

# Java/Spring 테스트를 추가하고 싶은 개발자들의 오답노트

잘못된 테스트 사례 학습

마무리와 부록

테스트에 대한 팁들

# INDEX

01

범위

테스트는 어디까지 해야하는가?  
장단점

02

테스트 팁

ParameterizedTest  
assert  
메소드 체인  
Thread.sleep

03

문화

트로이 목마

# 01. 범위

○ 테스트는 어디까지 해야하는가?

## ○ 테스트는 어디까지 해야하는가?

“ 나는 얼마나 마음 편하게 소프트웨어를 배포할 수 있느냐를 테스트의 성공 기준으로 삼으면 된다고 생각한다. 테스트를 실행한 후에 소프트웨어를 배포해도 될 만큼 테스트를 신뢰한다면 그것으로 된 것이다.

톰 홈버그, 만들면서 배우는 클린 아키텍처 자바 코드로 구현하는 클린 웹 애플리케이션, 박소은 옮김 조영호 감수, (위키북스, 2022-03-30), 95p

## 범위

---

### ○ 테스트는 어디까지 해야하는가?

“ 커버리지가 중요한게 아닙니다. 릴리즈 할 때 확신을 주면 됩니다.

# 범위

## ○ 테스트는 무엇을 해야하는가?

### Right-BICEP

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)



## ○ 테스트는 무엇을 해야하는가?

### Right-BICEP

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

“ Right - 결과가 올바른가?

다른 관점으로, 어떤 작은 부분의 코드에 대한 행복 경로 테스트를 할 수 없다면 그 내용을 완전히 이해하지 못한 것입니다. 그리고 앞의 질문에 대답할 수 있을 때까지 잠시 추가 개발은 보류하면 좋습니다.

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

## ○ 테스트는 무엇을 해야하는가?

### Right-BICEP

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

“ Boundary conditions - 경계 조건은 맞는가?

코드에 있는 분명한 행복 경로는 입력 값의 양극단을 다루는 코드 시나리오의 경계 조건에 걸리지 않을 수도 있습니다.

여러분이 마주치는 수많은 결함은 이러한 모서리 사례(coner case)이므로 테스트로 이것들을 처리해야합니다.

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

# 범위

## ○ 테스트는 무엇을 해야하는가?

### Right-BICEP

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

“ Boundary conditions - 경계 조건은 맞는가?

코드에 있는 분명한 행복 경로는 입력 값의 양극단을 다루는 코드 시나리오의 경계 조건에 걸리지 않을 수도 있습니다.

여러분이 마주치는 수많은 결함은 이러한 모서리 사례(corder case)이므로 테스트로 이것들을 처리해야합니다.

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

Edge case      시스템 내부 조건에 의해 발생하는 특별한 케이스를 의미 (ex. Long의 Max 값은 언어마다 다르다.)

Corder case      시스템 내/외부 조건에 의해 발생하는 특별한 케이스들을 의미 (ex. 네트워크가 단절된 경우, 서버의 메모리가 128MB인 경우)

# 범위

## ○ 테스트는 무엇을 해야하는가?

### Right-BICEP

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

“ Boundary conditions - 경계 조건은 맞는가?

#### Edge case 위주

C <sub>onformance</sub>	데이터 형식이 제대로 됬는지 검사해야 합니다
O <sub>rdering</sub>	데이터 순서가 맞는지 검사해야 합니다.
R <sub>ange</sub>	최댓값, 최솟값일 때에도 시스템이 의도한 대로 동작하는지 테스트해야 합니다
R <sub>eference</sub>	협력 객체의 상태가 어떤지에 따라, 시스템이 의도한 대로 동작을 해야하는지 테스트해야 합니다
E <sub>xistence</sub>	값이 존재할 때와 안할 때를 테스트합니다
C <sub>ardinality</sub>	값이 충분히 존재할 때와 존재하지 않을 때를 테스트합니다
T <sub>ime</sub>	시간 순서가 제대로 지켜지는지를 테스트합니다

# 범위

## ○ 테스트는 무엇을 해야하는가?

### Right-BICEP

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

“ Boundary conditions - 경계 조건은 맞는가?

