Unit Test

Chansik Yoon

Goals

- 단위 테스트의 정의를 이해하고, 효과를 이해합니다.
- 단위 테스트 프레임워크의 특징을 이해합니다.
- 단위 테스트 프레임워크를 사용하여 단위 테스트를 작성하는 방법을 배웁니다.
- 단위 테스트 패턴을 통해서 좋은 단위 테스트를 작성하는 방법을 배웁니다.
- 테스트 대역에 대해서 이해하고, 활용 방법을 배웁니다.
- 테스트 가능 설계에 대해서 배웁니다.

소프트웨어 테스팅이란?

- 요구사항에 의해 개발된 산출물이 요구사항과 부합되는지 여부를 검증하기 위한 작업
- 컴포넌트가 기능적으로 잘 동작하는지 뿐 아니라, 고객의 요구 사항이 제대로 이해되어 반영 되었는지 등의 모든 검증 작업
- 많은 소프트웨어 개발자들이 테스트의 필요성과 중요성에 대해서 인정한다.
- 그러나 테스트에 시간을 투자하기 보다는 비지니스 로직에 시간을 투자하거나, 구현체에 시간을 투자하는 것을 더 중요한 일로 생각한다.
- 테스트 되지 않은 시스템은 버그에 의해서 그보다 더 많은 시간을 버그 수정에 쏟아 붓게 되고, 그러다 보면 코드는 점점 더 복잡해질 것이고, 유지보수성은 떨어질 것이다.

소프트웨어 테스트 원칙 7가지

- 원칙 1. 테스팅은 결함이 존재함을 밝히는 것
  - 결함 발견 = 소프트웨어는 결함이 있으며, 장애 발생 가능
  - 결함 없음 = 소프트웨어는 결함을 **찾지 못함**

테스팅은 결함이 존재함을 드러내지만, 결함이 없다는 것을 증명할 수 없다.

소프트웨어 테스트 원칙 7가지

- 원칙 2. 완벽한 테스팅은 불가능하다.
  - 모든 가능성(입력과 사전 조건의 모든 조합)을 테스팅 하는 것은 아주 간단한 프로그램을 제외하고는 불가능하다.
  - 소프트웨어 내부 조건은 무수히 많다. (무한 경로)
  - 소프트웨어 입력 값의 조합은 무수히 많다. (무한 입력값)
  - 소프트웨어 이벤트 발생 순서에 대한 조건은 무한하다. (무한 타이밍)

완벽한 테스팅 대신에, 리스크 분석과 결정된 우선순위에 대해 테스팅 노력을 집중시켜야 한다.

소프트웨어 테스트 원칙 7가지

- 원칙 3. 개발 초기에 테스팅 시작
  - 개발의 시작과 동시에 테스트를 계획하고 접근하는 것을 고려하는 것은 물론, 요구 사항 분석서와 설계 등의 개발 중간 산출물을 분석하여 테스트 케이스를 도출하는 과정을 통해 결함을 발견하는 것을 의미한다.
  - 개발 과정에서 조기 테스트 설계를 하면, 코딩이 끝나자마자 개발 초기부터 준비된 테스트 케이스를 테스트 레벨별로 실행하게 되므로 테스팅 결과를 단기간에 파악할 수 있고, 개발 후반부에 발견될 결함을 예방할 수 있다.
  - 테스팅 팀이 독립적일 경우, 객관적이고 깊이 있는 테스팅 수행 가능
    - Google 은 소프트웨어를 어떻게 테스트하는가? -

소프트웨어 테스트 원칙 7가지

- 원칙 4. 결함 집중 현상
  - 많은 결함들이 특정 기능 또는 모듈에 집중되어 발생된다.
  - 결함이 집중될 가능성이 높은 모듈 특징
    - 복잡한 구조
    - 다른 시스템 또는 모듈과 복잡하고 많은 상호 작용을 함
    - 새롭게 개발
    - 경험이 부족한 개발팀에서 개발한 모듈

소프트웨어 테스트 원칙 7가지

- 원칙 5. 살충제 패러독스
  - 동일한 테스트가 반복적으로 수행된다면, 새로운 버그를 찾아내지 못한다.
  - 테스트 케이스를 정기적으로 리뷰하고, 새로운 내용으로 갱신하거나 개선해야 한다.
  - 테스트 케이스가 공식적인 기법을 사용하여 도출한 경우 테스트 케이스가 정형화되어 시간이 지날수록 새로운 결함을 찾아낼 수 없다.
  - 탐색적 테스팅 등의 새로운 접근법을 통해 테스트 케이스를 더하고 늘려가는 것이 필요하다.

소프트웨어 테스트 원칙 7가지

- 원칙 6. 테스팅은 정황에 의존적
  - 테스트는 정황에 따라 다르게 진행되어야 한다.
  - 원자력 발전소 vs 전자 상거래

  - 변하지 않는 것
    - 테스트 프로젝트 계획
    - 테스트 기법
    - 독립적인 테스트 환경
    - 테스트 팀 조직
    - 리포팅 방법

소프트웨어 테스트 원칙 7가지

- 원칙 7. 오류 부재의 궤변
  - 오류가 없어도 사용자의 요구에 맞지 않는다면 테스팅의 결과는 의미 없다.
  - 사용자 요구에 부합되지 않는 경우 오류가 없어도 제품의 품질이 높다고 할 수 없다.

단위 테스트 정의

- 다른 코드를 호출한 후, 몇가지 가정이 성립되는지 검사하는 코드이다.
- 가정이 성립되지 않으면 단위 테스트는 실패한다.
- 단위 = '메소드' 또는 '함수'
- 단위 테스트는 테스트 대상 시스템(SUT, System Under Test)에 대해 수행된다.

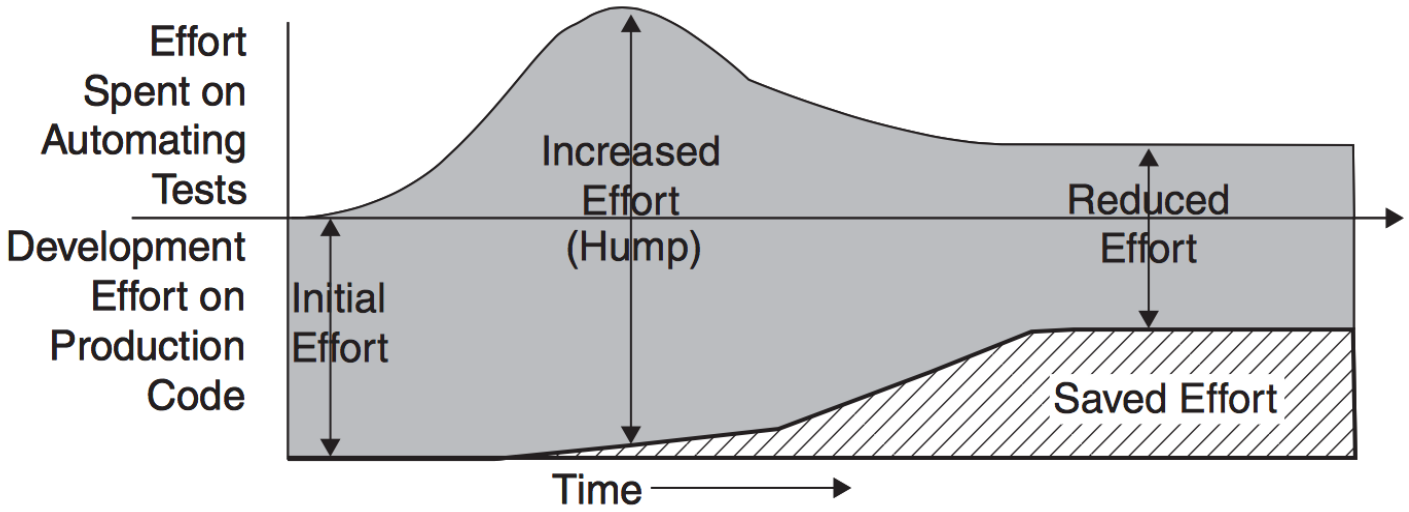
- SUT: 테스트 대상 시스템
  - 테스트 대상 코드(Code Under Test)
  - 테스트 대상 클래스(Class Under Test)

통합 테스트 정의

- 전체 시스템 통합이 완료될 때까지 소프트웨어 하드웨어 구성 요소들이 결합되어 테스트 되는 질서 정연한 테스트 과정 – 빌 허첼(The Complete Guide to Software Testing)

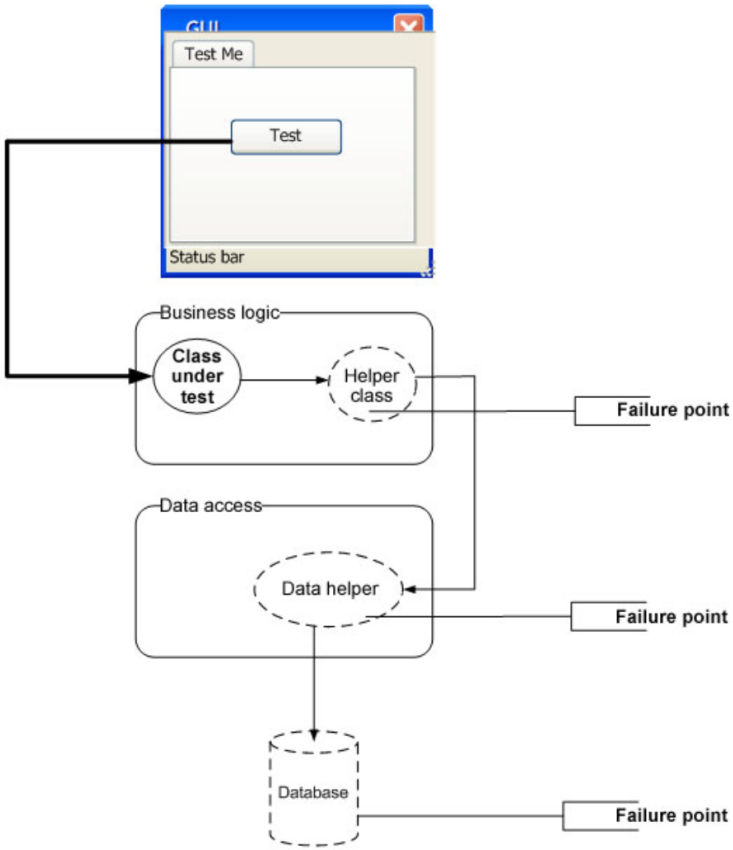
두 개 이상의 독립적인 소프트웨어 모듈을 하나의 그룹으로 테스트 하는 것을 의미한다.

