JUnit 프레임워크

상태	Java

목차

단위 테스트(Unit Test) vs 통합 테스트(Integration Test)

- **단위 테스트 (Unit Test)**

: 하나의 모듈(애플리케이션에서 작동하는 하나의 기능 또는 메소드)을 기준으로 독립적으로 진행되는 가장 작은 단위의 테스트

→ 애플리케이션을 구성하는 하나의 기능이 올바르게 동작하는지 독립적으로 테스트

- **통합 테스트 (Integration Test)**

: 모듈을 통합하는 과정에서 **모듈 간의 호환성을 확인**하기 위해 수행되는 테스트

→ 웹 페이지로부터 API를 호출하여 **통합된 모듈들이 올바르게 연계**되어 동작하는지 검증하는 것

- **단위 테스트를 해야 하는 이유**

  1) 코드를 수정하거나 기능을 추가할 때 수시로 **빠르게 검증** 가능

  : 실제로 어플리케이션을 실행하여 수동으로 테스트하지 않아도 된다.

  직접 테스트하는 비용이 크다.

  2) 리팩토링 시 **안정성 확보** 가능

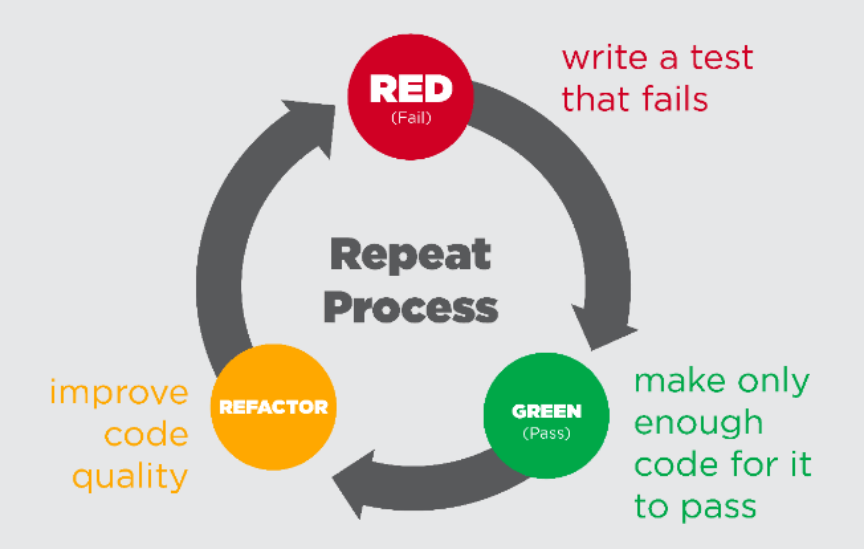
  3) 개발 및 테스팅에 대한 **시간과 비용 절감** 가능

TDD (Test Driven Development)

: 개발자가 아직 구현되지 않은 기능을 테스트 가능한 작은 단위로 나누고 이러한 단위에 대해 단위 테스트 코드를 먼저 작성하는 개발 방법론 중 하나로 소프트웨어 개발 프로세스를 품질 관리 및 안정성 향상을 위한 중요한 도구로 활용

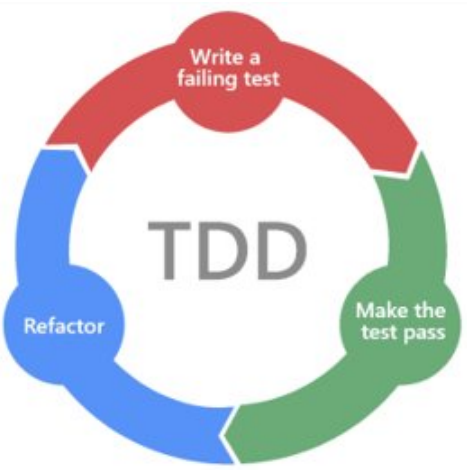
TDD의 사상 : "내가 설계한 모든 잘못된 점을 수정하면 비로소 올바름이 된다"