#### Edge case 위주

Conformance 사용자의 Id가 이메일 형식인지 테스트합니다

Ordering 당첨자가 청약 점수에 따라 정렬이 됬는지 테스트합니다

Range 덧셈 함수가 Long.Max + Long.Max일 때 어떻게 동작해야 하는지 테스트합니다

Reference 휴면 계정 처리된 사용자를 초대하기 했을 때 어떻게 동작해야 하는지 테스트합니다

Existence 입력값 null, 0, "" 일 때 어떻게 동작하는지 테스트합니다

Cardinality 입력값의 길이가 0일 때, 1일 때, n일 때 어떻게 동작하는지 테스트합니다

Time 병렬 처리하지만 메시지 순서는 제대로 처리되는지 테스트합니다

# 범위

## ○ 테스트는 무엇을 해야하는가?

### Right-BICEP

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

“ Inversion relationship - 역 관계를 검사할 수 있는가?



역 관계를 검증할 수 있다면 검증하라

```
@Test
void 제곱근의_결과를_제곱하면_원래_값에_근사해야_한다() {
    // given + when
    double root2 = Math.sqrt(2);

    // then
    Percentage percentage = Percentage.withPercentage(0.9999999);
    assertThat(actual: root2 * root2).isCloseTo(expected: 2, percentage);
}
```

## 범위

---

### ○ 테스트는 무엇을 해야하는가?

#### Right-BICEP

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

“ Cross check - 교차 검사할 수 있는가?



내가 구현한 유사한 라이브러리가 있다면, 구현한 값과 라이브러리 값을 비교하여 검증합니다.

## ○ 테스트는 무엇을 해야하는가?

### Right-BICEP

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

“ Error conditions - 오류 상황을 강제로 일어나게 할 수 있는가?



오류 상황을 강제로 발생시키고 시스템이 어떻게 동작하는지를 테스트해야 합니다.

- 메모리가 가득 찰 때
- 디스크 공간이 가득 찰 때
- 서버와 클라이언트 간 시간이 다를 때
- 네트워크 가용성 및 오류들
- 시스템 로드
- 제한된 색상 팔레트
- 매우 높거나 낮은 비디오 해상도

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

## 범위

---

### ○ 테스트는 무엇을 해야하는가?

#### Right-BICEP

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

“ Performance characteristics - 성능 조건은 기준에 부합하는가?

시스템 성능이 제대로 요구에 부합하는지를 테스트합니다.

일반적으로 e2e(대형) 테스트에 사용되기 때문에 단위 테스트의 범위와는 약간 거리감이 있습니다.

