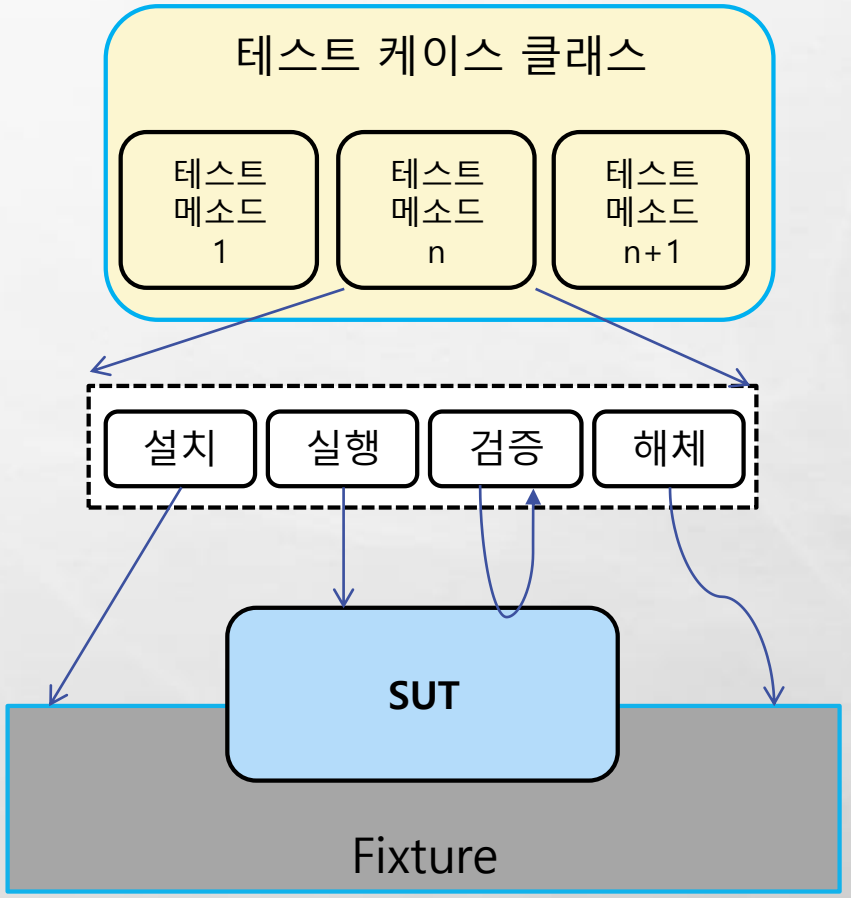
# xUNIT 테스트 패턴

# 목차

# Test 구성



- 설치(Setup)
  - 원하는 동작을 위해 테스트 Fixture를 설치하거나  Test Double을 넣는 작업
- 실행(Exercise)
  - SUT(System Under Test)에  테스트를 실행
- 검증(Verify)
  - 기대하는 결과가 나왔는지 확인
- 해체(TearDown)
  - Test Fixture 를 해체. 환경을 테스트 시작 전 상태로 되돌림




- Fixture : 테스트를 하기 위해 필요한 환경
  Test Double : SUT가 의존하는 Component 의 대체
- SUT : 테스트 하고자 하는 대상

# Fixture 정의



- xUnit 계열 framework 에서의 fixture
  - SUT 를 실행하기 위해 필요한 모든 것
  - Fixture 설치 = fixture 를 설치하기 위해 호출하는 테스트 로직

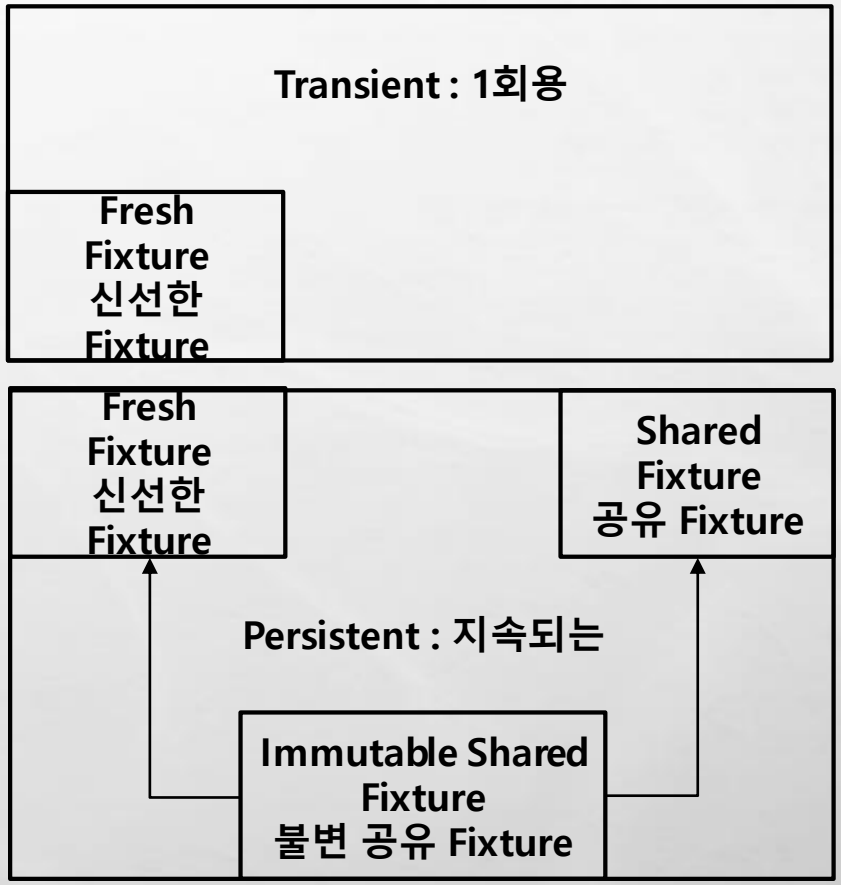
- Framework 별 Fixture 비교
  - JUnit : Fixture 와 Fixture를 생성하는 테스트케이스 클래스 구분
  - NUnit : 테스트케이스 클래스 객체를 Fixture 로 간주(.NET)
  - RSpec : 테스트 관련 클래스의 테스트 선조건(Behavior Driven Development for Ruby)
  - Fit : 데이터 주도 테스트 인터프리터 중에서 고객이 작성하는 부분(Framework for Integrated Test)

- xUnit patterns.com
  - Fixture : 모든 테스트 선조건
  - 테스트케이스 클래스 : 테스트 Fixture 설치에 필요한 모든 코드와 테스트 메소드가 들어있는 클래스

# Fixture Setup



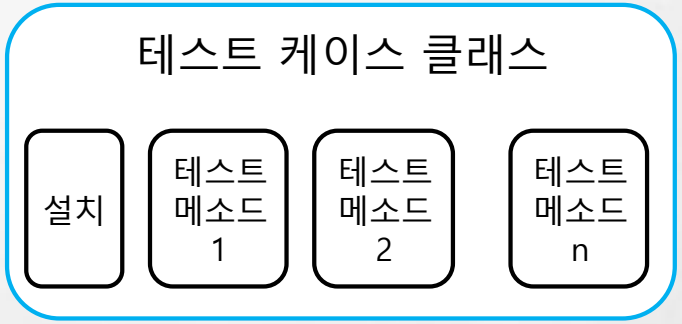
- Fixture 크기에 따른 구분
  - Minimal Fixture
    - 테스트에 필요한 최소한의 환경
  - Standard Fixture
    - 모든 테스트에 필요한 표준 환경
- Fixture 설치 방법에 따른 구분
  - Fresh Fixture
    - 필요시 설치하는 Fixture
  - Shared Fixture
    - 여러 테스트가 공유하는 Fixture
- Fixture 지속성에 따른 구분
  - Transient
  - Persistent



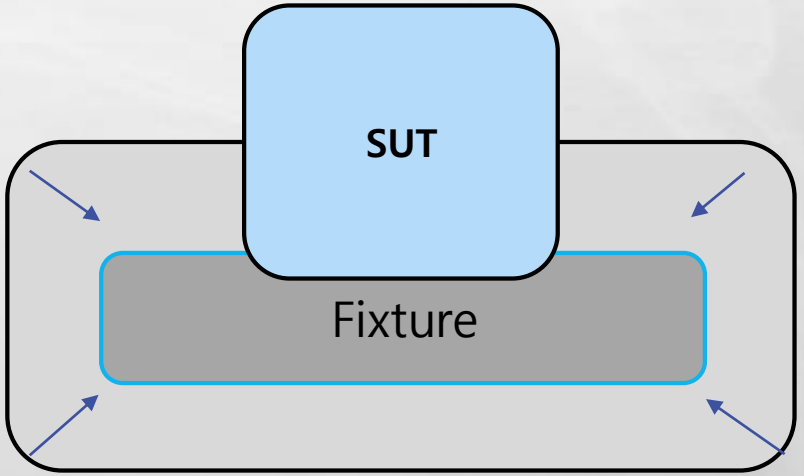
# Setup - Minimal Fixture



- Test 를 실행에 필요한 최소한의 component 를 설치




- 문서로서의 테스트
  - 작고 단순한 Fixture => 이해하기 쉬운 테스트
- 빠른 테스트
  - 작고 단순한 Fixture => 빠른 테스트

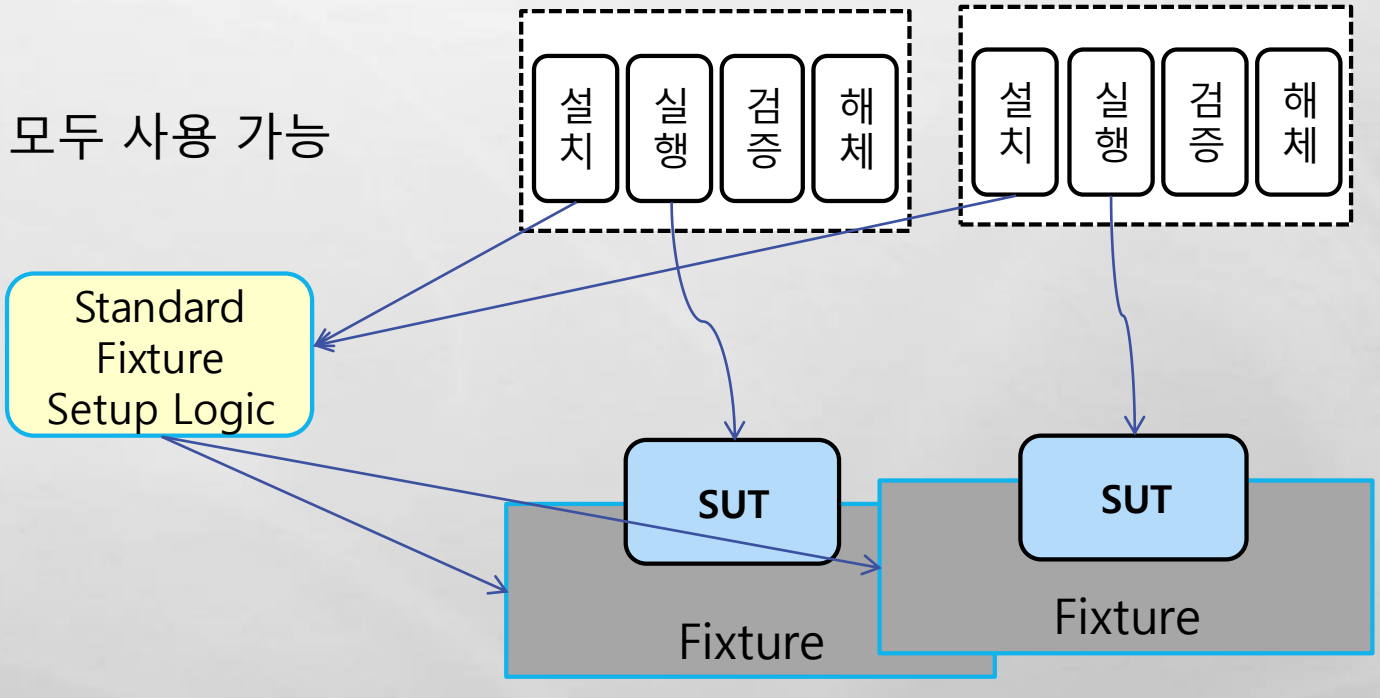



- By Fresh Fixture
  - 테스트할 때 마다 fixture 를 구성
- By Shared Fixture
  - 여러 테스트가 함께 사용하는 Fixture를  Minimal Fixture로 구성

# Setup - Standard Fixture



- 여러 테스트를 수행하는데 필요한 Fixture 를 설계, 재사용
- QA 테스팅에서 전통적으로 사용
- 여러 테스트에서 다른 목적으로 재사용할 수 있게 하면 할수록 Standard Fixture 는 더 커지고 복잡해진다.
  =>다른 Test에 영향을 주는 Fragile Fixture가 될 수 있음
- Fresh Fixture나 Shared Fixture 형태로 모두 사용 가능



# Setup - Fresh Fixture

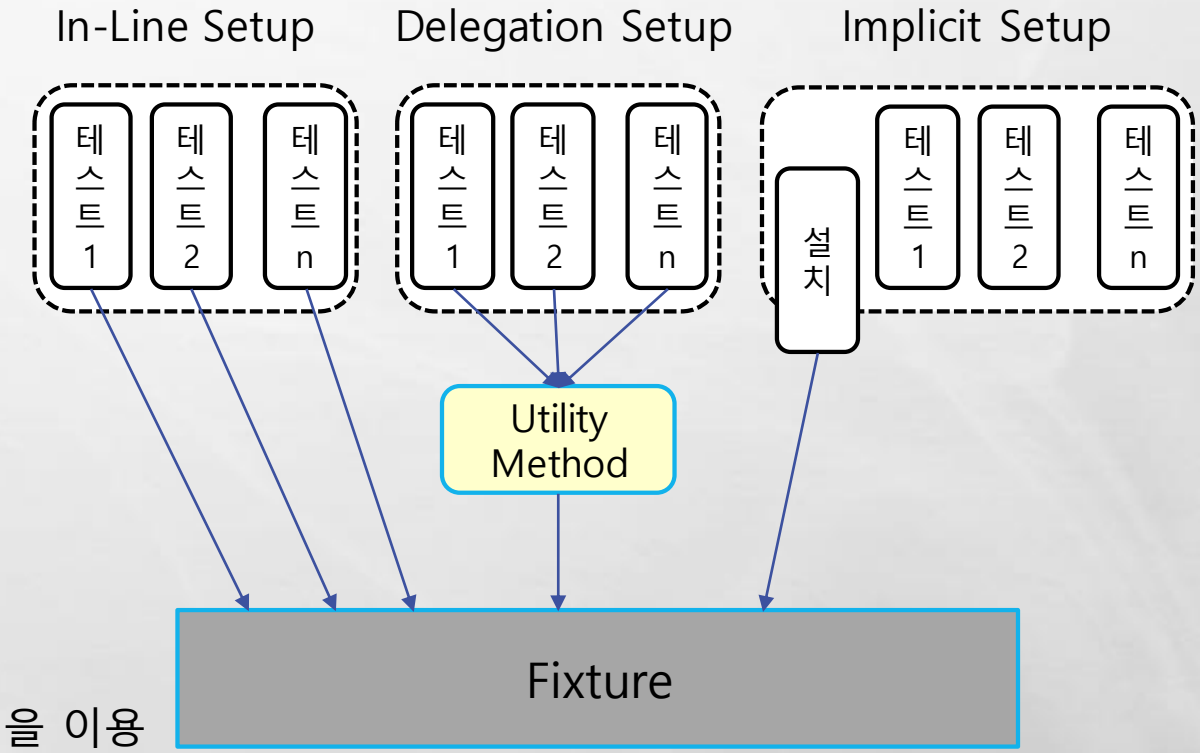


- Test 를 실행할 때 마다 전용 Fixture 설치 방식

- In-line Setup
  - Test 별로 필요한 Fixture 생성

- Delegation Setup
  - 생성 Method 를 이용
  - 테스트 코드 중복 방지, 유지보수성 유지
  - Parameterized creation method,
    Anonymous creation method,

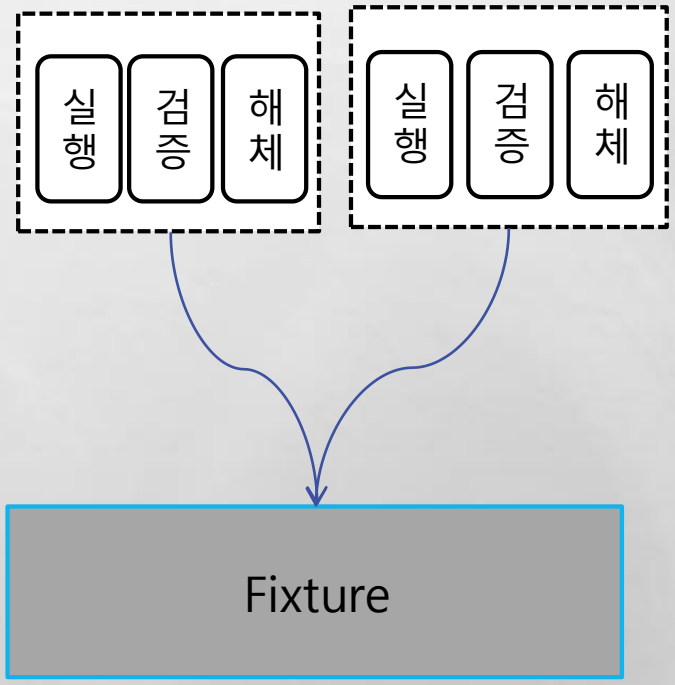
- Implicit Setup
  - Annotation 등 test framework 의 Fixture 설치 방식을 이용
    (gtest: TEST_F, Junit : @BeforeEach, Nunit : [Setup] 등)
  - 테스트가 이해하기 어려워질 수 있음
  - 지역변수 => 인스턴스변수 사용



# Setup - Shared Fixture



- 같은 Test Fixture 를 여러 테스트에서 재사용
- Shared Fixture 의 문제점
  - 어떤 테스트가 다른 테스트의 결과에 의존하는 결과를 만들기 쉽다.
  - 여러 테스트에 맞춰 설계되어 Minimal Fixture 에 비해 복잡하고 깨지기 쉽다.
- Shared Fixture 가 필요한 경우
  - 테스트마다 Fresh Fixture 를 생성하는 데 오래 걸릴 때.
  - 여러 단계의 작업이 이전 작업에 의존할 때.

