

테스트 주도 개발 방법 워크샵

(TDD: Test Driven Development)

강사: 한동준

handongjoon@gmail.com

dongjoon.han@synetics.kr

교육 목표

- ◆ 테스트 주도 개발의 특징과 장/단점 이해
- ◆ 테스트 주도 개발 방법 이해
- ◆ 임베디드 C 개발 시 유의 사항 이해
- ◆ 기본적인 Assert 문의 사용 이해

01 TDD를 위한 SW 테스트 기본

소프트웨어 품질(Software Quality)

- 정의

- 소프트웨어가 지닌 바람직한 속성의 정도 [IEEE]
- 요구되는 기능을 발휘할 수 있는 소프트웨어 특성의 정도 [DoD]
- 소프트웨어가 기능, 성능 및 만족도에 있어서 명시된 요구사항 및 내재된 요구사항을 얼마나 충족하는 가를 나타내는 소프트웨어 특성의 총체 [Pressman]

- 소프트웨어 품질의 구분

- 제품(Product) 품질
 - 제품 자체가 가지는 품질
 - 완성된 소프트웨어가 운용될 환경에 올려져 최종 시스템이 완성되었을 때, 소비자가 요구하는 바에 얼마나 부합되는지를 나타내는 품질
- 프로세스(Process) 품질
 - 소프트웨어를 개발하기 위해 필요한 모든 개발 활동이 계획을 준수하여 개발하였는가를 나타내는 품질
 - 그 활동이 효과적인지에 대해 검토 (Review) 및 감사 (Audit) 활동을 통해 확인

소프트웨어 제품 품질 보증

- Verification

- 제품이 올바르게 생성되고 있는가?(Are we building the product **right**?)
『Boehm』
- 소프트웨어가 정확한 요구사항에 부합하여 구현되었음을 보장하는 활동
- '요구사항 명세서에 맞게 올바른 방법으로 제품을 만들고 있음'을 보장

- Validation

- 올바른 제품을 생성하고 있는가?(Are we building the **right** product?)
『Boehm』
- 소프트웨어가 고객이 의도한 요구사항에 따라 구현되었음을 보장하는 활동
- '고객이 의도한 환경이나 사용 목적에 맞게 올바른 제품을 만들고 있음'을 보장

SW 테스트 종류

● 정적(Static)인 방법

- 소프트웨어를 실행하지 않고 결함을 찾아내는 것
- 여러 참여자들이 모여 소프트웨어를 검토하여 결함을 찾아내거나 정적 검증 도구 이용
- 소프트웨어 개발 중에 생성되는 모든 산출물들에 대해서 적용 가능
- 대표적인 방법:
 - 동료검토(Peer Review)
 - 인스펙션(Inspection)
 - 워크스루(Walk-through)
 - 데스크체크(Desk Check)
 - 도구를 이용한 정적 분석
 - 룰 기반 정적분석(PMD, BugFind 등)

● 동적(Dynamic)인 방법

- 소프트웨어를 실행하여 결함을 찾아냄
- 발견된 결함은 디버깅 활동으로 확인하여 수정함
- 대표적인 방법
 - 테스트
 - 블랙박스 테스트
 - 화이트박스 테스트

개발 단계 별 주요 산출물과 테스트 기법

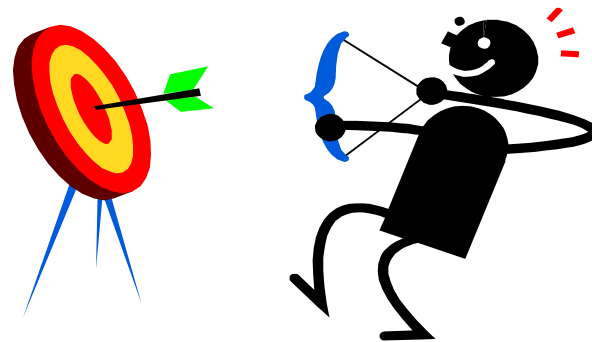
구분	공학 산출물	주요 품질 관리 기법
요구사항 정의	요구사항 정의서 인수 테스트 케이스	프로세스 감사 검토
요구사항 분석	요구사항 명세서 시스템 테스트 케이스	프로세스 감사 검토
상위 설계	아키텍처 설계서 인터페이스 설계서 통합 테스트 케이스	프로세스 감사 검토
상세 설계	상세 설계서 단위 테스트 케이스	프로세스 감사 검토
구현	소스코드	프로세스 감사 검토 정적 분석
테스트	테스트 결과서 결함	프로세스 감사 테스트

테스트의 목적

프로그램이 사용할 만 한 것인지 확인하기 위하여 결함을 발견할 목적으로 프로그램을 실행하는 작업

완벽한 테스트는 불가능

한 프로그램 내의 조건은 무수히 많을 수 있음
입력이 가질 수 있는 모든 값의 조합이 무수히 많음
GUI이벤트가 발생하는 순서에 대한 조합도 무수히 많음



결함이 없음을 보이려는 것이 아님

"프로그램 테스트는 결함이 있음을 보여줄 뿐, 결함이 없음을 증명할 수는 없다."

[Dahl Dijkstra Hoare]

현실적인 테스트의 목적

주어진 시간과 인력으로 오류를 발견할 확률이 높은, 가장 효율적인 테스트 케이스를 찾아내고 실행하는 일

테스트를 하면,
소프트웨어 품질 수준이 높아지는가?