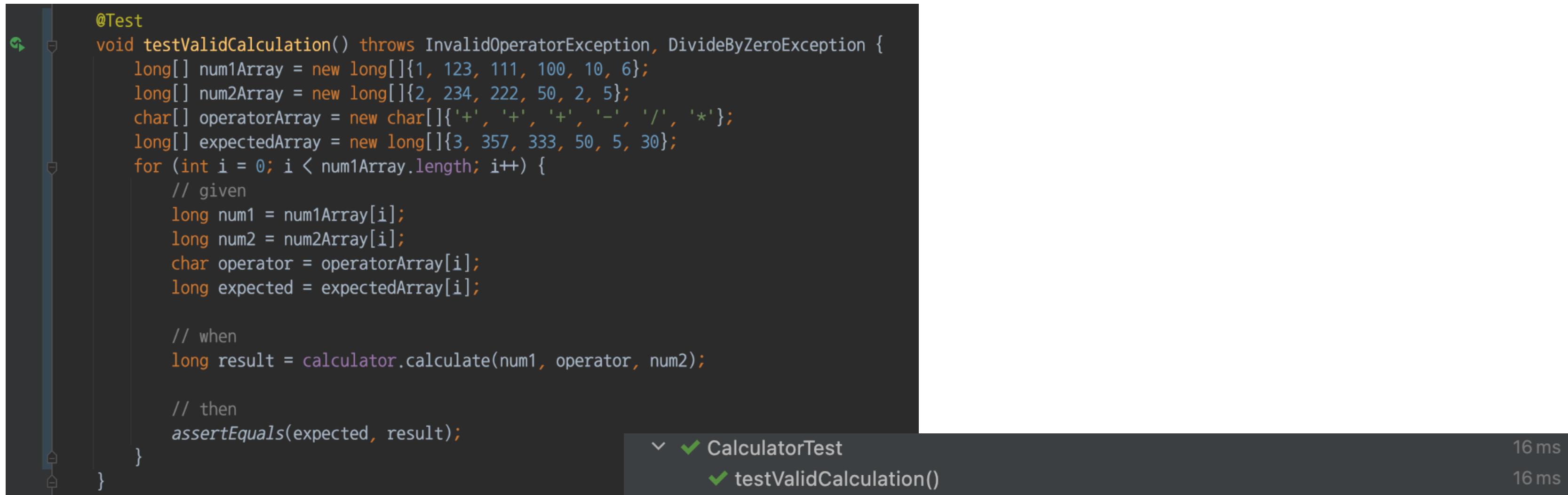
## 02. 테스트 팁

- ParameterizedTest
- assertAll
- 하나의 테스트
- assert는 하나만?
- Thread.sleep
- FIRST 원칙

# 테스트 팁

## ○ ParameterizedTest

가끔 동일 로직을 여러번 돌리고 싶습니다.



```
@Test
void testValidCalculation() throws InvalidOperatorException, DivideByZeroException {
    long[] num1Array = new long[]{1, 123, 111, 100, 10, 6};
    long[] num2Array = new long[]{2, 234, 222, 50, 2, 5};
    char[] operatorArray = new char[]{'+', '+', '+', '-', '/', '*'};
    long[] expectedArray = new long[]{3, 357, 333, 50, 5, 30};
    for (int i = 0; i < num1Array.length; i++) {
        // given
        long num1 = num1Array[i];
        long num2 = num2Array[i];
        char operator = operatorArray[i];
        long expected = expectedArray[i];

        // when
        long result = calculator.calculate(num1, operator, num2);

        // then
        assertEquals(expected, result);
    }
}
```

▼ ✓ CalculatorTest  
✓ testValidCalculation()  
16 ms  
16 ms

# 테스트 팁

## ○ ParameterizedTest

실패하면 뭐 때문에 실패했는지도 눈에 잘 안들어옵니다.

The screenshot shows a code editor with Java test code. The code defines a test method `testValidCalculation()` that performs calculations on arrays of numbers using various operators. It uses assertions to check if the calculated result matches the expected result. The test fails at the last iteration of the loop. Below the code, a test runner interface shows the failure details:

▼ ✘ CalculatorTest	13 ms
└ ✘ testValidCalculation()	13 ms

# 테스트 팁

## ○ ParameterizedTest

반복적인 테스트를 논리 로직 없이 돌릴 수 있는 방법

The screenshot shows a Java code editor with a test class and its execution results.

```
    @ParameterizedTest
    @MethodSource("provideValidInputs")
    void testValidCalculation(long num1, char operator, long num2, long expected) throws Inv
        // when
        long result = calculator.calculate(num1, operator, num2);

        // then
        assertEquals(result, expected);
    }

    private static Stream<Arguments> provideValidInputs() {
        return Stream.of(
            Arguments.of(1, '+', 2, 3),
            Arguments.of(123, '+', 234, 357),
            Arguments.of(111, '+', 222, 333),
            Arguments.of(100, '-', 50, 50),
            Arguments.of(10, '/', 2, 5),
            Arguments.of(6, '*', 5, 30)
        );
    }
}
```

The execution results on the right show the test method running 6 times with different arguments:

Test Case	Time
[1] num1=1, operator=+, num2=2, expected=3	33 ms
[2] num1=123, operator=+, num2=234, expected=357	2 ms
[3] num1=111, operator=+, num2=222, expected=333	1 ms
[4] num1=100, operator=-, num2=50, expected=50	1 ms
[5] num1=10, operator=/, num2=2, expected=5	1 ms
[6] num1=6, operator=*, num2=5, expected=30	1 ms

