

# Unit Test

2023-1 KNU SCSE Software Testing Theory

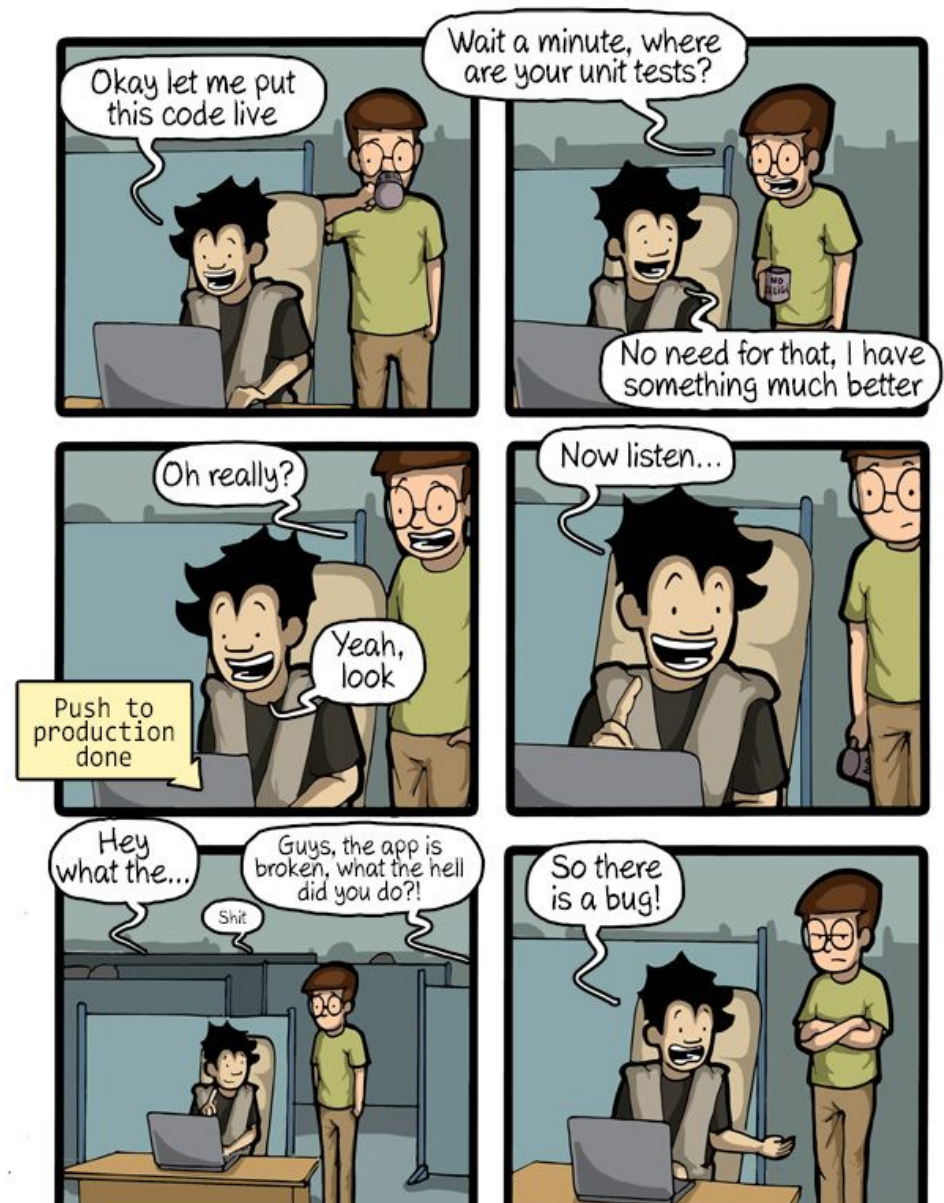
# Contents

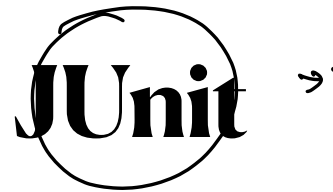
- Unit test using JUnit
- Test Driven Development (TDD)

