

# TDD with JUnit

## Test-Driven Development (TDD)

- TDD Life Cycle
- TDD vs DLP (Debug Later Programming)
- Why TDD is matter
- Unit Testing with F.I.R.S.T
- Code and Test Coverage
- Structure of good unit testing (GUT)

## Unit Testing with JUnit

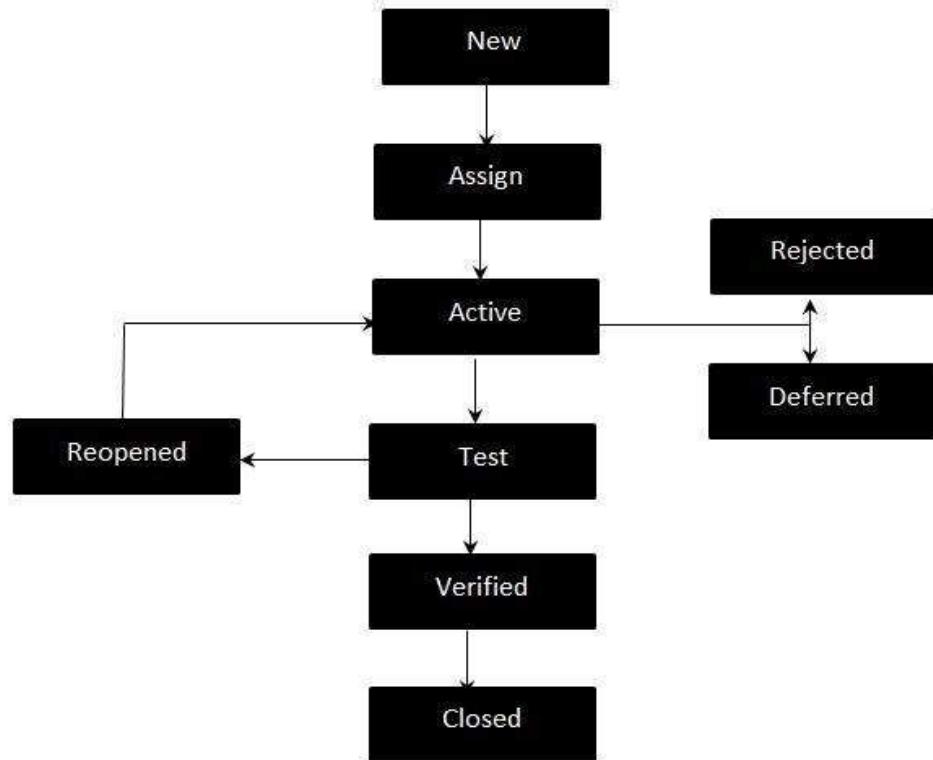
- JUnit lifeCycle
- Assertion
- Data-Driven Test with JUnit
- JUnit features
- Timeout
- Conditional
- Category
- Suite
- Running testing

## • Test Double

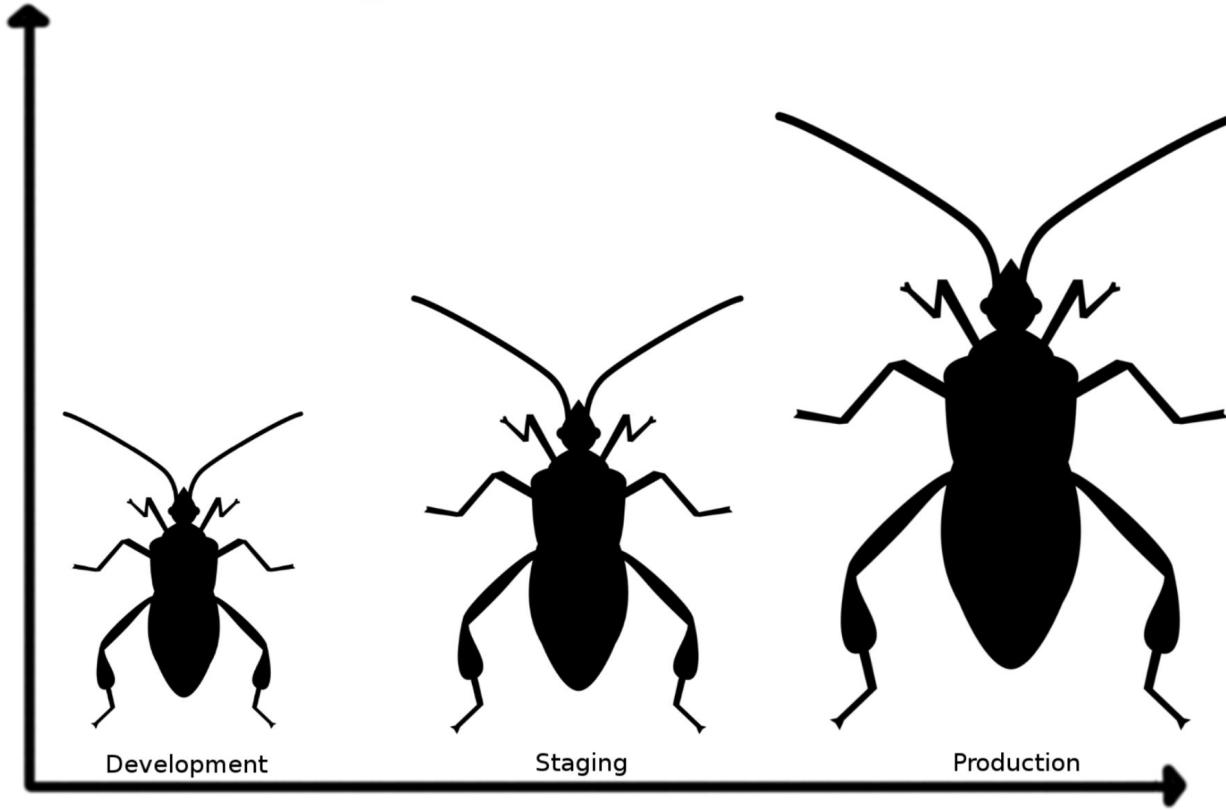
- Dummy
- Stub
- Spy
- Mock
- Fake

# Defect

A **defect** is a bug or an error created in the application. A programmer, while designing and building the software, can make mistakes or errors. These errors mean that there are flaws in the software. These are called defects.



Cost to fix a bug



Stage when a bug is found

# Technical Excellence



ARCHITECTURE  
& DESIGN



CLEAN CODE



# TDD

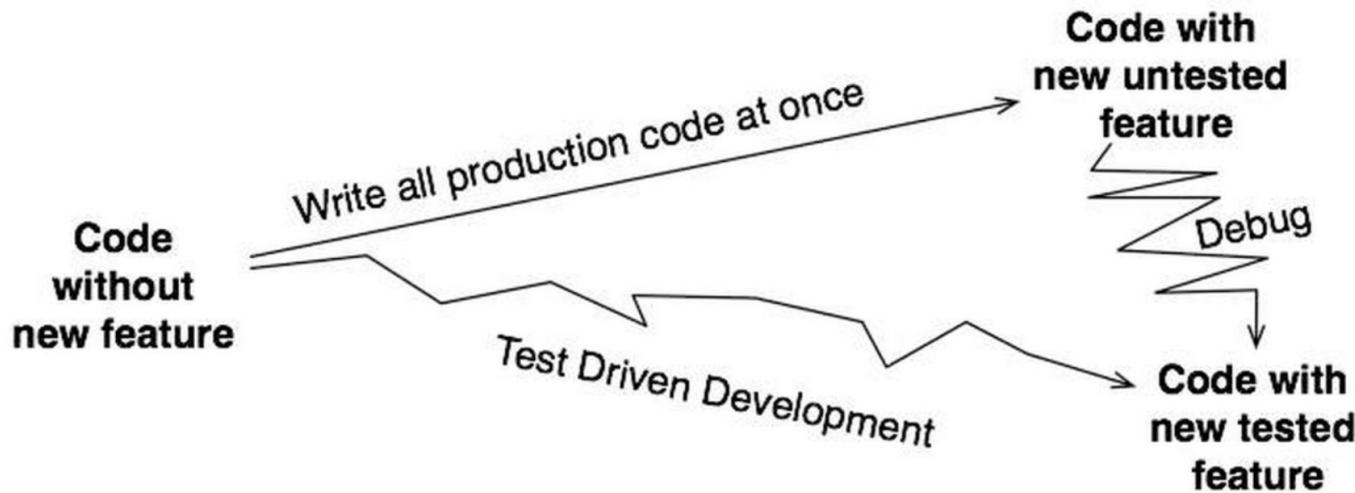
**ALL CODE IS GUILTY  
UNTIL PROVEN INNOCENT**

<https://www.freecodecamp.org/news/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2/>

# TDD

## Test-Driven-Development Test-Driven Design

# TDD vs DLP (Debug Later)

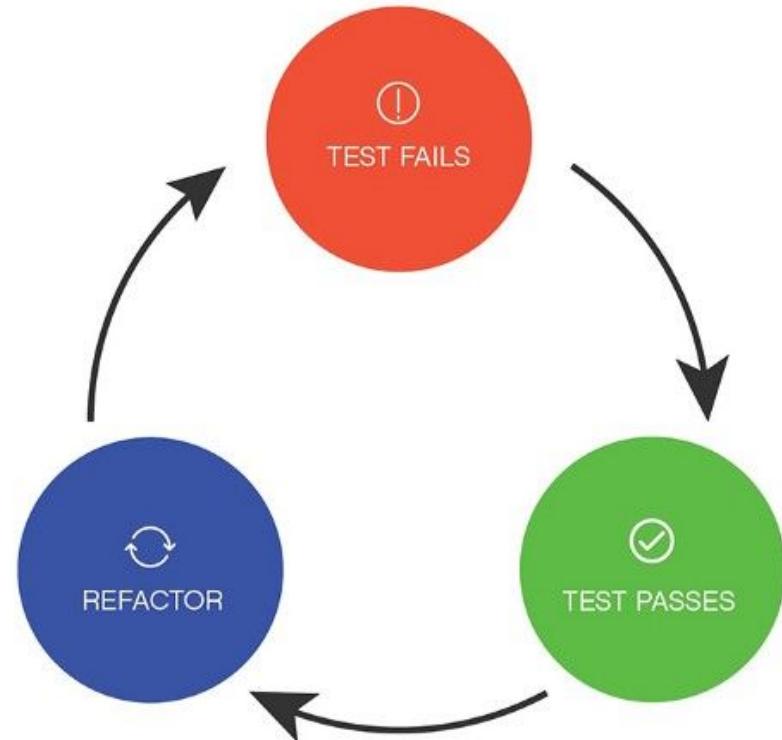


# What Is Test-Driven Development

Test-driven development is a development style that drives the design by tests developed in short cycles of:

1. Write one test.
2. Implement just enough code to make it pass.
3. Refactor the code so it is clean.

TDD Cycle



Red, Green, Refactor

# Red Green Refactor



# A TDD cycle Should Be ...

## Short

The turnaround time for passing each test is short. It could take 5 mins per cycle.

