

테스트 주도 개발 (Test-Driven Development)



소속 : NewHeart
이름 : 박상준
작성일 : 2014.09.30
E-mail : enif57@gmail.com

목차

1. 들어가기에 앞서.....	5
2. 테스트 주도 개발.....	5
2.1. 테스트 주도 개발이란?.....	5
2.2. 테스트 주도 개발의 과정.....	6
2.2.1. 실패하는 테스트 코드를 작성하라.....	6
2.2.2. 테스트를 돌려서 실패하는 것을 확인하라.....	7
2.2.3. 테스트를 통과하는 최소한의 구현 코드를 작성하라.....	7
2.2.4. 테스트를 통과시켜라.....	7
2.2.5. 리팩토링을 하라.....	7
2.3. 테스트 주도 개발의 장점.....	7
2.3.1. 개발의 방향을 잃지 않게 해준다.....	7
2.3.2. 좀 더 정확한 테스트 코드를 작성하게 해준다.....	7
2.3.3. 개발 능력을 높여준다.....	8
2.3.4. 설계를 개선할 수 있다.....	8
2.3.5. 모듈화 된 코드를 작성하게 된다.....	8
2.3.6. 회귀 에러(Regression Error)을 쉽게 테스트할 수 있다.....	8
2.3.7. 문서를 대체할 수 있다.....	8
2.4. 테스트 주도 개발에서 사용하는 개념들.....	8
2.4.1. 테스트 픽스처.....	8
2.4.2. 목 개체(Mock Object, or Mock).....	9
2.4.2.1. 목 개체란?.....	9
2.4.2.2. 목 개체는 언제 사용하는가?.....	9
2.4.2.3. 목 개체의 기본적인 분류.....	9
2.4.2.3.1. 더미 객체(Dummy Object).....	9
2.4.2.3.2. 테스트 스텝(Test Stub).....	10
2.4.2.3.3. 페이크 객체(Fake Object).....	11
2.4.2.3.4. 테스트 스파이(Test Spy).....	11
2.4.2.3.5. 모의 객체(목 객체, Mock Object).....	12
2.4.2.3.6. 더미 객체, 테스트 스텝, 페이크 객체, 테스트 스파이의 차이점 정리.....	12
2.4.2.4. 목 객체 사용시의 유의사항.....	13
2.4.3. xUnit.....	13
2.4.3.1. 테스트 러너(Test Runner).....	13
2.4.3.2. 테스트 케이스(Test Case).....	13
2.4.3.3. 테스트 픽스처(Test Fixture).....	13
2.4.3.4. 테스트 스위트(Test Suite).....	14
2.4.3.5. 단언문(Assertions).....	14
2.5. 테스트 주도 개발 예제.....	14
3. 구글 C++ 테스트 프레임워크.....	22
3.1. 구글 C++ 테스트 프레임워크란 무엇인가?.....	22
3.2. Google Test 설치 방법.....	23
3.3. 기본 개념.....	24
3.4. 간단한 예제.....	24
3.5. 구글 테스트에서 사용하는 단언문들(Assertions).....	25
3.5.1. 참/거짓 단언문.....	25
3.5.2. 비교 단언문.....	25
3.5.3. 문자열 관련 단언문.....	25

3.5.4. 명시적인 성공과 실패.....	26
3.5.5. Exception 단언문.....	26
3.5.6. 더 나은 에러 메시지를 위한 단정문.....	27
3.5.6.1. Boolean 함수 사용하기.....	27
3.5.6.2. AssertionResult 를 반환하는 함수 사용하기.....	28
3.5.6.3. 서술 형식자(Predicate-Formatter) 사용하기.....	29
3.5.7. 부동소수점 비교 단정문.....	30
3.5.8. 타입 단정문.....	30
3.6. 단정문의 위치.....	31
3.7. 서브루틴에서 단정문 사용하기.....	32
3.8. SCOPED_TRACE() 사용 팁.....	33
3.9. 실패 전파하기.....	33
3.9.1. 서브루틴에서의 단정 확인.....	34
3.9.2. 현재 테스트에서의 실패 확인하기.....	34
3.10. 테스트 픽처를 이용한 테스트.....	34
3.10.1. 테스트 픽처 생성 방법.....	35
3.10.2. 테스트 픽처 사용법.....	35
3.10.3. 테스트 픽처 사용 예제.....	35
3.10.4. 테스트 픽처를 이용한 테스트 실행 과정.....	36
3.11. 같은 테스트 케이스 안의 테스트들끼리의 자원 공유하기.....	37
3.12. 전역 Set-Up 과 Tear-Down.....	38
3.13. 값을 파라미터로 하는 테스트(Value Parameterized Test).....	39
3.14. 타입 테스트 하기(Typed Test).....	42
3.15. Private Code 테스트하기(Testing Private Code).....	43
3.15.1. Static Function.....	43
3.15.2. Private Class Member.....	44
3.16. 현재 테스트 이름 얻기.....	45
3.17. 고급 옵션으로 테스트 실행하기(Running Test Program : Advanced Option).....	46
3.17.1. 테스트의 이름들 나열하기.....	48
3.17.2. 테스트의 일부만 실행하기.....	48
3.17.3. 임시로 테스트 사용하지 않기(Temporarily Disabling Tests).....	49
3.17.4. 임시로 불활성화된 테스트 실행하기(Temporarily Enabling Disabled Tests).....	49
3.17.5. 테스트 반복하기(Repeating the Tests).....	49
3.17.6. 테스트 순서 섞기(Shuffling the Tests).....	50
4. 소프트웨어 테스트.....	50
4.1. 소프트웨어 테스트의 원칙.....	51
4.2. 들어가기 전 실력 확인 문제.....	52
4.3. 테스트 케이스 설계.....	52
4.3.1. 화이트 박스 테스트(White-Box Testing).....	53
4.3.1.1. 문장 커버리지(Statement Coverage).....	53
4.3.1.2. 결정 커버리지(Decision Coverage).....	53
4.3.1.3. 조건 커버리지(Condition Coverage).....	53
4.3.1.4. 결정/조건 커버리지(Decision/Condition Coverage).....	54
4.3.1.5. 다중 조건 커버리지(Multiple-condition Coverage).....	54
4.3.1.6. 각각의 커버리지 간의 포함관계.....	54
4.3.2. 블랙박스 테스트(Black-Box Testing).....	54
4.3.2.1. 동등 분할(Equivalence Partitioning).....	55
4.3.2.2. 경계값 분석(Boundary Value Analysis).....	55

5.결론.....	56
6.참고 문헌.....	56

1. 들어가기에 앞서

소프트웨어 개발 방법론은 여러가지가 있으며, 개인/팀마다 다를 수 있고 다른 사람은 추천하는 개발 방법이 자신에게 맞지 않을 수도 있다. 하지만 특정 방법론을 꼭 따르지 않더라도 이러한 개발 방법에 대해 알아보는 것은 자신의 개발 습관에 큰 영향을 미칠 수 있으며, 장점을 취해 더 좋은 개발 방식을 몸에 익힐 기회가 될 수 있을 것이라 생각한다.

테스트 주도 개발은 '구현 코드 작성 후 테스트 코드를 통한 테스트'라는 방식에서 '테스트 코드 작성 후 테스트 코드를 통과하기 위한 구현 코드 작성'이라는 거꾸로 된 방식을 택함으로써 개발자의 사고 습관을 바꾸고 좀 더 깔끔한 코드 작성을 목적으로 하고 있다.

이 문서는 크게 3 부분으로 나누어져 있다. 첫 번째 부분은 테스트 주도 개발에 대한 내용을 다루는 부분으로, 테스트 주도 개발의 개념과 방법, 장점과 테스트 주도 개발에서 사용하는 개념들에 대한 설명, 테스트 주도 개발 예제 등으로 이루어져 있다.

두 번째 부분은 테스트 주도 개발에서 중요한 유닛 테스트 툴(Unit Test Tool, 단위 테스트 툴) 중 C++에서 사용 가능한 Google C++ 테스팅 프레임워크의 사용법에 대해 이야기하고 있다. 테스트 주도 개발의 특성상 테스트를 자주 하게 되는데, 이러한 테스트를 하는 함수를 직접 만들어서 사용해도 되지만 그런 방식 보다는 테스팅 프레임워크를 사용하는 것이 더 편하다. 수많은 테스팅 프레임워크 중 구글 C++ 테스팅 프레임워크를 사용한 이유는 크게 세 가지이다.

- 1) 언어가 C++이다.
- 2) 사용 방법과 준비 과정이 간단하다.
- 3) Google에서는 Google Mock이라는 Mock 개체도 제공하고 있다.

이러한 이유로 현재 나 자신이 사용하기 편하고, 이후 Mock 개체에 대한 부분으로의 확장도 편할 것으로 판단되어 공부할 테스팅 프레임워크로 구글 C++ 테스팅 프레임워크를 선택하였다.

마지막 부분은 소프트웨어 테스팅 공학에 대한 부분이다. 테스트 주도 개발에서의 테스트는 '어떻게 구현을 할 것인가' 혹은 '어떤 입력 혹은 출력을 허용하지 않을 것인가'와 같은 가이드를 잡아주는 역할이 크다. 그에 반해 소프트웨어 테스팅 그 자체는 프로그램의 명세에서부터 설계, 구현에 이르기까지 '문제 찾아내기' 그 자체에 집중하고 있다. 약간 방향성은 다르지만 앞으로 개발에서 빼놓을 수 없는 테스팅에 대한 공부도 되고, 그 이유가 아니더라도 '소프트웨어 테스트'라는 같은 개념을 테스트 주도 개발과 소프트웨어 테스팅 공학에서 다루고 있으므로 조금이지만 같이 공부를 해 보았다.

2. 테스트 주도 개발

2.1. 테스트 주도 개발이란?

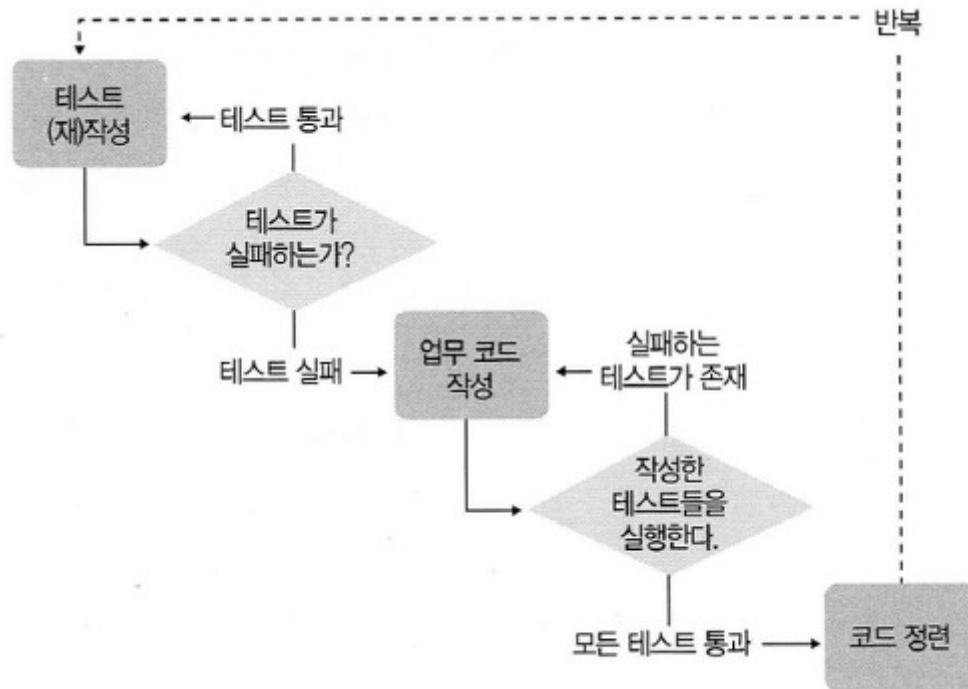
테스트 주도 개발(Test-Driven Development, TDD)은 켄트 벡이 주장한 소프트웨어 개발 방법론의 하나로 익스트림 프로그래밍(eXtreme Programming, XP) 방법론의 일부이다.

소프트웨어 개발에 관한 여러 가지 방법론이 있지만 모든 방법론들의 목적은 단 하나일 것이다. 좀 더 편하고 완벽하게 개발하는 것이다. 개발을 어렵게 하는 것에는 크게 두 가지를 들 수 있다. 버그와 요구사항의 추가/변경이다. 테스트 주도 개발은 이러한 어려움을 해결하기 위한 방법으로 테스트 코드를 먼저 작성하는 방법을 주장하고 있다.

테스트 주도 개발의 목표를 한마디로 말하자면 '잘 동작하는 깔끔한 코드(Clean code that works)'이다. 당연한 소리라고도 생각할 수도 있겠지만, 잘 동작하는 것뿐만 아니라 깔끔함도 동작성과 함께 요구한다는 것이 포인트라고 할 수 있다. 깔끔하다는 것은 유지보수의 편의성 및 가독성, 그에 따른 관리비용의 감소와 안전성의 증가 등을 의미한다.

테스트 주도 개발은 **테스트 코드 작성 → 테스트 실패 확인 → 구현 코드 작성 → 테스트 통과 확인 → 리팩토링**의 5 단계를 짧은 주기로 반복하면서 프로그램을 점차 완성시켜나가는 방식이며, 특징은 테스트 코드를

먼저 작성한다는 점과 짧은 주기로 반복한다는 것, 그리고 리팩토링이 개발 단계에 포함되어 있다는 점이다.



TDD 순서도.

사진 1: 테스트 주도 개발 순서도

테스트를 중심으로 한다는 점에서 테스트 주도 개발과 유닛 테스트를 혼동하는 경우가 있다. 둘 모두 테스트를 중심으로 생각한다는 점에서 공통점이 있지만 근본적인 차이점이 있다. 테스트 주도 개발은 테스트를 먼저 생각하는 ‘개발 방식’이고, 유닛 테스트는 구현한 기능을 테스트하는 자동화된 ‘테스트 케이스’ 혹은 이 테스트 케이스를 이용한 ‘테스트’를 의미한다. 따라서 보통 유닛 테스트는 테스트 주도 개발로 작업한 결과의 일부분에 속하게 된다. 당연한 말이지만 유닛 테스트를 위한 도구들은 테스트 주도 개발에서도 사용하게 된다.

2.2. 테스트 주도 개발의 과정

테스트 주도 개발은 다음의 5 단계를 하나의 사이클로 한다.

1. 실패하는 테스트 코드를 작성하라.
2. 테스트를 돌려서 실패하는 것을 확인하라.
3. 테스트를 통과하는 최소한의 구현 코드를 작성하라.
4. 테스트를 통과시켜라.
5. 리팩토링을 하라.

2.2.1. 실패하는 테스트 코드를 작성하라.

: 테스트 코드를 먼저 작성하면 구현 코드의 목적을 명확하게 인식한다는 장점이 있다. 테스트 코드를 만들기 위해서는 어떤 함수의 입력과 출력이 어떻게 될 지를 정확히 이해해야 하며, 어떤 경우에 예외상황이 발생하는지 혹은 어떤 상황을 문제 상황으로 여길지도 제대로 파악해야 한다. 이러한 고민을 구현 코드 작성 전에 함으로써 좀 더 객관적으로 문제를 바라보게 만든다. 이러한 테스트 코드를 작성함으로써 구현 코드는 테스트 코드를 통과하기만 하면 되는 쉬운 목표가 된다.

2.2.2. 테스트를 돌려서 실패하는 것을 확인하라.

: 구현 코드가 없이 테스트 코드를 먼저 만든 이상 테스트 코드를 실행시키면 당연히 실패한다. 만약 실패하지 않는다면

- 1) 테스트 코드를 잘못 작성했다.
- 2) 만들려는 코드와 같은 기능을 하는 코드가 이미 존재한다.

의 두가지 상황 중 하나일 것이다. 이 단계는 이러한 상황을 피하게 만든다.

2.2.3. 테스트를 통과하는 최소한의 구현 코드를 작성하라.

: 구현 코드를 작성하는 단계이다. 여기서의 포인트는 '최소한의' 구현 코드라는 점이다. 모든 기능과 모든 알고리즘을 도입하여 완벽한 코드를 만드는 것이 아니라 테스트 코드만 통과하는 코드를 만들면 된다. 복잡한 구현 코드를 개발하는 것은 테스트 주도 개발의 컨셉인 짧은 개발 사이클을 무시하는 것이기도 하며 그 목적 중 하나인 빠른 디버깅을 방해한다.

2.2.4. 테스트를 통과시켜라.

: 구현한 코드가 테스트를 통과하는지 확인하고, 실패한다면 이유를 찾아서 수정한다.

2.2.5. 리팩토링을 하라.

: 위의 4 단계와는 이질적인 부분이라고 생각한다. 3 단계에서 의아함을 느꼈겠지만, 구현 코드를 작성하는 단계에서 완벽하고 깔끔한 코드를 작성하진 않는다. 이 단계에 와서야 지저분하고 중복된 코드를 수정하며 이를 통해서 테스트 주도 개발이 완성된다.

이 5 단계로 이루어진 하나의 사이클을 최대한 짧게 이루어져야 한다. 즉 테스트 코드도 짧게 시작을 해야 하고, 이를 통과하기 위한 구현 코드도 짧게 짧게 구현 되어야 한다. 이러한 방식은 심플한 코드를 갖게 해주고 목적에 집중하게 해주며 빠르게 에러를 찾게 해준다. 만약 사이클이 길게 된다면 작성하고 있는 코드를 줄이고 단계를 늘리거나, 구현 기능 자체를 분할하는 방식을 생각해 볼 필요가 있다.

