

Testy jednostkowe

Agenda

1. Omówienie Unit Testing
2. Piramida testowania
3. TDD jako dobra praktyka programisty
4. Chronologia i podejście do pisania testów
5. Zalety / wady TDD
6. Przydatne biblioteki w Javie i Pythonie

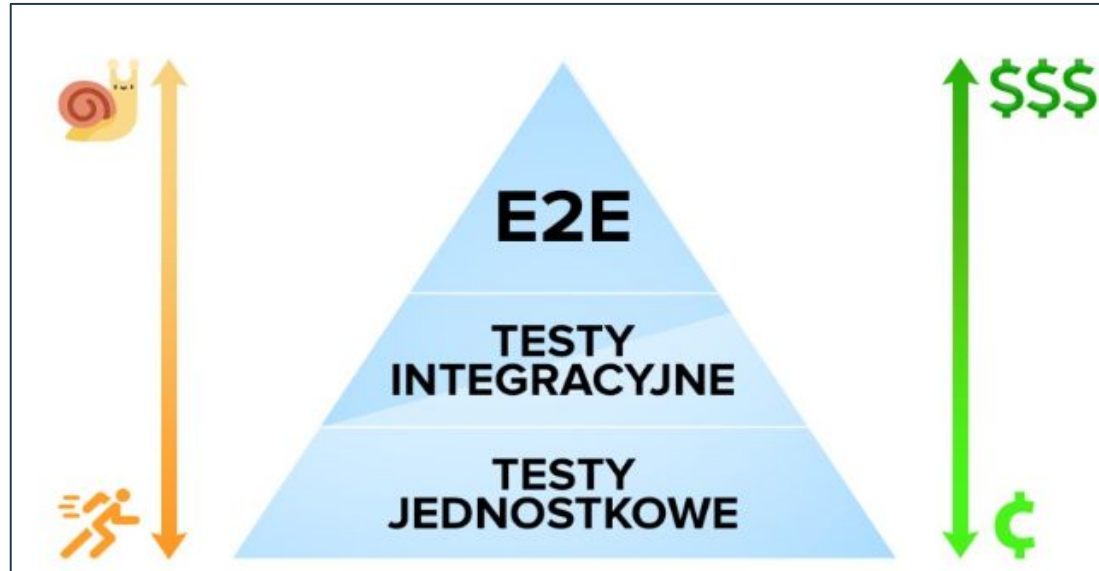
Czym są testy jednostkowe?

- Sprawdzenie czy dana funkcjonalność działa jak należy.
- Testowanie następuje wyłącznie dla wyizolowanego obszaru np. klasa, lub konkretna metoda.
- Pisane i utrzymywane wyłącznie w jednym celu.
- Nie ma wymogu że jedna metoda/klasa to jeden test - często testowane metody mają kilka/ kilkanaście testów.

Po co pisać testy?

- **Szybkość** - w ciągu sekundy może zostać odpalone kilkanaście / kilkadziesiąt testów.
- **"Tanie"** - nawet z narzutem jaki programista poświęcił na napisanie testu, to i tak jest tańsze niż opłacenie manualnego testera sprawdzającego daną funkcjonalność.
- **Walidacja** - raz napisany test, może być wielokrotnie używany do testowania kiedy dodamy jakąś nową funkcjonalność.
- **Dokumentacja** - dobrze napisane testy i przypadki testowe, w przejrzysty sposób pokazują oczekiwane zachowanie aplikacji

Poziomy testowania



Dobre praktyki pisania testów

- Testy powinny wykonywać się szybko - pojedynczy test powinien być relatywnie krótki
- Powinny skupiać się na jak najmniejszej jednostce - testujemy metody składowe zamiast dużych serwisowych
- Każda funkcjonalność biznesowa powinna mieć odpowiedni test jednostkowy