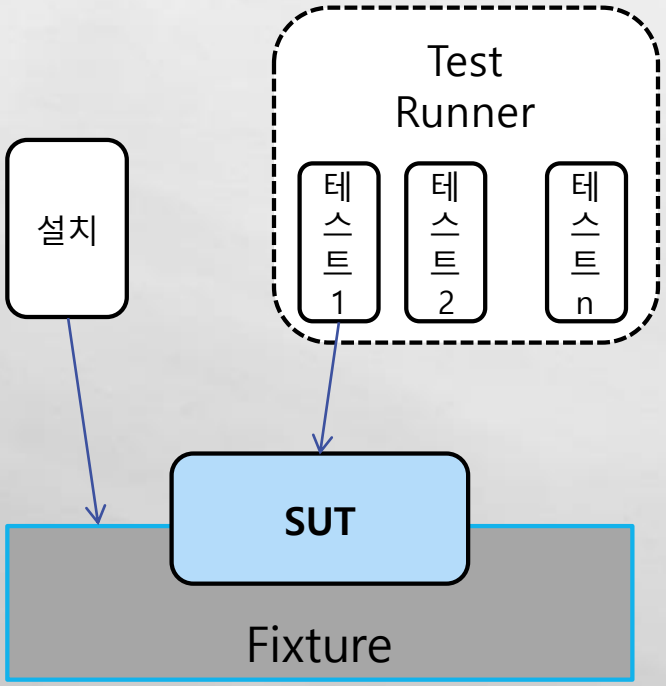


# Setup - Shared Fixture



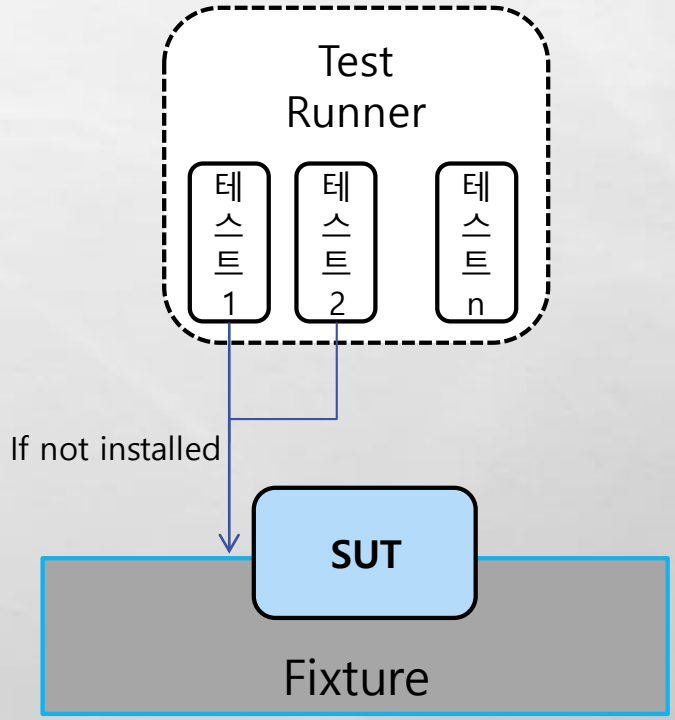
## Prebuilt Fixture

- Test 와 별도의 Fixture 설치
  - Test 를 실행하기 전에 Fixture 생성
  - Global Fixture




## Lazy Setup

- Fixture 를 필요로 하는 Test 에서 Lazy initialization

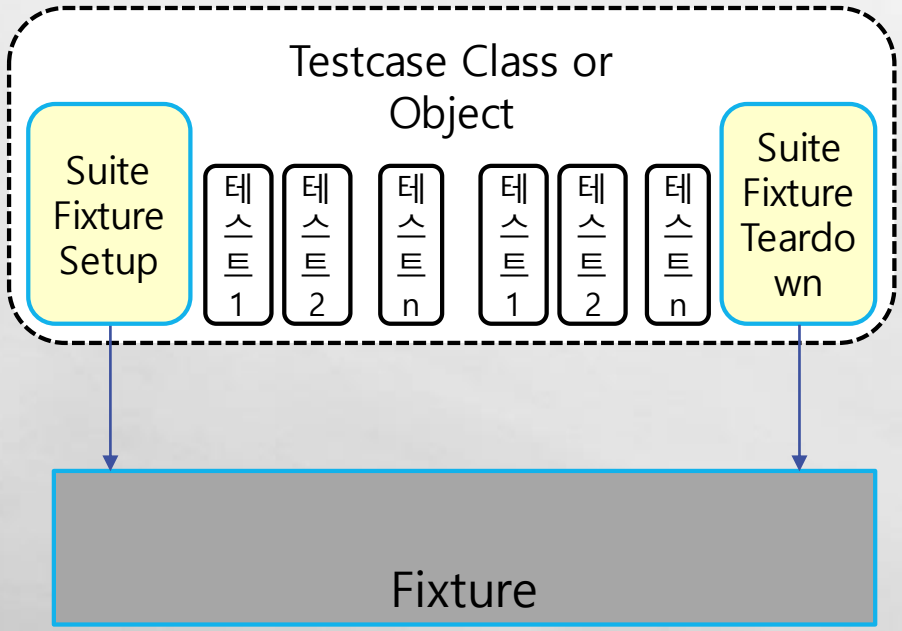


# Setup - Shared Fixture



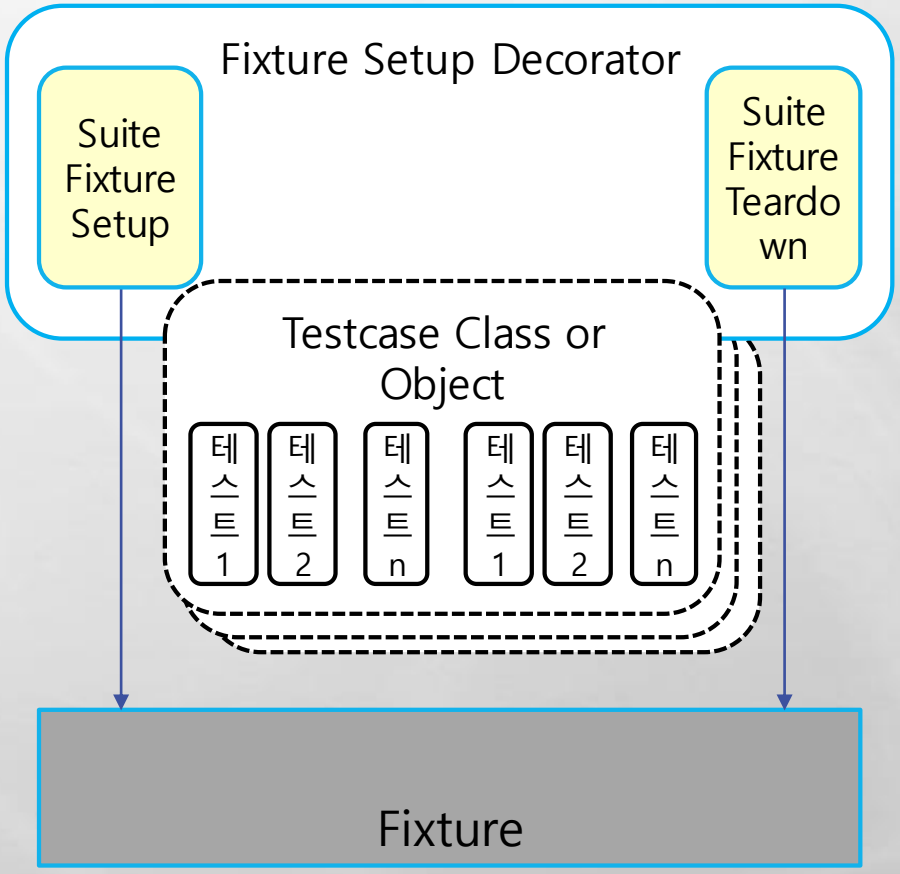
## Suite Fixture Setup

- 테스트 최초/최종에서 Fixture 를 setup




## Setup Decorator

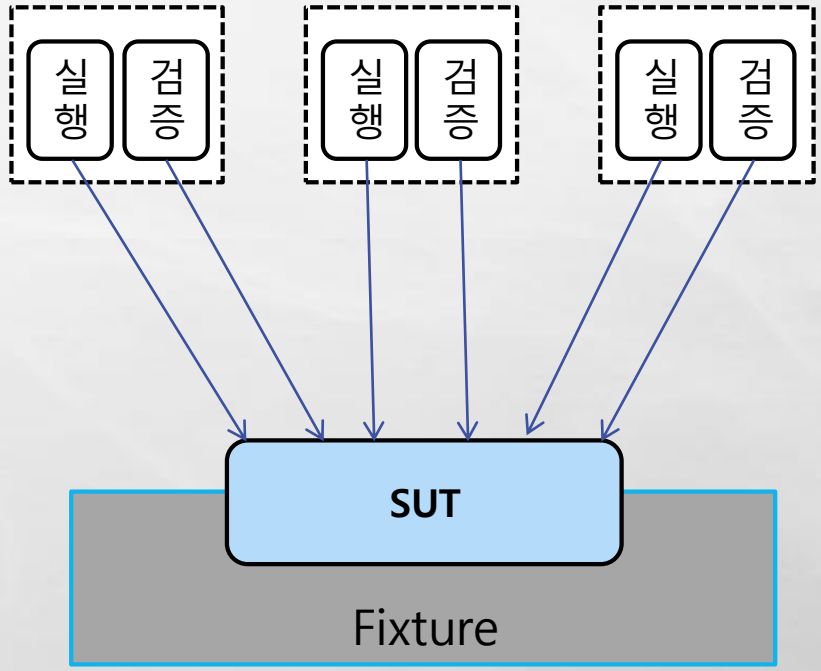
- Fake DataBase 를 만들어서 넘겨주는 등



# Setup - Shared Fixture



- Chained Test
  - 이전 테스트에서 설치하고 남겨둔 Test Fixture 가 다음에 실행되는 테스트의 Shared Fixture 로 재사용
  - Fragile Test: 이전 테스트의 영향으로 깨지지 쉬운 테스트가 될 수 있다.
  - Slow Test : Shared Fixture 를 사용하므로 인해 생기는 Slow Test 를 방지할 수 있다.



# Setup - Shared Fixture



- Immutable Shared Fixture
  - Shared Fixture 의 문제점을 해결하기 위한 방안.
  - Fixture 를 논리적으로 두 부분으로 분리하여 구성.
  - 어떤 테스트에서도 변경할 수 없는 부분

    => immutable Shared Fixture
  - 테스트에서 변경하고 삭제해야 하는 객체들로 이루어진 부분

    => Fresh Fixture 로 생성

# Exercise



- Named Test Suite
- Test Utility Method
- Parameterized Test
- Testcase Class per Class
- Testcase Class per Feature
- Testcase Class per Fixture
- Testcase Superclass
- Test helper

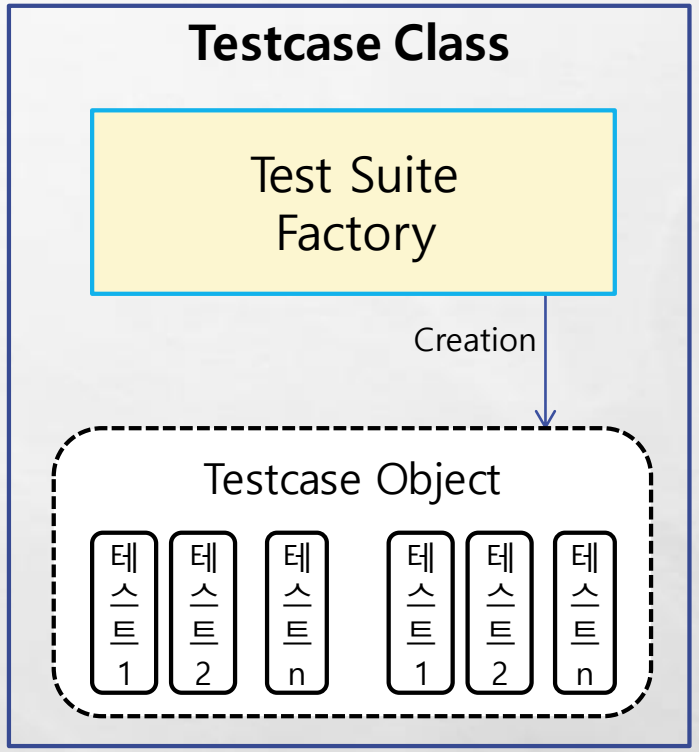
# Exercise – Named Test Suite

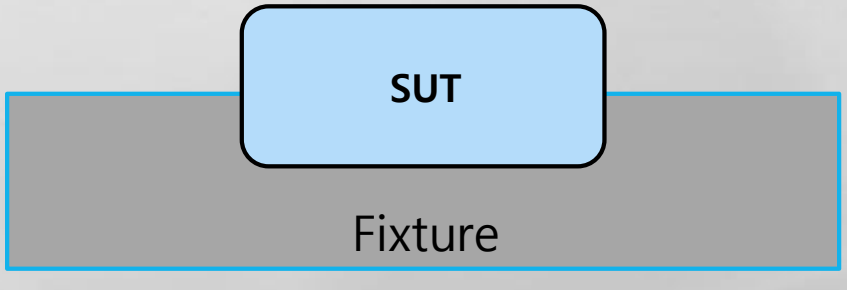


- 같은 이름으로 묶인 테스트들
- 기능적으로 서로 밀접하게 관련 있는 테스트 묶음
- 선택 실행 가능

```
TestSuite suite = new TestSuite();
suite.addTest(dbtest);
suite.addTest(uitest);
return suite;
```

```
TEST(TestSuiteName, TestName)
```



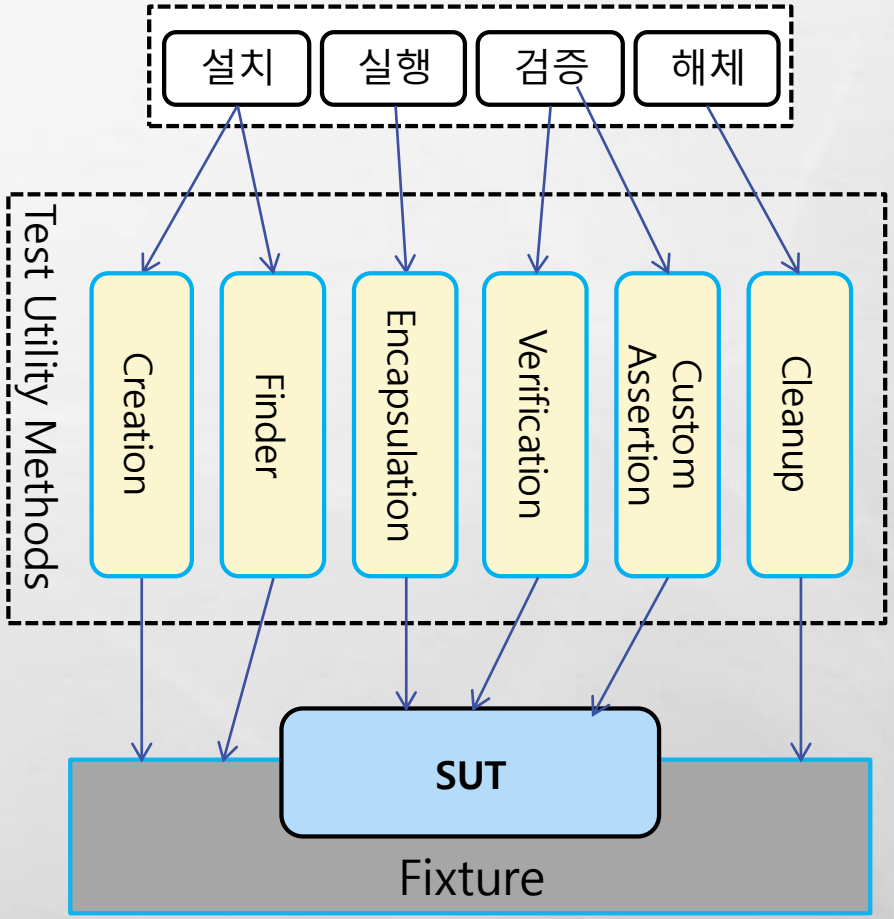
# Exercise – Test Utility Method



- 재사용을 위한 테스트 로직을 Utility Method 로 캡슐화
- 각 Test 의 Setup, Exercise, Verify, Teardown 에 중복된 코드를 Method 화 하는 것

- Creation Method : Fixture 설치의 일부
- Finder Method : Shared Fixture 에서 사용

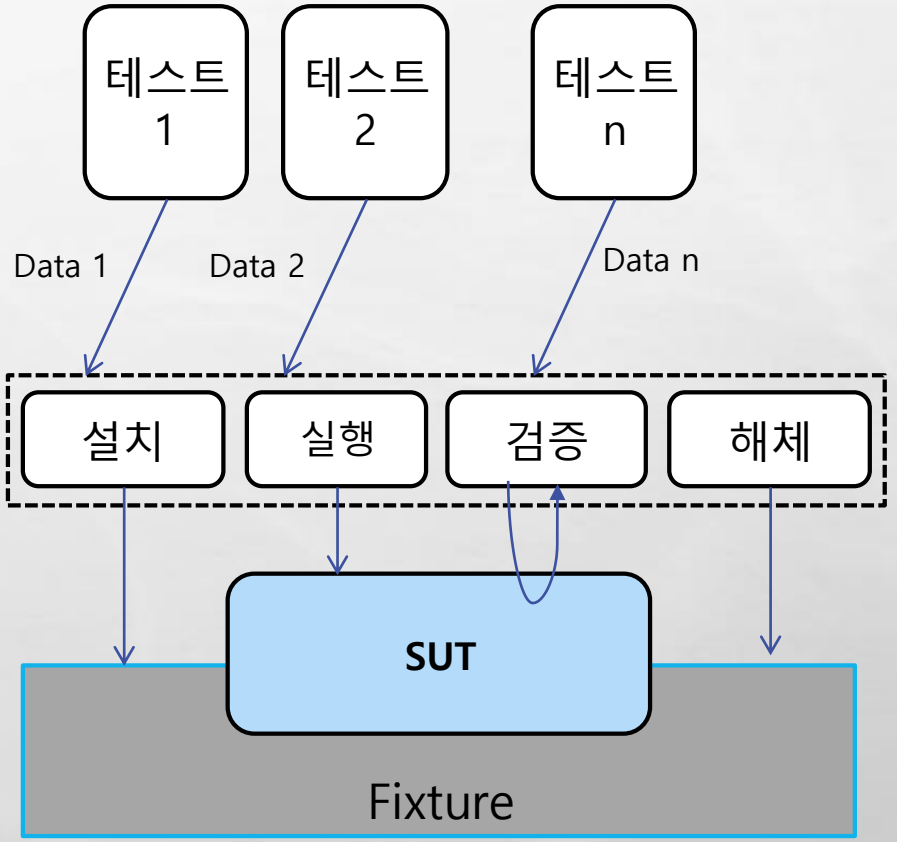
- Encapsulation Method : SUT Encapsulation
- Verification Method : SUT 와 상호작용
- Custom Assertion Method
  - 기대값과 실제값 비교 메소드.
  - SUT 와 상호작용 없어 부작용 없다.