# 테스트 팁

## ○ assertAll

중간에 실패해도 assert를 모두 돌릴 수 있는 방법

The screenshot shows a Java code editor and a terminal window. The code editor displays a test method named `void 사용자_정보를_save로_저장_할_수_있다()`. The test setup includes creating a `UserEntity` object and setting its status to `PENDING`. The test then saves the entity and asserts that its ID is not null, status is `ACTIVE`, certification code is "hash1234567891", and address is "Seoul". The terminal window shows the test results for the `UserRepositoryTest` class. The test case `사용자_정보를_save로_저장_할_수_있다()` failed with an `AssertionFailedError`. The error message indicates that the expected status was `ACTIVE` but the actual status was `PENDING`. The terminal also shows the expected and actual values for the status field.

```
@Test
void 사용자_정보를_save로_저장_할_수_있다() {
    // given
    UserEntity userEntity = new UserEntity();
    userEntity.setEmail("kok202@naver.com");
    userEntity.setStatus(UserStatus.PENDING);
    userEntity.setCertificationCode("hash1234567890");
    userEntity.setAddress("Seoul");

    // when
    userEntity = userRepository.save(userEntity);

    // then
    assertThat(userEntity.getId()).isNotNull();
    assertThat(userEntity.getStatus()).isEqualTo(UserStatus.ACTIVE);
    assertThat(userEntity.getCertificationCode()).isEqualTo("hash1234567891");
    assertThat(userEntity.getAddress()).isEqualTo("Seoul");
}
```

Test Results

- UserRepositoryTest
- 사용자\_정보를\_save로\_저장\_할\_수\_있다()

238 ms org.opentest4j.AssertionFailedError:  
expected: ACTIVE  
but was: PENDING  
Expected :ACTIVE  
Actual :PENDING  
<Click to see difference>

# 테스트 팁

## ○ assertAll

중간에 실패해도 assert를 모두 돌릴 수 있는 방법

The screenshot shows a Java code editor and a terminal window. The code editor displays a test method named `void 사용자_정보를_save로_저장_할_수_있다()`. The test setup includes creating a `UserEntity` object and setting its status to `PENDING`. The test assertion part contains four `assertThat` statements. Two of these assertions have their expected values circled in red: the first one expects the status to be `ACTIVE`, and the fourth one expects the address to be `"Seoul"`. The terminal window to the right shows the test results. It indicates that the test failed with an `AssertionFailedError`. The error message states that the expected value was `ACTIVE` but the actual value was `PENDING`. It also shows the expected and actual values for the other assertions.

```
@Test
void 사용자_정보를_save로_저장_할_수_있다() {
    // given
    UserEntity userEntity = new UserEntity();
    userEntity.setEmail("kok202@naver.com");
    userEntity.setStatus(UserStatus.PENDING);
    userEntity.setCertificationCode("hash1234567890"); // Circled in red
    userEntity.setAddress("Seoul");

    // when
    userEntity = userRepository.save(userEntity);

    // then
    assertThat(userEntity.getId()).isNotNull();
    assertThat(userEntity.getStatus()).isEqualTo(UserStatus.ACTIVE); // Circled in red
    assertThat(userEntity.getCertificationCode()).isEqualTo("hash1234567891"); // Circled in red
    assertThat(userEntity.getAddress()).isEqualTo("Seoul");
}
```

Test Results

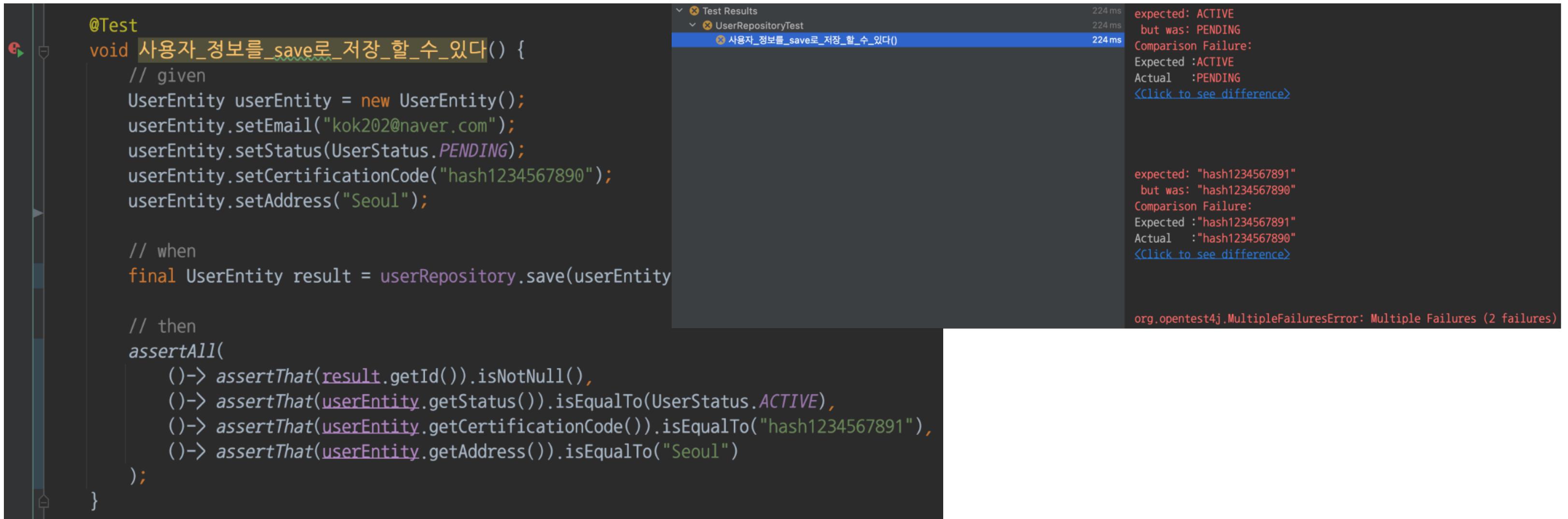
- userRepositoryTest
- 사용자\_정보를\_save로\_저장\_할\_수\_있다() 238ms

```
org.opentest4j.AssertionFailedError:  
expected: ACTIVE  
but was: PENDING  
Expected :ACTIVE  
Actual   :PENDING  
<Click to see difference>
```