단위 테스트 작성 비용



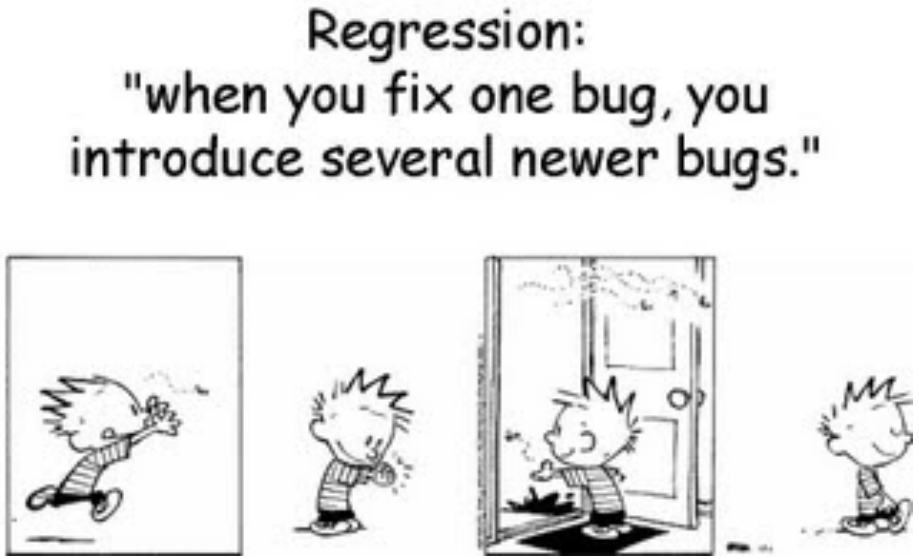
- 단위 테스트를 작성하기 위한 기술과 비용은 발생한다.
- 시간이 지나면, 추가적으로 발생하는 비용 전부를 이득으로 벌충하는 안정적인 상태에 도달한다.

단위 테스트의 가치



- 통합 테스트는 테스트가 실패할 수 있는 지점이 여러곳 존재한다.
- 문제의 근원지를 찾아내기 내는 것이 어렵다.
- 단위 테스트는 하나의 단위에 대한 테스트를 수행함으로써 어디서 문제가 발생했는지를 빠르게 찾을 수 있다.
- 결함 국소화
  - 어떤 테스트가 실패했는지를 보고 버그가 무엇인지를 금방 알아낼 수 있다.

단위 테스트의 가치

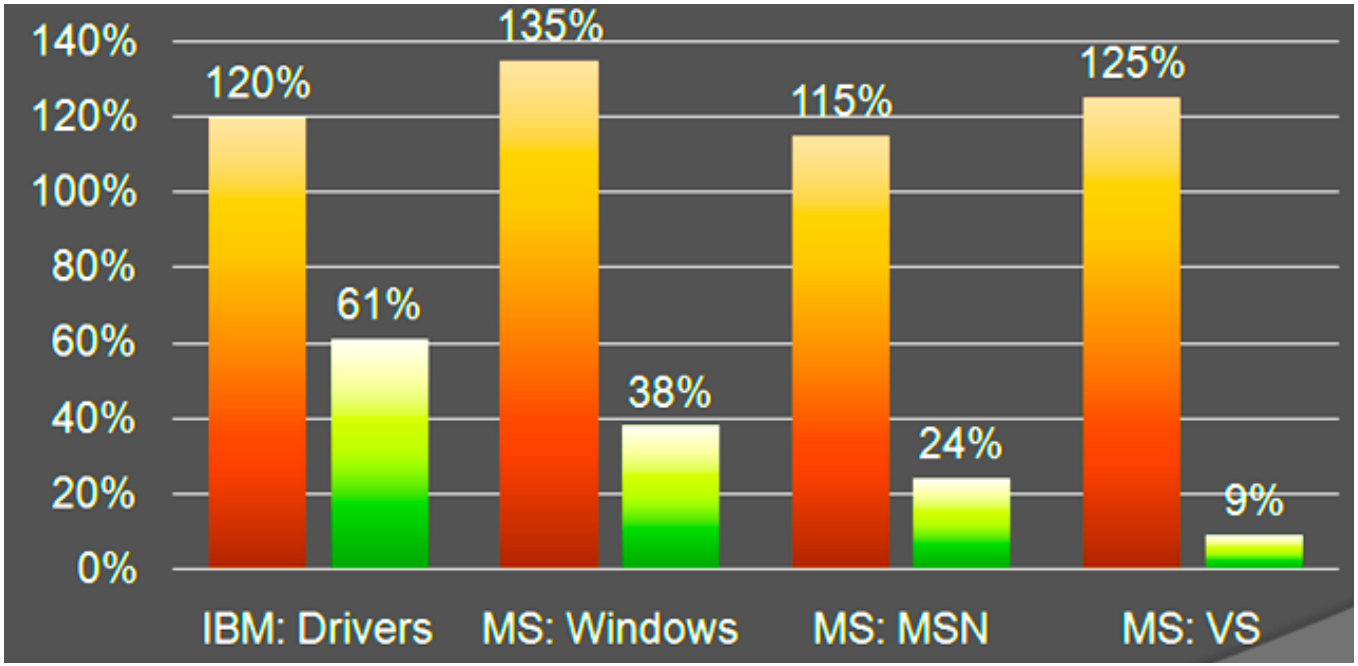


- 코드를 변경한 후 이전에 잘 동작하던 모든 기능들이 아직도 제대로 작동하는지 여부를 어떻게 검증할 수 있는가?
- 단위 테스트는 '우발적 버그 생성' 을 막아준다.

단위 테스트의 가치

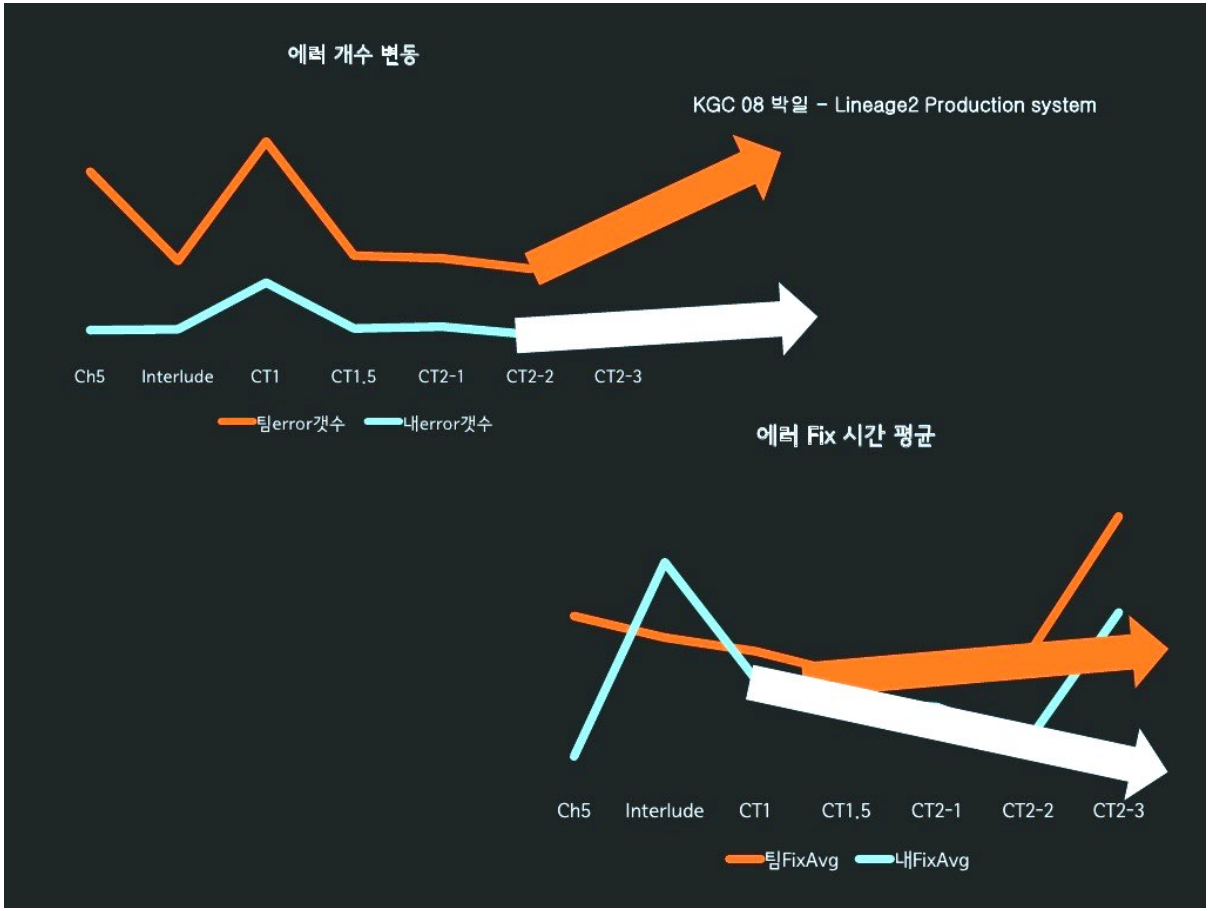
- 레거시 코드 수정의 안정망을 제공한다.
- 회귀 테스트 스위트가 갖춰진 코드로 작업을 한다면 훨씬 빠르게 작업할 수 있으며, 실험적인 방법으로 소프트웨어를 변경할 수 있다.
- **레거시 코드: 회귀 테스트 스위트가 없는 코드를 의미한다.** – 레거시 코드 활용 전략 –

단위 테스트의 가치



- 개발자의 코딩 부담은 증가한다.
- 하지만 결함이 감소하기 때문에, 전체 소프트웨어 개발 비용은 감소한다.

