ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 46, No. 4, pp. 341-354, 2019. 4
https://doi.org/10.5626/JOK.2019.46.4.341

Java 프로그램 단위 테스트 코드에서 발생하는 결함의 분류와 패턴 정의

(Categories and Patterns of Java Program Unit Test Code Bugs)

최 한 솔 [†]　　홍 신 [††]
(Hansol Choe)　　(Shin Hong)

요 약 단위 테스트 케이스를 이용한 자동 회귀 테스팅 방법론이 널리 사용됨에 따라, 단위 테스트 케이스 작성 중 발생하는 '테스트 코드 결함'이 소프트웨어 제품의 품질과 프로젝트의 생산성을 저하하는 새로운 소프트웨어 품질 문제로 대두되고 있다. 이러한 단위 테스트 코드 결함의 체계적인 이해와 탐지를 위하여, 본 논문에서는 Java 프로그램 단위 테스트 케이스 결함을 분류하는 결함 분류 체계와 실제 결함 사례에 기반한 단위 테스트 케이스 결함 패턴을 소개한다. 테스트 케이스 결함에 대한 단편적인 분류 기준을 제시하는 기존 연구와 달리, 본 연구에서는 단위 테스트 코드의 다양한 구조적, 기능적, 의미적 구성 요소의 범주를 제시한 후 이에 기반한 총체적인 분류 체계를 제안하며, 이를 이용해 실제 결함 사례와 정적 결함 검출기의 검출대상을 분류한 결과 소개한다. 이에 더하여, 본 논문에서는 실제 테스트 결함 사례로부터, 테스트 코드 결함의 주요 조건을 구체적이고 명확하게 표현한 새로운 8종의 테스트 결함 패턴을 소개한다.

키워드: 단위 테스팅, 테스트 결함, 결함 패턴, 결함 분류, 정적 결함 검출

Abstract Since unit testing is widely used in many software projects, the threat of unit test bugs (i.e., bugs in the test case code) is becoming a more important issue of software quality assurance. Test code bugs are critical threats because they may invalidate the quality assurance process, which consequently hurts quality of products and performance of the project. This paper presents a set of test bug categories and a set of bug patterns extracted from real-world cases. Unlike the existing work on test code bugs, this paper suggests a classification method to systematically categorize different features of test code bugs (i.e., structures, operations, and requirements). In addition, this paper defines eight new bug patterns in unit test code, based on previous bug reports from well-known open-source projects. Each pattern is formally specified as source code patterns so that it can be used for to construct a static bug pattern checker.

Keywords: unit testing, test code bug, bug patterns, bug classification, static bug detection

· 본 연구는 한동대학교 교내연구지원사업(제20160063호)과 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행되었음 (NRF-2017R1C1B1008159, NRF-2017M3C4A7068179)
· 이 논문은 2018 한국컴퓨터종합학술대회에서 'Java 프로그램의 유닛 테스트 코드에서 발생하는 결함의 분류'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 한동대학교 전산전자공학부
hansolchoe@handong.edu
†† 정 회 원 : 한동대학교 전산전자공학부 교수(Handong Global Univ.)
hongshin@handong.edu
(Corresponding author임)

논문접수 : 2018년 10월 23일
(Received 23 October 2018)
논문수정 : 2019년 2월 6일
(Revised 6 February 2019)
심사완료 : 2019년 2월 13일
(Accepted 13 February 2019)

정보과학회논문지 제46권 제4호(2019. 4)

1. 서 론

연속적 통합 환경에서 회귀 테스팅(regression testing)이 소프트웨어 품질 관리 방법으로 널리 이용됨에 따라, 회귀 테스팅의 자동 수행을 지원하기 위한 단위 테스트 케이스 작성이 소프트웨어 개발에 일상화되고 있다. 단위 테스트 케이스란 테스트 시나리오에 따라 검증대상 모듈을 독립실행 가능한 형태로 재구성하고, 해당 모듈에 특정 입력을 설정하여 실행한 후, 그 실행 결과가 올바른 지 검사하는 일련의 과정을 자동화한 코드로, 개발자는 특정 모듈의 요구사항과 시스템 내에서의 실행 맥락을, 단위 테스트 케이스에 정확하고 총체적으로 기술하여야 한다. 단위 테스트 케이스를 통한 회귀 테스팅 방법의 확산과 함께, 단위 테스트 케이스 작성 시 프로그래밍 실수로 인하여 발생하는 테스트 코드 결함(test code bug)이 소프트웨어 품질 관리의 새로운 난제로 부각되고 있다[1,2]. 테스트 케이스에 존재하는 결함은, 특정 모듈의 품질 관리 문제는 물론, 결함이 있는 테스트 케이스와 동시에 수행되는 단위 테스팅 과정 전반의 정확성과 생산성 하락을 야기할 수 있어, 일반적인 소프트웨어 결함에 비하여 실패의 위험성이 큰 것으로 인식되고 있다.

오픈소스 Java 프로젝트를 대상으로 한 최근 조사 연구[1]는 다양한 증상과 형태를 가진 여러 종류의 테스트 결함이 실제 소프트웨어 프로젝트에서 빈번히 발생하고 있음을 보고하고 있는 반면, 테스트 케이스 결함에 대한 총체적이고 체계적인 이해가 제한적인 실정이다. 현재 개발된 자동 결함 검출 기법은, 여러 종류의 테스트 코드 결함 중 '의도하지 않은 테스트 실행 순서 종속성'(Test-Order Dependency) 결함 검출[3-5]에 한정된 실정으로, 다양한 테스트 결함 검출의 효과적인 검출이 어려운 실정이다.

본 연구는 Java 프로그램에서 발생하는 다양한 종류의 테스트 코드 결함을 보다 체계적인 분석하고 자동 검출할 수 있도록 지원하기 위해 (1) 테스트 결함에 특화된 결함 분류 체계를 제안하고 (2) 정형화된 8개의 새로운 테스트 결함 패턴을 정의한다. 우선, 본 논문에서는 테스트 코드의 일반적인 요구사항, 구성형태, 실행 순서를 모델링한 후, 이를 바탕으로 정립한 테스트 결함 체계를 소개한다. 그리고 제안한 분류 체계를 통해 최근 연구결과[1]에서 보고된 총 44개의 실제 테스트 결함 사례와 FindBugs[6], ErrorProne[7], PMD[8], Fb-contrib[9]에 존재하는 45개의 결함 검출기의 검출 대상 결함을 분류한 결과를 설명한다.

특정 종류에 특화된 소프트웨어 결함 분류 체계는 해당 결함의 구조적 형태, 오류 발생 조건, 오류 실행 증상을 통합적으로 설명하고, 이를 바탕으로 실제 결함 사례를 분류하고 연관 요소를 종합하는 방법론을 제공함으로써, 해당 결함에 대한 정적/동적 결함 검출기 개발을 지원하는 것을 목표로 한다(예: [10-12]). 하지만, 테스트 코드 결함에 대하여 현재 제안된 분류 체계의 경우[1], 결함에 관련된 코드 요소(예: Exception Handling), 결함 증상(예: Resource Leak), 오류가 발생하는 상황(예: Difference in Operating System) 등 분류 기준이 한 가지 특성에 대해 단편적이고 비체계적으로 제시되어 있어, 특정한 결함 사례가 주어졌을 때를 이를 분석하거나 관련된 결함을 조사할 수 있는 기준으로 사용하기는 제한적이다. 본 논문에서 제안하는 분류 체계는 하나의 테스트 결함 요소를 3가지 관점에서 조합적으로 분류할 수 있는 기준을 제시함으로써, 테스트 결함에 대한 체계적인 이해와 분류가 가능하도록 하였다는 점에서 기존에 제안된 결함 분류 체계의 한계점을 개선함으로써 결함 사례를 보다 체계적으로 표현하고 분류할 수 있도록 지원한다.

또한, 본 논문에서는 실제 오픈소스 프로젝트에서 보고된 테스트 결함 사례를 바탕으로 새롭게 정의한 8개의 테스트 결함 패턴을 정의하였다. 8개의 테스트 결함 패턴은 기존의 관련 도구에서 탐지하지 못하는 새로운 종류의 테스트 결함을 대상으로 정의되었다. 본 논문은 8개의 테스트 결함 패턴이 FindBugs와 같은 코드 패턴 기반 정적 결함 검출기에 효과적으로 활용할 수 있도록, 구체적인 패턴 정의를 소개하도록 한다.

본 논문의 2장에서는 단위 테스트 코드의 기능과 구조를 설명한 후, 일반적으로 단위 테스팅을 구성하는 요소를 정의한다. 3장에서는 2장에서 파악한 단위 테스트 구성 요소를 바탕으로 한 단위 테스트 결함 분류 체계를 제안한다. 3장에서는 2장에 제안한 분류 체계를 활용하여 총 39개의 결함 사례와 45개의 결함 검출기의 검출 대상을 분류한 결과를 소개한다. 4장에서는 총 8개 결함 패턴이 각각 어떠한 결함을 검출 대상으로 정의되었으며, 어떠한 코드 패턴으로 해당 결함을 포착하는 지를 구체적으로 기술한다. 마지막으로 5장에서는 결론으로 논문을 마무리한다.