# Exercise – Parameterized Test



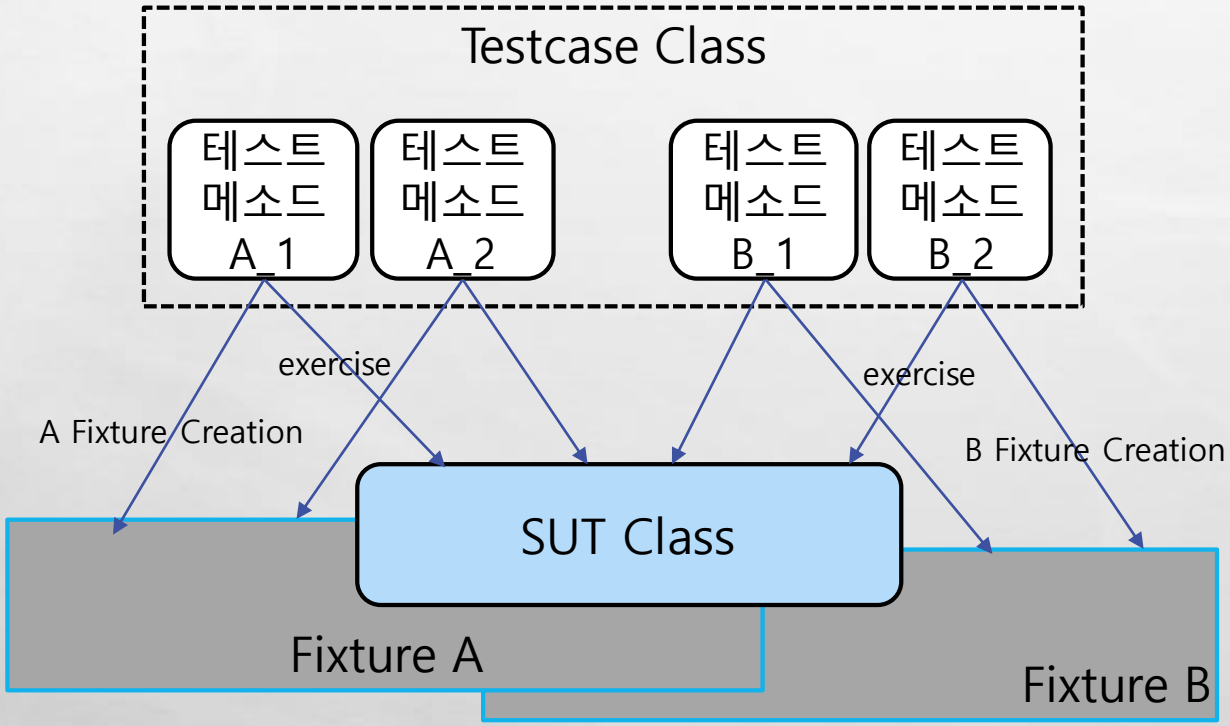
- 동일한 테스트 로직을 데이터만 약간씩 다르게 반복하는 경우
- 테이블 테스트
- 반복문 주도 테스트
- Framework가 지원하는 테이블 테스트



# Exercise – Testcase Class per Class



- 하나의 SUT(System Under test) Class 에 대해 하나의 Testcase Class 로 만든다
- 테스트 구성을 시작하는 간단한 방법

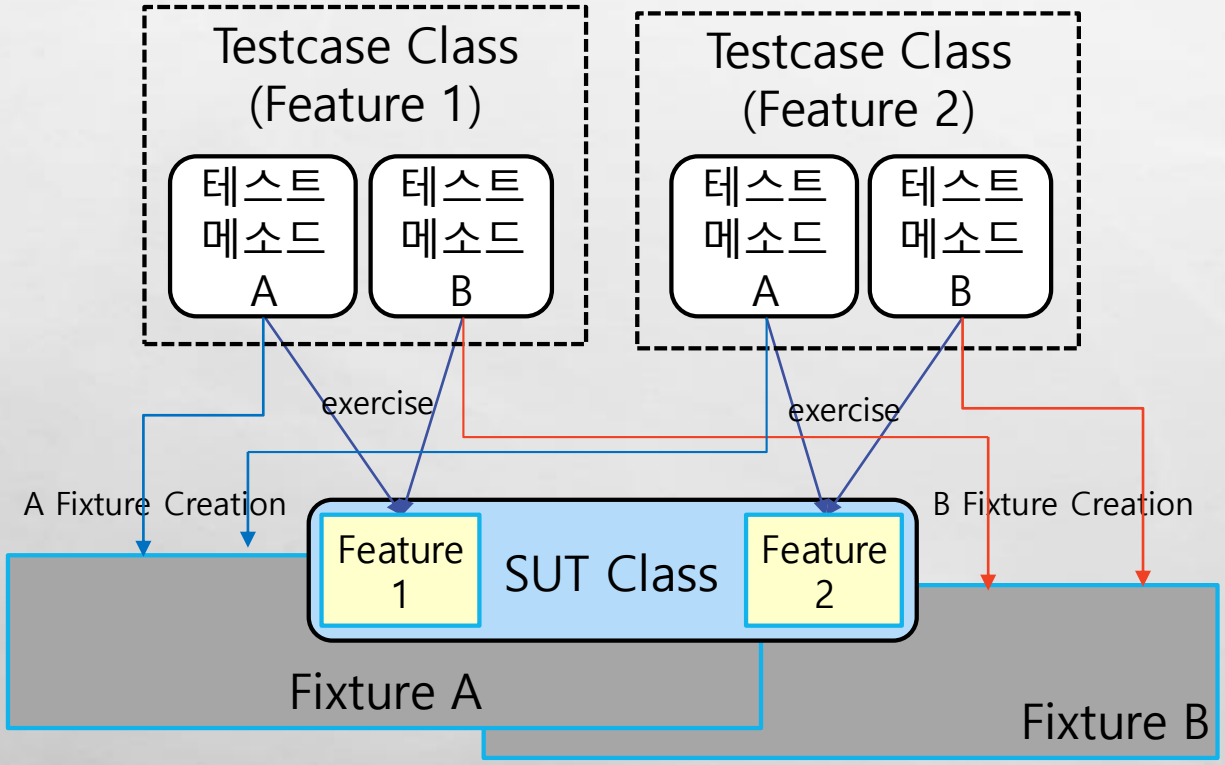


# Exercise – Testcase Class per Feature



- 동일한 SUT 의 기능에 대해 하나의 Testcase Class로 만든다.

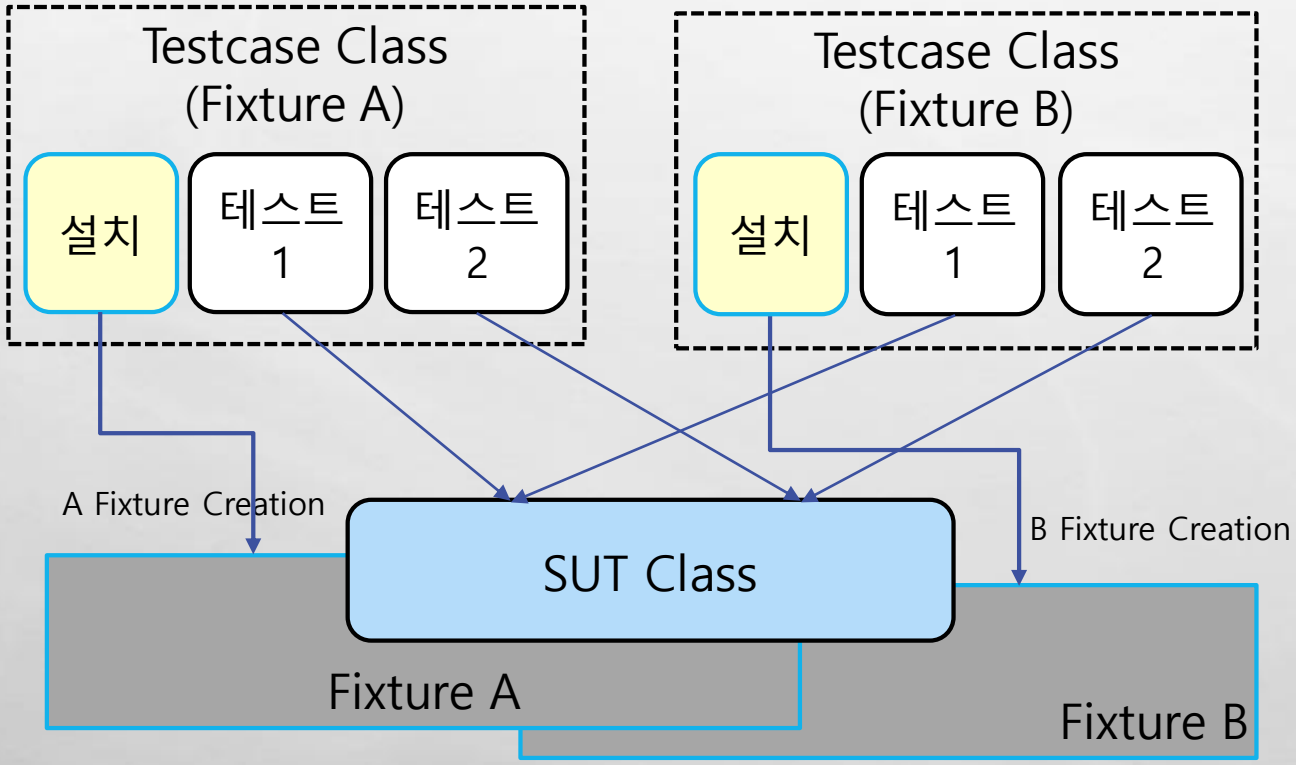
- Feature에 대한 Test가 많아 지고, 상세한 Test를 하는 경우



# Exercise – Testcase Class per Fixture



- 동일한 Fixture 에 대해 하나의 Testcase Class
- Minimal fixture, Implicit fixture setup
- Test as Document : Testcase class per Fixture 의 테스트 메소드 + 테스트 기대 결과 값

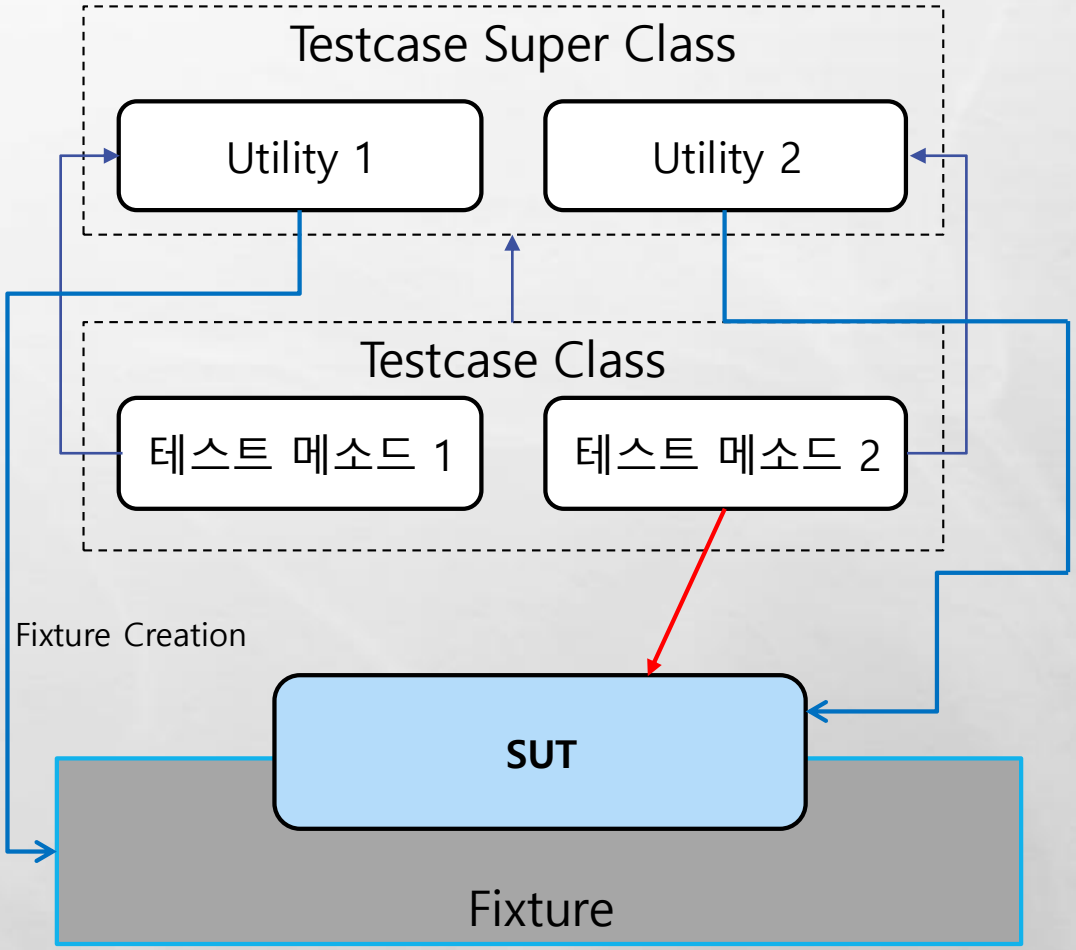


# Exercise – Testcase Superclass



- 동일한 테스트 로직이 반복되는 경우

- Testcase Class 간에 재사용 가능한 Test Utility Method 를 추상 상위 클래스로 정의.

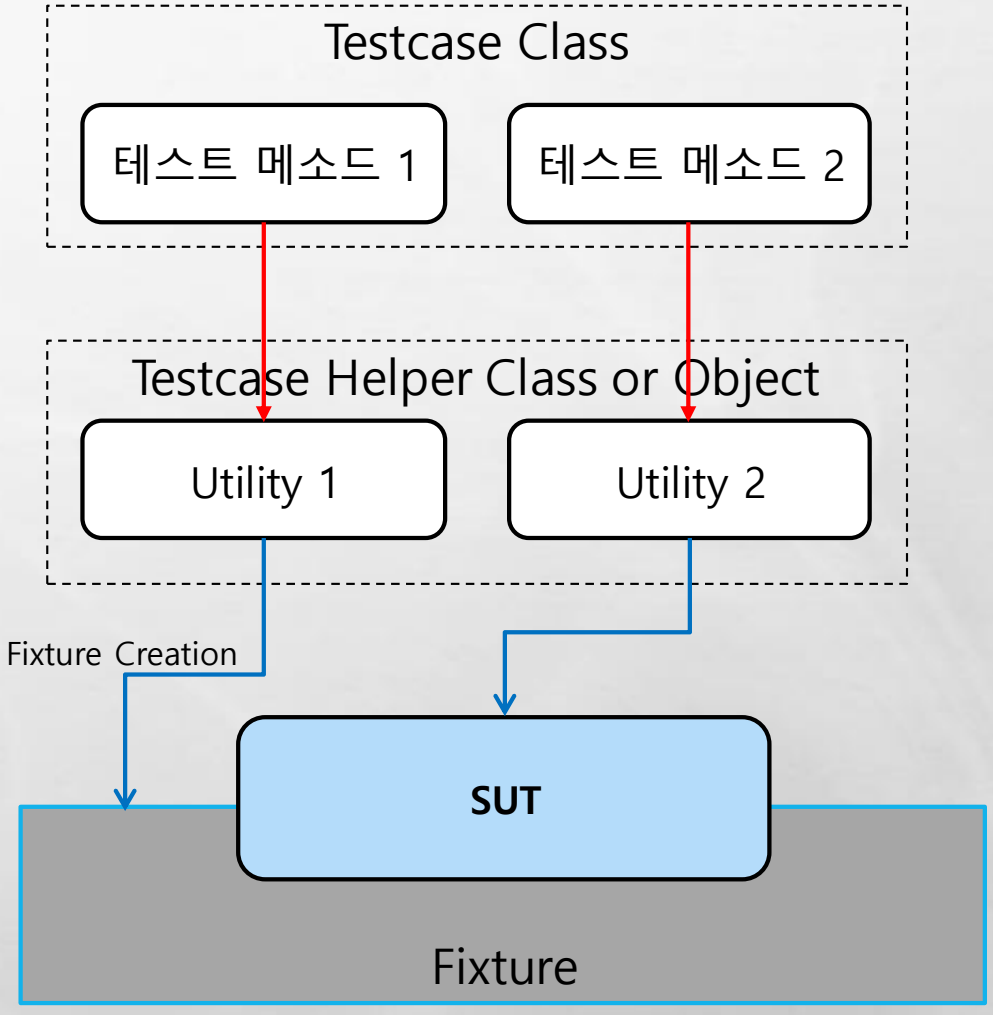


# Exercise – Test Helper



- Test Utility Method 들의 클래스를 정의
- Test Utility Method 재사용
- Test Superclass, Helper Class 는 타입 가시성의 문제

```
TestHelper helper;
helper.assert(myTestResult);
```



