든다.

향후 Mockito 2.0부터는 SMART NULLS가 기본값으로 채택될 예정이다.

05 행위 주도 개발(BDD) 스타일 지원

Mockito는 //given //when //then 식의 행위 주도 개발(Behavior-Driven Development, BDD) 스타일로 테스트 케이스 작성할 수 있게 지원해준다. BDD 스타일을 사용하려면 Mockito 클래스 대신 BDDMockito를 static import 한다. 다음은 해당 예다.

```
import static org.mockito.BDDMockito.*;

Seller seller = mock(Seller.class);
Shop shop = new Shop(seller);

public void shouldBuyBread() throws Exception {
```

```
        // given
        given(seller.askForBread()).willReturn(new Bread());

        // when
        Goods goods = shop.buyBread();

        // then
        assertThat(goods, containBread());
    }
```

BDD에 대해서는 8.7절 '행위 주도 개발'에서 좀 더 자세히 다룰 예정이다.

응용 예제

앞서 jMock에서 사용했던 테스트 케이스의 Mockito 버전은 다음과 같다.

```
public class INoiseTest {

    @Test
    public void testSound_MUTE() {
        final INoise noise = mock(INoise.class);
        final int MUTE = 0;
        when(noise.sound()).thenReturn(MUTE);

        NoiseChecker checker = new NoiseChecker();
        assertThat(checker.checkDecibel(noise), is(SoundType.MUTE));
        verify(noise).sound();    // ❶
    }

    @Test
    public void testSound_NOISY() {
        final INoise noise = mock(INoise.class);
        final int NOISY_SOUND = 11;
        when(noise.sound()).thenReturn(NOISY_SOUND);

        NoiseChecker checker = new NoiseChecker();
        assertThat(checker.checkDecibel(noise), is(SoundType.NOISY));
```

```
        verify(noise).sound();    // ❶
    }
}
```

❶ 만일 noise.sound() 호출 여부가 테스트 결과에 영향을 미친다면, 보는 것과 같이 verify 메소드를 명시적으로 호출할 수도 있다. 물론 테스트 결과에 관여하지 않는다면, 해당 부분을 제거한다.

Mockito 정리

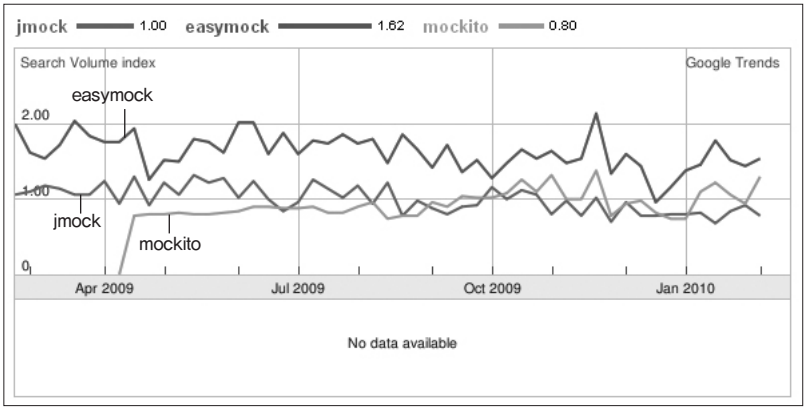
Mockito는 사용법이 간단하고 지향하는 바가 뚜렷하다. 'TDD를 위한 테스트 코드 작성에 전념할 수 있도록 도와준다. 최대한 간결하게 사용할 수 있도록 한다.' 이게 Mockito의 지향점이다. 만일 시간이 없어서 지금까지 언급한 세 개의 Mock 프레임워크 중 단 하나만 사용해야 한다면, 개인적으로는 Mockito를 추천한다.

4.4 Mock 프레임워크 마무리

이상으로 대표적인 Mock 프레임워크를 함께 살펴봤다. 봐서 알겠지만, Mock 프레임워크들은 그 기발함으로 인해, 종종 욕심 있는 엔지니어들을 빠져들게 만든다. 이건 사견이지만, 많은 테크니션들이 기교에 빠져들어 목적을 잊어버리는 것 같다. '지금 하려고 하는 일의 목적은 무엇인가?'라는 질문을 스스로에게 해볼 필요가 있다.