## Rhythmic

You'll feel the rhythm distinctly - "red, green, refactor... red, green refactor..."

## Incremental

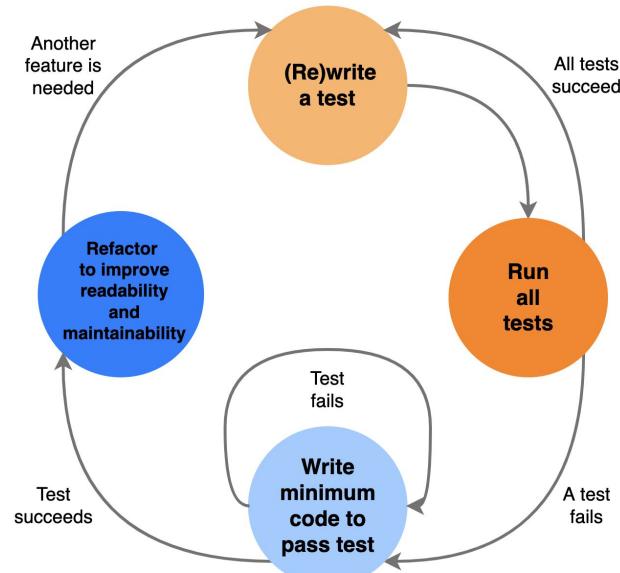
You'll know that as you write and pass more tests, working functionalities are being build up incrementally.

## Design-focused

With good knowledge of software design principles, you'll discover TDD is not a testing technique but a method of designing software.

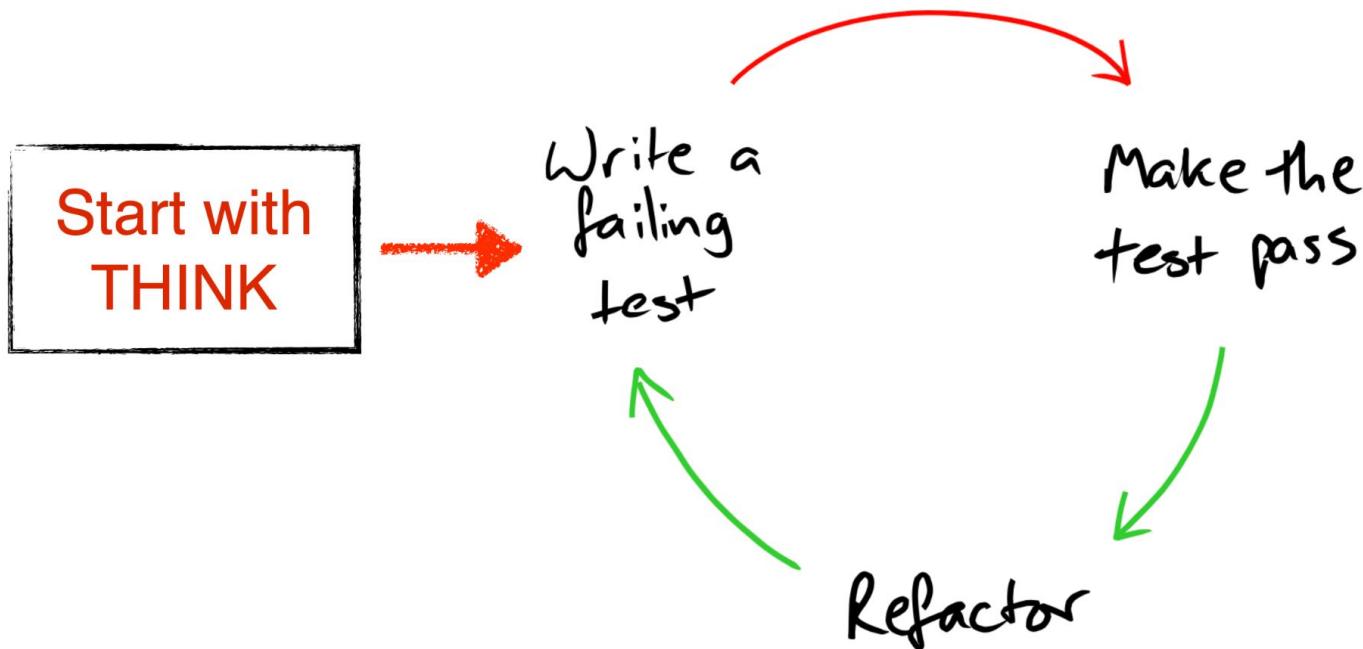
## Disciplined

TDD is a different way of developing software. To break the old habit of "code and fix" and to adopt a new habit will require discipline and persistence.

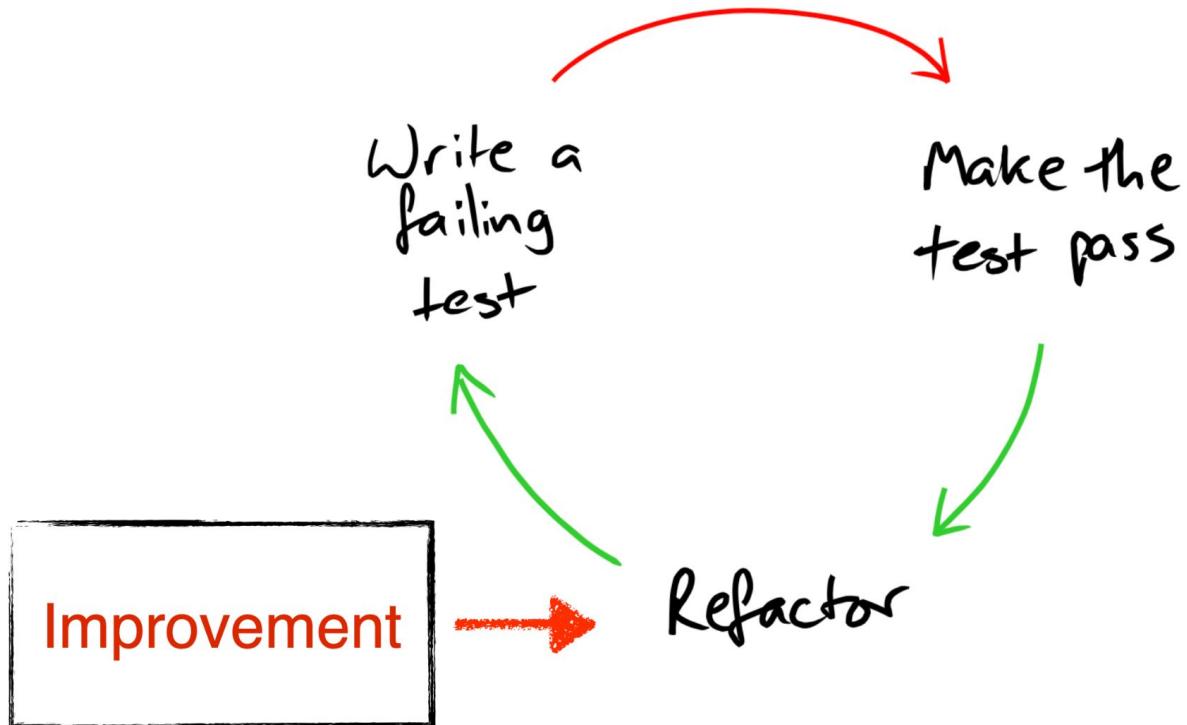


The cycle of test-driven development

# Improve TDD Cycle



# Improve TDD Cycle



# Code Smell

## Code Smells

- What? How can code "smell"??
- Well it doesn't have a nose... but it definitely can stink!

