

TDD

TEST WORKS 

# 목차

- 1일차
  - Introduction
  - Testing Overview
  - TDD
- 2일차
  - Testability
  - Mocking
- 3일차
  - BDD
  - Cucumber
  - 프로젝트 발표

## Introduction

- 강사 소개
- 각자 소개
  - 배경, 부서
  - 취미
  - 테스트 경험? 단위 테스트? code review에 대한 생각?

# 평가 기준

- 적극적인 참여
- 개인 과제:
  - 배운 내용 노트 정리
  - Programming assignment 제출
- 팀 과제

## Team Project

- Design & implement a “web crawler” – 웹크롤러 만들기
  - (다른 과제로 대체 가능)
- 한 팀당 3-4명
- Implement in any language
- Use TDD/BDD to cover all methods
- Presentation
  - What the team did
  - What was done/not done
  - Evaluate whether TDD/BDD helped

# Team Project

- Requirements: 요구 사항

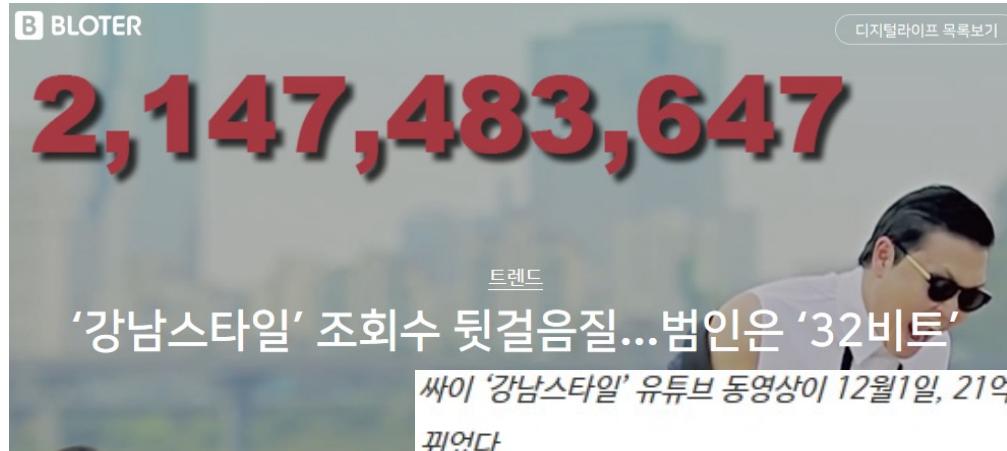
- Given a list of urls, download and save the text content.
- Get all hyperlinks (`<href="...">`)
- Crawl again from the parsed hyperlinks
- List most frequent words
- Detect duplicate hyperlinks
- Bonus Points:
  - Crawl search engine (bing, google, etc.) results
  - Send search texts again based on newly found words
  - Detect language automatically

# I. Testing Overview

- **What is testing?**
- **Why test?**
- **Software Development Life Cycle**



## Testing Overview – Why testing?



유튜브의 동영상 조회수는 원래 32비트로 개발됐다. 2진법의 0과 1이 채워지는 자리가 총 32개라는 뜻이다. 32비트 환경에서 표현할 수 있는 숫자의 범위는  $2,147,483,647$ (21억4748만3647)이다. 이는 양수의 범위고, 음수의 범위까지 더하면  $-2,147,483,648$ 에서  $2,147,483,647$ 까지다. 싸이의 동영상은 12월1일 32비트로 표현할 수 있는 최대 양수 범위인  $2,147,483,647$ 을 넘었다.

이 숫자를 초과하면 어떤 일이 발생할까. 숫자가 돌연 음수로 바뀐다.  $2,147,483,647$ 에서 1명이 더 '강남스타일'을 재생했을 때 현실 세계에서는  $2,147,483,648$ 이 돼야 하지만 컴퓨터 세상에서는  $-2,147,483,648$ 이 나온다. 이를 '정수(인티저) 오버플로우'라고 부른다.

<http://www.bloter.net/archives/214534>

# Testing Overview – Why testing?

[국제]"우버 자율주행차 사망사고 원인은 소프트웨어 결함 때문"

발행일 : 2018.05.08



[AD] [단독] 스마트폰 최대 13회 충전! 대용량 고밀배터리



지난 3월 미국 애리조나주에서 발생한 우버의 자율주행차 사망 사고가 보행자를 감지하고도 무시한 소프트웨어의 결함 문제라고 IT전문매체 디인포메이션 등이 7일(현지시간) 보도했다.

외신 보도에 따르면 우버 자율주행차는 사고 당시 차량의 센서가 보행자를 감지했지만, **소프트웨어는 해당 물체에 즉각적으로 대응할 필요가 없다는 잘못된 판단을 내렸다.**

자율주행차 소프트웨어는 차량 주변의 다양한 물체를 감지한 정보를 바탕으로 주행 관련 판단을 내린다. 예를 들어 도로 위에 날리는 비닐봉지 등 주행에 방해가 되지 않는 물체는 무시할 수 있다. 사고 당시 차량의 카메라, 라이다, 레이더 등 하드웨어는 모두 제대로 작동했지만 소프트웨어 감도 설정에 문제가 있었다는 것이다.

<http://www.etnews.com/20180508000142#>

## Testing Overview – Why testing?

### ■ 소프트웨어 테스팅의 개념

- 프로그램 또는 시스템이 그것에 부과된 것을 수행하는지 **확신을 수립**해가는 프로세스  
(Hetzl, 1973)
- **오류를 발견하려는 의도**로 프로그램 또는 시스템을 실행하는 프로세스  
(Myers, 1979)
- **명세된 요구사항을 만족하는지 검증**하거나 **실제와 기대되는 결과 간의 차이를 식별**하기 위해 시스템을 **수동 또는 자동화된 방법**으로 시험하고 평가하는 프로세스  
(IEEE, 1990)



응용 프로그램 또는 시스템의 동작과 성능, 안정성이 요구하는 수준을 만족하는지  
확인하기 위해 결함을 발견하는 메커니즘

# Verification & Validation

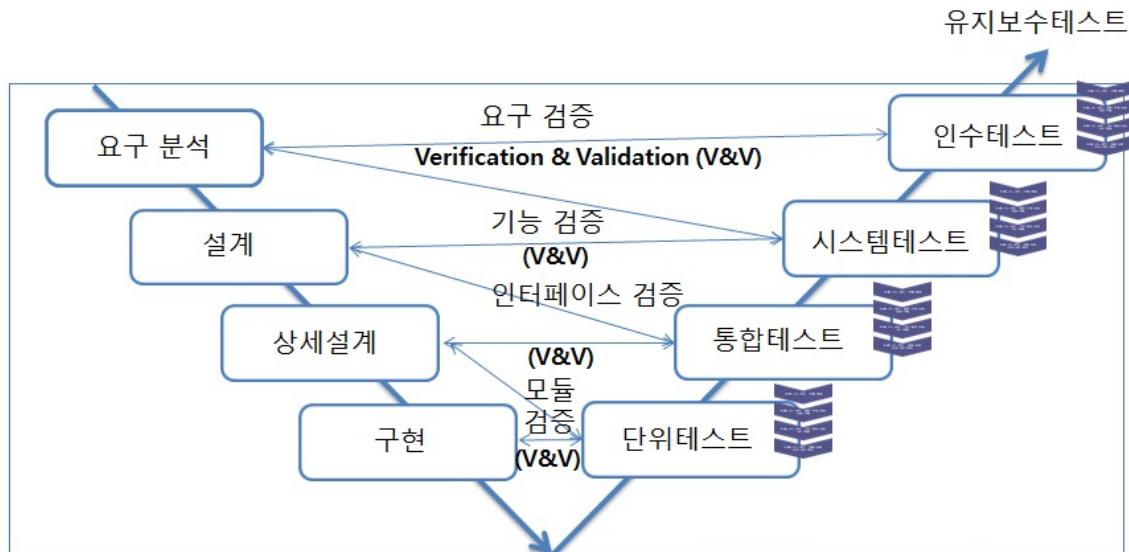


# Testing Overview – 소프트웨어 개발 생명주기와 테스트

## ■ 소프트웨어 개발 생명주기 (SDLC : Software Development Life Cycle)

### - V Model

- Sequential Development Model의 각 개발 단계에 매칭되는 4가지 단계의 테스팅 모델로 정의, 테스트 절차를 강화한 개발 모델
- 요구 분석 기준의 요구 검증 및 확인(V&V), (상세) 설계 기준의 기능 및 인터페이스 검증 및 확인, 구현 코드 기준의 모듈 검증 및 확인



# Test Overview – 소프트웨어 개발 생명주기와 테스트

- **소프트웨어 개발 생명주기 (SDLC : Software Development Life Cycle)**
  - **V Model**
    - Verification and Validation (V&V)

Criteria	Verification	Validation
정의	최종 산출물이 아닌 중간 단계 산출물(work product)이 정의된 요구사항을 만족하는지 평가하는 과정 The process of evaluating work-products(not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.	개발 과정의 최종 단계에서 개발된 소프트웨어 제품이 비즈니스 요구사항을 만족하는지 평가하는 과정 The process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements
평가 관점	소프트웨어 제품을 제대로 만들어 가고 있는지? Are we building the product correctly?	제대로 된 (사용자 요구사항을 만족하는) 소프트웨어 제품을 만들었는지? Are we building the right product?
평가 항목	계획서(Plans), 요구사항정의서(Requirement Specs), 설계 문서(Design Specs), 프로그램 코드(Code), 테스트케이스(Test Cases)....	소프트웨어 제품 (The actual product/software)
평가 활동	Reviews, Walkthroughs, Inspections	Testing

# Test Overview – 소프트웨어 테스트 유형

- 소프트웨어 테스트 분류 – 다양한 구분에 따른 분류

구 분	유 형	특 징
테스트 케이스 획득 (원천)	블랙박스 테스트	기능 요구사항 등 프로그램 외부 명세를 보면서 테스트. 따라서 상세한 기능 요구사항이 요구됨 (Incorrect outputs, incorrect or inconsistent system behavior, missing functionality)
	화이트박스 테스트	프로그램 내부 로직을 보면서 테스트 프로그램 구현 및 내부 구조 이해를 위한 지식이 요구됨 Code Coverage 테스트
프로그램 실행여부	동적 테스트 (Dynamic test)	프로그램을 실행하며 소프트웨어 시스템의 기능, 자원 사용 및 성능 확인 등 비기능 테스트. 모든 레벨에서 테스트 가능. 블랙박스 또는 화이트박스 테스트로 수행 Execute the software and validates the output with the expected outcome.
	정적 테스트 (Static test)	코드를 실행하지 않고 테스트 하는 기법으로 검토(review)같은 수동적(manual)기법과 정적 분석 같은 자동화된(automated) 기법이 있음. 정적 테스트는 개발 프로세스 초기에 개발 산출물들에 대해 결함을 발견함으로써 개발 비용을 낮추는데 도움이 됨.

# Test Overview – 소프트웨어 테스트 유형

## ■ 소프트웨어 테스트 분류 – 다양한 구분에 따른 분류

구 분	유 형	특 징
품질특성 에 따른 분류	기능 테스트 (vs. 비기능)	기능 요구사항 검증
	사용성 테스트 (Usability test)	소프트웨어를 쉽게 사용할 수 있는 정도, 편의 기능 제공 정도(SW's capability to be learned and understood easily and how attractive it looks to the end user)를 측정하고 검증 확인하는 테스트
	성능 테스트	시스템에 부하를 주면서 응답시간, 처리량, 속도 등 성능지표 추이 측정
	스트레스 테스트	복합데이터 또는 대량의 데이터를 이용한 고부하 장기(long-term) 테스트
	회복 테스트	문제가 발생했을 때 복귀 능력 검증. 예) 장애복구테스트
	보안 테스트	외부의 침입이나 해킹에 대응하는 능력 검증
접근 방법 (How to design a set of tests)	탐색적 테스트 (Exploratory test)	테스트 엔지니어의 경험, 지식과 직관에 기초한 테스트. 소프트웨어 시스템 분석을 위한 문서가 제대로 준비되어 있지 않은 경우 시스템 테스트에 유용
	위험기반 테스트 (Risk-based test)	리스크 분석과 결정된 우선순위에 따른 테스팅

# Test Overview – 소프트웨어 테스트 유형

## ■ 소프트웨어 테스트 분류 – 다양한 구분에 따른 분류

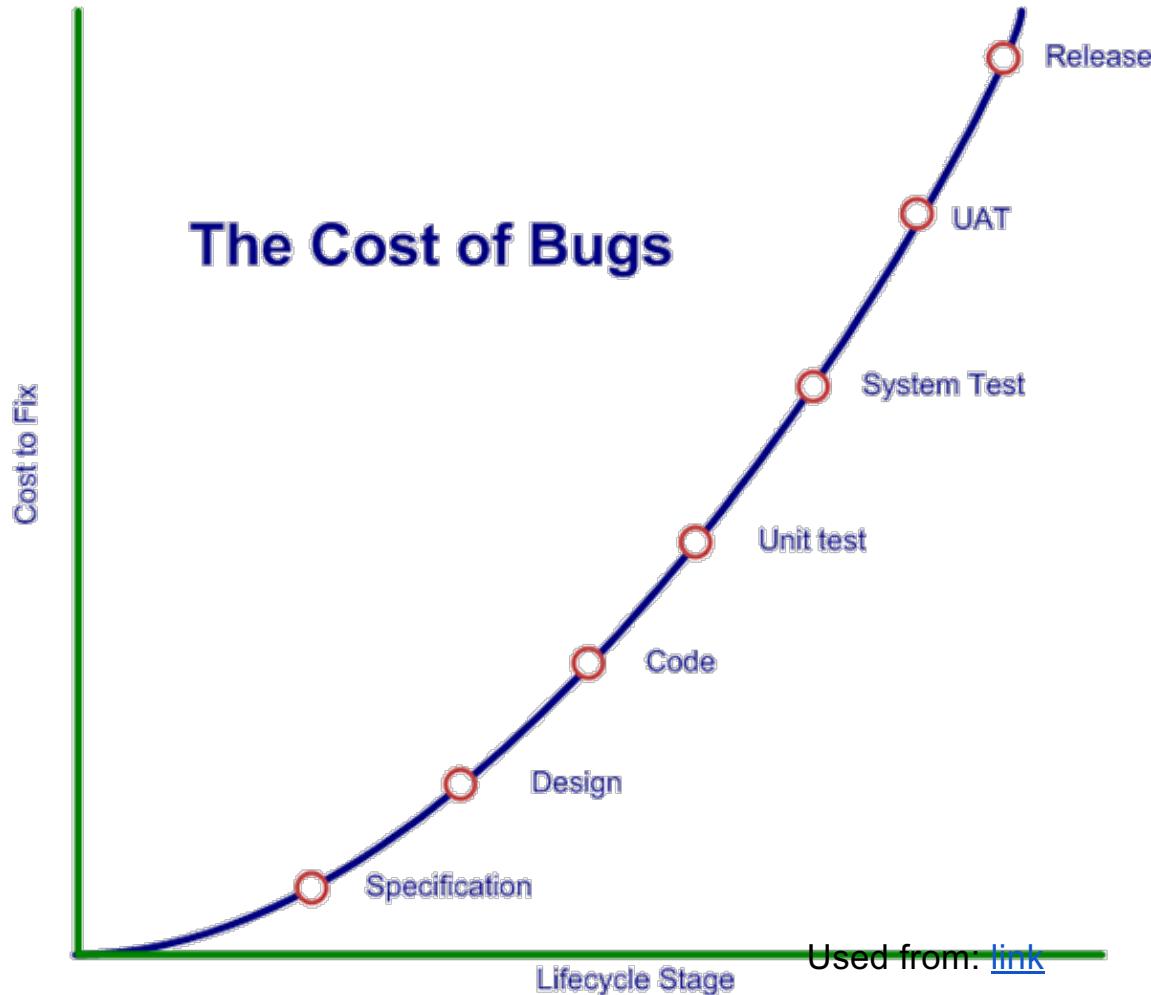
구 분	유 형	특 징
기타	설치/업그레이드 테스트	소프트웨어 제품의 설치 또는 제거, 이전 버전에서 업그레이드 확인.
	회기 테스트 (Regression test)	앞 단계에 정상적으로 수행된 테스트를 변경된 소프트웨어 시스템에 대해서 다시 테스트를 수행함으로써 변경에 의한 영향을 검증. Automated test의 좋은 후보!
	알파 테스트 (Alpha test)	개발이 거의 끝나는 단계이나 minor 변경이 있는 상태에서 개발팀 내 구성원이 아닌 사람 또는 사용자들에 의해 수행되는 테스트
	베타 테스트 (Beta test)	개발이 거의 끝나는 단계에서 출시(final release) 이전 출시에 문제가 되는 결함이 없음을 확인하기 위해 개발팀 내 구성원이 아닌 사람 또는 사용자들에 의해 수행되는 테스트

# Test Overview – 소프트웨어 개발 생명주기와 테스트

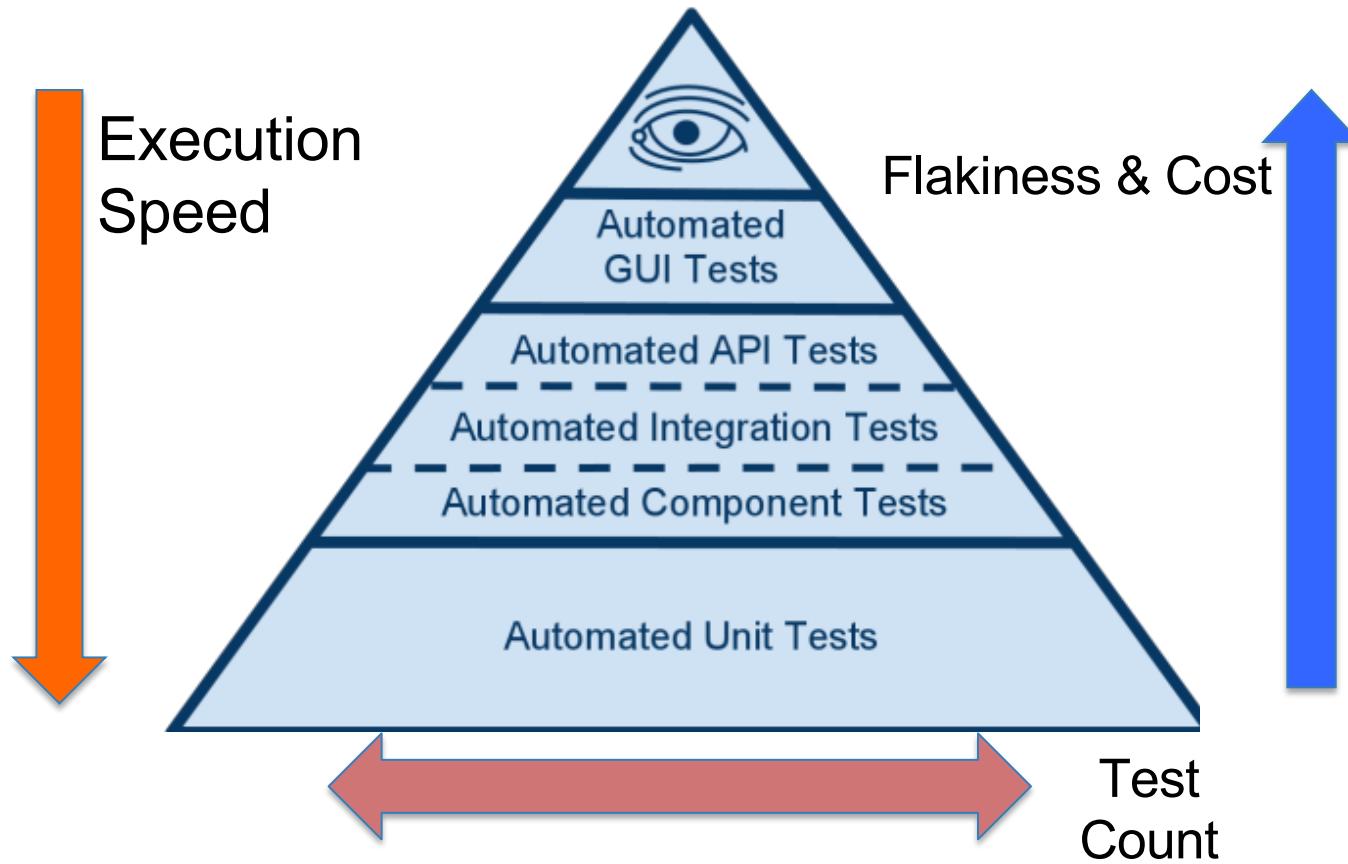
- 소프트웨어 개발 생명주기 (SDLC : Software Development Life Cycle)
  - V Model
    - 단계별 테스트

단계별 테스트	테스트 방법
인수 테스트 (Acceptance Test)	시스템이 사용자가 인수할만한 수준인지 판정하기 위한 테스트 - 사용자의 만족 여부를 테스트 하는 것, 결함을 찾는 것이 목적이 아님
시스템 테스트 (System Test)	전체 시스템의 기능, 비기능 테스트 - 실제 환경과 유사한 환경에서 수행 - 신뢰성, 견고성, 성능, 안전성, 보안성 등의 비기능적 요구사항 테스트
통합 테스트 (Integration Test)	모듈간 인터페이스 테스트 - 모듈을 결합하여 하나의 시스템으로 구성 시 테스트 수행
단위 테스트 (Unit Test)	각 모듈들의 독립적 평가 - 구현 단계에서 프로그램 개발자에 의해 수행되기도 함. - 개별 모듈 테스트를 위해 모듈의 단독 실행 환경 필요

## 테스팅을 하는 이유? - 비용



# Testing Pyramid (Ideal)



# How to test?

## ▪ Implications

- The earlier to test, the cheaper the cost
- The faster to test, the easier to test
- The easier to test, the higher the quality

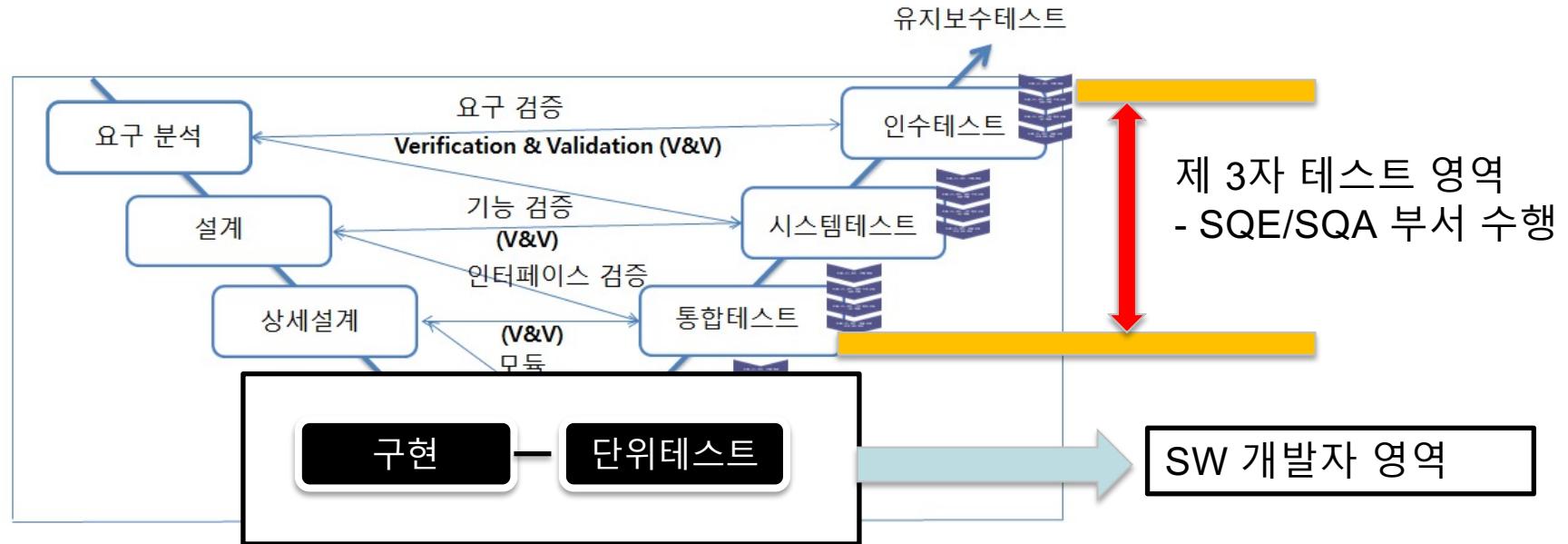
## II. 단위 테스트

- 단위테스트 개요
- **Code Coverage**
- **Junit**

## 단위 테스트 – 정의

- 테스트 대상이 되는 코드 기능의 작은 특정 영역(메소드 또는 클래스 등) 을 실행하며, 기대 값을 확인 하기 위해 개발자가 작성한 코드 조각을 말함 (A unit test is a piece of code written by a developer that executes a specific functionality in the code to be tested)
- 단위 코드가 의도한대로 제대로 동작하는지 확인하기 위해 수행. 단위 코드 가 원하는 대로 동작하지 않으면 실패(Failure)로 처리, 개발자의 코드 수정 을 유도 함

## 단위 테스트 - 개념



- 개발자는 본인이 구현하는 코드에 대한 검증을 수행
- 개발자가 실수를 개발자 스스로 찾아내지 않으면 제 3자 테스트를 통해 검출이 가능하지만 많은 비용이 발생

## 단위 테스트 - 중요성

SW를 완벽하게 테스팅하는 것은 불가능하다. 따라서, SW 결함은 출시 후 시장 불량으로 고객에 의해 보고된다. SW 시장 불량에 대한 책임은 어느 부서에 있는가?

