



내 모든 성공은 항상 실패를 바탕으로 만들어졌다.

- 벤저민 디즈레일리

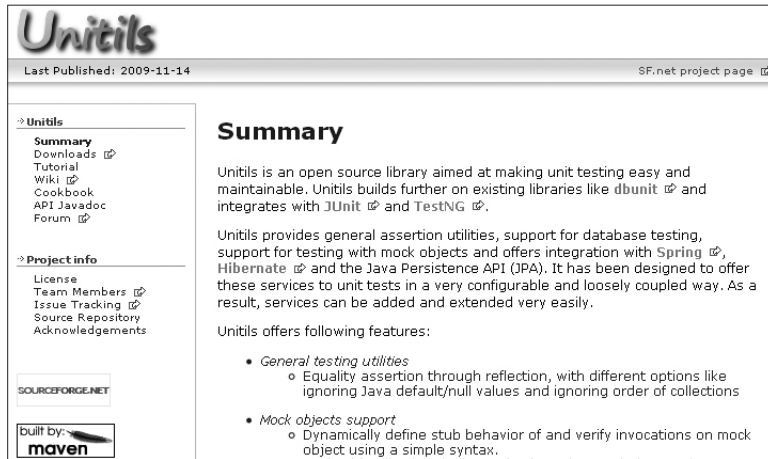
단위 테스트 지원 라이브러리: Unitils

오픈소스 중 하나인 Unitils(유닛틸즈)는 단위 테스트를 좀 더 쉽게 만들고 더 유연하게 유지할 수 있게 도와주는 일종의 단위 테스트 지원 라이브러리다. DbUnit과 마찬가지로, 독립적으로 사용된다기보다는 주로 다른 테스트 프레임워크와 함께 사용된다. 자칫, 수많은 라이브러리 중 하나쯤으로 간주하고 넘어갈 수도 있지만, 부디 그러지 않길 바란다. 왜냐하면 Unitils의 단위 테스트 지원 기능을 살펴보는 것만으로도, 테스트 케이스 작성시 흔히 발생하는 어려움에는 어떤 것이 있는지 배울 수 있는 좋은 기회가 되기 때문이다. 최대한 복잡하지 않도록 유용한 기능 위주로 설명을 할 테니, 힘들더라도 조금만 참고 따라와 줬으면 좋겠다. DbUnit과 마찬가지로, 매뉴얼적인 성격을 띤 부분이 중간에 나온다. 많이 지루하다 싶으면 앞서와 마찬가지로 편하게 훑고 넘어가자. 7장부터는 좀 더 재미있는 내용들이 기다리고 있다. 6장에서 포기하는 바람에, 7장과 그 이후를 읽지 못하는 우를 범하지 않기 바란다. 그럼 이제부터 Unitils에 대해 살펴보자.

6장 체크리스트

- 리플렉션 단정문(reflection assertion)
- 너그러운 단정문(lenient assertion)
- Unitils 모듈
- DBMaintainer

6.1 Unitils를 사용하기 위한 환경 준비



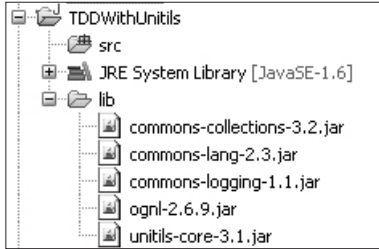
www.unitils.org

실제 예제를 살펴보기 전에, Unitils를 사용할 수 있는 환경을 먼저 구축해보자. Unitils를 사용하기 위해 Unitils 라이브러리를 사이트(www.unitils.org)에서 내려받는다. 참고로 필자는, `unitils-3.1-with-dependencies.zip` 파일을 내려받았다. 그 다음 압축을 풀어보면 다음과 같은 하위 폴더들이 나타날 것이다. 폴더 이름만 봐도 Unitils가 어떤 부분을 지원해주고 있는지 대략 유추가 가능하다.

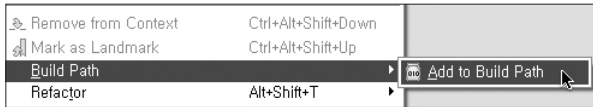
```
unitils-core
unitils-database
unitils-dbmaintainer
unitils-dbunit
unitils-easymock
unitils-inject
unitils-mock
unitils-orm
unitils-spring
unitils-testng
```

다운로드 받은 unitils 압축파일의 내부 모습

그중 핵심이 되는 부분은 unitils-core이다. 다음 실습을 따라 하려면, unitils-core 폴더와 unitils-core/lib 폴더 안의 라이브러리를 클래스패스에 포함시켜 줘야 한다.



위 라이브러리들을 빌드 경로(Build Path)에 추가한다.



그리고 소스에서는 JUnit 4와 마찬가지로 static import를 주로 사용한다. 이를테면 다음과 같은 식으로 말이다.

```
| import static org.unitils.reflectionassert.ReflectionAssert.*;
```

JUnit 4의 static 메소드들을 자동완성 기능으로 사용하기 위해 favorite으로 등록하는 방법을 2장에서 설명했었는데, 기억날지 모르겠다. Unitils의 static 메소드들도 같은 식으로 등록해놓으면 편리하게 사용할 수 있다.

6.2 Unitils의 단위 테스트 지원 기능들

객체 동치성 비교

TDD를 위해 테스트 케이스를 작성하다 보면, 객체끼리 동치성¹ 비교를 해야 하는 경우가 흔히 발생한다. 그런데, 이때 assertEquals만으로는 원하는 답이 나오지 않는 경우가 많다. 다음은 Book이라는 클래스와 그 클래스를 기반으로 만들어진 두 객체에 대한 동치성 비교 문장이다.

Book 클래스

```
public class Book {
    private String name;
    private String author;
    private int price;

    public Book(String name, String author, int price) {
        this.name = name;
        this.author = author;
        this.price = price;
    }
    ...
}
```

두 Book 객체를 비교하는 테스트 케이스

```
@Test
public void testBook() throws Exception {
    Book aBook = new Book("사람은 무엇으로 사는가?", "톨스토이", 9000);
```

1 **동일성과 동치성**: 두 객체를 비교할 때 흔히 생각해야 하는 부분은 동일성과 동치성이다. 동일성은 같은 객체인가를 판단하는 것이고, 동치성은 해당 객체가 표현하고자 하는 상태가 서로 일치하는가를 따져보는 것이다. 필요한 만큼의 레벨로 사물을 추상화하는 소프트웨어 설계에서는 특히 동치성 개념이 많이 사용된다. 이름, 나이, 주민번호 등의 내부 정보가 같은 두 객체가 서로 다른 객체라고 판단하게 되면 오히려 곤란해지는 경우가 소프트웨어 설계에서는 종종 발생한다. 그래서 객체 간의 비교에서는 본인이 하려고 하는 게 동치성 검사인지, 동일성 검사인지 잘 생각해둬야 한다. 이에 대해서는 '7장. 개발 영역에 따른 TDD 작성 패턴'에서 좀 더 자세히 다룬다.

```
Book otherBook = new Book("사람은 무엇으로 사는가?", "톨스토이", 9000);
assertEquals(aBook, otherBook);
}
```

실행 결과

```
Java.lang.AssertionError: expected:<main.Book@16f0472> but was:<main.
Book@18d107f>
    at org.junit.Assert.fail(Assert.java:91)
```

이럴테면 이런 종류의 객체 비교는 단순히 참조 비교를 하기 때문에 두 aBook과 otherBook은 의도된 객체의 상태라는 측면에서는 동일하지만, JUnit의 assertEquals 결과는 두 객체가 서로 다르다고 판정한다.

```
@Test
public void testBook() throws Exception {
    Book aBook = new Book("사람은 무엇으로 사는가?", "톨스토이", 9000);
    Book otherBook = new Book("사람은 무엇으로 사는가?", "톨스토이", 9000);

    assertEquals(aBook.getName(), otherBook.getName());
    assertEquals(aBook.getAuthor(), otherBook.getAuthor());
    assertEquals(aBook.getPrice(), otherBook.getPrice());
}
```

그래서 위와 같은 식으로 각 필드를 비교해야 한다. 하지만 필드가 많아지면 여간 불편한 게 아니다. Unitils는 이런 불편함을 해소하기 위해, 객체의 필드에 저장되어 있는 값을 알아서 비교해주는 리플렉션 단정문(reflection assertion)이라는 기능을 제공한다.