1. TDD의 개발주기 : **Red-Green-Refactor 사이클**



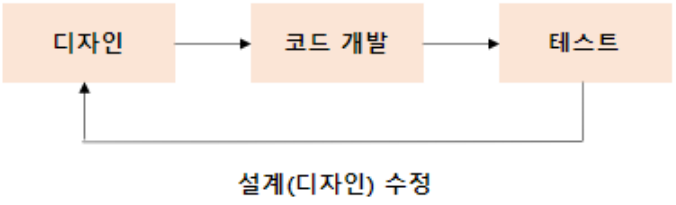
- **Red 단계** : 실패하는 테스트 케이스를 작성하는 단계. 해당 테스트 케이스는 아직 작성되지 않은 코드에 대한 요구 사항을 정의합니다.
- **Green 단계** : 실패하는 테스트 케이스를 통과하기 위한 최소한의 코드를 작성하고 테스트를 통과

- **Refactor 단계** : 중복 코드 제거, 일반화 등의 리팩토링



2. **일반적인 개발 방식 🆚 TDD 개발 방식**

[일반적인 개발 방식]

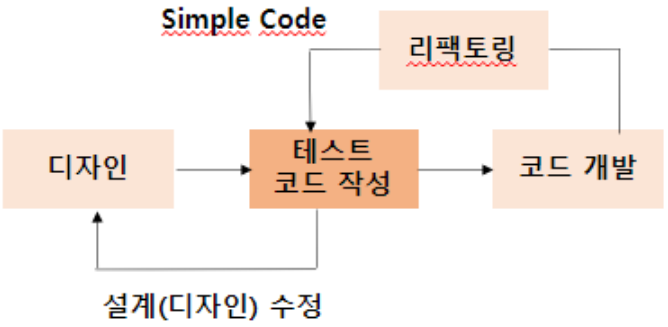


해당 방식은 개발을 느리게 하는 **잠재적 위험** 존재

- 소비자의 요구사항이 **처음부터 명확하지 않을 수 있음**
- 처음부터 **완벽하게 설계 어려움**
- 자체 버그 검출 능력이 저하로 품질 저하
- **자체 테스트 비용이 증가**할 수 있음

⇒ 즉, 어느 프로젝트든 초기 설계는 완벽할 수 없음

[TDD 개발 방식]



일반 개발 방식과 가장 큰 차이점은 **실제 코드를 작성하기 전에 테스트 코드를 작성한다는 것**

디자인(설계) 단계에서 프로그래밍 목적을 반드시 **미리 정의**해야만 하고,

또 무엇을 테스트해야 할지 **미리 정의(테스트 케이스 작성)**해야만 한다.

1. 테스트 코드를 작성하는 도중에 발생하는 **예외 사항(버그, 수정사항)들은 테스트 케이스에 추가**하고 설계를 개선한다.
2. 이후 테스트가 통과된 코드만을 코드 개발 단계에서 실제 코드로 작성한다.

→ **코드의 버그가 줄어들음**

→ 소스 코드가 간결해짐

→ 자연스럽게 설계를 하도록 지원해줌

🤢 현실적인 TDD의 단점

- 생산성의 저하 : TDD 방식의 개발 시간이 일반 개발 방식에 비해 대략 **10 ~ 30%** 정도로 늘어남 (특히 SI 업체에서 납기일 준수를 중요시하기 때문에 많이 힘들 수 있음)
- 지금까지의 **개인 개발 방식을 바꾸어야 함** → 적용하는 데에 시간 걸림

📎 단위 테스트와 TDD의 연관성

: TDD를 따를 때, 개발자는 아직 구현되지 않은 기능을 테스트 가능한 작은 단위로 나누고, 이러한 단위에 대한 단위 테스트를 작성합니다.