2.3. 테스트 주도 개발의 장점

2.3.1. 개발의 방향을 잃지 않게 해준다.

: 테스트 코드를 먼저 작성하기 위해서는 그 구현 방식이나 목표를 정확히 알아야 한다. 이러한 면은 장기적인 목표를 나타낸다. 또한 실패하는 테스트 케이스들은 극복해나가야 할 단기적인 목표가 된다. 혼자서 한 번에 개발을 마치는 것이 아니라, 여럿이서 혹은 잠깐잠깐 개발을 하는 경우라면 테스트 주도 개발은 개발의 목표를 유지하는데 도움이 될 것이다. 예를 들면 잠시 쉬었다가 개발을 다시 시작하는 경우, 실패한 테스트 케이스부터 다시 개발을 시작하는 것도 한 방법이 될 수 있다.

2.3.2. 좀 더 정확한 테스트 코드를 작성하게 해준다.

: 정확한 테스트를 위해선 수많은 요구사항과 예외 상황들을 고려해야 한다. 물론 구현 코드를 작성할 때에도 그런 생각을 해야 하지만, 아무래도 구현 방식에 대한 생각을 함께 하다보면 예외 상황에 대한 고려가 부족해 질 수도 있다. 또한, 구현 코드를 먼저 작성하고 테스트 코드를 작성하면 실패하는 코드보다는 구현 코드가 제대로 동작하는지를 확인하기 위한 코드를 작성하게 되기 쉽다는 문제도 있다.

테스트 코드를 먼저 작성하는 방식은 구현 방식에 대한 고려를 나중에 하게 함으로써 요구 사항과 예외 상황에 대한 생각을 더 깊게 할 수 있도록 도와주고, 빼먹는 예외 상황이 없도록 도와준다.

2.3.3. 개발 능력을 높여준다.

: 짧은 사이클 안에서 테스트 코드를 통과하는 구현 코드를 완성함으로써 개발자는 성취감을 느낄 수 있다. 정확한 구현 방법이나 내부 설계가 막막하더라도 테스트 주도 개발 방식을 통해 조금씩 구현을 해 나가다 보면 Divide & Conquer 방식을 자연스럽게 적용하게 된다.

2.3.4. 설계를 개선할 수 있다.

: 테스트 코드를 작성하기 위해선 인터페이스, 이름, 인자 등에 대한 고민을 먼저 해야 한다. 만약 테스트하기 어려운 구조의 코드라면 설계 방식을 제대로 따르지 않는 코드일 가능성이 높으며, 테스트 코드를 작성하면서 이러한 고민을 해봄으로써 더 나은 설계 디자인을 따르게 된다.

2.3.5. 모듈화 된 코드를 작성하게 된다.

: 짧은 사이클을 반복하는 테스트 주도 개발의 특성상 코드 구현은 작은 모듈 단위로 하게 된다.¹ 이러한 코드 구현은 이후 기능 추가나 수정 때 다른 부분과 상호작용하면서 생기는 문제를 방지하며, 테스트 케이스도 포함되어 있을 뿐 아니라 테스트도 완료된 모듈이므로 재사용이 용의하다.

2.3.6. 회귀 에러(Regression Error)을 쉽게 테스트할 수 있다.

: 회귀 에러는 이전에 제대로 작동하던 프로그램에 새로운 기능을 추가했는데, 예전에 작동하던 부분에서 발생하는 버그를 말한다. 이러한 경우는 과거에 작성했던 코드를 테스트하는 코드가 있지 않으면 알기 쉽지 않다. 테스트 주도 개발에서는 이미 만들어진 테스트 코드가 있으므로 테스트를 한번 실행하는 것만으로 이러한 에러를 잡을 수 있다.

2.3.7. 문서를 대체할 수 있다.

: 유지 보수를 위해 프로그램 코드를 작성하면서 같이 그 코드에 대한 문서도 작성해야 하지만, 코드를 수정하다보면 문서 수정을 꼬박꼬박 하는 것은 힘들다. 편한 개발을 위해 문서를 작성하는 것인데, 문서 작성이 오히려 개발을 방해하는 것이다.

테스트 코드는 실제로 동작하는 경우와 동작하지 않는 경우를 실제 예를 들어 보여주고 있다. 즉, 어떠한 목적과 기능이 있으며 어떤 방식으로 사용하는지에 대한 설명서가 될 수 있다. 원칙상 테스트 코드를 추가/수정하고 코드를 수정하므로 문서처럼 코드와 문서의 버전이 맞지 않는 경우도 방지하며 모든 코드가 테스트 됐다는 것을 보장할 수 있다.. 예를 들어 테스트 주도 개발을 엄밀히 적용하면 구현 코드에 else 문을 하나 넣더라도, 테스트 코드를 먼저 작성해야 한다. 따라서, 본 코드를 작성하고 테스트를 작성할 때 발생할 수 있는, “미처 테스트해보지 못한, 그래서 발생하는 에러”는 생기지 않는다.

2.4. 테스트 주도 개발에서 사용하는 개념들

2.4.1. 테스트 픽스처

테스트 픽스처는 여러 번 테스트를 실행해도 항상 같은 결과를 얻을 수 있도록 도와주는 **기반이 되는 상태나 환경**을 의미한다. 예를 들어, 어떤 변수의 값을 증가시키는 함수를 테스트 한다고 해보자. 처음 실행할 때에는 단언문(Assertion)에 입력한 예상값과 함수의 실행결과가 일치하겠지만, 두 번째 실행 때에는 일치하지 않을 수도 있다. 이미 첫 번째 테스트 때 변수의 값이 변했기 때문이다. 이러한 경우는 같은 변수를 같이 사용하는 여러 함수들을 테스트 할 때에도 발생할 수 있다. 이러한 일을 막기 위해 모든 테스트 프레임워크

1. 모듈화는 객체 지향 프로그래밍의 특징 중 하나이지만 실제로는 모듈화를 잘 안하게 된다

들은 테스트 픽스처를 제공하며 이러한 테스트 픽스처들은 테스트 전에 매번 새로 만들어지고, 테스트가 끝나면 정리된다. 테스트 픽스처가 있기 때문에 항상 같은 테스트 결과를 보장할 수 있다.

2.4.2. 목 개체(Mock Object, or Mock)

2.4.2.1. 목 개체란?

흔히 ‘목업(mock-up)을 만든다.’ 라는 말을 사용한다. 한번 만들어 볼 필요는 있는데 실제 제품과 같게 만들기에는 비용이 부담될 경우, 조형이 쉬운 물질(나무나 스티로폼)로 똑딱 만들어 디자인이나 기능 등을 확인하는 것이다. 이때 만든 물건을 목업이라 부른다. 목 개체는 이와 같은 역할을 한다. 겉만 대충 만들고 필요한 부분만 구현한 개체인 것이다.

2.4.2.2. 목 개체는 언제 사용하는가?

- 팀 작업을 하는데, 아직 객체가 완성이 안 된 경우
: 객체는 다른 팀원이 만들고, 나는 그 객체를 받아서 사용만 하는데, 아직 객체가 완성이 되지 않았다면 내가 만든 기능을 어떻게 테스트 할 것인가? 이럴 때 임시로 내 테스트에 필요한 역할만 하는 목 개체를 만들어 사용한다.
- 환경 구축에 시간이 오래 걸리는 경우
: 예를 들어 데이터베이스나 서버를 이용하는 경우를 들 수 있다. 데이터베이스나 서버를 구축하는데 시간이 오래 걸린다면, 데이터베이스나 서버처럼 보이는 목 개체를 만들어 테스트하는 것이 나을 수도 있다.
- 테스트가 특정한 상황에 의존적인 경우
: 예를 들어 FTP 클라이언트 프로그램을 만드는데, 네트워크 연결의 접속제한시간을 구현한다고 해보자. 접속 시도 후 5 초까지 1 초마다 접속을 재시도하고 그 이후엔 실패 메시지를 띄우는 기능을 테스트한다고 했을 때 어떻게 테스트를 할 수 있을까. 시계를 옆에 두고 네트워크 단자를 뽑았다 켜다 하면 될까? 이런 실제로 만들기 힘든 상황을 테스트해야 하는 경우 목 객체를 이용하는 것이 답일 수 있다.

2.4.2.3. 목 개체의 기본적인 분류

『xUnit Test Patterns』의 저자인 제라드 메스자로스(Gerard Meszaros)는 책에 테스트 더블(Test Double)²이라는 단어를 만들어 내면서 목 개체를 테스트 더블의 하위로 분류해놓았다. 제라드의 이러한 분류법은 일반적으로 받아들여지지만, 대부분의 사람들은 테스트 더블 대신 목(mock)이라는 단어를 사용한다. 목 개체를 이해하는 데에도 도움이 되니 테스트 더블의 분류에 대해 잠깐 알아보자.

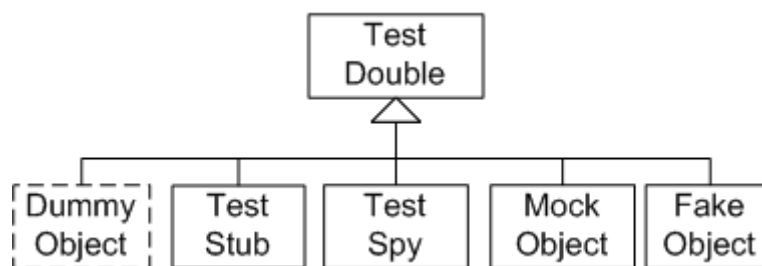


사진 2: 테스트 더블의 분류

2. 대역, 스텐드맨을 나타내는 스텐트 더블(stunt double)에서 차용해 온 단어이다.

2.4.2.3.1. 더미 객체(Dummy Object)

: 꺾테기만 구현한 객체이다. 예를 들어 온라인 마켓에서 사용 가능한 할인 쿠폰을 C++ 더미 객체로 만든다고 해보자. 그럼 다음과 같은 구현이 될 것이다.

```
class TestCoupon{
public :
    TestCoupon(){}

    int getDiscountPercentage(){
        return 0;
    }

    string getName(){
        return "";
    }

    bool isAvailable(TestItem item){
        return true;
    }

    bool isValid(){
        return false;
    }

    void doExpire(){}
};
```

이러한 객체는 딱 구현만 될 것이다. 그 외에는 어떠한 결과도 기대할 수 없다. 쿠폰이 생성만 되는 것으로 충분한 테스트³를 할 수 있다면 그때는 더미 객체로도 충분할 것이다. 하지만 만약 쿠폰을 사용하는 부분도 테스트를 해야 한다면 이것만으로는 부족하다. 멤버들이 제대로 구현되어 있지 않기 때문이다. 이 부분을 구현해 더미 객체보다 조금 더 나아간 것이 테스트 스텝이다.

2.4.2.3.2. 테스트 스텝(Test Stub)

: 테스트 스텝은 객체가 실제로 동작하는 것처럼 보이게 만든 객체이다. 다음의 코드는 위의 TestCoupo 클래스를 테스트 스텝을 만들어 본 것이다.

```
class TestCoupon{
public :
    TestCoupon(){}

    int getDiscountPercentage(){
        return 5;    // % 단위
    }

    string getName(){
        return "5% 할인쿠폰";
    }

    bool isAvailable(TestItem item){
        return true;
    }
}
```

3. 예를 들면, 만들어진 쿠폰이 사용자 계정에 잘 등록이 되는지를 테스트하는 경우

```

        bool isValid(){
            return true;
        }

        void doExpire(){}
};

```

이 객체는 5% 할인 쿠폰으로 사용할 수 있을 것이다. 이처럼 하는 행동이 딱 정해져 있지만, 객체의 특정한 예로 사용할 수 있는 정도로 구현된 테스트 더블을 테스트 스텝이라 부른다.

2.4.2.3.3. 페이크 객체(Fake Object)

: 페이크 객체는 테스트 스텝의 확장판이라 할 수 있다. 위의 테스트 스텝은 `isAvailable()` 함수가 항상 `true`를 반환하기 때문에, 모든 물품에 사용 가능한 5% 할인 쿠폰으로밖에 사용할 수 없다. 만약 어떤 물품에는 쿠폰을 적용 가능하고, 어떤 물품에는 쿠폰이 적용 불가능한지를 테스트 하려면 어떻게 해야 할까? 각각의 테스트를 할 때마다 클래스의 리턴값을 바꿔주어야 할 것이다. 이런 번거로운 일을 막기 위해 내부에 리스트를 가지고 있다가 그때그때 다른 리턴값을 리턴해줄 수 있도록 만들어진 테스트 더블이 페이크 객체이다.

```

class TestCoupon{
public :
    TestCoupon(){}
    int getDiscountPercentage(){
        return 5;        // % 단위
    }

    string getName(){
        return "5% 할인쿠폰";
    }

    bool isAvailable(TestItem item){
        if (item == "전자제품") return true;
        else if (item == "의류") return false;
        return false;
    }

    bool isValid(){
        return true;
    }
    void doExpire(){}
};

```

`isAvailable()` 함수 내부를 보면, 인자로 넘겨받는 `item`의 종류에 따라 리턴값을 바꿔주고 있다. 이런 경우 적용 가능한 경우와 불가능한 경우 모두를 테스트해 볼 수 있다. 이런 수준이 되면 실제 로직이 구현된 것처럼 보이기 때문에 페이크 객체라 부른다.

2.4.2.3.4. 테스트 스파이(Test Spy)

: 더미 객체~페이크 객체는 복잡함과 정교함의 차이가 있다면, 테스트 스파이는 앞의 것들과 기능상의 차이가 있다. 만약 객체 내부의 멤버 함수의 호출 횟수를 알고 싶다면 어떻게 해야 할까? 객체를 사용하는 쪽에서 체크를 할 수도 있지만, 객체 내부에 멤버 함수가 호출될 때마다 카운팅되는 변수를 하나 만드는 것이 더 편할 것이다. 이처럼 실제 구현될 객체에는 없는 내용이지만, 스파이처럼 객체에 관련된 정보를 수집하는 기능이 들어있는 테스트 더블을 테스트 스파이라 부른다. 테스트 스파이의 경우, 감시 기능을 제외한 부분은 더

미 객체일수도, 페이크 객체일 수도 있다.

```
class TestCoupon{
public :
    TestCoupon(){}
    int getDiscountPercentage(){
        return 5;    // % 단위
    }

    string getName(){
        countGetNameCall++;    // 함수 호출 횟수 감시
        return "5% 할인쿠폰";
    }

    bool isAvailable(TestItem item){
        countIsAvailableCall++;    // 함수 호출 횟수 감시
        if (item == "전자제품") return true;
        else if (item == "의류") return false;
        return false;
    }

    bool isValid(){
        return true;
    }
    void doExpire(){}

    int countGetNameCall;    // getName()이 호출된 횟수를 기억
    int countIsAvailableCall;    // isAvailable()이 호출된 횟수를 기억
};
```

2.4.2.3.5. 모의 객체(목 객체, Mock Object)

: 제라드의 정의에서의 모의 객체는 앞의 테스트 더블들과는 달리 상태가 아닌 행위를 기반으로한 테스트 객체를 의미했다. 하지만 현재는 더미 객체~테스트 스파이를 합친 것을 그냥 목 객체라 부르므로 우리도 그렇게 알고 쓰자.

2.4.2.3.6. 더미 객체, 테스트 스텝, 페이크 객체, 테스트 스파이의 차이점 정리

- 더미 객체 : 단지 인스턴스화 될 수 있는 정도로 구현
- 테스트 스텝 : 객체가 하나의 상태를 표현하는 정도로 구현
- 페이크 객체 : 여러 개의 객체를 표현할 수 있는 정도로 구현
- 테스트 스파이 : 객체 사용에 대한 감시를 할 수 있는 기능이 들어있는가

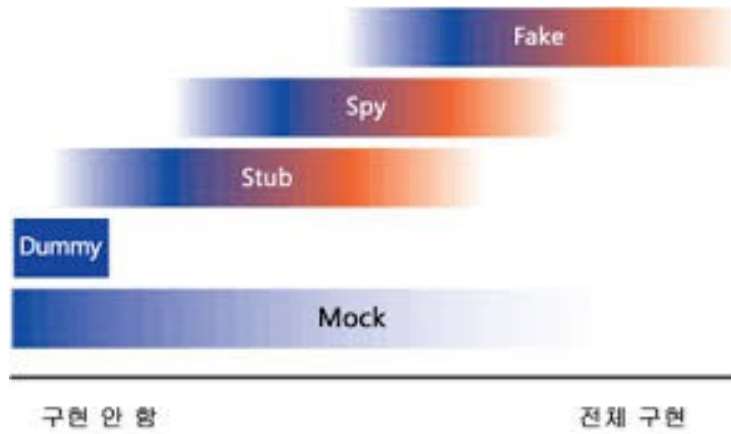


사진 3: 테스트 더블 비교

2.4.2.4. 목 객체 사용시의 유의사항

목 객체라는 것은 결국 개발자의 시간과 노력을 아끼기 위한 방법이다. 목 객체의 경우, 특정한 경우나 상황을 구현한 것이기 때문에 테스트 내용이 바뀌거나 하면 사용할 수 없게 되는 경우가 많다. 또한 최종적으로 보면 실제 객체를 구현해야 하고, 그 객체로 테스트를 해야 하는 순간이 온다. 따라서 실제 객체를 구현한 뒤 테스트를 하는 것이, 목 객체를 이용해 간단하게 테스트를 하는 것보다 장기적으로 보면 오히려 더 나을 수도 있다.

2.4.3. xUnit

xUnit 은 스톨토크⁴의 테스트 프레임워크인 SUnit 의 구조와 함수들을 따라한 유닛 테스트 프레임워크들을 총칭하는 이름이다. xUnit 이라는 이름은 SUnit 에서 따온 것으로 첫 글자는 사용하는 언어에서 따온다.