단위 테스트의 가치


에러 개수 변동
KGC 08 박일 – Lineage2 Production system
Ch5
Interlude
CT1
CT1.5
CT2-1
CT2-2
CT2-3
팀error갯수
내error갯수
에러 Fix 시간 평균
Ch5
Interlude
CT1
CT1.5
CT2-1
CT2-2
CT2-3
팀FixAvg
내FixAvg

단위 테스트의 가치

- 테스트는 SUT를 이해하는 데 도움이 되어야 한다.
- 문서로서의 테스트
  - 테스트가 없다면… 결과가 어떻게 될까? 를 알기 위해 SUT 코드를 직접 분석해야 한다.
  - 테스트가 있다면 그에 해당하는 문서로서의 테스트를 이용하면 된다.

단위 테스트 작성 방법

- **단위 테스트는 실행하기 쉬워야 한다.**
  1. 완전 자동 테스트: 수동 조정 없이 돌려볼 수 있어야 한다.
  2. 자체 검사 테스트: 직접 검사해보지 않아도 에러를 찾아내 알려줄 수 있어야 한다.
  3. 반복되는 테스트: 여러번 돌려 봐도 같은 결과를 얻을 수 있어야 한다.
  4. 독립적인 테스트: 테스트 별로 돌려볼 수 있어야 한다.

- **단위 테스트는 만들고 유지하기 쉬어야 한다.**
  - 단순한 테스트: 단위 테스트는 작아야 하고, 한번에 하나만 테스트 해야 한다.

- **단위 테스트에 필요한 유지 보수 비용이 최소화 되어야 한다.**
  - 하나를 변경하였을 때 영향 받는 테스트 수가 최소가 될 수 있는 방법으로 테스트를 작성해야 한다.
  - SUT를 최대한 격리시켜, 테스트 환경이 변경되어도 테스트에는 아무런 영향이 없게 해야 한다.

테스트를 작성하는 견해

먼저 작성할 것인가? 나중에 작성할 것인가?

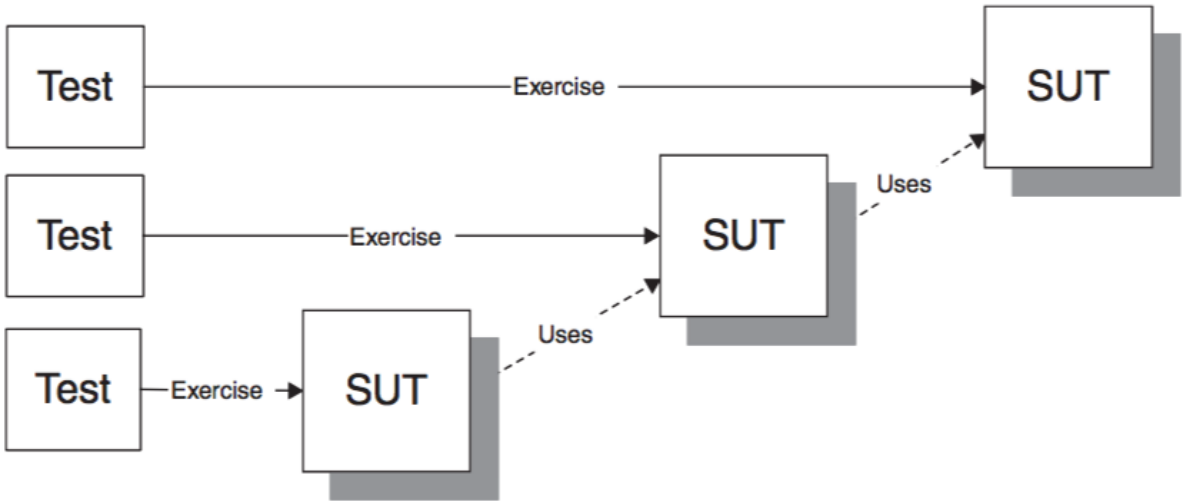
Last	First
전통적인 소프트웨어 개발	애자일 진영
단위 테스트 작성이 어렵다 (제품 코드를 건드리지 않으면서, 테스트 작성하기가 어렵다)	시스템 설계는 저절로 테스트 하기 쉽게 된다.
	제품 코드가 간결해진다.

테스트를 작성하는 견해

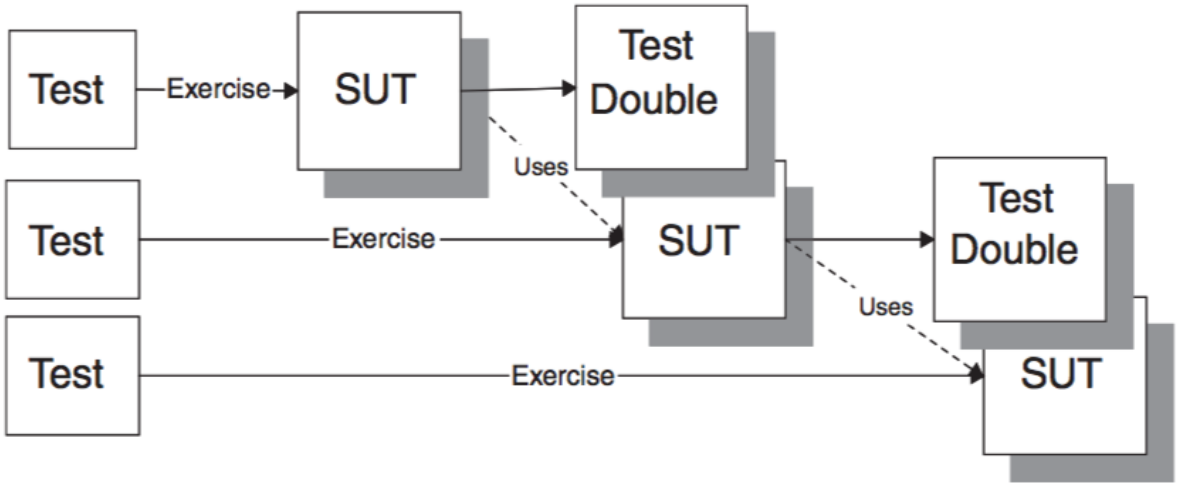
내부 모듈부터 작성할 것인가?
외부 모듈부터 작성할 것인가?

안에서 밖으로	밖에서 안으로
가장 내부 컴포넌트부터 개발을 시작해 이를 기반으로 점차 사용자 인터페이스까지 개발	밖에서부터 개발을 시작해 어떤 의존 컴포넌트가 필요한지를 봐서 내부로 개발을 진행해 나간다.
	소프트웨어 개발자처럼 생각 하기 전에 고객처럼 생각하게 해준다.
	테스트는 이들 사용 패턴을 찾아주고, 구현해야 하는 다양한 시나리오를 열거 할 수 있다.
의존 관계를 해결하지 않아도 된다.	의존 문제를 해결해야 한다.

테스트를 작성하는 견해


Test
Exercise
SUT
Test
Exercise
SUT
Uses
Test
Exercise
SUT
Uses

테스트를 작성하는 견해


Test
Exercise
SUT
Test Double
Uses
Test
Exercise
SUT
Test Double
Uses
Test
Exercise
SUT

테스트를 작성하는 견해

상태를 검증할 것인가?
동작을 검증할 것인가?

상태 검증	동작 검증
SUT 를 특정 상태에 두고, 실행한 후 SUT 가 원했던 상태인지 검증	SUT 의 시작과 끝의 상태 뿐 아니라, SUT 가 내부적으로 호출하는 것까지 검증
	모의 객체나 테스트 스파이를 많이 사용
쉽게 접근 가능하다.	어렵다.

xUnit Test Framework

xUnit is the collective name for several unit testing frameworks that derive their structure and functionality from Smalltalk's SUnit. SUnit, designed by Kent Beck in 1998, was written in a highly structured object-oriented style, which lent easily to contemporary languages such as Java and C#. Following its introduction in Smalltalk the framework was ported to Java by Kent Beck and Erich Gamma and gained wide popularity, eventually gaining ground in the majority of programming languages in current use. The names of many of these frameworks are a variation on "SUnit", usually replacing the "S" with the first letter (or letters) in the name of their intended language ("JUnit" for Java, "RUnit" for R etc.). These frameworks and their common architecture are collectively known as “xUnit".

xUnit Test Framework

- Test Runner
  - A test runner is an executable program that runs tests implemented using an xUnit framework and reports the test results.

- Test Case
  - A test case is the most elemental class. All unit tests are inherited from here.