# Unit Test

## Using JUnit

(Contents of this slides are borrowed from the lecture slides of Introduction to Software Testing by Ammann & Offutt )





- Open source Java testing framework used to write and run repeatable automated tests
- JUnit is open source (junit.org)
- A structure for writing test drivers
- JUnit features include:
  - Assertions for testing expected results
  - Test features for sharing common test data
  - Test suites for easily organizing and running tests
  - Graphical and textual test runners
- JUnit is widely used in industry
- JUnit can be used as stand alone Java programs (from the command line) or within an IDE such as Eclipse

# JUnit Test

JUnit ?  
↑  
Integration test → Unit들 간의 서비스가 잘 동작하는지.  
System test → 구현하고 통합된 소프트웨어를  
실제적으로 테스트.

- JUnit can be used to test ...
  - ... an entire object
  - ... part of an object – a method or some interacting methods
  - ... interaction between several objects
- It is primarily intended for unit and integration testing, not system testing.
- Each test is embedded into one test method
- A test class contains one or more test methods
- Test classes include :
  - A collection of test methods
  - Methods to set up the state before and update the state after each test and before and after all tests
- Get started at [junit.org](http://junit.org)

# Writing Tests for JUnit



- Need to use the methods of the `junit.framework.assert` class
  - javadoc gives a complete description of its capabilities
- Each test method checks a condition (assertion) and reports to the test runner whether the test failed or succeeded  
*test case 1개 → 여러 개 test case를 넣을 수 있지만 권장 X.*
- The test runner uses the result to report to the user (in command line mode) or update the display (in an IDE)
- All of the methods return void
- A few representative methods of `junit.framework.assert`
  - `assertTrue (boolean)`
  - `assertTrue (String, boolean)`
  - `fail (String)`  
*↓ print 해라.*  
*↓*  
*무조건 fail 해라.*

# JUnit Test Fixtures

↓  
고정물.

- A test fixture is the state of the test
  - Objects and variables that are used by more than one test
  - Initializations (prefix values)
  - Reset values (postfix values) → test 끝났고 initial 값으로 돌려 놓음.
- Different tests can use the objects without sharing the state
- Objects used in test fixtures should be declared as instance variables → static 변수 안됨.
- They should be initialized in a @Before method
- Can be deallocated or reset in an @After method

# Example

## JUnit4

```
public class Calc
{
    static public int add (int a, int b)
    {
        return a + b;
    }
}
```

Printed if  
assert fails

Expected  
output

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalcTest
{
    @Test public void testAdd()
    {
        assertTrue("Calc sum incorrect",
            5 == Calc.add (2, 3));
    }
}
```

Test  
values



# Testing Min Class

```
import java.util.*;
```

```
public class Min
```

```
{
```

```
/**
```

```
 * Returns the minimum element in a list
```

```
 * @param list Comparable list of elements to search
```

```
 * @return the minimum element in the list
```

```
 * @throws NullPointerException if list is null or  
 * if any list elements are null
```

```
 * @throws ClassCastException if list elements are  
not mutually comparable
```

```
 * @throws IllegalArgumentException if list is empty
```

```
 */
```

```
 ...
```

```
}
```

```
public static <T extends Comparable<? super T>> T min  
(List<? extends T> list)
```

```
{
```

```
 if (list.size() == 0)
```

```
 {
```

```
   throw new IllegalArgumentException ("Min.min");
```

```
 }
```

```
 Iterator<? extends T> itr = list.iterator();
```

```
 T result = itr.next();
```

```
 if (result == null) throw new NullPointerException  
 ("Min.min");
```

```
 while (itr.hasNext())
```

```
 { // throws NPE, CCE as needed
```

```
   T comp = itr.next();
```

```
   if (comp.compareTo (result) < 0)
```

```
   {
```

```
     result = comp;
```

```
   } }
```

```
 return result;
```

```
 }
```

주요 7개의 테스트 케이스

→ 7개의 테스트 케이스

list가 null 이거나  
list element가 null

→ 이 경우  
NullPointerException  
를 던진다

→ 이 경우  
IllegalArgumentException  
를 던진다

happy path

→ 이 경우  
정상적으로  
실행된다

# MinTest Class

Standard imports for all JUnit classes