테스팅 vs 디버깅



테스팅



디버깅

목적

- 알려지지 않은 결함의 발견

- 이미 알고 있는 결함의 수정

수행

- 테스터, QA 등 외부의 제 3자
- 개발자

- 개발자

주요 작업

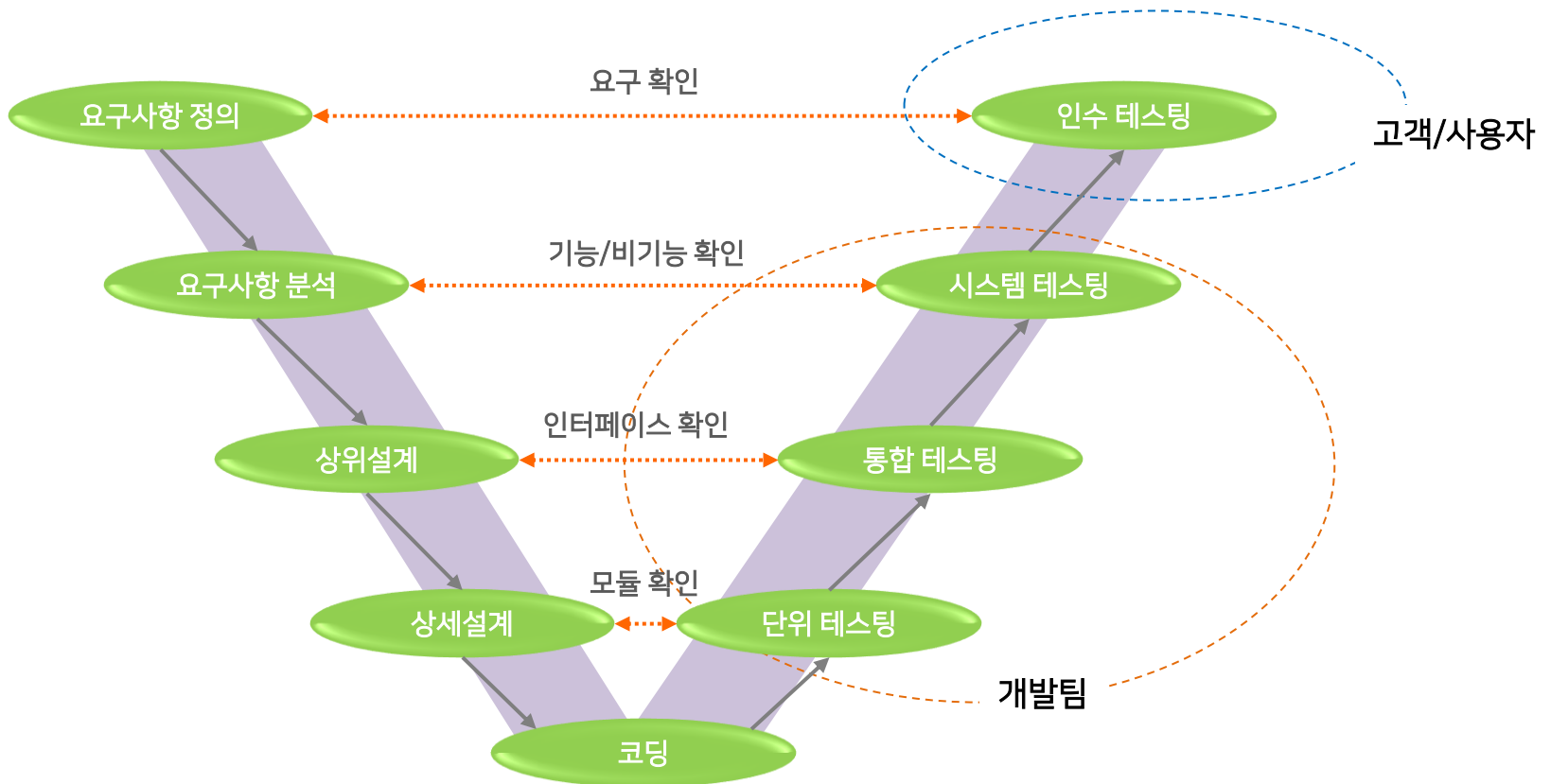
- 숨겨진 결함 **발견**

- 결함의 정확한 위치 파악
- 결함의 타입 식별
- 결함 **수정**

V모델 - 테스트 단계

소프트웨어 개발 프로세스로 폭포수 모델의 확장된 형태 중 하나
개발 생명주기의 각 단계와 그에 상응하는 소프트웨어 시험 각 단계의 관계를 나타냄

[위키백과]



단위 테스트(Unit Testing)

설계된 **모듈**이 **정확히 구현**되었는지 **확인**하고, 모듈과 같은 하나의 소프트웨어 구성요소나 소프트웨어 구성요소의 집합이 프로그램의 요구사항에 맞는지 확인하는 테스트

Testing conducted to verify the correct implementation of the design and compliance with program requirements for one software element (e.g., unit, module) or a collection of software elements. [IEEE 1012-1998 VVP]

목적

- 모듈이 올바르게 코딩 되었는지 확인

수행주체

- 개발자

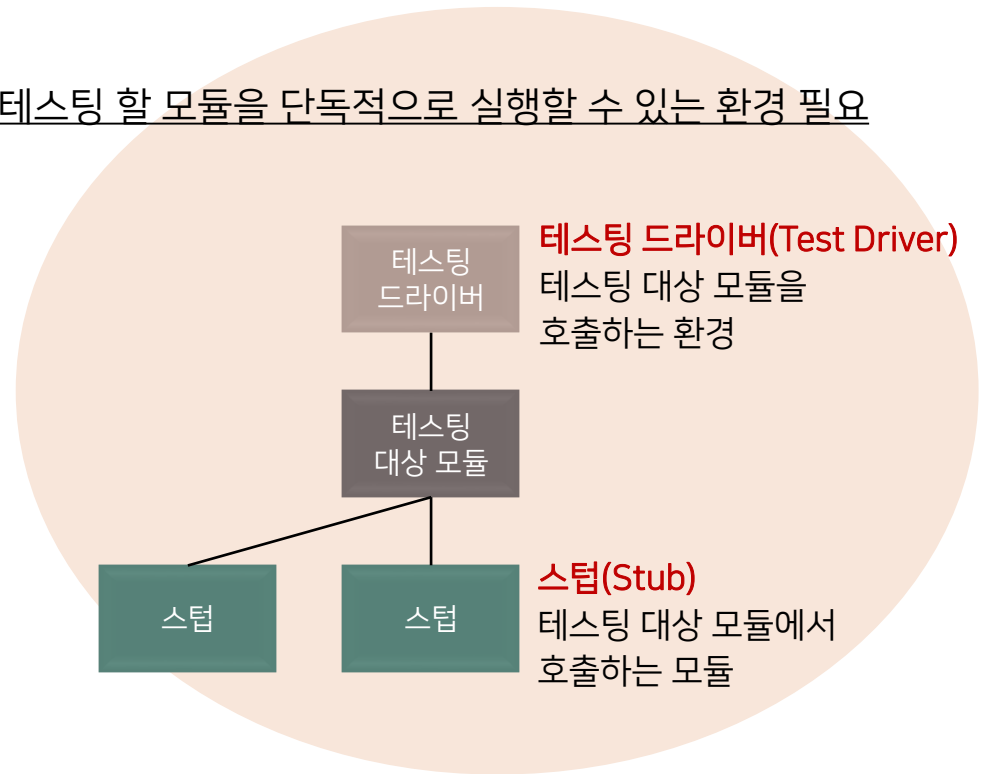
테스트 대상

- 모듈 수행기능(블랙박스), 코드 내부표현(화이트박스), 경계조건

완료시점

- 개발자가 더 이상의 오류가 없다고 판단될 때

테스팅 할 모듈을 단독적으로 실행할 수 있는 환경 필요



통합 테스트(Integration Testing)

시스템의 설계와 요구사항에 부합하는지 보이기 위해 소프트웨어 구성요소, 하드웨어 구성요소 등이
점차 통합되어 전체 시스템으로 통합될 때까지 이루어지는 절차적인 테스트

An orderly progression of testing of incremental pieces of the software program in which software elements, hardware elements, or both are combined and tested until the entire system has been integrated to show compliance with the program design, and capabilities and requirements of the system.

[IEEE 1012-1998]

빅뱅(Big Bang) 기법

- 모듈을 한꺼번에 통합하여 테스트를 하는 방법
- 오류가 발생하였을 경우 어느 부분에서 오류가 났는지 찾기가 힘들

하향식(Top-Down) 기법

- 가장 상위 모듈부터 하위 모듈로 점진적으로 통합하는 방법
- 상위 모듈 테스트 시, 하위 모듈에 대한 스텝이 필요

상향식(Bottom-Up) 기법

- 하위 모듈부터 테스트 하고 상위 모듈로 점진적으로 통합하는 방법
- 하위 모듈 테스트

시스템 테스트(System Testing)

모듈이 모두 통합된 후, 사용자의 요구사항이 만족되었는지 검사하는 테스트

고객에게 시스템을 전달하기 전, 시스템을 개발한 조직이 주체가 되는 마지막 테스트

The activities of testing an integrated hardware and software system to verify and validate whether the system meets its original objectives.

[IEEE 1012-1998 VVP]

테스팅 대상

- 요구사항 명세서를 기초로 하여 사용자의 기능 요구사항
- 보안, 성능, 신뢰성, UX 등의 비 기능 요구사항

인수 테스트(Acceptance Testing)

시스템이 **사용자에게 인수되기 전**, **사용자**에 의해 **실시**되는 테스트

실제 사용자가 운영하는 환경에서 실시

인수 테스트를 통과해야만 시스템이 정상적으로 사용자에게 인수되고 프로젝트는 종료됨

Testing conducted in an operational environment to determine whether a system satisfies its acceptance criteria (i.e., initial requirements and current needs of its user) and to enable the customer to determine whether to accept the system.

