



실패라는 건 없다. 모든 건 피드백일 뿐이다.

— 로버트 알렌

개발 영역에 따른 TDD 작성 패턴

이번 장에서는 TDD로 애플리케이션을 작성하면서 마주하게 되는 다양한 테스트 케이스 작성 상황을 살펴보고, 다양한 대응 방법에 대해 살펴볼 예정이다. 일부는 사례를 소개하고 있고, 일부는 필자가 생각하는 답을 제시하고 있다. 또한, 이제까지 배워왔던 부분들을 정리해보는 기회로 삼아보자.

7장 체크리스트

☐ 일반적인 애플리케이션

- 생성자 테스트
- DTO 객체 생성 테스트
- 닭과 달걀 메소드 테스트
- 배열 테스트
- 객체 동치성 테스트
- 컬렉션 테스트

☐ 웹 애플리케이션

- MVC 아키텍처
- 뷰 TDD
- 컨트롤러 TDD
- 모델 TDD

☐ 데이터베이스

☐ 전통적으로 잘못 인식되어 있는 테스트 메소드의 리팩토링

7.1 일반적인 애플리케이션

TDD가 가장 적극적으로 사용되고 효율이 높은 부분은 애플리케이션의 업무로직을 구현할 때다. 그렇다면 ‘로직 구현’이 개발의 중심이 되는 일반적인 개발에서는, 테스트 케이스를 작성할 때 어떠한 상황에 접하게 되는지 살펴볼 필요가 있다. 그리고 각각의 경우 어떻게 테스트 케이스를 작성하고, 또 어느 수준까지 진행할 것인지 미리 알아두면 시간절약에 많은 도움이 된다. 자 그럼 이제부터, 흔히 접하게 되는 상황별 테스트 케이스 작성에 대해 살펴보자.

생성자 테스트

- 단순히 클래스를 생성한다는 의미를 갖는 생성자(constructor)는 굳이 테스트 케이스를 작성하지 않는다. 다만, 객체 사용을 위해 반드시 갖춰야 하는 값을 생성자에 설정하는 경우는 필요에 따라 테스트를 작성한다.
- 단, 가끔 생성자에서 객체 생성의 의미뿐 아니라, 선행조건이나 업무로직을 직접 기술하는 경우도 있는데, 이럴 경우에는 테스트 케이스를 작성해야 한다. 이를테면 DAO 객체 생성 시 DB 커넥션까지 확보해야 하는 경우라면 다음과 같이 테스트 케이스를 만들 것이다.

```
public class EmployeeDaoTest {  
    @Test  
    public EmployeeDaoTest {  
        EmployeeDao dao = new EmployeeDao();  
        assertTrue(dao.isConnected());  
    }  
}
```

주의

일반적으로 좋은 애플리케이션이라면, 생성자의 파라미터로 지정된 값 외의 항목을 이용하는 코드를 생성자 메소드 안에 넣지 않는다. 자칫, 외견상 파악할 수 없는 숨겨진 로직이 되어서 모듈 자체의 복잡도를 높이기 때문이다.

DTO 스타일의 객체 테스트

- 클래스가 속성 필드와 단순 setter/getter로만 이뤄진 DTO¹ 스타일로 만들어진 경우에는 굳이 테스트 케이스를 작성하지 않는다.
- 단, 특정한 목적을 갖고 만들어진 **불변 객체**(immutable object)의 경우에는 getter 계열 메소드나 상태를 확인할 수 있는 is 계열(isReady, isConnected 등)의 메소드를 이용한 테스트 케이스를 작성하기도 한다.

7-1-1
한 번만
만들어
놓는다

불변 객체

한번 만들어진 다음엔, 내부 상태 값을 바꿀 수 없는 객체를 '불변 객체'라고 부른다. 단순히 생각하면 setter가 존재하지 않는 클래스라고 보면 된다. 대표적으로 자바의 String 클래스가 이런 불변 객체에 해당한다. String은 한번 만들어진 다음에는 내부 값을 수정할 수 없다. 새로 생성해서 새롭게 변수에 할당하는 수밖에 없다.

```
String name = "성운";  
name = "김" + name;
```

자바에서는 문자열을 위와 같이 사용할 수 있다. 마치 name의 앞에 "김"이 붙는 것처럼 보이지만, 실제로 문장은 아래와 같이 처리된다.

```
String name = new String("성운");  
name = new String("김" + name);
```

답과 달걀 메소드 테스트

메소드가 서로 맞물려 있어서, 완전히 하나만 독립적으로 테스트하기 어려운 경우가 있다. 보통 로직 메소드(add, remove, set 계열 메소드)와 상태확인 메소드(get, show, is 계열 메소드)가 짝을 이루는 경우에 해당한다. 이런 경우엔 어느 한쪽의 테스

1 DTO(Data Transfer Object): 데이터를 전달하기 위해 사용되는 객체. 보통, 로직은 들어가 있지 않고 단순히 저장하고(setter) 불러오는(getter) 메소드들로 구성되어 있다. 예전에는 VO(Value Object, 값 객체)라는 용어가 많이 쓰였으나 현재는 DTO로 통칭한다.

트 케이스를 먼저 만들기가 어렵다. 이를테면 참가자(Attendee) 목록을 관리하는 클래스가 있다고 가정해보자. 테스트 케이스로 작성해야 하는 메소드는 다음과 같다.

Attendee 클래스

void add(String name)	참가자 목록에 이름을 추가한다.
String get(int order)	해당 번째(order)로 등록된 참가자의 이름을 보여준다.

이제, Attendee 클래스를 위한 테스트 클래스와 테스트 케이스를 작성해보자. 흔히 다음과 같은 흐름으로 작성될 것이다.

```
public class AttendeeTest {
    @Test
    public void testAdd() throws Exception {    // ❶
        Attendee attendee = new Attendee();
        attendee.add("백기선");    // ❷
        assertEquals("백기선", attendee.get(1));    // ❸
    }

    @Test
    public void testGet() throws Exception {    // ❹
        Attendee attendee = new Attendee();
        attendee.add("백기선");    // ❺
        assertEquals("백기선", attendee.get(1));    // ❻
    }
}
```

- ❶ 참석자를 추가하는 add 메소드에 대한 테스트 케이스 작성을 시작한다.
- ❷ '백기선'이라는 참가자를 추가한다.
- ❸ 시나리오의 흐름상 참석자가 정상적으로 추가됐는지 확인하기 위해 get 메소드를 호출한다. 그런데 아직 get 메소드는 만들어지지 않은 상태다. get 메소드가 선행 구현되어 있어야 add 메소드 구현을 위한 테스트 케이스 작성이 가능하다는 사실을 알게 된다.

- ④ get 기능을 구현하기 위해 testGet 테스트 메소드 작성을 시작한다.
- ⑤ 테스트 케이스 시나리오상, get 메소드의 기능을 테스트하려면 참가자 추가기능에 해당하는 add 메소드 작성이 선행되어 있어야 한다. 아.. 뭔가 조금 꼬이는 느낌이 다. add를 구현할 수 없어서 만들다 말고 get 먼저 구현하려고 넘어온 건데???
- ⑥ add가 선행 구현되어 있지 않으면 assertEquals 문장이 의미가 없어진다.

위의 두 테스트 케이스는 어느 한쪽만 먼저 구현하기가 어려운 상황을 보여주는 한 예다. 서로가 상호 의존적으로 테스트 케이스 작성에 참여하는 형태다. 좋은 테스트 케이스 작성법 중 하나는 **한 번에 실패하는 테스트 케이스 하나씩** 작성하는 것이라고 앞에서 이야기했었다. 현 상황은, 그 규칙을 지키기 어려운 상태다. 이런 상황에서 택할 수 있는 방법은 보통 세 가지다.

해결책1 실패하는 테스트 케이스가 두 개인 상태에서 작업하기(일반적인 방법)

위와 같은 케이스는 난이도가 낮아서 두 개의 실패하는 테스트를 놓고 작업하기가 어렵지 않다. 하지만 각각 쉽지 않는 난이도의 메소드이고, 작성 도중 실패가 계속된다면, 어느 쪽 문제로 테스트가 실패하는지 파악하기 어려울 수 있다.

해결책2 안정성이 검증된 제3의 모듈을 사용하기(가능만 하다면, 권장)

5장에서 DbUnit을 이용한 경우에서 한 번 설명했었다. 검증은 다른 모듈로 하는 방법이다. 굳이 외부 모듈이 아니더라도, 이미 검증된 다른 메소드를 사용해 검증할 수 있다면 해당 메소드를 이용한다.

해결책3 자바 리플렉션을 이용해 강제로 확인하기(대체적으로 비권장)

자바 리플렉션을 이용한 방식은, 어쩔 수 없는 한정된 상황일 경우에 사용한다. 다음은 위와 같은 상황에서 리플렉션을 사용해 작성한 테스트 케이스다.

```
@Test
public void testAddByReflection() throws Exception {
    Attendee attendee = new Attendee();
    attendee.add("백기선");
}
```

```

Field attendeeList = attendee.getClass().getDeclaredField("attendeeList");
// ❶
attendeeList.setAccessible(true); // ❷

assertEquals("백기선", ((List<String>)attendeeList.get(attendee)).get(0));
// ❸
}

```

- ❶ Attendee 클래스의 필드 중 attendeeList라는 이름의 필드를 강제로 소환한다. 이때 문자열, 즉 “attendeeList”가 필드명이라는 숨겨진 의미를 갖는다. 이런 식의 의미가 내포된 문자열을 리터럴(literal)이라 한다.
- ❷ private으로 되어 있는 필드 변수이지만 강제로 접근 가능하게 만든다.
- ❸ attendee 객체의 해당 필드 변수에 접근해서 값을 가져온 다음 강제로 첫 번째 객체를 추출해낸다. 사전에 이미 List 타입으로 이름이 저장되리라는 정보를 알고 있었기 때문에 가능했다. 추출된 값이 예상값과 일치하는지 확인한다.

테스트 케이스 작성에 리플렉션을 이용하려면 작성하고자 하는 대상 클래스의 내부에 대한 이해나 선행설계가 먼저 되어 있어야 한다. 이 예제에서는, 아래와 같은 식으로 Attendee 클래스가 구현될 것이라고, 미리 가정해서 작성했다.

```

public class Attendee {
    private List<String>attendeeList = new ArrayList<String>();
    ...
}

```

가끔, 작성하기 어려운 테스트 케이스를 리플렉션을 통해 해결했다고 이야기하는 걸 들곤 하는데, 대부분의 경우는 잘못된 접근 방식이다. 리플렉션을 테스트 케이스 작성에 사용하는 일은 정말 부득이한 경우가 아니면 사용하지 않는다. 이를테면 접근제한자(scope)를 변경할 수 없는 private 메소드를 부득이하게 테스트해야 하거나 소스코드 없는 외부 모듈을 받았는데, 신뢰도 검증을 위해 해당 모듈의 내부 메소드를 테스트해야 하는 경우 등이 특별한 예외에 해당한다. 그나마도 곧잘 깨지기 쉬운 테스트 케이스가 돼버리기 때문에, 좋은 테스트 케이스가 되진 않는다.

배열 테스트

해결책1 JUnit 4의 `assertArrayEquals`를 이용한다.

기본적으로는 순서까지 따지기 때문에, 만일 순서는 고려하지 않고 비교할 때는 `Arrays.sort`를 이용한다.

```
@Test
public void testArrayEqual() {
    String[] arrayA = new String[] {"A", "B", "C"};
    String[] arrayB = new String[] {"A", "B", "C"};
    assertArrayEquals("두 배열의 값과 순서가 같아야 함", arrayA, arrayB);
}

@Test
public void testArrayEqual_NotSorted() {
    String[] arrayA = new String[] {"A", "B", "C"};
    String[] arrayB = new String[] {"B", "A", "C"};
    Arrays.sort(arrayA);
    Arrays.sort(arrayB);
    assertArrayEquals("두 배열의 순서는 달라도 무방", arrayA, arrayB);
}
```

해결책2 `Unitils`의 `assertReflectionEquals`나 `assertLenientEquals`를 이용한다.

(상세 내용은 6장 '단위 테스트 지원 라이브러리: 유닛틸즈(Unitils)'를 참조한다.)

```
@Test
public void testArrayEqual_NotSorted() {
    String[] arrayA = new String[] {"A", "B", "C"};
    String[] arrayB = new String[] {"B", "A", "C"};
    assertLenientEquals("두 배열의 순서는 달라도 무방", arrayA, arrayB);
}
```

해결책3 JUnit 3의 경우라면 `List`로 변환해서 비교한다.

필요하다면 미리 정렬을 해도 된다.

```

public void testArrayEqual_NotSorted() {
    String[] arrayA = new String[] {"A", "B", "C"};
    String[] arrayB = new String[] {"A", "B", "C"};

    assertEquals("List로 만들어서 비교", Arrays.asList(arrayA),
        Arrays.asList(arrayB));
}

```

객체 동치성 테스트

객체와 객체를 비교할 때는 정말 동일한 객체인지 판별해내야 하는 **동일성 테스트**인지, 아니면 같은 값을 갖는 객체이지만 판별하면 되는 **동치성 테스트**(equivalent test)인지를 구분해서 생각해야 한다. 대부분의 경우 객체를 비교할 때는 동치성 테스트가 많이 쓰인다. 예를 한번 살펴보자. 노래 제목과 가수를 내부 상태로 갖는 Music 클래스가 있다.

```

public class Music {
    private String songName;
    private String performerName;

    public Music(String songName, String performerName) {
        this.songName = songName;
        this.performerName = performerName;
    }
    ...
}

```

다음은, 두 개의 Music 객체를 assertEquals로 비교하는 테스트 클래스다. 각각의 성공/실패 여부를 미리 따져보자. 둘 중 어느 것은 성공하고, 어느 것은 실패하게 될까?

```

@Test
public void testEquals_case1() {
    Music musicA = new Music("Better in time", "Leona Lewis");
    Music musicB = musicA;
    assertEquals(musicA, musicB);
}

```

```

@Test
public void testEquals_case2() {
    Music musicA = new Music("Better in time", "Leona Lewis");
    Music musicB = new Music("Better in time", "Leona Lewis");
    assertEquals(musicA, musicB);
}

```

testEquals_case1은 성공하고 testEquals_case2는 실패한다. case2의 실패 메시지는 다음과 같다.

```

java.lang.AssertionError: expected:<Music@1cde100> but was:<Music@16f0472>
    at org.junit.Assert.fail(Assert.java:91)
    ...

```

musicA와 musicB, 두 객체는 엄연히 서로 다른 객체이고 다른 ID 값²을 갖기 때문에 테스트가 실패한다. 하지만 하나의 객체를 어떠한 하나의 상태값으로 봤을 때 musicA와 musicB, 두 객체가 서로 다른 상태값을 의미하는가? 그렇지 않다. 노래도 같고 가수도 같은데, 비교했더니 다르다고 나오는 건 때때로 그다지 유쾌하지 않은 상황이 된다. 이렇게 객체의 상태값을 비교하는 것을 동치비교(equivalent comparison)라고 한다. 객체 안에 있는 필드값들을 상태라고 봤을 때, 두 객체의 상태가 모두 일치하면 동치값의 객체로 보는 것이다. 테스트 케이스 작성 시 객체에 대한 expected 값과 actual 값을 비교하거나 정할 때 종종 고민되는 사항이다. 업무적으로는 서로 일치한다고 봐야 하는 경우가 많다.

두 객체를 서로 동치비교하는 방법에는 몇 가지가 있다.

해결책1 내부 상태(보통은 필드값)를 직접 꺼내와서 각각 비교한다.

