2장 테스트 (2.1~2.3)

여재환

3줄 요약

- 테스트의 가치
- 테스트의 철학
- 테스트 기법

Intro

- 스프링의 핵심 가치

  1. 객체지향
  2. 테스트

- 1장에서 본 IoC / DI 를 통한 오브젝트의 생성, 관계, 사용에 관한 기술은 객체지향의 철학을 개발자가 손쉽게 적용할 수 있게 해준다.
- 그렇다면 테스트는 뭘까?

Intro

- 테스트는 만들어진 코드에 확신을 주고, 변화에 유연하게 대처할 수 있는 자신감을 주는 기술
- 다양한 기술을 이해하고 검증하며 실전에 적용하기 위해서 테스트는 스프링을 학습하는데 있어 가장 효과적인 방법의 하나이다.

2.1 UserDaoTest 다시보기

1장에서 만든 UserDaoTest는

- main 메서드를 통해 UserDao 오브젝트의 add(), get() 메서드를 호출하고 결과를 콘솔에 출력해서 눈으로 확인할 수 있게 해준다.
- 이 과정을 통해 코드를 수정하더라도 처음과 동일한 결과가 나오는지 쉽게 확인할 수 있었다.
- 만약 이러한 테스트 코드가 없었다면?

웹을 통한 Dao 테스트의 문제점

- 테스트 코드가 없었다면 직접 실행해야 한다.
- Dao만 테스트 하고 싶더라도 웹의 모든 계층을 통하여 Dao 가 잘 사용되나 확인해야 한다.
- 즉 Dao가 아닌 외부 요인에 의해 문제가 발생할 소지가 크고 문제가 생긴다면 어디가 문제인지 한번에 파악하기 힘들다.
  - 서비스, 컨트롤러, 뷰 등 어느 계층에서 나타나는 문제인지 파악하기 어렵다.

작은 단위의 테스트

- Unit test
- 한가지 관심에 집중할 수 있게 작은 단위로 만든 테스트
- UserDao를 테스트하기 위해 웹의 어떠한 계층도 필요없이 Dao 그 자체만을 테스트하는 기법
- 단위는 정량화된 것이 아니라, 하나의 관심사가 하나의 단위
  - 즉 관심사의 촛점에 따라 그 범위가 클수도, 혹은 단 하나의 함수일 수도 있다.

작은 단위의 테스트

- 효과적인 unit test 방법

1. 일반적으로 단위는 작으면 작을수록 좋다.

   에러 상황에 대해 좀 더 빠르고 간편하게 대처할 수 있다.

2. 외부 리소스에 의존하지 않아야 한다.

   동일 입력에 대해 동일 출력을 보장할 수 있다.
   즉 항상 같은 테스트 결과를 보장한다.(멱등성)

지속적인 개선과 점진적인 개발을 위한 테스트코드

- 처음 만든 초난감 Dao 코드를 스프링을 이용한 완성도 높은 객체지향적 코드로 쉽게 만들 수 있었던 것은 테스트 코드 덕분.
- 테스트코드로 인해 코드를 수정하고 개선해 나가는 시간적 비용을 크게 아낄 수 있었다.
- 단순한 방법으로 일단 통과하는 테스트를 만들고 코드를 개선해 나가는 과정을 통해 쉽고 안정적으로 전체 코드 완성이 가능했다.

UserDaoTest 의 문제점

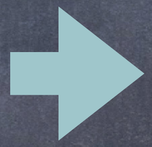
- 수동확인 작업의 번거로움
  - 콘솔의 출력값을 눈으로 보고 확인해야 함
- 실행 작업의 번거로움
  - 메인메서드를 통한 테스트는 한번에 하나의 테스트밖에 담당하지 못함
  - 따라서 코드의 규모가 증가함에 따라 매우 힘들어짐.