```
import static org.junit.Assert.*;
import org.junit.*;
import java.util.*;
```

Test fixture and pre-test setup method (prefix) :

```
private List<String> list; // Test fixture

// Set up - Called before every test method.
@Before
public void setUp()
{
    list = new ArrayList<String>();
}
```

Post test teardown method (postfix) :

```
// Tear down - Called after every test method.
@After
public void tearDown()
{
    list = null; // redundant in this example
}
```

# Min Test Cases: NullPointerException

```
@Test public void testForNullList()
```

```
{  
    list = null; ← 여기 선언  
    try {  
        Min.min (list);  
    } catch (NullPointerException e) {  
        return;  
    }  
    fail ("NullPointerException expected");  
}
```

Using fail assertion

여기 가리 실행되었으면 이 테스트 케이스는 실패해야  
NullPointerException이 발생하지  
않은 것임

Catching an easily overlooked special case

NullPointerException test

```
@Test (expected = NullPointerException.class)  
public void testForNullElement()  
{  
    list.add (null);  
    list.add ("cat");  
    Min.min (list);  
}
```

실패 하지 success

```
@Test (expected = NullPointerException.class)  
public void testForSoloNullElement()  
{  
    list.add (null);  
    Min.min (list);  
}
```

# More Exception Test Cases for Min

```
@Test (expected = ClassCastException.class)
@SuppressWarnings ("unchecked")
public void testMutuallyIncomparable()
{
    List list = new ArrayList();
    list.add ("cat");
    list.add ("dog");
    list.add (1);
    Min.min (list);
}
```

Handwritten red annotations: A red arrow points to the `ClassCastException.class` in the `@Test` annotation. A red circle is drawn around the `list.add (1);` line, with the handwritten text "비 호환" (incompatible) next to it.

Java generics do not prevent clients from using raw types

```
@Test (expected = IllegalArgumentException.class)
public void testEmptyList()
{
    Min.min (list);
}
```

Special case: Testing for the empty list

# Remaining Test Cases for Min

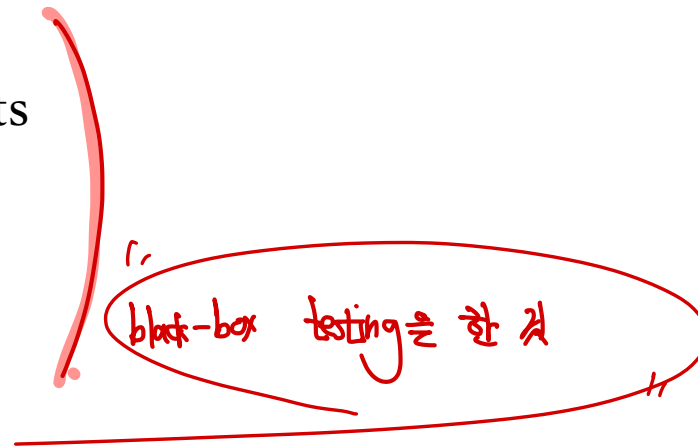
```
@Test
public void testSingleElement()
{
    list.add ("cat");
    Object obj = Min.min (list);
    assertTrue ("Single Element List", obj.equals ("cat"));
}

@Test
public void testDoubleElement()
{
    list.add ("dog");
    list.add ("cat");
    Object obj = Min.min (list);
    assertTrue ("Double Element List", obj.equals ("cat"));
}
```

Finally! A couple of  
“Happy Path” tests

# Summary: Seven Tests for Min

- Five tests with exceptions
  1. null list
  2. null element with multiple elements
  3. null single element
  4. incomparable types
  5. empty elements
- Two without exceptions
  6. single element
  7. two elements



tc 100 → test method 100+!

## Data-Driven Tests

- We want to avoid testing a function multiple times with similar values
  - e.g) Adding two numbers
- Data-driven unit tests call a constructor for each collection of test values
  - Same tests are then run on each set of data values
  - Collection of data values defined by method tagged with @Parameters annotation

# Date-Driven Unit Test

① parameterized.class

이

int a, b, sum  
은 2개의 배열 형태로 ...