필드가 많지 않을 경우에 흔히 사용된다. 다만, 필드가 많아지면 비교 구문 작성이 번거로워진다.

```

@Test
public void testEquals_case2() {

```

² 자바는 '클래스이름@해시값'을 사용한다.

```

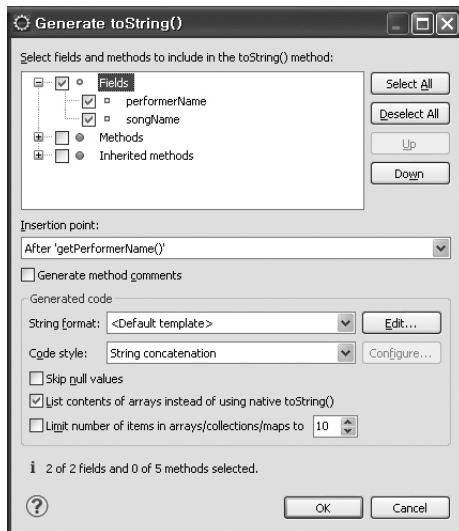
    Music musicA = new Music("Better in time", "Leona Lewis");
    Music musicB = new Music("Better in time", "Leona Lewis");
    assertEquals( musicA.getPerformerName(), musicB.getPerformerName());
    assertEquals( musicA.getSongName(), musicB.getSongName() );
}

```

내부에 있는 두 개의 필드를 각각 꺼내와서 동일한지 비교하고 있다. 테스트는 통과할 것이다.

해결책 2 toString을 중첩구현해(override)놓고, toString 값으로 비교한다.

편하게 비교할 수는 있지만, 만일 이미 toString 구현이 있을 경우 사용할 수 없다. 또한 toString의 목적 자체도 무언가 비교를 위한 것은 아니기 때문에 일종의 트릭이라고 볼 수 있다. 대상 클래스가 확장될 가능성이 적을 경우 사용한다. 이클립스가 toString 메소드 생성을 지원해주기 때문에 작성 자체는 쉽게 할 수 있다. 이클립스 메뉴에서 [Source] → [toString]을 선택한 다음, toString에 사용할 필드를 선택한 다음 OK를 누르면 끝이다.



다음은 이클립스에서 자동 생성한 toString 메소드의 모습이다.

```

public class Music {
    private String songName;
    private String performerName;
    ...
    @Override
    public String toString() {
        return "Music [performerName=" + performerName + ", songName="
            + songName + "];"
    }
}

```

이렇게 이클립스를 이용하면, 필드가 많아도 toString 메소드를 만드는 건 아주 쉽다.

```

@Test
public void testEquals_case2() {
    Music musicA = new Music("Better in time", "Leona Lewis");
    Music musicB = new Music("Better in time", "Leona Lewis");
    assertEquals( musicA.toString(), musicB.toString());
}

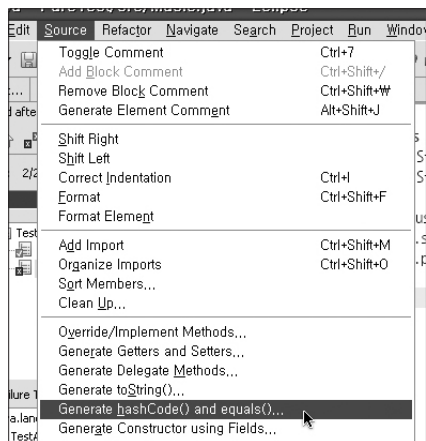
```

마찬가지로 테스트 케이스는 성공한다.

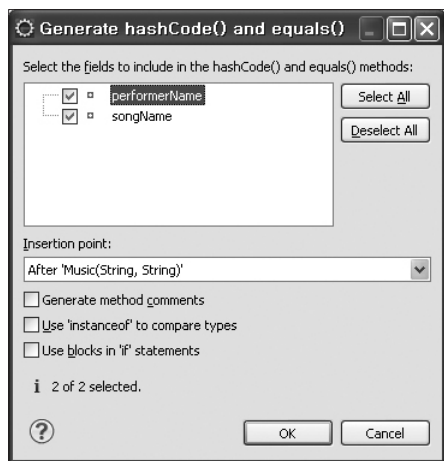
해결책3 equals 메소드를 중첩구현한다.

toString을 이용한 방법보다는 다소 번거롭지만, 개념적으로는 가장 올바른 방법이다. 그리고 Object 클래스로부터 내려오는 equals 메소드의 목적과도 잘 맞는다. 이 예제에서는 Music 클래스에서 equals를 중첩구현하면 된다. JUnit의 assertEquals는 앞에서도 이야기했지만, 이미 중첩되어 있는 기본 타입이 아닐 경우 두 클래스를 equals로 비교하기 때문이다. 직접 equals를 구현할 수도 있지만, 이펙티브 자바(Effective Java)로 유명한 조슈아 블로체(Joshua Bloch)³의 그 유명한 equals 구현을 이클립스에서 자동 생성으로 지원해주니까 이용해보자.

3 조슈아 블로체(Joshua Bloch). 조쉬(Josh)라고 줄여 부른다. 썬(Sun)의 엔지니어였고, Collection을 비롯한 Java 플랫폼의 많은 부분을 직접 설계했다. 물론, Java 5의 신 기능들 추가에도 크게 공헌했고, 새로 나올 Java 7의 여러 기능에도 참여했다. 그가 2001년에 출간한 『Effective Java』라는 책은 아키텍트가 되고자 하는 Java 개발자들의 필독서라 불린다. 개인적으로 2007년에 그 책을 놓고 읽을까 말까 고민했었다. 1년이 멀다 하고 신기술이 나오는데, 과연 6년이나 지난 Java 언어책을 봐야 할까? 하고 말이다. 결국 봤는데, 고민하면서 안 봤던 걸 후회했다. 2008년 말에 Java 5 기반의 2판이 나왔으니 이젠 필자와 같은 고민을 할 필요도 없다. 시간을 내서 꼭 읽어보자. 조슈아는 현재 구글(Google)에서 선임 자바 아키텍트로 일하고 있다.



이클립스 메뉴에서 [Source] → [Generate hashCode() and equals()]를 선택한다.



두 개의 필드를 모두 선택한 다음 OK를 누르면 equals와 hashCode 메소드가 자동으로 생성된다.⁴

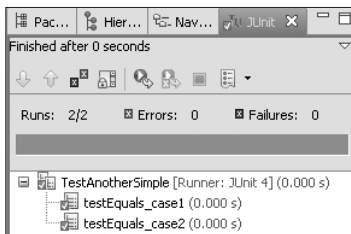
4 equals 구현 코드는 여기에서 직접 다루진 않을 테니, 꼭 한번 수행해보길 바란다. 만일 이펙티브 자바에서 소개된 equals를 본 적이 없다면, 굉장히 당황스런 수준의 equals와 hashCode 메소드가 생성된다. 그렇다고 걱정할 필요는 없다. 전 세계 수많은 엔지니어들이 충분히 잘 작동하는 **적절한 구현**이라는 걸 이미 검증해줬다. 이클립스가 자동으로 만들어주는 equals 메소드는 equals가 가져야 하는 특징인, 재귀성(reflectivity), 대칭성(symmetry), 이행성(transitivity), 추상성(abstract), 일관성(consistency), null 검사 등의 성격을 모두 만족시키는 훌륭한 코드다.

앞에서 작성한 최초의 테스트 케이스를 다시 실행해보자.

```
@Test
public void testEquals_case1() {
    Music musicA = new Music("Better in time", "Leona Lewis");
    Music musicB = musicA;
    assertEquals( musicA, musicB);
}

@Test
public void testEquals_case2() {
    Music musicA = new Music("Better in time", "Leona Lewis");
    Music musicB = new Music("Better in time", "Leona Lewis");
    assertEquals( musicA, musicB);
}
```

다시 테스트 케이스를 수행하면 이번엔 모두 통과할 것이다. 객체 비교가 아닌 동치비교를 했기 때문이다.



두 경우 모두 성공!

참고로, equals 메소드 재정의를 통한 동치비교를 할 때는, 아파치 커먼즈(apache commons)의 EqualsBuilder를 사용하는 것도 한 가지 방법이다.

EqualsBuilder를 이용한 equals 메소드 구현

```
import org.apache.commons.lang.builder.EqualsBuilder;

public class Music {
    private String songName;
```

```

private String performerName;
...
@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj);
}

```

이 외에도 아파치 커먼즈의 builder 패키지에는 비교 시에 유용하게 사용할 수 있는 좋은 유틸리티가 많이 있다. 수레바퀴를 다시 발명하려 하지 말고 이용할 수 있는 것들은 최대한 이용하자.⁵

org.apache.commons.lang.builder 패키지의 클래스

```

CompareToBuilder
EqualsBuilder
HashCodeBuilder
ReflectionToStringBuilder
StandardToStringStyle
ToStringBuilder
ToStringStyle

```

해결책 4 Unitils의 assertReflectionEquals를 이용한다.

Unitils를 사용하는 프로젝트라면 당연 권장하는 방식이다. 내용 자체는 Unitils를 소개하는 부분에서 이미 다뤘던 내용이다. 예제만 다시 한번 보자.

```

import static org.unitils.reflectionassert.ReflectionAssert.*;
...
@Test
public void testEquals_case2() {

```

5 일련의 연구원들이 개발 생산성을 높이는 방법에 대해 다각도로 연구를 한 적이 있었다. 그중 ‘가장 저렴한 비용으로 우리가 원하는 소프트웨어를 만드는 방법’에 대한 최종 연구 결과는 다음과 같았다. “시장에서 사온다.”


```

        Music musicA = new Music("Better in time", "Leona Lewis");
        Music musicB = new Music("Better in time", "Leona Lewis");
        assertEquals( musicA, musicB);
    }

```

마찬가지로 테스트를 통과한다.

7-1-1
자바
테스트

테스트 수행을 위한 assertEquals 메소드 재정의

단위 테스트 케이스를 수행할 때 객체를 비교하기 위해, equals 메소드를 재정의하는 것은 여러 가지 측면에서 좋은 방법이다. 하지만 만들려면 확실하게 만들어줘야 한다. 만일 equals 메소드를 대충 작성한다면, 향후 문제가 발생할 수 있다. 조금은 부담이 되는 부분이다. 그리고 그렇다 하더라도, 경우에 따라서는 equals나 toString 같은 메소드를 구현하지 않은 서드파티 모듈의 객체를 비교해야 할 일이 생길 수도 있다. 이럴 때는 equals나 toString 재정의 대신, 비교 대상 클래스를 지원하는 assertEquals를 정의하는 것도 하나의 해결책이 될 수 있다.

```

private boolean assertEquals(Car expected, Car actual){
    assertEquals(expected.getModelNo(), actual.getModelNo());
    assertEquals(expected.getBrandAlias(), actual.getBrandAlias());
    assertEquals(expected.getDoorType(), actual.getDoorType());
    assertEquals(expected.getPrice(), actual.getPrice());
}

```

자주 쓰는 타입들에 대해 작성해놓은 다음, 유틸리티처럼 사용하면 된다. 항상 그렇듯, 문제는 그때그때 여러 가지 방향으로 풀 수도 있다는 점을 명심하자.

컬렉션 테스트

객체를 담을 수 있는 자바의 컬렉션(collection)에는 종류가 많이 있지만, 대표적으로 많이 쓰는 것으로 List, Set, Queue가 있다. 한 부모 아래 자식들이라 사용법이 대동소이하다. 그중 List를 예제로 사용해서 컬렉션 객체 비교를 어떻게 할 것인지 살펴보자.

java.util

Interface Collection<E>

All Superinterfaces:

[Iterable<E>](#)

All Known Subinterfaces:

[BeanContext](#), [BeanContextServices](#), [BlockingDeque<E>](#),
[BlockingQueue<E>](#), [Deque<E>](#), [List<E>](#), [NavigableSet<E>](#), [Queue<E>](#),
[Set<E>](#), [SortedSet<E>](#)

All Known Implementing Classes:

[AbstractCollection](#), [AbstractList](#), [AbstractQueue](#), [AbstractSequentialList](#),
[AbstractSet](#), [ArrayBlockingQueue](#), [ArrayDeque](#), [ArrayList](#), [AttributeList](#),
[BeanContextServicesSupport](#), [BeanContextSupport](#),
[ConcurrentLinkedQueue](#), [ConcurrentSkipListSet](#), [CopyOnWriteArrayList](#),
[CopyOnWriteArraySet](#), [DelayQueue](#), [EnumSet](#), [HashSet](#), [JobStateReasons](#),
[LinkedBlockingDeque](#), [LinkedBlockingQueue](#), [LinkedHashSet](#), [LinkedList](#),
[PriorityBlockingQueue](#), [PriorityQueue](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#),
[SynchronousQueue](#), [TreeSet](#), [Vector](#)

Collection의 하위 인터페이스와 구현 클래스들. Java 6 기준

Case 1 자바 기본형(primitive type)이나 String이 컬렉션에 들어 있는 경우

이런 경우엔 곧바로 비교가 가능하다. 기본적으로 컬렉션들은 안에 담겨 있는 객체들을
열거 형태로 꺼내서 순차적 equals를 하게 되어 있다.

```
@Test
public void testListEqual_Primitive() {
    List<String> listA = new ArrayList<String>();
    listA.add("변정훈");
    listA.add("조연희");

    List<String> listB = new ArrayList<String>();
    listB.add("변정훈");
    listB.add("조연희");

    assertEquals("리스트 비교", listA, listB);
}
```

테스트를 정상 통과한다.

Case 2 일반 객체가 컬렉션에 들어 있는 경우

assertEquals는 기본적으로 기대값과 실제값을 서로 equals 비교한다. 그리고 컬렉션은

equals 비교 시에 원소를 하나씩 꺼내서 다시 각각에 대해 equals 비교를 한다.⁶ 그런데 담긴 객체가 이미 equals를 중첩구현해 놓았다면 문제가 없지만, 아니라면 제대로 비교가 안된다. 이때는 toString 메소드를 이용해서 문자열로 비교할 수 있게 하자.

```
@Test
public void testListEqual_NotSorted() {
    List<Employee> listA = new ArrayList<Employee>();
    listA.add( new Employee("변정훈"));
    listA.add( new Employee("조연희"));

    List<Employee> listB = new ArrayList<Employee>();
    listB.add( new Employee("변정훈"));
    listB.add( new Employee("조연희"));

    assertEquals("리스트 비교", listA, listB);
}

-----실행 결과-----
java.lang.AssertionError: 리스트 비교 expected:<[Employee@6b97fd,
Employee@1c78e57]> but was:<[Employee@5224ee, Employee@f6a746]>
```

만일 위 예제처럼, 두 개의 List 객체를 바로 비교한다면 List 안에 들어간 객체의 Object ID에 해당하는 toString 값으로 비교한다.⁷ 실행 결과로 출력된 ID 값을 보면, 네 개의 객체가 모든 다른 객체라는 사실을 알 수 있다. 객체 ID가 직접 찍히지 않도록, Employee 클래스에 toString을 오버라이드해보자.

```
public class Employee {
    private String name;

    public Employee(String name) {
        this.name = name;
    }
}
```

6 toString 비교를 하지 않고, 직접 등호 비교가 일어나게 하는 방법이 기억나는가? 아니라면, JUnit 절을 다시 살펴보자.

7 컬렉션들의 상위 클래스에 해당하는 AbstractCollection 클래스의 toString 메소드는 내부 원소들의 toString 값을 () 안에 나열하도록 구현되어 있다.

```

@Override
public String toString() {
    return this.name;
}
}

```

다시 테스트 케이스를 실행해보자.

-----실행 결과-----

```

java.lang.AssertionError: 리스트 비교
    expected: java.util.ArrayList<[변정훈, 조연희]> but was:
    java.util.ArrayList<[변정훈, 조연희]>
    at org.junit.Assert.fail(Assert.java:91)
    at org.junit.Assert.failNotEquals(Assert.java:618)
    ...

```

응? 기대값(Expected) [변정훈, 조연희]와 실제값(but was) [변정훈, 조연희], 두 값이 동일한 것 같은데, 왜 다르다고 하는 걸까? 이유는 Array 클래스의 equals가 사용돼서 listA와 listB의 객체 ID 값이 서로 다르다고 인식하기 때문이다. 따라서 객체를 비교하는 것이 아닌, 문자열만 비교할 수 있도록 listA와 listB를 각각 toString으로 변경해서 비교하자.