좋은 단위 테스트의 특징

1개의 테스트 함수는 1가지 개념만 테스트

[FIRST 규칙] - CleanCode 책

(1) Fast : 테스트는 빠르게 동작하여 자주 돌릴 수 있어야 함

(2) Independent : 각각의 테스트는 독립적이며 서로 의존 X

(3) Repeatable : 어느 환경에서든 반복 가능

(4) Self-Validating : 테스트는 성공 또는 실패로 Boolean값을 결과로 내어 자체적으로 검증되어야함

(5) Timely : 테스트는 테스트하려는 실제 코드를 구현하기 직전에 구현되어야 함

TDD의 대표적인 Tool JUnit 프레임워크

: 지금 최신 버전은 **5.10.0**

JUnit 기본 개념

: Java에서 주로 사용되는 독립된 단위 테스트 프레임워크 중에 하나로 보이지 않고 숨겨진 단위 테스트를 끌어내어 정형화시켜 단위 테스트를 쉽게 해주는 테스트용 Framework

JUnit 특징

- 단위 테스트 Framework 중 제일 많이 사용되는 프레임워크

  (다른 프레임워크 : TestNG - 쓰이는 기능들이나 Annotation들이 다름)
- @Test 메서드가 호출할 때마다 새로운 인스턴스가 생성되어 독립적인 테스트 가능
- 단정문(AssertJ)으로 테스트케이스의 수행결과를 판별
- 결과는 성공(녹색), 실패(붉은색) 표시

JUnit 기본 Annotation → 실습

: JUnit4와 JUnit5 버전 Annotation들이 조금씩 다름

(1) **@Test** : 테스트 메소드 지정하기

(2) **@DisplayName**

: 테스트 클래스 또는 테스트 메서드의 이름 정의 가능

(3) **@Test(timeout=5000)** : 시간 단위는 밀리 초

: 해당 메소드가 결과를 반환하는데 5000 밀리 초를 넘기면 테스트 실패

(4) **@Test(expected=RuntimeException.class)**

: 해당 메소드가 RuntimeException이 발생해야 테스트 성공

(5) **@BeforeClass, @AfterClass (ver.JUnit4) ⇒ @BeforeAll, @AfterAll (ver.JUnit5)**

: 대상은 class, 클래스 안에 정의된 모든 @Test 메소드들이 수행되기 전과 후에 한 번씩만 호출

: 해당 메서드는 static 으로 정의되어야 함

(6) **@Before, @After (ver.JUnit4) ⇒ @BeforeEach, @AfterEach**

: @Test 메소드들이 호출되기 직전과 직후에 각각 실행된다.

: @Test 메소드들의 성공/실패 여부와는 상관없이 실행

: 필요한 도메인 객체를 미리 생성하거나, **특정 상태로 미리 세팅하기 위한 공동 코드를 뽑아내는 목적으로 많이 사용**

(7) **@Disabled**

: 테스트 클래스 또는 메서드를 비활성화

: 이전의 @Ignore와 동일

JUnit 사용법 : given/when/then 패턴

- **given (준비)** : 테스트를 위한 준비 과정 (변수 선언 및 Mock 객체 정의)
- **when (실행)** : 테스트 실행 (테스트하고자 하는 내용 작성)
- **then (검증) :** 테스트 검증 (예상값과 결과값 일치 여부 확인) → 단정문(Assertions) 사용

단정문 (Assertions)

: assert 구문은 기대값과 반환된 실제 값을 비교

Assertions 메서드 정리 (자주 쓰이는 메서드 위주로)

JUnit Jupiter

JUnit 프레임워크에서 기본적으로 제공하는 Assertions

- assertTrue, assertFalse (boolean conditions, String message)

  : 가장 기본적인 assert 구문

  : 참과 거짓 비교

  : boolean 값을 리턴하는 메서드를 테스트하기에 적합함

  : 해당 테스트 실패할 경우, message 출력

- assertNull, assertNotNull(Object object, String message)

  : 객체를 넘겨 null인지 아닌지 판단

- assertEquals, assertNotEquals (int a, int b)

  : 파라미터값 2개가 일치하는지 판단

  : a - expected / b - actualz

  - assertEquals(a, b, c)

    : a - 예상값, b - 결과값 , c - 오차범위

⇒ 이 메소드들은 assert에서 Exception이 발생할 경우 바로 즉시 모든 Test를 멈추고 에러메시지를 띄우게 된다.