⁵xUnit 들은 보통 다음의 기본적인 구조와 요소를 공통으로 가진다.

2.4.3.1. 테스트 러너(Test Runner)

: 테스트 러너는 xUnit 프레임워크를 사용해 구현한 프로그램을 실행하고 결과를 보고하는 실행 가능한 프로그램을 말한다.

2.4.3.2. 테스트 케이스(Test Case)

: 테스트 하려는 프로그램이 목적대로 만들어 졌는지를 확인하기 위한 조건이나 변수의 집합을 의미한다. 가장 기본적인 요소이다.

2.4.3.3. 테스트 픽스처(Test Fixture)

: 테스트 픽스처(또는 테스트 컨텍스트(test context))는 테스트를 돌리기 위해 필요한 ‘전제 조건이나 상태의 모임’을 의미한다.

4. 객체지향 프로그래밍 언어. <http://en.wikipedia.org/wiki/Smalltalk>

5. Java 의 JUnit, R 의 RUnit

2.4.3.4. 테스트 스위트(Test Suite)

: 테스트 스위트는 같은 테스트 픽스처를 공유하는 테스트들의 집합이다. 테스트의 순서는 고려하지 않는다.

2.4.3.5. 단언문(Assertions)

: 단언문은 테스트 되는 코드를 확인하는 함수나 매크로이다. 단언문은 보통 현재 테스트 중인 프로그램의 결과가 참인지 거짓인지를 확인하며, 실패시 현재 테스트를 중단한다.

2.5. 테스트 주도 개발 예제⁶

목표 : 여러개의 통화단위를 지원하는 주식 총액 계산 프로그램을 만든다. 예를 들면 다음과 같은 합계를 보여주는 프로그램이다.⁷

종목	주	가격	합계
IBM	1000	25USD	25000USD
GE	400	150CHF ⁸	60000CHF
		합계	65000USD

※ 환율 : 1.5CHF = 1USD

어떤 기능들이 필요할까 생각해보았다.

- 가격과 주를 곱해서 총 합계를 구하는 기능 : $2 * 5\text{USD} = 10\text{USD}$
- USD 와 CHF 를 더해서 USD 로 나타내는 기능 : $10\text{USD} + 15\text{CHF} = 20\text{USD}$ ⁹

일단 이 두 기능만 있으면 될 것 같다. 테스트 주도 개발의 개발 원칙에 따라 우선 하나의 기능에 대해서만 테스트 코드를 먼저 작성해보았다.

```
TEST(TDDEExample, TestMultiply) {  
    Dollar five(5);  
    EXPECT_EQ(10, five.times(2));  
}
```

컴파일조차 되지 않는다. 그 이유는 (이 코드를 작성하는 순간부터 알 수 있지만) Dollar 클래스를 아직 선언하지 않았기 때문이다. 바로 문제가 있다는 것을 알 수 있지만, 테스트 주도 개발의 단계를 차근차근 밟아가기 위해 이렇게 작성하였다. 테스트 코드를 작성하고 실패하는 것까지 확인했으니 이 테스트를 통과만 할 수 있는 코드를 작성할 차례이다.

```
class Dollar{  
public:  
    Dollar(int amount){  
    }
```

6. 켄트 벡의 『Test-Driven Development : By Example』 나오는 예제

7. 객체 지향 방식으로 만들 필요도 없을 정도로 간단한 프로그램이지만, 연습을 위해 최대한 책의 방식을 따라했다.

8. 스위스의 화폐단위. 스위스 프랑

9. 굳이 큰 숫자를 대입할 필요가 없으므로 적당히 작은 숫자들로 테스트를 한다.

```
int times(int multiplier){
    return 0;
};
```

컴파일이 될 수만 있도록 Dollar 클래스를 구현하고 돌려보았다. 장렬하게 바로 Test Failed 가 뜨는 것을 볼 수 있다.

```
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from TDDEExample
[ RUN      ] TDDEExample.TestMultiply
c:\Users\bluemoon\documents\visual studio 2013\projects\tddeexample\tddeexample\tddeexamplemain.cpp(22): error: Value of: five.times(2)
Actual: 0
Expected: 10
[ FAILED   ] TDDEExample.TestMultiply (1 ms)
[-----] 1 test from TDDEExample (1 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (4 ms total)
[ PASSED   ] 0 tests.
[ FAILED   ] 1 test, listed below:
[ FAILED   ] TDDEExample.TestMultiply

1 FAILED TEST
계속하려면 아무 키나 누르십시오 . . .
```

사진 4: 테스트 주도 개발 예제 결과 - 클래스 선언만 했을 때

좀 더 구현 코드를 수정해야 한다.

```
int times(int multiplier){
    return 10;
}
```

우선 테스트를 통과할 수 있는 값인 10 을 리턴하도록 했다. 돌려본 결과 모두 성공이다.

```
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from TDDEExample
[ RUN      ] TDDEExample.TestMultiply
[ OK       ] TDDEExample.TestMultiply (0 ms)
[-----] 1 test from TDDEExample (1 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (4 ms total)
[ PASSED   ] 1 test.
계속하려면 아무 키나 누르십시오 . . .
```

사진 5: 테스트 주도 개발 예제 결과 - 리턴값을 정해줬을 때

여기까지 하면 테스트 주도 개발의 5 단계 중 테스트 코드 작성-실패, 구현코드 작성-실패의 4 단계를 거친

것이다. 마지막으로 남은 것은 리팩토링이다.

리턴값으로 사용한 10은 처음 생성시 받은 값인 5와 인자로 받은 2를 우리가 직접 곱해서 얻은 값이다. 이 부분을 프로그램에게 맡기자.

```
class Dollar{
public:
    Dollar(int amount_){
        amount = amount_;
    }

    int times(int multiplier){
        return amount * multiplier;
    }

private:
    int amount;
};
```

다음과 같이 구현 코드를 수정해서 테스트를 돌려본 결과 역시 성공했다. 이제야 비로소 테스트 주도 개발의 한 사이클이 종료되었다. 보통 이 정도로 간단한 코드는 크게 고민하지 않고도 바로 작성할 수 있겠지만, 테스트 주도 개발이 무엇인지 개념을 알기 위해 한 번에 할 수 있는 것들도 단계를 나누어 해보았다. 사실 테스트 주도 개발의 목적 중 하나가 당연하다고 생각하고 작성하는 코드에 의해 ‘나중에 한꺼번에’ 발생하는 에러들을 막기 위한 것이니 웬만하면 한 번에 하나씩만 작성하는 것이 옳은 것 같긴 하다.

이제 구현한 기능은 지우고, 두 번째 기능을 구현해보도록 하자.

```
- 가격과 주를 곱해서 총 합계를 구하는 기능 : 2 * 5USD = 10USD
- USD와 CHF를 더해서 USD로 나타내는 기능 : 10USD + 15CHF = 20USD
```

구현을 해보려 하니 CHF를 위한 클래스가 필요한 것 같다. Dollar와 같이 다음과 같은 테스트를 통과하게 하고 싶다.

```
TEST(TDDExample, TestFranc){
    Franc five(5);
    EXPECT_EQ(10, five.times(2));
}
```

Dollar의 코드를 이용해 Franc 클래스를 만들었다.

```
class Franc{
public:
    Franc(int amount_){
        amount = amount_;
    }

    int times(int multiplier){
        return amount * multiplier;
    }

private:
    int amount;
};
```


테스트를 돌려본 결과, 성공한다. 어차피 Dollar 코드를 복사-붙여넣기 한 것이니 크게 문제가 생길리도 없다. 리팩토링도 딱히 할 것은 없어 보인다.

그 다음은 USD와 CHF를 합할 수 있는 기능을 구현해야 한다. 두 객체를 합하려고 하니, 클래스가 다른 것이 문제가 된다. 어차피 큰 형태는 같으니 Money 클래스를 만들고 Dollar와 Franc 클래스가 Money 클래스를 상속받는 식으로 바꾸어야 할 것 같다. 우선 할일 리스트에 구현해야 할 기능을 추가했다.

- ~~가격과 주를 곱해서 총 합계를 구하는 기능 : $2 * 5\text{USD} = 10\text{USD}$~~
- USD와 CHF를 더해서 USD로 나타내는 기능 : $10\text{USD} + 15\text{CHF} = 20\text{USD}$
- **Dollar와 Franc 클래스의 상위 클래스인 Money 클래스**
- Dollar와 Franc 클래스의 비교 : $\text{Dollar}(5) \neq \text{Franc}(5)$

우선 Money 클래스에 대한 테스트를 작성해보자. 다음과 같은 테스트를 통과하도록 만들고 싶다.

```
TEST(TDDEExample, TestMoney){
    Money * fiveD = new Dollar(5);
    Money * fiveF = new Franc(5);
    EXPECT_EQ(10, fiveD->times(2));
    EXPECT_EQ(10, fiveF->times(2));
}
```

당연히 이 상태로 돌리면 컴파일 에러가 난다. 다음과 같이 클래스들을 수정했다.

```
class Money{
public:
    Money(){}

    Money(int amount_) : amount(amount_){
    }

    virtual int times(int multiplier){
        return amount*multiplier;
    }

    virtual void show() {
    }

protected:
    int amount;
};

class Dollar : public Money{
public:
    Dollar(int amount_) : amount(amount_) {
    }

    int times(int multiplier){
        return amount * multiplier;
    }

private:
    int amount;
};

class Franc : public Money{
```

```

public:
    Franc(int amount_) : amount(amount_){
    }

    int times(int multiplier){
        return amount * multiplier;
    }

private:
    int amount;
};

```

TestMoney 테스트 결과 아무 문제없이 성공하는 것을 확인했다.

이제 리팩토링을 할 차례이다. 지금 Money 클래스와 Money 클래스를 상속받는 Dollar, Franc 클래스 모두에 같은 역할을 하는 코드(times()와 amount 선언)가 작성되어 있다. Dollar 클래스와 Franc 클래스는 Money 클래스의 멤버 함수를 상속받아 사용만 하면 되므로, 중복되는 부분은 지우도록 하자.

```

class Money{
public:
    Money(){}

    Money(int amount_) : amount(amount_){
    }

    virtual int times(int multiplier){
        return amount*multiplier;
    }

    virtual void show() {
    }

protected:
    int amount;
};

class Dollar : public Money{
public:
    Dollar(int amount_) : Money(amount_) {
    }
};

class Franc : public Money{
public:
    Franc(int amount_) : Money(amount_){
    }
};

```

이것으로 또 하나의 TDD 사이클이 끝났다.

Dollar 와 Franc 는 같은 클래스를 상속받기만 해서 코드 상으론 차이가 나지 않지만, amount 가 같아도 그 둘은 다른 객체이다. Dollar 와 Franc 클래스를 비교하는 기능을 구현해보자.

— 가격과 주를 곱해서 총 합계를 구하는 기능 : $2 * 5\text{USD} = 10\text{USD}$

- USD와 CHF를 더해서 USD로 나타내는 기능 : $10\text{USD} + 15\text{CHF} = 20\text{USD}$
- Dollar와 Franc 클래스의 상위 클래스인 Money 클래스
- Dollar와 Franc 클래스의 비교 : $\text{Dollar}(5) \neq \text{Franc}(5)$

다음과 같은 테스트를 통과할 수 있도록 하는 것이 목적이다.

```
TEST(TDDEExample, TestMoney){
    ...
    Money * tenD = new Dollar(10);
    EXPECT_TRUE(fiveD->equal(new Dollar(5)));
    EXPECT_TRUE(fiveF->equal(new Franc(5)));
    EXPECT_FALSE(fiveD->equal(fiveF));
    EXPECT_TRUE(tenD->equal(new Franc(15)));
}
```

위 테스트를 통과시키려면 해결해야 할 일이 두 가지가 있다. 첫 번째는 Money 클래스 안에 `equal(Money * money)`를 구현하는 것이고, 두 번째는 Dollar와 Franc를 구분할 수 있는 변수를 추가하는 것이다. 우선 `equal()` 먼저 구현해보자.

```
class Money{
public:
    ...
    bool equal(Money * money){
        if (amount == money->amount) return true;
        else return false;
    }
    ...
};
```

다음과 같이 코드를 구현하고 테스트를 돌려보니 두개의 실패가 발생했다.

```
[ RUN      ] TDDEExample.TestMoney
c:\Users\Wbluemoon\documents\visual studio 2013\projects\TddExample\TddExample\TddExamplemain.cpp(56): error: Value of: fiveD->equal(fiveF)
  Actual: true
Expected: false
c:\Users\Wbluemoon\documents\visual studio 2013\projects\TddExample\TddExample\TddExamplemain.cpp(57): error: Value of: tenD->equal(new Franc(15))
  Actual: false
Expected: true
[ FAILED ] TDDEExample.TestMoney (3 ms)
```

사진 6: 테스트 주도 개발 예제 결과 - 비교

같은 Dollar나 Franc끼리 비교한 경우엔 정상적으로 동작하지만, 다른 통화끼리 비교할 때에는 값만 비교하는 형식 때문에 환율이 고려가 되지 않았다. 이 부분을 해결해보자.

일단 Dollar와 Franc에서 각각의 통화가 무엇인지 기억하고, 환율을 기억하고 있다가 비교 시 기준 통화(USD)로 바꿔서 비교를 하면 될 것 같다. 그런데 환율을 누가 기억해야 할까? Money 객체에서 기억하는 것은 뭔가 이상한 것 같다. 나중에 환율에 대한 정보를 추가해주어야 하는데, 그럴 경우 만들어진 모든 객체에 환율 정보를 추가해야 하기 때문이다. 환율에 관한 정보를 가진 Bank 객체를 따로 만드는 것이 좋을 것 같다.

다음과 같은 테스트를 통과하는 것이 이번 목표이다.

```
TEST(TDDEExample, TestMoney){
```

```

    Money * fiveD = new Dollar(5);
    Money * fiveF = new Franc(5);
    Money * tenD = new Dollar(10);

    EXPECT_TRUE(fiveD->equal(bank, new Dollar(5)));
    EXPECT_TRUE(fiveF->equal(bank, new Franc(5)));
    EXPECT_FALSE(fiveD->equal(bank, fiveF));
    EXPECT_TRUE(tenD->equal(bank, new Franc(15)));
}

```

다음은 Money 클래스에 추가된 코드와 Bank 클래스를 정의한 코드이다.

```

class Money{
friend class Bank;

protected:
    int amount;
    string currency;
public:
    Money(int amount_);
    int times(int multiplier);
    bool equal(Bank bank, Money * money);
    string getCurrency();
};

class Bank{
private:
    map<string, double> rateMap;
public:
    Bank();
    void addRate(string currency_, double rate_);
    double getRate(Money * money);
};

Money::Money(int amount_) : amount(amount_){
}

int Money::times(int multiplier){
    return amount*multiplier;
}

bool Money::equal(Bank bank, Money * money){
    if (amount / (bank.getRate(this)) == (money->amount / bank.getRate(money))) return true;
    else return false;
}

string Money::getCurrency(){
    return currency;
}

Bank::Bank(){
}

void Bank::addRate(string currency_, double rate_){
    pair<string, double> rate1(currency_, rate_);
}

```

```

        rateMap.insert(rate1);
    }

    double Bank::getRate(Money * money){
        map<string, double>::iterator i;
        for (i = rateMap.begin(); i != rateMap.end(); i++){
            if (i->first == money->getCurrency()) return i->second;
        }
        cout << "There is no match " << money->currency << " " << money->amount << endl;
        return 0;
    }

    class Dollar : public Money{
    public:
        Dollar(int amount_):Money(amount_) {
            currency = "USD";
        }
    };

    class Franc : public Money{
    public:
        Franc(int amount_):Money(amount_){
            currency = "CHF";
        }
    };

```

Bank 클래스를 만들어 통화와 환율을 Pair 로 기억하는 변수를 만들고 그 변수에 환율 정보를 더하거나 (addRate()) 통화 단위로부터 환율값을 알아내는 함수(getRate())를 만들었다.

Money 클래스에는 currency 변수를 만들어 각각의 개체가 어떤 통화 단위인지를 알 수 있도록 했으며, equal 함수를 실행할 때 환율을 알아와 기준 통화로 바꾼 뒤 비교하도록 바꾸었다.¹⁰

```

[ RUN      ] TDDEExample.TestMoney
There is no match USD 5
There is no match USD 5
There is no match USD 5
There is no match USD 10
c:\users\bluemoon\documents\visual studio 2013\projects\tddeexample\tddeexample\tddeexamplemain.cpp(102): error: Value of: tenD->equal(bank, new Franc(15))
Actual: false
Expected: true
[ FAILED   ] TDDEExample.TestMoney <7 ms>

```

사진 7: 테스트 주도 개발 예제 결과 - Dollar 와 Franc 비교

수행 결과 에러가 발생했다. CHF 에 대한 환율은 넣었지만, USD 에 대한 환율 정보가 없기 때문이다. Bank 의 생성자에 USD 환율에 관한 정보를 넣어주자.

```

Bank::Bank(){
    addRate("USD", 1.0);
}

```

10. 사실 이렇게 한번에 많이 만드는 것은 테스트 주도 개발의 원칙에 어긋난다. 좀 더 작은 단계로 쪼개어 개발을 해야한다. 하지만 그것들을 다 쓰는 것은 공간낭비라고 생각하여 몇 사이클 정도를 생략했다.

```

[=====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 2 tests from TDDEExample
[ RUN      ] TDDEExample.TestMultiply
[      OK   ] TDDEExample.TestMultiply (0 ms)
[ RUN      ] TDDEExample.TestMoney
[      OK   ] TDDEExample.TestMoney (0 ms)
[-----] 2 tests from TDDEExample (2 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test case ran. (5 ms total)
[ PASSED   ] 2 tests.
계속하려면 아무 키나 누르십시오 . . .

