

Software Testing

라이엇게임즈코리아
소프트웨어 엔지니어 정종오

1주차

- Software Testing의 정의와 목적
- 테스트의 종류
- 단위 테스트(Unit Test)
- 단위 테스트 작성해 보기

2주차

- Mocking 개념과 사용 방법
- TDD 개념 소개



1 주차

테스트를 왜 해야 할까요?

생산된 제품들을
사용자들이
아무런 문제없이
사용할 수 있어야 한다.



소프트웨어도
테스트를 해야 하나요?

인도네시아 여객기 추락

737 맥스 보잉사 치명적 소프트웨어 결함 발견

입력 2018.11.08 18:49

👍 좋아요 1개



가

가



이황구

정책사회부장



정치

경제

사회

국제

문화

연예

라이프

스포츠

피플

지역

오피니언

기획특집

디지털스페셜

멀티미디어

비즈니스경제 / 애플리케이션

©2013.10.18

코딩 오류로 880만 달러 비용 초래... 호주 통신사 사례

Adam Bender | Computerworld

2년 이상이나 발견되지 않은 소프트웨어 코딩 오류 때문에 호주 통신사 옵터스(Optus)가 모바일 선불 서비스를 이용하는 일반 소비자와 중소기업 고객들에게 약 23만 5,000호주달러를 잘 못 청구한 것으로 알려졌다.



“코드 한 줄 오류 때문에 리콜”...차량 SW 및 기능 안전 중요성 갈수록 커진다

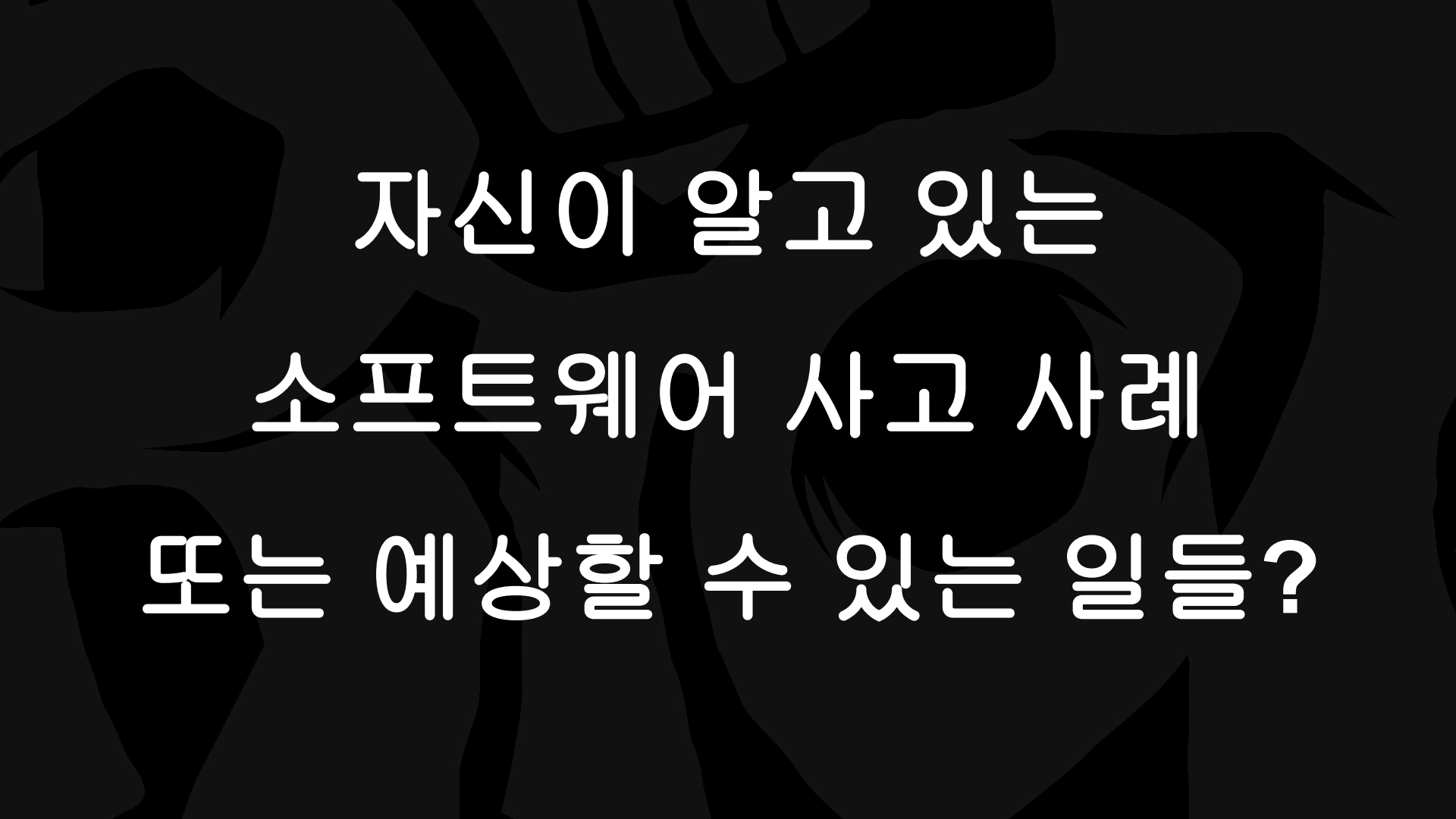
발행일 : 2015.02.26



가

가





자신이 알고 있는
소프트웨어 사고 사례
또는 예상할 수 있는 일들?