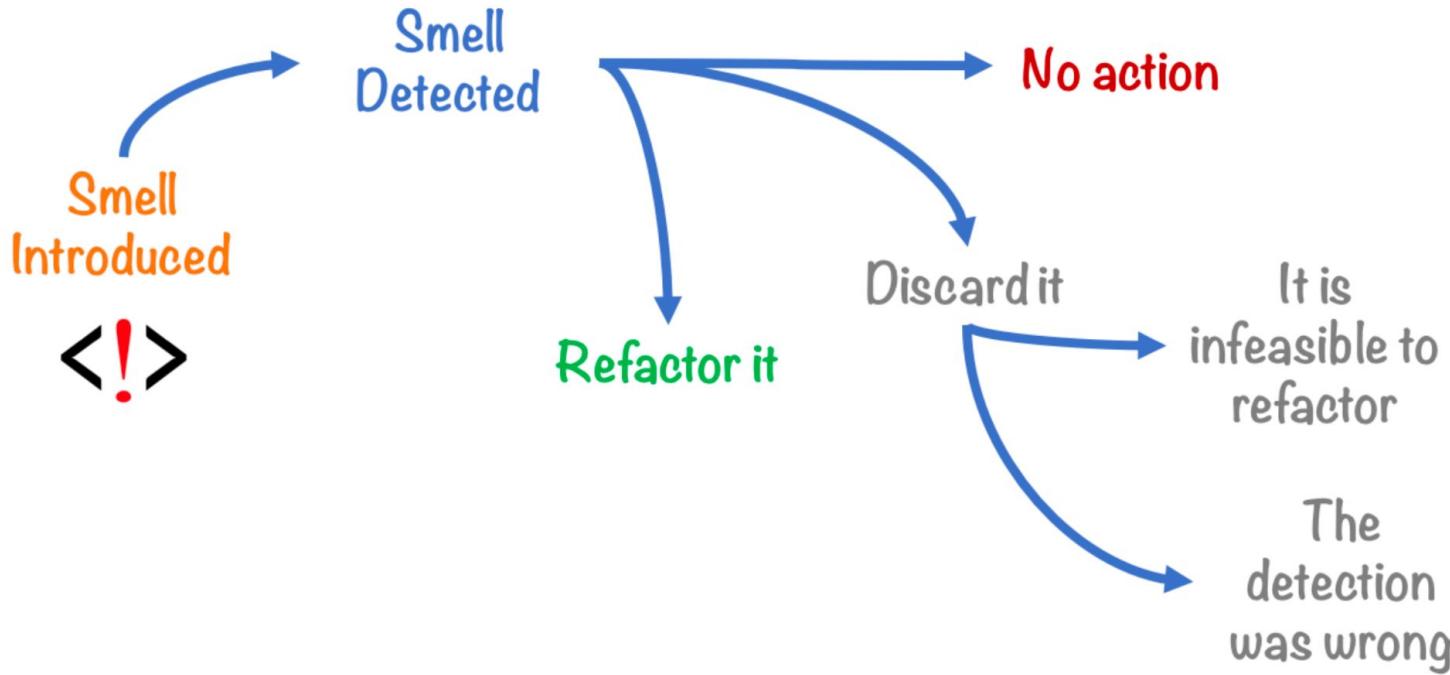


### Bloaters

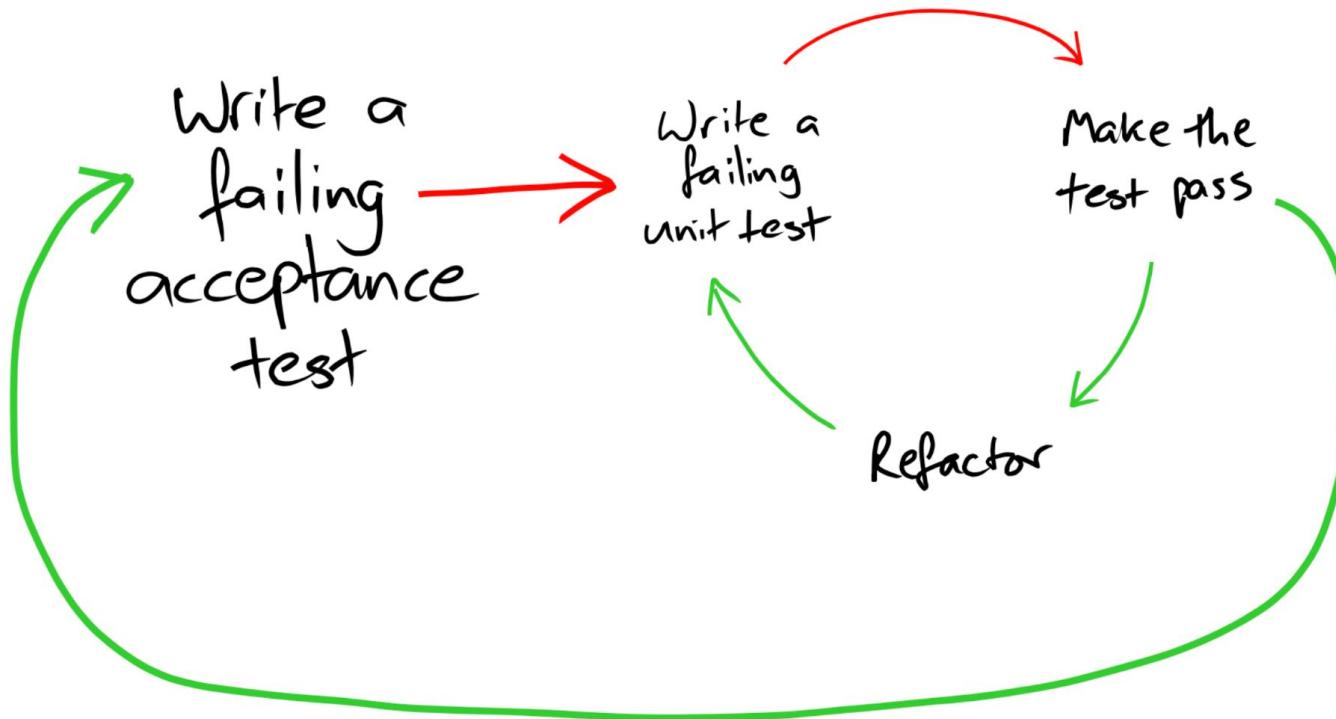
Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

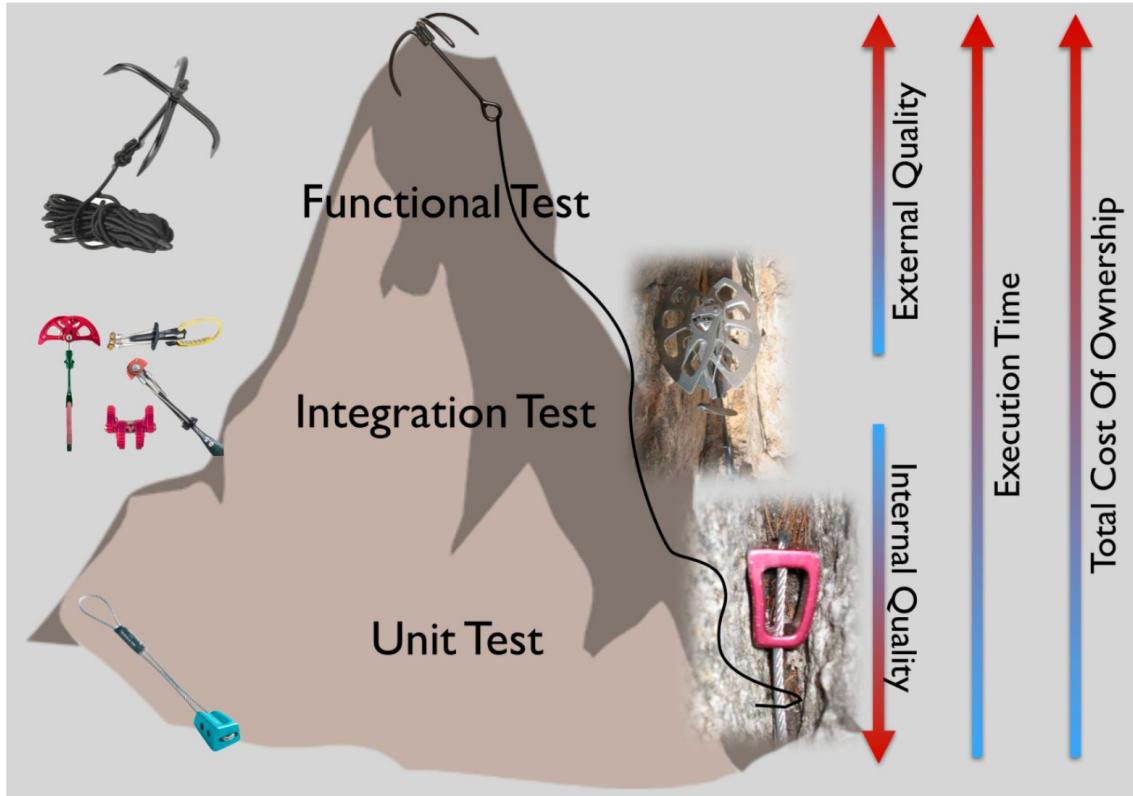
# Code Smell



# Acceptance tests

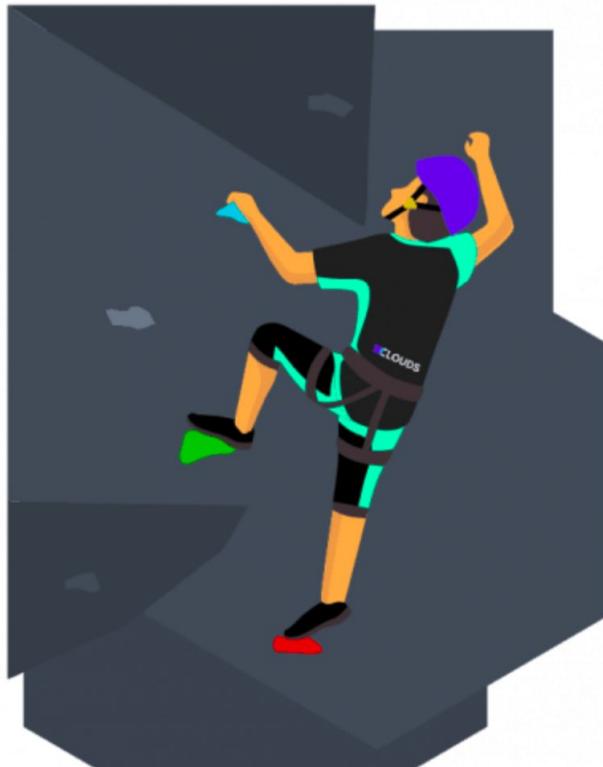


# The Golden rule of TDD



<https://less.works/jp/less/technical-excellence/unit-testing.html>

# The Golden rule of TDD



# Discipline #1

You are not allowed to write production code until you have first written a failing unit test.

# Discipline #2

You are not allowed to write more of a unit test than is sufficient to fail, and not compiling is failing.

# Discipline #3

You are not allowed to write more production code than is sufficient to pass the currently failing test.

# The Golden rule of TDD

“Never write a line of code  
without a failing test”

- *Kent Beck* -

# Make the test pass !!

Fake or hard code  
Triangulation  
Professional code

# What Is Unit Test?

**Unit Tests** are software programs written to exercise other software programs (called **Code Under Test**, or **Production Code**) with **specific** preconditions and verify the **expected behaviours** of the CUT.

Unit tests are usually written in the same programming language as their code under test.

Each **unit test** should be small and test only limited piece of code functionality. Test cases are often grouped into **Test Groups** or **Test Suites**. There are many open source **unit test frameworks**. The popular ones usually follow an **xUnit** pattern invented by Kent Beck, for example, **JUnit** for Java and CppUTest for C/C++.

Unit tests should also run very fast. Usually, we expect to **run hundreds of unit test cases within a few seconds**.

# Good Unit Tests (F.I.R.S.T)

Fast

Independent/Isolate

Repeat

Self-validate

Thorough/Timely

# Good Unit Test Patterns

A good pattern to follow in a unit test is **“AAA”**: **Arrange**, **Act** and **Assert**.

The AAA protocol is a recommended approach for structuring unit tests. As a unit testing best practice, it improves your test’s readability by giving it a logical flow. AAA is sometimes referred to as the “Given/When/Then” protocol.