# 테스트 팁

## ○ assertAll

어느 지점이 실패했는지 정확히 짚어줍니다.



The screenshot shows an IDE interface with a Java test class and its execution results.

```

    @Test
    void 사용자_정보를_save로_저장_할_수_있다() {
        // given
        UserEntity userEntity = new UserEntity();
        userEntity.setEmail("kok202@naver.com");
        userEntity.setStatus(UserStatus.PENDING);
        userEntity.setCertificationCode("hash1234567890");
        userEntity.setAddress("Seoul");

        // when
        final UserEntity result = userRepository.save(userEntity)

        // then
        assertAll(
            ()-> assertThat(result.getId()).isNotNull(),
            ()-> assertThat(userEntity.getStatus()).isEqualTo(UserStatus.ACTIVE),
            ()-> assertThat(userEntity.getCertificationCode()).isEqualTo("hash1234567891"),
            ()-> assertThat(userEntity.getAddress()).isEqualTo("Seoul")
        );
    }
  
```

The test method is annotated with `@Test`. It creates a `UserEntity`, sets its email to "kok202@naver.com", status to `PENDING`, certification code to "hash1234567890", and address to "Seoul". It then saves the entity using `userRepository.save` and performs assertions on the returned `result`.

The test results window shows two failures:

- Assertion Failure:** `userEntity.getStatus()` expected `ACTIVE` but was `PENDING`. Click to see difference.
- Assertion Failure:** `userEntity.getCertificationCode()` expected "hash1234567891" but was "hash1234567890". Click to see difference.

At the bottom right, the error message is: `org.opentest4j.MultipleFailuresError: Multiple Failures (2 failures)`.

# 테스트 팁

## ○ 하나의 테스트

한 개의 테스트는 한 개만 테스트해야 합니다

```
@Test
void testValidData() throws InvalidOperatorException, DivideByZeroException {
    // when
    long result = calculator.calculate( num1: 1, operator: '+', num2: 2);

    // then
    assertThat(result).isEqualTo(3);

    // when
    result = calculator.calculate( num1: 4, operator: '*', num2: 2);

    // then
    assertThat(result).isEqualTo(8);
}
```

# 테스트 팁

## ○ 하나의 테스트

그렇지 않으면 여러분의 후임자들이 계속해서 이 테스트에 테스트 케이스를 추가할 겁니다.

```
@Test
void testValidData() throws InvalidOperatorException, DivideByZeroException {
    // when
    long result = calculator.calculate( num1: 1, operator: '+', num2: 2);

    // then
    assertThat(result).isEqualTo(3);

    // when
    result = calculator.calculate( num1: 4, operator: '*', num2: 2);

    // then
    assertThat(result).isEqualTo(8);

    // when
    result = calculator.calculate( num1: 4, operator: '-', num2: 2);

    // then
    assertThat(result).isEqualTo(2);

    // when
    result = calculator.calculate( num1: 12, operator: '/', num2: 2);

    // then
    assertThat(result).isEqualTo(6);
```

