Java 프로그램의 유닛 테스트 코드에서 발생하는 결함의 분류

A Classification of Unit Test Bugs in Java Programs

최한솔, 홍신
한동대학교 전산전자공학부
{hansolchoe, hongshin}@handong.edu

요 약

유닛 테스팅이 소프트웨어 품질 관리 방법론으로 널리 시행됨에 따라, 유닛 테스팅을 위한 코드에서 발생하는 테스트 결함이 프로젝트의 정확도와 생산성을 해치는 새로운 품질 관리 문제로 떠오르고 있다. 본 논문에서는 Java 프로그램에서 발생하는 유닛 테스트 결함을 체계적으로 기술/분류하는 새로운 결함 분류체계를 제안한다. 단편적인 결함 패턴을 제시하는 기존 테스트 결함 연구와는 달리, 본 연구에서는 유닛 테스트 코드의 다양한 구조적 요소와 기능적 요소의 범주를 제시하고, 이에 기반 한 총체적 분류 체계를 제안한다. 또한, 제안한 분류 체계를 활용해 최근에 보고된 유닛 테스트 결함 사례와 결함 패턴을 분류한 결과를 소개한다.

1. 서론

유닛 테스팅(unit test)이 소프트웨어 품질관리 방법으로 널리 시행됨에 따라, 유닛 테스팅을 위해 개발된 테스트 코드에서 발생하는 테스트 결함(test bug)이 오늘날 또 다른 품질관리 문제로 부각되고 있다[1][10]. 유닛 테스팅 수행을 위해서는 테스트 케이스(test case) 별 테스트 코드(test cod)의 작성이 필수적인데, 이는 테스트 코드를 통해 검증대상 모듈을 독립실행 가능한 형태로 재구성하고, 테스트 입출력을 실행 상태(program state)의 형태로 설정/검사하는 실행 과정이 필요하기 때문이다. 이러한 테스트 코드의 개발은 검증대상의 맥락과 관련 요구사항을 복합적으로 고려해야 하는 난이도가 높은 개발 작업이므로, 테스트 결함은 실제 소프트웨어 프로젝트에서 빈번히 발생하고 있다[1].

테스트 결함은 특정 모듈의 정확성 문제를 일으킬 뿐 아니라, 유닛 테스팅을 통한 품질관리 과정 전반의 정확성/생산성과 관련된 결정적인 문제를 야기할 수 있으므로 위험성이 크다. 반면 테스트 결함에 대한 이해와 이를 검출하기 위한 기술은 매우 제한적인 실정이다. 최근 조사 연구는 실제 프로젝트에서 다양한 원인과 형태를 가지는 테스트 결함이 발생하고 있음을 보고하는 반면, 기존 연구는 의도 하지 않은 테스트 실행 순서 종속성(Test-Order Dependency)이라는 특정한 종류의 결함에 집중되어 한정적인 상황이다[2][3][4]. 이러한 이유로, 현재로서는 개발자가 현장에서 당면하는 테스트 결함 문제를 체계적으로 설명하고 효과적으로 디버깅을 지원하기 어려운 상황이다.

본 연구는 Java 프로그램에서 발생하는 테스트 결함의 체계적인 이해와 분석을 지원하기 위해 결함 분류체계(classification)를 제안한다. 본 연구는 테스트 코드의 일반적인 요구사항, 구성형태, 실행순서를 모델링한 후, 이를 바탕으로 각 테스트 결함을 분석하고, 연관된 테스트 결함을 분류하는 방법을 제안한다. 본 연구에서는 제안한 결함 체계를 통해 SpotBugs [6], ErrorProne [7], PMD [8], Fb-contrib [9]에 존재하는 45개의 결함 검출기와 최근 연구결과[1]에서 보고된 21개의 결함사례를 분류하였으며, 그 범위를 상호간 비교하는 결과를 소개한다.

2. 유닛 테스트 코드의 구성

소프트웨어 결함 분석을 위해서는 결함과 관련된 잘못된 코드 요소의 위치/형태, 잘못된 코드 요소 실행에 따른 잘못된 동작 발생, 잘못된 동작 발생에 따른 요구사항(기능) 달성 실패로 이어지는 연쇄의 파악이 필요하다[5]. 본 장에서는 Java 프로그램의 유닛 테스트 코드에서 발견되는 코드 요소의 형태, 유닛 테스트 코드의 동작 과정, 유닛 테스트에 대한 일반적 요구사항을 여러 가지 경우로 나누어 설명 한다. 본 연구에서는, 본 장에서 소개하는 테스트 코드 요소를 기준으로 테스트 결함을 분석, 분류하는 방법을 제안한다.

