# TDD

Test Driven Development

# 아리안5의 교훈 (1/2)

**[출처: 중앙일보 1996.06.06] 아리안5호 로켓 발사실패 따른 파장**

유럽의 「우주산업」이 큰 차질을 빚게 됐다.
유럽이 2000년대 세계 상업위성시장 석권을 위해 야심작으로내놓았던 위성 적재 로켓 아리안5호가 4일 **발사 1분도 채 안돼 공중폭발**했기 때문이다. 이번 발사는 위성 발사체의 기술개발이라는 측면과 함께 경제적이유에서 중요한 의미를 담고 있었다. 상업위성의 **중량화 추세**에 맞춰 이를 실어나를 **발사체**를 개발하는 한편 장기적으로는 **유럽도 인간을 우주로 보낼 수** 있느냐는 **가능성을 타진**하는 시험이었다.

* 참고) https://play-tv.kakao.com/embed/player/cliplink/v9ad5OWQlPumpnFmQQFncFW@my?service=daum_tistory 폭발 영상

https://play-tv.kakao.com/embed/player/cliplink/vaf662uYJgrYVyVrggVyg3y@my?service=daum_tistory 인터뷰 영상

- 직접적인 원인 : 숫자 전환상의 버그, overflow handler -> 시스템 shut-down
- 근본적인 원인 : **S/W 개발 관행(아리안4 재사용)에 따른 System Validation의 문제**

- System Validation 실패의 징후
  - 아리안5에서는 필요하지 않은 부분까지 재사용되었고, **그 부분**이 **실행** (**요구사항, 테스트도 없**었음)
  - 아리안4에서는 overflow가 발생불가 -> 아리안5와는 다른 상황
  - simulator를 통해서만 검증.(Simulator에는 그 기능이 포함되지 않았고, 직접 문제된 코드가 **검증될 기회가 없**었음)
  - 엄격한 설계리뷰와 코드리뷰를 거쳤음에도 불구하고 발견하지 못함.
  - 시스템 개발 동안의 프로세스 기록이 완전하지 않아서, 리뷰에서 뭐가 수행되었고 빠졌는지를 완전히 알지 못함

# 아리안5의 교훈 (2/2)

- **교훈** *(in Hard-Realtime System)*
  - 꼭 필요한 S/W가 아닌 것은 실행 시키지 않도록 해야 하다.
  - 시스템이 해서는 안되는 것에 대해 테스팅도 고려되어야 한다.
  - Shut-down하는 내용의 default exception handler 는 사용하지 않아야 한다.
  - Simulator보다는 실제시스템으로 test 해야 한다.
  - **매우 엄격한 리뷰 프로세스를 가지는 것이 중요하다.**
    - *(모든 원칙들이 리뷰에 포함되었더라면, 아리안5의 수평속도가 아리안의 속도와 다르다는 사실을 알 수 있었을 지도 모른다.)*

문제가 된 코드는 Document, Test, Review 대상에 **포함**되지 않았고,
나름 엄격한 Test process, review process 가졌으나, 발견하지 못했음
보다 엄격한 review process 를 거쳤다면(**review를 보다 잘 했다면**),..

- 실패분석보고서 14 RECOMMENDATIONS 중 *(Review 부분:4, Test 부분:8, 원천기술 부분:2)*
  - **R4** Organize, for each item of equipment incorporating software, a specific software qualification review.
  - **R5** Review all flight software (including embedded software), and ...
  - **R9** Include external (to the project) participants when reviewing specifications, code and justification documents.
  - **R11** Review the test coverage of existing equipment and extend it where it is deemed necessary.

# Testing

# 코드를 수정하는 방법



- 변경 후 기도하기(Edit and Pray)
  - 신중한 검토 -> 변경 대상 이해 -> 변경 -> 정상 동작 확인
    -> 의도하지 않은 변경 여부 확인(Side effect)
  - 전문적인 방법처럼 보이지만, 신중 ≠ 안정

- 보호 후 수정하기(Cover and Modify)
  - 테스트 루틴으로 코드를 보호
  - 문제 발생시 빠르게 찾을 수 있음
  - **테스트 루틴 = 피드백**

# Testing 이란 무엇인가?



- Testing 은 S/W 의 품질을 조사하는 과정

  Software testing is an investigation conducted to
  Provide stakeholders with information about the
  Quality of the product or service under test.

  출처 : Exploratory Testing, Cem Kaner, Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, Nov 2006

- Customer 가 진행하는 것? Closed Beta Test
- QA 가 진행하는 것? Regression Test
- 개발자가 수행하는 것? Integration Test? Unit test?

# Testing 의 종류



## Functional Testing

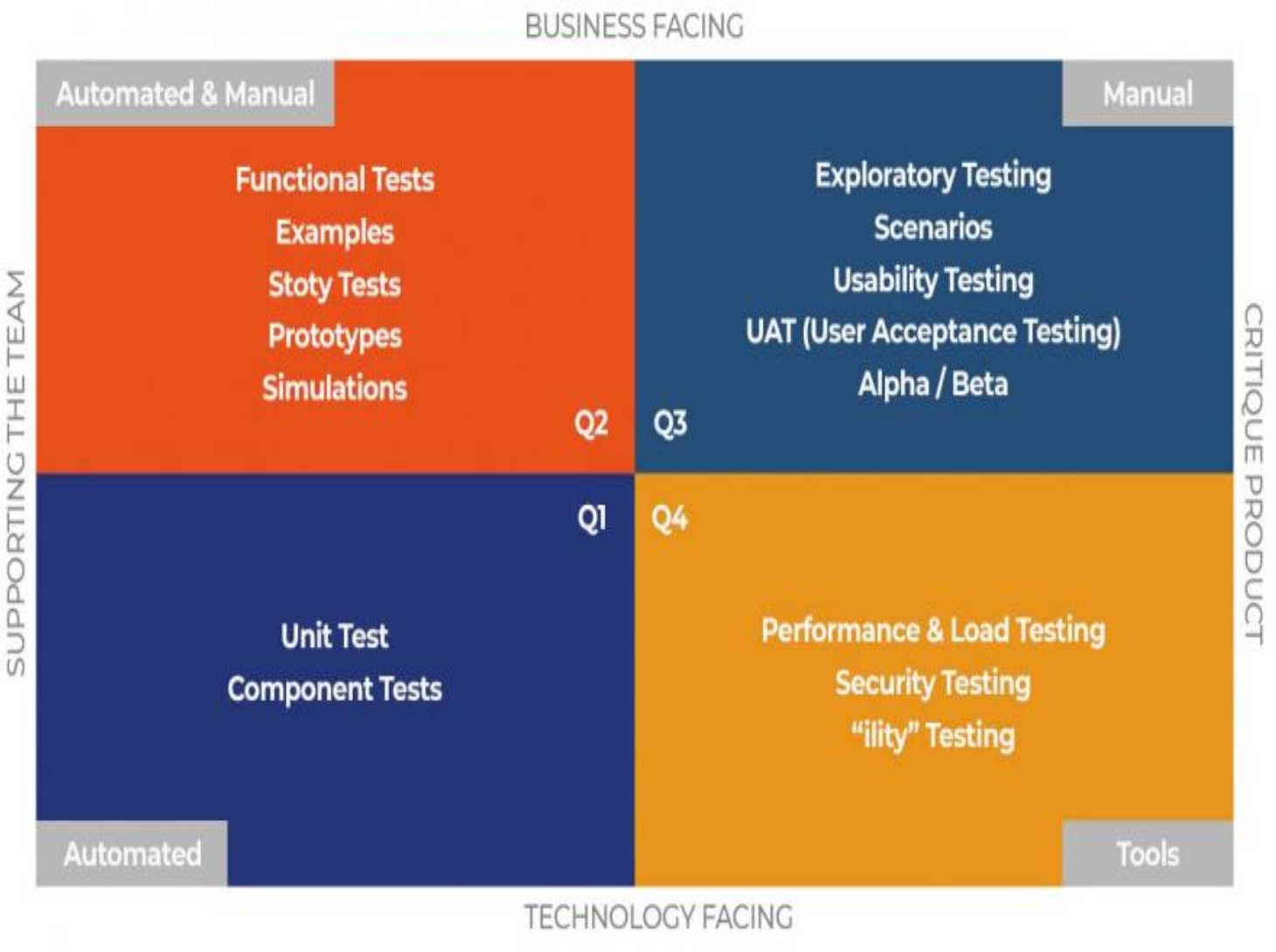
- Unit Testing
- Integration Testing
- System Testing
- Smoke Testing
- Sanity Testing
- Interface Testing
- Regression Testing
- Beta/Acceptance Testing

## Non-Functional Testing

- Performance Testing
- Load Testing
- Stress Testing
- Volume Testing
- Security Testing
- Compatibility Testing
- Install Testing
- Recovery Testing
- Reliability Testing
- Usability Testing
- Compliance Testing
- Localization Testing

# Testing의 4 영역



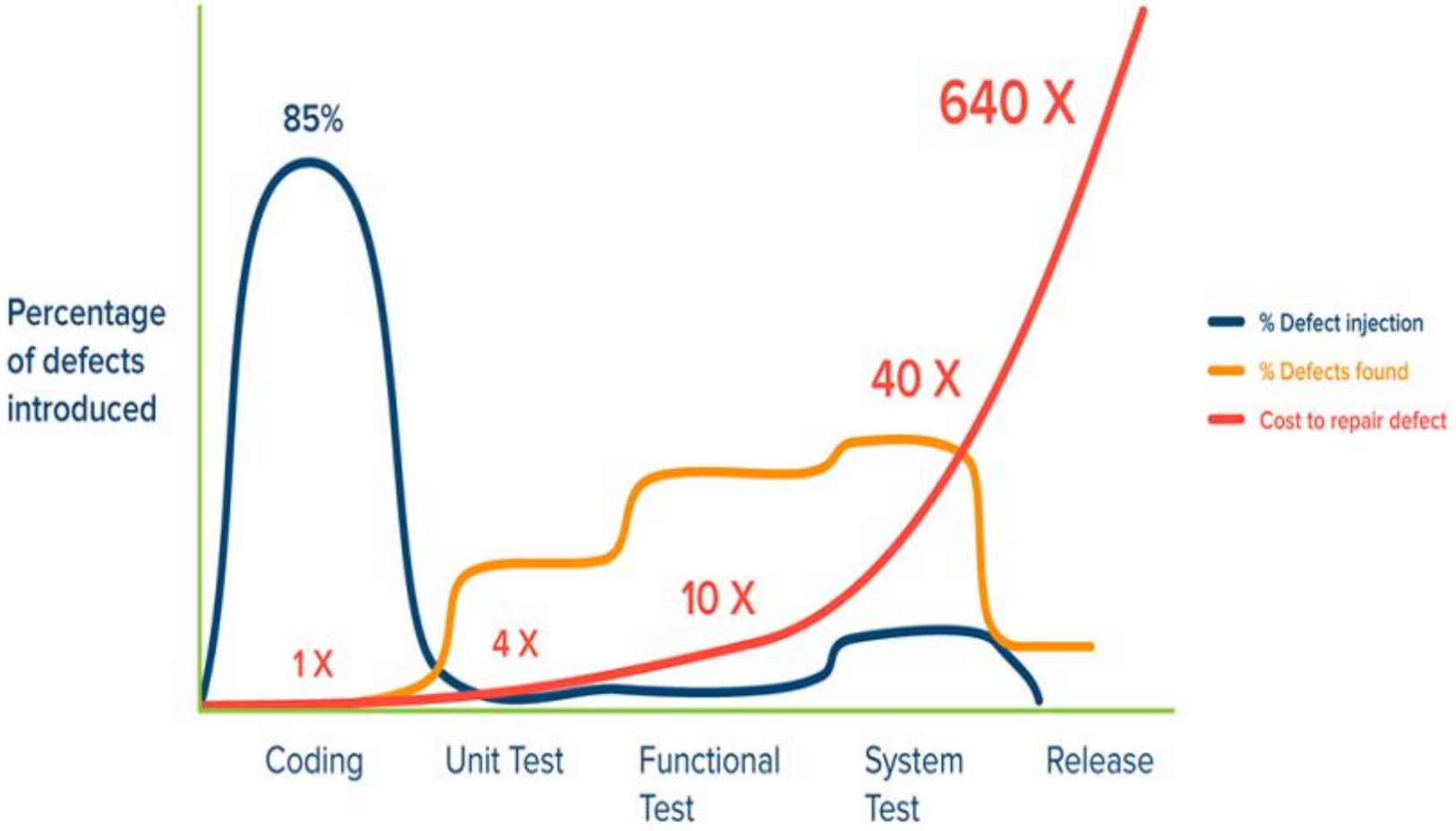



- **Q1**: Test Automation to improve the code.
- **Q2**: Test Automation + Manual to improve business outcomes.
- **Q3**: Manual tests to provide detailed feedback on product performance, functionality, and user experience. (often it is rechecking things done in Q1 and Q2)
- **Q4**: Tools to ensure security and compatibility.