# 테스트 팁

## ○ assert는 하나만 해야한다?

이런 이야기를 듣다보면, 한 개의 테스트는 한 개의 assert만을 가져야 한다고 오해되기도 함.



```
@Test
void 사용자_정보를_save로_저장_할_수_있다() {
    // given
    UserEntity userEntity = new UserEntity();
    userEntity.setEmail("kok202@naver.com");
    userEntity.setStatus(UserStatus.PENDING);
    userEntity.setCertificationCode("hash1234567890");
    userEntity.setAddress("Seoul");

    // when
    final UserEntity result = userRepository.save(userEntity);

    // then
    assertAll(
        ()-> assertThat(result.getId()).isNotNull(),
        ()-> assertThat(userEntity.getStatus()).isEqualTo(UserStatus.ACTIVE),
        ()-> assertThat(userEntity.getCertificationCode()).isEqualTo("hash1234567891"),
        ()-> assertThat(userEntity.getAddress()).isEqualTo("Seoul")
    );
}
```

## 테스트 팁

---

### ○ assert는 하나만 해야한다?

“ 몇몇 TDD 실천가들은 각 테스트에 예상 구문이나 단정을 하나만 담아야한다고 제안하기도 한다. (중략)  
우리는 그 방법이 실용적이지 않다는 사실을 알게 됐다. (중략)

**다시 한번 강조하지만 표현력이 핵심이다.**

테스트를 읽는 사람으로서 뭐가 중요한지 가늠할 수 있는가?

스티브 프리먼, 넷 프라이스 지음, 테스트 주도 개발로 배우는 객체 지향 설계와 실천, 이대엽 옮김, (인사이트(insight), 2019-12-26), 291p

## 테스트 팁

### ○ Thread.sleep

- “ 비동기 처리를 Thread sleep을 이용해서 테스트하면, 개발자 개별의 데스크탑 성능에 따라 테스트 결과가 달라질 수 있습니다.

```
@Getter
public class CountUpThread extends Thread {
    private final int goal;
    private int currentCount;

    public CountUpThread(int goal) {
        this.goal = goal;
    }

    @Override
    public void run() {
        for (currentCount = 1; currentCount < goal; currentCount++) {
            System.out.println(currentCount);
        }
    }
}
```

## 테스트 팁

### ○ Thread.sleep

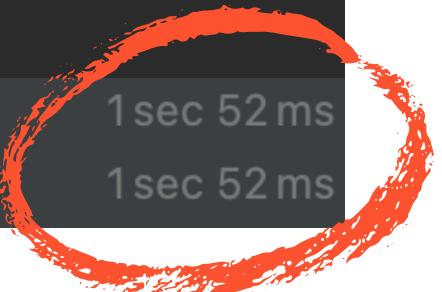
- “ 비동기 처리를 Thread sleep을 이용해서 테스트하면, 개발자 개별의 데스크탑 성능에 따라 테스트 결과가 달라질 수 있습니다.

```
@Test
void CountUpThread_는_목표치까지_카운트한다() throws InterruptedException {
    // given
    int goal = 100000;
    CountUpThread countUpThread = new CountUpThread(goal);

    // when
    countUpThread.start();
    Thread.sleep( millis: 1000 );

    // then
    assertThat(countUpThread.getCurrentCount()).isEqualTo(goal);
}
```

✓ CountUpThreadTest  
✓ CountUpThread\_는\_목표치까지\_카운트한다()



# 테스트 팁

## ○ Thread.sleep

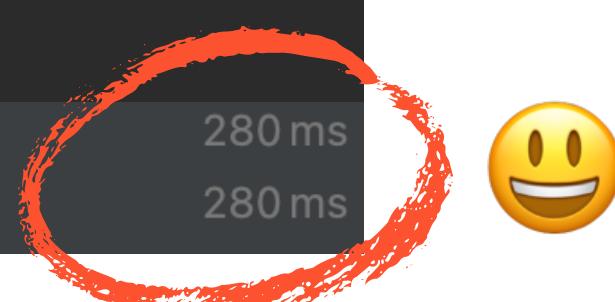
“ 간단한 해결책

```
@Test
void CountUpThread_는_목표치까지_카운트한다() throws InterruptedException {
    // given
    int goal = 100000;
    CountUpThread countUpThread = new CountUpThread(goal);

    // when
    countUpThread.start();
    countUpThread.join();

    // then
    assertThat(countUpThread.getCurrentCount()).isEqualTo(goal);
}
```