본 논문에서 소개하는 결함 분류 체계는 본 연구의 선행 연구 결과에 발표되었다[2]. 선행 연구에 추가하여, 본 논문은 실제 정적 결함 검출에 활용할 수 있는 8개의 결함 패턴을 새롭게 소개하였다(4장). 또한, 결함 분류 사례 연구(3.2절)에서 실제 결함 분류 사례를 확충하여 (선행 연구[2] 대비 86% 증대) 논의하였으며, 분류 체계의 분류 기준을 한 가지 추가(2.2절)함으로써, 선행 연구 결과를 보완한 테스트 결함 분류 방법을 소개한다.

2. 단위 테스트 코드의 구성

소프트웨어 결함은, PIE 모델[13]에 따라, (1) 잘못된 코드 요소(결함)의 실행(execution), (2) 잘못된 코드 요소의 실행에 따른 의도치 않은 상태 발생(infection), (3) 의도치 않은 상태로부터 잘못된 동작이 연쇄되어 요구사항 달성 실패(propagation)의 세 가지 요소를 파악함으로써 체계적으로 분석이 가능하다[5]. 본 장에서는 Java 프로그램의 일반적인 단위 테스트 코드의 구성 형태, 동작 과정, 그리고 단위 테스트에 대한 일반적 요구사항을 여러 가지 경우로 나누어 설명함으로써, 3장에서 소개할 단위 테스트 결함 모델의 배경을 제공한다.

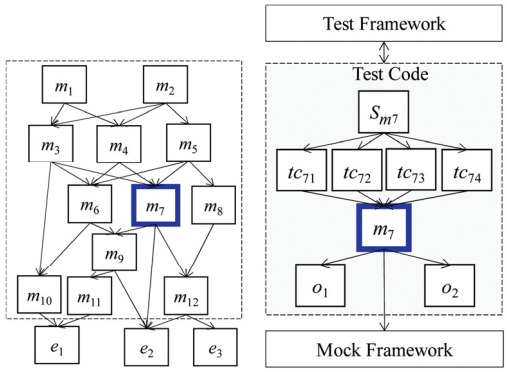
2.1 일반적인 단위 테스트 코드의 구성과 기능

Java 프로그램에서 하나의 단위 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, 단위 테스트케이스를 구현하는 테스트 코드는 다음과 같은 기능을 제공해야 한다:

- 검증대상 모듈을 전체 시스템으로부터 분리시켜 독립적으로 동작 가능하도록 가상적 실행 환경을 제공해야 함
- 테스트 실행이 시스템 전체 동작에 영향을 주지 않도록 검증대상 모듈을 고립시켜야 함
- 개발자가 의도한 테스트가 수행되도록 테스트 입력을 생성하고 테스트를 실행하며 그 결과를 검사하여야 함
- 테스트 생산성을 위해 연관된 단위 테스트 케이스가 효율적으로 실행되도록 관리해야 함
- 사용자 인터페이스를 통해 테스트 운영자의 명령을 수신하여 그에 따라 테스트를 실행하고, 테스트 결과를 올바로 테스트 운용자에게 전달해야 함

이와 같이, 다양한 기능적 요구사항을 효율적으로 구현하기 위해, Java 프로그램의 단위 테스트 코드는 일반적으로 JUnit, TestNG, Google Truth와 같은 단위 테스팅 프레임워크 상에서 개발된다. 예를 들어, JUnit 프레임워크는 테스트 코드를 테스트케이스 클래스(Test Case class)와 테스트 메소드(Test Method)를 중심으로 구조화된 양식에 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 프레임워크 상에서 구현된 검증대상 모듈(m_2, m_7)에 대한 단위 테스트 코드의 구성의 예시다.



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

그림 1 검증대상 모듈 m_7에 대한 단위 테스트 코드

Fig. 1 An example of unit test code structure

그림 1(a)에서 묘사하는 바와 같이, 시스템 전체 실행에서 검증대상 모듈의 실행은 사용자 인터페이스(m_1, m_2)로부터 발생한 복잡한 맥락 속에서 호출되며, 실행 과정에서 중 다른 모듈(예: m_9, m_{12}), 외부 시스템(예: e_1, e_2)과의 상호작용을 수반한다. 그림 1(b)는 검증대상 모듈만을 면밀히 검사하기 위해 작성된 테스트 코드의 예시다. 테스트 코드의 각 테스트케이스(tc_{71}, …, tc_{74})는 검증대상 모듈(m_7)의 실행 맥락을 모사하며 다양한 테스트 입력 값을 지정하여 검증대상 모듈을 호출한다. 이때 같은 검증대상에 대해 작성된 연관된 테스트 케이스는 테스트 스위트(Test suite) 코드(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(o_1, o_2). 복잡도가 높은 테스트 모형 작성/수행에는(예: DBMS, 네트워크 통신 스택) 테스트 모형(Mock-up)의 효율적인 개발/관리를 지원하는 테스트 모형 프레임워크가 추가로 사용되기도 한다.

2.2 단위 테스트 구성요소 정의

단위 테스트 코드는 (1) 요구사항 요소, (2) 실행과정 요소, (3) 코드구성 요소의 세 가지 측면에서 설명할 수 있다. 본 절에서는 3가지 단위 테스트 구성요소의 총체적인 범주(category)를 정의한다.

○ **요구사항 요소:** Java 프로그램 단위 테스트가 일반적으로 만족해야 하는 요구사항은 다음의 일곱 종류로 구분할 수 있다:

R1 사용자의 명령에 따라 테스트 코드로 실행이 도달해야 함

R2 테스트가 의도한 시나리오에 따라 올바른 테스트 입력을 테스트 대상에게 입력해야 함

R3 상이한 환경에서도 동일한 테스트 코드는 동일한 테

스트 동작을 실행해야 함

R4 단위 테스트 코드의 실행은 상호 간 영향을 받지 않아야 함

R5 실행 결과가 올바르지 않은 경우, 테스트 실패(test fail)로 관찰되어야 함

R6 테스트 실행 결과가 올바른 경우, 테스트 성공(test pass)으로 관찰되어야 함

R7 테스트 실행 결과가 사용자에게 올바르게 전달되어야 함

○ **실행과정 요소:** 단위 테스트 케이스는 실행의 세부 과정은 다음 일곱 가지 종류의 동작으로 분류할 수 있다:

S1 사용자 입력에 따른 단위 테스트 구동: 테스트 프레임워크를 통하거나 사용자의 직접 조작을 통해 주어지는 명령을 해석하여, 실행해야 할 단위 테스트 코드를 지정하여 시행

S2 테스트 실행환경 설정: 단위 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

○ **코드영역 요소:** 그림 1(b)이 설명하고 있는 바와 같이, 단위 테스트 코드는 다양한 구조의 복합으로 구성되는데, 일반적인 단위 테스트의 각 코드 요소는 다음 네 가지 영역 중 하나에 위치하는 것으로 구분할 수 있다:

P1 테스트케이스 별로 고유 시나리오를 정의하는 코드 영역(예: tc_{71})

P2 연관된 테스트케이스를 아울러 하나의 테스트 스위트를 구성하는 위한 영역(예: s_{m7})

P3 검증대상 모듈이 호출하는 테스트 모형을 정의하거나 혹은 테스트 모형 프레임워크와 인터페이스 영역(예: o_1, o_2)

P4 테스팅 프레임워크 인터페이스 부위로, 테스트 프레임워크의 기능을 호출/참조하는 영역

그림 2는 앞서 정의한 총 18개 요소(R1-7, S1-7, P1-4)가 실제 테스트 코드와 어떻게 연관되는지를 설명하기 위한 예시다. 그림 2에서 테스트 코드는 그래프로 표현되었다. 예시 코드는 13개의 구문을 갖는 것으로 하여, 각 구문을 노드(원으로 표현)로, 구문 간의 제어흐름은 간선으로 표현하였다. 이 중 7번 구문에 결함이 있는 것으로 가정하여, 그림 상에 특별한 표식을 붙였다. 그림 2에서 나타내는 바와 같이, 특정 테스트 케이스를 구현한 코드는 여러 영역(P1-4)에 걸쳐 작성된 코드 조각의 복합으로 볼 수 있다. 테스트 코드는 다양한 실행과정(S1-7)을 달성하기 위해 순차적으로 실행되며, 최종적으로는 실행 결과를 사용자(테스트 운영자)에게 전달하게 된다. 테스트 코드영역과 실행과정은 독립적인 요소로, 같은 실행과정에 속한 구문이 서로 다른 영역에 정의되어 있을 수 있으며(예: 그림 2의 구문2, 구문3), 또한 같은 영역 내에 서로 다른 실행과정이 존재할 수 있다(예: 그림 2의 구문3, 구문4).