2.1. 배경: 일반적인 유닛 테스트 코드의 구성과 기능

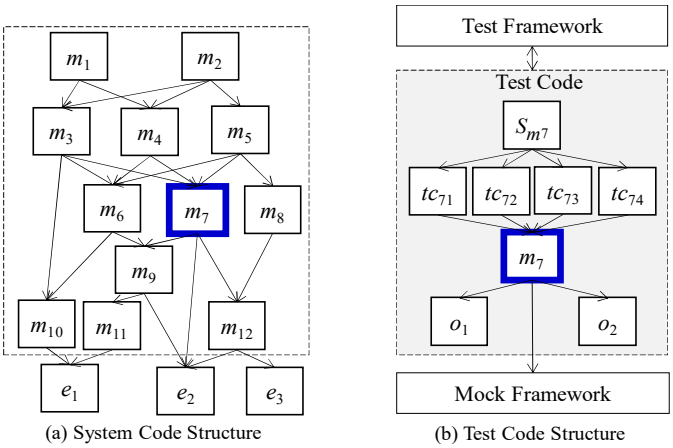
Java 프로그램에서 하나의 유닛 테스트 케이스(unit test case)는 검증대상 프로그램 내 하나의 모듈(예: 메소드, 함수)을 특정한 환경 아래에서 특정한 입력 값을 주어 실행시킨 후 그 결과를 해당 입력에 대한 기대 결과 값과 비교함으로써 해당 모듈의 동작 정확성을 판별하는 과정에 대한 프로그램이다. 이때, 검증대상 모듈의 면밀한 동작 검사를 위해, 해당 모듈을 다양한 입력 값들로 실행하는 여러 개의 연관된 테스트 케이스가 존재하는 것이 일반적이다. 일반적으로, 유닛 테스트케이스를 구현하는 테스트 코드는 다음과 같은 기능을 제공해야 한다:

- 검증대상 모듈이 시스템 전체의 실행 없이 동작할 수 있도록 가상적 실행 환경을 제공해야 함
- 테스트 실행이 시스템에 영향을 주지 않도록 검증대상 모듈을 고립시켜야 함
- 개발자가 의도한 테스트가 수행되도록 테스트 입력을 생성하고 테스트를 실행하며 그 결과를 검사하여야 함
- 테스트 효율성을 위해 연관된 유닛 테스트케이스가 효율적으로 실행되도록 관리해야 함
- 사용자 인터페이스를 제공하여, 사용자 명령에 따라 테스트를 실행하고 그 결과를 사용자에게 전달해야 함

이와 같이, 다양한 기능적 요구사항을 효율적으로 구현하기 위해, Java 프로그램의 유닛 테스트 코드는 일반적으로 JUnit, TestNG, Google Truth와 같은 유닛 테스팅 프레임워크 상에서 개발된다. 예를 들어, JUnit 프레임워크는 테스트 코드를 테스트 클래스(Test Class)와 테스트 메소드(Test Method)를 중심으로 구조화 한 양식(template)에 따라 작성하도록 요구하며, 이 양식을 맞춘 테스트 코드에 대해서는 다양한 API를 통해 테스트의 독립적이고 고립된 실행, 사용자 인터페이스 등의 기능을 간단히 사용할 수 있도록 제공한다.

그림 1은 테스팅 프레임워크를 활용하여 검증대상 모듈(i.e., m_7)에 대한 유닛 테스트 코드 구성을 개략적으로 설명하고 있다. 그림 1-(a)에서 설명하는 바와 같이, 시스템 전체 실행에서 검증대상 모듈의 실행은 사용자 인터페이스(i.e., m_1, m_2)로부터 발생한 복잡한 맥락 속에서 호출되며, 실행 과정에서 중 다른 모듈(e.g., m_9, m_{12}), 외부 시스템(i.e., e_1, e_2, e_3)과의 통신을 수반한다. 그림1-(b)는 검증대상 모듈만을 면밀히 검사하기 위해 작성된 테스트 코드의 예시다. 테스트 코드의 각 테스트케이스(i.e., tc_1,…,tc_7)는 검증대상 모듈의 실행 맥락을 모사하며 다양한 테스트 입력 값을 지정하여 검증대상 모듈을 호출한다. 이때