- 1) 개발팀
- 2) 테스트팀 (SQE, SQA 등)
- 3) 개발팀 내 코딩을 작성한 담당 개발자
- 4) 1)/2)/3) 모두 해당

## 단위 테스트 – Who owns the SW quality?

- 시장 불량에 대한 책임은 결함을 검출하지 못한 테스트팀의 단독 책임이 아니며 모든 유관 부서가 책임을 져야 함
- SW 품질에 대한 1차 책임자는 코딩을 구현하는 개발자임
- SQE/SQA는 개발자의 산출물을 검증하지만 개발자가 생성하는 모든 경우의 수에 대한 검증이 불가능하며 따라서 모든 결함을 검출하기 어려움
- Google 및 Microsoft 경우 SW Quality는 개발자에게 있다고 명시하며 개발자는 반드시 코드리뷰, 단위테스트, 통합테스트를 수행해야 함

## 단위 테스트 – 실 사례 (Len bytes에 대한 checksum 생성)

```
int checksum(void *p, int len)
{
    int accum = 0;
    unsigned char* pp = (unsigned char*)p;
    int i;
    for (i = 0; i <= len; i++)
    {
        accum += *pp++;
    }
    return accum;
```

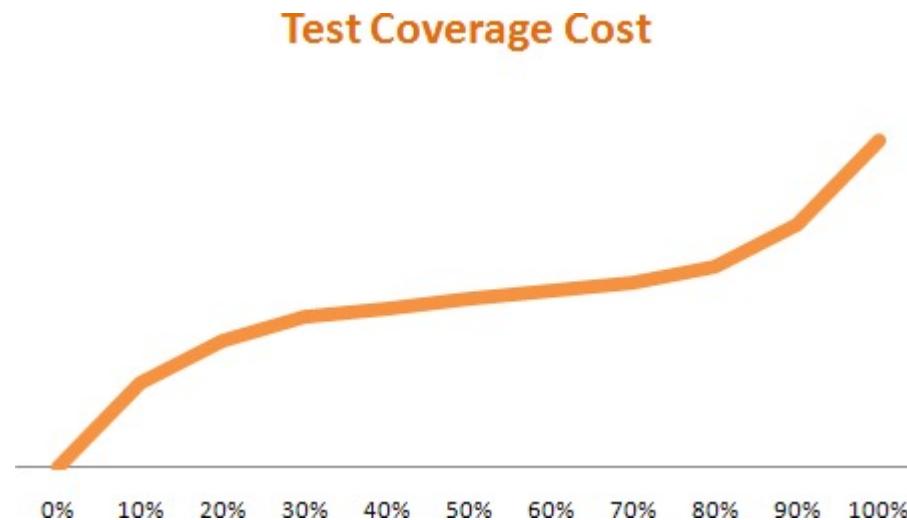
```
char *myString = "testing";
Int testval = checksum(myString,
strlen(myString))
```

for loop의 i <= len 은  
for loop i < len 으로 수정 필요

- 상기의 결함을 개발자가 단위 테스트를 통해서 검출하지 못한다면?
- QE에서 Aging Test (8 – 20시간)를 통해서 결함을 검출하며 결함 재현 증상이 명확하지 않음
- QE에서 결함을 등록하여 개발팀에서 원인 분석을 위해 별도의 Aging Test 수행
- 결함은 코드리뷰/단위테스트 1시간으로 검출 가능한 for loop의 단순 로직 실수임.

## Code Coverage – 필요성

- **Code Coverage Cost**



- 완벽한 테스트란?

출처: <http://povilasb.com>

## IV. 단위 테스트

- 단위테스트 개요
- Code Coverage
- Junit

# Code Coverage – 커버리지 비율과 결함 탐지율의 관계

- Coverage Requirement 와 테스트 Effort 관계

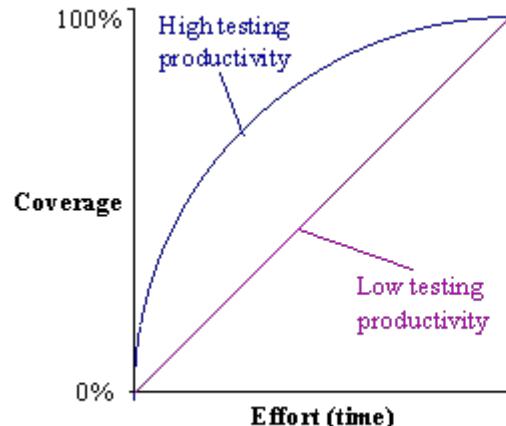


Figure 1: Coverage rate

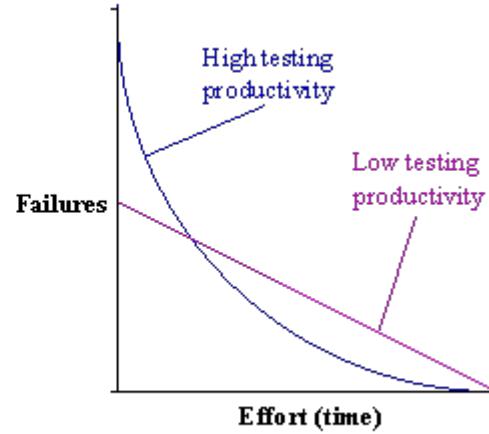


Figure 2: Failure discovery rate

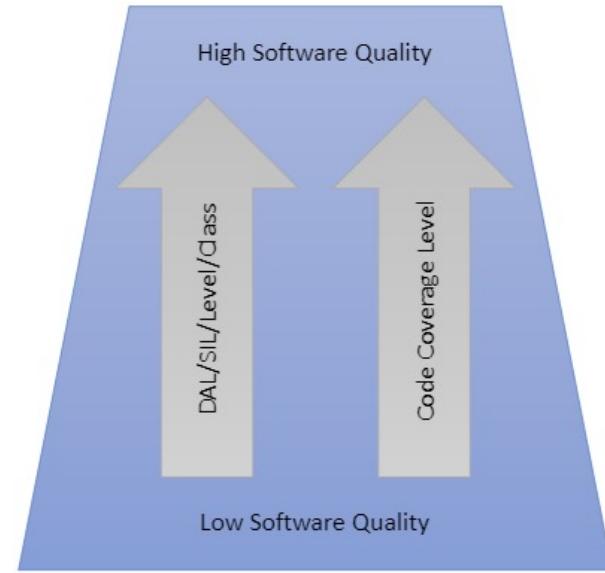
출처: [www.bullseye.com](http://www.bullseye.com)

- 우리 인건비는 얼마인가?

## Code Coverage – Code Coverage Level

- 과연 Code Coverage는 얼마를 목표로 하는가?
- SIL 이란?
- SIL Level에 따른 Code Coverage

Requirement	Statement	Branch	MC/DC
DO-178 B/C (Avionics)			
Level A	.	.	.
Level B	.	.	.
Level C	.		
ISO-26262 (Automotive)			
ASIL D	.	.	.
ASIL B/C	.	.	.
ASIL A	.		
IEC-61508 (Industrial)			
SIL 4	.	.	.
SIL 3	.	.	.
SIL 1/2	.		
EN-50128 (Railway)			
SIL 4	.	.	.
SIL 3	.	.	.
SIL 1/2	.		
IEC-62304 (Medical)			
Class C	.	.	.
Class B	.	.	
Class A	.		



# Code Coverage – 1장으로 보는 Code Coverage

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

&lt;Condition Coverage&gt;

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

&lt;Decision Coverage&gt;

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

&lt;Condition/Decision Coverage&gt;

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

&lt;Modified Condition/Decision Coverage&gt;

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

&lt;Multiple Condition Coverage&gt;

# Code Coverage – 이해

- 첫번째, Condition Coverage의 경우는,

Condition A의 결과인 1, 0 그리고 Condition B의 결과인 1, 0이 적어도 한번씩 Test Situation으로 선택된다.  
이것을 만족하는 Test Situation은 위의 예 말고도 더 있다.

- 두번째, Decision Coverage의 경우는,

Decision의 결과인 1과 0이 각각 적어도 한번씩 Test Situation으로 선택된다.  
이것을 만족하는 Test Situation은 위의 예 말고도 더 있다.

- 세번째, Condition/Decision Coverage의 경우는,

각각의 개별 Condition의 결과값이 0, 1 모두 적어도 한번씩 Test Situation으로 선택되고,  
Decision의 결과값이 0, 1 모두 적어도 한번씩 Test Situation으로 선택된다.

- 다섯째, Multiple Condition Coverage의 경우는,

그냥 단순히 모두를 선택하면 된다. 선택되는 테스트의 개수는  $2^n$ (이때, n은 Condition의 개수)가 될 것이다.

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

&lt;Condition Coverage&gt;

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

&lt;Decision Coverage&gt;

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

&lt;Condition/Decision Coverage&gt;

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

&lt;Modified Condition/Decision Coverage&gt;

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

&lt;Multiple Condition Coverage&gt;

## Code Coverage – 실습 : 강사가 제시하는 예문에 대하여

- 모든 경우의 수의 Statement Coverage를 구하여라.
- 모든 경우의 수의 Condition Coverage를 구하여라.
- 모든 경우의 수의 Condition/Decision Coverage를 구하여라.
- 모든 경우의 수의 Multiple Condition Coverage를 구하여라.

## Code Coverage – 실습

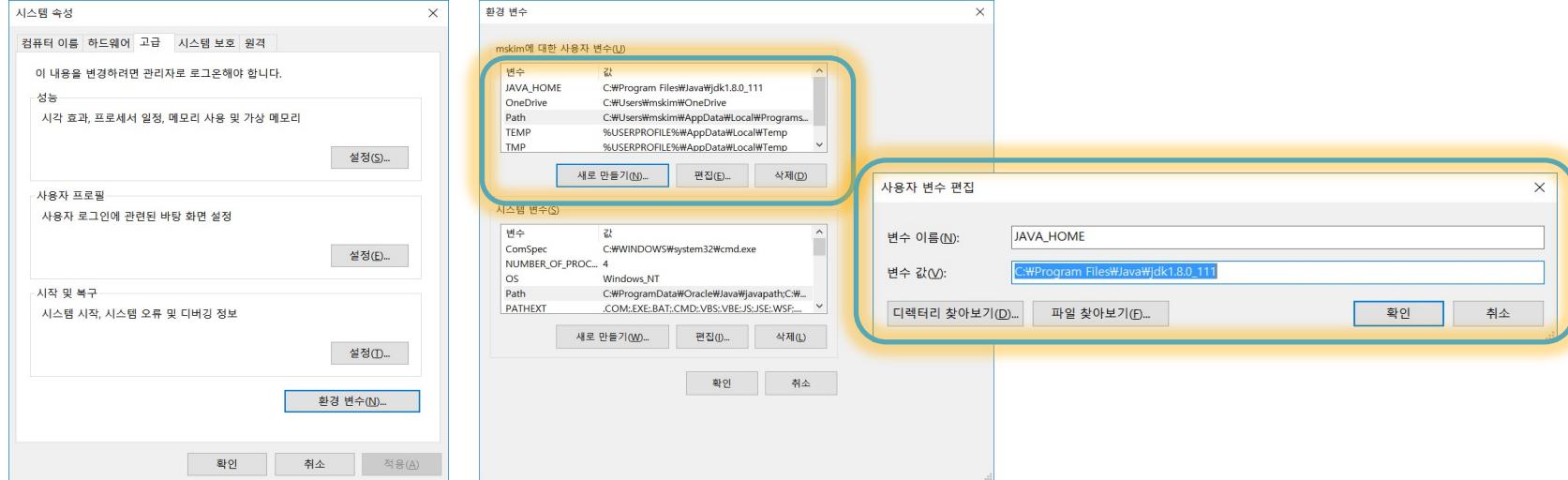
- MC/DC Coverage를 구하여라.
  - 앞에 까지는 쉬웠을 것이다. 이제 시작이다.
- 이렇게 어려운 MC/DC는 도대체 왜 쓸 가?
  - 꼭 5분 이상 생각한다. 타인을 인내하고 기다리자.
  - 정의란 무엇인가 생각해보지 않았는가?
- 이렇게 복잡한 MC/DC는 어디에 쓸까?
  - 꼭 5분 이상 생각한다. 타인을 인내하고 기다리자.
  - 정의란 무엇인가 생각해보지 않았는가?

## II. 단위 테스트

- 단위테스트 개요
- JUnit
- Code Coverage

# JDK 설치

- <https://www.oracle.com/java/technologies/javase-downloads.html> 에서 PC 환경에 적합한 버전 다운 → 설치파일 실행
- “JAVA\_HOME” 환경 변수 추가
- “Path”에 %JAVA\_HOME%\bin 추가



## 단위 테스트 – JUnit

- JUnit Assert statements : 실제 값과 기대 값을 비교하여 참, 거짓 구별

Statement	Description
fail(String)	Let the method fail. The String parameter is optional. 테스트를 바로 실패 처리함. 선택 사항인 message가 있다면 출력. 종종 절대 수행되지 않아야 될 부분 (예를 들면 예외가 발생하는 부분)을 표시하는데 사용
assertTrue([message], boolean condition)	Checks that the boolean condition is true. Boolean 조건이 참인지 판정. 조건이 거짓이면 실패로 처리.
assertFalse([message], boolean condition)	Checks that the boolean condition is false.
assertEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message], object)	Checks that the object is null. 인자로 넘겨 받은 객체가 null인지(또는 null이 아닌지) 판정하고 반대인 경우 실패로 처리. message 매개변수는 생략할 수 있다.
assertNotNull([message], object)	Checks that the object is not null.

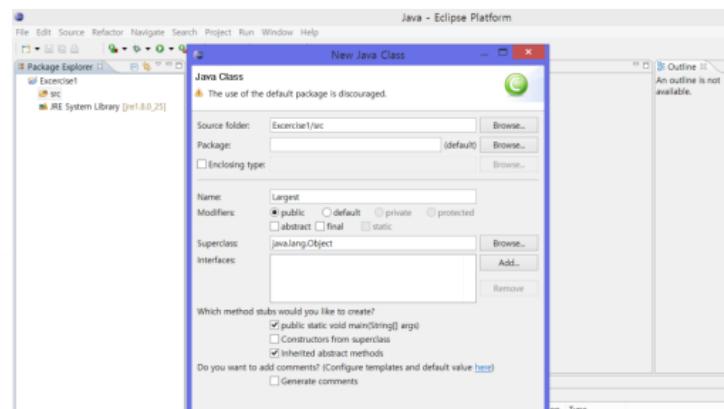


## 단위 테스트 – JUnit 예제 (1/7)

[숫자 목록에서 가장 큰 숫자를 찾는 정적 메서드]에 대한 단위 테스트를 통해 결함을 찾아 코드를 수정하는 과정에 대한 설명이다

### 1. 코드 (단위 테스트 대상 클래스) 구현

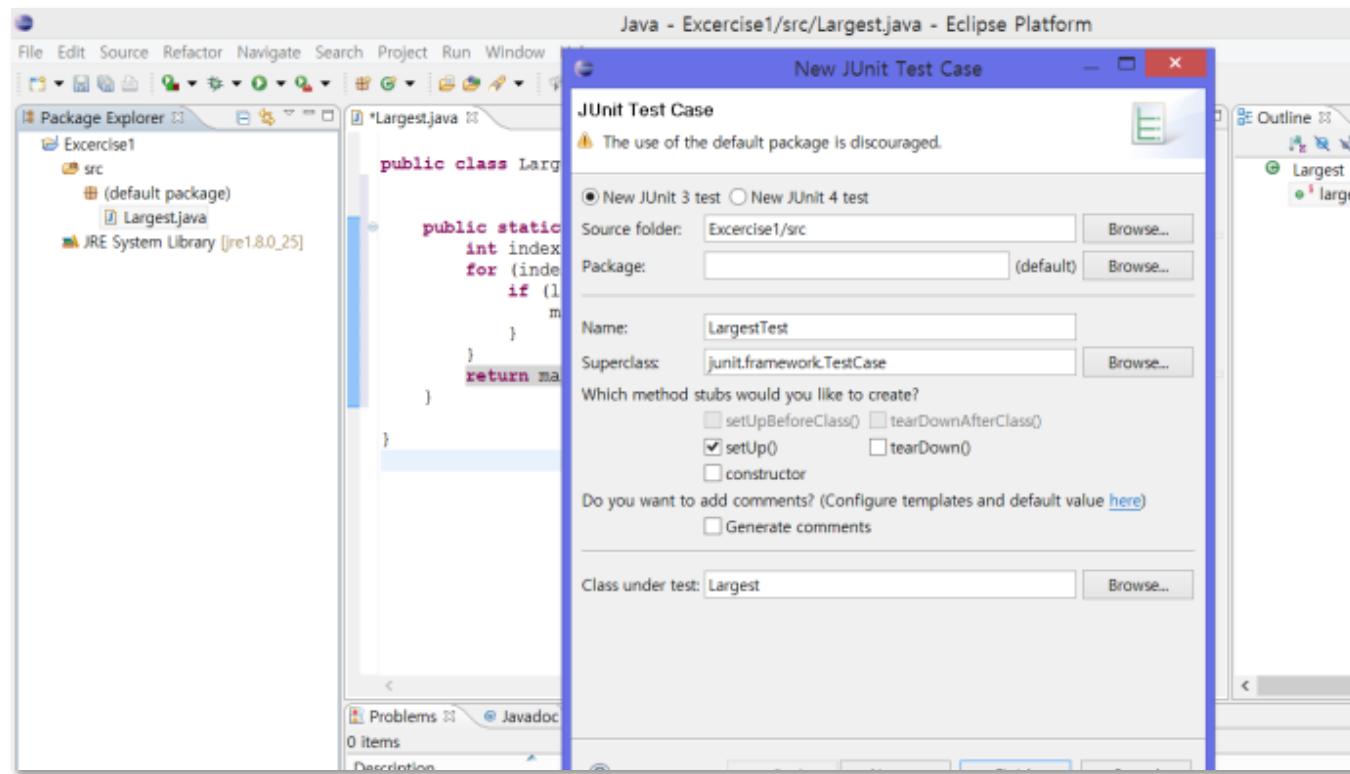
```
public class Largest {  
  
    public static int largest(int[] list) {  
        int index, max=Integer.MAX_VALUE;  
        for (index= 0; index <list.length-1; index++) {  
            if (list[index] > max) {  
                max = list[index];  
            }  
        }  
        return max;  
    }  
}
```



## 단위 테스트 – JUnit 예제 (2/7)

### 2. 단위 테스트 (Unit Test) 작성 코드

- 1) 테스트 대상 코드 선택
- 2) File -> New -> JUnit Test Case



## 단위 테스트 – JUnit 예제 (3/7)

### 2. 단위 테스트 (Unit Test) 작성 코드

- 1) 테스트 대상 코드 선택
- 2) File -> New -> JUnit Test Case
- 3) 단위 테스트 작성: 코드가 원하는 대로 동작하는지 확인하기 위해 **assertion** 사용. 이 경우는 **assertEquals** 사용

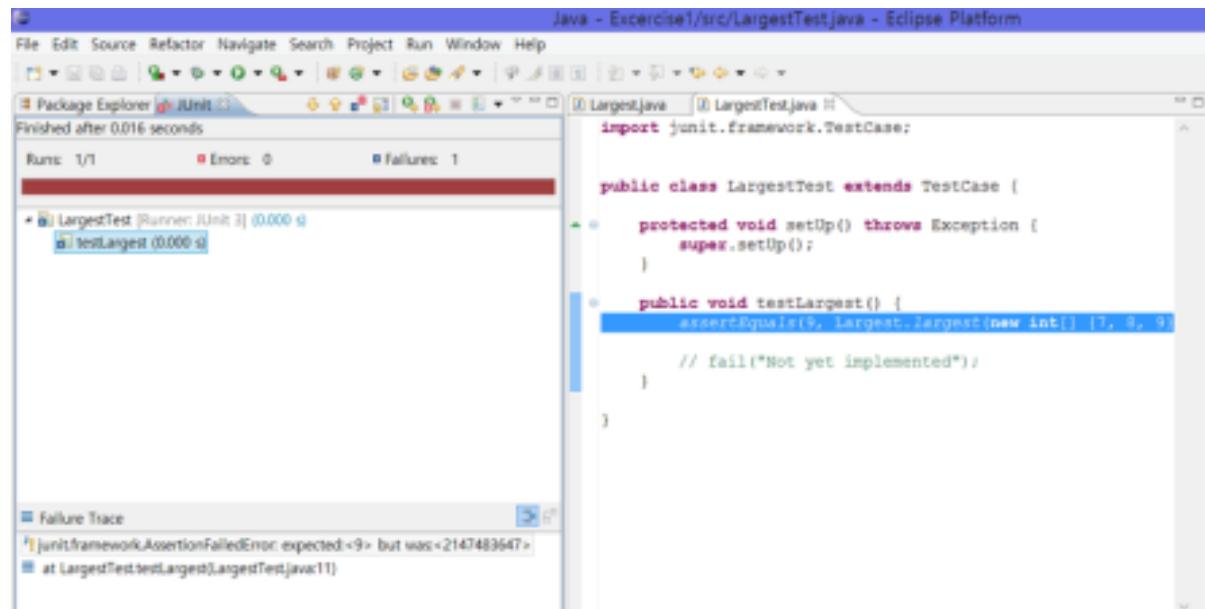
```
import junit.framework.TestCase;  
  
public class LargestTest extends TestCase {  
  
    protected void setUp() throws Exception {  
        super.setUp();  
    }  
  
    public void testLargest() {  
        assertEquals(9, Largest.largest(new int[] {7, 8, 9}));  
  
        // fail("Not yet implemented");  
    }  
}
```



## 단위 테스트 – JUnit 예제 (4/7)

### 3. JUnit Test 수행

- Run as JUnit Test



junit.framework.AssertionFailedError: expected:<9> but was:<2147483647>  
at junit.framework.Assert.fail(Assert.java:47) ....

