



성공은 열정을 잃지 않고 실패를 거듭하는
과정 속에 나온다.

- 원스틴 처칠

데이터베이스 테스트: DbUnit

이번 장에서는 데이터베이스(이하 DB)와 연동되는 테스트 케이스를 작성할 때 발생하는 문제점과 그에 대한 가장 발전된 해결책을 제시하는 DbUnit에 대해 살펴볼 예정이다. 중반 이후로는 다소 참조 매뉴얼 같은 느낌을 주는 부분이 나오는데, 그 부분은 나중에 필요한 시점에서 다시 찾아볼 수 있는 수준으로만 학습하는 것도 좋은 접근 방법이다. 물론, 차분히 다 읽는다면 DbUnit이 제공하는 장점을 최대한으로 사용할 수 있게 될 것이다.

5장 체크리스트

- ☐ 데이터셋
- ☐ 데이터셋 비교
- ☐ 데이터셋의 종류
- ☐ DbUnit의 DB 관리 작업 지원 기능
- ☐ DbUnit과 ANT

5.1 DbUnit의 장점

크고 작고를 떠나, 근래에 만들어지는 애플리케이션 중에서, 특히 다수의 사용자를 가정해놓고 만들어진 애플리케이션 중에서 DB를 사용하지 않는 경우는 드물다. Oracle,

SQLServer, DB2 같은 기업용에서부터 MySQL, PostgreSQL 등의 오픈소스 DB, 그리고 JavaDB(이전의 Apache Derby)나 SQLite, HSQLDB(이전의 Hypersonic SQL) 같은 경량형 DB에 이르기까지, DB는 애플리케이션의 다양한 부분에서 다양한 방식으로 사용된다. DB를 사용하는 부분은 프로그래밍 언어 외적인 부분이 상당 부분 포함되기 때문에 TDD를 적용하기가 종종 쉽지 않다. 특히 데이터베이스의 상태를 일정하게 유지하면서 테스트를 지속적으로 수행 가능하게 만드는 데는 고통스러운 작업이 필요하다. 또한 테스트 전후의 데이터베이스 상태를 비교하는 것도 손쉬운 일은 아니다.

이럴 때 도움을 받을 수 있는 대표적인 유틸리티로 DbUnit(www.dbunit.org)이 있다. DbUnit은 다음과 같은 기능을 제공함으로써 DB를 사용하는 테스트 케이스 작성 시에 도움을 준다.

- 독립적인 데이터베이스 연결을 지원한다.

JDBC, DataSource, JNDI 방식 지원

- 데이터베이스의 특정 시점 상태를 쉽게 내보내거나(export) 읽어들일(import) 수 있다.

xml 파일이나 csv 파일 형식을 지원한다. import 계열 동작일 경우에는 엑셀 파일도 지원한다.

- 테이블이나 데이터셋을 서로 쉽게 비교할 수 있다.

보통 DbUnit은 독립적으로 사용하기보다는 JUnit 등의 테스트 프레임워크 등과 함께 사용한다. 그래서 DbUnit은 테스트 프레임워크라기보다는 테스트 지원 라이브러리에 더 가깝다. DbUnit을 알면 테스트 케이스 작성 시 DB 관리가 매우 편리해진다. 하지만 그러기 위해서는 제일 먼저 알아야 하는 DbUnit의 개념이 하나 있다. 바로 데이터셋(Dataset)이다.



DB? DBMS?

보통 흔히 우리는 오라클이나 MSSQL 서버를 DB라고 부른다. 그런데 엄밀히 말하자면 DB, 즉 데이터베이스는 '다루기 쉽도록 논리적인 구조로 저장된 데이터'를 의미한다. 그리고 오라클이나 MSSQL 서버는 그런 DB를 관리해주는 DBMS(Database Management System)라고 부르는 것이 맞다. 하지만 이미 많은 이들이 혼용해서 사용하기 때문에, 엄밀히 구분해야 한다고 강력히 주장하진 않겠다.

5.2 데이터셋

DbUnit에서 이야기하는 데이터셋(DataSet)은 데이터베이스나 그 안에 존재하는 테이블 혹은 그 일부를 xml이나 csv(comma separated value) 파일로 나타낸 모습이다. 이를테면 다음과 같은 판매자(SELLER) 테이블이 있다고 하자.

SELLER

ID
NAME
EMAIL

판매자 테이블

판매자 테이블에 들어 있는 데이터

ID	NAME	EMAIL
horichoi	최승호	megaseller@hatmail.com
buymore	김용진	shopper@nineseller.com
mattwhew	이종수	admin@maximumsale.net

위 내용을 DbUnit의 데이터셋으로 만들면 다음과 같이 표현된다.

seller.xml 파일

```
<?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>
```

위 XML은 DbUnit에서 제공하는 여러 데이터셋 중 가장 대표적인 형식인 FlatXmlDataSet 형식의 데이터셋이다. XML에서 seller라고 표시된 요소(element)는 DB 테이블 이름에 해당하고 ID, NAME, EMAIL 등의 속성(attribute)명은 해당 테이블의 컬럼 이름을 나타낸다. DbUnit은 이런 데이터셋이라 불리는 형식을 이용해서 DB의 상태를 저장하거나 변경, 유지한다. 그리고 데이터베이스 전체나 특정 테이블 등을 비교할 때도 이 데이터셋을 사용한다. 이제 우선 간단한 예제를 먼저 본 다음, DbUnit의 각 요소를 살펴보겠다.

데이터베이스 연결과 테이블 초기화

앞서 살펴본 판매자 테이블을 이용해서 판매자 정보를 DB에 저장하는 DatabaseRepository 클래스를 만들어보자. 이제, DatabaseRepository 클래스를 구현하기 위해 테스트 케이스를 작성해야 한다. 그래서 DatabaseRepository 클래스가 가져야 하는 기능들을 떠올려 보니, 문득 향후에는 DB가 아닌 파일이나 다른 매체에 저장될 수 있다는 생각이 들었다. 그렇다면, 인터페이스를 먼저 만들면 좋겠다. 그리고 이 인터페이스를 이용해서 DatabaseRepository를 만들면 좀 더 유연한 구조가 될 것 같다. 다음은 저장소를 대표하는 Repository 인터페이스다.