그림 2는 단위 테스트 오류 결함에 다면적 요소가 있음을 도식적으로 보여주는 예시다. 본 예시의 구문2와 구문3은 동일한 코드 영역에 속한 구문으로서 서로 다른 실행 요소를 담당하는 경우가 발생할 수 있음을 보이며, 또한 구문5와 구문6은 동일한 실행 요소를 담당하는 구문이라도 서로 다른 코드 영역에 속할 수도 있음을 보인다. 또한, 결함이 위치하는 구문7이 실행되어 잘못된 상태가 발생(infection)한 직후 오류 증상이 관찰되지 않고, 추가적인 실행 과정을 거친 후, 특정한 요구사

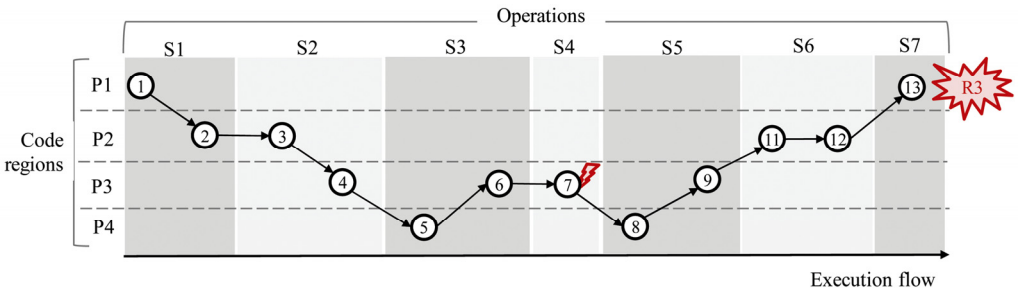


그림 2 단위 테스트 결함의 요구사항, 실행과정, 코드영역 요소 예시

Fig. 2 An example of a unit test execution and its requirement, operation and code region aspects

항 검사와 연관된 구문(구문13)에서 관찰되는 상황이 있을 수 있음을 예시하고 있다. 따라서 단위 테스트를 분류하고 분석하기 위해서는 한 가지 분류기준 대신 요구사항, 실행단계, 코드영역의 세 가지 기준을 종합적으로 사용한 분류 체계가 필요하다.

3. 단위 테스트 결함 분류 체계

3.1 분류 방법

앞서 정의한 3가지 구성요소의 총 18개 범주를 특정 단위 테스트 결함(혹은 결함패턴)의 오류 증상과 결함 코드 요소(fault)와 연관시킴으로써, 해당 결함을 체계적으로 기술하고 분류할 수 있다. 이 때 각 결함 사례는 3개의 구성요소별로 최소 1개 이상의 범주와 연관시킬 수 있다. 먼저, 테스트 결함의 오류 증상으로부터 잘못된 테스트 결과가 어떠한 종류의 요구사항의 실패로 일어난 것인지 파악하여 R1에서 R7 중 관련 범주와 연관시킨다. 이어서, 테스트 결함이 오동작을 발생시키는 과정을 분석하여, 어떠한 실행단계에서 잘못된 동작이 발생하였는지를 파악하여 S1에서 S7 중 관련 범주와 연관시킬 수 있다. 마지막으로, 테스트 결함의 코드 요소를 파악하여 해당 코드 요소가 속한 부위를 확인하여 P1에서 P4 중 관련 범주와 연관시킬 수 있다. 이 때 단위 테스트의 각 코드 요소는 P1과 P2 중 하나에 속하는 동시에 P3 혹은 P4에 속할 수 있다. 예를 들어, 그림2의 경우, 결함(구문7)이 P3, S4, R3 요소를 갖는 것으로 판별할 수 있다.

각 결함 사례를 구성요소별 범주와 연관한 이후에, 같은 범주와 연관된 결함 사례의 집합을 검토함으로써, 유사하거나 혹은 상호 간에 연성이 있는 테스트 결함 사례를 식별할 수 있으며, 이를 바탕으로 여러 테스트 결함을 체계적으로 분류하여 관찰할 수 있다.

3.2 사례연구: 단위 테스트 결함 검출 검출기와 실제 결함 사례의 분류

본 논문에서 제안한 분류 체계를 활용하여 (1) 현재 Java 프로그램 개발에 널리 사용되고 있는 오픈소스 정적 결함 검출 도구에 포함된 총 45개의 결함 검출기의 검출 대상 결함 패턴과 (2) 테스트 결함 조사연구[1]에서 보고하는 실제 테스트 코드 결함 사례 중 선정한 39개의 사례를 합한 총 84개 테스트 결함을 분류하였다. 또한, 분류 결과를 검토함으로써 테스트 결함 검출기와 실제 사례 간의 연관성을 논의하였다.

ㅇ 조사 대상: 단위 테스트 결함을 검출하는 오픈소스 기반 정적 결함 검출기로는 FindBugs [6] 중 6개, ErrorProne [7] 중 17개, PMD [8] 중 8개, Fb-contrib [9] 중 14개로, 총 45개를 선정했다. 이들은 네 종류의 오픈소스 정적 결함 검출기 프레임워크에 등록된 총 1238개 결함 검출기 중 동작명세에 테스트 코드 결함을 검출 대상으로 명시한 모든 경우이다(Best practice, Refactoring 등 규칙 검사기 제외).

실제 테스트 코드 결함 사례로는 Vahabzadeh 등이 발표한 연구[1]에서 수집한 총 443개의 코드 수정 사례 중 임의의 60개를 선정한 후, 각 사례의 결함 보고(bug report), 코드 수정 내역 등을 검토하여, 최종적으로 39개를 테스트 결함 사례로 판별하여, 조사에 사용하였다. 조사에 쓰이지 않은 21개의 코드 수정은 테스트 결함이 아닌 경우(예: 요구사항 변화에 따른 수정, 기능 변화가 동반되지 않는 리팩토링)이거나, 단순한 프로그래밍 실수이거나, 혹은 결함에 대한 정보가 불충분하여 분석이 불가능한 것으로 판별할 수 있는 경우였다.

ㅇ 분류 결과: 표 1은 조사대상으로 선정한 45개 테스트 결함 검출기의 검출대상 결함 패턴과 조사대상으로 선정한 39개 실제 결함 사례를 일반화하여 정의한 55개 결함 패턴을 3.1절에서 제안한 모델에 따라 분류한 결과다. 표 1의 첫 번째 열은 결함 패턴에 대한 설명으로 조사 대상 결함 검출기의 검출 대상 혹은 실제 결함 사례의 핵심적인 조건을 간단히 서술한다. 두 번째부터 네 번째 열은 결함과 연관된 구성요소 범주를 제시하고 있다. 다섯 번째 열은 결함 패턴과 연관된 조사 대상을 나타낸다. 결함 검출기의 경우, 도구명과 해당 결함 검출기의 고유 명칭의 복합으로 표기하였으며, 'FB:'는 FindBugs, 'FBC:'는 FB-contrib, 'PMD:'는 PMD, 'EP:'는 ErrorProne을 지칭한다. 실제 결함의 경우, 프로젝트 내 이슈 관리 번호로 표기하였다. 여러 개의 결함 검출기나 실제 결함 사례가 동일한 결함 패턴에 대응되는 경우, 표 1의 같은 칸에 함께 표현하였다.

각 결함 패턴 혹은 결함 사례가 어떠한 결함 분류 요소에 해당하는 지는 대학원생 1명과 교수 1명이 관련 소스코드, 결함 보고 등을 참고하여 종합적으로 논의한 결과에 따라 판별하였다. 특정 결함 패턴이 같은 범주에 있는 두 개 이상의 요소에 동시에 연관되어 있는 경우, 같은 칸에 함께 표현하였다.

ㅇ 관찰 및 논의: 표 1에서 소개하는 총 60개의 테스트 결함 패턴을 살펴볼 때, 본 논문의 2.2절에서 제시한 18개의 테스트 결함 분류 요소(R1-7, S1-7, P1-4)가 최소 한 1개 이상의 결함 패턴과 연관되어 있음을 확인할 수 있다. 예를 들어, 각 요구사항 요소는 최소 1개(R7)에서 최대 17개(R3)의 결함 패턴과, 각 실행과정 요소는 최소 1개(S3)에서 최대 16개(S6)의 결함 패턴과, 마지막으로 코드영역 요소는 최소 3개(P3)에서 최대 40개(P1)의 결함 패턴 요소와 연관된다. 이 결과를 통해 본 논문에서 제안한 테스트 결함 분류 방법이 실제 테스트 코드 결함의 다양한 요소를 효과적으로 구분함을 알 수 있다.