- Test Fixtures
  - A test fixture (also known as a test context) is the set of preconditions or state needed to run a test. The developer should set up a known good state before the tests, and return to the original state after the tests.

- Test Suites
  - A test suite is a set of tests that all share the same fixture. The order of the tests shouldn't matter.

xUnit Test Framework

- Test Execution
  - The execution of an individual unit test proceeds as follows:

```
setup(); /* First, we should prepare our 'world' to make an isolated environment for testing */
...
/* Body of test - Here we make all the tests */
...
teardown(); /* At the end, whether we succeed or fail, we should clean up our 'world' to
               not disturb other tests or code */
```

The setup() and teardown() methods serve to initialize and clean up test fixtures.

- Test Result Formatter
  - A test runner produces results in one or more output formats. In addition to a plain, human-readable format, there is often a test result formatter that produces XML output. The XML test result format originated with JUnit but is also used by some other xUnit testing frameworks, for instance build tools such as Jenkins and Atlassian Bamboo.

xUnit Test Framework

- Assertions
  - An assertion is a function or macro that verifies the behavior (or the state) of the unit under test. Usually an assertion expresses a logical condition that is true for results expected in a correctly running system under test (SUT). Failure of an assertion typically throws an exception, aborting the execution of the current test.

xUnit frameworks

- https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

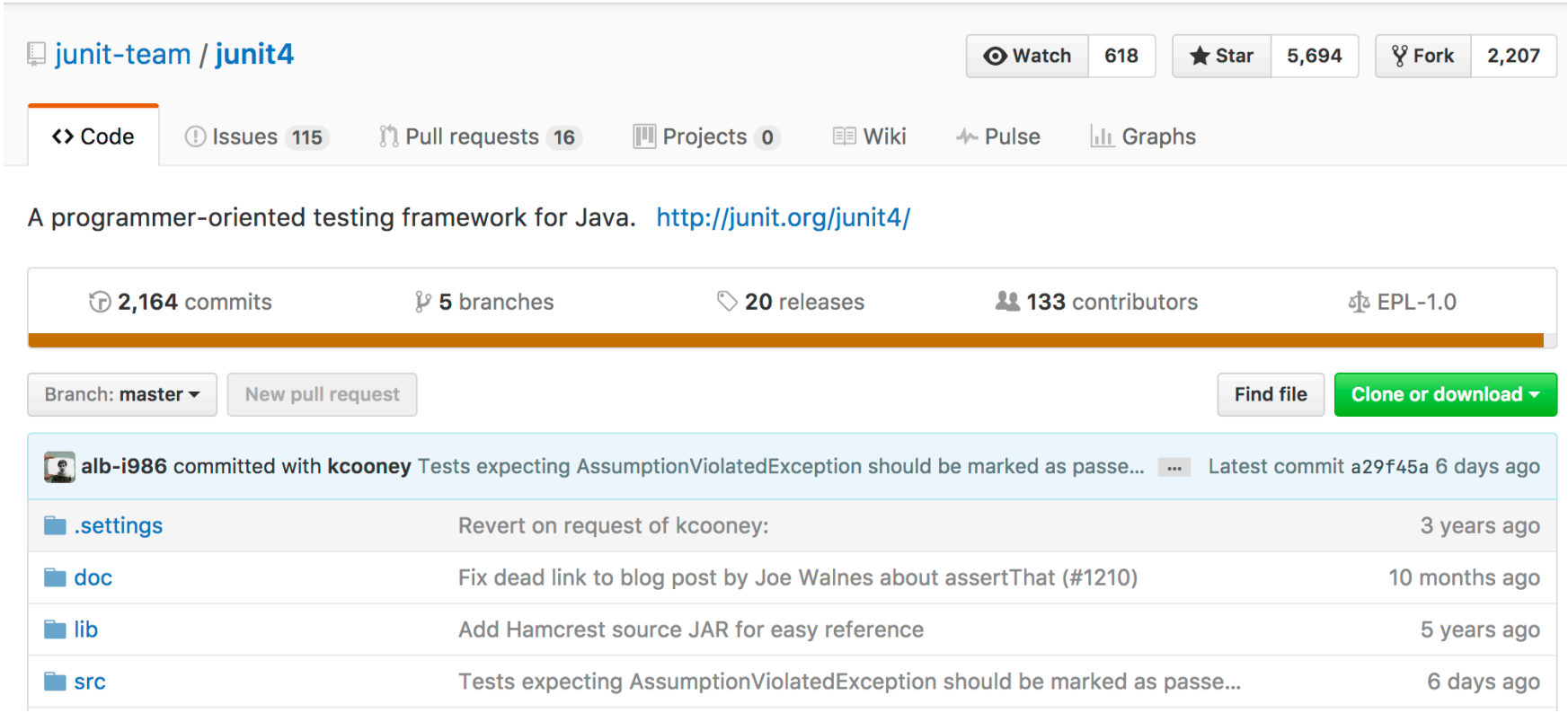
C++: Google Test, CppUnit
Java: JUnit, TestNG
C#: NUnit

Google Test Framework

https://github.com/google/googletest

JUnit Test Framework

https://github.com/junit-team/junit4



Chapter 1. 테스트를 구성하는 방법

```
// 1. gtest를 사용하기 위해서는 아래 헤더파일을 꼭 포함해야 합니다.
#include "gtest/gtest.h"

// 절대 실패하지 않는 테스트
// 1. 만약 테스트를 작성하는 도중이라면, 실패하는 테스트로 작성해야 한다.
// 2. 실패한 테스트의 이유는 반드시 작성해야 한다.

// - TEST(TestCaseName, TestName)
TEST(SampleTestCase, SampleTest) {
          FAIL() << "작성 중 입니다.";
}

// main 함수에서 특별히 할 일이 없다면, gtest_main.cc도 라이브러리에 포함하면 된다.
#if 0
int main(int argc, char* argv[])
{
          // 2. RUN_ALL_TESTS()를 수행하기 전에 gtest 라이브러리를 초기화해야 합니다.
          testing::InitGoogleTest(&argc, argv);

          // 3. 모든 단위 테스트를 수행하는 명령입니다.
          return RUN_ALL_TESTS();
}
#endif
```

Chapter 2. 3A

```
TEST(CalcTestCase, CalcTest1) {
        // Arrange
        Calculator* calculator = new Calculator;

        // Act
        calculator->enter(2);
        calculator->pressPlus();
        calculator->enter(2);
        calculator->pressEqual();

        // Assert
        if (calculator->display() != 4)
                FAIL();
        else
                SUCCEED();
}
```

유닛 테스트를 구성하는 방법 1
1. 객체를 생성하고, 필요한 경우 적절하게 설정하고 준비한다.(Arrange)
2. 객체에 작용을 가한다.(Act)
3. 기대하는 바를 단언한다.(Assert)

Chapter 2. 3A

```
// 테스트가 많아지면, 각 테스트가 어떤 시나리오로 테스트를 하는지 여부를
// 구분할 수 있도록 테스트의 이름을 지어야 한다.(애매모호한 테스트)
// ex) 테스트대상메소드_시나리오_기대값
TEST(CalcTestCase, getResult_AddingTwoPlusTwo_ReturnsFour) {
        Calculator* calculator = new Calculator;

        calculator->enter(2);
        calculator->pressPlus();
        calculator->enter(2);
        calculator->pressEqual();

        // Assert
        // 1. 테스트 함수는 별도의 테스트가 불가능하기 때문에, 조건문이나
        //    반복문은 피해야 한다.
        // 2. 기대하는 바를 단언하기 위해서는 gtest가 제공하는 단언 전용 매크로를
        //    사용하면 된다.
        // 3. ASSERT_XX(기대값, 실제값)
        //   EQ(==), NE(!=), LT(<), GT(>), LE(<=), GE(>=)
                // ASSERT_EQ(4, calculator->display());
        // 4. 실패의 이유 또한 명확하게 표현하자.
        ASSERT_EQ(4, calculator->display()) << "When Adding 2+2";
}
```

Chapter 3. 테스트 픽스쳐

픽스쳐(Fixture)
1. 정의: xUnit에서는 SUT를 실행하기 위해서 준비해야 하는 모든 것을
테스트 픽스쳐라고 부릅니다.

2. 픽스쳐를 설정하는 모든 로직 부분을 '픽스쳐 설치'라고 합니다.

Chapter 3. 테스트 픽스쳐

픽스쳐 설치 방법 1 – 인라인 픽스쳐 설치
- 모든 픽스쳐 설치를 테스트 함수 안에서 처리한다.

장점: 픽스쳐 설치와 검증 로직이 테스트 안에 존재하기 때문에 인과 관계를 쉽게 파악할 수 있다.