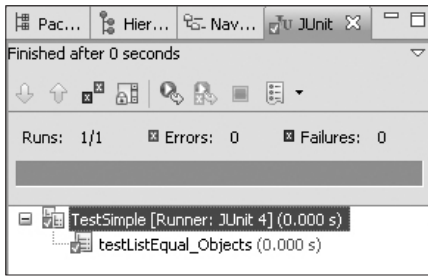
```

@Test
public void testListEqual_Objects() {
    List<Employee> listA = new ArrayList<Employee>();
    listA.add(new Employee("변정훈"));
    listA.add(new Employee("조연희"));

    List<Employee> listB = new ArrayList<Employee>();
    listB.add(new Employee("변정훈"));
    listB.add(new Employee("조연희"));

    assertEquals("리스트 비교", listA.toString(), listB.toString());
}

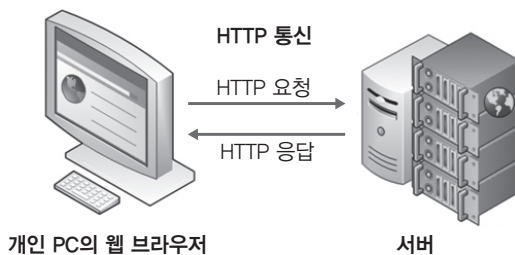
```



녹색 막대가 나왔다!! 성공!!

7.2 웹 애플리케이션

불과 수년 사이에 세상 사람들이 사용하는 애플리케이션의 상당수가 웹 애플리케이션으로 바뀌었다. 이제 일반인들 사이에서는 PC와 웹(www)은 거의 동일 개념으로 사용된다. 당장 우리 집만 해도, 'PC가 고장났다'는 말과 '인터넷이 안 된다'는 말은 같은 의미로 쓰인다. 그만큼 웹과 웹 애플리케이션은 현대 애플리케이션 개발의 중심에 있다. 웹 애플리케이션을 작성할 때, TDD를 어떻게 사용할 것인가에 대해 고민하기 전에 우선 '웹 애플리케이션은 무엇인가'부터 살펴볼 필요가 있다. (길게는 안 볼 테니 걱정은 마시길!)



웹 애플리케이션의 기반구성

웹 애플리케이션의 기본 흐름은 사실 지나칠 정도로 간단하다. HTTP(Hypertext Transfer Protocol) 기반에서의 요청(request)/응답(response)이 전부다. 사용자는

화면의 브라우저에서 무언가를 요청하면 서버는 적절한 응답을 한다. 기반이 HTTP(하이퍼텍스트 전송 규약)라고 했으니 개인 PC와 서버가 서로 주고받는 것도 하이퍼텍스트가 전부다. 말은 간단한데, 자세히 살펴보지 않으면 잘 드러나지 않는 여러 개발 영역이 그 사이에 자리잡고 있다. 따라서 ‘웹 애플리케이션을 개발한다! 그리고 TDD로 개발하겠다!’라고 마음을 먹는다면 어떤 개발 영역이 있는지 살펴보고, 영역별로 접근 전략을 잘 세워야 한다.

■ 개인 PC의 웹 브라우저

웹 브라우저는 서버로 요청하는 부분과 응답을 표현하는 부분을 갖는다. 표현은 HTTP로 넘어온 하이퍼텍스트로 이뤄진다. 사용자는 웹 브라우저 위에 시각(view)적으로 표현된 부분을 이용한다.

■ 서버

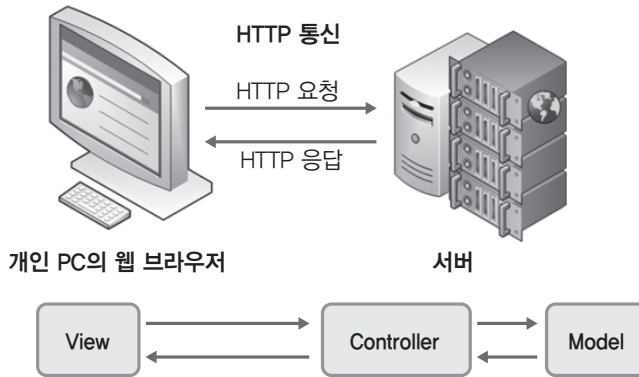
웹을 지탱하는 기반시설 중 하나다. 사용자의 요청을 해석(control)해서 적절한 동작을 취한 다음, 다시 사용자에게로 응답을 보낸다. 사용자에게 적절한 반응을 보이기 위한 업무로직(business logic)이 동작하고, 그때 사용하게 되는 데이터(domain data)가 보관되어 있다.

■ 웹 애플리케이션의 구조

개인 PC와 서버 사이에서 동작하는 웹 애플리케이션은 앞에서 이야기한 시각적인 표현요소, 요청의 해석, 업무로직 처리, 데이터 관리 등의 작업을 한다. 사람들은 이런 요소들을 한곳에서 처리하려면 복잡하고 어렵다는 사실을 알게 됐다. 그래서 관심의 분리(separation of concerns)를 적용해서 여러 가지 아키텍처를 구상해냈는데, 그중 현재 가장 널리 일반적으로 받아들여지는 구조가 MVC 아키텍처다.

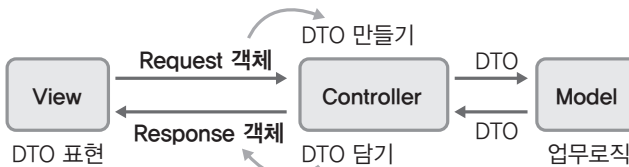
7.2.1 MVC 아키텍처

MVC 아키텍처는 앞서 말한 웹 애플리케이션을 모델-뷰-컨트롤러(Model-View-Controller)라는 세 개의 영역으로 나누어 구성하는 것을 지칭한다.



MVC 아키텍처로 본 일반적인 웹 애플리케이션의 논리적인 구조

시각적인 요소를 갖고 있는 웹 애플리케이션의 표현 영역을 뷰라 부르고, 업무로직과 데이터를 묶어서 모델이라고 부른다. 그리고 뷰와 모델을 중간에서 중재해주는 역할을 컨트롤러가 책임진다. 뷰는 컨트롤러하고만 대화하고, 모델도 마찬가지로 컨트롤러하고만 대화하는 게 원칙이다. 이때 대화의 매개체로 DTO가 종종 사용된다.



DTO를 사용한 MVC 아키텍처

대표적인 MVC의 장점은 다음과 같다.

- 표현(View)과 로직(Model)의 느슨한 결합(loose coupling)
- 관심의 분리
- 테스트 주도 개발 작성이 좀 더 용이함
- 각 영역에 대한 재사용성이 높아짐

결과적으로 변화에 대한 적응성이 높아지기 때문에, MVC 아키텍처를 많이 사용한다.

뷰

뷰(view)는 웹 애플리케이션의 사용자와 직접적으로 대화가 이뤄지는 부분이다. 웹 애플리케이션 측면에서는 HTML, JSP, PHP, ASP 등에 해당한다. 뷰는 사용자의 동작과 연관되어 있는 부분이 많고, 비교적 자주 변경되는 부분이라 웹 애플리케이션 개발에서 많은 시간이 소요되는 영역이다. 때문에 뷰를 TDD로 개발하려면 쉽지 않다. 그리고 뷰에는 다양한 요소들이 혼재되어 있다. 그럼에도 보통은 이를 분리해서 생각하지 않고 뷰라는 한 가지 개념으로만 접근하기 때문에 더욱 어렵게 느껴지기 쉽다. 다음은 구성요소에 따라 뷰를 분류해본 표다. 각 분류별로 TDD 적용과 적용의 효율성에 대한 의견을 함께 적었다.

구성요소	주요 사용	TDD 적용	TDD 효율
UI (User Interface)	<ul style="list-style-type: none"> • 심미적인 부분이나 사용 편의성 • 이미지나 테두리, 글자모양 등이 이에 해당한다. 	TDD 적용이 불가능하거나 TDD 적용의 가치를 찾기 어렵다.	매우 낮음
페이지 구성요소 (Page Elements)	<ul style="list-style-type: none"> • 각종 정보나 사용자의 의도가 담기는 화면요소 • 화면상의 텍스트, 아이디/비밀번호 입력, 검색, 게시판 글 쓸 때 사용하는 영역, 각종 버튼 등 	화면 설계 시 필요한 요소를 미리 정해놓고 다음과 같은 조건으로 판단한다. <ul style="list-style-type: none"> - 화면 내에 해당 요소가 존재하는가? - 해당 요소 내에 들어 있는 값이 예상과 같은가? 	보통
이벤트 (Event)	<ul style="list-style-type: none"> • 동작(action)은 일어나지만 사용자가 개입하지 않는 것 • 유효성 검사나 커서 이동, 경고창 팝업 등이 이에 해당한다. 웹 애플리케이션에서는 스크립트가 흔히 사용되며, 일반적으로 서버 쪽으로의 요청은 발생하지 않는다. 	보통 웹 애플리케이션에서는 Ajax를 제외한 뷰 영역의 이벤트를 컨트롤러에서 직접 제어하지 않기 때문에, 자바 언어 레벨에서는 TDD 적용이 힘들다. 자바스크립트용 xUnit 라이브러리인 JsUnit 등을 사용하면 테스트 가능하다.	낮음 페이지 구성요소, 특히 HTML 폼 필드들과의 커플링이 높기 때문에 변경의 요인이 크다. JsUnit을 사용해서 개발할 때조차도 쉽지 않다.

사용자 액션 (User Action)	<ul style="list-style-type: none"> • 사용자의 의도가 반영되는 요청 • 서버 쪽으로 요청이 발생한다. 	액션의 요청이 제대로 설정됐는지 확인할 수 있어야 하는데, 이견 뷰 영역에서 테스트하기 어렵다. 다만, 사용자 테스트 측면으로 접근해서 특정 화면에서 사용자가 특정 액션을 요청했을 때 결과 화면이 정상인지를 테스트하는 방식은 가능하다.	보통
---------------------------------------	--	---	----

뷰에 대한 TDD 적용은 매우 까다롭고, 유지보수 시 변경 작업도 잦은 편이라 ROI가 잘 나오는 작업은 절대 아니다. 굳이 TDD를 적용한다면, 보통은 HTML 폼 필드들과 사용자 액션 위주로 작성하게 된다. 웹 페이지 내의 이벤트는 자바스크립트가 흔히 사용되는데, 이때는 JUnit 같은 자바스크립트 기반의 테스트 라이브러리를 사용한다. 하지만 JUnit을 사용했을 때조차도 자바스크립트가 함수적인 기능, 예를 들면 글자를 30자마다 잘라서 다음 줄로 넘긴다든가 하는 식의 기능이 아닌, 폼 구성요소와 결합되어 있는 이벤트라면 TDD를 위한 테스트 케이스 작성이 효과가 적거나 거의 없다. 어찌 됐든, 뷰를 어떻게 테스트할 것인가에 대한 몇 가지 시도는 잠시 뒤, 7.2.2절 ‘뷰 TDD’에서 좀 더 자세히 살펴볼 예정이다.



집을 만들기 전에 테스트 케이스를 먼저 작성한다면?

개인적으로는 웹 애플리케이션에 대한 테스트 케이스를 작성하는 일은, 집을 짓기 전에 제대로 된 시공 여부를 어떻게 판단할 것인지를 따져보는 일과 비슷하다는 생각이 든다.

- 유리창 → 원하는 방향에 존재하는가? (상태 테스트, No Action)
- 방문 → 정상적으로 열고 닫히는가? (행동 테스트, User Action)
- 화재경보 → 연기가 나거나 열이 발생하면 경보가 울리는가? (이벤트 테스트, System Action)

웹 애플리케이션의 사용자 측면에서의 테스트, 즉 뷰 영역에서의 테스트 케이스와 비슷하다는 생각이 들지 않는가?

컨트롤러

컨트롤러(controller)는 모델과 뷰를 분리하기 위해 사용되는 중간 층이다. 뷰로부터 넘어온 요청에서 데이터 모델을 발췌해서 적절한 모델 쪽으로 넘겨주고, 모델로부터 받은 응답을 다시 뷰로 돌려준다. 뷰 입장에서는 컨트롤러만 알면 되고, 모델 입장에서도 컨트롤러만 알면 된다. 모델과 뷰를 재사용할 수 있게 해준다는 부가적인 장점도만 들어준다. 앞에서 뷰에는 TDD를 적용하기가 어렵다고 말했는데, 그럼 컨트롤러 작성 시 TDD를 적용하는 것은 어떨까? 결론부터 이야기하자면, 컨트롤러에 대한 TDD 작성은 비교적 수월하다. 요청으로부터 적절한 데이터를 발췌해내고, 해당 데이터를 모델로 넘긴다. 다시, 그 결과로 모델로부터 받은 데이터를 응답에 담아서 뷰로 넘기는 구조가 가장 기본적인 것이다. 요즘은 웹 애플리케이션 개발을 할 때 소위 말하는 프레임워크 없는 날코딩을 하는 경우가 매우 드물기 때문에, 컨트롤러를 직접 만들 일은 없다. 하지만 기본 개념을 알아두면 다양하게 응용할 수 있다. 전통적인 MVC 모델 방식의 서블릿(Servlet)을 컨트롤러로 사용한 경우의 예는 잠시 뒤 7.2.3절 ‘컨트롤러 TDD’에서 다루기로 한다.

모델

MVC에서는 모델(model)과 뷰를 완전히 분리하는 방식으로 작성할 것을 권장한다. 모델이 통신하는 컨트롤러와의 관계도 사실 자세히 살펴보면 모델이 컨트롤러를 호출하는 경우는 없다. 모델은 오로지 무언가의 호출에 응답할 뿐이고, MVC 모델의 웹 애플리케이션에서는 컨트롤러가 모델을 호출하는 역할을 맡고 있을 뿐이다. 혹자는 이런 방식을 객체 지향 원칙 중 하나인 헐리우드 법칙(Don't ask me, I'll tell you)⁸이라고 부르기도 한다. 모델에 대한 TDD는 앞에서 계속 이야기했던, 일반적인 애플리케이션을 작성하던 방식대로 TDD를 진행하면 된다. 웹 애플리케이션 구성요소 중 가장 TDD가 쉽게 적용되는 부분이고, 또 적용해야 하는 부분이다.

8 헐리우드에서는 수많은 배우들이 연출자에게 시시때때로 자신을 오디션 봐달라고 요청한다고 한다. 그럴 때 기획자가 흔히 사용하는 대사가 “명함을 놓고 가세요. 제가 연락드릴게요(Don't ask me, I'll tell you)”라고 한다. 그럼 배우는 집으로 돌아가서 연출자에게 연락이 올 때까지 기다린다. 물론 언제 연락이 올지는 모르지만 말이다. 이처럼 객체 지향 설계에서도 모듈을 만들 때, 언제 누가 자신을 호출하게 될지는 알 수 없지만 호출되면 제대로 동작할 수 있게 모듈을 구성해놓을 것을 권장한다. 의존성이 낮아지기 때문이다. 그리고 그렇게 구성하는 것을 앞의 상황에 빗대어 헐리우드 법칙(Hollywood Principle)이라고 부른다.

7.2.2 뷰 TDD

MVC 모델에 대한 TDD 적용 중 뷰 영역에 대한 접근을 살펴보자. 대표적으로 많이 사용되는 방법 위주로, 어떻게 접근이 가능하고, 어느 정도 효율이 있는지 함께 살펴보자.

첫 번째 시도, HttpUnit⁹

HttpUnit

Home

- Home
- License
- Download 1.7
- Cookbook
- Tutorial
- JavaDoc
- FAQ
- External Citations
- JavaScript Support
- ServletUnit Intro
- Developers
- Subversion Repository
- Mailing List
- Bug Database
- Feature Requests

Automated testing is a great way to ensure that code being maintained works. The *Extreme Programming* (XP) methodology relies heavily on it, and practitioners have available to them a range of testing frameworks, most of which work by making direct calls to the code being tested. But what if you want to test a web application? Or what if you simply want to use a web-site as part of a distributed application?

In either case, you need to be able to bypass the browser and access your site from a program. HttpUnit makes this easy. Written in Java, HttpUnit emulates the relevant portions of browser behavior, including form submission, JavaScript, basic http authentication, cookies and automatic page redirection, and allows Java test code to examine returned pages either as text, an XML DOM, or containers of forms, tables, and links. When combined with a framework such as JUnit, it is fairly easy to write tests that very quickly verify the functioning of a web site.