본 연구는NRF-2017R1C1B1008159,NRF-2017M3C4A7068179의 지원으로 수행됨



(a) System Code Structure (b) Test Code Structure

그림 1. 검증대상 모듈 m_7에 대한 유닛 테스트 코드의 작성

같은 검증대상에 대해 작성된 연관된 테스트 케이스는 테스트 스위트(Test Suite) 코드(e.g., S_{m7})에 의해서 호출된다. 또한 테스트 코드는 검증대상 모듈이 호출하는 시스템 내 다른 모듈을 테스팅 목적으로 간략히 표현한 테스트 모형(Test Mock Object)과 연결한다(e.g., o_1, o_2). 복잡도가 높은 테스트 모형 작성/수행에는(예: DBMS, Network protocol) 테스트 모형의 효율적인 작성과 관리를 지원하는 테스트 모형 프레임워크(Mock-up Framework)가 별도로 연결되어 사용되기도 한다.

2.2. 유닛 테스트 구성요소 정의

유닛 테스트 코드는 요구사항 구성요소, 실행과정 구성요소, 코드 구성요소의 세 가지 측면에서 설명할 수 있다. 본 절에서는 3가지 유닛 테스트 구성요소의 총체적인 범주(category)를 정의한다.

요구사항 구성요소: Java 프로그램 유닛 테스트는 일반적으로 다음 6개의 요구사항을 각각 독립적으로 만족해야 하므로, 각 요구사항 속성에 따라 범주(R1–R6)를 구별할 수 있다:

- **R1** 사용자가 지시한 테스트 케이스가 실행되어야 함
- **R2** 상이한 환경에서도 동일한 테스트 코드는 동일한 테스트 케이스를 실행해야 함
- **R3** 유닛 테스트 코드의 실행은 상호 간 영향을 받지 않아야 함
- **R4** 실행 결과가 올바르지 않은 경우, 테스트 실패(test fail)로 관찰되어야 함
- **R5** 테스트 실행 결과가 올바른 경우, 테스트 성공(test pass)으로 관찰되어야 함
- **R6** 테스트 결과가 사용자에게 올바르게 전달되어야 함.

실행과정 구성요소: 유닛 테스트 케이스는 각 실행과정은 다음의 7가지 종류로 구분하여 범주(S1–S7)로 정의할 수 있다:

- **S1** 사용자 입력에 따른 유닛 테스트 구동: 테스트 프레임워크를 통하거나 사용자의 직접 조작을 통해 주어지는 명령을 해석하여, 실행해야 할 유닛 테스트 코드를 지정하여 시행
- **S2** 테스트 실행환경 설정: 유닛 테스트의 실행을 통제하기 위해 검증대상 모듈의 실행에 영향을 줄 수 있는 여러 변수(환경변수, 공유 변수 등)를 특정 값으로 설정
- **S3** 테스트 입력 지정: 테스트케이스의 검증 목적에 따라 조정한 테스트 입력 값을 지정
- **S4** 검증대상 모듈 실행: 지정된 테스트 입력 값으로 검증대상 모듈의 기능을 실행
- **S5** 테스트 실행 결과 검사: 실행 결과를 테스트 입력 값에 대한 기대 값과 비교하여 테스트 통과 유무를 판별
- **S6** 테스트 실행환경 설정 해제: S2에서 설정한 실행환경을 해제
- **S7** 사용자에게 테스트 결과 보고: 테스트 통과 유무를 테스트 프레임워크를 통하거나 혹은 직접 사용자에게 전달

코드 구성요소: 그림1 (b)에서 설명하고 있는 바와 같이, 유닛 테스트 코드는 다음의 서로 다른 구조로 구성되므로, 각 코드 요소는 연관된 코드 부위에 따라 4가지 코드 구성요소(P1–P4)로 분류할 수 있다:

- **P1** 테스트케이스 별로 고유 시나리오를 정의하는 부위(예: tc_{71})
- **P2** 연관된 테스트케이스를 아울러 하나의 테스트 스위트를 구성하는 위한 부위(예: S_{m7})
- **P3** 검증대상 모듈이 호출하는 테스트 모형을 정의하거나 테스트 모형 프레임워크와 인터페이스 부위(예: o_1, o_2)
- **P4** 테스팅 프레임워크 인터페이스 부위로, 테스트 프레임워크의 기능을 호출/참조하는 부위