저장소(Repository) 인터페이스

```
public interface Repository {
    public Seller findById(String id);
    public void add(Seller seller);
    public void update(Seller seller);
}
```

```

        public void remove(Seller seller);
    }

```

이제 Repository 인터페이스를 이용해 DatabaseRepository를 만들어보자. DatabaseRepository는 Repository 스펙에 맞춘 구현체 중 하나로 판매자들을 관리하는 기능을 제공한다. 우선, Repository 인터페이스에 정의되어 있는 여러 기능 중, 판매자의 ID를 기준으로 판매자 정보를 가져오는 findById 기능을 구현해보자. 다음은 findById 기능을 구현하기 위해 만든 테스트 클래스와 테스트 메소드다.

```

public class DatabaseRepositoryTest {
    @Test
    public void testFindById() throws Exception {
        Seller expectedSeller = new Seller("horichoi", "최승호",
                                           "megaseller@hatmail.com");

        Repository repository = new DatabaseRepository();
        Seller actualSeller = repository.findById("horichoi");

        assertEquals(expectedSeller.getId(), actualSeller.getId());
        assertEquals(expectedSeller.getName(), actualSeller.getName());
        assertEquals(expectedSeller.getEmail(), actualSeller.getEmail());
    }
    ...
}

```

위 테스트 케이스는 단순히 ID 값으로 조회해서 나온 결과를 예상 결과와 비교하는 로직이기 때문에, 이 상태에서 곧바로 DatabaseRepository 구현에 들어가도 별 무리는 없다. 하지만 현재 가정한 데이터베이스 내의 데이터 내용이 변경된다면, 기능 구현에 문제가 없음에도 불구하고 테스트 케이스가 실패할 수 있다. 따라서 현재 가정되어 있는 DB 안의 데이터 상태가 테스트를 수행하기 전에 가정했던 모습으로 한결같이 유지됐으면 좋겠다. 그래서 다음과 같은 전략을 세워봤다.

테스트 관련 테이블 초기화



테스트 케이스 수행

테스트 실행 계획

이제 DbUnit을 테스트 케이스 작성에 이용해보자. DbUnit의 테이블 초기화 기능을 이용해서 위와 같은 형태로 테스트 케이스가 수행되도록 만들 생각이다. DBMS는 Apache Derby DB를 사용했다. Derby DB는 설치 및 사용 방법이 매우 간단한 DB이다. 예제를 따라 해보고자 한다면, 부록 A.4 '3분 안에 Java DB 설치하고 확인까지 마치기'를 참고하기 바란다.

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

    private IDatabaseTester databaseTester; // ❶

    @Before
    public void setUp() throws Exception{
        databaseTester = new JdbcDatabaseTester(driver, protocol +
            dbName); // ❷

        try {
            IDataset dataSet
                = new FlatXmlDataSetBuilder().build(new
                    File("seller.xml")); // ❸
            DatabaseOperation.CLEAN_INSERT.execute(
                databaseTester.getConnection(), dataSet); // ❹
        } finally {
            databaseTester.getConnection().close();
        }
    }
}
```

❶ DbUnit을 사용하기 위해서는 테스트 클래스가 DbUnit에서 제공하는 DBTestCase를 상속하도록 작성한다. 하지만 상속받아도 도움받는 부분이 많지 않은데다, 이 경우 JUnit 3 버전을 사용해야 한다는 단점이 있다. DBTestCase 클래스를 상속하지 않고, DbUnit에서 제공하는 기능을 이용하려면 IDatabaseTester라는 인터페이스를 사용하면 된다. 사실 DBTestCase 클래스도 내부적으로는 IDatabaseTester를 사용

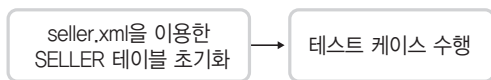
하고 있기 때문에 동작상 큰 차이는 없다. IDatabaseTester에는 DB 연결과 데이터셋 관련 기능이 정의되어 있다.

- ❷ 이 예제에서는 IDatabaseTester 구현체로 JDBC 연결 방식을 이용하는 JdbcDatabaseTester 클래스를 사용했다. DbUnit에서는 기본적으로 다음과 같은 네 개의 구현체를 제공한다.

JdbcDatabaseTester	DriverManager를 이용해 DB 커넥션을 생성
PropertiesBasedJdbcDatabaseTester	DriverManager를 이용해 DB 커넥션을 생성. 단, 연결 설정은 시스템 프로퍼티로부터 읽어들인다.
DataSourceDatabaseTester	javax.sql.DataSource를 이용해 DB 커넥션을 생성
JndiDatabaseTester	JNDI를 이용해 DataSource를 가져온다.

- ❸ 데이터셋을 지정한다. 위 예제의 경우엔 앞에서 보여준 판매자 데이터셋인 seller.xml을 지정했다.

- ❹ DB 커넥션과 데이터셋을 이용해 DB에 특정 작업을 수행한다. DatabaseOperation에는 여러 종류가 있는데, 그중 CLEAN_INSERT는 데이터셋에 지정된 DB 테이블의 내용을 모두 지운 다음, 데이터셋에 들어 있는 값으로 채워넣는다. 의미적으로 DatabaseOperation의 DELETE_ALL과 INSERT 두 동작을 연속적으로 수행한 것과 동일하다. 기타 동작에 대해서는 뒤에 나오는 표를 참고한다.



이제 위 setUp 메소드는 테스트 메소드가 수행되기 전에 항상 seller.xml에 지정된 상태로 테이블을 초기화한다. 다음은 위 테스트 케이스를 이용해 작성한 DatabaseRepository 클래스다. JDBC 드라이버를 이용해 DB에 접속한 다음, ID에 해당하는 판매자를 받아오는 아주 기본적인 코드다. 최대한 간결히 짜려고 많은 부분을 줄였다. 여유가 된다면, 잘못 짠 부분은 어디일지 생각해보자(답은 265쪽 아래의 주석에 있다).

