chapter 9: Unit Tests

Three Laws of TDD (Test-Driven Development)

1. 실패한 단위 테스트가 있다면 코드를 작성하지 마라
2. 컴파일되지만 충분히 실패한 테스트가 있다면 테스트를 더 만들지 마라
3. 실패했던 테스트를 통과하는게 아니라면 코드를 더 쓰지 마라

테스트의 중요성

> 코드 작성과 테스트 작성은 **같이 이루어져야** 한다.

- dirty test → hard to change → dirty code!
- test code is Just as Important as Production Code
- 테스트가 잘 짜여있다면, 코드를 변경할 때 겁먹지 않아도 된다.
    - 오히려 전체 구조를 improve하게 되는 것이다!

Test Name convention

> given-when-then

ex. GetPageHierarchyHasRightTags()

Clean Test

> READABILITY!!!

변경 전

SerializedPageResponderTest.java

```
public void testGetPageHieratchyAsXml() throws Exception
{
  crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
       new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}
```

- “terrible amount of duplicate codes”
- 무슨 테스트인지 도통 이해할 수가 없다
- 또한 중간중간 쓸데없이 뭘 바꾸는 코드들이 많아서 테스트의 본 의미를 상실하고 있다.

변경 후

```
public void testGetPageHierarchyAsXml() throws Exception {
  makePages("PageOne", "PageOne.ChildOne", "PageTwo");

  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
  );
}
```

- 전부 함수로 뺐다. 또한 중복도도 줄어 가독성이 향상되었고, 필요없는 코드들이 많이 제거되었다.
- 오직 data type, function, 이름만 가지고 테스트할 수 있다!

두번째 예시

```
@Test
  public void turnOnLoTempAlarmAtThreashold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
  }
```

변경 후

```
@Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

- 흠...HBchL이 뭔데? 할 수 있겠으나 변경 전 코드처럼 최소한 “eyes glide across” 상황은 피할 수 있다.
- 또한 getState()의 경우 오른쪽처럼 조금 비효율적으로 구현되어 있다.
  - 만약 성능이 중요한 코드였다면 StringBuffer을 사용했겠으나, test하는 입장에서는 사실 별 상관이 없다고 한다.
  - 오히려 깔끔하게 작성하는 것이 더 중요하다.

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
    return state;
  }
```

F.I.R.S.T Rule

1. **Fast**
   - 빠르게 돌아가야 한다!
2. **Independent**
   - 다른 테스트끼리 서로 영향을 주거나, 하나를 실패하면 다른것도 실패한다거나 하는 식이면 안된다.
3. **Repeatable**
   - 어느 환경에서든(배포, QA, 노트북, ...) 동일하게 돌아가야 한다.

4. Self-**validating**
    - Boolean 형식으로 반환해야 한다. 로그파일, 출력파일을 뒤져가면서 확인하면 안 된다는 뜻이다.
5. **Timely**
    - code를 쓰기 **직전**에 test를 작성해라.

One Assert Per Test

테스트 하나당 assert문을 단 하나만 작성하는 convention이 있다고 한다.
그러나 그걸 지키기 위해 억지로 테스트를 쪼개다 보면 코드 중복이 생긴다.

따라서 적당히 두세개 정도는 같이 써도 되고, template을 사용하던가 하는 방식을 사용할 수도 있겠다.

Single Concept per Test

딱 하나의 assert를 쓰는 것 보다 지켜야 하는 것은

하나의 컨셉을 유지하는 것이다.

- 한 테스트 함수에서 여러가지를 테스트하면 복잡해질 뿐이다
- 따라서 테스트를 여러개로 나눠서 잘 작성하도록 하자