UnitTest (ver 3.4.1)

목차

1. 테스트 개요

1.1 테스트 정의 (1/2)

✓ 테스트는 아래와 같이 정의합니다.

- IEEE
- 미국 전기 전자 학회
- Institute of Electrical and Electronics Engineers

소프트웨어 테스팅은 기대되는 결과와 실제 결과의 차이(즉, 버그)를 식별하고 기능을 평가하는 분석과정이다.

- *"Software testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item."* [1]

- Glenford J. Myers
- The Art of Software Testing 저자
- 스펙트럼 시그널 사의 CEO
- 레디시스 사의 창업자이자 대표
- IBM 시스템연구소의 선임연구원

테스팅은 에러를 찾기 위해 프로그램을 실행하는 프로세스이다.

- *Testing is the process of executing a program with the intent of finding errors."* [2]
- 테스팅와 디버깅을 처음으로 구분함

- Cem Kaner
- Testing Computer Software 저자
- Lessons Learned in Software Testing 저자
- Professor of Software Engineering at Florida Institute of Technology
- Director of Florida Tech's Center for Software Testing Education & Research

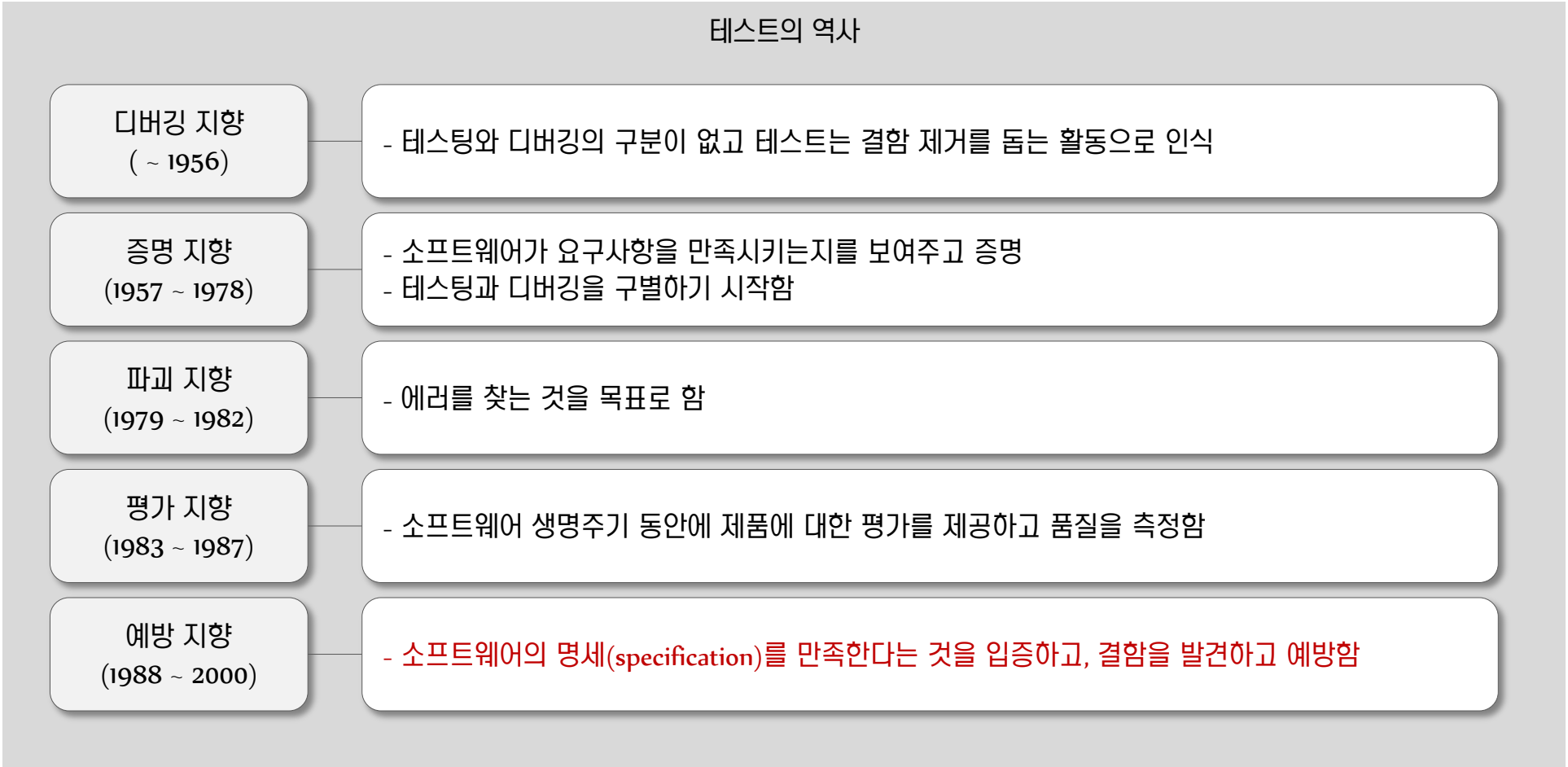
소프트웨어 테스팅은 이해관계자들에게 테스트 대상 제품 또는 서비스의 품질에 관한 정보를 제공하는 조사과정이다.

- *"Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test."* [3]

1) 1059-1993 - IEEE Guide for Software Verification and Validation Plan
2) Glenford. J. Myers. The Art of Software Testing. 1979
3) Cem Kaner. Quality Assurance Institute Worldwide Annual Software Testing Conference. 2006

1.1 테스트 정의 (2/2)

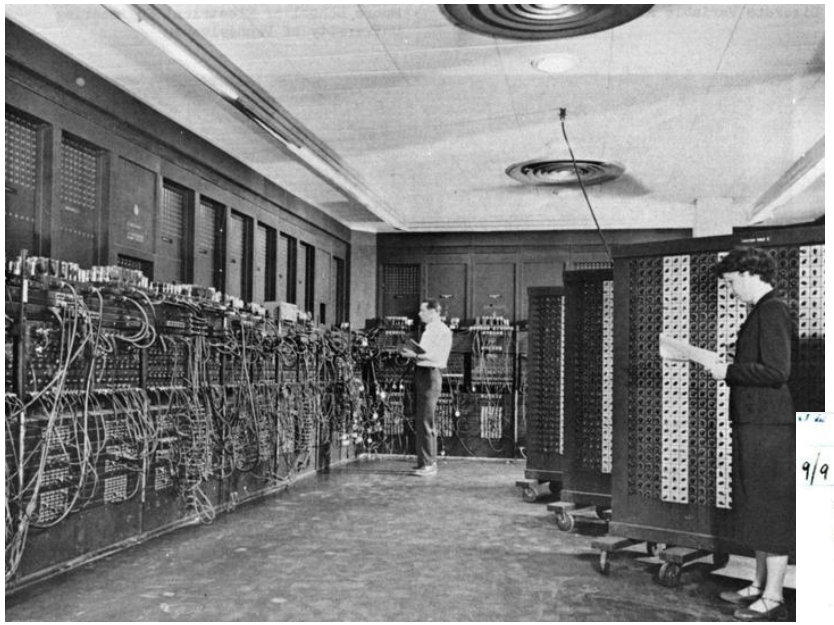
✓ Dave Gelperin과 William C. Hetzel은 1988년에 소프트웨어 테스팅이 다음과 같은 단계 및 목표를 따른다고 정의하였습니다.[1)]



1) Dave Gelperin; William C. Hetzel. The Growth of Software Testing. 1988

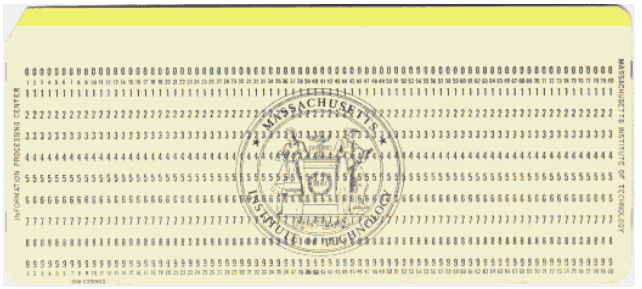
1.2 테스트 역사

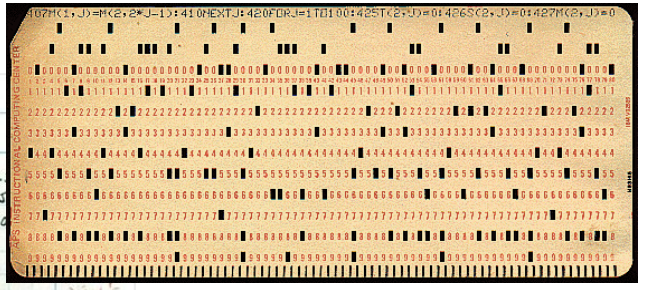
✓ 최초로 발생한 컴퓨터의 오류는 벌레 때문이었습니다.



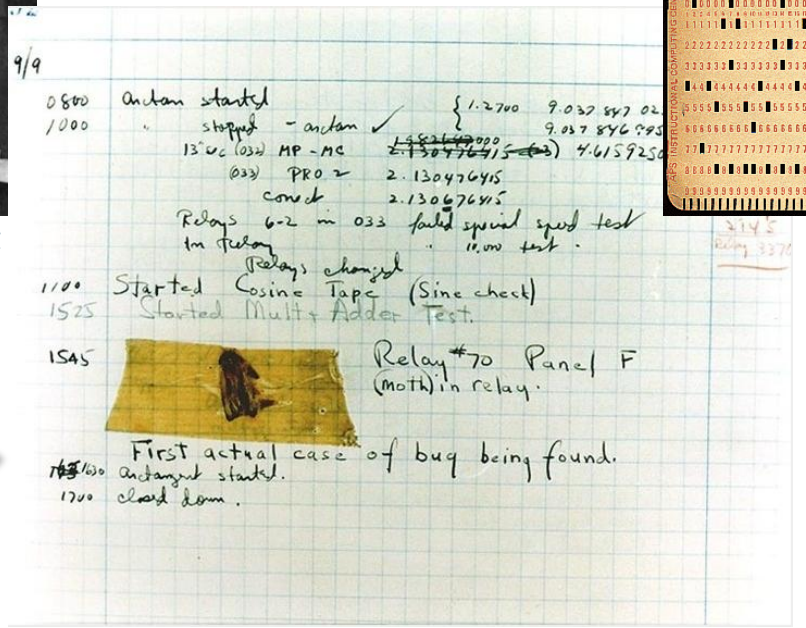
초창기의 컴퓨터

출처 : https://en.wikipedia.org/wiki/ENIAC





편치 카드



최초의 버그 리포트

출처 : http://collections.si.edu/search/results.htm?q=computer%20bug

1.3 테스트 필요성 (1/2)

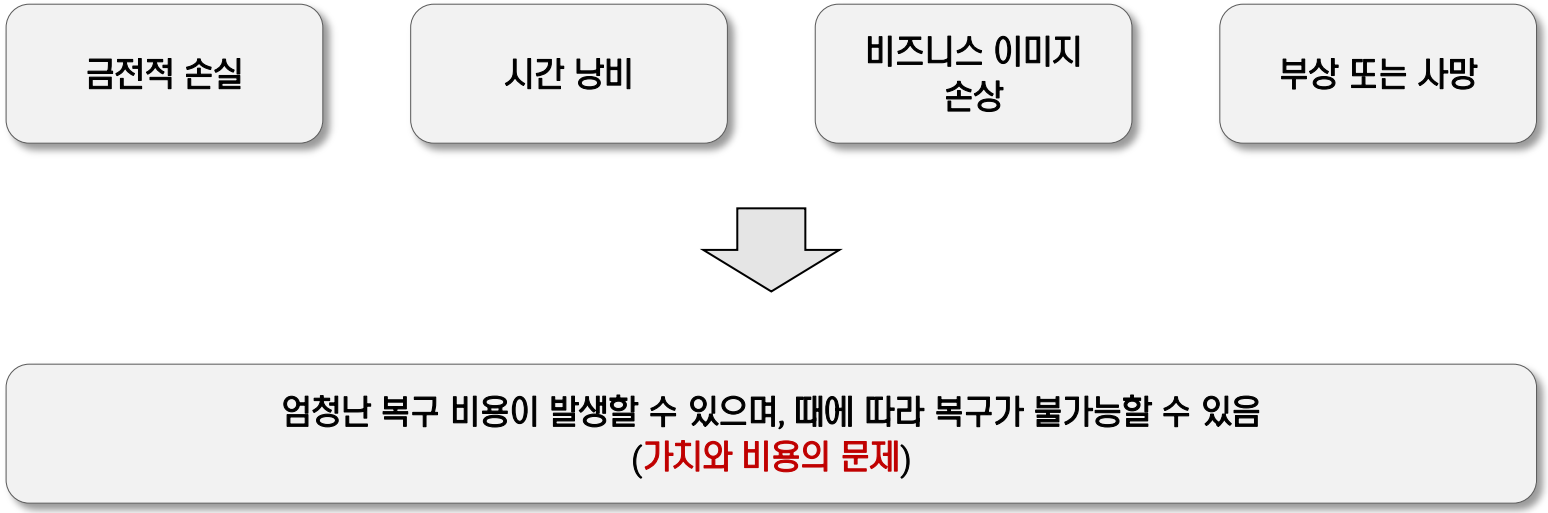
- ✓ 많은 질병들은 건강 검진을 통한 조기 발견 및 예방이 중요합니다.
- ✓ 건강 검진 비용이 아까워서 건너 뛰기도 하지만, 병에 걸린 후 치료에 필요한 비용보다는 훨씬 저렴합니다.
- ✓ 소프트웨어도 '별 문제 없겠지' 라는 생각으로 테스트하지 않은 소스 코드들이 있습니다.
- ✓ 이 코드들은 나중에 많은 비용을 삼키는 골치 덩어리 오류가 될 수도 있습니다.

건강검진(테스트)과 수술(디버그) 어느 쪽을 택할 것인가?

출처
1) https://pixabay.com
2) http://www.imbc.com/broad/tv/drama/whitepower/news/1547088_19902.html

1.3 테스트 필요성 (2/2)

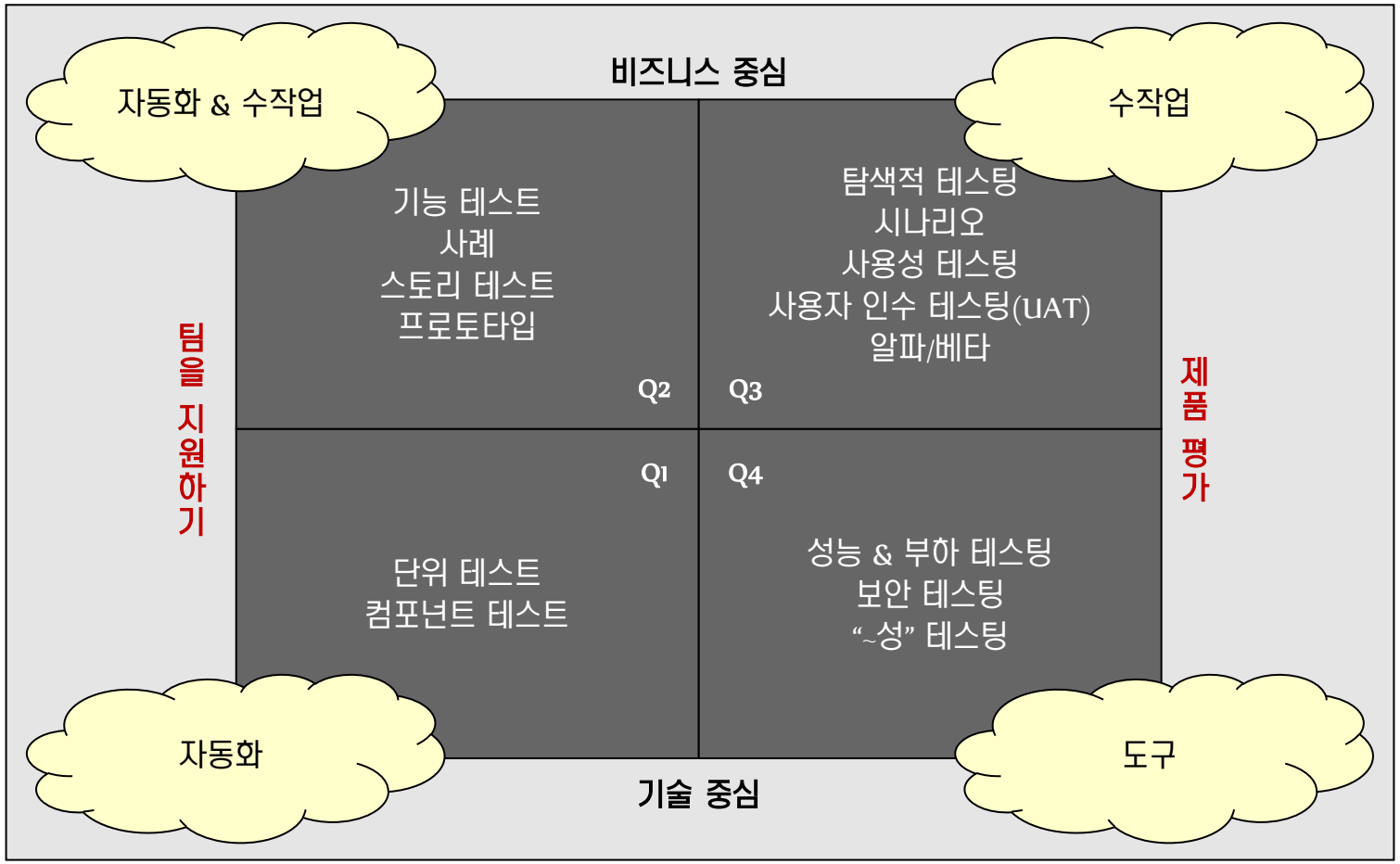
- ✓ 소프트웨어가 올바르게 동작하지 않는 경우 다양한 문제가 발생할 수 있습니다. (금전, 시간, 비즈니스 이미지, 부상, 사망)
- ✓ 문제가 발생했을 때 이를 해결하기 위해서 들어가는 비용은 테스트를 통한 사전 예방활동에 대한 비용보다 더 크며, 때에 따라서 해결하지 못하거나 복구가 불가능한 경우도 있을 수 있습니다.
- ✓ 따라서 테스트를 통해서 문제 발생 확률을 낮추어 문제해결 비용을 절감하고, 품질과 신뢰성을 높여야 합니다.



1.4 테스트 목적 (1/6)

✓ 테스트는 크게 **팀을 지원**하려는 목적과 **제품을 평가**하기 위한 목적을 가지고 있습니다. [1)]

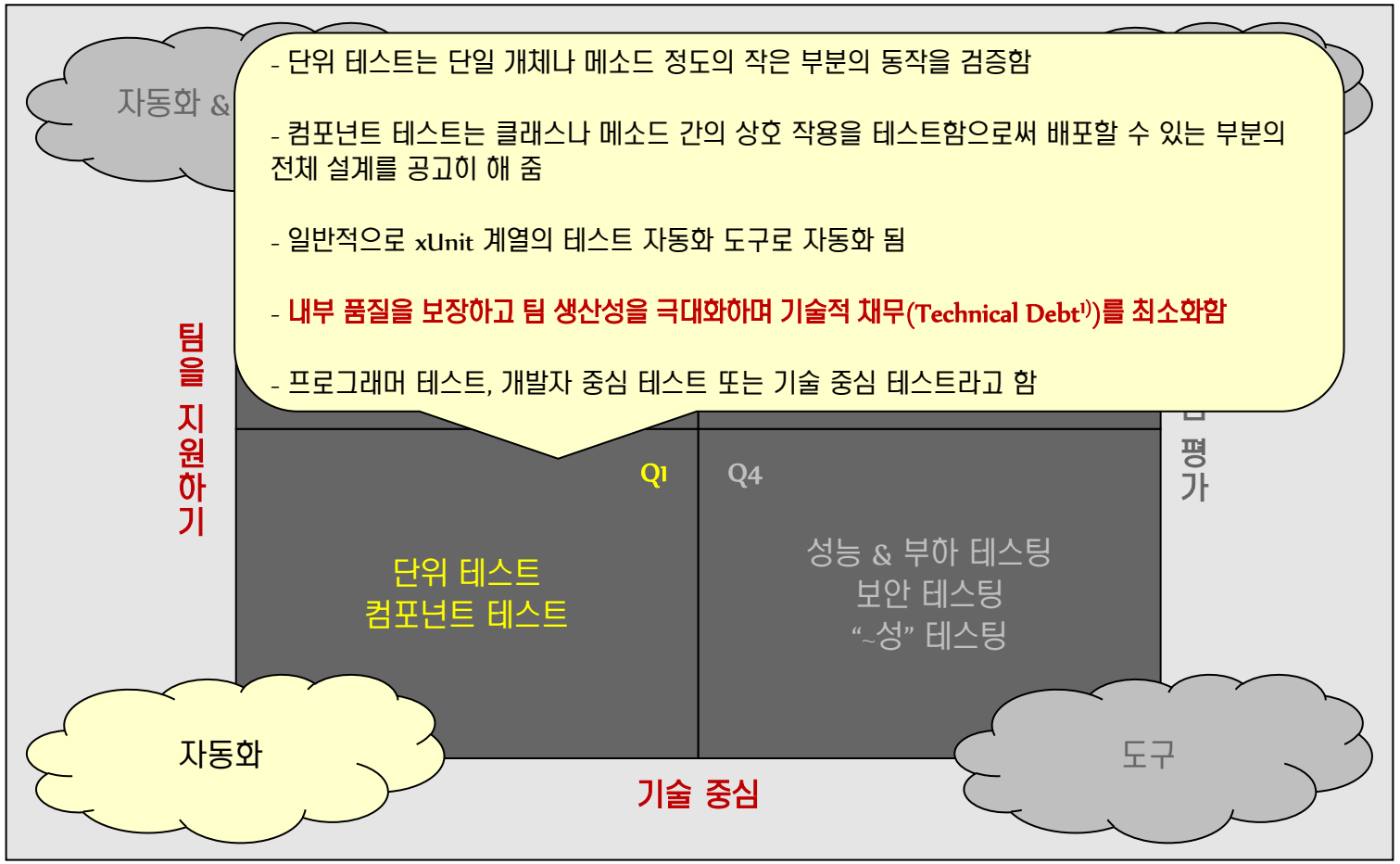
애자일 테스팅 사분면



1) Lisa Crispin; Janet Gregory. Agile Testing: A Practical Guide For Testers and Agile Teams. 2009

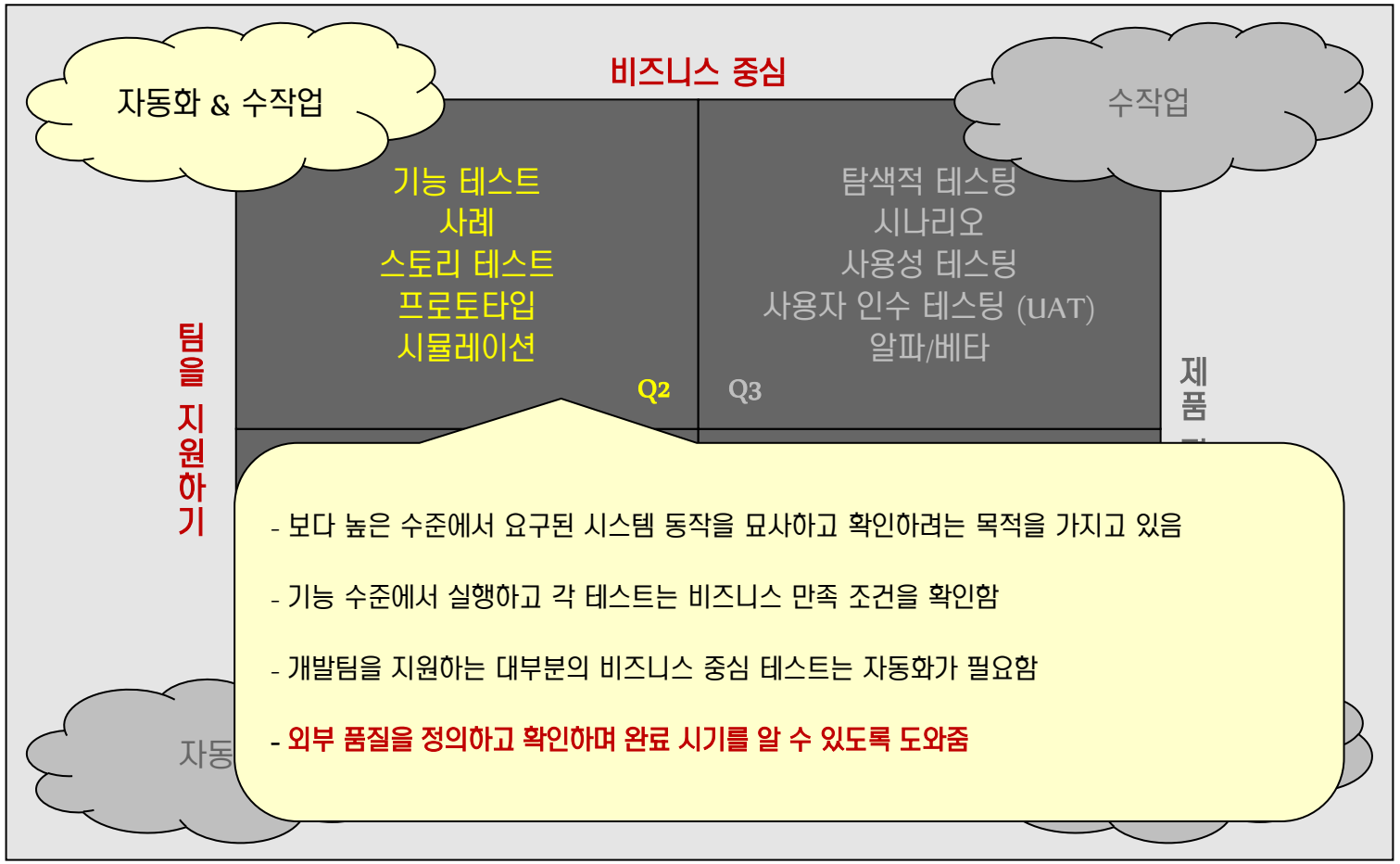
1.4 테스트 목적 (2/6)

✓ Q1의 주 목적은 테스트 주도 개발(TDD)를 통해서 프로그래머가 자신의 코드를 잘 설계하도록 돕는 것입니다.



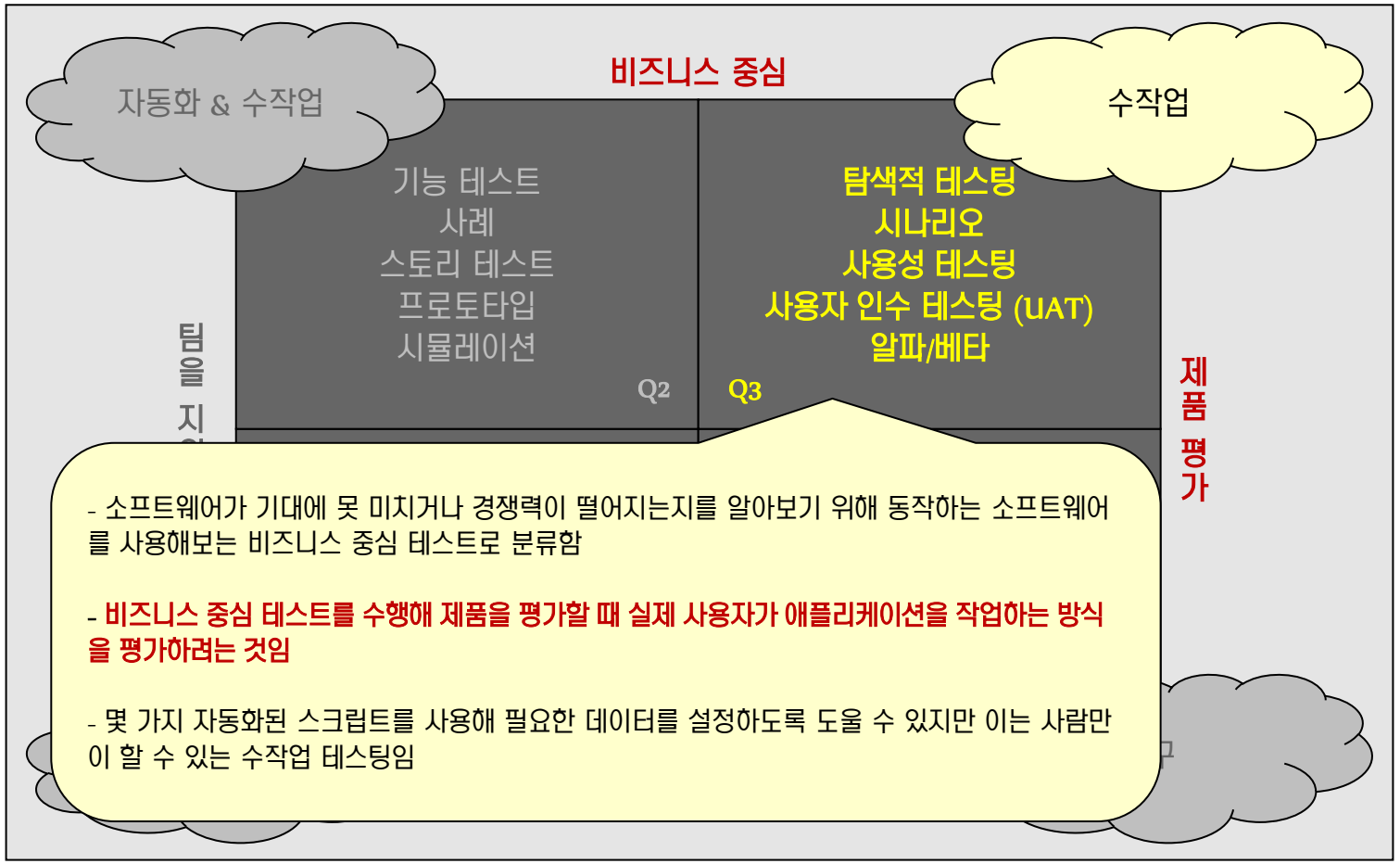
1.4 테스트 목적 (3/6)

✓ Q2의 주 목적은 비즈니스 요구사항대로 개발을 하고 있는지 개발팀에 피드백을 주는 것입니다.



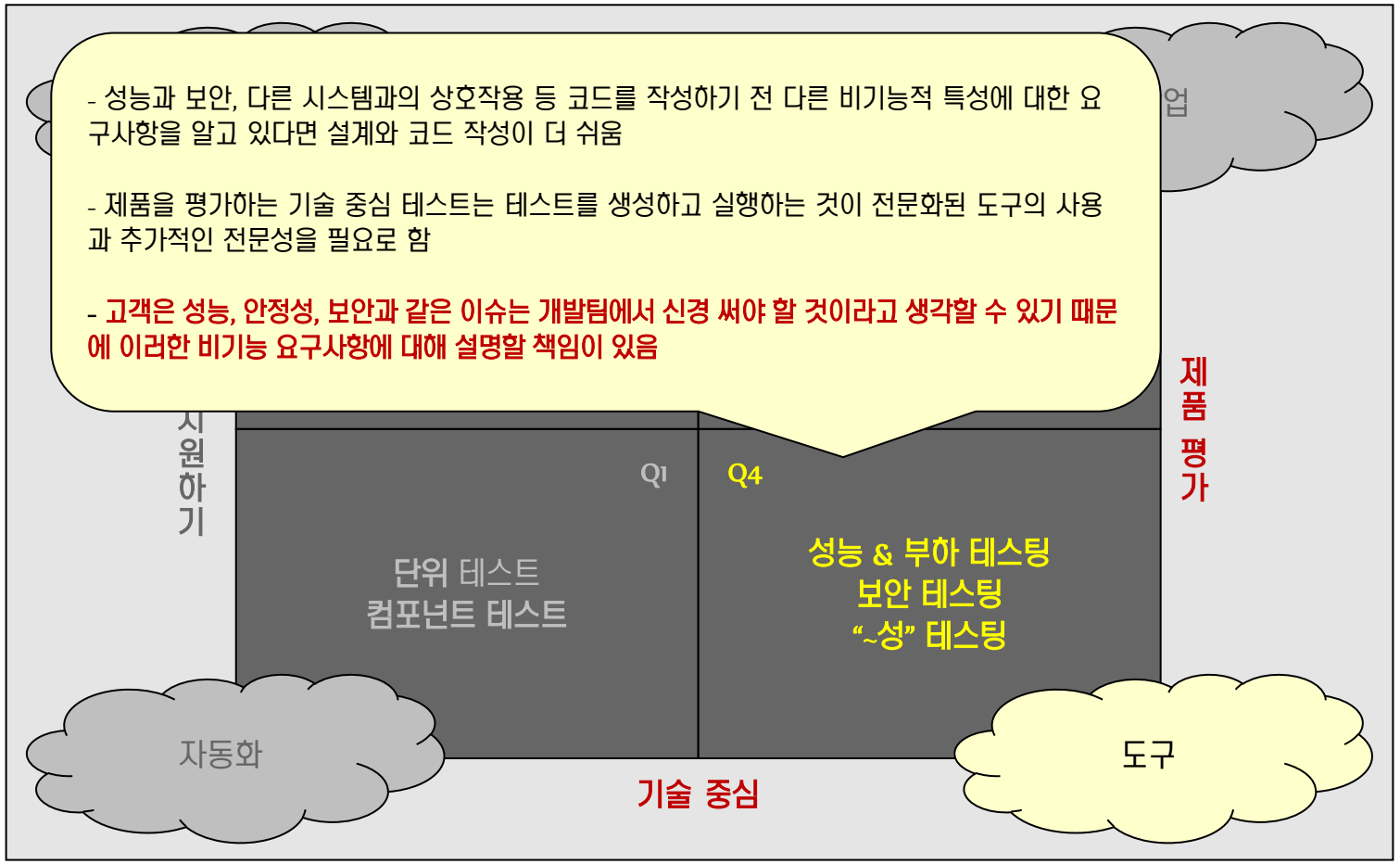
1.4 테스트 목적 (4/6)

✓ Q3의 주 목적은 고객이 원하는 비즈니스 요구사항이 반영되었는지 평가하는 것입니다.



1.4 테스트 목적 (5/6)

✓ Q4의 주 목적은 고객이 원하는 비기능 요구사항(품질)이 반영되었는지 평가하는 것입니다.



1.4 테스트 목적 (6/6)

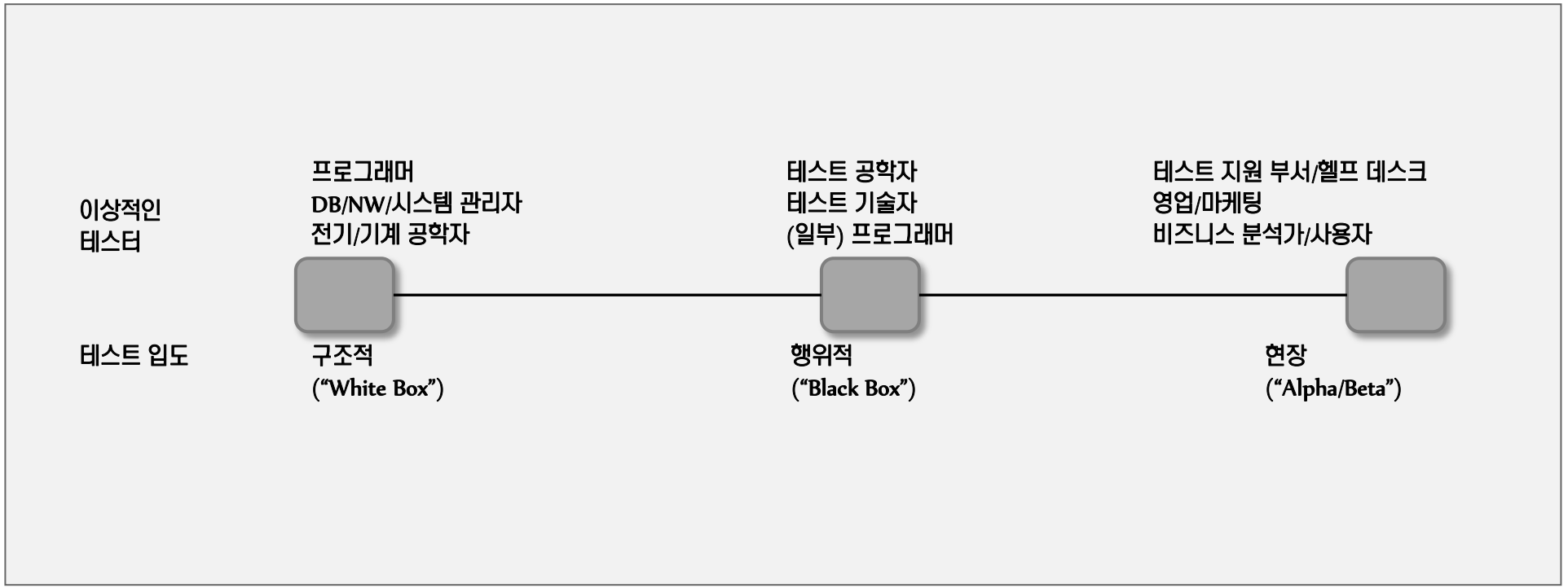
✓ 애자일 테스팅 사분면을 체크 리스트로 활용 할 수 있습니다.