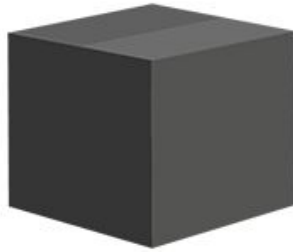
소프트웨어 테스트

- *Software testing is the process of evaluation a software item to detect difference between given input and expected output*
- 소프트웨어 테스트: 소프트웨어를 검사해 주어진 입력 내용과 기대되는 출력 내용 간 차이를 확인하는 과정

“테스트는 프로그램이나 시스템이
예상대로 작동할 것이라는 확신을
증진시키는 과정이다.”

- Hetzel 1973

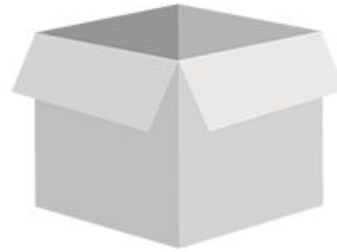
Black Box vs. White Box Testing



Interface-Level

Testers execute functional and regression tests via user interface

E.g. functional tests, regression tests, UI tests



Code-Level

Testers (or developers) look directly at code to find bugs and errors

E.g. unit tests, load tests



rainforest

Verification

Validation

?



Verification

- 개발자가 의도한 대로 동작하는가

Validation

- 각각의 요구 사항에 부합하게 동작하는가

Unit Testing

Performance Testing

Integration Testing

Usability Testing

Functional Testing

Acceptance Testing

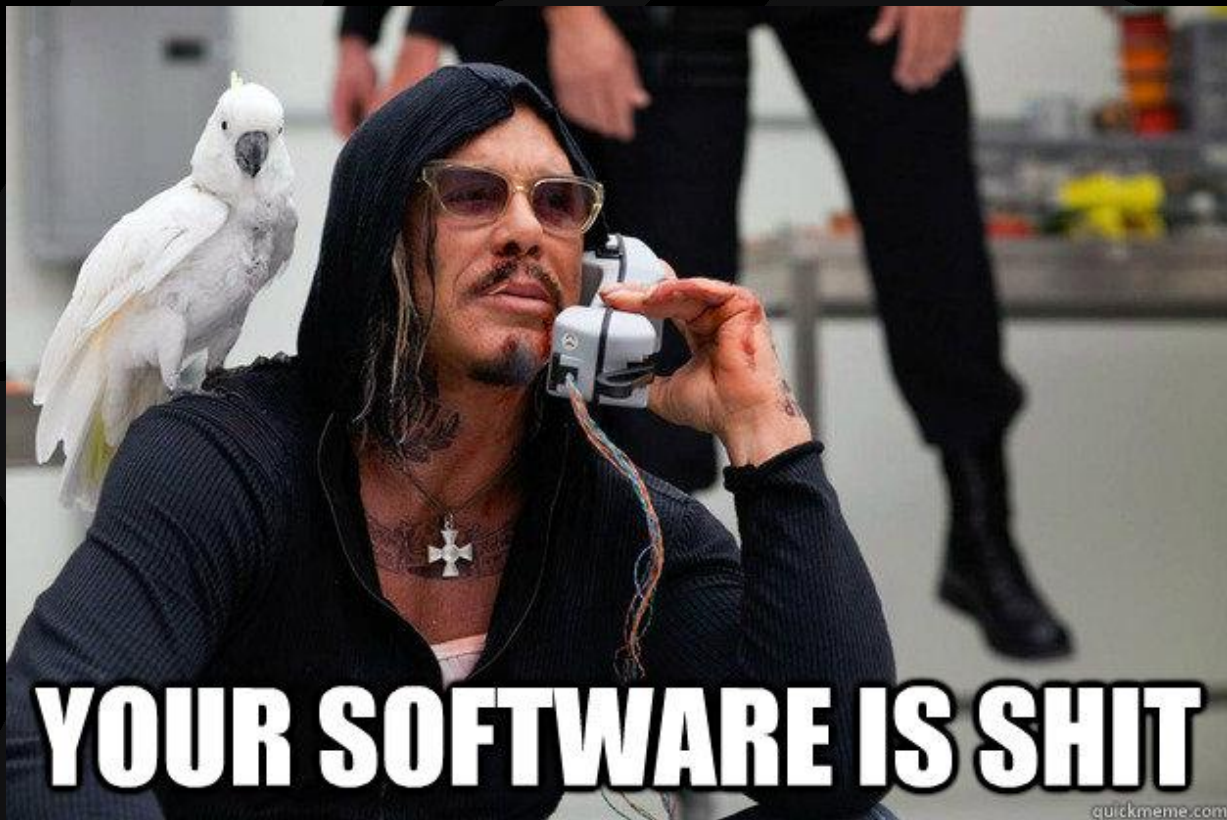
System Testing

Regression Testing

Stress Testing

Beta Testing

실사용자로부터의 신랄한 평가 피드백



단위 테스트(Unit Test) - 특징

- 가장 작은 단위의 테스트
- 개발 의도를 간단 명료하게 표현해 줌
- 가장 빠름
- 외부 요인에 영향을 받지 않음

단위 테스트(Unit Test) - 장점

- 테스트 코드량이 적다!
- 개발자를 위한 코드 안내서 역할을 할 수 있다!
- 코드 작성 중에도 반복적으로
실행해 볼 수 있다!

