2장 테스트

2.1 UserDaoTest 다시보기

2.1.1 테스트의 유용성

테스트란 결국 내가 예상하고 의도했던 대로 코드가 정확히 동작하는지를 확인해서 만든 코드를 확신할 수 있게 해주는 작업니다.

2.1.2 UserDaoTest의 특징

main메서드 사용, UserDao직접 호출

- 웹을 통한 DAO 테스트 방법의 문제점
  - DAO뿐만 아니라 서비스클래스,컨트롤러, JSP 뷰등 모든 레이어의 기능을 다 만들고 나서야 테스트가 가능하다.
  - 테스트가 실패했다면 어디서 문제가 발생했는지 찾는 수고도 필요하다.
- 작은 단위의 테스트
  - 관심사의 분리라는 원리가 적용된다.
  - 테스트의 관심이 다르면 테스트할 대상을 분리하고 집중해서 접근해야 한다.
  - 단위테스트를 하는 이유는 개발자가 설계하고 만든 코드가 원래 의도한 대로 동작하는지를 개발자 스스로 빨리 확인받기 위해서다.
- 자동수행 테스트 코드
  - 자동으로 수행되는 테스트의 장점은 자주 반복할 수 있다.
- 지속적인 개선과 점진적인 개발을 위한 테스트
  - 테스트를 이용하면 새로운 기능도 기대한 대로 동작하는 지 확인할 수 있을 뿐만 아니라, 기존에 만들어뒀던 기능들이 새로운 기능을 추가하느라 수정한 코드에 영향을 받지 않고 여전히 잘 동작하는지를 확인할 수도 있다.

2.1.3 UserDaoTes의 문제점

- 수동 확인 작업의 번거로움
- 실행 작업의 번거로움

2.2 UserDaoTest 개선

2.2.1 테스트 **검증**의 자동화

```
// 전
System.out.println(user.getId());
System.out.println(user.getPw());
System.out.println(user.getId()+"조회 성공");
// 후
        if(!user.getId().equals(user2.getId())){
            System.out.println("테스트 실패 (id)");
        }
        else if(!user.getPw().equals(user2.getPw())){
            System.out.println("테스트 실패 (password)");
        }
        else {
            System.out.println("조회 테스트 성공");
        }
```

2.2.2 테스트의 효율적인 수행과 결과 관리

- JUnit 테스트로 전환
  - Junit 프레임워크
- 테스트 메소드 전환
  - public 메소드+@Test
- 검증 코드 전환

리스트 2-5 JUnit을 적용한 UserDaoTest

```
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;
...
public class UserDaoTest {
    @Test
    public void addAndGet() throws SQLException {
        ApplicationContext context = new GenericXmlApplicationContext(
            "applicationContext.xml");

        UserDao dao = context.getBean("userDao", UserDao.class);
        User user = new User();
        user.setId("gyumee");
        user.setName("박성철");
        user.setPassword("springno1");

        dao.add(user);

        User user2 = dao.get(user.getId());

        assertThat(user2.getName(), is(user.getName()));
        assertThat(user2.getPassword(), is(user.getPassword()));
    }
}
```

JUnit 테스트로 전환하는 김에 새로운 기분으로 테스트 데이터도 바꿔보자.

- JUnit 테스트 실행

2.3 개발자를 위한 테스팅 프레임워크 JUnit

2.3.1 JUnit 테스트 실행 방법

- IDE
- 빌드 툴

2.3.2 테스트 결과의 일관성

- 코드에 변경사항이 없으면 항상 동일한 결과를 내야 한다.
- 일관성 있는 결과를 보장하는 테스트를 만들기 위해 deleteAll과 getCount()를 추가한다.

```
@Test
void andAddGet() throws SQLExcpetion{
...
  dao.deleteAll();
  assertThat(dao.getCount(), is(0));

  User user = new User();
  user.setId("gyumee");
```

```
    user.setName("박성철");
    user.setPassword("springno1");

    dao.add(user);
    assertThat(dao.getCount(), is(1));

    User user2 = dao.get(user.getId());

    assertThat(user2.getName(), is(user.getName()));
    assertThat(user2.getPassword(), is(user.getPassword()));
  }
```