단점: 모든 테스트 함수안에서 '코드 중복'이 발생한다.

```
TEST(CalcTestCase, CalcTest1) {
        // Arrange
        Calculator* calculator = new Calculator;

        // Act
        calculator->enter(2);
        calculator->pressPlus();
        calculator->enter(2);
        calculator->pressEqual();

        // Assert
        if (calculator->display() != 4)
                FAIL();
        else
                SUCCEED();
}
```

Chapter 3. 테스트 픽스쳐

픽스쳐 설치 방법 2 – 위임 설치

1. 동일한 픽스쳐를 가진 테스트 함수를 클래스로 묶는다.
2. 픽스쳐 설치에 관한 중복되는 코드를 함수로 제공한다.

```
class CalculatorTest : public ::testing::Test
{
protected:
        Calculator* create() {
                return new Calculator;
        }
};

TEST_F(CalculatorTest, CalcTest1) {
        // Arrange
        Calculator* calculator = create();

        // Act
        calculator->enter(2);
        calculator->pressPlus();
        calculator->enter(2);
        calculator->pressEqual();
        ...
```

Chapter 3. 테스트 픽스쳐

픽스쳐 설치 방법 3 – 암묵적 설치
- 여러 테스트에서 같은 테스트 픽스쳐를 SetUp() 메소드에서 생성한다.
- 장점: 테스트 코드 중복을 제거하고, 꼭 필요하지 않은 상호 작용 코드를 캡슐화 할 수 있다.
- 단점: 픽스쳐 설치 코드가 테스트 함수 밖에 존재하기 때문에 코드를 이해하기 어렵다.

```
class CalculatorTest : public ::testing::Test
{
protected:
        Calculator* calculator;

        virtual void SetUp()
        {
            calculator = new Calculator;
        }

        virtual void TearDown()
        {
            delete calculator;
        }
};
```

Chapter 3. 테스트 픽스쳐

4단계 테스트(Four-Phase Test)

1단계: 테스트 픽스쳐를 설치하거나 실제 결과를 관찰하기 위해
필요한 것을 집어넣는 작업을 한다.

2단계: SUT와 상호 작용한다.

3단계: 기대 결과를 확인한다.

4단계: 테스트 픽스쳐를 해체하여, 테스트 시작 상태로 되돌려 놓는다.

```
TEST_F(CalculatorTest, CalcTest1) {
        // Act
        calculator->enter(2);
        calculator->pressPlus();
        calculator->enter(2);
        calculator->pressEqual();

        // Assert
        ASSERT_EQ(4, calculator->display()) << "When Adding 2+2";
}
```

Chapter 4. GTest 기능

1. EXPECT_XX
- 테스트 함수 안에서 첫번째 테스트가 실패할 경우, 나머지 검증은 수행하지 않고 테스트는 실패한다.
- ASSERT_XX 대신 EXPECT_XX 을 사용하면 나머지 검증을 수행할 수 있다.

```
TEST(GoogleTestSample, Sample1)
{
        int expected = 3;
        int actual = 4;

        EXPECT_EQ(expected, actual) << "REASON1";
        ASSERT_EQ(expected, actual) << "REASON2";
}
```

Chapter 4. GTest 기능

2. 문자열 비교 단언(기대)

```
TEST(GoogleTestSample, Sample2)
{
        string s1 = "hello";
        string s2 = "hello";

        ASSERT_EQ(s1, s2); // s1 == s2;

        const char* s3 = "hello";
        const char* s4 = s1.c_str();

        // 문자열 비교는 ASSERT_STREQ 사용해야 한다.
        // ASSERT_EQ(s3, s4);
        ASSERT_STREQ(s3, s4);
}
```

Chapter 4. GTest 기능

3. 부동 소수점 비교 단언(기대)

```
TEST(GoogleTestSample, Sample3)
{
        // ASSERT_EQ(0.7, 0.1 * 7);
        ASSERT_DOUBLE_EQ(0.7, 0.1 * 7);  // 4 ULP's

        // 직접 오차 범위를 지정할 수 있다.
        ASSERT_NEAR(0.7, 0.1 * 7, 0);
}
```

Chapter 4. GTest 기능

4. 예외 테스트

```
TEST(GoogleTestSample, ExceptionTest)
{
	string filename = "";

	try
	{
		foo(filename);
		FAIL() << "기대한 예외가 발생하지 않았음!!";
	}
	catch (invalid_argument& e)
	{
		SUCCEED();
	}
	catch (...)
	{
		FAIL() << "예상했던 예외가 발생하지 않았다.";
	}
}
```

Chapter 4. GTest 기능

4. 예외 테스트

```
TEST(GoogleTestSample, ExceptionTest2)
{
        string filename = "";

        EXPECT_THROW(foo(filename), invalid_argument);
        EXPECT_ANY_THROW(foo(filename));
}
```

Chapter 4. GTest 기능

5. 테스트 비활성화
   - 테스트를 주석을 통해서 막는 것보다는 테스트 프레임워크에서 제공하는 비활성화 기능을 사용해야 한다.
   - 테스트 이름 또는 테스트 픽스쳐 클래스 이름 앞에 DISABLED_XXXX 를 붙이면 된다.

```
TEST(GoogleTestSample, DISABLED_SampleTest4)
{
        //...
}
```

Chapter 5. 스위트 픽스쳐

SUT

```
class Database
{
public:
        // 시간이 걸리는 작업.
        void netConnect() {
                delay(3);
        }

        void netDisconnect() {
                delay(1);
        }

        void logIn(const string& id, const string& password) {}
        void logOut() {}

        bool isLogin() { return false; }
};
```

Chapter 5. 스위트 픽스쳐

```
class DatabaseTest : public ::testing::Test
{
public:
        static string TEST_ID;
        static string TEST_PASSWORD;

protected:
        Database* database;

        virtual void SetUp()
        {
                database = new Database();
                database->netConnect();
        }

        virtual void TearDown()
        {
                database->netDisconnect();
                delete database;
        }
};
```

Chapter 5. 스위트 픽스쳐

암묵적 설치/해체 함수는 테스트 메소드가 추가될 때마다 수행된다. - SetUp(), TearDown()
느린 테스트(Slow Test): 테스트가 너무 느려서 개발자들이 SUT가 변경되어도 매번
테스트를 실행하지 않는다.
(테스트를 실행하는 개발자의 생산성을 떨어뜨려, 비용을 발생시킨다)

```
TEST_F(DatabaseTest, DatabaseLoginTest)
{
        database->logIn(DatabaseTest::TEST_ID, DatabaseTest::TEST_PASSWORD);
        ASSERT_TRUE(database->isLogin());
}

TEST_F(DatabaseTest, DatabaseLogoutTest)
{
        database->logIn(DatabaseTest::TEST_ID, DatabaseTest::TEST_PASSWORD);
        database->logOut();
        ASSERT_FALSE(database->isLogin());
}
```

Chapter 5. 스위트 픽스쳐

해결 방법: 스위트 픽스쳐 설치
: 정적 멤버 함수인 SetUpTestCase() / TearDownTestCase()를 통해서 설치/해체 할 수 있다.

```
class DatabaseTest : public ::testing::Test
{
public:
        static string TEST_ID;
        static string TEST_PASSWORD;

protected:
        static Database* database;
        static void SetUpTestCase()
        {
                database = new Database();
                database->netConnect();
        }
        static void TearDownTestCase()
        {
                database->netDisconnect();
                delete database;
        }
};
```

Chapter 5. 스위트 픽스쳐

문제점
: 테스트 케이스는 더 이상 독립적이지 않다.
공유 픽스쳐 상태에 따라서 테스트가 성공할 수도 실패할 수 있는
[변덕스러운 테스트]의 문제가 발생할 수 있다.

Chapter 6. 전역 픽스쳐

프로그램 단위의 픽스쳐를 설치/해체 할 수 있는 Google Test 고유의 기능

```
class MyTestEnvironment : public ::testing::Environment
{
public:
        void SetUp()
        {
                cout << "전역 픽스쳐 설치" << endl;
        }

        void TearDown()
        {
                cout << "전역 픽스쳐 해체" << endl;
        }
};

int main(int argc, char* argv[])
{
        testing::InitGoogleTest(&argc, argv);
        testing::AddGlobalTestEnvironment(new MyTestEnvironment);
        return RUN_ALL_TESTS();
}
```

Chapter 6. 전역 픽스쳐

main을 사용하지 않을 경우, 전역 변수를 통해 설치할 수 있다.

```
::testing::Environment* const env
= ::testing::AddGlobalTestEnvironment(new MyTestEnvironment);
```

Chapter 6. 전역 픽스쳐

main을 사용하지 않을 경우, 전역 변수를 통해 설치할 수 있다.

```
::testing::Environment* const env
= ::testing::AddGlobalTestEnvironment(new MyTestEnvironment);
```

Chapter 7. Time Test

JUnit 4/5 에서는 지원하는 기능이지만, Google Test 에서는 지원하지 않는다.

```
class TimeCriticalTest : public ::testing::Test
{
protected:
        time_t startTime;

        static const int PIVOT = 3;

        virtual void SetUp() { startTime = time(0); }
        virtual void TearDown()
        {
                time_t endTime = time(0);
                time_t duration = endTime - startTime;

                EXPECT_TRUE(duration < PIVOT) << "테스트가 제한시간을 초과하였습니다 - " << PIVOT;
        }
};

TEST_F(TimeCriticalTest, Sample)
{
        Sleep(1000 * 3);
}
```

Chapter 7. Time Test

재사용 가능한 테스트 픽스쳐 – CRTP

```
template <typename T, int N>
class TimeCriticalTest : public ::testing::Test
{
protected:
        time_t startTime;
        static const int PIVOT = N;
        virtual void SetUp() { startTime = time(0); }
        virtual void TearDown()
        {
            time_t endTime = time(0);
            time_t duration = endTime - startTime;
            EXPECT_TRUE(duration < PIVOT) << "테스트가 제한시간을 초과하였습니다 – " << PIVOT <<
"sec";
        }
};

class MyTest : public TimeCriticalTest<MyTest, 1> {};
TEST_F(MyTest, foo)
{
        Sleep(1 * 1000);
}
```

Chapter 7. Memory Leak Test

메모리 릭 감지 테스트 – operator new / operator delete

```
class Image
{
private:
	static int allocObjectCount;
public:
	static int getAllocObjectCount() { return allocObjectCount; }

	void* operator new(size_t sz)
	{
		++allocObjectCount;
		return malloc(sz);
	}

	void operator delete(void* p, size_t)
	{
		--allocObjectCount;
		free(p);
	}
};

int Image::allocObjectCount = 0;
```