출처 : https://blog.qatestlab.com/2020/06/04/agile-testing-practices/

# Cost of Defect






Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

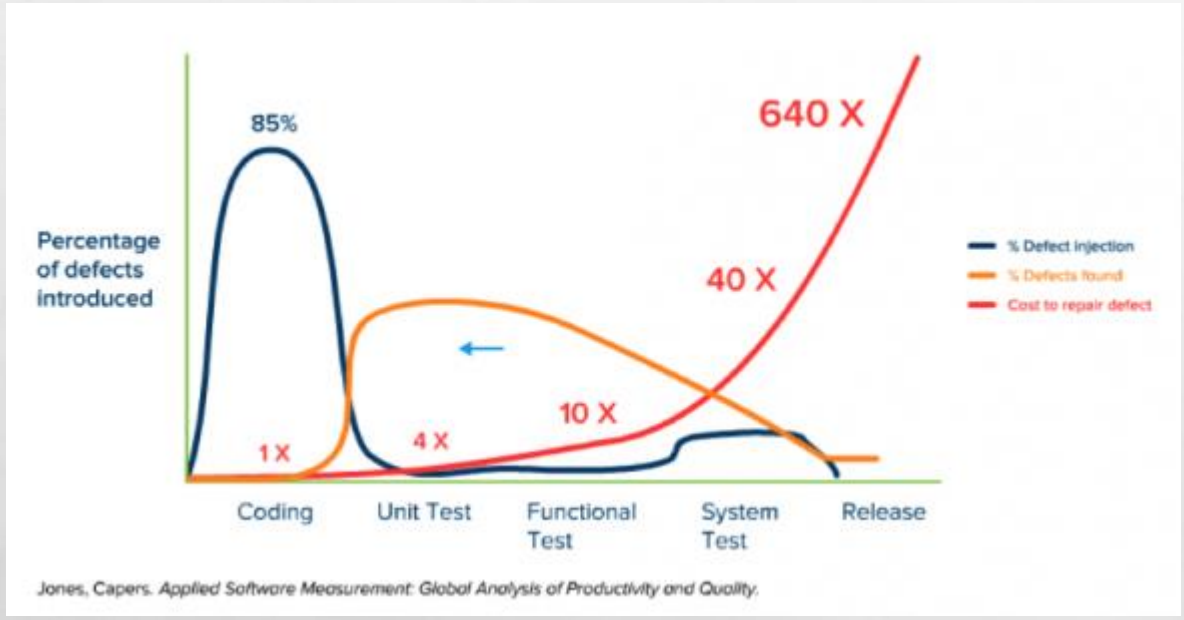
출처 : https://www.stickyminds.com/article/shift-left-approach-software-testing

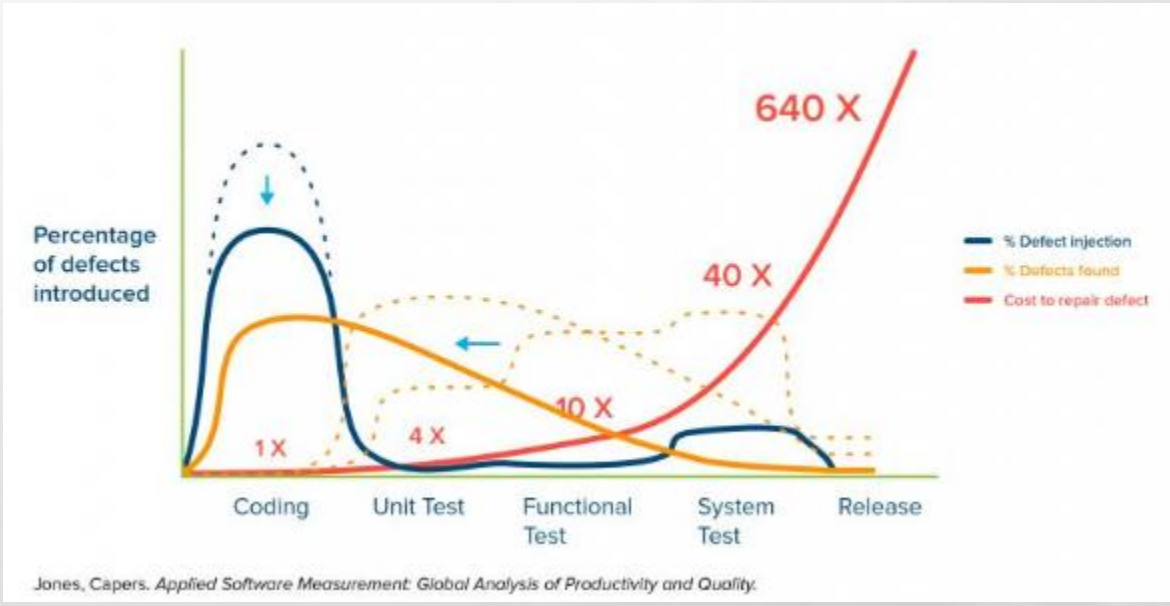
- 결함 투입 시점
  : 코딩 단계
- 결함 발견 시점
  : Test 단계
- 결함 수정 비용
  : 개발 후반부로 갈수록 급격히 증가

# Cost of Defect



- 테스트를 개발 초기에 시행하여 defect 를 찾아낼수록 defect 비용은 감소한다.




출처 : https://www.stickyminds.com/article/shift-left-approach-software-testing

# 코드 수정의 어려움



- 코드를 수정하는 시간이 오래 걸리는 이유
  - 테스트 지연 시간
  - 오류의 위치 파악
  - 컴파일 등 환경 영향

- 테스트 지연 시간
  - Test 가 늦다는 것이 최대의 단점

  => Defect 은 최대한 빨리 발견되어야 한다. – 빠른 feedback

  코드를 수정하자마자, 누군가가 "즉시" 코드로 인해 문제가 생기지 않는 지 확인해서 알려주면 좋겠다.
- 오류의 위치 파악
  - 코드 분석 시간 : 코드의 양에 Exponential 비례
  - Regression Test : 위치 파악이 어렵다.

  => 오류의 위치 파악이 쉬워야 한다.

  원인을 쉽게 파악할 수 있는 과정을 가지는 테스트로 이루어져야 한다.
- 컴파일 등 환경 영향

**defect 의 발생이 가장 많은 시점에서 Test 가 잘 이루어져야 한다.**

# 우리는 시간이 없다



- 스트레스(압박) -> 적은 테스트 -> 에러 증가 -> 스트레스(압박)의 악순환

- **테스트는 자동화되어야 한다.**

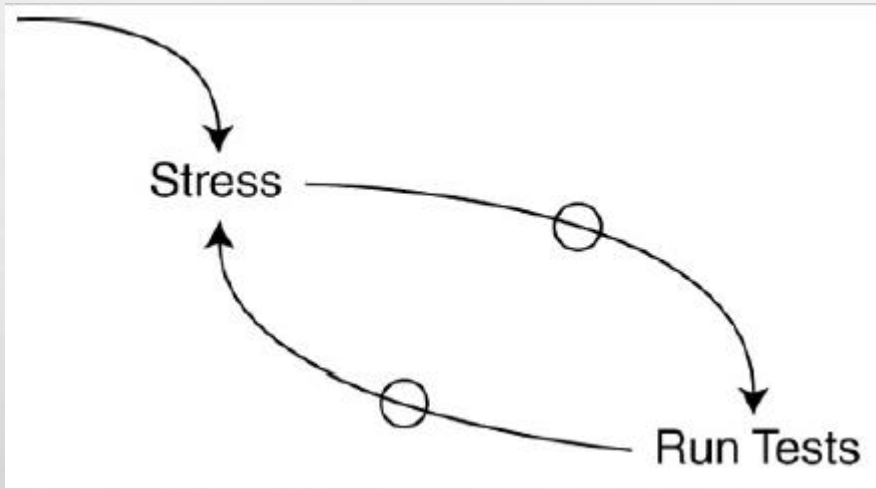
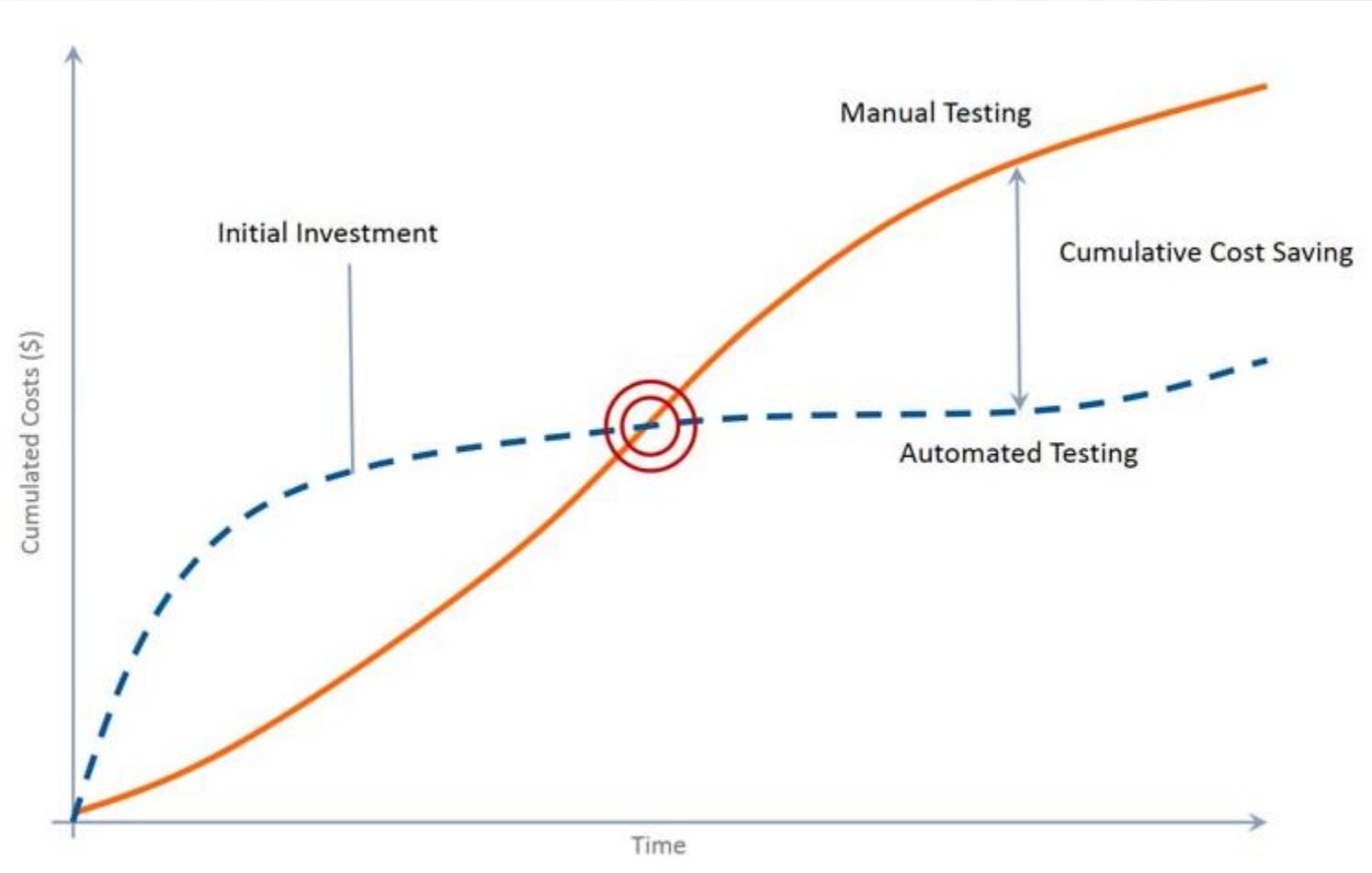



그림 : The “no time for testing” death spiral
출처 : Quality Software Management, Gerry Weinberg

# 테스트 자동화



- 테스트는 자동화되어야 한다.
- 테스트는 빨라야 한다.
- 테스트는 어떤 경우에도 수행되어야 한다.
- 수동 테스트를 하는 시간
  .VS.
  자동 테스트를 작성하고 테스트 하는 시간

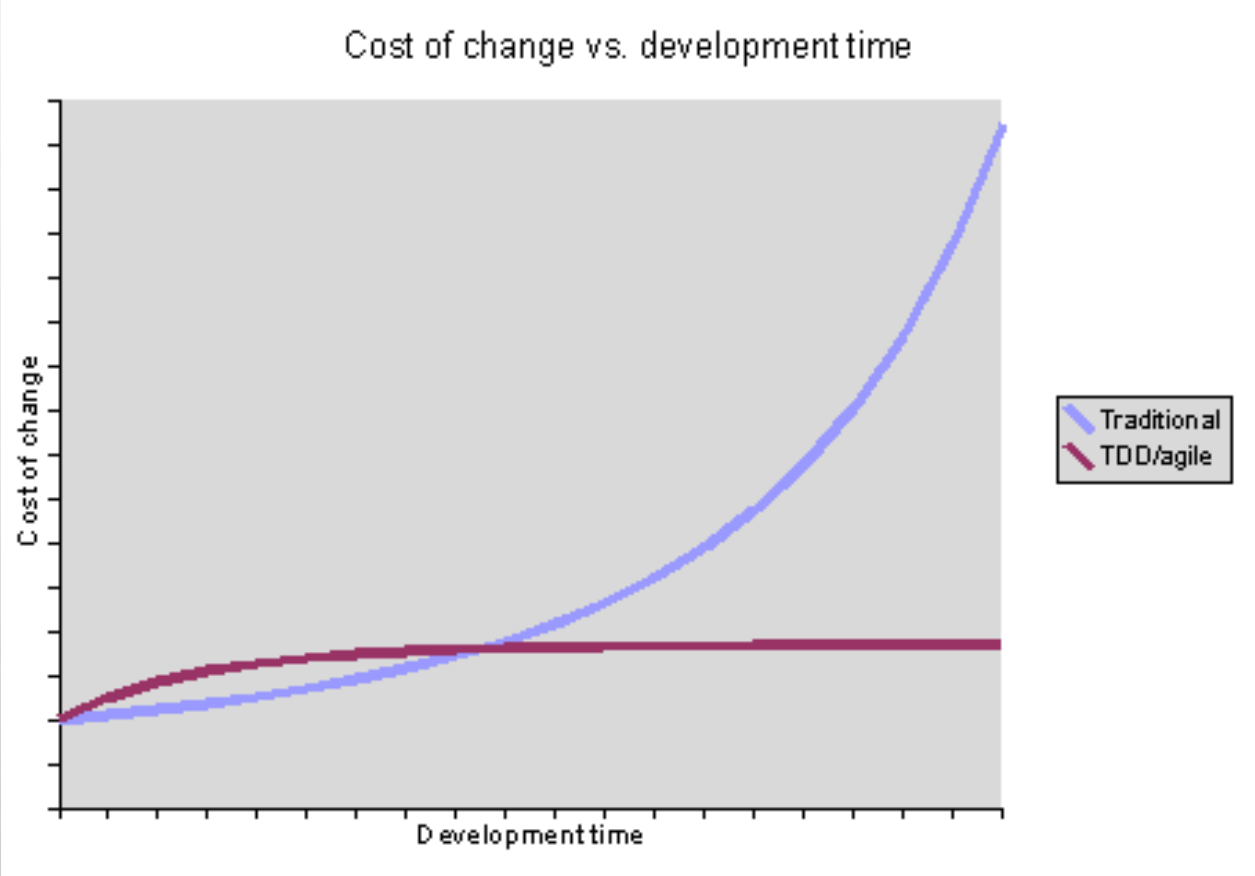



출처 : https://blog.qatestlab.com/2018/06/12/when-automate-testing/

# 테스트 자동화



- TDD : 15~35% 의 개발시간 증가, 40~90% 의 결함률 감소
  - 출처 : Microsoft Exploding Software-Engineering Myths By Janie Chang




출처 : http://gamesfromwithin.com/backwards-is-forward-making-better-games-with-test-driven-development

# 테스트 자동화



- 자동화 테스트 적용에 대한 높은 장벽
- 잘 적용된 자동화 테스트 안정화 -> Production 개발에 대한 노력 감소
- 테스트 작성이 어렵고 유지보수가 높은 자동화 테스트는 지속적인 노력을 필요로 함.