You can use the AAA protocol to structure your unit tests with the following steps:

- **Arrange**: Arrange the setup and initialization for the test
- **Act**: Act on the unit for the given test
- **Assert**: Assert or verify the outcome

## Arrange, Act, and Assert (AAA)

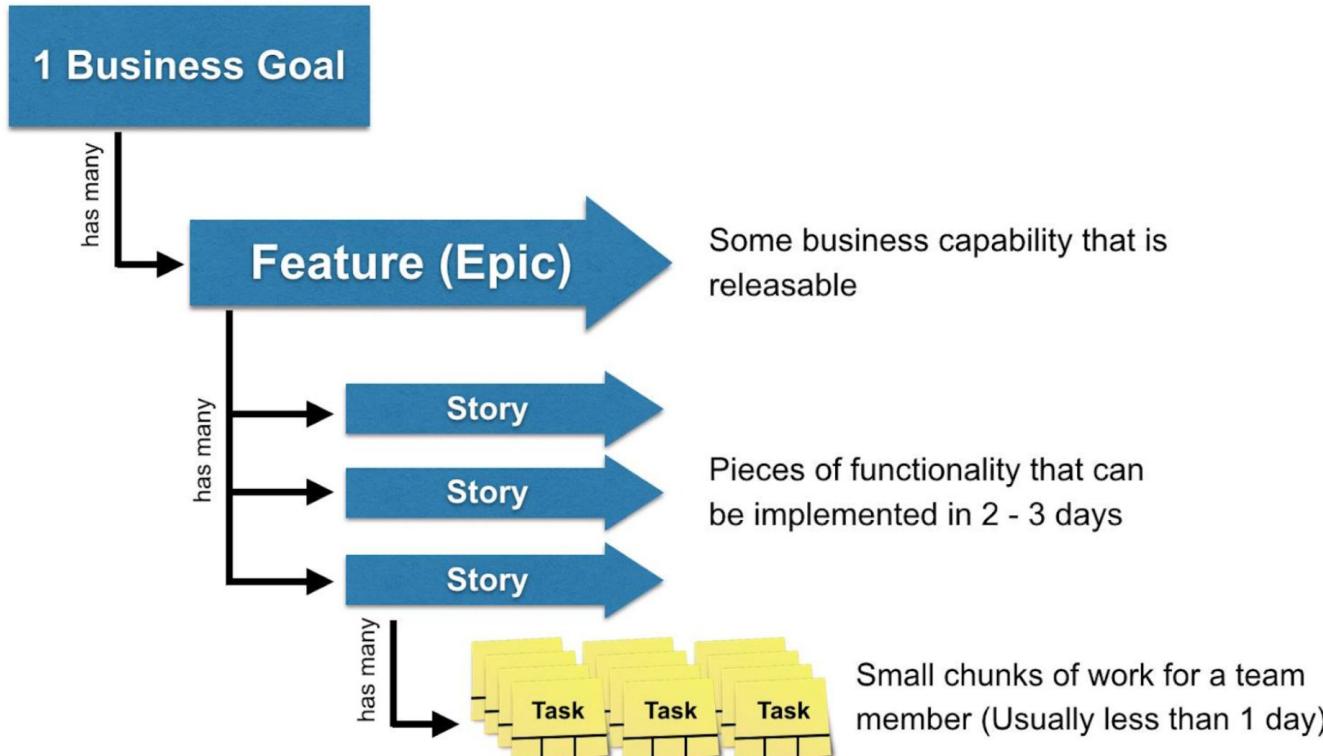
```
def test_abs_for_a_negative_number():

    # Arrange
    negative = -2

    # Act
    answer = abs(negative)

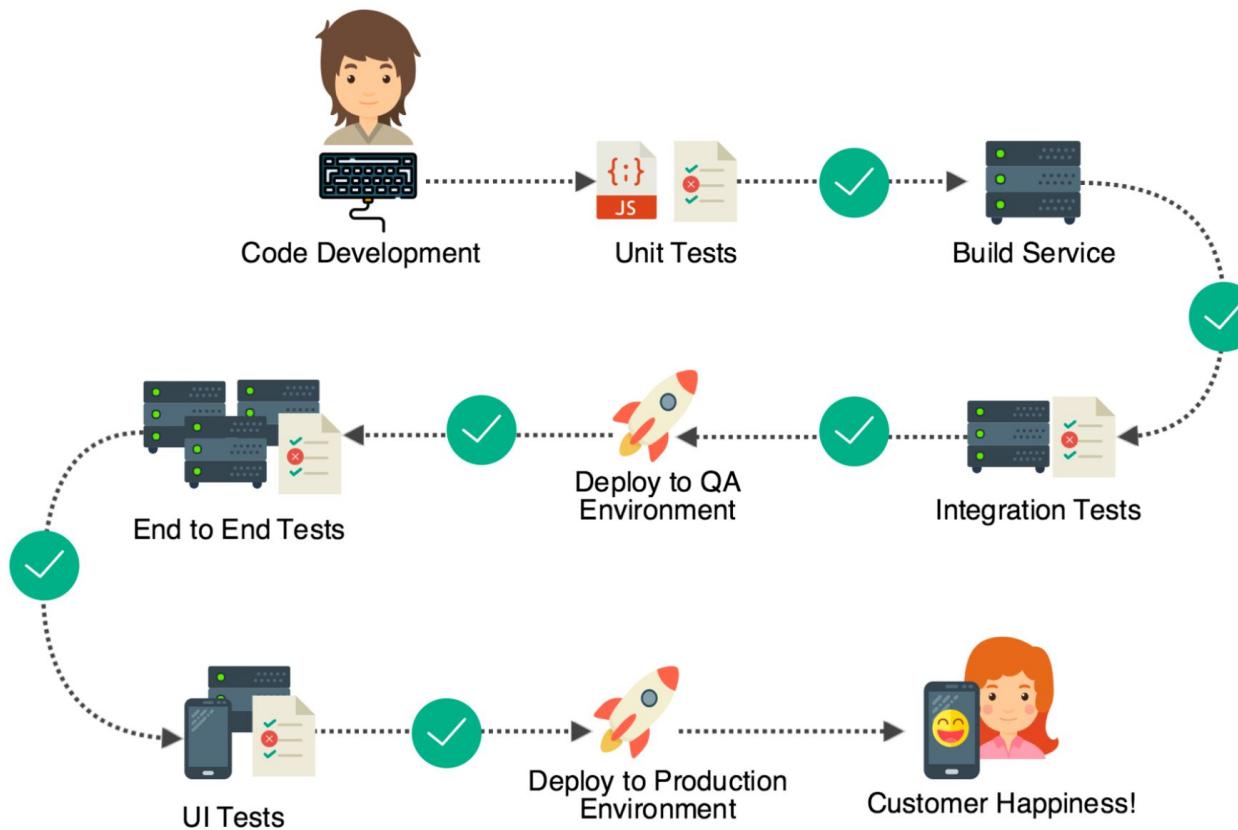
    # Assert
    assert answer == 2
```

# Start with Business Goal to Tasks



Source: The Whole Team Approach to Agile Testing by Janet Gregory and Lisa Crispin