```

사진 8: 테스트 주도 개발 예제 결과 - Dollar 와 Franc 비교 성공

테스트가 모두 통과되었다.

- 가격과 주를 곱해서 총 합계를 구하는 기능 : $2 * 5\text{USD} = 10\text{USD}$
- USD 와 CHF 를 더해서 USD 로 나타내는 기능 : $10\text{USD} + 15\text{CHF} = 20\text{USD}$
- Dollar 와 Franc 클래스의 상위 클래스인 Money 클래스
- Dollar 와 Franc 클래스의 비교 : $\text{Dollar}(5) \neq \text{Franc}(5)$

이제 통화 단위가 다른 두 값을 더하는 기능을 구현할 차례가 되었다. 우선 테스트를 작성하자.

```

TEST(TDDEExample, TestAdd){
    Money * tenD = new Dollar(10);
    Money * fifteenF = new Franc(15);
    Money * Sum;

    Bank bank;
    bank.addRate("CHF", 1.5);
    EXPECT_TRUE((new Dollar(20))->equal(bank, tenD->add(bank, fifteenF, "USD")));
}

```

Money 클래스의 add 부분을 구현한 코드이다.

```

Money * Money::add(Bank bank, Money * money, string currency_){
    return new Money((int)(amount / (bank.getRate(this)) + (money->amount /
bank.getRate(money))), currency_);
}

```

리턴하는 Money 객체의 통화를 지정하려다 보니 통화 단위를 받아서 Money 객체를 생성하는 생성자가 필요하다. 생성자 부분을 수정했다.

```

Money(int amount_ = 0, string currency_ = "USD") : amount(amount_), currency(currency_){}

```

이것으로 모든 테스트가 성공했다. 이제 Money 클래스와 그 자식 클래스들(Dollar 와 Franc) 일정 금액을 나타내는 객체를 만들 수 있으며, 그 객체를 이용해 금액을 몇배로 만들거나, 각각의 객체끼리 더하는 기능을 이용할 수 있다. 더 많은 기능을 추가하거나, 코드가 길어진 부분을 private member function 으로 쪼개는 방식으로 좀 더 리팩토링을 할 수 있지만 여기서 마치겠다.

3. 구글 C++ 테스트 프레임워크

유닛 테스트이든, 테스트 주도 개발이든 테스트를 주로 사용하는 경우에는, 테스트들을 쉽고 빠르게 사용할 수 있도록 도와주는 프레임워크가 필수적이다. 그러한 테스트 프레임워크는 언어별로 나누어져 있고, 각 언어별로도 여러종류가 있다. C++에도 많은 테스트 프레임워크¹¹가 있지만, 그 중에 ‘구글 C++ 테스트 프레임워크’를 선택해 사용해보았다.

3.1. 구글 C++ 테스트 프레임워크란 무엇인가?

구글 C++ 테스트 프레임워크(또는 구글 테스트)는 C++에서 사용 가능한 Testing Framework로 xUnit¹²같은 테스트 프레임워크이다. 다양한 환경에서 특별한 프로그램 설치 없이 사용가능하며, 그를 위해 Exception을 사용하지 않는 것이 특징이다.

구글 테스트의 지향점은 다음과 같다.

1. 테스트는 독립적이고 반복 가능해야 한다. 하나의 테스트가 다른 테스트로부터 영향을 받는다면 문제를 찾기 참 곤란할 것이다. 구글 C++ 테스트 프레임워크는 각각의 테스트를 다른 객체를 이용해 테스트함으로써 독립적으로 테스트가 이루어 질 수 있도록 하였다. 이런 방식은 테스트 실패 시 문제점을 빨리 찾는 것을 도와준다.
2. 테스트는 잘 구성되어야 하며 테스트되는 코드의 구조를 반영해야 한다. 구글 C++ 테스트 프레임워크는 테스트를 테스트 케이스와 연결시키는데, 이 테스트 케이스는 데이터와 서브루틴을 공유할 수 있게 해준다. 이러한 패턴은 테스트를 쉽게 만들고 쉽게 유지할 수 있도록 해준다.
3. 테스트는 이식성이 좋아야 하고 재사용이 가능해야 한다. 오픈 소스 커뮤니티의 코드들은 플랫폼에 영향받지 않고(platform-neutral) 따라서 테스트들도 플랫폼에 영향받지 않아야 한다. 구글 테스트는 여러 OS들(Windows, Mac OS, Linux)과 여러 컴파일러들(gcc, MSVC, etc)에서 사용 가능하다.
4. 테스트가 실패하면 최대한 많은 정보를 제공해야 한다. 구글 C++ 테스트 프레임워크는 첫 번째 실패에도 정지하지 않는다. 대신 현재 테스트만을 멈추고 다음 테스트를 계속 수행한다. 이러한 방식은 한 번에 여러개의 문제점들을 찾을 수 있게 해준다.
5. 테스트 프레임워크는 개발자가 자잘한 일이 아닌, 테스트 내용에 집중할 수 있게 해주어야 한다. 구글 C++ 테스트 프레임워크는 모든 테스트를 자동으로 추적하므로, 여러 테스트들을 한 번에 실행하기 위해 고생할 필요가 없다.

11. http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C.2B.2B

12. 자바를 위한 테스트 프레임워크인 JUnit을 모방한 테스트 프레임워크들의 총칭. x 자리에 사용하는 언어가 들어간다. 예를 들어, Python을 위한 PyUnit, C++을 위한 CppUnit 등

6. 테스트는 빨라야한다. 구글 C++ 테스트 프레임워크를 사용하면 set-up/tear-down 과정을 이용해 테스트들끼리 영향받는 일 없이 공유된 자원을 사용할 수 있게 해준다.

3.2. Google Test 설치 방법

사용환경 : Window 7 & Visual Studio 2013

구글 테스트는 여러 OS 와 여러 컴파일러를 지원하지만, 이 문서에서는 윈도우 7 & 비주얼 스튜디오(이하 VS)2013 을 기준으로 설명한다.

구글 테스트를 사용하기 위한 준비과정은 다음과 같다.

1. 구글 테스트를 다운받는다. (<https://code.google.com/p/googletest/downloads/list>)
2. 라이브러리 파일을 생성한다.
3. 라이브러리 파일과 기타 필요한 파일들을 프로젝트에 포함시킨다.

매우 쉽다. 파일을 받고 컴파일 한번 한 뒤, 구글 테스트 파일들을 자신의 프로젝트에서 사용하면 되는 것이다. 좀 더 자세히 알아보자.

1. 구글 테스트를 다운받는다.

: 구글 테스트의 메인 페이지(<https://code.google.com/p/googletest/>)로 들어가면, Downloads 탭과 Wiki 탭이 보인다. Downloads 탭에서 구글 테스트를 받을 수 있으며, Wiki 탭에는 약간의 샘플과 구글 테스트에 관련한 공식 문서들을 볼 수 있다.

2. 라이브러리 파일을 생성한다.

: 받은 파일의 압축을 풀면 다양한 폴더들이 보인다. 여러 환경에서 사용할 수 있는 파일들을 모아놓았기 때문에 폴더의 수는 많으나 VS 에서 사용할 폴더는 두 가지이다. include 폴더와 msvc 폴더이다.

msvc 폴더에 들어가면 gtest.sln 와 gtest-md.sln 두개의 솔루션 파일을 볼 수 있다. gtest.sln 은 정적 라이브러리를 생성하며 gtest-md.sln 은 DLL 버전의 라이브러리를 생성한다. 작성하는 프로젝트의 특성에 따라 선택하면 된다. VS 프로젝트 기본 설정¹³이 MDd 로 되어 있었으므로 gtest-md.sln 을 사용하겠다.

gtest-md.sln 파일을 실행 전, 파일들의 읽기 전용 옵션을 해제해야 한다.(VS 2013 에서 열면 단방향 업그레이드를 해야 한다고 나오는데, 그냥 실행을 했을 경우 권한이 없다면서 종료된다.) 실행을 한 뒤 컴파일을 하면 msvc/gtest-md/Debug 에 gtestd.lib 파일이 생성된다.

3. 라이브러리 파일과 기타 필요한 파일들을 프로젝트에 포함시킨다.

: 크게 보면 두가지 방법이 있다. 첫 번째 방법은 필요한 파일들을 복사해서 자신의 프로젝트 파일에 붙여 넣는 것이다. include 폴더와 위에서 생성한 gtestd.lib 파일을 복사해 사용할 프로젝트 폴더에 붙여 넣으면 된다. 두 번째 방법은 구글 테스트 파일들이 있는 폴더를 프로젝트에 포함시키는 것이다. 프로젝트의 [속성] - [구성 속성] - [VC++ 디렉터리] - [포함 디렉터리]에 include 폴더를 추가하고, [라이브러리 디렉터리]에 gtestd.lib 가 있는 Debug 폴더를 포함시키면 된다.

여러 프로젝트에서 구글 테스트를 사용할 경우 두 번째 방법이 편할 것으로 생각되나, 이번에는 그냥 파일들을 복사-붙여넣기 하는 방식으로 사용하였다.

마지막으로 소스 파일에 다음의 코드를 추가한다.

13. 프로젝트의 [속성] - [구성 속성] - [C/C++] - [코드 생성] - [런타임 라이브러리] 에서 확인할 수 있다.


```
#include "gtest/gtest.h"
#pragma comment (lib, "gtestd.lib")

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

    return RUN_ALL_TESTS();
}
```

이제 구글 테스트를 사용할 준비가 다 끝났다.

3.3. 기본 개념

구글 테스트는 기본적으로 단정문을 사용한다. 단정문의 결과로는 성공, 치명적인 실패, 치명적이지 않은 실패의 3가지가 있다. 치명적인 실패가 발생하면 실행중인 함수가 끝나지만(`abort`), 나머지 경우엔 계속 진행된다.

3.4. 간단한 예제

기본적인 테스트 방법은 `TEST(test_case_name, test_name)` 매크로를 만들고 그 바디에 원하는 실행문과 단정문을 넣는 것이다. 여기서 `test_case_name`은 테스트들의 그룹을 의미하며 `test_name`은 각각의 테스트의 이름이다. Google Test는 test case 별로 그 결과를 출력하므로 연관성이 있는 테스트들은 같은 `test_case_name`으로 묶는 것이 좋다. `test_case_name`은 test fixture를 사용할 때에도 쓰인다.

`test_case_name`과 `test_name` 모두 C++ identifier 조건에 맞아야 하며, `_`로 시작하는 것은 권장하지 않는다.

```
// 0 팩토리얼의 결과값이 1 인지를 테스트
TEST(FactorialTest, HandlesZeroInput)
EXPECT_EQ(1, Factorial(0));

// 팩토리얼값이 제대로 나오는지 테스트
TEST(FactorialTest, HandlesPositiveInput)
EXPECT_EQ(1, Factorial(1));
EXPECT_EQ(2, Factorial(2));
EXPECT_EQ(6, Factorial(3));
EXPECT_EQ(40320, Factorial(8));
```

두 테스트 모두 `test_case_name`으로 `FactorialTest`를 사용했고, `test_name`으로는 각각의 특성을 반영한 이름을 사용했다.

3.5. 구글 테스트에서 사용하는 단언문들(Assertions)

ASSERT_XXXX 형식의 단언문의 실패 치명적인 실패이며, 실행중인 함수를 정지시킨다. 만약 이 형식의 단언문을 통과하지 못하면 그 이후에도 연속적인 실패가 예상될 때 사용한다.

EXPECT_XXXX 형식의 실패는 치명적이지 않은 실패이며, 성공/실패 여부와 상관없이 계속 진행한다.

3.5.1. 참/거짓 단언문

치명적인 실패	치명적이지 않은 실패	확인하는 내용
ASSERT_TRUE(<i>condition</i>)	EXPECT_TRUE(<i>condition</i>)	<i>condition</i> 이 참인가?
ASSERT_FALSE(<i>condition</i>)	EXPECT_FALSE(<i>condition</i>)	<i>condition</i> 이 거짓인가?

3.5.2. 비교 단언문

치명적인 실패	치명적이지 않은 실패	확인하는 내용
ASSERT_EQ(<i>expected</i> , <i>actual</i>);	EXPECT_EQ(<i>expected</i> , <i>actual</i>);	<i>Expected</i> 와 <i>actual</i> 가 같은가
ASSERT_NE(<i>val1</i> , <i>val2</i>);	EXPECT_NE(<i>val1</i> , <i>val2</i>);	<i>val1</i> 과 <i>val2</i> 가 같지 않은가
ASSERT_LT(<i>val1</i> , <i>val2</i>);	EXPECT_LT(<i>val1</i> , <i>val2</i>);	<i>val1</i> 가 <i>val2</i> 보다 작은가
ASSERT_LE(<i>val1</i> , <i>val2</i>);	EXPECT_LE(<i>val1</i> , <i>val2</i>);	<i>val1</i> 가 <i>val2</i> 보다 작거나 같은가
ASSERT_GT(<i>val1</i> , <i>val2</i>);	EXPECT_GT(<i>val1</i> , <i>val2</i>);	<i>val1</i> 가 <i>val2</i> 보다 큰가
ASSERT_GE(<i>val1</i> , <i>val2</i>);	EXPECT_GE(<i>val1</i> , <i>val2</i>);	<i>val1</i> 가 <i>val2</i> 보다 크거나 같은가

여기서 주의점은 ASSERT_EQ와 EXPECT_EQ이다. 왼쪽에 예상값이, 오른쪽에 실행결과가 들어간다. 그 아래의 것들은 그러한 조건이 없다.¹⁴

3.5.3. 문자열 관련 단언문

치명적인 실패	치명적이지 않은 실패	확인하는 내용
ASSERT_STREQ(<i>expected_str</i> , <i>actual_str</i>);	EXPECT_STREQ(<i>expected_str</i> , <i>actual_str</i>);	두 문자열의 내용이 같은가?
ASSERT_STRNE(<i>str1</i> , <i>str2</i>);	EXPECT_STRNE(<i>str1</i> , <i>str2</i>);	두 문자열의 내용이 다른가?
ASSERT_STRCASEEQ(<i>expected_str</i> , <i>actual_str</i>);	EXPECT_STRCASEEQ(<i>expected_str</i> , <i>actual_str</i>);	두 문자열의 내용이 같은가? (대소문자 구분 무시)
ASSERT_STRCASENE(<i>str1</i> , <i>str2</i>);	EXPECT_STRCASENE(<i>str1</i> , <i>str2</i>);	두 문자열의 내용이 다른가? (대소문자 구분 무시)

14. 구글 테스트 Document에는 그렇게 써져 있지만, 직접 해본 결과 들어가는 위치가 바뀌어도 문제는 발생하지 않았다.

주의점 : Null pointer 는 비어있는 문자열과 다른 것으로 취급된다.

‘CASE’는 대소문자(Case) 구분을 무시하는 것을 의미한다.

STREQ 단언문과 STRNE 단언문은 확장문자¹⁵(wide C strings, wchar_t 타입)도 사용 가능하다.

3.5.4. 명시적인 성공과 실패

SUCCEED()

하나의 성공을 의미한다.

함수	결과
FAIL();	치명적인 실패
ADD_FAILURE();	치명적이지 않은 실패
ADD_FAILURE_AT("file_path", line_number)	file_path 의 파일의 line_number 에서 치명적이지 않은 실패를 생성한다.

이러한 단정문들은 ASSERT_TRUE(1!=1)이나 EXPECT_TRUE(1!=1)과 같이 참/거짓 단언문을 이용해 실패를 생성하는 수고를 줄여준다.

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

주로 위의 코드와 같은 형식으로 사용하게 된다.

주의사항 : FAIL()은 리턴값이 void 인 함수 내부에서만 사용해야 한다.

3.5.5. Exception 단언문

치명적인 실패	치명적이지 않은 실패	확인하는 내용
ASSERT_THROW(statement, exception_type);	EXPECT_THROW(statement, exception_type);	statement 가 exception_type 타입의 예외를 발생하는가
ASSERT_ANY_THROW(statement);	EXPECT_ANY_THROW(statement);	statement 가 예외를 발생하는가

15. http://en.wikipedia.org/wiki/Wide_character

ASSERT_NO_THROW(<i>statement</i>);	EXPECT_NO_THROW(<i>statement</i>);	<i>statement</i> 가 예외를 발생하지 않는가
--------------------------------------	--------------------------------------	------------------------------------

*statement*에 들어가는 함수가 예외를 발생하는지를 확인할 때 사용한다.

3.5.6. 더 나은 예리 메시지를 위한 단정문

구글 테스트는 많은 종류의 단정문을 제공하고 있지만, 사용자가 원하는 모든 단정문을 제공할 수는 없다. 그런 이유로 사용자는 보통 EXPECT_TRUE()를 사용하여 복잡한 표현을 테스트하게 되는데 이 경우 표현식(expression)에 사용한 값을 알 수가 없어서 어떤 부분이 문제인지 확인하기 어려운 경우가 발생한다. 실패 메시지를 만들어 사용하는 방법도 있지만 부작용(Side Effect)가 발생할 수도 있으므로 번거롭다. 이런 문제를 해결하기 위해 구글 테스트에서는 3 가지 해결책을 제안하고 있다.