표 1 조사대상 결함 사례[1]와 결함 검출기의 결함 패턴[7-10]의 분류 결과
Table 1 Test bug patterns observed in the studied bug checkers and real bug cases and their classifications

Bug Pattern Description	R	S	P	Bug Checkers and/or Test Bug Cases
A TestCase class of JUnit 3 has no test method	R1	S1	P1	FB: IJU_NO_TESTS
suite() is defined incorrectly in JUnit 3	R1	S1	P2	FB: IJU_BAD_SUITE_METHOD, FB: IJU_SUITE_NOT_STATIC, PMD:JUnitStaticSuite
suite() is overridden in JUnit 4	R1	S1	P2	PMD: JUnit4SuitesShouldUseSuiteAnnotation
The name of a TestCase class does not satisfy the rule of Maven	R1	S1	P4	SLIDER-41
A test method is not declared with @Test annotation in JUnit 4	R1	S1	P4	PMD: JUnit4TestShouldUseTestAnnotation, EP: JUnit4TestNotRun
A TestCase class uses JMock objects without using JMock Runner	R1	S1	P4	EP: JMockTestWithoutRunWithOrRuleAnnotation
A overridden setUp() does not invokde super.setUp()	R1	S2	P3	FB: IJU_SETUP_NO_SUPER, HDFS-725
setUp() is defined without @Before annotation in JUnit 4	R1	S2	P4	PMD: JUnit4TestShouldUseBeforeAnnotation EP: JUnit4SetUpNotRun
A non-static method is declared with @AfterClass or @BeforeClass	R1	S2	P4	EP: JUnit4ClassAnnotationNonStatic
Reserved method names are incorrectly used in JUnit 3	R1	S2,S6	P1	EP: JUnit3TestNotRun
Incorrect use of Mockito with JDK 9	R1	S3	P1	EP:MockitoCast
A overridden tearDown() does not invoke super.tearDown() in JUnit 3	R1	S6	P3	FB: IJU_TEARDOWN_NO_SUPER
tearDown() is defined without @After annotation in JUnit 4	R1	S6	P4	PMD: JUnit4TestShouldUseAfterAnnotation, EP: JUnit4TearDownNotRun
A test suite fails to add a test method using Reflection because the test method is not built depending on a running environment (JDK)	R1,R3	S1	P2	DERBY-3588
A test method terminates unintentionally when an Exception is thrown before executing the test target method	R2	S2	P3	MAPREDUCE-6125
There exist statements after an expected exception throwing in the lambda passed to assertThrows() in JUnit 5	R2	S4	P1	EP: AssertThrowsMultipleStatements
A test method receives the status of an external file system as input, and it unintentionally fails for a subtle case of the file system status.	R2,R3	S4	P1	HBASE-8567
A test method does not give a commit command to DBMS, so that the following tests that use DBMS are unintentionally interfered	R2,R3	S6	P1	DERBY-4607
A test case is falsely bound to a specific time zone, so that it's execution on other time zone unintentionally fails	R3	S2	P1	ABMARI-4243
A test case waits for an arbitrary amount of time for I/O operation, and it fails when the time is not enough for I/O	R3	S2	P1	ACCUMULO-3113
A test case fails depending on platforms because it uses a platform-specific feature of a third-party library to set up the test case	R3	S2,S6	P1,P2	FLUME-2441, JCR-495
Since a test case is hardwired with a specific port number for network, it fails when the port is occupied by another process.	R3	S2	P1	HBASE-3794, HIVE-1716
Since the test case falsely assumes a certain order of test execution order, it unintentionally fails if the tests run in different orders	R3	S2	P1	MAPREDUCE-5868
A test case falsely uses a platform-specific path separator ('/')	R3	S2,S6	P2	MAPREDUCE-4983, MAPREDUCE-5259
Since a test case sets a waiting time for a network communication tightly, it fails depending on the running circumstances.	R3	S4	P1	DERBY-6701
A test case shows race condition as it communicates with a concurrent service without proper synchronizations	R3	S4	P1	FLUME-1788
A test case falsely assumes that every line of a file is delaminated by '\n'	R3	S4	P1	HADOOP-9294
A test case fails unintentionally when another process concurrently accesses an external resource that the test case is accessing	R3	S6	P1	HDFS-824
A test case creates a FileInputStream to open a file, and then terminates without releasing the acquired resource.	R3	S6	P1	FLUME-349
A test case spawns a thread for a task, and then accesses the results while falsely assuming that the task had done in certain time	R3,R4	S5	P1	UIMA-2912
A test case opens a DBMS transaction, and then terminates without closing the transaction (i.e., clse())	R3,R4	S6	P1	DERBY-3323
A test case temporary updates a variable shared by other test cases, but terminates without restoring it back to the original value	R4	S2,S6	P1,P2	FLUME-571, ACCUMULO-2198

표 1 조사대상 결함 사례[1]와 결함 검출기의 결함 패턴[7-10]의 분류 결과 (이어짐)

Table 1 Test bug patterns observed in the studied bug checkers and real bug cases and their classifications (Cont'd)

Bug Pattern Description	R	S	P	Bug Checkers and/or Test Bug Cases
A test method spawned threads and then terminates without waiting the terminations of the created threads (i.e., no join() operation)	R4	S4,S6	P1	DERBY-5708, HDFS-4693
There is no assert statement in a test method	R4	S5	P1	PMD: JUnitTestsShouldIncludeAssert, FBC: NO_ASSERT
A boxed primitive is passed to assertNull()	R4	S5	P1	FBC: ASSERT_NULL
assert() is used in the Test framework	R4	S5	P1	EP: UseCorrectAssertInTests, FBC: ASSERT_USED
An assert statement meaninglessly checks whether a reference to an object is identical to itself	R4	S5	P1	EP: TruthSelfEquals, EP: SelfEquals
A test method using Mockito does not use verify()	R4	S5	P1	EP: MockitoUsage
An assertion checks the equality of an object itself	R4	S5	P1	EP: JunitAssertSameCheck
There is no @Test(expected=…) annotation for a test method which is to check whether a target method throws an exception correctly	R4	S5	P1	PMD: JUnitUseExpected
A test case or test suite creates a Table or Sequence at DBMS, and then does not drop it before the termination.	R4	S6	P1,P2	DERBY-4393, DERBY-5174, HBASE-8122
A test method opens a socket, and terminates without close()	R4	S6	P1	HDFS-7282
A test case opens a file with a new InputSteam object, however the test case does not close the file when an exception occurs in a middle.	R4	S6	P1	JCR-267
A test method occupies a mutually-exclusive shared resource and does not return the acquired resource before termination.	R4	S6	P2	DERBY-2708
A test case assigns the reference of a new object to a shared variable and does not delete it before the termination (memory leak).	R4	S6	P2	DERBY-5717
A test method holds a mutex and then terminates without releasing the mutex	R4	S6	P2	HDFS-7109
An assertion is executed by a thread spawned by a test method	R4	S7	P1	FB: IJU_ASSERT_METHOD_INVOKED_FROM…
An assert statement meaninglessly checks a condition over constants	R4,R5	S5	P1	EP: TruthConstantAsserts, FBC: ACTUAL_CONSTANT
An assert checks the equality of two objects of different types	R4,R6	S5	P1	EP: TruthIncompatibleType
No fail() exist after a test target method call which is expect to throw an exception	R5	S4	P1	MAPREDUCE-5421, DERBY-3852, DERBY-6088, JCR-500
assertionEquals() is not given with a tolerance when it checks the equality of two floating-point numbers	R5	S5	P1	EP: JUnit3FloatingPointComparison WithoutDelta
fail() exists in a try-block while one of its corresponding catch-block catches an assertionError() exception	R5	S5	P1	EP: AssertionFailureIgnored
Multiple statements may throw an Exception that is expected to be thrown by a test target method	R7	S5	P1	JCR-498
assertTrue() checks whether two objects are equivalent using equals()	R6	S5	P1	PMD: UseAssertEqualsInsteadOfAssertTrue, FBC: USE_ASSERT_EQUALS
assertTrue() checks whether an object is null or not	R6	S5	P1	PMD: UseAssertNullInsteadOfAssertTrue, FBC: USE_ASSERT_NULL

표 2는 요구사항 요소-실행 과정 요소-코드영역 요소의 조합(1-3열)을 범주로 하여, 각 범주에 해당하는 조사대상 결함 검출기의 개수(4열)와 조사대상 실제 결함 사례 개수(5열)을 보고한다. 이 때, 하나의 조사대상이 같은 범주에 속한 복수 개의 결함 요소와 연관이 있는 경우, 표 2에서는 각각의 경우를 개별적으로 계수하였다. 표 2는 84개 조사대상 결함을 총 33개의 구성 요소 조합으로 분류하고 있으며, 이 때 각 조합에 평균 2.7개(최소 1개, 최대 14개)의 조사 대상이 대응된다.

표 2에서 결함 검출기(4열)와 실제 결함 사례(5열) 간의 관계를 살펴보면, 총 33개 범주 중 총 29개의 범주에서는 결함 검출기만 대응되거나(예: R1-S1-P1) 혹은 실제 결함 사례만 대응되고 있음(예: R2-S2-P2)을 알 수 있다. 표 2의 분류를 바탕으로, 같은 범주로 속한 조사대상 결함 검출기와 실제 결함 사례 중 동일한 결함 패턴에 해당하는 지를 추가로 조사한 결과, 조사대상 실제

표 2 테스트 요소 조합에 따른 조사 대상 결함 검출기와 실제 결함 사례의 수

Table 2 The number of studied bug checkers and real test bug cases per classification category

Category			Checker	Bug
R1	S1	P1	1	0
R1	S1	P2	3	1
R1	S1	P4	3	1
R1	S2	P1	1	0
R1	S2	P3	1	0
R1	S2	P4	3	0
R1	S2	P1	1	0
R1	S3	P1	1	0
R1	S6	P1	1	0
R1	S6	P3	1	0
R1	S6	P4	2	0
R2	S2	P2	0	1
R2	S2	P3	0	1
R2	S4	P1	1	1
R2	S6	P1	0	1
R3	S1	P2	0	1
R3	S2	P1	0	6
R3	S2	P2	0	1
R3	S4	P1	0	4
R3	S5	P1	0	2
R3	S6	P1	0	4
R3	S6	P2	0	2
R4	S2	P1	0	1
R4	S4	P1	0	1
R4	S5	P1	13	1
R4	S6	P1	0	6
R4	S6	P2	0	5
R4	S7	P1	1	0
R5	S4	P1	0	5
R5	S5	P1	4	0
R6	S4	P1	0	3
R6	S5	P1	5	0
R7	S4	P1	0	1

결함을 조사대상 결함 검출기를 통해 검출할 수 있는 경우는 1건 (HDFS-725를 FindBugs의 IJU_SETUP_NO_SUPER로 검출할 수 있음)으로 매우 제한적임을 관찰할 수 있었다. 반면, 같은 범주로 분류된 서로 다른 결함 검출기가 실제로 동일한 결함 패턴에 대응되는 경우는 총 11건으로 상대적으로 빈번하게 발견되었다.