```
package main.eshop;
import java.sql.*;

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();
        conn = DriverManager.getConnection(protocol + dbName);
    }

    public Seller findById(String id) {
        PreparedStatement stmt = null;
        ResultSet rs = null;
        Seller seller = null;

        try {
            String query = "select ID, name, email"
                + " from seller where ID = ?"; // ❶
            stmt = conn.prepareStatement(query);
            stmt.setString(1, id);
            rs = stmt.executeQuery();
            if ( !rs.next() ){
                throw new SQLException("No Data Found!"); // ❷
            }
            seller = new Seller(rs.getString(1), rs.getString(2),
                               rs.getString(3)); // ❸
            rs.close();
            stmt.close();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```

        return seller;
    }

    @Override
    public void remove(Seller seller) {}

    @Override
    public void update(Seller seller) {}

    @Override
    public void add(Seller seller) {}
}

```

- ❶ ID를 기준으로 판매자 정보(ID, NAME, EMAIL)를 불러온다.
- ❷ 1건도 존재하지 않으면 SQLException을 발생시킨다.
- ❸ SQL 실행 결과를 Seller 클래스에 담는다.

주의

Derby의 내장탐색 모드(embedded mode)에서는 DB 접속 연결이 오직 하나만 생성 가능하다. 명령어 실행창 유틸인 따라서 DerbyJ 등으로 접속해 있는 상태에서는 테스트 케이스를 비롯한 다른 애플리케이션에서 DB 접근이 안 된다. 이때는 DerbyJ를 exit; 명령어로 빠져나온 다음에 실행하도록 하자. Derby DB에 접근해서 작업을 할 수 있게 도와주는 DerbyJ에 대해서는 부록에서 다루고 있다.

데이터셋 비교

이번엔 판매자를 추가하는 기능을 구현하고자 한다. 테스트 케이스를 다음과 같이 추가했다.

판매자 추가 기능 구현을 위해 작성한 테스트 케이스

```

@Test
public void testAddNewSeller() throws Exception {
    Seller newSeller = new Seller("hssm", "이동욱", "scala@hssm.kr");
    Repository repository = new DatabaseRepository();
    repository.add(newSeller); // 새로운 판매자 추가
}

```

답: finally 처리를 안 했기 때문에 자원누수 가능성이 있다.

```

Seller sellerFromRepository = repository.findById("hssm");

assertEquals(newSeller.getId(),sellerFromRepository.getId());
assertEquals(newSeller.getName(),sellerFromRepository.getName());
assertEquals(newSeller.getEmail(),sellerFromRepository.getEmail());
}

```

테스트 케이스의 흐름을 살펴보면, 이동욱이라는 판매자를 생성해서 저장소에 추가하고, 정상 저장됐는지 확인하기 위해 판매자의 아이디로 저장소에서 찾아온다. 그리고는 저장소에서 ID로 찾아낸 판매자의 정보가 예상값과 같은지 비교한다. 앞서 작성한 @Before 애노테이션이 붙은 setUp 메소드로 인해 seller 테이블은 계속 초기화될 테니, 향후에도 지속적으로 동일한 테스트가 가능할 것이다. 이제는 특별히 고민할 필요 없이 테스트 케이스를 기준으로 구현을 진행해나가면 된다. 그런데 위 테스트 메소드에는 다소 미묘한 부분이 한 군데 존재한다. 계속 읽어나가기 전에, 어느 부분일지 생각해보자. 잘 모르겠으면, 몇 줄 안 되니까 그냥 짚어보기라도 하자.



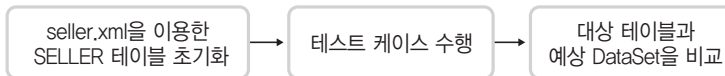
흔히 CRUD¹ 기능을 구현할 때 테스트 케이스 작성 시 취하는 흔한 방식은, 우선 조회 계열 기능을 제일 먼저 TDD로 구현하고, 그 다음 추가/수정/삭제 등은 해당 조회 기능을 이용해 검증하도록 작성한다. 위 예제에서도 비슷한데, testAddNewSeller 테스트 메소드는 테스트의 성공/실패 판단을 findById 기능에 전적으로 의존하고 있다. 자, 이제 앞에서 미묘하다고 표현한 부분이 어디일지 감이 오는가? 사실 대부분의 경우 이런 식의 테스트 케이스 작성이 별다른 문제처럼 느껴지진 않는다. 다만 나중에는 문제

1 데이터를 생성하고(create), 조회하고(retrieve), 수정하고(update), 삭제하는(delete) 작업들을 지칭

가 될 소지는 남아 있다. 그럼, 어떤 경우에 문제가 생길까? 어느 날, 다음과 같은 상황이 발생했다고 생각해보자.

- 조회 기능 구현에 오류가 있는 걸 발견한다.
- 조회 기능을 수정해야 할 일이 발생한다.

이런 경우, 조회 기능을 수정했더니 다른 테스트 메소드들이 한꺼번에 와장창 깨져버리는 일이 발생할 수 있다. 해당 테스트 메소드 내에서 검증하고자 하는 실제 기능들엔 아무런 문제가 없음에도 말이다. 비슷한 식으로, 만일 TDD로 작업된 소스에서 어느 한 부분을 고쳤더니 여러 개의 테스트 케이스가 동시에 실패한다면, 이런 경우가 아닌지 의심해볼 필요가 있다. 계속해서 이야기하게 되는 부분인데, 테스트 케이스 작성의 기본 원칙은, 하나의 테스트 케이스는 다른 부분에서 영향받는 부분이 최소화되어 있어야 한다는 것이다. 위 예제도 조회 기능에 의존하지 않고, 결과를 검증할 수 있게 만들려면 어떻게 하면 좋을까? 여러 가지 방법을 사용할 수 있지만, DbUnit을 설명하는 부분이니까 여기서는 DbUnit의 데이터셋을 이용해 검증해보자.² 다음은 결과 비교에 데이터셋을 사용하는 테스트 실행계획이다.



DbUnit의 테이블 비교 기능을 이용해 다시 작성한 테스트 메소드

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

    IDataset currentDBdataset =
        databaseTester.getConnection().createDataSet(); // ❶
    ITable actualTable = currentDBdataset.getTable("seller"); // ❷
    IDataset expectedDataSet = new FlatXmlDataSetBuilder().build(
```

2 이와 관련 주제는 7장 '개발 영역에 따른 TDD 작성 패턴'에서 따로 다루고 있다.

```

        new File("expected_seller.xml")); // ❸
        ITable expectedTable = expectedDataSet.getTable("seller"); // ❹

        Assertion.assertEquals(expectedTable, actualTable); // ❺
    }

```

- ❶ 현재 데이터베이스의 상태를 데이터셋으로 추출한다. createDataSet에 파라미터로 테이블 이름을 지정할 수도 있다. 지정하지 않으면 접속 유저 소유의 전체 테이블과 데이터를 데이터셋으로 만든다.
- ❷ 데이터셋에서 특정 테이블(seller)을 가져온다.
- ❸ 미리 만들어놓은 예상 데이터셋을 읽어들인다.

expected_seller.xml

```

<?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"/>
    <seller ID="hssm" NAME="이동욱" EMAIL="scala@hssm.kr"/>
</dataset>

```

- ❹ 예상 데이터셋 중에서 비교에 사용할 테이블을 읽어들인다.
- ❺ DbUnit에서 제공하는 Assertion 클래스의 메소드를 이용해 결과를 비교한다.