The same techniques used to test web sites can be used to test and develop servlets without a servlet container using ServletUnit, included in the download.

HttpUnit은 일종의 가상 웹 브라우저를 이용해 웹 페이지의 요소들을 테스트하기 위해 사용하는 프레임워크다. 특정 웹 페이지를 지정해서 페이지 내의 각종 요소들, 이를테면 일반적인 텍스트(plain text), 폼(form)이나 링크(link), 프레임(frame) 등에 접근해서 상태를 값으로 불러올 수 있게 해준다. 다음은 HttpUnit을 이용한 전형적인 페이지 구성요소에 대한 테스트 케이스다.

```
@Test
public void connectTest() throws Exception {
    WebConversation wc = new WebConversation();
    WebResponse response = wc.getResponse("http://www.google.com/");
    assertNotNull("No response received", response);
}
```

⁹ <http://httpunit.sourceforge.net/>

```

    assertEquals("content type", "text/html", response.getContentType());
    assertTrue("메시지 출력 확인", response.getText(),
        containsString("google"));
}

```

HttpUnit을 사용하면, 보통 위와 같은 식으로 테스트 케이스를 작성한다. 응답(response)을 직접 해석하거나 Form이나 Table 같은 페이지 구성요소를 나타내는 객체로 변환해서 비교한다.

페이지 내의 특정 요소를 객체로 만들어서 예상값과의 일치 여부를 판단하는 HttpUnit 테스트 코드

```

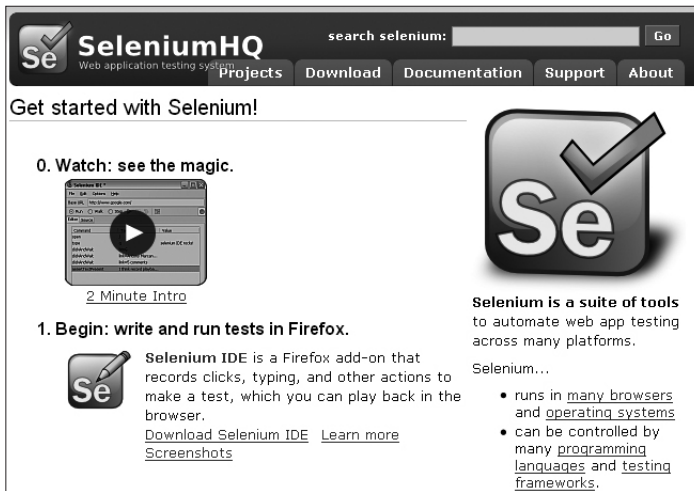
WebTable table = resp.getTables()[0];
assertEquals( "rows", 4, table.getRowCount() );
assertEquals( "columns", 3, table.getColumnCount() );
assertEquals( "links", 1, table.getTableCell( 0, 2 ).getLinks().length );

```

그런데, 미안한 이야기지만 현실적으로 HttpUnit을 통한 TDD 웹 개발은 거의 불가능에 가깝다. 학습용이거나 실험적인 부분에서는 가능하겠지만, 일반적인 상용 웹 애플리케이션을 개발하기엔 거쳐 가야 하는 길이 너무 멀고 험하다.¹⁰ 그리고 무엇보다 TDD 적용에 대한 ROI가 제대로 나오지 않는다. 만일 누군가가 HttpUnit으로 TDD를 적용해 개발하라고 권한다면, 다시 한번 잘 생각해볼길 바란다. 자바스크립트 없는 순수 HTML 환경이 아니면, 제대로 동작하지 않을 것이다. 다만, 서블릿 같은 요청/응답 테스트 작성에는 활용할 만하다.

10 조금 복잡한 자바스크립트가 페이지 안에 들어가기 시작하면 다양한 에러를 접하게 될 것이다.

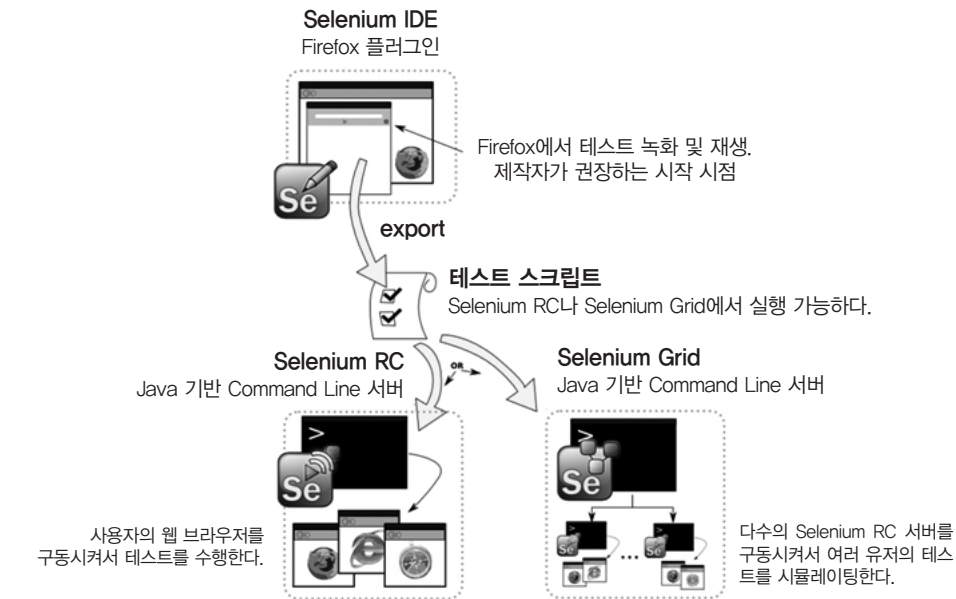
두 번째 시도, Selenium¹¹



오픈소스 웹 UI 테스트 분야에서는 openQA의 Selenium(셀레늄)이 가장 유명하다. Selenium은 크게 세 가지 형태로 테스트를 지원해준다.

- Selenium IDE: 브라우저와 통합돼서 녹화/실행/확인(Record & Run & Verify) 기능을 제공한다. 녹화된 스크립트는 JAVA, PHP, Python, Ruby, C# 등 대부분의 프로그래밍 언어로 내보낼(export) 수 있다. 현재는 파이어폭스 브라우저만 지원한다.
- Selenium RC(Remote Control): 브라우저를 원격으로 조종해서 웹 페이지에 대한 테스트를 수행한다. 이때 직접 스크립트를 작성할 수도 있고, Selenium IDE에서 녹화된 스크립트를 이용할 수도 있다.
- Selenium Grid: 여러 개의 Selenium RC 서버를 구동해서 테스트 수행의 스케일업(scale-up)을 지원한다. 부하 테스트, 경합 테스트 등에 사용할 수 있다.

¹¹ <http://seleniumhq.org/>



Selenium의 구조도

이 중에서 TDD로 뷰를 작성하기 위해 사용할 수 있는 부분은, 직접 스크립트 작성이 가능한 Selenium RC이다.

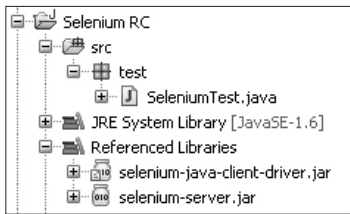
Selenium RC를 이용한 뷰 영역의 TDD를 진행해보자. 작성에 앞서, 테스트 케이스의 시나리오를 먼저 정하면 이렇다.

- ❶ 특정 사이트(지금은 www.google.com)에 접속한다.
- ❷ 검색창에 '펭귄너구리'라고 입력한다.
- ❸ 결과 페이지에 '툼과 제리 #1'이라는 문구가 있으면 테스트 성공이다.

이 시나리오를 자바로 작성한 소스는 다음과 같다.



구글의 기본 검색창. 검색 input 폼의 이름은 알파벳 q이고, 검색버튼(Google Search)의 이름은 'btnG'이다.



참조 라이브러리

시나리오에 맞춰 작성한 테스트 케이스

```
import com.thoughtworks.selenium.*;

public class SeleniumTest extends SeleneseTestCase {    // ❶

    public void setUp() throws Exception {
        setUp("http://www.google.com/", "*firefox");    // ❷
    }

    public void testSearch() throws Exception {
        selenium.open("/");    // ❸
        selenium.type("q", "펭귄너구리");    // ❹
        selenium.click("btnG");    // ❺
        selenium.waitForPageToLoad("30000");    // ❻
    }
}
```

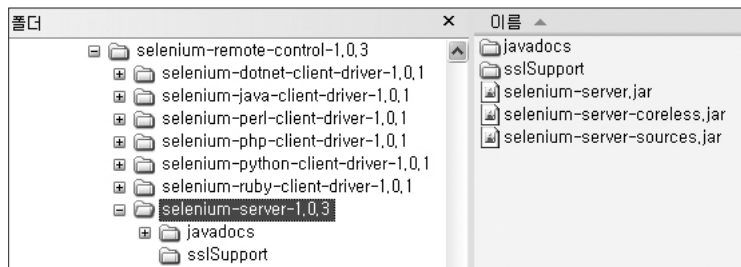
```

        assertTrue(selenium.isTextPresent("툼과 제리 #1"));    // ❶
    }
}

```

- ❶ SeleneseTestCase를 상속받는다. JUnit 3 기반으로 작성된 Test Runner이다.
- ❷ 접속할 웹 페이지의 URL과 이때 사용할 웹 브라우저를 지정한다. 참고로 MS IE로 바꾸고 싶으면 *iexplore라고 넣는다. 지원 브라우저의 종류를 알고 싶은 경우, 영어 이름 넣어서 실행하면 친절히 알려준다.
- ❸ 페이지 경로를 적는다.
- ❹ form type의 이름과 그 안에 넣을 값을 지정한다.
- ❺ btnG 이름의 버튼을 클릭한다.
- ❻ 최대 30초까지 페이지 로딩 완료를 기다린다. 넘어서면 실패로 간주한다.
- ❼ 현재 '툼과 제리 #1'이라는 글자가 화면에 존재하는지 확인한다.

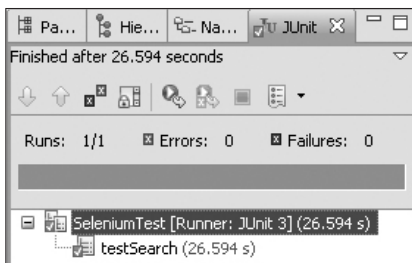
HttpUnit에서 웹 브라우저를 에뮬레이션하던 방식의 코드와 비슷한데, Selenium은 테스트 수행 시에 에뮬레이션으로 동작하는 것이 아니라 웹 브라우저를 실제로 기동시킨다. 그 다음에 웹 브라우저를 조종해서 테스트를 수행하기 때문에 좀 더 실제와 같은 테스트가 가능하고, 브라우저의 동작 모습을 개발자의 눈으로 직접 확인할 수 있다. 주의할 점은, 테스트 케이스를 실행하기 전에 우선 Selenium RC 서버를 구동시켜 놓아야 한다는 것이다. 다운로드 받은 Selenium RC 압축파일 안에 서버 모듈도 들어 있다.



실행은 명령행을 이용한다.¹²

```
D:\selenium-server-1.0.3>java -jar selenium-server.jar
19:53:19.125 INFO - Java: Sun Microsystems Inc. 11.0-b15
19:53:19.125 INFO - OS: Windows XP 5.1 x86
19:53:19.140 INFO - v2.0 [a2], with Core v2.0 [a2]
19:53:19.468 INFO - RemoteWebDriver instances should connect to:
http://127.0.0.1:4444/wd/hub
19:53:19.468 INFO - Version Jetty/5.1.x
19:53:19.468 INFO - Started HttpContext[/selenium-server/driver,/selenium-
server/driver]
19:53:19.468 INFO - Started HttpContext[/selenium-server,/selenium-server]
19:53:19.468 INFO - Started HttpContext[/,/]
19:53:19.562 INFO - Started org.openqa.jetty.jetty.servlet.
ServletHandler@1a16869
19:53:19.562 INFO - Started HttpContext[/wd,/wd]
19:53:19.578 INFO - Started SocketListener on 0.0.0.0:4444
19:53:19.578 INFO - Started org.openqa.jetty.jetty.Server@1c184f4
```

이제 테스트 케이스를 실행해보자. 이클립스의 Run 버튼을 누르면 개발자 PC의 웹 브라우저가 실행된 다음 마치 사용자가 직접 웹 서핑을 하는 것처럼 테스트 케이스 실행이 진행된다. 실제로 처음 보면 웹 브라우저가 혼자 움직이는 모습이 신기하게 보일 수 있다.



테스트 케이스가 성공한다.

¹² 이클립스에서 Run 설정으로 만들어도 무방하다. 이클립스에서 Run 설정하는 방법은 부록 A.4절 '3분 안에 Java DB 설치하고 확인까지 마치고'를 참조하라.

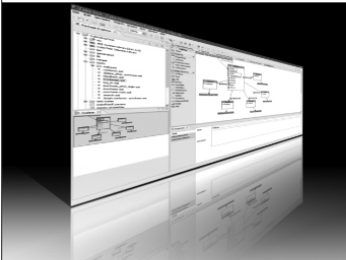
비록 지금은 미리 만들어져 있는 페이지에 대해 테스트를 했지만, 정상적으로 뷰 페이지가 작성된 다음에야 녹색 막대가 보일 테니, TDD 식 개발이 가능하다. 참고로, Selenium은 기본적으로 JUnit 3 기반으로 동작한다. 만일 JUnit 4 버전을 함께 사용하고 싶다면, 오픈소스 프로젝트인 selenium4junit¹³을 함께 사용한다.

자, 그럼 두 번째 시도로, 뷰에 대한 테스트는 그럼 끝인 건가? 아쉽지만, Selenium RC를 이용하는 방법도 최선의 답은 아니다. Selenium을 이용한 방식으로 뷰를 개발하는 것도 사실 따지고 보면 ROI가 높은 방식은 아니다. 화면의 요소를 미리 자바 코드로 짜고, 변경에 맞춰서 해당 소스를 유지한다는 게 생각보다 쉽진 않다. 좀 더 간단한 방법이 필요하다. 뭔가 좀 더 GUI 방식으로 만들 수는 없을까?

세 번째 시도, CubicTest¹⁴

CubicTest: CubicTest


WELCOME TO FUNCTIONAL WEB TESTING WITH CUBICTEST!




CubicTest is a graphical Eclipse plug-in for writing Selenium and Watir tests. It makes web tests faster and easier to write, and provides abstractions to make tests more robust and reusable.

CubicTest's test editor is centered around pages/states and transitions between these pages/states. The model is intuitive for both Ajax and traditional web applications and supports most user interaction types.


CubicTest features an innovative test recorder and test runner based on Selenium RC which are fully integrated with the graphical test editor. Tests can also run standalone from Maven 2.



Page/state concept
A unique page/state concept that lets you model your application as a graph of states.



Recorder
Record your tests directly from Firefox into the editor.

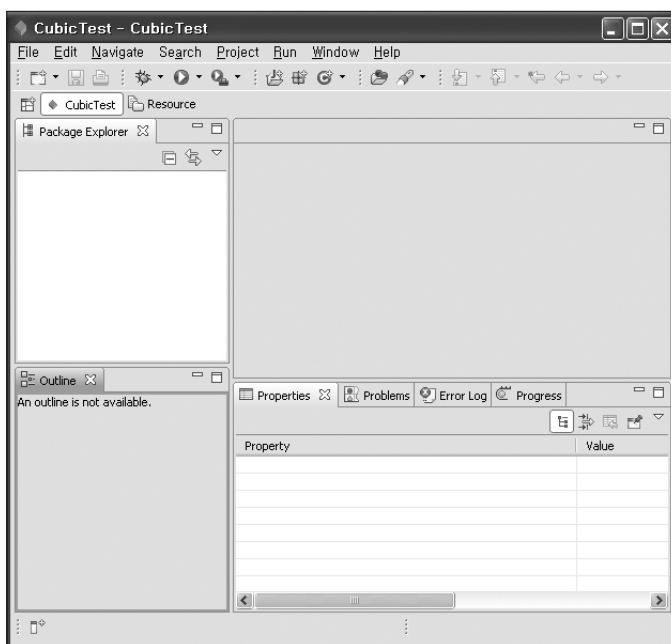


Runner
Run your tests instantly, using your favorite browser. No instrumenting of target application necessary.