[IEEE 1012-1998 VVP]

테스트 케이스(Test Case)

특정한 요구사항이 제대로 구현되었는지를 검증하기 위하여 테스트 조건, 입력 값, 예상 출력 값, 그리고 실제 수행한 결과를 기록하는 것

A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement

[IEEE-Std-610]

테스트 케이스 ID: ST-0001					
목적	로그인 시 아이디와 패스워드의 대소문자를 구분하여 처리한다.				
테스트 사전조건	아이디/비번 : abcd / abcd 가 DB에 이미 입력되어 있음.				
테스터	홍길동	테스트 일자	2006.10.01~2006.10.01		
단계	입력값	예상 출력 값	실행 결과	조치사항	조치 후 시험결과
1	아이디: ABCD 패스워드: abcd	아이디 없음 경고	정상 로그인	디버깅 필요	아이디 없음 경고
2	아이디: abcd 패스워드: ABCD	패스워드 틀림 경고	패스워드 틀림 경고	-	-
3	아이디: abcd 패스워드: abcd	정상 로그인	정상 로그인	-	-

테스트 케이스(Test Case)의 필수 항목

입력값

테스트 절차

예상출력값

테스트 자동화도 동일

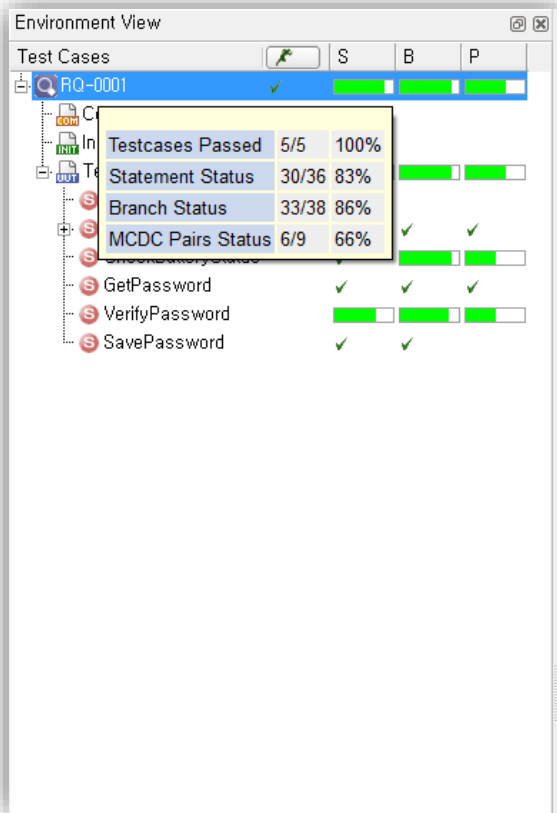
ISO 26262에서 요구하는 커버리지

Table 12 — Structural coverage metrics at the software unit level

Methods		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)	+	+	+	++

◆ 커버리지 달성 목표 %는?

도구 별 커버리지 측정 대상 - VectorCAST



```
else
3 5      ( ) ( )      if(
3 5.1    ( ) ( )      No_External_Open_Switch_in == ON){
3 6      Sound(NO_EXTERNAL_OPEN);
3 7      result = FAIL;
        }
3 8      return result;
    }
enum return_type CheckCloseFlag(enum switch_id_
enum return_type result = FAIL;

4 0      (T)      CheckCloseFlag
4 1      (T) (F)    if((
4 1.1    (T) (F)    Auto_Lock_Switch_in == ON)&&(
4 1.2    (T) ( )    Door_Sensor_in == CLOSE)){
4 2      *          result = SUCCESS;
        }
        else
4 3      (T) ( )      if(
4 3.1    (T) ( )      Auto_Lock_Switch_in == OFF){
4 4      *          result = ElecKeySensing();
```

도구 별 커버리지 측정 대상 - Junit + Cobertura

Coverage Report - All Packages

Package /	# Classes	Line Coverage	Branch Coverage
All Packages	257	86% 3521/4091	84% 1093/1302
junit.extensions	6	82% 52/63	87% 7/8
junit.framework	17	76% 399/525	90% 139/154
junit.runner	3	46% 67/144	41% 20/48
junit.textui	2	76% 99/130	76% 23/30
org.junit	14	86% 211/244	86% 85/98
org.junit.experimental	2	91% 21/23	83% 5/6
org.junit.experimental.categories	11	93% 149/159	94% 83/88
org.junit.experimental.max	8	85% 92/108	86% 26/30
org.junit.experimental.results	6	92% 37/40	87% 7/8
org.junit.experimental.runners	1	100% 7/7	100% 4/4
org.junit.experimental.theories	15	93% 167/178	83% 72/86
org.junit.experimental.theories.internal	8	91% 189/207	85% 77/90
org.junit.experimental.theories.suppliers	2	100% 7/7	100% 2/2
org.junit.internal	13	92% 132/164	94% 33/36
org.junit.internal.builders	8	98% 57/58	92% 13/14
org.junit.internal.matchers	4	75% 40/53	0% 0/18
org.junit.internal.requests	3	100% 29/29	75% 3/4
org.junit.internal.runners	18	73% 312/423	63% 84/132
org.junit.internal.runners.model	3	100% 26/26	100% 4/4
org.junit.internal.runners.rules	1	100% 40/40	100% 24/24
org.junit.internal.runners.statements	8	92% 128/139	73% 28/38
org.junit.matchers	1	9% 1/11	N/A N/A
org.junit.rules	21	87% 233/265	100% 30/30
org.junit.runner	19	94% 210/222	90% 49/54
org.junit.runner.manipulation	9	90% 38/42	100% 18/18
org.junit.runner.notification	14	97% 140/144	80% 16/20
org.junit.runners	16	97% 346/354	96% 110/114
org.junit.runners.model	15	96% 225/233	91% 102/112
org.junit.validator	9	88% 47/53	91% 11/12

Report generated by Cobertura 2.0.3 on 14. 4. 6 오후 2:03.

Coverage Report - org.junit.internal.requests.ClassRequest

Classes in this File	Line Coverage	Branch Coverage
ClassRequest	100% 13/13	75% 3/4

```

1 package org.junit.internal.requests;
2
3 import org.junit.internal.builders.AllDefaultPossibilitiesBuilder;
4 import org.junit.runner.Request;
5 import org.junit.runner.Runner;
6
7 public class ClassRequest extends Request {
8     209 private final Object fRunnerLock = new Object();
9     private final Class<?> fTestClass;
10    private final boolean fCanUseSuiteMethod;
11    private volatile Runner fRunner;
12
13    209 public ClassRequest(Class<?> testClass, boolean canUseSuiteMethod) {
14        209     fTestClass = testClass;
15        209     fCanUseSuiteMethod = canUseSuiteMethod;
16    209 }
17
18    public ClassRequest(Class<?> testClass) {
19        178     this(testClass, true);
20    178 }
21
22    @Override
23    public Runner getRunner() {
24        212     if (fRunner == null) {
25        209         synchronized (fRunnerLock) {
26        209             if (fRunner == null) {
27                209                 fRunner = new AllDefaultPossibilitiesBuilder(fCanUseSuiteMethod).safeRunnerFor(
28                    }
29        209             }
30        }
31    212     return fRunner;
32    }
33 }

```

Report generated by Cobertura 2.0.3 on 14. 4. 6 오후 2:03.