리플렉션 단정문(Reflection Assertion)

```
assertReflectionEquals(예상 객체, 실제 객체);
assertReflectionEquals([메시지], 예상 객체, 실제 객체);
```

형식은 일반적으로 사용되는 assertEquals와 동일하며, 마찬가지로 첫 번째 인자로 ‘메시지’를 적을 수도 있다. 실제 적용 시에는 가급적이면 의도를 확실히 하기 위해 assert의 의도를 설명해주는 ‘메시지’를 적는 걸 권장한다.

```
import static org.junit.reflectionassert.ReflectionAssert.*;
...
public class BookTest {

    @Test
    public void testBook() throws Exception {
        Book aBook = new Book("사람은 무엇으로 사는가?", "톨스토이", 9000);
        Book otherBook = new Book("사람은 무엇으로 사는가?", "톨스토이", 9000);

        assertReflectionEquals("Book 객체 필드 비교", aBook, otherBook);
    }
    ...
}
```

리플렉션 단정문을 사용하면 테스트가 깔끔하게 성공한다.

Unitils의 assertEquals를 사용하면 이런 식으로 객체의 동치성을 증명해 준다. 이것 Unitils에서는 리플렉션을 이용한 단정(reflection assertion)이라고 부른다. Unitils의 이런 리플렉션 단정문에 몇 가지 옵션을 지정할 수 있는데, 대부분은 소스코드가 리팩토링이 일어나면서 발생하는 테스트 케이스의 깨짐(fragility of test)을 보완하기 위해 사용된다. 이 방식을 너그러운 단정문(lenient assertions) 적용이라고 부른다.

리플렉션 단정문의 너그러운 비교(Lenient Assertion)

```
assertReflectionEquals(예상 객체, 실제 객체, ReflectionComparatorMode);
```

앞서 본 assertEquals의 맨 마지막에 ReflectionComparatorMode(리플렉션 비교모드)라는 옵션이 하나 더 붙어 있는 모습이다. ReflectionComparatorMode는

리플렉션으로 대상을 비교할 때 좀 더 유연한 비교를 할 수 있도록 다음과 같은 세 가지 옵션을 제공한다.

ReflectionComparatorMode	설명
LENIENT_ORDER	컬렉션이나 배열을 비교할 때 순서는 무시한다.
IGNORE_DEFAULTS	예상 객체의 필드 중 타입 기본값을 갖는 필드에 대해서는 비교를 하지 않는다.
LENIENT_DATES	시간이나 날짜 타입은 서로 비교하지 않는다.

LENIENT_ORDER 너그러운 순서

배열 객체를 서로 비교하게 될 때 순서가 다르면 다른 객체로 판정한다. 그런데 경우에 따라서 배열 안에 들어 있는 원소들이 중요하지, 들어가 있는 순서는 상관없는 경우가 있다. 버스에 아이들이 어떤 순서로 탔는지보다는 타야 하는 아이들이 모두 예상대로 탑승했는지를 판단할 경우가 그러하다. 이럴 때 LENIENT_ORDER라는 옵션을 사용한다.

```
List<Integer> myList = Arrays.asList(3, 2, 1);  
assertReflectionEquals(Arrays.asList(1, 2, 3), myList, LENIENT_ORDER);
```

참고로 ReflectionComparatorMode를 사용하려면 org.unitils.reflectionassert.ReflectionComparatorMode 클래스를 static import로 추가해줘야 한다.

IGNORE_DEFAULTS 필드 기본값 무시

Java의 타입 기본값, 이를테면 int 타입의 0, 객체의 null, boolean의 false는 각 타입의 기본값이다. assertReflectionEquals로 두 객체의 동치성을 비교할 때, 만일 필드의 값에 위에 같은 타입 기본값이 할당되어 있을 경우 동치성 판단 여부에 포함시키지 않는다.

```
Item expectedItem = new Item("IKH-001", null, 24000);  
Item actualItem   = new Item("IKH-001", "20040601", 24000);  
assertReflectionEquals(expectedItem, actualItem, IGNORE_DEFAULTS);
```

위와 같은 경우에는 Item 객체의 두 번째 필드 값에 할당하는 값은 동치성 비교에 포함되지 않도록 해줬기 때문에 `assertReflectionEquals`는 참(true)이 되고 테스트는 성공(success)한다. `IGNORE_DEFAULTS` 옵션이 없었다면 테스트가 실패하고 붉은색 막대(false)가 나타났을 것이다. 이때 유의할 점이 하나 있는데, 필드 기본값을 비교에서 제외하는 건 `expected` 객체를 기준으로 한다는 점이다. 만일 위 예제에서 `actualItem`과 `expectedItem`의 순서를 바꿔서 테스트했다면, 실패로 단정한다.

```
Item expectedItem = new Item("IKH-001", "20040601", 24000);
Item actualItem = new Item("IKH-001", null, 24000);
assertReflectionEquals(expectedItem, actualItem, IGNORE_DEFAULTS);
```

즉, 위와 같이 `expected`에는 type 기본값이 없지만, `actual`에는 0이나 null 같은 타입 기본값이 존재한다면, 테스트는 아래와 같은 메시지와 함께 실패로 판정된다.

```
junit.framework.AssertionFailedError:
Expected: Item<productID="IKH-001", inhouseDate="20040601", stock=24000>
  Actual: Item<productID="IKH-001", inhouseDate=null, stock=24000>

--- Found following differences ---
inhouseDate: expected: "20040601", actual: null

--- Difference detail tree ---
  expected: Item<productID="IKH-001", inhouseDate="20040601", stock=24000>
    actual: Item<productID="IKH-001", inhouseDate=null, stock=24000>

inhouseDate expected: "20040601"
inhouseDate actual: null
```

미묘한 부분이긴 한데, 어쨌든 `IGNORE_DEFAULTS` 옵션을 사용한다면 `expected`와 `actual`의 위치를 지켜줘야 한다.

LENIENT_DATES 너그러운 날짜

객체값을 비교할 때 종종 테스트를 만들기가 어려운 경우는, 로그성 날짜가 들어가는 필드값이 쓰일 때다. 이를테면 주문일자, 등록일자, 접수시간 등의 필드는 실제 해당 값이 생성되는 시점의 '시스템 타임'이 값으로 채워지는 경우가 많다. 그런데 테스트 케이스를 만들어서 수행하는 경우에는 테스트할 때마다 시스템 시간이 달라지기 때문에 예상값을 특정해놓기가 어렵다. 그래서 이런저런 편법을 사용해 날짜 비교를 무시하도록 유도하곤 하는데, Unitils에서는 해당 부분을 LENIENT_DATES라는 옵션으로 해결해주고 있다.

```
Item expectedItem = new Item("IKH-001", null, 24000,
                             new Date(System.currentTimeMillis()+100));
Item actualItem = new Item("IKH-001", null, 24000,
                           new Date(System.currentTimeMillis()));
assertReflectionEquals(expectedItem, actualItem, LENIENT_DATES);
```

위 경우, 두 객체의 필드값을 비교할 때 Date 타입에 해당하는 필드는 동치성 비교에서 제외한다. 즉, 위 테스트 케이스는 녹색 막대로 표시된다.