> JUnit Jupiter에서 제공하는 어설션 기능이 많은 테스트 시나리오에 충분하더라도 더 많은 성능과 *일치기와*같은 추가 기능이 필요하거나 필요한 경우가 있습니다. 이러한 경우 JUnit 팀은 AssertJ , Hamcrest , Truth 등과 같은 타사 주장 라이브러리를 사용할 것을 권장합니다. 따라서 개발자는 원하는 주장 라이브러리를 자유롭게 사용할 수 있습니다.

AssertJ

JUnit5 에서 자주 사용하는 라이브러리 : AssertJ 라이브러리에서 제공하는 Assertions

```
import static org.assertj.core.api.Assertions.*;


assertThat("확인할 대상")
    .isNotNull() // null 값이 아닌가
    .startsWith("Hello") // "Hello 로 시작하는가?"
    .contains("good")  // "good" 을 포함하는가?
    .isInstanceOf(String.class);  // "String class" 타입인가?
```

메서드 체이닝이 가능하여 코드 깔끔하고 표현이 명확함

❓ AssertJ vs Hamcrest

TDD 방식으로 테스트 코드 작성하는 방법

규칙

- **쉬운 경우 → 어려운 경우**

  : 어려운 경우로 테스트 코드 작성을 시작하면 한 번에 구현해야 하는 코드가 많아짐

- **예외적인 경우 → 정상적인 경우**

  : 처음부터 예외 상황을 고려하지 않고 코드 작성할 경우 나중에 코드를 뒤집거나 복잡하게 만들 수 있음.

- **예시**
  - 회원 가입 : **동일 이메일이 존재 O** 🆚 동일 이메일이 존재 x (예외적인 케이스)
  - 만료일 계산 : 1월 31일에서 한달 뒤 🆚 **1월 25일에서 한달 뒤** (쉬운 케이스)
  - 회원 주소 변경 : **회원이 없는 경우** 🆚 회원이 있는 경우 (예외적인 케이스)

TDD에서 중요한 것은 **완급 조절**

- TDD로 통과시키는 과정
  - 정해진 값 바로 리턴 → 값 비교를 이용하여 정해진 값 리턴 → 테스트 추가하며 일반화 진행
- 구현이 생각나면 빠르게 구현
  - 단, 테스트 통과시킬 만큼만
  - 앞서서 테스트 **X**
- 구현이 막히면 다시 뒤로 돌아와서 천천히 진행