Junit으로 Junit을 테스트 한 커버리지

테스트 자동화는 테스트 케이스 개발의 자동화가 아닌 테스트 수행의 자동화

테스트 케이스는 사람이 개발하고, 수행을 컴퓨터가 함

02 테스트 주도 개발 (Test Driven Development)

TDD

- ✓ Test Driven Development: 테스트 주도 개발
- ✓ Test First Development: 테스트 우선 개발
- ✓ 테스트 케이스를 먼저 만들고, 테스트를 통과하는 코드를 작성
테스트가 통과하면 코드를 리팩토링
- ✓ 한 번에 모든 테스트 케이스를 만들지 않고, 조금씩 만들어 감
(Baby Step)

TDD의 사용 예

✓ 테스트 케이스 작성

SumTest.c

```
static void testSum(void)
{
    assert(sum(3, 2) == 5);
}
```

✓ 테스트 케이스 실행 - 실패

✓ 테스트 케이스를 통과하는 코드 작성

Sum.c

```
int sum(int a, int b)
{
    return a + b;
}
```

✓ 테스트 케이스 실행 - 성공

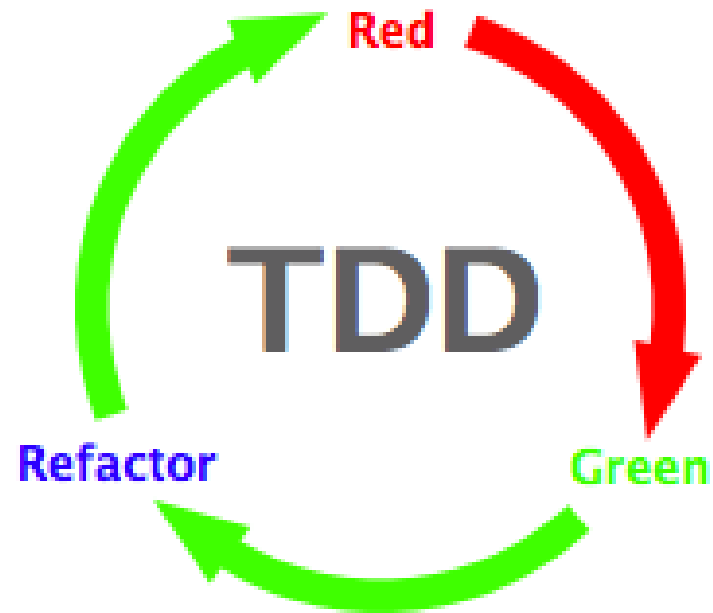
✓ 코드 구조 개선(리팩토링)

✓ 테스트 케이스 실행 - 성공

반복



TDD 방식



TDD 장단점

장점

- ✓ 코드의 목적을 명확히 하고 코딩 진행
- ✓ 지속적인 코드 개선
- ✓ 테스트 케이스가 문서의 역할 수행

단점

- ✓ 단위 테스트에 과도한 몰입

어떤 단위 테스트를 사용해야 하는가?

◆ TDD는 코딩과 단위 테스트 행위를 구분하지 어려움

- 코딩 = 테스트 코드 + 제품 코드 작성. 즉, 하나의 IDE에서 작성/실행 필요

◆ 일반 SW의 경우 대표적인 단위 테스트 프레임워크 활용

- Java, C#, JS... : Xunit 프레임워크와 Mock 도구
- 일반적인 C/C++: Google Test, CppUTest, 기본 assert 등

◆ 기능 안전의 경우

- 일반적으로 단위 테스트에 상용 테스트 도구를 활용
- Assert 문 및 C/C++ 테스트 프레임워크의 활용이 Tool Certi.와 관련이 없을지 확인은 필요
- 상용 테스트 도구는 커버리지 측정 및 추가/보완 테스트 작성의 용도로 활용하는 방안 검토 필요

assert() 사용하기

◆ assert(boolean): boolean이 True이면 테스트 성공, False이면 테스트 실패

- Assert: 단언하다.
- assert(1): 테스트 성공

◆ 사용 방법

- Include <assert.h>
- assert 문 내부에, 테스트 하려는 함수의 비교 결과를 작성
- assert(3*3 == 9); // 3*3이 9가 맞니?
- 관례적으로 좌측에 함수 실행 결과, 우측에 내가 예상하는 결과를 작성

◆ 테스트 파일 작성(관례적으로)

- 제품 코드는 src 폴더에, 테스트 코드는 test 폴더에
- "제품코드.c"의 테스트 파일은 "제품코드.tests.c" 또는 "제품코드Test.c"

◆ 테스트 함수 작성(관례적으로)

- "제품함수()"를 테스트하기 위한 "test제품함수()" 작성

assert() 예제


```
#include "calc.h"

int sum(int a, int b)
{
    return a + b;
}
```

```
#include "calc.h"
#include <assert.h>
#include <stdio.h>

static void testSum(void)
{
    assert(sum(3, 2) == 5);
}

int main(void)
{
    testSum();
    puts("All tests passed");
}
```



```
1 :stdout:
2 All tests passed
```

[참고] GoogleTest

- ◆ xUnit 아키텍처를 기반으로 하는 C++용 단위 테스트 라이브러리
 - xUnit: AdaUnit, Junit을 기반으로 하는 단위 테스트 프레임워크의 총칭
- ◆ Assertion 함수 호출을 통해 테스트 수행
 - Fatal assertion: 테스트가 실패하면 테스트 중단
 - Nonfatal assertion: 테스트가 실패해도 모든 테스트 실행

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_EQ(val1, val2);	EXPECT_EQ(val1, val2);	val1 == val2
ASSERT_NE(val1, val2);	EXPECT_NE(val1, val2);	val1 != val2
ASSERT_LT(val1, val2);	EXPECT_LT(val1, val2);	val1 < val2
ASSERT_LE(val1, val2);	EXPECT_LE(val1, val2);	val1 <= val2
ASSERT_GT(val1, val2);	EXPECT_GT(val1, val2);	val1 > val2
ASSERT_GE(val1, val2);	EXPECT_GE(val1, val2);	val1 >= val2

- ◆ Setup/Teardown을 지원

Google Test 코드 예제 (C의 경우)

```
TEST(TestSuiteName, TestName) {  
    ... test body ...  
}
```

```
#include <gtest/gtest.h>  
  
extern "C"  
{  
    #include "hiker.h"  
}  
  
using namespace ::testing;  
  
TEST(Hiker, Life_the_universe_and_everything)  
{  
    EXPECT_EQ(42, answer());  
}
```

Google Test 설명

1. 테스트는 독립적이고 반복할 수 있어야 한다. 다른 테스트의 결과에 따라 결과가 달라지는 테스트는 좋은 테스트가 아니다. 구글테스트는 각각의 테스트를 분리하여 다른 오브젝트로 관리할 수 있도록 도와준다.
2. 테스트는 잘 구조화되고 테스트하는 코드를 잘 반영해야 한다. 구글테스트는 관련된 테스트를 test suite로 그룹화하여 데이터와 subroutine을 공유할 수 있도록 한다.
3. 테스트는 재사용 가능하고 플랫폼 종속되지 않아야 한다. 구글테스트는 다른 OS에서도 돌 수 있도록 한다.
4. 테스트가 Fail했을 때 왜 실패했는지를 보고해주기 때문에 버그를 쉽게 찾을 수 있다.
5. 테스트 프레임워크는 테스트 작성자들의 귀찮음을 덜어주고 테스트 자체에 집중할 수 있도록 만들어준다.
6. 테스트는 빨라야 한다. 구글테스트를 이용해서 shared resource를 테스트 간에 재사용 할 수 있고, 한번만 실행되는 set-up/tear-down 메소드도 사용할 수 있다.