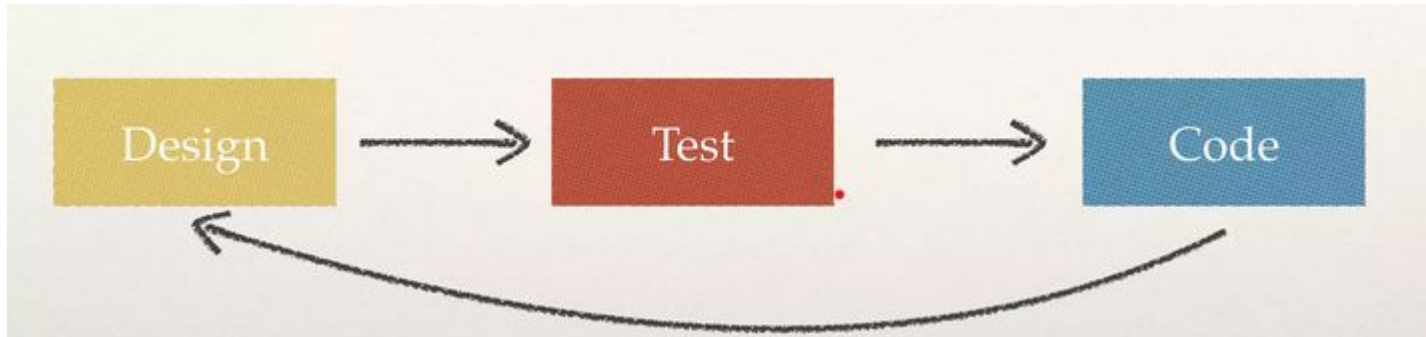
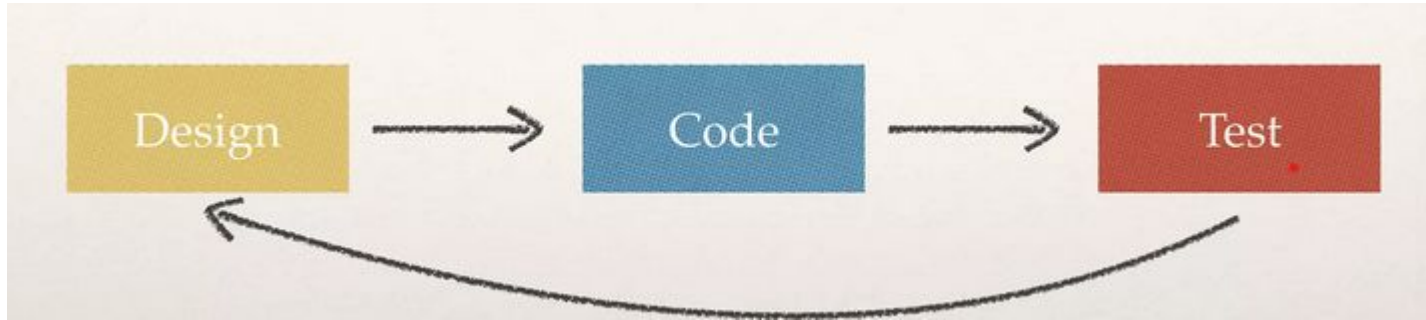
Antywzorce testowe

- Nastawienie na 100% pokrycie kodu testami.
- Nieintuicyjne nazwy metod testowych.
- Długodziałające testy.
- Testy które po sobie nie sprzątaj.
- Wiele niepowiązanych asercji w jednym teście.
- Traktowanie testów jako kodu drugiej kategorii.
- Ręczne uruchamianie testów.