흔히 날짜/시간 등은 비교 시 불편한 점이 많아, Date 타입을 만들어 사용하기보다는 SimpleDateFormat 등을 이용해 String으로 변환해서 사용하는 경우가 종종 있다. 이럴 경우 LENIENT_DATES를 사용하면 String으로 굳이 변환하는 작업을 줄일 수 있고, DB 등을 이용하지 않고 시스템 내에서 날짜 관련 작업을 처리해야 하는 경우에 유용한 옵션이다.

참고로 이야기하자면, Unitils에는 `assertLenientEquals`라는 것이 있는데, `ReflectionComparatorMode`를 사용하는 `assertReflectionEquals`의 간략화 버전으로, 위에 소개한 세 개의 옵션 중 LENIENT_DATES를 제외한 두 개를 동시에 적용해서 비교하는 버전이다.

```
// 컬렉션의 순서가 다른 경우
List<Integer> bag = Arrays.asList(100, 200, 300);
assertLenientEquals(Arrays.asList(300, 200, 100), bag);
```

```

// 배열의 순서가 다른 경우
assertLenientEquals(new String[]{"a", "B", "c"}, new String[]{"B",
"c","a"});

// 필드값이 타입 기본값일 경우 비교에서 제외
Item expectedItem = new Item("IKH-001", null, 24000);
Item actualItem = new Item("IKH-001", "20040601", 24000);
assertLenientEquals(expectedItem, actualItem);

```

위 세 개의 단정문은 모두 참이 된다.

프로퍼티 단정문(Property Assertions)

보통, 객체의 특정 필드에 예상하는 값이 제대로 할당됐는지 확인하는 가장 간단한 방법은 getter 메소드를 통해 해당 필드 변수에 들어가 있는 값을 직접 확인해보는 것이다.

```

@Test
public void testLoadPlayerTest() throws Exception {
    // 저장소에서 주장 캐릭터의 정보를 불러온다.
    Player player = VolleyballTeamRepository.getCaptain();
    assertEquals("Ku Min-jung", player.getName());
}

```

이런 식으로 비교를 하면 된다. 그런데 경우에 따라서는 getter 메소드가 제공되지 않는 경우도 있다. 아래는 배구 게임의 선수에 해당하는 Player 클래스의 모습이라고 가정해보자.

```

public class Player {
    private String name;
    private int age; // 나이
    private int experienceYear; // 경력

    public Player(String name, int age, int experienceYear) {

```

```

        this.name = name;
        this.age = age;
        this.experienceYear = experienceYear;
    }

    public String getName() {
        return this.name;
    }

    public int getAbilityPoint(){ // 나이 30이 넘으면 능력이 떨어진다? --;
        return (30 - this.age) + experienceYear;
    }
}

```

나이와 경력 값이 제대로 할당되는지를 확인하고 싶은데 현재 해당 값에 접근할 수 있게 하는 getter가 없다. 뭐, 그냥 Player 클래스에 getter를 만들어도 되긴 되지만, 다음 같은 상황 중 하나라고 가정해보자.

- 테스트할 때 외에는 해당 getter 메소드를 사용할 일이 없다.
- 외부에서 받은 라이브러리라서 Player 클래스를 수정할 수 없다.

이런 경우 Java의 강력한 기능 중 하나인 리플렉션을 이용하면 소스를 수정하지 않고도 문제 상황을 돌파할 수 있다. 직접 만들 수도 있는데, 부디 그러지 말고 Unitils의 `assertPropertyLenientEquals`를 이용하자.²

`assertPropertyLenientEquals(속성 이름, 예상되는 속성 값, 실제 객체)`

객체의 특정 필드값(속성값)만을 비교하고자 할 때 사용하는 기능이다.

² 우린 이미 충분함을 넘어설 만큼 많은 수레바퀴를 다시 발명하고 있다.

출처: Post Modern Programming, <http://www.postmodernprogramming.org/>

```

@Test
public void testPlayerPropertyTest() throws Exception {
    Player player = VolleyballTeamRepository.getCaptain();
    assertPropertyLenientEquals("age", 31, player);
    assertPropertyLenientEquals("experienceYear", 15, player);
}

```

getter 메소드가 없지만 비교가 가능하다. `assertPropertyLenientEquals`의 장점 중 하나는 자바빈(JavaBeans) 규칙을 따르는 식으로 프로퍼티를 비교한다는 점이다. 무슨 말인가 하면, 만일 추후 `Player` 클래스 내에 getter 메소드가 생기게 된다면, `assertPropertyLenientEquals`는 해당 bean 규약에 맞는 getter 메소드를 이용해 프로퍼티 값을 불러와서 비교를 수행한다는 뜻이다. 단순히 리플렉션으로 클래스의 필드 변수값만을 가져와서 비교하는 수준을 넘어서고 있다.



프레임워크, 프레임워크, 프레임워크

세상에 무궁무진한 프레임워크들이 있다. 특히 Java 언어는 다른 모든 프로그래밍 언어에 존재하는 프레임워크를 다 합친 것보다 더 많은 프레임워크가 존재하는 언어다. 어떤 이들은 그게 Java 언어가 아직 많이 부족하다는 하나의 증거라고 이야기한다. 프레임워크가 좋은 건 알겠지만, 그렇다고 다 배울 순 없지 않은가? 그래서 이 책에서도 분량이 넘치지 않는 범위 내에서, 정말 알면 유용하다 생각되는 것 위주로 설명을 하고 있다. 투자 대비 효과를 높일 수 있도록 노력할 테니, 힘들더라도 따라와 줬으면 좋겠다.

다양한 웹 애플리케이션 프레임워크

Echo	Cocoon	Millstone	OXF
Struts	SOFIA	Tapestry	WebWork
RIFE	Spring MVC	Canyamo	Maverick
JPublish	JATO	Folium	Jucas
Verge	Niggle	Bishop	Barracuda
Action Framework	Shocks	TeaServlet	wingS
Espresso	Bento	jStatemachine	jZonic
OpenEmcee	Turbine	Scope	Warfare

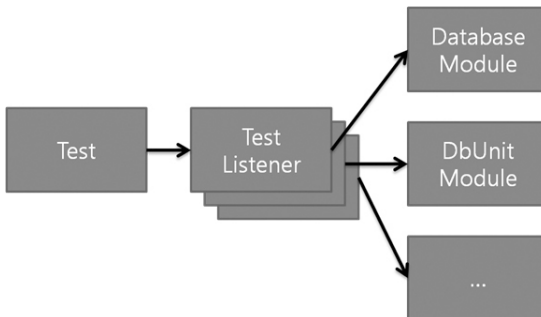
JWAA	Jaffa	Jacquard	Macaw
Smile	MyFaces	Chiba	JBanana
Jeenius	JWarp	Genie	Melati
Dovetail	Cameleon	JFormular	Xoplon
Japple	Helma	Dinamica	WebOnSwing
Nacho	Cassandra	Baritus	Stripes
Click	GWT	Wicket	

출처: 아파치 위켓(Apache Wicket) 홈페이지(<http://wicket.apache.org>)

중급 이상 도전!