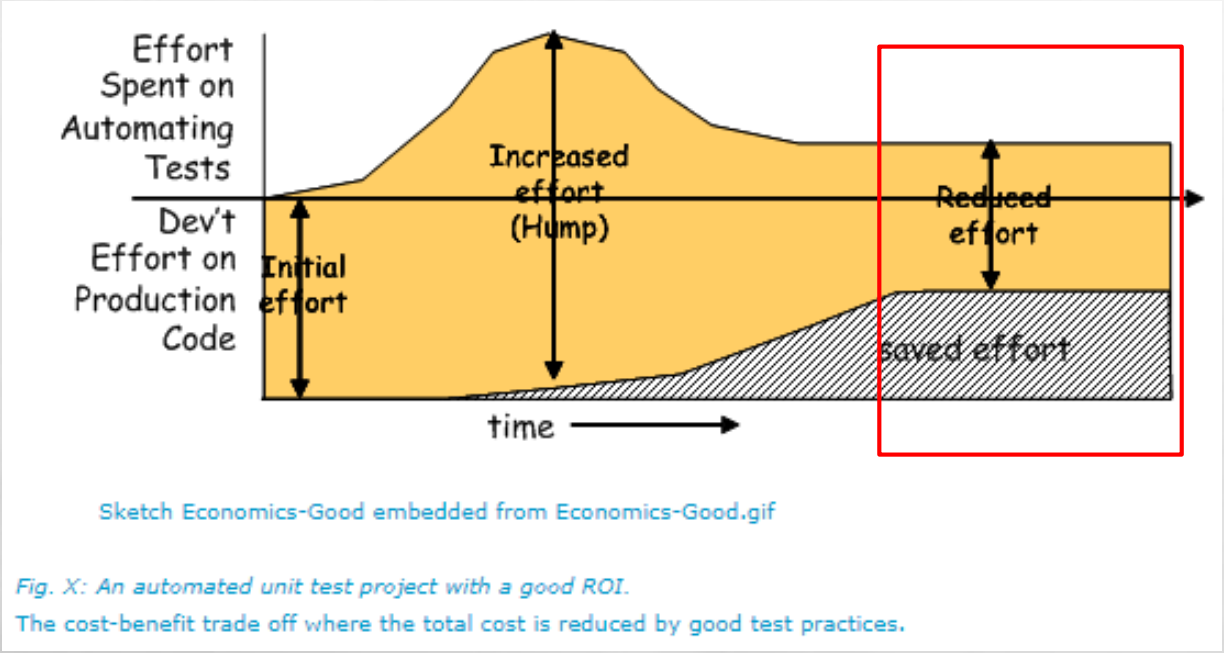



*Fig. X: An automated unit test project with a good ROI.*
The cost-benefit trade off where the total cost is reduced by good test practices.

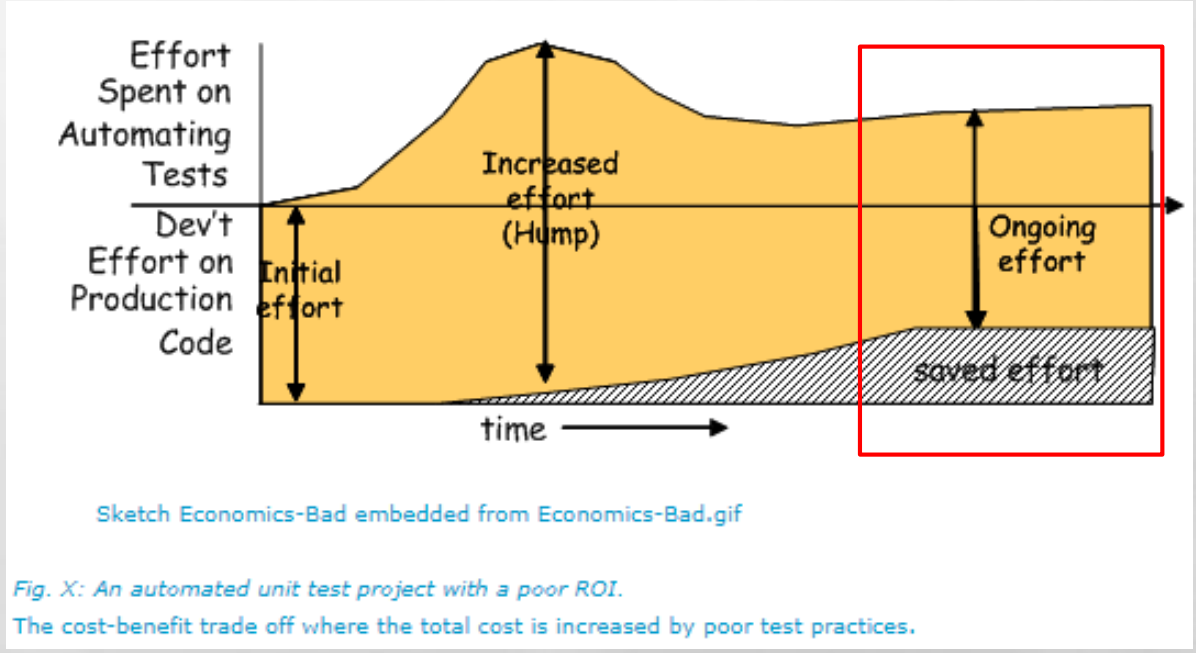



*Fig. X: An automated unit test project with a poor ROI.*
The cost-benefit trade off where the total cost is increased by poor test practices.

출처 : http://xunitpatterns.com/Goals%20of%20Test%20Automation.html

# Test Automation Manifesto



***Automated test should be:***

***Concise*** *– As simple as possible and no simpler.*

***Self Checking*** *– Test reports its own results ; needs no human interpretation.*

***Repeatable*** *– Test can be run many times in a row*
*without human intervention.*

***Robust*** *– Test produces same result now and forever.*

*Tests are not affected by changes in the external environment.*

***Sufficient*** *– Tests verify all the requirements of the software being tested.*

# Test Automation Manifesto



***Necessary*** – *Everything in each test contributes to the specification of desired behavior.*

***Clear*** – *Every statement is easy to understand.*

***Efficient*** – *Tests run in a reasonable amount of time.*

***Specific*** – *Each test failure points to a specific piece of broken functionality; unit test failures provide "defect triangulation".*

***Independent*** – *Each test can be run by itself or in a suite with an arbitrary set of other tests in any order.*

***Maintainable*** – *Tests should be easy to understand and modify and extend.*

***Traceable*** – *To and from the code it tests and to and from the requirements.*

*출처 : xUnitPatterns.com*

# Test Framework



- google test
  - google test C/C++ 프로그래밍 언어의 unit test framework
- JUnit
  - Java 프로그래밍 언어용 unit test framework
- NUnit
  - .Net 을 위한 unit test framework
- CppUTest
  - CppUTest is a C/C++ unit test framework based on xUnit
- KUnit : Kernel unit test framework

# 테스트 코드의 작성



- Should Check external behavior

- Unit test
  - Test Framework 를 활용
  - Assert 등으로 확인
  - 코드에 대한 이해도 필요
  - 코드 변경시 관련 Test 만 수정/삭제
  - 코드 이해도를 바탕으로 커버리지가 높음

- Golden master testing
  - 확인이 필요한 출력을 master file 로 저장
  - 코드 변경이 일어날 때마다 실행되는 출력을 저장된 master file 과 비교
  - 코드가 복잡한 경우에 사용
  - 코드가 변경될 경우 재생성 필요
  - 코드 커버리지가 낮을 수 있음

# TDD
# - Test Driven Development

***Test the program before you write it.***

-TDD by Example, Kent Beck

# TDD 란?



- TDD 의 기원
  - Agile 개발 방법론의 하나인 eXtreme Programming 의 실천 방식 중 하나.

- TDD의 정의
  - 코드를 작성하기 전에 테스트 코드를 먼저 만드는 것
  - 설계 문서를 만들어 생각하고 코딩하여 눈으로 확인하던 개발 방식
    => 예상 결과를 코드로 표현해 놓고 해당 코드가 자동으로 판단하게 하는 개발 방식

- TDD의 목표
  - Clean code that works. – Ron Jeffries
  - 소프트웨어의 품질 : 유지보수의 편의성, 가독성, 비용, 안정성 고려

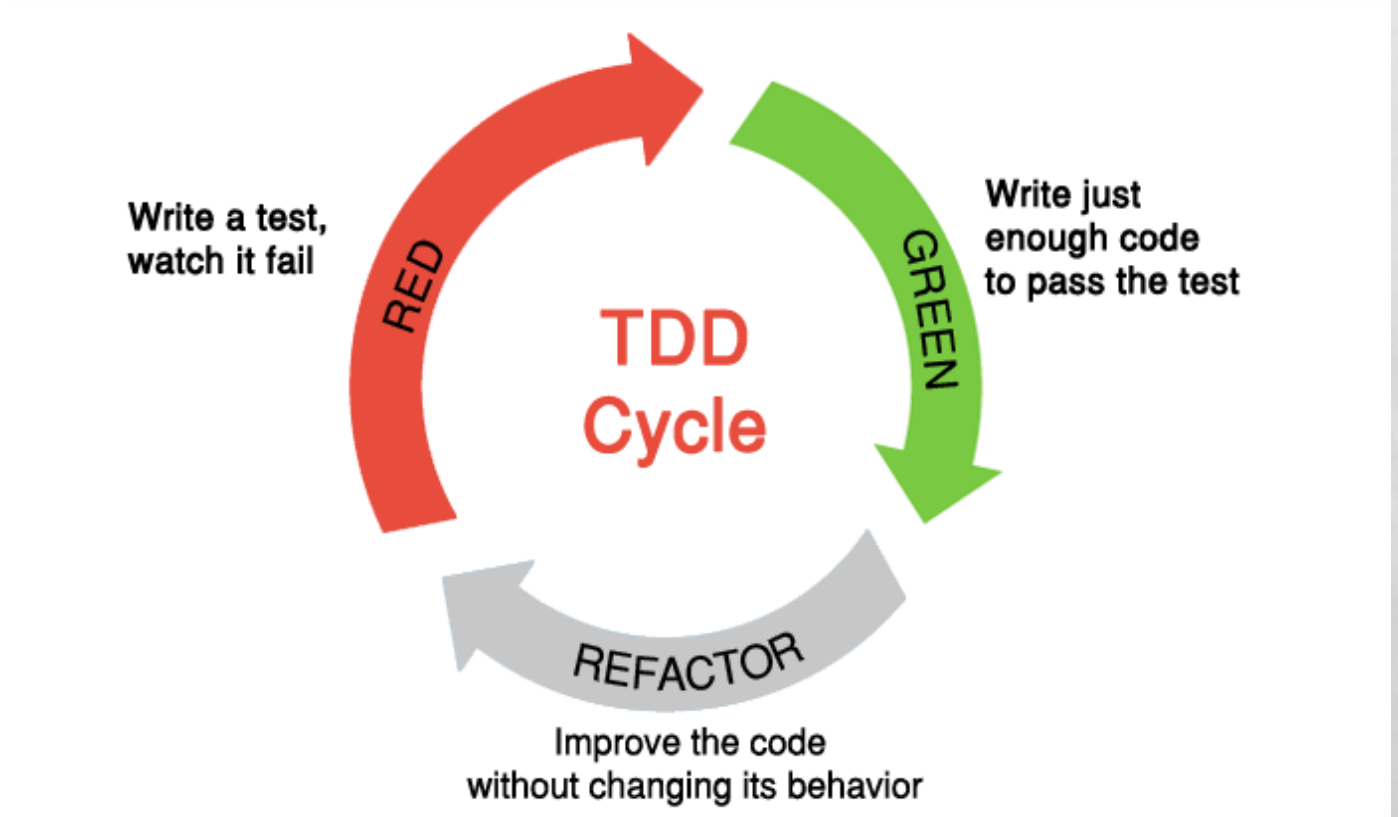
# TDD 개발 방식



- Unit Test
  - 개발자가 수행하는 테스트
  - 메소드 단위 테스트 vs 사용자 측면의 기능 테스트

- Unit test ≠ TDD

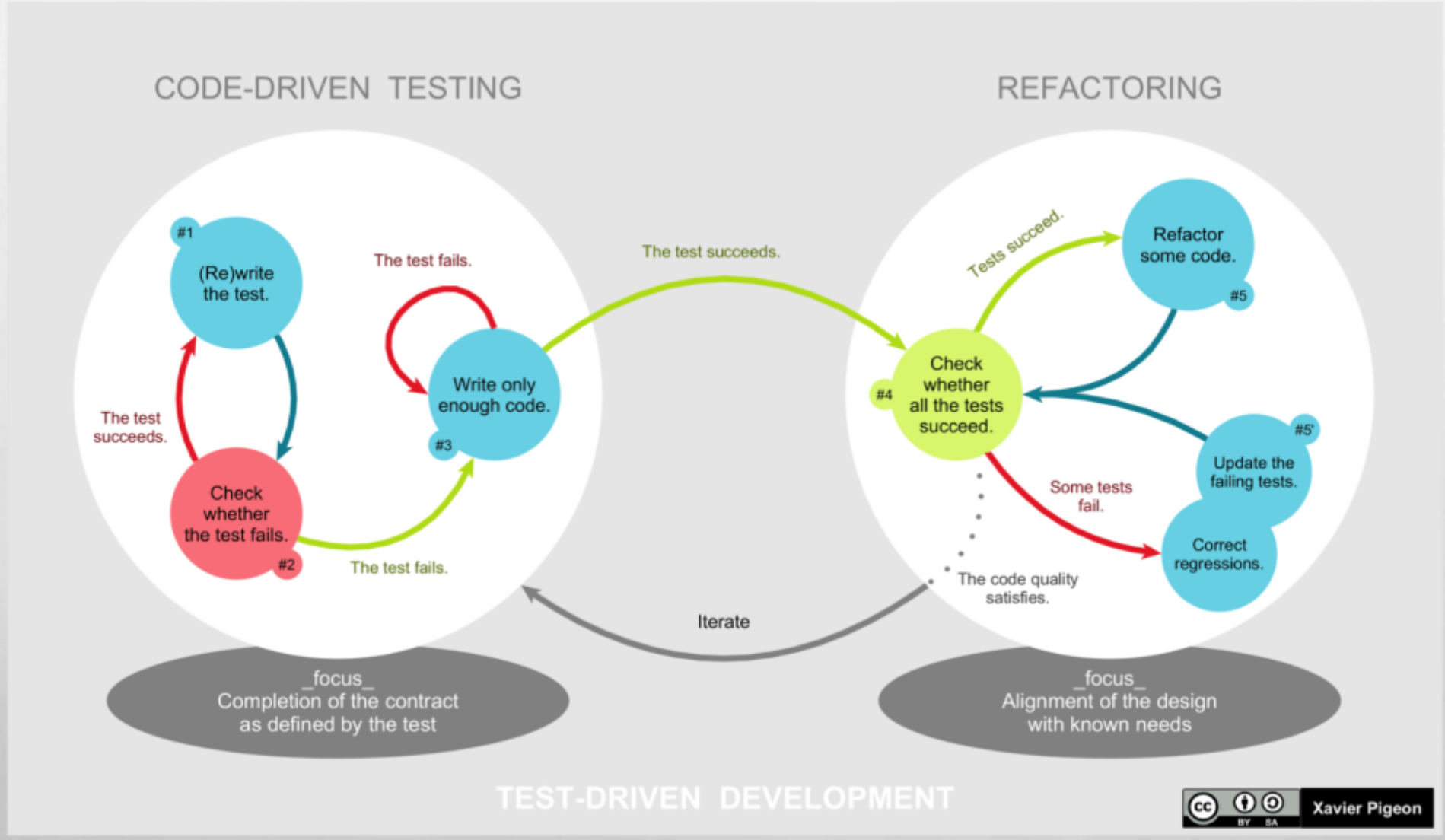
- TDD Process
  - Ask – Response – Refine 의 반복
  - Red -> Green -> Refactor 의 짧은 주기를 반복하는 Test First 개발 방식