DbUnit에서 제공하는 Assertion 클래스의 메소드

```

public class Assertion {
    public static void assertEquals(ITable expected, ITable actual)
    public static void assertEquals(IDataSet expected, IDataSet actual)
}

```

(당연하지만) JUnit에서는 제공하지 않는 ITable 타입과 IDataset 타입의 비교를 지원해준다. 만일 데이터셋을 직접 비교하고 싶으면 다음과 같이 작성한다.

```
...
IDataset currentDBdataset
    = databaseTester.getConnection().createDataSet(new String[]
        {"seller"});
IDataset expectedDataSet
    = new FlatXmlDataSetBuilder().build(new File("expected_seller.xml"));
Assertion.assertEquals(expectedDataSet, currentDBdataset);
...
```

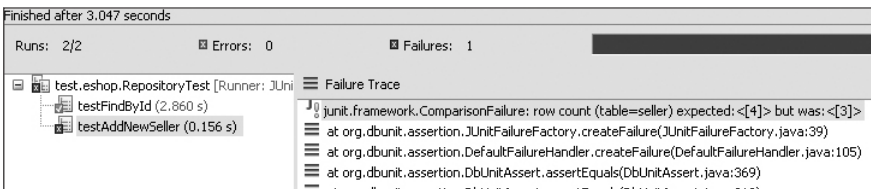
주의

DbUnit은 위 예제의 경우처럼 데이터베이스의 스냅샷(Snapshot, 특정 시점의 상태나 이미지)을 잡아 테이블로 추출할 때, 해당 테이블의 기본키(primary key, PK) 값으로 정렬한다. 하지만 데이터셋을 파일로부터 읽어들이는 때는 해당 테이블을 정렬하지 않는다. 그래서 만일 새로운 데이터가 테이블에 추가되어 PK 정렬로 인해 순서가 달라지면 동일한 내용의 데이터셋임에도 오류가 발생할 수 있다. 이럴 경우에는 SQL을 이용해 데이터셋의 일부만을 추출할 수 있게 도와주는 createQueryTable을 사용해서 직접 sql에 "ORDER BY" 구문을 포함시켜 버리거나, 아니면 파일로부터 읽어들이는 데이터셋 테이블을 SortedTable 클래스로 정렬한 다음 비교하면 된다.

```
// 자동 정렬이 일어나지 않도록 SQL 문을 지정하든가
ITable actualTable = connection.createQueryTable("seller", "select *
from seller");

// 예상 결과 데이터셋 값을 정렬시켜 버리든가
Assertion.assertEquals(new SortedTable(expectedTable), actualTable);
```

아래 이미지는 테스트 케이스를 실제로 실행했을 때의 결과다. 예상 테이블 값과 건수(row count)가 다르다며 실패하고 있다.



실패하는 테스트 케이스

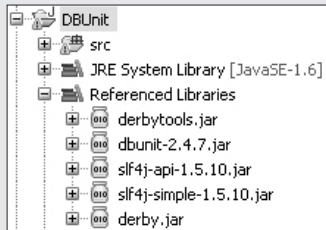
이제 테스트 케이스가 만들어졌으니, 실제 로직 구현에 들어가면 된다. 이런 식으로 테스트 케이스를 작성하면 테스트 메소드 수행 결과가 테스트 대상 클래스의 다른 로직에서 받게 되는 영향을 최소한으로 만들어줄 수 있다. 하지만 모든 테스트 케이스를 이렇게 만들긴 어려울 수 있다. 무조건 더 좋은 것이 꼭 답은 아닐 때가 있기 때문이다. 더 비싼 자동차가 더 좋은 건 알지만, 대신 더 비싼 비용을 치뤄야 하는 것처럼 말이다.

이제, DbUnit을 어떻게 사용할지 조금은 감이 왔을 것 같다. 이제부터는 DbUnit의 주요 부분을 좀 더 자세히 알아볼 예정이다. 5.3절 'DbUnit 데이터셋의 종류'부터는 필요할 때 참조할 수 있도록 매뉴얼처럼 만들어뒀으므로, 보다가 잘 와 닿지 않는다면 필요할 때 다시 찾아볼 필요가 없다. 한번 훑어보고, 필요할 때 떠올릴 수 있는 정도면 충분하다. 음, 그런데 이런 말도 있긴 하다. '디테일의 힘'이라든가, '작은 차이가 큰 결과 차이를 만든다'든가 뭐 그런... 아, 뭐 말이 그렇다는 이야기다. 크게 신경 쓰진 말자. :)



실습 시 클래스패스 및 라이브러리 관련 유의점

현재 예제는 DbUnit과 Apache Derby DB를 이용하고 있다. 실습으로 진행할 때는 각각에 해당하는 라이브러리와 의존 라이브러리가 클래스패스 내에 함께 존재해야 한다. 다음은 실습에 사용한 참조 라이브러리 목록이다.



SLF4J(Simple Logging Facade for Java) 라이브러리는 <http://www.slf4j.org/>에서 내려받는다. 그런데 이런 식으로 매번 의존성 라이브러리를 찾아서 내려받는 것도 생산성을 상당히 저하시키는 요소다. 특히 오픈소스 라이브러리는 이런 의존성 관련 작업이 많다. 리눅스에 아파치 서버 한번 컴파일할라 치면, 정말이지 참조 라이브러리 찾아서 멀고도 먼 길을 떠나야 한다. 이런 문제를 해결하기 위한 노력이 각각의 플랫폼에서 있었다. 우분투의 apt-get, RedHat의 yum, Java 계열의 Maven이나 Apache IVY 등이 그 예다. 근래 Java 계열에서는 특히 Maven을 많이 사용하

는데, 이런 의존성 라이브러리 찾기 문제가 좀 더 유연하게 해결된다. 기회가 닿으면 한번 사용해 보길 권장한다.

참고

1. Maven + m2eclipse 설치하기(<http://blog.doortts.com/59>)
2. 『자바 프로젝트 필수 유틸리티: Maven, TeamCity, Subversion, Trac』(한빛미디어, 박재성 저)

5.3 DbUnit 데이터셋의 종류

앞에서 DbUnit은 DB 데이터의 구조를 나타내기 위해 데이터셋이라는 개념을 사용한다고 이야기했다. 이 데이터셋이란 개념은 하나의 타입을 나타냄과 동시에 테이블들의 집합체를 표현하는 IDataset 인터페이스의 구현체를 의미하기도 한다. 앞서 본 seller.xml 파일은 그중 FlatXmlDataSet으로 표현된 데이터셋이다.