13 <http://code.google.com/p/selenium4junit/>

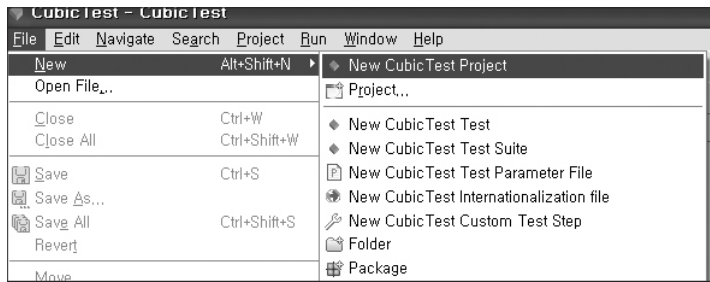
14 <http://cubictest.seleniumhq.org/> CubicTest를 쓰려면 좋은 사양의 PC가 필요하다.

CubicTest도 Selenium과 마찬가지로 openQA에서 지원하고 있는 웹 애플리케이션 테스트 프로젝트다. 웹 애플리케이션의 기능을 테스트하는 부분에 기능이 집중되어 있다. 기존 Selenium RC가 프로그램 언어로 작성된 스크립트 기반으로 동작한다고 하면, CubicTest는 GUI 기반에서 순서도를 작성하듯 웹 애플리케이션에 대한 기능 테스트를 작성할 수 있다. 이클립스 플러그인으로 동작한다. 다운로드 페이지에서 이클립스 RCP로 작성되어 있는 독립실행(stand-alone) 버전도 내려받을 수 있다. 내부적으로는 Selenium RC가 사용되고 있다.

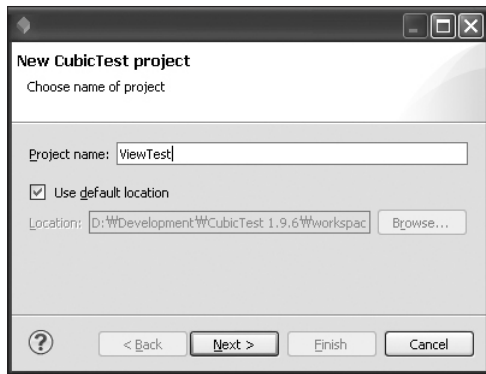


CubicTest의 독립실행 버전. 응? 이클립스???

플러그인을 업데이트 받든가 독립실행 버전을 받아서 실행시킨 모습은 위와 같다. 앞의 경우와 비슷하게 View 영역에 대한 테스트를 작성해보자. 메뉴에서 [File] → [New] → [New CubicTest Project]를 선택한다.

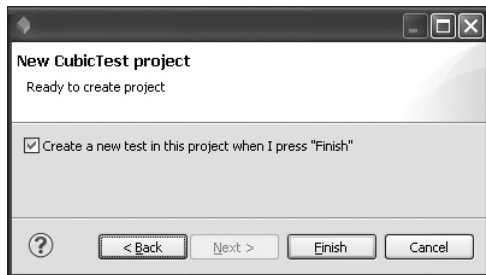


새로운 CubicTest 프로젝트를 생성한다.



프로젝트 이름 정하기

프로젝트 이름은 ViewTestProject라고 입력한다.



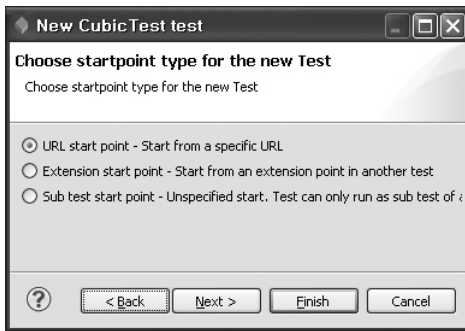
테스트를 바로 만들 것인지 선택

테스트 케이스까지 함께 만드는 것이 기본값이다.



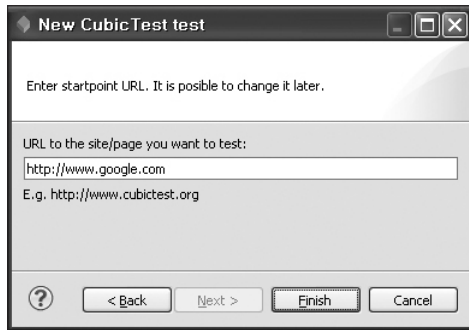
테스트 케이스 이름 정하기

테스트의 이름이다. 정확히는 현재 웹 애플리케이션의 기능 테스트를 만들고 있다.

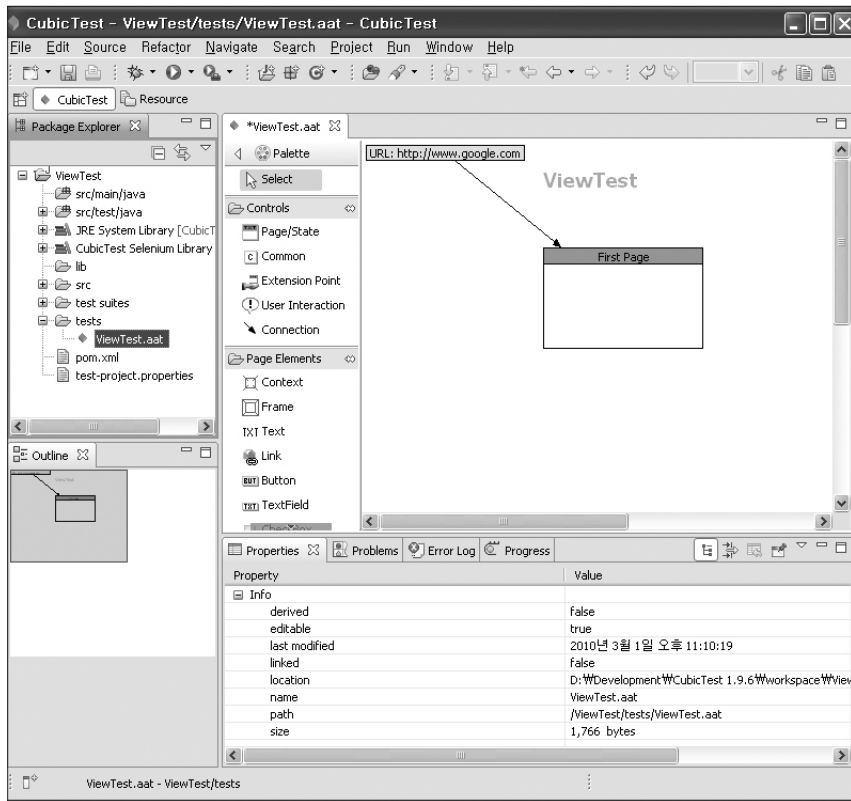


테스트의 시작 지점 정하기

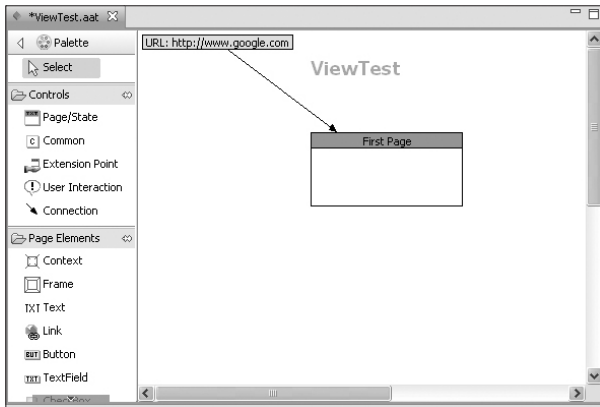
URL을 지정해서 시작할 것인지, 아니면 기존에 작성한 테스트의 확장점에서 시작할 것인지, 아니면 다른 테스트의 서브 테스트로만 시작하게 만들 것인지 결정한다. 우리는 우선 첫 번째, 'URL을 이용한 테스트 시작'을 선택한다.



테스트할 사이트나 URL 주소 적기



시작 화면

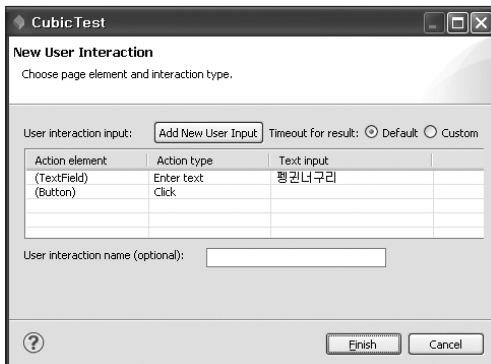


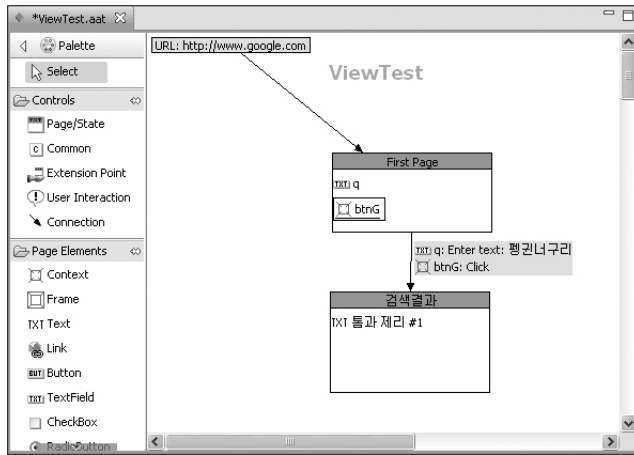
URL 주소로 접근했을 때의 첫 페이지(First Page)를 다이어그램으로 표현하고 있다.

첫 페이지의 구성요소를 정한 다음 왼편의 팔레트에서 드래그&드롭한다. 시나리오는 앞의 경우와 같다.

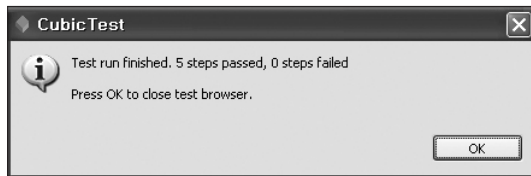
- ❶ 특정 사이트(지금은 www.google.com)에 접속한다.
- ❷ 검색창에 '펭귄너구리'라고 입력한다.
- ❸ 결과 페이지에 '툼과 제리 #1'이라는 문구가 있으면 테스트 성공이다.

결과 페이지를 하나 더 추가한 다음 Controls의 Connection으로 두 페이지를 연결하면 다음과 같은 창이 뜬다. 사용자의 액션을 지정하는 부분이다. 텍스트 필드에 값을 넣고 버튼을 누른다.



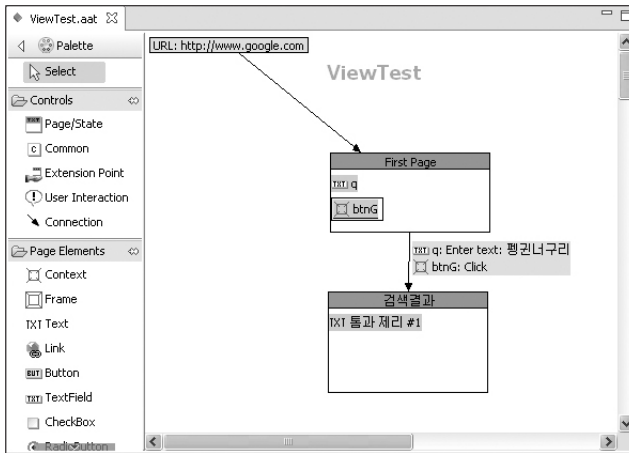


최종 작성된 테스트 케이스는 위와 같다. 검색 결과로 '톰과 제리 #1'이 텍스트로 화면에 보이면 된다. 이클립스의 실행 버튼을 누르면 테스트가 실행된다. 실행해보면 알겠지만, 내부적으로는 Selenium RC가 동작한다. 이전과 마찬가지로 웹 브라우저가 뜨고, 시나리오대로 테스트 케이스가 실행된다.¹⁵



5개의 스텝이 모두 통과됐다고 표시된다.

15 브라우저를 종류를 바꾸고 싶으면 test-project.properties 파일을 참고한다.



테스트를 통과한 요소들은 녹색으로 표시된다. 시작 페이지에 q라는 이름의 text 필드 존재(통과), btnG라는 버튼 존재(통과), 텍스트 필드에 ‘펭귄너구리’ 정상 입력(통과), submit 버튼에 해당하는 btnG 정상 클릭(통과), 결과 페이지에 ‘통과 제리 #1’ 표시됨(통과), 이렇게 5개의 스텝이 모두 통과했다. Selenium RC를 이용해 직접 스크립트를 작성하는 것보다는 작업이 좀 수월해졌다. TDD라고 꼭 코드를 작성해야 하는 건 아니다. 이렇게 시나리오를 먼저 만들어놓고 뷰를 작성해서 위와 같은 툴 테스트를 통과하게 만들어도 상관없다.

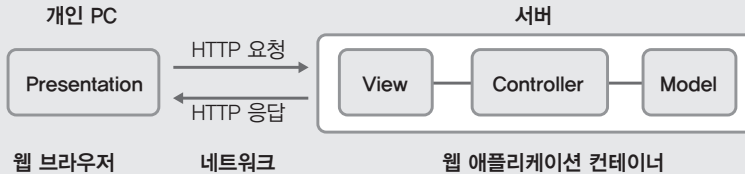
페이지 구성요소에 대한 테스트는 이 정도로 마친다. 지금까지 살펴본 세 가지 방법은 어떤 식으로든, 미리 목표 이미지를 정해놓게 개발을 할 수 있도록 만드는 방법이다. Selenium 같은 경우엔 웹 애플리케이션, 그중에서도 기능 테스트 쪽의 자동화 테스트 툴에 더 가깝지만 TDD에도 이용할 수 있다. 다만, 앞에서도 이야기했던 것처럼, 뷰에 대한 TDD는 노력 대비 효율이 좋지 않다. 차라리 완성된 후에 녹음(record) 방식으로 기능 테스트를 만들어놓은 다음, 변경 시에 조금씩 수정해서 유지하는 방식을 더 권장한다. 마지막에 CubicTest를 자세히 소개한 이유도 자동화 테스트에 적극 사용하길 바라는 마음에서이다.

마지막으로 뷰 영역을 TDD로 개발하는 것에 대한 필자의 소견은 가능하다고 어떻게든 하는 게 꼭 답은 아니라는 것이다.



MVC의 뷰는 서버 쪽인가, 사용자의 화면 쪽인가?

앞에서 MVC를 설명할 때 뷰를 사용자 PC의 웹 브라우저 쪽인 것처럼 표현했다. 하지만 잘 생각해보면, MVC는 모두 서버 쪽에 있다. JSP가 됐든, PHP가 됐든, ASP든, 실제 사용자 PC의 브라우저 입장에서 HTML 페이지와 다를 바가 없다. 왜? 앞에서도 이야기했지만, HTTP 통신에서 넘어오는 정보는 HTML(Ajax의 경우 JSon이나 XML이 되겠지만)뿐이다. HTTP 자체가 그렇게 되도록 설계된 프로토콜이기 때문이다. 그래서 좀 더 실제적인 구조를 표현해보면 아래와 같다.

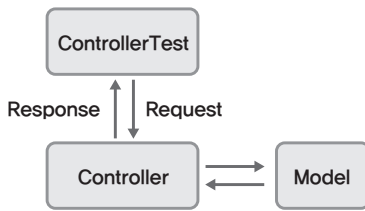


그래서 뷰를 테스트하겠다는 말의 의미는 JSP 같은 Dynamic Page를 서버 차원에서 테스트하겠다는 말인 건지, 아니면 사용자의 입장에서 보이는 Presentation 차원의 기능 테스트를 하겠다는 건지, 조금 고민해볼 필요가 있다.