✓ CountUpThreadTest  
✓ CountUpThread\_는\_목표치까지\_카운트한다()



## 테스트 팁

### ○ Thread.sleep의 대안

“ 조금 더 보편적인 해결책 - Awaitility

```
@Test
void CountUpThread_는_목표치까지_카운트한다() throws InterruptedException {
    // given
    int goal = 100000;
    CountUpThread countUpThread = new CountUpThread(goal);

    // when
    countUpThread.start();
    await()
        .atMost( timeout: 10, TimeUnit.SECONDS )
        .until(countUpThread::getCurrentCount, equalTo(goal));

    // then
    assertThat(countUpThread.getCurrentCount()).isEqualTo(goal);
}

✓ CountUpThreadTest                               277 ms
✓ CountUpThread_는_목표치까지_카운트한다()          277 ms
```

## 테스트 팁

---

### ○ FIRST 원칙

ref. 제프 랭어, 앤디 헌트, 데이브 토마스 저, 자바와 JUnit을 활용한 실용주의 단위 테스트 클린 코드의 핵심, 단위 테스트로 소프트웨어 품질을 향상시킨다!, 유동환 역, (길벗, 2019-07-19)

- “ **F**ast 테스트는 빨라야한다
- I**ndependent 테스트는 독립적이어야 한다
- R**epeatable 테스트는 반복 수행해도 결과가 같아야한다
- S**elf-validating 테스트를 수행하면 시스템의 성공/실패 알 수 있어야 한다
- T**imely 테스트는 적시에(코드 구현 전) 작성되어야 한다