TDD와 지속적 통합

◆ 지속적 통합 (참고자료 참조)

- 개발한 소프트웨어를 빠르게 통합하는 것
- 모든 SW 개발에 적용 가능한 활동이며, 널리 확산됨

◆ TDD와 지속적 통합은 SW 개발 관점에서 동일한 목적을 가짐

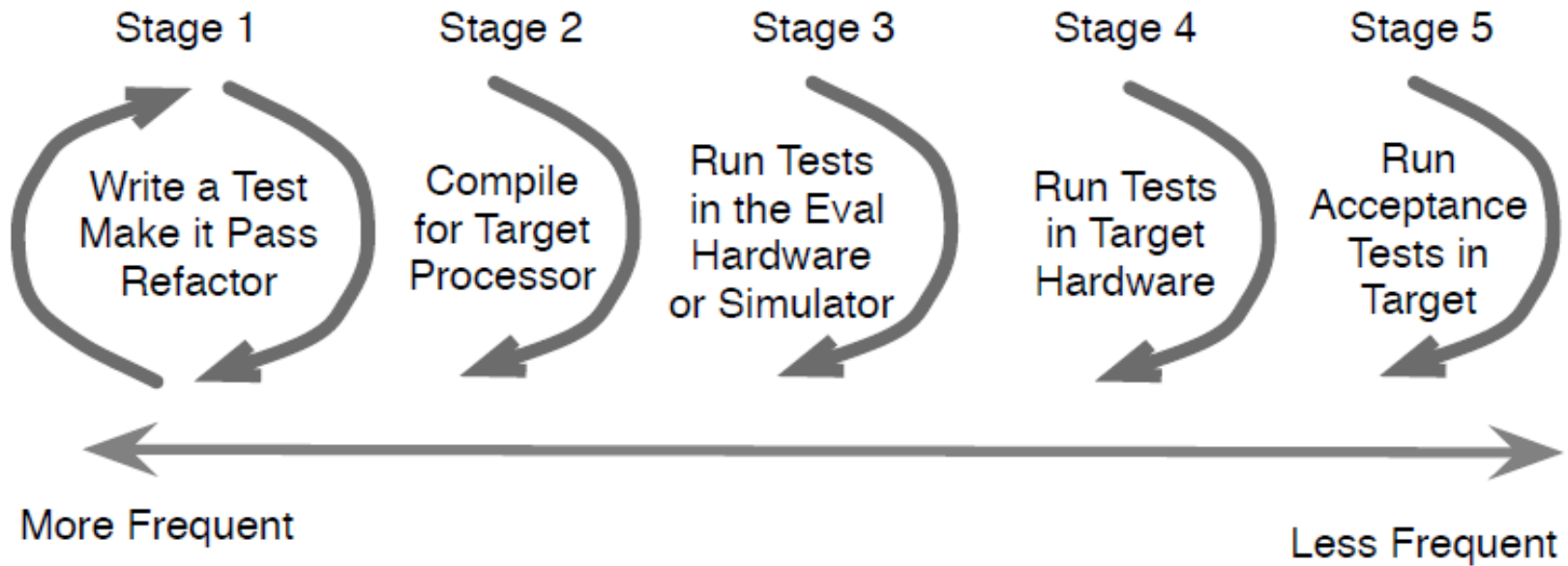
- 문제의 크기가 작을 때 빠르게 해결
- 지속적으로 문제를 검증하고 해결

[참고]

임베디드 테스트 환경 구축

테스트 환경 구축의 필요성

◆ 임베디드 환경에서 권장되는 테스트 환경



시뮬레이션 PC 환경		Embedded 제품환경	
장점	단점	장점	단점
<ul style="list-style-type: none"> - 조기 구축 가능 - H/W와 S/W 문제 분리 - 테스트 수행에 적은 비용 - 가장 높은 ROI 	<ul style="list-style-type: none"> - 테스트 정확도가 떨어질 수 있음 - 테스트 환경 구축에 많은 비용 발생 가능성 	<ul style="list-style-type: none"> - 정확한 테스트 수행 	<ul style="list-style-type: none"> - 테스트 환경 구축 및 수행에 많은 비용 발생 가능성

출처: 임베디드 C를 위한 TDD

스텝 만들기

◆ 유형

- 링크타임 치환
- 함수포인터 치환
- 전처리기 치환

◆ 목적

- 구현된 코드 중 시뮬레이션 환경상에서 빌드 및 실행을 위해 OS Dependency 제거
- 미 구현된 코드로 테스트 수행을 위해 교체
- 테스트의 독립성을 확보하기 위하여 교체
 - 3rd Party Library 등 외부 간접을 제거 하기 위한 용도

03 리팩토링(Refactoring)

리팩토링 개요

◆ 정의

- 소프트웨어의 동작 변화 없이 내부 구조를 개선하는 것
- = 함수의 입력과 출력을 고정시킨 상태로 내부 구조를 개선하는 것

◆ 왜 해야 하는가?

- 소프트웨어 설계 개선
- 소프트웨어를 더 이해하기 쉽게
- 더 빠르게 결함을 찾기 위해
- 더 빠르게 소프트웨어를 개발하기 위해

◆ 언제 하는가?

- 코드 구린내(Code Smell)가 확인되면
 - 코드 구린내: 중복 코드, 긴 코드, 긴 매개변수, 이해하기 힘든 변수/함수명, 불필요한 Switch, 여러 일을 수행하는 하나의 함수...
 - 기능 안전의 경우 표준 또는 내부 요건

단위 테스트의 존재

- ◆ 함수의 구조를 변경해도, 정상 동작하는지 확인이 가능해야 함
- ◆ 리팩토링은 함수의 입/출력은 고정

= 단위 테스트는 함수의 입/출력을 검증

= 함수 내부를 개선해도, 동작은 문제가 없음을 확인하기 위함

TDD는 단위 테스트를 먼저 작성하고 제품 코드를 작성함

가장 기본적인 리팩토링 - 함수 추출(Extracting)

- ◆ 하나로 묶을 수 있는 코드가 있으면, 코드 목적이 잘 표현되는 이름을 지어 함수로 추출
- ◆ 언제 사용하는가?
 - 너무 긴 함수
 - 중복 되는 함수
 - 너무 복잡한 함수
- ◆ 어떻게 사용하는가?
 - 함수를 대표할 이름을 만들고, 복사
 - 변수 및 파라미터 처리
 - 컴파일
 - > IDE의 리팩터 메뉴를 활용
- ◆ 주의점
 - 지역 변수가 있는 경우의 처리

기능 안전 SW의 주요 소스코드 품질 매트릭: 리팩토링 대상

◆ 규모 관련

- 라인 수(LOC: Line of Code)
- 주석 제외 라인 수
- 주석 비율
- 함수 별 라인 수

◆ 복잡도 관련

- 순환 복잡도(Cyclomatic Complexity): 함수의 제어 흐름이 얼마나 복잡한지 측정. 분기문 +1
- Nesting Depth: if안의 if 안의 if...

◆ 테스트 관련

- 기능 커버리지
- 함수 커버리지
- 문장/분기/MCDCC 커버리지

◆ 정적분석 관련

- MISRA 룰 위반 수

기능 안전 SW의 주요 소스코드 품질 매트릭: 리팩토링 대상

◆ 의존성 관련

- 함수 별 호출하는 건수
- 함수 별 호출되는 건수
- 변경 영향 비율(Stability): 한 함수가 변경되면 코드의 몇 %가 영향받는가?
- 상호 참조 비율: 상호 호출하는 파일이 있는가? (변경에 대한 영향도 파악 목적)

소스코드 품질 매트릭의 정의

지표명	순환 복잡도 위반 건수 (LDRA: Cyclomatic Complexity)	
지표 정의	LDRA 분석 수행을 통해 발견한 함수의 순환 복잡도 10 이상 건수	
산식	A 건	A: 순환 복잡도 10 이상 건수 B: -
수집 단계	구현 단계 이후	
수집 주기	매일(Jenkins 자동 실행)	
수집 대상	LDRA (Jenkins 자동 실행) / *.mts.htm	
검토 주기	매일	
해석	높을수록 소스코드를 이해하기 어려운 코드가 많아, 유지보수 노력과 잠재 결함의 발생 가능성을 증가시킴	
목표 정의	0 건	

아키텍처 설계 원칙 요건(ISO 26262)