# Delivery process



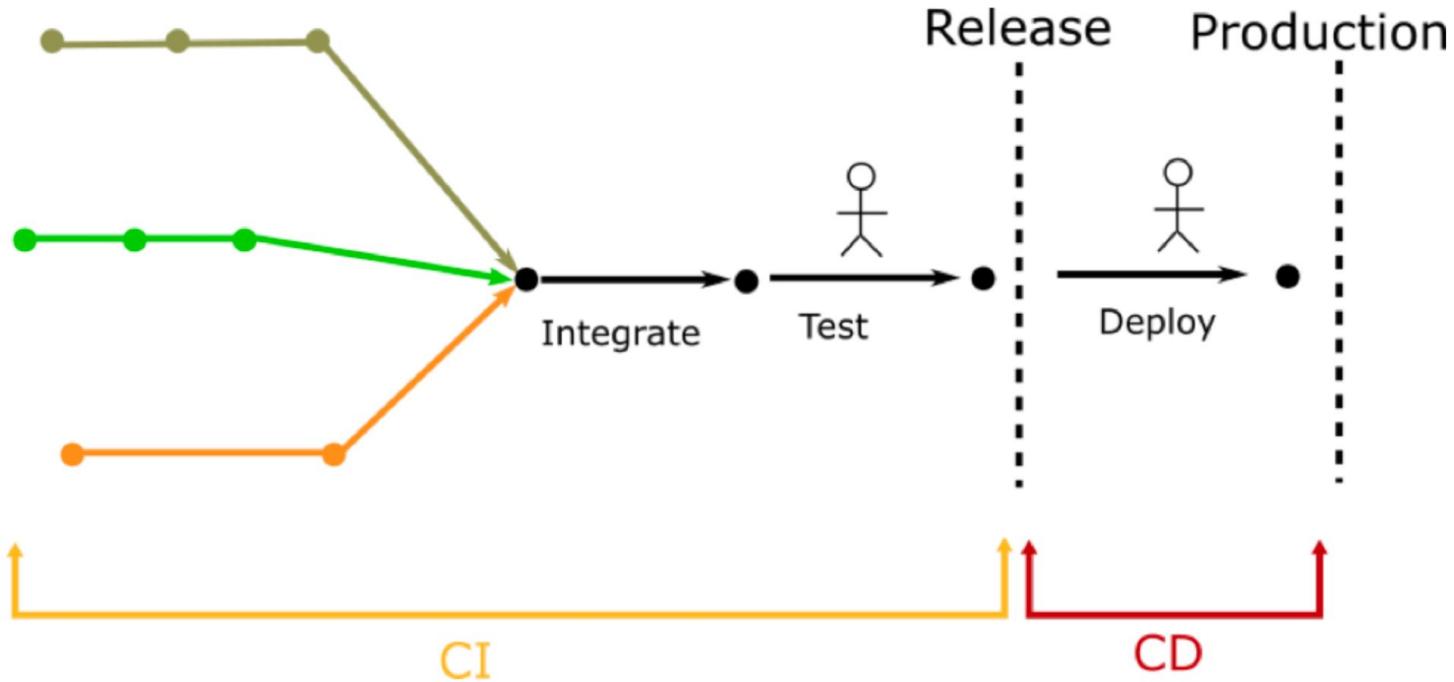
# TDD in Iteration development

Iteration cycle 2-4 weeks

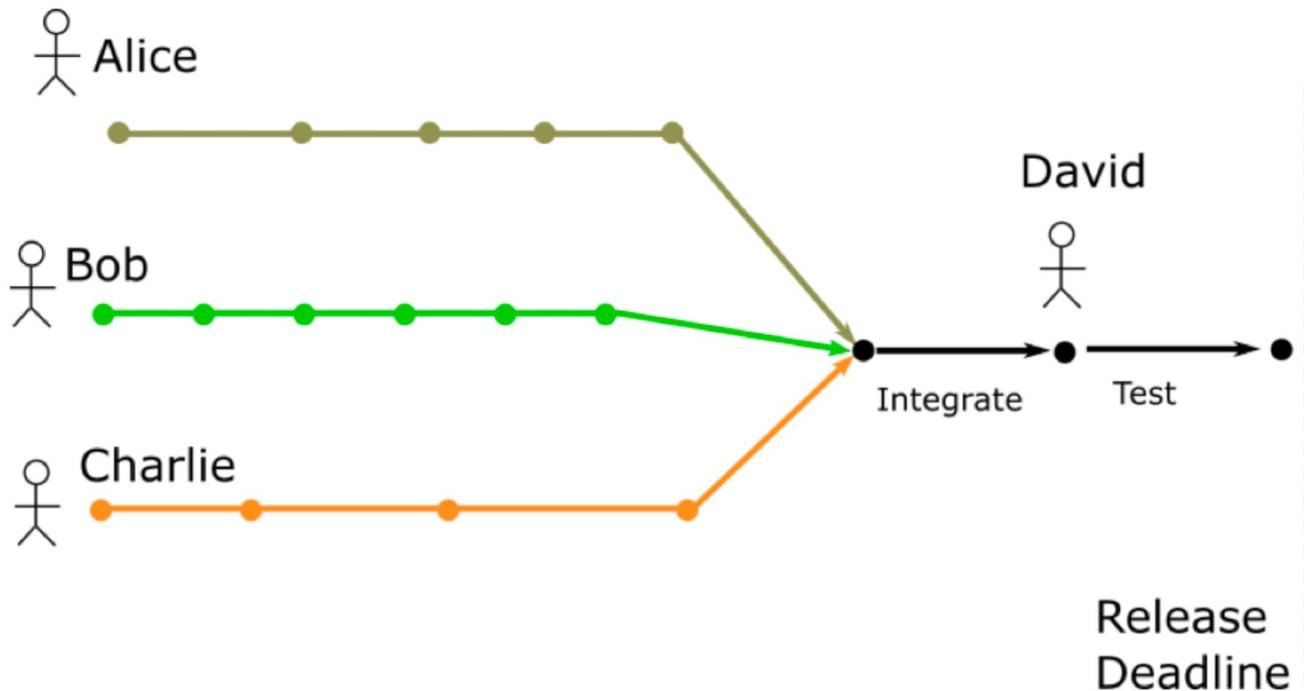
Continuous integration cycle  
(multiple times a day)

TDD cycle in a few minutes (3-10 minutes)

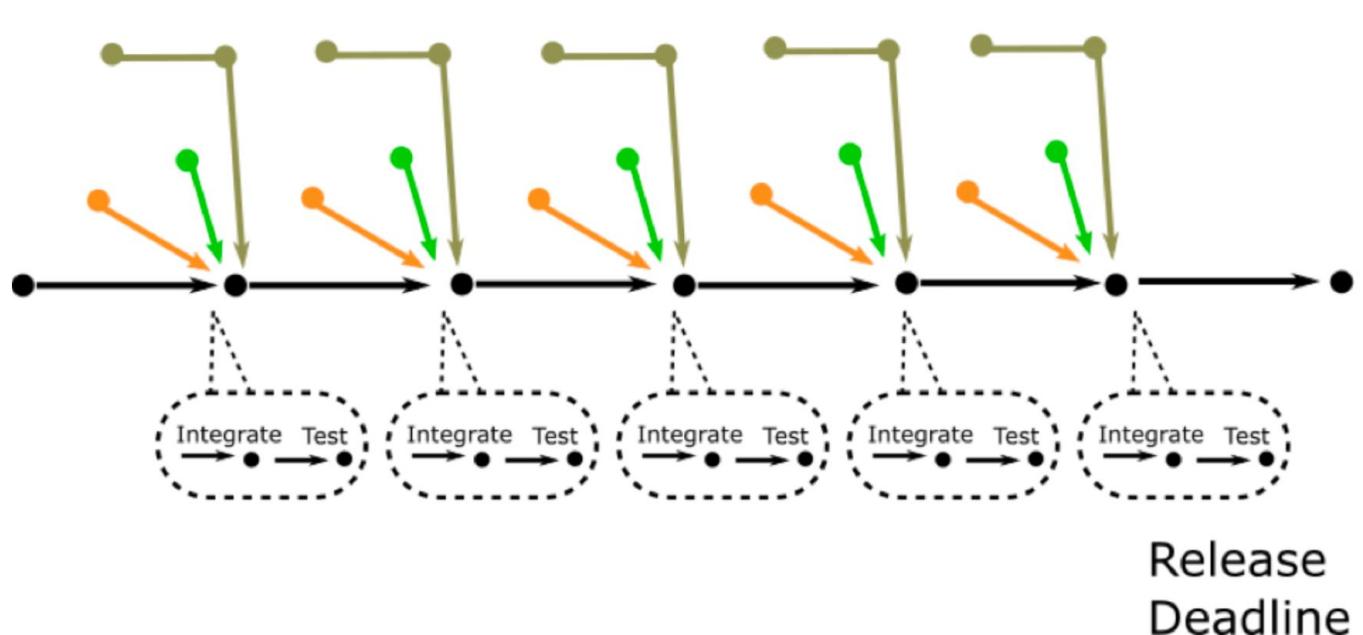
# TDD in Iteration development



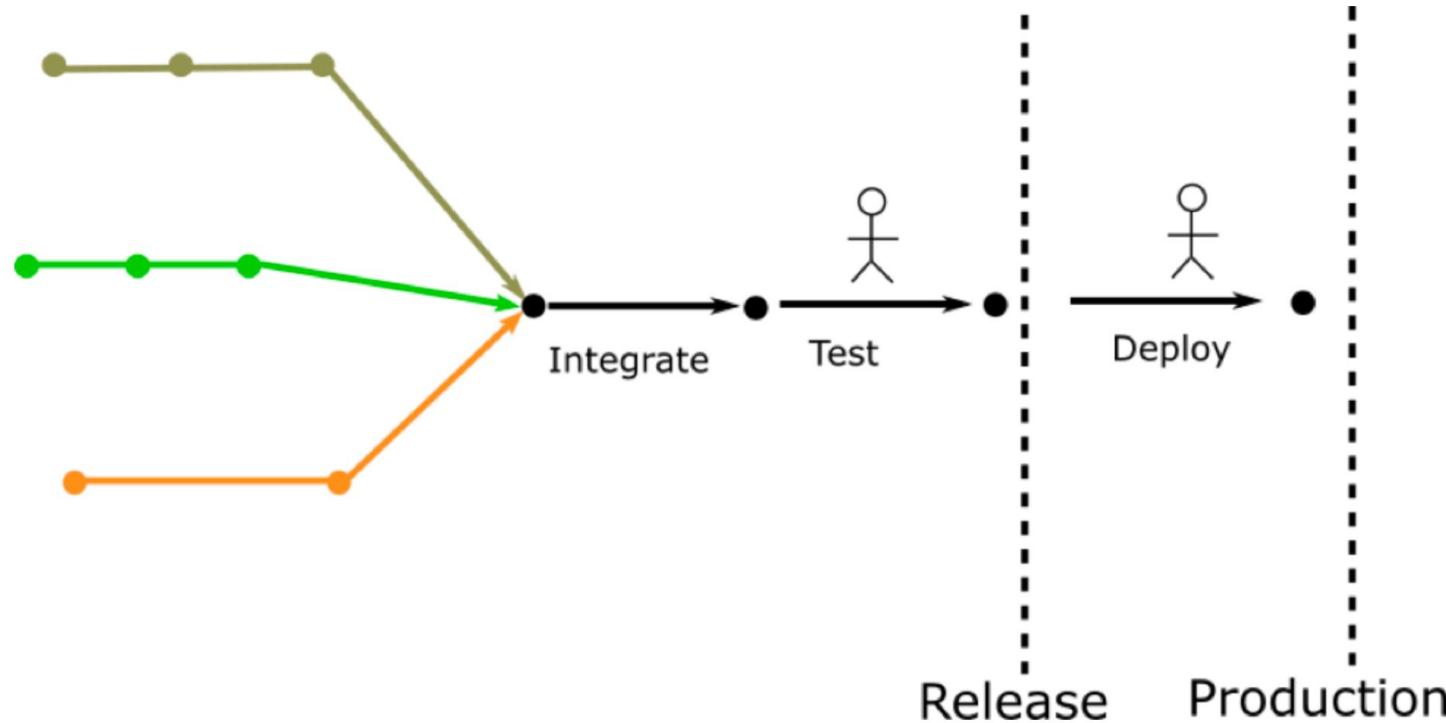
# TDD in Iteration development



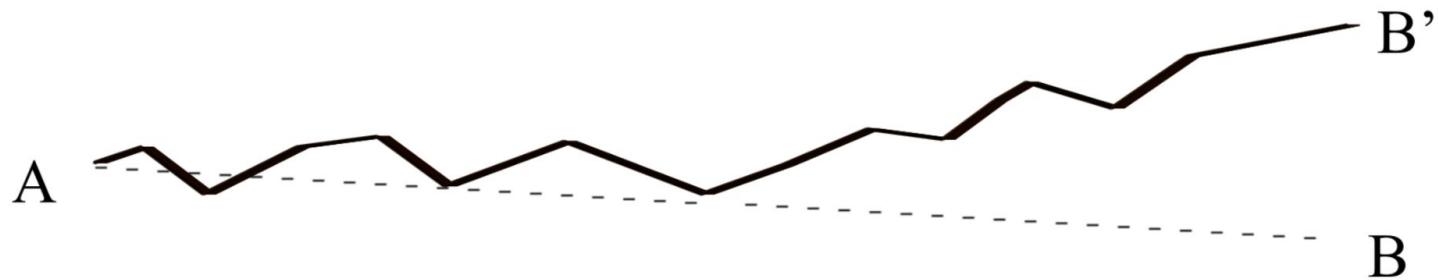
# TDD in Iteration development



# TDD in Iteration development



# Small step



# Exercise

- Fast
  - Isolated/Independent
  - Repeatable
  - Self-validating
  - Timely
1. Test Speed: According to the F.I.R.S.T principles, a well-written unit test should run quickly and not be bogged down by external dependencies like databases or network calls.
    - True / False
  2. Test Dependency: A good unit test should depend on other tests to be executed first, ensuring the proper sequence for validation.
    - True / False

# Exercise - Key answer

- Fast
  - Isolated/Independent
  - Repeatable
  - Self-validating
  - Timely
1. Test Speed: According to the F.I.R.S.T principles, a well-written unit test should run quickly and not be bogged down by external dependencies like databases or network calls.
- **True / False**
2. Test Dependency: A good unit test should depend on other tests to be executed first, ensuring the proper sequence for validation.
- **True / False**

# Code coverage

# Code coverage

A tool to measure how much of your code is covered by tests that break down into classes, methods and lines.