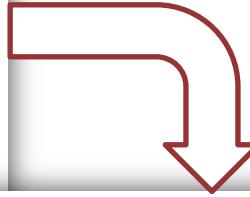
단위 테스트 수행  
결과 실패 (Failures:1)  
→ 테스트 대상  
코드 수정

## 단위 테스트 – JUnit 예제 (5/7)

### 4. 테스트 대상 코드 수정

- ✓ Unit Test 실행 결과 실패 (Failures: 1).
- ✓ 실패의 원인 분석 결과 max 초기값 수정 필요. max=0 코드 추가 (수정)

```
public class Largest {  
  
    public static int largest(int[] list) {  
        int index, max=Integer.MAX_VALUE;  
        for (index= 0; index <list.length-1; index++) {  
            if (list[index] > max) {  
                max = list[index];  
            }  
        }  
        return max;  
    }  
}
```



max 초기값 수정

```
public class Largest {  
  
    public static int largest(int[] list) {  
        int index, max=Integer.MAX_VALUE;  
        max=0;  
        for (index= 0; index <list.length-1; index++) {  
            if (list[index] > max) {  
                max = list[index];  
            }  
        }  
        return max;  
    }  
}
```

## 단위 테스트 – JUnit 예제 (6/7)

### 5. JUnit Test 재 실행

- Run as JUnit Test (Largest.java 코드 수정 후 JUnit Test 재 실행)

The screenshot shows the Eclipse IDE interface. The top menu bar says "Java - Excercise1/src/Largest.java - Eclipse Platform". The left sidebar has "Package Explorer" and "JUnit" tabs. The "JUnit" tab shows "Runs: 1/1", "Errors: 0", and "Failures: 1". It lists a single test: "LargestTest [Runner: JUnit 3] (0.000 s)" with "testLargest (0.000 s)". The right side is the code editor for "Largest.java" and "LargestTest.java". The "Largest.java" code is:

```
public class Largest {  
    public static int largest(int[] list) {  
        int index, max=Integer.MAX_VALUE;  
        max=0;  
        for (index= 0; index <list.length-1; index++) {  
            if (list[index] > max) {  
                max = list[index];  
            }  
        }  
        return max;  
    }  
}
```

The "LargestTest.java" code is:

```
import org.junit.Test;  
import static org.junit.Assert.*;  
  
public class LargestTest {  
    @Test  
    public void testLargest() {  
        int[] list={1,2,3,4,5,6,7,8,9};  
        assertEquals(9, Largest.largest(list));  
    }  
}
```

The bottom "Failure Trace" panel shows the error message:

```
junit.framework.AssertionFailedError: expected:<9> but was:<8>  
at junit.framework.Assert.fail(Assert.java:47) ....
```

재 수행 후에도 단위  
테스트 수행 결과 실패  
→ 테스트 대상  
코드 수정

junit.framework.AssertionFailedError: expected:<9> but was:<8>  
at junit.framework.Assert.fail(Assert.java:47) ....

## 단위 테스트 – JUnit 예제 (7/7)

### 6. 테스트 대상 코드 재 수정

- ✓ JUnit Test 실행 결과 실패 (Failures: 1).
- ✓ 실패의 원인 분석 결과 list 배열의 마지막 값의 크기가 비교 되지 않음 확인
- ✓ “`index < list.length-1`” 을 “`index <= list.length-1`”, 또는 “`index < list.length`”로 변경

```
public class Largest {  
  
    public static int largest(int[] list) {  
        int index, max=Integer.MAX_VALUE;  
        max=0;  
        for (index= 0; index <list.length-1; index++) {  
            if (list[index] > max) {  
                max = list[index];  
            }  
        }  
        return max;  
    }  
}
```

index  $\leq$  list.length-1

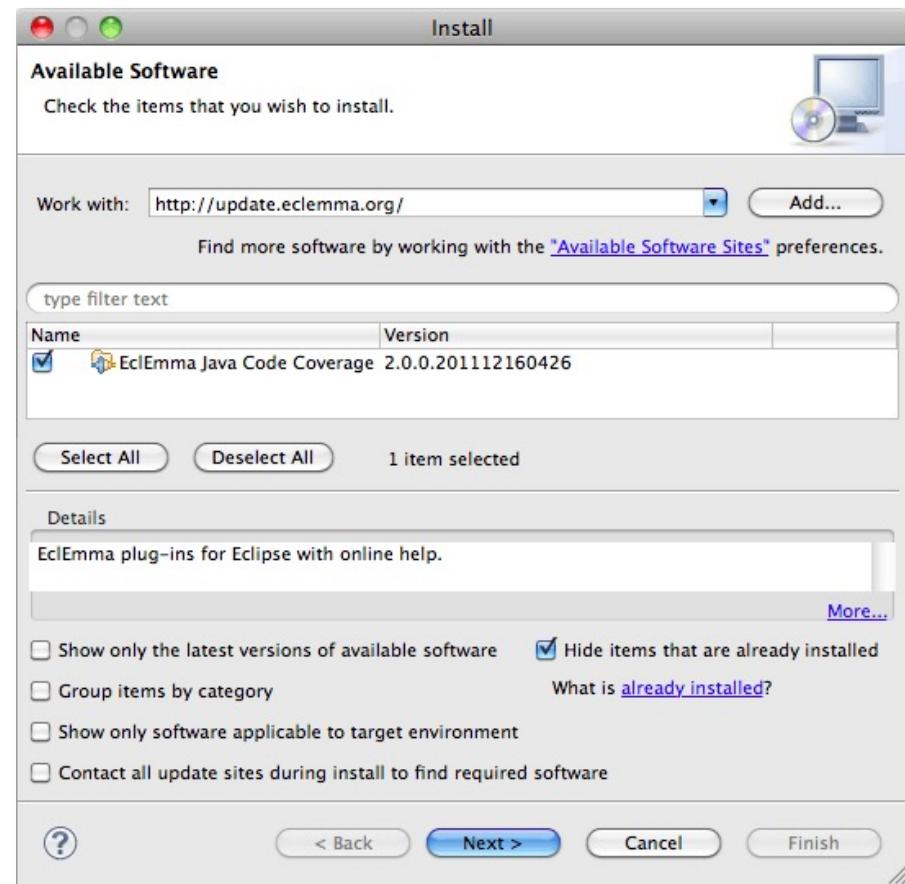
```
public class Largest {  
  
    public static int largest(int[] list) {  
        int index, max=Integer.MAX_VALUE;  
        max=0;  
        for (index= 0; index <list.length-1; index++) {  
            if (list[index] > max) {  
                max = list[index];  
            }  
        }  
        return max;  
    }  
}
```

## [부록] Code Coverage 측정 도구-\*\*JUnit에 적용

- **ECLEmma**
  - Eclipse에서 사용 가능한 오픈 소스 Java Coverage Tool
  - Statement, Branch, Methods 등의 코드 커버리지 측정 가능
- 참고 사이트
  - <http://eclemma.org/>
- 다른 커버리지 측정 도구
  - Code Cover
  - JaCoCo
  - Etc

## [부록] Code Coverage 측정 도구-\*\*JUnit에 적용

- EclEmma 설치
  - Eclipse 실행 -> Help -> Install New Software...
  - Work with 항목에  
<http://update.eclemma.org>  
 입력



## [부록] Code Coverage 측정 도구-\*\*JUnit에 적용

### ■ ECIEmma 실행

- Tool bar에서 아래 아이콘 클릭



- 커버된 Code는 초록색으로  
커버되지 않는 코드는  
빨간색으로 하이라이트 됨.

```

HelloWorldJava  HelloWorldT...  StringHelper...  StringHelper...  TDDStringHel...  TDDStringHel...
2
3 public class StringHelper {
4
5     //AACD => CD, ACD => CD, CDEF => CDEF, CDAA => CDAA
6     public String truncateAInFirst2Positions(String str) {
7         if (str.length() <= 2) {
8             //System.out.println(result);
9             return str.replaceAll("A", "");
10        }
11        String first2Chars = str.substring(0, 2);
12        String stringMinusFirst2Chars = str.substring(2);
13
14        return first2Chars.replaceAll("A", "") +
15            stringMinusFirst2Chars;
16    }
17
18    public boolean areFirstAndLastTwoCharactersTheSame(String str) {
19
20        if (str.length() <= 1)
21            return false;
22        if (str.length() == 2)
23            return true;
24
25        String first2Chars = str.substring(0, 2);
26
27        String last2Chars = str.substring(str.length() - 2);
28
29        return first2Chars.equals(last2Chars);
30    }
31

```

Problems Javadoc Declaration Console Coverage StringHelperTest (2016. 3. 17 오전 9:23:25)

Element	Coverage	Covered Branches	Missed Branches	Total Branches
JunitCourse	66.7 %	4	2	6
src/main/java	66.7 %	4	2	6
com.testworks.junit.helper	66.7 %	4	2	6
src/test/java		0	0	0

## [부록] Code Coverage 측정 도구-\*\*Junit에 적용

- **Code Coverage is not about tool but about quality goal with consensus in an organization**
  - 100% Line Coverage Or Branch Coverage?
- **Safety critical software requires branch coverage**
  - Sometimes, you are forced to have branch coverage due to poor quality for your product
- **100% code coverage will guarantee quality?**
  - For Microsoft, 80% is something realistic
  - In Agile community, there is one consensus
    - If your code coverage ratio is low, that's a bad smell. If it's high... well... we can't say anything...
  - There will be always bugs beyond code coverage

### III. TDD

- TDD 개요
- TDD 예제 및 실습

## TDD 개요

- TDD 란?
  - “작성 해야 하는 프로그램에 대한 테스트를 먼저 작성” 후 “테스트를 통과할 수 있도록 실제 프로그램의 코드를 작성”하는 것

# TDD 개요

## ■ TDD 소개

### 1 기원

#### TDD 출현의 기원

- SW 개발 방법의 하나로 XP의 Test-first 프로그래밍에서 발전

### 2 특징

#### 방법론적 특징

- 증분형 개발 방법론
- SW 설계의 점진적 개선 (Test-driven design)

### 3 검증방법

#### 개발 도중 검증

- 선행된 테스트 코드와 매칭된 개발 코드의 테스트를 통한 피드백 중심 개발

### 4 고려사항

#### 테스트 측면

- 시스템 검증 및 Testability의 고려
- 그러나, TDD는 테스트가 아니므로 그에 따른 별도의 개념이 필요

### 5 고려사항

#### 자동화 측면

- 단위테스트 기능을 겸비한 자동화 도구 필요

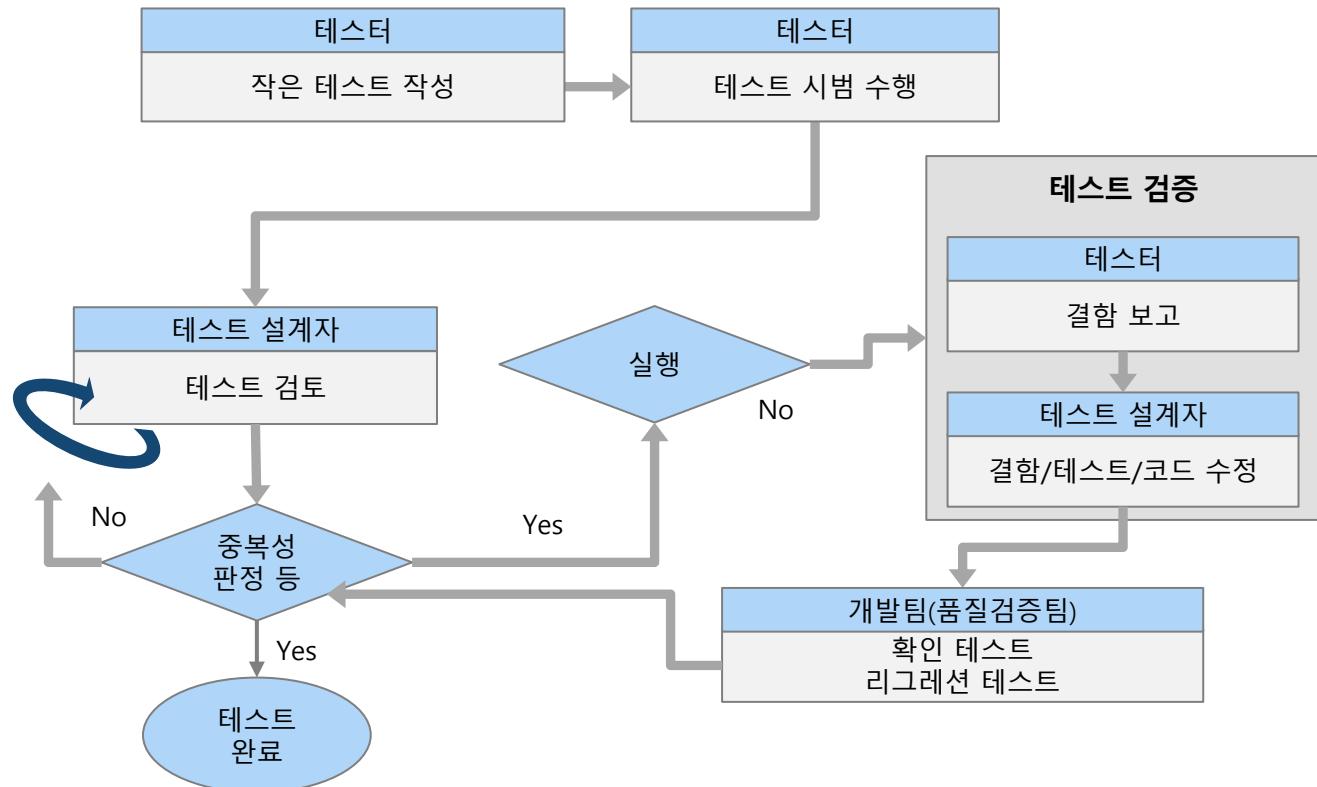
### 3 고려사항

#### 시스템 측면

- Full Build 되지 않은 코드 테스트
- Compile 되지 않은 코드 테스트

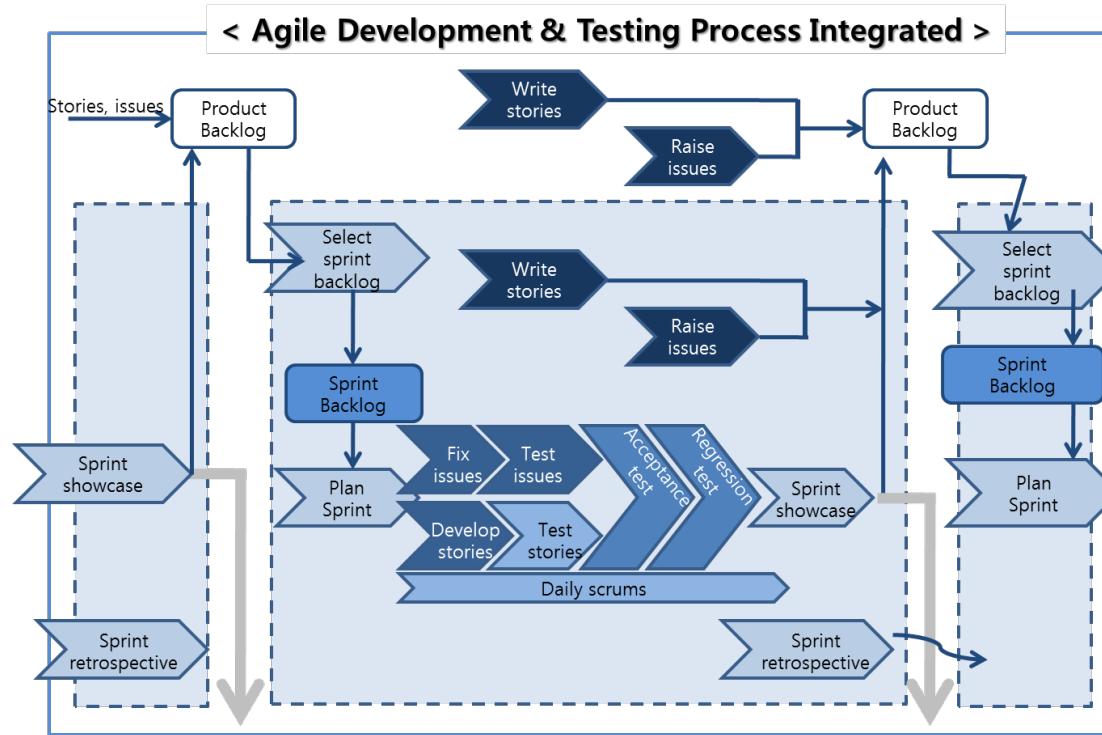
# TDD 개요

## ■ TDD 개발 주기



# TDD 개요

## ■ Scrum - Agile Development & Testing Process Integrated



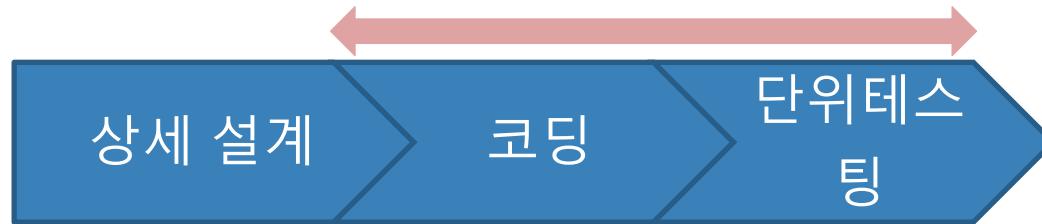
Ref. Testing Options in Agile Projects by Stuart Reid, 2009

Note : There are many testing activities in a sprint.  
But if sprint integration, non-functional and final acceptance testing are combined...?

## TDD 개요

- 기존 방식 vs. TDD (Agile)

- 기존 방식



- TDD (애자일)



## TDD 절차

- 세가지 색깔의 모자를 쓰고 있음!

### 1. 처음 Red phase - failing test

- 최소한의 코드로 실패하는 케이스만 만들면 됨

### 2. Green phase-코드가 돌아갈 만큼만 작성

- 최소한의 코드로 돌아가게만 하면 됨

### 3. Blue phase- refactor하는 단계

- 중복이 없어야 함
  - 동일한 코드 블록이 다른 곳에 중복되게 쓰이는 것만 의미하지 않음
  - 구조의 가독성을 개선하는 작업

### 4. 종료 - 더이상 생각나는 테스트가 없으면 멈춤

## TDD의 3가지 원칙

- **Fail Test가 있을 때만 Production Code에 코드를 작성한다**
  - **Write NO production code except to pass a failing test**
- **실패를 나타낼 수 있는 정도의 테스트만 작성하고 너무 한 곳에 머물러 있지 말라**
  - 내가 원하는 기능이 작동하지 않음을 표현하는 정도까지만
  - **Write only ENOUGH of a test to demonstrate a failure**
- **실패하는 테스트가 있으면 성공하는 만큼만 코드를 작성해라**
  - 코드가 견고해져서 뭔가를 할 수 없을 경우가 발생
  - 디버깅-테스트를 잘못된 순서를 추구할때 발생함
  - **Write only ENOUGH production code to pass the test**

## TDD의 원칙과 팁 (1/2)

### ▪ Most Simple

- 제안 단순 한 테스트 작성:  
**Degenerate(수준 이하의, 레벨 이하의) test case** 를 우선 작성
- 테스트 작성 순서 : 간단 → 복잡
- 단순 테스트 작성 이유
  - 스파게티 코드화 - 코드가 잔뜩 꼬여서 나중에 단순한 것을 테스트 할 수 없다
  - 테스트 중복성 - 이미 테스트 수행 된 경우 발생, 단위 테스트 불가능 상태 유발
  - 예시 - 계산기를 할때 제일 쉬운 기능이 무엇일까?  
거꾸로 뭐가 가장 어려울까? **divide by zero** 아닐까?
- 어려운 것부터 하지 말고 **0+0** 부터 하자

## TDD의 원칙과 팁 (2/2)

- **Little Golf Game**

- 테스트가 fail될 때 너무 많은 코드를 짜지 말고 돌아갈 정도만 코딩한다

- **As the tests get more specific, the code gets more generic**

- 테스트는 개별 케이스에 대해서 점점 더 구체화 되어 가고  
코드는 처음에는 구체적이었지만 점점 더 범용이 되어간다
    - 모든 (테스트)케이스를 수용할 수 있는
  - 이 부분이 이해가 된다면 테스트를 어떤 식으로 추가하면 되겠구나 하고 알게 된  
다

## TDD의 장점 (1/3)

- **Debugging Time 감소**

- 동작 코드 개발 시간 증가
    - 디버깅은 줄고 줄이고 동작하는 코드를 작성하는데 시간 증대
    - TDD가 디버깅 시간을 **1/10**로 줄여 줄 것이다
    - 실은, **1/10**이 아니라 **1/2**만 줄어도 된다
    - 증가된 TC → 디버깅 감소

## TDD의 장점 (2/3)

### ▪ Decoupling 감소

- 개발 초기 미완성 제품 코드 테스트 가능
  - 기존 방식은 완성형 제품 코드가 필요
  - 완성형 제품 코드에서는 시기적=으로 늦춰진 테스트 수행
  - 단위테스트 코드 자체의 폭증
  - 코드 증가 → 의존성 증가 → 단위테스트 수행 불가
- 공학적 기준에 부합하는 코드의 자동 생성
  - 응집도 향상
  - 결합도 감소
  - 테스트하기 좋은 코드 양산 → 코드 품질 강화

## TDD의 장점 (3/3)

- **Design Documents 자동 생성**
  - TDD의 세가지 법칙 준수 → 설계 문서 자동 생성
  - **TDD Test = Low Level Design document**
    - 테스트 코드 → 코드 수행 알고리즘 직관적 이해 가능

## TDD 수행방법

- TDD 과정 요약

- 1) 테스트 작성
- 2) 작성한 테스트에 대한 코드 작성
- 3) 테스트 통과 시 작성한 코드에서 중복 제거 실패 시 2번으로 재 작성
- 4) 테스트 수행
- 5) 테스트 통과 시 완성이면 다음 테스트를 1번부터 시작 하고 실패 시 4로 돌아감

### III. TDD

- TDD 개요
- TDD 예제 및 실습

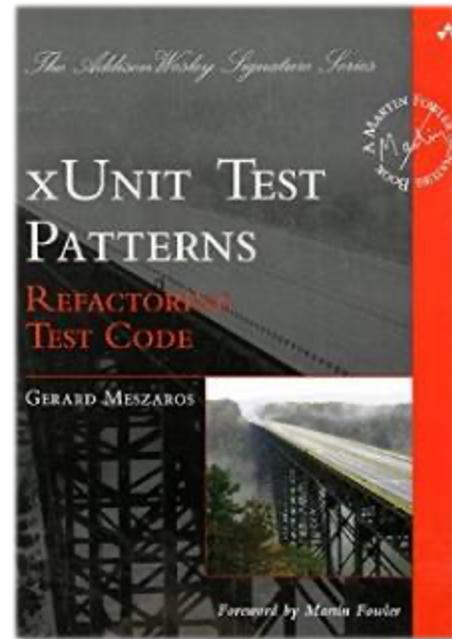
## IV. Mocking

- Test Doubles 개요
- Mocking 예제 및 실습

# Mocking & Stabbing

- 1. Test Doubles 이란?**
- 2. Component Isolation (요소 분리)**
- 3. Behavior Injection (행동 주입)**
- 4. Behavior Verification (행동 검증)**

# Test Doubles



## When to use What?

<b>Test Double</b>	<b>Testing Type</b>	<b>Can Directly Fail Test ?</b>
Stub	State Verification	No
Mock	Behavior Verification	Yes
Spy	Behavior Verification	Depends on framework

# Component Isolation

- **Problem:**
  - My code's dependencies cause my tests to fail unexpectedly.
  - (내 코드의 종속성으로 인해 테스트가 예기치 않게 실패합니다)

# Component Isolation

- Test doubles **isolate** the system under test (SUT) from its dependencies. (테스트 더블은 테스트중인 시스템 (SUT)을 종속성에서 분리합니다.)