# 03. 문화

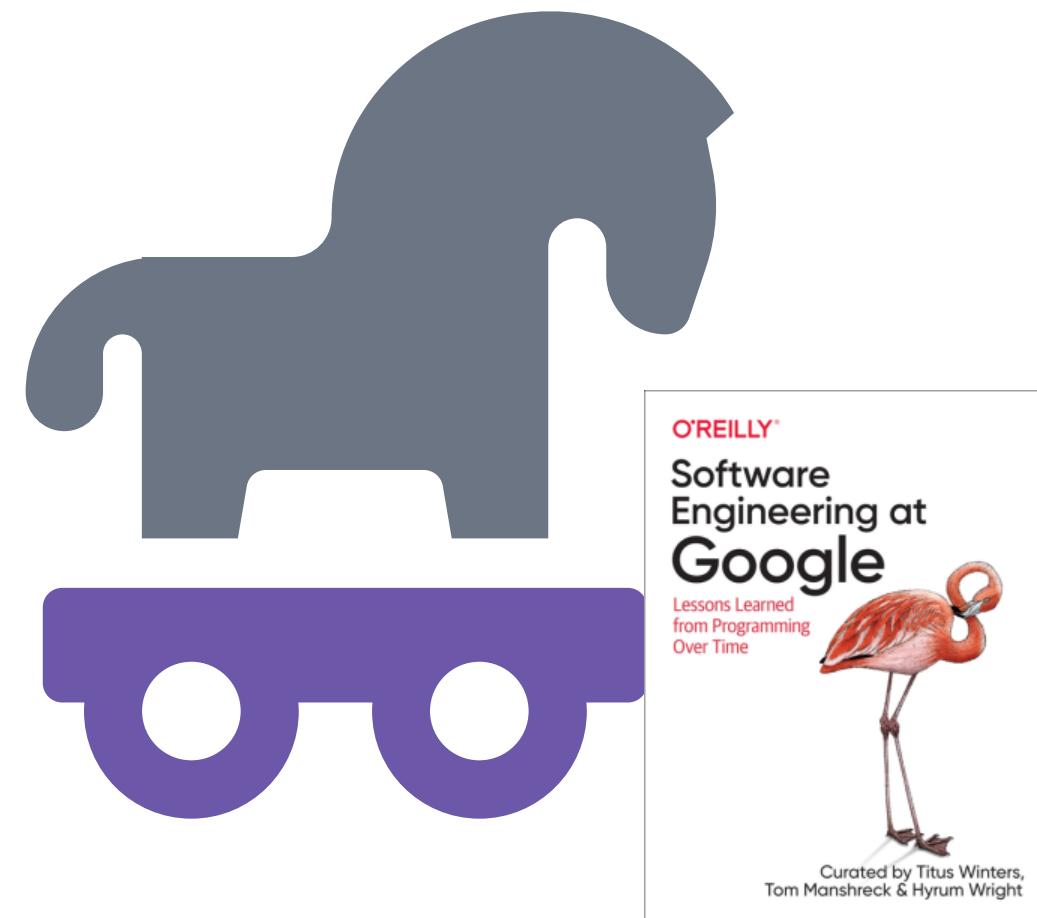
○ 트로이 목마

○ 마틴 파울러

## 트로이 목마

---

### ○ 테스트를 전파하는 트로이 목마



## 트로이 목마

---

### ○ 테스트를 전파하는 트로이 목마



## 트로이 목마

---

### ○ 테스트를 전파하는 트로이 목마



## 트로이 목마

---

### ○ 테스트를 전파하는 트로이 목마



## 마틴 파울러

---

### ○ 어록 1. 완전한 리팩터링

“ 리팩터링하기 전에 제대로 된 테스트부터 마련한다. 테스트는 반드시 자가진단하도록 만든다.

마틴 파울러 저, 리팩터링 2판 코드 구조를 체계적으로 개선하여 효율적인 리팩터링 구현하기, 개앞맵시, 남기혁 역, (한빛미디어), 2020-04-01), 28p

“ 누군가 "리팩터링하다가 코드가 깨져서 며칠이나 고생했다"라고 한다면, 십중팔구 리팩터링한 것이 아니다.

마틴 파울러 저, 리팩터링 2판 코드 구조를 체계적으로 개선하여 효율적인 리팩터링 구현하기, 개앞맵시, 남기혁 역, (한빛미디어), 2020-04-01), 80p

## 마틴 파울러

---

### ○ 어록 2. 테스트

“ 자주 테스트하라. 작성 중인 코드는 최소한 몇 분 간격으로 테스트하고, 적어도 하루에 한 번은 전체 테스트를 돌려보자.

마틴 파울러 저, 리팩터링 2판 코드 구조를 체계적으로 개선하여 효율적인 리팩터링 구현하기, 개앞맵시, 남기혁 역, (한빛미디어), 2020-04-01, 141p

⇒ 그러기 위해선 테스트를 돌리는데 부담이 없어야 하고 빨라야 합니다.

## 마틴 파울러

---

### ○ 어록 2. 좋은 코드

“ 좋은 코드를 가늠하는 확실한 방법은 "얼마나 수정하기 쉬운가"다.

마틴 파울러 저, 리팩터링 2판 코드 구조를 체계적으로 개선하여 효율적인 리팩터링 구현하기, 개앞맵시, 남기혁 역, (한빛미디어), 2020-04-01), 76p

## 마틴 파울러

---

“ Any fool can write code that a computer can understand.  
Good programmers write code that humans can understand.

컴퓨터가 이해할 수 있는  
코드는 누구나 작성할 수 있다.  
좋은 프로그래머는 사람이  
이해할 수 있는 코드를 작성한다.

## 테스트

---

### ○ 과거의 글

“ 그동안 개발을 해오면서 설계나 패턴에 대한 고민을 안 해본 것이 아닙니다. 남들 다 읽는 설계, 패턴 책들 다 읽어봤고 원리 원칙들도 남들보다는 더 많이 아는 편에 속한다 생각합니다. 그런데 테스트 코드를 짜면서 드는 생각이, 사실 알고 보니 그런 내용들 다 필요 없었구나 싶습니다.

그리고 좋은 설계는 "테스트 코드를 쉽게 짜도록 설계하는 법", "테스트할 수 있게 코드를 짜는 법"을 고민하다 보면 자연스럽게 따라온다는 생각이 듭니다. 조금 극단적인 생각이긴 합니다. 그럼에도 불구하고 이런 말을 하는 이유는 테스트 코드를 작성해서 얻은 설계와 아닌 설계를 비교해보니, 확실히 품질 차이가 있다는 것을 느꼈기 때문입니다.

다 떠나서 Testability와 좋은 설계는 강한 상관관계에 있는 것은 분명한 것 같습니다.

# RETURN;

아키텍처