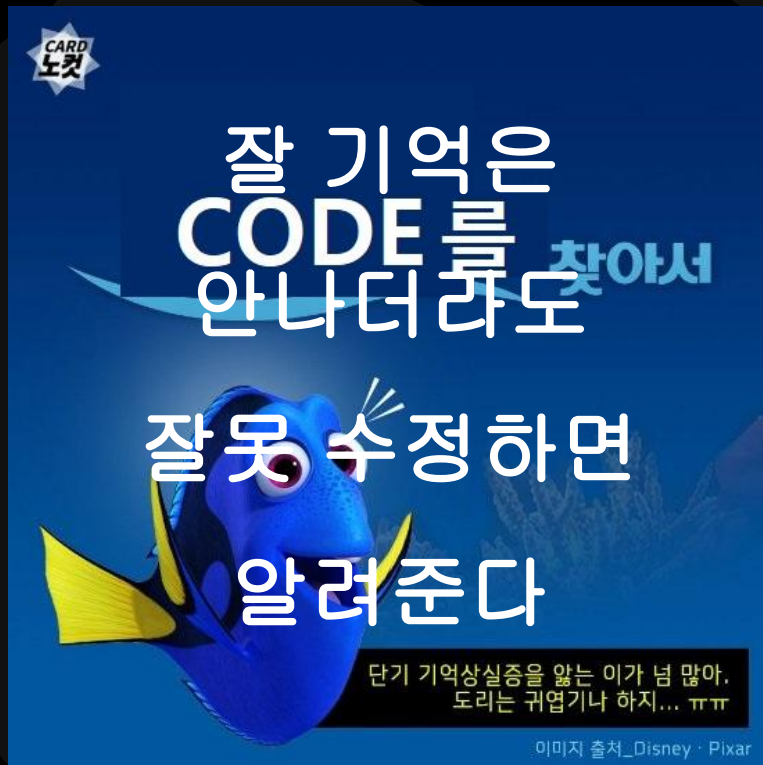
개발자들은 왜 Unit Test 를 만들까?

본인의 코드를
신뢰하지 못함



개발자들은 왜 Unit Test 를 만들까?

사라지는 기억
이걸 내가
짚다고?



개발자들은 왜 Unit Test 를 만들까?

다른 코드로부터
내 코드를
보호한다



다른 코드의
잘못된 영향을
코드로부터
에러를 발생한다

개발자들은 왜 Unit Test 를 안만들까?

- 버그 고치기에도 시간이 빠듯하다
- 내가 만들어도 다른 사람은 안만들어서 소용 없다
- 내가 작성하는 코드에 관해서는 내가 제일 잘 알고 있어서 필요 없다

개발자들은 왜 Unit Test 를 안만들까?

- 빨리 수정사항을 배포해야 하는데 테스트를 돌리면 오래 기다려야 한다
- 테스트가 깨지면 테스트 코드를 고치느라 배보다 배꼽이 더 커진다

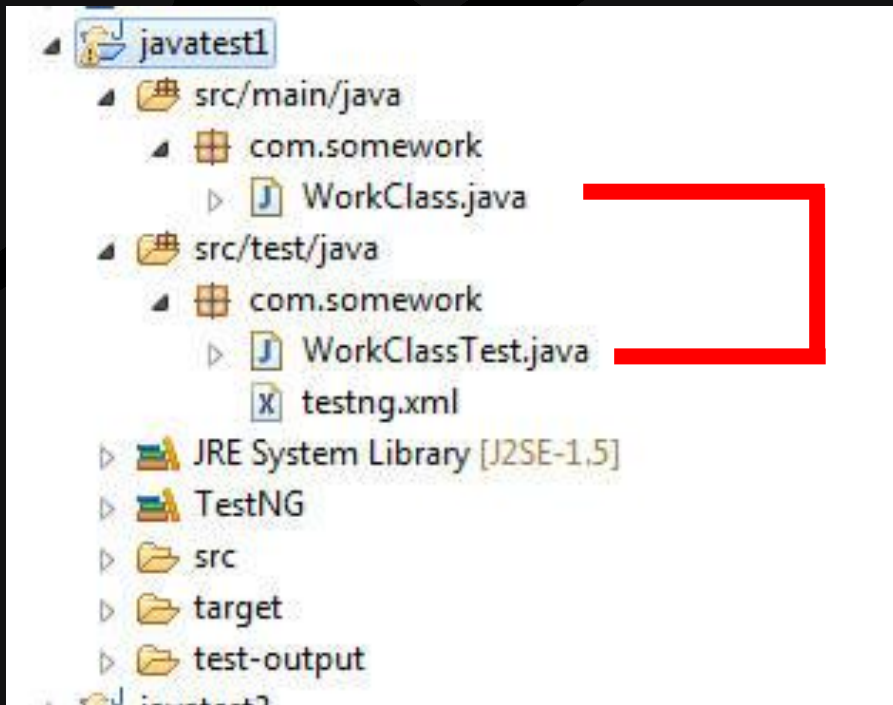
개발자들은 왜 Unit Test 를 안만들까?

테스트를 만들 줄 모른다

나?