6.3 Unitils 모듈

일반적인 수준에서는 앞에서 설명한 Unitils의 기능만으로도 테스트 케이스를 작성하는 데 도움이 많이 된다. 하지만 Unitils는 이 외에도 DB 관련 기능, Hibernate, JPA, Spring, Mock Object 등에 대해서도 유용한 기능을 모듈(module)이라는 개념으로 제공하고 있다.



Unitils의 모듈 지원 구성도

Unitils의 모듈은 다양한 서비스를 제공하는데, 테스트 케이스 작성 시에 해당 모듈을 사용하려면 UnitilsJUnit3, UnitilsJUnit4, UnitilsTestNG 등의 Test Listener³가 필요하다. Test Listener를 사용하는 방법은 해당 Unitils의 Test Listener를 테스트 클래스가 상속하거나 @RunWith 같은 애노테이션으로 Test Runner로 지정하면 된다.

JUnit 3의 경우, 상속을 이용한다.

```
import org.unitils.UnitilsJUnit3;

public class MyTest extends UnitilsJUnit3 {
}
```

JUnit 4의 경우, UnitilsJUnit4 클래스를 상속해도 되고 위와 같이 Runner를 지정해도 된다.

```
import org.junit.runner.RunWith;
import org.unitils.UnitilsJUnit4TestClassRunner;

@RunWith(UnitilsJUnit4TestClassRunner.class)
public class MyTest {
}
```

현재 Unitils에서 제공하는 모듈은 다음과 같다.

DatabaseModule	데이터베이스 관리와 커백션 풀 관련
DbUnitModule	DbUnit을 사용할 때 사용하는 테스트 데이터 관리
HibernateModule	하이버네이트 설정 지원과 DB 매핑 체크
MockModule	Unitils에서 제공하는 Mock 프레임워크
EasyMockModule	EasyMock 지원 기능
InjectModule	오브젝트를 강제로 할당시켜 버리는 Injection 기능
SpringModule	스프링의 애플리케이션 컨텍스트 지원과 스프링 빈(bears)의 주입 기능 지원

³ 테스트 케이스를 실행하는 일종의 테스트 러너(Test Runner)에 해당한다.

이 중에서 우리는 몇 가지 부분만 살펴볼 예정이다. 우선 데이터베이스 관련 기능 지원을 살펴보자. 다음 소스는 DbUnit을 설명할 때 사용했던 판매자 관리 기능을 테스트하던 클래스다.

```
public class RepositoryTest {
    private final String driver = "org.apache.derby.jdbc.EmbeddedDriver";
    private final String protocol = "jdbc:derby:";
    private final String dbName = "shopdb";

    private IDatabaseTester databaseTester;
    private IDatabaseConnection connection;

    @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);
    }

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

    @Test
    public void testFindById() throws Exception {
        Repository repository = new DatabaseRepository();
        Seller actualSeller = repository.findById("horichoi");

        assertPropertyLenientEquals("id", "horichoi", actualSeller);
        assertPropertyLenientEquals("name", "최승호", actualSeller);
        assertPropertyLenientEquals("email", "megaseller@hatmail.com",
            actualSeller);
    }
}
```

앞으로 이전 페이지의 내용 기반으로 설명을 진행할 예정이다.

6.4 DbUnit과 함께 사용하는 데이터베이스 지원 모듈

DbUnit은 데이터베이스 단위 테스트에 매우 유용하다. Unitils는 DbUnit을 좀 더 편하게 사용할 수 있도록 편리한 기능들을 제공한다. 어떠한 기능을 제공하는지 한 번 살펴보자.

환경 준비를 위한 unitils.properties 파일 설정

Unitils에서 DB 접속 기능을 지원받기 위해 클래스패스 내에 unitils.properties라는 이름의 텍스트 파일을 하나 만들자(src 폴더 바로 아래에 놓아도 된다). 다음은 unitils.properties 파일의 모습이다.

```
database.driverClassName=org.apache.derby.jdbc.EmbeddedDriver
database.url=jdbc:derby:shopdb
database.userName=
database.password=
database.schemaNames=APP
database.dialect=derby
DatabaseModule.Transaction.value.default=disabled
```

driverClassName이나 url은 계속 사용해왔던 부분이다. 만일 오라클 DB를 사용했다면 이 부분은 다음과 같을 것이다.

```
database.driverClassName=oracle.jdbc.driver.OracleDriver
database.url=jdbc:oracle:thin:@yourmachine:1521:YOUR_DB
```

Derby DB 임베디드 모드의 기본 스키마 이름은 APP이다. database.dialect에는 Unitils에서 DB에 맞는 내부 SQL을 사용할 수 있게 DB 종류를 적어줬다. 마지막으로 현재 애플리케이션은 자동커밋(auto-commit) 기준으로 작성됐기에, Unitils가 트랜잭션 처리를 하지 않도록 disabled로 지정해줬다. 참고로, Unitils에서 DB 사용 시의 기본 트랜잭션 관리는 커밋이다.

@DataSet

Unitils는 @DataSet이라는 애노테이션을 지원하는데, 클래스이름.xml이라는 파일을 기본 데이터셋으로 인식하고 DB로 읽어들인다. 이때 데이터셋 파일은 FlatXmlDataSet 타입이어야 한다. DB로 읽어들이는 때의 기본 동작은 모두 삭제하고 집어넣는 CLEAN_INSERT이다. 위 예제 소스를 Unitils의 @DataSet을 이용해 작성하면 다음과 같다.

```
@RunWith(UnitilsJUnit4TestClassRunner.class) // ❶
@DataSet // ❷
public class DatabaseRepositoryTest {

    @Test
    public void testFindById() throws Exception {
        Repository repository = new DatabaseRepository();
        Seller actualSeller = repository.findById("horichoi");

        assertEquals("id", "horichoi", actualSeller); // ❸
        assertEquals("name", "최승호", actualSeller);
        assertEquals("email", "megaseller@hatmail.com",
            actualSeller);
    }
}
```

- ❶ Unitils 모듈의 서비스를 이용하기 위해 Test Listener를 지정했다.
- ❷ 클래스 레벨에서 @DataSet 애노테이션이 사용되면 클래스이름.xml 파일을 데이터셋으로 인식한다. 따라서 이 예제에서는 DatabaseRepositoryTest.xml 파일을 읽어들이게 되어 있다. seller.xml 파일을 테스트 클래스가 있는 위치에 DatabaseRepositoryTest.xml라는 이름으로 복사해놓자.
- ❸ actualSeller.getId()가 horichoi인지 equals 비교를 한다.

앞 소스에 비해 테스트 클래스가 좀 더 간결해졌다. 또한 데이터셋과 클래스 파일의 이름을 일치시킨다는 관례적인 규칙(conventional rule)으로 데이터셋을 관리하니까 일

관성 측면에서도 더 나아졌다. 현재 DatabaseRepositoryTest.xml 파일의 모습은 다음과 같다.

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <seller ID="horichoi" NAME="최승호" EMAIL="megaseller@hatmail.com"/>
  <seller ID="buymore" NAME="김용진" EMAIL="shopper@nineseller.com"/>
  <seller ID="mattwhew" NAME="이종수" EMAIL="admin@maximumsale.net"/>
</dataset>
```

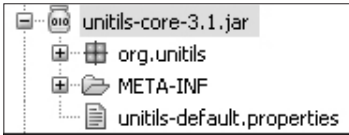
만일 클래스 이름과 데이터셋 파일 이름이 일치하지 않는다면 직접 지정해줄 수도 있고, 필요하다면 여러 개의 데이터셋을 지정할 수도 있다.