자, 이제는 Mock 프레임워크의 트렌드와 Mock 사용 시의 유의점을 살펴보는 걸로 이야기를 마무리할까 한다. 어렵고도 긴 다리를 건너오느라 고생이 많았다. 조금만 더 힘을 내자.

Mock 프레임워크 트렌드 비교



위는 2010년 2월 기준으로 지난 12개월간의 Google Trends 그래프다. 즉, 구글 검색에서 활발하게 찾아지는 빈도를 보여주는 그래프인데, 높으면 높을수록 인기가 있는 분야(검색어)라고 보면 된다. jMock에 대한 검색비율을 1로 봤을 때 EasyMock은 1.6, Mockito는 0.8이다. 여전히 전 세계적으로 봤을 때는 EasyMock이 강세다. 오래됐기도 하고, 기존에 이미 EasyMock으로 작성된 소스들이 많기 때문이다. 하지만 그래프 추이를 보면, Mockito의 성장세가 두드러지는 걸 알 수 있다. 아마 앞으로 당분간은 더 활발히 증가하리라 예상된다. 사실 프레임워크 간의 우위를 이야기하는 건 다소 논란의 소지가 있는 부분이다. 바질리 시즈코프(Vasily Sizov)라는 한 외국 개발자가 대표적인 7개의 Mock 프레임워크를 비교해서, 항목별로 점수를 매겨 인터넷에 올린 자료가 있다. Mockito가 1등을 했고 그 뒤로 EasyMock과 jMock이 2, 3등을 했다. 아직도 다양한 논쟁이 해당 글에 댓글로 달리고 있으니 관심 있는 사람은 찾아보길 바란다.

해당 내용은 http://www.sizovpoint.com/2009/03/java-mock-frameworks-comparison.html에서 찾아볼 수 있다.

Mock 사용 시 유의사항

– Mock 프레임워크가 정말 필요한지 잘 따져본다

적지 않은 개발자들이 Mock 객체와 Mock 프레임워크를 접하고는, 그 기능과 방식에 매료돼서는 Mock을 사용하는 것 자체가 목적이 돼버리는 경우를 종종 본다. 개발을 진행해나가면서 Mock 객체가 필요한 부분이 나오는 것이 아니라, Mock 객체가 적용될 수 있는 부분을 찾아내려고 애쓰는 역전현상이 발생하는 것이다. 한번 Mock 프레임워크를 사용하면, 그 뒤로는 해당 테스트 케이스를 유지하는 데 드는 비용이 지속적으로 발생한다. 그리고 Mock 객체들은 흔히 깨지기 쉬운 테스트 케이스(fragile testcase)가 돼버리는 경향이 있다. 만일 가능만 하다면 설계를 바꿔서라도 Mock 필요 없는 의존성 적은 구조로 만들어놓길 바란다. 그 편이 Mock 객체를 사용하려고 노력하는 것보다 훨씬 낫다.

– 투자 대비 수익(ROI)[18]이 확실할 때만 사용한다

테스트용 DB를 설치하는 데 반나절이 걸린다고 가정하자.[19] 이럴 때 DB와 연관된 기능 부분을 Mock 객체로 만드는 것이 옳은 선택일까, 잘못된 선택일까? Mock을 사용하려 할 때는 좀 더 길게 볼 필요가 있다. 장기적으로 이득일지 고민해보자. 테스트를 편리하게 만들기 위해 사용하기 시작한 Mock 객체들이 시간이 지날수록 불어날 텐데, 자칫 개발 진척의 발목을 잡는 족쇄가 될 수도 있다.

– 어떤 Mock 프레임워크를 사용하느냐는 핵심적인 문제가 아니다

어떤 Mock 프레임워크를 사용할 것인지에 대해 지나치게 많은 고민을 하는 경우가 있다. 그리고 어떤 Mock 프레임워크가 선정되느냐에 따라 향후 테스트 케이스 작성에 커다란 영향을 미칠 것처럼 느껴질 수 있다. 하지만 사실은 그렇지 않다. 앞에서 소개한 어떤 Mock 프레임워크를 사용한다 하더라도 익숙해지기 전까지는 불편할 수밖에 없다. 그러니, 유행 따라 최고의 Mock 프레임워크를 찾아 떠나는 여정은 하

18 ROI(Return On Investment): 투자비용 대비 수익

19 요즘 같은 시절엔 DB를 설치하는 데 30분이면 되는 경우도 많다. Oracle 같은 대형 DBMS도 Express 버전은 20분이면 설치가 끝난다.