3. 유닛 테스트 코드 구성요소 범주에 기반한 결함 분류 체계

3.1. 분류 방법

앞서 정의한 3가지 구성요소의 총 17개 범주를 특정 유닛 테스트 결함(혹은 결함패턴)의 오류 증상과 결함 코드 요소(fault)와 연관시킴으로써, 해당 결함을 체계적으로 기술하고 분류할 수 있다. 우선, 테스트 결함의 오류 증상으로부터 잘못된 테스트 결과가 어떠한 종류의 요구사항 실패로 일어난 것인지 파악하여 R1–R6 중 관련 범주와 연관시킨다. 이 때 각 결함 사례는 3개의 구성요소별 최소 1개 이상의 범주와 연관시킬 수 있다. 또한, 테스트 결함이 오동작을 발생시키는 과정을 분석하여, 어떠한 실행단계에서 잘못된 동작이 발생하였는 지를 파악하여 S1–S7 중 관련 범주와 연관시킬 수 있다. 마지막으로, 테스트 결함의 코드 요소를 파악한 후, 해당 코드 요소가 속한 부위를 확인하여 P1–P4 중 관련 범주와 연관시킬 수 있다. 이 때 유닛 테스트의 각 코드 요소는 P1과 P2 중 하나의 범주에 속하며, 동시에 P3 혹은 P4에 속할 수 있다.

각 결함 사례를 구성요소별 범주와 연관한 이후에는, 같은 범주와 연관된 결함 사례의 집합을 검토함으로써, 유사한/연관된 결함 사례를 식별할 수 있으며, 이를 바탕으로 결함사례 분류를 수행할 수 있다.

3.2. 사례연구: 유닛 테스트 결함 검출 도구와 실제 결함 사례 분류

본 논문에서 제안한 분류 체계를 활용하여 현재 Java 프로그램 개발에 널리 사용되고 있는 오픈소스 정적 결함 검출 도구에 포함된 총 26종의 결함 검출기와 최근 조사연구에서 보고하는 실제 테스트 코드 결함 사례 중 선정한 22개의 사례를 분류하고, 그 연관성을 조사하였다.

조사대상으로 선정된 결함 검출기는 FindBugs [6] 중 6개, ErrorProne [7] 중 17개, PMD [8] 중 8개, Fb-contrib [9] 중 14개로 총 45개가 선정되었다. 이들은 4종의 도구가 제공하는 총 1238개의 결함 검출기 중 동작명세에 테스트 코드 결함을 검출 대상으로 명시한 모든 경우이다(refactoring, best practice 등 결함 검출과 직접 관련성이 없는 검출기는 제외하였다). 실제 테스트 코드 결함 사례로는 Vahabzadeh *et al.* [1]에서 조사한 총 443개의 실제 결함 사례 중, 논문에서 제시한 다섯 개의 카테고리를 고려하고 21개를 임의로 선정하였으며, 각 결함 사례는 결함 보고(bug report)와 논문의 설명을 참조하여 분류하였다.

표1은 조사대상 결함 검출기가 검출대상으로 삼은 테스트 코드

결함패턴과 조사대상으로 선정한 21개 실제 결함 사례를 분류한 결과다. 표의 첫 번째 열은 결함에 대한 간단한 설명이며, 두 번째부터 네 번째 열은 결함과 연관된 구성요소 범주를 제시하고 있다. 마지막 열은 결함의 출처로, 실제 결함의 경우, Apache 이슈 번호로 표기하였다.

조사대상인 총 66개 사례는 총 46가지 경우로 요약할 수 있었으며, 각 경우를 범주와 연관시킨 결과, R1–R6은 최소 3개에서 최대 18개, S1–S7은 최소 2개에서 최대 17개, P1–P4는 최소 2개에서 최대 33개의 경우와 연관되었다. 본 사례에서는 제안한 범주가 테스트 코드 결함의 다양한 요소를 효과적으로 구분함을 확인할 수 있었다.