FlatXmlDataSet

- 테이블 이름을 XML TAG 구성요소로 적는다.
- 컬럼 이름은 속성으로 적는다.
- 널(null)값을 넣을 컬럼은 표현하지 않는다. 자동으로 널값이 들어간다.
- XML DTD(Document Type Definitions)를 지정하지 않아도 된다.
- 데이터셋 중 가장 흔하게 사용된다.

FlatXmlDataSet 형식으로 작성된 XML 파일

```
<dataset>
  <EMPLOYEE NO="101" NAME="안병현" EMAIL="megane@hssm.kr" />
  <EMPLOYEE NO="102" NAME="김상욱" />
  <DEPARTMENT />
</dataset>
```

XmlDataSet

- 다소 장황한 버전의 데이터셋
- DTD를 반드시 포함해야 한다.
- 잘 사용하지 않는다.

StreamingDataSet

- 데이터베이스의 커서(cursor) 개념처럼 단방향으로 동작하며 현재 레코드만 메모리에 존재한다.
- UPDATE, INSERT, REFRESH 같은 동작을 하는 XML 데이터셋을 읽어들이기 때 매우 효율적으로 동작한다.

```
IDatasetProducer producer =  
    new FlatXmlProducer(new InputSource("dataset.xml"));  
IDataset dataSet = new StreamingDataSet(producer);
```

DatabaseDataSet

- 데이터베이스 인스턴스에 대한 접근을 제공한다.
- 직접 new로 생성하지 않고 팩토리 메소드로 만들어낸다.

```
IDataset currentDBdataSet = IDatabaseConnection.createDataSet();
```

QueryDataSet

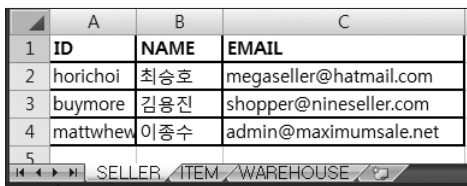
- 쿼리문으로 데이터셋을 만들어낸다.

사번이 600번 이상인 직원과 DEPARTMENT 테이블 전체를 데이터셋에 담는다.

```
QueryDataSet dataSet = new QueryDataSet(connection);
dataSet.addTable("NEW_EMPLOYEE", "SELECT * FROM EMPLOYEE WHERE EMPNO >
    600");
dataSet.addTable("DEPARTMENT");
```

XlsDataSet

- MS 엑셀 문서를 데이터셋으로 인식한다.
- 엑셀 문서 내의 각 시트(sheet)를 테이블로 인식한다.
- 시트의 첫 번째 줄을 컬럼 이름으로 인식한다.
- 나머지 줄은 데이터 값으로 인식한다.



	A	B	C
1	ID	NAME	EMAIL
2	horichoi	최승호	megaseller@hatmail.com
3	buymore	김용진	shopper@nineseller.com
4	mattwhew	이종수	admin@maximumsale.net
5			

SELLER 테이블을 엑셀에 표현한 모습

ITEM 테이블과 WAREHOUSE 테이블도 다른 시트에서 표현하고 있다.

ReplacementDataSet

- 데이터셋에서 특정한 문자열을 치환하기 위해 사용한다.
- 보통은 null 값을 다르게 표현하고 런타임 시에 치환하는 데 많이 사용한다.

```
<dataset>
  <EMPLOYEE NO="101" NAME="안병현" EMAIL="megane@hssm.kr"/>
  <EMPLOYEE NO="102" NAME="김상옥" EMAIL="[null]"/>
</dataset>
```

데이터셋 파일은 위와 같이 만든 다음 아래와 같은 코드로 읽어들이어서 null 값을 변환한다.

```
ReplacementDataSet dataSet = new ReplacementDataSet( new FlatXmlDataSet( ...
...));
dataSet.addReplacementObject("[NULL]", null);
```

이 외에도 CompositeDataSet과 FilteredDataSet 등이 있다. 이제 DbUnit이 지원하는 DB 작업 지원 기능에는 어떠한 것이 있는지, DbUnit을 Ant를 이용해 실행하려면 어떻게 해야 하는지 차례대로 살펴보자.

5.4 DbUnit의 DB 지원 기능

앞부분의 예제에서 테스트 케이스를 수행하기 전에 DB를 지정한 데이터셋으로 초기화하도록 만들 때 DatabaseOperation이라는 클래스를 이용했던 것이 기억나는지 모르겠다.

DatabaseRepositoryTest 클래스의 setUp 메소드 중 일부

```
public void setUp() throws Exception{
    ...
    DatabaseOperation.CLEAN_INSERT.execute(databaseTester.getConnection(),
        dataSet);
    ...
}
```

DbUnit에서는 데이터셋을 이용한 DB 관리 작업을 DatabaseOperation이라는 개념으로 만들어놓았다. 사용 방법은 아래와 같다.

```
DatabaseOperation.오퍼레이션이름.execute( DB커넥션, 데이터셋 );
```

다음은 DbUnit에서 사용할 수 있는 오퍼레이션의 종류다.

DatabaseOperation의 종류	설명
DatabaseOperation.INSERT	데이터베이스에 데이터셋 내용을 INSERT한다. PK(primary key)를 기준으로 대상 테이블에 중복 데이터가 들어 있지 않다는 가정하에서 동작하기 때문에 중복 데이터가 존재하면 실패로 간주한다. FK(foreign key, 참조키)가 걸려 있는 테이블의 경우 데이터셋의 순서에 따라 정상적으로 INSERT가 안 될 수 있으므로 유의한다.
DatabaseOperation.DELETE_ALL	데이터셋에 지정된 테이블들의 데이터를 모두 지운다. 지정되지 않은 테이블들은 건드리지 않는다.
DatabaseOperation.CLEAN_INSERT	데이터셋에 지정된 테이블에 대해 DELETE_ALL을 수행한 다음, 데이터셋에 있는 데이터 값을 INSERT한다. 즉, DELETE_ALL + INSERT와 동일하다. REFRESH와 함께 매우 잘 사용되는 기능이다.
DatabaseOperation.UPDATE	데이터셋의 내용으로 테이블을 업데이트한다. UPDATE의 기준은 INSERT와 마찬가지로 PK 컬럼이 된다.
DatabaseOperation.REFRESH	CLEAN_INSERT와 함께 가장 많이 사용되는 기능. 대상 테이블에 존재하지 않는 데이터는 INSERT, 이미 존재하는 데이터일 경우에는 UPDATE한다. 둘 다에 속하지 않는, 이미 테이블에 존재하는 데이터는 건드리지 않는다.
DatabaseOperation.DELETE	데이터셋과 일치하는 데이터를 테이블에서 지운다. 테이블 전체를 지우지는 않는다.
DatabaseOperation.TRUNCATE	데이터셋에 지정된 테이블들의 데이터를 모두 지운다. TRUNCATE는 DELETE와 달리 롤백(rollback)이 불가능하다. 테이블 데이터 삭제 작업은 데이터셋에 지정된 테이블 순서의 역순으로 적용된다.
CompositeOperation	<p>여러 개의 DatabaseOperation을 하나로 묶어서 한 번에 실행한다. 이런 방식을 DatabaseOperation 클래스를 데코레이트³한다고 표현한다.</p> <pre>DatabaseOperation op = new CompositeOperation(DatabaseOperation.DELETE_ALL, DatabaseOperation.INSERT); op.execute(connection, xmlDataSet);</pre> <p>위 코드는 DELETE_ALL과 INSERT를 하나의 DatabaseOperation으로 묶은 모습이다.</p>