항목	확인
애플리케이션에 적합한 설계를 찾는 것을 돕기 위해 단위 테스트와 컴포넌트 테스트를 사용하고 있는가?	
신속한 피드백을 위해 자동화된 단위 테스트를 실행하는, 자동화된 빌드 프로세스를 가지고 있는가?	
비즈니스 중심 테스트를 수행해 고객의 기대를 **만족하는 제품을 제공하도록 돕고 있는가?**	
원하는 시스템 동작의 바른 예를 잡아내고 있는가? 더 필요한 것은 없는가? 테스트가 이러한 사례에 기반을 두고 있는가?	
코드를 작성하기 전에 사용자에게 ui 프로토타입을 보여주고 보고하는가? 최종 소프트웨어가 어떻게 동작할 것인지를 사용자에게 들려줄 수 있는가?	
탐색적 테스트를 위한 충분한 시간을 확보하고 있는가? 사용성 테스트를 어떻게 해결하고 있는가? **충분한 고객이 참여하고 있는가?**	
성능과 보안 같은 기술적 요구사항을 개발 주기 맨 처음부터 고려하고 있는가? 테스트의 다양한 측면을 수행하는 적절한 도구를 가지고 있는가?	

1.5 테스트 종류 (1/5)

✓ 테스트 입도

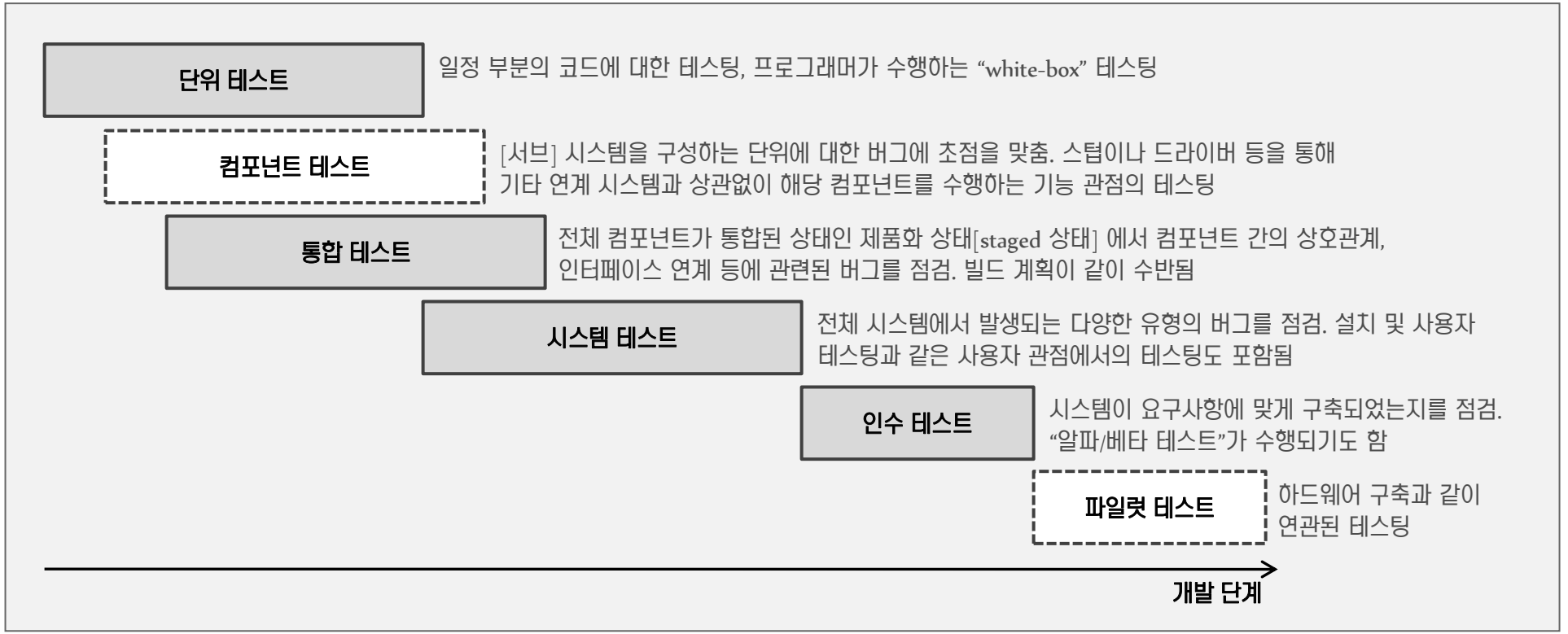
- 테스트의 관점이 얼마나 세밀하고 거친지를 가리킵니다.
- 정밀한 테스트 케이스는 낮은 수준의 세부사항을 점검하거나 시스템의 내부를 점검합니다.
- 거친 입도의 테스트 케이스는 일반적인 시스템 행위에 대한 정보를 제공합니다.



1.5 테스트 종류 (2/5)

✓ 단계별 테스트 접근방법

- 단계별로 필요한 테스트를 수행하고, 이에 맞는 대응 전략을 마련할 수 있습니다.
- 테스트 목표, 테스트 베이시스[1], 테스트 대상, 주로 발견되는 장애와 결함의 종류, 테스트 수행 주체(조직) 등과 같은 항목은 각 테스트 레벨[2]의 특징에 맞게 다르게 정의되고 식별되어야 합니다.



1) test basis. 요구사항을 내포하고 있는 모든 문서. 테스트 케이스는 테스트 베이시스를 토대로 만들어 진다.
문서가 오직 공식적 수정절차의 방법에 의해 수정될 수 있다면, 해당 테스트 베이시스를 동결(frozen) 테스트 베이시스라 부름.
2) 개발 단계와 대응하는 테스트 단계를 의미한다.

1.5 테스트 종류 (3/5)

✓ 테스트 유형

- 테스트하는 목적 및 품질 특성을 염두에 두고, 소프트웨어 시스템을 검증하는 일련의 테스트 활동 입니다.
- 크게 기능 테스트(Functional Test), 비기능 테스트(Non-functional Test)으로 나눌 수 있습니다.

1.5 테스트 종류 (4/5)

✓ 성능/부하 테스팅

- 성능 테스팅(performance testing)은 특정 부하 상태에서 시스템이나 서브 시스템이 얼마나 빨리 반응하는지 점검합니다.
  - 확장성(scalability), 신뢰성(reliability), 자원 사용에 대한 품질도 측정 가능합니다.

- 부하 테스팅(load testing)은 대량의 데이터나 다수의 사용자와 같은 특정 부하에서 기능을 제대로 수행하는지 점검합니다.
  - 확장성(scalability)과 관련된 테스팅입니다.
  - 내구성 테스팅(endurance testing) 이라고도 함

✓ 안정성(stability) 테스팅

- 인정할 수 있는 기간 동안 혹은 넘어서 지속적으로 기능을 수행하는지를 점검합니다.
- 부하(load) 테스팅이라고도 합니다.

✓ 사용성(usability) 테스팅

- 사용자 인터페이스가 사용하기 쉽고 이해하기 쉬운지를 점검합니다.

1.5 테스트 종류 (5/5)

✓ 보안성 테스팅

- 외부 침입자에 의한 시스템 침투를 방어하는 데이터 암호화에 대한 절차 점검합니다.

✓ 국제화 및 지역화

- 임의의 지역에 대한 SW 적응성을 테스팅 합니다.
- 새로운 언어나 문화 환경에서도 시스템이 기능을 제대로 수행하는지를 점검합니다.

✓ 기타 테스팅

- 파괴(destructive) 테스팅
  - 시스템의 견고성을 위해 시스템의 실패를 유발시키는 테스팅을 의미합니다.

- 탐험적(exploratory) 테스팅
  - 테스터의 개인적인 자유와 책임을 강조한 테스팅의 유형으로 프로젝트를 통해 동시에 수행되는 상호 지원 활동입니다.
    테스트 관련 지식, 테스트 설계, 테스트 실행, 테스트 결과 해석을 다룸으로써 테스트 작업의 품질을 지속적으로 최적화시킵니다.
  - 테스터는 테스트를 수행하는 동안에 경험과 함께 새롭고 더 좋은 테스트를 생성시키는 창의성을 습득합니다.
  - 블랙 박스 테스팅 기법으로도 불립니다.
  - 개발 프로세스의 특정 단계에서 특정 테스트 기법에 적용될 수 있는 테스트 접근 방식으로 고려됩니다.
  - 핵심은 테스트 기법이나 테스트 항목, 검토가 아니라, 테스터의 인지적인 참여와 자신의 시간을 관리하는 책임성에 있습니다.

2. 테스트와 개발

2.1 테스트와 문화 (1/2)

✓ 조직 문화는 조직의 가치, 규범, 가정에 의해서 정해지며, 사람들이 의사소통하고 관계를 맺고 결정을 내리는 방식을 지배합니다. 이러한 조직 문화는 직원의 행동을 통해서 쉽게 알 수 있습니다.

✓ 아래와 같은 내용에 대해서 조직 문화를 개선해야 테스트를 적용하기가 쉽습니다.

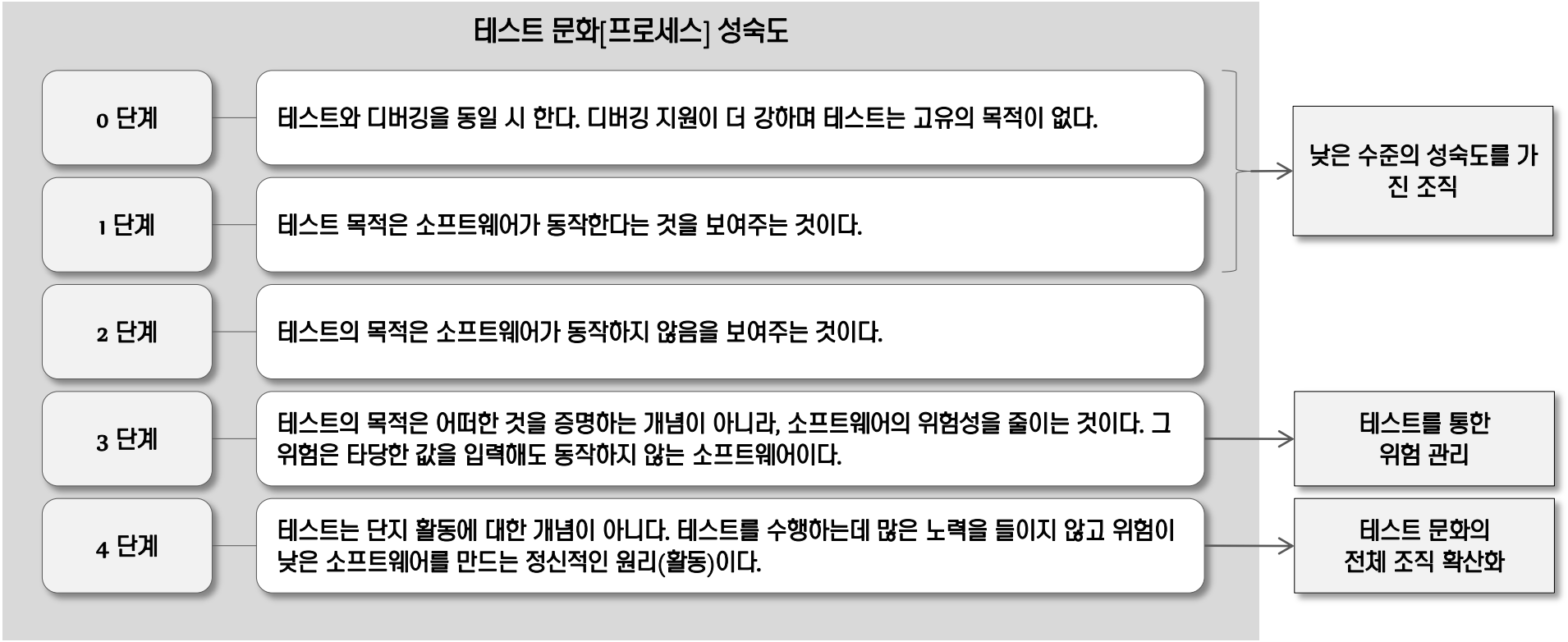
- 품질 철학
- 적당한 진행 속도 유지
- 고객 관계
- 조직 규모
- 팀에 권한 부여

✓ 관리자의 경우 조직 문화와 충돌했을 때 테스터의 성공을 돕기 위해 지원과 훈련을 제공해 주어야 합니다.

✓ 테스터는 진행상황을 추적하고 ROI를 결정하기 위해 관리자가 필요로 하는 정보를 제공함으로써 팀이 관리자의 기대에 부응하도록 도울 수 있어야 합니다.

2.1 테스트와 문화 (2/2)

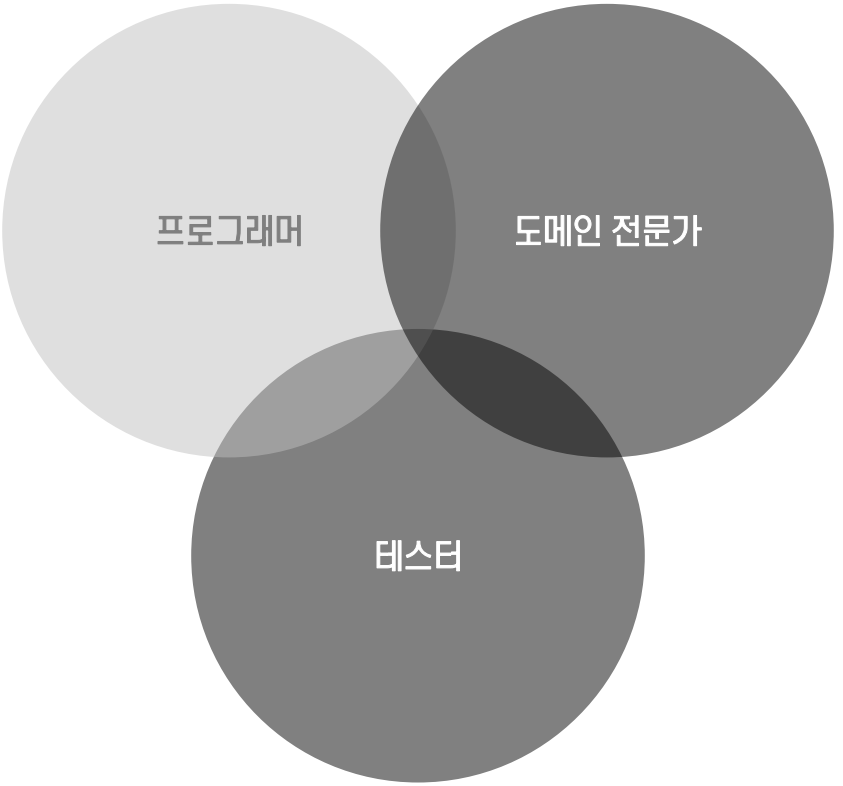
✓ 테스트(개발자) 의 정신세계 5단계



2.2 테스트와 조직 (1/3)

✓ **고객팀**

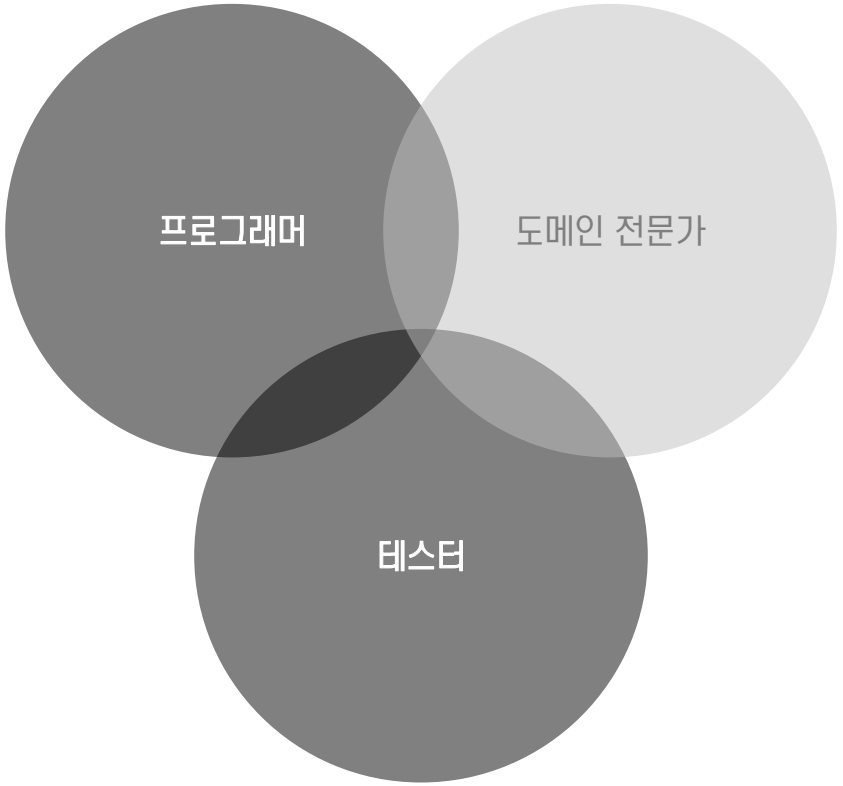
- 비즈니스 전문가, 제품 책임자, 도메인 전문가, 제품 관리자, 비즈니스 분석가, 주제 관련 전문가를 포함한 모두 프로젝트의 “비즈니스” 쪽에 있는 사람들입니다.
- 테스터는 고객이 요구사항과 예제를 이끌어 내고 그들의 요구사항을 테스트를 통해 표현하도록 도와야 합니다.



2.2 테스트와 조직 (2/3)

✓ **개발팀**

- 코드를 만드는 데 관여된 모든 사람을 의미합니다.
- 팀에서의 전문화된 역할을 제한하고 모든 팀원들이 가능한 한 자신들의 기술을 다른 이들에게 전수해주도록 권장합니다.
  그럼에도 불구하고 각 팀은 프로젝트에서의 필요한 전문분야를 결정해야 합니다.
- 테스터는 고객의 입장에서 품질을 대변하고 개발팀이 최상의 비즈니스 가치를 만들어낼 수 있도록 지원해야 합니다.



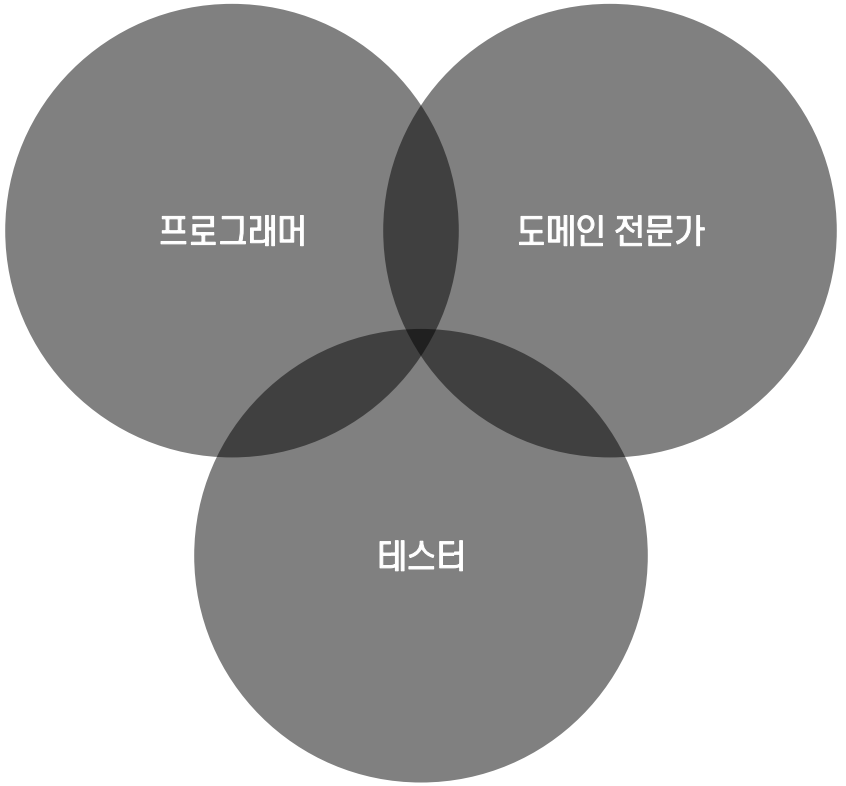
2.2 테스트와 조직 (3/3)

✓ **조직의 가치를 전달한다는 공통의 목표를 가진 한 팀이므로 고객팀과 개발팀 간의 상호작용은 매우 중요합니다.**

- 개발자의 의견을 듣고 고객팀은 개발 우선순위를 결정하고, 개발팀은 얼마만큼의 일을 맡을 수 있을 지 결정해야 합니다.
- 테스트와 사례들로 요구사항을 정의하고 테스트를 통과할 코드를 작성하기 위해서 협력해야 합니다.

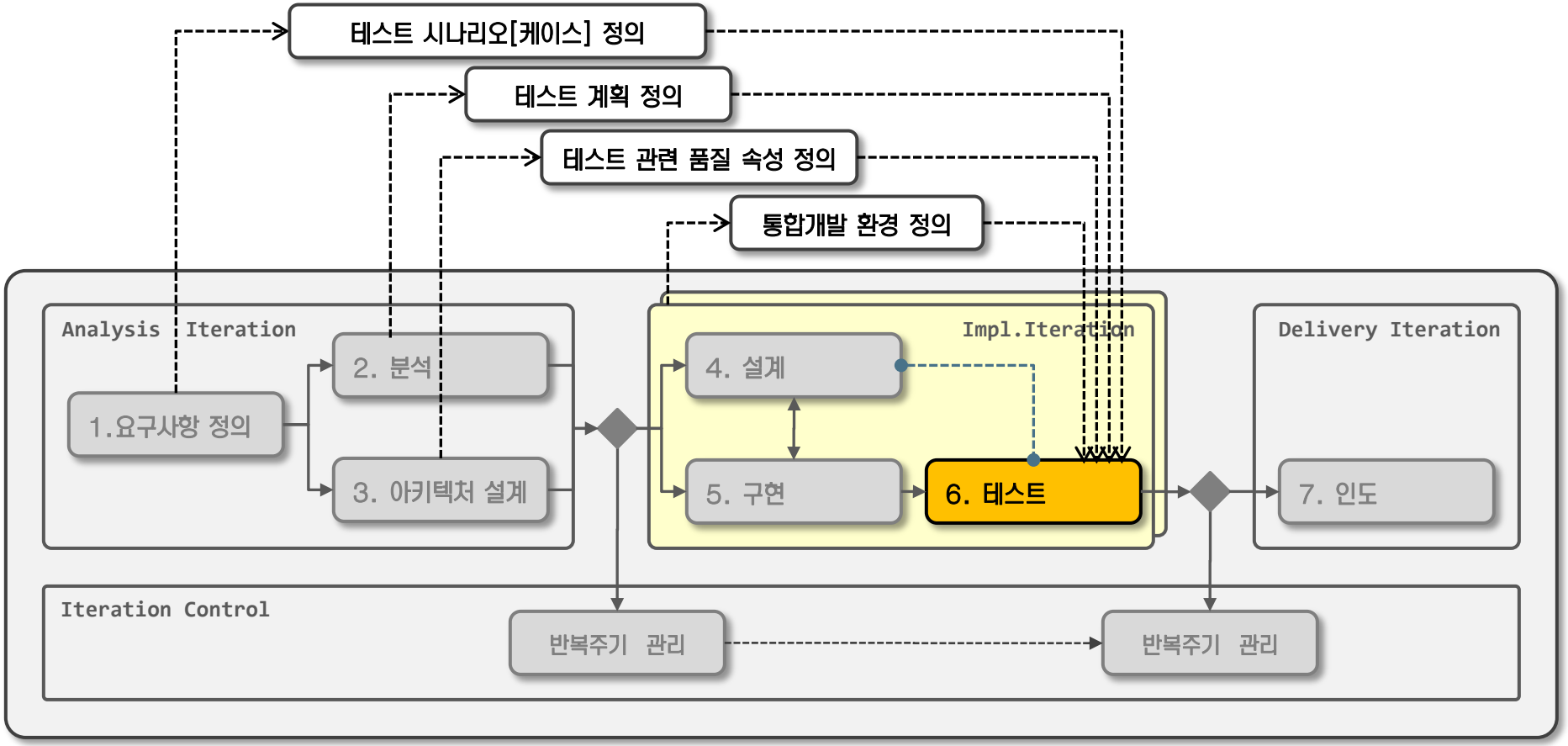
✓ **테스터는 기술적 구현의 복잡성뿐만 아니라 사용자 관점 이해라는 두 영역 모두 발을 담그고 있습니다.**

- 테스터가 없으면 누군가는 테스터의 역할을 수행해야 합니다.



2.3 테스트와 프로세스 (1/3)

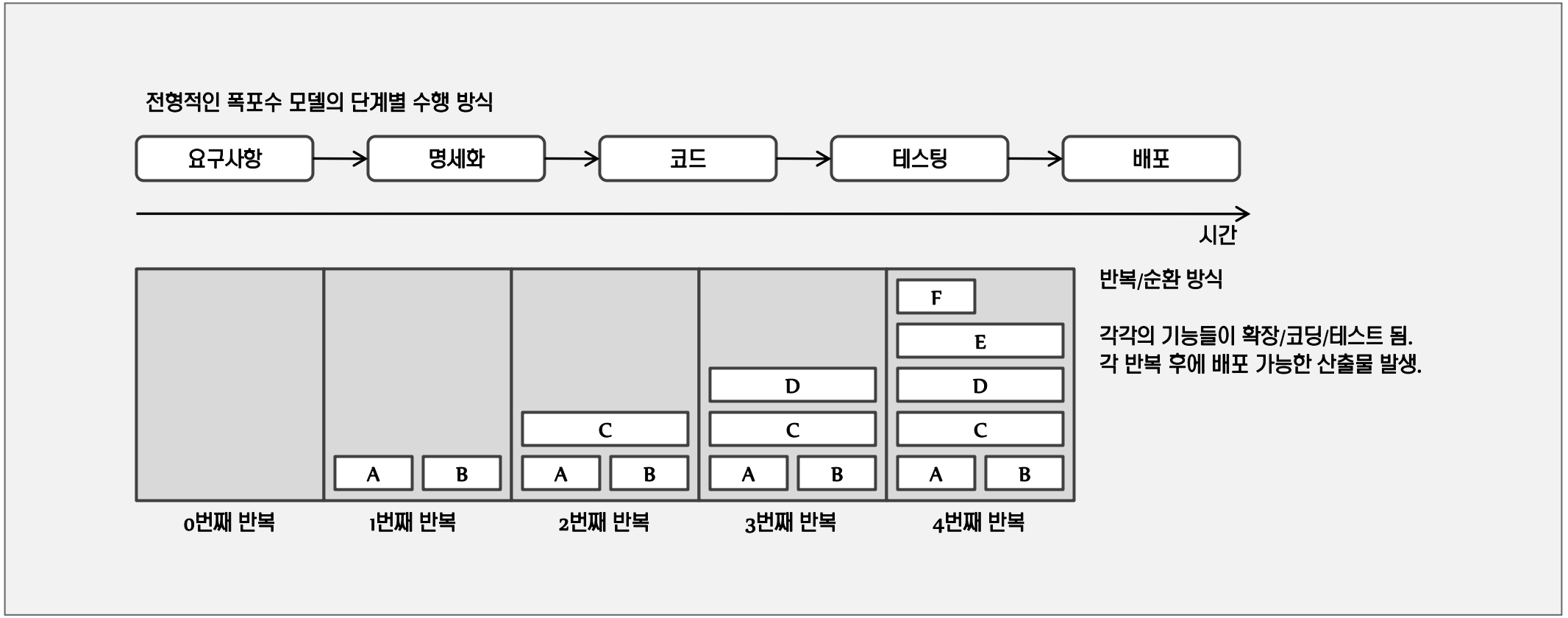
✓ 테스트는 각 개발 단계와 연관이 있으며, 각 활동에서부터 생성된 산출물들을 입력, 참조하게 됩니다.



2.3 테스트와 프로세스 (2/3)

✓ 전형적인 CMMi/폭포수 개발 모델과 Agile/XP 개발 모델

- 각 단계별로 기능 개발 후 독립적인 그룹의 테스터들에 의해 수행되는 보편적 테스팅을 진행합니다.
- Agile/XP 개발에서는 Test-Driven 방식의 개발 모델을 사용합니다.
  - 단위 테스트 작성 -> 테스트에 성공하는 코드 작성 -> 설계 재조정 (리팩토링)



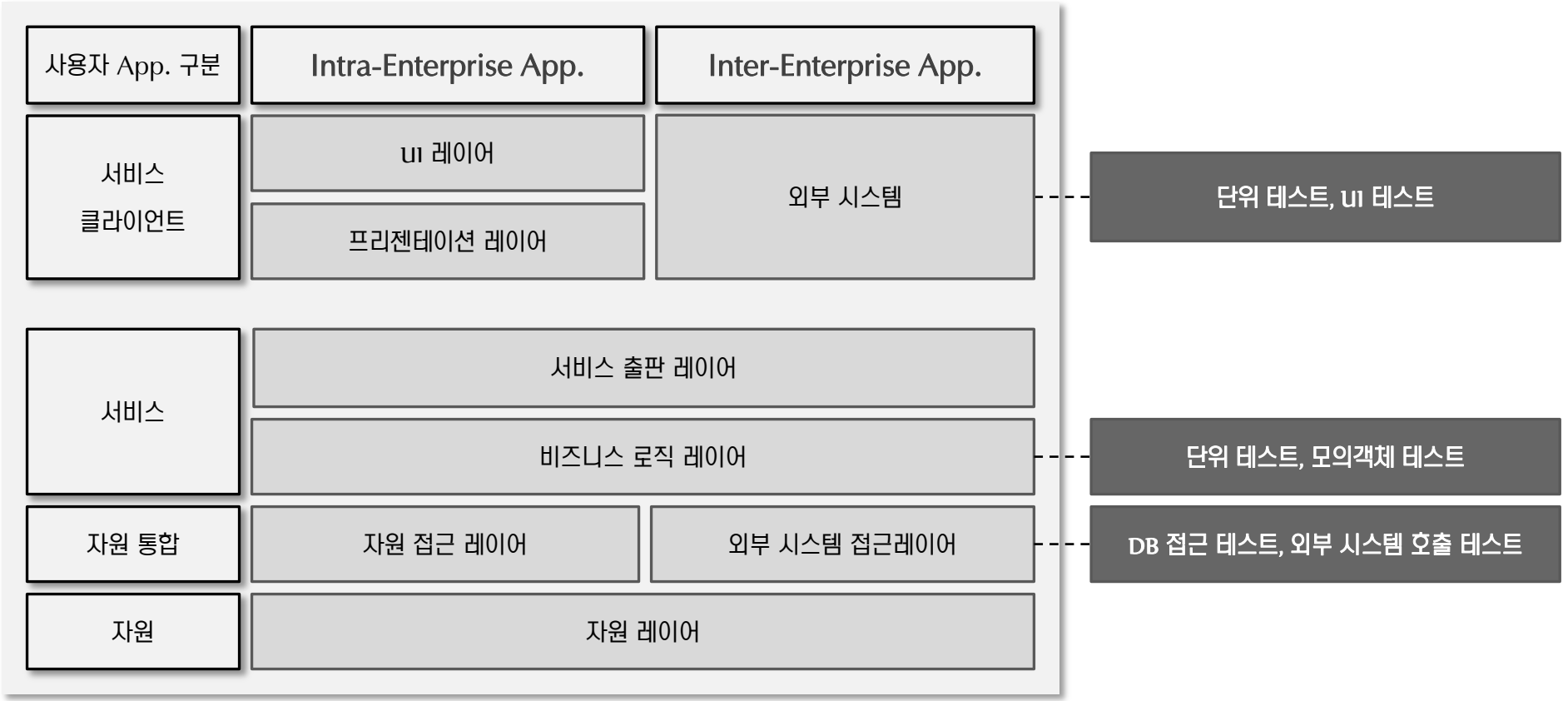
2.3 테스트와 프로세스 (3/3)

✓ 일반적인 테스팅 절차

테스트 절차는 각 프로젝트의
규모/상황에 맞게
조정될 필요가 있으며,
테스트 관련 산출물 역시
테스트 유형과 접근 방법에 따라
조정됨.

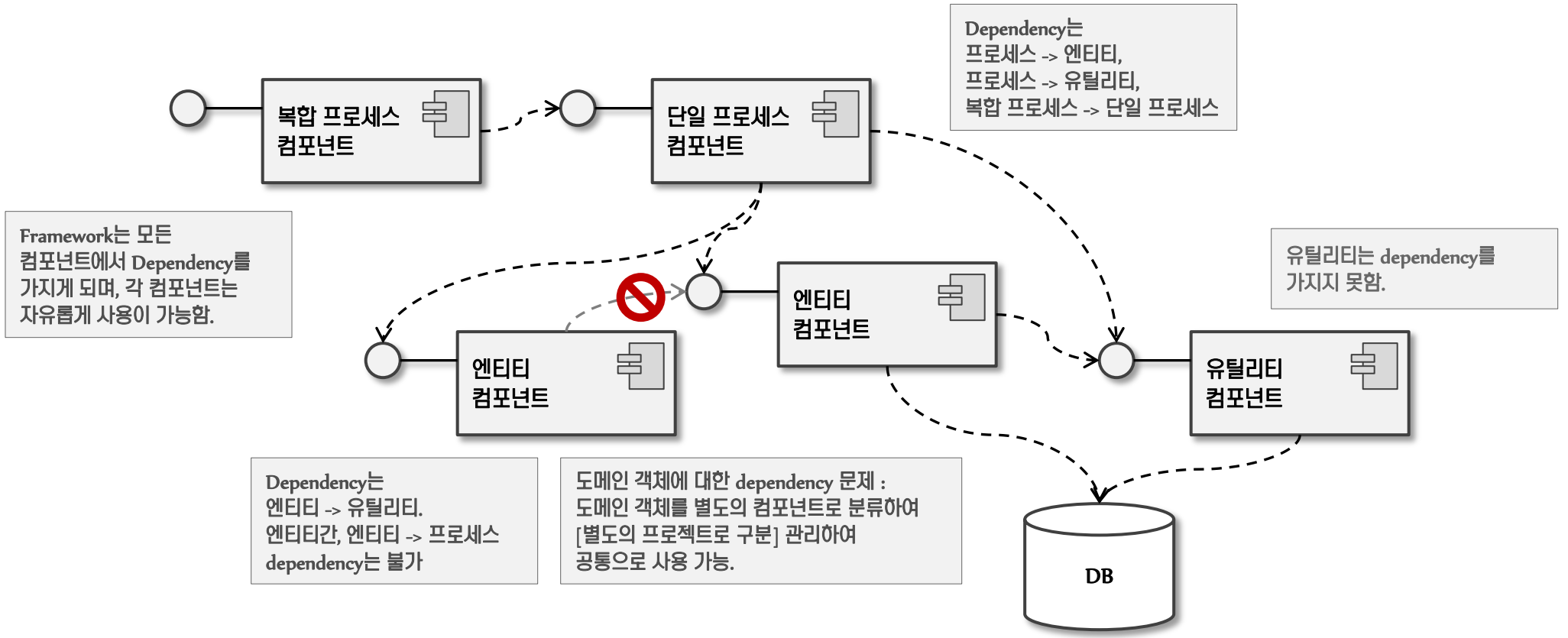
2.4 테스트와 아키텍처 (1/3)

✓ 테스트가 용이한 아키텍처 설계는 애플리케이션에서 서로 다른 기능은 다른 계층에서 수행하도록 분리하는 것입니다.
✓ 다른 애플리케이션이나 자원에 접근을 시도하는 대신 모의객체를 사용해 비즈니스 로직을 고유한 계층으로 분리합니다.
✓ 프리젠테이션 계층을 비즈니스 로직과 자원 접근에서 분리할 수 있다면 입력 유효성을 신속하게 테스트할 수 있습니다.