```
@DataSet("seller.xml")
@DataSet("seller.xml", "item.xml")
```

그리고 클래스 레벨로 데이터셋을 지정하는 것이 아니라, 메소드 레벨에서 데이터셋을 지정할 수도 있다.

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

하지만 메소드 레벨로 데이터셋을 지정하는 경우가 많아지면 데이터셋 관리가 또 하나의 부담이 될 수 있으니 유의해서 사용하자. 그리고 FlatXMLDataSet 외의 데이터셋 타입, 이를테면 XlsDataSet 타입이나 csv 파일 등을 이용하고 싶다면, DataSetFactory 인터페이스를 직접 구현한 다음, 설정파일에 DbUnitModule.DataSet.factory.default 항목으로 지정하면 된다. 다음은 unitils-core 라이브러리 안에 unitils의 기본 설정과 일이 들어 있는 모습과, 그 설정파일 안에 들어 있는 기본 데이터셋 타입이 지정된 모습이다. Unitils-default.properties 파일을 직접 수정할 일은 없지만 기본 세팅을 확인할 때 사용한다.



Unitils 코어 라이브러리 안에 들어가 있는 기본 설정파일

기본 설정파일 안에 들어 있는 기본 데이터셋 형식 지정 부분

```
# Default factory that is used to create a dataset object from a file for  
the @DataSet annotation  
DbUnitModule.DataSet.factory.default=org.unitils.dbunit.datasetfactory.  
impl.MultiSchemaXmlDataSetFactory
```

직접 만든 DataSetFactory의 구현체에 이 부분을 직접 적어놓을 순 없으니까, 앞서와 마찬가지로 클래스패스 내에 있는 unitils.properties 파일에 적어서 오버라이드되도록 만들면 된다.

데이터셋 로드 전략

앞서 DbUnit을 다룰 때 이야기했지만, 데이터셋을 사용하는 동작(operation)에는 몇 가지 종류가 있었다. @DataSet 애노테이션은 CLEAN_INSERT가 기본 동작이지만, 원할 경우 다른 동작을 지정할 수 있다. 두 가지 방식이 있는데, 설정파일(unitils.properties)에 지정하는 방법과 애노테이션 사용 시에 지정하는 방법이다.

unitils.properties에 설정한 경우

```
DbUnitModule.DataSet.loadStrategy.default=org.unitils.dbunit.  
datasetloadstrategy.InsertLoadStrategy
```

애노테이션으로 지정한 경우

```
@DataSet(loadStrategy = InsertLoadStrategy.class)
```

Unitils에서 지원하는 데이터셋 지원 전략은 다음과 같다(DbUnit의 모든 동작을 지원하지는 않는다).

CleanInsertLoadStrategy	대상 테이블의 내용을 모두 지우고 데이터셋의 내용을 INSERT한다.
InsertLoadStrategy	데이터셋을 INSERT만 한다.
RefreshLoadStrategy	데이터셋의 내용으로 DB를 갱신한다. 이미 존재하는 데이터는 UPDATE, 없는 데이터는 INSERT한다.
UpdateLoadStrategy	DB에 존재하는 데이터를 UPDATE한다.

동작과 관련해서는 필요 시 DbUnit의 DatabaseOperation 항목을 살펴보길 권한다.

@TestDataSource, 테스트에 사용하는 데이터소스 접근하기

Unitils는 설정파일에 지정된 DB 관련 값을 이용해 DataSource를 구성해 자동으로 DB 연결을 만들어 테스트를 진행하도록 설계되어 있다. 경우에 따라서는 DataSource에 직접 접근해야 할 필요가 있을 수 있다. 이럴 때 사용할 수 있는 방법은 두 가지로, @TestDataSource 애노테이션을 DataSource에 지정해서 Unitils가 자동으로 주입(injection)하도록 만드는 방법과 DatabaseUnitils.getDataSource() 메소드를 이용하는 방법이다.

@TestDataSource 애노테이션을 이용해 Unitils가 직접 자신의 DataSource를 주입하도록 만든 모습

```
public class ShopDAO0Test extends UnitilsJUnit4 {

    @TestDataSource
    private DataSource dataSource;
    private ShopDAO dao

    @Before
    public void initialize () {
        this.dao = new ShopDAO();
        dao.setDataSource(dataSource);
    }
}
```

```
...  
}
```

DatabaseUnitils.getDataSource()를 이용해 DataSource를 얻어오는 모습

```
dao.setDataSource( DatabaseUnitils.getDataSource() );
```

@ExpectedDataSet, 예상 데이터셋을 이용한 테스트 메소드 레벨의 결과 비교

때로는 결과값을 비교해야 할 때도 있다. DbUnit에서는 예상값을 마찬가지로 데이터셋으로 만들어서 상태를 비교했다. Unitils에서는 이럴 때 사용할 수 있도록 @ExpectedDataSet이라는 애노테이션을 제공한다. 예를 보자. 다음은 DbUnit 설명서에 사용했던 예제와 동일한 예제다. 새로운 판매자를 입력한 다음, 정상적으로 입력됐는지 확인하는 테스트다.

```
@Test  
public void testAddNewSeller() throws Exception {  
    Seller newSeller = new Seller("hssm", "이동욱", "scala@hssm.kr");  
    Repository repository = new DatabaseRepository();  
    repository.add(newSeller);  
  
    ITable actualTable = connection.createQueryTable("SELLER", "select *  
        from seller");  
    IDataSet expectedDataSet = new FlatXmlDataSetBuilder().build(  
        new File("expected_seller.xml"));  
    ITable expectedTable = expectedDataSet.getTable("seller");  
  
    Assertion.assertEquals(expectedTable, actualTable);  
}
```

판매자를 추가한 다음 DB에서 데이터셋을 추출해내고, 그걸 미리 만들어놓았던 expected_seller.xml의 테이블과 비교하고 있다. 위 코드를 @ExpectedDataSet 애노테이션을 이용해 고치면 다음과 같아진다.