Chapter 7. Memory Leak Test

메모리 릭 감지 테스트 – operator new / operator delete

```
class ImageTest : public ::testing::Test
{
protected:
        int initAllocCount;

        virtual void SetUp() { initAllocCount = Image::getAllocObjectCount(); }
        virtual void TearDown()
        {
                int diff = Image::getAllocObjectCount() - initAllocCount;
                EXPECT_EQ(diff, 0) << " 메모리 누수 발생 - " << diff << " 객체";
        }
};

TEST_F(ImageTest, foo)
{
        Image* p = new Image;

        // 이미지 객체의 메모리 누수!
        // delete p;
}
```

Chapter 7. Memory Leak Test

조건부 컴파일을 통한 Leak Test 코드 삽입 방법

```
#define LEAK_TEST
#include "Image.h"

#ifdef LEAK_TEST
#define DECLARE_LEAK_TEST()                    \
private:                                       \
static int allocObjectCount;                   \
public:                                        \
static int getAllocObjectCount();              \
void* operator new(size_t sz)                  \
{                                              \
        ++allocObjectCount;                    \
    return malloc(sz);                         \
}                                              \
void operator delete(void* p, size_t)          \
{                                              \
        --allocObjectCount;                    \
        free(p);                               \
}
```

Chapter 7. Memory Leak Test

조건부 컴파일을 통한 Leak Test 코드 삽입 방법

```
#define IMPLEMENT_LEAK_TEST(classname)          \
        int classname::allocObjectCount = 0;    \
        int classname::getAllocObjectCount() { return allocObjectCount; }
#endif

class Image
{
    DECLARE_LEAK_TEST()
};

IMPLEMENT_LEAK_TEST(Image)
```

Chapter 8. Parameterized Test

입력 데이터를 바꿔가면서 수차례 반복 검사하는 데이터 중심 테스트의 중복을 없애주는 기법

```
class PrimeTest : public testing::TestWithParam<int>
{
protected:
        virtual void SetUp() {}
        virtual void TearDown() {}
};

INSTANTIATE_TEST_CASE_P(PrimeValues, PrimeTest,
        ::testing::Values(3, 4, 11, 23));

TEST_P(PrimeTest, isPrime)
{
        ASSERT_TRUE(IsPrime(GetParam()));
}

TEST_P(PrimeTest, isPrime2)
{
        ASSERT_TRUE(IsPrime(GetParam()));
}
```

Chapter 9. Test Specific Subclass

자동차의 시동을 걸었을 때, 엔진이 제대로 시작했는지 여부를 검증하고자 한다.

문제점 1. Engine 객체는 검증할 수 있는 상태가 없다.

2. Engine 객체는 상태를 제공하는 메소드가 없다.

```
class Engine {
public:
    virtual void start() { cout << "Engine Start" << endl; }
};

class Car {
    Engine* engine;
public:
    Car(Engine* p) : engine(p) {}
    void start() { engine->start(); }
};
```

Chapter 9. Test Specific Subclass

테스트 전용 하위 클래스

```
class TestEngine : public Engine
{
        // 테스트를 위한 상태
        bool _isRunning;
public:
        TestEngine() : _isRunning(false) {}

        // 상태를 얻기 위한 메소드
        bool isRunning() { return _isRunning; }
        void start()
        {
                Engine::start();  // 원래 기능 사용.
                _isRunning = true;
        }
};
```

Chapter 9. Test Specific Subclass

테스트 전용 하위 클래스

```
class CarTest : public ::testing::Test {};

TEST_F(CarTest, EngineIsStartWhenCarStarts)
{
        TestEngine engine;
        Car car(&engine);

        car.start();

        ASSERT_TRUE(engine.isRunning());
}
```

Chapter 10. Test Specific Subclass

테스트 전용 하위 클래스 – SUT의 필드가 protected 인 경우

```
class User
{
protected:
    int level;
    int getLevel() { return level; }

public:
    User() : level(1) {}
    void levelUp() { ++level; }
};
```

Chapter 10. Test Specific Subclass

테스트 전용 하위 클래스 – SUT의 필드가 protected 인 경우

```
class TestUser : public User
{
public:
    using User::getLevel;
};

TEST(UserTest, LevelUpTest)
{
        TestUser user;

        user.levelUp();
        ASSERT_EQ(user.getLevel(), 2);
}
```

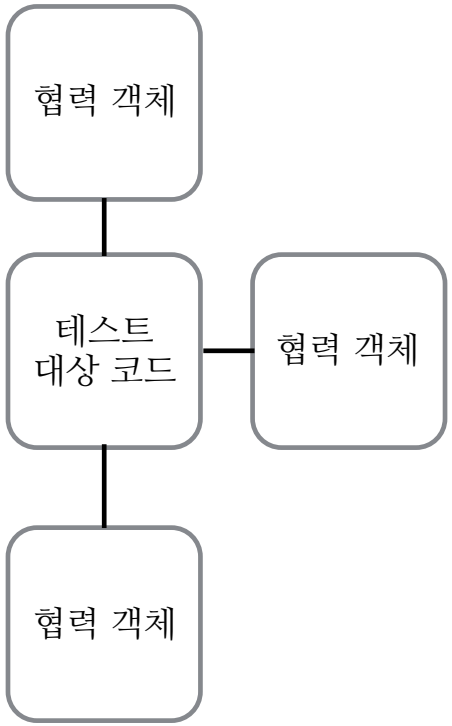
Chapter 11. Test Specific Subclass

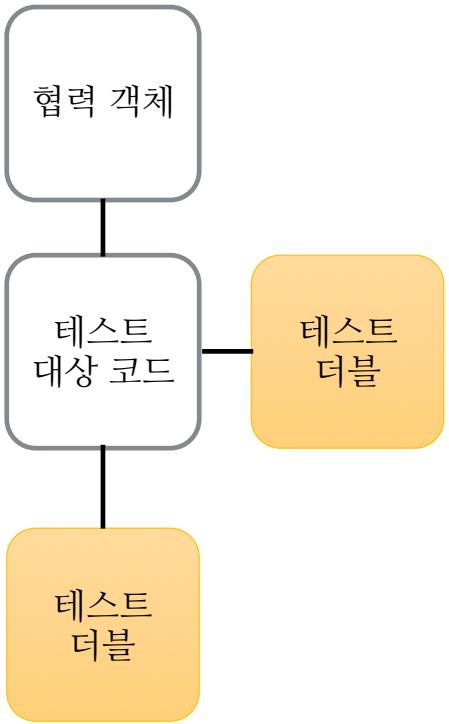
테스트 전용 하위 클래스 -FRIEND_TEST

```
class User
{
private:
    string name;

public:
    FRIEND_TEST(UserTest, CheckNameIsEmpty);
};

class UserTest : public testing::Test
{
};

TEST_F(UserTest, CheckNameIsEmpty)
{
    User user;
    ASSERT_TRUE(user.name.empty());
}
```

테스트 대역(Test Double)

- 테스트 시에 실제 객체를 대신할 수 있는 객체를 총칭하여 테스트 대역이라고 한다.
- 영화에서의 스턴트 대역(stunt double) 에서 유래.
- 테스트 대상 코드와 협력 객체를 잘 분리하여 테스트 환경 전반을 통제하는 것이 그 용도이다.





Chapter 12. Test Double – Stub

```
class ConnectionError : public exception
{
public:
        virtual const char* what() {
                return "Bad Connection Error!!";
        }
};

struct IConnection {
        virtual void move(int x, int y) = 0;
        virtual ~IConnection() {}
};

class Connection : public IConnection
{
public:
        void move(int x, int y)
        {
            // 연결이 끊겼을 때 요청시, 예외를 던져야 한다.
        }
};
```

Chapter 12. Test Double – Stub

```
// 연결이 성립되지 않았을 때, move()를 호출하면 예외를 던지는지 여부를 테스트하고 싶다.
TEST_F(UserTest, TestMoveWhenConnectionIsDisconnectedThrowsException)
{
        Connection connection;
        User user(&connection);

        ASSERT_THROW(user.move(10, 20), ConnectionError);
}
```

Chapter 12. Test Double – Stub

목적 – 특수한 상황을 시뮬레이션 한다.

```
class BadConnection : public IConnection
{
public:
        void move(int x, int y) {
                throw ConnectionError();
        }
};

TEST_F(UserTest, TestMoveWhenConnectionIsDisconnectedThrowsException)
{
        BadConnection connection;
        User user(&connection);

        ASSERT_THROW(user.move(10, 20), ConnectionError);
}
```

Chapter 13. Test Double – Fake Object

```
class UserInfo
{
        string id_;
        int level_;
        int gold_;
public:
        UserInfo(string id, int level, int gold)
                : id_(id), level_(level), gold_(gold) {}

        string id() { return id_; }
        int level() { return level_; }
        int gold()  { return gold_; }

        // ASSERT_EQ(==)를 사용하기 위해서는 이항 연산자(==)를 재정의해야 한다.
        friend bool operator==(const UserInfo& a, const UserInfo& b) {
                return (a.id_ == b.id_) && (a.level_ == b.level_) && (a.gold_ == b.gold_);
        }
};
```

Chapter 13. Test Double – Fake Object

실제 데이터 베이스에 접근해서 데이터를 읽고 쓰는 테스트를 작업하는 것은 많은 시간이 소모될 수 밖에 없다.

```
class Database {
public:
        void save(string id, UserInfo* ui) {
                // 사용자 정보를 데이터 베이스에 저장한다.
                Sleep(1000 * 2);
        }

        UserInfo* load(string id) {
                // 사용자 정보를 데이터 베이스에서 불러온다.
                Sleep(1000 * 2);
                return 0;
        }
};
```