Table 3 — Principles for software architectural design

Principles		ASIL			
		A	B	C	D
1a	Appropriate hierarchical structure of the software components	++	++	++	++
1b	Restricted size and complexity of software components ^a	++	++	++	++
1c	Restricted size of interfaces ^a	+	+	+	++
1d	Strong cohesion within each software component ^b	+	++	++	++
1e	Loose coupling between software components ^{b,c}	+	++	++	++
1f	Appropriate scheduling properties	++	++	++	++
1g	Restricted use of interrupts ^{a,d}	+	+	+	++
1h	Appropriate spatial isolation of the software components	+	+	+	++
1i	Appropriate management of shared resources ^e	++	++	++	++
<p>^a In principles 1b, 1c, and 1g “restricted” means to minimize in balance with other design considerations.</p> <p>^b Principles 1d and 1e can, for example, be achieved by separation of concerns which refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal, task, or purpose.</p> <p>^c Principle 1e addresses the management of dependencies between software components.</p> <p>^d Principle 1g can include minimizing the number, or using interrupts with a clear priority, in order to achieve determinism.</p> <p>^e Principle 1i applies for shared hardware resources as well as shared software resources in the case of coexistence. Such resource management can be implemented in software or hardware and includes safety mechanisms and/or process measures that prevent conflicting access to shared resources as well as mechanisms that detect and handle conflicting access to shared resources.</p>					

* 출처) ISO26262, Part 6

상세 설계 원칙 요건(ISO 26262)

Table 6 — Design principles for software unit design and implementation

Principle		ASIL			
		A	B	C	D
1a	One entry and one exit point in subprograms and functions ^a	++	++	++	++
1b	No dynamic objects or variables, or else online test during their creation ^a	+	++	++	++
1c	Initialization of variables	++	++	++	++
1d	No multiple use of variable names ^a	++	++	++	++
1e	Avoid global variables or else justify their usage ^a	+	+	++	++
1f	Restricted use of pointers ^a	+	++	++	++
1g	No implicit type conversions ^a	+	++	++	++
1h	No hidden data flow or control flow	+	++	++	++
1i	No unconditional jumps ^a	++	++	++	++
1j	No recursions	+	+	++	++
^a Principles 1a, 1b, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.					

* 출처) ISO26262, Part 6

모듈화 요건

◆ 모듈의 의미

- 수행 가능 명령어, 자료구조 또는 다른 모듈을 포함하고 있는 독립 단위

◆ 특성

- 이름을 가지며
- 독립적으로 컴파일 되고
- 다른 모듈을 사용할 수 있고
- 다른 프로그램에서 사용될 수 있다

◆ 모듈의 크기

- 되도록 쉽게 이해될 수 있도록 가능한 한 작아야 함
- 너무 작은 모듈로 나뉘지지 않도록 함

정보 은닉(Information Hiding)

◆ 의미

- 각 모듈 내부 내용에 대해서는 비밀로 묶어두고, 인터페이스를 통해서만 메시지를 전달 할 수 있도록 하는 개념
- 설계상의 결정 사항들이 각 모듈 안에 감추어져 다른 모듈이 접근하거나 변경하지 못하도록 함

◆ 장점

- 모듈의 구현을 독립적으로 맡길 수 있음
- 설계 과정에서 하나의 모듈이 변경되더라도 설계에 영향을 주지 않음

모듈의 응집력 (1/2)

◆ 모듈의 응집력이란?

- 모듈을 이루는 각 요소들의 서로 관련되어 있는 정도
- 강력한 응집력을 갖는 모듈을 만드는 것이 모듈 설계의 목표

◆ Myers의 응집력 정도 구분

1. 기능적 응집(Functional cohesion)	응집력 강함
2. 정보적 응집(Informational Cohesion)	↑ 중간 ↓
3. 교환적 응집(Communication cohesion)	
4. 절차적 응집(Procedural cohesion)	
5. 시간적 응집(Temporal cohesion)	↓ 응집력 약함
6. 논리적 응집(Logical cohesion)	
7. 우연적 응집(Coincidental cohesion)	

모듈의 응집력 (2/2)

◆ 응집력의 종류

- 기능적 응집(Functional cohesion)
 - 모듈이 잘 정의된 하나의 기능만을 수행할 때 기능적 응집도가 높아짐
- 교환적 응집(Communication cohesion)
 - 한 모듈 내에 2개 이상의 기능이 존재하고 단계별 순서에 의해서만 수행되는 경우
 - 각 기능은 동일한 입력 자료를 사용하면서도 서로 다른 출력을 생성함
- 절차적 응집(Procedural cohesion)
 - 모듈 안의 작업들이 큰 테두리 안에서 같은 작업에 속하고, 입출력을 공유하지 않지만 순서에 따라 수행될 필요가 있는 경우
- 시간적 응집(Temporal cohesion)
 - 프로그램의 초기화 모듈 같이 한 번만 수행되는 요소들이 포함된 형태
- 논리적 응집(Logical cohesion)
 - 비슷한 성격을 갖거나 특정 형태로 분류되는 처리 요소
- 우연적 응집(Coincidental cohesion)
 - 아무 관련 없는 처리 요소들로 모듈이 형성되는 경우

모듈의 결합도 (1/2)

◆ 의미

- 모듈간에 연결되어 상호 의존하는 정도
- 낮은 결합도를 갖는 모듈(Losely coupled)을 만드는 것이 모듈 설계의 목표

◆ 모듈간의 의존도

1. 자료 결합(Data coupling)
2. 구조 결합(Stamp coupling)
3. 제어 결합(Control coupling)
4. 공통 결합(Common coupling)
5. 내용 결합(Content coupling)

결합도 약함



결합도 강함

모듈의 결합도 (2/2)

◆ 결합도의 종류

- 자료 결합(data coupling)
 - 모듈 간의 인터페이스가 자료 요소로만 구성된 경우
 - 가장 이상적인 형태의 결합
- 구조 결합(structure coupling)
 - 모듈 간의 인터페이스로 배열이나 레코드 등의 자료 구조가 전달되는 경우
- 제어 결합(control coupling)
 - 한 모듈이 다른 모듈에게 제어 요소(function code, switch, tag 등)를 전달하는 경우
- 공통 결합(common coupling)
 - 여러 모듈이 공통 자료 영역을 사용하는 경우
- 내용 결합(content coupling)
 - 한 모듈이 다른 모듈의 일부분을 직접 참조 또는 수정하는 경우

04 실습

실습은 단위 테스트/TDD 연습 사이트 진행

◆ 사이트: Cyber Dojo

- <https://cyber-dojo.org/>
- TDD 수련 사이트였으나, 프로그래밍 연습 사이트로 발전
- 대표적인 개발언어와 테스트 프레임워크를 사용 가능
 - 개발 언어와 테스트 프레임워크는 지속적으로 변경

◆ 별도의 회원 가입은 필요 없음

◆ 사이트 내에서 컴파일/테스트 수행 가능

- 별도의 도구 설치나 설정 불필요
- 실습 파일은 다운로드 가능

◆ 주어진 주제를 구현하며 연습 가능

- 약 30개의 주제 선택 가능

첫 번째 실습: 간단한 할인 조건 계산

프로젝트 : Big Sale



1. 성별과 나이에 따라 할인을 계산
2. 여자에게만 할인 적용

◦ 할인율 공식

남자일 경우: 나이에 상관없이 할인 없음

여자의 경우

- 19세 미만은 30% 할인
- 19세 이상은 10% 할인

[설계 요건]

1. 입력값:
 1. 남자는 1, 여자는 2 (주민번호 뒷자리)
 2. 나이는 int
2. 할인율은 int로 반환 (30%는 30)

두 번째 실습: FizzBuzz

◆ 문제

- 1부터 100까지 출력하며,
- 3의 배수는 "Fizz" 출력
- 5의 배수는 "Buzz" 출력
- 3과 5의 공배수는?

◆ 완료 후 Download 받기

세 번째 실습

- ◆ 첫, 두 번째 실습 결과에 따라 난이도 결정!

04 임베디드 SW 적용을 위한 논의

#1. 어디에 적용할 것인가?

- ◆ TDD를 임베디드 SW 개발의 어느 부분에 적용하는 것이 적절한가?
 - 전체
 - BSW
 - ASW

- ◆ 레거시인 경우 어떻게 할 것인가?

#2. ISO 26262의 단위 테스트를 대체할 수 있는가?