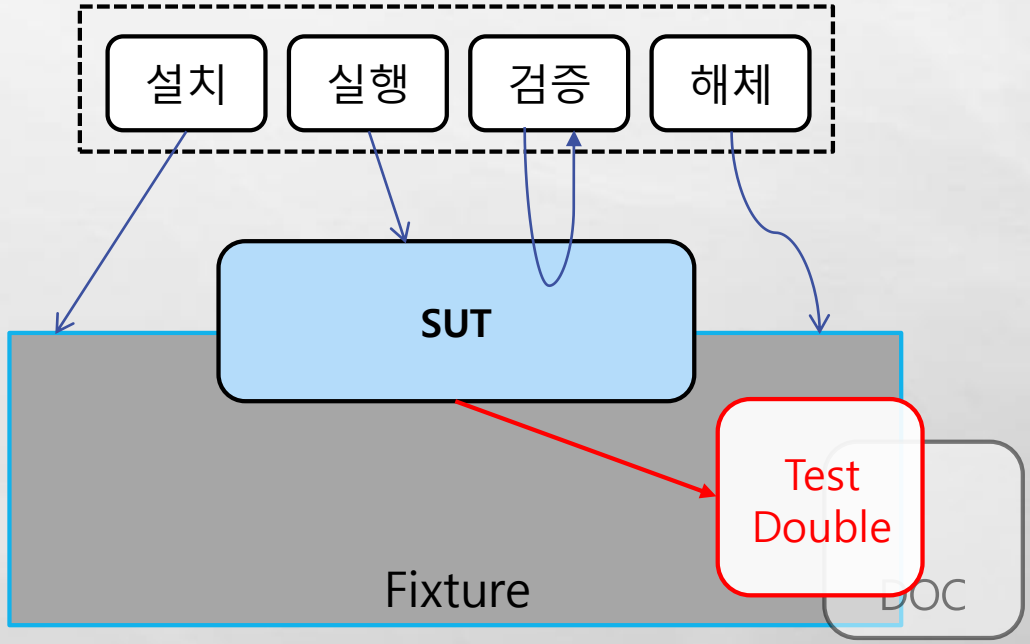
# Exercise – Test Double



- SUT가 의존하는 컴포넌트(DOC : Depend-On Component)를 대체하는 것
  - 느린 component
  - 예측할 수 없는 component
  - 간접 입력으로 SUT를 실행할 수 있는 제어 위치가 없는 경우
  - 간접 출력으로 동작 검증을 위한 관찰 위치가 없는 경우

- Test Stub
- Test Spy
- Mock Object
- Fake Object

- Configurable Test Double
- Hard-Coded Test Double
- Test-Specific Subclass

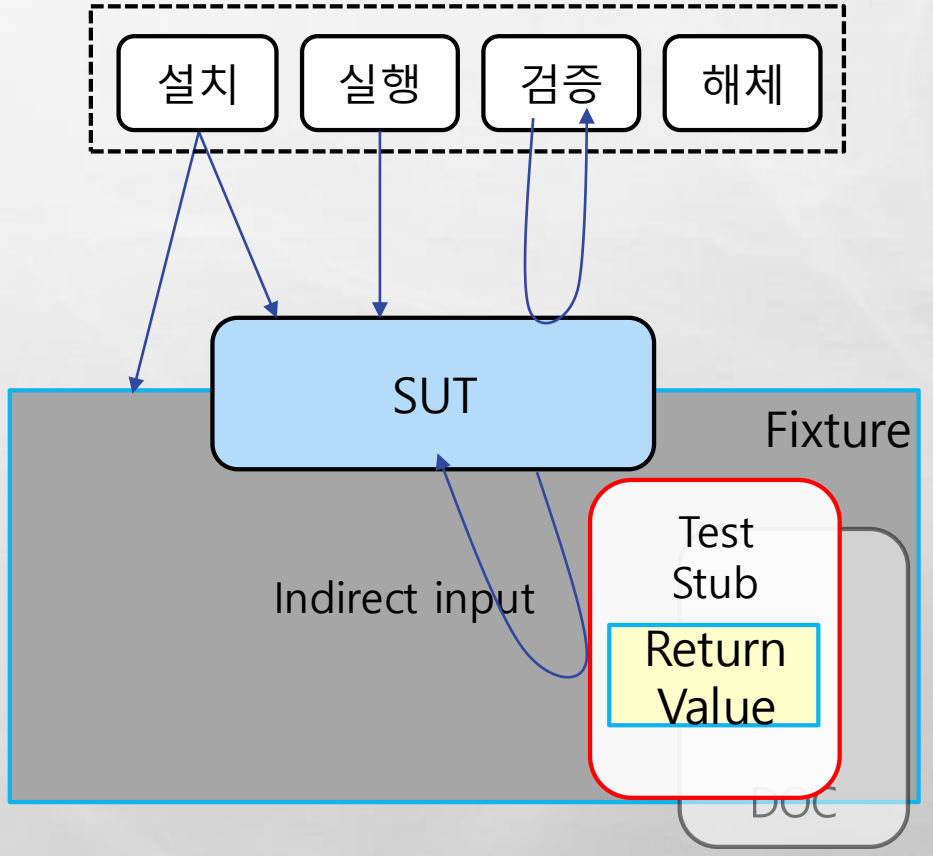


# Exercise – Test Stub



- SUT 에 간접 입력을 보내는 테스트용 객체
- SUT 가 호출하면 SUT 내의 테스트 안 된 코드가 실행될 수 있게 하는 값이나 예외 리턴.

- 유형
  - Simple Success Test
  - Expected Exception Test
  - Not implemented DOC
  - Entity Chain Snipping

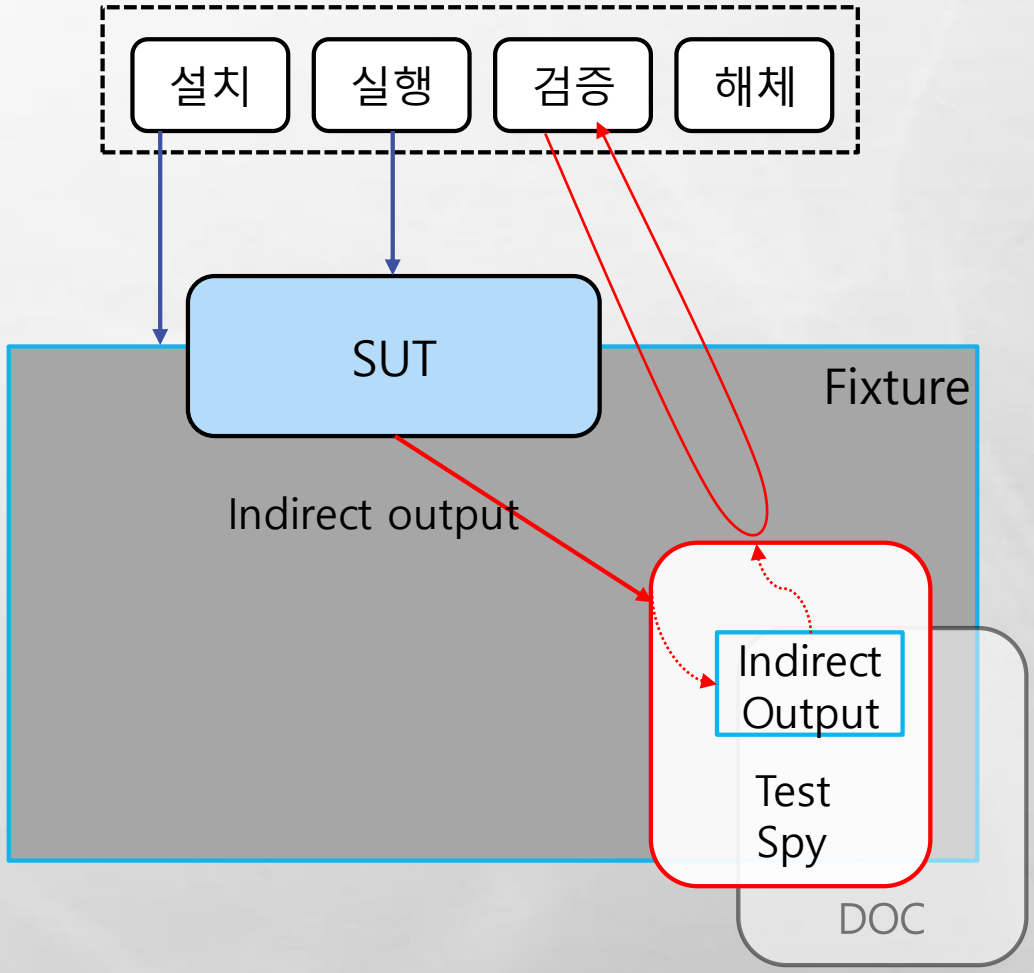


# Exercise – Test Spy



- Behavior Verify 를 위해, SUT 의 간접 출력을 확인하는 경우

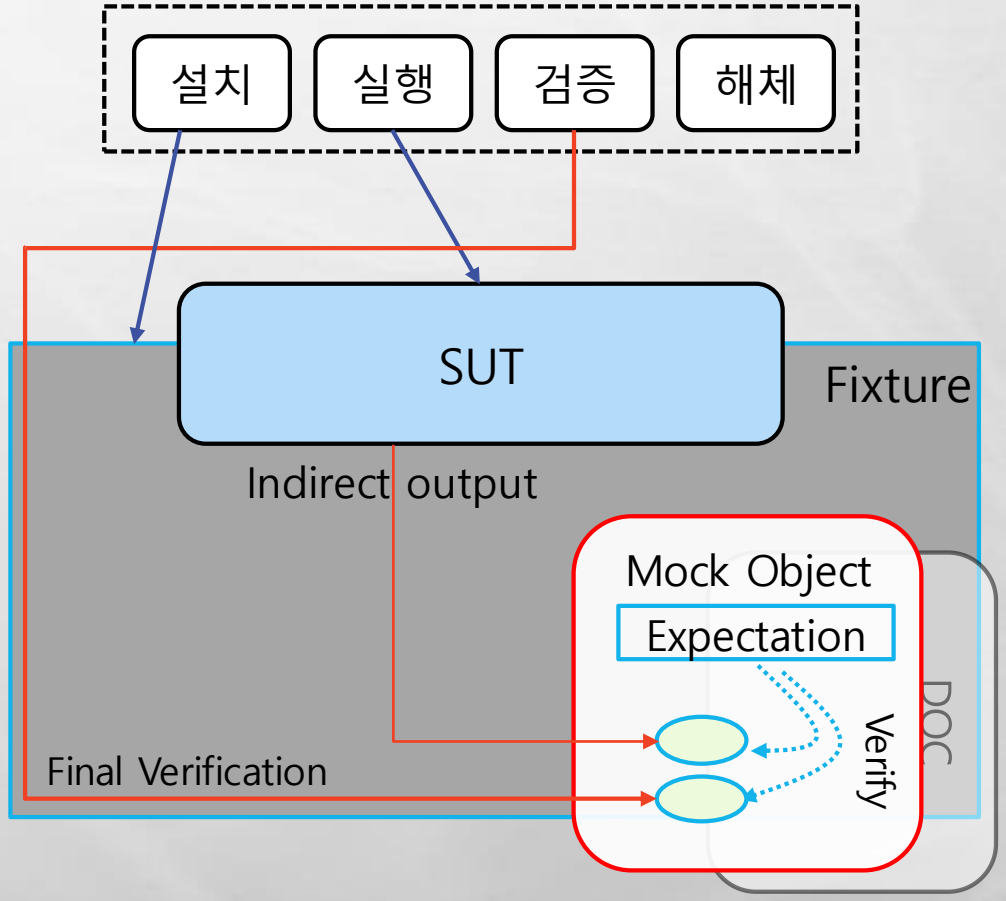
- SUT의 method 호출을 기록, 관찰하며,
  Verify 에서 이를 확인하기 위한 대역(double)



# Exercise – Mock Object



- SUT의 간접 출력이 DOC를 정확하게 사용하는지를 확인하기 위한 Double
- 테스트할 때 SUT에게 응답해 줄 값과 SUT에서 호출할 것으로 예상되는 메소드 호출을 설정.
- SUT method를 호출했을 때의  부작용 확인을 위한 동작 검증 목적.
- Mock Object 에서 Assert을 통해 테스트.

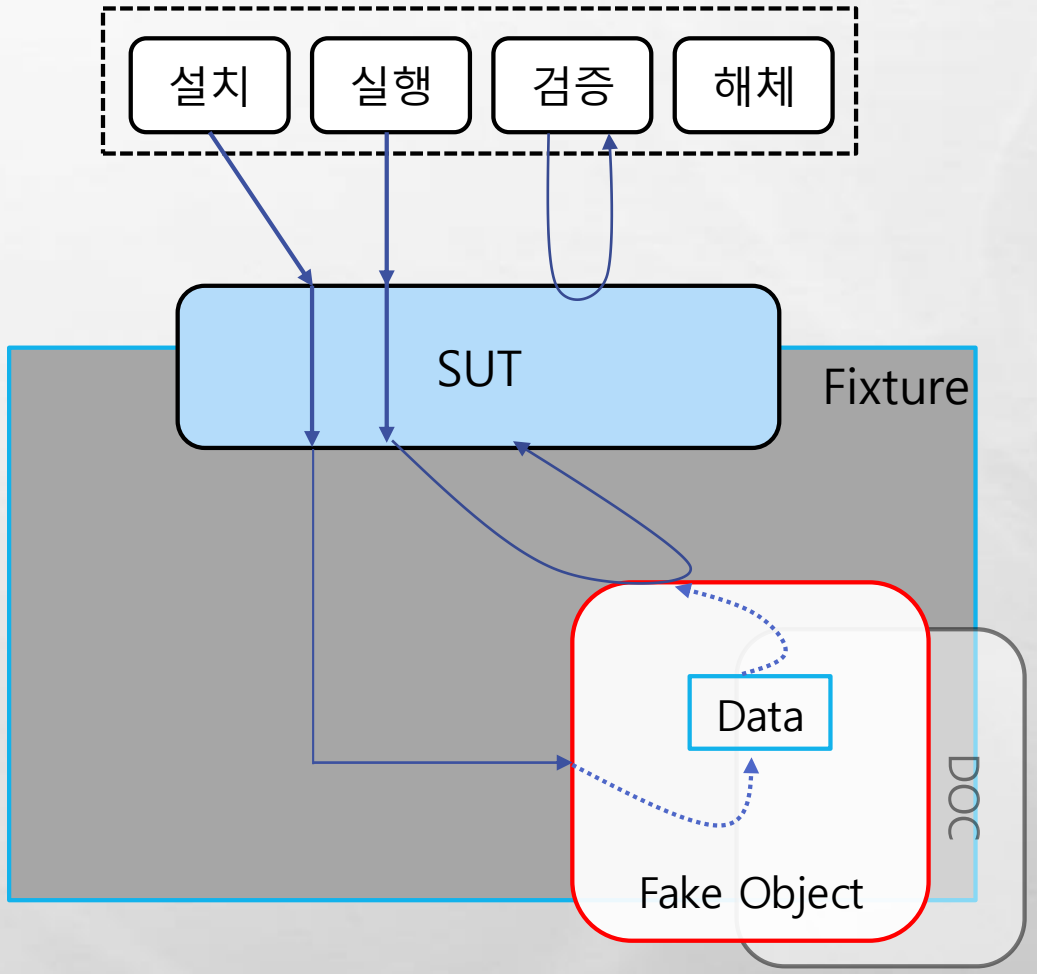


# Exercise – Fake Object



- DOC 의 동작을 가볍게 구현한 대역
- SUT 에 일관된 상호 작용 제공

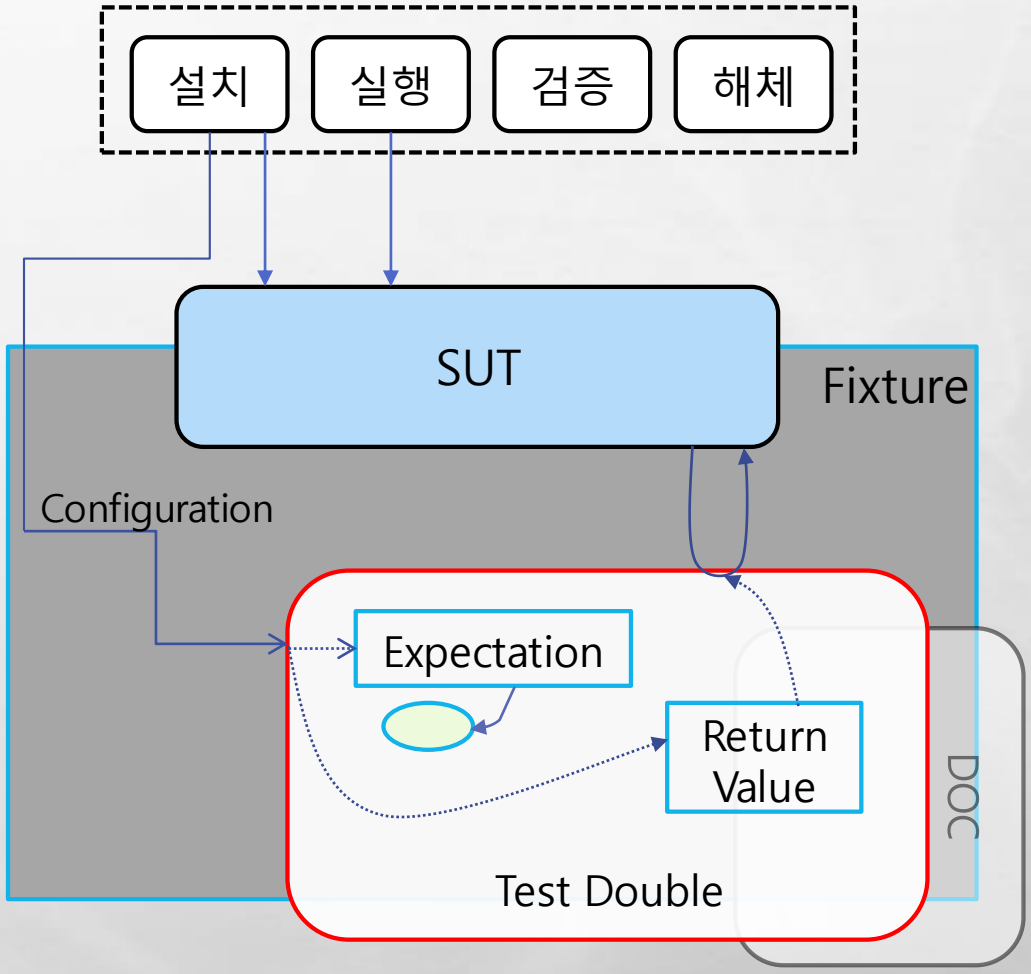
- Fake Database
- Fake Memory Database
- Fake web service
- Face service layer