2.4 테스트와 아키텍처 (2/3)

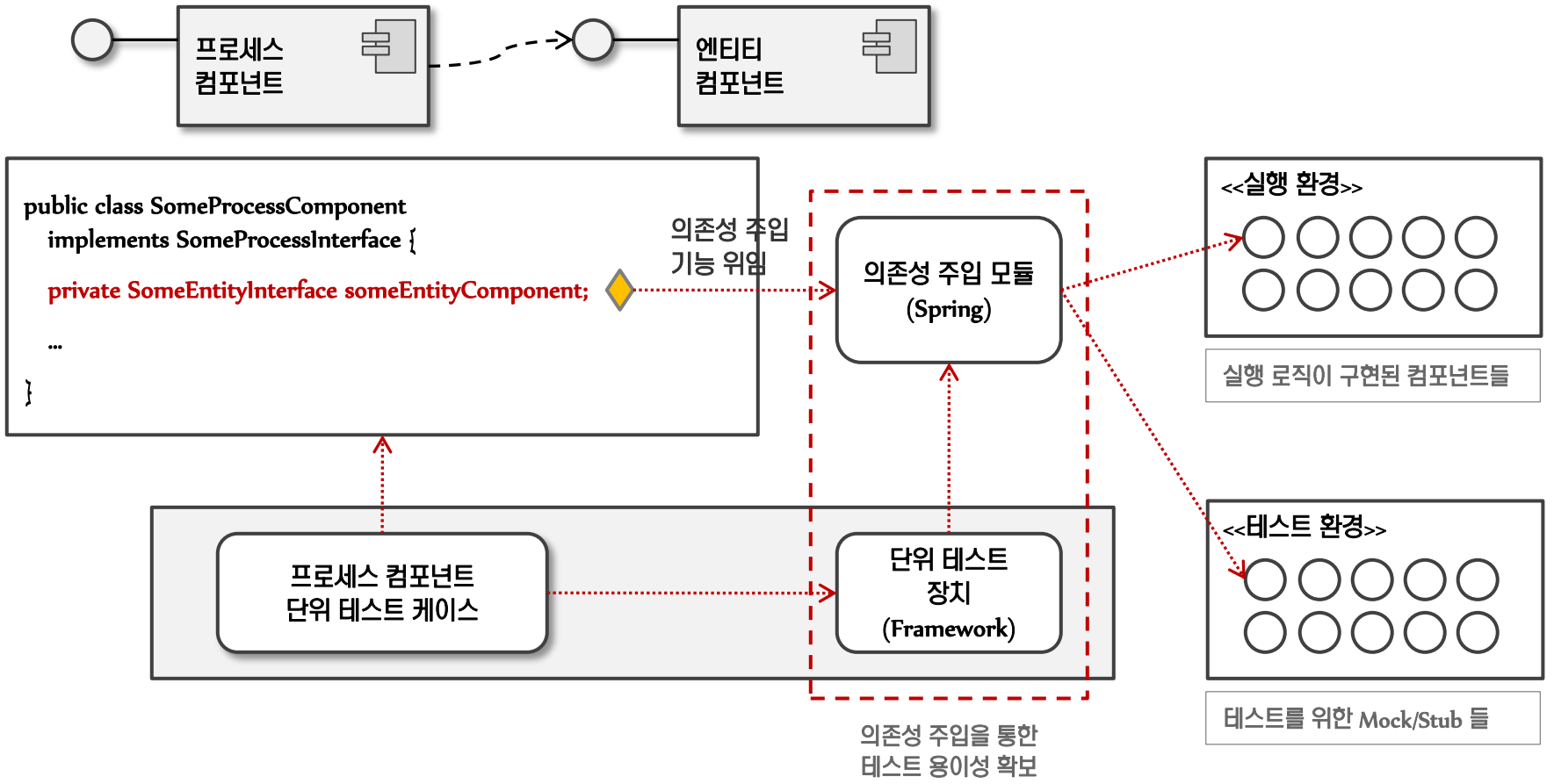
- ✓ 측정 가능한 단위로 비즈니스 로직을 설계해야 합니다.
- ✓ 테스트가 가능한 크기로 설계해야 합니다.
- ✓ 소스코드와 테스트 코드는 동일한 패키지로 작성하되, 물리적으로는 분리되어있어야 합니다.
- ✓ 의존관계(dependency) 구성 시 상호참조(cycling)하지 않도록 주의해야 합니다.



2.4 테스트와 아키텍처 (3/3)

✓ 의존성 주입을 통해 의존관계를 형성합니다.

✓ 느슨한 의존관계를 통한 테스트 용이성을 확보합니다.



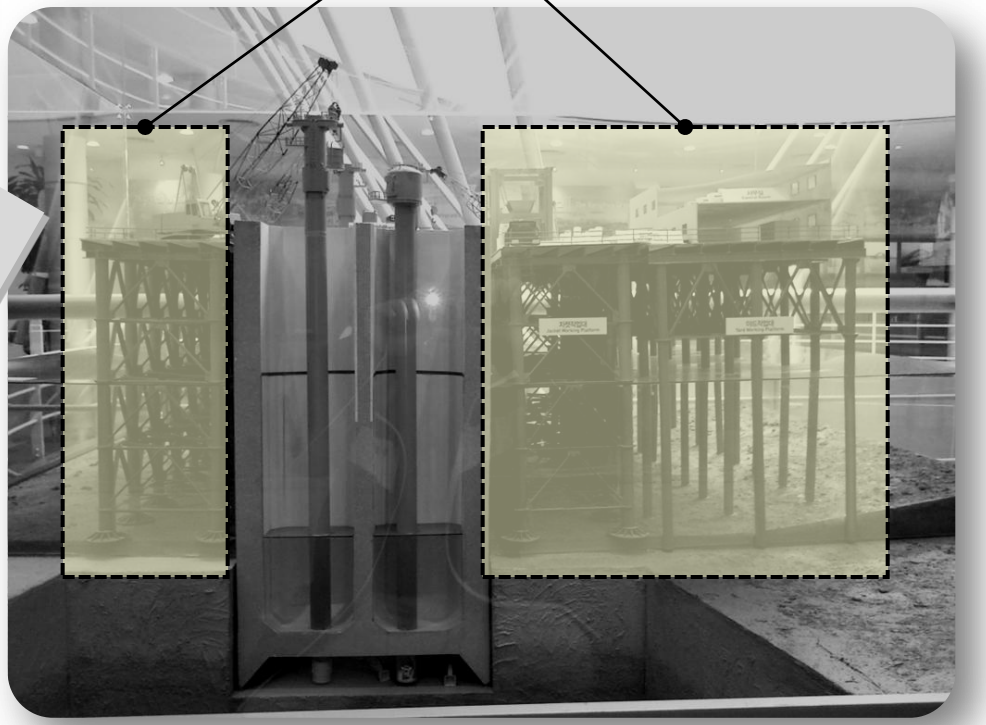
2.5 테스트와 개발환경 (1/5)

✓ 소프트웨어의 규모에 따라 개발환경의 규모를 결정합니다.

✓ 철산대교 공사현장과 인천대교 공사현장은 다를 수 밖에 없습니다.

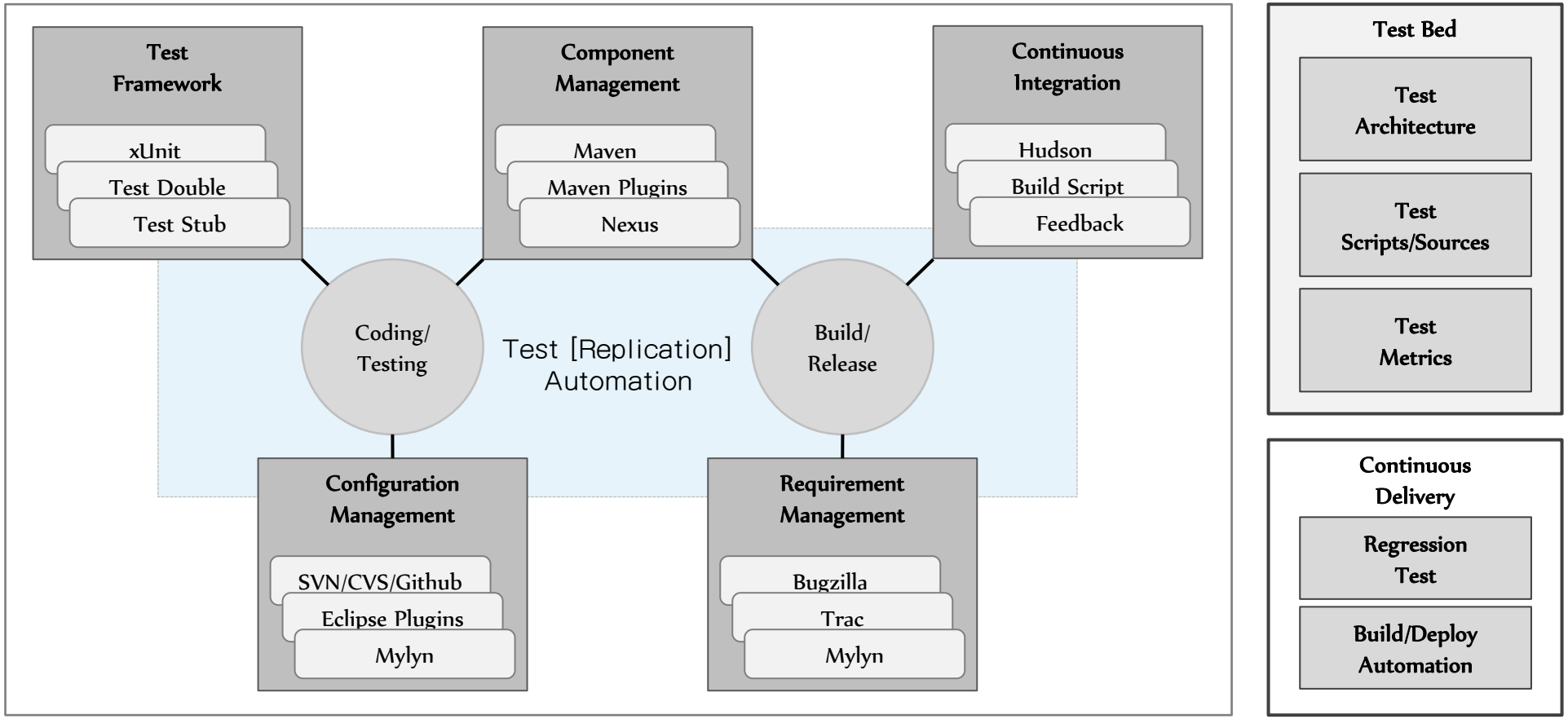
출처 : http://www.incheonbridge.com/Data/ADMIN_GALLERY/희망의문_L.gif

작업자(개발자)가 안전하고 편하게 작업(구현)할 수 있는
[작업] 영역은 구현 대상의 규모에 따라 달라짐



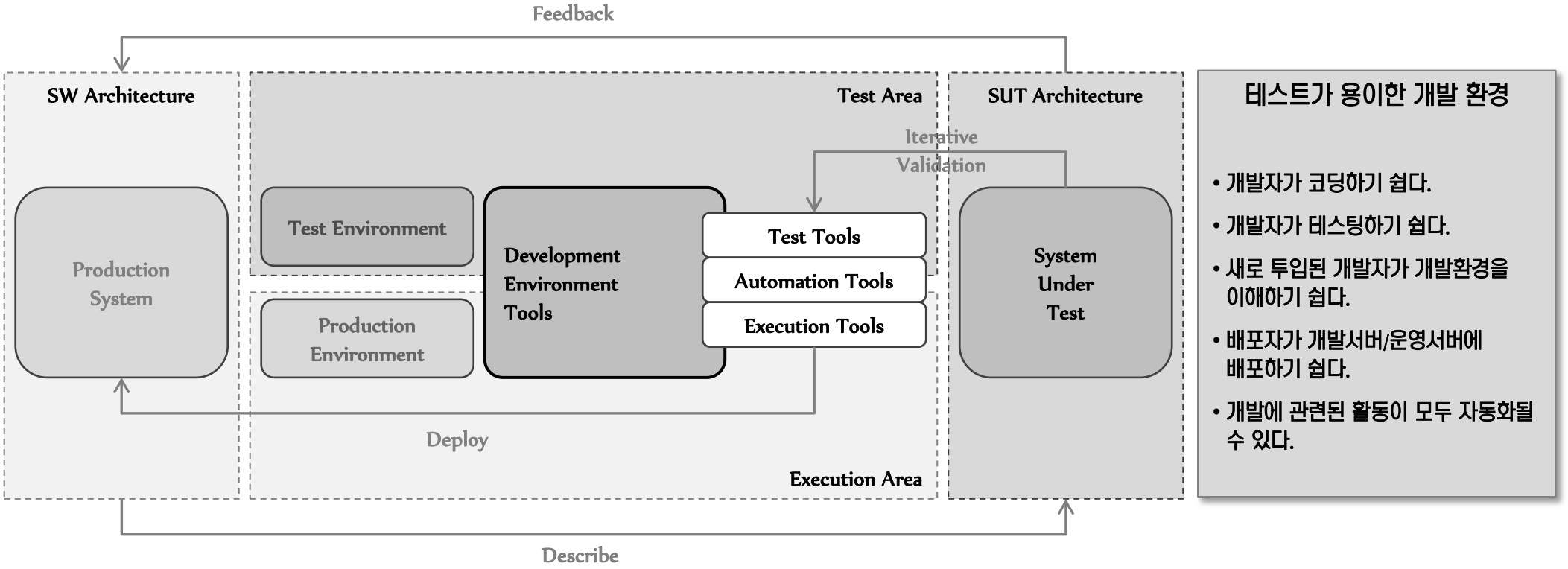
2.5 테스트와 개발환경 (2/5)

- ✓ 개발자가 장비 연결 없이 테스트를 수행할 수 있는 환경을 구축합니다.
- ✓ 개별 로직에 대한 단위테스트를 수행합니다.
- ✓ 테스트, 컴포넌트 빌드, 배포, 소스코드 관리 등을 위해 많은 도구를 사용합니다.



2.5 테스트와 개발환경 (3/5)

✓ 개발자 환경이 운영 환경과 유사할수록 환경 설정의 노력이 많이 들어갑니다.
✓ 각 개발자의 테스트 환경 역시 형상관리 대상이 되어야 합니다.
✓ 작업 환경이 풍부해질수록(최적화될수록) 품질은 비례하며, 개발 생산성 역시 증가합니다.

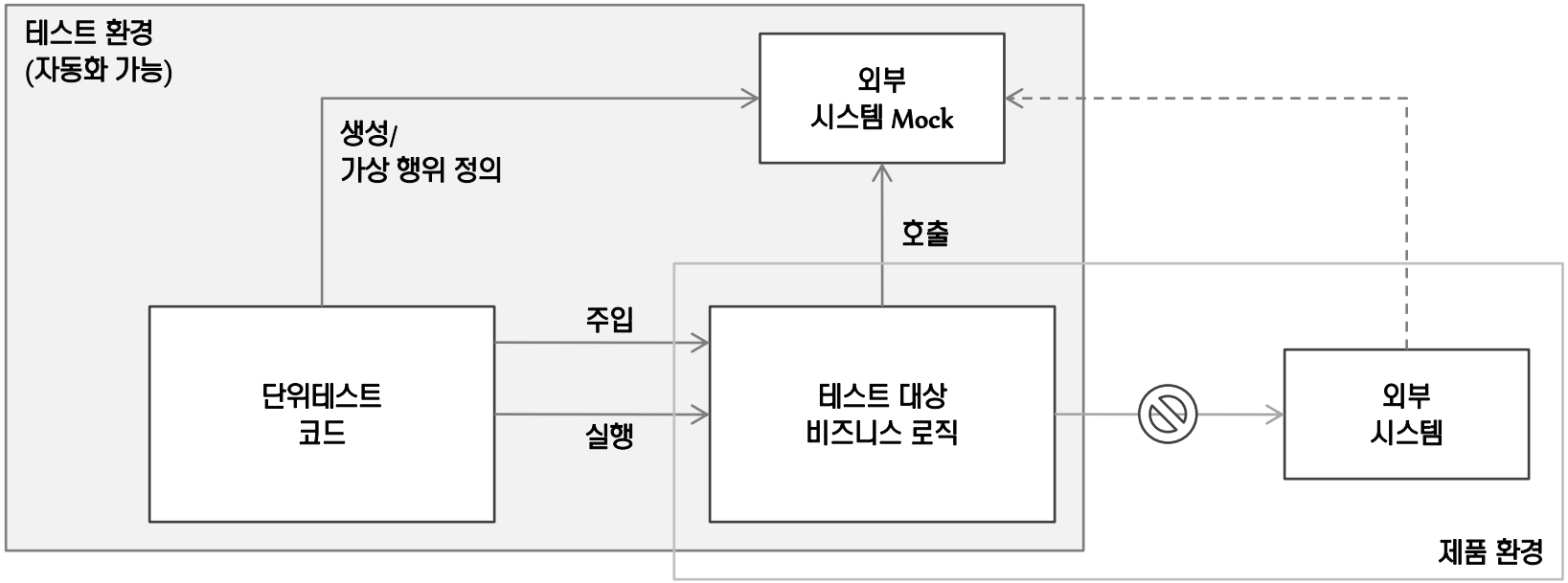


테스트가 용이한 개발 환경

- 개발자가 코딩하기 쉽다.
- 개발자가 테스팅하기 쉽다.
- 새로 투입된 개발자가 개발환경을 이해하기 쉽다.
- 배포자가 개발서버/운영서버에 배포하기 쉽다.
- 개발에 관련된 활동이 모두 자동화될 수 있다.

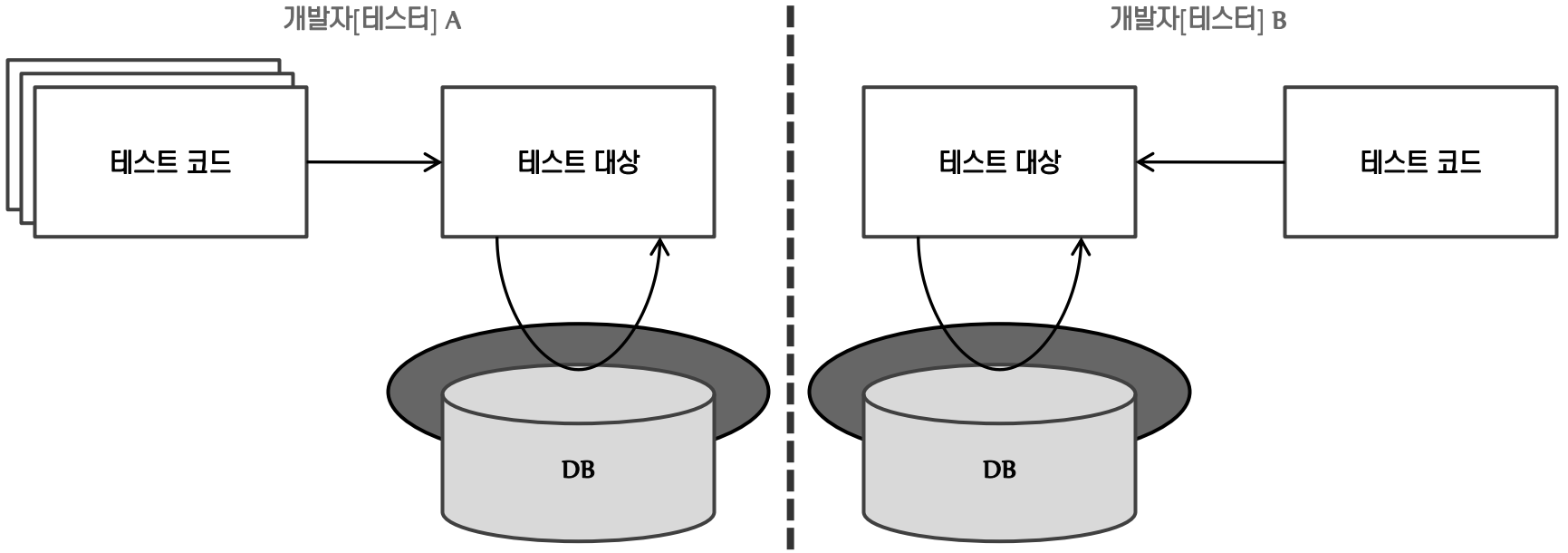
2.5 테스트와 개발환경 (4/5)

- ✓ JUnit, EasyMock, Mockito, Unitils, DBUnit 등의 단위테스트 프레임워크를 사용합니다.
- ✓ 의존성 관계를 해결하기 위해 Maven을 사용합니다.
- ✓ 의존성 주입을 사용해서 외부 시스템을 테스트하기 용이하도록 설계합니다.
- ✓ 외부 인터페이스를 테스트하기 위해 모의객체(mock)를 사용합니다.



2.5 테스트와 개발환경 (5/5)

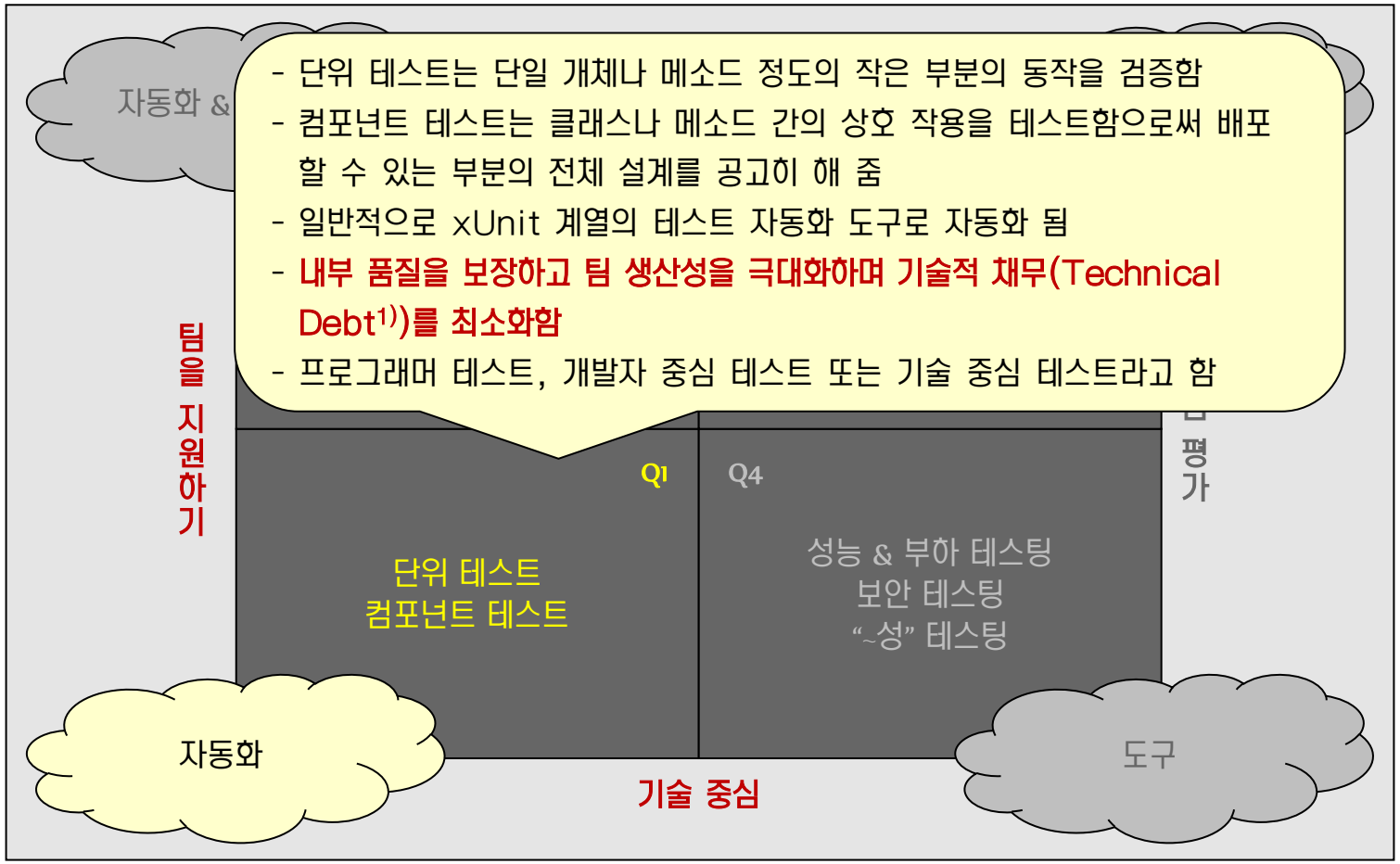
- ✓ DB 테스트를 진행하기 위해서 테스트용 DB 또는 테이블을 생성합니다.
- ✓ 개발자들끼리 테스트 DB 환경 분리를 통한 독립성을 유지합니다. (Sandbox)
- ✓ 지속적인 테스트를 하기 위해서는 INSERT 테스트 후에 테스트 데이터를 삭제합니다.
- ✓ SELECT 테스트를 하기 위해서 테스트 전 초기데이터를 생성합니다.



개발자[테스터]마다 분리된 DB 사용
(DB Sandbox)

2.6 팀을 지원하는 기술 중심 테스트의 목적

✓ 주 목적은 테스트 주도 개발(TDD)를 통해서 프로그래머가 자신의 코드를 잘 설계하도록 돕는 것입니다.



2.7 테스트와 디버그

✓ 테스트(Test)

- 주요 이해관계자들에게 테스트 대상 제품 또는 서비스의 품질에 대한 정보를 제공하는 조사과정을 의미합니다.
- 오류를 발견하기 위한 방법을 말합니다.
- 요구사항 명세서에 기술 했던 내용들에 대한 타당성을 검토합니다.
- 요구된 소프트웨어 제품 기능 타당성을 검토합니다.

✓ 디버그(Debug)

- 컴퓨터 프로그램의 정확성이나 논리적인 오류(버그)를 찾아내는 과정을 의미합니다.
- 이미 발견된 오류의 원인을 진단하고 수정하는 방법을 의미하기도 합니다.

✓ 테스트 vs 디버그

- 대다수의 개발자들이 전체 테스트 시간의 많은 부분을 디버깅을 하는데 소요합니다.
- 선(先) 테스트를 늘리면, 디버깅을 통한 오류 수정시간을 단축할 수 있습니다.
- 에러 발생 가능한 코드의 작은 부분들에 대한 테스트가 이미 존재한다면, 복잡한 프로그램의 오류도 쉽게 발견할 수 있습니다.

2.8 Test-Driven Development (TDD) (1/3)

✓ TDD의 정의

- 테스트 코드를 먼저 작성하고, 그 테스트를 통과하는 실제코드를 단계적으로 만들어 가면서 중복되는 것들은 삭제하는 개발 방법을 지칭 – 켄트 벡

✓ TDD의 목표

- 작동하는 깔끔한 코드 – 론 제프리즈[1)]

✓ TDD의 기원

- 애자일 개발 방식 중 하나의 XP의 실천 방식 중 하나 입니다.

✓ TDD의 위치

- 개발자가 처음으로 수행하는 테스트 입니다.

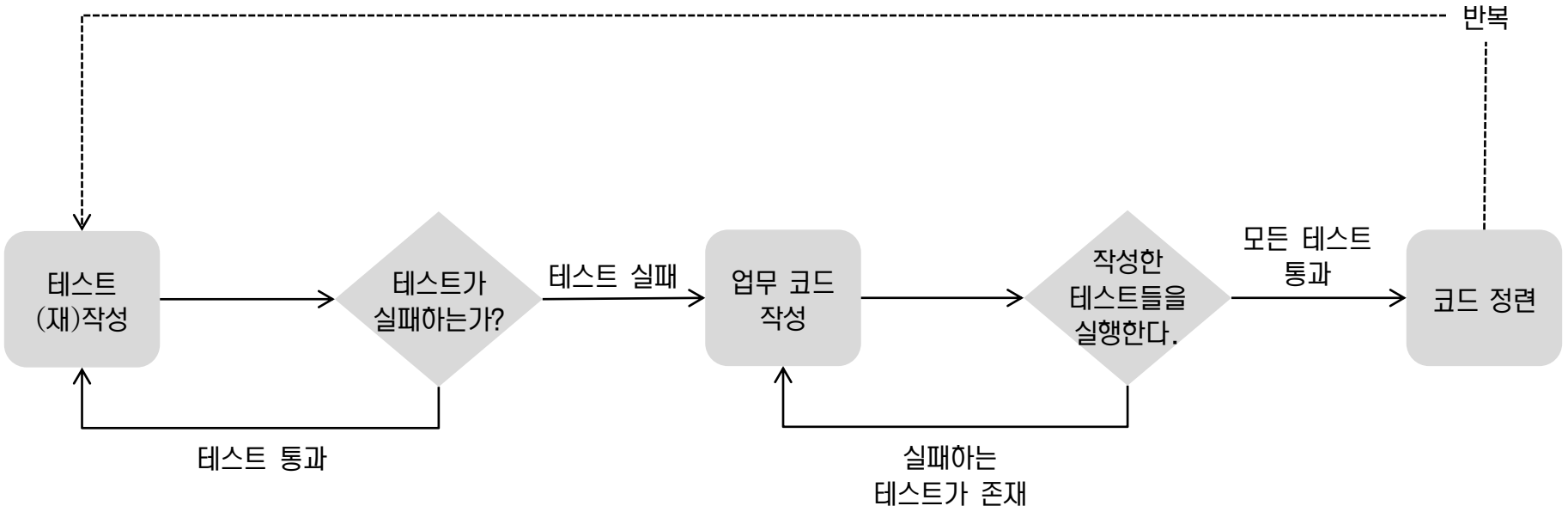
✓ TDD의 규칙

- 오직 자동화된 테스트가 실패할 경우에만 새로운 코드를 작성합니다.
- 중복을 제거합니다.

2.8 Test-Driven Development (TDD) (2/3)

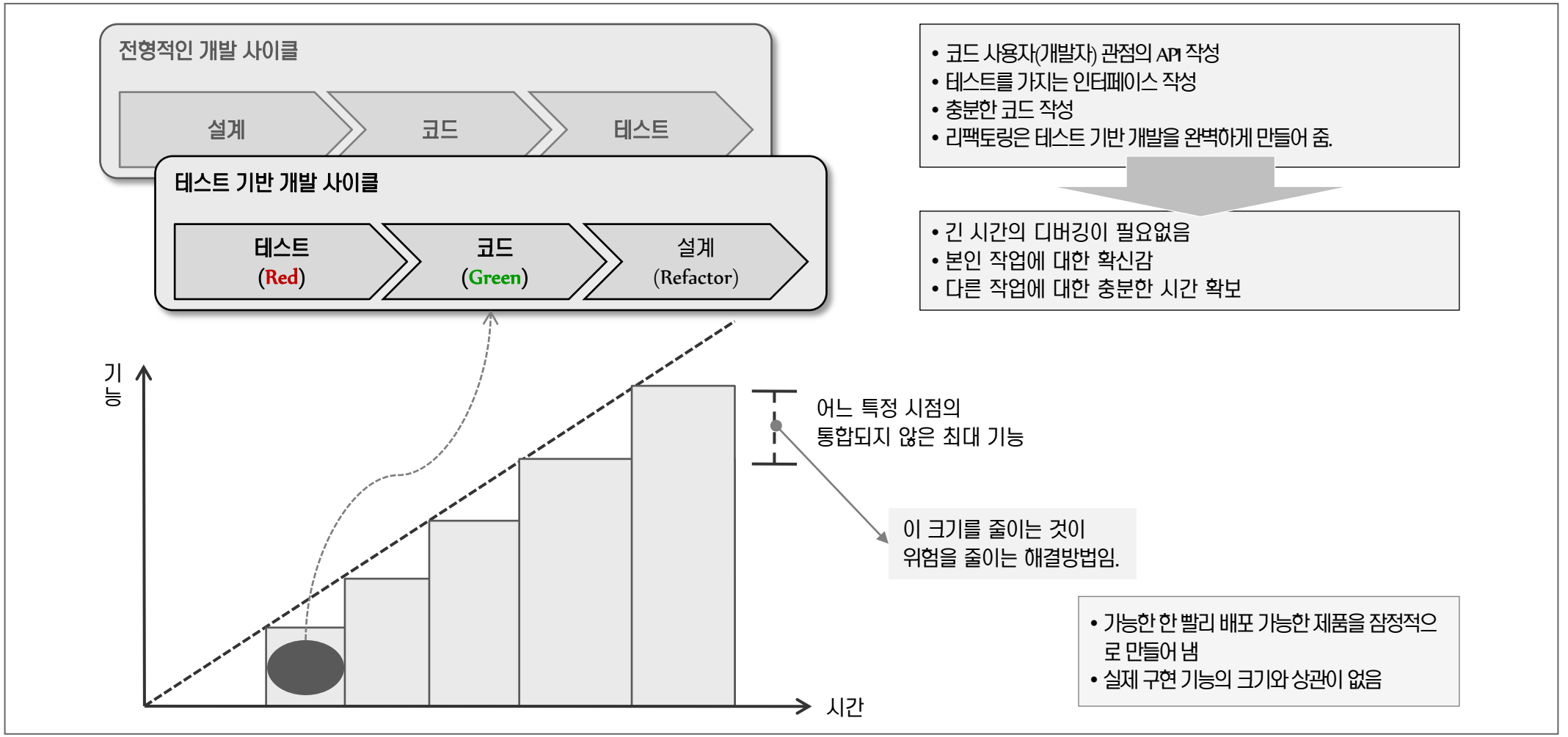
✓ TDD의 진행방식

- 질문(Ask) : 테스트 작성을 통해 시스템에 질문합니다. (테스트 수행결과는 실패)
- 응답(Respond) : 테스트를 통과하는 코드를 작성해서 질문에 대답합니다. (테스트 성공)
- 정제(Refine) : 아이디어를 통합하고, 불필요한 것은 제거하고, 모호한 것은 명확히 해서 대답을 정제합니다. (리팩토링)
- 반복(Repeat) : 다음 질문을 통해 대화를 계속 진행합니다.



2.8 Test-Driven Development (TDD) (3/3)

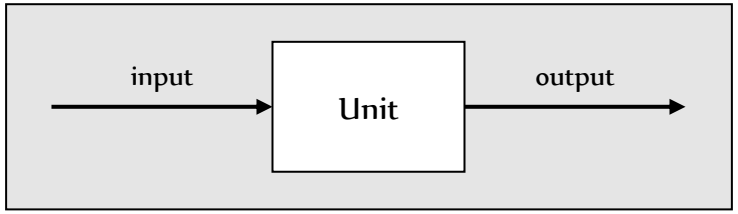
✓ 테스트 코드를 먼저 작성하고, 그 테스트를 통과하는 실제코드를 단계적으로 작성하면서 중복되는 소스는 삭제하는 개발 방법을 지칭합니다.