3 데코레이트(decorate)는 대상이 되는 객체를 감싸서 대상 객체의 기능을 확장하는 기법을 말한다. 데코레이트된 클래스는, 보통 대상이 되는 클래스를 자신의 멤버로 갖는다. 그리고 대상 클래스와 동일한 부모 타입을 갖고, 동일한 메소드 시그니처(signature)를 갖도록 만든다. 이렇게 구성하면, 사용하는 쪽에서는 자신이 다루는 객체의 타입이 오리지널인지 데코레이트된 클래스인지 구별하지 않고 동일한 방식으로 이용 가능하다. 대상 클래스 자체를 더 이상 상속할 수 없거나, 일부 기능만을 확장하고자 할 때 많이 사용하는 방식이다. 좀 더 살펴보고 싶다면 『Head First Design Patterns』(한빛미디어)를 참고하길 권장한다.

TransactionOperation	<p>데이터셋을 처리할 때 트랜잭션으로 묶어서 처리할 것인지를 결정한다. DatabaseOperation 클래스를 데코레이트한다.</p> <pre>DatabaseOperation op = new CompositeOperation(DatabaseOperation.DELETE_ALL, DatabaseOperation.INSERT); op = new TransactionOperation(operation); op.execute(connection, xmlDataSet);</pre> <p>위 코드는 DELETE_ALL과 INSERT를 하나의 DatabaseOperation으로 묶은 다음 그 오퍼레이션에 트랜잭션을 걸어놓은 모습이다. 중간에 실패하는 부분이 있으면, 전부 롤백한다.</p>
IdentityInsertOperation	<p>MS SQL 서버의 IDENTITY 컬럼을 잠시 비활성화시킨 상태로 만들어 INSERT 시에 오류가 발생하지 않도록 도와준다. 참고로, IDENTITY 컬럼은 자동으로 숫자가 증가해서 입력되는 컬럼으로, 특정 값을 강제로 넣는 것이 일반적인 INSERT 문으로는 불가능하다.</p>

5.5 DbUnit과 Ant

DbUnit은 DbUnit 라이브러리 코드를 테스트 케이스 내에서 직접 선언해 수행할 수도 있지만, Ant를 이용해 처리할 수도 있다. DB를 다루는 작업인 만큼, 이제까지 본 DbUnit의 기능을 Ant로 수행하는 것이 더 편한 경우가 많다.

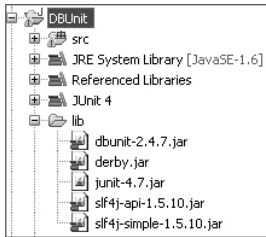
그리고 Ant를 사용하면 개발툴이나 IDE에 의존하지 않고, 시스템 레벨에서 배치 작업 등을 이용해 좀 더 높은 수준의 자동화를 이룰 수 있다. 그러나 Ant가 비록 사용법이 어렵진 않다곤 하지만, 낯설어서 거부감이 들거나 이후 내용이 지루하다는 생각이 들면 우선은 훑고만 넘어가자. 혹은 11장 ‘테스트 자동화와 커버리지’에서 Ant에 대한 내용과 자동화에 대한 부분을 자세히 설명하고 있으니, 11장을 먼저 학습한 다음에 다시 이 부분으로 돌아와 읽는 것도 괜찮은 학습법이다. 각 장 맨 처음에 나오는 체크리스트를 이럴 때 적극 이용하면 좋겠다. :)

DbUnit에서 Ant를 이용하기 위한 준비 단계

Ant의 클래스패스 내에 dbunit.jar 파일을 추가한다. 그리고 DbUnit은 Ant에서 지원하는 기본 태스크가 아니기 때문에, 사용자 태스크로 지정할 필요가 있다. Ant 빌드파일 내에 태스크를 정의한다.

```
<taskdef name="dbunit" classname="org.dbunit.ant.DbUnitTask"/>
```

그런데 위 태스크 정의가 정상적으로 동작하려면 dbunit.jar 파일이 클래스패스 내에 존재해야 한다. 빌드파일에서 참조하는 라이브러리는 가급적 해당 파일 내에서 확인할 수 있도록 하는 것이 좋다.



lib 폴더 밑에 의존 라이브러리들이 위와 같이 존재한다고 가정하고 진행한다.

Ant의 빌드파일로 사용할 build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="SellerManagement" default="test" basedir=".">

    <target name="test" depends="">

        <taskdef name="dbunit" classname="org.dbunit.ant.DbUnitTask">
            <classpath>
                <fileset dir="${basedir}/lib" includes="**/*.jar"/>
            </classpath>
        </taskdef>
    </project>
```

이제 dbunit이라는 이름의 Ant 태스크가 정의됐다. 이제 DbUnit을 Ant에서 사용할 준비가 됐다.

앞에서 작성한 RepositoryTest 클래스의 @Before 부분을 살펴보자.

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

@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);
}
```

테스트를 수행하기 전에 데이터를 초기화하는 코드다. 이걸 Ant 스크립트로 전환하면 다음과 같다.

```
<target name="sellerdb-init">
    <dbunit driver="org.apache.derby.jdbc.EmbeddedDriver"
        url="jdbc:derby:shopdb"
        userid=""
        password=""
    >
        <operation type="CLEAN_INSERT" src="seller.xml"/>
    </dbunit>
</target>
```

실행해보면 다음과 같다.

```
Buildfile: D:\Development\DBUnit\build.xml
sellerdb-init:
    [dbunit] Executing operation: CLEAN_INSERT
    [dbunit]           on file: D:\Development\DBUnit\seller.xml
```



```
[dbunit]                with format: null
test:
BUILD SUCCESSFUL
Total time: 1 second
```

이런 식으로 무언가 일괄적으로 DbUnit을 이용해 DB를 조작할 때 Ant를 이용하면 유리하다. 특히 특정 데이터베이스나 테이블의 스냅샷을 잡아 스크립트로 만들어놓을 때 Ant와 DbUnit을 함께 쓰면 좋다.

dbunit.xml 파일의 뼈대가 아래와 같다고 가정한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="dbunit" default="export" basedir=".">
  <property name="driver" value="org.apache.derby.jdbc.EmbeddedDriver" />
  <property name="uri" value="jdbc:derby:shopdb" />
  <property name="userid" value="" />
  <property name="password" value="" />

  ...