# Exercise – Configurable Test Double



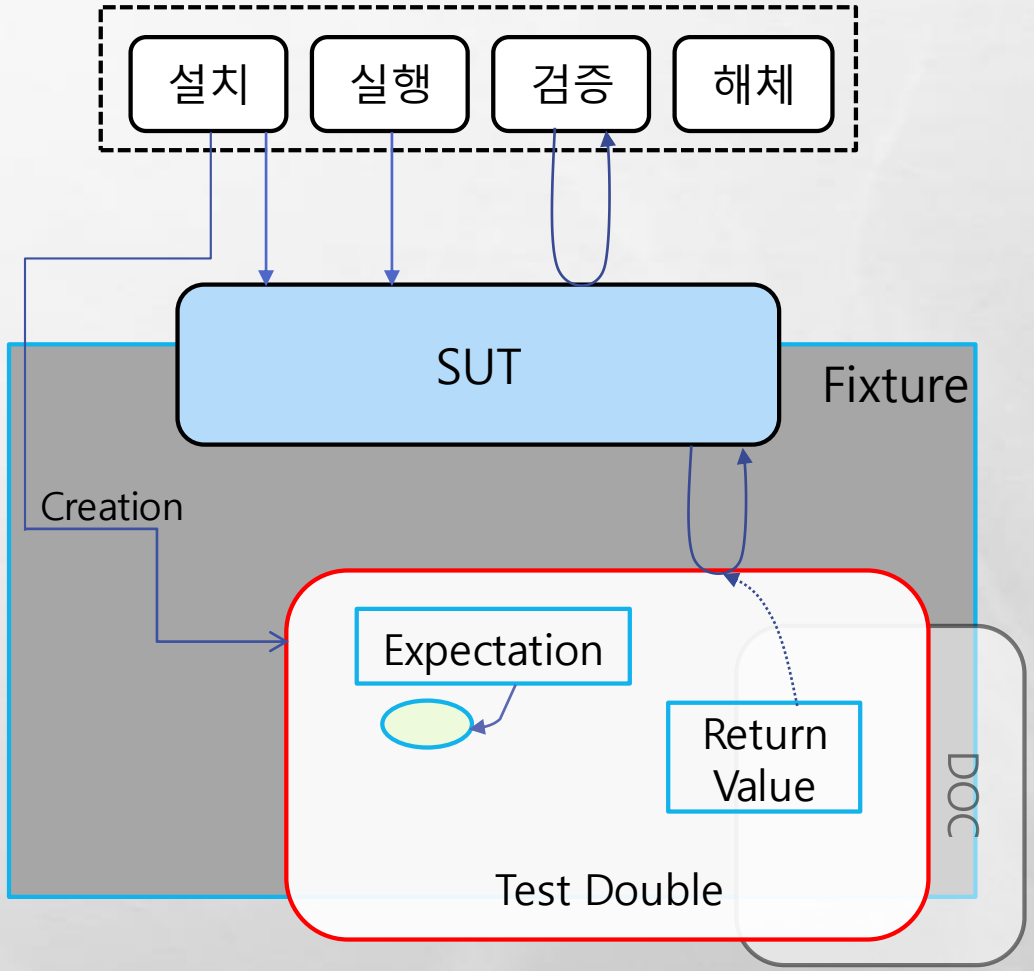
- 테스트 Fixture 설치 단계에서 Test double에 Return 값이나 예상 메소드 인자값을 설정
- 재사용으로 테스트 코드 중복 방지



# Exercise – Hard-Coded Test Double



- Return value와 expectation 을 하드 코딩한 테스트 대역
- Test Double이 단순하거나, 하나의 테스트에만 사용될 때의 해결책
- SUT 가 변경될 때마다 테스트 코드의 유지보수, 리팩토링 필요.

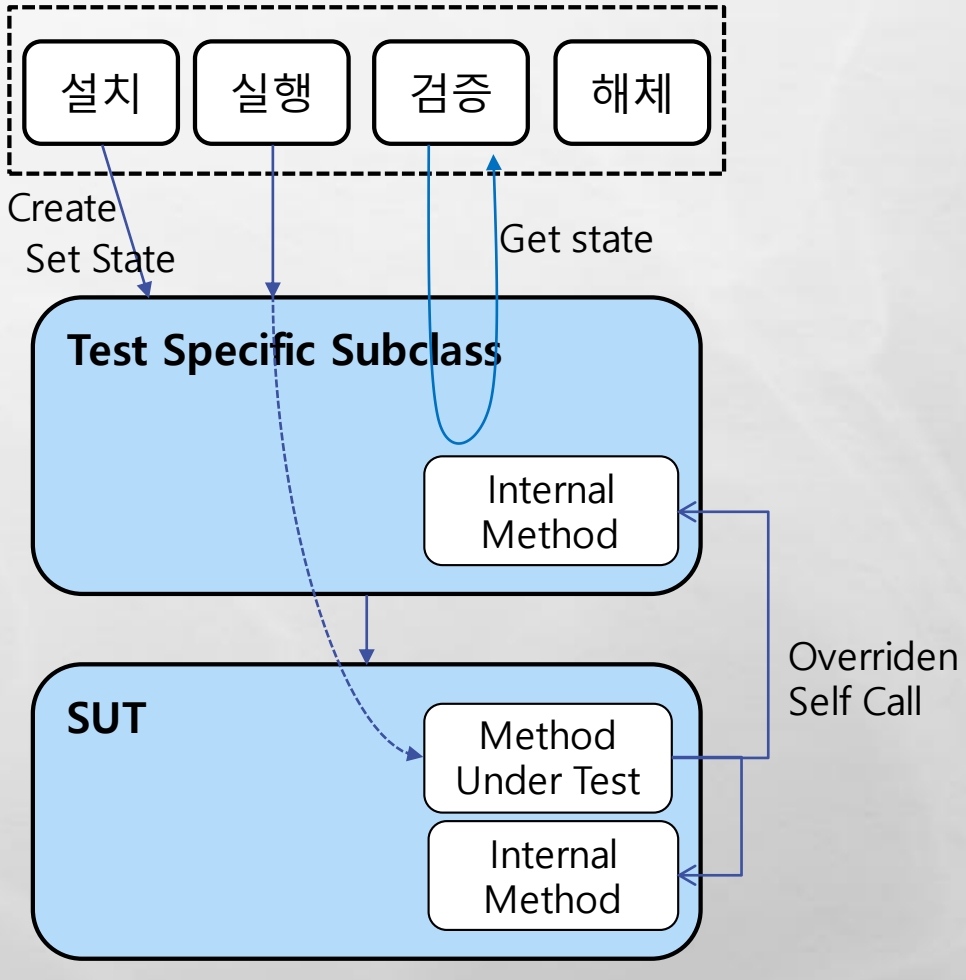


# Exercise – Test-specific Subclass



- SUT가 테스트 가능하게 설계되지 않은 경우, 테스트에 필요한 상태나 동작을 드러내는 방식
- SUT는 고치지 않은 채, SUT의 상태를 공개하는 방법.

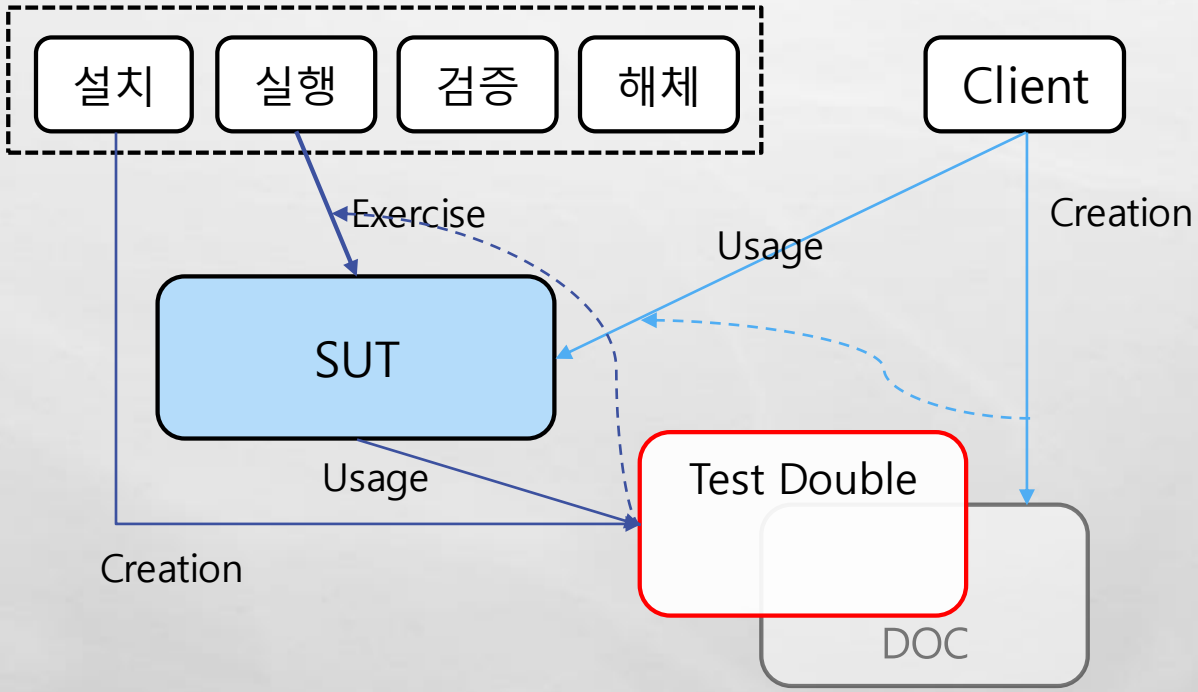
- 상태 노출 하위 클래스
- 동작 노출 하위 클래스
  - Superclass 의 method를 노출한다.
- 동작 변경 하위 클래스
  - Superclass 의 동작을 바꿈.



# Exercise - 테스트를 하기 쉬운 설계 패턴(1/4)



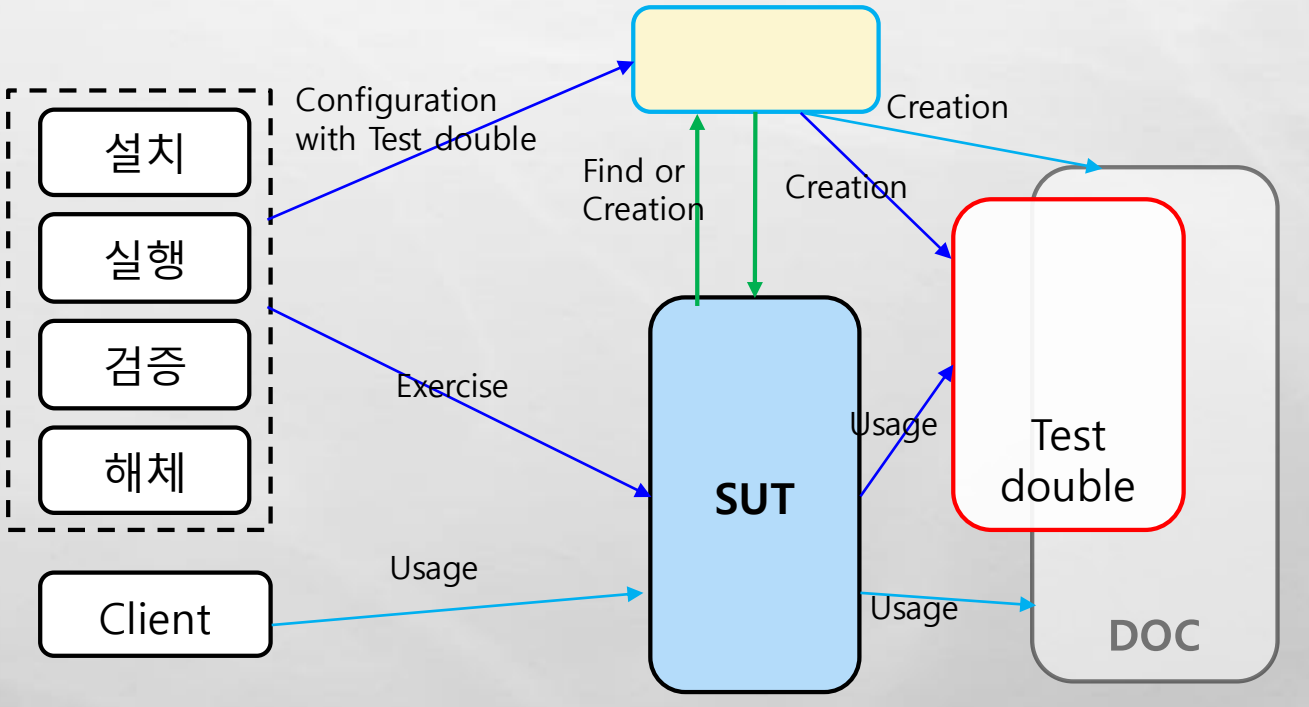
- 의존 주입(Dependency Injection)
  - 클라이언트나 시스템 설정에서 SUT가 실행될 때 DOC 를 제공.
  - 의존을 명시할 수 있는 방법을 SUT API 로 작성.
  - 여러 테스트에서 DOC를 Test Double 로 교체하는 방법



# Exercise - 테스트를 하기 쉬운 설계 패턴(2/4)



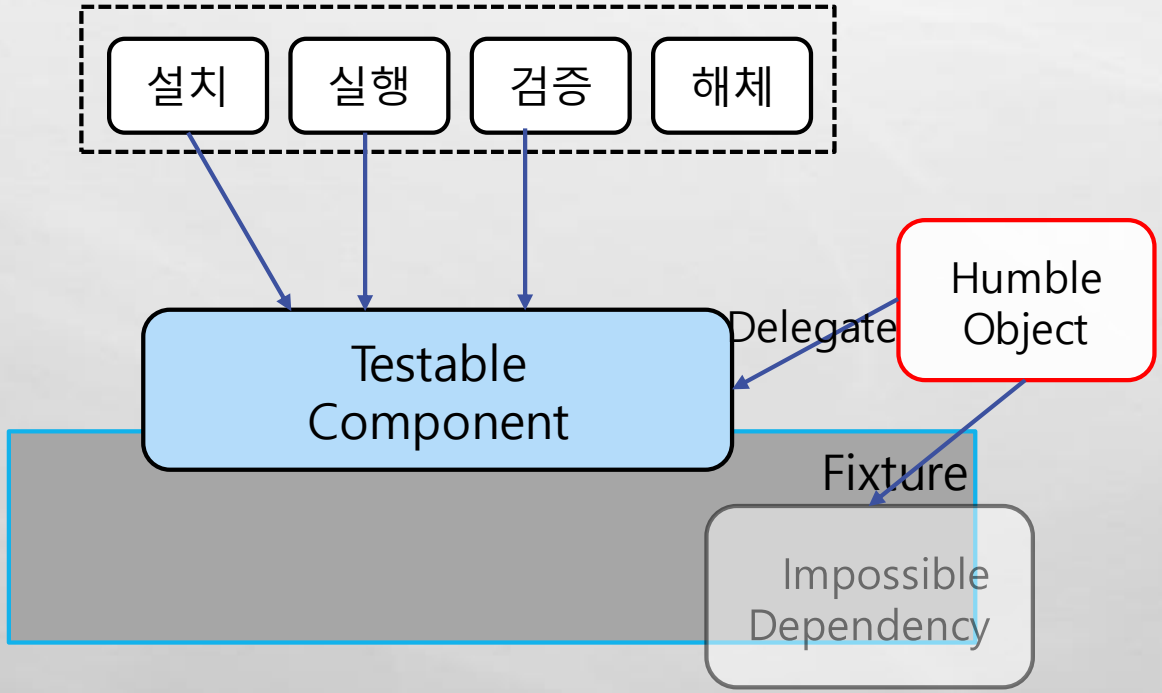
- 의존 찾기(Dependency Lookup)
  - SUT 가 어떤 Component 를 사용할 지를 알려 줄 수 있는 방법을 제공.
  - 다루기 어려운 의존이 있거나 테스트 성능을 높이기 위해 테스트할 때 의존을 바꿔 쓰는 경우
  - 예 : Fake DB 와 In-Memory DB, 웹 사이트와 어플리케이션 서버 의존
  - 기존 레거시 소프트웨어 도입을 간단하게 해 준다.