2.2 UserDaoTest 개선

테스트 검증의 자동화

- 기존의 테스트는 결과가 올바른지 눈으로 보고 머리로 판단해야 했다.

```
System.out.println(user2.getName());
System.out.println(user2.getPassword());
System.out.println(user2.getId() + " 조회 성공");
```



```
if(!user.getName().equals(user2.getName())) {
    System.out.println("test fail (name)");
} else if(!user.getPassword().equals(user2.getPassword())) {
    System.out.println("test fail (password)");
} else {
    System.out.println("test success");
}
```

- 테스트의 성공과 실패 결과 전달하여 보다 직관적인 테스트 결과 전달

테스트 검증의 자동화

- 테스트 검증 자동화로 얻은 장점
  - 테스트의 성공 실패만을 확인하면 된다.
  - 이후 코드 수정에 대해 테스트 코드만 통과한다면 side-effect 가 발생하지 않았음이 보장된다.
  - 코드 수정 이후 side effect 가 발생했다면 발생한 부분을 빠르게 찾고 대응할 수 있다.
  - 마음 편한 퇴근, 휴가가 보장된다.

테스트의 효율적인 수행과 결과 관리

- main 메서드에서 테스트를 수행함으로써 생기는 단점
  - 한번에 여러 테스트를 돌리기 어렵다.
  - 일정한 패턴을 공통적으로 적용시키기 까다롭다.

- 해결법 : JUnit

검증 코드 전환

```java
@Test
public void addAndGet() throws SQLException {
    ApplicationContext context = new ClassPathXmlApplicationContext( s: "application.xml");

    UserDao dao = context.getBean( bean: "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()));
}
```

- JUnit이 제공하는 assertThat() 메서드를 통해 테스트 코드 결과 파악 가능
- asseltThat의 2번째 파라미터 matcher를 조건으로 일치하면 다음으로 넘어가고 아니라면 테스트가 실패된다.

2.3
개발자를 위한 테스팅 프레임워크
JUnit

동일한 결과를 보장하는 테스트

- 책에서는 매 순간 동일한 상태를 만들기 위해
  getCount, deleteAll 과 같은 메서드를 만들어서 사용함
- 테스트를 위해 추가적인 메서드 작성을 권유하는것이 옳은 방향인지 의문
- 하지만 경우에 따라 필요한 상황이 있을 것으로 추정
  ex) NoSQL, 검색엔진, 메세지 큐 등을 활용하는 경우 등

예외조건에 대한 테스트

- 통과되는 테스트보다 실패하는 테스트가 더 중요하다!
  edge 케이스, 극단적인 경우를 항상 고려하자!

```java
@Test(expected = EmptyResultDataAccessException.class)
public void getUserFail() {
    ApplicationContext context = new ClassPathXmlApplicationContext( s: "application.xml");

    UserDao dao = context.getBean( bean: "userDao", UserDao.class);
    dao.deleteAll();

    assertThat(dao.getCount(), is( value: 0));

    dao.get("unknown_id");
}
```

테스트가 이끄는 개발

- 먼저 구상하는 바, 추가하고 싶은 기능에 대해 테스트 코드를 작성하고 테스트 코드를 통과하는 코드를 작성하는 기법
- 요구사항, 기능명세 == 테스트 코드
- Test Driven Development
- 실패한 테스트를 성공시키기 위한 목적이 아닌 코드는 만들지 않는다.
  - 즉, 일단 테스트 코드를 짜고, 이것을 성공시키기 위한 코드만을 작성한다.
- TDD != Test code

테스트 코드 개선

- 테스트 코드도 리팩토링이 필요하다.
- JUnit 동작 방식
  @Before -> @Test -> @After

```
private UserDao dao;

@Before
public void setUp() {
    ApplicationContext context = new ClassPathXmlApplicationContext( s: "application.xml");
    this.dao = context.getBean( bean: "userDao", UserDao.class);
}

@Test
public void test1() { }

@Test
public void test2() {}
```

테스트 코드 개선

- JUnit 은 각각의 테스트 코드를 실행할 때마다 인스턴스를 새로 생성
  - 다른 테스트 코드로 인한 side-effect를 철저히 막고 독립적으로 실행하기 위해
- Fixture
  - 테스트를 수행하는데 필요한 정보나 오브젝트
  - 여러 테스트에서 반복적으로 사용되므로 @Before으로 추출하는 것이 용이