테스트 코드 위치



같은 패키지
구조를
유지한다

테스트 코드 예제

```
import org.junit.Test;  
import static org.junit.Assert.*;
```

```
public class JunitSampleTest{
```

```
    @Test
```

← Test Annotation

```
    public void testMethod() {
```

← Test Method

```
        int sum = 10 + 20;
```

```
        assertEquals(30, sum, 0);
```

← Test Code

```
    }
```

```
}
```

Class 이름 끝에는
Test를 붙여준다

코드에서 테스트 대상 추출하기

```
public class UpperCaseCounter {  
    public int getNumberOfUpperCaseCharactersInString(String str) {  
        if (str == null) {  
            return 0;  
        }  
  
        int numberOfUpperCaseChars = 0;  
        for (int index = 0; index < str.length(); index++) {  
            char charAt = str.charAt(index);  
            if (charAt >= 'A' && charAt <= 'Z') {  
                numberOfUpperCaseChars++;  
            }  
        }  
  
        return numberOfUpperCaseChars;  
    }  
}
```



만약
입력값이
null이라면
0을
리턴한다.

코드에서 테스트 대상 추출하기

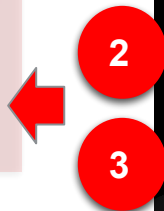
```
public class UpperCaseCounter {  
    public int getNumberOfUpperCaseCharactersInString(String str) {  
        if (str == null) {  
            return 0;  
        }  
  
        int numberOfUpperCaseChars = 0;  
        for (int index = 0; index < str.length(); index++) {  
            char charAt = str.charAt(index);  
            if (charAt >= 'A' && charAt <= 'Z') {  
                numberOfUpperCaseChars++;  
            }  
        }  
  
        return numberOfUpperCaseChars;  
    }  
}
```



첫글자부터
한글자씩
검사하면서
A~Z 글자
수를
세겠다

코드에서 테스트 대상 추출하기

```
public class UpperCaseCounter {  
    public int getNumberOfUpperCaseCharactersInString(String str) {  
        if (str == null) {  
            return 0;  
        }  
  
        int numberOfUpperCaseChars = 0;  
        for (int index = 0; index < str.length(); index++) {  
            char charAt = str.charAt(index);  
            if (charAt >= 'A' && charAt <= 'Z') {  
                numberOfUpperCaseChars++;  
            }  
        }  
  
        return numberOfUpperCaseChars;  
    }  
}
```



다른 글자들
(소문자, 숫자)
이
함께 포함되어
있다면?

코드에서 테스트 대상 추출하기

```
public class UpperCaseCounter {  
    public int getNumberOfUpperCaseCharactersInString(String str) {  
        if (str == null) {  
            return 0;  
        }  
  
        int numberOfUpperCaseChars = 0;  
        for (int index = 0; index < str.length(); index++) {  
            char charAt = str.charAt(index);  
            if (charAt >= 'A' && charAt <= 'Z') {  
                numberOfUpperCaseChars++;  
            }  
        }  
  
        return numberOfUpperCaseChars;  
    }  
}
```



2

3

4

str에
빈 값("")이
들어
있다면?

테스트 프레임워크를 몰랐을 때

```
public class UpperCaseCounter {  
    public int getNumberOfUpperCaseCharactersInString(String str) {  
        if (str == null) {  
            return 0;  
        }  
  
        int numberOfUpperCaseChars = 0;  
        for (int index = 0; index < str.length(); index++) {  
            char charAt = str.charAt(index);  
            if (charAt >= 'A' && charAt <= 'Z') {  
                numberOfUpperCaseChars++;  
            }  
        }  
  
        return numberOfUpperCaseChars;  
    }  
}
```