# Exercise - 테스트를 하기 쉬운 설계 패턴(3/4)



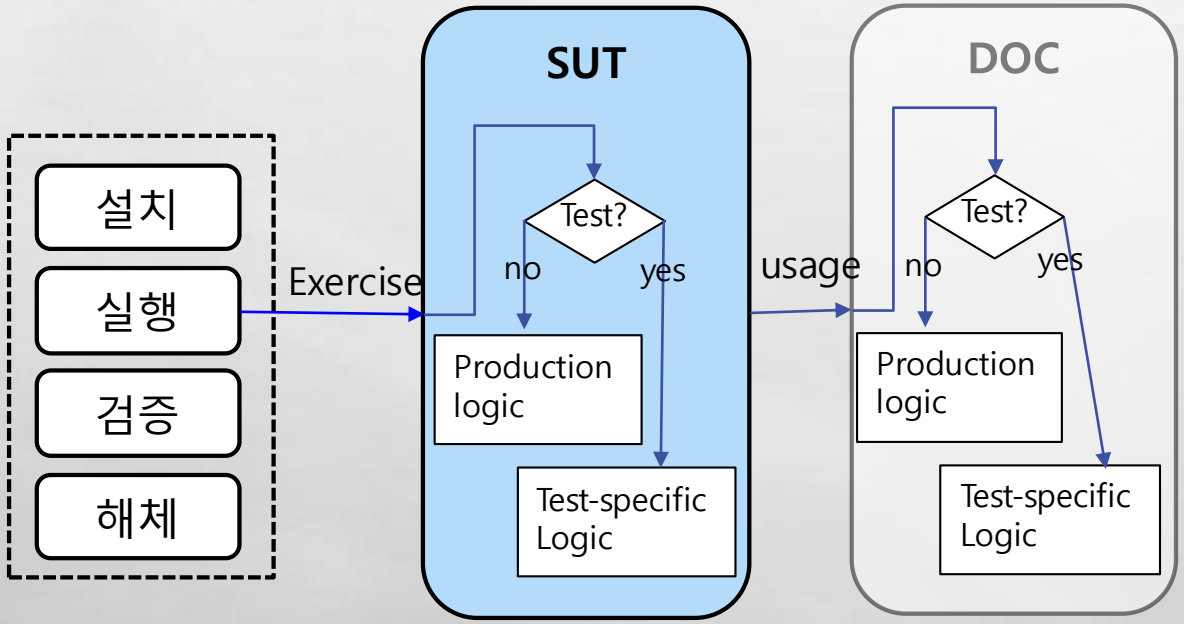
- 대강 만든 객체(Humble Object)
  - 환경에 강하게 결합된 코드를 테스트하기 쉬운 Component 를 테스트할 수 있는 방법
  - 중요한 로직이 들어있는 Component가 Framework 에 의존하거나 비동기적으로 접근하는 경우 유용.
  - 예 : Visual Component, Transaction Component Plug-In, Thread, Process, User Interface
  - Humble Object Component는 코드가 거의 없는 얇은 Adapter Layer 가 된다.



# Exercise - 테스트를 하기 쉬운 설계 패턴(4/4)



- 테스트 훅(TEST HOOK)
  - SUT나 DOC 안에 HOOK 을 직접 걸어 SUT 의 동작을 테스트에 맞게 변경하는 방법.
  - 의존 주입, 의존 찾기, 모두 할 수 없을 때 사용하는 최후의 수단.
  - 레거시 코드를 테스트 범위 안으로 두기 위한 변경 전략으로 사용 가능.
    => 테스트 훅으로 레거시 코드를 테스트 가능하게 한 후 리팩토링하여 테스트 훅 제거.

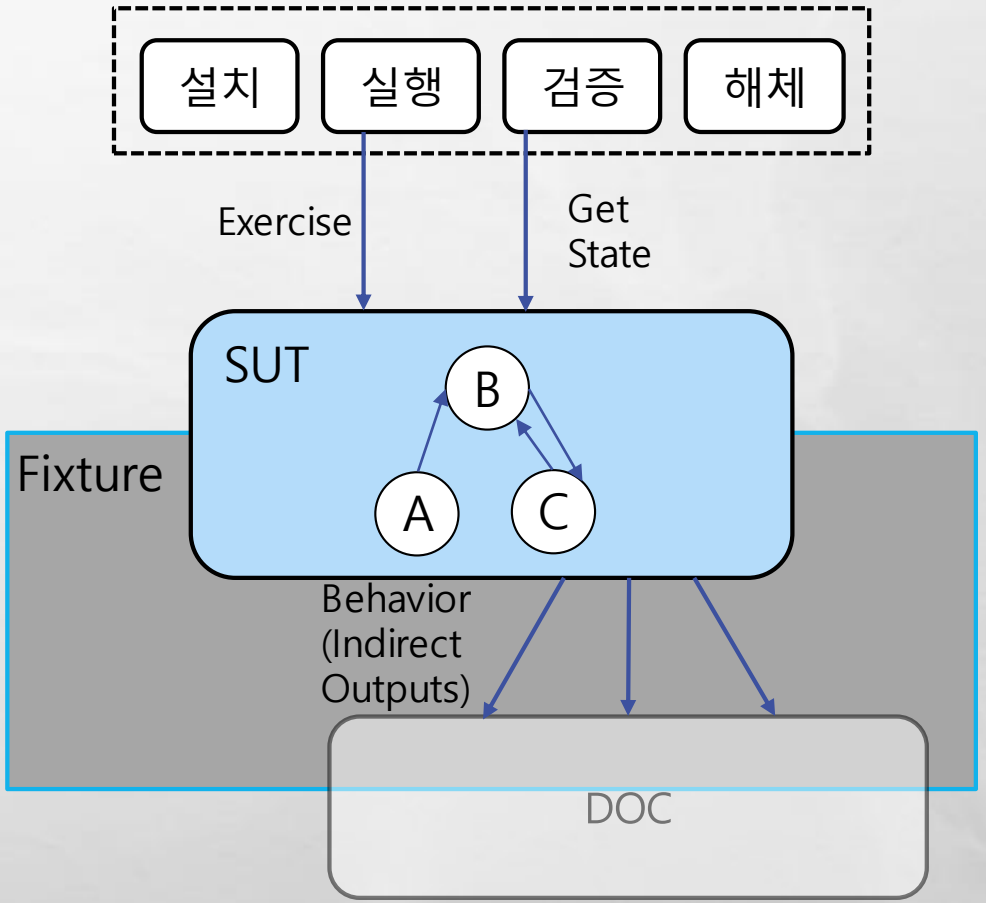


# Verify - Verification



- 상태 검증(State verification)
  - SUT 실행한 후 SUT의 변경된 상태 확인하여 기대값과 비교.
  - SUT의 최종 상태만 검증할 때 사용.

- 상태 검증 구현 방식
  - 절차형 상태 검증
    - 단언 메소드를 여러 번 호출하는 방식으로 구현
    - 애매한 테스트, 코드 중복 가능
  - 기대 상태 명세
    - 기대 속성이 설정된 하나  이상의 객체를 만들어 실제 객체의 상태와 비교
    - Test as Document 만드는 데 용이.

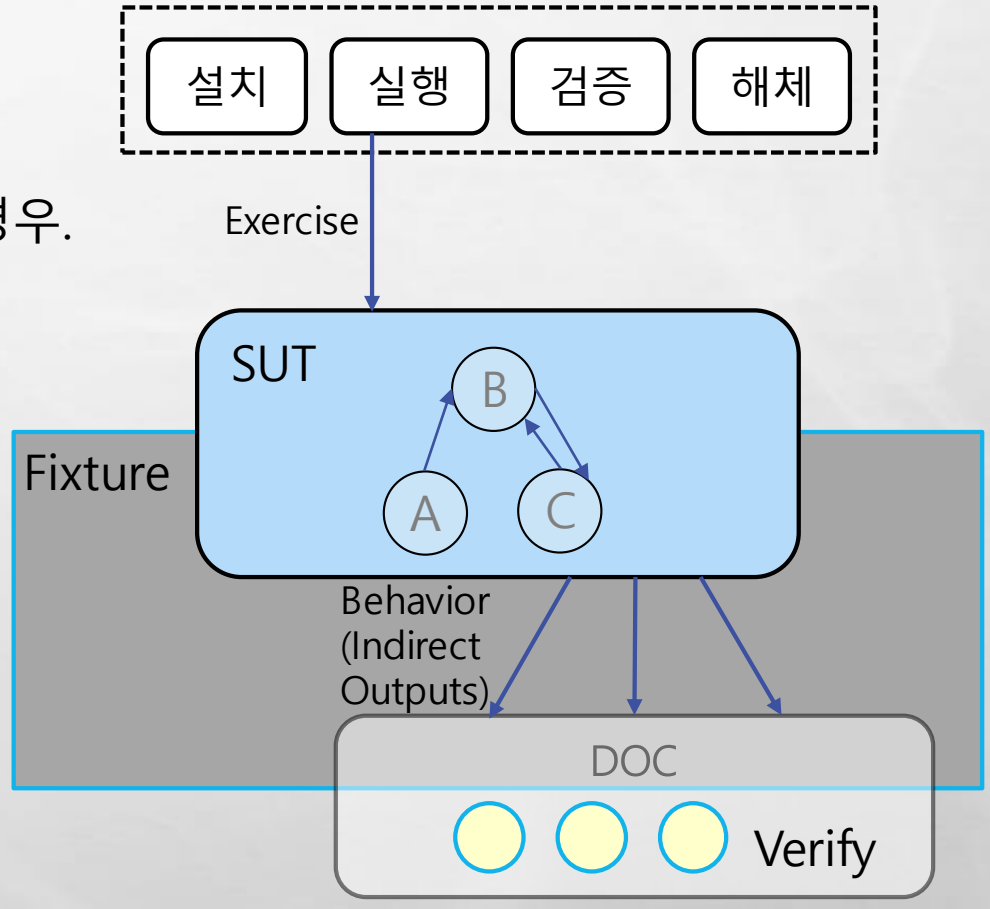


# Verify - Verification



- 동작 검증(Behavior verification)
  - SUT 가 실행되는 동안 발생하는 간접 출력 검증.
  - SUT의 상태가 없거나 바뀌지 않는 경우
  - 내부 동작의 상세한 확인이 필요한 경우
  - SUT 가 다른 객체나 컴포넌트 메소드를 호출하는 경우.
  - 많은 Test Double 필요해지는 문제점 발생
    => 깨지기 쉬운 테스트, 높은 테스트 유지 비용

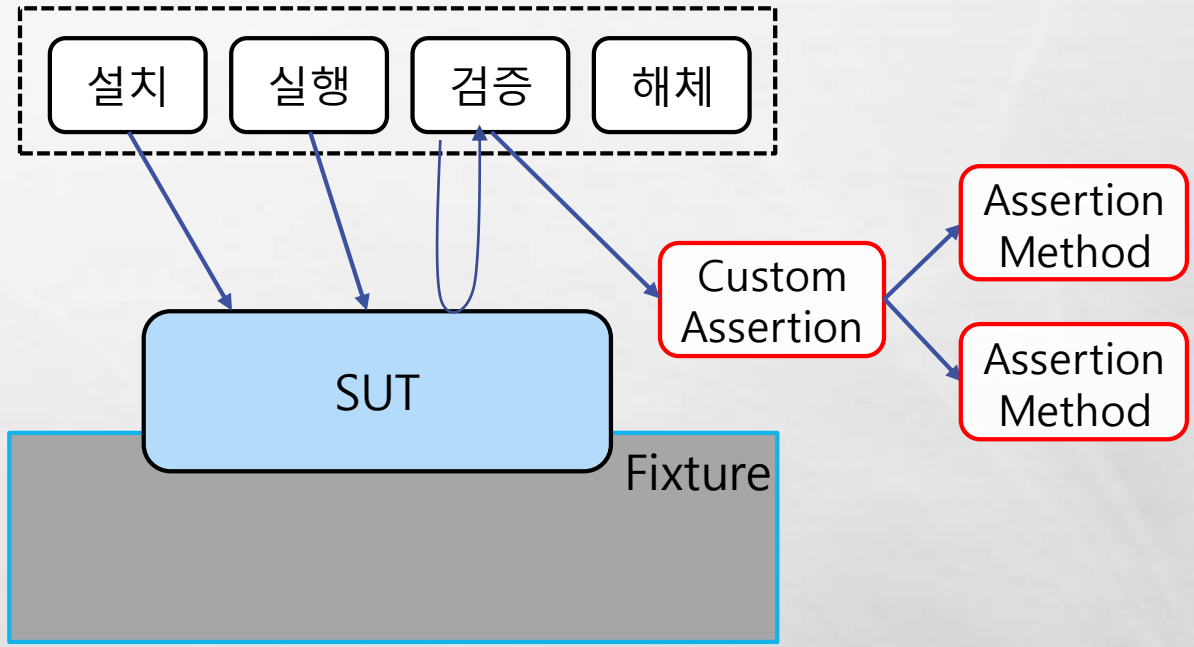
- 동작 검증 구현 방식
  - Spy 객체를 이용한 절차형 동작 검증
  - Mock 객체를 이용한 기대 동작 명세



# Verify – Assertion – Custom Assertion



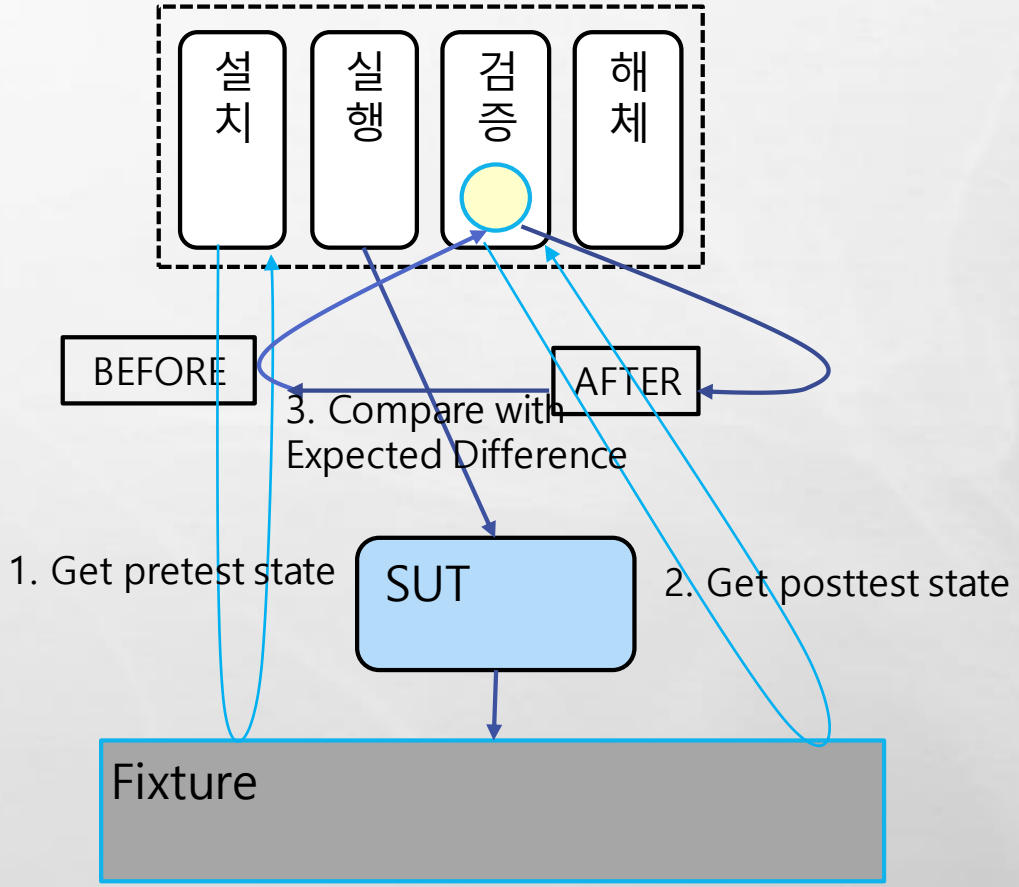
- ~를 해주는 단언문이 있으면 테스트가 쉬울 텐데 => Custom Assertion 작성
- 여러 테스트에서 동일한 단언문 중복
- 테스트 결과 확인이 이해하기 어렵고 순차적이다 – 애매한 테스트(Obscure Test)
- 조건문/반복문 내의 단언문
- 단언문이 실패해도 정보가 충분치 않다.
  - 잦은 디버깅(Frequently Debugging)



# Verify – Assertion – Delta Assertion



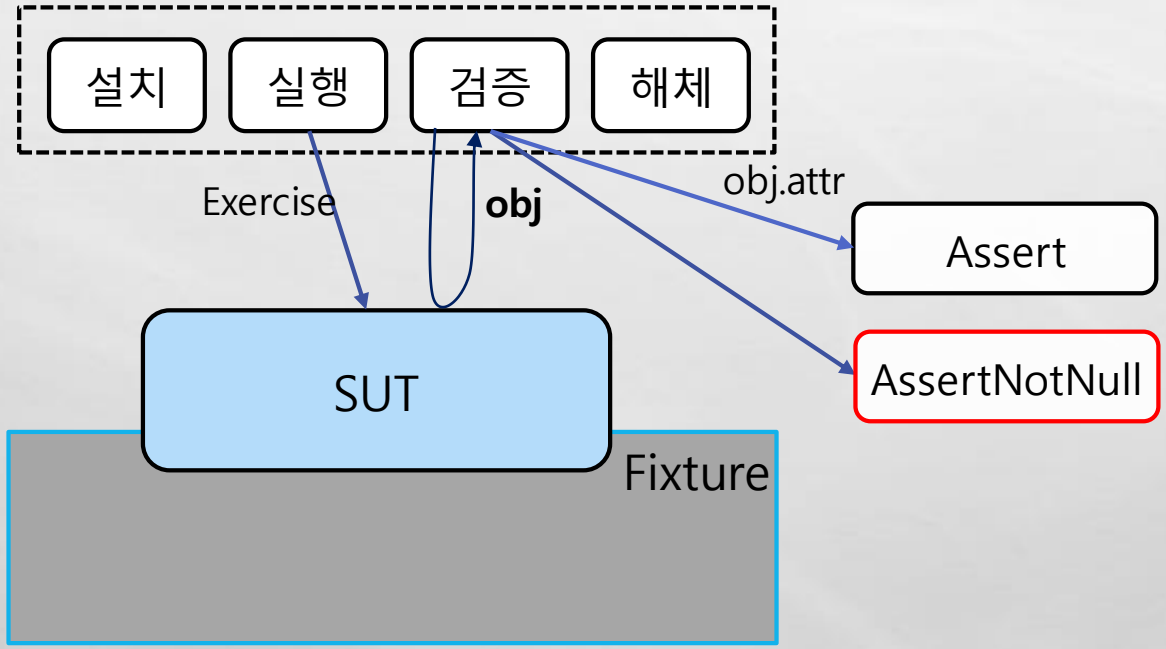
- SUT 실행 전후의 상태 차이에 대한 단언문
  - SUT 실행 전 Fixture 상태에 대한 스냅샷
  - 실행 후 이전 스냅샷과의 비교
    - 원하는 변경이 일어났는지 확인
    - 기대한 객체가 생겼는지 확인



# Verify – Assertion – Guard Assertion



- 테스트 내의 if 조건문을 단언문으로 교체해, 조건을 만족시키지 않으면 테스트 실패
- SUT 가 리턴하는 결과값에 따라 오류가 발생할 만한 코드가 있는 경우
- If 조건절을 사용 => 보호 단언문
- Fixture 가 테스트의 필요상태를 충족하는지  여부 검증.