```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;
```

import:

```
@RunWith (Parameterized.class)
public class DataDrivenCalcTest
{ public int a, b, sum; → 이 파라미터를 받는다.
```

Constructor is  
called for each  
triple of values

Test 1  
Test values: 1, 1  
Expected: 2

Test 2  
Test values: 2, 3  
Expected: 5

생성

```
public DataDrivenCalcTest (int v1, int v2, int expected)
{ this.a = v1; this.b = v2; this.sum = expected; }
```

constructor.

```
@Parameters public static Collection<Object[]> parameters()
{ return Arrays.asList (new Object [][] {{1, 1, 2}, {2, 3, 5}}); }
```

Test method

```
@Test public void additionTest()
{ assertTrue ("Addition Test", sum == Calc.add (a, b)); }
}
```

가장 처음 실행



parameter 등으로는 값들의  
일의성 보장만으로  
테스트를 통과할 수 있다.

# Test with Parameters: JUnit Theories

- Unit tests can have actual parameters
  - So far, we've only seen parameterless test methods
- Contract model: Assume, Act, Assert
  - Assumptions (preconditions) limit values appropriately
  - Action performs activity under scrutiny
  - Assertions (postconditions) check result

값을 쓸 수 있게 함

true라고 가정하겠다.

```
@Theory public void removeThenAddDoesNotChangeSet (  
    Set<String> someSet, String str) {           // Parameters!  
    assumeTrue (someSet != null)                 // Assume  
    assumeTrue (someSet.contains (str));          // Assume  
    Set<String> copy = new HashSet<String>(someSet); // Act  
    copy.remove (str); → 제거한다. 다시 추가  
    copy.add (str);  
    assertTrue (someSet.equals (copy));          // Assert  
}
```

# Where Do the Data Values Come From?

- All combinations of values from @DataPoints annotations where assume clause is true
- Four (of nine) combinations in this particular case
- Note: @DataPoints format is an array

*array 형태*

```
@DataPoints
public static String[] animals = {"ant", "bat", "cat"};

@DataPoints
public static Set[] animalSets = {
    new HashSet (Arrays.asList ("ant", "bat")),
    new HashSet (Arrays.asList ("bat", "cat", "dog", "elephant")),
    new HashSet (Arrays.asList ("Snap", "Crackle", "Pop"))
};
```

*assumption을 만족하는 대상들 (또는 조합) parameterized tests를 진행하겠단다.*

*Nine combinations of animalSets[i].contains (animals[j]) is false for five combinations*

*5개의 조합은 재의를 해버린다.*

*3중 for loop 조합한 test!*

# JUnit Theories Need BoilerPlate

```
import org.junit.*;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;
import static org.junit.Assume.*;

import org.junit.experimental.theories.DataPoint;
import org.junit.experimental.theories.DataPoints;
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;

import java.util.*;