- ◆ Automotive SW에 필연인 기능 안전 요건을 만족할 수 있는가?
 - ISO 26262의 요건을 준수하기 위한 의도적인 활동이어야 하는가?
 - ISO 26262에 얽매이지 않는 보다 나은 개발을 위한 활동이어야 하는가?

[참고] ISO 26262와 단위 테스트

◆ 어떤 테스트를 해야 하는가?

Table 7 — Methods for software unit verification

Methods		ASIL			
		A	B	C	D
1a	Walk-through ^a	++	+	0	0
1b	Pair-programming ^a	+	+	+	+
1c	Inspection ^a	+	++	++	++
1d	Semi-formal verification	+	+	++	++
1e	Formal verification	0	0	+	+
1f	Control flow analysis ^{b, c}	+	+	++	++
1g	Data flow analysis ^{b, c}	+	+	++	++
1h	Static code analysis ^d	++	++	++	++
1i	Static analyses based on abstract interpretation ^e	+	+	+	+
1j	Requirements-based test ^f	++	++	++	++
1k	Interface test ^g	++	++	++	++
1l	Fault injection test ^h	+	+	+	++
1m	Resource usage evaluation ⁱ	+	+	+	++
1n	Back-to-back comparison test between model and code, if applicable ^j	+	+	++	++

[참고] ISO 26262와 단위 테스트

◆ 어떻게 테스트 케이스를 도출해야 하는가?

Table 8 — Methods for deriving test cases for software unit testing

Methods		ASIL			
		A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes ^a	+	++	++	++
1c	Analysis of boundary values ^b	+	++	++	++
1d	Error guessing based on knowledge or experience ^c	+	+	+	+
^a Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.					
^b This method applies to interfaces, values approaching and crossing the boundaries and out of range values.					
^c Error guessing tests can be based on data collected through a “lessons learned” process and expert judgment.					

[참고] ISO 26262와 단위 테스트

- ◆ 소스코드의 어느 정도를 테스트해야 하는가?

Table 9 — Structural coverage metrics at the software unit level

Methods		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)	+	+	+	++

#3. 어떤 테스트를 사용할 것인가?

- ◆ 어떤 테스트 프레임워크를 사용할 것인가?
 - 전통적인 프레임워크: CppUTest, Unity
 - 요즘 프레임워크: Google Test
- ◆ 테스트 프레임워크의 무게에 대한 고민
 - 가볍다: Unity
 - 무겁다: CppUTest, Google Test
- ◆ 기존 테스트 도구와의 연계는?
 - 이미 VectorCAST를 사용하는 경우의 역할 구분

더 고민해야 하는 것

- ◆ 테스트 코드와 제품 코드의 분리
- ◆ 멀티 타켓팅
- ◆ 테스트 커버리지

05 마무리

마무리

- ◆ TDD가 모든 경우에 항상 옳은 것은 아님 (유용하나, 모든 경우에 딱 맞는 것은 아님)
 - SW 개발에 은총알은 없다.
 - TDD는 죽었다: <https://sangwook.github.io/2014/04/25/tdd-is-dead-long-live-testing.html>
 - TDD는 죽었는가?: <https://jinson.tistory.com/271>
- ◆ 기능 안전 SW 적용의 제약
 - TDD로 단위 테스트를 대체해도 되는가?
 - 기능적으로 적용이 안되는 것이 아니라, 결과를 인정 받을 수 있는가의 의미
 - ISO26262의 Tool Certification 문제, 커버리지 측정 문제
 - Tool Certification과 연계되는 경우, SW 개발자가 더 쉽게 TDD를 적용할 수 있는가?
 - IDE 따로, 테스트 도구 따로... 의 경우는 몰입도 깨지고, 진행이 느림



[참고] **지속적 통합과 테스트 자동화**

지속 통합(CI: Continuous Integration)

- ✓ 팀 구성원들이 자신이 한 일을 자주 통합하는 소프트웨어 개발 실천 방법
- ✓ 각 통합은 자동화된 빌드를 검증하여 최대한 빨리 통합 오류를 탐지
- ✓ 하루에 여러 번 통합 빌드를 수행하는 것