지 않길 바란다. 앞에서도 살펴봤지만, 대부분의 경우 간단한 Mock 객체만으로도 충분하다. 어떤 측면에서는 Mock 프레임워크를 사용했다는 것만으로도 이미 어려운 길에 들어서는 것이다.

– Mock은 Mock일 뿐이다

Mock 객체를 사용해 아무리 잘 작동하는 코드를 만들었다 하더라도, 실제 객체(real object)가 끼어들어 왔을 때도 잘 동작하리라는 보장은 없다. Mock 객체는 실제 객체나 실제 시스템에 대해 단지 흉내만, 그것도 오로지 당신이 원하는 형태로만 낼 뿐이란 걸 잊어서는 안 된다. 결국 언제가 됐든, Mock 객체는 실제 객체로 대체되어 테스트를 해야 하는 시점이 온다. 따라서 초반부터 실제 객체를 테스트에 사용할 수 있다면, 그리고 그 비용이 많이 크지 않다면, Mock 객체를 사용하지 말자.

Mockito 개발자가 이야기하는 궁극의 TDD 템플릿

Mockito 개발자가 자신의 블로그에 테스트 케이스 작성을 위한 궁극의 템플릿(ultimate test template)이라며 호들갑스럽게 밝히는 템플릿이 있다. BDD(Behavior Driven Development) 스타일을 따르는 템플릿인데, 모습은 다음과 같다.

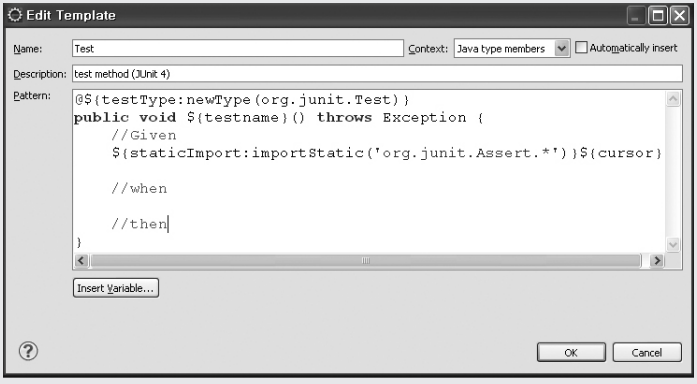
```
@Test
public void shouldDoSomethingCool() throws Exception {
     //given : 선행조건 기술

     //when  : 기능 수행

     //then  : 결과 확인
}
```

주석을 달고 문맥을 나누어 테스트 케이스를 작성하는 방식이다.

생각보다 시시한가? 하지만 그에 비해 매우 강력한 템플릿이라고 써본 사람들은 이야기한다. 테스트 케이스를 작성하는 데도 도움이 많이 되고, 작성된 테스트 케이스를 보게 될 다른 개발자에게도 도움이 많이 된다고 한다. 별것 아닌 것 같지만, 의미적으로 각 단계를 구분해주고 있다. 확실히 가독성이 높아진다. 이클립스의 템플릿을 수정해서 테스트 케이스를 만들 때 자동적으로 주석이 달리도록 만들 수도 있다. 다음은 그 방법이다.

BDD 스타일을 이클립스 템플릿으로 등록하기

이클립스 메뉴에서 [Window] → [Preferences] → [Java] → [Editor] → [Templates] 순으로 찾아간다. JUnit 4 테스트 템플릿에 해당하는 Test 템플릿을 선택한 다음 아래 그림처럼 주석을 넣는다.



광고

"저처럼 어리석은 짓을 하지 마시고, 바로 오늘부터 //given //when //then 템플릿을 쓰기 시작하세요. 삽질하기엔 인생은 너무 짧답니다. 소프트웨어 장인 레벨 85에 빨리 올라갔으면 싶으시죠? 이렇게 한번 해보세요. 우선 먼저, 개발 주기[20]를 하나 선정해서 그동안에 이 템플릿을 써보세요. 그러고 나서 결과가 맘에 들지 않는다면 전액 환불해드립니다! 정말 진지하게 말씀드리는 건데요, 생각할 필요도 없습니다. 지르세요!" – Mockito 개발자

참고

http://monkeyisland.pl/2009/12/07/given-when-then-forever/

20 애자일 개발에서 이야기하는 반복개발 단위. 보통 2~4주 정도가 된다.