3.5.6.1. Boolean 함수 사용하기

치명적인 실패	치명적이지 않은 실패	확인하는 내용
ASSERT_PRED1(<i>pred1</i> , <i>val1</i>);	EXPECT_PRED1(<i>pred1</i> , <i>val1</i>);	<i>pred1(val1)</i> 가 참을 반환하는가
ASSERT_PRED2(<i>pred2</i> , <i>val1</i> , <i>val2</i>);	EXPECT_PRED2(<i>pred2</i> , <i>val1</i> , <i>val2</i>);	<i>pred2(val1, val2)</i> 가 참을 반환하는가
...

`bool predN(val1, val2..., valN)` 형태이 함수에 인자로 {*val1*, *val2*..., *valN*}을 집어넣어 리턴값을 확인하는 단언문들이다.

```
// Returns true iff m and n have no common divisors except 1.
bool MutuallyPrime(int m, int n) { ... }
const int a = 3;
const int b = 4;
const int c = 10
```

이러한 예제를 생각해보자.

EXPECT_PRED2(MutuallyPrime, a, b); 의 경우엔 테스트에 성공하며

EXPECT_PRED2(MutuallyPrime, b, c); 은 테스트 실패와 함께 다음 메시지를 출력한다.

```
!MutuallyPrime(b, c) is false, where
b is 4
c is 10
```

단지 성공/실패 뿐만이 아닌, 어떤 변수에 어떤 값이 들어갔기 때문에 실패했는지를 확인할 수 있다.

주의사항 : 현재 인자가 5 개 이하인 경우에만 사용가능하다.

3.5.6.2. AssertionResult 를 반환하는 함수 사용하기

위의 방법은 확실적이고 조금 사용하기 불편할 수도 있다. 그런 경우, 예러 메시지를 마음대로 조작할 수 있는 이 방법을 사용하는 것도 좋은 방법이 될 것이다.

다음은 ::testing::AssertionResult 클래스이다

```
namespace testing {  
  
    // Returns an AssertionResult object to indicate that an assertion has succeeded.  
    AssertionResult AssertionSuccess();  
  
    // Returns an AssertionResult object to indicate that an assertion has failed.  
    AssertionResult AssertionFailure();  
  
}
```

<< 연산자를 이용해 자신이 원하는 메시지를 추가시킬 수 있다. 불리언을 리턴하는 함수 대신 내용이 추가된 AssertionResult 를 리턴하는 함수를 만들면, 테스트 실패 시 더 자세한 리포트를 받을 수 있다.

예를 들어 다음의 함수 대신

```
bool IsEven(int n) {  
    return (n % 2) == 0;  
}
```

이런 형태의 함수를 사용하면 추가적인 정보를 얻을 수 있다.

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

테스트 결과 예시

```
테스트 코드 : EXPECT_FALSE(IsEven(8));  
Value of: !IsEven(8)  
Actual: true (8 is even)  
Expected: false  
  
테스트 코드 : EXPECT_TRUE(IsEven(3));
```

```
Value of: !IsEven(3)
Actual: false (3 is odd)
Expected: true
```

3.5.6.3. 서술 형식자(Predicate-Formatter) 사용하기

<< 오퍼레이션을 지원하지 않거나 사용할 경우 부작용(Side Effect)가 발생하는 경우, 혹은 위의 두 방법으로 부족한 경우에는 서술 형식자(Predicate-Formatter)를 사용해 에러 메시지를 사용자가 원하는 대로 구성할 수 있다.

치명적인 실패	치명적이지 않은 실패	확인하는 내용
<code>ASSERT_PRED_FORMAT1(pred_format1, val1);</code>	<code>EXPECT_PRED_FORMAT1(pred_format1, val1);</code>	<code>pred_format1(val1)</code> 이 성공인가
<code>ASSERT_PRED_FORMAT2(pred_format2, val1, val2);</code>	<code>EXPECT_PRED_FORMAT2(pred_format2, val1, val2);</code>	<code>pred_format2(val1, val2)</code> 이 성공인가
...

기본적으로 첫 번째 방법과 비슷하지만 함수(`predN`) 대신 서술 형식자(`pred_formatN`)을 사용하는 것에서 차이가 있다.

서술 형식자(Predicate_Formatter)의 형식은 다음과 같다.

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

`expr1...exprn`은 소스 코드에서 사용하는 변수의 이름이고, `val1...varn`은 각 변수들의 값을 의미한다. `T1...Tn`에는 값 또는 참조자 타입 두가지 모두 사용가능하다. 예를 들면 타입이 `Foo` 라면, `Foo` 와 `const &Foo` 모두 사용 가능하다.

실제 예를 보는 것이 이해가 빠르다. 다음의 예를 보자.

```
// 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);
}

```

이런 형식의 함수와 서술 형식자가 있다고 하자.

EXPECT_PRED_FORMAT2(AssertMutuallyPrime, b, c); 를 수행하면 다음과 같은 메시지를 출력한다.

```

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

```

3.5.7. 부동소수점 비교 단정문

부동소수점 비교는 그 특유의 연산방식 때문에 정확히 같은 값을 비교하는 것은 거의 불가능하다. 그런 이유로 EXPECT_EQ() 로 비교할 경우 의도와 다른 결과를 얻을 수도 있다. 따라서 부동소수점 비교를 위해선 적절한 오차 범위 내에서 비교를 수행해야 하는데, 구글 테스트에서는 3 가지의 단정문을 지원하고 있다.

치명적인 실패	치명적이지 않은 실패	확인하는 내용
ASSERT_FLOAT_EQ(<i>expected</i> , <i>actual</i>);	EXPECT_FLOAT_EQ(<i>expected</i> , <i>actual</i>);	두 float 타입의 값이 거의 같은가
ASSERT_DOUBLE_EQ(<i>expected</i> , <i>actual</i>);	EXPECT_DOUBLE_EQ(<i>expected</i> , <i>actual</i>);	두 double 타입의 값이 거의 같은가
ASSERT_NEAR(<i>val1</i> , <i>val2</i> , <i>abs_error</i>);	EXPECT_NEAR(<i>val1</i> , <i>val2</i> , <i>abs_error</i>);	두 값의 차이가 <i>abs_error</i> 보다 작은가

위의 두가지 단언문은 ULPs¹⁶를 이용한 일종의 디폴트 단언문이다. 세 번째의 ASSERT/EXPECT_NEAR에서는 사용자가 직접 오차 범위를 지정할 수 있다.

3.5.8. 타입 단정문

```

::testing::StaticAssertTypeEq<T1, T2>();

```

정확히 말하면 단정문은 아니라고 생각한다. 이전의 단정문들은 성공이든 실패이든 컴파일되어 실행 되지만, 이 함수는 조건이 맞지 않으면 컴파일 자체가 안 되기 때문이다. 위 함수는 T1 와 T2 의 타입이 같으면 아무런 일도 하지 않고 넘어가지만, 타입이 다르면 컴파일에 실패하게 되며 (컴파일러에 따라 다르겠지만) T1 와 T2 의 값을 보여준다. 보통 템플릿 코드 내부에서 사용한다.

```

template <typename T> class Foo {

```

16. http://en.wikipedia.org/wiki/Unit_in_the_last_place

```
public:
    void Bar() { ::testing::StaticAssertTypeEq<int, T>(); }
};
```

간단한 예제이다. 다음의 코드는 템플릿 타입으로 int 가 들어오면 정상적으로 실행되겠지만, int 외의 타입이 들어오면 컴파일에 실패한다.

주의사항 : StaticAssertTypeEq<T1, T2>()가 실행되어야지만 타입의 일치 여부를 확인할 수 있다. 위의 코드를 예로 들어보자.

```
void Test1() { Foo<bool> foo; }
```

이런 형태의 코드는 Foo 클래스를 생성하지만 StaticAssertTypeEq<T1, T2>()가 들어있는 Bar() 함수를 사용하지 않는다. 따라서 비교가 수행되지 않으며 정상적으로 컴파일 된다. 하지만

```
void Test2() { Foo<bool> foo; foo.Bar(); }
```

위 코드와 같이 Bar()가 실행되는 코드가 들어가는 순간, 비교가 수행되고 컴파일에 실패하게 된다.

3.6. 단정문의 위치

단정문은 모든 위치에서 사용 가능하다. 하지만 치명적인 실패를 만들어내는 FAIL()이나 ASSERT_XXXX()의 경우엔 리턴 타입이 void 인 함수 내부에서만 사용 가능하다. 구글 테스트는 Exception 을 사용하지 않기 때문이다. 만약 리턴 타입이 void 가 아닌 함수 내부에서 두 종류의 함수를 사용하면 컴파일 에러를 띄우게 된다.

주의사항 : 클래스의 생성자와 소멸자는 보통 리턴 타입이 void 인 함수로 여겨지지 않는다. 따라서 생성자와 소멸자 내부에서 FAIL()이나 ASSERT_XXXX()를 사용하면 컴파일 에러가 뜬다.

이 문제를 해결할 수 있는 방법은 리턴타입이 void 인 private 함수를 하나 만들어 생성자와 소멸자에 들어갈 내용을 그 함수로 모두 옮기고, 생성자와 소멸자에서는 해당 private 함수를 호출만 하는 것이다. 하지만, 이 경우 단정문 실패가 발생하면 생성자/소멸자에서 도중에 빠져나오므로, 객체는 일부만 생성되거나 일부만 소멸된 상태일 수 있다. 이런 상황에서는 단정문 사용에 주의해야 한다.

3.7. 서브루틴에서 단정문 사용하기

단정문들이 들어있는 하나의 서브루틴을 여러 곳의 테스트에서 호출 할 경우¹⁷ 이 서브루틴이 실패할 경우, 어떤 테스트가 부른 서브루틴에서 실패했는지 알기 힘들다. 이때 다음의 매크로를 사용하면 편리하다.

```
SCOPED_TRACE(message);
```

`message`에는 `std::ostream`에 전달할 수 있는 내용이라면 어떤 것이든 가능하다. 이 매크로는 현재 파일 이름, 줄 번호, `message` 내용을 실패 메시지에 추가한다.

```
10: void Sub1(int n) {
11:     EXPECT_EQ(1, Bar(n));
12:     EXPECT_EQ(2, Bar(n + 1));
13: }
14:
15: TEST(FooTest, Bar) {
16:     {
17:         SCOPED_TRACE("A"); // This trace point will be included in every failure in this
scope.
19:         Sub1(1);
20:     }
21:     // Now it won't.
22:     Sub1(9);
23: }
```

이 예제를 실행해보면 다음과 같은 출력메세지를 얻을 수 있다.

```
path/to/foo_test.cc:11: Failure
Value of: Bar(n)
Expected: 1
Actual: 2
Trace:
path/to/foo_test.cc:17: A

path/to/foo_test.cc:12: Failure
Value of: Bar(n + 1)
Expected: 2
Actual: 3
```

기본적인 메시지는 에러가 난 위치를 단정문이 작성된 줄 번호로 표시한다. 하지만 `SCOPED_TRACE()`를 추가한 경우 그와 함께 `SCOPED_TRACE()`가 적힌 줄 번호도 함께 표시하므로 어느 스코프에서 불린 테스트 서브루틴이 문제가 됐는지 쉽게 파악 가능하다.

17. 코드 중복을 막기 위해 동일한 내용의 단정문들을 하나의 테스트 서브루틴으로 만드는 경우

3.8. SCOPED_TRACE() 사용 팁

1. 적절한 메시지를 작성하면, 서브루틴을 호출한 곳이 아니라 서브루틴 앞에 SCOPED_TRACE()를 추가하는 것으로 충분할 수 있다.
2. 루프문 안에서 서브루틴을 사용할 경우, loop iterator 를 메시지에 포함시키는 방법으로 몇 번째 루프에서 문제가 발생했는지 추적할 수 있다.
3. 문제가 발생한 줄 번호를 아는 것만으로 충분하다면, 메시지 내용을 작성할 필요 없이 “” 만 넣어도 충분하다.
4. SCOPED_TRACE()를 바깥 스코프에서도 사용하고, 안쪽 스코프에서도 사용한 경우, SCOPED_TRACE()를 만난 순서의 반대로 메시지를 출력한다. 즉, 안쪽 스코프의 SCOPED_TRACE() 메시지를 출력하고 바깥쪽 스코프의 메시지를 출력한다.

3.9. 실패 전파하기

ASSERT_XXX()와 FAIL()을 사용하다보면, 해당 단정문들이 실패할 경우, 테스트 전체가 중단될 것이라고 생각하기 쉽다. 하지만 사실은 해당 단정문이 들어있는 **함수만이 종료**될 뿐이다.

```
void Subroutine() {
    // Generates a fatal failure and aborts the current function.
    ASSERT_EQ(1, 2);
    // The following won't be executed.
    ...
}

TEST(FooTest, Bar) {
    Subroutine();
    // The intended behavior is for the fatal failure in Subroutine() to abort the entire
    test.
    // The actual behavior: the function goes on after Subroutine() returns.
    int* p = NULL;
    *p = 3; // Segfault!
}
```

예를 들어, 다음과 같은 예제의 경우 서브루틴에서 치명적인 실패를 하기때문에,이 서브루틴이 사용된 Bar 테스트 또한 중단되어 Null 포인터에 값을 집어넣는 일이 발생하지 않을 것이라 예상하기 쉽지만, 실제로는 서브루틴만 도중에 빠져나왔을 뿐, Bar 테스트의 나머지는 정상적으로 진행해 당연하게 에러를 만들어낸다.

이러한 일을 막기 위해 구글 테스트에서는 두가지 해결책을 제시한다.

3.9.1. 서브루틴에서의 단정 확인

치명적인 실패	치명적이지 않은 실패	확인하는 내용
<code>ASSERT_NO_FATAL_FAILURE(<i>statement</i>);</code>	<code>EXPECT_NO_FATAL_FAILURE(<i>statement</i>);</code>	<i>statement</i> 가 현재 스레드에서 치명적인 실패를 하지 않았는가

오직 단정문을 실행한 스레드에서의 실패만이 확인된다. 만약 *statement*가 새로운 스레드를 생성했다면, 그 스레드 내부에서의 단정문에 의한 실패는 무시된다.

```
ASSERT_NO_FATAL_FAILURE(Foo());

int i;
EXPECT_NO_FATAL_FAILURE({ i = Bar(); });
```

위의 예와 같은 방법으로 사용한다.

3.9.2. 현재 테스트에서의 실패 확인하기

`::testing::TEST`에 있는 다음의 함수들을 이용해 현재 테스트에서 실패가 발생했는지를 확인할 수 있다,

함수	내용
<code>HasFatalFailure();</code>	현재 테스트에서 치명적인 실패가 발생했는가
<code>HasNonfatalFailure();</code>	현재 테스트에서 치명적이지 않은 실패가 발생했는가
<code>HasFailure();</code>	현재 테스트에서 어떤 종류의 실패가 발생했는가

다음과 같은 방식으로 사용할 수 있다. `if`문을 이용해서 이전 과정에서 실패가 발생하면 `return;` 하는 방식이다.

```
TEST(FooTest, Bar) {
    Subroutine();
    // Aborts if Subroutine() had a fatal failure.
    if (HasFatalFailure())
        return;
    // The following won't be executed.
    ...
}
```

3.10. 테스트 픽스처를 이용한 테스트

항상 균일한 테스트 결과를 얻기 위해 구글 테스트에서는 각각의 테스트 전과 후에 매번 테스트 픽스처를 만들고 정리하는 기능을 지원하고 있다.

3.10.1. 테스트 픽스처 생성 방법

1. `::testing::Test` 클래스를 상속받는 테스트 픽스처 클래스를 만든다.
2. 사용할 객체들을 정의한다.
3. 필요하다면 생성자나 `SetUp()` 내부에 테스트를 위한 준비를 한다. 이때, `Setup()`이 아닌 `SetUp()`인 것을 조심.
4. 필요하다면 소멸자나 `TearDown()` 내부에 생성자나 `SetUp()`에서 할당한 변수에 대한 정리를 하도록 한다.
5. 마지막으로 필요한 서브루틴들을 정의한다.

3.10.2. 테스트 픽스처 사용법

1. `TEST(Test_case_name, Test_name)`이 아닌 `TEST_F(Test_case_name, Test_name)`을 사용해 테스트를 작성한다. `TEST_F`의 F는 Fixture를 의미한다. 또한, `Test_case_name`은 테스트 픽스처 클래스의 이름과 동일해야 한다.(테스트 픽스처 클래스의 이름을 테스트 케이스의 이름과 같게 짓는다고도 볼 수 있다.)
2. 테스트를 실행한다.

3.10.3. 테스트 픽스처 사용 예제

```
template <typename E>
class Queue {
public:
    Queue();
    void Enqueue(const E& element);
    E* Dequeue();
    size_t size() const;
    ....
};
```

이런 형태의 큐를 테스트 해본다고 하자. 큐가 제대로 동작하는지 알기 위해서는 생성한 뒤 몇 개의 엘리먼트를 넣어두어야 할 것이다.¹⁸ 이 부분은 반복적이면서 큰 의미가 없는 부분이므로 몇 개의 엘리먼트가 들어간 큐를 준비해 두고, 그 큐들을 이용해 테스트를 하면 편할 것이다. 이때 엘리먼트가 들어있는 큐가 테스트 픽스처가 된다.

```
class QueueTest : public ::testing::Test {
protected:
    Queue<int> q0_;
    Queue<int> q1_;
    Queue<int> q2_;

    virtual void SetUp() {
        q1_.Enqueue(1);
        q2_.Enqueue(2);
        q2_.Enqueue(3);
    }

    //virtual void TearDown() {}
}
```

18. 큐에 엘리먼트를 삽입하는 기능은 테스트가 되어있다고 가정.

```
};
```

만들어진 테스트 픽스처의 모습이다. 3 개의 큐 q0_, q1_, q2_를 선언하고 SetUp()에서 q1_에 한 개의 엘리먼트를, q2_에는 두 개의 엘리먼트를 넣었다. 딱히 TearDown()에서 해야 할 일이 없으므로 TearDown()은 비워두었다.