← else 로 분기해서 값을 출력해 본다

←

←

중간 중간에 값을 출력해
본다



테스트는 테스트 프레임워크를 사용한다

CUnit

JUnit

PyUnit

Java 언어를 위한 단위 테스트 프레임워크

JUnit Annotation

1. **@Test**

가장 기본이 되는 annotation으로 테스트하고자 하는 method에 선언한다

2. **@BeforeClass, @AfterClass**

해당 테스트 class가 실행전과 후에 동작할 class를 선언한다, static 선언 필요

3. **@Before, @After**

class내 test method들이 실행되기 전과 후에 동작할 method를 선언한다

4. **@FixMethodOrder**

테스트가 실행되는 순서를 정의한다

5. **@SuiteClasses(Class[])**

테스트 클래스들을 묶어서 실행할 때 사용한다

```
import org.junit.*;
public class Junit4Test {
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        System.out.println("테스트 수행 전 한번만 실행 됨");
    }

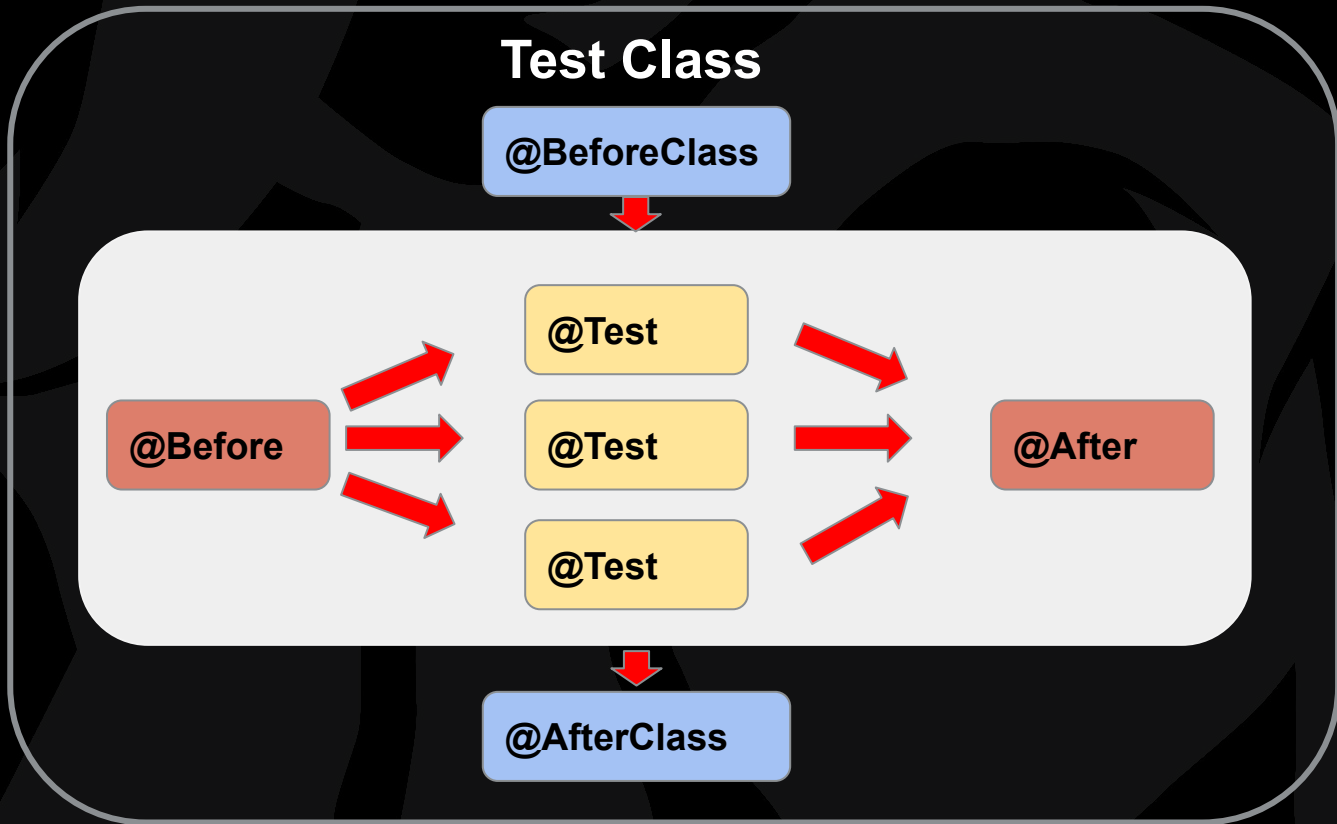
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
        System.out.println("테스트 수행 후 한번만 실행 됨");
    }

    @Before
    public void setUp() throws Exception {
        System.out.println("테스트 케이스를 실행할 때마다 먼저 반복 실행 됨");
    }

    @After
    public void tearDown() throws Exception {
        System.out.println("테스트 케이스를 실행할 때마다 수행 후 반복 실행 됨");
    }

    @Test
    public void testCase1() throws Exception {
        System.out.println("테스트 메소드를 지정함");
    }
}
```

@BeforeClass, @AfterClass, @Before, @After Flow



JUnit Annotation - @Test

가장 기본이 되는 annotation으로 테스트하고자 하는 method에 선언한다

@Test

```
public void getNumberOfUpperCaseCharactersInString_return_0_for_null_input() {
```

```
    String str = null;
```

```
    int numberOfUpperCaseCharactersInString = upperCaseCounter.getNumberOfUpperCaseCharactersInString(str);
```

```
    assertThat(numberOfUpperCaseCharactersInString, is(0));
```

```
}
```


JUnit Annotation - @Ignore

테스트를 진행하지 않을 때 사용한다

@Ignore

@Test

```
public void IgnoreTest() {
```

```
    System.out.println("이 테스트는 무시합니다");
```

```
}
```

JUnit - Exception 처리

기대(예상)하는 Exception이 발생하는지를 테스트 한다

```
@Test(expected = IndexOutOfBoundsException.class)
```

```
public void testIsEmptyGetIndexOutOfBoundsException() {
```

```
    new ArrayList<Object>().get(0);
```

```
}
```

JUnit - timeout

주어진 시간내에 정상적으로 실행이 되는 지를 테스트 한다

```
@Test(timeout = 5000)
```

```
public void timeinMethodTest() throws InterruptedException {
```

```
    Thread.sleep(3000);
```

```
}
```

```
@Test(timeout = 5000)
```

```
public void timeinMethodTest() throws InterruptedException {
```

```
    Thread.sleep(6000);
```

```
}
```

JUnit 단정문 - assert*

JUnit에서 사용되는 단정문과 hamcrest 단정문이 있음.
JUnit 4.x부터 hamcrest 가 포함되어 바로 사용할 수 있음.