# TDD 개발 방식

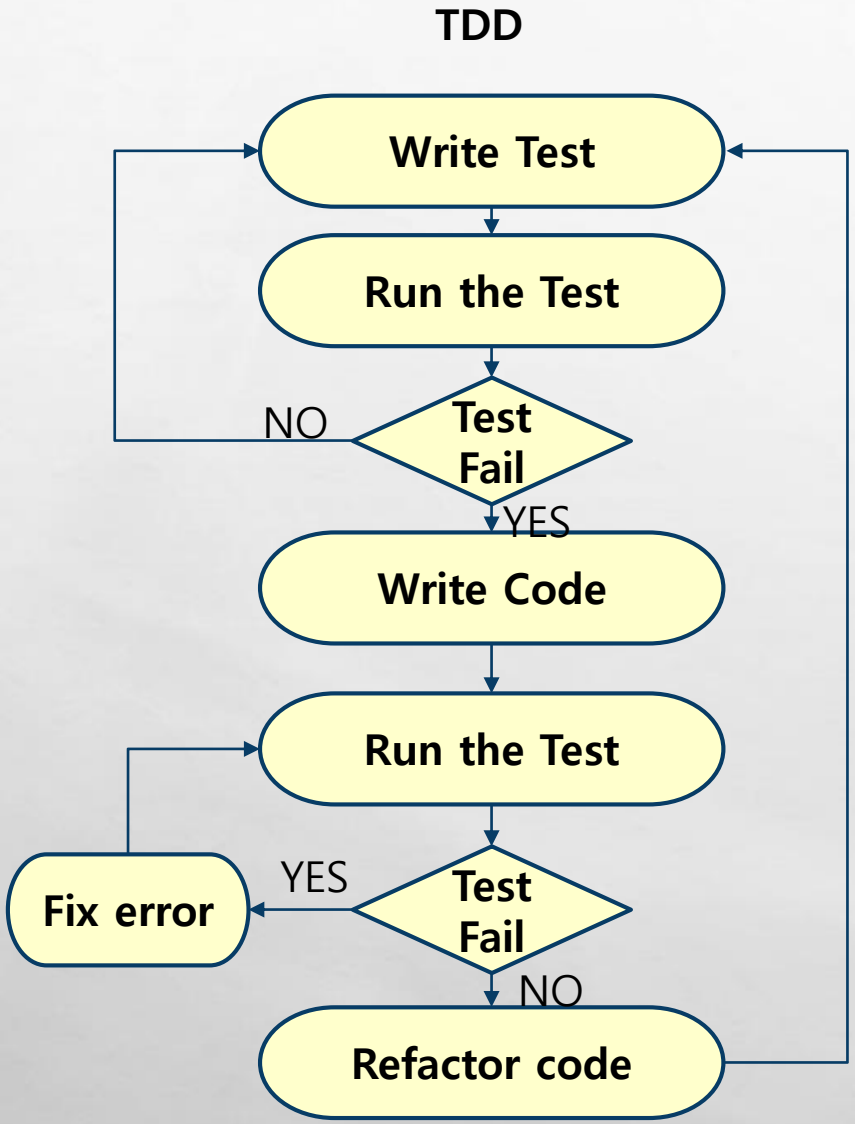


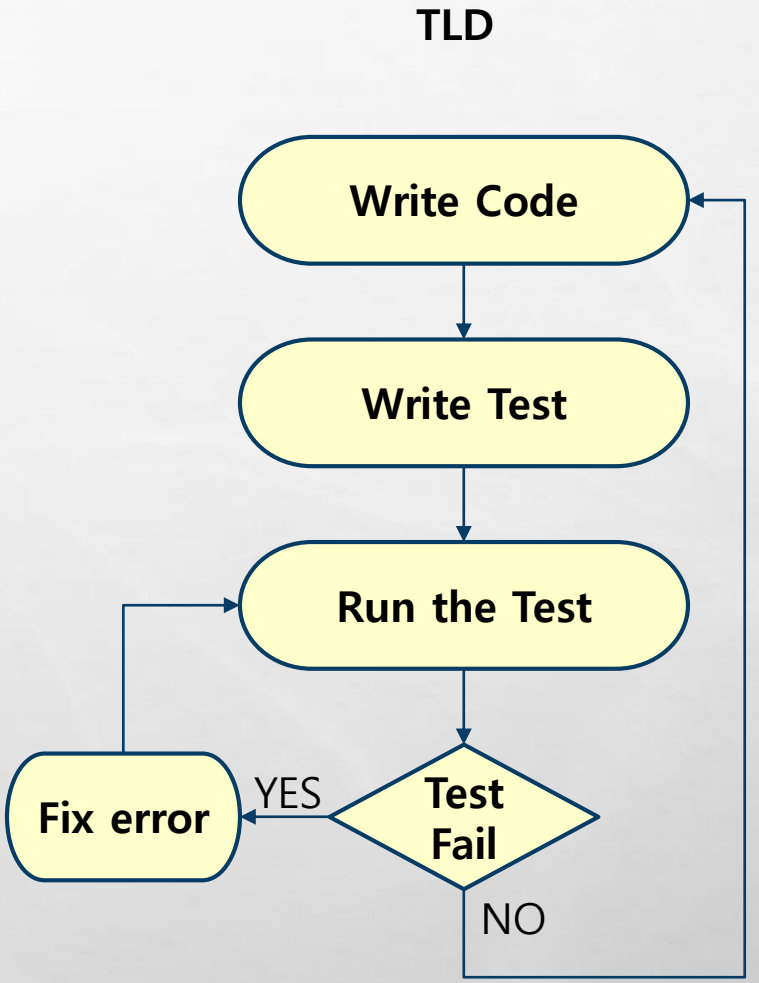
- 출처 : https://en.wikipedia.org/wiki/Test-driven_development



# TDD vs TLD(Test Last Development)





# Test First ? – Legacy code



- 기존 코드(Legacy code)에 Test 를 만드는 것은 매우 어렵다.
- 기존 코드에 만들어지는 Test 는 Business logic 을 모두 커버하기 어렵다.
- Test 가 신규 Business logic 보다 먼저 만들어 져야 한다.
- Legacy code 를 수정하는 경우에도 **Failed Test case** 가 먼저 만들어 져야 한다.

# TDD 의 장점



- 개발의 방향을 잃지 않게 해준다.
  - TDD 의 테스트케이스들이 개발의 나침반 역할.
- Good architecture로의 변화
  - TDD는 단위 테스트를 기반으로 하므로, 작은 단위의 설계를 만들어낸다.
  - 테스트케이스를 작성하고 통과시키려면 설계가 더 명확해져야 한다.
  - 작은 단위의 설계는 모듈의 동작을 보다 더 정확하게 설계되고 테스트하게 된다.
  - 이러한 디자인으로 인해 다른 모듈에 대한 의존성이 적게 작성되고, SW 의 복잡도가 낮아진다.
- 품질 높은 소프트웨어
  - 이해하기 쉽고 모듈화된 테스트 코드
  - SW 개발 비용 부담 감소
- 자동화된 단위 테스트 케이스를 가진다.
  - 자동화된 단위테스트 케이스는 현 시스템을 증명하기도 하지만,
    기능을 추가하거나 수정하게 됐을 때 테스트 부담을 줄여준다.

# TDD 의 장점



- 테스트 근거 및 문서화
  - TDD 로 작성된 모듈은 개발 종료 후, 테스트 코드가 남는다.
  - 테스트 케이스를 잘 작성하면 개발문서, 테스트 문서가 된다.
- 짧은 성공 주기와 성취감.
  - TDD 는 매 주기를 짧게 설정하도록 권장
  - 짧은 주기의 성공 경험으로 성취감 증대.
- 리팩토링에 대한 확신
- 휴먼 에러에 대한 방어
- 객체 지향 설계로의 사고 전환

# TDD 단점



- 테스트 코드의 유지보수 노력
- 테스트 코드 작성의 어려움
- 좋은 테스트 코드 작성은 더 어려움
- 단기간 내 개발 속도 저하(Slow down development in short time)
- 테스트가 없는 기존 코드에 적용하기 어려움

# TDD 유의사항



- 테스트 메소드 이름 : 의도를 전달할 수 있는 이름
- 중복된 테스트 케이스, 더 이상 동작하지 않는 테스트 케이스 제거
- 모든 상황에 대한 테스트케이스는 불필요
- 하나의 테스트케이스는 하나의 항목만 테스트.
- 최대한 독립적인 테스트 케이스 작성
  - 의존성 : 테스트 케이스가 작성되지 않은 모듈, DB, 외부 인터페이스, IO, Network

- 테스트 코드의 리팩토링은 준비-실행-검증-해체 단계(또는 준비-행동-단언 AAA 구조) 간의 구별이 보존되어야 한다.

- 테스트 커버리지 100% 의 비용은 테스팅 실력과 테스트가 용이하고 복잡성이 낮으며 결합이 느슨한 코드를 작성하는 데 달려 있다.
  이 비용에 비해 돌아오는 품질 보상과의 트레이드 오프는 문제 도메인에 따라 달라진다.

# TDD 세 가지 법칙



1. 실패하는 단위 테스트를 작성할 때까지 실제 코드를 작성하지 않는다.
2. 컴파일은 실패하지 않으면서 실행이 실패하는 정도로만 단위 테스트를 작성한다.
3. 현재 실패하는 테스트를 통과할 정도로만 실제 코드를 작성한다.

- 출처 : Clean Code, Robert C Martin

# TDD 에 대한 전문가 의견



"코드의 미래에 대해 고려하지 않음으로 인해, 코드가 더 뛰어난 적응성을 가질 수 있게 한다.
비록 발생하지 않은(혹은 아직 발생하지 않은) 변주 종류는 잘 표현할지 못할지라도,
발생하는 변주 종류는 잘 표현하게 해준다."

"TDD는 시간이 지남에 따라 (개발자의) 코드에 대한 자신감을 점점 더 쌓아가게 해준다.
시스템 행위에 대한 자신감을 더 많이 얻게 된다.
프로젝트를 시작하고 시간이 지난 후에 더 좋은 느낌을 갖게 하는데 도움을 준다."

"어떤 판단을 테스트에 담아 낼 수 있을 때, 설계논의는 훨씬 더 흥미로워진다.
우선 시스템이 이런 식으로 동작해야 하는지 저런 식으로 동작해야 하는지 논의할 수 있다.
일단 올바른 행위에 대해 결정을 내린 후에, 그 행위를 얻어낼 수 있는 최상의 방법에 대해 논의할 수 있다."

* 참조 : 켄트 백, TDD By Example

# TDD 에 대한 질문



- Legacy code에 Test 를 반드시 만들어야 하는가?
- Code Review 와 Test 는 어떻게 연관되어야 하는가?
- TDD 는 반드시 필요한가?