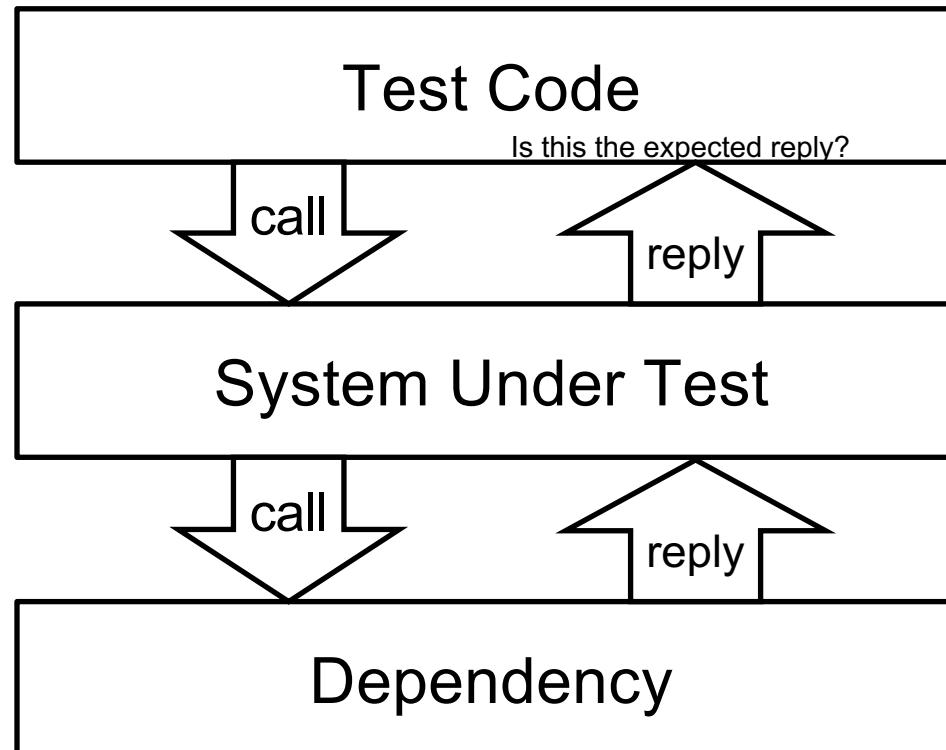
## Test with Real Dependency

Test Code

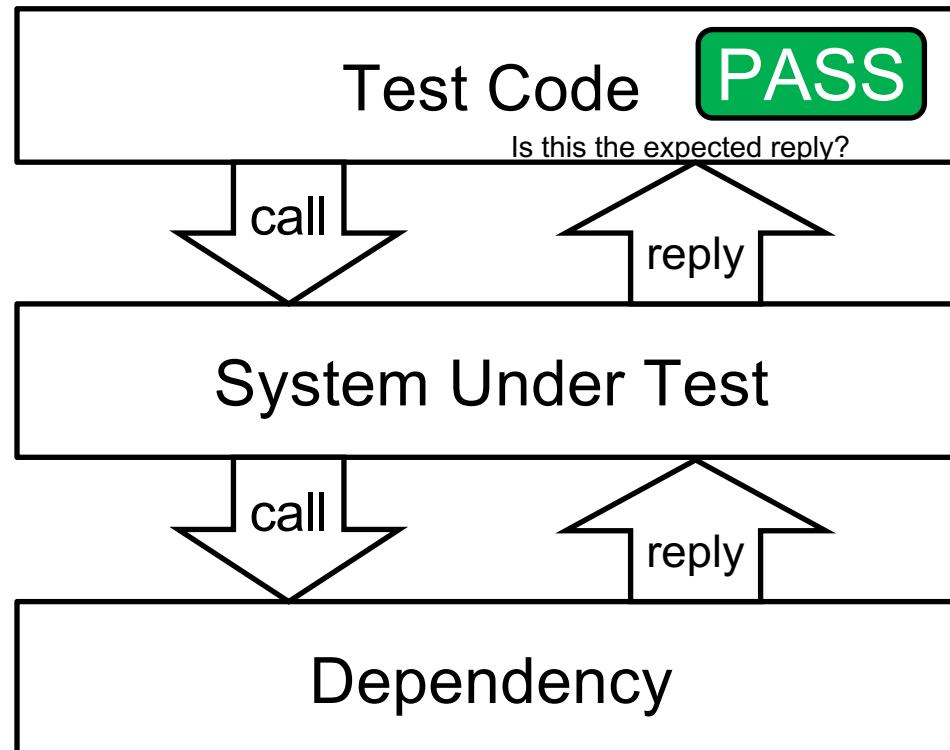
System Under Test (SUT)