2.3.3 포괄적인 테스트

- 테스트의 결과가 테스트 실행 순서에 영향을 받는다면 테스트를 잘못 만든 것이다.
  → 모든 테스트는 실행 순서에 상관없이 독립적으로 항상 동일한 결과를 낼 수 있도록 해야한다.
- 성공하는 테스트만 골라 만들지 말자
- 테스트를 작성할 때 부정적인 케이스 먼저 만드는 습관을 들이는게 좋다.

2.3.4 테스트가 이끄는 개발

- 테스트 주도 개발

2.3.5 테스트 코드 개선

- 공통 준비 작업과 정리 작업 : @Before, @After
- 테스트를 수행하는 방식
  1. 테스트 클래스에서 @Test가 붙은 public, void 형이며 파라미터가 없는 테스트 메소드를 모두 찾는다.
  2. 테스트 클래스의 오브젝트를 하나 만든다.
  3. @Before가 붙은 메소드가 있으면 실행한다.
  4. @Test가 붙은 메소드를 하나 호출하고 테스트 결과를 저장해둔다.
  5. @After가 붙은 메소드가 있으면 실행한다.
  6. 나머지 테스트 메소드에 대해 2~5번을 반복한다.
  7. 모든 테스트의 결과를 종합해서 돌려준다.

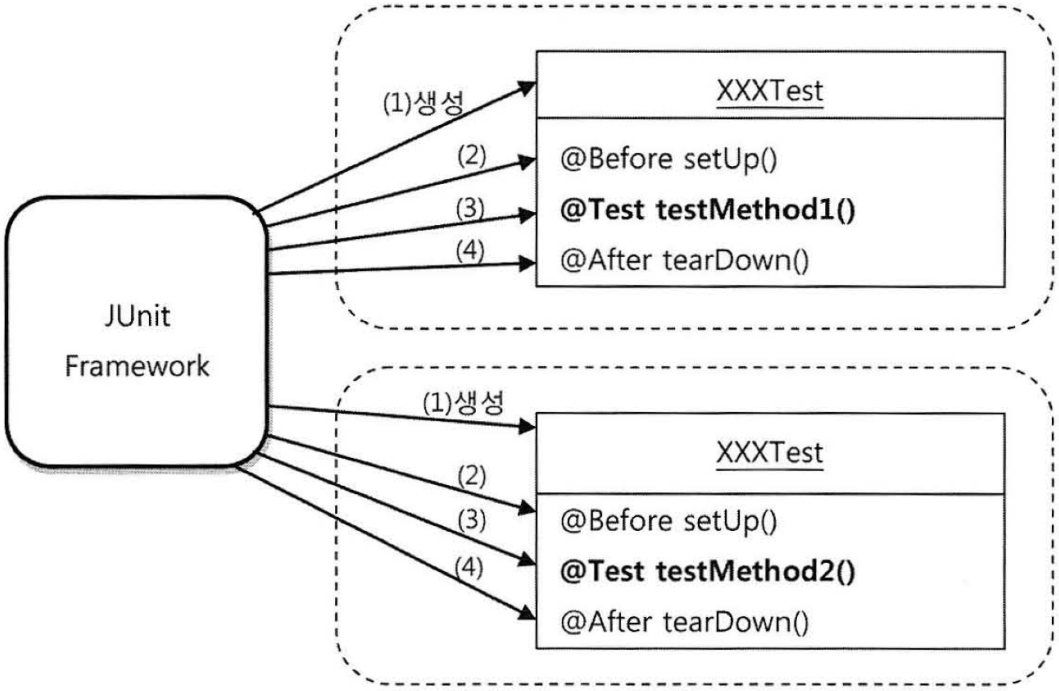

그림 2-4 JUnit의 테스트 메소드 실행 방법

각 테스트가 서로 영향을 주지 않고 독립적으로 실행됨을 보장하기 위해 매번 새로운 오브젝트를 만든다.