# Code coverage

```
1. public class MyRange {  
2.     public boolean startWithInclude(String input) {  
3.         return input.startsWith("[");  
4.     }  
5.  
6.     public boolean endWithInclude(String input) {  
7.         if(input == null) {  
8.             return false;  
9.         }  
10.        return input.endsWith("]");  
11.    }  
12.  
13.    public boolean startWithInclude2(String input) {  
14.        return input.startsWith("[");  
15.    }  
16.}
```

# **Code coverage**

**Class coverage**

**Method coverage**

**Line coverage**

**Branch coverage**

# Code coverage

But 100% of code coverage **does not mean** that  
your code is 100% correct

**"Code coverage can show the  
high risk areas in a program,  
but never the risk-free."**

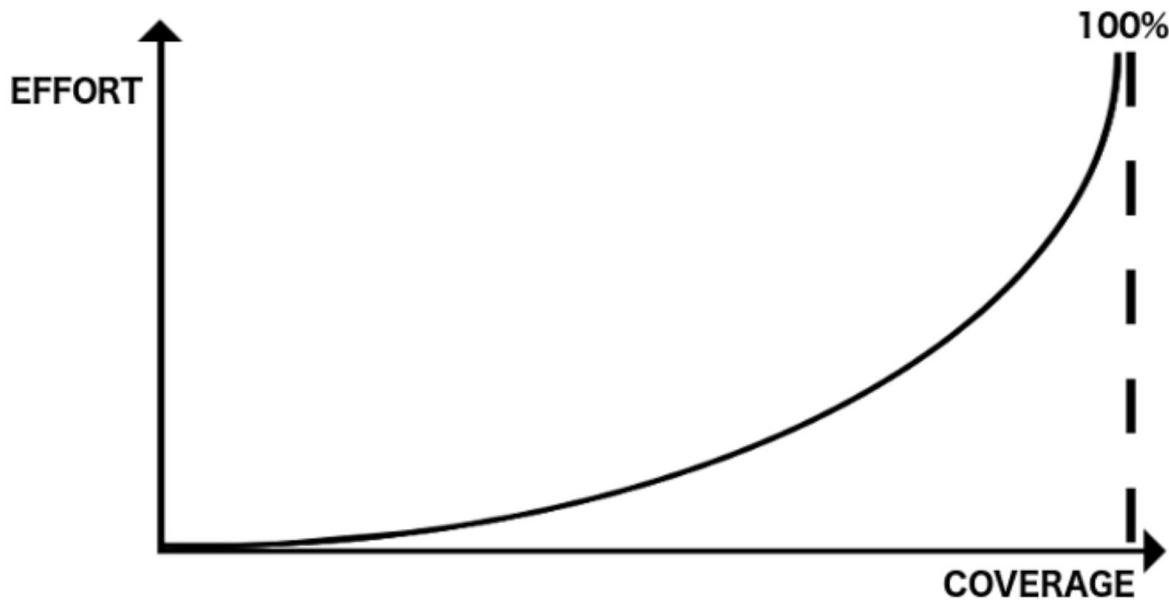
Paul Reilly, 2018, Kotlin TDD with Code Coverage

# **Code coverage**

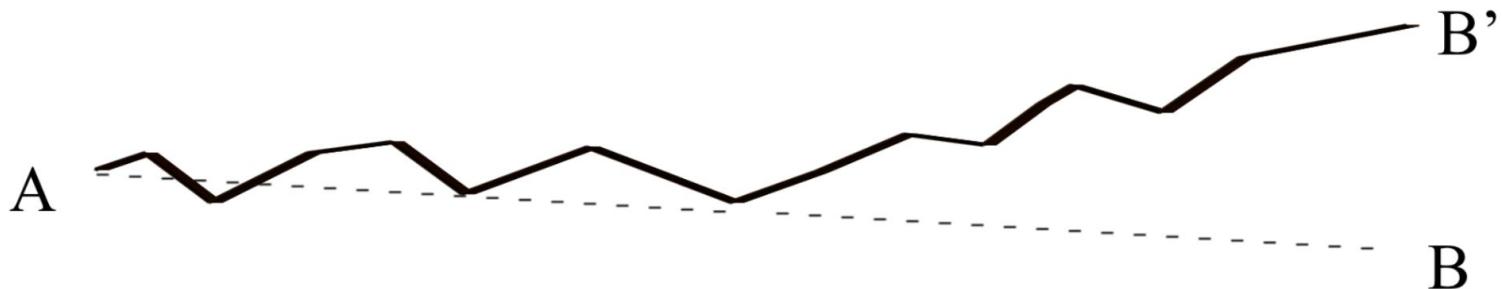
Powerful tool to improve the quality of your code

**Code coverage != quality of tests**

# Code coverage 100% ?

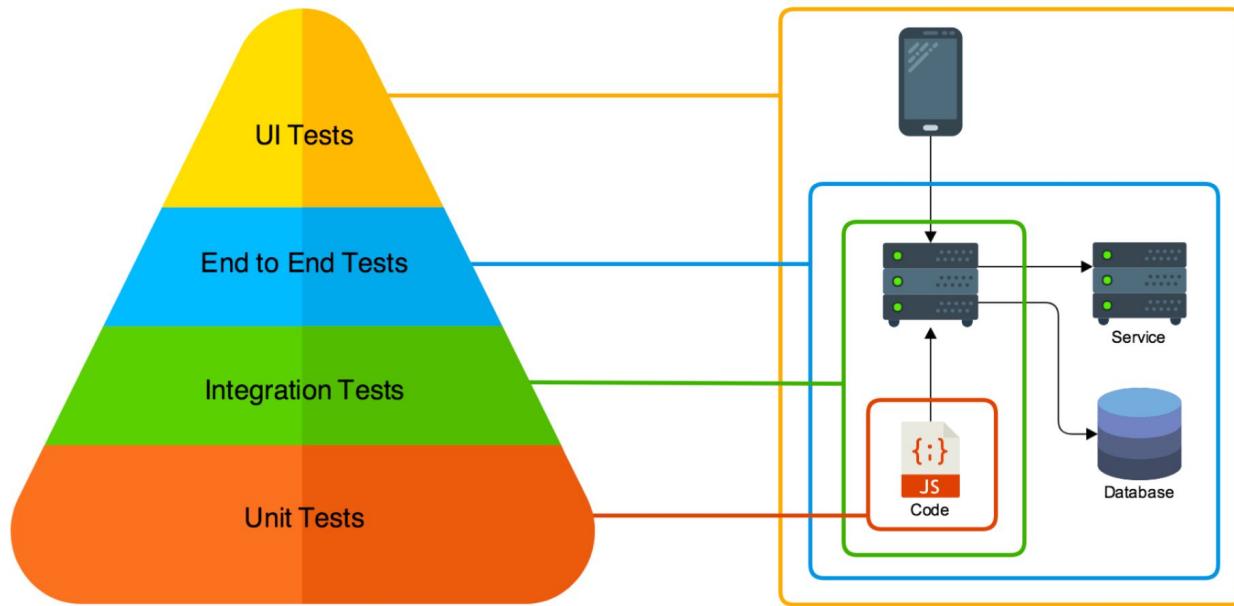


# Small steps



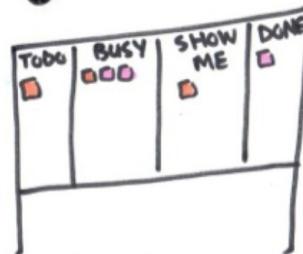
# **Types of Test**

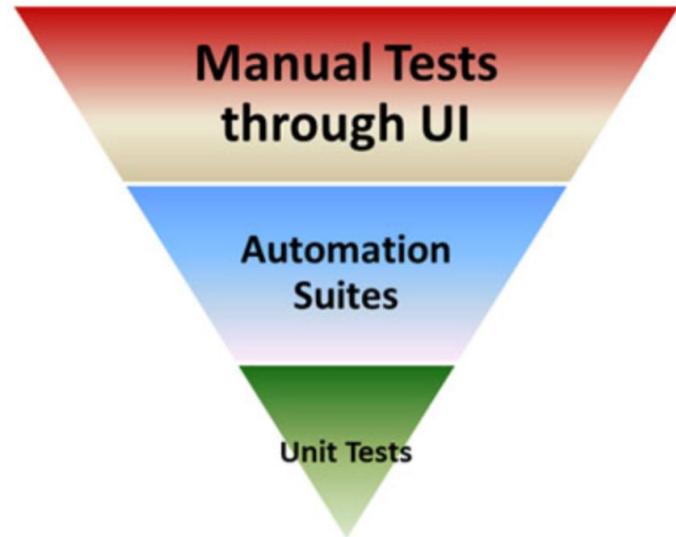
## Scope of each test involved



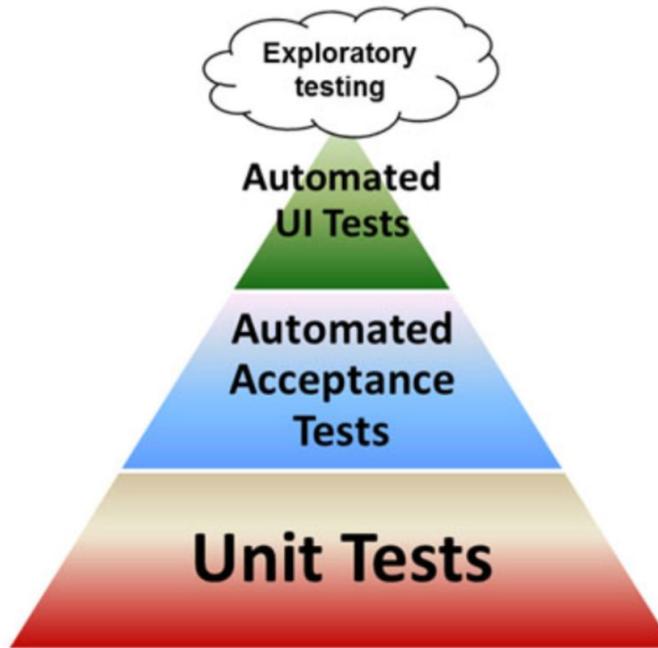


# Testing is an ACTIVITY!

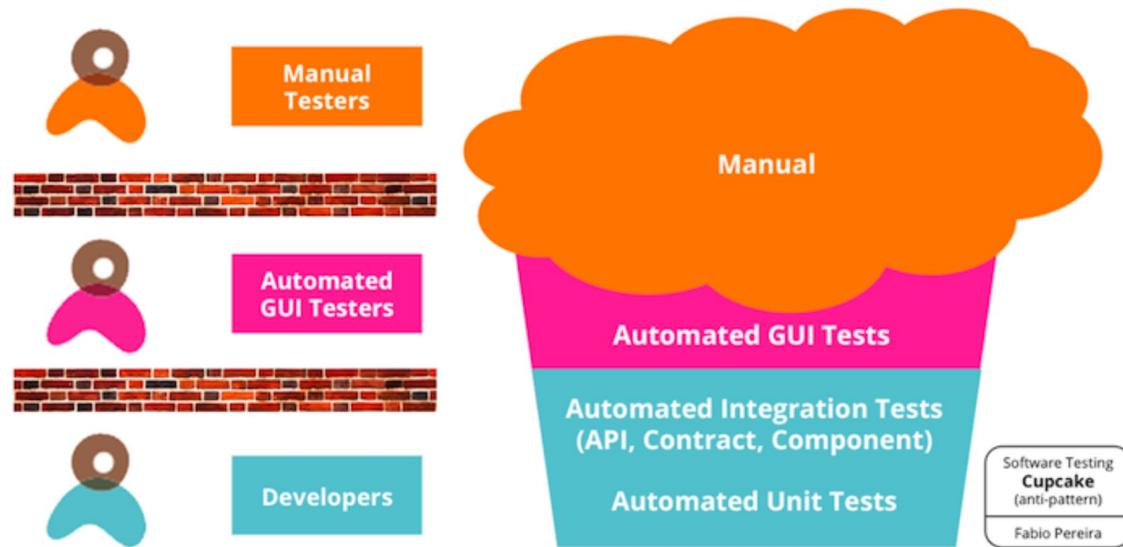




**Traditional**  
(find bugs)



**Agile**  
(prevent bugs)



<https://www.thoughtworks.com/insights/blog/introducing-software-testing-cupcake-anti-pattern>