7.2.3 컨트롤러 TDD

컨트롤러(controller)는 모델과 뷰를 분리하기 위해 사용되는 중간 층이다. 뷰로부터 넘어온 요청에서 데이터 모델을 발췌해 적절한 모델 쪽으로 넘겨주고, 모델로부터 받은 응답을 다시 뷰로 돌려준다. 뷰 입장에서는 컨트롤러만 알면 되고, 모델 입장에서도 컨트롤러만 알면 된다. 모델과 뷰를 재사용할 수 있게 해준다는 부가적인 장점도 제공한다.

MVC 모델에서 컨트롤러를 테스트하는 가장 간단한 방법은 뷰로부터 넘어오는 요청(Request)를 가상으로 만들어주고, 그 결과에 해당하는 응답이 예상과 일치하는지 판단하는 것이다. 이런 방식의 장점 중 하나는 뷰를 먼저 작성하지 않아도 되고, 뷰로부터 발생하는 액션을 정상 동작시키기 위한 Tomcat이나 WebLogic 같은 웹 컨테이너가 따로 필요 없다는 점이다.



ControllerTest는 뷰를 대신해서 요청을 컨트롤러로 전달하고 응답을 결과로 받아 예상값과 비교한다.

흔히 MVC 아키텍처에서는 서블릿이 컨트롤러의 역할을 맡게 된다. 자, 그럼 우리도 서블릿을 컨트롤러 역할로 사용해서 테스트 케이스를 작성해보자.

컨트롤러에 대한 테스트 케이스 작성

화면에서 사원번호를 입력하면, 해당 사원의 정보를 출력해주는 상황을 가정해보자.

직원정보 검색
Employee Search 1.0

테스트 케이스 작성에 사용할 데이터

이름	박성철
사번	5874
아이디	fupfin
직위	회장

컨트롤러에 해당하는 서블릿은 EmployeeSearchServlet이라는 이름으로 만들 예정이다. TDD 방식이 의례 그렇듯, 시나리오에 맞춰서 ControllerTest를 작성해보자. 단, 이때 Request 객체와 Response 객체를 자유롭게 다루기 위해서 스프링 프레임워크의 테스트 라이브러리를 이용했다.¹⁶ Request나 Response는 서블릿 표준 스펙상 개발자

¹⁶ 몇 개의 Mock 객체들을 사용해봤는데, 스프링의 웹 지원 Mock 객체가 가장 사용이 편리했다. 또한 나중에 스프링을 배울 때도 도움이 되리라 생각되어서 사용하게 됐다.

가 마음대로 내부 상태를 변경할 수 없는데, 스프링 프레임워크의 테스트 라이브러리는 이 부분을 극복할 수 있게 도와준다. 이에 대해서는 ‘스프링 프레임워크’를 살펴보는 부분에서 좀 더 다룰 예정이다. 여기서는 Request나 Response의 Mock 구현체를 이용하기 위해 필요한 라이브러리만 클래스패스에 넣어 간략하게 사용했다. 일반적인 Mock 객체의 개념이나 사용법은 이미 앞에서 다뤘으니까 그 내용을 떠올리면서 가능한 한 부담 없이 살펴보자. 테스트 케이스 작성에 쓰인 참조 라이브러리는 소스코드 설명 뒤에 그림으로 첨부했다.

```
package test;

import static org.junit.Assert.assertEquals;
import main.EmployeeSearchServlet;
import org.junit.Test;
import org.springframework.mock.web.*;

public class EmployeeSearchServletTest {
    @Test
    public void testSearchByEmpid() throws Exception {
        MockHttpServletRequest request = new MockHttpServletRequest(); // ❶
        MockHttpServletResponse response = new MockHttpServletResponse(); // ❷

        request.addParameter("empid", "5874"); // ❸

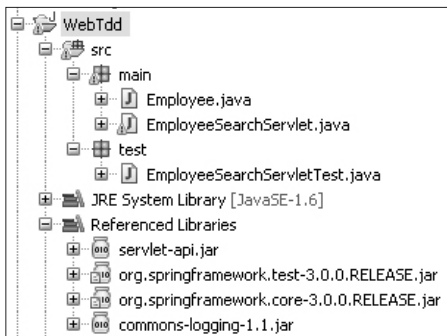
        EmployeeSearchServlet searchServlet = new
            EmployeeSearchServlet(); // ❹
        searchServlet.service(request, response); // ❺

        Employee employee = (Employee)request.getAttribute("employee"); // ❻
        assertEquals("박성철", employee.getName()); // ❼
        assertEquals("5874", employee.getEmpid());
        assertEquals("fupfin", employee.getId());
        assertEquals("회장", employee.getPosition());

        assertEquals("/SearchResult.jsp", response.getForwardedUrl()); // ❽
    }
}
```

- ❶ 스프링 프레임워크의 테스트 라이브러리를 이용해 Mock Request를 생성했다.
- ❷ 마찬가지로 Mock Response를 생성했다.
- ❸ Request에 empid라는 파라미터 이름으로 5874라는 사번이 담겨왔다고 가정한다.
- ❹ 컨트롤러에 해당하는 EmployeeSearchServlet을 생성한다.
- ❺ EmployeeSearchServlet 컨트롤러를 동작시킨다.
- ❻ MVC 모델에서 컨트롤러가 해야 하는 역할 중 하나는 모델에서 수행된 결과를 Request에 담는 것이다. 정상적으로 컨트롤러가 실행돼서 Request에 예상한 값이 담겨 있는지 확인해야 한다. 예제에서는 employee라는 이름의 Employee 객체를 Request에서 꺼냈다. 앞에서 설명한 DTO라고 보면 된다.
- ❼ 단정문을 이용해 값을 확인한다.
- ❽ 응답 페이지 설정이 예상과 일치하는지 확인한다. 마찬가지로, MVC 모델에서 컨트롤러가 해야 하는 역할이다.

보면 알겠지만 Request, Response 객체를 사용한다는 것 말고는 일반적인 애플리케이션의 테스트 케이스 작성과 별반 다르지 않다. 이제 EmployeeSearchServlet을 구현하면 된다.



실습 예제를 실행하기 위한 의존성 라이브러리



실습에 필요한 라이브러리 내려받기

Spring Community Downloads

- Spring Framework
 - Latest GA release: 3.0.1.RELEASE-A
 - spring-framework-3.0.1.RELEASE-A-dependencies.zip (sha1) 133.3 MB
 - spring-framework-3.0.1.RELEASE-A-with-docs.zip (sha1) 45.3 MB
 - spring-framework-3.0.1.RELEASE-A.zip (sha1) 21.8 MB
 - More >>

실습에 필요한 파일은 각각 찾아도 되지만, 스프링 공식 사이트의 다운로드 사이트(<http://www.springsource.com/download/community>)에서 의존성 라이브러리가 포함되어 있는 버전 (spring-framework-3.0.1.RELEASE-A-dependencies.zip)을 내려받으면 실습에 필요한 라이브러리가 안에 다 들어 있다. 다만 이 경우, com.springsource나 프로젝트 접두어가 붙어서 이름은 조금씩 다를 수 있다. commons-logging-1.1.jar 대신에 com.springsource.org.apache.commons.loggings-1.1.1.jar 같은 식으로 말이다.

직원정보 검색

Employee Search 1.0

이름	박성철
사번	5874
아이디	fupfin
직위	회장

SearchResult.jsp 페이지

다음은 위 테스트 케이스를 기반으로 작성한 EmployeeSearchServlet이다. 하나의 예 일 뿐이라는 걸 감안 하고 살펴보자.

```
import javax.servlet.*;
import javax.servlet.http.*;

public class EmployeeSearchServlet extends HttpServlet {
    @Override
```

```

        protected void service(HttpServletRequest request, HttpServletResponse
            response) throws ServletException, IOException {
            SearchBiz searchBiz = new SearchBiz();
            Employee employee = searchBiz.getEmployeeByEmpid(
                request.getParameter("empid"));

            request.setAttribute("employee", employee);
            RequestDispatcher dispatcher =
                request.getRequestDispatcher("/SearchResult.jsp");
            dispatcher.forward(request, response);
        }
    }

```

한 가지 트집을 잡자면, 컨트롤러에 해당하는 위의 EmployeeSearchServlet은 SearchBiz와 강하게 결합되어 있다. 좀 더 약하게 결합시키기 위해서는 인터페이스를 사용하거나 객체를 주입할 수 있는 창구, set 메소드를 만드는 것도 하나의 방법이다. 다음 코드는 모델을 따로 주입할 수 있도록 set 메소드를 추가한 모습이다.

```

public class EmployeeSearchServlet extends HttpServlet {
    private SearchBiz searchBiz;

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
        response)
        throws ServletException, IOException {

        Employee employee = this.searchBiz.getEmployeeByEmpid(
            request.getParameter("empid"));

        request.setAttribute("employee", employee);
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/SearchResult.jsp");
        dispatcher.forward(request, response);
    }
}

```

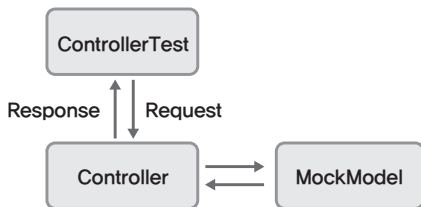
```

    public void setModel(SearchBiz biz){
        this.searchBiz = biz;
    }
}

```

의존성을 줄인 컨트롤러 테스트 케이스 작성

좀 더 나아가서, 완전히 컨트롤러만 테스트하고자 한다면, Mock을 이용해 모델까지도 Mock으로 전환할 수 있다.



MockModel을 이용한 테스트 케이스를 작성해보자. 이왕 Mock을 이용하는 김에 앞에서 배웠던 Mockito와 Unitils를 이용해보자. MockModel을 만드는 건 Mockito에게 맡기고, 객체에 대한 동치비교는 Unitils의 `assertLenientEquals`를 이용할 생각이다.

```

package test;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.*;
import static org.unitils.reflectionassert.ReflectionAssert.*;
import main.*;
import org.junit.Test;
import org.springframework.mock.web.*;

public class EmployeeSearchServletTest {

    @Test
    public void testSearchByEmpid() throws Exception {
        MockHttpServletRequest request = new MockHttpServletRequest();
    }
}

```



```

MockHttpServletResponse response = new MockHttpServletResponse();

request.addParameter("empid", "5874");

SearchBiz biz = mock(SearchBiz.class); // ❶
Employee expectedEmployee = new Employee("박성철", "5874", "fupfin",
    "회장"); // ❷
when(biz.getEmployeeByEmpid(anyString())).thenReturn(
    expectedEmployee); // ❸

EmployeeSearchServlet searchServlet = new EmployeeSearchServlet();
searchServlet.setModel( biz ); // ❹
searchServlet.service(request, response);

Employee employee = (Employee)request.getAttribute("employee");
assertLenientEquals(expectedEmployee, employee); // ❺

    assertEquals("/SearchResult.jsp", response.getForwardedUrl());
}
}

```

- ❶ Mockito를 이용해 Mock 객체를 만든다. 현재는 SearchBiz가 인터페이스가 아닌 일반 클래스다. 만일 모델에 대한 설계가 좀 더 잘 이뤄졌다면, 아마 인터페이스가 됐을 것이다.
- ❷ 검색 결과로 돌려줄 예상값에 해당하는 DTO를 미리 작성했다.
- ❸ Mockito로 Stub을 만들었다. getEmployeeByEmpid의 파라미터로는 어떤 문자열이 되어도 무방하고, 그러면 expectedEmployee를 돌려준다.
- ❹ 의존성 모듈인 Biz 모델을 주입한다.
- ❺ 동치비교를 위해 Unitils의 assertLenientEquals를 사용했다.

앞 예제 소스와 동일한 테스트이지만, 이번엔 모델까지도 격리시켰다. 이제 순수하게 컨트롤러의 기능에만 집중되어 있는 테스트 케이스가 됐다.



예제 실습을 위한 참조 라이브러리들의 모습

지금까지 웹 애플리케이션의 MVC 모델에서 컨트롤러를 테스트하는 방식에 대해 살펴봤다. 처음 보면 다소 낯설 수 있지만, 크게 복잡한 내용은 없기 때문에 금새 익숙해질 것이다. 대부분의 경우 컨트롤러는 요청에서 데이터를 받채하는 역할과 적절한 모델을 찾아 해당 데이터를 넘기는 일을 주로 하는데, 프레임워크 차원에서 지원해주는 경우에는 자칫 컨트롤러에 대한 테스트가 아니라, 프레임워크 자체를 테스트하는 모양이 될 수도 있으니 유의하자.

7.2.4 모델 TDD

모델은 MVC에서 M에 해당한다. 모델에 대한 TDD의 접근은, 우선 모델이 실제로는 두 부분으로 크게 구분된다는 사실을 인식하는 데서 시작한다. 바로, 도메인 모델(domain model)과 서비스 모델(service model)이다. 흔히 데이터(data)와 로직(logic)이라고 부르기도 하고, 도메인 모델로 DTO가 많이 사용되기 때문에 DTO와 Biz(비즈니스)라고도 한다.

모델에 대한 TDD는 앞서서부터 지속적으로 봐왔던 일반적인 TDD의 경우에 해당한다. 간략하게 정리하면서 넘어가 보자.

도메인 모델에 대한 TDD

도메인 모델의 대표적인 예가 DTO이고, 앞에서 살펴봤듯이 대부분의 DTO는 단순하기 때문에 TDD로 작성할 필요가 거의 없다. 하지만 반대로 이야기하면, 단순하기 때문에 TDD로 만드는 데도 거의 시간이 소요되지 않는다. 테스트 커버리지를 측면에서라면 사실 간단히 만들어놓는 것도 나쁘지만은 않다.

서비스 모델에 대한 TDD

서비스 모델은 기능 위주로 구성된 클래스들로, DTO를 비롯한 여러 다른 도메인 클래스들을 사용하는 애플리케이션의 핵심적인 부분이다. 지금까지 TDD의 예제로 사용했던 부분들이 대부분 서비스 모델에 해당한다. 그리고 MVC 모델에서 TDD로 작성했을 때 가장 빛을 발하는 부분이 바로 이 부분이다. 다른 부분들보다도 테스트 커버리지를 최대한 높일 필요가 있다. 필요하다면 100%가 된 상황에서도 추가 케이스를 작성한다. 이 부분에서는 별다른 논의점이 없다. 무조건 최대한 TDD로 만든다고 생각하면 편하다. 얼마만큼? 지겨움이 익숙함이 돼서 두려움이 없어질 때까지!

대신, 논의할 만한 부분이 하나 있는데, 서비스 모델이 단순히 화면과 저장영역(DB) 사이의 통로가 될 뿐인 형태의 경우 TDD를 할 가치가 있느냐에 대한 부분이다.

스쳐 지나가는 서비스 모델 pass through service model

인증, 권한처리 등의 공통사항은 기반이 되는 프레임워크가 처리해주고, 모델이 하는 일이라고는 SQL과 DAO를 연결해주는 것과, 결과값을 객체로 받는 것이 전부인 서비스 모델이 있다. 이를테면 사용자 등록, 전화번호 검색, 공지사항 출력, 목록 화면 등의 경우는, 조건에 맞도록 SQL을 작성해 DB에서 정보를 가져오면 따로 더 이상 할 일이 없다.

```
SELECT 화면에필요한컬럼 FROM 테이블들 WHERE 조건절;
```

배송 검색

Shipping List Search

검색

대리점코드	대리점이름(한글)	대리점이름(영문)	국가코드	유형
AKOR001	서울본사	Seoul Head Office	KOR	직영
AKOR002	서울강북지사	Seoul North Branch	KOR	직영
AKOR003	서울강서지사	Seoul West Office	KOR	직영

.....