- 픽스처
  - 테스트를 수행하는데 필요한 정보나 오브젝트를 말한다.

리스트 2-16 User 픽스처를 적용한 UserDaoTest

```
public class UserDaoTest {
    private UserDao dao;
    private User user1;
    private User user2;
    private User user3;

    @Before
    public void setUp() {
        ...
        this.user1 = new User("gyumee", "박성철", "springno1");
        this.user2 = new User("leegw700", "이길원", "springno2");
        this.user3 = new User("bumjin", "박범진", "springno3");
    }
    ...
}
```

2.4 스프링 테스트 적용

2.4.1 테스트를 위한 애플리케이션 컨텍스트 관리

- 스프링 테스트 컨텍스트 프레임워크 적용

리스트 2-17 스프링 테스트 컨텍스트를 적용한 UserDaoTest

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="/applicationContext.xml")
public class UserDaoTest {
    @Autowired
    private ApplicationContext context;
    ...

    @Before
    public void setUp() {
        this.dao = this.context.getBean("userDao", UserDao.class);
        ...
    }
```

스프링의 테스트 컨텍스트 프레임워크의 JUnit 확장기능 지정

테스트 컨텍스트가 자동으로 만들어줄 애플리케이션 컨텍스트의 위치 지정

테스트 오브젝트가 만들어지고 나면 스프링 테스트 컨텍스트에 의해 자동으로 값이 주입된다.

`@RunWith` JUnit프레임워크의 테스트 실행 방법을 확장할 때 사용하는 어노테이션. `SrpingJUnit4ClassRunner` 라는 JUnit용 테스트 컨텍스트 프레임워크 확장 클래스를 지정해주면 JUnit이 테스트를 진행하는 중에 테스트가 사용할 애플리케이션 컨텍스트를 만들고 관리하는 작업을 진행한다.

`@ContextConfiguration` 은 자동으로 만들어줄 애플리케이션 컨텍스트의 설정파일 위치를 지정한 것이다

- 테스트 메소드의 컨텍스트 공유
  - 하나의 애플리케이션 컨텍스트가 만들어져 모든 테스트 메소드에 사용되고 있다
  - 스프링의 JUnit확장 기능은 테스트가 실행되기 전에 **딱 한번만 애플리케이션 컨텍스트를 만들어 두고** 테스트 오브젝트가 만들어질 때마다 특별한 방법을 이용해 **애플리케이션 컨텍스트 자신을 테스트 오브젝트의 특정 필드에 주입**해주는 것이다. (DI같지만 애플리케이션 오브젝트 사이의 관계를 관리하기 위한 DI와는 성격이 다르다)
- 테스트 클래스의 컨텍스트 공유
  - 여러 개의 테스트 클래스가 있는데 모두 **같은 설정 파일**을 가진 애플리케이션 컨텍스트를 사용한다면, 스프링은 테스트 클래스 사이에서도 **애플리케이션 컨텍스트를 공유**하게 해준다.
- @Autowired
  - DI에 사용되는 특별한 어노테이션
  - 테스트 컨텍스트 프레임워크는 변수 타입과 일치하는 컨텍스트 내의 빈을 찾는다. 타입이 일치하는 빈이 있으면 인스턴스 변수에 주입해준다. (타입에 의한 자동 와이어링)
  - 스프링 애플리케이션 컨텍스트는 초기화 할 때 자기 자신도 빈으로 등록한다.
  - 같은 타입의 빈이 두 개 이상 잇는 경우에는 타입만으로 어떤 빈을 가져올지 결정할 수 없다.
  - 테스트는 필요하다면 애플리케이션 클래스와 밀접한 관계를 맺고 있어도 상관없다. 개발자가 만드는 테스트는 코드 내부 구조와 설정 등을 알고 있고 의도적으로 그 내용을 검증해야 할 필요가 있기 때문이다.

    하지만 꼭 필요하지 않다면 테스트에서도 가능한 인터페이스를 사용해서 애플리케이션 코드와 느슨하게 연결해두는 편이 좋다.

2.4.2 DI와 테스트

- 인터페이스를 두고 DI를 적용해야 하는 이유
  1. 소프트웨어 개발에서 절대로 바뀌지 않는 것은 없다
  2. 클래스의 구현 방식은 바뀌지 않는다고 해도 인터페이스를 두고 DI를 적용하게 두면 다른 차원의 서비스를 도입할 수 있다.
  3. 테스트 때문이다. → 테스트를 잘 활용하려면 자동으로 실행 가능하며 빠르게 동작하도록 테스트 코드를 만들어야 하는데 그러기 위해선 가장 작은 단위의 대상에 국한해서 테스트 해야한다.

     DI는 테스트가 작은 단위의 대상에 대해 독립적으로 만들어지고 실행되게 하는 중요한 역할을 한다.
- 테스트 코드에 의한 DI
  - `@DirtiesContext` 테스트 컨텍스트 프레임워크에게 해당 클래스의 테스트에서 애플리케이션 컨텍스트의 상태를 변경한다는 것을 알려주고, 이 애노테이션이 붙은 클래스에는 애플리케이션 컨텍스트를 공유하지 않는다.
- 테스트를 위한 별도의 DI 설정
  - 두 가지 종류의 설정 파일을 만들어 하나에는 서버에서 운영용으로 사용할 DataSource 빈으로 등록해두고, 다른 하나에는 테스트에 적합하게 준비된 DB를 사용하는 가벼운 DataSource가 빈으로 등록되게 만드는 것이다.