# Verify – Assertion – Unfinished Test Assertion



- 테스트 작성 중 무조건 실패하는 단언문을 둬서 항상 실패하게 한다.

```
void underWritingTest () {
    // install Fixture
    // Something DO
    // Verify
    fail("Unfinished Test!");
}
```

# Teardown



- Garbage-Collected Teardown : Garbage Collection 에 의한 Fixture 해체
- Automated Teardown : 생성시 객체를 기록 후 자동으로 해체
- In-line Teardown : Test 내에 직접적인 Teardown
- Implicit Teardown

# Teardown – Garbage-collected Teardown



- 프로그래밍 언어에서 제공하는 Garbage Collection 에 의한 Fixture 해체

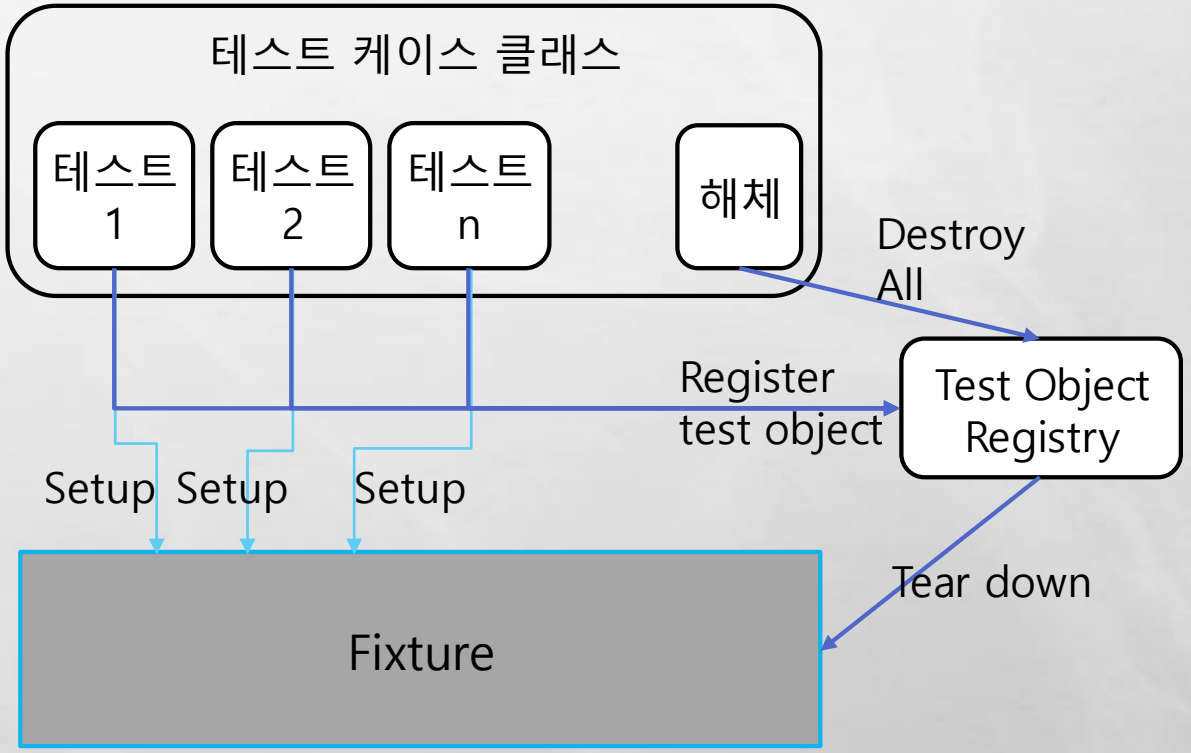
```
Public void testWithGC() {
    Fixture sut = createInlineFixture();
    sut.doMethod();
    assertEquals( expectedResult, sut.result());

    //GC 에 의한 Fixture 해체
}
```

# Teardown – Automated Teardown



- 테스트에서 생성한 모든 자원을 기록, 테스트가 해체될 때 Destroy 한다.
- 객체 정리 도중 하나가 실패해도 나머지 객체의 정리 작업은 계속 진행되게 해 주어야 한다.

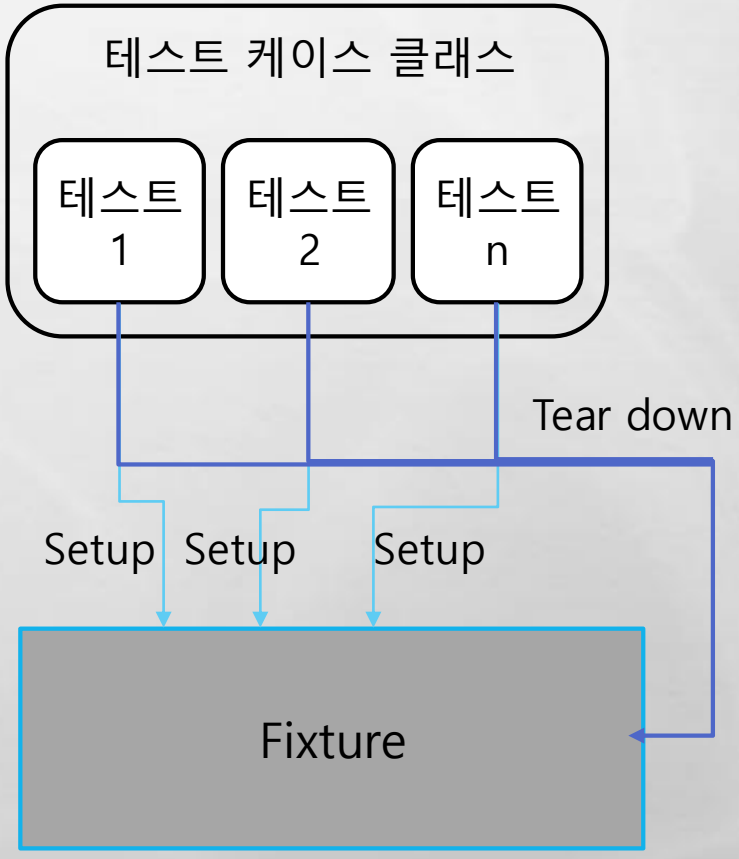
- Automated Fixture Teardown
  - Creation method 에서 생성한 객체 자동 해체
  - Inline Fixture, Shared Fixture
- Automated Exercise Teardown
  - SUT 실행시 생성한 객체에 대한 자동 해체
  - Object Factory를 만들어 객체 생성



# Teardown – Inline Teardown



- 테스트 메소드 내에서 Fixture 해체
- GC 의 대상이 아닌 자원
- Assert fail 이나 Guard assert 로 인해 중단된 테스트에서도 실행되어야 함
- 단순한 Test 생성
- Implicit teardown 으로 가는 중간 절차



# Teardown – Implicit Teardown



- Test framework 이 지원하는 teardown 을 이용

- Test framework에 의해 테스트 성공/실패 여부와 관계없이 항상 실행

# 참고 문헌




- xUnit Test Patterns, Refactoring Test Code, Gerard Meszaros, Addison-Wesley, 2007
- http://xunitpatterns.com/

# THANK YOU.

# Test Smell

# Test smell



- Test smell 의 종류
  - Code smell
    - 애매한 테스트, 테스트 내 조건문 로직, 테스트 하기 힘든 코드, 테스트 코드 중복, 제품 코드 내 테스트 로직
  - Behavior smell
    - 단언 룰렛, 변덕스러운 테스트, 깨지기 쉬운 테스트, 잦은 디버깅, 수동 조정, 느린 테스트
  - Project smell
    - 버그투성이 테스트, 테스트를 작성하지 않는 개발자, 높은 테스트 유지 비용, 제품 버그

# Code smell – Obscure Test



- 한눈에 이해하기 어려운 테스트

- 욕심쟁이 테스트(Eager Test)
  - 한번에 너무 많은 기능을 검증하려는 테스트
  - 하나의 조건만 검증하는 단일 조건 테스트로 만들어라.

- 미스터리한 손님 (Mystery Guest)
  - Fixture 와 Verify 일부가 테스트 Method 밖에 있어 인과 관계가 보이지 않는 테스트
  - 인라인 설치 (in-line Setup)로 Fresh Fixture 사용하라.
  - Shared Fixture나 암묵적 설치를 꼭 써야 할 경우, finder 메소드로 접근하라.

- 일반 Fixture (General Fixture)
  - 필요 이상으로 큰 Fixture 를 사용
  - 최소 Fixture(Minimum Fixture) 사용하라.
  - Shared Fixture를 써야 한다면, 테스트에서 사용하는 모든 리소스의 이름을 고유하게 만들어라.

# Code smell – Obscure Test



- 관련 없는 정보 (Irrelevant Information)
  - 너무 상세한 정보로 인해 SUT 동작에 영향을 미치는 관계 확인이 어려운 테스트
  - 관련 있는 정보만 인자로 받는 생성 메소드 호출로 변경하라.
  - 중요하지 않은 Fixture는 생성 메소드의 기본 값으로 정하거나 더미 객체로 바꿔라.
  - 관련 없는 정보를 결과 검증 로직에서 숨겨주는 custom assertion을 만들어 한 번에 단언하게 하라.

- 하드 코딩된 테스트 데이터 (Hard-Coded Test Data)
  - 데이터 값이 테스트에 하드 코딩되어 입/출력의 인과 관계 확인이 어려운 테스트
  - 리터럴 상수를 생성자 메소드의 기본값으로 지정하거나, Fixture의 초기화한 상수로 바꾸어준다.

# Code smell – Obscure Test



- 간접 테스팅 (Indirect Testing)
  - 검증하려는 객체가 아닌 다른 객체를 통해 SUT를 확인하려는 테스트
  - 프레젠테이션 레이어를 통해 비즈니스로직을 테스트하려는 경우
  - 테스트에서 접근하려는 클래스의 SUT 부분이 private 일 가능성 크다.
  - SUT의 테스트 용이성을 위한 설계를 개선하라.
  - SUT를 간접적으로 접근해야만 한다면, 테스트 유틸리티 메소드 안에 간접 테스팅 로직을 캡슐화하라.

# Code smell – Conditional Test Logic



- Test 내에 실행이 안 될 수도 있는 코드가 있는 테스트

- 유연한 테스트 (Flexible Test)
  - 언제, 어디서 실행하느냐에 따라 다른 기능을 검증하는 테스트
  - SUT 를 의존과 분리하지 못해 테스트 로직을 환경에 맞췄을 가능성 크다.
  - 의존을 테스트 stub 이나 mock 으로 교체하여 분리한다.

- 검증 조건문 로직 (Conditional Verification Logic)
  - 기대 결과를 검증하는데 조건문을 사용하는 테스트
  - 보호 단언문(AssertNot(), Assert())으로 if를 제거한다.
  - 복잡한 객체를 검증하는 조건문은 동등 단언문(AssertEquals()) 또는 맞춤 단언문(custom assertion)으로 바꾼다.
  - 검증 로직의 반복은 맞춤 단언문으로 변경한다.

# Code smell – Conditional Test Logic



- 테스트 내 제품 로직 (Production Logic in Test)
  - Production Logic 과 유사한 Logic을 테스트 내에 쓰는 테스트
  - 원인 : 단일 테스트메소드에서 여러 테스트 조건을 검증하려는 경우
  - SUT 로 테스트하려던 값들을 미리 계산해 열거한 집합으로 테스트한다.

- 복잡한 해체 (Complex Teardown)
  - Teardown 이 너무 복잡한 테스트 – 다른 테스트가 분명한 이유없이 실패하게 만들 가능성
  - 데이터 누수를 만들어 다른 테스트가 실패하게 만들기 쉽다.
  - 해결책 : 암묵적 해체로 변경하거나, 자동 해체를 써야 한다.

# Code smell – Conditional Test Logic



- 여러 테스트 조건 (Multiple Test Conditions)
  - 여러 입력값과 기대값을 테스트 내에 적용하는 테스트
  - 여러 값을 적용[parameterized], 그러나 expected value 에 조건이 있음
  - 실패하는 조건을 만났을 때 더 이상 진행하지 않고 멈춰버려 버그의 결함 국소화를 제공하지 못한다는 문제점이 있다.
    이 문제는 각 테스트 조건에 대해 별개의 테스트 메소드를 만들고 거기에서 인자를 받는 테스트를 호출하게 한다.
  - 메소드 추출로 인자를 받는 테스트를 반복문 안에서 호출하도록 리팩토링하여 가독성을 높일 수 있다.

# Code smell – Hard-to-Test Code



- 코드가 테스트하기 어렵다
  - GUI component, Multi Thread, Highly Coupled to other class.
  - Private constructor, Many other class for parameter
- 강하게 결합된 코드 (Highly Coupled Code)
  - 여러 다른 클래스가 있어야 테스트가 가능한 코드
  - test doubles 을 사용하여 해결.
- 비동기 코드 (Asynchronous Code)
  - 직접적인 method 호출로 테스트 할 수 없는 코드
  - 스레드, 프로세스, 프로그램의 실행시키고 상호작용할 때까지 기다려야 하는 테스트
  - 로직과 비동기 접근 매커니즘의 분리가 중요.
  - 대강 만든 객체(Humble object) 패턴으로 테스트

# Code smell – Hard-to-Test Code



- 테스트할 수 없는 테스트 코드 (Untestable Test Code)
  - 테스트 코드가 너무 복잡하거나, 테스트가 올바른지 확신 할 수 없는 테스트 내 조건문 로직이 있는 경우.
  - 테스트 메소드를 단순화시켜 테스트할 필요 자체를 없애고, 테스트 내 조건문 로직은 테스트 유틸리티 메소드로 바꿔 테스트한다

# Code smell – Test Code Duplication



- 여러 번 반복되는 테스트 코드

- '잘라 붙여 넣기' 코드 재사용 (Cut-and-Paste Code Reuse)
  - 코드를 작성하는 빠른 방법이지만, 유지보수비용 증가
  - 메소드 뽑아내기로 리팩토링
  - 생성메소드나 Finder 메소드, 맞춤 단언문이나 검증 메소드로 추출한다.

- 바퀴 재발명하기 (Reinventing the Wheel)
  - 이미 있는 Test Utility Method 를 다시 작성
  - Test Utility Method 들을 확인하여 적용하기

# Code smell – Test Logic in Production



- 제품 코드에 테스트에서만 실행돼야 하는 코드가 들어있음

- 테스트 훅 (Test Hook)
  - SUT 내에서 실제 코드가 동작할 지, 테스트 전용 로직이 동작할 지를 결정하는 조건문
  - 제품에서 실행돼야 하는 코드는 strategy 객체에 두고 이를 기본으로 설치하게 한 후, 테스트할 때는 Null 객체로 바꾼다.
  - SUT 에서 테스트할 때 추가로 코드를 실행시키려면 strategy 객체를 테스트용 버전으로 교체한다.

- 테스트 전용 (For Tests Only)
  - SUT에 테스트만을 위한 코드, 원인은 테스트 더블, 레거시 코드 테스트
  - 테스트에서 필요한 속성이나 초기화 로직을 SUT 의 테스트용 하위 클래스로 만들어, 테스트에서 접근할 수 있게 한다.
  - 테스트용 하위 클래스로 옮길 수 없다면, 테스트 전용 메소드임을 보여주게 네이밍한다.

# Code smell – Test Logic in Production



- 제품 코드 내 테스트 의존 (Test Dependency in Production)
  - 제품 코드 실행이 테스트 실행에 의존
  - 원인 : 모듈간의 의존을 신경 쓰지 않아서 발생
  - 어떤 제품 코드도 테스트 코드에 의존하지 않게 의존 관계를 신중하게 관리해야 한다.
  - 테스트와 제품 코드에 둘 다 필요한 코드는 제품 모듈에 두고 양쪽에서 모두 접근할 수 있게 해야 한다.

- 동등 오염 (Equality Pollution)
  - SUT의 equals method에 Test 를 위한 동등을 구현하는 경우
  - 테스트용 동등이 필요하다면, equals 메소드를 수정하는 대신 custom assertion 으로 내장 equality assertion 을 쓸 수 있게 한다.
  - 동적 모의 객체 생성 도구를 쓰고 있다면, SUT 의 equals 대신에 비교자(comparator) 를 써야 한다.

# Behavior smell – Assertion Roulette



- Test 내 여러 단언문 중 어디에서 실패가 발생했는지 확인이 어려움

- 욕심쟁이 테스트 (Eager Test)
  - 너무 많은 기능을 검증하는 테스트
  - 욕심쟁이 테스트 메소드를 쪼개어 단일 조건 테스트 스위트로 구성, 메소드 추출
  - 복잡한 Fixture 설치로 인한 경우, 테스트 초반부와 독립적인 테스트 후반부를 Back Door Setup Fixture 설치로 변경한다.

- 빠진 단언 메시지 (Missing Assertion Message)
  - 동일한 단언문을 사용하거나, 단언 메시지를 빼먹은 경우
  - Assert 가 실패하는 경우, 알려주기 위한 Message 가 포함되어 있다.
  - 사용하지 않는 경우 어떤 단언문이 실패했는지 확인이 빠르게 되지 않을 수 있다.
  - IDE 를 활용하거나, 오답 제외 방식의 단언문 제거로 접근하여 찾아내거나, 단언 메소드 호출에 단언 메시지를 넣어준다.