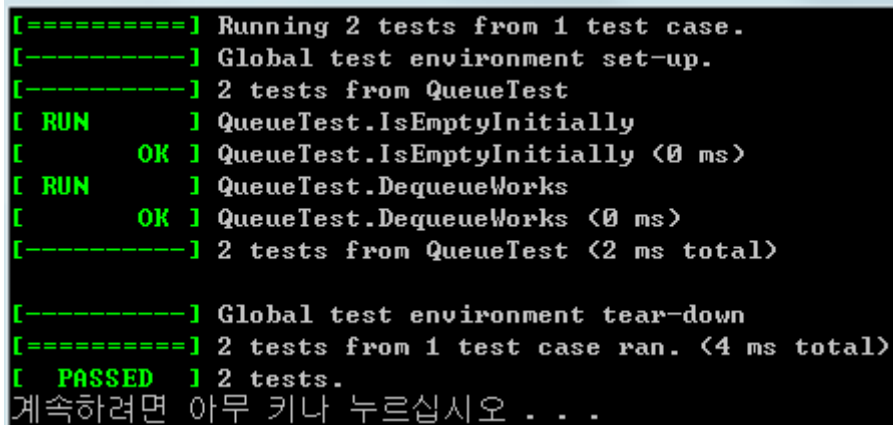
```
TEST_F(QueueTest, IsEmptyInitially) {
    EXPECT_EQ(0, q0_.size());

    q1_.Dequeue();
}

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

    n = q1_.Dequeue();
    ASSERT_TRUE(n != NULL);
    EXPECT_EQ(1, *n);
    EXPECT_EQ(0, q1_.size());
    delete n;

    n = q2_.Dequeue();
    ASSERT_TRUE(n != NULL);
    EXPECT_EQ(2, *n);
    EXPECT_EQ(1, q2_.size());
    delete n;
}
```



```
[=====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 2 tests from QueueTest
[ RUN     ] QueueTest.IsEmptyInitially
[         OK ] QueueTest.IsEmptyInitially (0 ms)
[ RUN     ] QueueTest.DequeueWorks
[         OK ] QueueTest.DequeueWorks (0 ms)
[-----] 2 tests from QueueTest (2 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test case ran. (4 ms total)
[ PASSED ] 2 tests.
계속하려면 아무 키나 누르십시오 . . .
```

사진 9: 테스트 픽스처 사용 예제 결과

테스트 픽스처를 사용하는 두개의 테스트와 그 결과이다. IsEmptyInitially 테스트에서 q1_.Dequeue()를 실행하지만, DequeueWorks 테스트에는 아무런 영향도 미치지 않은 것을 볼 수 있다.

3.10.4. 테스트 픽스처를 이용한 테스트 실행 과정

위의 코드가 실행되는 것을 예로 들자면,

1. 픽스처 객체를 생성하고 SetUp()을 실행한다. 생성된 픽스처 객체를 t1 이라 하자.
2. t1 을 이용해 첫 번째 테스트를 실행한다.

3. 첫 번째 테스트가 끝나면 `t1.TearDown()`을 실행해 정리한다.
4. `t1` 을 소멸시킨다.
5. 새로운 픽스처 객체 `t2` 를 생성하고 `t1.Setup()`을 실행한다.
6. `t2` 를 이용해 두 번째 테스트를 실행한다.
7. 다 끝나면 `t2.TearDown()`을 실행하고 `t2` 를 소멸시킨다.

이런 식으로 새로운 테스트(`Test_name` 이 다른 테스트)를 할 때마다 매번 픽스처 객체를 새로 만들어 `SetUp()`을 이용해 초기설정을 한 뒤 이 픽스처 객체를 이용해 테스트를 실행한다.

3.11. 같은 테스트 케이스 안의 테스트들끼리의 자원 공유하기

구글 테스트는 테스트 별로 테스트 픽스처를 새로 만들어서 제공하지만 비용이 많이 들거나, 픽스처 내용에 변화가 없어서 같은 테스트 픽스처 개체를 여러 테스트에서 사용해도 된다면 하나의 픽스처 개체를 공유하게 만들 수 있다. 방법은 다음과 같다.

1. 테스트 픽스처 클래스에 공유할 변수들을 `static` 으로 정의한다.
2. 같은 픽스처 클래스에 `static void SetUpTestCase()`와 `static void TearDownTestCase()`를 정의해 사용한다.

공유자원이 있는 테스트 픽스처 클래스를 사용하는 테스트는 다음과 과정을 통해 동작한다.

1. `SetUpTestCase()`가 먼저 실행되어 픽스처 개체의 공유 자원이 초기화된다.
2. `SetUp()`이 호출되어 그 외의 자원들도 초기화된다.
3. 첫 번째 테스트가 실행된다.
4. `TearDown()`이 호출되어 사용한 자원이 정리된다.
5. `SetUp()`이 호출되어 자원을 다시 초기화한다.
6. 두 번째 테스트가 실행된다.
7. `TearDown()`이 호출되어 사용한 자원이 정리된다.
8. 5~7 과정을 해당 테스트 케이스 내의 모든 테스트가 수행될 때까지 반복한다.
9. `TearDownTestCase()`를 수행하여 공유 자원을 정리한다.

주의사항

1. 픽스처 클래스의 사용 방법상, 이 방법으로는 다른 테스트 케이스간의 자원 공유는 불가능하다.
2. 테스트 간의 선후관계는 정해져있지 않다는 것에 주의한다. 즉, 먼저 테스트가 공유 자원을 변경시켰을 것이라 가정하는 테스트는 의도와 다른 수행결과를 보여줄 수 있다.
3. 만약 테스트가 공유 자원을 변경시켰을 경우엔 테스트의 끝에서 다시 자원을 원상복귀 시켜야 한다.

다음은 사용 예제이다.

```

class FooTest : public ::testing::Test {
protected:
    // Per-test-case set-up.
    // Called before the first test in this test case.
    // Can be omitted if not needed.
    static void SetUpTestCase() {
        shared_resource_ = new ...;
    }

    // Per-test-case tear-down.
    // Called after the last test in this test case.
    // Can be omitted if not needed.
    static void TearDownTestCase() {
        delete shared_resource_;
        shared_resource_ = NULL;
    }

    // You can define per-test set-up and tear-down logic as usual.
    virtual void SetUp() { ... }
    virtual void TearDown() { ... }

    // Some expensive resource shared by all tests.
    static T* shared_resource_;
};

T* FooTest::shared_resource_ = NULL;

TEST_F(FooTest, Test1) {
    ... you can refer to shared_resource here ...
}
TEST_F(FooTest, Test2) {
    ... you can refer to shared_resource here ...
}

```

3.12. 전역 Set-Up 과 Tear-Down

테스트 레벨의 SetUp()과 테스트 케이스 레벨의 SetUpTestCase()처럼 테스트 프로그램 레벨에서도 같은 일을 할 수 있다. 방법은 다음과 같다.

1. ::testing::Environment 클래스를 상속받는 클래스를 만든다.

```

class Environment1 : public ::testing::Environment {
public:
    // Override this to define how to set up the environment.
    virtual void SetUp() {
    }

    // Override this to define how to tear down the environment.
    virtual void TearDown() {};
};

```

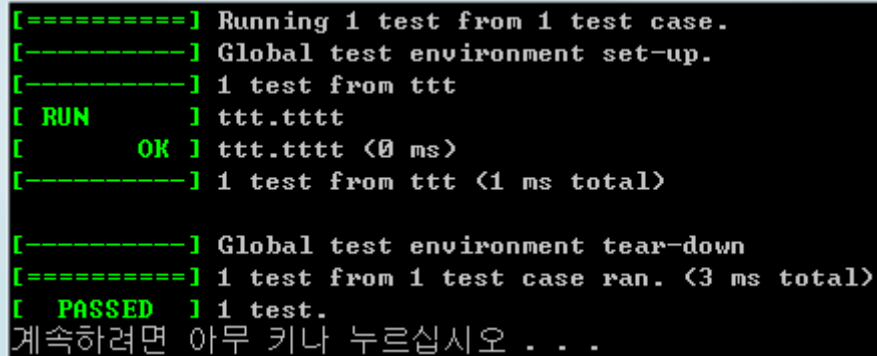
2. 환경 변수(Environment 객체)를 만들고 ::testing::AddGlobalTestEnvironment() 함수를 이용해 구글 테스트에 등록한다.

```
int main(int argc, char ** argv){
    Environment1 env;           // make Environment object
    AddGlobalTestEnvironment(&env); //register Environment object
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}
```

3. RUN_ALL_TEST()를 실행한다.

이와 같은 방법을 이용하면, 환경 변수의 SetUp()을 수행하고 테스트들을 진행한 뒤, 치명적인 실패가 없다면 환경 변수의 TearDown()을 수행한다.



```
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from ttd
[ RUN      ] ttd.tttt
[          OK ] ttd.tttt (0 ms)
[-----] 1 test from ttd (1 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (3 ms total)
[ PASSED   ] 1 test.
계속하려면 아무 키나 누르십시오 . . .
```

사진 10: 테스트의 수행 결과

테스트 전/후로 Global test environment 의 set-up 과 tear-down 을 수행한 것이 표시된다.

주의사항

1. 여러 개의 환경 변수를 등록할 수 있다. 등록된 순서대로 SetUp()이 실행되며, 역순으로 TearDown()이 실행된다.
2. 구글 테스트가 환경 변수의 소유권을 가지고 있으므로, 사용자가 삭제를 해서는 안된다.
3. RUN_ALL_TEST()전에 AddGlobalTestEnvironment() 를 이용해 객체를 등록해야 한다.

3.13. 값을 파라미터로 하는 테스트(Value Parameterized Test)

같은 종류의 테스트인데 매개변수만 다르게 하여 테스트를 해야 할 경우가 있다. 이를 위해 매개변수를 일일이 입력해가면서 단언문과 테스트를 작성하는 것은 힘든 일이다. 이런 경우를 위해 단 한번만 테스트를 작성한 뒤, 그 테스트에 다양한 값의 매개변수를 넘겨주는 식으로 테스트를 할 수 있다. 방법은 다음과 같다.

1. 텍스트 픽스처 클래스를 정의한다. 이때 보통의 테스트 픽스처¹⁹와 달리 `::testing::TestWithParam<Type>`을 상속받아야 한다. 만약 `Type`이 포인터라면 수명관리를 직접 해야 한다.
2. `TEST()`가 아닌 `TEST_P()`를 이용해 테스트를 작성한다. 여기서 `P`는 `Parameter` 혹은 `Pattern`을 의미한다.
3. `INSTANTIATE_TEST_CASE_P(instantiation_name, test_case_name, parameter_list)`를 이용해 원하는 매개변수 리스트와 함께 테스트 케이스를 인스턴스화 한다.

구글 테스트는 다음과 같은 방법으로 매개변수 리스트를 지정할 수 있도록 하고 있다.

문법	내용
<code>Range(begin, end[, step])</code>	{begin, begin+ step, begin+ step*2...end-step}. begin 값부터 시작해 step 만큼 증가시킨다. end 값이 포함되지 않는 것에 주의. step의 디폴트 값은 1이다
<code>Values(v1, v2, ..., vN)</code>	{v1, v2, ..., vN}
<code>ValuesIn(container)</code> , <code>ValuesIn(begin, end)</code>	배열이나 STL의 컨테이너, [begin, end) 범위의 이터레이터로부터 값을 얻어낸다. begin과 end에는 런타임에 값이 결정되는 표현식이 올 수 있다.
<code>Bool()</code>	{false, true}
<code>Combine(g1, g2, ..., gN)</code>	모든 콤비네이션(수학에서의 곱집합). <tr1/tuple> 헤더를 지원해야 한다. ²⁰

다음은 매개변수를 이용한 테스트를 실습해본 예제이다.

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

TEST_P(ParamTest, test1){
    EXPECT_TRUE(2 <= GetParam());
}

INSTANTIATE_TEST_CASE_P(ParamTestName1, ParamTest, ::testing::Values(1, 2, 3, 4, 5));

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

    return RUN_ALL_TESTS();
}
```

¹⁹ `::testing::Test`를 상속받는다.

²⁰ 만약 `Combine()` 사용시, 시스템에서 지원하는데, 구글 테스트에서 실행이 안된다면 `GTEST_HAS_TR1_TUPLE=1`로 재정의함으로써 사용할 수 있다.

Param 테스트 픽스처를 생성해서 매개변수로 {1, 2, 3, 4, 5}를 가지는 테스트를 수행했다. 결과는 다음과 같은 형식으로 나온다.

```
[=====] Running 5 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 5 tests from ParamTestName1/ParamTest
[ RUN      ] ParamTestName1/ParamTest.test1/0
c:\#users\#bluemoon\#documents\#visual studio 2013\#projects\#googletest\#googletest\#testfixture.cpp(9): error: Value of: 2 <= GetParam()
  Actual: false
Expected: true
[  FAILED  ] ParamTestName1/ParamTest.test1/0, where GetParam() = 1 <1 ms>
[ RUN      ] ParamTestName1/ParamTest.test1/1
[      OK  ] ParamTestName1/ParamTest.test1/1 <0 ms>
[ RUN      ] ParamTestName1/ParamTest.test1/2
[      OK  ] ParamTestName1/ParamTest.test1/2 <0 ms>
[ RUN      ] ParamTestName1/ParamTest.test1/3
[      OK  ] ParamTestName1/ParamTest.test1/3 <0 ms>
[ RUN      ] ParamTestName1/ParamTest.test1/4
[      OK  ] ParamTestName1/ParamTest.test1/4 <0 ms>
[-----] 5 tests from ParamTestName1/ParamTest <7 ms total>

[-----] Global test environment tear-down
[=====] 5 tests from 1 test case ran. <8 ms total>
[  PASSED  ] 4 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] ParamTestName1/ParamTest.test1/0, where GetParam() = 1

1 FAILED TEST
계속하려면 아무 키나 누르십시오 . . .
```

사진 11: 매개변수를 이용한 테스트 예제의 테스트 결과

수행 결과를 표시해 부분의 이름은 “*instantiation_name/test_case_name.test_name/순서*”의 형식으로 매겨지며, ‘순서’는 0 부터 자동으로 매겨진다.

주의사항

INSTITUTE_TEST_CASE_P()는 해당 테스트 케이스의 모든 테스트를 인스턴스화 한다. 이는 INSTANTIATE_TEST_CASE_P() 사용보다 전에 있던 테스트 케이스들도 인스턴스화 되며, 테스트가 GetParam()을 사용하지 않아도 역시 인스턴스화 된다.

예를 들어 다음과 같이 테스트를 작성했어도 테스트는 매개변수의 숫자만큼 실행된다.

```
TEST_P(ParamTest, test1){
    EXPECT_TRUE(1 == 1);
}

INSTITUTE_TEST_CASE_P(ParamTestName1, ParamTest, ::testing::Values(1, 2, 3, 4, 5));
```

```

[=====] Running 5 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 5 tests from ParamTestName1/ParamTest
[ RUN      ] ParamTestName1/ParamTest.test1/0
[       OK ] ParamTestName1/ParamTest.test1/0 (0 ms)
[ RUN      ] ParamTestName1/ParamTest.test1/1
[       OK ] ParamTestName1/ParamTest.test1/1 (0 ms)
[ RUN      ] ParamTestName1/ParamTest.test1/2
[       OK ] ParamTestName1/ParamTest.test1/2 (0 ms)
[ RUN      ] ParamTestName1/ParamTest.test1/3
[       OK ] ParamTestName1/ParamTest.test1/3 (0 ms)
[ RUN      ] ParamTestName1/ParamTest.test1/4
[       OK ] ParamTestName1/ParamTest.test1/4 (0 ms)
[-----] 5 tests from ParamTestName1/ParamTest (4 ms total)

[-----] Global test environment tear-down
[=====] 5 tests from 1 test case ran. (7 ms total)
[ PASSED   ] 5 tests.
계속하려면 아무 키나 누르십시오 . . .

```

사진 12: GetParam()을 사용하지 않는 테스트의 결과

매개변수가 필요 없는 테스트이지만 5번 실행된 것을 알 수 있다.

3.14. 타입 테스트 하기(Typed Test)

같은 형태의 테스트를 여러 타입에 대해 수행하려 한다. 예를 들면 어떤 계산을 하는 함수를 변수 종류가 `int` 때와 `double` 일 때 모두 테스트 하는 경우이다. 수행할 테스트와 타입의 수가 늘어나면 작성해야 할 테스트의 수가 [테스트의 수 * 타입의 개수]로 금방 커지게 된다. 이때 타입 테스트를 이용해 하나의 테스트를 이용해 여러 타입에 대해 테스트 해 볼 수 있다.

사용법

1. 템플릿을 사용하는 테스트 픽스처 클래스를 정의한다.

```

template <typename T>
class FooTest : public ::testing::Test {
public:
    ...
    typedef std::list<T> List;
    static T shared_;
    T value_;
};

```

2. 테스트 케이스와 타입들의 리스트를 연결한다. 리스트에 있는 타입들이 테스트에서 반복된다.

```

typedef ::testing::Types<char, int, unsigned int> MyTypes;
TYPED_TEST_CASE(FooTest, MyTypes);

```

주의사항 : 꼭 `typedef` 을 사용해야 한다.

3. TEST_F() 대신 TYPED_TEST()를 사용하여 테스트를 정의한다.