- 마틴 파울러

소프트웨어 통합 오류를 개발 초기부터 예방하는 것

통합: 빌드, 테스트, 정적분석, 배포를 포함

통합 지옥

Integration HELL



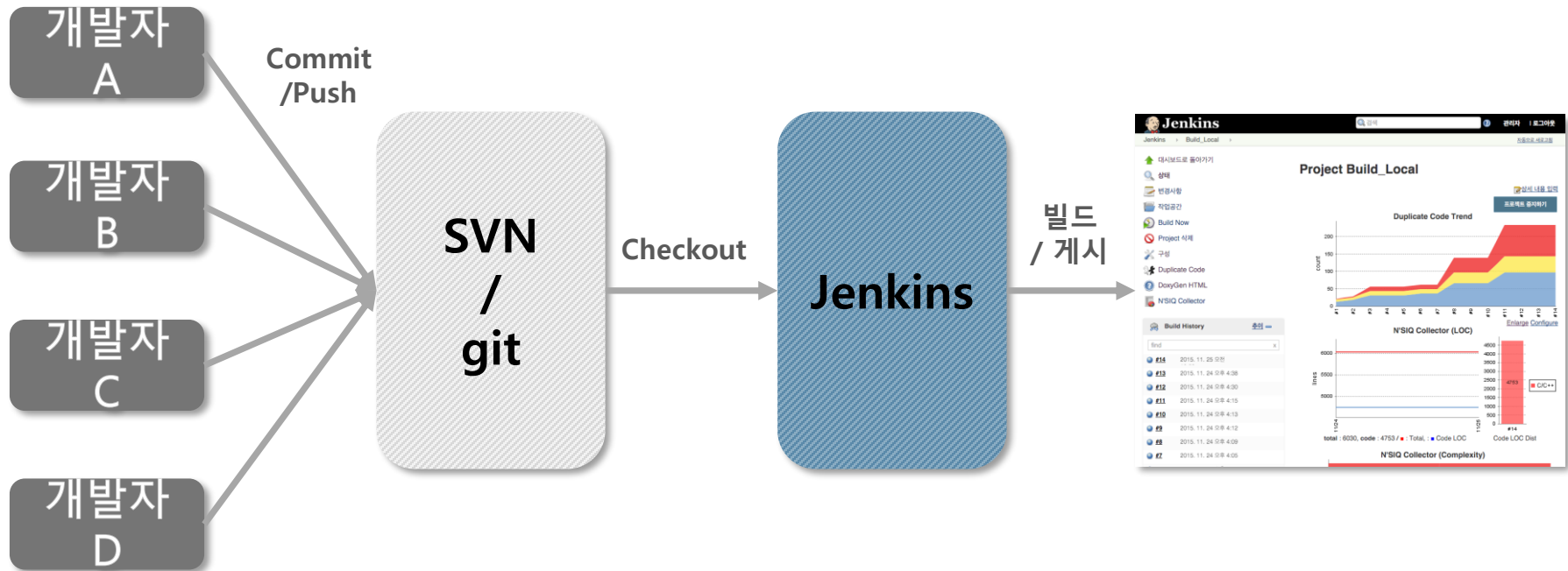
Jenkins

<http://www.jenkins-ci.org>

- ✓ 웹 기반 오픈소스 CI 도구
- ✓ 점유율 약 70%
- ✓ 1,300 개의 플러그인

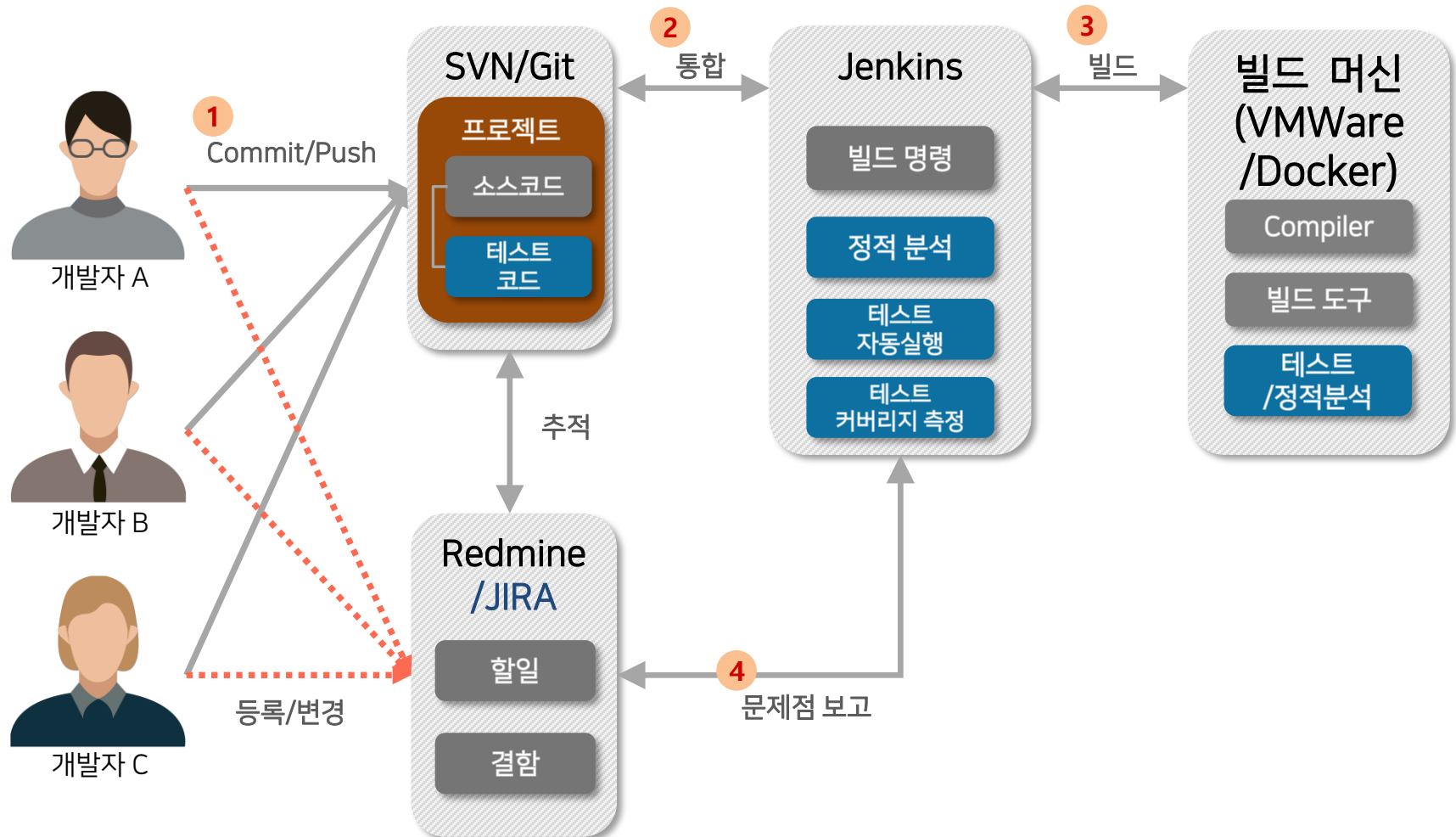
지속적 통합 - 동작 방식

- ① 버전관리 도구에서 최신 리비전을 체크아웃 받아
- ② 주어진 명령대로 빌드/테스트/정적분석하여
- ③ 결과를 게시/전달함

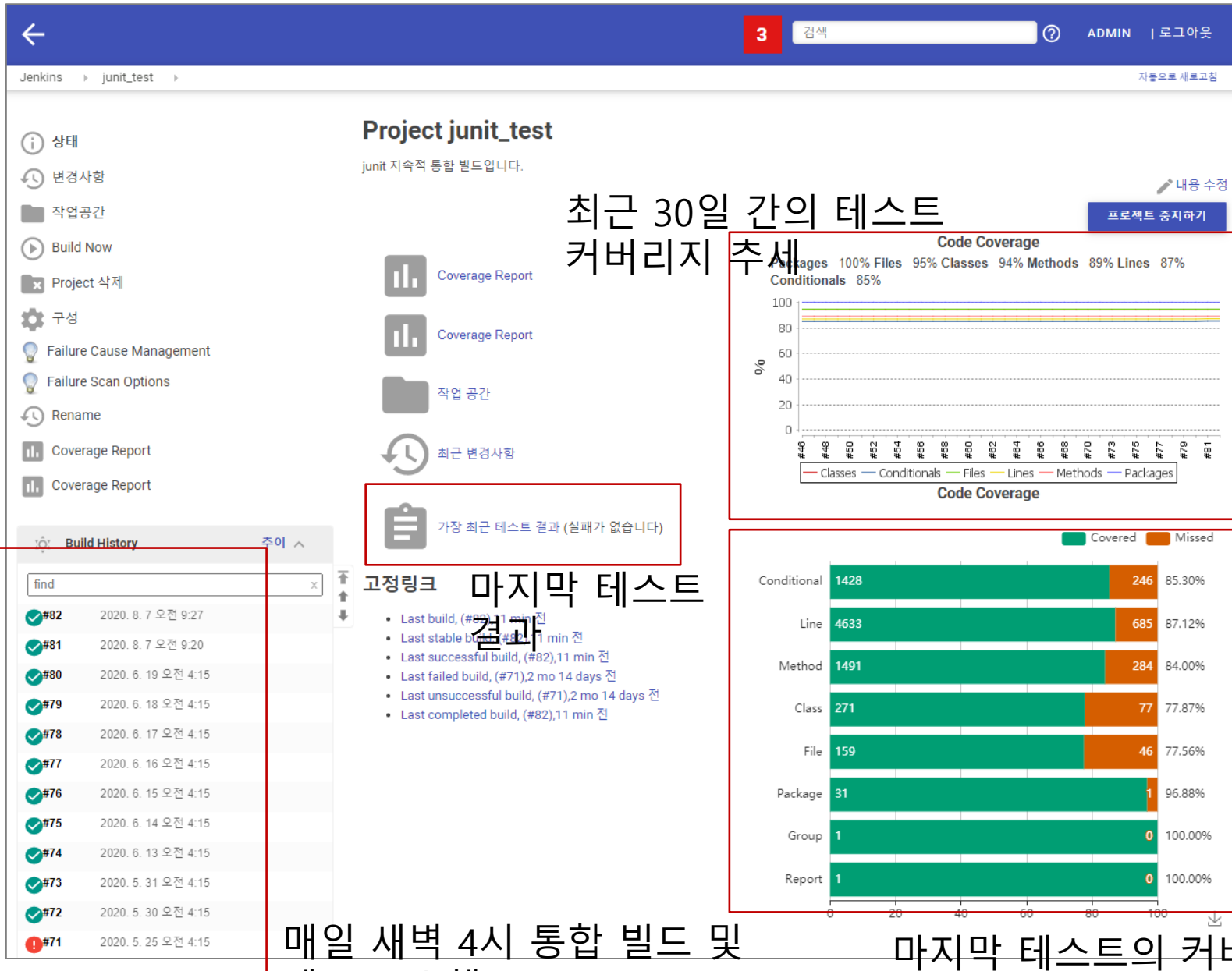


지속적 통합과 테스트 자동화 연동 구성도

지속적 통합 도구인 Jenkins를 사용하여 SW 개발 문제점(테스트 실패, 낮은 커버리지 등)을 조기에 발견하고 해결하는 것을 자동화 하기 위함 (2~4 단계가 도구에 의한 자동화)



각 프로젝트의 Jenkins 메인 화면



감사합니다.