Chapter 13. Test Double – Fake Object

```
class UserManager {
        Database* database;
public:
        UserManager() {
                database = new Database;
        }

        void save(UserInfo* ui) {
                database->save(ui->id(), ui);
        }

        UserInfo* load(string id) {
                return database->load(id);
        }
};
```

Chapter 13. Test Double – Fake Object

진짜 데이터베이스 처럼 동작하는 가짜 객체를 도입하여 느린 테스트의 문제를 해결 가능하다.

```
class UserManagerTest : public ::testing::Test
{
protected:
        static const string TEST_ID;
        static const int TEST_LEVEL = 10;
        static const int TEST_GOLD = 1000;
};

const string UserManagerTest::TEST_ID = "TEST_ID";
// 느린 테스트!
TEST_F(UserManagerTest, SaveLoadUserInfo) {
        UserManager manager;

        UserInfo ui(TEST_ID, TEST_LEVEL, TEST_GOLD);
        manager.save(&ui);

        UserInfo *actual = manager.load(TEST_ID);
        ASSERT_NE(nullptr, actual) << "로딩 실패";
        ASSERT_EQ(ui, *actual) << "잘못된 정보가 로딩되었습니다.";
}
```

Chapter 13. Test Double – Fake Object

목적 – 느린 테스트를 개선한다.

```
class MemoryDatabase : public IDatabase {
        map<string, UserInfo*> datas;
public:
        void save(string id, UserInfo* ui) {
                datas[id] = ui;
        }

        UserInfo* load(string id) {
                return datas[id];
        }
};
TEST_F(UserManagerTest, SaveLoadUserInfo2) {
        MemoryDatabase mdb;
        UserManager manager(&mdb);

        UserInfo ui(TEST_ID, TEST_LEVEL, TEST_GOLD);
        manager.save(&ui);
        UserInfo *actual = manager.load(TEST_ID);
        ASSERT_NE(nullptr, actual) << "로딩 실패";
        ASSERT_EQ(ui, *actual) << "잘못된 정보가 로딩되었습니다.";
}
```

Chapter 14. Test Double – Spy

메소드를 호출하였을 때 발생하는 부작용을 관찰할 수 없어 테스트 안된 요구사항이 있다.

```
enum Level {
    INFO, WARN, CRITICAL
};

struct DLogTarget {
    virtual void write(Level level, const string& message) = 0;
    virtual ~DLogTarget() {}
};

class DLog {
    vector<DLogTarget*> targets;
public:
    void addTarget(DLogTarget* p) { targets.push_back(p); }

    void write(Level level, const string& message) {
        for (int i = 0; i < targets.size(); ++i) {
                targets[i]->write(level, message);
        }
    }
};
```

Chapter 14. Test Double – Spy

목적 – 목격한 일을 기록해두었다가, 나중에 테스트가 확인할 수 있도록 한다.

```
class SpyTarget : public DLogTarget {
        vector<string> log;
        string concat(Level level, const string& message) {
                static const char* level_string[] = {
                        "INFO", "WRAN", "CRITICAL"
                };

                return string(level_string[level]) + " : " + message;
        }

public:
        bool received(Level level, const string& message) {
                return find(log.begin(), log.end(),
                        concat(level, message)) != log.end();
        }
        void write(Level level, const string& message) {
                log.push_back(concat(level, message));
        }
};
```

Chapter 14. Test Double – Spy

목적 – 목격한 일을 기록해두었다가, 나중에 테스트가 확인할 수 있도록 한다.

```
class DLogTest : public ::testing::Test {};

TEST_F(DLogTest, WritesEachMessageToAllTargets) {
        DLog log;
        SpyTarget spy1, spy2;

        log.addTarget(&spy1);
        log.addTarget(&spy2);

        Level level = INFO;
        string message = "test_message";

        log.write(level, message);

        EXPECT_TRUE(spy1.received(level, message));
        EXPECT_TRUE(spy2.received(level, message));
}
```

Chapter 15. Test Double - Mock Object

메소드를 호출하였을 때 발생하는 부작용을 관찰할 수 없어 테스트 안된 요구사항이 있다.

```
struct IMP3 {
        virtual void play() = 0;

        virtual ~IMP3() {}
};

class iPod : public IMP3 {
public:
        void play() {
                cout << "play mp3 with iPod" << endl;
        }
};


class Person {
public:
        void playMusic(IMP3* mp3) {
                mp3->play();
        }
};
```

Chapter 15. Test Double - Mock Object

목적 - 행위 기반 검증

```
// Person의 playMusic()을 호출하면 iPod의 play가 제대로 호출되는지
// 여부를 검증하고 싶다.
TEST_F(PersonTest, playMusic) {


}
```

1. 상태 검증(state verification) == 상태 기반 테스트(state base testing)
  : SUT를 실행 후, 상태를 보고 이를 기대값과 비교한다.
2. 동작 검증(behavior verification) == 행위 기반 테스트(behavior base testing)
  : SUT의 간접 출력을 갈무리 안 후, 이를 기대 동작과 비교한다.
  => 리턴값이 없거나, 리턴값을 확인하는 것만으로는 예상대로 동작했음을 보증하기 어려울 때 사용한다.

일반적으로 동작 검증을 위해 직접 목을 만들어 사용하는 것이 아니라, Mock Framework 를 이용한다.

Chapter 15. Test Double – Mock Object

목적 – 행위 기반 검증

```
#include <gmock/gmock.h>

class Mock : public iPod
{
public:
        MOCK_METHOD0(play, void());
};

class PersonTest : public ::testing::Test {};
TEST_F(PersonTest, playMusic) {
        Person person;
        Mock mock;

        EXPECT_CALL(mock, play());

        person.playMusic(&mock);
}
```

Test Double 목적

1. 테스트 시간 단축
2. 난수나 현재 시간등의 비 결정적 요소를 제어
3. 특수한 상황 시뮬레이션
4. 숨겨진 정보를 확인
5. 테스트 코드 격리
6. 행위 기반 검증

Test Double 종류

1. Stub: 원하는 결과 제어
2. Fake: 진짜 객체가 사용하기 어렵거나, 속도 개선 용도
3. Spy: 접근 불가능한 내부 정보 확인
4. Mock: 기대한 상호 작용이 정말로 일어나는지 확인하는 행위 기반 테스트에 사용.

Chapter 15. Google Mock 기능

1. Mock 객체를 만드는 방법

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

        virtual void stop() = 0;
        virtual void say(const string& msg) = 0;

        virtual void attack(Unit* target) = 0;
        virtual void move(int x, int y) = 0;

        virtual int getX() const = 0;
        virtual int getY() const = 0;
};
```

Chapter 15. Google Mock 기능

MOCK_METHOD[파라미터개수](함수명, 함수타입);
MOCK_CONST_METHOD[파라미터개수](함수명, 함수타입);

```
class MockUnit : public Unit {
public:
        MOCK_METHOD0(stop, void());
        MOCK_METHOD1(say, void(const string&));

        MOCK_METHOD1(attack, void(Unit*));
        MOCK_METHOD2(move, void(int, int));

        MOCK_CONST_METHOD0(getX, int());
        MOCK_CONST_METHOD0(getY, int());
};
```

Chapter 15. Google Mock 기능

gmock-1.7.0/scripts/generator/gmock_gen.py

```
class MockUnit : public Unit {
public:
        MOCK_METHOD0(stop, void());
        MOCK_METHOD1(say, void(const string&));

        MOCK_METHOD1(attack, void(Unit*));
        MOCK_METHOD2(move, void(int, int));

        MOCK_CONST_METHOD0(getX, int());
        MOCK_CONST_METHOD0(getY, int());
};
```

Chapter 15. Google Mock 기능

2. 검증 작업은 목 객체가 파괴되는 시점에 일어난다.

```
TEST_F(MockUnitTest, Sample2) {
        // 1. mock 생성
        MockUnit mock;

        // 2. 기대 행위를 명시한다.
        EXPECT_CALL(mock, stop());

        // 3. mock이 적용된 코드를 수행한다.
        foo(&mock);
}
```

```
// 안 좋은 방법
TEST_F(MockUnitTest, Sample1) {
        MockUnit* mock = new MockUnit;

        EXPECT_CALL(*mock, stop());

        foo(mock);

        delete mock;
}
```

Chapter 15. Google Mock 기능

3. 횟수 검증

```
void goo(Unit* p) {
        p->stop();
        p->stop();
}

using ::testing::AnyNumber;
TEST_F(MockUnitTest, Sample3) {
        MockUnit mock;

        EXPECT_CALL(mock, stop()).Times(2);
        EXPECT_CALL(mock, stop()).Times(AnyNumber());

        goo(&mock);
}
```

Chapter 15. Google Mock 기능

4. 인자 검증

```
void hoo(Unit* p) {
        // p->move(10, 20);
        // p->move(20, 20);

        for (int i = 0; i < 10; ++i) {
                p->move(10 * i, 20);
        }
}

using ::testing::_;
TEST_F(MockUnitTest, Sample4) {
        MockUnit mock;

        EXPECT_CALL(mock, move(_, 20)).Times(AnyNumber());

        hoo(&mock);
}
```

Chapter 15. Google Mock 기능

5. 순서 검증

```
void koo(Unit* p) {
        p->say("hello");
        // p->move(10, 20);
        p->attack(0);
        p->move(10, 20);
}

using ::testing::InSequence;
TEST_F(MockUnitTest, Sample5) {
        InSequence s;
        MockUnit mock;

        EXPECT_CALL(mock, say("hello"));
        EXPECT_CALL(mock, move(10, 20));
        EXPECT_CALL(mock, attack(0));

        koo(&mock);
}
```

Chapter 15. Google Mock 기능

6. 결과 제어 – ON_CALL

```
class Time
{
public:
        MOCK_METHOD0(getTimeString, string());
};

using ::testing::Return;
TEST(TimeTest, Sample)
{
        Time mock;
        ON_CALL(mock, getTimeString()).WillByDefault(Return("00:00"));

        cout << mock.getTimeString() << endl;
}
```

Chapter 16. Google Test Listener

Google Test 결과를 원하는 형식으로 출력 가능하다.