```
@ContextConfiguration(location="/test-applictionContext.xml")
```

- 컨테이너 없는 DI 테스트
  - 스프링 컨테이너에 의존하지 않는다.

리스트 2-23 애플리케이션 컨텍스트 없는 DI 테스트

```
public class UserDaoTest {
    UserDao dao; ──→ @Autowired가 없다.
    ...

    @Before
    public void setUp() {
        ...
        dao = new UserDao();
        DataSource dataSource = new SingleConnectionDataSource(
            "jdbc:mysql://localhost/testdb", "spring", "book", true);
        dao.setDataSource(dataSource);
    }                    └──→ 오브젝트 생성, 관계설정 등을 모두 직접 해준다.
```

테스트를 위한 DataSource를 직접 만드는 번거로움은 있지만 애플리케이션 컨텍스트를 아예 사용하지는 않으니 코드는 더 단순해지고 이해하기 편해졌다.

애플리케이션 컨텍스트가 만들어지는 번거로움이 없어져 그만큼 테스트시간도 절약할 수 있다.

- 침투적 기술과 비침투적 기술
  - 침투적 기술: 기술을 적용했을 때 애플리케이션 코드에 기술 관련 API가 등장하거나, 특정 인터페이스나 클래스를 사용하도록 강제하는 기술
    - 애플리케이션 코드가 해당 기술에 종속되는 결과를 가져온다
  - 비침투적 기술: 애플리케이션 로직을 담은 코드에 아무런 영향을 주지 않고 적용 가능한 기술 → 기술에 종속적이지 않은 순수한 코드를 유지할 수 있게 해준다.
    - 스프링이 대표적인 예. 그래서 컨테이너 없는 DI테스트도 가능하다.
- DI를 이용한 테스트 방법 선택
  - 항상 스프링 컨테이너 없이 테스트할 수 있는 방법을 우선적으로 고려하자
    - 가장 빠르고 테스트 자체가 간결하다.
  - 여러 오브젝트와 복잡한 의존관계를 갖고 있는 오브젝트를 테스트해야 할 경우: 스프링의 설정을 이용한 DI방식의 테스트를 이용하면 편리하다.

    - 예외적인 의존관계를 강제로 구성해서 테스트해야할 경우: 컨텍스트에서 DI받은 오브젝트에 다시 테스트 코드로 수동 DI해서 테스트하는 방식을 사용하면 된다. @DirtiesContext를 붙여야 한다.

2.5 학습 테스트로 배우는 스프링

자신이 만들지 않은 프레임워크나 다른 개발팀에서 만들어 제공한 라이브러리 등에 대해서도 테스트를 작성해야하는데 이를 학습 테스트 라고 한다.