3. 단위테스트와 JUnit

3.1 단위테스트 소개 (1/5)

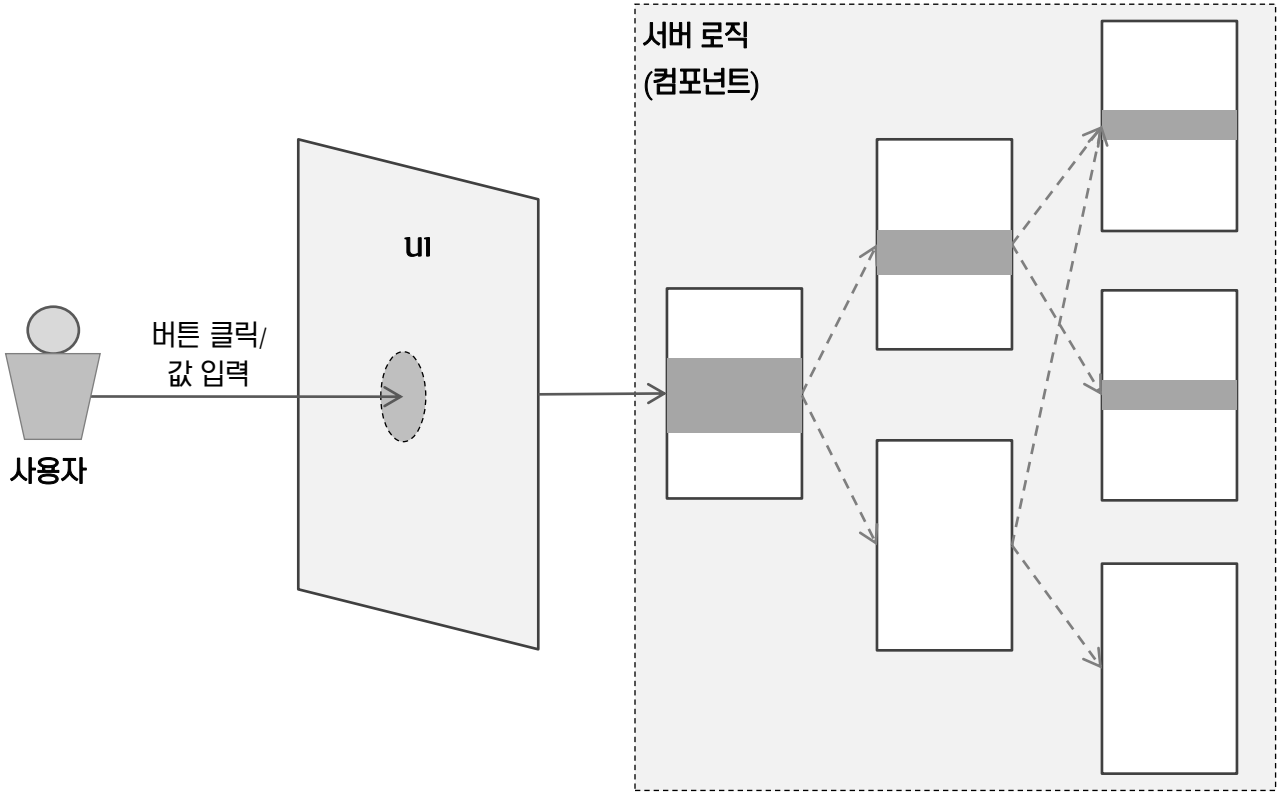
- ✓ 테스트 가능한 최소의 소프트웨어 요소(모듈, 객체 등)가 의도한 대로 동작하는지 증명하는 행위입니다.
- ✓ 하나의 단위는 대개 한 개발자가 책임을 집니다.
- ✓ 쉽게 예상 가능한 결과가 반환되는 수준의 코드 레벨을 가지게 됩니다.
- ✓ 주된 테스트 방법은 white-box 테스트이고, 에러 발생 소지가 있는 모든 부분을 테스트 합니다.



White-box test

3.1 단위테스트 소개 (2/5)

- ✓ 개발자가 본인이 작성한 로직을 테스트 합니다.
- ✓ 단위테스트는 지속적으로 수행해야 합니다. 지속적인 테스트를 위해서 도구를 사용하기도 합니다.
- ✓ 하나의 로직을 테스트하기 위해 많은 테스트코드가 작성되기도 합니다.
- ✓ 사용자(또는 테스터)가 UI에서 기능단위로 테스트를 하면 전체 코드 중 일부분만 테스트 하는 형태가 됩니다.

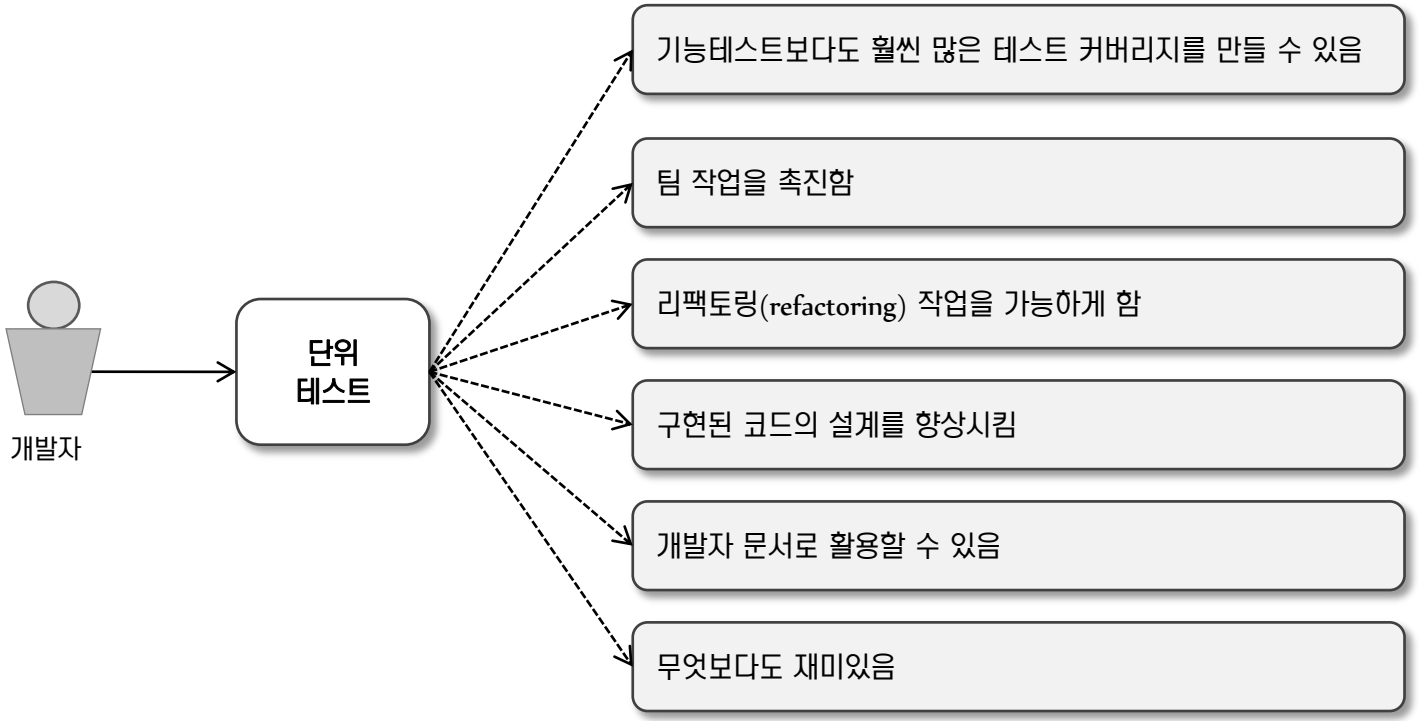


3.1 단위테스트 소개 (3/5)

- ✓ 단위테스트는 '개발자 테스트' 혹은 '객체 테스트' 라고 하며, 특정 객체를 선택해서 주변 환경 안에서 영향을 주는 역할에 상관없이 객체 자체를 테스트하는 방식을 의미합니다.
- ✓ 객체로 구성된 컴포넌트/애플리케이션 전체가 아닌 개별 객체의 메소드를 직접 호출하는 테스트를 의미합니다.
- ✓ 테스트 결과 값 검증을 통해 성공/실패 여부를 판단합니다.

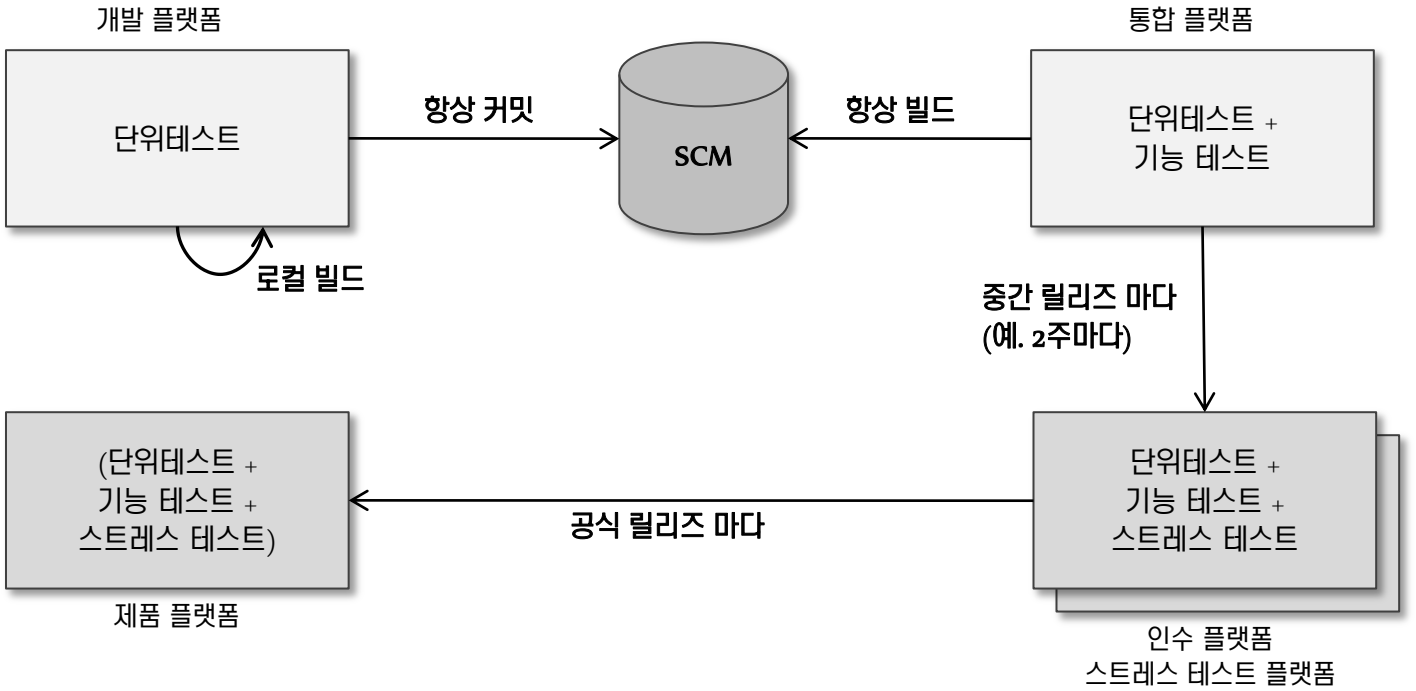
3.1 단위테스트 소개 (4/5)

- ✓ 단위테스트를 테스트 유형 중 하나라거나 품질관리 활동의 하나라고 인식하는 것보다, 적은 비용으로 보다 나은 코드를 빠르게 만들 수 있는 간단한 방법이라는 인식이 필요합니다.
- ✓ 특히, 단위테스트를 위한 개발환경과 자동화는 개발자에게 반드시 필요합니다.
- ✓ 잘 구현한 단위테스트 코드는 개발자 문서로 활용할 수도 있습니다.



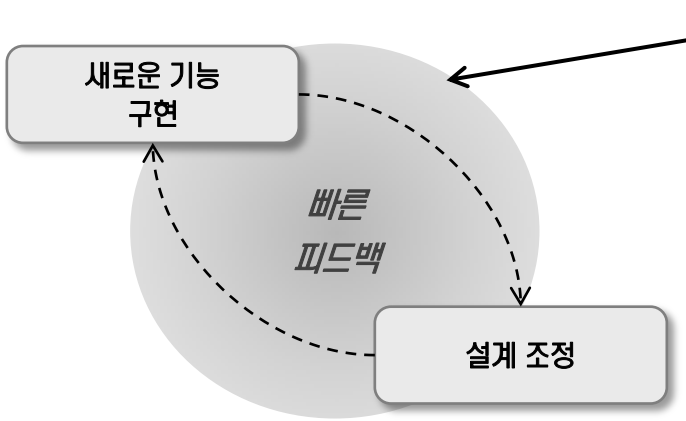
3.1 단위테스트 소개 (5/5)

✓ 단위테스트 코드 뿐만 아니라, 테스트 환경은 반드시 코드와 같이 유지해야 하고, 관리해야 할 대상입니다.
✓ 단위테스트를 통해서 다른 사람이 만든 코드를 이해할 수 있어야 합니다.
✓ 테스트가 언제든지 동일한 결과를 만들 수 있게 관리해야 애플리케이션 변경 시 코드 깨짐을 방지할 수 있습니다.



3.2 단위테스트의 목적 (1/2)

✓ 소프트웨어에 문제가 발생하게 되면 이 부분에 대한 피드백을 빠르게 전달함으로써 개발자는 문제점이 더 커지기 전에 조치할 수 있습니다.



✓ 단위 테스트는 품질 보증(Quality Assurance)의 한 방법
✓ 개발 중인 소프트웨어에 남아 있는 버그가 존재하지 않을 때까지 테스트 수행
✓ 기능 테스터는 개발이 끝난 소프트웨어를 대상으로 기능 테스트를 수행함
  • 이런 종류의 피드백은 매우 가치가 있지만,
  • 너무 늦게 발견되므로 그 가치가 줄어듬
  • 버그 수정을 위해 스케줄을 재조정해야 함
✓ 개발자가 보다 빨리 피드백을 받을 수 있는 테스트가 필요함
  • 개발자 테스트 / 단위테스트

기존 방식	점증적인 방식
모든 것을 설계 후 개발하는 방식으로 한번에 완전하고 완벽한 아키텍처를 만들기 어려움	점증적인 설계는 행위가 덧붙여짐에 따라서 조그만 증가로 코드의 구조를 조정하는 것임.
한번에 자그마한 단계를 설계하여 반복적으로 덧붙여 나아감 [점증]	개발 생명주기 동안 특정 단계에서 코드는 개발자가 현재의 기능을 지원하도록 느끼는 최상의 설계를 표현함.

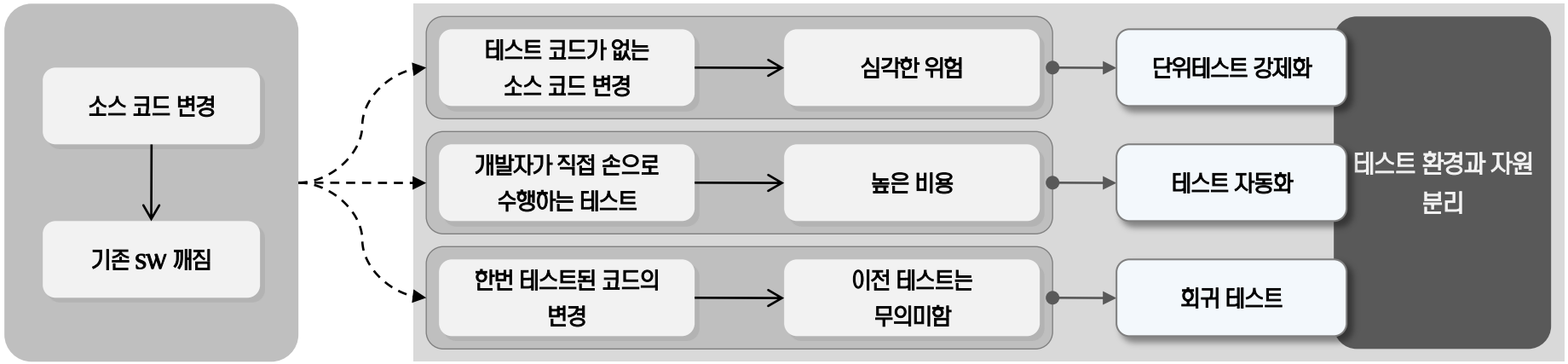
실제로 증명된 아키텍처의 점증적인 발전

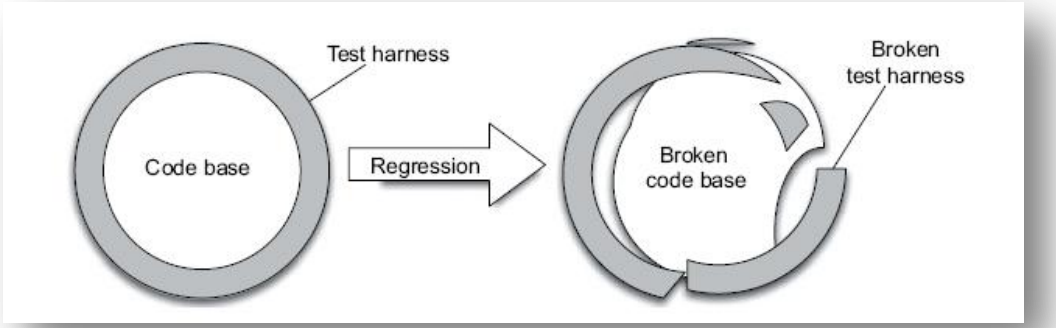
단계	수정에 필요한 상대적인 비용
요구사항 정의	$1
High-level 설계	$2
Low-level 설계	$5
코딩	$10
단위 테스트	$15
통합 테스트	$22
시스템 테스트	$50
인도 이후	$100+

*참고 : B. Littlewood, ed., Software Reliability: Achievement and Assessment (Henley-on-Thames, England: Alfred Waller, Ltd., Nov. 1987)

3.2 단위테스트의 목적 (2/2)

- ✓ 테스트 안전장치(Harness)로써, 소프트웨어를 안전하게 보호합니다.
- ✓ 소스코드 변경으로 인해 기존 SW가 깨지면 테스트 안전장치도 깨진 상태가 됩니다.
- ✓ 다시 안전한 상태가 되도록 테스트 코드를 변경합니다.
- ✓ 변경한 테스트 코드를 기반으로 이전에 수행했던 테스트를 다시 수행합니다. (회귀테스트)





- ✓ 테스트 안전장치harness는 코드 기반 주변에 틀mold을 형성함
- ✓ 코드 기반을 깨는 변경이 가해짐에 따라 틀 mold은 더 이상 균형상태가 아니며, 따라서 깨어진 상태가 됨

3.3 단위 테스트 필요성

✓ 하나의 모듈에 대해 특정 입력 값을 통해 예상되는 결과를 확인하고, 문제점을 발견하여 빨리 버그를 수정 합니다.
✓ 각각의 모듈마다 테스트 케이스가 작성되어 있어, 모듈 또는 클래스 마다 코드의 리팩토링을 손쉽게 합니다.
✓ 각각의 모듈은 수많은 테스트를 거쳐 완성되었기 때문에 시스템 통합을 간단하게 합니다.

3.4 JUnit 개요 (1/3)

✓ 단위테스트 프레임워크(JUnit)

- 1997년, Erich Gamma와 Kent Beck이 개발하였습니다.
- Kent Beck이 Smalltalk를 위해 개발한 프레임워크, SUnit의 후속입니다.
- 공개 소스 소프트웨어로서 상업적인 용도로 사용할 수 있습니다.
- IBM의 공개 라이선스 1.0 버전에서 배포되었으며, SourceForge가 후원하고 있습니다.
- Java 개발에서 단위 테스트의 de facto standard[1] 프레임워크로 빠르게 자리잡았습니다

✓ 프레임워크

- 프레임워크는 반-완성된(semi-complete) 어플리케이션을 의미합니다.
- 어플리케이션간에 공유할 수 있는 재사용 가능한 공통 구조를 제공합니다.
- 단순한 유틸리티를 제공하기 보다는 응집된 구조를 제공합니다.

1) De facto standard : 사실상 표준, 사실상의 표준. 즉, 비공식적인 표준을 의미함.

3.4 JUnit 개요 (2/3)

✓ JUnit 3.x 버전에서 4.x로 넘어오면서 몇 가지 큰 변화가 있습니다.

✓ 3.x 버전에서는 TestCase 클래스를 상속받아 테스트케이스 클래스를 작성해야 합니다.

✓ Naming 규칙이 3.x는 정해진 패턴에 따라 구현 -> 4.x는 Annotation을 통해 자유롭게 작성 가능합니다.

```
public class JUnit3Test extends TestCase {

	private List emptyList;

	// Sets up the test fixture.
	public void setUp() {
		emptyList = new ArrayList();
		System.out.println("sutUp call");
	}

	// Tears down the test fixture.
	public void tearDown() {
		emptyList = null;
		System.out.println("tearDown call");
	}


	public void testSomeBehavior() {
		assertEquals("Empty list should have 0
		elements", 0, emptyList.size());
		System.out.println("testSomeBehavior call");
	}
}
```

```
public class JUnit4Test {

	private List emptyList;

	// Sets up the test fixture.
	@Before
	public void setUp() {
		emptyList = new ArrayList();
		System.out.println("sutUp call");
	}
	// Tears down the test fixture.
	@After
	public void tearDown() {
		emptyList = null;
		System.out.println("tearDown call");
	}
	@Test
	public void testSomeBehavior() {
		assertEquals("Empty list should have 0
		elements", 0, emptyList.size());
		System.out.println("testSomeBehavior call");
	}
}
```

3.4 JUnit 개요 (3/3)

- ✓ Java 5 Annotation을 지원합니다.
- ✓ TestCase 클래스 상속받아야 한다는 제약을 해소합니다.
- ✓ Test라는 글자로 method 이름을 시작해야 한다는 제약을 해소합니다.
- ✓ 조금 더 유연한 픽스처 를제공합니다.
  - @BeforeClass, @AfterClass, @Before, @After
- ✓ 예외 테스트를 지원합니다.
  - @Test(expected=NumberFormatException.class)
- ✓ 시간제한 테스트를 지원합니다.
  - @Test(timeout=1000)
- ✓ 테스트 무시 기능을 제공합니다.
  - @Ignore("this method isn't working yet")

3.5 Main 메소드 테스트 (1/3)

✓ 테스트 프레임워크가 개발되기 이전의 테스트 방법은 어떤 방식을 사용했을까요?

```
public class Calculator {

   public static int add(int number1, int number2) {
      return number1 + number2;
   }

   public static int subtract(int number1, int number2) {
      return number1 - number2;
   }

   public static int multiply(int number1, int number2) {
      return number1 * number2;
   }

   public static int divide(int number1, int number2) {
      return number1 / number2;
   }

}
```

3.5 Main 메소드 테스트 (2/3)

✓ 테스트 프레임워크가 개발되기 이전의 테스트 방법은 콘솔출력문을 이용하여 개발자가 의도한대로 값이 지정되거나 설정되고 있는지 확인하였습니다.

```
public class Calculator {

   public static int add(int number1, int number2) {
      return number1 + number2;
   }

   public static int subtract(int number1, int number2) {
      return number1 - number2;
   }

   public static int multiply(int number1, int number2) {
      return number1 * number2;
   }

   public static int divide(int number1, int number2) {
      return number1 / number2;
   }

}
```

```
public class CalculatorTest {

   public static void main(String[] args) {
      System.out.println
         (" 덧셈 검증 " + (Calculator.add(1,2) == 3));

      System.out.println
         (" 뺄셈 검증 " + (Calculator.subtract(3,2) == 1));

      System.out.println
         (" 곱셈 검증 " + (Calculator.multiply(3,4) == 12));

      System.out.println
         (" 나눗셈 검증 " + (Calculator.divide(10,2) == 5));

   }

}
```

3.5 Main 메소드 테스트 (3/3)

- ✓ 제품 코드가 테스트 코드를 포함합니다.
- ✓ Main 메소드를 사용하면 모든 테스트를 실행하기 위해 불필요한 코드를 추가해야 합니다.
- ✓ 테스트 결과를 GUI로 확인하는 경우, 필요한 코딩을 더 추가 해야 합니다.
- ✓ 테스트 결과를 HTML 형식이나 테스트와 같은 보고서 형식으로 기록하고 싶은 경우, 추가적인 코딩을 해야 합니다.
- ✓ 테스트 하나가 실패하면 다른 테스트가 동작하지 않습니다.
- ✓ 다른 사람이 작성한 테스트 코드를 이해하기 어렵습니다.

1) 사실상 표준, 사실상의 표준. 즉, 비공식적인 표준을 의미함.

3.6 단위테스트 작성 (1/8)

- ✓ JUnit 단위테스트 클래스의 기본 구조입니다.
- ✓ 한 클래스에 여러 개의 테스트 메소드를 작성할 수 있습니다.
- ✓ 테스트 클래스의 이름은 [테스트 대상 클래스이름+Test]와 같은 형태로 명명하는 것이 좋습니다.
- ✓ @Before, @After 어노테이션을 사용해서 테스트 전, 후 작업을 정의할 수 있습니다.

```
public class CalculatorTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void test() {
        double result = Calculator.add(10, 50);
        assertEquals(60, result, 0);
    }

}
```

xxxTest 형식으로명명

testXXX 메서드가 실행되기 전에 수행.
테스트를 위한 사전 작업을 담당.

testXXX 메서드가 실행된 후 수행.
테스트 종료 후 작업을 담당.

자동으로 수행될 단위 테스트 메서드.
@Test 어노테이션으로 인식

단위 테스트를 수행할 객체의 인스턴스를 생성하고 결과를 얻음

단정 메서드를 통해 테스트 결과가 정상적으로 수행되었는지 확인

3.6 단위테스트 작성 (2/8)

- ✓ Money 클래스를 테스트하는 MoneyTest 클래스를 작성합니다.
- ✓ 테스트 대상 메소드를 호출한 후 검증 메소드(assert)를 이용해서 수행결과를 검증합니다.
- ✓ 테스트 메소드에서 검증 메소드(assert)를 여러 번 사용할 수 있습니다.
- ✓ JUnit 실행결과 화면에서 초록색 막대는 테스트 통과, 빨간색 막대는 테스트 실패를 나타냅니다.

```java
import static org.junit.Assert.assertEquals;

import java.util.Currency;
import java.util.Locale;

import org.junit.Test;

/**
 * Money 객체 테스트
 *
 * @author Elvis
 */
public class MoneyTest {

    /**
     * Money 객체의 add 메소드 단위 테스트<br/>
     * 한국 돈과 미국 돈의 덧셈. 환율 인터페이스 호출 포함.
     *
     * @throws Exception
     */
    @Test
    public void testAdd() throws Exception {
        Money addend = new Money(5000); // 5천원
        addend.setCurrencyExchangable(new CurrencyExchangeImpl());  // 환율구현 클래스
        Money augend = new Money(20.05, Currency.getInstance(Locale.US));   // 20.05 $

        Money sum = addend.add(augend); // 5천원 + 20.05 $
        assertEquals(27055, sum.getAmount().intValue());     // 27,055 원
        assertEquals(24, sum.getAmount(Currency.getInstance(Locale.US)).intValue());    // 24 $
    }
}
```

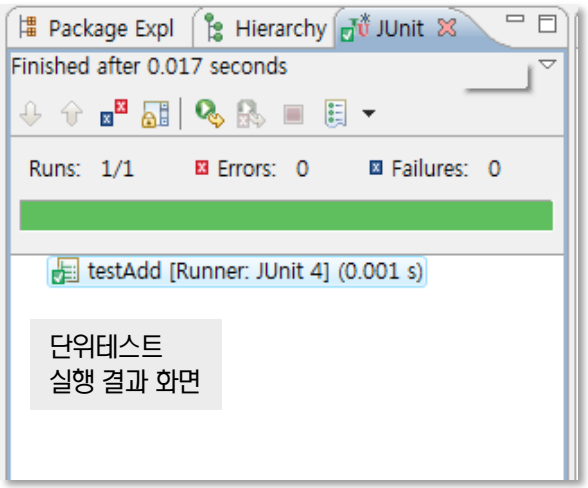
테스트케이스 클래스
(대상 객체명 뒤에 Test를 붙임)

단위테스트 코드 주석

단위테스트
어노테이션(JUnit 지원)

단위테스트 메소드
(대상 오퍼레이션 앞에 test 붙임)

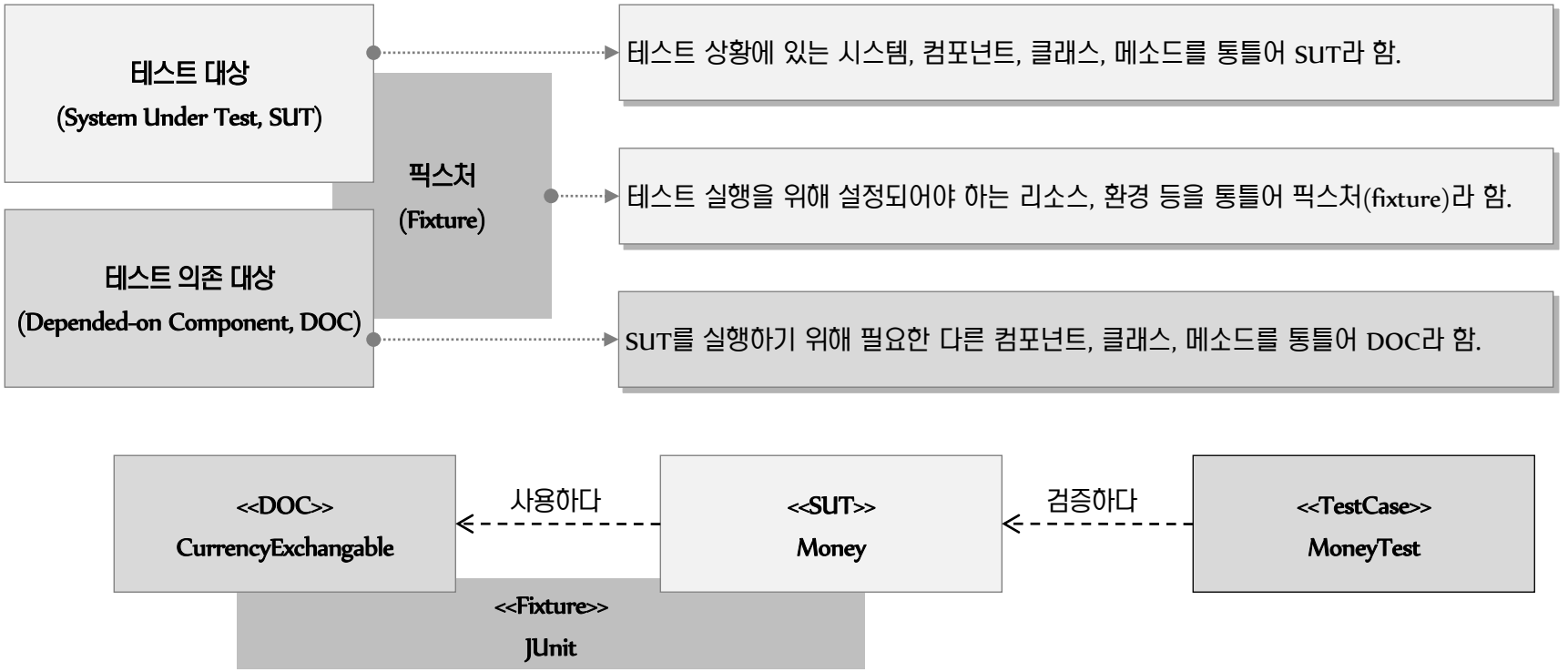
검증(assertion)
메소드 (jUnit 지원)



단위테스트
실행 결과 화면

3.6 단위테스트 작성 (3/8)

✓ 테스트 대상 (개발자가 구현한 로직)을 SUT(System Under Test)라고 합니다.
✓ SUT가 가용하는 다른 컴포넌트, 외부 시스템 등을 DOC(Depended on Component)라고 합니다.
✓ 테스트 픽스처는 JUnit, DBUnit, Mock등 여러 가지 존재합니다.



3.6 단위테스트 작성 (4/8)

- ✓ assert 메소드를 사용해서 SUT수행결과를 검증합니다.
- ✓ assertSame()과 assertEquals()는 다르게 동작하므로 주의해서 사용해야 합니다.
- ✓ assertNull(), assertNotNull()을 사용해서 수행결과 객체의 null여부를 검증합니다.
- ✓ 검증 실패시 표시할 메시지를 지정할 수 있습니다.

assertTrue(boolean condition)	Condition이 true일 경우 검증 성공. false는 검증 실패
assertEquals(Object expected, Object actual)	expected 객체와 actual 객체가 equals 오퍼레이션에 의해 동일하지 않으면 실패.
assertEquals(int expected, int actual)	expected 값과 actual 값이 == 오퍼레이션에 의해 동일하지 않으면 실패. 여러 primitive 값에 대해 overloading 오퍼레이션들이 존재
assertSame(Object expected, Object actual)	expected 객체와 actual 객체가 메모리에서 서로 다른 객체를 참조하면 실패. assertSame에서 실패한 객체라고 하더라도 equals 비교에서 동일할 수 있음.
assertNull(Object object)	object가 null이면 성공

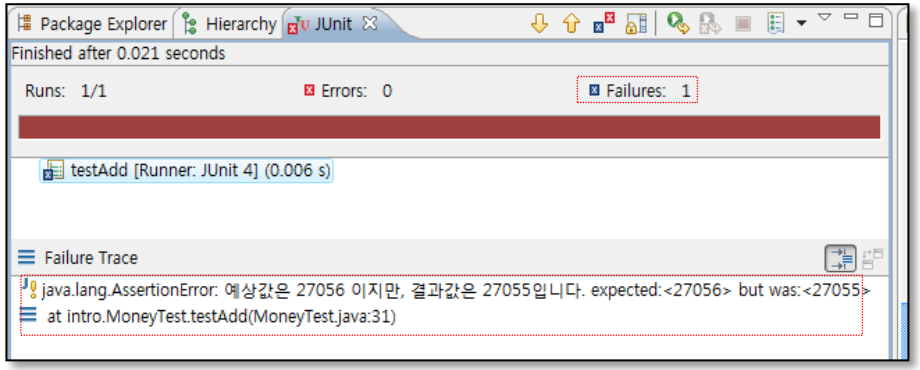
위의 검증 메소드 이외에 assertArrayEquals, assertFalse, assertNotNull, assertNotSame, assertThat 메소드가 있습니다.

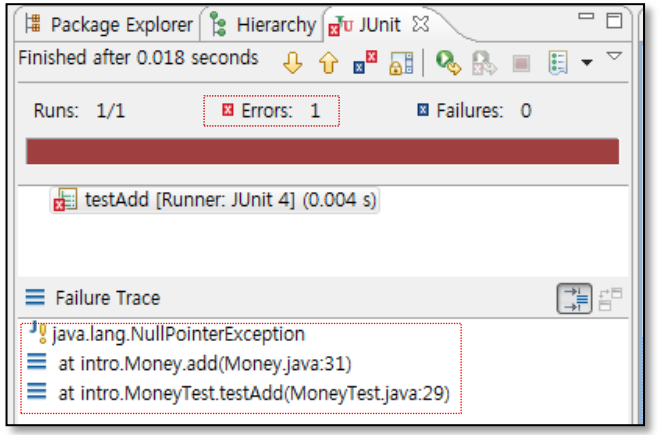
```
Money sum = addend.add(augend); // 5천원 + 20.05 $
int expectedValue = 27055;
assertEquals("예상값은 " + expectedValue + " 이지만, 결과값은 " + sum.getAmount().intValue() + "입니다.",
        expectedValue, sum.getAmount().intValue()); // 27,055 원
expectedValue = 24;
assertEquals("예상값은 " + expectedValue + "이 결과값과 다릅니다.",
        24, sum.getAmount(Currency.getInstance(Locale.US)).intValue()); // 24 $
```

검증 실패 메시지 지정

3.6 단위테스트 작성 (5/8)

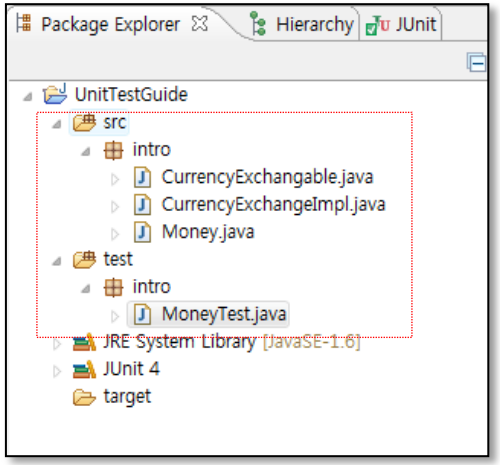
- ✓ 테스트 케이스가 정상적으로 수행되었지만, 검증 결과가 fail인 경우를 테스트 실패(failure)라고 합니다.
- ✓ 테스트 케이스 또는 테스트 대상 로직 수행 중 예외로 인해 수행이 중단된 경우를 에러(error)라고 합니다.
- ✓ JUnit은 테스트 실패 또는 에러 사유를 화면에 표시합니다.
- ✓ 테스트 실패나 에러 모두 로직을 점검하고 다시 수행해야 합니다.