@RunWith (Theories.class)
public class SetTheoryTest
{
    ... // See Earlier Slides
}
```

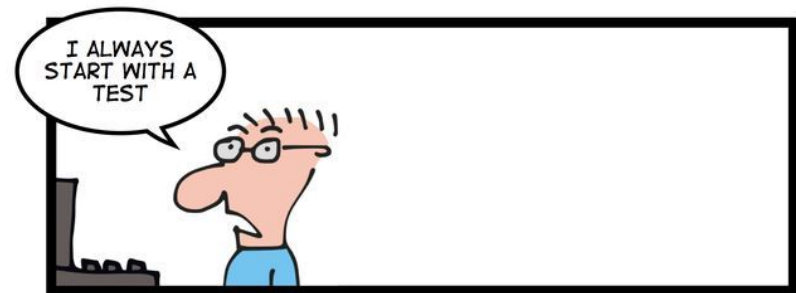
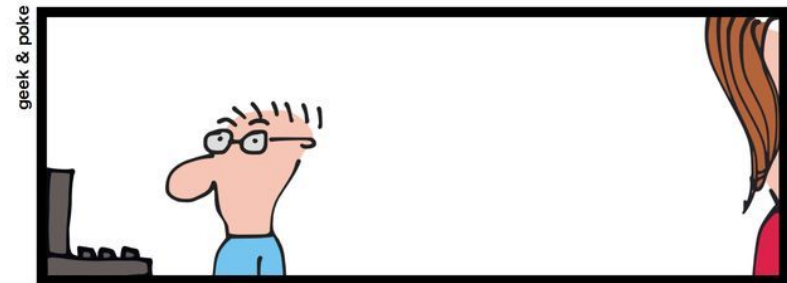
# JUnit Resources

- Some JUnit tutorials
  - <http://open.ncsu.edu/se/tutorials/junit/> (Laurie Williams, Drigh Ho, and Sarah Smith )
  - <http://www.laliluna.de/eclipse-junit-testing-tutorial.html> (Sascha Wolski and Sebastian Hennebrueder)
  - <http://www.diasparsoftware.com/template.php?content=jUnitStarterGuide> (Diaspar software)
  - <http://www.clarkware.com/articles/JUnitPrimer.html> (Clarkware consulting)
- JUnit: Download, Documentation
  - <http://www.junit.org/>

# Test Driven Development

TDD

↓  
part.



TDD

Agile → (테스트 프로그래밍  