# TDD 는 반드시 필요한 것인가?



- TDD 에 관한 질문

- 생각해 볼 만한 사항
  - Is TDD DEAD?
    - https://martinfowler.com/articles/is-tdd-dead/
    - https://junho85.pe.kr/975
  - TDD is the best thing that has happened to software design
    - https://www.thoughtworks.com/insights/blog/test-driven-development-best-thing-has-happened-software-design

# 참고 문헌




- http://xunitpatterns.com/
- https://www.stickyminds.com/article/shift-left-approach-software-testing
- https://blog.qatestlab.com/2018/06/12/when-automate-testing/
- https://en.wikipedia.org/wiki/Test-driven_development
- http://gamesfromwithin.com/backwards-is-forward-making-better-games-with-test-driven-development
- Test Driven Development By Example, Kent Beck
- Clean Code, Robert C. Martin
- TDD 실천법과 도구, 채수원, 한빛미디어
- Quality Code, Stephen Vance, Addison-Wesley

# THANK YOU.

# Google Test

# Google Test 기초

# A Unit Test



- Arrange

  Set-up preconditions and inputs

- Act

  Invoke the method under test

- Assert

  Verify test result

```
TEST(Multiply, AddTwoNumbers) {
   Calculator calc;

   int result = calc.Add(2, 3);

   ASSERT_EQ(5, result);
}
```

⇒하나의 테스트 결과를 확인 한 후 다음 테스트로 가도록

⇒하나의 테스트 안의 다중 assertion 은 중간에 fail이 난 경우 이후 무슨 일이 일어날 지 불확실

⇒단일 결과의 여러 측면을 테스트해야하는 경우

# 테스트 작성하기



```
#include <gtest/gtest.h>                    // test.cpp
#include "main.cpp“
TEST(Multiply, TwoMultiplyThree) {
   EXPECT_EQ(multiply(2, 3), 6);
}
TEST(Multiply, MultiplyZero) {
    EXPECT_EQ(multiply(2, 0), 0);
    EXPECT_EQ(multiply(-1, 0), 0);
    EXPECT_EQ(multiply(0, 0), 0);
}
TEST(Max, PositiveMax) {
    ASSERT_EQ(myMax(1, 3, 2), 3);
    ASSERT_EQ(myMax(2, 3, 2), 3);
    ASSERT_EQ(myMax(5, 5, 5), 5);
}
int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

```
// main.cpp
int multiply(int a, int b) {
    return a * b;
}

int myMax(int a, int b, int c) {
    if(a >= b && a >= c)
        return a;
    else if(b >= a && b >= c)
        return b;
    return c;
}
```

- Header 파일 include
- InitGoogleTest 호출
- RUN_ALL_TESTS의 결과를 반환

**::testing::InitGoogleTest() 함수**

- googletest flag의 command line 분석, 모든 flag 제거
- RUN_ALL_TESTS()

# main() 작성



- 대부분의 경우 main() 작성 필요 없음
- gtest_main : googletest 에서 제공하는 기본 구현 main()
- 사용자 정의 작업들과 gtest_main() 을 연결하기 위해 main() 작성
- 테스트들을 수행하기 전에 test suite, test fixture 프레임워크에서 정의할 수 없는 작업이 필요한 경우에만 적용
- main() 에서는 반드시 RUN_ALL_TEST() 결과값 리턴해야 함.

# 테스트 호출



TEST(), TEST_F() 에서 정의한 테스트들은 내부적으로 googletest 에 등록

- RUN_ALL_TEST()
  - 연결된 유닛 안의 모든 테스트 수행, 서로 다른 test suites 이거나 다른 소스 파일일 수도 있음.
  - RUN_ALL_TEST 수행 절차 => 치명적 오류 발생시 단계 skip됨.
    - 모든 googletest flag 상태값 저장
    - 첫번째 테스트의 test fixture 객체 생성
    - SetUp() 실행으로 초기화하고, fixture 하에서 테스트 수행
    - TearDown() 실행으로 fixture 정리
    - fixture 삭제
    - 모든 googletest flag 상태값 복원
    - 모든 테스트 수행될 때까지 위의 단계 반복, fatal error 발생하면 다음 단계로 건너뜀.
  - RUN_ALL_TEST() 리턴값 반드시 처리
    : 자동화된 테스트는 종료 코드 기반의 테스트 통과 여부 결정되어야 하므로, main() 함수는 반드시 RUN_ALL_TEST() 결과값을 리턴해야 한다.
  - RUN_ALL_TEST() 는 한 번만 호출해야 함. : 그러지 않으면 thread-safe death tests 와 충돌.

# TEST



- TEST()

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

  - TestSuiteName, TestName  명명 규칙 : 유효한 C++ 식별자 이름, (_) 제외.
  - TestSuiteName 이 다른 테스트들 간의 같은 TestName 허용.
  - 여러 테스트가 있는 경우는 TestSuiteName 과 TestName 의 조합의 중복 주의
    => 빌드 오류 발생.

- TEST_F() : 테스트 픽스처를 사용한 테스트
  - 첫 번째 인수는 테스트 픽스처로 정의된 클래스 이름

- TEST_P() : 하나의 테스트 매개 변수를 바꿔가면서 여러 번 테스트를 수행하게 한다.

# Assertions



- 2-type Assertion
  - ASSERT
    - 테스트 실패시 fatal failures 발생시키고 현재 함수 종료.
    - fail 시 종료되면서 메모리 정리 코드를 건너뛰어 memory leak 발생할 수 있으니 주의
  - EXPECT : 테스트 실패시 nonfatal failure 발생시키고 종료 없이 테스트 진행.

- 기본 Assertions
  - 조건의 참, 거짓 여부 테스트

| Fatal Assertion | Nonfatal Assertion | |
|---|---|---|
| ASSERT_TRUE(condition); | EXPECT_TRUE(condition); | condition is true |
| ASSERT_FALSE(condition); | EXPECT_FALSE(condition); | condition is false |

Ex) ASSERT_EQ(2+2, 5);

# Assertions : 이진 비교 Assertions



■ 이진 비교 Assertions

| Fatal Assertion | Nonfatal Assertion | |
|---|---|---|
| ASSERT_EQ(expected, actual) | EXPECT_EQ(expected, actual) | expected == actual |
| ASEERT_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 |

- User-defined type의 경우 비교 연산자 정의시 의미를 가짐.
  2 개의 C string 에 대한 ASSERT_EQ() => 같은 메모리에 있는 지 테스트
- 포인터의 null확인
  - *_EQ(ptr, nullptr) 또는 *_NE(ptr, nullptr) 로 비교

# Assertion : 사용자 정의 fail 메시지



- 사용자 정의 fail 메시지
  - << 연산자 사용하여 매크로로 메시지 전송

```
ASSERT_EQ(x.size(), y.size()) << "Vectors x and y are of unequal length";

for (int i = 0; i < x.size(); ++i) {
  EXPECT_EQ(x[i], y[i]) << "Vectors x and y differ at index " << i;
}
```

  - ostream 에 전송될 수 있는 것들은 모두 assertion 매크로로 전송 가능

# Failure Messages



- Understand why the test failed
  - ASSERT_TRUE() vs ASSERT_EQ()
- Reduce debugging time
- It's the purpose of test
  - 실패가 발생하는 경우 이전에 테스트를 만든 이유를 이해
  - 오류생성자와 테스트작성자가 다른 경우 중요

# Test 예제 : SimpleAssertionTest



```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
    EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(2, Factorial(2));
    EXPECT_EQ(6, Factorial(3));
    EXPECT_EQ(40320, Factorial(8));
}
```

```
//Source.cc
int Factorial(int n); // Returns the factorial of n
```

# Assertions : 문자열 비교 Assertions



- 문자열 비교 Assertions : 두 개의 C string 비교

| Fatal Assertion | Nonfatal Assertion | |
|---|---|---|
| ASSERT_STREQ( expected_str, actual_str) | EXPECT_STREQ( expected_str, actual_str) | 두 Cstring 의 값이 같은 지 여부 |
| ASSERT_STRNE(str1, strl2) | EXPECT_STRNE(str1, strl2) | 두 Cstring 의 값이 다른지 여부 |
| ASSERT_STRCASEEQ( expected_str, actual_str) | EXPECT_STRCASEEQ( expected_str, actual_str) | 대소문자 구분 없이, 두 Cstring 의 값이 같은 지 여부 |
| ASSERT_STRCASENE(str1, str2) | EXPECT_STRCASENE(str1, str2) | 대소문자 구분 없이, 두 Cstring 의 값이 다른 지 여부 |

- CASE: 대소문자 상관 없음.
- std::string 객체 비교하려면 EXPECT_EQ, EXPECT_NE 사용
- Null pointer 와 빈 문자열은 다른 것으로 간주.
- ASSERT_STREQ(C_string, NULL) : C string 이 Null 인지 체크

# Test 예제 : StringAssertionTest



```
// Tests the default c'tor.
TEST(MyString, DefaultConstructor) {
    const MyString s;
    EXPECT_STREQ(nullptr, s.c_string());
    EXPECT_EQ(0u, s.Length());
}

const char kHelloString[] = "Hello, world!";

// Tests the c'tor that accepts a C string.
TEST(MyString, ConstructorFromCString) {
    const MyString s(kHelloString);
    EXPECT_EQ(0, strcmp(s.c_string(), kHelloString));
    EXPECT_EQ(sizeof(kHelloString)/sizeof(kHelloString[0]) - 1,
s.Length());
}
```

```
//MyString.cc
class MyString {
private:
    const char* c_string_;
    const MyString& operator=(const MyString& rhs);

public:
    static const char* CloneCString(const char* a_c_string);
    MyString() : c_string_(nullptr) {}
    explicit MyString(const char* a_c_string) : c_string_(nullptr) {
        Set(a_c_string);
    }
    MyString(const MyString& string) : c_string_(nullptr) {
        Set(string.c_string_);
    }
    ~MyString() { delete[] c_string_; }
    const char* c_string() const { return c_string_; }
    size_t Length() const { return c_string_ == nullptr ? 0 : strlen(
c_string_); }
    void Set(const char* c_string);
};
```

# Test Fixture 를 사용한 테스트



- 여러 테스트 메소드에  같은 구성의 객체를 사용하는 경우, test fixture 를 사용하여 재사용.
- Test fixture class 작성
  - **::testing::Test** 로부터 상속받은 fixture 클래스 정의 :  **protected** 로 접근 제한자
  - 테스트 메소드들에서 사용할 객체를 클래스 내부에서 선언
  - 필요에 따라, 개별 테스트 객체 준비를 위한 기본 생성자를 **SetUp()** 함수 작성
  - SetUp() 에서 할당한 모든 자원을 소멸하기 위한 **TearDown()** 함수 작성
  - 테스트들이 공유할 함수 정의
- TEST_F() : 테스트 픽스처를 사용한 테스트 작성하기
  - TEST_F(TestFixtureName, TestName)
  - TEST_F() 로 선언된 unit test 마다, 런타임시 새로운 test fixture 생성.
    SetUp()에서 정의한 대로 초기화, 테스트 수행, TearDown()에 의한 정리 후, test fixture 삭제.
  - 같은 test suite 의 테스트들은 다른 test fixture 객체를 가지며,
    다음 test fixture 를 생성하기 전에 기존 test fixture 객체 삭제 됨.

# Test 예제 : TestFixtureTest



```
class QueueTest : public ::testing::Test {
 protected:
  void SetUp() override {
     q1_.Enqueue(1);
     q2_.Enqueue(2);
  }

  // void TearDown() override {}

  Queue<int> q0_;
  Queue<int> q1_;
  Queue<int> q2_;
};
```

```
TEST_F(QueueTest, IsEmptyInitially) {
  EXPECT_EQ(q0_.size(), 0);
}

TEST_F(QueueTest, DequeueWorks) {
  int* n = q0_.Dequeue();
  EXPECT_EQ(n, nullptr);

  n = q1_.Dequeue();
  ASSERT_NE(n, nullptr);
  EXPECT_EQ(*n, 1);
  EXPECT_EQ(q1_.size(), 0);
  delete n;
}
```

# 실행 옵션



- Test Selection
  - --gtest_list_tests
  - --gtest_filter
    - Run only specified Tests
      - xyzTest
      - xyz*, *zTest, *yz*
      - xyzTes?
    - Run all except specified test
    - Separated by ':'
      - xyz.*:abc.*-xyz.old
  - --gtest_also_run_disabled_tests
- Test Execution
  - --gtest_repeat=count
  - --gtest_shuffle
    - gtest_random_seed = number
- Test Output
  - --gtest_color
    - On/Off/Auto
  - --gtest_print_time
    - Test execution time or use '0' to disable
  - --gtest_output=xml
    - JUnit compatible
    - Can specify file/directoryAssertion Behavior
- Assertion Behavior

# Advanced Google Test

# Explicit Success & Failure



- 실제 테스트를 수행하지 않고, 성공, 실패를 generate.
- 테스트 flow 의 control 에 유용
- SUCCEED()
  - 테스트가 성공한 것으로 간주됨. 테스트 전체의 성공은 아님 주의.
- FAIL()
  - fatal failure를 generate.
  - 리턴이 없는(void) 함수에서만 사용 가능.
- ADD_FAILURE()
  - non-fatal failure를 generate.
- ADD_FAILURE_AT("file_path", line_number)
  - non-fatal failure를 generate.

# Explicit Success & Failure 예제



```
switch(expression) {
  case 1:
    ... some checks ...
  case 2:
    ... some other checks ...
  default:
    FAIL() << "We shouldn't get here.";
}
```

# Exception Assertions



| Fatal Assertion | Nonfatal Assertion | |
|---|---|---|
| ASSERT_THROW(statement, exception_type) | EXPECT_THROW(statement, exception_type) | statement 가 명시된 exception 발생시키는 지 여부 |
| ASSERT_ANY_THROW(statement) | EXPECT_ANY_THROW(statement) | statement 가 임의의 exception 발생시키는 지 여부 |
| ASSERT_NO_THROW(statement) | EXPECT_NO_THROW(statement) | statement 가 어떤 exception 도 발생시키지 않는 지 여부 |

```
ASSERT_THROW(Foo(5), bar_exception);

EXPECT_NO_THROW({
  int n = 5;
  Bar(&n);
});
```

# Predicate Assertions



- 좀 더 복잡한 형태 표현을 확인(Assertion)하기 위하여
- 세 가지 방법
  - (ASSERT|EXPECT)_PRED()
  - ::testing::AssertionResult 클래스
  - (ASSERT|EXPECT)_PRED_FORMAT()

# Predicate Assertions – (ASSERT|EXPECT)_PRED$_N$



- bool 값을 리턴하는 함수를 이용하는 방법

| Fatal Assertion | Nonfatal Assertion | |
|---|---|---|
| ASSERT_PRED1(pred1, val1) | EXPECT_PRED1(pred1, val1) | pred1(val1) 이 참인지 |
| ASSERT_PRED2(pred2, val1,val2) | EXPECT_PRED2(pred2, val1,val2) | pred2(val1,val2) 이 참인지 |
| … | … | … |

```
 //m 과 n 이 서로소이면 true 리턴
bool MutuallyPrime(int m, int n) { … }

const int a = 3;
const int b = 4;
const int c = 10;
```

```
EXPECT_PRED2(MutuallyPrime, a, b);    //성공
EXPECT_PRED2(MutuallyPrime, b, c);    //실패
```

```
EXPECT_PRED1(IsPositive, 5);          // 1
EXPECT_PRED1(static_cast<bool
(*)(int)>(IsPositive), 5);            // 2

ASSERT_PRED1(IsNegative<int>, -5);
ASSERT_PRED2(GreaterThan<int, int>, 5, 0);    // 3
ASSERT_PRED2((GreaterThan<int, int>), 5, 0);  // 4
```

# Predicate Assertions – ::testing::AssertionResult



- AssertionResult 객체를 리턴하는 함수 이용
  - AssertionSuccess(), AssertionFailure() 로 Assertion 결과와 메시지를 나타낼 수 있다.
  - factory function 을 이용하여 AssertionResult 객체 생성하게 작성할 수 있다.