  ...

</project>
```

유저테이블 전체를 export 받기

```
<dbunit driver="${driver}" url="${uri}" userid="${userid}"
        password="${password}">
  <export dest="export.xml" />
</dbunit>
```

참고로, export는 기본적으로 UTF-8로 인코딩해서 파일로 만들어진다. export 옵션 중 encoding이 있는데, xml 파일에 한정되어 적용된다는 점에 주의하자.

특정 쿼리의 결과나 특정 테이블만 export 받기

```
<dbunit driver="${driver}" url="${uri}" userid="${userid}"
  password="${password}">
  <export dest="export.xml" >
    <query name="item" sql="select PRODUCTNO, NAME, PRICE from item" />
    <table name="seller" />
  </export>
</dbunit>
```

현재 폴더에 특정 테이블만 csv 파일로 내려받기

```
<dbunit driver="${driver}" url="${uri}" userid="${userid}"
  password="${password}">
  <export dest="." format="csv">
    <table name="seller" />
  </export>
</dbunit>
```

DELETE_ALL + INSERT

```
<dbunit driver="${driver}" url="${uri}" userid="${userid}"
  password="${password}" >
  <operation type="CLEAN_INSERT" src="export.xml" />
</dbunit>
```

DB 유저 전체 테이블에 대해 비교하기

```
<dbunit driver="${driver}" url="${uri}" userid="${userid}"
  password="${password}">
  <compare src="export.xml" />
</dbunit>
```

위 경우 export.xml은 미리 만들어놓은 예상 데이터셋이다.

특정 테이블이나 데이터만 비교하기