# Exercise: Preparing your Local environment for test

1. Red-Green-Refactor Cycle: In TDD, the cycle usually begins with writing a failing test (Red), followed by writing the minimal amount of code to make the test pass (Green), and finally refactoring the code to meet the proper standards (Refactor).
  - True / False
2. Initial Test Writing: In TDD, it's recommended to write all the tests for the entire application upfront before writing any functional code.
  - True / False
4. Test Granularity: TDD emphasizes writing tests at the system or integration level rather than the unit level.
  - True / False

# Exercise: Preparing your Local environment for test

1. Red-Green-Refactor Cycle: In TDD, the cycle usually begins with writing a failing test (Red), followed by writing the minimal amount of code to make the test pass (Green), and finally refactoring the code to meet the proper standards (Refactor).

- **True / False**

2. Initial Test Writing: In TDD, it's recommended to write all the tests for the entire application upfront before writing any functional code.

- True / **False**

4. Test Granularity: TDD emphasizes writing tests at the system or integration level rather than the unit level.

- True / **False**

5. Preparing your local dependency for JUnit+Jacoco Coverage

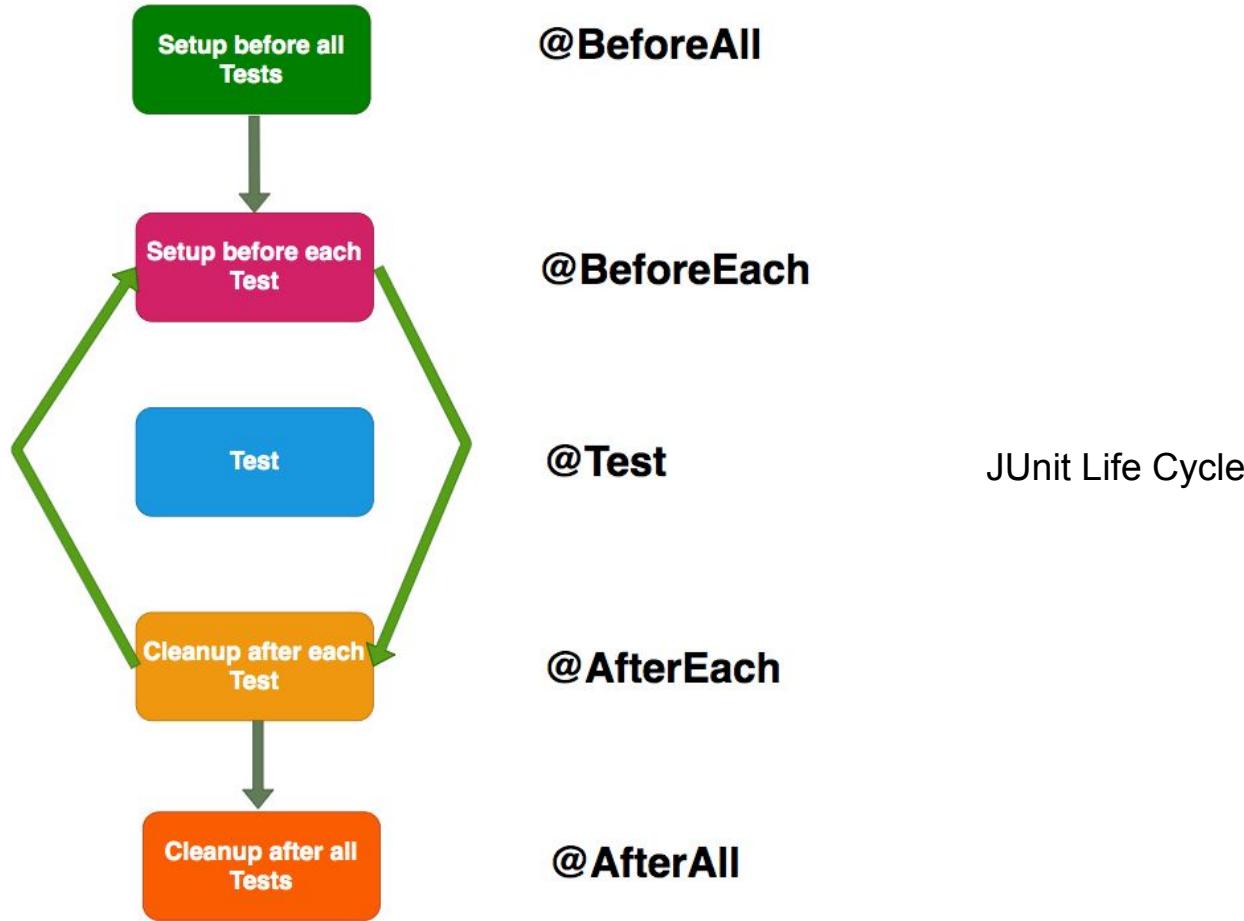
# TDD with JUnit

## Test-Driven Development (TDD)

- TDD Life Cycle
- TDD vs DLP (Debug Later Programming)
- Why TDD is matter
- Unit Testing with F.I.R.S.T
- Code and Test Coverage
- Structure of good unit testing (GUT)

## Unit Testing with JUnit

- **JUnit lifeCycle**
- **Assertion**
- **Data-Driven Test with JUnit**
- **JUnit features**
- **Timeout**
- **Conditional**
- **Category**
- **Suite**
- **Running testing**



# Assertions and assumptions

Assert statement	Example
assertEquals	assertEquals(4, calculator.multiply(2, 2),"optional failure message");
assertTrue	assertTrue('a' < 'b', () → "optional failure message");
assertFalse	assertFalse('a' > 'b', () → "optional failure message");
assertNotNull	assertNotNull(yourObject, "optional failure message");
assertNull	assertNull(yourObject, "optional failure message");

# AssertTimeout

if you want to ensure that a test fails, if it isn't done in a certain amount of time you can use the `assertTimeout()` method. This assert fails the method if the timeout is exceeded.

```
import static org.junit.jupiter.api.Assertions.assertTimeout;
import static java.time.Duration.ofSeconds;
import static java.time.Duration.ofMinutes;

@Test
void timeoutNotExceeded() {
    assertTimeout(ofMinutes(1), () -> service.doBackup());
}

// if you have to check a return value
@Test
void timeoutNotExceededWithResult() {
    String actualResult = assertTimeout(ofSeconds(1), () -> {
        return restService.request(request);
    });
    assertEquals(200, request.getStatus());
}
```

COPY JAVA

```
=> org.opentest4j.AssertionFailedError: execution exceeded timeout of
1000 ms by 212 ms
```

# Testing for exceptions

Testing that certain exceptions are thrown are be done with the

`org.junit.jupiter.api.Assertions.expectThrows()` assert statement. You define the expected Exception class and provide code that should throw the exception.

```
import static org.junit.jupiter.api.Assertions.assertThrows;  
  
@Test  
void exceptionTesting() {  
    // set up user  
    Throwable exception = assertThrows(IllegalArgumentException.class,  
() -> user.setAge("23"));  
    assertEquals("Age must be an Integer.", exception.getMessage());  
}
```

COPY

JAVA

# JUnit Features

## Conditions

### Operating System Conditions

```
@Test  
@EnabledOnOs({OS.WINDOWS, OS.MAC})  
public void shouldRunBothWindowsAndMac() {  
    //...  
}
```

```
@Test  
@DisabledOnOs(OS.LINUX)  
public void shouldNotRunAtLinux() {  
    //...  
}
```

# JUnit Features

## Conditions

### Java Runtime Environment Conditions

```
@Test  
@EnabledOnJre({JRE.JAVA_10, JRE.JAVA_11})  
public void shouldOnlyRunOnJava10And11() {  
    //...  
}
```

```
@Test  
@EnabledForJreRange(min = JRE.JAVA_8, max = JRE.JAVA_13)  
public void shouldOnlyRunOnJava8UntilJava13() {  
    // this test will only run on Java 8, 9, 10, 11, 12, and 13.  
}
```

```
@Test  
@DisabledForJreRange(min = JRE.JAVA_14, max = JRE.JAVA_15)  
public void shouldNotBeRunOnJava14AndJava15() {  
    // this won't run on Java 14 and 15.  
}
```

# JUnit Features

## Conditions

### System Property Conditions

```
@Test  
@EnabledIfSystemProperty(named = "java.vm.vendor", matches = "Oracle.*")  
public void onlyIfVendorNameStartsWithOracle() {  
    //...  
}
```

```
@Test  
@DisabledIfSystemProperty(named = "file.separator", matches = "[/]")  
public void disabledIfFileSeperatorIsSlash() {  
    //...  
}
```

# JUnit Features

## Conditions

### Environment Variable Conditions

```
@Test
@EnabledIfEnvironmentVariable(named = "GDMSESSION", matches = "ubuntu")
public void onlyRunOnUbuntuServer() {
    //...
}

@Test
@DisabledIfEnvironmentVariable(named = "LC_TIME", matches = ".*UTF-8.")
public void shouldNotRunWhenTimeIsNotUTF8() {
    //...
}
```

# What is not Unit ?

External service

UI specification

File system

Compare screen image

Database

Compare HTML

WebService API calls

# Test Naming

**Test Name**

```
public class FizzBuzzTest {
```

**Annotation**

```
@Test
```

**Test Case Name**

```
    public void sayFizzWhenNumberIsDividedByThree() {
```

```
        FizzBuzz fizzBuzz = new FizzBuzz();
```

```
        String actualResult = fizzBuzz.say(3);
```

```
        assertEquals("Fizz", actualResult);
```

```
}
```

```
}
```

“What 's in a name ?”