TDD, CI, Refactoring, ...)  
→ Incrementally.

# Test-driven development

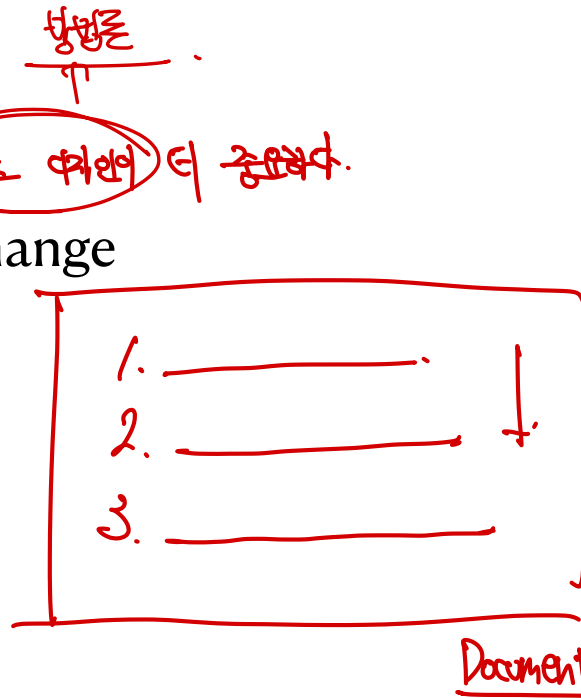
fail.

하루에 한 번씩 실패를 반복하여  
개선하는 방법

- TDD was introduced as part of agile methods such as Extreme Programming.
- Tests are written before code and 'passing' the tests is the critical driver of development.
- You develop code incrementally, along with a test for that increment.
- You don't move on to the next increment until the code that you have developed passes its test

# A perspective on TDD

- TDD is mostly about design 테스트 케이스가 중요하다.
- Gives confidence and enables change
- It is Documentation By Example
- Provides rapid feedback on
  - Quality of implementation
  - Quality of design



변형이 더 쉬움.  
문제를 미리 형태로 추론할 수 있음.  