```
namespace testing {
    AssertionResult AssertionSuccess();
    AssertionResult AssertionFailure();
}
```

```
bool IsEven(int n) {
  return (n % 2) == 0;
}
...
EXPECT_TRUE(IsEven(Fib(4)))
```

```
testing::AssertionResult IsEven(int n) {
  if ((n % 2) == 0)
    return testing::AssertionSuccess() << n << " is even;
  else
    return testing::AssertionFailure() << n << " is odd";
}

TEST(AssertionResultTest, IsEven) {
    EXPECT_TRUE(IsEven(5));
    EXPECT_FALSE(IsEven(6))
}
```

# Predicate Assertions – (ASSERT|EXPECT)_PRED_FORMAT$_N$ (1/2)



- Assertion 을 fully customize 하는 방법 : predicate-formatter의 signature 준수

| Fatal Assertion | Nonfatal Assertion | |
|---|---|---|
| ASSERT_PRED_FORMAT1(pred_format1, val1) | EXPECT_PRED_FORMAT1(pred_format1, val1) | pred_format1(val1)의 성공 |
| ASSERT_PRED_FORMAT2(pred_format2, val1,val2) | EXPECT_PRED_FORMAT2(pred_format2, val1,val2) | pred_format2(val1,val2) 의 성공 |
| … | … | … |

**//predicate-fomatter() function or functor 의 signature**

```
testing::AssertionResult PredicateFormattern(const char* expr1,
                                             const char* expr2,
                                               …
                                             const char* exprn,
                                             T1 val1,
                                             T2 val2,
                                              …
                                             Tn valn);
```

# Predicate Assertions – (ASSERT|EXPECT)_PRED_FORMAT$_N$ (2/2)



```
// Returns the smallest prime common divisor of m and n, or 1 when m and n are mutually prime.
int SmallestPrimeCommonDivisor(int m, int n) { ... }

// A predicate-formatter for asserting that two integers are mutually prime.
testing::AssertionResult AssertMutuallyPrime(const char* m_expr,
                                             const char* n_expr,
                                             int m,
                                             int n) {
  if (MutuallyPrime(m, n)) return testing::AssertionSuccess();

  return testing::AssertionFailure() << m_expr << " and " << n_expr
      << " (" << m << " and " << n << ") are not mutually prime, "
      << "as they have a common divisor " << SmallestPrimeCommonDivisor(m, n);
}
```

```
//predicate-formatter 호출
EXPECT_PRED_FORMAT2(AssertMutuallyPrime, b, c);
```

```
//실행 결과 메시지
b and c (4 and 10) are not mutually prime, as they have a common divisor 2.
```

# Test 예제 : AssertPredTest



```
TEST(AssertPredTest, IsPositive) {
    EXPECT_PRED1(static_cast<bool (*)(int)>(IsPositive),
5);
    ASSERT_PRED1(IsNegative<int>, -5);
    ASSERT_PRED2((GreaterThan<int, int>), 5, 0);
}

TEST(AssertionResultTest, IsEven) {
    EXPECT_TRUE(IsEven(4));
    EXPECT_FALSE(IsEven(6));
}
TEST(AssertionResultTest, AssertMutuallyPrime) {
    const int b = 4;
    const int c = 8;

    EXPECT_PRED_FORMAT2(AssertMutuallyPrime, b, c);
}
```

```
//Source.h
template <typename T>
inline bool IsNegative(T x) {
  return x < 0;
}

template <typename T, typename F>
inline bool GreaterThan(T x, F y) {
  return x > y;
}
```

# Assertions : Floating-Point 비교 Assertions



- floating-point 값의 round-off error 로 정확한 비교 불가
  => naïve한 비교와 bound 를 준 비교 방식 제공

| Fatal Assertion | Nonfatal Assertion | |
|---|---|---|
| ASSERT_FLOAT_EQ(val1, val2) | EXPECT_FLOAT_EQ(val1, val2) | 두 float의 값이 almost equal |
| ASSERT_DOUBLE_EQ(val1, val2) | EXPECT_DOUBLE_EQ(val1, val2) | 두 double의 값이 almost equal |
| ASSERT_NEAR(val1, val2, abs_error) | EXPECT_NEAR(val1, val2, abs_error) | \|val1-val2\| <= abs_error |

* almost equal : 4 ULP(Units in Last Place) 이내 의미

- Floating-Point Predicate-format 함수를 이용한 비교

```
(ASSERT|EXPECT)_PRED_FORMAT2(testing::FloatLE, val1, val2);
(ASSERT|EXPECT)_PRED_FORMAT2(testing::DoubleLE, val1, val2);
```

# Argument Matchers



- gMock matcher library 를 사용
  - googletest에 gMock 이 포함되어 build 되기 때문에 build dependency 추가 필요 없으나, include gmock/gmock.h 는 필요.

| Fatal Assertion | Nonfatal Assertion | |
|---|---|---|
| ASSERT_THAT(value, matcher) | EXPECT_THAT(value, matcher) | value와 matcher 일치 |

```
#include "gmock/gmock.h"

...
  EXPECT_THAT(Foo(), StartsWith("Hello"));
  EXPECT_THAT(Bar(), MatchesRegex("Line ₩₩d+"));
  ASSERT_THAT(Baz(), AllOf(Ge(5), Le(10)));

  ASSERT_THAT(result, AllOf(NotNull(), StrNe("")));
  EXPECT_THAT(result, AnyOf(Gt(100), Le(-100)));
```

# FRIEND_TEST : Private code test



- Private 코드를 .cc 로 만들어 include ".cc "
- Namespace "internal" 로 하여 "*-internal.h"로
- Text fixture 를 클래스의 fiend 로

FRIEND_TEST(TestSuiteName, TestName) 의 사용

```
class Foo {
    friend class FooTest;
    FRIEND_TEST(FooTest, Bar);
    FRIEND_TEST(FooTest, Baz);
private:
    int Bar(void* x){ return 1; }
    int Baz(void* x){ return 2; }
};
```

```
class FooTest : public testing::Test {
protected:
    Foo foo;
};

TEST_F(FooTest, Bar) {
    EXPECT_EQ(foo.Bar(NULL), 1);
}
TEST_F(FooTest, Baz) {
    EXPECT_EQ(foo.Baz(NULL), 2);
}
```

# TEST_P : Value-Parameterized Tests



- 같은 테스트에 다른 parameter들을 가지고 테스트

```
class FooTest : public testing::TestWithParam<const char*> {
    ...
};

TEST_P(FooTest, DoesBlah) {
  EXPECT_TRUE(foo.Blah(GetParam()));
  ...
}

INSTANTIATE_TEST_SUITE_P(InstantiationName,
                         FooTest,
                         testing::Values("meeny", "miny", "moe"));
```

- 클래스 생성
  - testing::TestWithParam<T>
  - testing::Test로 상속받은 fixture가 있는 경우 testing::WithParamInterface<T>로 부터 추가 상속

```
class BaseTest : public testing::Test {...};
class BarTest : public BaseTest,
                public testing::WithParamInterface<const char*> {...};
```

- GetParam() : 테스트 파라미터 가져오기

# TEST_P : Value-Parameterized Tests



- Parameter Generator : name space testing

  - Range(begin, end [, step]) - {begin, begin+step, begin+step+step, ...} 산출
  - Values(v1, v2, ..., vn) - {v1, v2, ..., vN} 산출
  - ValuesIn(container) and ValuesIn(begin, end) - c-style 배열 산출
  - Bool() - {false, true} 산출
  - Combine(g1, g2, ..., gN) - n 개의 generator 들의 값들을 std::tuples 로 조합

# TEST_P : Value-Parameterized Tests



- Test Suite에서 테스트 파라미터별 테스트 인스턴스 생성
  - global, namespace 영역에 위치해야 함.

```
INSTANTIATE_TEST_SUITE_P(InstantiationName,
                         FooTest,
                         testing::Values("meeny", "miny", "moe"));

const char* pets[] = {"cat", "dog"};
INSTANTIATE_TEST_SUITE_P(AnotherInstantiationName, FooTest,
                         testing::ValuesIn(pets));
```

  - INSTANTIATE_TEST_SUITE_P()의 마지막 argument는 파라미터에 근거한 사용자가 정의 테스트 명 지정 함수

```
INSTANTIATE_TEST_CASE_P(AllAllowedCharacters, CustomFunctorNamingTest,
                        testing::Values("test1", "test2", "test3"),
                        CustomParamNameFunctor)
```

# Test 예제 : CustomParameterizedTest



```
class MyTestSuite : public testing::TestWithParam<int> {};

TEST_P(MyTestSuite, MyTest) {  std::cout << "Example Test Param: " << GetParam() << std::endl; }

INSTANTIATE_TEST_SUITE_P(MyGroup, MyTestSuite, testing::Range(0, 10), testing::PrintToStringParamName());

enum class MyType { MY_FOO = 0, MY_BAR = 1 };

class MyTestSuite : public testing::TestWithParam<std::tuple<MyType, std::string>> {  };

INSTANTIATE_TEST_SUITE_P(
        MeaningfulTestParameters, MyTestSuite,
        testing::Combine(testing::Values(MyType::MY_FOO, MyType::MY_BAR), testing::Values("1","2")),
        [](const testing::TestParamInfo<MyTestSuite::ParamType>& info) {
                std::string name = std::get<0>(info.param) == MyType::MY_FOO ? "Foo_" : "Bar_";
                return name + std::get<1>(info.param);
});
```

# Approval Test

# Assertion with approvals (1/2)



```
#define APPROVALS_GOOGLETEST
#include "ApprovalTests.hpp“


TEST(TestSuiteName, TestName){
    ApprovalTests::Approvals::verify("Hello Approvals");
    ApprovalTests::Approvals::verifyAll(“rectangles”, getRectangles());
}
```

- using namespace ApprovalTests;
- Default Golden master file : {sourceFileName}. {TestSuiteName}.{TestName}.approved.txt
- approval 실패시 {sourceFileName}. {TestSuiteName}.{TestName}. received.txt 생성

# Assertion with approvals (2/2)



```
// main.cpp:

#define APPROVALS_GOOGLETEST_EXISTING_MAIN
#include "ApprovalTests.hpp"

int main(int argc, char** argv)
{
    ::testing::InitGoogleTest(&argc, argv);

    ApprovalTests::initializeApprovalTestsForGoogleTests();

    return RUN_ALL_TESTS();
}
```

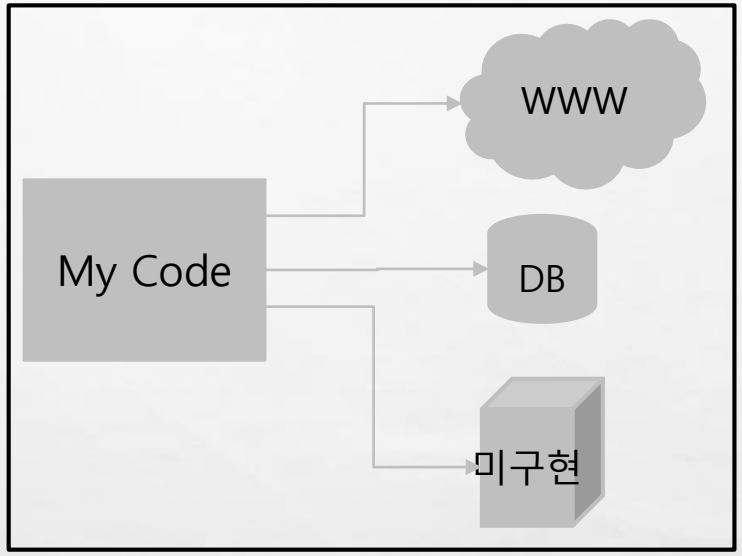
# Test Double & gMock

이미지출처 : https://news.hmgjournal.com/Tech/Item/car-clash-test

# Dependencies

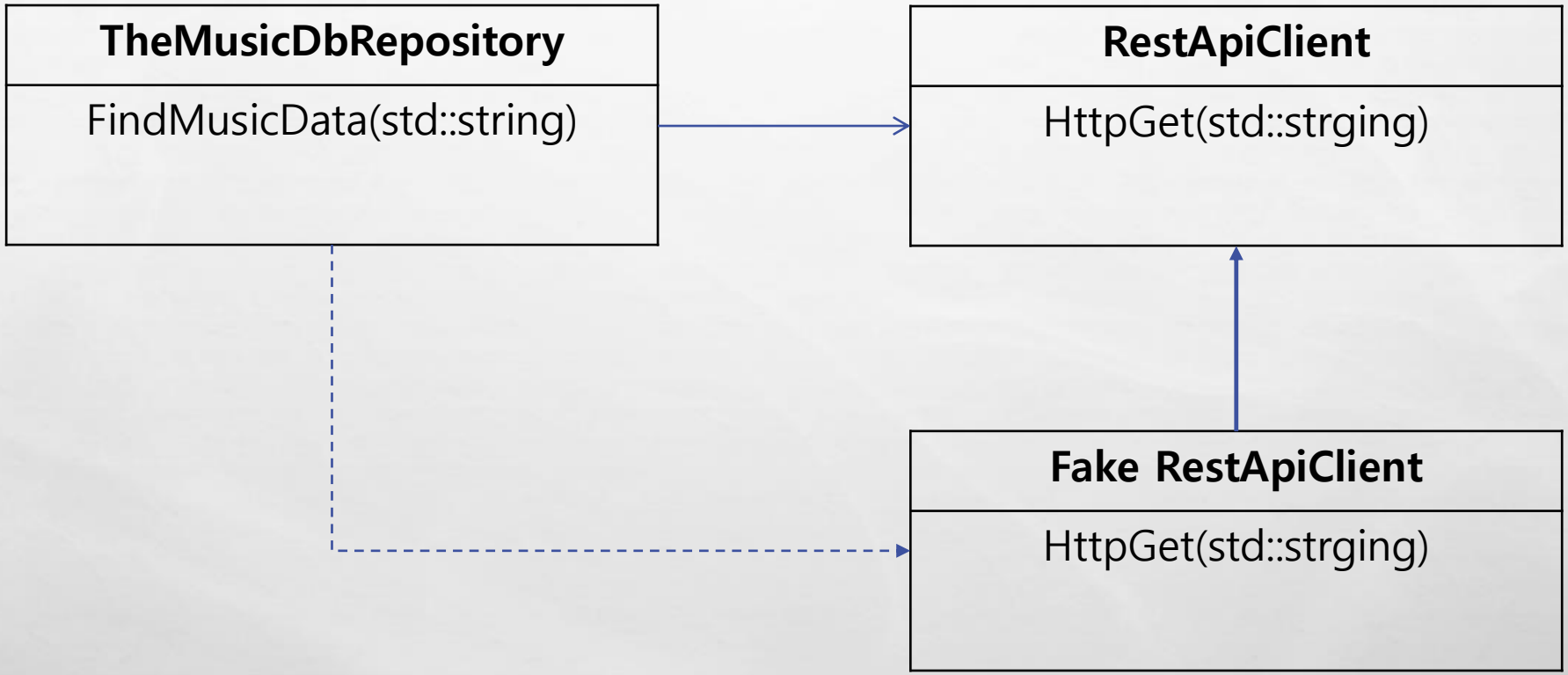


- Unit Tests vs. Integration Tests
- Integration Test의 어려움
  - 테스트 설정의 어려움
  - 테스하기 어려운 케이스가 존재
  - 테스트의 속도가 느림
  - 테스트 결과의 확인이 어려움
  - Side effects
  - 환경에 의존적임



# Own Test Double (1/2)





# Own Test Double (1/2)