전문가 인터뷰

황상철 삼성 SDS 소프트웨어아키텍처 Senior

Q. 본인 소개와 하시는 일, 혹은 관심 기울이고 있으신 일에 대해 말씀해 주시겠습니까?

A. 현재 삼성 SDS의 SW Eng.팀에서 엔터프라이즈 자바 CoE 리더를 맡고 있습니다. 팀에서는 주로 팀원들의 역량강화를 담당하고 있으며 프로젝트에 직접 애자일 개발을 도입하는 역할을 담당합니다.

Q. 현재 테스트 주도 개발(이하 TDD)을 업무에 적용하고 계신가요?

A. 제가 강의하는 팀의 개발자 필수 교육이 있는데 이 과정에서 진행되는 모든 개발을 TDD로 진행하고 있습니다. TDD를 적용하면서부터 유사한 개발 교육에 비해 큰 학습 효과를 보고 있습니다.

Q. 개발자 필수 교육 때 TDD를 사용한다고 하셨는데요, 어려움은 없으신지요?

A. 저도 처음 TDD를 가르치려고 마음먹었을 때 반신반의했습니다. 회사에서 반대를 하지는 않을까? 수강생들이 전혀 이해를 못하면 어쩌나. 그런데 오히려 상황은 반대였습니다. 제가 가르친 애자일 기법 중 가장 현실적이라고 뽑은 게 TDD로 나왔습니다. 놀랍죠^^

Q. TDD에 대한 경험담이 있으시면 소개해주실 수 있는지요? 좋은 기억이 아니어도 무방합니다.

A. 프로젝트에서 SA로 일하면서 Test Code 작성을 개발자에게 강요한 적이 있습니다. 물론 실패했죠^^ TDD가 아니더라도 테스트 코드가 개발에 필수 요소가 돼야 한다고 믿습니다. 그래서 테스트 코드를 잘 모르는 개발자들을 위해 테스트 코드를 만들어주는 툴을 개발해서 사내에 배포하고 있습니다.

Q. TDD를 바라보는 본인의 생각, 혹은 'TDD는 무엇이다!'라고 이야기한다면?

A. TDD는 '발상의 전환'이라고 생각합니다. 단순히 테스트 코드를 먼저 만드는 게 중요한 게 아니라 생각 자체를 뒤집는 것입니다. 그래서 배우기 어렵고 익숙해지기 어려운 것 같습니다.

Q. TDD를 적용하는 데 있어 중요하다고 생각하는 부분, 혹은 유의점은 무엇이라고 생각합니까?

A. TDD가 좋은 점이 있다고는 하나 항상 사용하기는 어려운 일입니다. 현실적으로 봐도 어렵다 생각합니다. 하지만 그렇다고 아예 시도도 안 해보는 것은 TDD에서 배울 수 있는 좋은 점을 포기하는 것입니다. 필요한 때에 필요한 만큼 쓸 줄 알아야 합니다.

Q. 언제 TDD를 쓰면 좋을지 후배가 묻는다면 어떻게 대답하시겠습니까?

A. 두 가지를 이야기하고 싶습니다.

첫 번째, 모르는 분야를 공부할 때 TDD를 써라. 사실 TDD를 제대로 하는 것은 약간 지겨울 수 있습니다. 그럴 만한 가치가 있을 때 써야 합니다. 잘 모르는 분야를 TDD로 하나하나 짚어가다 보면 감을 쉽게 잡을 수 있습니다.

두 번째, 풀리지 않는 어려운 문제를 해결할 때 TDD를 써라. 앞이 안 보이는 막막한 상황에서야말로 TDD가 적당한 해결책이 될 수 있다고 생각합니다.

Q. 바쁘신 와중에도 귀한 시간을 내어 인터뷰에 응해주셔서 감사합니다. 마지막으로, 좋은 소프트웨어를 만들고 싶어하는 후배들에게 업계 선배로서 해주고 싶은 조언이 있으시다면?

A. "준비하는 사람에게 기회는 온다"는 말을 해주고 싶습니다. 국내 소프트웨어 현실이 어렵고 힘들다고 불평만 하기보다는 더 나은 사람이 되기 위해 노력하는 사람에게 좋은 기회는 반드시 온다고 믿습니다. 모바일도 또 하나의 좋은 기회라는 생각이 듭니다.