실습 : 휴대폰 인증번호 SMS 문자 요청 API

< 해당 API 프로세스 설명 >

1) 핸드폰 번호를 parameter로 받기 → 핸드폰 번호 validation 체크 기능

2) 무작위 인증번호 6자리 생성 → 인증번호 생성 기능

3) 해당 데이터 DB에 저장

4) 해당 핸드폰 번호로 인증번호 문자 전송 → 문자 전송 기능

<예시> 핸드폰 번호 validation 체크

- 핸드폰 번호가 입력이 안된 경우(null값인 경우) → 예외상황 + 쉬움
- 핸드폰 번호가 11자리가 아닌 경우 → 예외상황 + 쉬움
- 핸드폰 번호가 정상적으로 들어오는 경우 → 쉬움
- 핸드폰 번호가 정상적인 형태가 아닌 경우 → 예외상황 + 쉬움

```
package org.compassion.common.example;

import lombok.extern.slf4j.Slf4j;
import org.compassion.common.controller.CommonController;
import org.compassion.common.dto.TextMessageDto;
import org.compassion.common.dto.smsAuthDto;
import org.compassion.common.service.CommonServiceImpl;
import org.junit.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.configuration.IMockitoConfiguration;
import org.mockito.junit.jupiter.MockitoExtension;

import javax.servlet.http.HttpServletResponse;
import javax.xml.soap.Text;
import java.security.SecureRandom;
```

```
import static org.junit.Assert.*;
import static org.assertj.core.api.Assertions.*;
import static org.mockito.Mockito.*;

@Slf4j
@ExtendWith(MockitoExtension.class)
public class requestPhoneVerifyTest {



    /******** 핸드폰 번호 validation 체크 기능 ********/

    // 핸드폰 번호가 null 인 경우
    @Test
    public void testPhoneNumIsNull() {
        String phoneNumber = null;

        checkPhoneNumber(phoneNumber);

    }

    // 핸드폰 번호가 11자리가 아닌 경우
    @Test(expected = RuntimeException.class)
    public void testPhoneNumIsNotCorrect() {
        String phoneNumber = "0000000";

        checkPhoneNumber(phoneNumber);

    }

    // 이동통신 전화번호가 아닌 경우 (010~ 으로 시작하지 않는 경우)
    @Test(expected = RuntimeException.class)
    public void testPhoneNumIsNotCorrect2() {
        String phoneNum = "02085695138";

        assertTrue(checkPhoneNumber(phoneNum));

    }


    private boolean checkPhoneNumber(String phoneNumber) {

        if(phoneNumber == null || phoneNumber.length() != 11) {
            throw new RuntimeException("핸드폰 번호 입력 안됨");
        }

        String phoneCode = phoneNumber.substring(0, 3);

        if(!("010".equals(phoneCode))) {
            throw new RuntimeException("핸드폰 번호 비정상");
        }

        return true;


    }

    /******** 인증번호 생성 체크 기능 ********/
    @Test
    public void checkVerificationCodeIsCorrect() throws Exception {
```

```
        if(makeVerificationCode().length() != 6) {
            throw new RuntimeException("인증번호 생성 잘못되었음");
        }

        log.info("인증번호 생성 잘 되었음 : {}", makeVerificationCode());

    }

    private String makeVerificationCode() throws Exception{

        SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
        secureRandom.setSeed(secureRandom.generateSeed(128));
        String verificationCode = String.valueOf(secureRandom.nextInt((999999 - 1000) + 1) + 1000);


        return verificationCode;
    }



}
```

Mockito를 이용한 단위 테스트

Mockito 란

: 단위 테스트를 위한 가짜 객체(Mock)를 지원하는 테스트 프레임워크

: mock을 쉽게 만들고 mock의 행동을 정하는 stubbing, 정상적으로 작동하는지 verify 등 다양한 기능을 제공해주는 프레임워크

: Mockito를 활용하면 가짜 객체에 원하는 결과를 **Stub** 하여 단위 테스트 진행 가능

Mock 객체 (모의 객체) 란

: “모의, 가짜의”

: 테스트할 때 필요한 실제 객체와 동일한 가짜 객체

: 그 안에 메소드 호출 시, 반드시 **스터빙(Stubbing)** 해야함

```
Stub 란?

- 사전적 의미 : 토막, 꽁초, 남은 부분
- 개발에서의 의미 : 실제로 준비는 되지 않았지만 원활한 테스트를 위해 메서드 호출에 미리
                    정해진 답을 반환하는 구현체
```

```
# Stubbing 방법
when(호출하고자 하는 메서드).then(미리 지정하는 값);
```