```
<dbunit driver="${driver}" url="${uri}" userid="${userid}"
    password="${password}">
    <compare src="export.xml" >
        <query name="item" sql="select PRODUCTNO, NAME, PRICE from item" />
        <table name="seller" />
    </compare>
</dbunit>
```

저자
한·지

Java DB? Apache Derby?

이제는 오라클(oracle)이 된 썬(SUN)은 Java 6 버전 이상부터 Java DB라는 경량형 데이터베이스를 포함시켜 배포하고 있다. 그런데 알고 보면 Java DB는 Apache Derby라는 오픈소스 데이터베이스의 SUN 배포판이다. Apache Derby는 SQLite 같은 경량형 데이터베이스를 지향하면서도, 커밋(commit)이나 롤백(rollback) 같은 OLTP(Online Transaction Processing)를 지원하며, 그 크기도 불과 2M 남짓에 불과하다. 한때 Hypersonic SQL이라고 불렸던 HSQLDB(Hyper Structured Query Language Database)와 함께 트랜잭션 업무가 크지 않은 분야에서 두루 사용되고 있다. 이를테면 데스크톱 애플리케이션 등을 만들 때, 환경설정 값이나 애플리케이션 데이터 등을 파일 대신에 이런 경량화된 DB를 이용하면 편리하다. 사용법도 매우 간단하다. 엔진에 해당하는 derby.jar와 관리 툴인 derbytools.jar, 이 두 파일만 클래스패스 내에 존재하면, 바로 RDBMS(관계형 데이터베이스)로 사용할 수 있다. Java DB는 사용 목적에 따라 네트워크 서버(network server) 타입 혹은 임베디드(embedded) 타입 중에서 선택할 수 있다. 단, 임베디드 타입으로 구동할 경우에는 Java 애플리케이션과 동일한 JVM 위에서 동작하고, 오로지 해당 애플리케이션에서만 접근 가능한 싱글유저 모드로 동작한다. 따라서 이 경우 Derby DB의 기동과 정지는 Derby를 사용하는 애플리케이션의 생명주기를 따른다. 네트워크 서버 모드는 우리가 흔히 알고 있는 형태인, 서버/클라이언트 방식으로 동작하는 독립형(stand-alone) DB라고 생각하면 된다. 클라이언트에서 접속하고자 할 때 사용되는 JDBC 드라이버의 경우, 임베디드 드라이버는 derby.jar 파일 안에, 네트워크 서버에 접속하기 위한 드라이버는 배포판에서 함께 제공되는 derbyclient.jar 파일 안에 들어 있다.

5.6 정리

DbUnit을 사용하는 것이 어렵지는 않지만, 데이터셋을 만드는 일은 비용과 노력이 많이 드는 작업이며, 때때로 ROI가 맞지 않곤 한다. 따라서 DB를 사용할 때, DbUnit을 만드시 써야 할지는 고민해볼 필요가 있다. 때에 따라서는 단순히 SQL 파일을 테스트 전후로 실행하는 편이 더 나을 수도 있기 때문이다.

DbUnit에 대해서는, DbUnit의 권장 사용법을 설명하는 걸로 마무리하기로 한다.

DbUnit 권장 사용법

- 개발자마다 데이터베이스 인스턴스나 스키마를 하나씩 쓸 수 있게 하라.
- 나중에 정리(tearDown)를 할 필요가 없도록 setUp 처리를 잘 하자.
- 데이터셋은 크기를 작게 하고 여러 개로 만들어라. 꼭 필요한 테스트 데이터 위주로 만들자.
- 데이터셋을 너무 많이 만들지 마라. 유지보수가 힘들다.
- 데이터셋은 테스트 클래스 기반으로 만들고, 여타 테스트 클래스와 공유해서 사용하지 말자.
- 테스트용 데이터베이스에서는 참조키(foreign key)나 널값 제약(not null constraint) 기능을 꺼놓으면 편리하다.

안영희 (주)아이티와이즈컨설팅 Wise엔지니어링그룹장

Q. 안녕하십니까? 안영희님. 바쁘신 와중에 인터뷰에 응해주셔서 감사합니다. 간략한 소개 부탁드립니다 될까요? 또 현재 하시는 일이나 근래에 관심 두고 계신 것은 어떤 게 있으신지요?

A. 소프트웨어 컨설팅 회사에서 일하고 있습니다. 지난달까지 금융회사 IFRS 프로젝트에 10달 동안 참여해서 소프트웨어 아키텍트 역할로 일했습니다. 실제로 한 일은 코드 품질을 높이기 위해 자동화 테스트를 작성하도록 독려하고 여러 시스템을 엮어서 '결산'이라고 하는 업무 절차를 자동화하는 프로그램을 설계했습니다. 프로젝트를 마친 후 호주로 휴가를 다녀오고 삶을 되돌아보는 중입니다. 제가 하는 일을 좋아하기 때문에 이 일을 오래하려면 즐거움을 찾아야 하고, 사랑하는 사람과 함께 하는 삶과 조화를 이뤄야 한다는 사실을 깨달았습니다.

Q. 영희님은 세미나 발표 등, 공식적인 자리에서 TDD에 대한 언급을 많이 하시는데요, 혹시 계기나 동기가 있으신지요?

A. 두 가지가 있습니다. 하나는 TDD 실천 과정에서 배우는 코딩의 리듬, 즉 코드를 진화해 가는 자신만의 페이스를 터득하는 일이 매우 유익하기 때문입니다. 이런 리듬은 비단 프로그래밍뿐 아니라 다른 작업을 처리하는 데도 응용할 수 있죠. 두 번째는 반골기질이 있는지 몰라도 '척박한 SI 현실에서 TDD는 무리'라고 하는 이상한 주장을 들으면 반론을 펼치게 됩니다. 실제 대형 프로젝트에서 자동화 테스트를 적용한 바 있지만, '현장을 충분히 고려하지 않고 책대로 따라 하려는 시도'나 '익숙하지 않은 기법에 대한 저항감'이 문제지, 자동화 테스트를 작성하는 과정이나 이를 축적하여 만드는 회귀 테스트의 효과는 투여하는 노력에 비해 보상이 탁월한 매우 생산적인 활동이라 생각합니다.

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

A. 네. 앞서 말씀드린 바대로입니다. 다만 TDD 전문가는 아닌 터라 장애물을 만나죠. 학습 목적이 아닌 경우에는 개발과 동시에 실험을 할 수는 없기 때문에 선택적으로 TDD를 적용합니다. SI 프로젝트에 참여하다 보면 처음 만나는 개발팀과 협력을 합니다. 이렇듯 실력을 알 수 없는 개발자 수십 명이 함께 하는 경우는 테스트 환경을 통일하고 테스트를 쉽게 해주는 유틸리티를 만들어 교육을 하고 주로 데이터 입출력 부분에 대해서만 자동화 테스트를 작성하도록 유도합니다. 불특정 다수를 대상으로 무리하게 TDD를 적용하려고 욕심을 부리다가 학원(?)으로 바뀔 우려가 있고 저항감도 있으니 현장에서 터득한 적정 수위죠.

반면 최근에는 일주일 만에 작은 단위 시스템을 개발해야 하는 급박한 상황에서 TDD를 통해 효과를 본 일이 있습니다. 생각과 그림만으로 설계하는 방법보다는 실제 코드를 만들어 가면서 인터페이스나 핵심 로직을 설계합니다. 그 과정에서 예상하지 못한 내용을 깨닫고 배웁니다. 설계와 학습이 빠른 사이클로 이루어지는데 이 경우에 TDD를 대체하는 다른 방법이 없어 보일 정도로 효율이 높더군요.

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

A. 어떤 프로젝트에서 테스트 작성을 번거롭게 생각한 개발자 의견을 취합해서 주간보고 때 해당 시스템 개발 PL이 개발자 테스트 때문에 작업이 버겁다고 보고를 하더군요. 그래서 다음 주 주간보고 때 개발자 개인별 작성 코드량, 작성하는 함수 개수 등을 취합해서 발표한 바 있습니다. PM이나 실제 코드에 대해 잘 모르는 이해관계자가 깜짝 놀라더군요. 개발 초기에 개발자의 진도는 관리자가 상상하는 것에 비해 무척 더디기 마련이니까요. 업무를 익히는 시간이라고 말하는데 사실 테스트를 작성하면서 경우의 수를 고민해보는 시간보다 더 빨리 업무를 배우는 방법이 있을까요?

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

A. 빠른 피드백을 통해 '만들어야 할 프로그램'을 학습해나가는 점진적인 기법이라고 이야기하고 싶습니다. '피드백'과 '학습'의 의미를 강조하고 싶어서죠.

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

A. 꼭 TDD에만 해당하는 내용은 아니고요, 무언가 새로운 기법을 적용할 때는 두 가지 조건이 필요하다고 생각합니다. 먼저 배울 때는 자기 생각을 버리고 열린 자세를 취하는 일이지요. 사실 저도 잘 못하는 일이라 말을 하고 나니 부끄럽네요. 두 번째는 학습 이후에 적용할 때는 현장을 충분히 반영하여 지속적으로 방법을 개선해야 한다고 생각합니다. 함께 하는 사람, 주어진 시간 등을 충분히 고려하고, 처음에 정한 방식을 계속 고수하기보다는 역시 피드백을 얻으면 이를 반영해서 현장에 꼭 맞는 방법을 찾아가야 한다고 생각합니다.

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

A. 마침 『김예슬 선언, 오늘 나는 대학을 그만둔다, 아니 거부한다』는 책을 읽었습니다. 저는 김예슬 씨처럼 당당하지 못해서 아쉬운데 공감하는 내용이 많더군요. 제 스스로에게 하는 말이기도 한데요. 좋은 소프트웨어를 만드는 일 이전에 자기 삶을 되돌아보라고 하고 싶습니다. 충분히 행복한지, 소중한 것을 잃어버리고 사는 건 아닌지. 먼저 삶에 대한 확신이 있어야 즐겁게 일하고 직업윤리도 생긴다고 봅니다. 얼마 전 켄트 벡(Kent Beck)이 Manipulation(조작)에 대해 블로그에 쓴 글이 좋아서 출력해서 들고 다니는데요. 현장에서 일하다 보면 협상할 일이 많은데 켄트 벡이 이겨내려고 했던 Manipulation이 마치 정석 기법처럼 쓰입니다. 비정규직이 흔해진 조직과 빈둥빈둥 노는 관리자, 앞서 예를 든 정적하지 않은 협상문화를 보면서 반성 없이 사회에 젖어서 살면 안 되겠다는 생각을 하게 됩니다. 여러분도 그런 기회를 갖고 더 나은 개발자 문화를 만들기 위해 연대하면 어떨까요?