# Behavior smell – Erratic Test



- 어떤 경우에는 통과하고 어떤 경우에는 실패하는 테스트

- 서로 반응하는 테스트 (Interacting Tests)
  - 원인
    - 다른 테스트의 Fixture 설치 단계에서 생성한 Fixture에 의존
    - 다른 테스트의 SUT 실행 단계에서 변경한 SUT에 의존
    - 같이 실행한 두 테스트 사이의 어떤 상호 배타적 실행으로 인한 충돌
  - 테스트끼리의 의존성을 확인하는 빠른 방법 : 테스트 스위트를 다른 순서로 실행해 보는 것이다.
  - 서로 반응하는 테스트 스위트의 경우 주의할 점
    - 서로 반응하는 테스트가 둘 이상일 수 있다
    - 실제 테스트메소드간의 충돌이 아닌 테스트케이스 클래스의 스위트 Fixture 설치나 설치 데코레이터의 충돌 때문일 수 있다.

# Behavior smell – Erratic Test



- 해결책
  - Fresh Fixture 사용
  - shared Fixture를 써야 한다면, 불변 shared Fixture로 테스트들이 Fixture를 변경하는 걸 막는다.
  - 다른 테스트에서 기대했던 객체나 데이터를 생성하지 않아 발생한 의존의 경우, Lazy Setup 으로 양쪽 테스트에서 객체나 데이터를 생성하는 방안 고려

■ 자원 누수 (Resource Leakage)

- 테스트가 점점 느려지다가 어느 순간 실패
- 테스트가 실패할 때 자원을 정리하지 못해 생기는 문제 : 확정 인라인 해체(Guaranteed In-line TearDown) 나 자동 해체를 쓴다
- 자원 풀을 하나로 하면 테스트 누수를 빨리 찾을 수 있다.

# Behavior smell – Erratic Test



- 자원 낙관주의 (Resource Optimism)
  - 외부 자원에 의존하는 테스트의 결과가 언제, 어디서 실행되느냐에 따라 결과가 다름
  - 가능한 Fresh Fixture를 쓴다.
  - 외부 자원을 써야 한다면, 자원을 소스 코드 저장소(SCM)에 저장해 모든 테스트 실행이 같은 환경에서 실행될 수 있게 한다.

- 반복 안 되는 테스트 (Unrepeatable Test)
  - 처음 실행했을 때와 다음에 실행했을 때 다르게 동작하는 테스트.
    실제론 이전 실행된 자신과 상호작용한다.
  - Fresh Fixture 사용
  - 개별 테스트가 끝난 후에 모든 shared Fixture를 전부 제거하여 테스트들을 완전히 격리한다.

# Behavior smell – Erratic Test



- 테스트 실행 전쟁 (Test Run War)
  - 여러 사람이 동시에 테스트를 실행하면 무작위로 실패하는 테스트
  - Fresh Fixture
  - 테스트 실행기마다 별도의 데이터베이스 샌드박스를 주어 테스트간의 실행 전쟁을 막는다.
  - shared Fixture를 통해 서로 반응하는 원인이 완전히 제거할 수 없는 경우,
    테스트에서 Fixture 객체를 변경하려고 할 때마다 새로운 객체를 만들게 해 불변 shared Fixture를 만든다.
  - 어떤 테스트에서 남긴 객체나 데이터베이스 row로 인해 나머지 테스트 Fixture가 오염된 경우,
    매번 테스트가 끝날 때마다 자동 해체하게 한다.

# Behavior smell – Erratic Test

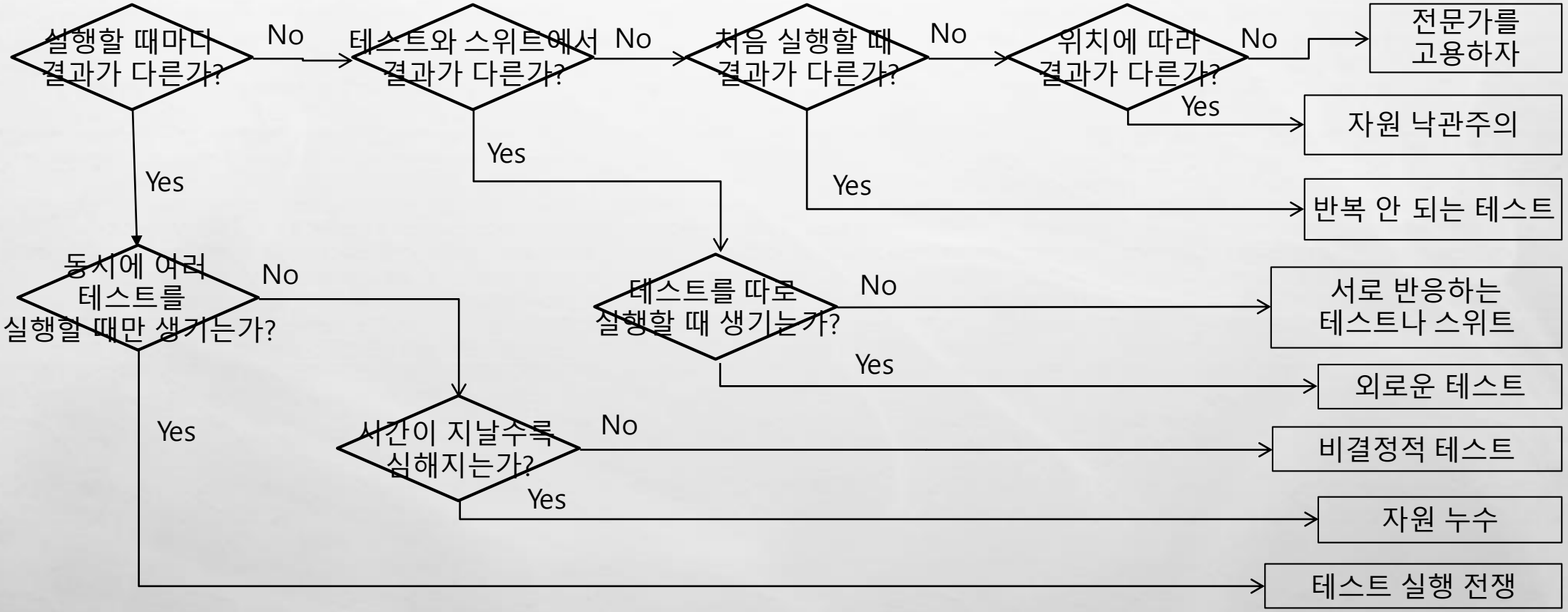


- 비결정적 테스트 (Nondeterministic Test)
  - 하나의 테스트 실행기에서 테스트를 실행시켰는데 무작위로 실패하는 테스트
  - 같은 테스트에서 실행할 때마다 다른 값을 쓰는 경우
    => 무작위 값을 결정적인 값으로 바꾸거나, 입력 값의 최적 집합을 인자로 받는 테스트로 뽑아낸다.
  - 테스트 내 조건문 로직이 있는 경우
    => 테스트 내 조건문 로직을 모두 제거

# Behavior smell – Erratic Test



- 문제 해결 과정



# Behavior smell – Fragile Test



- 인터페이스에 민감함 (Interface Sensitivity)
  - SUT 인터페이스 일부 변경으로 인해 테스트가 컴파일/실행이 안됨(기록 후 재생 테스트 등)
  - 에러를 발생시키는 인터페이스가 자동 테스트와 내부에서만 사용된다면, SUT API 캡슐화하고 인터페이스가 하위 호환성을 지원하게 한다.

- 동작에 민감함 (Behavior Sensitivity)
  - 대상 시스템(SUT)의 동작을 변경했을 때 다른 테스트가 실패
  - Fixture 설치 과정에서 SUT 동작에 대해 잘못된 가정을 하는 코드를 생성 메소드 안으로 캡슐화하여 수정한다.
  - SUT 의 테스트 후 상태에 대한 가정도 custom assertion이나 검증 메소드 안에 캡슐화하여 변경한다.

# Behavior smell – Fragile Test



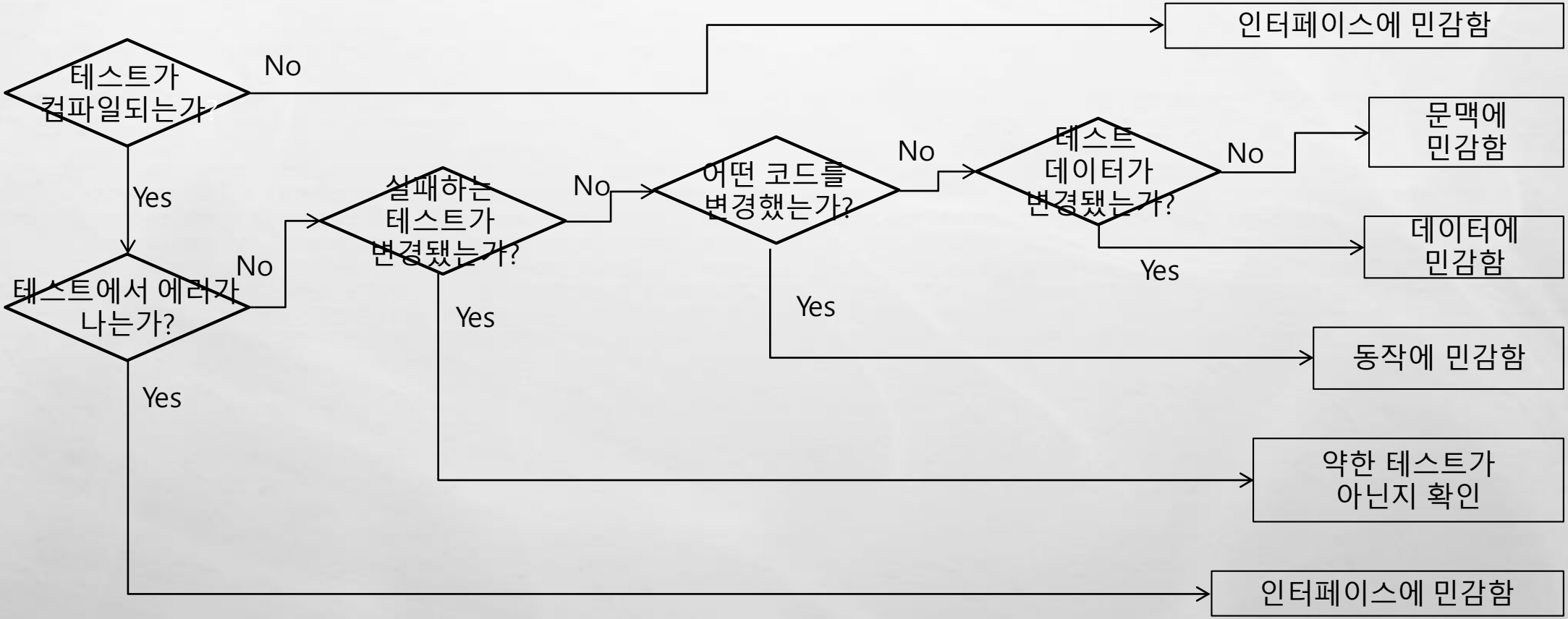
- 데이터에 민감함 (Data Sensitivity)
  - SUT를 테스트할때 사용하는 데이터가 변경된 경우 테스트가 실패
  - 주로 테스트가 데이터베이스 변경시 발생
  - 테스트를 데이터와 독립적으로 만들 수 있게 Fresh Fixture를 쓴다.
  - Fresh Fixture를 쓰는 게 불가능하다면, 데이터 베이스 분할 스키마를 사용하여 어떤 테스트에서 변경된 데이터가 다른 테스트에서는 건드리지 않게 한다.
  - 데이터가 제대로 변경되었는 지 검증한다

- 문맥에 민감함 (Context Sensitivity)
  - SUT가 실행되는 문맥의 상태나 동작이 변경된 경우 테스트가 실패
  - 근본 원인 : 시간이나 날짜에 의존 관계, SUT에서 의존하는 코드나 시스템 동작의 변경
  - 과하게 명세화된 소프트웨어 (Overspecified Software)
  - 민감한 동등 (Sensitive Equality)
  - 깨지기 쉬운 Fixture (Fragile Fixture)
  - 의존적인 입력값을 테스트 스텁으로 제어한다.

# Behavior smell – Fragile Test



- 깨지기 쉬운 테스트 문제 해결 과정



# Behavior smell – Frequent Debugging



- 테스트 실패의 원인을 찾으려면 직접 디버깅을 해야 한다.
  - Test Runner 의 출력만으로 문제를 파악하기 어려울 때
- Defect localization 부족
  - 문제를 드러낼 수 있는 테스트 작성
- Infrequently Run Test
  - Baby step 마다 자동화된 테스트를 실행
- Untested Requirement
  - 자동 테스트에서 발견하지 못한 문제를 수동테스트에 발견
  - Chained Defect
- 해결책 또는 문제를 드러내는 테스트 작성, TDD

# Behavior smell – Manual Intervention



- 실행할 때마다 수작업을 해야 하는 테스트

- 수동 Fixture 설치 (Manual Fixture Setup)
  - 서버 설치, 서버 프로세스 실행, Prebuilt Fixture 설치 등 자동테스트를 위한 수동 테스트 환경

- 수동 결과 검증 (Manual Result Verification)
  - 테스트 검증을 위해 수작업 필요한 경우

- 수동 이벤트 발생 (Manual Event Injection)
  - 테스트 진행을 위해 수동으로 어떤 작업 해야 하는 경우

- 해결책 : 자동으로 테스트할 수 있는 방법을 찾아라.

# Behavior smell – Slow Test



- 테스트 실행에 시간이 너무 오래 걸리는 경우

- 느린 컴포넌트 사용
  - SUT의 어떤 컴포넌트 응답이 느린 경우 -> Test double

- 일반 Fixture
  - 전체적으로 균일하게 느린 경우 -> Minimal Fixture, Shared Fixture

- 비동기 테스트
  - 명시적인 지연이 있는 테스트 코드 -> testable component

- 너무 많은 테스트
  - 테스트가 너무 많은 경우 -> 적절히 분할한 테스트

# Project smell – Buggy Test



- 자동 테스트에서 버그가 많은 경우
- 깨지기 쉬운 테스트 (Fragile Test)
- 애매한 테스트 (Obscure Test) : 문서로서의 테스트 검토
- 테스트하기 힘든 코드(Hard to Test Code)

- 이유 : 충분한 시간이 주어지지 않았다.
- 해결책
  - 테스트를 적절하게 작성하는 방법 배우기
  - 레거시 코드를 자동으로 테스트하기 쉽고 견고하게 만들기 위해 리팩토링
  - 테스트 먼저 작성하기

# Project smell – Developers Not Writing Test



- 개발자들이 테스트를 작성하지 않는 경우
- 부족한 시간 (Not Enough Time)
  - Code writing + Debugging = Test writing + Code Writing + Testing + Debugging
- 테스트하기 힘든 코드 (Hard to Test Code)
- 잘못된 테스트 자동화 전략 (Wrong Test Automation Strategy)
  - Test 환경/ 자동화 전략으로 인해 테스트가 Fragile Test나 Obscure Test가 되는 경우
- 개발자와 관리자 모두가 관심을 가져야 하는 문제

# Project smell – High Test Maintenance Cost



- 기존 테스트를 유지 보수 하기가 너무 어렵다.
- 깨지기 쉬운 테스트 (Fragile Test)
- 애매한 테스트 (Obscure Test)
- 테스트 하기 힘든 코드 (Hard to Test Code)

- 문제 해결을 위한 조언
  - 관리자는 개발팀에게 테스트 유지 보수에는 얼마나 걸리는 지,
    얼마나 자주 테스트를 유지 보수해야 하는 지,
    왜 유지 보수를 꼭 해야 하는 지 물어보고,
  - 높은 테스트 유지 보수 비용이 발생하지 않는 좋은 방법을 찾도록 독려한다.

  - 관리자는 고객에게 전달할 새로운 기능을 좀 줄여 개발 속도를 조절하고,
    테스트 리팩토링 스토리 기간을 만들어 시간을 벌어야 한다.

# Project smell – Production Bug



- 공식 테스트나 제품에서 버그가 많다
- 드문 테스트 실행 (Infrequently Run Tests)
  - Slow Test 등으로 인해 테스트를 자주 실행하지 않음
- 놓친 테스트 (Lost test)
  - 의도치 않게 제외된 테스트
- 빠진 단위 테스트 (Missing Unit Test)
  - 단위 테스트는 성공, 그러나 고객 테스트는 실패
- 테스트 안 된 코드 (Untested Code)
  - 어떤 테스트에서도 실행되지 않는 SUT 코드
- 테스트 안 된 요구 사항 (Untested Requirement)
  - 어떤 기능을 테스트하려고 하지만 공개 interface 로 보여지는 부분이 없다.
- 절대 실패하지 않는 테스트 (Neverfail Test)
  - 잘못 작성된 단언문, 비동기 테스트에서 Fail detect 불가

# 참고 문헌




- xUnit Test Patternls, Refactoring Test Code, Gerard Meszaros, Addison-Wesley, 2007
- http://xunitpatterns.com/

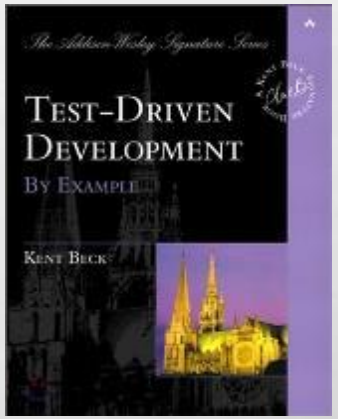
# THANK YOU.

# 별첨

# Test-Driven Development



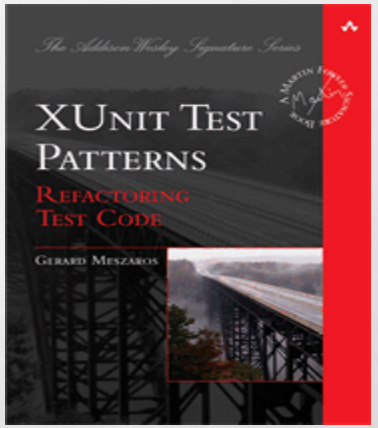
- pubulished on 2002 by kent back
- Example 을 통하여 TDD 를 실천하는 방법을 설명
- 화폐예제
- xUnit



# xUnit Test Patterns



- " xUnit Test Patterns, Refactoring Test Code" Gerard Meszaros,Addison-Wesley, 2007
- http://xunitpatterns.com/




- Opportunities to re-think the design process with Test Driven Development.
- Same as "Design pattern" for testing