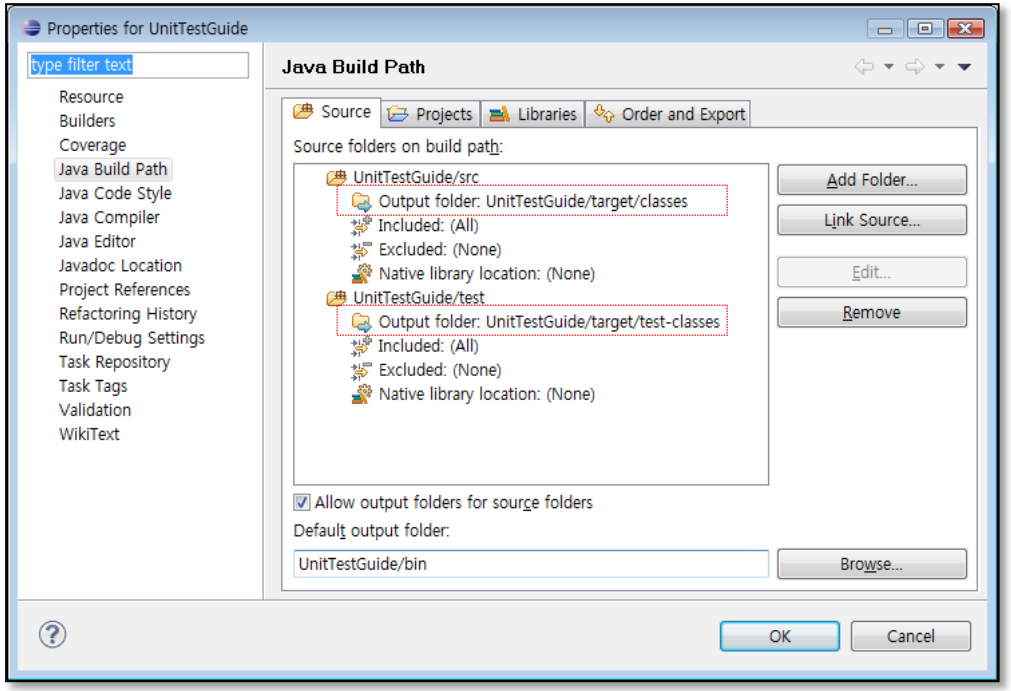


3.6 단위테스트 작성 (6/8)

- ✓ 소스코드와 단위 테스트 코드를 분리합니다. 소스코드는 src 폴더 하위에, 테스트 코드는 test 폴더 하위에 작성합니다.
- ✓ 물리적으로는 다른 위치에 작성하지만, 패키지는 동일한 패키지를 유지해야 테스트 코드를 관리하기 용이합니다.
- ✓ 예를 들어 intro 패키지의 Money.java를 테스트하기 위한 테스트 케이스도 intro 패키지 하위에 MoneyTest.java 파일로 작성합니다.



테스트 대상 소스 코드와 단위테스트 코드와의 분리

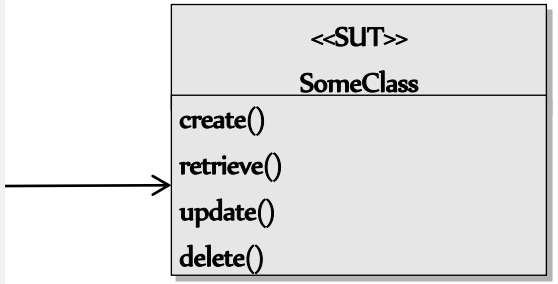


빌드 결과(바이너리 파일)에 대한 분리

3.6 단위테스트 작성 (7/8)

✓ 오퍼레이션 기준으로 단위테스트 작성하기보다는 행위를 초점으로 작성합니다.
✓ 단일 오퍼레이션에 대한 테스트라도 다른 오퍼레이션을 호출할 수도 있습니다.
✓ 즉, 생성(create) 오퍼레이션 테스트는 조회(retrieve) 오퍼레이션을 호출해서 검증할 수 있습니다.

```
package com.namoo.xunit.junit;

public class IsSomeClassTest {
    ...

    @Test
    public void testCreate() throws Exception {
        ...
        someClass.create();
        ...
        assert…(expected,someClass.retrieve());
        ...
    }
}
```



3.6 단위테스트 작성 (8/8)

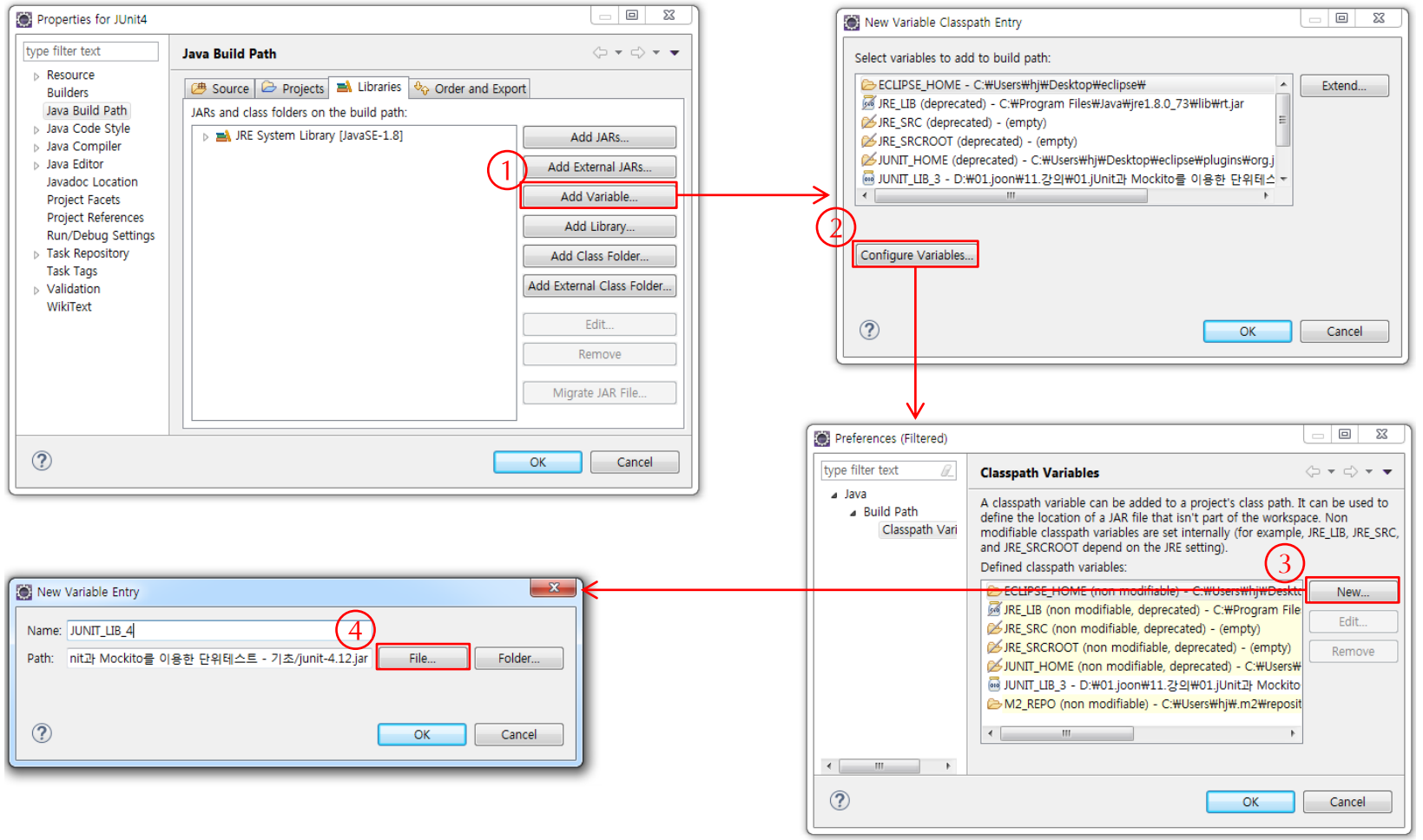
- ✓ 하나의 오퍼레이션은 N개의 테스트케이스 클래스/오퍼레이션이 나타납니다.
- ✓ 테스트 메소드가 많이 발생하는 경우, 테스트 클래스로 분리하여 다양한 테스트를 수행하는 것도 가능합니다.
- ✓ 예를 들면, MoneyAddTest, MoneyDelayAddTest 클래스로 분리하여 작성합니다.

```
    /**
     * 동일 금액 단위에 대한 합산 테스트
     *
     * @throws Exception
     */
    @Test
    public void testAdd_withSameCurrency() throws Exception {
        Money addend = new Money(5700); // 5천 7백원
        Money augend = new Money(3800); // 3천 8백원

        Money sum = addend.add(augend);
        assertEquals(9500, sum.getAmount().intValue());
    }
}
```

```
/**
 * Money 객체의 getAmount 메소드 테스트
 *
 * @author Elvis
 */
public class Money_getAmountTest {

    /**
     * 0 값에 대한 테스트
     *
     * @throws Exception
     */
    @Test
    public void testGetAmount_withZero() throws Exception {
        Money money = new Money(0);
        assertEquals(0, money.getAmount().intValue());
    }

    /**
     * 0이 아닌 값에 대한 테스트
     *
     * @throws Exception
     */
    @Test
    public void testGetAmount_withNonZero() throws Exception {
        Money money = new Money(3000);
        assertEquals(3000, money.getAmount().intValue());
    }

    /**
     * 환율에 대한 테스트
     *
     * @throws Exception
     */
    @Test
    public void testGetAmount_withCurrency() throws Exception {
        Money money = new Money(4000);
        money.setCurrencyExchangable(new CurrencyExchangeImpl());
        assertEquals(3, money.getAmount(Currency.getInstance(Locale.US)).intValue());
    }
}
```

3.7 개발환경 설정

- ✓ http://www.eclipse.org/downloads/packages/release/Luna/SR2 -> 이클립스 다운로드(luna 버전)
- ✓ https://github.com/junit-team/junit4/wiki/Download-and-Install -> JUnit jar 파일 다운로드(4.12 버전)



3.8 실습 (1/2) – Calculator 클래스 작성

✓ Calculator 클래스를 같이 작성해 봅시다.

```
public class Calculator {
	public int add(int num1, int num2) {
		return num1 + num2;
	}

	public int sub(int num1, int num2) {
		return num1 - num2;
	}

	public int mul(int num1, int num2) {
		return num1 * num2;
	}

	public int div(int num1, int num2) {
		return num1 / num2;
	}
}
```

3.8 실습 (2/2) – JUnit을 사용한 단위테스트 작성

✓ JUnit 프레임워크를 통해 Calculator 클래스의 단위 테스트를 작성해 봅시다.

```
public class CalculatorTest {

	// 덧셈 확인
	@Test
	public void testAdd() {

		Calculator calculator = new Calculator();
		int result = calculator.add(10, 20);
		assertEquals(30, result);
	}

	// 빼기 확인
	@Test
	public void testSub() {

		Calculator calculator = new Calculator();
		int result = calculator.sub(5, 1);
		assertEquals(4, result);
	}
```

```
	// 곱하기 확인
	@Test
	public void testMul() {

		Calculator calculator = new Calculator();
		int result = calculator.mul(5, 8);
		assertEquals(40, result);
	}

}
	// 나누기 확인
	@Test
	public void testDiv() {

		Calculator calculator = new Calculator();
		int result = calculator.div(100, 5);
		assertEquals(20, result);
	}
```

4. JUnit 기능

4.1 단정문 이해
4.2 테스트 fixture 설정
4.3 유형별 테스트
4.4 테스트 suite
4.5 JUnitParams library
4.6 비교표현의 확장: Hamcrest
4.7 사용자 관리 실습

4.1 단정문 이해 (1/2)

✓ Junit에서는 테스트 수행 결과 값을 검증할 메소드 들이 존재합니다.

- assertEquals() – 예상 값과 결과 값이 같은지 체크
- assertTrue() – 결과 값이 true 인지 체크
- assertFalse() – 결과 값이 false 인지 체크
- assertNull() – 결과 값이 null 인지 체크
- assertNotNull() – 결과 값이 not null 인지 체크
- assertNotEquals() – 예상 값과 결과 값이 다른지 체크
- assertArrayEquals() – 예상 배열과 결과 배열이 같은지 체크
- assertSame() – 예상 값의 객체와 결과 값의 객체가 같은지 체크
- assertNotSame() – 예상 값의 객체와 결과 값의 객체가 다른지 체크
- assertThat() – Hamcrest Library를 사용한다. 일반적인 큰 차이는 없지만 조금 더 읽기 편한 자연어로 표시할 수 있습니다.

4.1 단정문 이해 (2/2)

✓ 간단한 예제를 통해 단정문을 실습 해봅시다.

```
public class AssertExample {

      @Test
      public void assertEqualsTest() {

            int x = 5;
            assertEquals("예상 값과 실제 값이 다릅니다.", 5 , x);
      }

      @Test
      public void assertTrueTest() {

            boolean isFlag = true;
            assertTrue("예상 값이 'true'가 아닙니다.", isFlag);
      }

      @Test
      public void assertNullTest() {

            List<String> list = null;
            assertNull("예상 값이 'null'이 아닙니다.", list);
      }
```

```
      @Test
      public void assertArrayEqualTest() {

            String [] frutisNames = {“apple", “banana",
            “orange"};
            String [] frutisNamesTemp = {“mango", “plum",
            “ship"};

            assertArrayEquals("배열의 값이 다릅니다.",
            frutisNames, frutisNamesTemp);
      }
      @Test
      public void assertSameTest() {

            List<String> frutisNames = new
            ArrayList<String>();
            frutisNames.add(“apple");
            List<String> frutisNamesTemp = frutisNames;

            assertSame("객체가 다릅니다.", frutisNames,
            frutisNamesTemp);
      }
}
```

4.2 테스트 fixture 설정 (1/2)

✓ 테스트 수행 전 또는 수행 후 공통적으로 처리해야 할 내용이 있을 경우 관련된 부분을 정의 하는 영역입니다.

✓ 객체를 생성하는 로직, 데이터 베이스 드라이버 로딩과 연결등 초기화 작업을 정의합니다.

✓ @BeforeClass, @AfterClass, @Before, @After 어노테이션을 통해 간단하게 지정할 수 있습니다.

```
@BeforeClass
public static void setUpClass(){
    테스트 시작 전 테스트 대상 클래서에서 한 번만
    수행되어야 할 부분 설정
    ex.데이터베이스 드라이버 로딩정보
}

@AfterClass
public static void tearDownClass(){
    테스트가 끝난 후 테스트 대상 클래서에서 한 번만
    수행되어야 할 부분을 설정
    ex.데이터베이스 해제
}
```

```
@Before
public void setUp(){
    테스트 시작 전 매번 수행 되어야 할 부분 설정
    ex.데이터베이스 로그인 정보
}


@After
public void tearDown(){
    테스트 종료 후 매번 수행 되어야 할 부분 설정
    ex.데이터베이스 로그인 정보 해제
}
```

4.2 테스트 fixture 설정 (2/2)

✓ 앞에서 작성한 Calculator 클래스를 조금 수정하여 Test fixture를 실습 해봅시다.

```
public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        int result = calculator.add(10, 20);
        assertEquals(30, result);
    }

    @Test
    public void testDiv() {
        Calculator calculator = new Calculator();
        int result = calculator.div(100, 5);
        assertEquals(20, result);
    }
}
```



```
public class TestFixtureExample {

      private Calculator calculator;

      @Before
      public void setUp() {
            calculator = new Calculator();
      }

      @Test
      public void testAdd() {

            int result = calculator.add(10, 20);
            assertEquals(30, result);
      }

      @Test
      public void testDiv() {

            int result = calculator.div(100, 5);
            assertEquals(20, result);
      }
}
```

4.3 유형별 테스트 (1/7) - 동등비교 테스트

- ✓ 동등 비교 중에 equals 테스트는 테스트 대상 객체(SUT)의 equals 오퍼레이션과 관련이 있습니다.
- ✓ primitive 값이나 String과 같은 단순한 객체 비교는 assertEquals 을 사용하여 비교합니다.
- ✓ Collection 비교는 사이즈 비교와 동시에 Collection이 포함한 값까지 비교합니다.

```java
@Test
public void testNewListIsEmpty() {
    List<Object> list = new ArrayList<Object>();
    assertEquals(0, list.size());
}

@Test
public void testSynchronizedListHasSameContents_eachCompare() {
    List<String> list = new ArrayList<String>();
    list.add("Albert");
    list.add("Henry");
    list.add("Catherine");
    List<String> synchronizedList = Collections.synchronizedList(list);
    assertEquals("Albert", synchronizedList.get(0));
    assertEquals("Henry", synchronizedList.get(1));
    assertEquals("Catherine", synchronizedList.get(2));
}
```

List 객체의 size 비교에 대한 테스트는 테스트 코드를 작성하는데 있어서 empty나 null 테스트 이외에 테스트 목적에 부합하지 않음.
따라서, Collection 객체는 내부에 들어있는 값이 예상된 값으로 들어있는지를 하나하나 비교하는 것을 추천함.

값의 비교 방법

1. 검증하려는 대상과 특정 값을 적어서 비교
2. 예상되는 값을 특정 객체로 만들고 이를 서로 비교

가능하면 2번째 방법을 사용하는 것을 추천함.
예제에서 검증 대상인 synchronizedList 에 들어있는 값과 예상되는 값을 직접 String 값을 적어서 비교함.
list.add("..."); 와 assertEquals("...", ...); 의 String 값은 두 번 사용됨. 또한, 많은 테스트 코드를 유발시킴.

4.3 유형별 테스트 (2/7) - void 메소드 1

- ✓ void 메소드는 아무런 값도 반환하지 않기 때문에 이에 대한 테스트는 변화되는 값, 상태 등을 점검합니다.
- ✓ add() 오퍼레이션의 경우 contains() 오퍼레이션을 호출해서 검증할 수 있습니다.
- ✓ 변화된 상태를 감지할 수 있는 contains()와 같은 오퍼레이션이 없는 경우, 테스트를 위해서라도 구현하도록 합니다.
- ✓ 비즈니스 요구사항으로는 구현할 필요가 없지만 테스트를 위해서 구현하는 것도 충분히 의미가 있습니다.

```java
@Test
public void testListAdd() {
    List<String> list = new ArrayList<String>();
    assertFalse(list.contains("hello"));
    list.add("hello");
    assertTrue(list.contains("hello"));
}
```

```java
@Test
public void testLoadProperties() throws Exception {
    Properties properties = new Properties();
    properties.load(Thread.currentThread().getContextClassLoader()
            .getResourceAsStream("fundamental/application.properties"));
    assertEquals("jbrains", properties.getProperty("username"));
    assertEquals("1234", properties.getProperty("password"));
}
```

4.3 유형별 테스트 (3/7) - void 메소드 2

- ✓ 초기화되는 값이 올바로 생성되었는지를 점검하는 테스트를 주로 수행합니다.
- ✓ 내부 상태 값이 외부에 노출이 되지 않은 경우 (public 메소드가 아닌 경우), 별도의 장치 (동일 패키지 위치, reflection을 통한 private 접근) 등을 통해서 검증합니다.
- ✓ 플랫폼 테스트는 불필요한 코드입니다. 언어나 프레임워크가 정상 동작하는지 여부에 대해서는 테스트하지 않아도 됩니다.

```
@Test
public void testInitializationParameters() {
    Money money = new Money(3000);
    assertEquals(3000, money.getAmount().intValue());

    try {
        Field currencyField = Money.class.getDeclaredField("currency");
        currencyField.setAccessible(true);
        Object currencyValue = currencyField.get(money);
        assertTrue(currencyValue instanceof Currency);
        assertEquals(Currency.getInstance(Locale.KOREA), (Currency)currencyValue);
    } catch (SecurityException e) {
        e.printStackTrace();
    } catch (NoSuchFieldException e) {
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}
```

생성자를 통해 초기화된 외부에서 접근 가능한 내부 초기값에 대한 검증

생성자를 통해 초기화된 외부에서 접근 불가능한 내부 초기값에 대한 검증

Reflection 을 사용해서 접근 권한을 허락시키고 테스트를 수행 (소스 코드에서는 이와 같은 변수에 대한 접근 권한을 수정해서는 안됨)

```
@Test
public void testInitialization_invalid() {
    Money money = new Money(3000);
    assertNotNull(money);
    assertTrue(money instanceof Money);
}
```

생성자 실행이 not null인지, 해당 타입으로 생성되었는지에 대한 테스트를 플랫폼 테스트라고 함. (자바 언어가 기본적으로 수행하는 기능에 대한 검증)

이러한 플랫폼 테스트는 불필요한 테스트 코드임
(reflection을 사용한 로직인 경우에는 필요한 테스트임)

4.3 유형별 테스트 (4/7) – getter

- ✓ 단순한 getter는 테스트를 별도로 수행하지 않습니다.
- ✓ 다른 테스트를 수행 시에 테스트 데이터를 검증하는 과정에서 getter가 수행될 수 있습니다.
- ✓ 테스트 커버리지를 통해 getter 수행여부를 확인합니다.
- ✓ 내부 로직을 가지고 있어서 검증할 필요가 있는 경우 테스트 코드를 작성 합니다.

```java
/**
 * 소요시간에 대한 정밀도 점검
 */
@Test
public void testDurationInMinutes() {
    Song song = new Song("Bicyclops", "Fleck", 260);
    assertEquals(4.333333d, song.getDurationInMinutes(), 0.000001d);
}
```

getter 오퍼레이션의 정밀도를 검증하는 테스트 수행.
단순한 getter 오퍼레이션에 대한 테스트는 생성자 테스트 시 수행할 수 있음.

```java
public class Song {
    /** 곡명 */
    private String name;
    /** 가수명 */
    private String artistName;
    /** 소요시간 (단위는 시간) */
    private int duration;

    /**
     * 생성자
     *
     * @param name 곡명
     * @param artistName 가수명
     * @param duration 소요시간
     */
    public Song(String name, String artistName, int duration) {
        this.name = name;
        this.artistName = artistName;
        this.duration = duration;
    }

    /**
     * 소요시간을 분으로 환산
     *
     * @return 분으로 환산된 소요시간 (단위는 분)
     */
    public double getDurationInMinutes() {
        return (double) duration / 60.0d;
    }

    public String getName() {
        return name;
    }

    public String getArtistName() {
        return artistName;
    }

}
```

4.3 유형별 테스트 (5/7) – setter

- ✓ 단순한 setter는 테스트를 별도로 수행하지 않습니다.
- ✓ 해당 값에 대한 setter 메소드가 없는 경우 Test Spy와 같은 객체 주입을 이용해서 테스트 할 수도 있습니다.
- ✓ 내부 로직을 가지고 있어서 검증할 필요가 있는 경우 테스트 코드를 작성합니다.

```java
public class BankTransferActionTest {

    @Test
    public void test() {
        BankTransferAction action = new BankTransferAction();
        action.setSourceAccount("source");
        action.setTargetAccount("target");
        action.setAmount(100);

        action.execute(new Bank(){
            public void transfer(String sourceAccount, String targetAccount, int amount) {
                Assert.assertEquals("source", sourceAccount);
                Assert.assertEquals("target", targetAccount);
                Assert.assertEquals(100, amount);
            }
        }
        );
    }
}
```

setter 메소드에 대한 검증은 객체의 내부 값을 reflection을 사용해서 수행할 수도 있지만, 외부 객체로 전달되는 값에 대한 검증을 위해 중간에 특정 객체(테스트 spy)를 삽입(의존성 주입)하여 이를 모니터링 함으로써 검증이 가능함.

테스트 spy 객체의 주입을 위해서 메소드(비즈니스 로직)을 분리

```java
public class BankTransferAction {
    /** 송금 계좌번호 */
    private String sourceAccount;
    /** 이체 계좌번호 */
    private String targetAccount;
    /** 이체 금액 */
    private int amount;

    public void setSourceAccount(String sourceAccount) {
        this.sourceAccount = sourceAccount;
    }

    public void setTargetAccount(String targetAccount) {
        this.targetAccount = targetAccount;
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }

    /** 기본은행 계좌이체 */
    public void execute() {
        execute(Bank.getInstance());
    }
    /** 특정은행 계좌이체 */
    public void execute(Bank bank) {
        bank.transfer(sourceAccount, targetAccount, amount);
    }
}
```

4.3 유형별 테스트 (6/7) - 자바빈

✓ Sanity check 를 통해 자바빈의 데이터가 특정 비즈니스를 수행하기 위해 올바른지 정합성 체크를 합니다.
✓ 가벼운 정합성 체크 용도로 사용합니다.
✓ 비즈니스 로직이 복잡한 정합성 체크에 사용하면 자바빈이 복잡해질 수 있습니다.

```java
public class BankTransferAction {
    /** 송금 계좌번호 */
    private String sourceAccount;
    /** 이체 계좌번호 */
    private String targetAccount;
    /** 이체 금액 */
    private int amount;

    public void setSourceAccount(String sourceAccount) {

    public void setTargetAccount(String targetAccount) {

    public void setAmount(int amount) {

    /** 기본은행 계좌이체 */
    public void execute() {
        execute(Bank.getInstance());
    }
    /** 특정은행 계좌이체 */
    public void execute(Bank bank) {
        bank.transfer(sourceAccount, targetAccount, amount);
    }

    public boolean isReadyToExecute(){
        if(this.sourceAccount == null || this.sourceAccount.trim().length() <= 0){
            return false;
        }
        if(this.targetAccount == null || this.targetAccount.trim().length() <= 0){
            return false;
        }
        return true;
    }
}
```

```java
public class BankTransferActionTest {

    public void test() {

    @Test
    public void testIsValid() {
        BankTransferAction action = new BankTransferAction();
        action.setSourceAccount("source");
        action.setTargetAccount("target");
        action.setAmount(100);

        assertTrue(action.isReadyToExecute());
    }

    @Test
    public void testNeedAmount() {
        BankTransferAction action = new BankTransferAction();
        action.setSourceAccount("source");
        action.setTargetAccount("target");
        //action.setAmount(100);

        assertFalse(action.isReadyToExecute());
    }

}
```

4.3 유형별 테스트 (7/7) – 예외 테스트

- ✓ 비정상적인 환경을 조성하고, 기대하는 예외가 발생하는지 테스트 합니다.
- ✓ 기대와는 다른 예외가 발생하거나, 예외가 발생하지 않은 경우 실패 처리 합니다.
- ✓ 예외는 주로 외부 시스템 연결이나 인터페이스 호출 간에 발생합니다.
- ✓ 이러한 예외에 대한 테스트는 인터페이스에 대한 테스트와 같이 중요한 테스트 중의 일부입니다.

```java
/**
 * 0으로 나누는 연산에 대한 예외 테스트
 *
 * @throws Exception
 */
@Test
public void testConstructorDiesWithNull() throws Exception {
    try {
        @SuppressWarnings("unused")
        Fraction oneOverZero = new Fraction(1, 0);
        fail("Created fraction 1/0! That's undefined!");
    } catch (IllegalArgumentException expected) {
        assertEquals("denominator", expected.getMessage());
    }
}
```

```java
/**
 * 나누기 연산 객체
 *
 * @author Elvis
 */
public class Fraction {

    /** 제수 */
    private int divisor;
    /** 피젯수 */
    private int dividend;
    /** 결과값 */
    private BigDecimal result;

    /**
     * 생성자<br/>
     * divisor / dividend 의 결과값을 result에 세팅
     *
     * @param divisor 제수
     * @param dividend 피젯수
     */
    public Fraction(int divisor, int dividend) {
        if (dividend == 0) {
            throw new IllegalArgumentException("denominator");
        }
        this.divisor = divisor;
        this.dividend = dividend;

        this.result = BigDecimal.valueOf(this.divisor).divide(
                BigDecimal.valueOf(this.dividend));
    }
```

4.4 테스트 suite

- ✓ 관련된 단위테스트들을 묶어서 수행합니다.
- ✓ @RunWith 어노테이션을 이용해서 테스트 슈트 클래스를 지정합니다.
- ✓ @SuiteClasses 어노테이션을 이용해서 묶어서 실행할 테스트 클래스를 지정합니다.
- ✓ 테스트 자동화목적으로는 사용하지 않습니다. (테스트 자동화는 별도의 도구를 사용하는 방식을 더 선호합니다.)

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;


@RunWith(Suite.class)
@SuiteClasses({
    ArrayToArrayListTest.class, BankTransferActionTest.class,
    ListIteratorTest.class, ListTest.class,
    MoneyConstructorTest.class, PropertiesTest.class,
    RightExceptionTest.class, SongTest.class
})
public class AllTests {

}
```

4.5 JUnitParams library

- ✓ 파라미터를 이용하여 다양한 case에 대해 하나의 테스트 메소드에서 할 수 있도록 도와주는 library 입니다.
- ✓ 입력데이터와 테스트 로직 부분을 따로 관리하여 테스트 메소드 로직부분을 깔끔하게 작성할 수 있습니다.
- ✓ 테스트를 수행하는 메소드는 테스트 데이터에 따른 case에 대해 신경을 쓰지 않아도 됩니다.

```
public class MoneyTest {

      @Test
      public void
      constructorShouldSetAmountAndCurrency() {
         Money money = new Money(10, "USD");

         assertEquals(10, money.getAmount());
         assertEquals("USD", money.getCurrency());

         money = new Money(20, "EUR");

         assertEquals(20, money.getAmount());
         assertEquals("EUR", money.getCurrency());
      }
}
```



```
@RunWith(JUnitParamsRunner.class)
public class MoneyParameterizedTest {
      private static final Object[] getMoney() {
            return new Object[] {
                  new Object[] { 10, "USD" },
                  new Object[] { 20, "EUR" }
            };
      }

      @Test
      @Parameters(method = "getMoney")
      public void constructorShouldSetAmountAndCurrency
      (int amount, String currency) {

            Money money = new Money(amount, currency);

            assertEquals(amount, money.getAmount());
            assertEquals(currency, money.getCurrency());
      }
}
```

4.6 비교표현의 확장: Hamcrest (1/7)

- ✓ Hamcrest는 jMock 라이브러리 저자들이 참여해 만든 테스트 표현식 입니다.
- ✓ Hamcrest는 다양한 Matcher들이 모인 Macher 집합체 입니다.
- ✓ Macher라는 말 그대로 어떤 값들의 상호 일치 여부나 특정한 규칙 준수 여부 등을 판별하기 위한 것입니다.
- ✓ 공학적인 느낌보다는 조금 더 부드러운 문맥적인 흐름을 만들어 주도록 합니다.

```
import org.junit.Test;
import static org.hamcrest.CoreMatchers.*;

public class HamcrestExmaple {
	@Test
	public void HamcrestTest(){
		assertEquals("hyuk joon", user.getName);

		assertThat(user.getName(), is("hyuk joon"));
	}
}
```

4.6 비교표현의 확장: Hamcrest (2/7)

✓ 숫자를 비교하는 Matcher

- greaterThan() – 앞에 값이 크겠지?
- lessThan() – 앞에 값이 작겠지?

```
@Test
public void hamcrestNumberTest() throws Exception{

      assertThat("0이 아니겠지?", 1, not(0));

      assertThat("앞에값이 크겠지?", 2000, greaterThan(1000));

      assertThat("앞에값이 크거나 같겠지?", 1000,
      greaterThanOrEqualTo(1000));

      assertThat("앞에값이 작겠지?", 2000, lessThan(5000));

      assertThat("앞에값이 작거나 같겠지?", 1000,
      lessThanOrEqualTo(1000));
}
```

4.6 비교표현의 확장: Hamcrest (3/7)

✓ 문자열을 비교하는 Matcher
- equalTo() – 앞에 문자열과 같겠지?
- equalToIgnoringCase() – 앞에 문자열과 대소문자 구별 없이 같겠지?
- equalToIgnoringWhiteSpace() – 앞에 문자열과 비교하여 공백은 무시하고 같겠지?

```
@Test
public void hamcrestStrTest() throws Exception{

      assertThat("앞뒤가 같겠지?", "하이", equalTo("하이"));

      assertThat("앞뒤가 대소문자 구분없이 같겠지?", "aabbcc",
      equalToIgnoringCase("AaBbCc"));

      assertThat("앞뒤 공백은 좀 바주고 같겠지?", "하이 ",
      equalToIgnoringWhiteSpace(" 하이"));
}
```

4.6 비교표현의 확장: Hamcrest (4/7)

✓ 비어있는 값인지를 체크하는 Matcher
- notNullValue() – 앞에 값이 널이 아니겠지?
- empty() – 앞에 값이 비어있겠지?
- emptyArray() – 앞에 배열이 비어있겠지?

```
@Test
public void hamcrestEmptyCheckTest() throws Exception{

      assertThat("널이 아니겠지?", new String("널 아니에요."),
      is(notNullValue()));

      assertThat("리스트가 비어있겠지?", new
      ArrayList<Object>(), empty());

      assertThat("배열이 비어있겠지?", new String[0],
      emptyArray());
}
```

4.6 비교표현의 확장: Hamcrest (5/7)

✓ 사용자 정의 Matcher

- notANumber() – 앞의 값이 음수 겠지?(양수면 에러)

```
@Test
public void hamcrestUserMatcherTest() throws Exception{
      // 앞의 값이 음수 겠지?(양수면 에러)
      assertThat(Math.sqrt(1), is(notANumber()));
}
```

4.6 비교표현의 확장: Hamcrest (6/7)

- ✓ TypeSafeMatcher 클래스를 상속받아 사용자 정의 Matcher를 정의 할 수 있습니다.
- ✓ describeTo – 비교하여 틀렸을 경우 메시지를 정의하는 메소드입니다.
- ✓ matchesSafely – 예상하는 값을 비교하는 메소드 입니다.
- ✓ @Factory – 단위테스트 메소드에 호출되어 사용되는 메소드 이름을 정의하는 역할을 합니다.

```
public class IsNotANumber extends
TypeSafeMatcher<Double> {
	@Override
	public void describeTo(Description description) {
		System.out.println("describeTo 호출...");
		description.appendText("not a number");
	}

	@Override
	protected boolean matchesSafely(Double number) {
		System.out.println("matchesSafely 호출...");
		return number.isNaN();
	}

	@Factory
	public static <T> Matcher<Double> notANumber(){
		System.out.println("notANumber 호출...");
		return new IsNotANumber();
	}
}
```

```
	@Test
	public void hamcrestTest() throws Exception{
		assertThat(Math.sqrt(-1), is(notANumber()));
	}
```

4.6 비교표현의 확장: Hamcrest (7/7)

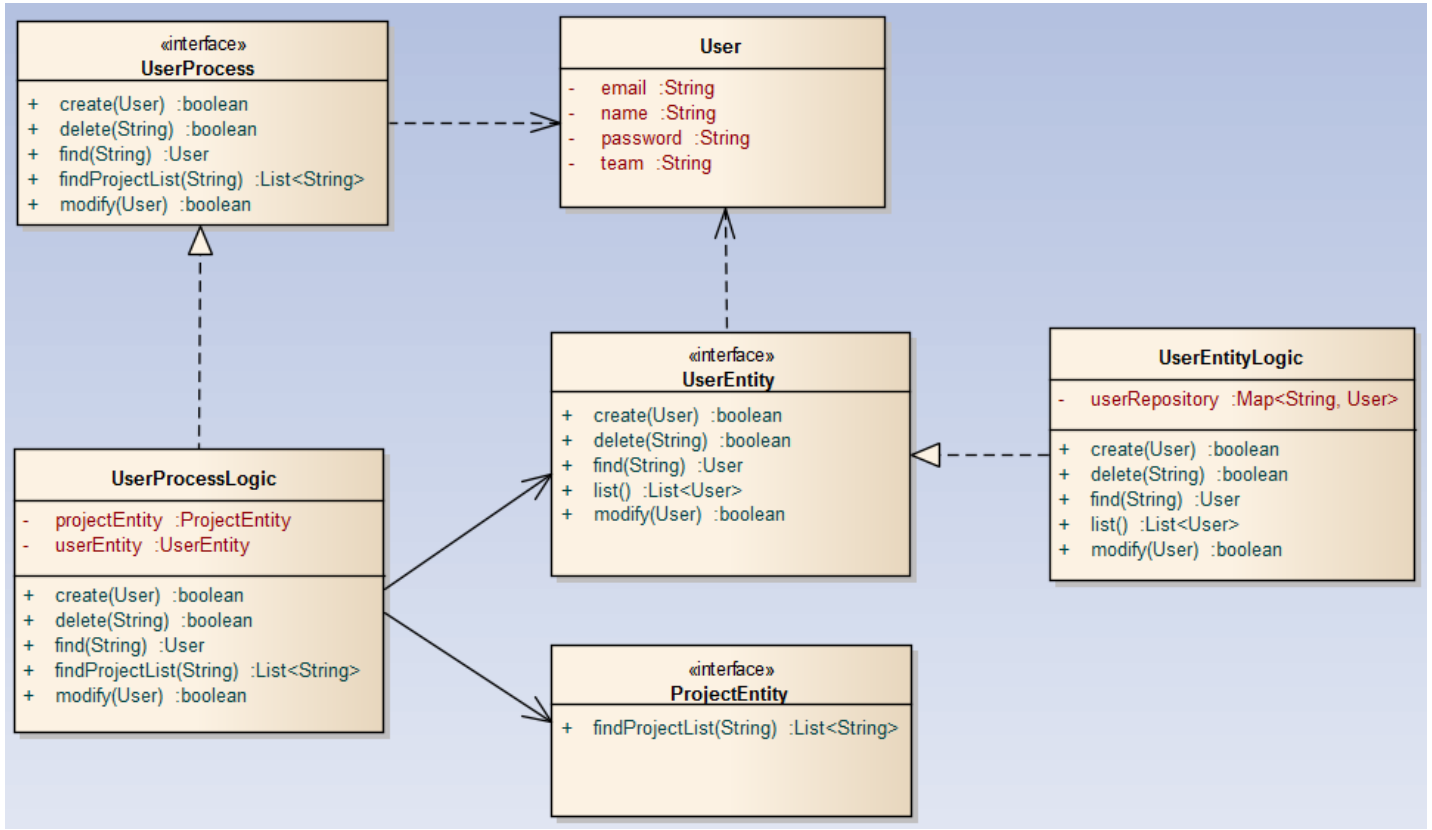
✓ Hamcrest패키지 설명

- org.hamcrest.core – 오브젝트나 값들에 대한 기본적인 Matcher들의 집합
- org.hamcrest.beans – Java 빈과 그 값 비교에 사용되는 Matcher들의 집합
- org.hamcrest.collection – 배열과 컬렉션에 사용되는 Matcher들의 집합
- org.hamcrest.number – 수 비교를 하기 위한 Matcher들의 집합
- org.hamcrest.object – 오브젝트와 클래스들을 비교하는 Matcher들의 집합
- org.hamcrest.text – 문자열 비교를 위한 Matcher들의 집합
- org.hamcrest.xml – XML문서를 비교를 위한 Matcher들의 집합

1) 테스트 주도개발 TDD 실천법과 도구 – 채수원 지음에서 참조

4.7 사용자 관리 실습 (1/7)

✓ 사용자 관리 프로젝트를 통해 조금 더 복잡한 관계의 프로젝트를 단위테스트 해봅시다.
✓ 사용자 정보를 담고 있는 User클래스와 이를 관리하는 UserEntity 인터페이스가 있습니다.
✓ UserProcessLogic에서는 UserEntity와 ProjectEntity를 가지고 있습니다.



4.7 사용자 관리 실습 (2/7)

✓ 사용자 정보를 담고 있는 User 클래스는 다음과 같습니다.