분류 결과를 살펴볼 경우, 상이한 도구 간에 유사한 패턴을 검출하는 경우는 자주 발견되나(9건), 결함 검출기의 검출 대상 패턴과 실제 결함 사례가 같거나 유사한 경우는 발견되지 않았다. 이러한 결과로 미루어 볼 때, 실제 프로젝트에서 발생하는 테스트 결함은 현재 오픈소스 결함 검출기의 검출 대상과는 차이가 있음을 알 수 있었다. 또한, 같은 범주로 연관된 결함/결함패턴 간에도 세부적인 사례에 있어 상이한 경우가 많았다. 이와 같은 관찰을 통해, 실제 테스트 코드에서 발생하는 구체적인 결함 사례에 대한 추가적인 연구와 실제 결함을 효과적으로 포착하는 결함 검출 도구의 개발이 필요함을 확인할 수 있다.

4. 결론

본 연구에서는 Java 프로그램 유닛 테스트 코드의 구조요소별 범주를 정의하여 이를 바탕으로 유닛 테스트 코드에서 발생하는 결함을 분석/분류하는 체계를 제안하였다. 또한, 제안한 분류 체계를 이용해 45개의 오픈소스 정적 결함 검출기의 결함 패턴과 21개의 실제 결함 사례를 분류하고 그 연관성을 체계적으로 비교하였다. 향후 연구에는 제안한 분류체계를 이용해 보다 많은 테스트 결함의 분류를 수행함으로써, 유사한 사례에서 발견되는 공통 요소를 결함 패턴으로 식별하여 새로운 테스트 코드 결함 검출기를 정의하고자 한다.

참조문헌

[1] A. Vahabzadeh et al., An Empirical Study of Bugs in Test Code, ICSME 2015

[2] J. Bell et al., Efficient Dependency Detection for Safe Java Test Acceleration, FSE 2015

[3] S. Zhang et al., Empirically Revisiting the Test Independence Assumption, ISSTA 2014

[4] A. Gambi, J. Bell, A. Zeller, Practical Test Dependency Detection, ICST 2018

[5] J. M. Voas, PIE: A Dynamic Failure-Based Technique, IEEE Transactions on Software Engineering, 18(8), August 1992