기본적인 사용 예제

```
public class JunitSampleTest{  
    @Test  
    public void testMethod() {  
        org.junit.Assert.assertTrue( new ArrayList().isEmpty() );  
    }  
}
```

JUnit 단정문 - assert*

Method	내 용
<code>assertTrue(a)</code>	a의 조건이 참이면 Pass
<code>assertFalse(a)</code>	a의 조건이 거짓이면 Pass
<code>assertEquals(a, b)</code>	a와 b의 값이 같으면 Pass
<code>assertSame(a, b)</code>	a 객체와 b 객체가 같으면 Pass
<code>assertNotNull(a)</code>	a 객체가 Null이 아니면 Pass
<code>assertArrayEquals(a, b)</code>	a 배열과 b 배열이 동일하면 Pass

Hamcrest - assertThat

Matcher 라이브러리 중 하나로 테스트 표현식 작성시 좀더 자연스러운 문맥과 문장을 만들 수 있도록 지원한다.
가장 큰 장점은 오류 메시지를 읽기가 쉽다.

```
public class JunitHamcrestTest{  
    @Test  
    public void testMethod() {  
        assertThat(myList, is(empty()));  
    }  
}
```

JUnit과 Hamcrest 비교

```
public class JunitHamcrestSampleTest{  
    @Test  
    public void testCorrectChampionName() {  
        assertEquals("Darius", champion.getName());  
        assertThat(champion.getName(), is("Darius"));  
    }  
}
```

Hamcrest

allOf - 조건에 사용된 matcher가 모두 통과되는지에 대한 테스트

anyOf - 조건에 사용된 matcher 중 하나만이라도 통과되는지에 대한 테스트

not - 조건에 사용된 matcher 의 결과와 맞지 않는지에 대한 테스트

equalTo - 예상값과 matcher 결과 이후 동일 할 경우 성공

is - equalTo와 동일하나 가독성 증진을 위해 사용

hasToString - 문자열이 포함되어 있는지에 대한 테스트

instanceOf, isCompatibleType - 타입에 대한 테스트

notNullValue, nullValue - null값에 대한 테스트

sameInstance - 두 Object가 같은 지에 대한 테스트

Hamcrest

hasEntry, hasKey, hasValue - Map 요소에 대해 포함 여부를 테스트

hasItem, hasItems - collection 에 item 포함 여부를 테스트

hasItemInArray - 배열에 해당 item이 존재하는 지에 대한 테스트

closeTo - 부동소수점 값에 대한 허용 근사치내 해당하는지에 대한 테스트

greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo

- 수치값의 크기에 따른 비교조건에 적합한 지에 대한 테스트(>, >=, <, <=)

equalToIgnoringCase - 대소문자가 달라도 같은 것으로 판단하여 비교

equalToIgnoringWhiteSpace - 공백을 제거한 상태에서 두 값이 같은지 비교

containsString, endsWith, startsWith - 문자열의 일부, 시작, 끝나는 부분에 대한 비교

테스트 메소드명 이름

1. 테스트 메소드명은 간결한 것보단 명확하게 설명해 주는 것이 좋다.
길더라도 테스트의 목적을 명확히 설명하라. 의미없는 메소드명은 금물.
2. 예전에는 메소드명 끝이나 앞에 **Test**를 붙여야 했으나, 최근에는 사용하지 않는 경우도 많다. 취향에 맞춰 선택하면 된다.
3. 영어가 익숙하지 않아 이름 짓는데 투자하는 시간이 많다면 그냥 한글로 작성해도 무방하다. 멋은 좀 떨어지지만 -_-;
4. 정답은 없다. 하지만 가장 많이 사용되는 **Convention**을 따르는 것을 추천한다



Unit Test 를 위한 Naming Convention

많이 사용되는 7가지 Naming Convention

<https://dzone.com/articles/7-popular-unit-test-naming>

MethodName_StateUnderTest_ExpectedBehavior:

- isAdult_AgeLessThan18_False
- withdrawMoney_InvalidAccount_ExceptionThrown
- admitStudent_MissingMandatoryFields_FailToAdmit

test[Feature being tested]:

- testIsNotAnAdultIfAgeLessThan18
- testFailToWithdrawMoneyIfAccountIsInvalid
- testStudentIsNotAdmittedIfMandatoryFieldsAreMissing



그러면 직접 테스트 코드를 작성해 봅시다

1. 다음 Git 주소에서 소스를 다운받습니다.

<https://github.com/ryan-riot/unit-test-1week>

2. IntelliJ 에서 다운받은 소스를 불러 옵니다.
3. 다운 받은 소스를 바탕으로 테스트 코드를 작성합니다.

1주차 과제

1. 금일 테스트한 코드를 본인의 github repository에 저장하고 해당 url 을 저에게 공유해 주세요.
2. 아래 사이트를 통해서 복습하시면 좋을 것 같습니다.
<https://www.baeldung.com/java-junit-hamcrest-guide>
3. 금일 강의에 대한 피드백을 부탁드립니다(좋았던 점, 개선할 점)
4. 진행이 잘 안되는 부분들은 주저말고 슬랙에 문의 주세요.

2 주차

Mock? 가짜?

테스트는 독립적이어야 한다

Independent(독립적)

테스트는 깔끔함과 단정함을 유지해야 한다. 즉, 확실히 한 대강에 집중한 상태여야 하며, 환경과 다른 개발자들(명심하라. 다른 개발자들이 동시에 같은 테스트를 실행해 볼 수도 있다)에게서 독립적인 상태를 유지해야 한다.