```
public class User {

	private String email;
	private String name;
	private String password;
	private String team;

	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
```

```
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getPassword() {
		return password;
	}
	public void setPassword(String password) {
		this.password = password;
	}
	public String getTeam() {
		return team;
	}
	public void setTeam(String team) {
		this.team = team;
	}
}
```

4.7 사용자 관리 실습 (3/7)

✓ 사용자 정보를 관리하는 UserEntity 인터페이스는 아래 왼쪽과 같습니다.
✓ UserEntity는 사용자의 정보를 생성, 수정, 삭제, 조회의 기능을 가지고 있습니다.
✓ 프로젝트 목록을 관리는 ProjectEntiy는 아래 오른쪽과 같고 프로젝트 목록을 조회하는 기능을 가지고 있습니다.

```
public interface UserEntity {

	public boolean create(User user);

	public User find(String email);

	public boolean modify(User user);

	public boolean delete(String email);

	public List<User> list();
}
```

```
public interface ProjectEntity {

	public List<String> findProjectList(String email);
}
```

4.7 사용자 관리 실습 (4/7)

✓ UserEntiy를 구현한 UserEntiyLogic는 다음과 같습니다.

```
public class UserEntityLogic implements UserEntity {

	private Map<String, User> userRepository;

	public UserEntityLogic() {
		userRepository = new HashMap<>();
	}

	public boolean create(User user) {
		userRepository.put(user.getEmail(), user);
		return true;
	}

	public User find(String email) {
		return userRepository.get(email);
	}
```

```
	public boolean modify(User user) {
		userRepository.put(user.getEmail(), user);
		return true;
	}

	public boolean delete(String email) {
		userRepository.remove(email);
		return true;
	}

	public List<User> list() {
		List<User> users = new
		ArrayList<>(userRepository.values());
		return users;
	}
}
```

4.7 사용자 관리 실습 (5/7)

✓ 사용자 관리 프로젝트의 로직을 담당하는 UserProcess 인퍼페이스는 다음과 같습니다.

✓ UserProcess 인터페이스에는 ProjectEntity를 사용하는 findProjectList 메소드가 존재합니다.

```
public interface UserProcess {

	public boolean create(User user);

	public User find(String email);

	public boolean modify(User user);

	public boolean delete(String email);


	/**
	 * 프로젝트 목록을 오름차순으로 정렬해서 반환한다.
	 * @return
	 */
	public List<String> findProjectList(String email);
}
```

4.7 사용자 관리 실습 (6/7)

✓ findProjectList는 projectEntity를 사용하여 프로젝트 목록을 조회하고 sort하여 리턴하는 역할을 합니다.

```
public class UserProcessLogic implements UserProcess {

	private UserEntity entity;
	private ProjectEntity projectEntity;

	public void setEntity(UserEntity entity) {
		this.entity = entity;
	}

	public void setProjectEntity(ProjectEntity
	projectEntity) {
		this.projectEntity = projectEntity;
	}

	public boolean create(User user) {
		return entity.create(user);
	}

	public User find(String email) {

	return entity.find(email);
	}
```

```
	public boolean modify(User user) {
		return entity.modify(user);
	}

	public boolean delete(String email) {
		return entity.delete(email);
	}

	public List<String> findProjectList(String email) {
		List<String> projects =
		projectEntity.findProjectList(email);

		//정렬로직 구현
		Collections.sort(projects);

		return projects;
	}
}
```

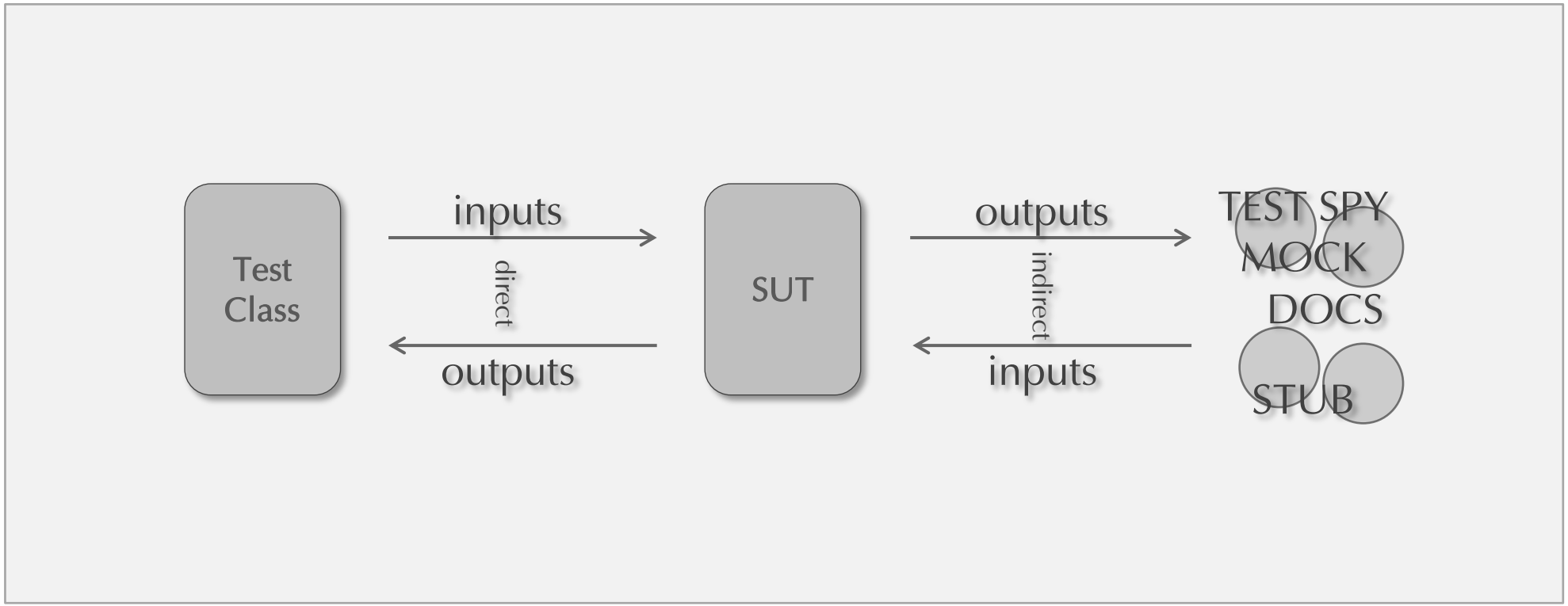
4.7 사용자 관리 실습 (7/7)

- ✓ UserEntityLogic과 UserProcessLogic 클래스에 대한 테스트 케이스 클래스를 작성해봅시다.
- ✓ UserProcessLogic 안에 있는 findProjectList 메소드 테스트 케이스를 어떻게 작성하면 좋을지 생각해 봅시다.

5. 단위테스트 상호작용과 Mockito

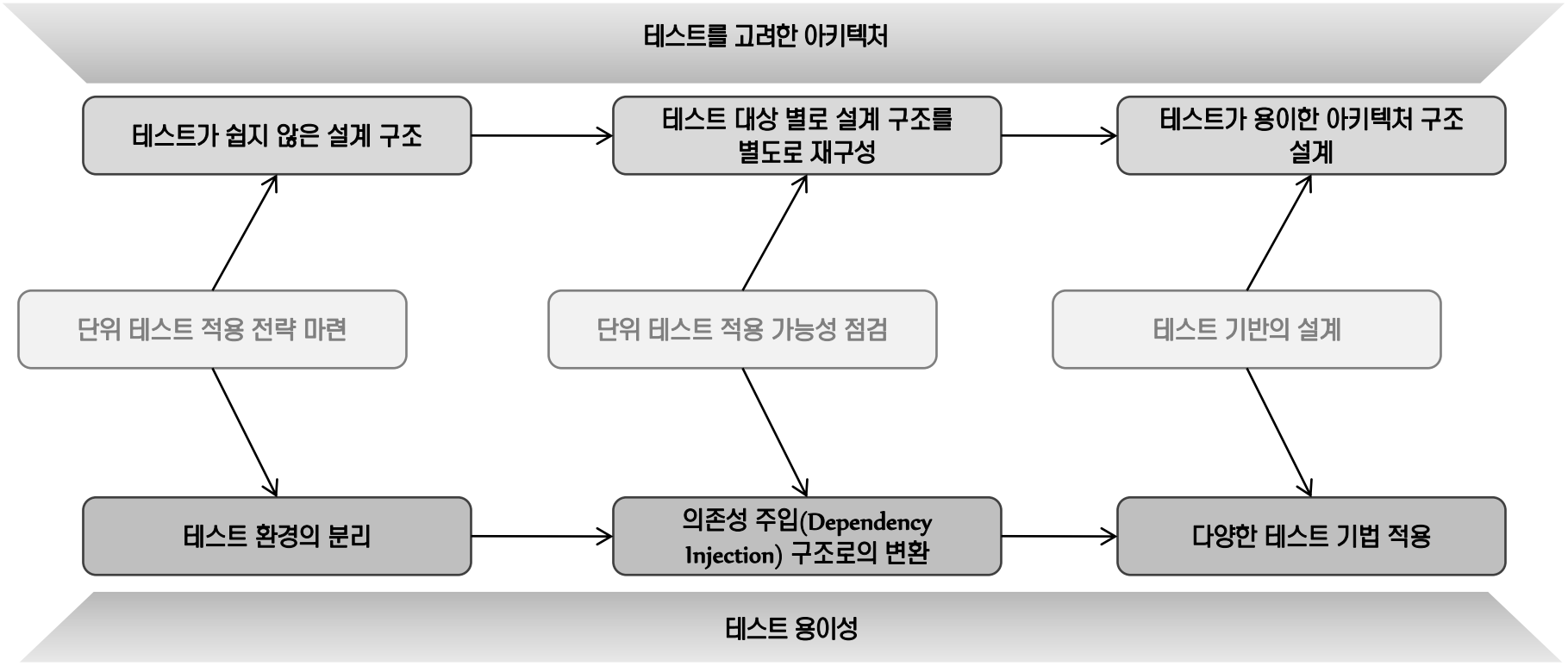
5.1 단위테스트의 상호작용

✓ SUT는 테스트 대상시스템, DOC는 테스트 대상시스템에서 의존하는 컴포넌트입니다.
✓ 단위 테스트를 진행할 때 테스트 대상 모듈이 의존하는 다른 컴포넌트에 대한 환경을 고려하여 테스트를 진행해야 합니다.
✓ 테스트 대상이 의존하는 모듈의 환경 문제로 단위 테스트 작성 시 테스트 더블로 테스트 케이스를 작성합니다.



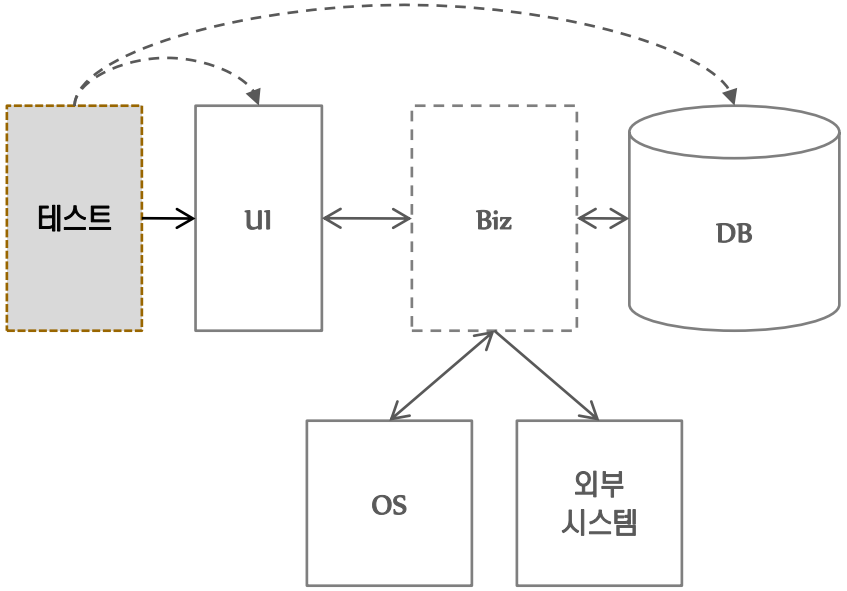
5.2 아키텍처와 테스트 (1/2)

✓ SW 아키텍처 설계시 테스트가 용이하도록 설계하여야 하며, 테스트가 어렵거나 불가능할 경우, 테스트가 가능한 형태로 재구성할(리팩토링) 필요가 있습니다.

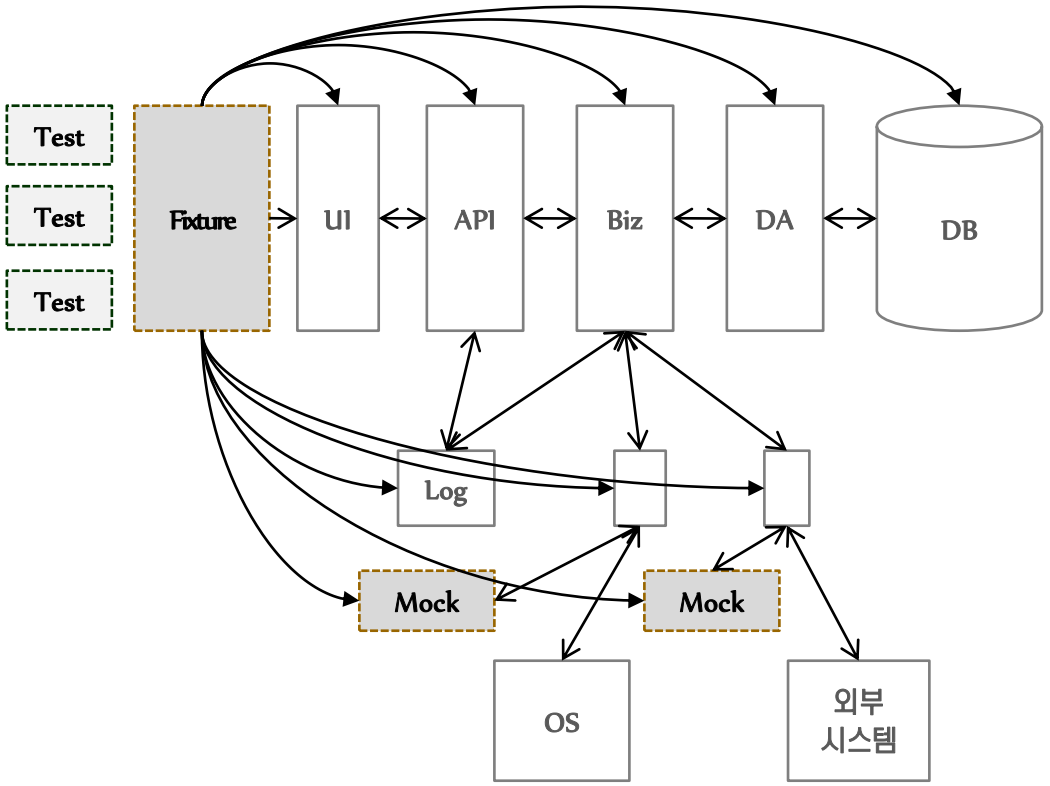


5.2 아키텍처와 테스트 (2/2)

- ✓ 테스트 가능한 아키텍처는 인터페이스를 통해 기능을 외부로 노출하고, 주입이 가능한 모듈 형태로 설계한 구조입니다.
- ✓ Biz로직을 자원접근, UI와 분리한 형태로 설계해야 합니다.
- ✓ Biz로직 계층을 잘못 구성하면 UI와 DB테스트만 수행하는 경우가 될 수도 있습니다.



일반적인 3-tier 아키텍처에서 테스트는 Biz 로직의 구성에 따라 테스트 범위를 결정함.



5.3 의존성 주입 (1/3)

✓ 의존성 주입은 방식에 따라 Parameter, Constructor, Setter 주입으로 나뉩니다.
✓ 컴포넌트가 주입 받은 객체에 대한 참조를 객체 내에 유지합니다.
✓ 의존성 주입 프레임워크를 사용하면 Field 주입을 할 수도 있습니다.

종류	설명
Parameter Injection	• 파라미터로 의존 컴포넌트에 주입하는 방식 • 컴포넌트가 의존 컴포넌트에 대한 참조를 유지하지 않을 경우 사용 • 클라이언트 측에서 컴포넌트가 어떤 의존 컴포넌트를 사용하는지 알아야 함
Constructor Injection	• 컴포넌트의 생성자를 이용하여 의존 컴포넌트를 주입하는 방식 • 객체가 생성될 때 의존 컴포넌트를 이용하여 초기화 작업을 수행해야 하는 경우 사용 • 컴포넌트가 의존 컴포넌트에 대한 참조를 객체 내에 유지함
Setter Injection	• 컴포넌트의 Setter 메소드를 이용하여 의존 컴포넌트를 주입하는 방식 • 컴포넌트가 의존 컴포넌트에 대한 참조를 객체 내에 유지함

5.3 의존성 주입 (2/3) – Parameter Injection

✓ 의존관계를 갖는 객체를 파라미터로 전달합니다.
✓ 테스트 시점에는 의존관계를 갖는 컴포넌트를 테스트용으로 구현해서 로직을 검증합니다.
✓ 비즈니스 로직은 전달받은 파라미터가 테스트용도인지 정상 구현된 객체인지 모른 상태로 의존관계 모듈을 호출합니다.

```java
/**
 * 날짜값 처리 유틸리티
 */
public class DateHandler {

    /**
     * 현재시간을 일정형태(yyyy-MM-dd)로 조회
     *
     * @param provider
     * @return
     */
    public static String getCurrentTime(TimeProvider timeProvider)
            throws Exception {
        Calendar c = null;
        c = timeProvider.getTime();

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

        return sdf.format(c.getTime());
    }
}
```

```java
public interface TimeProvider {

    /**
     * 특정시간을 제공
     * @return 특정 시간
     * @throws Exception
     */
    Calendar getTime() throws Exception;

}
```

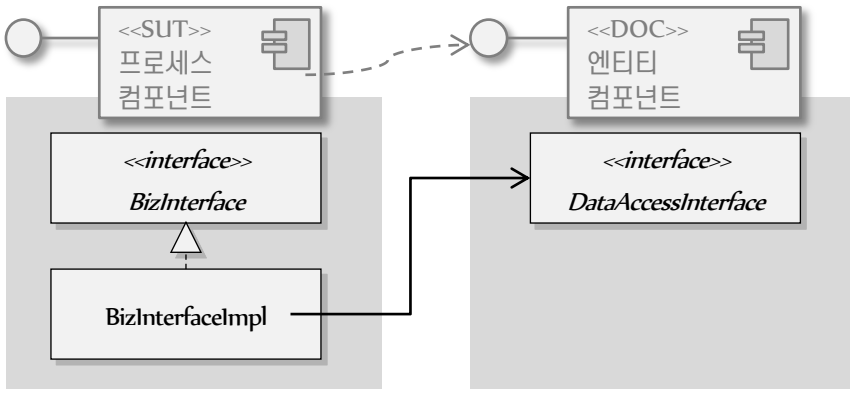
```java
public class DateHandlerTest {

    @Test
    public void testGetCurrentTime() throws Exception {
        String currentTime = DateHandler.getCurrentTime(new TimeProvider(){
            public Calendar getTime() throws Exception {
                Calendar c = Calendar.getInstance();
                c.set(Calendar.YEAR, 2014);
                c.set(Calendar.MONTH, Calendar.JANUARY);
                c.set(Calendar.DAY_OF_MONTH, 1);
                return c;
            }
        });
        assertEquals("2014-01-01", currentTime);
    }

}
```

5.3 의존성 주입 (3/3) – Setter Injection

✓ 의존관계를 갖는 객체를 setter 메소드로 전달합니다.

✓ SUT는 DOC(의존관계를 갖는 컴포넌트)에 대한 참조변수를 객체 내에 유지하고 있어야 합니다.

✓ DOC는 간단한 형태로 직접 구현하거나 테스트 라이브러리를 사용해서 구현합니다.




```java
@Test
public void testSomeMethod_withDependencyInjection() throws Exception {
    ((BizInterfaceImpl)this.bizInterface).setDaInterface(new DataAccessInterface() {
        @Override
        public Object findDataByKey(String key) {
            if ("123".equals(key)) return new Object();
            return null;
        }
    });

    bizInterface.someMethod("123");
    try {
        bizInterface.someMethod("1");
        fail("비즈니스 로직이 예상되는 행위를 수행하지 못했습니다.");
    } catch (RuntimeException e) {
        assertEquals("조회한 정보가 없습니다.", e.getMessage());
    }
}
```

의존관계 인터페이스에 대한 외부(테스트) 주입

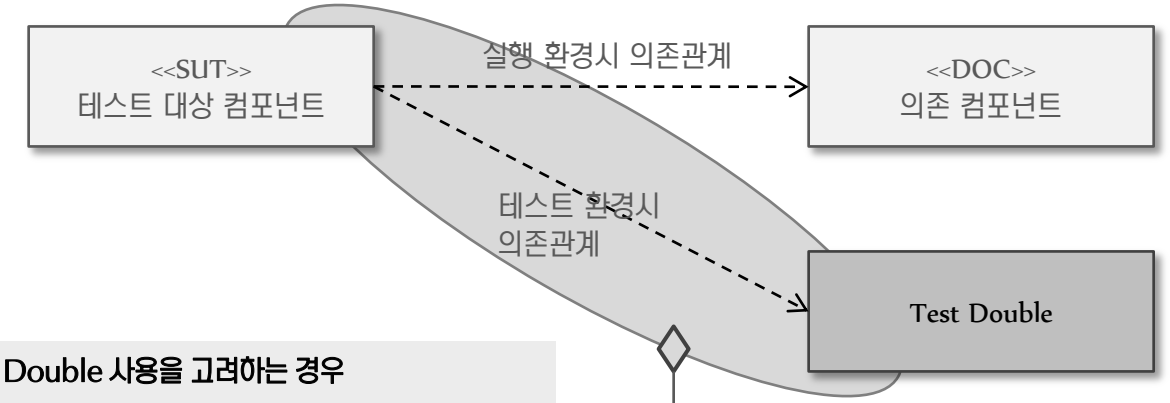
```java
public class BizInterfaceImpl implements BizInterface {

    /** 데이터 접근 인터페이스 */
    private DataAccessInterface daInterface;

    /**
     * 데이터 접근 인터페이스 setter
     * @param daInterface 데이터 접근 인터페이스
     */
    public void setDaInterface(DataAccessInterface daInterface) {
        this.daInterface = daInterface;
    }

    /**
     * 비즈니스 오퍼레이션 구현
     *
     * @see com.meritzfis.std.guide.unittest.advanced.BizInterface#someMethod(java.lang.String)
     */
    @Override
    public void someMethod(String param) {
        Object retrievedValue = this.daInterface.findDataByKey(param);
        if (retrievedValue == null)
            throw new RuntimeException("조회한 정보가 없습니다.");

        // Some Logics
    }

}
```

의존관계 인터페이스 선언 및 setter

의존관계 인터페이스 호출

5.4 Test Double (1/3)

✓ 테스트 대상 컴포넌트(SUT)가 의존하는 컴포넌트(DOC)를 테스트 환경에 맞는 컴포넌트로 대체하는 것을 말합니다.
✓ 테스트 더블은 테스트 환경 내에서 의존 컴포넌트와 동일하게 동작하도록 구성합니다.
✓ 테스트 시점에는 테스트 더블을 생성해서 SUT에 주입합니다.



Test Double 사용을 고려하는 경우

- 의존 컴포넌트가 대상이 동적으로 결정되는 경우
- 설정(Setup)하기 어려운 경우
- 속도가 느릴 경우
- 쓰레드나 프로세스로 동작할 경우
- 아직 개발되지 않았을 경우

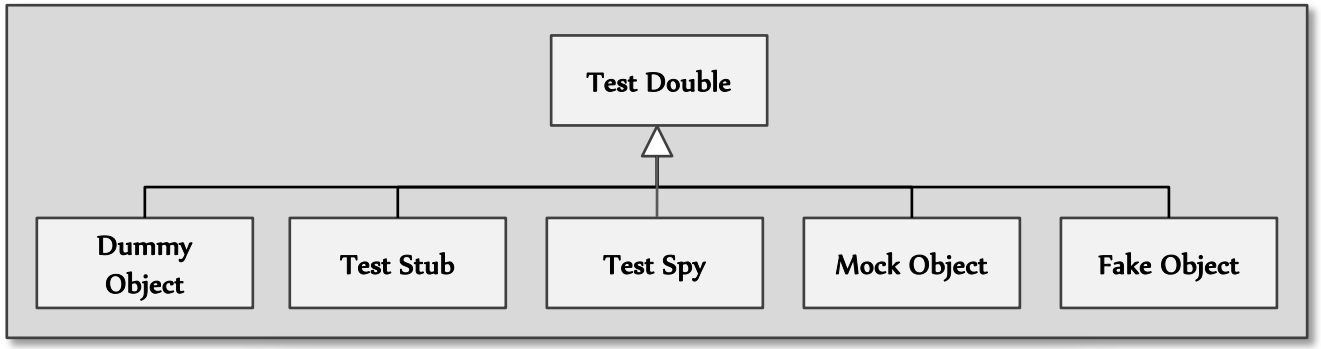
Test Fixture

테스팅시에 테스트를 실행하기 위해 기준으로 사용되는 고정된 상태를 말함. Test Fixture를 사용하는 목적은 테스트가 실행되고 그 결과가 반복 가능하게 하는 잘 알려지고 고정된 환경이 존재함을 보장하는 것임. Test Context라고도 함.

✓ - 102 -

5.4 Test Double (2/3)

✓ 테스트 더블의 유형으로는 Dummy, Stub, Spy, Mock, Fake가 있습니다.
✓ DOC의 실제 로직과 얼마나 비슷하게 구현하느냐에 따라 아래와 같이 구분합니다.
✓ 하지만, 그래프도 서로 영역이 겹쳐서 표현되듯이 구현 정도에 따라 Stub 또는 Spy와 같이 딱 잘라서 구분하지 않습니다.



1) 참조 : xUnit Test Patterns
2) 출처 : http://msdn.microsoft.com/en-us/magazine/cc163358.aspx

5.4 Test Double (3/3)

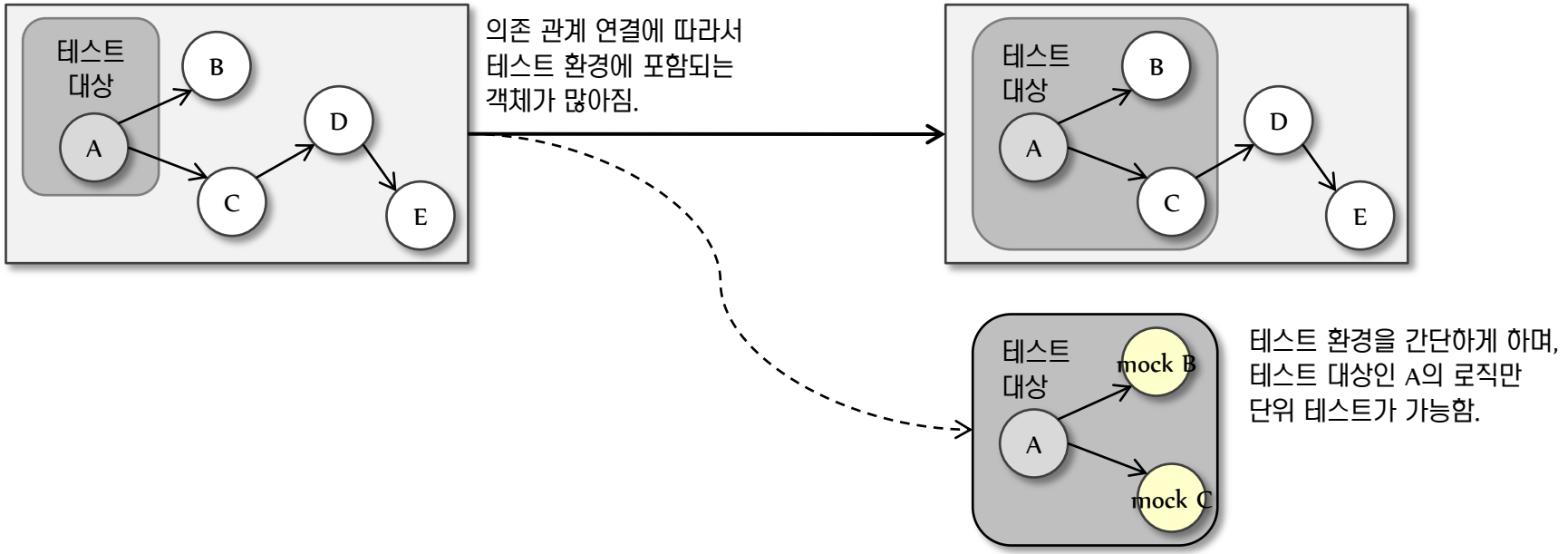
✓ 테스트 더블의 종류별 특징입니다.

✓ 비즈니스 로직과 비슷하게 구현할수록 테스트 더블을 생성하는 비용이 증가하게 됩니다.

종류	설명
Dummy Object	• 객체 전달에만 사용되는 가짜 객체 • 테스트 환경에서 실제로 사용하지 않음 • 단순히 매개 변수 목록을 만족시킴
Test Stub	• 테스트를 위해 미리 준비한 응답을 제공할 수 있도록 해주는 객체 • 테스트가 간접적인 입력을 제어할 수 있도록 해줌 • Responder, Saboteur로 구분
Test Spy	• Test Stub에 좀 더 많은 기능을 부여한 형태 • 테스트 환경 내부의 행위를 Stub내에 기록하고 검증하는 메커니즘을 가짐
Mock Object	• 테스트 환경과 의존 컴포넌트에서 발생하는 다양한 입/출력 행위에 대한 검증 • 간접적인 출력에 대한 검증 • 간접적인 입력에 대한 제어 • Test Stub이나 Test Spy 대신 사용 가능
Fake Object	• 실제 객체와 동일한 기능을 수행 • 빠른 구현을 위해 실제 환경과는 조금 다르게 구현 • RDBMS 대신 In Memory DB의 사용

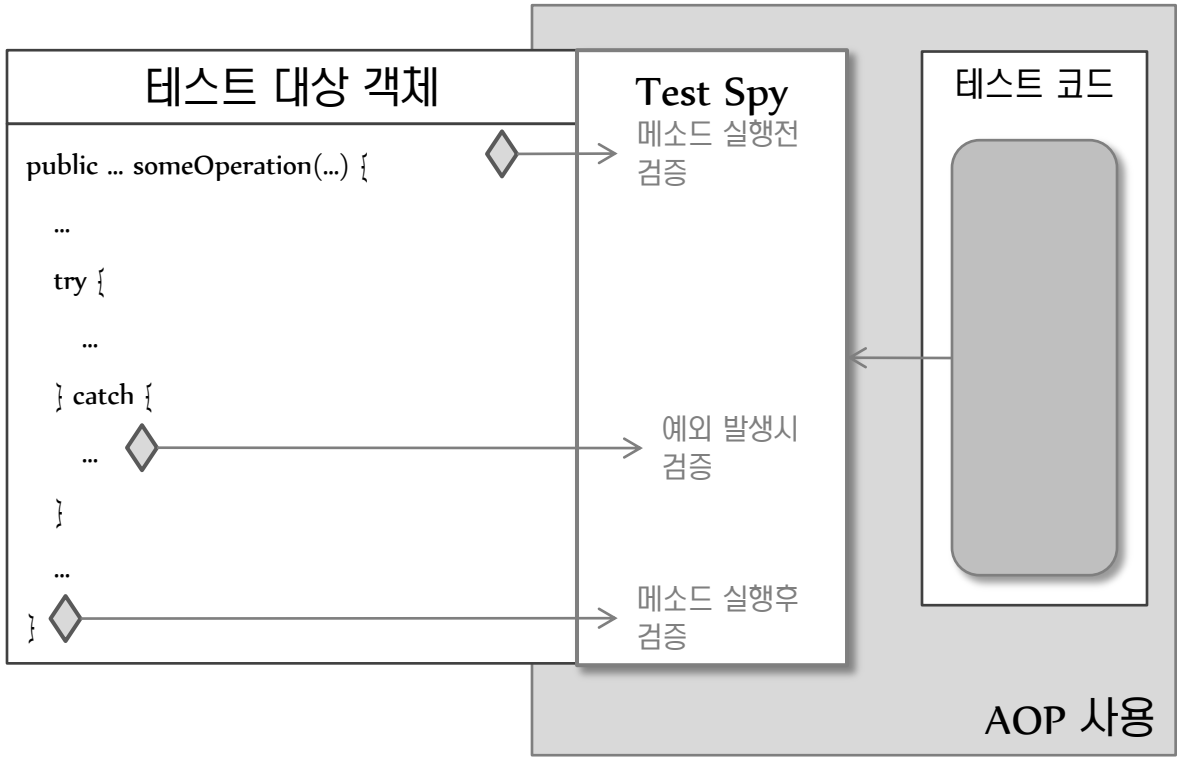
5.5 Mock Object

- ✓ Mock 객체를 사용해서 해당 오퍼레이션에 대한 입출력을 간접적으로 검증함으로써, 테스트 대상의 내부 행위를 검증합니다
- ✓ Mock 객체를 사용하지 않을 경우 단위 테스트에서 의존관계를 맺는 컴포넌트를 생성하여야 하며, 이는 또 다른 테스트 환경이 마련되어야 함을 의미합니다.
- ✓ 즉, 의존 관계가 깊어질수록 많은 테스트 환경이 필요한데, Mock 객체는 이러한 수고를 덜어줍니다.



5.6 Test Spy

- ✓ 테스트 환경에서 의존 컴포넌트(DOC)로 입력한 내용을 스텁에 저장하고, 테스트 환경에서 실행된 후에 테스트 환경의 행위가 올바른지 검증하는데 사용합니다.
- ✓ SUT가 컨테이너에서 수행될 경우, 예외 발생시 컨테이너가 자체적으로 처리하게 됩니다. 이 경우 Test Spy를 사용하면 테스트 실패에 대한 원인이 테스트 로직으로 전달되므로 원인파악이 용이합니다.



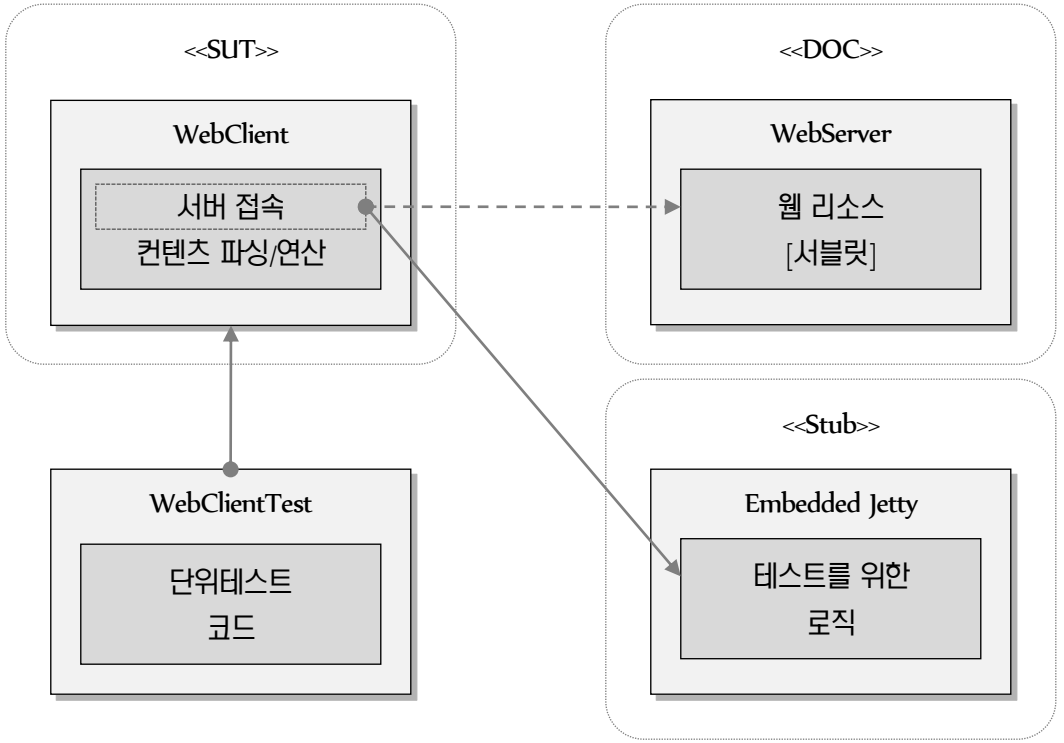
cf. AOP : Aspected-Oriented Programming

[테스트 스파이 사용시기]

- Test Stub의 기능을 확장하여 테스트 환경의 행위(Indirect Output)를 검증
- 기존 테스트 방법으로 검증할 수 없는 요구사항이 있을 때 사용
- 테스트 케이스 안에서 명백한 검증을 실행하고 싶은 경우
  - Mock일 경우, 검증 작업이 Mock 객체 내부에서 이루어짐.
- 테스트 의존적인 Assertion방법이 필요한 경우
  - Mock을 사용할 수 없음, 대체로 Mock은 일반적인 단정문만을 제공
- 테스트 실패가 테스트로 명확히 전달되지 않을 경우
  - 컨테이너 안에서 SUT가 실행되며, 컨테이너가 자체적으로 예외를 처리
- SUT가 모두 실행된 후 검증을 하기 원할 경우
  - 일부 Mock의 경우, 지정된 순서대로 SUT가 실행되지 않을 경우, 바로 테스트가 실패함

5.7 Test Stub

✓ 외부시스템이 항상 연결 준비가 되어있지 않고, 테스트 환경도 마련되어 있지 않는 경우 사용합니다.
✓ 결제 시스템과 같은 중요한 트랜잭션의 경우, 한 번 테스트한 후 이전 상태로의 복원이 어렵기 때문에 사용합니다.
✓ 테스트 후 변경된 사항을 반영하는 절차나 시간이 오래 걸리는 경우 사용합니다.



5.8 Mockito 개요

- ✓ 자바 단위테스트에서 가짜 객체를 지원해주는 프레임워크 입니다.
- ✓ Mockito는 Szczepan Faber and friends에 의해 서비스가 제공됩니다.
- ✓ Mock 객체 생성, Mock 객체 동작을 지정, 그리고 테스트 대상 로직이 제대로 수행 되었는지 확인합니다.