[6] FindBugs, http:// findbugs.sourceforge.net

[7] ErrorProne, http://errorprone.info

[8] PMD, http://pmd.github.io

[9] Fb-Contrib, http://fb-contrib.sourceforege.net

[10] Q. Luo et al., An Empirical Analysis of Flaky Tests, FSE 2014

표1. 조사대상 결함 사례[1]와 결함 검출기의 결함 패턴[6][7][8][9]의 분류 결과

결함(결함 패턴) 설명	요구사항	실행과정	코드부위	출처
JUnit3 에서 TestCase 클래스에 test method 가 없는 경우	R1	S1	P1	FindBugs
JUnit3 에서 suite() 선언이 잘못된 경우	R1	S1	P2	FindBugs, PMD
JUnit4 에서 suite()를 오버라이드한 경우	R1	S1	P2	PMD
JUnit4 에서 test method 에 @Test Annotation 이 없는 경우	R1	S1	P4	FindBugs, ErrorProne
JMock 객체를 사용하면서 동시에, 해당 Test Class 에서 JMock Runner 를 사용하지 않음	R1	S1	P4	ErrorProne
testPath 를 잘못 설정함	R1	S2	P1	JCR-524
JUnit3 에서 setUp()이 오버라이드되어 있으나, super.setUp() 을 호출하지 않음	R1	S2	P3	FindBugs
JUnit4 에서 setUp()을 정의하고 @Before Annotation 이 없는 경우	R1	S2	P4	FindBugs, ErrorProne
JUnit4 의 @AfterClass @BeforeClass annotation 을 사용하면서 method 가 static 이 아님	R1	S2	P4	ErrorProne
JUnit3 에서 약속된 메소드 명이 잘못된 경우	R1	S2, S6	P1	ErrorProne
JDK9 로 테스트 코드를 컴파일했을 때 오류가 나는 Mockito code pattern 을 사용한 경우	R1	S3	P1	ErrorProne
연속으로 동일한 Assertion 을 호출함	R1	S5	P1	DERBY-6716
FileInputStream 을 생성하였으나 close()를 호출하지 않음	R1	S6	P1	FLUME-349
JUnit3 에서 tearDown()이 오버라이드되어 있으나, super.tearDown() 을 호출하지 않음	R1	S6	P3	FindBugs
JUnit4 에서 tearDown()을 정의하고 @After Annotation 이 없는 경우	R1	S6	P4	FindBugs, ErrorProne
테스트 내에서 Thread 를 생성했으나 join()을 하지 않고 테스트가 종료됨	R1, R2	S2, S3, S6	P1	DERBY-5708
org.junit.Test 를 import 하는 java 파일의 public class 명이 Test 로 시작되지 않음	R1, R3	S1	P4	SLIDER-41
Exception 이 기대한 테스트케이스에서 Exception 이 발생하지 않아도 fail() 하지 않음	R1, R4	S5	P1	DERBY-6088,DERBY-3852
SSLContext 의 getInstance 메소드에 "SSL"을 파라미터로 넘긴 경우	R2	S2, S3	P1	FLUME-2441
file path 의 경로를 '/'으로만 구분하는 경우	R2	S3	P1	MAPREDUCE-4983
assertThrows 에 넘겨진 코드에서 Exception 을 throw 한 이후 statement 가 남아있음	R2	S4	P1	ErrorProne
for 문의 조건문을 잘못 입력하여 test 를 실행하지 못한 경우	R2	S4	P1	HDFS-2596
파일을 생성한 뒤 try-catch 블록 이후 finally 블록에 close()를 호출하지 않은 경우	R2	S6	P1	JCR-267
Test Suite 에서 하나의 DB 를 사용하고, Table 이나 Sequence 를 생성한 뒤 Drop 하지 않은 경우	R3	S2, S5	P2	DERBY-4393
Socket 을 생성하고 테스트가 끝나는 시점에 close() 하지 않은 경우	R3	S6	P1	HDFS-7282
테스트 실행 도중 공유 변수의 값을 임의로 변경한 후 되돌리지 않음	R3	S6	P1, P2	FLUME-571, ACCUMULO-2198
Ant 1.9.3 에서 ant 가 test 결과를 file 을 수정할 수 있는 권한을 설정하지 않음	R3	S7	P4	DERBY-6685
모든 test 가 실행되는지 확인하지 않음	R4	S1	P4	JENA-795
test 가 추가되기 전에 호출되어야 할 메소드 leasechecker.interruptAndJoin()이 누락됨	R4	S2	P1	HDFS-824
테스트 환경을 설정하는 동일한 메소드를 연속해 중복 실행하는 경우	R4	S2	P1	HBASE-11698
setup() 이나 teardown()에서 Exception 을 catch 하지 않음	R4	S2, S6	P2	JCR-505
Test Method 내부에서 Assertion 메소드의 호출이 없는 경우	R4	S5	P1	PMD, Fb-contrib,
assertNull() 안에 boxed primitive 가 파라미터로 넘겨지는 경우	R4	S5	P1	Fb-contrib
Test Framework 를 사용하는 동시에 assert()를 사용하는 경우,	R4	S5	P1	ErrorProne, Fb-contrib
Assertion method 에서 동일한 객체의 reference 가 서로 같은지 확인하는 경우	R4	S5	P1	ErrorProne
Mockito 를 사용하는 테스트에서 verify 메소드를 사용하지 않음	R4	S5	P1	ErrorProne
Assertion 메소드를 사용하여 동일한 객체에 대해 equality 를 검증하는 경우	R4	S5	P1	ErrorProne
Exception 이 기대한 테스트케이스에서 Exception 이 발생하지 않아도 fail() 하지 않음	R4	S5	P1	ErrorProne
IOException 를 발생시키는 메소드를 assertNull()에서 호출함	R4	S5	P1	MAPREDUCE-5421
Thread.run() 내부에 assertio 이 있는 경우	R4	S7	P1	FindBugs
Assertion 메소드에서 test 하고자 하는 객체가 상수인 경우	R4, R5	S5	P1	ErrorProne, Fb-contrib
assert 구문에서 타입이 다른 두 객체의 equality 를 비교하는 경우	R4, R6	S5	P1	ErrorProne
assertionEquals()에서 부동 소수점 비교에서 오차 허용 범위가 없는 경우	R5	S5	P1	Fb-contrib
Try 블록 내부에서 fail()이 있으나, catch 블록에서 assertionError 를 catch 하는 경우	R5	S5	P1	ErrorProne
assertTrue() 내부의 판별식에서 equals 를 호출하여 두 객체가 같음을 확인하는 경우	R6	S5	P1	PMD, Fb-contrib
assertTrue() 내부의 판별식에서 test 하고자 하는 객체가 null 인지를 확인하는 경우	R6	S5	P1	PMD, Fb-contrib