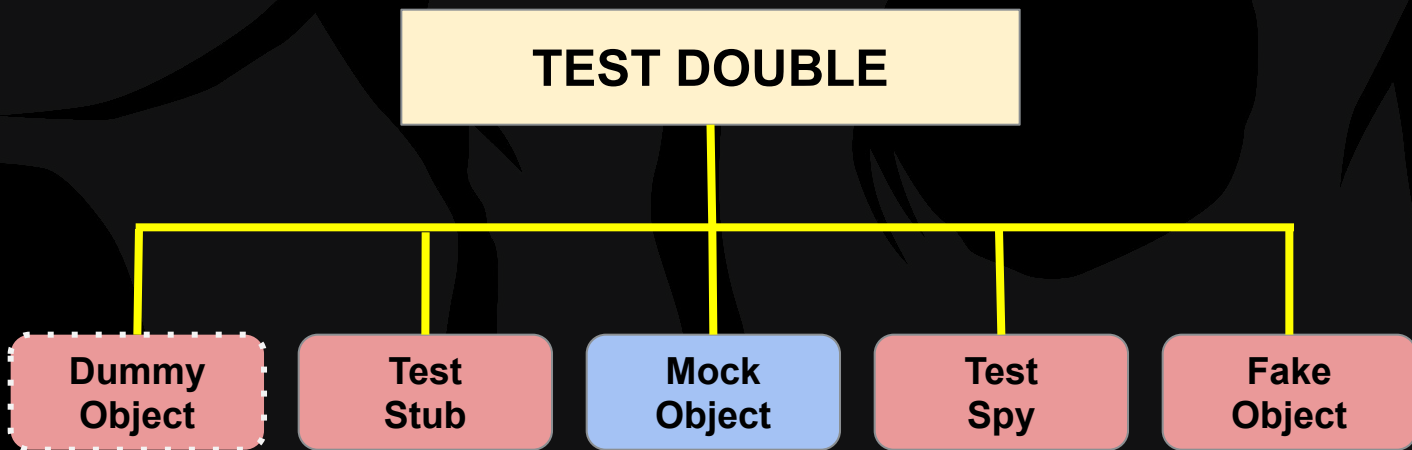
또한 독립적이라는 것은 어떤 테스트도 다른 테스트에 의존하지 않는다는 것을 의미한다. 어느 순서로든, 어떤 개별 테스트라도 실행해 볼 수 있어야 한다. 처음 것을 실행할 때 그 밖의 다른 테스트에 의존해야 하는 상황을 원하지는 않을 것이다.

모든 테스트는 섬이어야 한다.

from 실용주의 프로그래머를 위한 단위 테스트 with JUnit

Test Double 이란?

대역 배우(Stunt double)를 일컫는 단어에서 차용한 것으로 테스트의 의존성을 독립시킬 수 있도록 지원하는 대역 객체들을 지칭.



Test Double 사용시 장점

- 테스트 대상 코드 격리
- 테스트 속도 개선
- 예측 불가능한 실행 요소 제거
- 특수한 상황 테스트 가능
- 감춰진 정보를 확인 가능

Mock Test 란?

Mock? 가짜?

Mock Test 란?

**Mock 객체를 만들어
테스트를 진행**

Mock 을 왜 사용할까요?

1. 테스트 코드가 다른 코드와의 의존 관계에 있어 오류가 나거나 테스트 진행이 안되는 경우
2. 네트워크나 데이터베이스 등 외부적인 요인에 영향을 받아 정상적으로 테스트 진행이 어려운 경우

Mockito

자바에서 단위테스트를 하기 위해
Mock을 만들어주는 프레임워크

pom.xml dependency 추가

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <exclusions>
      <exclusion>
        <groupId>org.hamcrest</groupId>
        <artifactId>hamcrest-core</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-all</artifactId>
    <version>1.3</version>
  </dependency>
```

```
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.10</version>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.10.19</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Test Runner 지정 - @RunWith

테스트 환경의 기능 확장이나 초기화를 위해 사용된다. Spring에서 좀더 유연한 테스트를 위해서는 `@RunWith(SpringJUnit4ClassRunner.class)` 를 사용한다.

```
@RunWith(MockitoJUnitRunner.class)
```

```
public class studentMockTest(){
```

```
....
```

```
public void test강사능력평가(){
```

```
    assertThat(지금강사님은강의를잘하고있다, is(그렇리없다));
```

```
}
```

```
}
```

Mock 객체 만들기 - mock()

@Test

```
public void studentMockTest(){  
    Student student = mock(Student.class);  
    assertTrue( student != null );  
}
```


Mock Annotation - @Mock

@Mock

```
Student student;
```

@Test

```
public void studentMockTest(){  
    MockitoAnnotations.initMocks(this);  
    assertTrue( student != null );  
}
```

Mock Annotation - when(), then()

해당 Mock 객체가 호출되거나 사용될 때 원하는 값을 리턴하여 테스트를
진행

@Test

```
public void testStudentMockWhenThen(){  
    Student student = mock(Student.class);  
    when(student.getName()).thenReturn("카이사");  
    assertTrue("카이사".equals(student.getName()));  
}
```

Mock 예외 던지기 - doThrow()

```
@Test(expected = IllegalArgumentException.class)
public void testStudentDoThrow(){
    Student student = mock(Student.class);
    doThrow(new IllegalArgumentException()).when(p).setName(eq("홍길동"));
    p.setName(name);
}
```