```java
@Test
public void mockitoVerifyTest() {
      // mock creation
      List list = mock(List.class);

      // using mock object - it does not throw any
      "unexpected interaction"
      // exception
      list.add("one");
      list.clear();

      // selective, explicit, highly readable
      verification
      verify(list).add("one");
      verify(list).clear();
}
```

```java
@Test
public void mockitoStubTest() {
      // you can mock concrete classes, not only
      interfaces
      List list = mock(List.class);

      // stubbing appears before the actual execution
      when(list.get(0)).thenReturn("first");

      // the following prints "first"
      System.out.println(list.get(0));

      // the following prints "null" because get(999) was
      not stubbed
      System.out.println(list.get(999));
}
```

5.9 인터페이스를 구현한 mock객체 지정

- ✓ Mock 프레임워크를 사용하지 않고 테스트 케이스에 사용 될 mock 객체를 임의로 implements하여 작성합니다.
- ✓ 하지만 매번 mock객체를 위해 임의로 implements를 하는 것은 시간과 불필요한 코드가 작성되기 때문에 권장하는 방법이 아닙니다.

```
public class ProjectEntityImpl implements ProjectEntity{

	public List<String> findProjectList(String email) {
		List<String> result = new ArrayList<String>();

		result.add("정보원 프로젝트");
		result.add("직능원 프로젝트");
		result.add("인천공항 프로젝트");

	return result;
	}
}
```

```
@Test
public void findProjectListByImpl() {

	ProjectEntity mock = new ProjectEntityImpl();
	userProcess.setProjectEntity(mock);

	List<String> projectList =
	userProcess.findProjectList("hjkwon");

	assertNotNull(projectList);
	assertEquals(3, projectList.size());
}
```

6. Mockito 기능

6.1 mock 객체 생성

✓ mock 메소드를 사용하여 mock객체를 생성합니다.

✓ mock객체를 Constructor, Setter Parameter Injection 중 한가지 방식을 통해 주입하면 됩니다.

```
@Test
public void findProjectListByMock() {

      //1. 가상객체(목객체)를 생성
      ProjectEntity mock = mock(ProjectEntity.class);

      userProcess.setProjectEntity(mock);
}
```

가짜 객체 생성

가짜 객체 주입

6.2 stub 만들기 (1/3)

✓ When 메소드를 통해 하나의 메소드가 호출되었을 때 임의 값을 반환 하라고 설정합니다.

✓ 메소드의 파라미터 값까지 지정하고, 호출 시점에서 지정한 파라미터까지 동일해야 지정한 임의 값을 반환합니다.

```
@Test
public void findProjectListByMock() {
      //1. 가상객체(목객체)를 생성
      ProjectEntity mock = mock(ProjectEntity.class);
      userProcess.setProjectEntity(mock);

      //2. 결과값(예상값)을 녹화
      List<String> projects = new ArrayList<>();
      projects.add("정보원 프로젝트");
      projects.add("직능원 프로젝트");
      projects.add("인천공항 프로젝트");

      when(mock.findProjectList("hjkwon")).thenReturn(proj
      ects);

      List<String> projectList =
      userProcess.findProjectList("hjkwon");

      assertNotNull(projectList);
      assertEquals(3, projectList.size());
}
```

가짜 객체에서 반환 할 값 생성

가짜 객체에서 frndProjectList메소드의 파라미터로 'hjkwon'이 호출되면 위에저 지정한 값을 반환 하라고 설정

Stub으로 지정 한 값 리턴 받음

6.2 stub 만들기 (2/3)

✓ Argument Matcher를 사용한 인자매칭을통해 임의 값을 통해 stub을 지정할 수 있습니다.

✓ anyInt(), antString(), anyDouble(), anyLong(), anyList(), anyMap()등의 메소드가 있습니다.

```
@Test
public void findProjectListByMock() {
      //1. 가상객체(목객체)를 생성
      ProjectEntity mock = mock(ProjectEntity.class);
      userProcess.setProjectEntity(mock);

      //2. 결과값(예상값)을 녹화
      List<String> projects = new ArrayList<>();
      projects.add("정보원 프로젝트");
      projects.add("직능원 프로젝트");
      projects.add("인천공항 프로젝트");

      when(mock.findProjectList(anyString())).thenReturn(p
      rojects);

      List<String> projectList =
      userProcess.findProjectList("hjkwon");

      assertNotNull(projectList);
      assertEquals(3, projectList.size());
}
```

가짜 객체에서 반환 할 값 생성

가짜 객체에서 frndProjectList메소드의 파라미터를 임의의 값으로 호출 되어도 반환 값을 같도록 설정

Stub으로 지정 한 값 리턴 받음

6.2 stub 만들기 (3/3)

✓ thenThrow 메소드를 통해 mock 객체의 특정 메소드 호출 시 예외를 발생 시킬 수 있습니다.

```
@Test(expected=RuntimeException.class)
public void findProjectListByMock() {
      //1. 가상객체(목객체)를 생성
      ProjectEntity mock = mock(ProjectEntity.class);
      userProcess.setProjectEntity(mock);

      //2. 결과값(예상값)을 녹화
      List<String> projects = new ArrayList<>();
      projects.add("정보원 프로젝트");
      projects.add("직능원 프로젝트");
      projects.add("인천공항 프로젝트");

      when(mock.findProjectList(anyString())).thenThrow(ne
      w RuntimeException("invalid param"));

      List<String> projectList =
      userProcess.findProjectList("hjkwon");

      assertNotNull(projectList);
      assertEquals(3, projectList.size());
}
```

가짜 객체에서 반환 할 값 생성

가짜 객체에서 frndProjectList메소드가
호출되면 예외가 발생되도록 설정

6.3 mock 객체 검증

✓ verify 메소드를 통해 실제 로직 상에서 mock 객체의 메소드가 호출되었는지 확인하고 검증하는 기능을 할 수 있습니다.

```
@Test
public void findProjectListByMock() {
      //1. 가상객체(목객체)를 생성
      ProjectEntity mock = mock(ProjectEntity.class);
      userProcess.setProjectEntity(mock);

      //2. 결과값(예상값)을 녹화
      List<String> projects = new ArrayList<>();
      projects.add("정보원 프로젝트");
      projects.add("직능원 프로젝트");
      projects.add("인천공항 프로젝트");

      when(mock.findProjectList(anyString())).thenReturn(p
      rojects);

      List<String> projectList =
      userProcess.findProjectList("hjkwon");

      assertNotNull(projectList);
      assertEquals(3, projectList.size());

      verify(mock, times(1)).findProjectList(anyString());
}
```

가짜 객체에서 반환 할 값 생성

가짜 객체에서 frndProjectList메소드의 파라미터를 임의의 값으로 호출 되어도 반환 값을 같도록 설정

지정한 메소드가 호출되었는지 검증

6.4 메소드 종류

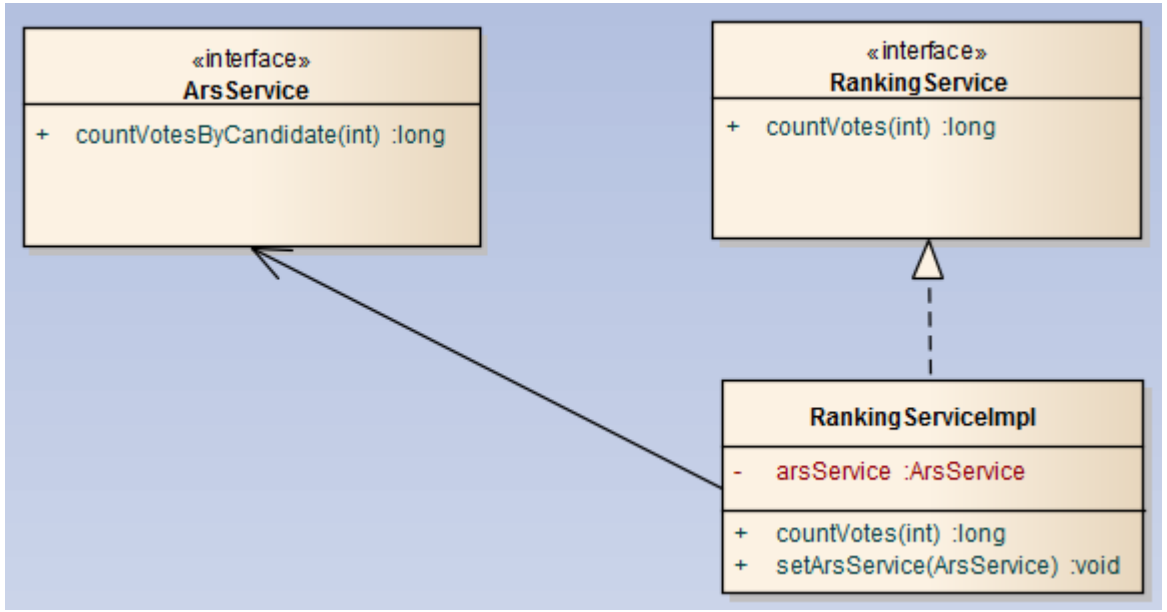
✓ Mockito 메소드 종류는 다음과 같습니다.

- Mock() – 모의 객체를 생성하는 역할
- when() – 협력객체 메소드 반환 값을 지정해주는 역할(stub)
- verify() – SUT안의 협력객체 메소드가 호출 되었는지 확인
- times() – 지정한 횟수 만큼 협력 객체 메소드가 호출 되었는지 확인
- nerver() – 호출되지 않았는지 여부 검증
- atLeastOnce() – 최소 한 번은 특정 메소드가 호출되었는지 확인
- atLeast() – 최소 지정한 횟수 만큼 호출되었는지 확인
- atMost() – 최대 지정한 횟수 만큼 호출되었는지 확인
- clear() – 스텁을 초기화 한다
- timeOut() – 지정된 시간 안에 호출되었는지 확인

6.5 kpopRanking 실습 (1/3)

✓ kpopRanking 프로젝트를 통해 mockito에 대해 조금 더 습득해봅시다.

✓ ArsService 인터페이스는 다른 외부 시스템이라고 간주합니다.



6.5 kpopRanking 실습 (2/3)

✓ 외부 시스템인 ArsService의 인터페이스만 알고 있습니다.

✓ RankingServiceImpl에서 ArsService서비스를 사용하고 있습니다.

```
public interface ArsService {
    //
    public long countVotesByCandidate(int number);
}
```

```
public interface RankingService {
      public long countVotes(int number);
}

public class RankingServiceImpl implements
RankingService {
      private ArsService arsService;

      public void setArsService(ArsService arsService) {
            this.arsService = arsService;
      }

      public long countVotes(int number) {
            long count = 0; //심사위원 + 방청객
            count +=
            arsService.countVotesByCandidate(number);