Dependency

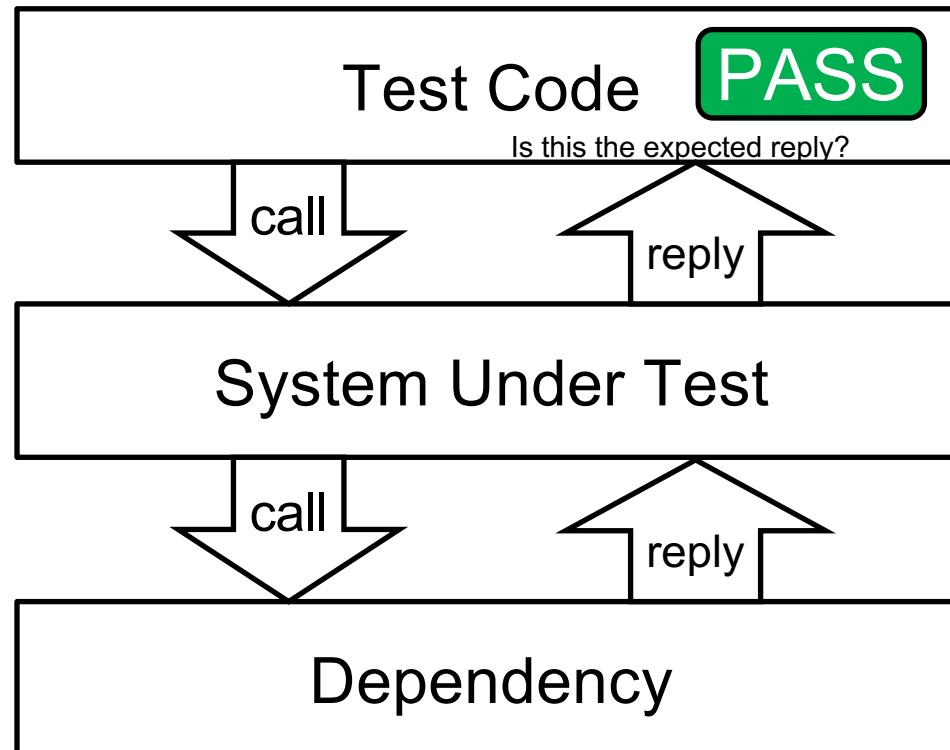
## Test with Real Dependency



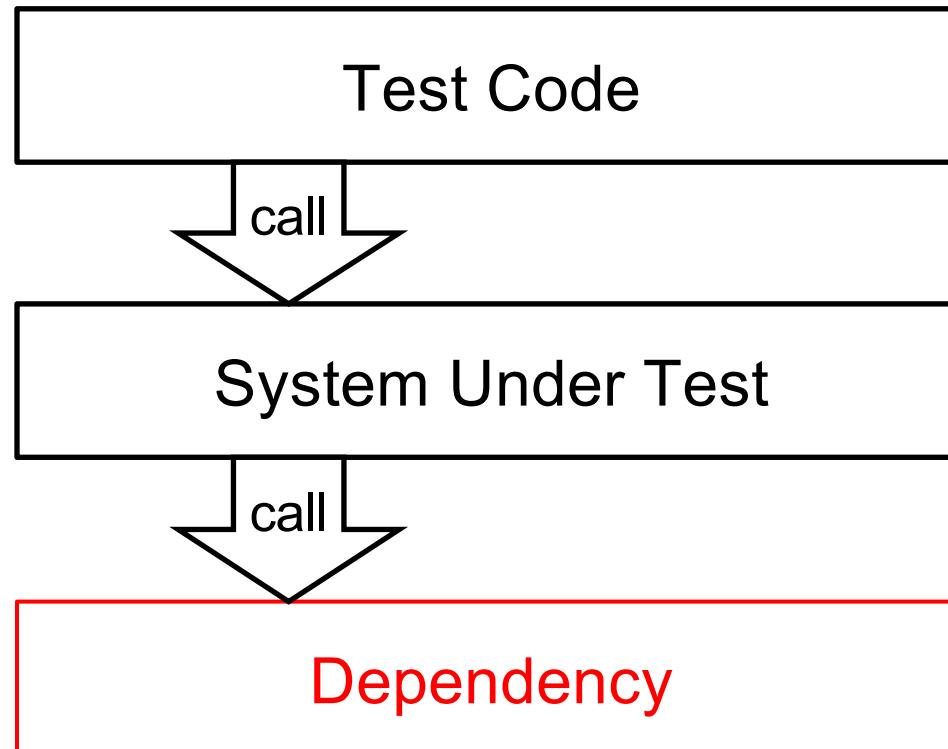
## Test with Real Dependency



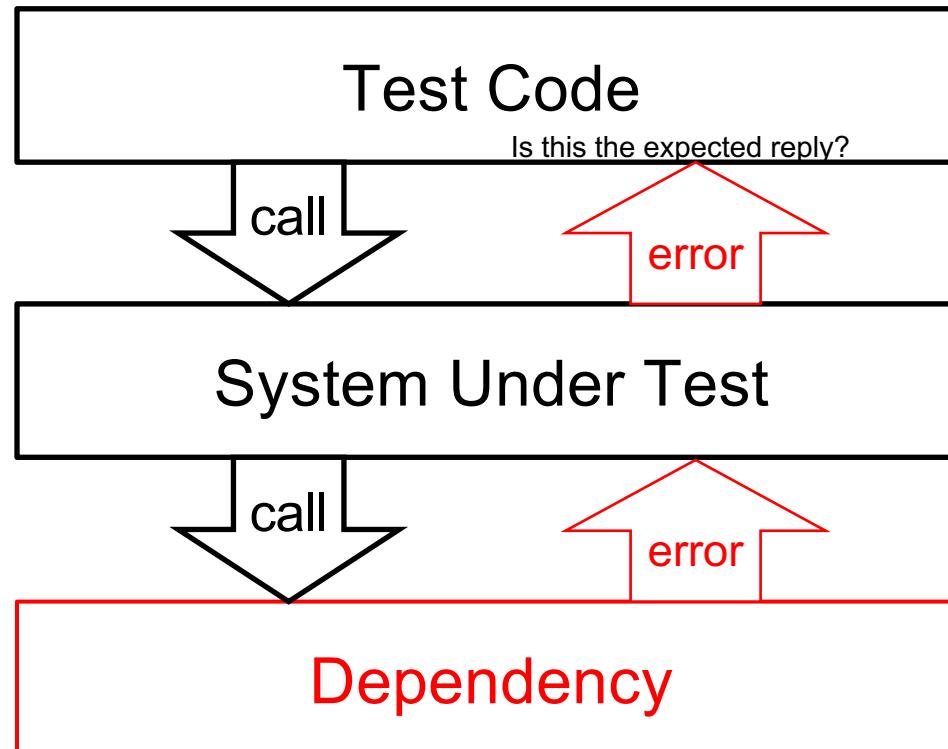
## Test with Real Dependency



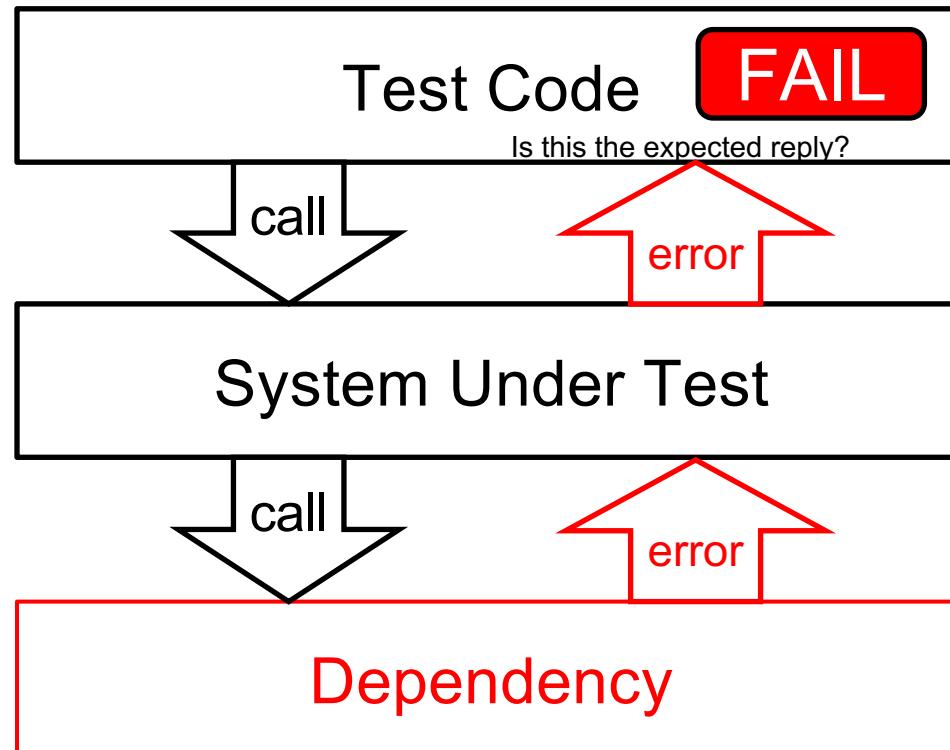
## Dependency Error



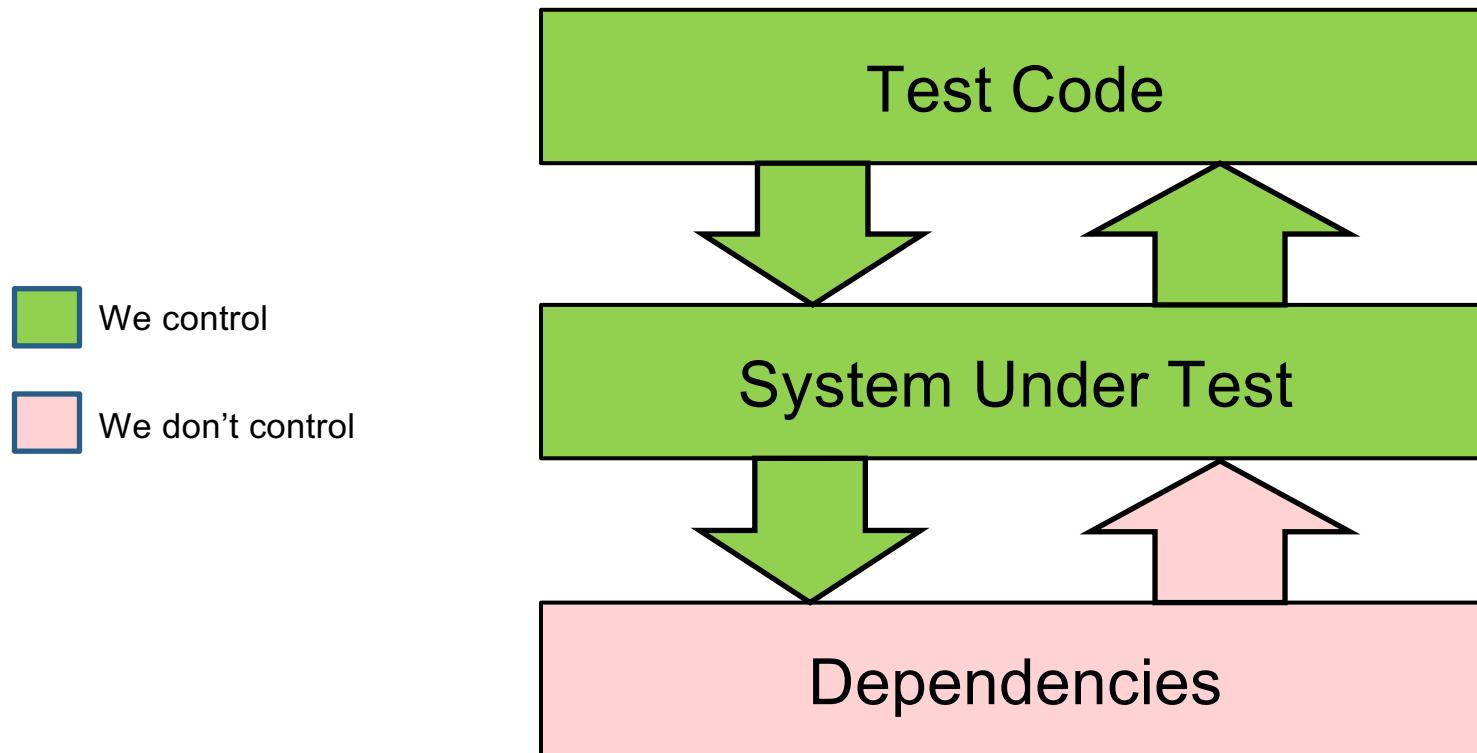
## Dependency Error



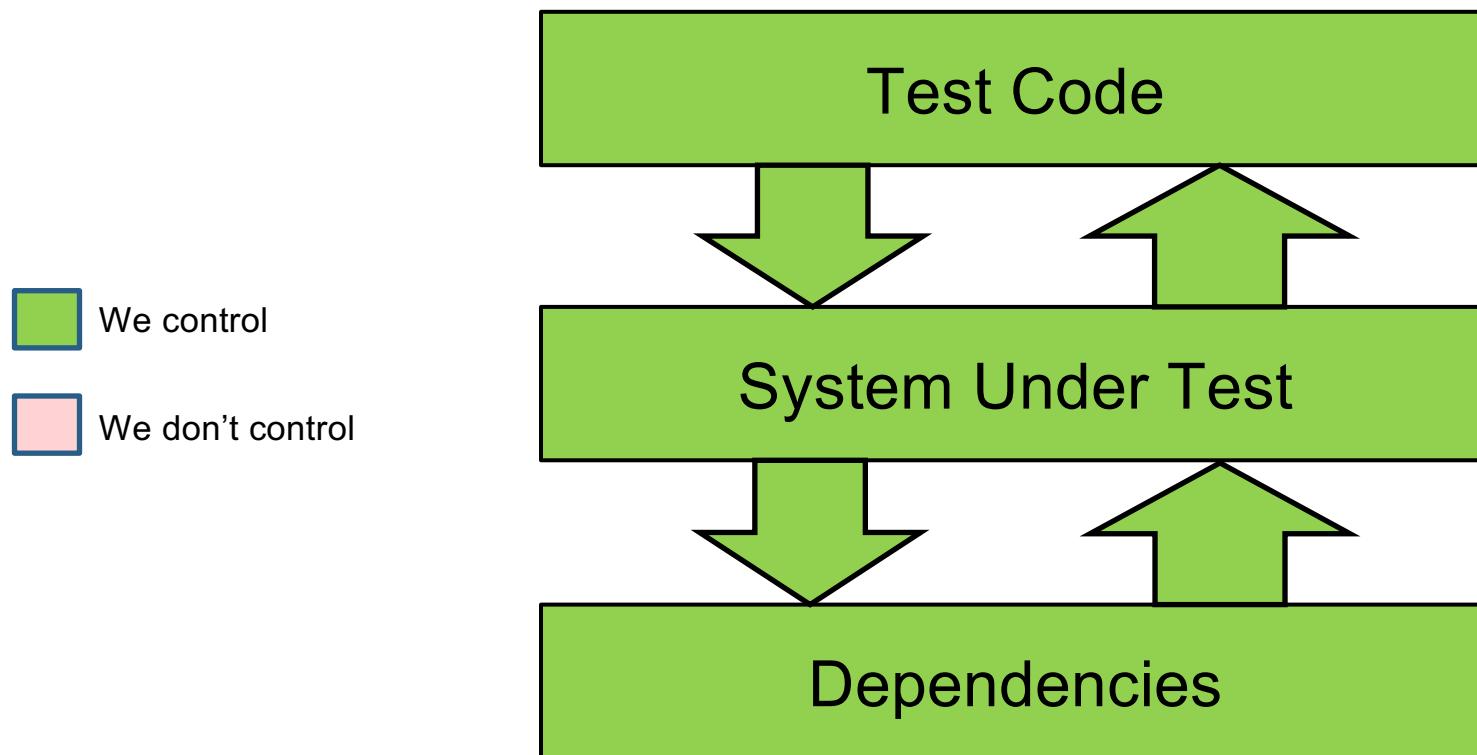
## Dependency Error



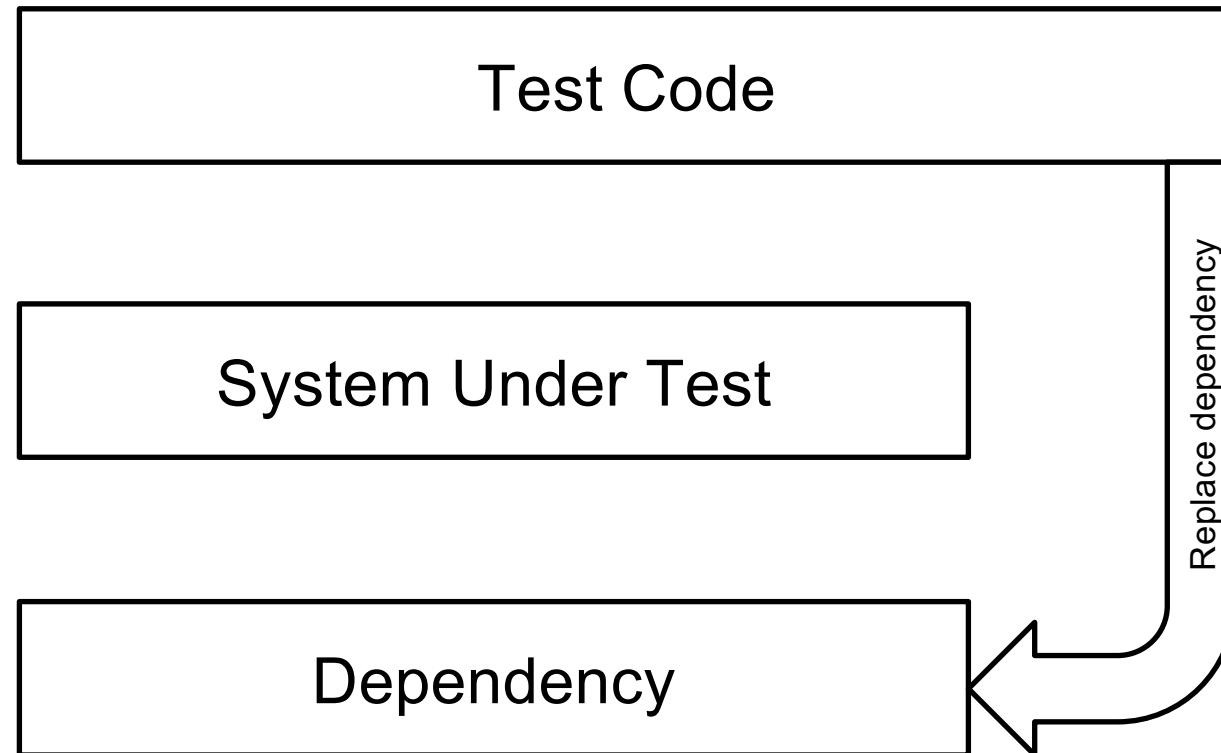
## Isolating the SUT



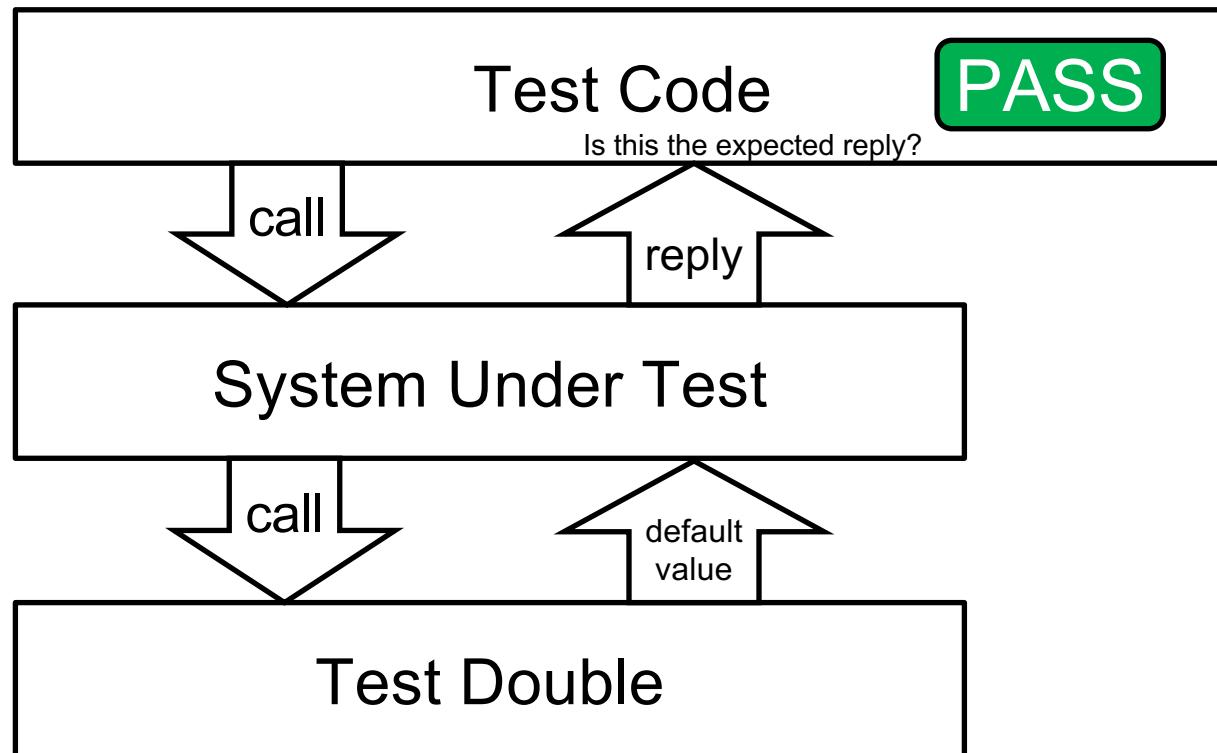
## Isolating the SUT



## Isolating the SUT



## Isolating the SUT



# Behavior Injection

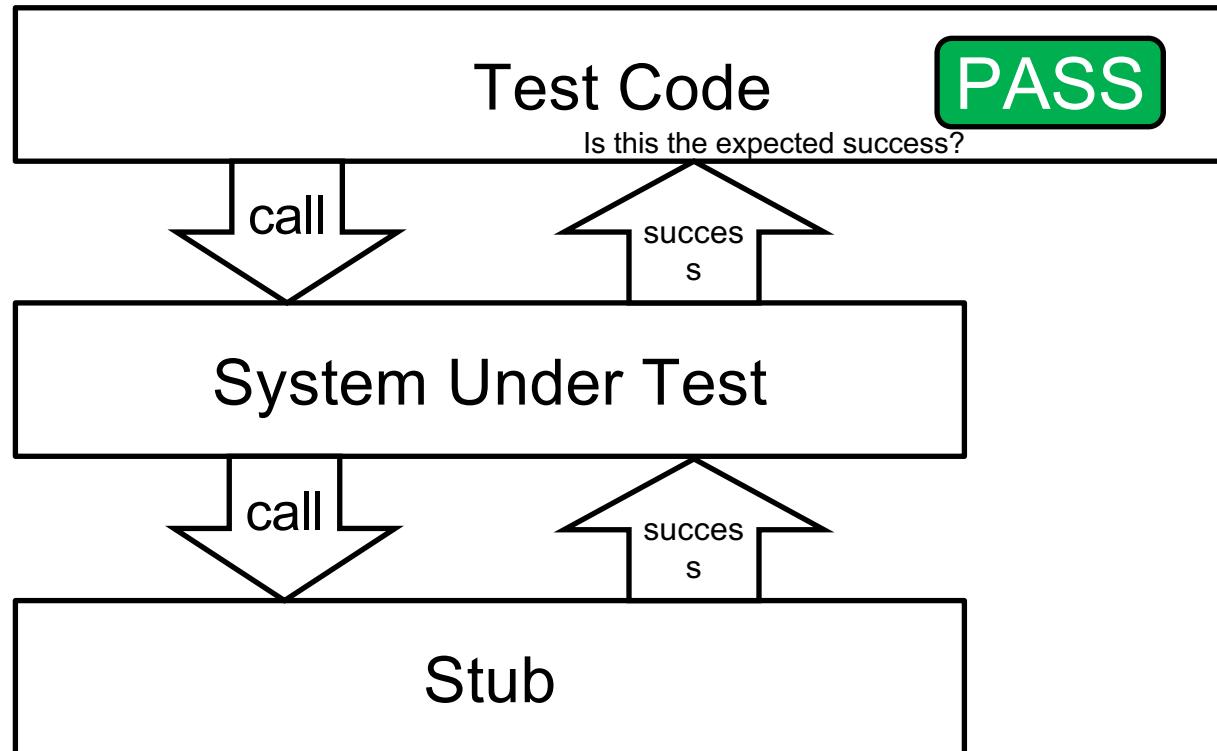
- **Problem:**

- I need my dependencies to behave in a specific way to fully test my SUT.

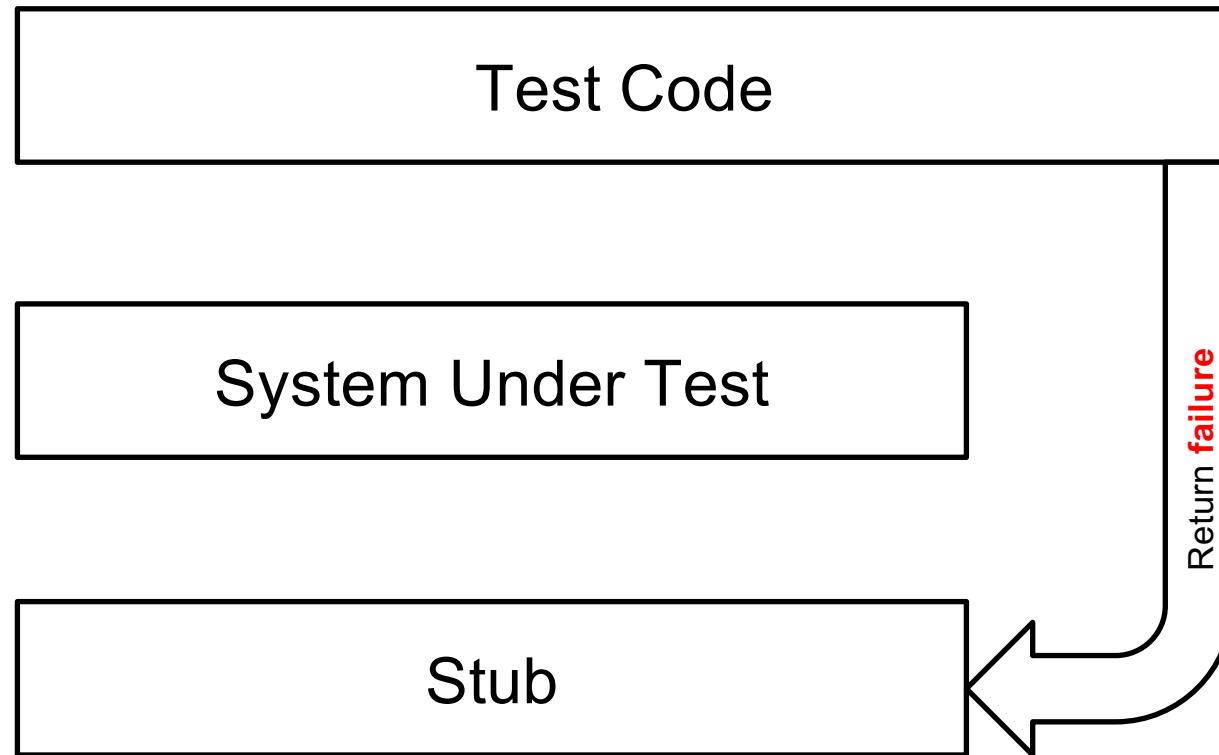
# Behavior Injection

- Stubs let you inject behavior.

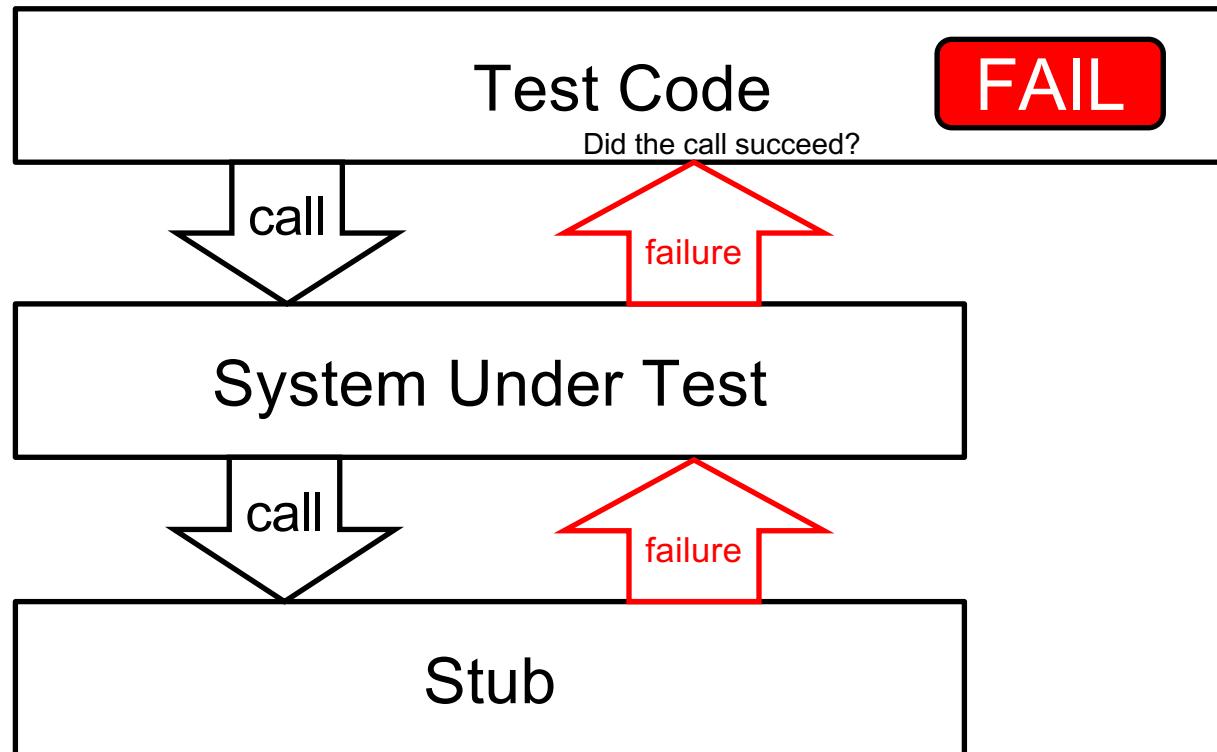
## Injecting a Value



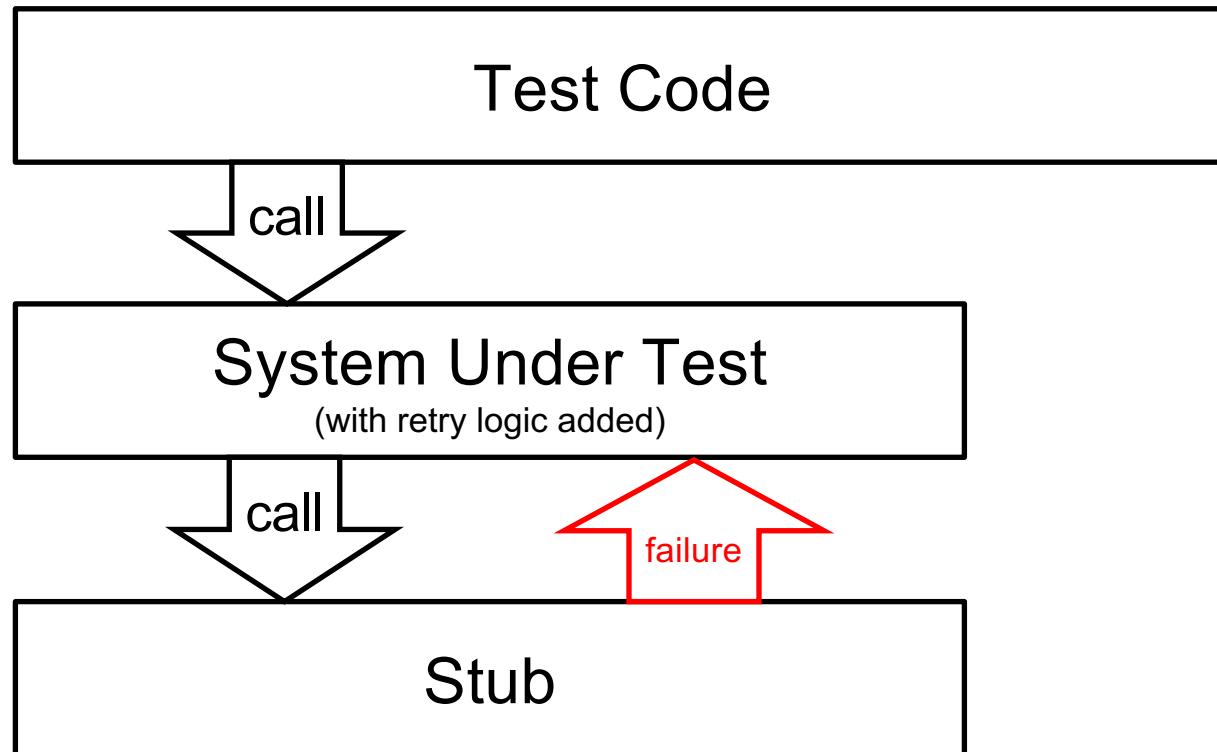
## Injecting a Failure



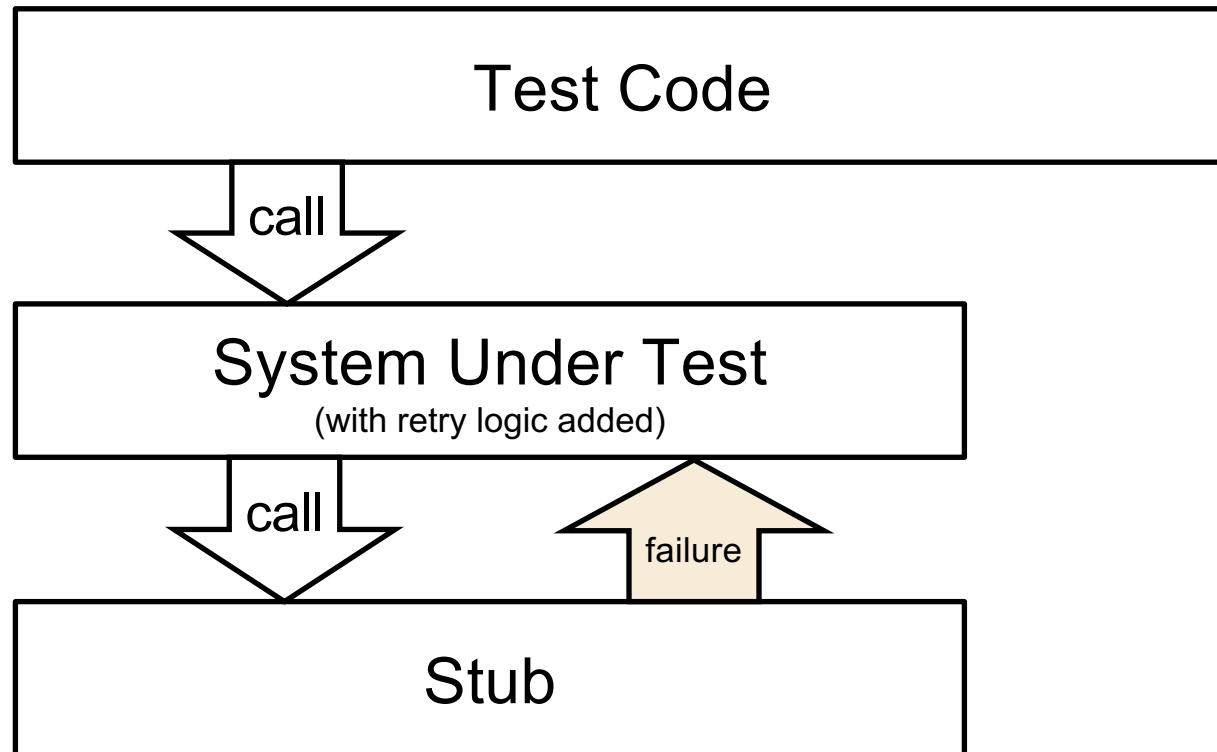
## Injecting a Failure



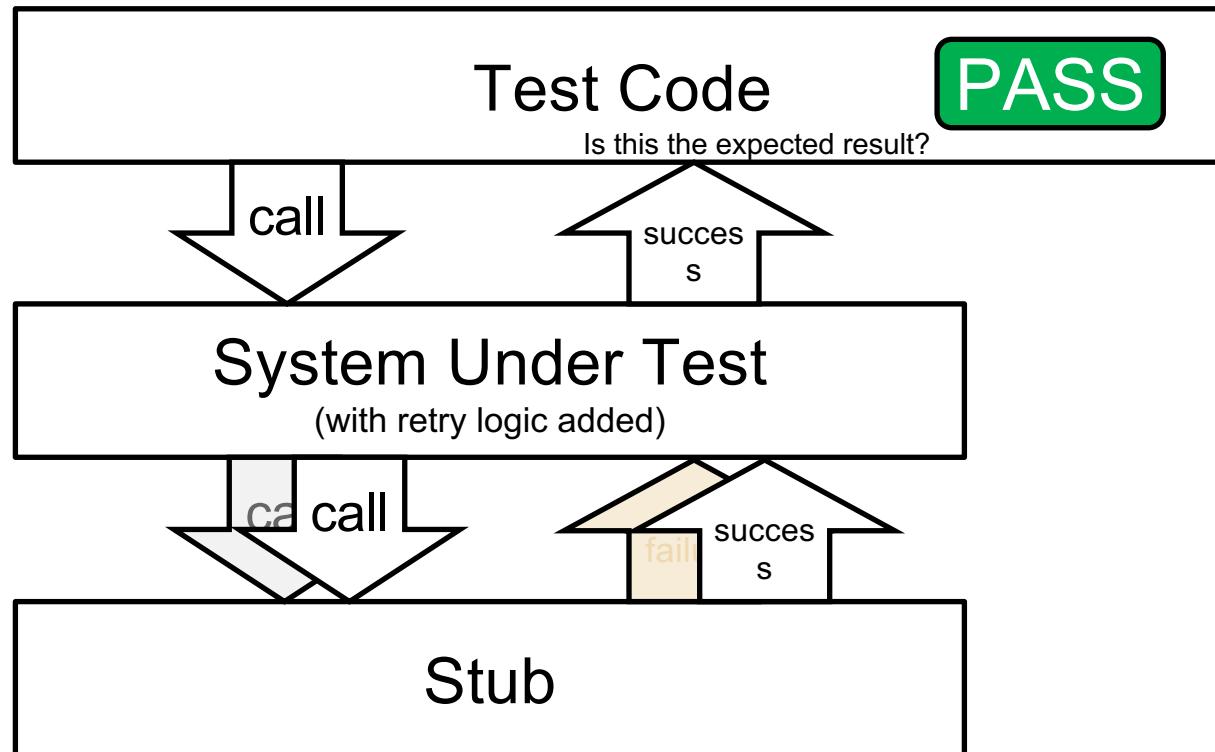
## Injecting Multiple Behaviors



## Injecting Multiple Behaviors



## Injecting Multiple Behaviors

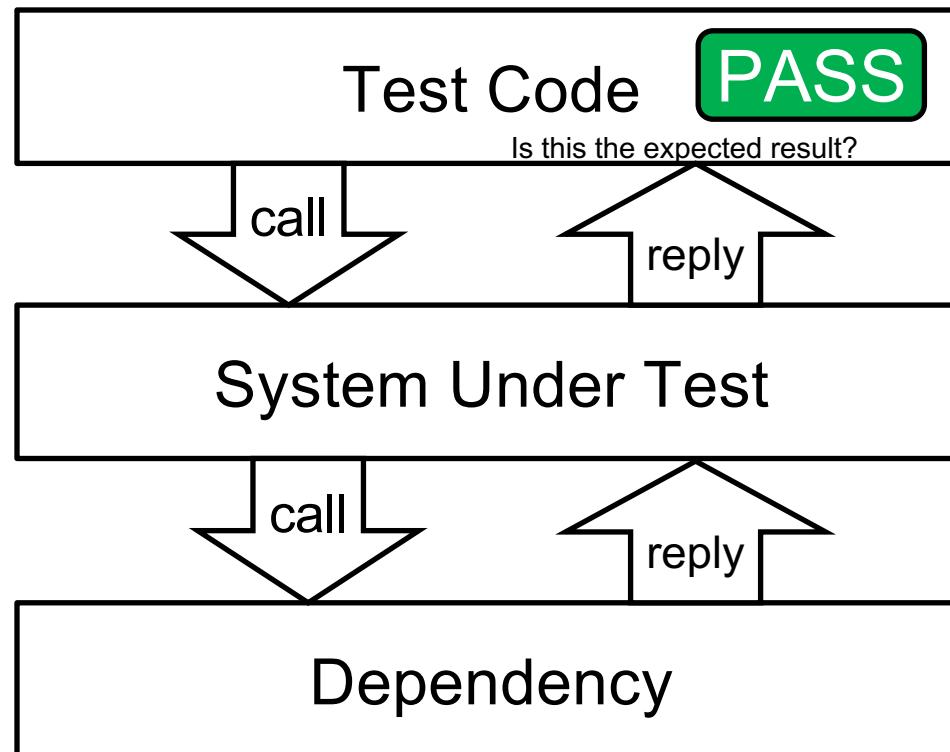


# Verification Methods

- **State Verification**
- **Behavior Verification**
- **State verification tests code by inspecting an object's state after a method call.**