That which we call a rose

by any other name would smell

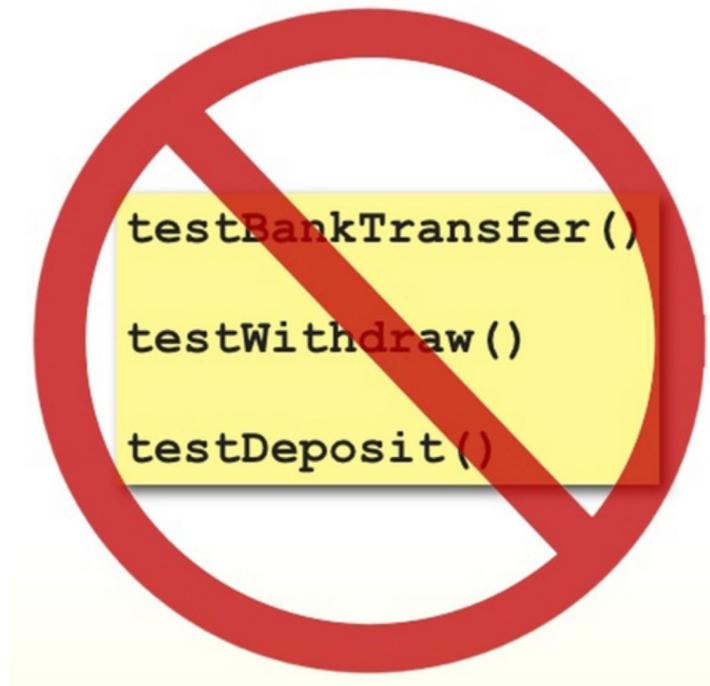
as sweet

Romeo and Juliet

# Guide to Test Writing

Don't say “**test**”, say “**should**”

# Don't use the word “test”



# Use the word “should”

```
transferShouldDeductSumFromSourceAccountBalance()
```

```
transferShouldAddSumLessFeesToDestinationAccountBalance()
```

```
depositShouldAddAmountToAccountBalance()
```

# Guide to Test Writing

Don't test your class, test **behaviour**

# Guide to Test Writing

**Test class names** are important too

**Structure** your test well

Tests are **deliverable** too

# Workshop

---

## FizzBuzz

Input	Expected Result
1	1
2	2
3	Fizz
4	4
5	Buzz
6	Fizz
7	7
8	8
9	Fizz
10	Buzz
15	FizzBuzz

1-20 =>

1,2,[Fizz] // divide by 3 ==0 -> Fizz  
1,2,[Fizz],4,[Buzz] // divide by 5 == 0 => buzz  
...[Fizz],13,14, [FizzBuzz],16

# Good Unit Test

```
@Test  
public void sayFizzWhenNumberIsDividedByThree() {  
Arrange     FizzBuzz fizzBuzz = new FizzBuzz();  
Act          String actualResult = fizzBuzz.say(3);  
Assert       assertEquals("Fizz", actualResult);  
}
```

---

# Setup Pattern

```
@Test  
public void sayFizzWhenNumberIsDividedByThree() {  
    FizzBuzz fizzBuzz = new FizzBuzz();  
  
    String expectedResult = fizzBuzz.say(3);  
  
    assertEquals("Fizz", expectedResult);  
}  
  
@Test  
public void sayBuzzWhenNumberIsDividedByFive() {  
    FizzBuzz fizzBuzz = new FizzBuzz();  
  
    String expectedResult = fizzBuzz.say(5);  
  
    assertEquals("Buzz", expectedResult);  
}
```

# Setup Pattern

Inline  
Setup

```
@Test
public void sayFizzWhenNumberIsDividedByThree() {
    FizzBuzz fizzBuzz = new FizzBuzz();

    String expectedResult = fizzBuzz.say(3);

    assertEquals("Fizz", expectedResult);
}

@Test
public void sayBuzzWhenNumberIsDividedByFive() {
    FizzBuzz fizzBuzz = new FizzBuzz();

    String expectedResult = fizzBuzz.say(5);

    assertEquals("Buzz", expectedResult);
}
```

# Setup Pattern

Delegate  
Setup

```
@Test
public void sayFizzWhenNumberIsDividedByThree() {
    FizzBuzz fizzBuzz = setup();
    String actualResult = fizzBuzz.say(3);
    assertEquals("Fizz", actualResult);
}
```

```
@Test
public void sayBuzzWhenNumberIsDividedByFive() {
    FizzBuzz fizzBuzz = setup();
    String actualResult = fizzBuzz.say(5);
    assertEquals("Buzz", actualResult);
}
```

```
private FizzBuzz setup() {
    FizzBuzz fizzBuzz = new FizzBuzz();
    return fizzBuzz;
}
```

# Setup Pattern

## Implicit Setup

```
@Before  
public void initial() {  
    fizzBuzz = new FizzBuzz();  
}
```

```
@Test  
public void sayFizzWhenNumberIsDividedByThree() {  
    String actualResult = fizzBuzz.say(3);  
    assertEquals("Fizz", actualResult);  
}
```

```
@Test  
public void sayBuzzWhenNumberIsDividedByFive() {  
    String actualResult = fizzBuzz.say(5);  
    assertEquals("Buzz", actualResult);  
}
```

# Implicit Teardown

```
@After  
public void clearResources(){  
    fizzBuzz = null;  
}
```

# Ignore Testcase

```
@Ignore("Pending implemetation")
@Test
public void sayFizzWhenNumberIsDevidedByThree() {
    FizzBuzz fizzBuzz = new FizzBuzz();
    String actualResult = fizzBuzz.say(3);
    assertEquals("Fizz", actualResult);
}
```

# Handle Exception

# Traditional approach

```
@Test  
public void invalidWhenNumberLessThanOne() {  
    try {  
        fizzBuzz.say(0);  
        fail();  
    } catch (IllegalStateException expected) {  
    }  
}
```

# Expected Annotation

```
@Test(expected=IllegalStateException.class)
public void invalidWhenNumberLessThanOne() {
    fizzBuzz.say(0);
}
```

# ExpectedException Rule

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void invalidWhenNumberLessThanOne() {
    thrown.expect(IllegalStateException.class);
    fizzBuzz.say(0);
}
```

# Data-Driven with JUnit

# Using parameterized

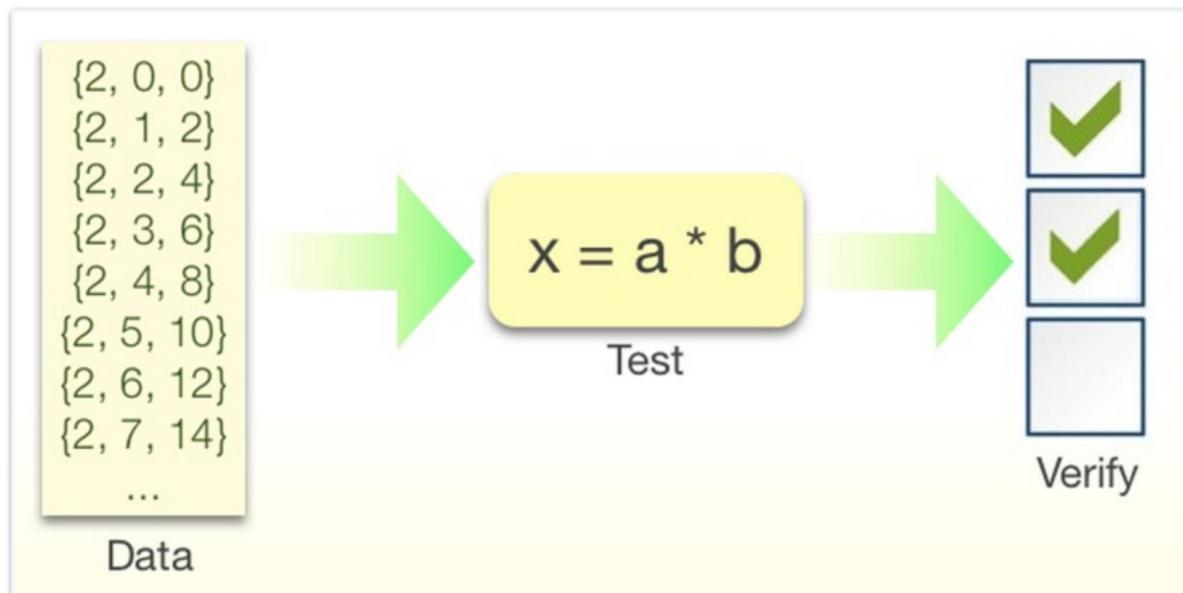
Set of test data

Expected result

Define test that uses the test data

Verify result against expected result

# Data-Driven Development



# @Parameterized

## with Calculate Grade

Input	Expected Result
80	A
70	B
60	C
50	D
40	F

# Step 1 :: Test Data

Input	Expected Result
80	A
70	B
60	C
50	D
40	F

# Step 2 :: Add Runner

```
@RunWith(Parameterized.class)
public class CalculateGradeTest {

    private int score;
    private String expectedGrade;

    public CalculateGradeTest(int score, String expectedGrade) {
        this.score = score;
        this.expectedGrade = expectedGrade;
    }

    @Test
    public void convertScoreToGrade() {
    }

}
```

# Step 3 :: Matching fields

```
@RunWith(Parameterized.class)
public class CalculateGradeTest {

    private int score;
    private String expectedGrade;

    public CalculateGradeTest(int score, String expectedGrade) {
        this.score = score;
        this.expectedGrade = expectedGrade;
    }

    @Test
    public void convertScoreToGrade() {
    }
}
```

# Step 4 :: Define test case

```
@RunWith(Parameterized.class)
public class CalculateGradeTest {

    private int score;
    private String expectedGrade;

    public CalculateGradeTest(int score, String expectedGrade) {
        this.score = score;
        this.expectedGrade = expectedGrade;
    }

    @Test
    public void convertScoreToGrade() {
    }

}
```

# Step 5 :: Add @parameters

```
@RunWith(Parameterized.class)
public class CalculateGradeTest {

    private int score;
    private String expectedGrade;

    @Parameters
    public static Collection<Object[]> data(){
        return Arrays.asList(new Object[][]{
            {80, "A"},
            {70, "B"},
            {60, "C"},
            {50, "D"},
            {40, "F"}
        });
    }
}
```

# Workshop

@Parameterized  
with Calculator

Operand 1	Operator	Operand 2	Expected
1	+	1	2
2	+	2	4
1	-	1	0
2	-	1	1
3	*	1	3
3	*	2	6

# Selfie