Mock 객체의 사용 이유

```
/*** 예시 1 ***/

CommonController의 requestPhoneVerificationCode에 대한 단위 테스트 진행 시,
CommonService의 insertSmsAuthCode() 를 실행하는 코드가 있을 때,
현재 단위테스트 진행 시, requestPhoneVerificationCode() 함수가 insertSmsAuthCode까지
정상적으로 호출하는 지에 대한 여부가 관심사다.
즉, insertSmsAuthCode()가 어떻게 구현되는지는 지금의 관심사가 아니기 때문에
정상적인 리턴 값을 던지는 Mock 객체로 만들어 오로지 requestPhoneVerificationCode() 메소드에
집중할 수 있도록 해줌


/*** 예시 2 ***/
UserController에 대한 단위 테스트 진행 시,
UserService를 주입받고 있다면
@Mock 어노테이션을 통해 가짜 UserService를 만들고,
@InjectMocks를 통해 UserController에 주입
```

Mockito에서 가짜 객체의 의존성 주입을 위한 Annotation

- **@Mock** : 가짜 객체를 만들어 반환

  → 그 안에 메소드 호출해서 사용하려면 **반드시 스터빙(stubbing)** 해야함

  when(…).thenReturn(…)

  → 만약, 스터빙 하지 않고 호출 시, **primitive type은 0 / 참조형은 null 반**

```
@Mock 어노테이션 : Mockito는 해당 필드를 Mock 객체로 초기화하고 필요한 설정을 자동으로 처리
=> 좀 더 간결하게 사용되어짐.


mock() 메서드 : Mockito 클래스의 정적 메서드로, Mock 객체를 수동으로 생성하고 설정
```

- **@Spy** : 실제 객체를 사용해서 mock하는 annotation

  진짜 오브젝트를 생성. Spy는 Mock과 달리 stub이 필요한 부분에만 stubbing 할 수 있음.

  → Mock을 사용할 경우, 모든 메서드가 mocking 처리가 되지만, Spy를 사용하면 하나의 메소드에 대한 Mocking 처리가 가능해진다.

- **@InjectMocks** : @Mock 또는 @Spy로 생성된 가짜 객체를 자동으로 주입

[Mockito annotation을 사용하는 field 초기화하는 방법 2가지]

1) JUnit test class 에 **@RunWith(MockitoJUnitRunner.class)** 추가

- @ExtendWith(MockitoExtension.class) : JUnit5와 Mockito 연동을 위한 Annotation

  @RunWith(MockitoJUnitRunner.class) : JUnit4와 Mockito 연동을 위한 Annotation

```
@RunWith(MockitoJUnitRunner.class)
public class MockitoTest {

    /** 코드 작성 **/

}
```

2) **MockitoAnnotations.initMocks(Object)** 을 @Before 메서드에서 실행

```
// 해당 소스코드 작성 안할 경우, Mock 객체에서 nullPointerException 발생

@Before
pubic void setUp(){
      MockitoAnnotations.initMocks(this);
  }
}
```

Controller 단위 테스트

```
@Slf4j
@RunWith(SpringJUnit4ClassRunner.class)
public class commonControllerTest extends TestConfig {

    private MockMvc mockMvc;

    @Autowired
    CommonController commonController;

    @Before
    public void setUp() throws Exception {

        mockMvc = MockMvcBuilders.standaloneSetup(commonController).build();
        log.info("setUp 완료");

    }

    @Test
    public void requestVerificationCodeTest() throws Exception {


        mockMvc.perform(post("/common/requestPhoneVerificationCode")
                .param("phoneNumber", "01022212"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.resultCode").value(0))
                .andDo(print());
    }

}
```

JUnit 세미나 ppt.pdf

참고자료

https://mangkyu.tistory.com/143

https://inpa.tistory.com/entry/QA-📚-TDD-방법론-테스트-주도-개발

https://recordsoflife.tistory.com/338

https://jwkim96.tistory.com/168

https://dding9code.tistory.com/117

https://velog.io/@wonizizi99/Spring-JUnit5-Mockito-이용한-테스트-코드

https://velog.io/@galaxy/JUnit을-활용해-Controller-단위테스트하기

https://minholee93.tistory.com/entry/JUnit-Stub

https://kimcoder.tistory.com/418

https://okky.kr/questions/269348

https://velog.io/@jkijki12/Spring-MockMvc