```

@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);
}

```

테스트 케이스가 훨씬 더 간결해졌다. 만일 직접 예상 데이터셋을 지정하지 않는다면, `Unitils`는 기본적으로 '클래스이름.메소드이름-result.xml'이라는 데이터셋 파일을 찾는다. 위 예제의 경우라면 `RepositoryTest.testAddNewSeller-result.xml` 파일을 예상 데이터셋으로 찾았을 것이다. 단, 주의할 점은 앞서도 이야기했지만, 메소드 레벨의 데이터셋을 사용하면 자칫 데이터셋 파일이 많아지는 불우한 상황이 벌어질 수 있음을 유의하자.

@Transactional, 트랜잭션 처리

DB 관련 기능을 테스트할 때, 트랜잭션 처리⁴가 필요한 경우가 많다.

- 업무 로직상 트랜잭션 기능 자체를 테스트해야 하는 경우
- 테스트 시 변경된 데이터가 테스트 종료 후에도 그 상태로 남게 하지 않으려고 롤백을 사용하는 경우
- SELECT FOR UPDATE처럼 트랜잭션 처리를 해야 제대로 동작하는 기능을 사용해야 할 경우
- 즉시 자동커밋(immediate auto-commit)으로 운영하기 어려운 제품을 사용하는 경우. 대표적인 예로 하이버네이트나 JPA, TOP-LINK 등을 사용할 때

4 트랜잭션이란 하나의 업무가 완결성을 갖고 정상처리되기 위해 필요한 하위 업무나 기능들의 집합을 말한다. 중간에 하나의 단위 업무라도 정상적으로 동작하지 않는 일이 발생하면 진행되고 있던 모든 기능이나 단위 업무가 작업 시작 시점의 상태로 돌아가야 한다. 흔히는 협소한 의미로, 하나의 커다란 단위로 묶어서 동작하는 UPDATE, INSERT, DELETE 같은 데이터베이스의 DML 문장들이 모두 성공하면 커밋, 실패하면 롤백으로 돌아가는 상황을 지칭하곤 한다.

이럴 때는 트랜잭션 기능을 사용해야 하는데, Unitils에서 마찬가지로 쉽게 처리할 수 있는 기능을 제공한다.⁵

기본적으로 Unitils의 DB 관련 기능을 이용해 테스트를 수행하게 되면, 모든 테스트는 트랜잭션이 발생하는 상태로 동작하고, 테스트 마지막에 커밋을 발생시킨다. 따라서 자신이 사용하는 DB가 트랜잭션을 지원하는지 살펴보고, 그리고 자신의 테스트 코드가 트랜잭션으로 동작해도 무방한지 고려해봐야 한다. 그 다음엔 어떻게 적용할지를 정해야 한다. 설정파일에 지정하는 방법과 애노테이션을 이용해 클래스 레벨로 지정하는 방법 두 가지가 있다. 지정 가능한 상태는 commit, rollback, disabled 세 가지다. 앞선 예제에서는 트랜잭션 처리를 하지 않게 하려고, 즉 자동커밋 상태를 만들기 위해 unitils의 트랜잭션 기능을 disabled로 만들었다. 다음은 테스트 케이스를 수행한 다음 롤백처리하도록 만드는 예다. 둘 중 하나만 사용하자.

unitils.properties 파일에 설정한 경우

```
DatabaseModule.Transactional.value.default=rollback
```

@Transactional 애노테이션으로 지정한 경우

```
@Transactional(TransactionMode.ROLLBACK)
public class ShopDaoTest extends UnitilsJUnit4 {
    ...
}
```

참고

Derby DB의 임베디드 모드에서는 트랜잭션 처리가 정상적으로 동작하지 않는다.

특이한 건, Unitils는 트랜잭션 관련 기능을 구현하는 데 스프링 프레임워크의 트랜잭션 관리 기능을 전적으로 차용해서 사용했다는 점이다. 의존 라이브러리를 열어보면 spring 라이브러리들이 들어 있는 것을 확인할 수 있다.

unitils-3.1-with-dependencies\unitils-3.1\unitils-database\lib 목록 중 일부

```
...
/TDDWithUnitils/unitilsDbSupport/lib/spring-beans-2.5.2.jar
/TDDWithUnitils/unitilsDbSupport/lib/spring-context-2.5.2.jar
```

⁵ 스프링 프레임워크 유저라면 이와 비슷한 기능을 스프링에서 본 적이 있을 것이다.

```
/TDDWithUnitils/unitilsDbSupport/lib/spring-core-2.5.2.jar
/TDDWithUnitils/unitilsDbSupport/lib/spring-jdbc-2.5.2.jar
/TDDWithUnitils/unitilsDbSupport/lib/spring-tx-2.5.2.jar
...
```

데이터베이스 관련 지원 기능 중 재밌는 것이 하나 있는데, DBMaintainer라는 기능이다.

6.5 DBMaintainer: DB를 자동으로 유지보수해주는 DB 유지보수 관리자

Unitils의 DBMaintainer는 개발자 각자의 DB 스키마를 SQL 스크립트를 이용해 자동으로 유지시켜 주는 기능이다. 기본적으로 동작하는 방식은 다음과 같다.

- 프로젝트 내에 폴더를 하나 만들어서 애플리케이션에서 필요한 DB스크립트를 넣어놓는다.
- 이때 DB스크립트는 숫자 형식의 버전넘버를 _(언더바)로 구분지어 갖도록 이름짓는다.
예: 001_DROP_ALL_TABLES.sql, 002_CREATE_TABLES.sql ...
- Unitils의 DataSource를 이용하는 테스트 클래스를 실행한다.
- DBMaintainer는 지정된 스크립트 폴더를 모니터링해서 변경된 내용이 있으면 반영한다.

DB 구조를 SQL 스크립트로 관리하고, 추가내용을 덧붙여 반영하도록 만들 때 매우 유용하다. 스크립트 폴더에 변경분만 넣어주면 끝나기 때문이다. 모니터링 후 동작하는 방식은 두 가지인데, 만일 새로운 스크립트가 추가된 경우라면 해당 스크립트만 실행한다. 기존 스크립트가 변경됐다면, 스키마를 전체 리셋하고 다시 처음부터 스크립트를 실행한다(해당 유저의 스키마가 리셋된다는 점을 유의하자). 참고로 이때, 번호가

매겨지지 않은 스크립트는 가장 나중에 실행된다. 필요에 따라 스크립트 폴더들도 번호를 붙여 순차적으로 실행되도록 만들 수 있다.

```
dbscripts/ 01_production/ 001_initial.sql
                                002_auditing_updates.sql
02_latest_dev/ 001_add_user_table.sql
                                002_rename_product_id.sql
```

DBMaintainer 기능 활성화시키기

DBMaintainer 기능은 기본적으로는 false 상태로 되어 있기 때문에 해당 기능을 사용하려면 설정파일에 다시 지정해서 true로 변경해줘야 한다.

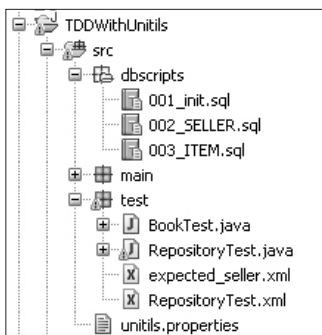
```
updateDataBaseSchema.enabled=true
```

적용 범위는 Unitils의 DataSource를 사용하는 모든 테스트 클래스가 된다.

그리고 스크립트 파일들이 존재하는 폴더를 지정해준다.

```
dbMaintainer.script.locations=src/dbscripts
```

실행 예



dbscripts라는 DB 관리용 스크립트 폴더를 만들었다. 원할 경우 하위 폴더들도 숫자를 붙여 만들 수 있다.

DBMaintainer를 위해 unitils.properties에 추가된 내용

```
dataSetStructureGenerator.xsd.dirName=xsd

updateDataBaseSchema.enabled=true
dbMaintainer.script.locations=src/dbscripts
dbMaintainer.autoCreateExecutedScriptsTable=true
```

dbMaintainer.autoCreateExecutedScriptsTable은 DBMaintainer가 스크립트들을 버전관리하기 위해 사용하는 테이블이다. 최초에는 없기 때문에 자동으로 만들도록 지정해줬다. 프로젝트 홈 아래에 xsd라는 폴더를 만들어서 dataSetStructureGenerator 위치로 지정해줬다. 해당 폴더에 DB 스키마의 구조를 만들어준다.

아래 내용은 앞에서 작성한 판매자 테스트를 실행했을 때 콘솔에 찍힌 내용이다. 스크립트가 실행되는 모습을 볼 수 있다.