범주별로 연관된 결함 검출기와 실제 결함 사례의 수를 살펴보면, R3(상이한 환경에서도 동일한 테스트 코드는 동일한 테스트 동작을 실행해야 함)에 해당하는 결함 사례는 빈번히 보고되고 있는 반면, 이에 해당하는 결함을 검출하는 결함 검출기는 없음을 알 수 있다. 또한 R4(단위 테스트 코드의 실행은 상호 간 영향을 받지 않아야 함)에 연관된 결함 사례는 많으나, 이에 연관된 결함 검출기는 R4-S5-P1의 특정한 조건에 국한되어 개발되어 있음을 확인할 수 있다.

이러한 관찰을 바탕으로, 실제 프로젝트에서 발생하는 테스트 결함과 현재 존재하는 오픈소스 결함 검출기의 검출 대상에 차이가 있음을 알 수 있다. 따라서 실제 테스트 코드에서 발생하는 실제 테스트 결함 패턴을 효과적으로 탐지하는 새로운 결함 검출기의 개발이 요청된다(4장 참고).

마지막으로, R7(테스트 실행 결과가 사용자에게 올바르게 전달되어야 함), S7(사용자에게 테스트 결과 보고), P4(테스팅 프레임워크 인터페이스 부위) 범주에 속한 결함 검출기와 결함 사례는 비교적 적게 조사되었는데, R7, S7, P4 모두 테스트 결과 확인과 연관된 범주임을 관찰할 수 있다.

4. 단위 테스트 코드 결함 패턴

본 장에서는 실제 테스트 결함 사례를 바탕으로 정의한 8종의 새로운 Java 테스트 결함 코드 패턴을 소개한다. 4.1절에서 4.8절은 각 테스트 결함 패턴의 탐지 대상 결함을 설명하고, 이를 포착하는 코드 구조 조건을 상세히 소개한다. 본 논문에서 소개하는 8종의 테스트 결함 패턴은 기존의 결함 검출 도구나 관련 연구에서 제시되지 않은 새로운 결함 패턴이다. 4.9절에서는 앞서 소개한 8개 결함 패턴을 FindBugs 프레임워크에서 정적 결함 검출기로 어떻게 구현하였는지 간단히 설명한다. 본 연구에서 구현한 8개 테스트 결함 검출기의 소스코드는 공개 소프트웨어 저장소를 통해 공개하여 일반에 활용이 가능하도록 하였다[1].

4.1 패턴1: Exception 발생 실패를 탐지하는 테스트 오라클이 없음

4.1.1 검출 대상 결함

테스트 대상 메소드가 의도대로 Exception을 발생시키는지 검사하는 테스트 케이스에서, 테스트 오라클의 부족으로 인하여, 테스트 대상 메소드가 Exception을 발생시키지 못한 오류 상황을 테스트 실패로 인식하지 못하는 결함이다. 이러한 결함은 실제 테스트 실패 상황을 테스트 성공으로 잘못 인식하는 오류 미탐지(false negative) 문제를 야기한다.

그림 3은 본 패턴이 검출하고자 하는 결함의 실제 사례인 DERBY-3852을 간략히 묘사한 코드 일부분이다. testDSRequestAuthentication는 테스트 대상 메소드인 ds.getConnection(5행)가 특정한 상황에 SQLException을 발생시키고(6행), 그 때 오류 메시지가 정확한 지(“5XJ015”) 검사하는(7행) 테스트 케이스다. 이 테스트 케이스는 ds.getConnection이 해당 테스트 실행 중 SQLException을 발생시키지 않는 오동작이 발생하는 경우에는 테스트 메소드가 정상 종료되어(5행), 테스트 성공으로 잘못 인식되게 된다. 이러한 문제를 해결하기 위해 DERBY-3852는 ds.getConnection이 반환되는 경우(5행 직후)에 테스트 오라클 구문인 fail을 삽입하여, 해당 상황이 오류로 인식되도록 고쳐졌다.

```
1 void testDSRequestAuthentication()
2 {
3   ...
4   try {
5     ds.getConnection();
6   } catch(SQLExecption sqle) {
7     assertSQLState(“XJ015”, sqle);
8   }
9 }
```

그림 3 DERBY-3852 결함의 예

Fig. 3 Buggy code of DERBY-3852

1) http://github.com/hansolchoe/FindTestBugs

4.1.2 결함 코드 패턴

그림 4는 패턴 1을 포착하기 위한 두 가지 경우의 코드 구조 조건을 묘사하고 있다. 그림 4(a)는 테스트 대상 메소드가 Exception을 발생시킬 수 있다고 것으로 선언되지 않은 경우(throws)에 해당하며, 그림 4(b)는 Exception을 발생시킬 수 있다고 선언된 경우에 해당한다. 그림 4(a)는 테스트 메소드의 body-block 내 마지막 구문(3-9행)이 E 타입의 Exception을 처리하는(7행) try-catch 구문일 경우, 이를 E 타입 Exception 발생을 검사하는 테스트 메소드로 판별한다. 그리고 해당 try-catch 구문의 try-block 내 마지막 구문이 E 타입의 Exception을 발생시키는 메소드 m을 호출하는 경우(5행)를 패턴1에 해당하는 결함으로 탐지한다. 이는, m이 정상적으로 반환되는 실행상황을 오류로 식별하는 테스트 오라클(예: fail)이 없기 때문이다.

그림 4(b)는 테스트 메소드가 E 타입의 Exception을 발생시킬 수 있는 경우(1행)에 대한 패턴이다. 본 패턴은 테스트 메소드의 body-block 내 마지막 구문이 E 타입 Exception을 발생시킬 수 있는 메소드 m의 호출인 경우(3행)를 결함으로 포착한다. 이 때, 메소드 m은 Exception 발생이 기대되는 대상 메소드로 추정되는데, m이 의도와 다르게 Exception을 발생시키지 않는 경우를 오류로 인식하는 테스트 오라클(예: fail)이 없는 상황이다.

```
1 test() {
2   ...
3   try {
4     ...
5     m(); // m may throw an exception of type E
6   } catch(SQLExecption sqle) {
7     assertSQLState("XJ015", sqle);
8   }
9 }
10
```

(a) Case for test methods not throwing any exception

```
1 test() throws E {
2   ...
3   m(); // m may throw an exception of type E
4 }
```

(b) Case for test method that may throw an exception

그림 4 결함 패턴1의 구문 패턴

Fig. 4 Code Patterns for Bug Pattern 1

4.2 패턴2: Exception 발생을 검사하는 try-block 내 복수의 Exception 발생 메소드 호출이 존재

4.2.1 검출 대상 결함

테스트 대상 메소드에서 발생하는 특정한 타입의 Exception을 감지하기 위해서 테스트 메소드를 Exception을 발생시킬 수 있는 타입으로 선언한 경우(i.e., throws), 테스트 대상 이외의 다른 메소드가 해당 타입의 Exception을 발생시키는 실행에 대해서 테스트 메소드가 종료되게 되는데, 이로 인해 의도치 않게 테스트가 중단되거나 테스트 결과가 오인될 수 있다. 실제 테스트 결함 사례인 JCR-498은 테스트 대상 메소드에서 발생할 수 있는 Exception을 전달하기 위해 Exception 발생 가능 메소드로 선언된 테스트 메소드에서 보고된 결함으로, 테스트 대상 메소드가 실행되기 전 테스트 입력 설정 명령에서 예외적인 상황으로 인해 Exception이 발생할 경우, 의도치 않게 테스트 메소드가 종료(Exception 전달)되는 오류가 발생한다. JCR-498에서는 문제가 되는 테스트 입력 설정 명령에 별도의 try-catch 구문을 추가해 예외적인 상황에도, 테스트가 올바르게 진행되도록 테스트 케이스를 수정하여 문제를 해결하였다.

4.2.2 결함 코드 패턴

그림 5에서 묘사한 바와 같이, 패턴2에 대한 코드 구조 조건은 테스트 시나리오로 Exception 발생을 의도하지 않은 경우 (그림 5(a))와 Exception 발생을 의도한 경우(그림 5(b))의 두 가지 경우로 나누어 정의하였다.