즉각적인 피드백 가능함.

Requirements의 역할.

• It isn't the only testing you'll need to do

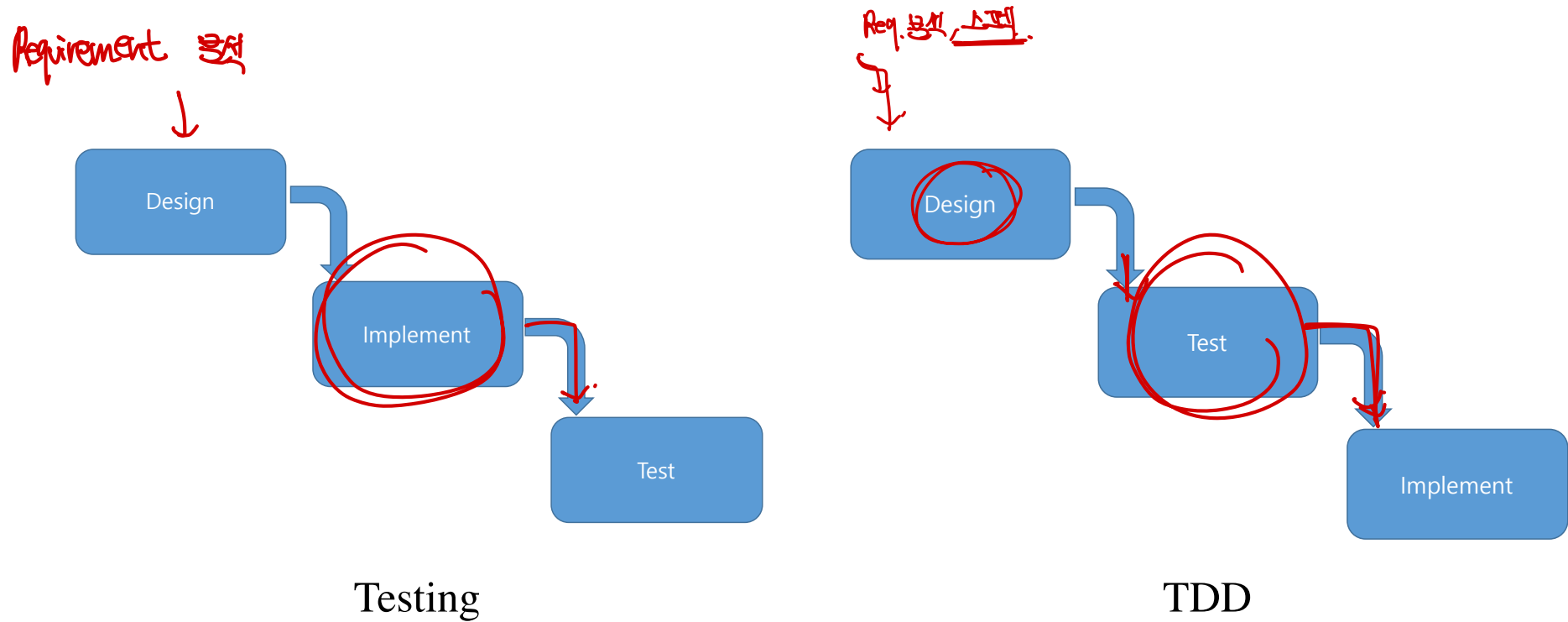
• Developers write tests

↓  
test designer가 요구 사항 설계.

↳ 다 개발 완료되면 Integration test

↳ system test도 수행해야 함.

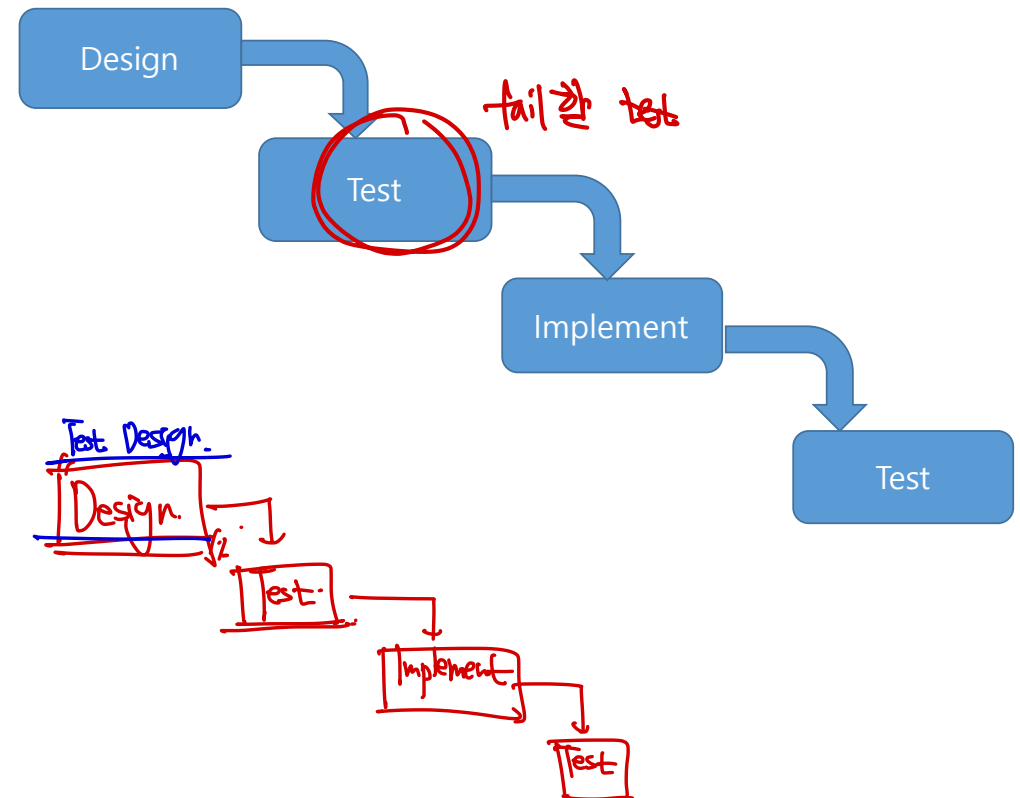
# Testing vs. TDD



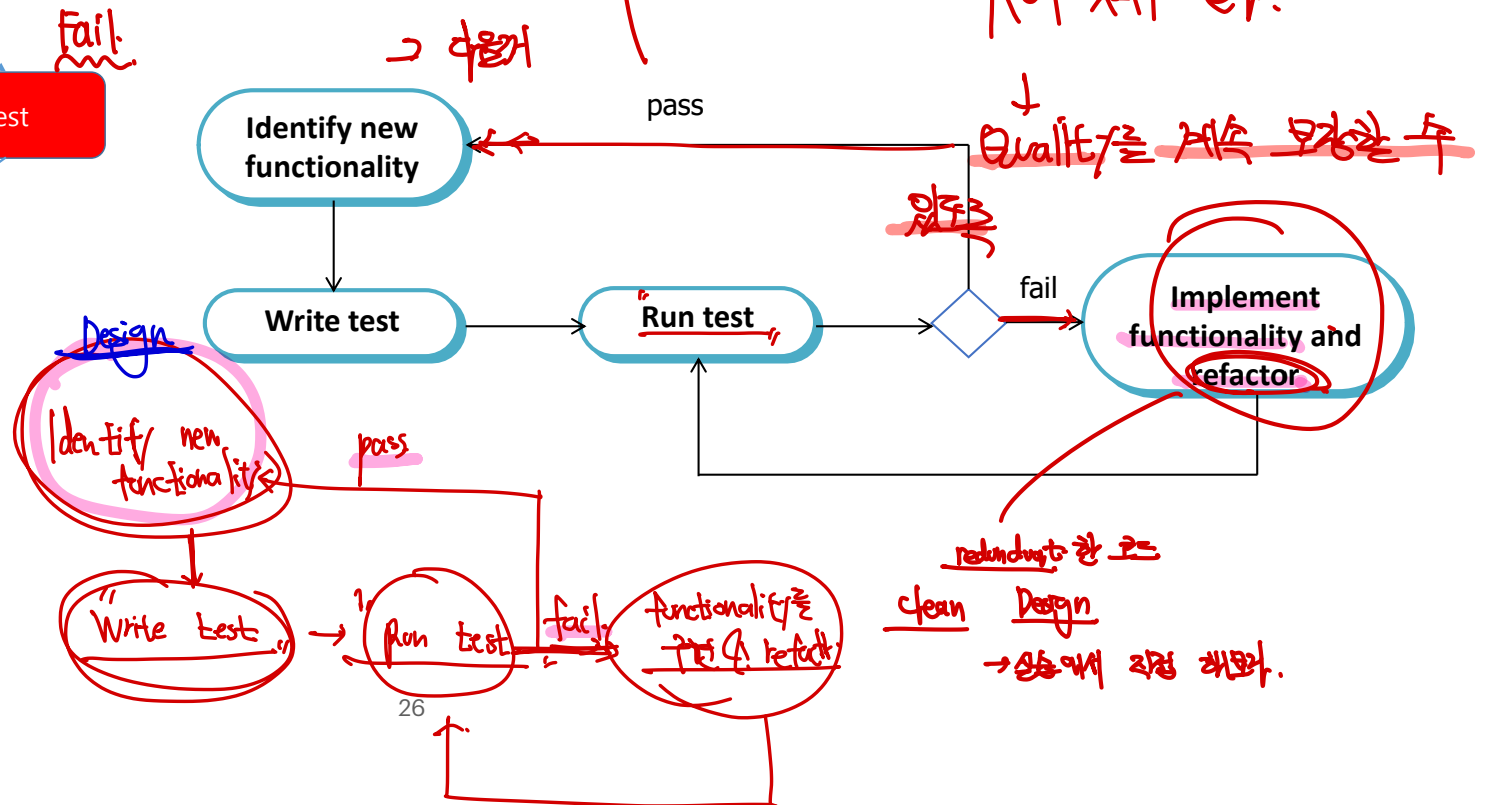
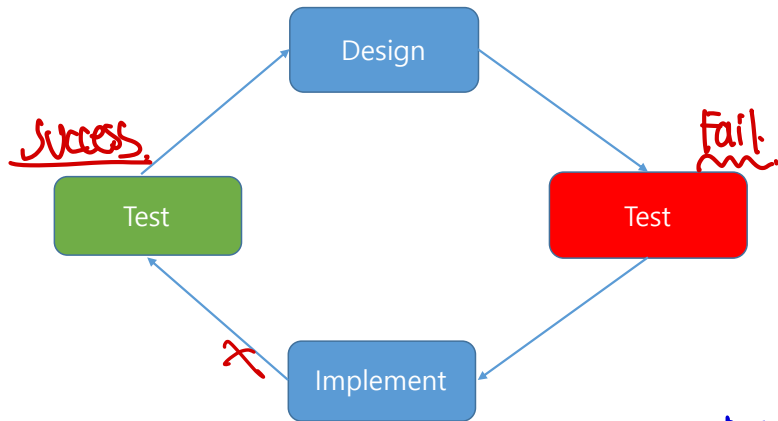


# TDD

- Design: figure out what you want to do
- Test: write a test to express the design
  - It should fail
- Implement: write the code
- Test again
  - It should pass



# TDD Process



# TDD example 주식

- To do: We need to be able to add amounts in two different currencies and convert the result given a set of exchange rates
- Test-first