Mock 아무것도 안하기 - doNothing()

@Mock

StudentService studentService;

@Test

public void testDoNothing(){

doNothing().when(studentService).getName(any());

studentService.getName("르블랑");

}

Mock 호출 검증 - verify()

해당 구문이 호출되었는 지를 검증

@Test

```
public void testStudentMockVerify(){  
    Student student = mock(Student.class);  
    student.setName("이즈리얼");  
    verify(student).setName("이즈리얼");  
}
```

Mock 호출 횟수 검증 - verify() with number of

`@Test`

```
public void testStudentMockVerifyWithTimes(){
```

```
.....  
verify(student, times(1)).size(); //일반 verify와 같습니다. 1번 실행되었는지 확인
```

```
verify(student, atLeastOnce()).size(); // 최소 한 번 이상만 실행되면 OK
```

```
verify(student, atMost(2)).size(); // 최소 두번은 실행되어야 함
```

```
verify(student, atLeast(1)).size(); // 최소한 번은 이하로 실행되면 OK
```

```
verify(student, never()).clear(); // 절대 호출되면 안됨
```

```
verify(student, timeout(100).atLeast(1)).setName(any(String.class));
```

```
//100ms 시간 내에 한 번이상 실행되면 OK, setName에 오는 string은 아무거나 OK
```

```
}
```

테스트 대상이 다른 객체를 포함할 때 - @InjectMocks

```
public class StudentService {  
    private Student student;  
    public void addStudent(){  
        ....  
    }  
}
```

```
@Mock  
Student student  
@InjectMocks  
StudentService studentService  
@Test  
public void testStudentServiceInjectMock(){  
    Student student = studentService.getStudent("애쉬");  
    assertTrue(student.getPosition() == "바텀");  
}
```

Given, When, Then Pattern

BDD 기반으로 테스트하는 방법을 의미하며, 다음 예제와 같은 형식을 갖는다.

@Test

```
public void testPhoneBookInsertWhenIsEmpty(){  
    given(phoneBookRepository.contains(momContactName))  
        .willReturn(false);  
    phoneBookService.register(momContactName, momPhoneNumber);  
    then(phoneBookRepository)  
        .should()  
        .insert(momContactName, momPhoneNumber);  
}
```


Mock 실습

1. 다음 Git 주소에서 소스를 다운받습니다.

<https://github.com/ryan-riot/unit-test-2week>

2. IntelliJ 에서 다운받은 소스를 불러 옵니다.
3. 다운 받은 소스를 바탕으로 테스트 코드를 작성합니다.

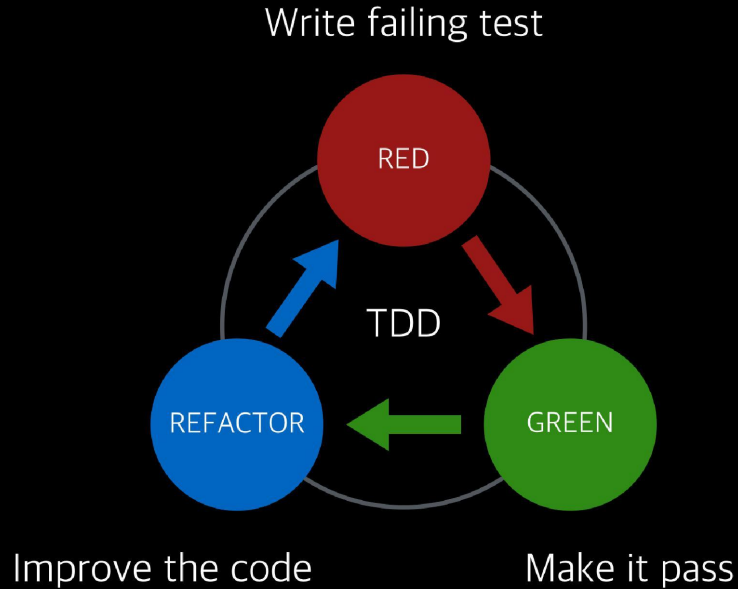
TDD - Test Driven Development

TDD 매우 작은 개발 세션의 반복
로 시작하는 방식이고,
테스트를 먼저 만들고 테스트

**ALL CODE IS GUILTY
UNTIL PROVEN INNOCENT**

TDD - Red, Green, Blue Circle

THE THREE LAWS OF TDD



TDD - 장점

1. 깨끗한 코드
2. 유지보수 비용 절감
3. 결함 감소

TDD - 단점

1. 개발 시간의 증가
2. 테스트 코드 작성이 어려운 경우 배보다 배꼽이 더 크다
3. 기존 개발 방법에 익숙한 경우 적응이 어렵다

TDD 시연 데모

간단한 은행 계좌 관리

<https://github.com/ryan-riot/unit-test-tdddemo>

소프트웨어 테스트 과제

1. 조별로 간단한 **Java** 프로그램을 작성합니다. 주제는 자유롭게 선정하세요.
2. 주제를 선정할 때에는 각 테스트코드를 사용하기 적합한 것이면 좋습니다.
3. 해당 프로그램을 테스트하기 위한 테스트 코드를 작성합니다.
4. 코드는 한 사람이 아닌 팀원이 함께 파트를 잘 구분해서 진행합니다.
팀원이 함께 작업한 코드를 **Github**에 올리고 해당 링크를 제출하세요.