업무화면

다음은 작성된 서비스 모델의 소스코드다.

```
public class ShipManager {
    public ShippingList getActualShippingList(String shipperId) throws
        Exception {
        String sqlName = "/gx/booking/retrieveActualShprList";
        CommonDao dao = new CommonDao(sqlName, shipperId);
        return (ShippingList)dao.executeQuery();
    }
    ...
}
```

SQL 파일을 이름으로 읽어들이어서 실행할 수 있게 도와주는 프레임워크 등을 사용했다. Biz 클래스라 불리는 서비스 모델의 getActualShippingList 메소드가 하는 일은 적절한 SQL 파일을 불러와서 DAO로 넘겨주는 게 전부다. 예제의 sqlName에 해당하는 SQL 파일은 다음과 같다.

```
<?xml version="1.0" encoding="UTF-8"?>
<sqls>
    <sql name="/gx/booking/retrieveActualShprList">
        SELECT agent.CODE,
               agent.KOR_NAME,
               agent.ENG_NAME,
               agent.NATION_CODE,
               agent.TYPE
```

```
FROM tb_gxa101 agent
WHERE agent.type IN ('01', '02')
...
```

이런 경우에 TDD로 작성하는 것이 가능한 한 건지, 가능하더라도 효과는 있는 건지에 대한 논란의 소지가 있다. 보통 이럴 때는 한정적으로 TDD를 적용하게 된다. DAO를 통해 넘어온 결과의 건수, 헤더(header)에 해당하는 컬럼이름의 일치 여부 정도를 목표표로 TDD를 적용한다.

```
@Test
public void testGetActualShippingList () {
    ShipManager shipManager = new ShipManager();
    ShippingList list = (ShippingList)shipManager.getActualShippingList("IT
        EM003");

    assertTrue( list.size() > 0 );
    assertEquals("[CODE, KOR_NAME, ENG_NAME, NATION_CODE, TYPE]",
        list.headers() );
}
```

모델 실행 결과의 데이터 건수와 헤더 컬럼 일치를 테스트한다. 건수는 보통 0, 1, 1 이상 정도로 체크한다. 모델 테스트임에도 사실 SQL 정상 작성 테스트에 더 가깝게 변했다. 장점은, SQL 문을 변경하면 그 즉시 에러가 나는 모델이 발견된다는 점이다. 크진 않지만 나름의 유용함이 있다. 하지만 이 방식을 적용할 때 유의해야 할 점이 하나 있다. 자칫 TDD의 의미를 잊고는, 헤더 비교하는 부분을 SQL 파일에서 복사해온다는가 SQL 실행 결과에서 가져다 붙이는 경우가 있는데, 절대 그러면 안 된다. 그건 DAO가 정상 동작하는지를 테스트하는 것이 돼버린다. TDD는 목표 이미지에 도달하도록 구현했느냐를 판별하는 게 목적이다. 따라서 헤더 컬럼이름은 화면이나 화면설계서를 보고 assert 문을 작성해야 한다. Biz의 역할이 화면에 필요한 데이터를 가져오는 것이기 때문이다.

```
assertEquals("[CODE, KOR_NAME, ENG_NAME, NATION_CODE, TYPE]", list.headers());
```

화면에서 필요한 부분

모델 실행 결과로 받은
헤더 부분

이런 모델의 테스트는 다소 효율이 낮긴 하지만, 없는 것보다는 낫다. SQL 문의 변화에도, 화면요소의 변화에도 민감하게 반응해주기 때문이다. 참고로 필자가 참여했던 프로젝트에서 이 방법을 사용했었는데, 개발자나 분석설계자들의 반응은 이랬다.

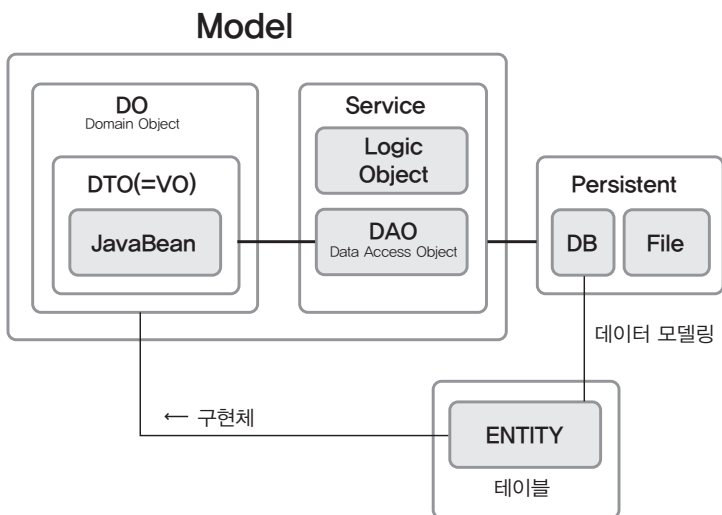
“화면에서는 이런저런 예러가 많은 경우에도 일단 화면을 넘어서 서버로 가면 예러가 나는 경우가 거의 없다는 게 신기하네요!”

하지만 신기한 게 아니라 SQL이 변경되거나 화면의 구성요소가 변경될 때, 이미 작성되어 있는 테스트 케이스들을 통해 개발자들이 빠르고 쉽게 Biz 클래스와 SQL 문장을 재작성할 수 있었기 때문이다.



DTO, VO, DO, DAO, ENTITY, JavaBean!!!

DTO(Data Transfer Object), VO(Value Object), DO(Domain Object), DAO(Data Access Object), ENTITY, JavaBean은 흔히 많이 혼용/오용되는 용어들이다. 소스코드의 모양으로 구분하려고 들면 혼란스러울 수 있다. 이야기가 나온 김에 구분하고 넘어가 보자. 모델은 서비스와 데이터로 나뉘고, 서비스에서 사용하는 데이터 객체는 일반적으로 DO(Domain Object, 혹은 Business Object)라고 부른다. DTO는 데이터 전달을 위해 사용되는 객체이며, 비즈니스적인 구현이 들어가 있지 않다. 예전에는 VO라고 불렸었다. JavaBean은 재사용을 위해 정의된 자바 컴포넌트를 지칭한다. 단, DTO로 많이 사용되는데, 좀 더 엄격한 점은 반드시 set/get으로 이름이 시작하는 메소드로만 데이터에 접근해야 한다는 점이다. DAO는 영속성 객체라 불리는 DB나 File 등과 데이터를 주고받기 위해 쓰이는 객체이며, 보통 매개체로 DTO를 사용한다. DAO를 서비스에 포함시키는 것이 일반적이지만, 따로 분리해서 생각하는 경우도 있다. ENTITY는 데이터 모델링에서 쓰이는 표현으로 DB의 테이블에 해당한다. ENTITY 클래스라고 불리는 클래스들은 DB의 ENTITY를 클래스 구조로 만들어놓은 모습이다. 프로그램으로 구현될 때는 종종 DO/DTO의 모습을 갖는다. 이렇게 이야기해도 헷갈릴 수 있는데, 사실 다른 건 다 잊고, DTO와 DAO만 구분해서 사용할 수 있으면 대개 별 문제가 없다.



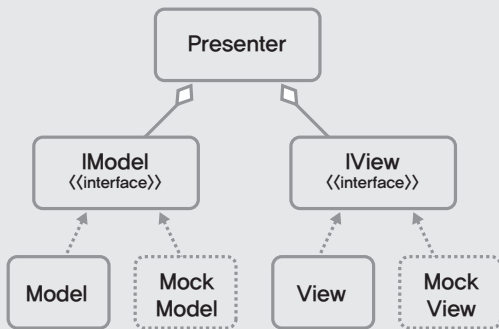
모델 영역의 용어 관계도

웹 애플리케이션에 대한 TDD 접근 전략 정리

- 모델과 뷰와 컨트롤러를 최대한 분리시킨다.
- 뷰는 단순히 표현 계층(presentation layer)으로 보고 업무로직이 들어가지 않도록 유지한다.
- 뷰에 대한 TDD는 ROI를 잘 따져보고, 필요하다면 TDD를 포기하고 Record & Play 방식의 툴을 사용해 기능 테스트나 회귀 테스트의 비용을 줄이는 쪽으로 생산성을 높이자.
- 컨트롤러가 프레임워크 차원에서 지원될 때는 굳이 테스트 케이스를 만들려고 하지 않는다.
- 모델에 대한 TDD는 최대한으로 적용한다.



MVP 모델에서의 프리젠퍼 우선(Presenter First) 접근법



웹 애플리케이션 개발에 MVC(Model-View-Controller)가 있다면, 데스크톱 UI 애플리케이션 개발에는 MVP(Model-View-Presenter)라는 것이 있다. MVC와 MVP는 C와 P를 제외하고는 동일한 구조다. 다만 MVP에서는 뷰가 프리젠퍼의 상태를 알 수 있게 프리젠퍼의 인스턴스를 갖고 있다. 또한 MVP 패턴의 특징 중 하나는 모든 비즈니스 로직이 프리젠퍼 안에 들어 있고, 프리젠퍼는 뷰의 인터페이스만 상대한다는 점이다. 따라서 해당 인터페이스를 모킹(mocking)하는 형태로 테스트가 가능해진다. 이 이야기는 화면 UI의 동작을 인터페이스로 지정해놓았기 때문에, UI가 바뀐다고 해도 프리젠퍼가 받는 영향이 최소화된다는 뜻이다. 이 구조의 또 다른 장점은 View와 Model을 최대한 분리시킬 수 있게 해준다는 점이다. 이건 MVC의 C의 장점과 일치한다. 마지막으로, 세 번째 장점은 사용자를 위한 UI가 미리 만들어지지 않은 상태에서도 유저 스토리에 해당하는 이벤트를 미리 작성할 수 있게 해준다는 점이다(만들어졌다 해도 UI는 곧잘 바뀌곤 한다). 왜냐하면 이미 IView라는 인터페이스가 있기 때문이고, 인터페이스니까 Mock으로 만들어서 로직이 들어 있는 프리젠퍼를 작성할 수가 있다. 이렇듯, 프리젠퍼를 먼저 만들고 나머지는 인터페이스를 통해 개발하는 방식을 프리젠퍼 우선(Presenter First) 접근법이라고 부른다. 이런 방식을 이용하면, 이벤트 기반의 데스크톱용 UI 애플리케이션을 만들 때도 좀 더 쉽게 TDD 적용이 가능하다.

참고

Presenter First: Organizing Complex GUI Applications for Test-Driven Development, Agile 2006

7.3 데이터베이스

TDD는 흔히 프로그래밍의 두 가지 경우를 예상해서 진행된다. 상태(state)와 동작(behavior). 동작만 테스트하는 건 어렵지 않다. 또한 특정 상태만 테스트하는 것도 어렵지 않다. 그런데 이 두 개가 함께 섞이면 복잡해진다. 그 대표적인 예가 데이터베이스가 사용될 때다.

데이터베이스 관련 테스트는 몇 가지 어려움이 있는데 그중 대표적인 걸 뽑아본다면 다음과 같다.

- 테스트를 진행하면 데이터베이스에 들어 있는 데이터의 상태가 바뀐다.
- 테스트 전/후의 데이터 비교가 쉽지 않다.

각각에 대한 일반적인 해결 방법을 살펴보자.

데이터베이스 상태가 바뀌는 문제의 일반적인 해결 방법

데이터베이스 상태 변경과 관련된 테스트 케이스는 어떻게 작성 가능한지 살펴보자. 소개되는 방법 중에서 자신에게 가장 잘 맞을 방법을 하나 선택해서 사용하면 된다. 사용하다가 한계가 오는 것 같으면 다른 방법을 이용해보자. 아쉽게도 모든 경우에 대한 최선의 방법은 없다. 필자 개인적으로는 DbUnit과 Unitils 조합을 선호하지만, 각각 학습이 필요하다는 점은 단점이다. 자, 그럼 하나씩 살펴보자.

해결책1 트랜잭션을 선언하고 테스트 케이스를 수행한 다음 롤백처리한다.

```
public class DBRepositoryTest {  
    private final String protocol = "jdbc:derby:";  
    private final String dbName = "shopdb";  
  
    private Connection connection;  
    private Repository repository;  
  
    @Before
```

```

public void setUp() throws Exception {
    repository = new DatabaseRepository();
    connection = ((DatabaseRepository)repository).getConnection();
    connection.setAutoCommit(false);    // ❶
}

@After
public void tearDown() throws Exception {
    this.connection.rollback();    // ❷
    this.connection.close();
}

@Test
public void testAddNewSeller() throws Exception {
    Seller newSeller = new Seller("hssm", "이동욱", "scala@hssm.kr");
    repository.add(newSeller);

    assertEquals(newSeller, repository.findById("hssm"));
}
}

```

❶ 트랜잭션을 선언한다.

❷ 테스트 케이스 수행이 끝난 다음엔 rollback으로 되돌린다.

테스트 케이스를 이용해 작성된 DatabaseRepository의 모습

```

public class DatabaseRepository implements Repository {
    private final String driver = "org.apache.derby.jdbc.EmbeddedDriver";
    private final String protocol = "jdbc:derby:";
    private final String dbName = "shopdb";
    private Connection conn;

    public DatabaseRepository() throws Exception {
        Class.forName(driver).newInstance();
        setConn(DriverManager.getConnection(protocol + dbName));
    }
}

```

```

public void setConn(Connection conn) {
    this.conn = conn;
}

public Connection getConn() {
    return conn;
}

@Override
public void add(Seller seller) {
    PreparedStatement stmt = null;
    try {
        String query = "insert into seller values (?, ?, ?)";
        stmt = getConn().prepareStatement(query);
        stmt.setString(1, seller.getId());
        stmt.setString(2, seller.getName());
        stmt.setString(3, seller.getEmail());

        int affectedRows = stmt.executeUpdate();
        if (affectedRows == 0){
            throw new SQLException("Seller adding fail!");
        }
        stmt.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

예제에서는 메소드 단위로 트랜잭션을 관리했지만, @BeforeClass를 이용해 테스트 클래스 단위로 트랜잭션을 관리할 수도 있다.

장점	트랜잭션 선언만 처리하면 되기 때문에, 테스트 케이스 작성이 간단하다.
단점	테스트 케이스의 작성 목적이 '트랜잭션 처리'인 경우에는 경우엔 적용 불가. 트랜잭션을 테스트 내에서 제어할 수 없는 경우에도 적용 불가.

해결책2 테스트 케이스 작성 시 '입력 → 수정 → 삭제' 순서대로 테스트 케이스가 실행되도록 만든다.