```
public void testMultiplication(){
    Dollar five = new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
}
```

getAmount:  
 → Dollar class는 값이 5를 가졌다  
 하고 times 메서드를 제공한다.

Instrument	Shares	Price	Total
IBM	1000	25	25000
GE	400	100	40000
		Total	65000

Instrument	Shares	Price	Total
IBM	1000	25 USD	25000 USD
Novartis	400	150 CHF	60000 CHF
		Total	65000 USD

From	To	Rate
CHF	USD	1.5

이러는 테스트  
 환율 정보 받아서 계산  
 할 수 있는 테스트 코드

# Exercise (~~AB~~)

← test.  
AA

# Benefits of TDD

→ coverage 측정을 할 수 있지만. 무조건 X.

## • Code coverage

- Every code segment that you write has at least one associated test so all code written has at least one test.

모든 코드는 하나의 test와 연관되어 있다.

## • Regression testing

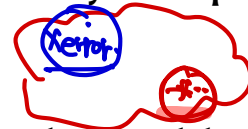
회귀 테스트



이전의 tests를 통과해야.

- A regression test suite is developed incrementally as a program is developed.

## • Simplified debugging



- When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.

1. : pass 2. pass 3. pass 4. fail

→ 구현중인 functionality를  
확인 해야 한다.

## • System documentation

- The tests themselves are a form of documentation that describe what the code should be doing

# Summary

- Unit test is a prerequisite of integration test and system test
  - It is no matter which development method is used (Waterfall, Agile, MDD, etc.)
- Unit test is usually performed by programmers themselves
- Automation is a must for efficient unit test
  - Partial support is already available through xUnit → 제공되는 Framework.
- TDD and Agile are heavily dependent on unit test