그림 5(a)는 E 타입의 Exception을 전달할 수 있다고 선언된(1행) 테스트 메소드로서 try-block의 마지막 구문이 E 타입의 Exception을 발생시키는 테스트 대상 메소드이면서(7행), 해당 try-block 내의 다른 구문에서도 E타입의 Exception이 발생 가능한 경우(5행)를 결함으로 포착한다. 이러한 패턴의 테스트 케이스에서는 pre() 에서 Exception이 발생할 경우, 의도치 않게 테스트가 종료되는 오류가 발생할 수 있다.

```
1 test() {
2  ...
3  try {
4   ...
5   pre(); // pre may throw an exception of type E
6   ...
7   target(); // target may throw an exception of type E
8   fail();
9  } catch (E e) {
10   ...
11  }
12 }
```

(a) Case for the test target method (i..e., target) that may throw an exception of type E

```
1 test() {
2   ...
3   try {
4     ...
5     m(); // m may throw an exception of type E
6   } catch(SQLExecption sqle) {
7     assertSQLState("XJ015", sqle);
8   }
9 }
10
```

(b) Case for the test target method (i.e., target) that throws an exception of type E as the correct behavior

그림 5 결함 패턴2의 구문 패턴

Fig. 5 Code Patterns for Bug Pattern 2

그림 5(b)는 정상동작으로 E 타입의 Exception을 의도한 테스트 케이스의 경우, 즉 해당 메소드가 Exception을 전달할 수 있는 타입으로 선언되었고(1행)과 메소드의 body-block의 마지막에 fail 구문(6행)이 존재할 때, 해당 테스트 메소드의 body-block에 E 타입의 Exception을 발생시키는 구문이 2개 이상 존재하는 경우(3행, 5행)를 결함으로 탐지한다. 이러한 패턴의 테스트 케이스의 경우, 앞서 실행되는 메소드 호출(3행)에서 의도치 않게 Exception이 발생하는 경우를 테스트 성공으로 잘못 인식하는 오류 미탐지 문제가 발생할 수 있다.

4.3 패턴3: TestCase 클래스에서 setUp을 정의했으나 tearDown을 정의하지 않음

4.3.1 검출 대상 결함

JUnit의 TestCase 클래스의 경우, setUp과 tearDown을 각각 정의함으로써 매 테스트 케이스 실행 직전과 직후에 테스트실행에 필요한 자원을 사용 전에 할당하고, 또한 사용이 끝난 자원을 해제하는 하는 작업을 정의할 수 있다. 이 때 특정 TestCase 클래스에서 자원을 setUp 메소드를 정의한 반면 이에 대응하는 tearDown을 정의하지 않을 경우, 필수적인 자원 해제 작업이 올바르게 시행되지 않아, 테스트 케이스 간 의존성 문제 등이 발생할 위험이 있다.

그림 6은 HBASE-8122에 해당하는 결함 사례를 간략히 표현한 코드로, JUnit4 환경에서 TestCase 클래스의 실행 전에 공유 자원의 설정하는 setUp 메소드인 일종인 setupBeforeClass에서 특정 자원 설정을 수행하지만(5행), 이후 이를 해제하지 않는 문제를 가지고 있다. HBASE-8122에서는 그림 6의 주석(8-11행)이 설명하는 바와 같이, tearDown에 해당하는 tearDownAfterClass를 정의하여 자원 해제를 명령(11행)을 하도록 수정했다.

```
 1 public class TestAccessController {
 2 @BeforeClass
 3 void setUpBeforeClass() {
 4 ...
 5 TEST_UTIL.startMiniCluster();
 6 ...
 7 }
 8 // FIX
 9 // @AfterClass
10 // void tearDownAfterClass
11 // TEST_UTIL.shutdownMiniCluster();
12 // }
```

그림 6 HBASE-8122 결함의 예
Fig. 6 Buggy code of HBASE-8122

4.3.2 결함 코드 패턴

그림 7의 결함 패턴은 JUnit의 TestCase 혹은 TestSetup을 확장하여(1행) 구현한 테스트 클래스의 정의를 검사하여 테스트 클래스의 setUp 메소드는 정의(overriding) 하였으나(2행), tearDown은 정의하지 않은 경우를 결함으로 탐지한다.

```
1 class Test extends TestCase {
2 setUp(){
3 ...
4 }

5 // tearDown() {
6 //
7 // }
8 }
```

그림 7 결함 패턴3의 구문 패턴
Fig. 7 Code Patterns for Bug Pattern 7

4.4 패턴4: TestCase setUp에서 공유변수에 저장한 참조를 tearDown에서 제거하지 않음

4.4.1 검출 대상 결함

본 패턴의 검출대상은 JUnit의 TestCase 클래스의 setUp에서 공유 자원의 레퍼런스(참조)를 공유 변수(멤버 변수)에 저장한 경우, tearDown에서 해당 참조를 제거하여 향후 공유 자원의 메모리 해제(garbage collection)가 이루어질 수 있도록 해야 하나, 이러한 참조 제거 명령을 tearDown에서 수행하지 않는 결함이다. JUnit의 TestCase 객체가 테스트 실행을 완료한 후에도 메모리 상에 남아 있을 경우, 멤버 변수에 저장된 참조가 공유 자원의 해제가 불가능한 상태가 되므로 메모리 과사용이나, 해당 객체와 연관된 자원의 미해제에 따라 다른 테스트 케이스 실행에 의도하지 않은 간섭을 일으키는 문제가 발생할 수 있다.

그림 8은 해당 패턴의 일종인 DERBY-5717 결함을 간략히 표현한 코드다. setUp에서는 각 테스트 케이스를 매번 실행할 때 마다 _databaseMetaData에 getMetaData()를 통해 얻은 DBMS 와의 연결에 관련된 객체의 참조가 대입되나(3행), tearDown에서 이를 제거하지 않아 테스트 케이스 실행 후에도 해당 데이터베이스 연결에 관련 자원이 해제되지 않는 문제가 보고되었다. 해당 결함은 tearDown에 _databaseMetaData를 삭제하도록 수정되었다(9행).

4.4.2 결함 코드 패턴

본 패턴은 그림 9에서 설명하는 바와 같이, setUp에서 해당 TestCase 클래스 멤버 필드에 참조 값을 정의하는 구문(5행)이 있는 반면 tearDown에는 해당 멤버 필드에 null 값을 삽입하는 구문이 없는 경우(9행)를 결함으로 탐지한다.

```
1 void setUp() {
2   super.setUp();
3   _databaseMetaData = getConnection.getMetaData();
4   dropSchema();
5 }

6 void tearDown() {
7   dropSchema();
8   // Fix: _databaseMetaData = null;
9 }
```

그림 8 DERBY-5717 결함 코드
Fig. 8 Buggy code of DERBY-5717

```
 1 class Test extends TestCase {
 2 T ref;
 3 setUp() {
 4 ...
 5 ref = ...;
 6 ...
 7 }
 8 tearDown() {
 9 // ref is not defined as null
10 }
11 }
```

그림 9 결함 패턴4의 구문 패턴
Fig. 9 Code pattern for Bug Pattern 4

4.5 패턴5: 테스트 케이스에서 생성한 외부 자원을 적합하게 제거하지 않음

4.5.1. 검출 대상 결함

테스트 케이스의 일시적인 사용을 위해 생성한 파일이나 데이터베이스 테이블을 해당 테스트 케이스가 종료되기 전에 적절하게 제거하지 않을 경우, 의도하지 않은 테스트 케이스 간 간섭 등이 발생하여 올바른 테스트 수행에 문제가 될 수 있다.

DERBY-5174는 테스트 케이스가 테스트 시나리오의 일부로 외부 DBMS에 테이블을 생성하여 사용하고 이를 제거하지 않는 결함을 보고하고 있다. 이 결함은 테스트 실행 이후 외부 환경을 의도하지 않은 방식으로 변형시키는 문제, 혹은 이러한 변형을 통해 같은 DBMS를 사용하는 테스트 케이스 간에 의도치 않은 간섭 문제를 일으킬 수 있다.

4.5.2 결함 코드 패턴

본 패턴은, 그림 10에서 나타내고 있는 바와 같이, 테스트 케이스의 setUp 혹은 테스트 메소드에서 외부 자원을 생성하고 있으나(3행, 8행), 테스트 메소드 혹은 tearDown에서 해당 자원을 삭제하는 명령이 없는 경우(10행, 14행)를 결함으로 인식한다. 구체적으로, 그림 10에서 create으로 표현된 외부 자원 생성 명령은 Closable 인터페이스를 구현한 파일 관련 객체의 생성자 혹은 CREATE TABLE 스키마를 인자로 받는 Statement.executeQuery 명령으로 정의할 수 있으며, 이에 대응하여 remove는 Closable 인터페이스를 구현한 객체의 close 메소드 호출, 혹은 DROP TABLE 스키마를 인자로 받는 Statement.executeQuery 호출로 정의할 수 있다.

```
 1 setUp() {
 2 ...
 3 create();
 4 ...
 5 }

 6 test() {
 7 ...
 8 create();
 9 ...
10 // remove() must not exist in test()
11 }

12 tearDown() {
13 ...
14 // remove() must not exist in test()
15 }
```

그림 10 결함 패턴5의 구문 패턴
Fig. 10 Code pattern for Bug Pattern 5

4.6 패턴6: 테스트 케이스가 쓰레드를 개시한 후 종료 전에 Join하지 않음

4.6.1 검출 대상 결함

테스트 케이스가 쓰레드를 생성하여 개시(Thread.start)한 경우, 테스트 케이스 종료 전에 쓰레드 객체에 Thread.join 명령을 호출하여 종료되도록 동기화해야 한다. 만약 이와 같은 동기화가 적절히 이루어지지 경우, 테스트 케이스는 종료된 반면 생성된 쓰레드가 계속 실행되는 상태로 남아 있을 수 있다. 특히, JUnit 등 프레임워크를 통해 여러 개의 테스트 스위트를 연속으로 실행하는 상황에서는, 개별 테스트 케이스의 실행이 종료되더라도 프로세스 전체가 종료되는 것이 아니기 때문에, 종료되지 않은 쓰레드가 계속해서 실행될 수 있다

이러한 이상 실행의 경우, 쓰레드와 관련된 자원이 적절히 해지되지 않거나, 쓰레드 실행이 다른 테스트 케이스의 실행에 의도하지 않은 간섭을 일으키거나, 혹은 쓰레드 실행에서 발견된 오류 증상을 테스트 케이스가 탐지하지 못하는 문제를 발생시킬 수 있다. DERBY-5708에서는 테스트 케이스가 생성한 쓰레드가 잘못된 동기화로 인해 정상종료 되지 않아 자원 과사용이 발생하는 문제가 있었다.