```
class MyPrinter : public ::testing::EmptyTestEventListener {
        virtual void OnTestStart(const ::testing::TestInfo& info) {
                cout << "*****************************************" << endl;
                cout << "name: " << info.name() << endl;
                cout << "test_case_name: " << info.test_case_name() << endl;
        }

        virtual void OnTestPartResult(const ::testing::TestPartResult& result) {
                bool fail = result.failed();
                cout << (fail ? "실패" : "성공") << endl;
        }

        virtual void OnTestEnd(const ::testing::TestInfo& info) {
                cout << "*****************************************" << endl;
        }
};
```

Chapter 16. Google Test Listener

Google Test 결과를 원하는 형식으로 출력 가능하다.

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

	::testing::TestEventListeners& listeners =
		::testing::UnitTest::GetInstance()->listeners();

	delete listeners.Release(listeners.default_result_printer());
	listeners.Append(new MyPrinter);

	return RUN_ALL_TESTS();
}

TEST(SampleTest, foo) { SUCCEED(); }
TEST(SampleTest, goo) { FAIL(); }
TEST(SampleTest, hoo) { FAIL(); }
```

Chapter 17. Google Test Runner Options

xml 출력 형식에 추가적인 정보 기록 방법
프로그램.exe --gtest_output=xml

```
TEST(SampleTest, foo)
{
        // xml에 추가적인 정보를 기록하고 싶다.
        RecordProperty("cpu", "1.2");
        RecordProperty("memory", "20");
}

TEST(SampleTest, goo) {}
TEST(SampleTest, hoo) {}

TEST(MyTest, foo) {}
TEST(MyTest, goo) {}
TEST(MyTest, hoo) {}
```

Chapter 17. Google Test Runner Options

1. 필터링 옵션
   프로그램.exe --gtest_filter=*.foo
                --gtest_filter=*.*foo*
                --gtest_filter=SampleTest.*-*.foo
2. 반복 테스트
   프로그램.exe --gtest_repeat=N --gtest_break_on_failure --gtest_shuffle

테스트 가능 설계

- 테스트 가능 설계는 테스트 코드에서 클래스를 생성하고, 구현 일부를 대체하고 다른 시나리오를 시뮬레이션하고, 원하는 실행 경로를 선택하는 등의 작업을 쉽게 해줄 수 있도록 해준다.
- 제품 코드는 단위 테스트를 쉽고 빠르게 작성할 수 있도록 설계되어야 한다.

The Art of Unit Testing With Examples in.NET

- 테스트 용이성(testability) – 소프트웨어를 얼마나 쉽게 테스트 할 수 있는가
- 테스트 용이성이 떨어질 수록 단위 테스트를 작성하는 프로그래머의 부담이 커진다.
- 기존 코드를 리팩터링해서 테스트 하기 쉽게 고치는 것보다, 아예 처음부터 테스트하기 쉬운 코드를 작성하는 것이 항상 더 쉽다.

테스트 가능 설계 – SOLID

- 로버트 마틴(Robert C. Martin) 이 고안한 객체 지향 설계 다섯 원칙

- 단일 책임의 원칙(SRP, Single Responsibility Principle)

✓ '클래스는 작고 한가지 역활에 충실해야 한다.'

✓ 단일 책임 원칙을 지키며 작성한 코드는 이해하기 쉽고 원하는 부분을 빠르게 찾을 수 있다. 자연스럽게 테스트 하기 쉽다.

✓ 개별 메소드가 간결해지므로, 테스트 복잡도도 같이 낮아진다.

테스트 가능 설계 - SOLID

- 개방 폐쇄 원칙(OCP, Open-Closed Principle)

✓ 클래스는 확장에는 열려 있고, 수정에는 닫혀 있어야 한다.

✓ 자신의 역활 일부를 다른 객체에 위임하도록 만들어진 클래스라면, 테스트에서도 그 기능을 테스트 더블로 교체하여 원하는 시나리오를 시뮬레이션 할 수 있다.

- 리스코프 치환 원칙(LSP, Liskov Substitution Principle)

✓ 상위 클래스는 하위 클래스로 대체 될 수 있어야 한다. (다형성)

✓ '자식 클래스의 공통된 특징은 부모로부터 와야 한다.'

✓ '계약 테스트' 를 가능하게 하므로 테스트 용이성이 높아진다.

✓ 계약 테스트 : 인터페이스에 정의된 기능을 제공하겠다는 계약을 그 구현체가 제대로 지키는지 검증하는 것. (인터페이스 동작을 확인하는 테스트 스위트 하나로 그 인터페이스를 구현한 모든 클래스 모두를 검증 가능하다)

테스트 가능 설계 - SOLID

- 인터페이스 분리 원칙(ISP, Interface Segregations Principle)

✓ 하나의 범용 인터페이스보다는 쓰임새별로 최적화된 인터페이스가 낫다.

✓ 인터페이스는 작고 한가지 목적을 충실하도록 만들어야 한다.

✓ 인터페이스가 작으면 테스트 더블을 구축하는 비용이 작아진다.

- 의존 관계 역전 원칙(DIP, Dependency Inversion Principle)

✓ 코드는 구현체가 아닌 추상 개념에 의존해야 한다.

✓ 클래스는 협력 객체를 직접 생성하지 말고, 인터페이스로 건네 받아야 한다.

✓ 협력 객체를 외부에서 넘겨줄 수 있기 때문에, 테스트 더블을 쉽게 주입 가능하다.

✓ 제품 코드가 사용하는 방식 그대로 테스트할 수 있다.

테스트 가능 설계를 위한 지침

- 복잡한 private 메소드를 지양해야 한다.
  - private 메소드의 용도를 public 메소드의 가독성을 높이기 위한 모듈화의 역활로 제한해야 한다.

- 정적 멤버 함수를 피하라.
  - 정적 멤버 함수는 스텁을 통해 교체 불가능하다.
  - 협력 객체로서 사용될 객체라면 테스트 용이성을 위해 정적 메소드를 지양해라.

- new 는 신중하게 사용하라.
  - 객체를 생성하는 것은 정확한 구현이 그것이라고 못 박는 행위이다.
  - 구체 클래스에 의존하는 형태로 코드를 작성한다는 것은 테스트 더블로 대체할 가능성이 없는 객체에 대해서만 직접 생성해야 한다.

테스트 가능 설계를 위한 지침

- 싱글톤을 피하라
  - 싱글톤 : 클래스의 인스턴스가 단 하나만 만들어진다는 것을 보장하고, 어디에서나 접근할 수 있도록 하는 패턴
  - 싱글톤을 사용하는 코드는 검사하려 할 때마다 상당히 거치적 거린다.

- 상속보다는 컴포지션을 사용하라.
  - 상속의 용도는 다형성이지 재사용이 아니다.
  - 상속을 사용하게 되면 상위 클래스에 영구히 묶여 버린다.
  - 컴포지션은 상위 클래스에 종속적이지 않다.
  - 실행 중에도 교체 가능하다.

테스트 가능 설계를 위한 지침

- 종속성을 줄여라.
  - 클래스가 다수의 다른 클래스에 종속되어 특정 상태로 셋팅되어 있다면, 테스트도 복잡해진다.
  - 코드에서 객체 생성을 담당하는 메소드와 애플리케이션 로직 수행을 처리하는 메소드를 분리하는 것이다.

- 생성자는 간단하게 만들어라.
  - 모든 테스트 케이스는 다음의 과정을 따른다.
    - 테스트 하려는 클래스를 생성한다.
    - 생성한 클래스를 특정한 상태가 되도록 설정한다.
    - 클래스의 최종 상태를 확인한다.

- 생성자에서 작업을 수행하게 되면, 첫 번째와 두 번째 작업이 섞여 버린다.
  - 코드의 유지보수성을 떨어뜨리는 안좋은 코드이다.

테스트 가능 설계를 위한 지침

- 최소 지식의 원칙을 따르라
- 디미터의 법칙(The Law of Demeter, 최소 지식의 원칙)
  - 클래스는 반드시 자신에게 꼭 필요한 만큼만 알아야 한다.

```
class Car {
private:
    Driver* driver;
public:
    Car(Context* context) {
        this->driver = context->getDriver();
    }
}
```

```
class Car {
private:
    Driver* driver;
public:
    Car(Driver* driver) {
        this->driver = driver;
    }
}
```

- Car 클래스는 Context 가 getDriver() 라는 메소드가 있어야 한다는 사실을 알아야 한다.
- 이 생성자를 테스트 하기 위해서는, 생성자 호출에 앞서 Context 가 유효한지 확인해야 한다.

테스트 가능 설계를 위한 지침

- 숨겨진 종속성과 전역 상태를 피하라.

```
void reserve() {
    DBManager* manager = new DBManager();
    manager->initDatabase();

    Reservation* r = new Reservation();
    r->reserve();
}
```

```
public void reserve() {
    DBManager* manager = new DBManager();
    manager->initDatabase();

    Reservation* r = new Reservation(manager);
    r->reserve();
}
```

Notes