            return count;
      }
}
```

6.5 kpopRanking 실습 (3/3)

- ✓ RankingServiceImpl의 인터페이스만 알고 있는 countVotes 메소드를 mockito를 사용해서 테스트 해봅시다.
- ✓ 테스트 시 먼저 countVotesByCandidate가 정상적으로 호출되었는지 verify메소드를 통해 검증합니다.
- ✓ 두번째로 countVotestByCandidate메소드가 파라미터 케이스로 stub을 정의하고 verify 메소드를 통해 검증합니다.

7. 테스트 커버리지

7.1 검사와 Test의 차이

- ✓ Test는 동적인 활동으로서, 소프트웨어를 직접 실행시켜 그 기능성을 확인합니다.
- ✓ 검사는 미리 정의한 규칙 집합에 따라 코드를 분석하는 활동입니다.
- ✓ 코딩 문법 표준, 코드 중복 검사, Test coverage 등이 있습니다.
- ✓ Test와 검사는 둘 다 소프트웨어를 변경하지는 않습니다.
- ✓ 검사와 Test가 보고한 문제에 대한 명확한 조치가 필요합니다.

7.2 Test Coverage

✓ 테스트 커버러지

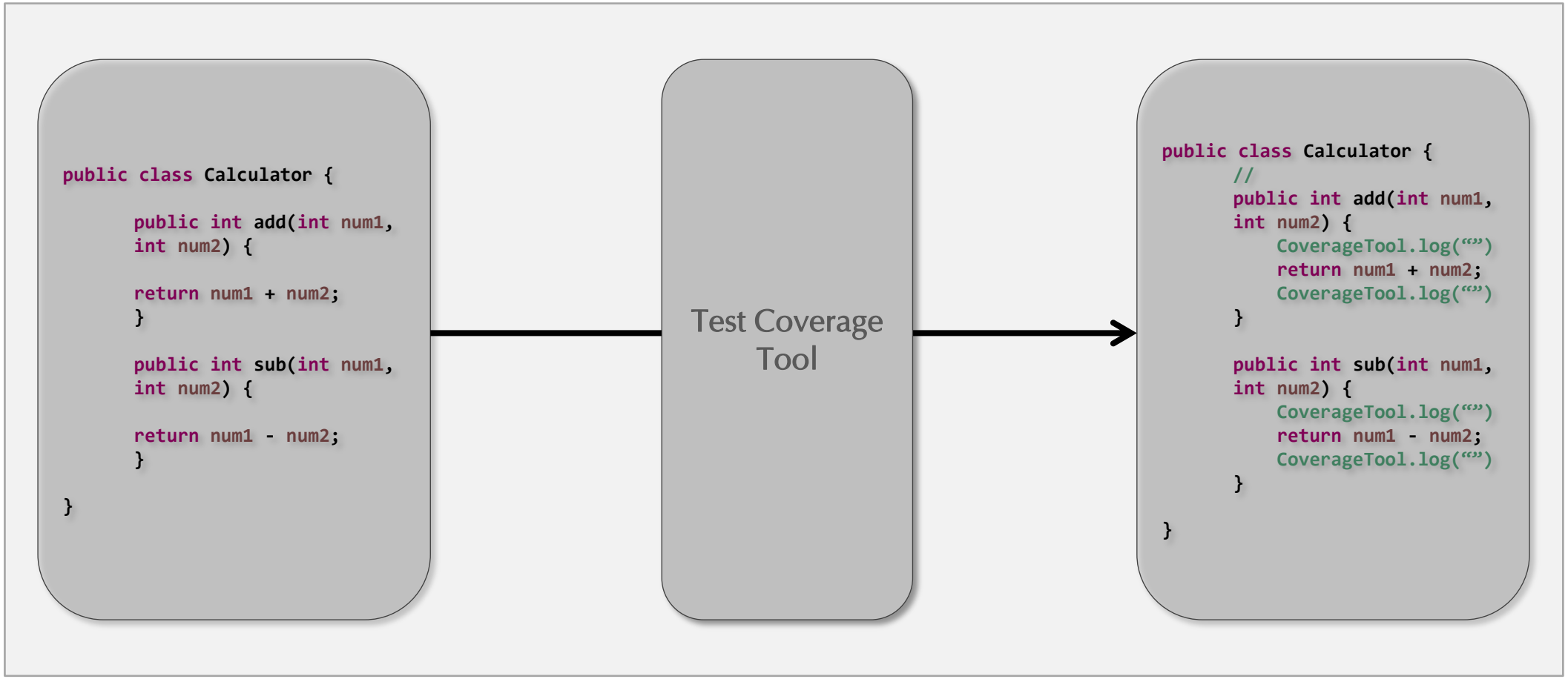
- 단위테스트 또는 통합테스트 수행 시 테스트코드가 테스트 대상 시스템에 대하여 얼마나 점검되었는지 판단해야 합니다.
- 테스트코드가 테스트 대상의 로직들에 대하여 커버하는지 테스트 커버러지 비율로서 확인 할 수 있습니다.

✓ 테스트 커버러지 척도

- 테스트 커버러지 분석중 가장 논란이 많은 부분이 테스트 코드의 최종 커버러지 비율입니다.
- 커버러지 비율을 "85%이상 유지해야 한다" 또는 "70% 이상 유지해야 한다"라는 기준을 세워 놓고 개발팀에 이 기준을 맞추도록 요구하는 것은 테스트 커버러지를 잘못 이해해서 발생하는 상황으로 볼 수 있습니다.
- 단순한 수치적 비율을 놓고 이야기하게 되면 고객은 테스트 대상 시스템이 충분히 테스트 되지 않았다고 오해를 할 수 있습니다.
- 테스트 커버러지에 대한 정책을 결정할 때는 중요한 모듈을 선정하고 ValueObject 또는 Utility 와 같은 클래스를 제외한 상태에서 비율을 정의해야 합니다.
- 테스트 커버러지를 도입 시에는 업무상 중요도를 판단하여 테스트 대상 컴포넌트의 범위와 목표 커버리지를 정의하고 이해당사자들의 동의를 받는 것이 중요합니다.

7.3 Test Coverage 툴의 원리 (1/2)

✓ Code Coverage 툴의 주요 기능은 실행 중에 대한 code 라인이 수행되었는지를 검증하는 것입니다.
✓ 이를 위해 Coverage 툴은 class의 각 실행 라인에 Code Coverage 툴로 log-in하는 logic을 추가합니다.
✓ 기존의 class에 Coverage 분석을 위한 code를 추가하는 작업을 Instrument라고 합니다.



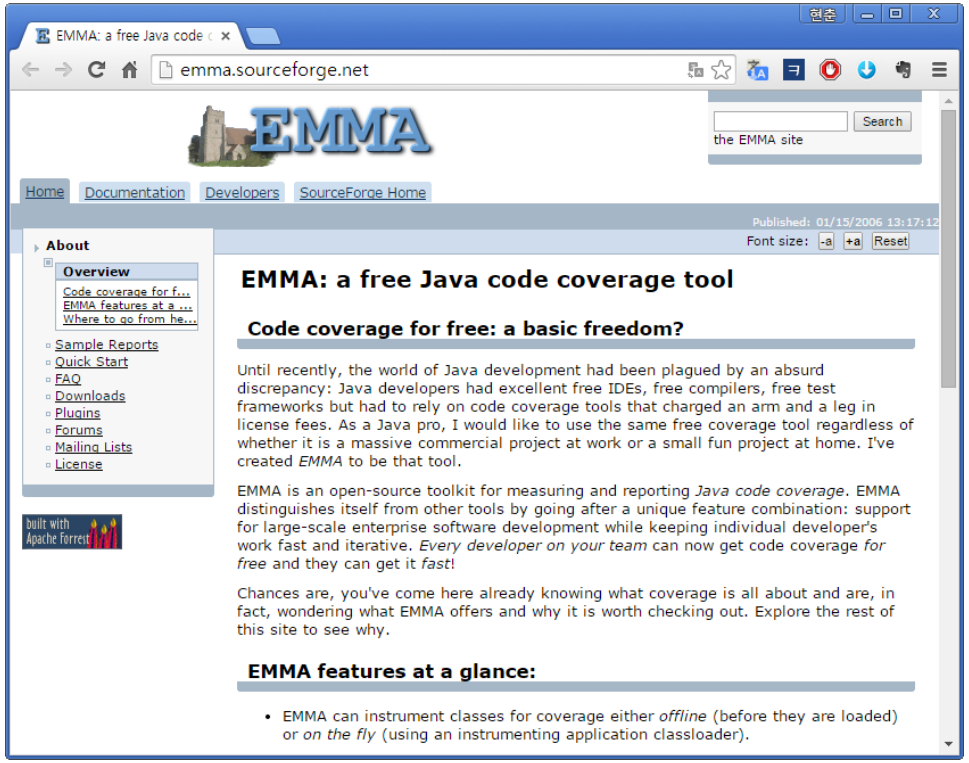
7.3 Test Coverage 툴의 원리 (2/2)

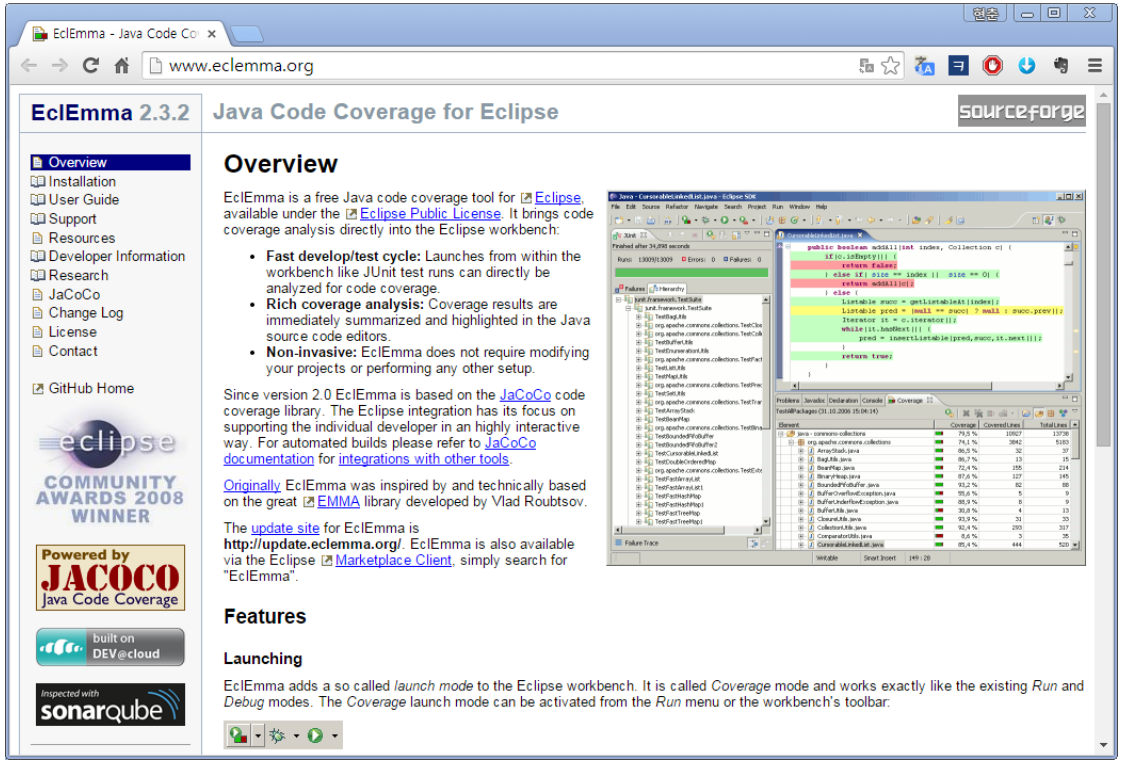
✓ Instrument

- 정적 기법 : 어플리케이션이 수행되기 이전에 source code나 compile이 완료되어 있는 class 파일을Instrument하여 Instrumented class들을 만든 후 그것을 수행하는 방식입니다.
- 동적 기법 : 원본 Class를 가지고 어플리케이션을 수행하여 runtime시에 class가 loading되는 순간에 class에 Instrumentation을 하는 방식입니다.
- 동적 기법은 runtime에서 Code Instrumentation을 하는 부하가 발생하기 때문에, Instrumentation 양이 압도적으로 많은 Code Coverage 툴의 경우에는 정적 Instrumentation방식이 좀 더 유리합니다.
- EMMA는 동적, 정적 Instrumentation을 모두 지원합니다.
- 유사한 Coverage Tool인 Cobertura는 정적 Instrumentation만 지원합니다.

7.4 Emma (1/6)

- ✓ Emma 는 Java Code Coverage Tool로서 Open Source 입니다.
- ✓ EclEmma는 Emma 기반의 이클립스 Plug-in 프로그램입니다.
- ✓ 코드 커버러지 분석결과는 자바 소스 편집기에서 확인가능 합니다.
- ✓ 테스트 케이스를 통해 소스의 각 라인 별로 호출여부를 확인할 수 있으며 점검결과 통계를 조회할 수 있습니다.

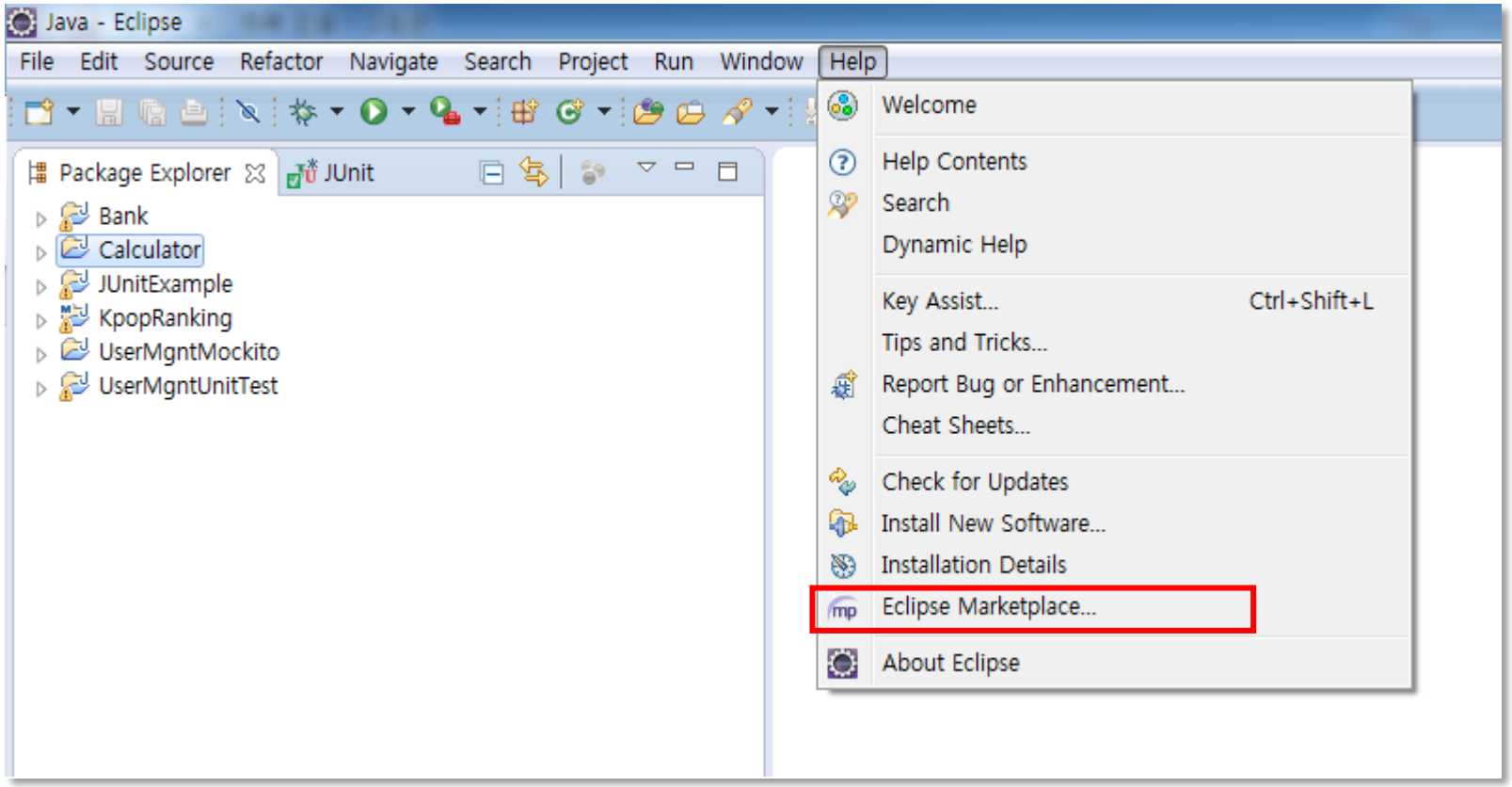




7.4 Emma (2/6)

✓ Code Coverage를 확인하기 위하여 이클립스 IDE에 EclEmma Plug-in을 설치합니다.

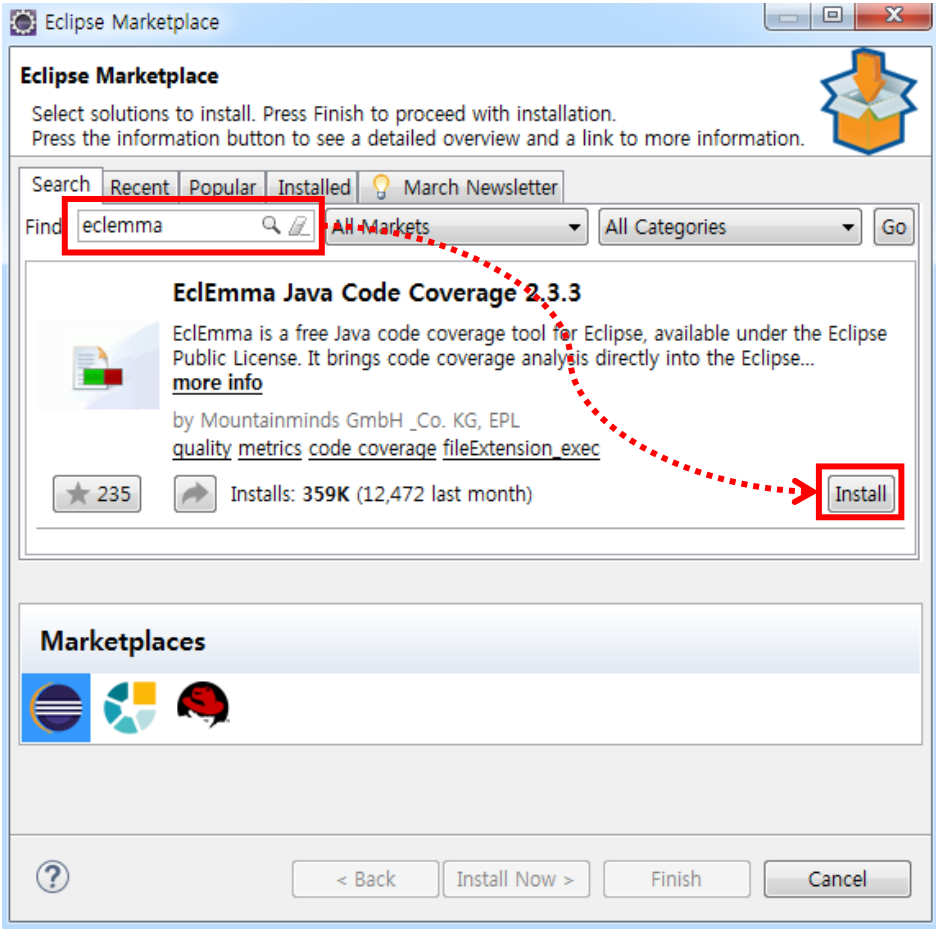
✓ Plug-in을 설치하기 위하여 Eclipse Marketplace 에 접속합니다.



7.4 Emma (3/6)

✓ Eclipse Marketplace 의 Find 창에서 eclemma 를 입력 후 검색을 실행합니다.

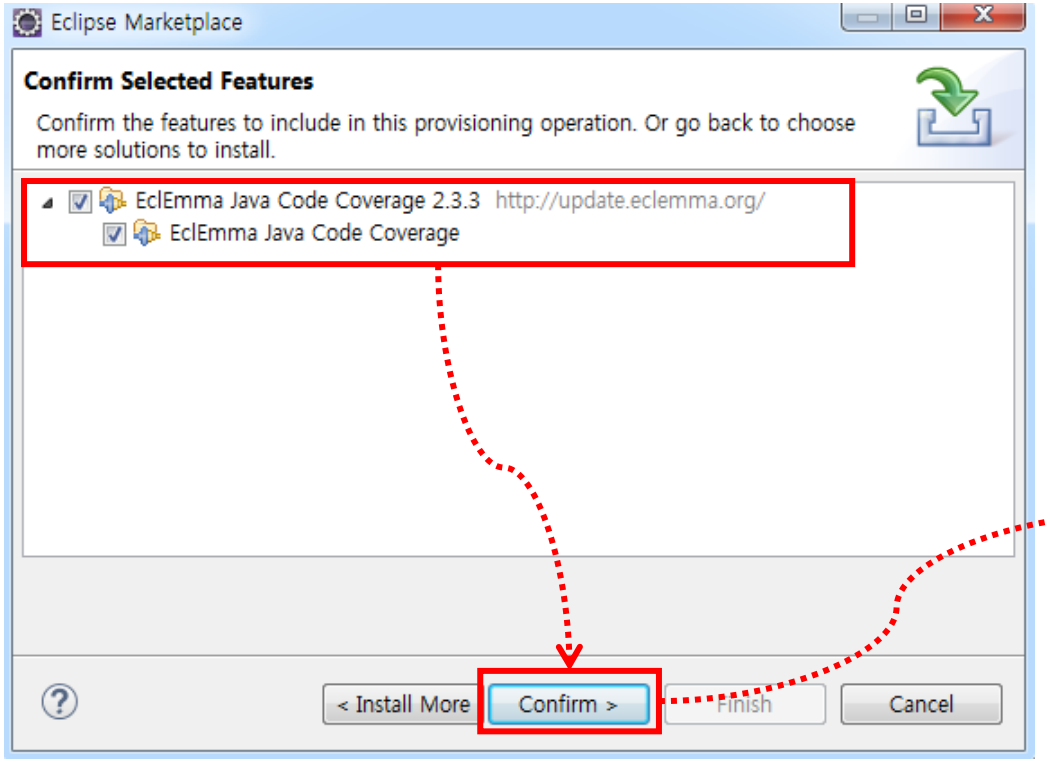
✓ EclEmma Java Code Coverage가 검색되면 install 버튼을 실행하여 해당 Plug-in을 설치합니다.

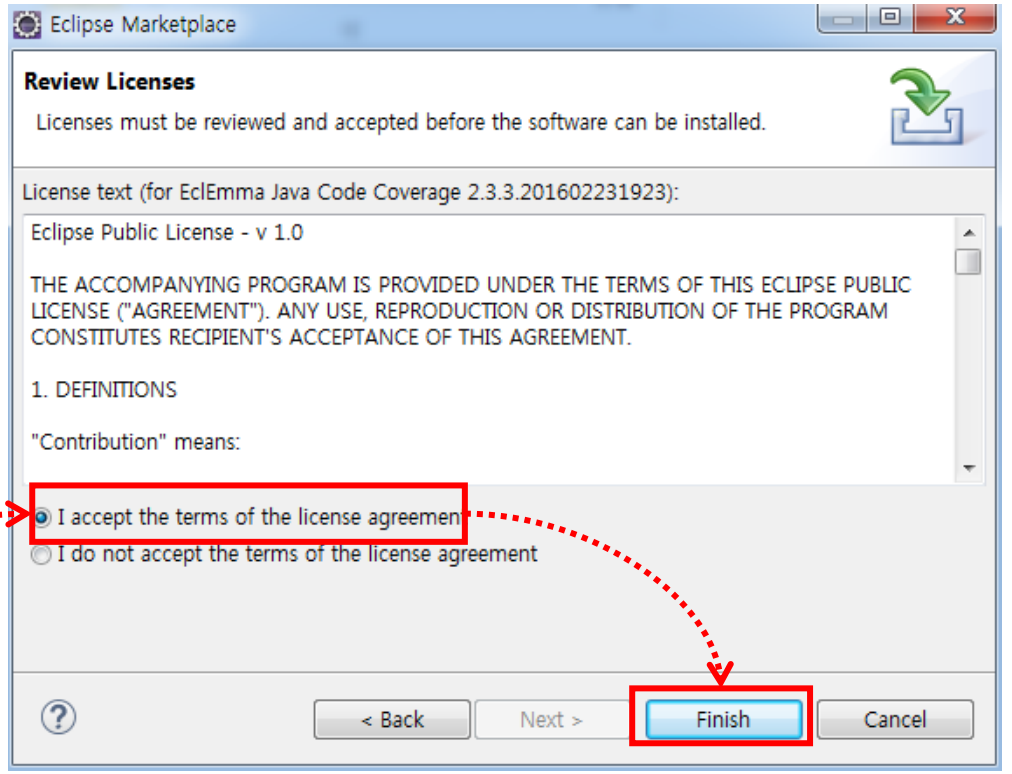


7.4 Emma (4/6)

✓ 설치할 항목을 체크한 후 Confirm 버튼을 실행합니다.

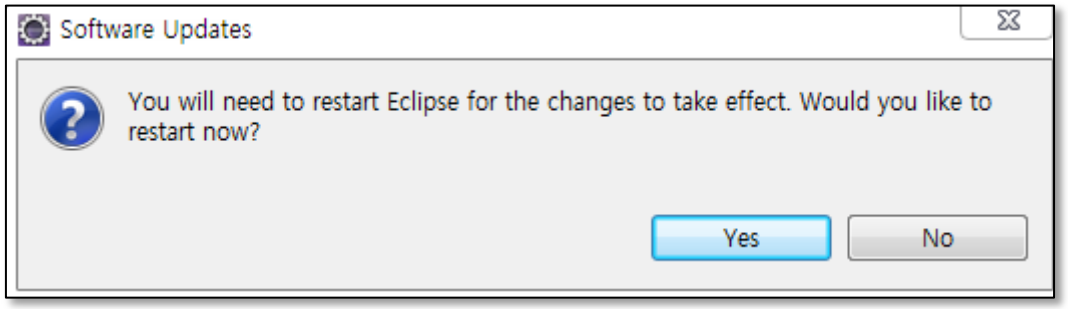
✓ 라이선스에 동의하고 Finish 를 실행하며 설치가 시작 됩니다.





7.4 Emma (5/6)

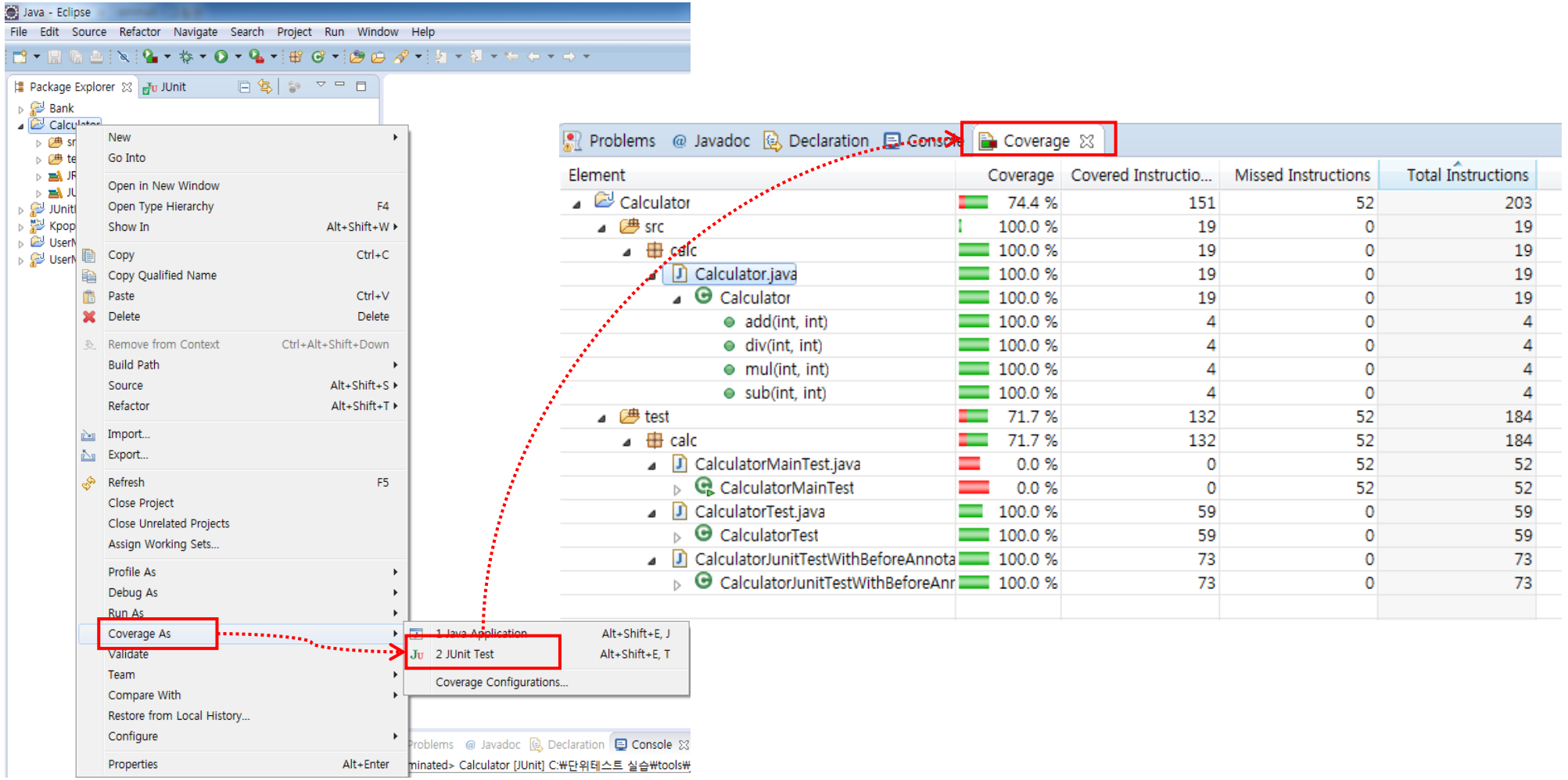
✓ 설치가 완료되고 이클립스를 재시작하면 EclEmma 를 사용하여 Code Coverage를 분석할 수 있습니다.



7.4 Emma (6/6)

✓ 테스트하고자 하는 프로젝트의 coverage As 메뉴를 선택하고 Junit Test를 실행합니다.

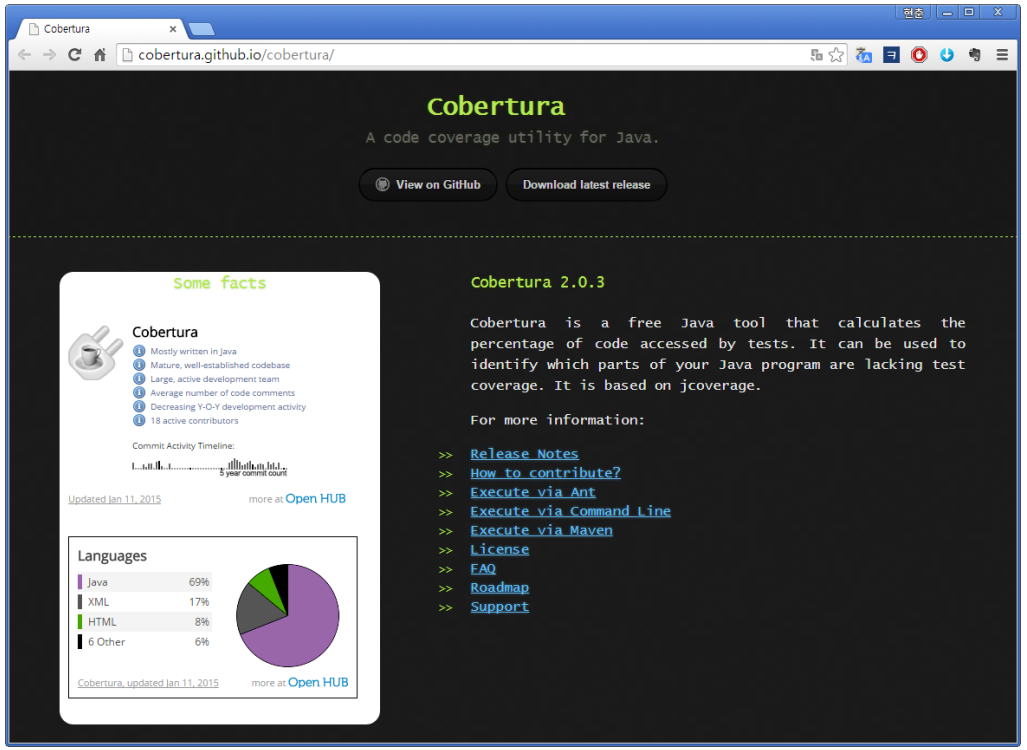
✓ Coverage 탭의 실행결과를 확인합니다.



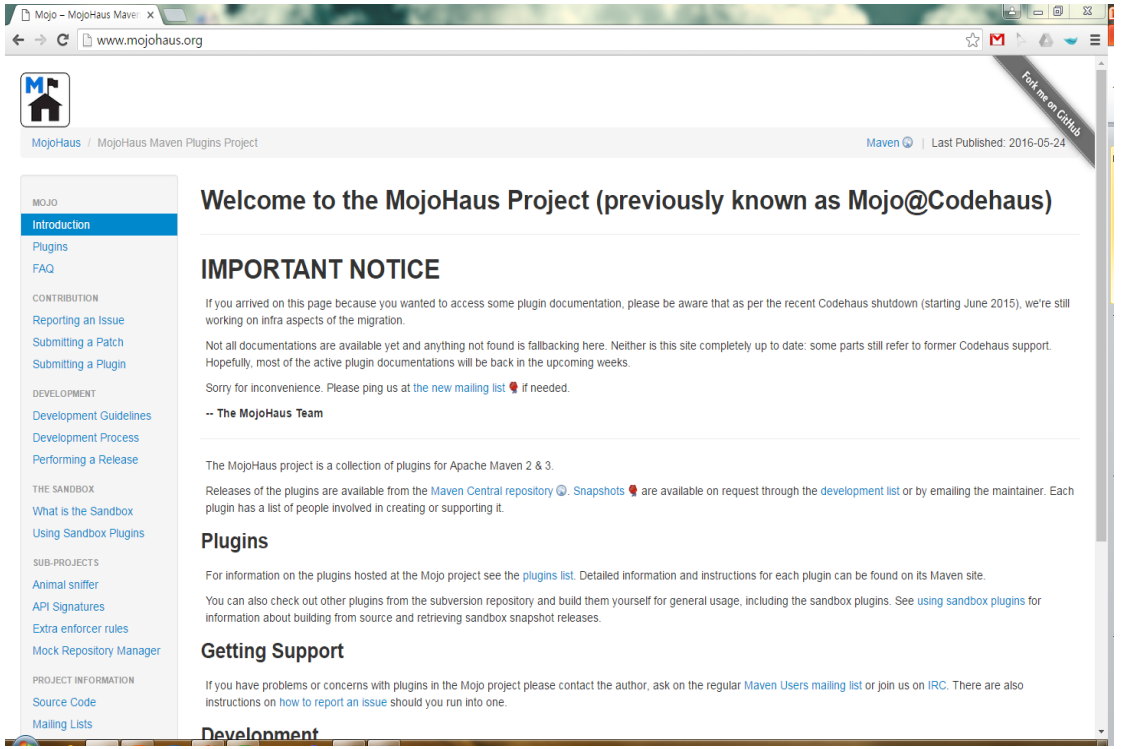
7.5 Cobertura (1/7)

✓ Cobertura 는 Java Code Coverage Tool로서 Ant, CommandLine, Maven을 지원 합니다.

✓ JCoverage를 기반으로 테스트 대상 소스의 테스트 커버리지 비율을 분석합니다.



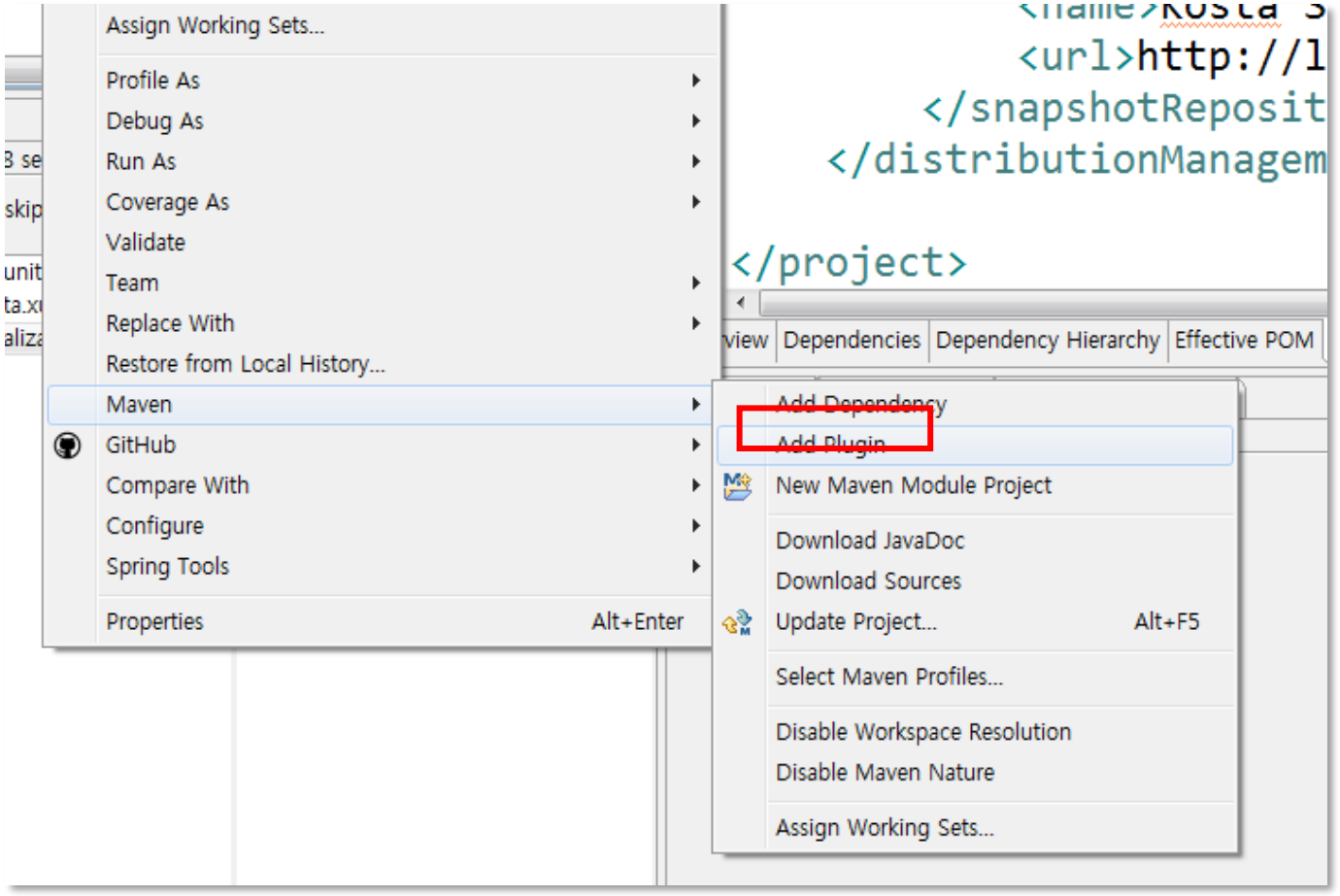
출처 : http://cobertura.github.io/cobertura/



출처 : http://www.mojohaus.org/

7.5 Cobertura (2/7)

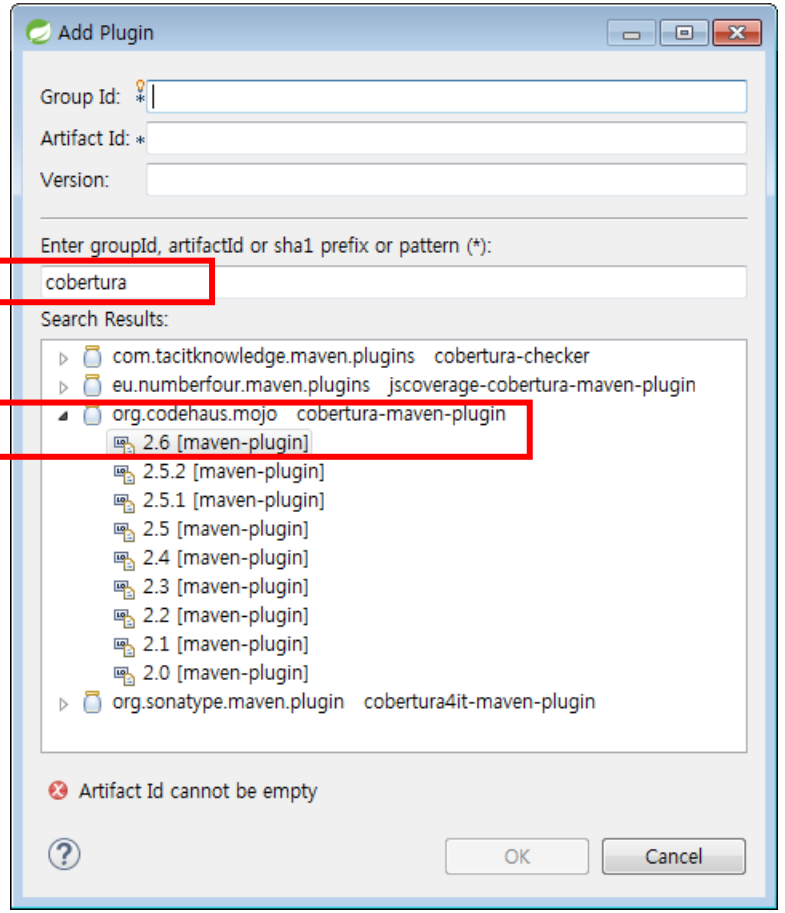
✓ Cobertura 를 이클립스에서 사용하기 위하여 Maven Plug-in을 설치 합니다.



7.5 Cobertura (3/7)

✓ 검색창에 cobertura 를 입력하고 검색된 결과에서 mojo Plug-in 최신버전을 선택 합니다.

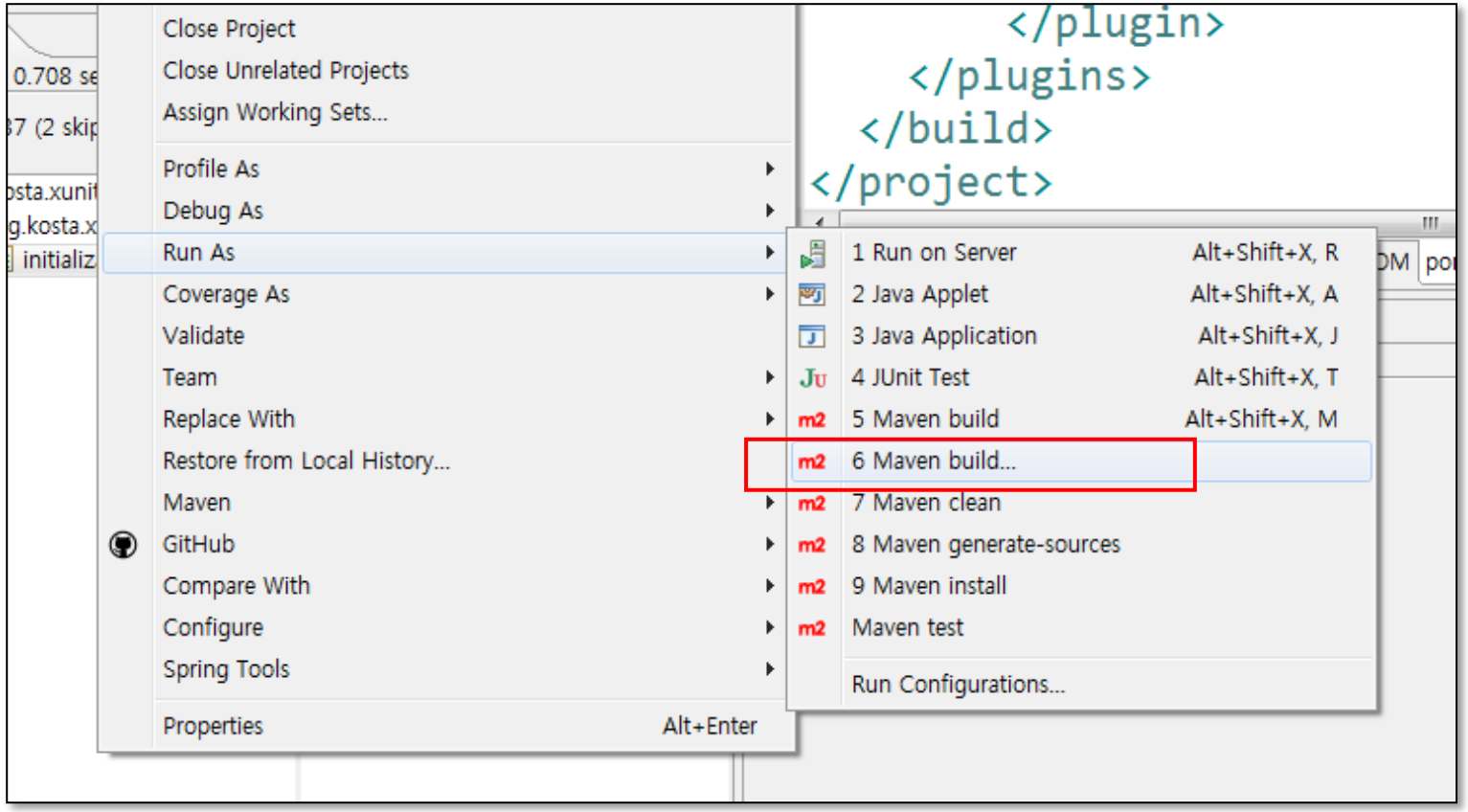
✓ 실행후 POM.xml 에 Cobertura Plugin 정보가 셋팅된 것을 확인할 수 있습니다.




```
<build>
  <plugins>
      <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>cobertura-maven-plugin</artifactId>
          <version>2.6</version>
      </plugin>
  </plugins>
</build>
```

7.5 Cobertura (4/7)

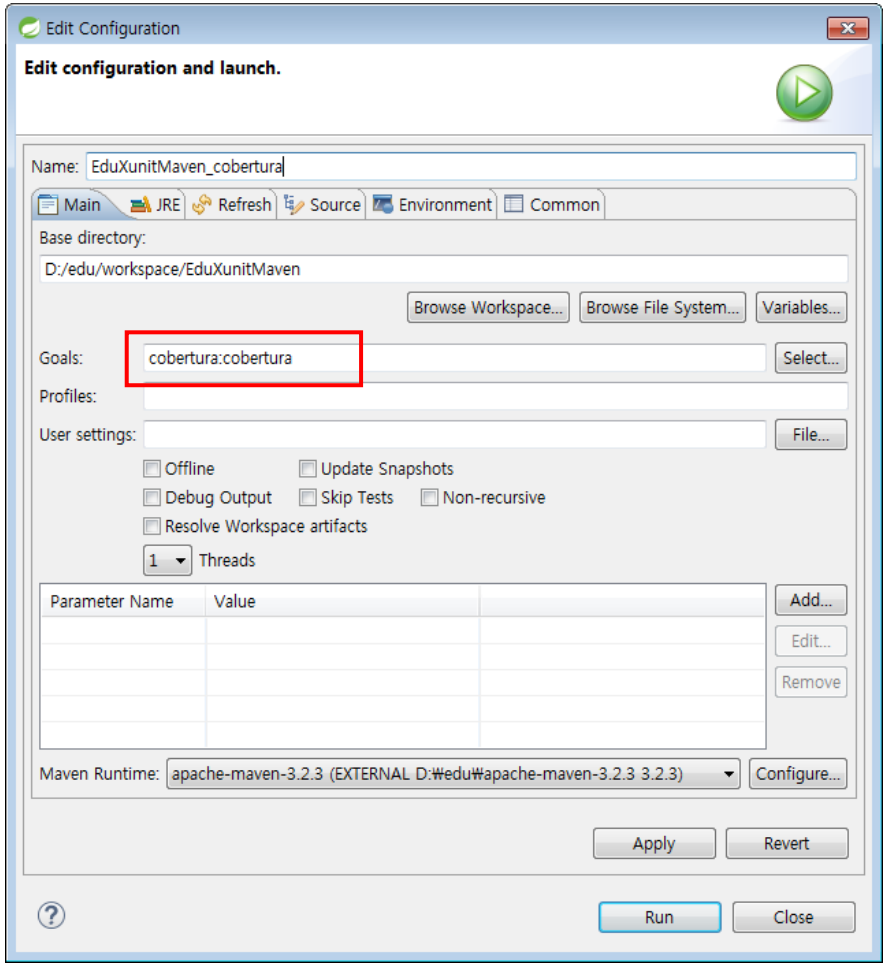
✓ 테스트하고자 하는 프로젝트에서 Maven 을 실행 합니다.



7.5 Cobertura (5/7)

✓ Maven 실행 윈도우에서 명령어를 Goals 에 입력 후 Run 을 수행합니다.

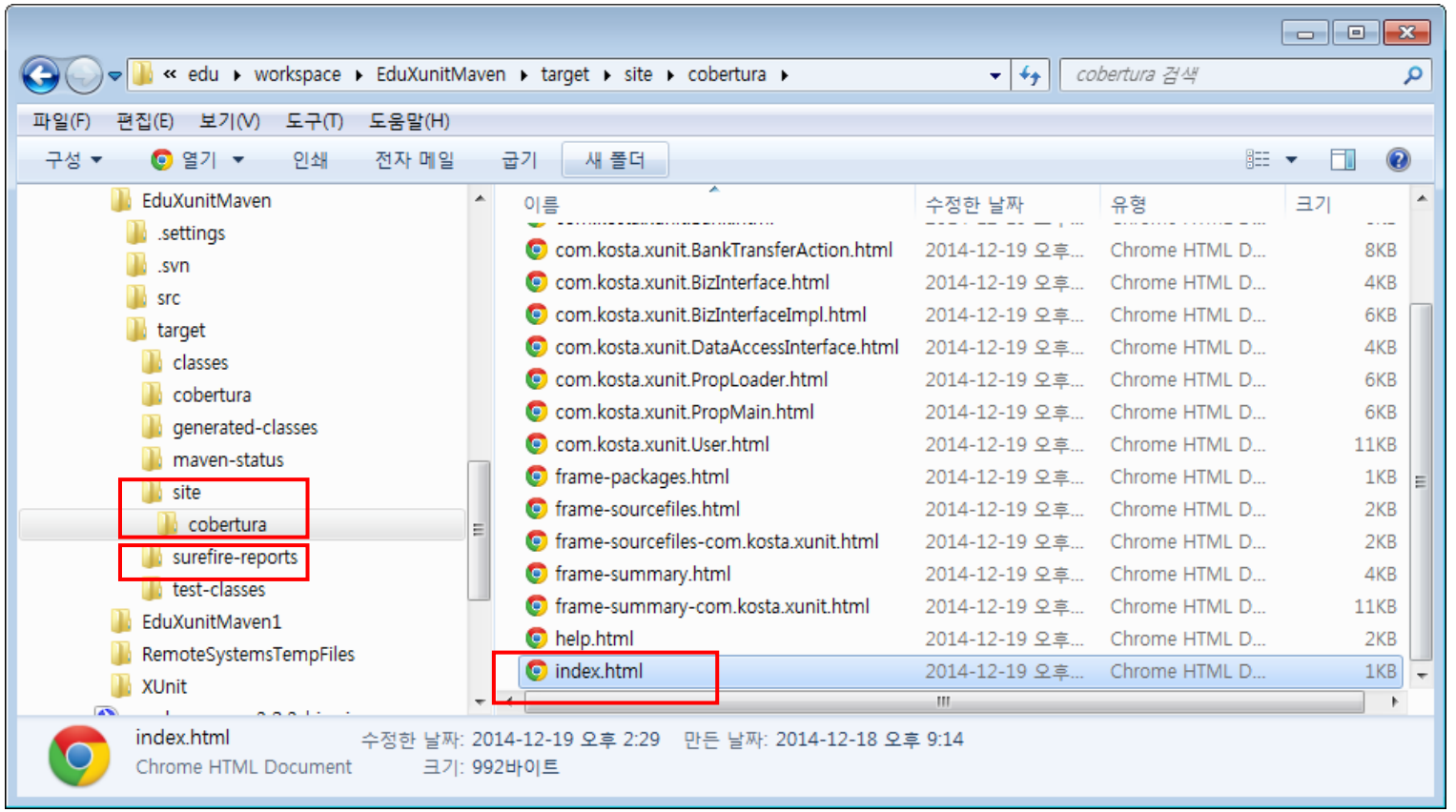
✓ 입력명령어 : Cobertura:Cobertura




```
Cobertura Maven Goals

- cobertura:check
- cobertura:clean
- cobertura:dump-datafile
- cobertura:instrument
- cobertura:cobertura
```

7.5 Cobertura (6/7)

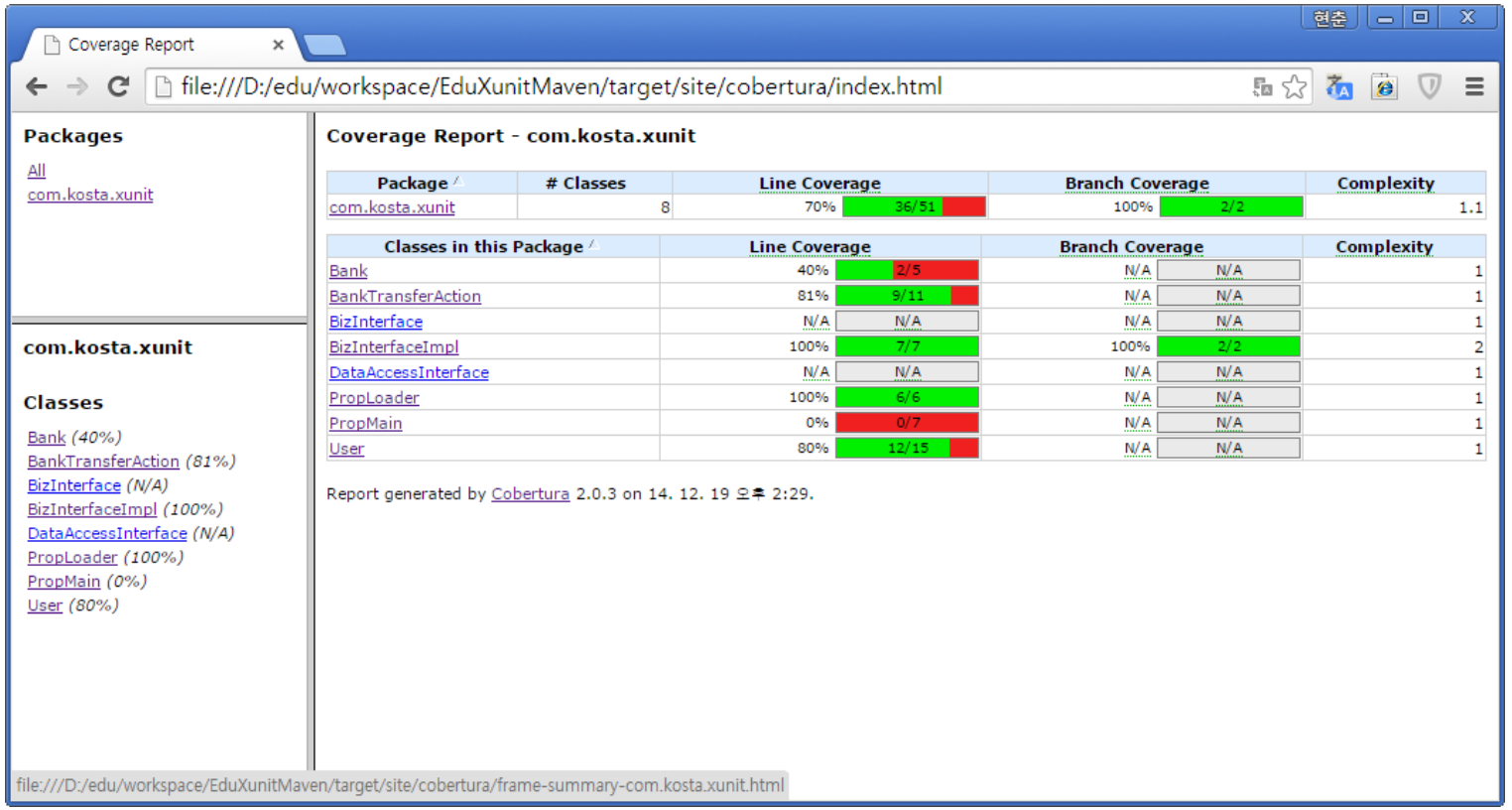
- ✓ 실행 결과는 workspace의 해당 프로젝트 target 디렉토리에서 확인합니다.
- ✓ Surefire-reports 디렉토리는 결과파일을 저장합니다.
- ✓ Site 디렉토리의 cobertura 디렉토리는 CodeCoverage 탐색결과 리포트를 가지고 있습니다.
- ✓ index.html 파일을 실행하면 커버리지 측정결과를 확인할 수 있습니다.



7.5 Cobertura (7/7)

✓ 패키지, 클래스별로 확인할 수 있습니다.

✓ Line Coverage, Branch Coverage 모두 확인할 수 있습니다.



토론

✓질문과 대답

✓토론