4.6.2 결함 코드 패턴

본 결함 패턴은, 그림 11에서 설명하고 있는 바와 같이, 테스트 케이스에서 Thread 클래스의 객체(혹은 Thread의 하위 클래스의 객체)를 생성한 후(3행), 동일한 테스트 케이스에서 생성한 Thread 객체에 대한 Thread.join 명령이 없는 경우(5행)의 코드를 결함으로 정의한다.

4.7 패턴7: 파일 경로에 플랫폼-종속적인 디렉토리 이름 구분 문자 사용

4.7.1 검출 대상 결함

파일 경로에서 디렉토리 이름을 '/' 혹은 '\'로 구분할 경우, 테스트 코드가 실행되는 운영체제에 따라 올바르게 인식되지 않는 문제를 발생시킬 수 있다. 이와 같은 결함은 디렉토리 구분자로 특정 문자열이 아닌 File.separatorChar 값을 사용하여 플랫폼 독립적인 동작이 가능하도록 함으로써 해결할 수 있다. 그림 12는 MAPREDUCE-5259로 파일 경로를 비교하는 과정에서 '/' 경로 구분자를 사용하여, 해당 테스트 케이스가 Windows 운영체제에서는 올바르게 동작하지 않는 문제가 발생한 실제 사례다.

```
1 test() {
2 ...
3 Thread t = new Thread();
4 ...
5 /* t.join() must not eixist in test() */
```

그림 11 결함 패턴 6의 구문 패턴
Fig. 11 Code pattern for Bug Pattern 6

```
1 void testTaskLog() {
2   ...
3   File f = TaskLogFile(...);
4   ...
5   assertTrue(f.getAbsolutePath().
6     endsWith("test/stdout");
7 }
```

그림 12 MAPREDUCE-8589 결함의 예
Fig. 12 Buggy code of MAPREDUCE-8589

```
1 test() {
2   ...
3   File f = new File("/path/to/file");
4   ...
5   assert(path, f.getPath());
6 }
```

그림 13 결함 패턴 7의 구문 패턴
Fig. 13 Code pattern for Bug Pattern 7

4.7.2 결함 코드 패턴

본 결함 코드 패턴은, 그림 13과 같이, 테스트 케이스에서 파일과 연관된 클래스(예: File, FileReader, FileWritter 혹은 그 하위 클래스) 객체 혹은 해당 객체의 멤버의 메소드를 호출하는 경우, 인자로 전달되는 문자열에 '/' 혹은 '\'가 포함될 때 이를 결함으로 검출한다.

4.8 패턴8: Maven이 인식하지 못하는 테스트 클래스명 사용

4.8.1 검출 대상 결함

오늘날 널리 사용되는 빌드 시스템인 Maven은 테스트 실행 시, 프로젝트 내의 Java 클래스 중에서 특정 패턴에 해당하는 클래스를 테스트 클래스로 인식하여 실행한다. Maven이 인식하는 테스트 클래스명 패턴은 pom.xml로 정규식(regular expression)으로 개발자가 명시하거나, 개발자의 명시가 없는 경우 클래스명이 "Test"로 시작하거나 끝나는 경우를 테스트 클래스로 인식한다. 이 때, 테스트 케이스가 작성된 클래스의 이름이 Maven 규칙에 맞지 않을 경우, 테스트 클래스로 인식되지 않아, 개발자의 의도와 다르게 테스트 실행이 이루어지지 않는 문제가 발생할 수 있다. SLIDER-41은 실제 이와 같은 결함 사례에 대한 결함 보고로 pom.xml에는 테스트 클래스명 패턴이 Test로 시작하는 경우로 기술되어 있으나, 개발자가 테스트 클래스의 이름을 SliderUtilsTest으로, 정의한 규칙에 맞지 않아, Maven을 통해 해당 테스트 케이스가 실행되지 않는 문제가 발생한 사례다.

```
1 public class SomeTestCases extends TestCase {
2   ...
3   test() {
4     ...
5   }
6 }
```

그림 14 결함 패턴 8의 구문 패턴
Fig. 14 Code pattern for Bug Pattern 8

4.8.2 결함 코드 패턴

본 패턴에서는 테스트 클래스로 구현된 클래스를 식별하고, 해당 클래스의 이름이 Maven 인식 규칙에 맞지 않는 경우를 결함으로 검출한다. 특정 클래스가 테스트 클래스로 의도된 경우는 해당 클래스가 JUnit의 TestCase 클래스를 상속한 하위 클래스인 경우, 혹은 클래스 이름에 "Test"가 부분문자열로 포함된 경우를 대상으로 하였다. 이러한 클래스의 이름이 pom.xml에 기술된 테스트 클래스명 정규식 혹은 그림 14와 같이 기본 패턴("Test"로 시작하거나 끝나는 경우)에 맞지 않는 클래스를 본 패턴은 결함으로 인식한다.

4.9 논 의

4.9.1 결함 검출기 구현

본 연구에서는 앞서 소개한 8종의 테스트 결함 패턴을 JUnit 3 테스트 코드에 대해 동작하는 정적 결함 검출기로 구현함으로써, 각 코드 구조 조건이 구체적인 결함 검출기 개발에 활용될 수 있음을 확인하였다.

앞서 소개한 8개 패턴에 대한 결함 걸출기는 모두 FindBugs의 BytecodeScanningDetector 클래스를 활용하여, 검증 대상 Java 프로그램의 바이트코드 명령의 구문 구조를 탐색하며 특정 구문 조건을 검사하는 구조로 구현하였다. 기본적으로, 모든 결함 검출기는 특정 클래스가 JUnit 3의 TestCase 클래스를 상속하였는지 여부와 특정 메소드의 이름이 "test"로 시작하는 지를 검사함으로써 테스트 메소드를 인식한다.

패턴 1과 패턴 2에 대한 결함 검출기의 경우, 테스트 메소드 내에 Exception 테이블과 이와 관련된 메소드 호출 명령어를 검사하여, 해당 구조 조건 검사를 수행한다. 패턴 3에 대한 결함 검출기는 TestCase 클래스를 상속받은 클래스에 setUp 및 tearDown 메소드가 정의되어 있는 지 유무를 확인함으로써 결함 구문 조건을 검사한다. 패턴 4와 패턴 7에 대한 결함 검출기는 검증 대상 TestCase 클래스의 멤버 변수를 파악한 후, 각 멤버 변수에 대해 해당 TestCase 클래스 내에 특정한 명령이 존재하는 지를 확인함으로써, 결함 구문 조건을 검사한다. 패턴 6의 경우, 테스트 메소드가 직접 Thread.start() 호출 명령을 포함하거나 혹은 테스트 메소드가 호출하는 다른 메소드에 Thread.start() 명령이 존재하는 지 검사하는 방식으로 작동한다. 마지막으로, 패턴 5와 패턴 8에

해당하는 결함 검출기의 경우, 특정한 메소드 호출에 주어지는 입력 값을 파악한 후, 해당 입력 값에 대한 조건을 검사함으로써 결함을 검출하도록 구현하였다. 예를 들어, 패턴 8에 대한 결함 검출기의 경우는 테스트 메소드를 구성하는 바이트코드 명령으로 File, FileWriter의 생성자의 입력으로 주어지는 문자열 상수에 '/' 혹은 '\'이 포함된 경우를 결함으로 검출하도록 구현하였다.

4.9.2 제안한 결함 패턴의 한계

본 연구에서 제안한 결함 패턴은 구체적인 결함 사례로부터 특징적인 결함 조건을 유추한 사례로, 각 패턴이 실제 결함과 높은 연계성(precision)을 갖도록 정의된 반면, 일반적인 테스트 결함 패턴의 전체를 포괄하지 못한다는 한계를 가진다. 보다 많은 테스트 결함을 포괄하기 위해서는, 실제 소프트웨어 프로젝트에서 발생하는 다양한 유형의 테스트 결함을 지속적으로 분류하고 같은 분류 내의 결함을 일반화하여 패턴을 정의하는 연구가 필요할 것으로 보인다.