```
class RestApiClient
{
public:
    RestApiClient();
    ~RestApiClient();

    std::string HttpGet(std::string& url);
};

class TheMusicDbRepository
{
    RestApiClient m_client;

public:
    TheMusicDbRepository(const MusicDataFactory& musicDataFactory)
    ~TheMusicDbRepository();

    std::shared_ptr<MusicData> FindMusicData(const std::string&
musicName);
}:
```

```
class RestApiClient
{
public:
    RestApiClient();
    virtual ~RestApiClient();

    virtual std::string HttpGet(std::string& url);
};

class TheMusicDbRepository
{
    RestApiClient& m_client;
public:
    TheMusicDbRepository(const MusicDataFactory& musicDataFactory,
RestApiClient& client)
    ~TheMusicDbRepository();

    std::shared_ptr<MusicData> FindMusicData(const std::string&
musicName);
};
```

# Own Test Double (2/2)



```
class FakeRestApiClient : public RestApiClient {
{
    std::string m_result;

public:
    FakeRestApiClient(std::string result) : m_result(result){}
    std::string HttpGet(std::string& url) override {
        return m_result;
    }
}

TEST()
{
    TheMusicDbDataFactory factory;
    FakeRestApiClient fakeClient("");

    TheMusicDbRepository repository(factory, fakeClient);
    ASSERT_THROW(repository.FindMusicData("name"), MusicNotFoundException);
}

TEST()
{
    TheMusicDbDataFactory factory;
    FakeRestApiClient fakeClient("{}");

    TheMusicDbRepository repository(factory, fakeClient);
    ASSERT_THROW(repository.FindMusicData("name"), MusicNotFoundException);
}
```

# Test Double



## Dummy

- 인스턴스 수준의 객체

## Stubs

- 특정 상태, 모습을 가정한 객체. 특정 값 리턴하거나 메시지 출력 형태

## Fake

- 로직이나 동작을 비교적 단순화하여 구현한 객체.
  다른 객체나 클래스들과의 의존성 제거하기 위해 사용.

## Spies

- 테스트하는 동안의 호출을 기록하였다가 알려준다.

## Mocks

- 특정한 동작이 올바르게 수행되었는 지 여부와 같은 behavior를 테스트.

# Interaction Testing



- State Based Testing vs. Interaction Testing
  - Return value or object state vs. Method was/wasn't called
  - Using Assertion vs. Using Fakes/Mocks

- Interaction Testing 의 사용
  - 테스트에서 테스트 결과에 액세스 할 수 없는 경우
    - Fake object를 사용하여 간접적으로 결과를 확인
  - 결과가 테스트 된 개체의 외부에 있는지 여부
    - 이메일이 전송, 엔티티를 데이터베이스에 저장, 일부 외부 서비스 호출. 실제로 모든 서비스는 제어 할 수 없으므로 이러한 종속성을 mocking 할 때 올바른 인수로 올바르게 호출되었는지 확인
  - 행동이 비즈니스 서비스 연결이  요구 사항의 일부인 경우
    - ex) 클라이언트와의 끊어진 경우 X 번 시도해야한다.

# Mock 실행



```
#include "gtest/gtest.h"
#include "gmock/gmock.h“

int main(int argc, char** argv)
{
    ::testing::InintGoogleMock(&argc, argv);
    return RUN_ALL_TETSS();
}
```

# Mock 클래스 정의



■ 일반 클래스의 mocking

```
class Foo {
  ...
  virtual ~Foo();
  virtual int GetSize() const = 0;
  virtual string Describe(const char* name) = 0;
  virtual string Describe(int type) = 0;
  virtual bool Process(Bar elem, int count) = 0;
};
```

```
#include "gmock/gmock.h"

class MockFoo : public Foo {
Public:
  //MOCK_METHOD(ReturnType, MethodName, (Args...));
  //MOCK_METHOD(ReturnType, MethodName, (Args...), (Specs...));
  MOCK_METHOD(int, GetSize, (), (const, override));
  MOCK_METHOD(string, Describe, (const char*), (override));
  MOCK_METHOD(string, Describe, (int), (override));
  MOCK_METHOD(bool, Process, (Bar, int), (override));
};
```

```
Cf. MOCK_METHOD2(TakeUnique, int(const std::unique_ptr<int>&, std::unique_ptr<int>));
```

# Mock 클래스 정의



## Specs

- const - const 메소드의 재정의 시 필수
- override - virtual 메소드의 재정의 시 권장
- Noexcept – noexcept 메소드의 재정의 시 필수
- Calltype(…) – 메소드의 call type 을 지정(windows api에 유용)
  - Mock 함수의 기본 호출 규칙을 사용하지 않는 경우, CallType(convention) 을 지정.

  *STDMETHODCALLTYPE : 윈도우, <objbase.h>에서 정의

# Mock 클래스 정의



## 일반 클래스의 mocking 의 주의점

- Real class 의 protected, private, public 메소드에 대한 MOCK_METHOD 는 public에
- , 다루기
  - MOCK_METHOD(bool, CheckMap, (std::map<int, double>, bool));

  <Solution 1>

  MOCK_METHOD(bool, CheckMap, ((std::map<int, double>), bool));

  <Solution 2>

  using MapIntDouble = std::map<int, double>;

  MOCK_METHOD(bool, CheckMap, (MapIntDouble, bool));
- virtual method
  - Real class의 virtual method이나 테스트 하기를 원하지 않는 경우. MOCK_METHOD를 안 만들면 경고 발생

```
class Foo { ...
    virtual int Add(Element x);
    virtual int Add(int times, Element x);
}
```

```
class MockFoo: public Foo { ...
    using Foo::Add;
    MOCK_METHOD(int, Add, (Element x),(override)
}
```

# Mock 클래스 템플릿 정의



- 클래스 템플릿의 mocking

```
template <typename Elem>
class StackInterface {
  ...
  virtual ~StackInterface();
  virtual int GetSize() const = 0;
  virtual void Push(const Elem& x) = 0;
};
```

```
template <typename Elem>
class MockStack : public StackInterface<Elem> {
  ...
  MOCK_METHOD(int, GetSize, (), (const, override));
  MOCK_METHOD(void, Push, (const Elem& x), (override));
};
```

# Mock 클래스 예제 : SimpleMockTest



```
EXPECT_CALL(mock_object, method(matchers))
    .Times(cardinality)
    .WillOnce(action)
    .WillRepeatedly(action);
```

```
EXPECT_CALL(mock_object, non-overloaded-method)
    .Times(cardinality)
    .WillOnce(action)
    .WillRepeatedly(action);
```

```
#include "gmock/gmock.h"

TEST(TestForMyClass, TestGetSize)
{
    MockFoo obj1;
    //EXPECT_EQ(0, obj1.GetSize());
    EXPECT_CALL(obj1, GetSize())
        .Times(1)
        .WillOnce(Return(1))
        .WillOnce(Return(2))
        .WillRepeatedly(Return(3));
    EXPECT_EQ(0, obj1.GetSize());
}
```

```
TEST(TestForMyClass, TestDescribe)
{
    MockFoo obj1;
    //EXPECT_CALL(obj1, Describe(1))
    //    .Times(AtLeast(0));

    EXPECT_CALL(obj1, Describe("123"))
        .Times(AtLeast(0));

    EXPECT_CALL(obj1, Describe(::testing::_))
        .Times(0);

    EXPECT_CALL(obj1, Describe)
        .Times(0);
}
```

# Mock 객체 생성



- *Uninteresting* calls vs *unexpected* calls
  - uninteresting: EXPECT_CALL 이 없음.
  - unexpected: EXPECT_CALL 이 설정되어 있으나 expectation과 일치하는 호출이 발생하지 않음.
- NiceMock : uninteresting call 무시
- NaggyMock : uninteresting call 경고, default
- StrickMock : uninteresting call 에 대해 failure 처리

```
using ::testing::NiceMock;
using ::testing::NaggyMock;
using ::testing::StrictMock;

NiceMock<MockFoo> nice_foo;      // The type is a subclass of MockFoo.
NaggyMock<MockFoo> naggy_foo;    // The type is a subclass of MockFoo.
StrictMock<MockFoo> strict_foo;  // The type is a subclass of MockFoo.
```

# Mock 객체 생성 예제: NiceStrictTest



```
class Foo {
public:
  virtual ~Foo() {}

  virtual void DoThis() = 0;
  virtual int DoThat(bool flag) = 0;
};

class MockFoo : public Foo {
public:
  MockFoo() {}
  void Delete() { delete this; }

  MOCK_METHOD(void, DoThis, ());
  MOCK_METHOD(int, DoThat, (bool));
};
```

```
TEST(RawMockTest, IsNaggy_IsNice_IsStrict) {
  MockFoo raw_foo;
  EXPECT_TRUE(Mock::IsNaggy(&raw_foo));
  EXPECT_FALSE(Mock::IsNice(&raw_foo));
  EXPECT_FALSE(Mock::IsStrict(&raw_foo));
}

TEST(NiceMockTest, WarningForUninterestingCall) {
  NiceMock<MockFoo> nice_foo;

  nice_foo.DoThis();
  nice_foo.DoThat(true);
}

TEST(NiceMockTest, WarningForUninterestingCallAfterDeath) {
  NiceMock<MockFoo>* const nice_foo = new NiceMock<MockFoo>;

  ON_CALL(*nice_foo, DoThis())
      .WillByDefault(Invoke(nice_foo, &MockFoo::Delete));

  nice_foo->DoThis();
}
```

# Mocking Frameworks



- Mock 객체 생성
  - 아무 것도 하지 않는 오브젝트를 생성

- 동작을 정의하고 예외에 대한 특정 값을 반환
  - 동작을 하지 않아도 메소드를 호출하면 반환 타입에 따른 반환이 필요 => 기본 반환 값
  - 필요 시 기본 반환 값을 변경
  - 행위별 동작을 지정

- 특정 메소드가 특정 인수로 호출되었는지 확인

# Mock 의 Default Actions (1/2)



- 기본 반환 값
  - Void method -> do nothing
  - Bool : false
  - Numeric : 0
  - ptr : NULL
  - 객체 : 기본 생성자가 있는 경우 해당 객체의 인스턴스

  => 필요시 기본 반환 값의 변경 가능

- 기본 반환 값의 중요성
  - Reduce test code
  - Increase readability
  - 향후 변경이나 리팩터링에 의해 실패할 가능성이 줄어 듦

# Mock 의 Default Actions (2/2)



- Return type 이 T인 함수에 대한 default action 지정

```
using ::testing::DefaultValue;
// Sets the default value to be returned. T must be CopyConstructible.
DefaultValue<T>::Set(value);
// Sets a factory. Will be invoked on demand. T must be MoveConstructible.
DefaultValue<T>::SetFactory(&MakeT);
// ... use the mocks ...
DefaultValue<T>::Clear();       // Resets the default value.
```

- ON_CALL (): Mock 객체의 특정 메서드에 대한 기본 동작 지정

```
//ON_CALL(mock-object, method(matchers)).WillByDefault(action);
ON_CALL(fakeFoo, method(_)).WillByDefault(Return(-1));
ON_CALL(fakeFoo, method(0)).WillByDefault(Return(0));
```

# Mock 의 Test Behavior 설정 - Return



```
EXPECT_CALL(mock-object, method (matchers)?)
    .WillOnce(action)              *
    .WillRepeatedly(action)        ?
```

- .WillOnce를 사용하면 동작이 한 번만 발생
- .WillRepeatedly 동작이 두 번 이상 발생하도록

# Mock 의 Test Behavior 설정 - Return



```
EXPECT_CALL(fakeFoo, MyMethod("abc"))
    .WillOnce(Return(-1));
EXPECT_CALL(fakeFoo, MyMethod(_))
    .WillOnce(ReturnRef(bar1));
EXPECT_CALL(Const(fakeFoo), MyMethod(_))
    .WillOnce(...);    // const method

// Return에 전달 된 값은 한 번만 평가.
int n = 0;
EXPECT_CALL(fakeFoo, MyMethod("abc"))
    .WillRepeatedly(Return(n++));
```

- 참조를 반환해야하는 경우 Return 대신 ReturnRef
- Return에 전달 된 값은 한 번만 평가. 여러 번 호출 되더라도 항상 0을 반환
  변경되는 값을 전달해야하는 경우 포인터, 참조, Invoke 동작 사용

# Mock 의 Test Behavior 설정 – Side Effects



```
EXPECT_CALL(fakeFoo, MyMethod(true, _))
    .WillOnce(SetArgPointee<1>(10));

EXPECT_CALL(fakeFoo, MyMethod(true, _)
    .WillOnce(DoAll(SetArgPointee<1>(10), Return(true));
```

- SetArgPointee: 출력용 매개변수의 지정
- DoAll: 둘 이상의 인수를 설정하거나 반환 값 지정 및 다수의 동작 지정

# Mock 의 Test Behavior 설정 – Throwing Exception



```
EXPECT_CALL(fakeFoo, MyMethod())
    .WillOnce(Throw(exception);
```

- 메소드 실행 중에 예외를 발생시킴
- 예) 네트워크 종료 중에 서버를 호출 할 때 어떤 일이 발생하는지 테스트하려면 서버에 연결하는 메서드가 호출 될 때 적절한 예외를 발생시켜 동일한 동작을 시뮬레이션 할 수 있습니다.

# Mock 의 Test Behavior 설정 – Invoking a Function



```
EXPECT_CALL(fakeFoo, MyMethod())
    .WillOnce(Invoke(OtherMethod));

//인수 없이 OtherMethod 실행
EXPECT_CALL(fakeFoo, MyMethod())
    .WillOnce(InvokeWithoutArgs(OtherMethod));

//OtherMethod의 결과를 반환하지 않고 실행
EXPECT_CALL(fakeFoo, MyMethod())
    .WillOnce(InvokeWithoutArgs(IgnoreResult(OtherMethod));

//1번째 argument(function pointer) 에 5의 인수로 실행
EXPECT_CALL(fakeFoo, MyMethod())
    .WillOnce(InvokeArgument<1>(5));
```

# Mock 의 Test Behavior 설정 – Composite Actions



- DoAll(a1, a2, …, an)
  - : a1~an 까지의 action 수행. 마지막 동작 결과 만 return
- IgnoreResult(action)
  - 작업을 수행하고 결과를 무시 – 재정의를 목적으로 사용 됨
- WithArg<N>(action)
  - N번째 argument 를 전달받아 action a 실행
- WithArgs<N1, N2, …, Nk>(action)
  - <N1, N2,.. , Nk> argument를 가지고 a 실행
- WithoutArgs(action)
  - 인수를 전달하지 않고 메소드를 수행

# Mock 의 Test Behavior 설정 – Defining Actions



- Invoke를 위해 action 을 빠르게 생성

- ACTION(Sum) { return arg0 + arg1; }

- ACTION_P(Plus, n) { return arg0 + n }

- ACTION_PK(MyAction, p1, …, pk) { … }

# Mock 의 Test Behavior 설정 – 행위 간 관계



```
EXPECT_CALL(fakeFoo, MyMethod(100)).WillOnce(Return(true));
EXPECT_CALL(fakeFoo, MyMethod(200)).WillOnce(Return(false));
EXPECT_CALL(fakeFoo, MyMethod(300)).WillOnce(Return(exception));
```

- 100 => false, 200 => true

```
EXPECT_CALL(fakeFoo, MyMethod(_)).WillRepeatedly(Return(true));
EXPECT_CALL(fakeFoo, MyMethod(100)).WillRepeatedly(Return(false));
```

- Always true

```
EXPECT_CALL(fakeFoo, MyMethod(100)).WillRepeatedly(Return(false));
EXPECT_CALL(fakeFoo, MyMethod(_)).WillRepeatedly(Return(true));
```

- 100=>?, 200=>?