**State verification tests code  
by inspecting an object's state  
after a method call.**

## Benefits of State Verification

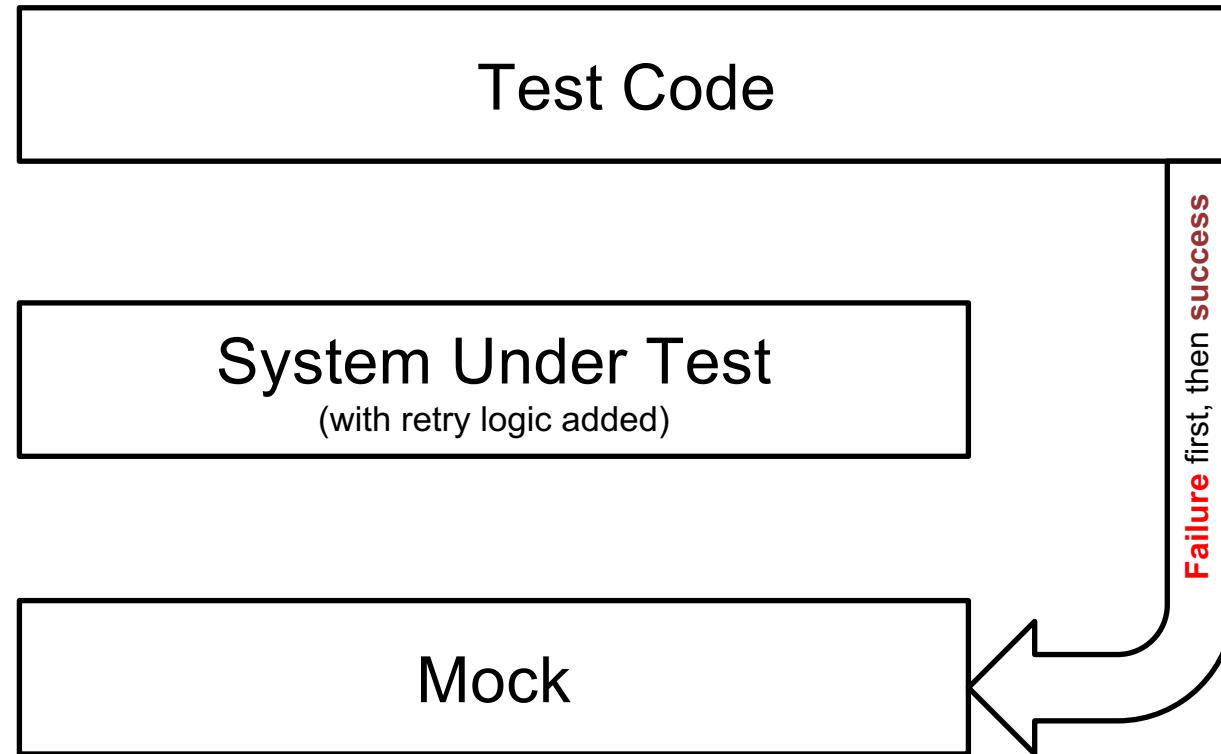


**Behavior verification involves using a mock or spy to verify interactions with dependencies.**

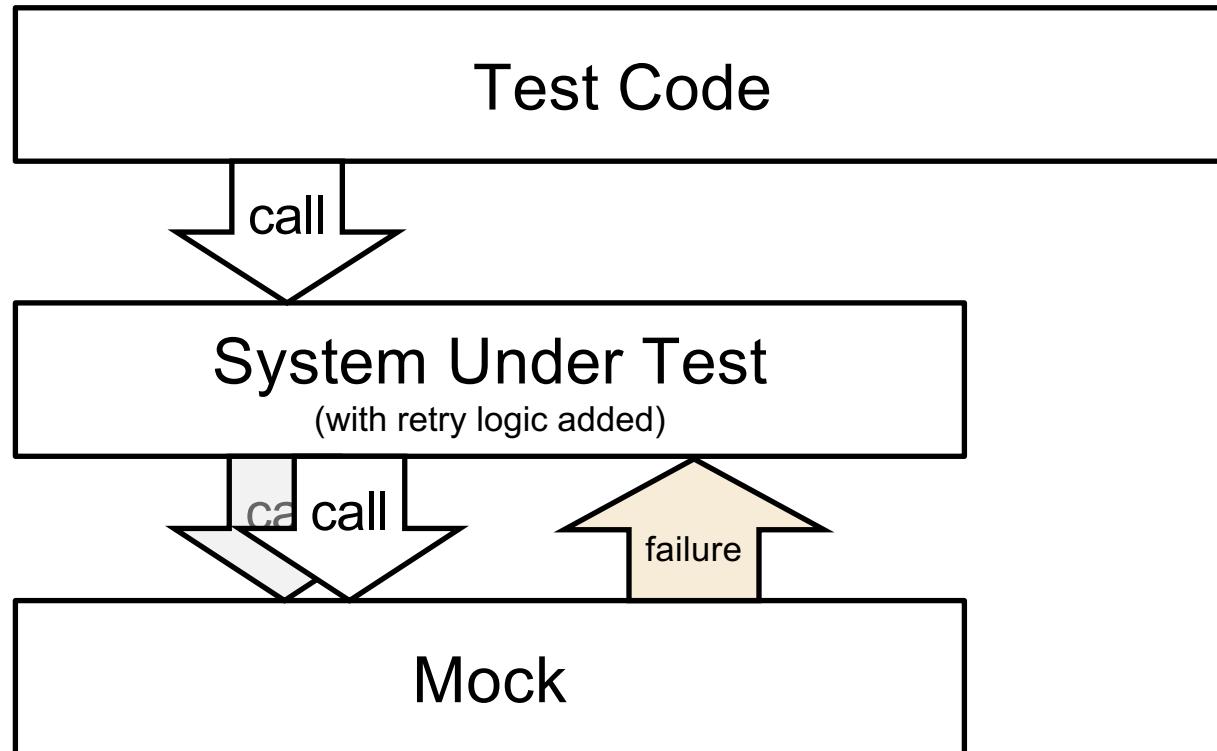
# Problem

**There's no state to verify and  
you can predict values in advance.**

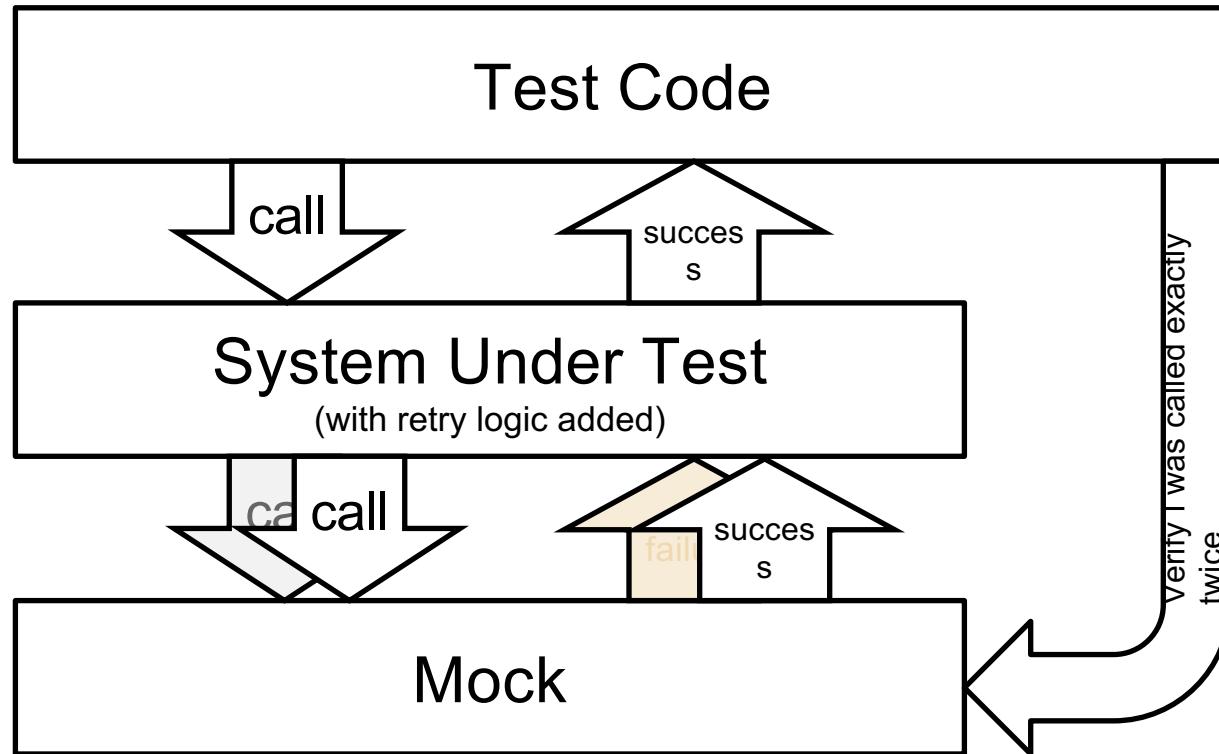
## Behavior Verification via Mock



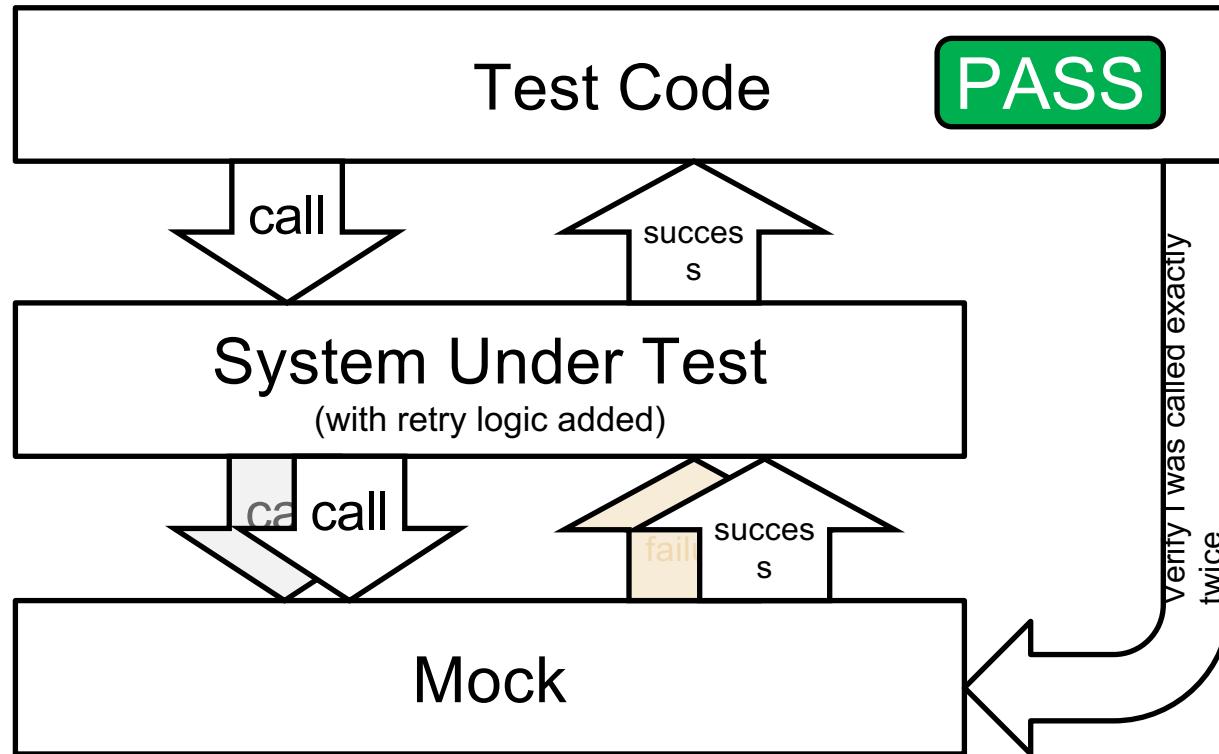
## Behavior Verification via Mock



## Behavior Verification via Mock

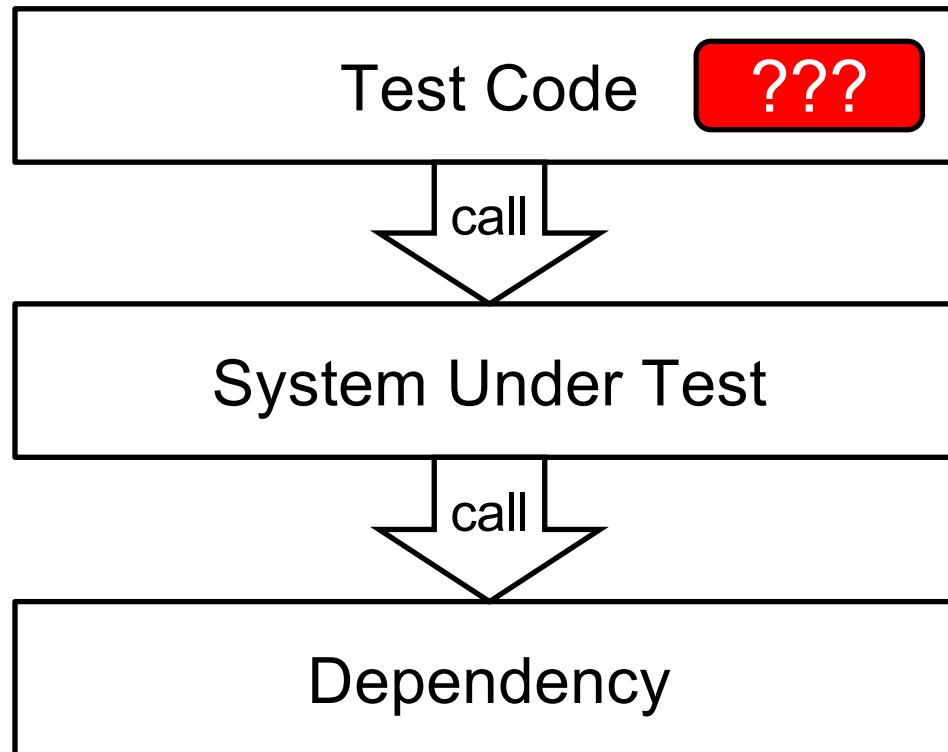


## Behavior Verification via Mock



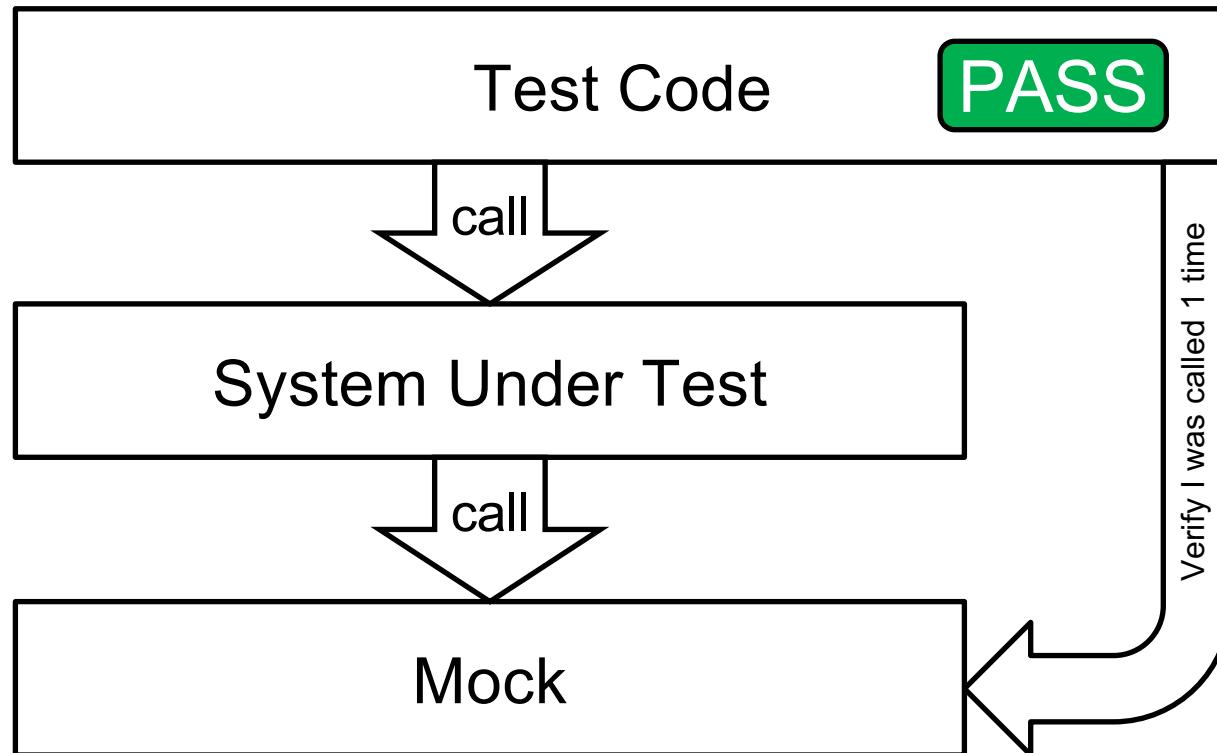
**What if your system under test  
doesn't get a response?**

## No Response

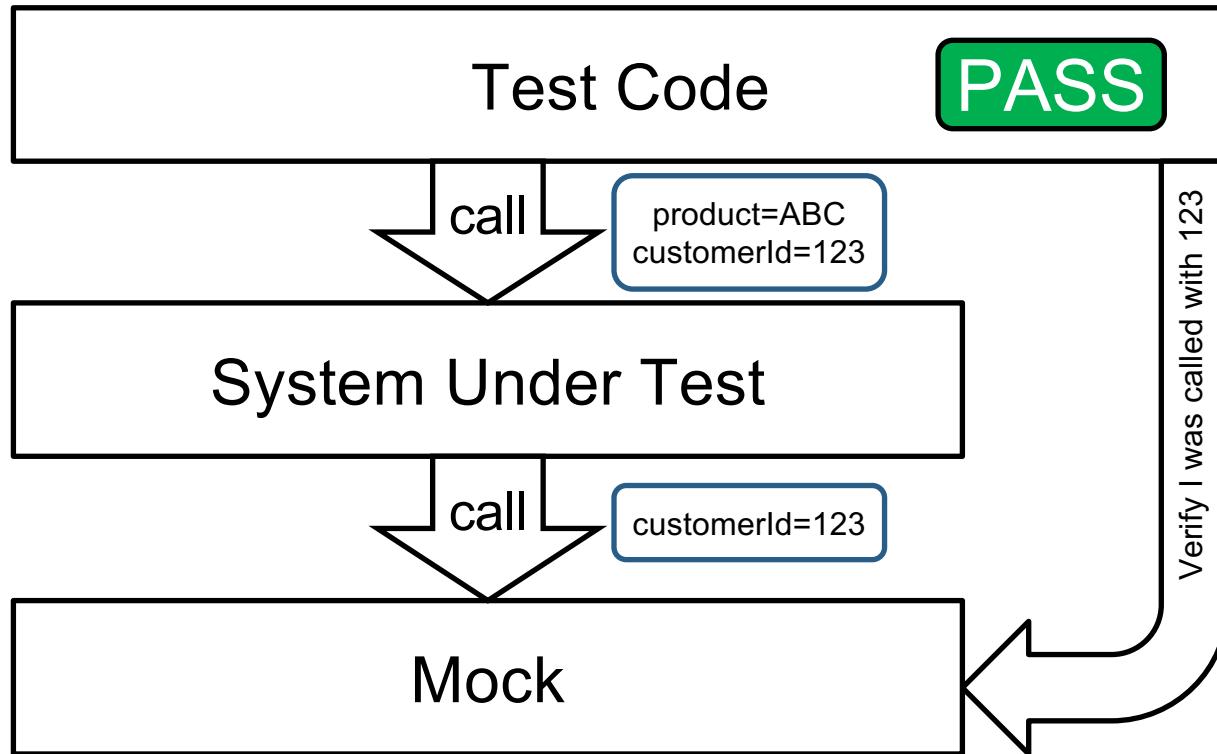


**If there's no state to inspect,  
you can still use behavior verification.**

## Verifying Void Method



## Verifying Specific Arguments



# Problem

- There's no state to verify and you can't predict values in advance.