앞서 소개한 테스트 결함 패턴은 의사코드와 이에 대한 부연 설명으로 정의되어 있으므로, 이를 해석함에 있어서 모호성이 있을 수 있다는 한계점이 있다. 반면, 의사코드와 같은 추상적인 기술을 통해, 본 논문에서 정의한 패턴이 특정 단위 테스트 라이브러리에 국한되지 않고, 일반적인 Java 단위 테스트 코드에 대해 활용될 수 있도록 하였다.

이와 같이, 본 논문이 제안한 테스트 결함 패턴은 포괄성과 정확성에서 한계를 가지고 있으며 이를 보완하기 위해서는 보다 많은 테스트 결함 사례에 대한 조사와 분석이 필요하다. 이러한 개선을 위하여, 본 논문은 앞서 테스트 결함 분류 체계(3장)를 제시하여 단위 테스트 결함 사례를 체계적으로 수집하고 분류하는 방법론을 제시하였다. 또한, 본 장에서 상술한 패턴 정의 방법이 향후 테스트 결함 패턴 확충에 있어서 결함 패턴을 정의하는 방법을 보이는 사례로 활용될 수 있으리라 기대한다.

5. 관련연구

단위 테스트 코드에 대한 연구로 실제 소프트웨어 개발 과정에서 관찰된 단위 테스트 결함에 대한 보고에 대한 조사와 특정한 유형의 단위 테스트 결함을 검출하기 위한 기법에 대한 연구가 이루어져왔다. Vahazadeh 등[1]은 오픈소스 프로젝트 저장소의 결함 리포트를 조사하여 수집한 단위 테스트 결함 사례의 수집을 소개하며, Luo 등[14]의 연구에서는 테스트 결함 중 반복되는 실행 중 테스트 결과가 달라지는 'flaky test'에 대한 조사 결과를 소개한다.

본 연구와 유사하게, Vahazadeh 등[1]의 논문에서는 16개의 카테고리로 테스트 결함을 분류하는 체계를 소개하며, 각 분류에 따라 대표적인 결함 사례를 소개한다. Vahazadeh 등[1]이 특정 사례를 중심으로 분류 기준을 체계적으로 제시하는 데 한계를 가지는 것과 달리, 본 논문에서 제안하는 분류 체계는 하나의 테스트 결함을 구조적 요소, 기능정 요소, 의미적 요소의 3가지 관점에서 조합적으로 분류할 수 있는 기준을 제시함으로써, 테스트 결함에 대한 체계적인 이해와 분류가 가능하도록 하였다.

테스트 결함 검출 기법으로는 테스트 종속성으로 인한 결함을 검출하는 동적 분석 기법[3,5]이 소개되었다. 일반적인 테스트 결함에 대한 검출 방법으로는 Ramler 등[15]은 특정 소프트웨어 프로젝트 중에 관찰된 테스트 결함 사례를 바탕으로 빈번히 발생하는 테스트 결함 유형을 패턴으로 정의한 사례연구가 발표되었으나, 구체적인 결함 패턴의 정의는 소개되지 않아 활용에 한계가 있다.

면밀한 정적/동적 분석의 적용이 어려운 복잡한 대규모 코드로부터 결함 검출을 효과적으로 지원하는 방안으로 코드 패턴 매칭을 통한 결함 검출에 대한 연구가 개발되어 왔다[6,16,17]. FindBugs[6]는 Java 프로그램의 바이트코드에서 결함에서 특징적으로 관찰되는 패턴을 포착하여 Null pointer deference 등 일반적인 프로그램 결함 검출기[17]는 물론, 보안 약점을 발생시키는 라이브러리 사용 패턴[18] 등 특정 도메인에 특화된 패턴 기반 결함 검출기를 개발하는 방법론을 제안하였다[19]. COBET은 패턴 기반 결함 검출 방법론을 멀티쓰레드 프로그램에 적용한 기술로, 멀티쓰레드 프로그램에서 발생하는 동시성 결함의 구조적, 의미적 특징을 패턴으로 효과적으로 기술할 수 있는 프레임워크를 제공한다[20].

특정 프로그램 버전에 존재하는 코드 패턴으로 결함을 포착하는 방식 이외에 버전 간 코드 변화에서 결함을 유발하는 코드 수정 패턴을 포착하는 기법도 개발되어 왔다. Kim 등의 연구에서는 결함을 유발하는 코드 변화를 조사하여[21], 결함을 유발하는 코드 변화를 예측하고[22], 또한 이를 패턴으로 정의하여 결함 수정 등에 활용하는 기법[23]을 제안하였다.

6. 결 론

본 논문은 Java 프로그램 단위 테스트 코드의 구조요소별 범주를 정의하여 이를 바탕으로 단위 테스트 코드에서 발생하는 결함을 분석/분류하는 체계를 제안하였다. 또한, 제안한 분류 체계를 이용해 45개의 오픈소스 정적 결함 검출기의 결함 패턴과 39개의 실제 결함 사례를 분류하고 그 연관성을 체계적으로 비교하였다. 또한, 본 논문에서는 실제 테스트 결함 사례로부터 테스트 결함의 특징적 조건을 기술한 8개의 새로운 테스트 결함 패턴을 구체적으로 정의하였으며, 이를 바탕으로 정적 결함 검출기를 구현한 내용을 소개한다. 향후에는,

테스트 결함 문제의 효과적이고 정확한 탐지를 위해서 테스트 케이스 대상 동적 분석(예: 테스트 코드의 커버리지 정보)을 활용하는 연구와 Java 이외의 다른 프로그래밍 언어, 다른 테스트 플랫폼 상에서 발생하는 테스트 결함에 대한 연구를 진행할 계획이다.

References

[1] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An Empirical Study of Bugs in Test Code," *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 101-110. 2015.

[2] H. Choe, and S. Hong, "A Classification of Unit Test Bugs in Java Programs," *Korea Computing Congress (KCC)*, pp. 493-495, 2018. (in Korean)

[3] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient Dependency Detection for Safe Java Test Acceleration," *Joint Meeting of the European Software Engineering Conference on the ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE)*, pp. 770-781, Aug. 2015.

[4] S. Zhang, D. Jalali, J. Wutkke, K. Muslu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically Revisiting the Test Independence Assumption," *International Symposium on Software Testing and Analysis (ISSTA)*, pp. 385-396, Jul. 2014.

[5] A. Gambi, J. Bell, and A. Zeller, "Practical Test Dependency Detection," *International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1-11, Apr. 2018.

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

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

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

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

[10] E. Farchi, Y. Nir, and S. Ur, "Concurrent Bug Patterns and How to Test Them," *International Symposium on Parallel and Distributed Processing*, pp. 286, Apr. 2003.

[11] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real-World Concurrency Bug Characteristics," *International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, Vol. 36, No. 1, pp. 329-339, Mar. 2008.

[12] Z. Yin, M. Caesar, and Y. Zhou, "Toward Understanding Bugs in Open Source Router Software," *ACM SIGCOMM Computer Communication Review*, Vol. 40, No. 3, Jul. 2010.

[13] J. M. Voas, "PIE: A Dynamic Failure-Based Technique," *IEEE Transactions on Software Engineering (TSE)*, Vol. 18, No. 8, pp. 712-727, Aug. 1992.

[14] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An Empirical Analysis of Flaky Tests," *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 643-653, Nov. 2014.

[15] R. Ramler, M. Moser, and J. Pichler, "Automated Static Analysis of Unit Test Code," *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 2, pp. 25-28, Mar. 2016.

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking System Rules using System Specific, Programmer-written Compiler extensions," *Conference on Symposium on Operating System Design and Implementation (OSDI)*, Vol. 4, pp. 1, Oct. 2000.

[17] D. Hovemeyer, and W. Pugh, "Finding More Null Pointer Bugs, But Not Too Many," *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pp. 9-14, Jun. 2007.

[18] National Vulnerability Database, [Onlile]. Available: https://nvd.nist.gov/vuln/categories

[19] Find Security Bugs, [Onlile]. Available: https://find-sec-bugs.github.io

[20] S. Hong and M. Kim, "Effective Pattern-driven Concurrency Bug Detection for Operating Systems," *Journal of Systems and Software (JSS)*, Vol. 86. No. 2, pp. 377-388, Feb. 2013.

[21] S. Kim, K. Pan, and E. E. Whitehead Jr, "Memories of Bug Fixes," *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 35-45, Nov. 2006.

[22] S. Kim, T. Zimmermann, K. Pan, and E. James Jr, "Automatic Identification of Bug-Introducing Changes," *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 81-90, Sep. 2006.

[23] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-written Patches," *International Conference on Software Engineering*, pp. 802-811, May. 2013.

최 한 솔

2018년 한동대학교 전산전자공학부(학사) 2018년~현재 한동대학교 전산전자공학부 석사과정. 관심분야는 유닛 테스팅, 심볼릭 테스팅, 자동 소프트웨어 테스팅

홍 신

2007년 KAIST 전산학부(학사). 2010년 KAIST 전산학부(석사). 2015년 KAIST 전산학부(박사). 2016년~현재 한동대학교 전산전자공학부 조교수. 관심분야는 소프트웨어 테스트 자동 생성, 소프트웨어 자동 디버깅, 멀티쓰레드 프로그램 분석 및 테스팅