```
EXPECT_CALL(fakeFoo, MyMethod(_))
    .WillOnce(Return(false))
    .WillRepeatedly(Return(true));
```

# Mock 의 Test Expectations 설정



- EXPECT_CALL()
  - mock 메소드의 expectation 설정

```
EXPECT_CALL(mock-object, method (matchers)?)
    .With(multi-argument-matcher)   ?
    .Times(cardinality)             ?
    .InSequence(sequences)          *
    .After(expectations)            *
    .WillOnce(action)               *
    .WillRepeatedly(action)         ?
    .RetiresOnSaturation();         ?
```

  - ?: 최대 한번만 사용. *: 여러 번 사용 가능
  - EXPECT_CALL 은 method call 호출 전에 사용
  - Matcher 가 생략된 경우에는 argument 들이 anything-matcher로 지정된 것과 동일((_, _, _, _) for a four-arg method)

# Mock 의 Test Expectations 설정 – 호출 횟수



```
EXPECT_CALL(fakeFoo, MyMethod()).Times(3);
EXPECT_CALL(fakeFoo, MyMethod()).Times(Exactly(3));

//3번 호출을 기대, 첫번 째 호출시에만 10을 리턴
EXPECT_CALL(fakeFoo, MyMethod()).Times(3).WillOnce(Return(10));

EXPECT_CALL(fakeFoo, MyMethod()).Times(AtLeast(1));
EXPECT_CALL(fakeFoo, MyMethod()).Times(AtMost(3));
EXPECT_CALL(fakeFoo, MyMethod()).Times(Between(1, 3));
EXPECT_CALL(fakeFoo, MyMethod()).Times(AnyNumber());   //AtLeast(1) 과 동일
```

- Times 가 없는 경우 WillOnce와 WillRepeatedly 로 호출횟수 결정됨
- 메서드를 호출해야하는 횟수에 관심이 있다면 항상 Times를 사용

# Mock 의 Test Expectations 설정 – 호출 순서(1/3)



- 정해진 순서의 호출 평가: After 와 Sequences 의 사용
- After 절의 사용
  - 먼저 실행될 Expectation 이용

```
using ::testing::Expectation;
...
Expectation init_x = EXPECT_CALL(foo, InitX());
Expectation init_y = EXPECT_CALL(foo, InitY());
EXPECT_CALL(foo, Bar())
     .After(init_x, init_y);
```

  - ExpectationSet 이용 : 선행되어야 할 expectation 이 많은 경우

```
using ::testing::ExpectationSet;
...
ExpectationSet all_inits;
for (int i = 0; i < element_count; i++) {
  all_inits += EXPECT_CALL(foo, InitElement(i));
}
EXPECT_CALL(foo, Bar())
     .After(all_inits);
```

# Mock 의 Test Expectations 설정 – 호출 순서(2/3)



- Sequence : 순서 명시하는 방법

```
using ::testing::Return;
using ::testing::Sequence;
Sequence s1, s2;
...
EXPECT_CALL(foo, Reset())
    .InSequence(s1, s2)
    .WillOnce(Return(true));
EXPECT_CALL(foo, GetSize())
    .InSequence(s1)
    .WillOnce(Return(1));
EXPECT_CALL(foo, Describe(A<const char*>()))
    .InSequence(s2)
    .WillOnce(Return("dummy"));
```

Reset() 은 GetSize() 와 Describe() 호출 이전에 실행되어야 하고, GetSize(), Describe()의 순서와는 무관.

# Mock 의 Test Expectations 설정 – 호출 순서(3/3)



- InSequence : 명시할 순서가 많은 경우

```
using ::testing::InSequence;
{
    InSequence seq;

    EXPECT_CALL(fakeFoo, MyMethod(1));
    EXPECT_CALL(fakeFoo, MyMethod(2)).Times(2);
    EXPECT_CALL(fakeFoo, MyMethod(_));
}
```

# Mock 의 Test Expectations 설정 – LifeCycle



```
EXPECT_CALL(fakeFoo, HttpGet(_)).WillOnce(Return(0));

EXPECT_CALL(fakeFoo, HttpGet(_)).Times(1).WillOnce(Return(1));
```

- 첫번째 호출에 대하여 1을 return하고 두번째는 0을 return?

⟹Fail 발생

⟹호출 횟수가 포화되면 중단 : RetiresOnSaturation

```
EXPECT_CALL(fakeFoo, HttpGet(_))
    .WillOnce(Return(0));

EXPECT_CALL(fakeFoo, HttpGet(_))
    .Times(1)
    .WillOnce(Return(1))
    .RetiresOnSaturation();
```

# Mock 의 Test Expectations 검증



- 테스트 종료시 자동으로 Verify 수행
- 명시적인 검증

```
using ::testing::Mock;
...
// Verifies and removes the expectations on mock_obj;
// returns true if and only if successful.
Mock::VerifyAndClearExpectations(&mock_obj);
...
// Verifies and removes the expectations on mock_obj;
// also removes the default actions set by ON_CALL();
// returns true if and only if successful.
Mock::VerifyAndClear(&mock_obj);
```

# 호출 인수 설정 Matcher



- **EXPECT_CALL**(mock-object, method (matchers)?), **ON_CALL()**
- 특정 값이 사용되는 경우 해당 특정 값의 동작에 대한 기대 값 만 설정
- 개체의 동작을 미세 조정

```
ON_CALL (fakeFoo, MyMethod(_))
    .WillByDefault(Return(42));
EXPECT_CALL(fakeFoo, MyMethod(Eq(100)))
    .Times(AtLeast(1))
    .WillRepetedly(Throw(exception));
```

- 사용
  - 같은 메소드에 다른 행위 지정
  - 복잡한 워크플로어 생성
  - 정확한 expectation 검증
  - Assertion 기능 향상

```
ASSERT_THAT(result, AllOf(NotNull(), StrNe("")));
EXPECT_THAT(result, AnyOf(Gt(100), Le(-100)));
```

# 호출 인수 설정 Matcher – Wildcards



```
// 모든 인수에 대하여
EXPECT_CALL(fakeFoo, MyMethod(_))
//특정 타입에 대하여
EXPECT_CALL(fakeFoo, MyMethod(A<int>()))
EXPECT_CALL(fakeFoo, MyMethod(An<int>()))
```

# 호출 인수 설정 Matcher – Comparisons



```
EXPECT_CALL(fakeFoo, MyMethod(Eq(100))          // arg == 100
EXPECT_CALL(fakeFoo, MyMethod(Ne(100))          // arg != 100
EXPECT_CALL(fakeFoo, MyMethod(Gt(100))          // arg > 100
EXPECT_CALL(fakeFoo, MyMethod(Lt(100))          // arg < 100

EXPECT_CALL(fakeFoo, MyMethod(IsNull())         // arg == NULL/nullptr
EXPECT_CALL(fakeFoo, MyMethod(NotNull())        // arg != NULL/nullptr

EXPECT_CALL(fakeFoo, MyMethod(Ref(str))
```

# 호출 인수 설정 Matcher – Type Based



```
// 정수형 인수 호출
EXPECT_CALL(fakeFoo, MyMethod(An<int>()))
// 50인 정수형 인수 호출
EXPECT_CALL(fakeFoo, MyMethod(TypedEq<int>(50)))
// 50 보다 큰 정수형 인수 호출
EXPECT_CALL(fakeFoo, MyMethod(Matcher<int>(Gt(50))))
```

```
EXPECT_CALL(fakeFoo, MyMethod(_)).WillRepeatedly(Return(1));
EXPECT_CALL(fakeFoo, MyMethod(Gt(10))).WillRepeatedly(Return(5));
EXPECT_CALL(fakeFoo, MyMethod(Gt(20))). .WillRepeatedly(Return(10));
EXPECT_CALL(fakeFoo, MyMethod(A<char>())).WillRepeatedly(Return(200));
```

# 호출 인수 설정 Matcher – String



```
// C string(char *), std::string
ContainsReges(string)
EndsWith(suffix)
HasSubstr(string)
MatchesRegex(string)
StartsWith(suffix)
// C string(char *), std::string, wide strings
StrCaseEq(string)
StrCaseNe(string)
StrEq(string)
StrNe(string)
```

# 호출 인수 설정 Matcher – Combining



AllOf(m1, m2, ...)

AnyOf(m1, m2, ...)

Not(m)

MatcherCast<T>(m)

MatcherSafeCast<T>(m)

```
EXPECT_CALL(fakeFoo, MyMethod(AllOf(NotNull(), Not(StrEq(""")), 5)))
```

# 호출 인수 설정 Matcher – Member



```
Field(&class::field, m)
Property(&class::property, m)

Key(v/m)
Pair(m1, m2)
```

```
EXPECT_CALL(fakeFoo, MyMethod(Key(42)))
```

# 호출 인수 설정 Matcher – Container



```
//Whole matchers
ContainerEq(other)
IsEmpty()
SizeIs(m)
Contains(e)
Each(e)

//Individual items metchers
ElementsAre(e0, e1, ...)
ElementsAreArray({})
Pointwise(m, container)
UnorderedElementsAre(...)
WhenSorted(m)
WhenSortedBy(comparator, m)
```

```
std::vector<std::string> result;
EXPECT_EQ(3, result.size());
EXPECT_EQ("John", result[0]);
EXPECT_EQ("Jane", result[1]);
EXPECT_EQ("Some", result[2]);

ASSERT_THAT(result, ElementsAre("John",
"Jane", "Some"));

ASSERT_THAT(result,
UnorderdElementsAre("John", "Jane", "Some"));
```

# Mock 의 Test Expectations 설정 – 인수



```
EXPECT_CALL(fakeFoo, MyMethod(a, b))
    .With(Eq())
EXPECT_CALL(fakeFoo, MyMethod(a, b, c))
    .With(Allargs(Eq()))
EXPECT_CALL(fakeFoo, MyMethod(a, b, c))
    .With(Args<1,2>(Eq()))
```

- Argument 에 대한 Test Expectation 설정
- Multiargument matcher 사용
  - Eq, Ge, Gt, Le, Lt, Ne

# 호출 인수 설정 Matcher – 그 외



// Floating

**DoubleEq, FloatEq**

**DoubleNear, FloatNear**

**NanSensitive**

// Pointer

**Pointee(m)**

**WhenDynamicCastTo<T>(m)**

// Result of a function

**ResultOf(f, m)**

# 호출 인수 설정 Matcher – 사용자 정의



```
//MATCHER(name, description) {...}
MATCHER(IsEven, “”){ return arg%2==0; }

//MATCHER_P(name, param_name, description) {...}
MATCHER_P(IS_DIVISABLE, value, “”) {return arg%value == 0; }

MATCHER_P2(InCloseRange, low, high, “”) {
    return low <= arg && arg <= high;
}

Template <typename T>
class MatcherInterface {
public:
    virtual ~MatcherInterface();
    virtual bool MatchAndExplain(T x, MatchResultListener* listener) const = 0;
    virtual void DescibeTo(ostream* os) const = 0;
    virtual void DescribeNegationTo(ostream* os) const;
};
```

# Mock 테스트 기본 흐름



1. 사용할 gMock 이름 가져오기.
2. mock 객체 생성
3. 필요한 경우, mock 의 기본 작업 설정.
4. mock의 expectation 설정 : 호출 방법, 동작 내용
5. mock 을 사용한 assertion 으로 mock 테스트
6. mock 객체 destruction 후,
   gMock이 자동으로  expectation 결과 확인

```
using ::testing::Return;                                          // #1

TEST(BarTest, DoesThis) {
  MockFoo foo;                                                    // #2

  ON_CALL(foo, GetSize())                                         // #3
      .WillByDefault(Return(1));
      // ... other default actions ...

  EXPECT_CALL(foo, Describe(5))                                   // #4
      .Times(3)
      .WillRepeatedly(Return("Category 5"));
       // ... other expectations ...

  EXPECT_EQ("good", MyProductionFunction(&foo));                  // #5
}                                                                 // #6
```

# 테스트 예제: GmockOutput



- 다양한 시나리오 테스트

# Write better tests



- Treat Test Code as Production Code
- Use Test Patterns to achieve great readability
- Avoid Unreliable Tests
- Test at The Appropriate Level
- Do Use Test Doubles

출처 : https://www.thoughtworks.com/insights/blog/write-better-tests-5-steps

# 참고 문헌




- https://github.com/google/googletest
- https://github.com/approvals/approvaltests.cpp
- TDD 실천법과 도구, 채수원, 한빛미디어

# THANK YOU.