```
TYPED_TEST(FooTest, DoesBlah) {
    // Inside a test, refer to the special name TypeParam to get the type
    // parameter. Since we are inside a derived class template, C++ requires
    // us to visit the members of FooTest via 'this'.
    TypeParam n = this->value_;

    // To visit static members of the fixture, add the 'TestFixture::'
    // prefix.
    n += TestFixture::shared_;

    // To refer to typedefs in the fixture, add the 'typename TestFixture::'
    // prefix. The 'typename' is required to satisfy the compiler.
    typename TestFixture::List values;
    values.push_back(n);
    ...
}
```

TypeParam 형으로 만들어진 변수에 값을 담을 수 있으며, this 를 이용해 픽스처 개체의 값에 접근이 가능하다.

3.15. Private Code 테스트하기(Testing Private Code)

내부 구현을 바꾸어도, 인터페이스에 변화가 없다면 테스트 결과에 영향을 미쳐서는 안 된다. 따라서 보통 내부 구현 내용을 테스트 할 때에는 public 함수들을 이용해 테스트한다. 만약 내부 구현 내용을 테스트해야 한다면 우선 더 좋은 설계로 바뀌 그런 상황을 피할 수 있는지 생각해보고, 그래도 해야 한다면 다음의 두 가지 방법을 고려해볼 수 있다.

1. static function(static 멤버 함수와는 다르다!) 또는 이름 붙여지지 않은 네임스페이스(unnamed namespace)
2. private 또는 protected 클래스 멤버

3.15.1. Static Function

static 함수나 unnamed namespace 는 같은 번역 단위(translation unit)²¹에서만 볼 수 있다. 테스트될 .cc 파일들을 전부 #include 를 이용해 메인 .cc 파일에 포함시키므로써 테스트 할 수 있다.(절대 좋은 방식이 아니니까 실제 제품 코드에는 이렇게 하지 말 것!)

더 좋은 방법은 테스트 할 비공개 코드를 전부 foo::internal namespace 로 옮기고, 비공개 코드의 선언을

21. http://en.wikipedia.org/wiki/Translation_unit_%28programming%29

*-internal.h 파일에 넣는 것이다. (여기서 foo 는 프로젝트에서 사용하는 일반적인 namespace 이다) 제품 코드나 테스트 코드에서는 이 internal header 를 include 하게하고, 클라이언트에게는 허용하지 않는 것이다. 이런 방식을 통해 비공개 코드의 외부에 노출하지 않고 테스트 할 수 있다.

3.15.2. Private Class Member

Private 타입의 멤버들은 그 클래스나 프렌드 클래스에서만 접근 가능하다. 이를 이용해 테스트 픽스처를 프렌드 클래스로 선언하고, 테스트 픽스처에 접근자를 만들면 테스트에서는 그 접근자를 통해 private 멤버에 접근할 수 있다. 주의점은 테스트 픽스처를 프렌드 클래스로 선언하더라도, 테스트들도 프렌드가 되지는 않는다는 것이다.

Private 멤버를 테스트하는 다른 방법은 구현 클래스로 private 멤버들을 옮기고 그것을 *-internal.h 에서 선언하는 것이다. 테스트에서만 이 헤더를 include 하게 하면 된다. 이러한 방식을 Pimpl(Private Implementation) idiom 이라 한다.

또는 클래스 내부에 다음과 같은 방식으로 특정 클래스를 프렌드로 선언할 수도 있다.

```
FRIEND_TEST(TestCaseName, TestName);
```

다음의 방식으로 사용한다.

```
// Defines FRIEND_TEST.
class Foo {
    ...
private:
    FRIEND_TEST(FooTest, BarReturnsZeroOnNull);
    int Bar(void* x);
};

// foo_test.cc
...
TEST(FooTest, BarReturnsZeroOnNull) {
    Foo foo;
    EXPECT_EQ(0, foo.Bar(NULL));
    // Uses Foo's private member Bar().
}
```

주의사항 : 테스트할 private 멤버가 있는 클래스와 테스트 픽스처, 테스트는 같은 namespace 에 있어야 한다.

3.16. 현재 테스트 이름 얻기

현재 진행 중인 테스트의 이름을 얻어야 할 때가 있다. `::testing::TestInfo` 클래스가 이러한 정보를 가지고 있다.

```
// Gets information about the currently running test.  
// Do NOT delete the returned object - it's managed by the UnitTest class.  
const ::testing::TestInfo* const test_info= ::testing::UnitTest::GetInstance()-  
>current_test_info();
```

이렇게 TestInfo 객체를 만든 뒤 test_info->test_case_name()와 test_info->name()를 이용해 테스트 케이스 이름과 테스트 이름을 얻을 수 있다.

3.17. 고급 옵션으로 테스트 실행하기(Running Test Program : Advanced Option)

구글 테스트 프로그램은 일반적으로 실행가능하다. 한번 빌드를 끝내면 그냥 실행을 할 수도 있고, 환경 변수나 커맨드 라인 플래그와 같이 실행할 수도 있다. 플래그가 제대로 동작하게 하기 위해선 RUN_ALL_TESTS() 전에 ::testing::InitGoogleTest()를 호출해야 한다. 플래그들과 그것들의 사용법을 보려면 --help 플래그(또는 -h, -?, /?)와 함께 프로그램을 실행하면 된다. 이 기능은 1.3.0 버전부터 지원한다.

```

C:\Users\Jun>GoogleTest -h
This program contains tests written using Google Test. You can use the
following command line flags to control its behavior:

Test Selection:
  --gtest_list_tests
    List the names of all tests instead of running them. The name of
    TEST<Foo, Bar> is "Foo.Bar".
  --gtest_filter=POSTIVE_PATTERNS[-NEGATIVE_PATTERNS]
    Run only the tests whose name matches one of the positive patterns but
    none of the negative patterns. '?' matches any single character; '*'
    matches any substring; ':' separates two patterns.
  --gtest_also_run_disabled_tests
    Run all disabled tests too.

Test Execution:
  --gtest_repeat=[COUNT]
    Run the tests repeatedly; use a negative count to repeat forever.
  --gtest_shuffle
    Randomize tests' orders on every iteration.
  --gtest_random_seed=[NUMBER]
    Random number seed to use for shuffling test orders (between 1 and
    99999, or 0 to use a seed based on the current time).

Test Output:
  --gtest_color=<yes|no|auto>
    Enable/disable colored output. The default is auto.
  --gtest_print_time=0
    Don't print the elapsed time of each test.
  --gtest_output=xml[:DIRECTORY_PATH#[:FILE_PATH]]
    Generate an XML report in the given directory or with the given file
    name. FILE_PATH defaults to test_details.xml.

Assertion Behavior:
  --gtest_break_on_failure
    Turn assertion failures into debugger break-points.
  --gtest_throw_on_failure
    Turn assertion failures into C++ exceptions.
  --gtest_catch_exceptions=0
    Do not report exceptions as test failures. Instead, allow them
    to crash the program or throw a pop-up (on Windows).

Except for --gtest_list_tests, you can alternatively set the corresponding
environment variable of a flag (all letters in upper-case). For example, to
disable colored text output, you can either specify --gtest_color=no or set
the GTEST_COLOR environment variable to no.

For more information, please read the Google Test documentation at
http://code.google.com/p/googletest/. If you find a bug in Google Test
(not one in your own code or tests), please report it to
googletestframework@googlegroups.com.

```

만약 같은 옵션이 환경 변수와 플래그에 의해 동시에 설정되면, 플래그가 우선이 된다.

다음과 같은 방식으로 플래그를 설정할 수도 있다.

```
int main(int argc, char** argv) {
```



```
// Disables elapsed time by default.
::testing::GTEST_FLAG(print_time) = false;

// This allows the user to override the flag on the command line.
::testing::InitGoogleTest(&argc, argv);

return RUN_ALL_TESTS();
}
```

주의사항 : 플래그 설정은 `::testing::InitGoogleTest()` 보다 먼저 나와야 한다.

3.17.1. 테스트의 이름들 나열하기

`--gtest_list_tests` 옵션과 함께 프로그램을 실행하면 다음과 같은 형태로 테스트 목록을 출력한다.

```
TestCase1.
  TestName1
  TestName2
TestCase2.
  TestName
```

이 플래그를 실행하면 나열된 테스트들은 실제로 실행되지 않는다. 이에 해당하는 환경 변수는 없다.

3.17.2. 테스트의 일부만 실행하기

`GTEST_FILTER` 환경 변수나 `--gtest_filter` 플래그로 필터링 문자열을 설정하면, 테스트 케이스나 테스트 중 이름이 필터와 매치된 테스트들만 수행한다.

필터는 ‘.’와 ‘-’를 이용해 설정하며 ? 와일드 카드와(한글자 대체), * 와일드 카드(문자열 대체)를 지원한다. ‘.’와 와일드 카드들을 이용해 positive patterns 를 지정하며, ‘-’와 ‘:’, 와일드 카드들들을 이용해 negative patterns 를 지정한다. 즉, ‘.’를 이용해 구분한 필터링 문자열을 포함한 테스트 들 중에 ‘-’ 다음에 나온 필터링 문자열을 포함하지 않는 테스트들만 실행된다.

예를 들어보자.

<code>./foo_test</code>	플래그 없음. 모든 테스트를 실행한다.
<code>./foo_test --gtest_filter=*</code>	플래그로 * 사용. 모든 테스트를 실행한다.
<code>./foo_test --gtest_filter=FooTest.*</code>	FooTest 테스트 케이스에 있는 테스트들만 실행한다.

<code>./foo_test --gtest_filter=*Null*:~*Constructor*</code>	“Null”이나 “Constructor”를 포함한 테스트들만 실행한다.
<code>./foo_test --gtest_filter=~*DeathTest.*</code>	“DeathTest”로 끝나는 테스트 케이스들의 테스트는 실행하지 않는다.
<code>./foo_test --gtest_filter=FooTest.*~FooTest.Bar</code>	FooTest 테스트 케이스의 테스트들을 실행하되, FooTest.Bar 테스트는 실행하지 않는다.

3.17.3. 임시로 테스트 사용하지 않기(Temporarily Disabling Tests)

만약 바로 고칠 수 없는 부서진 테스트가 있다면 DISABLED_ 접두사를 이름 앞에 붙임으로써 테스트를 불활성화 할 수 있다. 이 테스트는 실행에서 제외될 것이다. 이 방법은 실행은 되지 않지만 컴파일은 되므로 코드를 주석처리 하거나 `#if 0` 를 사용하는 방식보다는 낫다.

예를 들어 다음의 테스트들은 컴파일은 되지만 실행은 되지 않는다.

```
// Tests that Foo does Abc.
TEST(FooTest, DISABLED_DoesAbc) { ... }

class DISABLED_BarTest : public ::testing::Test { ... };

// Tests that Bar does Xyz.
TEST_F(DISABLED_BarTest, DoesXyz) { ... }
```

각각 테스트를 불활성화 한 예제와 테스트 케이스를 불활성화 한 예제이다.

주의사항 : 이 기능은 임시적으로만 사용해야 한다. 이 사실을 기억할 수 있도록 구글 테스트는 불활성화 된 테스트가 있다면 경고를 띄운다.

3.17.4. 임시로 불활성화된 테스트 실행하기(Temporarily Enabling Disabled Tests)

`--gtest_also_run_disabled_tests` 플래그나 `GTEST_ALSO_RUN_DISABLED_TESTS` 환경 변수를 0 이 아닌 값으로 설정함으로써 불활성화된 테스트도 실행하도록 만들 수 있다.

3.17.5. 테스트 반복하기(Repeating the Tests)

확률적으로 실패하는 테스트의 경우엔 여러 차례 돌려야만 제대로 된 원인을 찾을 수 있다. 이런 경우 다음의 플래그들을 이용해 반복할 수 있다.

플래그	내용
--gtest_repeat=1000	테스트를 1000 번 반복하고 실패가 있어도 멈추지 않는다.
--gtest_repeat=-1	음수를 넣으면 무한히 반복한다.
--gtest_repeat=1000 --gtest_break_on_failure	테스트를 1000 번 반복하고 첫 번째 실패에서 정지한다. 디버그 모드에서 실행할 경우 실패와 동시에 디버거를 띄워준다.
--gtest_repeat=1000 --gtest_filter=FooBar	필터와 매치하는 테스트(이 경우 FooBar)를 1000 번 반복한다.

주의사항 : 만약 AddGlobalTestEnvironment()를 이용해 추가한 global set-up/tear-down 코드가 있으면 테스트를 반복할 때마다 이것들도 같이 반복된다.

또한 GTEST_REPEAT 환경변수를 이용해 반복 횟수를 지정할 수도 있다.

3.17.6. 테스트 순서 섞기(Shuffling the Tests)

--gtest_shuffle 플래그나 GTEST_SHUFFLE 환경 변수를 이용해 테스트가 실행되는 순서가 랜덤이 되도록 할 수 있다. 만약 테스트 간에 의존성이 있다면 이를 통해 찾을 수 있다.

기본적으로 구글 테스트는 현재 시각을 기준으로 랜덤 시드를 생성한다. 만약 임의로 시드를 지정하고 싶다면 --gtest_random_seed=SEED 나 GTEST_RANDOM_SEED 환경변수를 통해 시드를 지정할 수 있다. SEED에 들어갈 수 있는 값은 0에서 99999까지의 정수이다. 만약 SEED에 0을 넣으면 현재 시각을 기준으로 시드를 계산하는 디폴트 시드를 사용한다.

만약 --gtest_repeat=N 플래그와 함께 사용하면 반복할 때마다 시드를 재생성해서 테스트한다.

4. 소프트웨어 테스트

이 챕터는 테스트 주도 개발이 아닌, 소프트웨어 테스트 자체에 대한 내용으로, 위에서 사용하는 테스트 주도 개발에서 필요한 테스트와는 약간 차이가 있을 수 있다. 테스트 주도 개발에 사용되는 테스트가 개발을 위한 테스트라면, 이 챕터의 테스트들은 말 그대로 테스트를 위한 테스트이기 때문이다. 다른 목적을 가지고 있으므로 다른 경향성을 지니는 것은 당연할지도 모른다. 하지만 ‘문제가 발생하지 않게 한다’라는 목적을 공유하는 이상 같이 공부하면 좋을 것이라 생각한다.

소프트웨어 테스트는 의도한 것은 의도한 대로 일어나고, 의도하지 않은 것은 일어나지 않는 것을 확인하기 위한 과정이다. 하지만 이러한 목표를 성공적으로 달성하는 것은 쉽지 않고, 그에 따라 여러가지 방법이나 도구들이 만들어졌다. 이 챕터에서는 소프트웨어 테스트를 성공적으로 하기 위해 알아야 할 것들을 소개해보고자 한다.

4.1. 소프트웨어 테스트의 원칙

소프트웨어 테스트에서 심리학적인 면은 중요한 고려사항이다. 이러한 점을 중심으로 테스트 기준 10 가지를 뽑아보았다.

1. 테스트 케이스에는 예상 출력이나 결과에 대한 정의가 필요하다.
2. 프로그래머는 자신의 프로그램을 테스트하려 해서는 안된다.
3. 프래그래밍 조직은 자신의 프로그램을 테스트 해서는 안된다.
4. 모든 테스트 과정에는 각 테스트의 결과에 대한 검사가 포함되어야 한다.
5. 테스트 케이스는 유효하고 예상 가능한 입력 조건만이 아니라 무효하고 예상치 못한 조건에 대해서도 작성되어야 한다.
6. 프로그램이 해야 할 일을 수행하는지 확인하는 것이 50%, 하지 말아야 하는 일을 하지 않는지 확인하는 것이 50%이다.
7. 폐기시킬 프로그램이 아니라면, 폐기할 테스트 케이스를 사용해서는 안된다.
8. 아무런 오류가 발생하지 않을 것이라는 가정하에 테스트를 계획해서는 안된다.
9. 프로그램의 특정 부분에서의 오류 존재 가능성은 그 부분에서 발견된 오류의 수에 비례한다.
10. 테스트는 매우 창조적이고 지적인 업무이다.

각 원칙에 대해 좀 더 알아보자.

1. 테스트 케이스에는 예상 출력이나 결과에 대한 정의가 필요하다.

: 예상 결과를 테스트 전에 정하지 않는다면, 잘못된 결과가 나와도 좋은 쪽으로 해석할 수 있다. 이런 문제를 방지하기 위해 테스트 케이스는 다음의 두 가지 요소를 갖추어야 한다.

- 입력 데이터에 대한 기술
- 입력 데이터에 대한 프로그램의 올바른 출력에 대한 기술

2. 프로그래머는 자신의 프로그램을 테스트하려 해서는 안된다.
3. 프래그래밍 조직은 자신의 프로그램을 테스트 해서는 안된다.

: 자신이 작업한 것들의 오류를 찾는 것은 심리적으로 거부감을 느끼게 한다. 또한, 프로그래머가 잘못된 이해한 프로그램 명세를 토대로 프로그램을 짰을 경우, 올바른 테스트가 이루어지지 않는다.

4. 모든 테스트 과정에는 각 테스트의 결과에 대한 검사가 포함되어야 한다.

: $1+1=3$ 이라고 풀었는데, 답지에 3 이 정답이라고 적혀있다면 그 순간 동그라미는 쳐지겠지만, 나중에는 결국 틀렸다는 것을 알게 될 것이다. 이러한 일을 막기 위해서 테스트 결과에 대한 확인은 필요하다.

5. 테스트 케이스는 유효하고 예상 가능한 입력 조건만이 아니라 무효하고 예상치 못한 조건에 대해서도 작성되어야 한다.

: 보통 테스트 케이스를 작성할 때 유효한 입력에 대해 똑바로 일을 하는지에 집중을 하게 된다. 하지만 보통 무효한 경우에 대한 테스트 케이스가 훨씬 많다.

6. 프로그램이 해야 할 일을 수행하는지 확인하는 것이 50%, 하지 말아야 하는 일을 하지 않는지 확인하는 것이 50%이다.

: 값을 하나만 바꿔야 하는데 두 개를 바꾼다던가, 출력만 해야 하는데, 출력과 함께 삭제를 한다거나 하는 일이 있으면 안된다.

7. 폐기시킬 프로그램이 아니라면, 폐기할 테스트 케이스를 사용해서는 안된다.

: 테스트 케이스를 테스트하고 바로 지워버리는 것은 소모적인 일이다. 나중에 테스트를 할 일이 생길 경우 다시 만들어야 하기 때문이다.

8. 아무런 오류가 발생하지 않을 것이라는 가정하에 테스트를 계획해서는 안된다.

: 테스트는 프로그램이 잘 작동하는지를 보여주는 작업이 아니다. 프로그램의 오류를 찾기 위한 과정이다.

9. 프로그램의 특정 부분에서의 오류 존재 가능성은 그 부분에서 발견된 오류의 수에 비례한다.

: 오류는 밀집하여 발생하는 경향이 있다. 따라서 특정 부분에서 많은 오류가 발생한다면, 그 부분에 테스트 역량을 집중하는 것이 좋은 전략일 수 있다.

10. 테스트는 매우 창조적이고 지적인 업무이다.

: 완벽한 테스트는 불가능하다. 즉, 어떤 일상적인 절차로는 부족하다는 얘기이다. 각각의 프로그램에 대해 적절한 테스트 케이스를 만들기 위해선 창의력이 있어야 한다.

4.2. 들어가기 전 실력 확인 문제

다음의 프로그램이 정확히 동작한다는 것을 확인하기 위한 테스트 케이스를 작성하라.

프로그램의 목적 : 세 정수를 입력받는다. 세 정수는 삼각형의 변의 길이를 나타낸다. 입력받은 값을 기준으로 해당 삼각형이 정삼각형인지, 이등변 삼각형인지, 부등변 삼각형인지 판단하여 결과를 출력한다.

쉬운 문제이다. 잠깐 답을 생각해보자. 대표적인 테스트 케이스들은 다음과 같다.

1. 유효한 부등변 삼각형을 나타내는 테스트 케이스
2. 유효한 정삼각형을 나타내는 테스트 케이스
3. 유효한 이등변 삼각형을 나타내는 테스트 케이스
4. 두 변이 같으며, 같은 삼각형을 나타내는 3 개의 테스트 케이스. ex) 3,3,4 ; 4,3,3 ; 3,4,3
5. 한 변의 값이 0으로 이루어진 테스트 케이스
6. 한 변의 길이가 음수로 이루어진 테스트 케이스
7. 3 개의 값이 0보다 크고 두 변의 합이 나머지 한 변과 같은 테스트 케이스
8. 7 변의 조건을 만족하는 값들의 순열로 이루어진 3 개의 테스트 케이스. ex) 1,2,3; 2,3,1; 3,2,1
9. 두 변의 합이 나머지 변의 값보다 작은 테스트 케이스
10. 9 변의 조건을 만족하는 값들의 순열로 이루어진 3 개의 테스트 케이스. ex) 1,2,4 ; 2,4,1 ; 4,2,1
11. 적어도 하나는 정수가 아닌 값을 가진 테스트 케이스
12. 모든 변이 0으로 되어있는 테스트 케이스

생각보다 많은 테스트 케이스들이 나온다. 이러한 테스트 케이스에서 문제가 생기지 않는다고 문제가 발생하지 않는다는 보장은 없지만, 다른 프로그램들에서 발생했던 오류들을 모아본 것이다. 간단한 프로그램에 대한 테스트 케이스들도 이렇게 많다면 복잡한 프로그램들은 더 많은 테스트 케이스들을 가지고 있을 것이고, 필요한 테스트 케이스들을 빼먹지 않고 작성하는 것도 상당히 어려울 것이다. 이러한 문제점들을 해결할 수 있는 방법들을 알아보자.

4.3. 테스트 케이스 설계

테스팅은 크게 화이트박스 테스팅과 블랙박스 테스팅으로 나눌 수 있다. 화이트 박스 테스팅은 구현 방법에 대해 알고 있는 상태에서 수행하는 테스팅이고, 블랙 박스 테스팅은 구현 방법과는 상관없이, 해당 프로그램의 명세를 가지고 수행하는 테스팅이다. 여기서 명세는 해당 프로그램이 어떤 입력값을 가지고 어떤 동작을 통해 어떤 결과값을 알려주는지를 의미한다. 우선 화이트 박스 테스팅에 대해 알아보자.

4.3.1. 화이트 박스 테스트(White-Box Testing)

화이트 박스 테스트는 테스트 케이스가 프로그램의 코드를 실행하고 포함하는 정도에 초점을 맞춘다.

4.3.1.1. 문장 커버리지(Statement Coverage)

프로그램 내 모든 문장을 한 번 이상 실행하는 것도 유용한 목표가 될 수 있다.