**Spies can retrieve what you  
send to a dependency.**

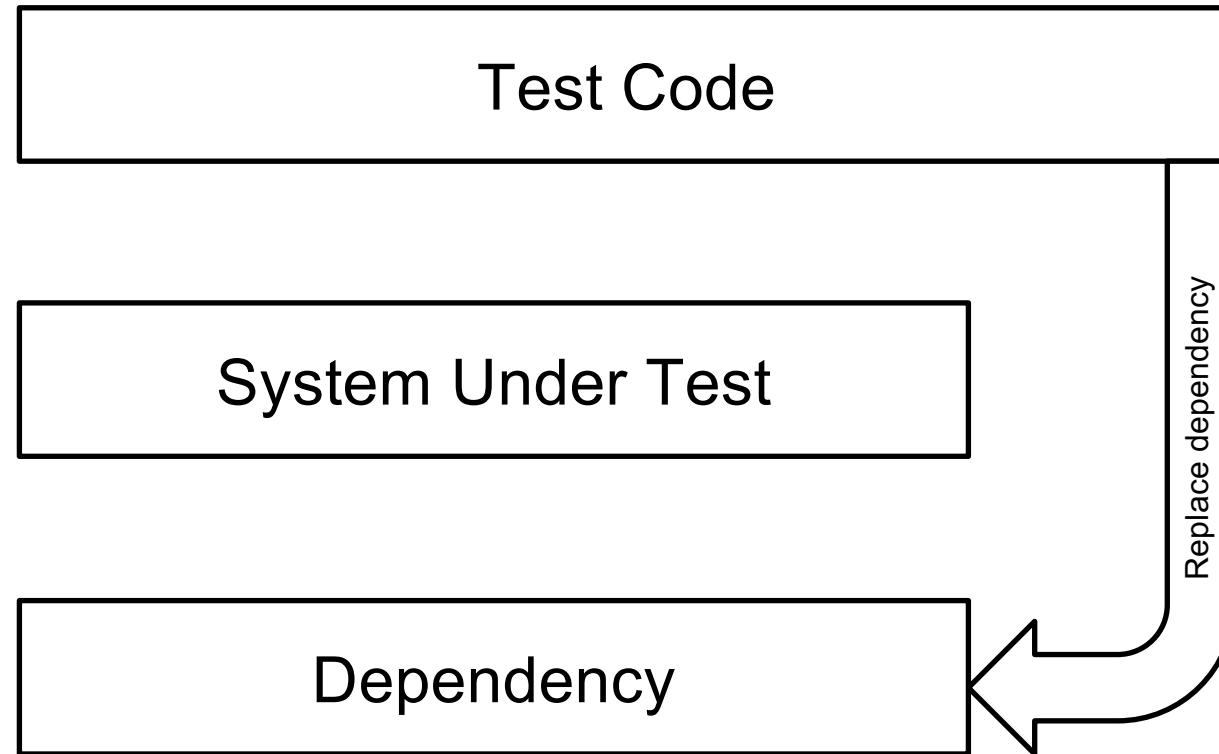
## Behavior Verification via Spy

Test Code

System Under Test

Dependency

## Behavior Verification via Spy



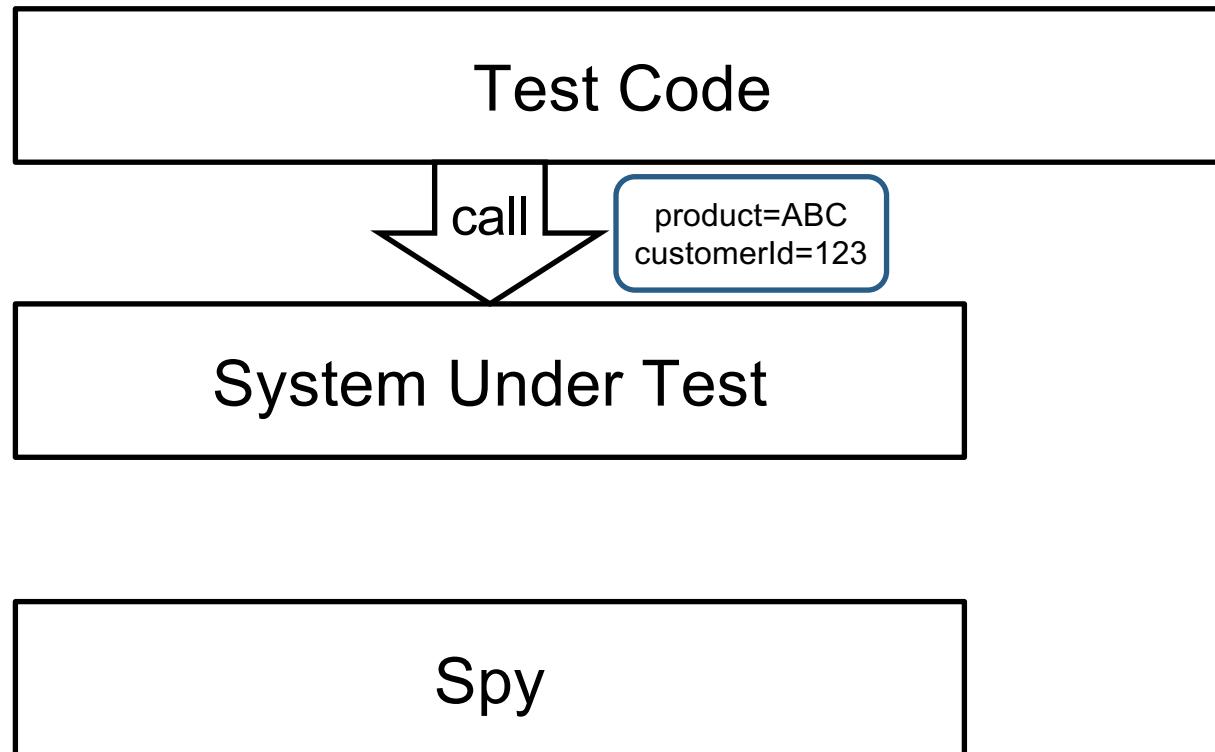
## Behavior Verification via Spy

Test Code

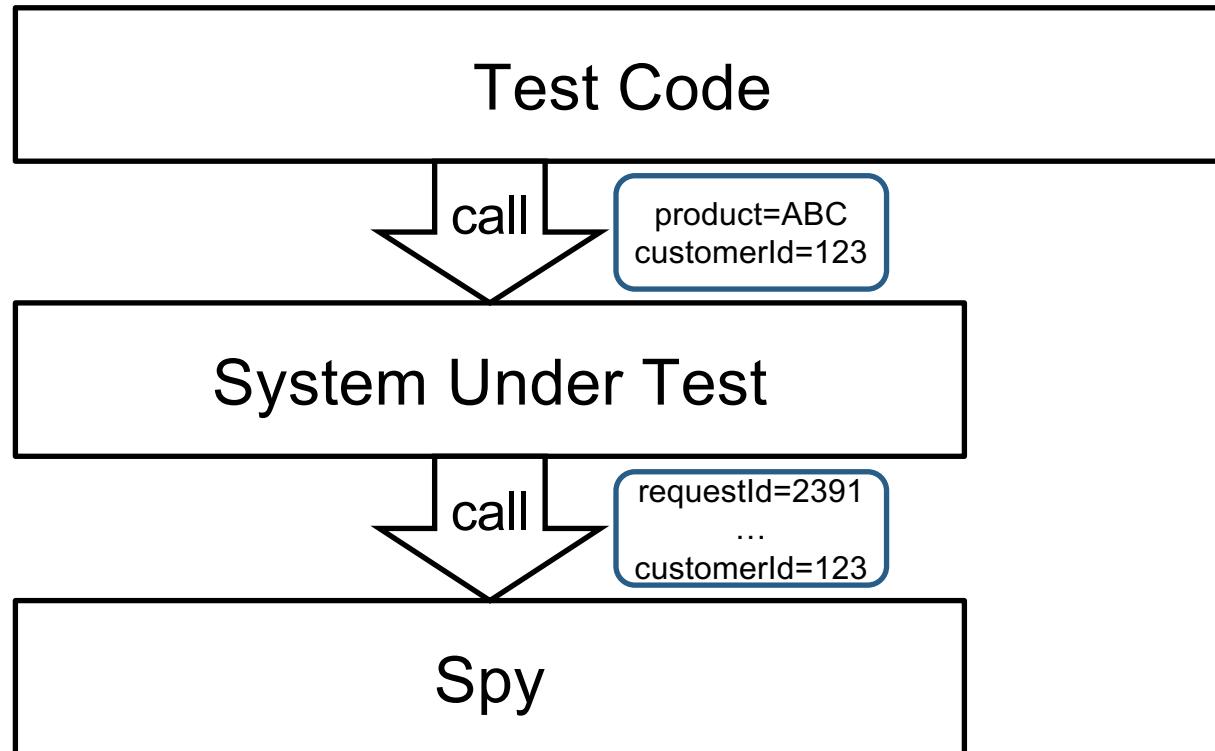
System Under Test

Spy

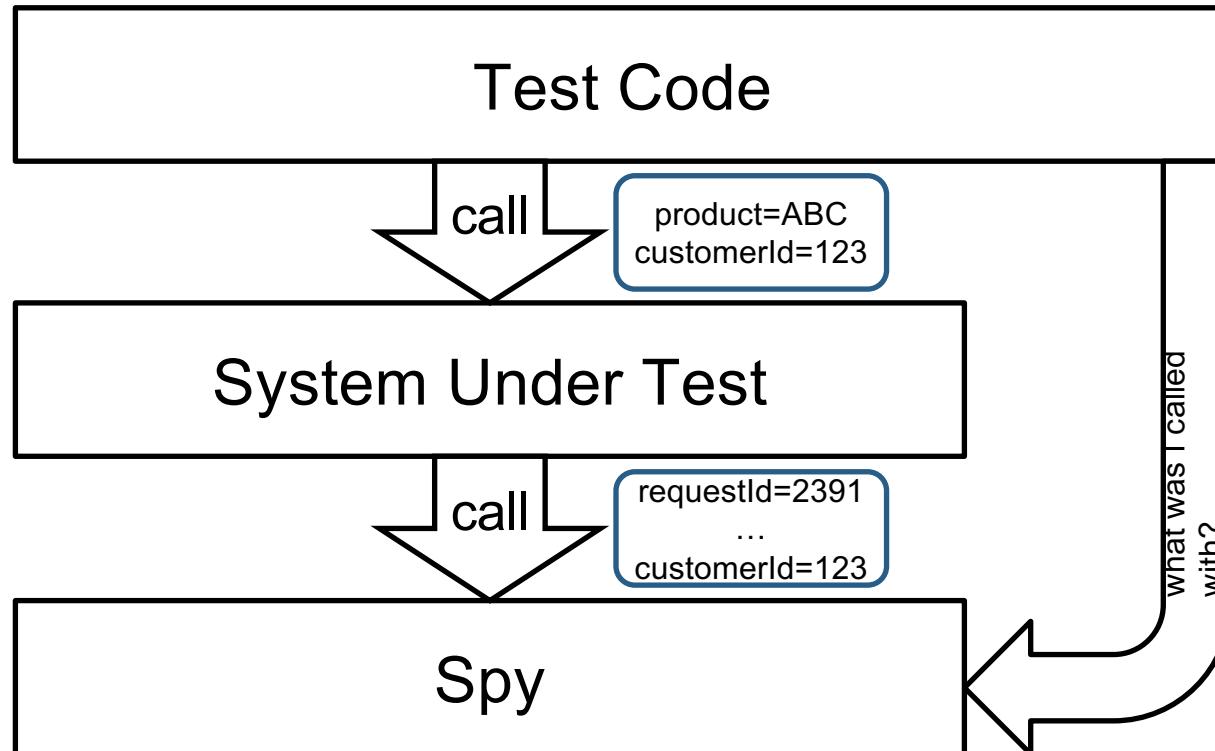
## Behavior Verification via Spy



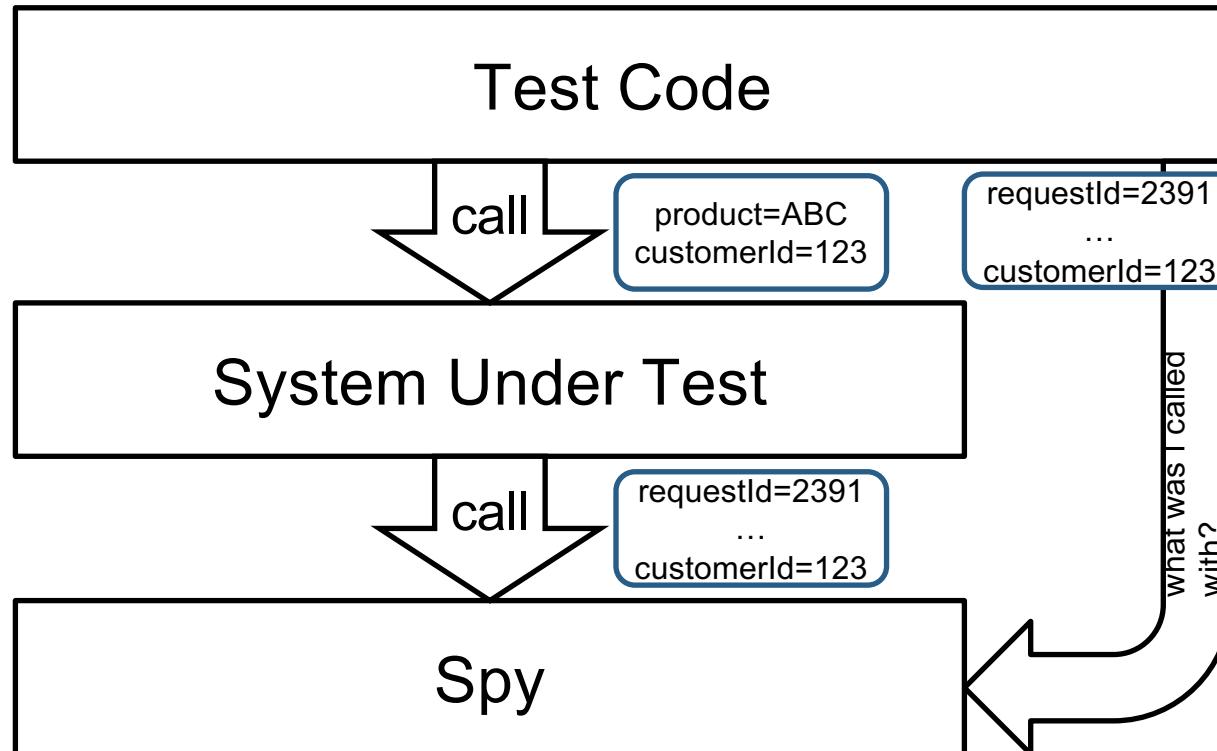
## Behavior Verification via Spy



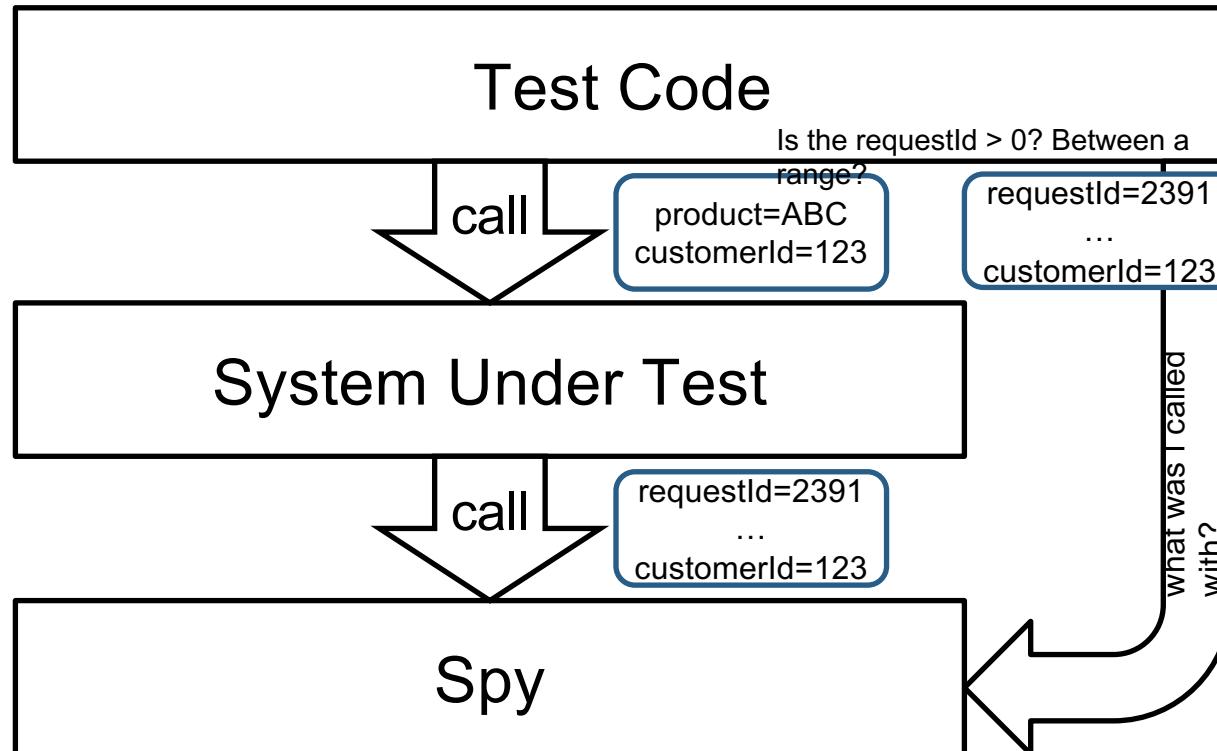
## Behavior Verification via Spy



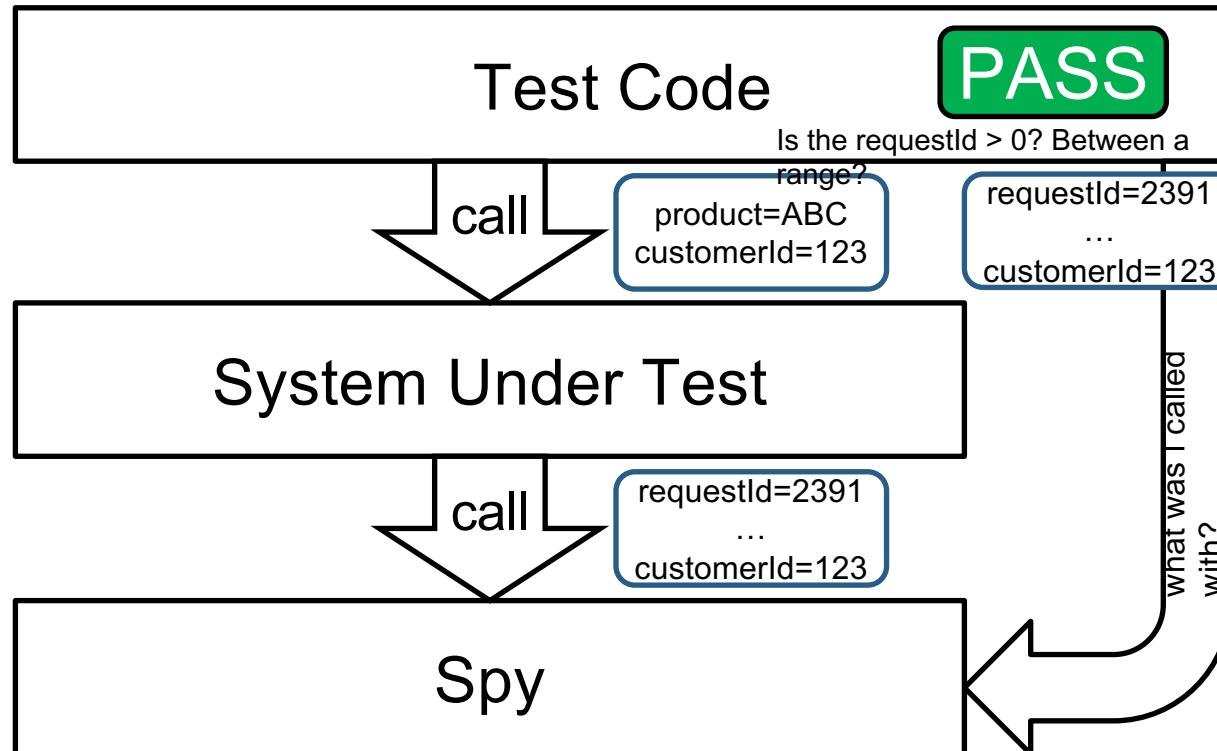
## Behavior Verification via Spy



## Behavior Verification via Spy



## Behavior Verification via Spy



# V. Testability

- Simplicity
- Open/Closed
- Liskov Substutability
- Interface Segregation
- Dependency Injection

# What is Testability?

- **Software is *testable*:**
  - if it can be tested in different contexts.
- **High correlation between:**
  - A testable code and a good design.
  - A testable code and quality.
- **Testability is a property of code.**

# SW 설계 품질 – 잘못된 설계로 나타날 수 있는 현상들

잘못된 설계의 냄새	내용
경직성 (Rigidity)	하나의 변경에 따라 다른 곳도 변경해야 하는 곳이 연쇄적으로 발생하는 경우
깨지기 쉬움 (Fragility)	시스템에서 한 부분을 변경할 때 전혀 상관없는 다른 부분이 작동을 멈추는 경우
부동성 (Immobility)	시스템을 여러 컴포넌트로 분해해서 다른 시스템에 재사용하기 어려운 경우
끈적함 (Viscosity)	개발환경이 과도하게 연결되어, 편집-컴파일-테스트 순환 시간이 오래 걸리는 경우
과도한 복잡도 (Needless Complexity)	현재 사용되지는 않지만, 향후 사용될지도 모른다고 예상되는 기능까지 무리하게 반영한 경우
불필요한 반복 (Needless Repetition)	비슷한 기능이 코드의 여러 곳에 반복되는 경우
불투명 (Opacity)	코드의 작성 의도가 명확하게 설명되지 않는 경우

\* Ref: UML for Java Programmers, Robert C. Martin, 2002

## SW 설계 품질 – 좋은 설계를 위한 원칙들

- 원칙1 단일 책임 원칙 (Single Responsibility Principle)
  - 어떤 클래스를 변경해야 하는 이유는 오직 하나 뿐이어야 한다.
  - 하나의 클래스에 너무 많은 기능을 포함하지 않아야 함

## SW 설계 품질 – 좋은 설계를 위한 원칙들

- 원칙2 개방-폐쇄 원칙 (Open-Closed Principle)
  - SW 앤티티(클래스, 모듈, 함수 등)는 확장에 대해서는 개방되어야 하지만, 변경에 대해서는 폐쇄되어야 한다.
  - 모듈 자체를 변경하지 않고도 그 모듈을 둘러싼 환경을 변경할 수 있어야 함
  - 예) 모델-뷰-컨트롤러 패턴

## SW 설계 품질 – 좋은 설계를 위한 원칙들

- 원칙3 리스코프 교체 원칙 (Liskov Substitution Principle)
  - 서브타입은 언제나 자신의 기반 타입(base type)으로 교체할 수 있어야 한다.
  - instanceof, down casting을 과도하게 사용하는 경우 바람직하지 않음

## 설계 품질 – 좋은 설계를 위한 원칙들

- 원칙4 인터페이스 격리원칙 (Interface Segregation Principle)
  - 클라이언트는 자신이 사용하지 않는 메서드에 의존 관계를 맺으면 안 된다.
  - 어떤 객체의 사용자에게 그 사용자에게 필요한 메서드만 인터페이스를 제공해야 함

## SW 설계 품질 – 좋은 설계를 위한 원칙들

- 원칙5 의존관계 역전 원칙 (Dependency Inversion Principle)
  - 고차원 모듈은 저차원 모듈에 의존하면 안된다. 이 두 모듈 모두 다른 추상화된 모듈에 의존해야 한다.
  - 추상화된 것은 구체적인 것에 의존하면 안된다. 구체적인 것이 추상화된 것에 의존해야 한다.
  - 자주 변경하는 concrete class 대신 인터페이스/추상클래스에 의존해야 함

# Benefits of “testable” code

- Execution in isolation
  - Code review
  - Code readability/maintenance
  - Less costly to test
- 
- These things result in code with fewer defects.