```
2010. 2. 17 오전 12:24:26 org.unitils.database.DatabaseModule updateDatabase
정보: Checking if database has to be updated.
2010. 2. 17 오전 12:24:27 org.unitils.dbmaintainer.DBMaintainer
updateDatabase
정보: Database update scripts have been found and will be executed on the
database.
2010. 2. 17 오전 12:24:27 org.unitils.dbmaintainer.clean.impl.
DefaultDBCleaner cleanSchemas
정보: Cleaning database schema APP
2010. 2. 17 오전 12:24:27 org.unitils.dbmaintainer.DBMaintainer
executeScripts
정보: Executing script 001_init.sql
2010. 2. 17 오전 12:24:28 org.unitils.dbmaintainer.DBMaintainer
executeScripts
정보: Executing script 002_SELLER.sql
2010. 2. 17 오전 12:24:28 org.unitils.dbmaintainer.DBMaintainer
executeScripts
정보: Executing script 003_ITEM.sql
2010. 2. 17 오전 12:24:28 org.unitils.dbmaintainer.structure.impl.
DefaultConstraintsDisabler disableConstraints
```

정보: Disabling constraints in database schema APP

2010. 2. 17 오전 12:24:28 org.unitils.dbmaintainer.structure.impl.

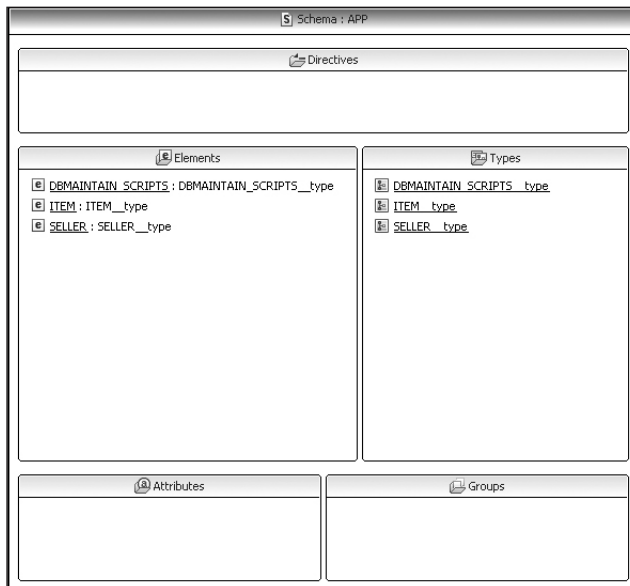
DefaultSequenceUpdater updateSequences

정보: Updating sequences and identity columns in database schema APP

DBMaintainer는 단순히 스키마의 데이터를 업데이트하는 것과는 좀 다르다.

추가적으로 다음과 같은 기능을 함께 제공한다.

- 모든 참조키와 Not Null 제약조건을 비활성화(disable)시키기 때문에 INSERT 스크립트 실행을 편리하게 만들어준다.
- 시퀀스(sequence) 타입들을 높은 숫자로 업데이트시켜 주기 때문에, 테스트 수행 시에 넣게 되는 데이터들의 기본키(primary key) 값을 고정값으로 지정해도 충돌이 나지 않도록 도와준다(시퀀스가 PK 컬럼으로 쓰일 때).
- 데이터베이스의 구조를 XSD(XML Schema Definition) 파일로 만들어준다. 해당 파일은 이클립스에서 GUI 형태로 볼 수 있다.



XSD 파일을 이클립스에서 열어본 모습. 각각의 테이블 구조도 볼 수 있다.

현재 DBMaintainer가 지원하는 DBMS의 종류와 버전은 다음과 같다.

- Oracle – Tested with version 9i, 10g, 10XE
- Hsqldb – Tested with version 1.8.0
- MySQL – Tested with version 5.0
- PostgreSQL – Tested with version 8.2
- DB2 – Tested with version 9
- Derby – Tested with version 10.2.2.0
- MS-SQL Server – Tested with version 2005

출처: unitils 공식 사이트

Unitils는 애플리케이션을 만들 때 가장 괴롭히는 부분 중 하나인 DB 관련 부분에 다양한 기능을 제공하는 걸 알 수 있다. 많이 사용되기 때문에 좀 자세히 다뤘다. 여타 모듈에 대해서는 특징적인 부분만 살펴볼 예정이다.

6.6 기타 지원 모듈들

하이버네이트 지원 모듈

@HibernateSessionFactory

@HibernateSessionFactory 애노테이션으로 SessionFactory를 쉽게 얻을 수 있게 도와준다.

```
@RunWith(UnitilsJUnit4TestClassRunner.class)
public class ShopDaoTest {

    @HibernateSessionFactory({"hibernate.cfg.xml", "mapped-classes.cfg.xml"})
    private SessionFactory sessionFactory;
```

```
    ...  
}
```

하이버네이트 매핑 테스트

실제 데이터베이스의 구조와 하이버네이트 매핑 파일 내의 구조가 일치하는지 테스트 해준다.

```
@HibernateSessionFactory("hibernate.cfg.xml")  
public class HibernateMappingTest extends UnitilsJUnit4 {  
  
    @Test  
    public void testMappingToDatabase() {  
        HibernateUnitils.assertMappingWithDatabaseConsistent();  
    }  
}
```

스프링 지원 모듈

Unitils는 스프링 프레임워크의 애플리케이션 컨텍스트 설정을 읽어오는 부분과 스프링 빈(beans)을 주입시켜 주는 부분을 편리하게 쓸 수 있도록 도와준다.

@SpringApplicationContext, 스프링 애플리케이션 컨텍스트 설정 지원

```
@RunWith(UnitilsJUnit4TestClassRunner.class)  
public class UnitilsMusicPlayerTest {  
  
    @SpringApplicationContext("/context-musicplayer.xml")  
    private ApplicationContext context;  
  
    @Test  
    public void testGetFileName() throws Exception {  
        MusicPlayer player = (MusicPlayer)context.getBean("musicPlayer");  
    }  
}
```

```

        assertEquals("Belong to me.mp3", player.getFileName());
    }
    ...

```

스프링 빈 주입

이름지정, 이름일치, 타입일치, 이렇게 세 개의 애노테이션을 지원해준다.

```

@SpringBean("userMusicPlayer")
private MusicPlayer player;

@SpringBeanByName
private MusicPlayer player;