```
void foo(int a, int b, int c){  
    if (a > 1 && b == 0){  
        c = c / a;  
    }  
  
    if (a == 2 || c > 1){  
        c++;  
    }  
}
```

위와 같은 함수를 생각해보자. A=2, b=0, c=3 으로 설정하면 모든 문장은 실행된다. 하지만 이러한 입력으로는

‘첫 번째 if에 && 대신 || 가 들어간 경우’나 ‘두 번째 if에 c>0 으로 되어 있는 경우’에 그 문제를 찾아내지 못한다. 또한 c가 변하지 않는 경우도 포함되어 있으므로 위의 테스트 케이스로는 c에 대한 테스트도 제대로 되지 않는다.

따라서 문장 커버리지만 가지고는 충분한 테스트가 되었다고 할 수 없다.

4.3.1.2. 결정 커버리지(Decision Coverage)

분기 커버리지(Branch Coverage) 라고도 불리는 이 기준은 모든 결정문²²에 대해 적어도 한 번은 참 또는 거짓을 갖도록 테스트 케이스를 작성하는 방법이다. 일반적으로 결정 커버리지는 문장 커버리지를 만족한다. 다만 예외가 있다.

- 결정문이 없는 경우
- 다중 시작점을 가진 프로그램
- 서브루틴

위의 예제를 다시 이용한다면 [a=3, b=0, c=3]인 테스트 케이스와 [a=2, b=1, c=1]인 테스트 케이스가 결정 커버리지를 만족하는 테스트 케이스이다.

4.3.1.3. 조건 커버리지(Condition Coverage)

결정문 내의 개별적인 조건들에 대해 한 번은 참 또는 거짓을 갖도록 테스트 케이스를 작성하는 방법이다. 결정 커버리지는 결정문 전체(if문이 참인지 거짓인지)를 기준으로 한다면, 조건 커버리지는 결정문을 이루는 각각의 조건식(if문 안에 있는 각각의 논리연산)을 기준으로 한다.

22. if 문, switch 문 등

예제에서는 $a > 1$, $b == 0$, $a == 2$, $c > 1$ 의 4 가지 조건이 있다. 따라서 $a > 1$, $a <= 1$, $b == 0$, $b != 0$, $a == 2$, $a != 2$, $c > 1$, $c <= 1$ 의 8 가지 경우에 대한 테스트 케이스가 필요하다. 예) $[a=1, b=0, c=3]$ 와 $[a=2, b=1, c=1]$

결정 커버리지는 전체 조건에 대해 따지고, 조건 커버리지는 개별적인 조건에 대해 따지니 조건 커버리지가 결정 커버리지를 포함할 수 있을 것으로 생각된다. 하지만 실제로는 그렇지 않다.

바로 위에서 예로 든 테스트 케이스를 보면, 4 가지 조건의 참/거짓을 모두 가지고 있다. 하지만 테스트 케이스가 수행되는 과정을 추적해보면, 두 테스트 케이스 모두 첫 번째 if문에서 거짓으로, 두 번째 if문에서는 참으로 통과한다. 즉, 첫 번째 if문의 참과 두 번째 if문의 거짓은 테스트 되지 않는 것이다. 이러한 문제를 해결하기 위한 기준이 다음의 결정/조건 커버리지이다.

4.3.1.4. 결정/조건 커버리지(Decision/Condition Coverage)

이름에서 알 수 있듯이 결정 커버리지와 조건 커버리지의 합집합입니다. 모든 결정문이 참/거짓이 되는 경우와 각각의 조건문이 참/거짓이 되는 경우를 포함한 것이다.

$[a=2, b=0, c=4]$, $[a=1, b=1, c=1]$ 가 그 예가 될 수 있다.

4.3.1.5. 다중 조건 커버리지(Multiple-condition Coverage)

&&나 ||의 경우 같이 쓰이는 다른 조건의 평가를 가리거나 방해한다. 예를 들어, and로 이어진 두 조건 중 첫 번째 조건이 거짓인 경우 두 번째 조건의 결과는 무시된다. 이런 결과를 막기 위해 다중 조건 커버리지에서는 모든 조건에 대한 모든 조합을 테스트하는 것을 기준으로 삼는다.

예제의 경우

$a > 1, b == 0$	$a == 2, c > 1$
$a > 1, b != 0$	$a == 2, c <= 1$
$a <= 1, b == 0$	$a != 2, c > 1$
$a <= 1, b != 0$	$a != 2, c <= 1$

의 8 가지 경우를 조합해 테스트 케이스를 만들면 이게 바로 다중 조건 커버리지를 만족하는 테스트이다. 그 예로는 $[a=2, b=0, c=4]$, $[a=2, b=1, c=1]$, $[a=1, b=0, c=2]$, $[a=1, b=1, c=1]$ 의 4 가지 입력값 조합이 있다.

4.3.1.6. 각각의 커버리지 간의 포함관계

구문 커버리지 < 조건 커버리지, 결정 커버리지 < 조건/결정 커버리지 < 다중 조건 커버리지
--

의 관계를 가지며, 조건 커버리지와 결정 커버리지는 교집합을 갖는 서로 다른 집합이다.

4.3.2. 블랙박스 테스트(Black-Box Testing)

블랙박스는 구현이 아닌 명세를 가지고 수행을 하기 때문에 결정문이 아닌 입력값에 주 초점을 맞추게 된다.

4.3.2.1. 동등 분할(Equivalence Partitioning)

모든 입력값에 대한 테스트는 현실적으로 불가능하다. 따라서 오류를 찾을 가능성이 높은 입력값의 집합을 선택해야 한다. 이런 기준으로 선택된 잘 구성된 테스트 케이스는 '테스팅 목표를 만족하며, 다른 테스트 케이스를 포함한다.'라는 조건을 만족해야 한다. 즉, 어느 테스트 케이스 집합을 대표할 수 있는 테스트 케이스를 뽑아야 한다는 것이다.

예를 들어 동등하다고 판단된 테스트 케이스의 집합의 한 테스트 케이스가 어느 오류를 발견했다면, 같은 집합의 다른 테스트 케이스도 그 오류를 찾을 수 있어야 한다. 이런 식으로 같은 결과를 낼 수 있는 테스트 케이스의 집합을 동등 클래스라고 한다.
동등 클래스의 예는 다음과 같다.

- 입력 조건이 “정수 1~999 까지” 라면, 하나의 유효한 동등 클래스($1 < \text{입력값} < 999$)와 두 개의 무효한 동등 클래스($\text{입력값} < 1$, $\text{입력값} > 999$)가 있다.

- 입력 조건이 “문자”라면, 하나의 유효한 동등 클래스(입력값이 문자)와 하나의 무효한 동등 클래스(입력값이 문자가 아님)가 있다.

테스트 케이스 식별 방법

다음의 과정을 통해 테스트 케이스를 식별한다.

1. 동등 클래스를 나눈다.
2. 테스트 케이스가 모든 유효한 동등 클래스를 포함할 때까지 테스트 케이스를 작성한다.
3. 테스트 케이스가 모든 무효한 동등 클래스에 대한 테스트 케이스를 작성한다.

예를 들어보자.

입력값으로 "2 개 이상의 값을 받으며, 각각의 값은 정수나 실수이고, 각 값의 크기는 5 보다는 커야하고 10 보다는 작아야 한다" 는 조건이 있다고 가정하자. 그 경우 다음의 동등 클래스로 나눌 수 있다.

조건	유효한 동등 클래스	무효한 동등 클래스
입력받는 값의 개수	2 개(1)	1 개(2)
값의 타입	정수(3), 실수(4)	문자(5)
값의 최소값 조건	$x > 6$ (6)	$x \leq 5$ (7)
값의 최대값 조건	$x < 10$ (8)	$x \geq 10$ (9)

※ 각각의 동등 클래스 옆의 괄호안의 숫자는 이후 설명을 위한 넘버링이다.

이제 유효한 테스트 케이스를 작성한다.

입력값 (7,8)의 경우 (1), (3), (6), (8)를 만족시킨다.
입력값 (7.2, 8.3)은 나머지 유효한 동등클래스인 (4)를 만족시킨다.

그 다음 무효한 테스트 케이스를 작성한다. 이 예제의 경우 and 로 조건이 묶여 있으므로 각각의 무효한 동등 클래스에 대한 테스트 케이스를 따로따로 작성해야 원하는 결과를 얻을 수 있다.

즉, 이 문제의 경우 유효한 동등 클래스에 대한 테스트 케이스 2 개와 무효한 동등 클래스에 대한 테스트 케이스 4 개를 합한 6 개의 테스트 케이스가 있으면 충분한 테스트를 할 수 있다.

4.3.2.2. 경계값 분석(Boundary Value Analysis)

경험적으로 경계값에 대한 테스트 케이스가 그렇지 않은 테스트 케이스보다 오류 발견 성공률이 높다. 따

라서 경계값에 초점을 맞춰 테스트를 하는 것도 좋은 전략이다. 예를 들면 다음과 같은 테스트이다.

- 입력값이 1~255 까지라면, 입력값이 0, 1, 255, 256 인 테스트 케이스를 작성한다.
- 입력 갯수가 1~5 개라면, 0 개, 1 개, 5 개, 6 개를 입력했을 때의 테스트 케이스를 작성한다.
- 1 의 조건을 출력에 적용하여, 결과값이 최소 10 에서 최대 100 까지만 되어야 한다면, 10 미만 혹은 100 초과가 될 수 있는 입력값으로 테스트 케이스를 작성한다.
- 2 의 조건을 출력에 적용하여, 출력 결과의 개수가 3 개까지만 표시되어야 한다면, 0 개나 5 개가 출력될 수 있는 입력으로 테스트 케이스를 작성한다.
- 프로그램의 입/출력이 선형 리스트와 같은 경우에 맨 처음과 맨 끝에 주의를 기울인다.

5. 결론

테스트 주도 개발은 코드 작성->테스트가 아닌, 테스트 작성->코드 작성이라는 발상의 전환에서 태어난 개발 방법이다. 말로 하면 간단한 순서 바꿈이지만 그로 인해 고민의 우선 순위와 선후 관계가 바뀌고, 그 결과 자유롭지만 막판에 고생하는 프로그래밍에서 초반에 좀 더 고생하고 나중에 그만큼 덜 고생하는 프로그래밍이 되었다고 생각한다.

테스트를 먼저 작성하는 방식은 각 메소드들의 인터페이스에 대해 먼저 고민하게 해보고, 이러한 방식은 설계 우선 프로그래밍 방법론과 닮았다고 생각한다. 실제로 테스트 주도 방식으로 프로그램을 만들어보면서 우선 종이에 설계를 먼저 하고 나중에 코드를 구현하는 방식으로 하게 되는 경우도 있었다. 하지만 설계 변경에 부담이 없다는 점, 그리고 구현 결과를 바로바로 확인할 수 있다는 점에서 테스트 주도 방식이 좀 더 유연하고 안정적인 방식이라고 느꼈다.

테스트 주도 개발 방식으로 프로그램을 짜야겠다고 마음먹고 시작을 해도, 예전 습관 때문에 테스트 하나 만들고 구현코드 10 개 만드는 식으로 프로그램을 짰 경우도 많았다. 아마 이러한 습관을 바로잡기 위한 비용이 테스트 주도 개발을 사람들이 널리 이용하지 않는 이유일 것이다. 하지만 테스트 주도 개발 방법은 분명 프로그램 작성에 대한 패러다임을 바꿀 수 있는 경험이 될 것이라 생각한다.

마지막으로 테스트 주도 개발이 완벽하고 무조건 해야할 방법은 아니다 라는 말을 하고 싶다. 간단한 프로그램의 경우엔 일반적인 구현 방법을 사용하는 것이 프로그램을 더 빨리 구현해 낼 수 있기도 하며, 꼭 이런 점이 아니더라도 테스트 주도 개발에 반대하는 의견들도 나오고 있다. 다음의 링크는 테스트 주도 개발에 반대하는 의견과 그에 대한 켄트 벡(테스트 주도 개발을 처음 주장한 사람)의 동의하는 의견이다.

DHH : <http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html>

윗 글의 번역 : <http://visangwook.tumblr.com/post/83725422949/tdd-is-dead-long-live-testing>

이에 대한 켄트 벡의 의견 : <https://www.facebook.com/notes/kent-beck/rip-tdd/750840194948847>

6. 참고 문헌

1. 켄트 벡, 테스트 주도 개발(Test-Driven Development : By Example), 김창준, 강규영 역, 2014, 인사이트
2. 채수원, 테스트 주도 개발: 고품질 채속개발을 위한 TDD 실천법과 도구, 2013, 한빛미디어
3. 구글 공식 문서, https://code.google.com/p/googletest/wiki/V1_7_Documentation
4. 블로그, <http://surpreem.com/archives/273>