# Where to start?

- Requirements
- Architecture
- Documentation
- Class Design/Implementation
- We focus on implementation.

# SOLID

- Design principles for Testability

# Single Responsibility

- Can you describe what it does in one sentence?

# Single Responsibility

- Can you describe what it does in one sentence?
- Cohesive components don't require “and”
- Classes have only one reason to change
- Functionality should not be duplicated somewhere else

# Single Responsibility

- **Warning signs:**
  - High churn rate in a single file
  - Dependencies unused in some code paths
  - Conditional logic outside of algorithms

# Single Responsibility 실습

```
class StreamAnalyzer {  
    public List<Double> findKLargest(int k) {  
        Iterator<Double> s = getPrices();  
        Heap<Double> h = new MaxHeap<>();  
        while (s.hasNext()) {  
            h.add(s.next());  
        }  
        return h.stream().limit(k).collect();  
    }  
  
    private Iterator<Double> getPrices() {  
        // Open a connection  
        Connection conn = DoubleService.create();  
        // Create a stream around it  
        return conn.iterator();  
    }  
}
```

What's the problem?

# Single Responsibility - 장단점

- 장점
  - Clearer code reviews
  - Easier failure diagnosis
  - Lower likelihood of change
- 단점
  - More classes
  - More indirection

# Open/Closed

- Do your types change to add features?

# Open/Closed

- Do your types change to add features?
- Be open for extension and composition
- Publish contract via an interface
- Well defined contract (input/output types)

# Open/Closed 실습

```
Payment checkOut(List<Items> items, PaymentMethod pm) {  
    // calculate totals...  
  
    if (credit) {  
        return pm.chargeCredit(total);  
    } else {  
        return pm.chargeCash(total);  
    }  
}
```

# Open/Closed

- **Warning signs:**
  - **Adding suspiciously similar methods**
  - **Data is protected, not private**
  - **Objects not immutable after construction**

# Open/Closed 장단점

- 장점
  - Simpler to reason about state
  - Fewer reasons to change the original type
- 단점
  - Changing interfaces is hard
  - Requires a clear contract
  - Prevents short-term “workarounds”

# Substitutability

- Is the type contract the same after inheritance?

# Substitutability

- Is the type contract the same after inheritance ?
- Parents and children are interchangeable
- All inheritance is additive
- All children pass their parent's tests

# Substitutability

- Warning signs:
- Subclasses ignore functionality in parents
- Use of runtime type information
- Subclasses restrict type invariants

# Substitutability 실습

```
public class Person {      public class PersonTest {  
    int xPos = 0;          @Test  
    int yPos = 0;          public void canMove() {  
  
        void moveX(int dx) {  Person p = getPerson();  
            xPos += dx;       p.moveX(2);  
        }                     assertEquals(2, p.xPos);  
  
        void moveY(int dy) {  }  
            yPos += dy;  
        }  
    }  
}
```

# Substitutability 장단점

- 장점

- Having contracts for your types
- Subclasses receive test coverage “for free”
- Wide hierarchies prevent runtime “yoyo”

- 단점

- Defining contracts for your types
- Results in wider hierarchies

# Interface Segregation

- Does every client use your entire interface?

# Interface Segregation

- Does every client use your entire interface?
- Don't force clients to use unneeded things
- Define your own interface

# Interface Segregation

- **Warning signs:**
- **Clients only use some methods**
- **You have lots of stubs that return null**
- **No cohesion**

# Interface Segregation 실습

```
interface BigInterface {  
    void doSomething();  
    String getSomething();  
    double calculateThing1();  
    double calculateThing2();  
    String getSomethingElse();  
    ...  
}
```

# Interface Segregation 장단점

- 단점
  - More, smaller interfaces
  - More indirection (adapter)
  - Must pick a pivot
- 장점
  - Test doubles easier to make
  - Implementations are cheaper

# Dependency Inversion

- Can everything you depend on be replaced?

# Dependency Inversion

Can everything you depend on be replaced?

- Ask for things, don't create them
- Ask for the most generic version you can use

# Dependency Inversion 실습

```
class DataHandler {  
    DatabaseProvider provider;  
  
    DataHandler() {  
        provider = new DatabaseProvider(...);  
    }  
    ...  
}
```

# Dependency Inversion

## Warning signs:

- Declared types are specialized
- Use of the new operator
- Extending rather than implementing

# Dependency Inversion 장단점

## 장점

- Easier injection of test doubles
- Separation of concerns (creation vs. use)
- Decoupling from implementations

## 단점

- Push object graph creation up the stack
- More up-front work worrying about plumbing
- More indirection
  - Dependency Injection

## VI. BDD

- BDD 개요
- BDD 예제 및 실습

# BDD

- **Behavior Driven Development is an extension of TDD**
- **Focuses on**
  - Where to start in the process
  - What to test and what not to test
  - How much to test in one go
  - What to call the tests
  - How to understand why a test fails

# BDD

- **Gives a common language for all stakeholders: the product owner, business analysts, PMs, QAs, developers, etc.**
- **Written in natural language**
- **The test cases are the scenarios so no need to write a separate test cases**
- **The documentation is up-to-date**

## BDD Example

Given a 5 by 5 game  
When I toggle the cell at (3, 2)  
Then the grid should look like

.....

.....

.....

..X..

.....

## BDD Code Implementation (Java)

```
@Given("a $width by $height game")
public void theGameIsRunning(int width, int height) {
    game = new Game(width, height);
    renderer = new StringRenderer();
    game.setObserver(renderer);
}

@When("I toggle the cell at ($column, $row)")
public void iToggleTheCellAt(int column, int row) {
    game.toggleCellAt(column, row);
}

@Then("the grid should look like $grid")
public void theGridShouldLookLike(String grid) {
    assertThat(renderer.toString(), equalTo(grid));
}
```

# Cucumber

- **BDD framework**
- **Uses Gherkin language**
  - **GIVEN: Setup**
  - **WHEN: test execution**
  - **THEN: Asserts, Verifications**

# Cucumber Example

**Feature:** Withdraw Money from ATM

A user with an account at a bank would like to withdraw money from an ATM.

Provided he has a valid account and debit or credit card, he should be allowed to make the transaction. The ATM will tend the requested amount of money, return his card, and subtract amount of the withdrawal from the user's account.

**Scenario:** Scenario 1

**Given** preconditions

**When** actions

**Then** results

# Cucumber in practice

- <https://cucumber.io/>
- <https://github.com/cucumber>
- Cucumber is supported in C++ too
  - <https://github.com/cucumber/cucumber-cpp>



감사합니다

[www.testworks.co.kr](http://www.testworks.co.kr)

## I. 부록: 테스트 설계 기법

- 명세기반 설계 기법
- 구조기반 설계 기법

# 명세기반 – 동등분할 (Equivalence Partitioning)

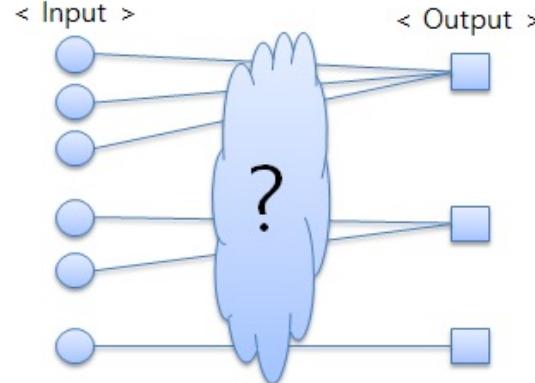
## ■ 개요

- 입력값이 범위가 정해져 있을 경우, 각 범위의 대표값을 이용하여 테스팅

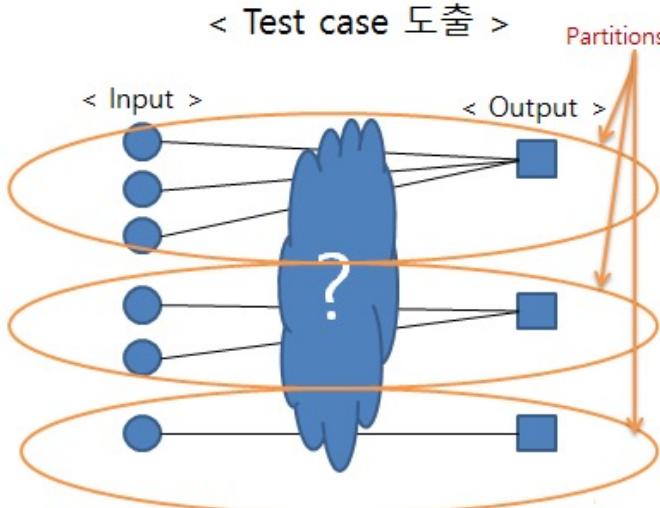
## ■ 적용 절차

- 소프트웨어나 시스템이 특정 범위의 입력값에 의해 결과값이 동일하다면 입력값의 범위를 하나의 그룹으로 구분
- 입력값/출력값 영역을 유한개의 상호 독립적인 집합으로 나누어 수학적인 등가집합으로 만듦
- 각 등가 집합의 원소 중 대표값을 선택해 테스트 케이스를 도출함

<Functional/Behavior Specification >



< Test case 도출 >



## 명세기반 – 동등분할 (Equivalence Partitioning) 실습1

- 100점이 만점이고 0 ~ 100점을 받을 수 있는 시험이 있다. 시험 점수를 입력하면, 점수에 따라 다음과 같이 A부터 F까지의 성적을 출력하라.

	성적
90점 이상 ~ 100점 이하	A
80점 이상 ~ 90점 미만	B
70점 이상 ~ 80점 미만	C
0점 이상 ~ 70점 미만	F

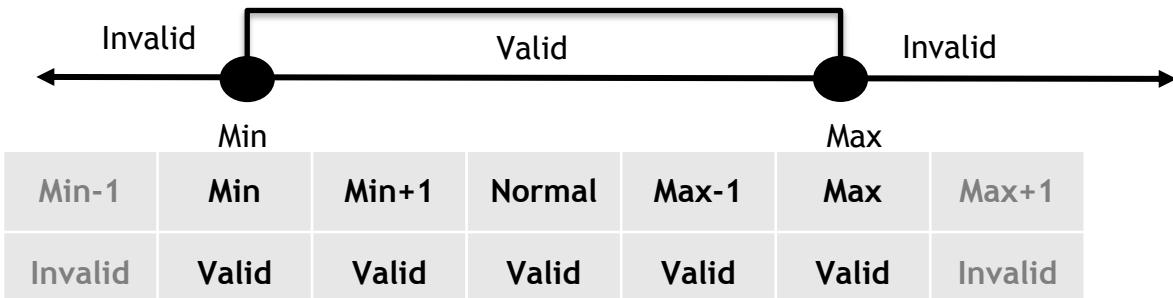
# 명세기반 – 동등분할 (Equivalence Partitioning) 실습1 풀이

	0 ≤ 점수 < 70	70 ≤ 점수 < 80	80 ≤ 점수 < 90	90 ≤ 점수 ≤ 100
테스트케이스	1	2	3	4
분할영역	0점 이상 ~ 70점 이하	70점 이상 ~ 80점 미만	80점 이상 ~ 90점 미만	90점 이상 ~ 100점 이하
입력 값(점수)	50점	75점	85점	95점
예상 결과값	F	C	B	A

분할영역	입력 값	예상 결과 값
실수로 입력	35.5	
문자열 입력	@점	
입력하지 않는 경우	Null	유효하지 않음
범위 밖의 값을 입력	103	

## 명세기반 – 경계 값 분석 (Boundary Value Analysis)

- 입력 값의 주요 오류 대상인 경계 값을 입력 값으로 테스트 케이스를 작성하여 테스팅
- Boolean값은 적합하지 않음
- 경계 값 분석은 동등 분할의 경계 부분에 해당되는 입력 값에서 결함이 발견될 확률이 경험적으로 높기 때문에 경계 값까지 포함하여 테스트 하는 기법임
  - 유효 경계 값 : 유효한 분할영역의 경계 값
  - 비유효 경계 값 : 비유효한 분할영역의 경계 값
- 경계 값 분석은 모든 테스트 레벨 및 모든 테스트 유형에서 적용이 가능함



## 명세기반 – 경계 값 분석 (Boundary Value Analysis) 실습

- ▶ 프린터 카트리지를 판매하는 도매업자가 있다. 최소 주문 수량은 5개이고, 100개 이상 살 경우 20%의 할인을 받을 수 있다. 경계 값 분석 활용하여 테스트 케이스를 도출 하시오.

## 명세기반 – 경계 값 분석 (Boundary Value Analysis) 실습 풀이

#TC	분할영역	입력	예상 결과
1	5보다 적은 수	4	주문 불가
2	5~100	5	주문 가능
3	5~100	99	주문 가능
4	100보다 큰 수	100	주문 가능, 20% 할인

## 명세기반 – 결정테이블

- 결정테이블은 동작을 유발시키는 조건 또는 상황(Triggering conditions) 주로 모든 입력 조건에 대한 참과 거짓의 조합으로 나타남), 각 해당 조합에 대한 예상 결과까지 포함.
- 입/출력값이 True, False로 결정될 수 있는 경우 모든 경우의 수를 확인해볼 수 있는 방법
- 테이블의 각 칼럼은 비즈니스 규칙과 대응관계를 갖는다.
- 해당 비즈니스 규칙(테이블의 각 칼럼)은 유일한 조건의 조합(Combination of conditions)을 정의하고, 조건의 조합은 해당 비즈니스 규칙과 연관된 동작을 수행

## 명세기반 – 결정테이블 실습

### 명세

Columbus의 16 - 65세 사이의 국민들은 소득세를 내야 한다. 소득이 20,000 달러 미만인 사람은 소득의 20%의 소득세를 내야 하고 그 이상인 사람은 50%의 소득세가 부가된다. 단 아이가 있는 경우는 10%를 감면 받을 수 있다.

Decision Table(결정 테이블) 테스트 설계 기법을 사용하여 Test Case를 개발 하세요

## 명세기반 – 결정테이블 실습 풀이

### 테스트 조건

속성	1	2	3	4	5	6	7	8
나이 16~65	T	T	T	T	F	F	F	F
소득 < 20000	T	T	F	F	T	T	F	F
자녀 = Y	T	F	T	F	T	F	T	F

### 예상결과

세금 20%	●	●						
세금 50%			●	●				
감세 10%	●		●					

## 명세기반 – 결정테이블 실습 풀이

A g l l y

û Ü	-	-	..	-	)	,	.	..
n r	-	-	-	-	"	"	"	"
ú C	-	-	"	"	-	-	"	"
† Å	-	"	-	"	-	"	-	"

ž } W-

á -	-	Dz	Dz					
á -	-			Dz	Dz			
b á	-	Dz		Dz				

A g l W r g

û Ü	-	-	..	)	)	,	.	..
n r	-	,	-	-	-	,	-	-
ú C	-	-	-	-	-	-	-	-
† Å	..	-	..	-	..	-	..	-
H W-	ú C	ú C	ú C	ú C	ú C	ú C	ú C	ú C

## I. 부록: 테스트 설계 기법

- 명세기반 설계 기법
- 구조기반 설계 기법

# 구조 기반 테스트 설계 기법 개요

- 구문 커버리지 (Statement Coverage)
  - 테스트에 의해 실행된 구문 (Statement)가 몇 %인지 측정
- 결정 커버리지 (Decision Coverage, Branch Coverage)
  - 테스트에 의해 실행된 결정 포인트 내의 전체 조건식이 최소한 참 한 번,거짓 한 번의 값을 가지는지 측정
  - 개별 조건식의 개수와 관계 없이 테스트 케이스의 최소 개수는 2개로 도출
- 조건 커버리지 (Condition Coverage)
  - 전체 조건식의 결과와 관계없이 각 개별 조건식이 참 한 번,거짓 한 번을 모두 가지도록 조건식을 조합

Decision Point	A	B
0	1	0
1	1	1

&lt;A AND B에 대한 결정 커버리지 결정 테이블&gt;

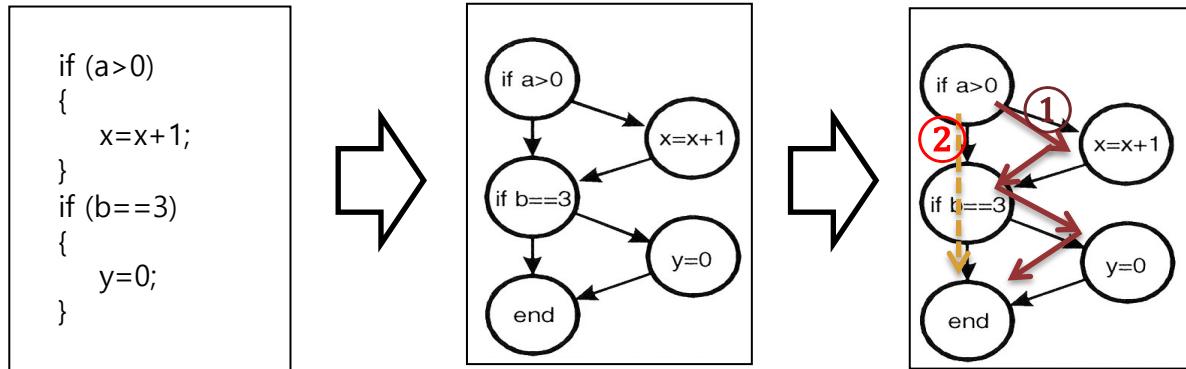
Decision Point	A	B
0	1	0
0	0	1

&lt;A AND B에 대한 조건 커버리지 결정 테이블&gt;

## 구조 기반 테스트 설계 기법 - 구문 커버리지 & 결정 커버리지

### ■ 실습 예제 – 구문 커버리지 & 결정 커버리지

- 아래 프로그램 코드에서 구문 커버리지와 결정 커버리지 100%를 만족하는 테스트 케이스를 도출하시오.



- 구문 커버리지 100%인 테스트케이스 ?

Test Case 1 : a = 6, b = 3

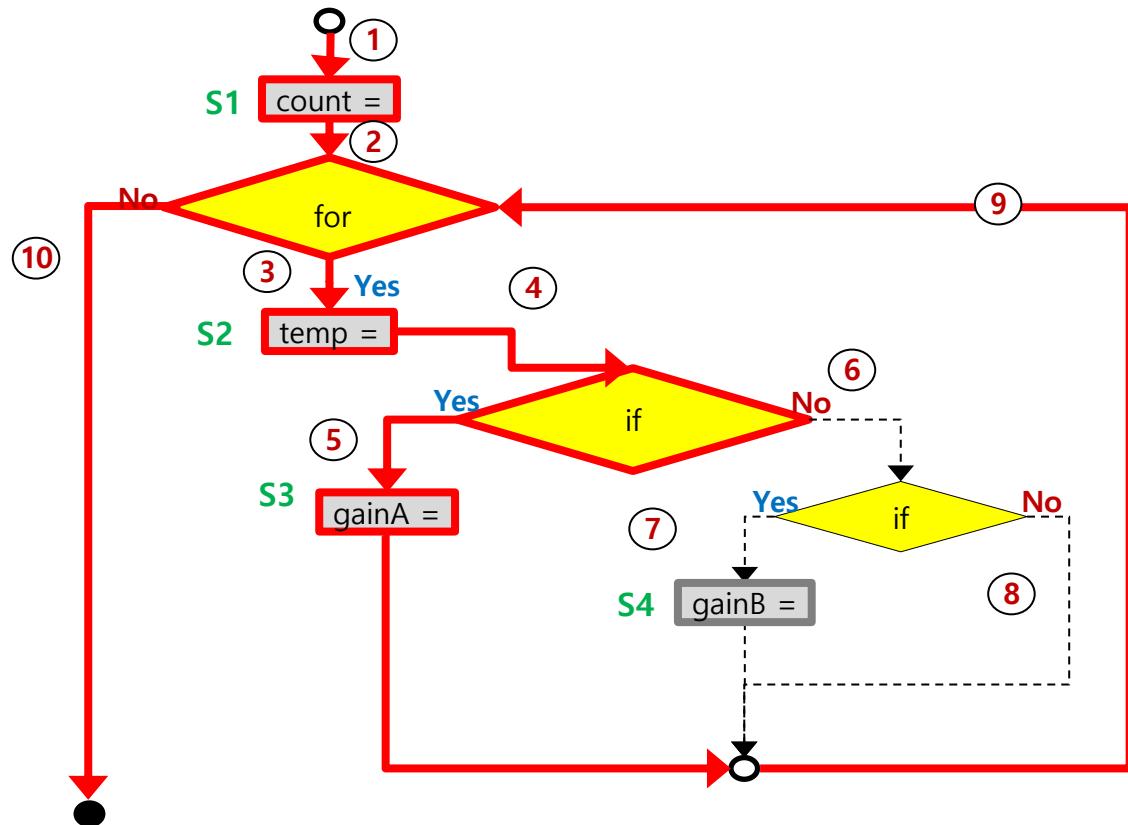
- 결정 커버리지 100%인 테스트케이스 ?

Test Case 1 : a = 6, b = 3

Test Case 2 : a = -1, b = 1

# 구조 기반 테스트 설계 기법 - 구문 커버리지 & 결정 커버리지 실습

## ▪ 구문 커버리지 vs. 결정 커버리지

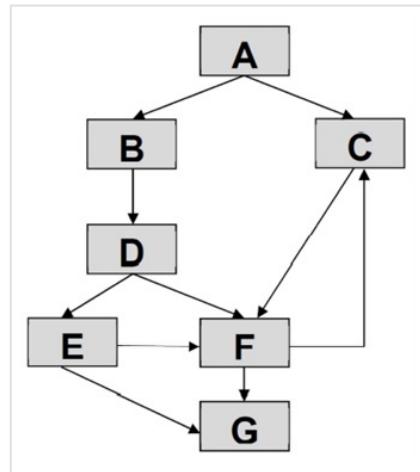


- TC : 1-2-3-4-5-9-10
  - 구문(명령문) 커버리지  
(Statement Coverage)?
  - 결정(분기) 커버리지  
(Decision/Branch Coverage)?
  - 조건 커버리지  
(Condition Coverage)?

# 구조 기반 테스트 설계 기법 – 실습풀이

## ■ 실습 예제 – 결정 커버리지

- 프로젝트의 테스트 목표 중 하나는 100% 결정 커버리지 달성이다. 아래의 제어흐름 그래프에 대해 다음의 3개의 테스트 케이스가 실행됐다.
  - 테스트 A가 커버하는 경로: A, B, D, E, G
  - 테스트 B가 커버하는 경로: A, B, D, E, F, G
  - 테스트 C가 커버하는 경로: A, C, F, C, F, C, F, G



다음 중 결정 커버리지(Decision coverage) 달성을 관해 올바르게 설명한 것은?

- a) 결정(Decision) D는 완전히 테스트 되지 않았다
- b) 100% 결정 커버리지를 달성했다
- c) 결정 E는 완전히 테스트 되지 않았다
- d) 결정 F는 완전히 테스트 되지 않았다

출처: Sample Exam. Certified Tester Foundation Level. 2011 Syllabus. Version 2.3