@SpringBeanByType
private MusicPlayer player;

```

만일 @ContextConfiguration 애노테이션과 @Autowired를 즐겨쓰는 입장이라면 별 장점이 되진 않을 것이다.

Mock 지원 모듈

Unitils는 예전에는 easymock을 지원해주는 수준이었는데, 2.0 버전부터 자체 Mock 프레임워크를 제공하고 있다. 다소 복잡하기도 하거니와 여타 Mock 프레임워크보다 큰 장점을 얻기가 어려워 설명을 생략한다. 일반적인 Mock 프레임워크에 대해서는 3장에서 자세히 다루고 있다.

저자
한·사·디

프레임워크 vs 라이브러리

우리는 지금, TDD를 좀 더 편하게 진행할 수 있도록 도와주는 단위 테스트 프레임워크와 지원 유틸리티(혹은 라이브러리)를 계속 살펴보고 있다. 그러면서 '프레임워크'와 '라이브러리'라는 용어를 계속 쓰고 있는데, 그 둘의 차이점이 무엇인지 잠깐 생각해보자. 둘 다 재사용을 목적으로 하고 있으며, 소프트웨어 개발에서 중요한 부분을 차지한다는 점에선 공통점을 갖는다. 그렇다면 소프트웨어의 세계에선 과연 무엇을 프레임워크라고 부르고, 어떤 걸 라이브러리라고 부르는

결과? (책을 덮고 잠깐 생각해보기!) 일반적으로 이야기하는 기본 정의는 이렇다. 프레임워크는 복잡한 문제를 해결하기 위해 사용되는 기본 개념적인 구조로, 보통 툴이나 컴포넌트들의 집합체를 의미한다. 라이브러리는 소프트웨어를 개발하기 위해 사용되는 클래스나 서브루틴들의 집합체다. 느낌으로도 프레임워크의 범위가 더 크다는 걸 알 수 있다. 하지만 정의만으로는 차이가 뚜렷히 잘 안 느껴지니까, 차이점 위주로 이야기를 다시 해보자. 프레임워크는 층(layer)을 갖고 라이브러리는 API만을 갖는다. 프레임워크는 층을 따른 데이터의 흐름(flow)이 있고, 라이브러리는 값의 IN/OUT에 가깝다. 따라서, 어느 사이트를 갔더니 구조를 층으로 나누어 설명하는 그림이 있으면 프레임워크라고 생각하자. 또한 경우에 따라서는 ‘개발자가 직접 API를 호출하느냐?’ 아니면 ‘개발자가 만든 코드를 시스템이 호출하느냐’에 따라 구분하기도 한다. 이 경우 전자는 라이브러리, 후자는 프레임워크에 해당한다. 물론, 아쉽게도 이건 정의(definition)적인 측면이고, 대부분은 곧잘 혼용해서 사용하며, 둘을 단순히 규모로 나누는 경우도 많다.

구별이나 정의라는 게 알아두면 좋지만, 그렇다고 강박을 가질 필요까진 없으니 이런 걸로 다른 사람을 놀리거나, 스트레스 받진 말자. 세상을 움직인 위대한 소프트웨어는 엄격한 용어와 절묘한 체계로 무장한 학자보다는, 열의와 꿈으로 뭉친 엔지니어의 손에서 더 많이 나왔다는 사실을 잊지 말자. 아 물론, 엄격한 용어와 절묘한 체계를 가진, 꿈과 열의가 가득 찬 엔지니어가 된다면 야 더할 나위 없이 좋겠지만 말이다.

Q. 안녕하세요? 박재성님. 간략한 소개 부탁드립니다. 그리고, 현재 하시는 일이나 근래에 관심 두고 계신 것이 있으시면 함께 이야기해주셨으면 좋겠습니다.

A. 자바지기 커뮤니티를 운영하고 있는 박재성입니다. NHN을 거쳐 현재는 XLGames라는 회사에서 근무하고 있습니다. 하는 일은 웹 서비스 개발 업무 및 프로젝트 관리입니다. 개발자가 행복하게 일할 수 있는 환경을 만드는 데 많은 관심을 갖고 있습니다.

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

A. NHN에서 개발 팀장으로서 테스트 개발 환경과 문화를 만드는 데 집중했습니다. 같이 일했던 개발자들은 TDD를 통해서나 다른 방식으로 많은 테스트 코드를 만들었습니다.

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

A. 실질적으로 TDD를 경험한 것은 2009년 초 TDD 스터디에서였습니다. 이 스터디에서 예제로 만들어본 볼링게임 프로그램이 가장 기억에 남습니다. 똑같은 기능을 몇 번에 걸쳐 TDD로 구현해보면서 점진적으로 설계가 개선되는 것을 느낄 수 있었습니다.

Q. 후배들에게 TDD를 권할 의향이 있으십니까? 있다면 이유는?

A. 당연히 의향 있습니다. 자신이 만드는 코드에 자신감이 생기기 때문입니다. 이러한 자신감은 소스코드를 지속적으로 변화시킬 수 있습니다.

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

A. 최근 소프트웨어 업계의 화두는 누가 뭐라 해도 모바일일 것입니다. 상당히 많은 개발자들이 지금까지 자신이 하던 일을 멈추고 모바일로 뛰어드는 모습을 볼 수 있습니다. 모바일로 뛰어드는 것은 좋지만 정말 그 일이 여러분이 하고 싶은 일인가요? 업계의 변화에 이리저리 휩쓸리면서 정작 자신이 하고 싶은 일과 기본적인 소양을 잃어버리는 경우를 종종 봅니다. 정말 자신이 하고 싶은 일을 찾아서 꾸준히 정진했으면 좋겠습니다. 다소 느린 발걸음이겠지만 시간이 지나 돌아켜보면 더 빠른 지름길이라고 생각합니다. 또한 개발자들을 보면 너무 컴퓨터와의 대화에만 익숙한 경우가 많습니다. 지금도 사람들 간의 협업이 중요한 화두이지만 시간이 지날수록 프로젝트의 성공에 더 중요한 요소가 될 것이라 생각합니다. 소프트웨어 개발도 사람이 하는 일이니만큼 너무 기술적인 부분에만 치우치지 말고 사람 공부도 하면서 소프트웨어 개발에 집중하면 좋겠습니다. 사람을 이해하려면 먼저 마음의 여유를 가질 수 있어야 할 겁니다. 마음의 여유를 가질 수 있는 개발자가 되어 정말 자신이 꿈꾸는 소프트웨어를 만들어낼 수 있기를 기대합니다.

이하에는 팀 내에 테스트 개발 문화를 만들기 위해 노력하셨던 박재성님의 팁을 정리해봤습니다.

팀 내에 테스트 개발 문화 만들기

팀장의 역할

- 단위 테스트의 필요성을 인지하고 적극적으로 지원해줘야 한다.
- 프로젝트 초반에 개발 생산성이 저하될 가능성이 크기 때문에 효과가 나타날 때까지 인내하고 기다려줄 수 있어야 한다.
- 단위 테스트 및 소스코드 품질의 중요성을 강조하고, 잘하는 개발자에게 포상을 해줘야 한다.

팀원의 역할

- 단위 테스트의 필요성을 인지하고 단위 테스트를 만드는 데 적극적으로 참여해야 한다. 필요 없다고 판단된다면 적극적으로 의견 개진한다.
- 모든 개발자가 단위 테스트를 작성해야 한다. 일부 개발자만 참여할 경우 효과가 희석되거나 실패할 가능성이 많다.
- 좋은 단위 테스트 코드를 만들기 위해 끊임없이 공부하고 노력할 자세가 되어 있어야 한다.

팀의 역할

- 단위 테스트의 필요성을 인지하고 팀의 문화로 만들기 위한 지속적인 노력을 한다.
- 팀 구성원끼리 지속적으로 정보를 공유하고 배우려는 문화를 정착시켜야 한다.

하지 말아야 할 일

- 단위 테스트를 내가 원해서가 아니라 위에서 시키니까 한다는 수동적인 자세로 시작할 계획이라면 하지 않는 편이 낫다. 오히려 시간 낭비가 될 가능성이 많다.
- 항상 100%가 실행되지 않는 테스트 코드는 추후 쓰레기가 될 가능성이 높다. 쓰레기가 된 테스트 코드를 유지보수하는 비용은 낭비되는 시간일 뿐이다.

단위 테스트의 효과를 높일 수 있는 팁

- 지속적 통합 툴(CI)을 도입하여 자동화된 단위 테스트가 가능하도록 한다.
- Test Coverage 리포트에서 복잡도가 높으나 단위 테스트 Coverage가 낮은 코드를 우선적으로 단위 테스트한다.
- 가능하면 테스트 코드를 먼저 만드는 TDD를 습관화한다. 단, 많은 시간과 노력이 필요하다.
- 코드의 복잡도를 낮출 수 있도록 지속적으로 리팩토링한다.
- 테스트하기 쉬운 코드를 만들기 위해 노력한다.