2.5.1 학습 테스트의 장점

- 다양한 조건에 따른 기능을 손쉽게 확인해 볼 수 있다.
- 학습 테스트 코드를 개발 중에 참고할 수 있다.
- 프레임워크나 제품을 업그레이드할 때 호환성 검증을 도와준다.
- 테스트 작성에 대한 좋은 훈련이 된다.
- 새로운 기술을 공부하는 과정이 즐거워 진다.

2.5.2 학습 테스트 예제

2.5.3 버그 테스트

코드에 오류가 있을 떄 그 오류를 가장 잘 드러내줄 수 잇는 테스트를 버그 테스트라 한다.

버그 테스트는 일단 실패하도록 만들어야 한다. 버그가 원인이 돼서 테스트가 실패하는 코드를 만드는 것이다. 그러고 나서 버그 테스트가 성공할 수 있도록 프로덕션 코드를 수정한다. 테스트가 성공하면 버그는 해결된 것이다.

- 테스트의 완성도를 높여준다
- 버그의 내용을 명확하게 분석하게 해준다.
- 기술적인 문제를 해결하는데 도움이 된다.
- 동등분할: 같은 결과를 내는 값의 범위를 구분해 각 대표 값으로 테스트를 하는 방법
- 경계값 분석: 에러는 동등 분할 범위의 경계에서 주로 많이 발생한다는 특징을 이용해 경계의 근처에 있는 값을 이용해 테스트하는 방법. 보통 숫자의 입력 값인 경우 0이나 그 주변 값 또는 정수의 최댓값 최솟값등으로 테스트한다.

2.6 정리

1. 테스트는 자동화돼야 하고, 빠르게 실행할 수 있어야 한다.
2. main()테스트 대신 JUnit프레임워크를 이용한 테스트 작성이 편리하다.
3. 테스트 결과는 일관성이 있어야 한다. 코드의 변경 없이 환경이나 테스트 실행 순서에 따라서 결과가 달라지면 안 된다.
4. 테스트는 포괄적으로 작성해야 한다. 충분한 검증을 하지 않는 테스트는 없는 것보다 나쁠 수 있다.
5. 코드 작성과 테스트 수행의 간격이 짧을수록 효과적이다.
6. 테스트하기 쉬운 코드가 좋은 코드다.
7. 테스트를 먼저 만들고 테스트를 성공시키는 코드를 만들어가는 테스트 주도 개발 방법도 유용하다.
8. 테스트 코드도 애플리케이션 코드와 마찬가지로 적절한 리팩토링이 필요하다.
9. @Before, @After를 사용해서 테스트 메소드들의 공통 준비 작업과 정리 작업을 처리할 수 있다.
10. 스프링 테스트 컨텍스트 프레임워크를 이용하면 테스트 성능을 향상시킬 수 있다.
11. 동일한 설정 파일을 사용하는 테스트는 하나의 애플리케이션 컨텍스트를 공유한다.
12. @Autowired를 사용하면 컨텍스트의 빈을 테스트 오브젝트에 DI할수 있다.
13. 기술의 사용 방법을 익히고 이해를 돕기 위해 학습 테스트를 작성하자.
14. 오류가 발견될 경우 그에 대한 버그 테스트를 만들어두면 유용하다.