```
@Test
public void testAddNewSeller() throws Exception {
    Seller newSeller = new Seller("hssm", "이동욱", "scala@hssm.kr");
    repository.add(newSeller);
    assertEquals(newSeller, repository.findById("hssm"));
}

@Test
public void testFindByIdSeller() throws Exception {
    Seller expectedSeller = new Seller("hssm", "이동욱", "scala@hssm.kr");
    assertEquals(expectedSeller, repository.findById("hssm"));
}

@Test
public void testUpdateSeller() throws Exception {
    Seller seller = new Seller("hssm", "이동욱", "scala@ksug.or.kr");
    repository.update(seller);
    assertEquals(expectedSeller, repository.findById("hssm"));
}

@Test
public void testRemoveSeller() throws Exception {
    Seller seller = new Seller("hssm", "이동욱", "scala@hssm.kr");
    repository.remove(seller);
    Seller actualSeller = repository.findById("hssm");

    assertEquals("존재하지 않는 ID입니다", actualSeller.getId());
}
```

add → findById(=select) → update → remove 순서로 테스트가 진행된다.

장점	테스트 케이스 작성 시에 큰 노력이 필요하지 않다.
단점	입력/수정/삭제 세 기능 중 일부 기능이 업무적으로 지원되지 않을 경우엔 적용 불가능하다. 또한 테스트 케이스 실행 순서를 고려해야 하는데, 일반적으로 테스트 케이스 내에 선후 관계가 존재하도록 테스트 케이스를 만드는 걸 권장하지 않는다. 각각 독립적으로 수행될 수 있어야 하기 때문이다.

해결책 3 SQL 스크립트가 테스트 수행 시에 실행되도록 만든다.

SQL 스크립트를 테스트 수행 전에 실행시키는 방법은 매우 다양하다. 어떤 방식이 가능한지 정도만 알 수 있도록, 몇 가지 사례를 간략히 소개한다.

3-1 ANT SQL TASK를 이용하는 방법(Ant를 이용해 테스트를 수행할 때)

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="sqlrun" default="initdb" basedir=".">
  <target name="initdb">
    <sql driver="org.apache.derby.jdbc.EmbeddedDriver"
        url="jdbc:derby:shopdb"
        userid="" password="">
      <classpath>
        <fileset dir="${basedir}/lib" includes="**/*.jar" />
      </classpath>
      DROP TABLE item;
      DROP TABLE seller;
      INSERT INTO SELLER VALUES('horichoi', '최승호',
                                'megaseller@hatmail.com');
      INSERT INTO SELLER VALUES('buymore', '김용진',
                                'shopper@nineseller.com');
      INSERT INTO SELLER VALUES('mattwhew', '이종수',
                                'admin@maximumsale.net');
    </sql>
  </target>
</project>
```

3-2 SQL 스크립트 파일을 실행(자바의 커맨드 실행 방법으로)

```

@Before
public void setUp() throws Exception {
    Process p
        = Runtime.getRuntime().exec("java org.apache.derby.tools.ij
            initdb.sql");
    p.waitFor();
}

```

3-3 Spring 프레임워크의 SimpleJdbcTestUtils를 이용(스프링 2.5 이후)

```

import javax.sql.DataSource;

import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.core.io.*;
import org.springframework.jdbc.core.simple.SimpleJdbcTemplate;
import org.springframework.test.jdbc.SimpleJdbcTestUtils;
import org.unitils.UnitilsJUnit4TestClassRunner;
import org.unitils.database.annotations.TestDataSource;

@RunWith(UnitilsJUnit4TestClassRunner.class)
public class JDBCUtilTest {

    @TestDataSource
    DataSource dataSource;

    @Before
    public void testScripts() throws Exception {
        SimpleJdbcTemplate template = new SimpleJdbcTemplate(dataSource);
        Resource resource = new ClassPathResource("/initdb.sql");
        SimpleJdbcTestUtils.executeSqlScript(template, resource, true);
    }
}

```

클래스패스 내에 존재해야 하는 참조 라이브러리

```
org.springframework.test-3.0.1.RELEASE-A.jar
org.springframework.jdbc-3.0.1.RELEASE-A.jar
org.springframework.core-3.0.1.RELEASE-A.jar
org.springframework.transaction-3.0.1.RELEASE-A.jar
unitils-core-3.1.jar
unitils-database-3.1.jar
```

이 외에도 방법은 매우 다양하다. 방식은 어찌 됐든, 기본적으로는 SQL 스크립트만을 유지보수하게 된다.

장점	테스트 실행 상태를 만드는 작업과 테스트 수행 작업을 분리해서 관리할 수 있다. SQL 문장만 제대로 작성하면 되기 때문에 편리하다.
단점	SQL 스크립트를 실행시킬 유틸리티나 Ant 등이 필요하다. 자체 SQL 파일이 많아지면 별도의 관리가 필요하다.

해결책4 DbUnit을 사용한다.

DbUnit을 사용하는 방식은 앞에서 설명했기에 따로 다루진 않는다. 기억을 되살리는 차원에서 소스만 한번 다시 살펴보자.

```
@Before
public void setUp() throws Exception {
    databaseTester = new JdbcDatabaseTester(driver, protocol + dbName);
    connection = databaseTester.getConnection();
    IDataset dataSet = new FlatXmlDataSetBuilder().build(new
        File("seller.xml"));
    DatabaseOperation.CLEAN_INSERT.execute(connection, dataSet);
}
```

장점	DbUnit의 여러 가지 기능을 사용해서 효과적으로 테스트 케이스를 작성할 수 있다. SQL 문으로 일일이 작성하지 않아도 테스트에 사용할 데이터를 준비하기 쉽다.
단점	DbUnit을 배워야 한다. 자칫 테스트 케이스 소스 여기저기에 데이터셋 파일이 산재하게 될 수 있다.

테스트 전후의 데이터베이스 상태를 비교하는 방법

해결책1 예상 결과를 미리 다른 테이블에 넣어놓고, 대상 테이블과 예상 테이블을 각각 Select 문으로 결과를 받아와서 비교한다(가능은 하지만 매우 불편).

해결책2 예상 결과를 미리 문자열로 만들어놓고 Select 문을 실행해서 이용해 비교한다.

오라클을 비롯한 대형 DBMS 제품들은 SQL 실행 결과를 파일로 만들어주는 기능을 제공한다. 그런 기능을 이용해서 파일비교의 문자열을 읽어들이어서 비교한다. 예전에 급할 때 사용했던 방식이다. 지금도 프로그래밍이 아닌 계열에서 가끔씩 사용한다.

해결책3 DbUnit과 Unitils를 함께 사용한다. 

앞에서 살펴봤으니, 소스코드만 다시 한번 살펴보는 걸로 기억을 되살려보자.

```
@Test
@ExpectedDataSet("expected_seller.xml")
public void testAddNewSeller() throws Exception {
    Seller newSeller = new Seller("hssm", "이동욱", "scala@hssm.kr");
    Repository repository = new DatabaseRepository();
    repository.add(newSeller);
}
```

데이터베이스가 연관된 부분에 대한 TDD 정리

데이터베이스와 연관되어 있는 프로그램에 대해 TDD를 진행할 때, 크게 두 가지가 방법이 가장 효율이 높았다. @BeforeClass를 이용해 트랙잭션을 선언해놓고 ‘입력’ → ‘수정’ → ‘삭제’ 순으로 동작시켰던 방식과 DbUnit, Unitils를 함께 사용하는 방식, 이렇게 두 가지다. 그중에서도 특히 후자를 더 선호한다. 이 외에 하이버네이트 등의 ORM¹⁷ 프레임워크를 사용해 DB 관리를 최소한으로 만들어놓고, 객체 모델을 테스트에 좀 더 집중하는 방법도 있지만, 아직 ORM 자체가 익숙하지 않다면 더 많은 문제가 발생할 수 있으므로, 여기서는 논외로 한다.

17 ORM(Object-Relational Mapping): 객관 관계 매칭. 관계형 데이터베이스와 객체 지향 언어 사이의 데이터 모델을 상호 전환시켜 주는 기술

7.4 안티패턴(anti-pattern), 전통적으로 잘못 인식되어 있는 테스트 메소드의 리팩토링

기본적으로 좋은 테스트 케이스는 다음과 같은 규칙을 따른다고 말했다.

- 하나의 테스트 케이스는 외부와 독립적이어야 한다.
따라서 다른 테스트 케이스에 영향을 주거나 받지 않아야 한다.
- 하나의 일관된 시나리오를 갖고 있어야 한다.

그런데 우리가 테스트 케이스 정련 작업을 할 때 종종 이 항목을 묘하게 위반하곤 한다. 이제 와서 고백하건대, 본인도 앞에서 마찬가지로 작업을 했다. 다만, 한꺼번에 설명하는 것이 부담스러웠고, 다소 논란의 소지가 있는 내용이기 때문에 이 부분까지 미뤘다. 여기까지 왔다면, 당신은 그게 무엇인지 들을 만한 자격이 있다. :)

다음 코드를 다시 한번 살펴보기 바란다. 앞에서 한동안 작성했던 바로 그 테스트 코드다.

```
public class AccountTest {
    private Account account;

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

    @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());
    }
}

```

어느 부분이 미묘하다고 이야기하려는 걸까?

출금하기(withdraw)를 테스트하는 메소드를 살펴보자.

```

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

```

테스트 메소드는 하나의 일관된 시나리오를 갖고 외부에 독립적이어야 하는데, 이 부분만으로는 시나리오가 연결성을 갖지 않는다. setUp을 함께 살펴봐야 문맥적으로 하나의 정상적인 테스트 시나리오가 만들어진다.

```

private Account account;

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

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

```

이건 AccountTest 클래스 내 여타 테스트 케이스의 경우도 마찬가지다. 앞 테스트 코드는 안타깝지만, 다음과 같이 적는 것이 문맥적으로는 오히려 더 적절하다는 생각이 든다.

```

public class AccountTest {

    @Before
    public void setUp(){
    }

    @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());
    }
}

```

“Account 생성이 중복이잖아요!!”라고 항의를 할 수도 있다. 알고 있다. 중복되는 부분이 존재한다.

“아.. 지금 무슨 소리세요? 중복은 나쁜 거고.. 나쁜 냄새로 불리며, 버그를 쉽게 만들어 내고, 또..”

맞는 말이다. 하지만 테스트 케이스 코드는 접근이 조금 다르다. 정련 시에 중복을 제거하는 것은 맞는데, 그 중복되는 부분이 테스트 시나리오의 일부라면 조금 고민해볼 필요가 있다. 비록 중복이 됐지만, 각각 독립된 테스트 시나리오에 해당하는 위의 두 테스트 메소드는 앞선 경우보다 가독성이 더 높다. 테스트 메소드가 하나의 문맥 측면에서 완결성을 갖기 때문이다.

그럼 setUp과 tearDown 같은 테스트 픽스처 메소드는 언제 사용할까? 테스트 시나리오에 참가하고 있는 객체들에게 테스트에 필요한 사전 조건이나 기반 환경을 제공

할 때 사용하는 것이 더 어울린다. 이를테면 다음과 같은 가상의 코드가 그 예에 해당한다.

```
public class AccountTest {

    Connectoin conn;

    @Before
    public void setUp(){
        conn = manager.getConnection();
        Transaction.start(conn);
    }

    @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());
    }

    @After
    public void tearDown(){
        Transaction.end(conn);
    }
}
```

테스트 시나리오를 진행하기 위해 선행되어야 하는 환경과 정리작업에 해당하는 부분을 setUp과 tearDown으로 분리해놓았다. 그리고 테스트에 필요한 테스트 재료는 각각의 테스트 메소드에서 준비했다. 만약 테스트 메소드 내에서 중복 코드가 지속적으로 발

생한다면, setUp 부분으로 옮기는 것보다, 메소드로 뽑아내는 것도 하나의 좋은 해결책이다. 물론 당연한 이야기지만, 이때 메소드 이름이나 변수 이름을 잘 짓는 일 또한 매우 중요하다.

이렇듯, 중복된 코드는 리팩토링의 대상이 되는 것이 맞지만, 테스트 케이스를 작성할 때는 조금 더 고민해보는 시간이 필요하다.

추가 논의

위 내용은 다소 논란의 여지가 있는 부분이라 생각되어 애자일 커뮤니티인 XPer 메일링리스트¹⁸에 본 절의 제목과 내용을 그대로 올렸다. 논의된 의견 중, 박영록님과 김창준 대표님의 답변은 많은 분들과 공유할 만한 좋은 접근법이라 생각되어 코드를 정리해 옮겨본다.

중복을 제거하기 위해 테스트 픽처 생성 부분을 메소드로 추출한 박영록님의 테스트 코드

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

private Account prepareTenThousandAccount() throws Exception {
    return new Account(10000);
}
```

최대한 중복이 줄어들어서 마치 템플릿 코드처럼 동작하는 김창준님의 테스트 코드

```
public class BaseAccountWithDeltasTest {
    private Account account;
    private int base, delta;

    @Before
    public void setUp(){
```

18 <http://groups.google.com/group/xper>

```

        base = 10000;
        delta = 1000;
        account = new Account(base);
    }

    @Test
    public void testDeposit() throws Exception {
        account.deposit(delta);
        assertEquals(base+delta, account.getBalance());
    }

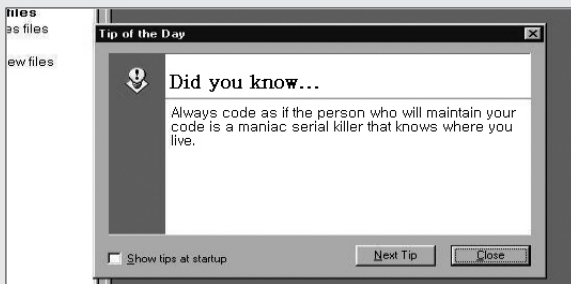
    @Test
    public void testWithdraw() throws Exception {
        account.withdraw(delta);
        assertEquals(base-delta, account.getBalance());
    }
}

```

위 글은 <http://blog.doortts.com/123>에서 찾아볼 수 있으며 찾아보기 쉽게 XPer 모임의 글도 링크로 걸어왔다.



‘최고의 프로그래밍 팁’으로 선정된 어떤 개발팁



“당신이 사는 곳을 아는 어떤 매니악한 연쇄살인범이, 당신이 작성한 코드의 유지보수 작업을 맡게 될 거라고 늘 생각하면서 코딩하세요!”

– Visual C++ Tip of the Day 중에서

이일민 Epril Pty Ltd, Managing Director

Q. 본인 소개와 하시는 일, 혹은 관심 기술이고 있으신 일에 대해 이야기 부탁드립니다.

A. 자바 오픈소스 프레임워크를 기반으로 한 엔터프라이즈 시스템을 구축 컨설팅과 프레임워크 개발, 교육 일을 하고 있습니다. 프레임워크에 대한 프레임워크(FoF)를 활용해서 개발 생산성과 품질일 높일 수 있는 방법에 관심이 많이 있습니다.

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

A. 코드 작성이 필요한 거의 모든 업무에 TDD를 적용하고 있습니다. 오래전에 TDD를 사용하지 않고 개발했던 시스템에 대해서도 유지보수 작업에 TDD를 적극 활용하고 있습니다. 새로운 기술을 학습할 때도 TDD를 이용해서 학습테스트와 예제를 만들기도 합니다.

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

A. 개발팀에 TDD를 도입하기 전에 몇 달간 TDD 훈련시간으로 삼았습니다. 연습문제를 준비해서 일주일에 하나씩 각자 TDD 원칙을 따라 만들어보고 정기적으로 모여서 개발 과정에서 느꼈던 점과 만들어진 코드를 공유하는 시간을 가졌습니다. 처음에는 부담스럽게 생각하고 대충 하려고 했던 팀원들도 시간이 지나면서 더 나은 코드를 만들 수 있으리라는 확신을 갖기 시작했습니다. 시키지 않았는데도 몇 번이고 반복해 다시 시도해보면서 어떻게 더 좋은 코드를 만들어낼 수 있을지 고민하는 사람들이 늘어났습니다. 이전에는 기능만 동작하면 코드는 대충 만들어도 된다고 생각했던 사람들이 깔끔하고 좋은 코드에 관심을 갖는 모습이 인상적이었습니다.

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

A. TDD를 적용하는 데 가장 큰 장애는 테스트 작성에 대한 부담인 것 같습니다. 유명 TDD 서적이나 아티클에 소개되는 예제와 달리 현장에서는 DB를 사용하는 웹 애플리케이션 개발이 대부분인데 TDD를 적용하려고 보면 테스트를 어떻게 만들어야 할지 막막한 경우가 많습니다. 일단 TDD를 본격적으로 시작하기 전에 자동화된 테스트 작성, DB 테스트, 웹 테스트, Mock 오브젝트를 이용한 단위 테스트 작성에 대한 충분한 학습과 꾸준한 훈련이 필요합니다. 동시에 테스트하기 편리한 코드를 만드는 방법에도 관심을 가져야 합니다. 테스트하기 어려운 코드를 리팩토링하는 방법에도 익숙해질 필요가 있습니다. 테스트 작성에 대해 충분한 자신감이 생기면 TDD를 시작하는 것이 바람직하다고 봅니다.

Q. 후배들에게 TDD를 권할 의향이 있으십니까?

A. 소개는 하지만 무작정 권하고 싶지는 않습니다. 일단 TDD는 아니더라도 자신이 만든 코드를 테스트로 검증하는 것에 관심을 가지라고는 권하고 싶습니다. 처음부터 TDD에 대한 환상을 심어주면 쉽게 실망하거나 좌절할 수 있다고 생각되기 때문입니다.

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

A. “모든 전문가는 연습한다”라는 말이 있습니다. 전문가들은 새로운 지식을 학습할 뿐만 아니라, 이미 알고 있는 기술이나 지식을 잘 활용하기 위해서 반복적으로 꾸준한 훈련을 해야 합니다. 개발자로 전문가가 되고 싶다면 최신 기술을 습득하는 것 이상으로, 잘 알고 있는 지식을 적용하는 훈련을 반복적으로 꾸준히 하기를 바랍니다. 특히 TDD는 꾸준한 연습 외에는 자기 것으로 만들 수 있는 방법이 없습니다.