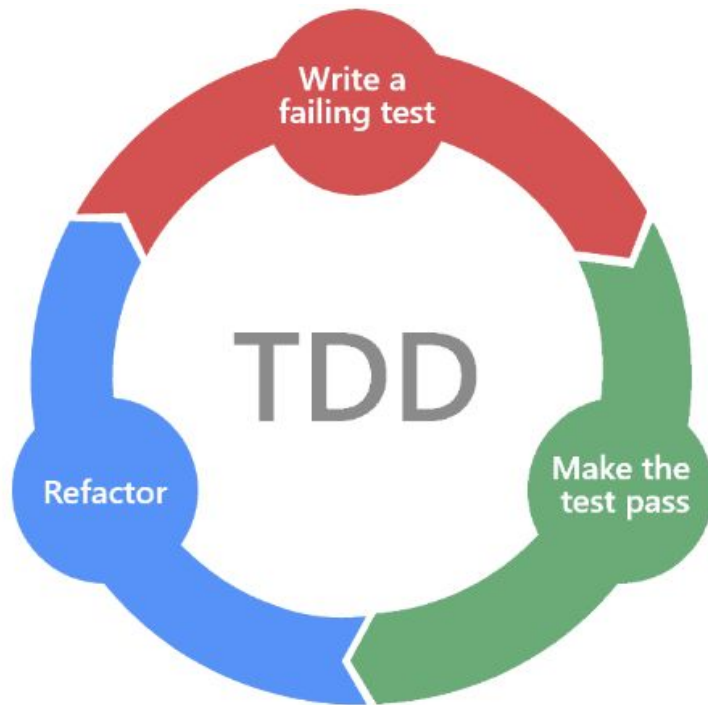
TDD - test driven development

- To praktyka pisania testów jednostkowych przed napisaniem kodu produkcyjnego.
- Jest to odwrotność sposobu, w jaki większość programistów pisze kod, ale ma wiele zalet.
- Główne założenie to eliminacja niepotrzebnych linii kodu
- Programując zgodnie z TDD, większość błędów zostanie wykryta podczas testów przed wprowadzeniem kodu w środowisko testowe lub produkcyjne

Pisanie kodu kiedyś vs dziś



Workflow podczas TDD



Cykle życia TDD

- **Faza czerwona** - polega na napisaniu nieudanego testu jednostkowego, dla danej funkcjonalności, którą chcemy zaimplementować w naszym kodzie produkcyjnym
- **Faza zielona** - piszemy minimalną ilość kodu, która jest niezbędna do przejścia testu
- **Faza refaktoryzacji kodu** - opiera się na czyszczeniu kodu testowego, aby był utrzymywany przy takich samych standardach, jak kod produkcyjny

Jak długie powinny być cykle życia TDD?

Wielu programistów zadaje sobie pytanie, ***“Jak długie powinny być testy?”*** , ***“Jak duży fragment kodu powinny pokrywać?”***

- Jednym z założeń TDD jest pisanie jak najkrótszych cykli czyli dużej ilości relatywnie krótkich testów, gdzie każdy będzie pokrywał jakąś część implementacji metody serwisowej.
- Cykle powinny być takie aby programista był w stanie zrozumieć to co dzieje się w danym przypadku testowym.

Kiedy nie należy stosować TDD ?

- Wykorzystanie tej praktyki nie sprawdzi się w przypadku małych, nieskomplikowanych projektów
- W przypadku kodu trywialnego lub kiedy służy jako “łącznik” między modułami, testy jednostkowe mogą być nadmiernym obciążeniem
- Kiedy rozwiązanie problemu nie jest oczywiste, wtedy TDD może być ograniczające, ponieważ wymaga pisania testów jednostkowych przed napisaniem właściwego kodu

Najważniejsze zalety TDD

- Programowanie zgodne z TDD umożliwia wykrycie większości błędów w kodzie, zanim zostanie wprowadzony w środowisko testowe lub produkcyjne.
- TDD wymaga rozpoczęcia testów od najprostszych funkcji, a to sprzyja rozkładowi głównego problemu na mniejsze fragmenty (łatwiejsze do rozwiązania sprawdzenia i weryfikacji).
- Pomaga w dobrym projektowaniu zorientowanym obiektowo, ponieważ sprawia że klasy i funkcje mogą być testowane w izolacji.

Najpopularniejsze biblioteki

JUnit 


pytest

mockito 


unittest

JUnit 5

- Najpopularniejsza biblioteka w swojej dziedzinie... i na Świecie.
- Idea polega na oznaczaniu metod odpowiednimi adnotacjami które dzięki temu są automatycznie egzekwowane przez framework.
- Pozwala na pisanie niezależnych od siebie testów, co ułatwia ich skalowanie w miarę rozwoju aplikacji.
- Bazujemy na pisaniu tzw. *asercji*.

JUnit 5

```
@Test
void testIfUnitWeightAfterUnloadingAllCargoIsCorrect() {
    Unit unit1 = new Unit(new Coordinates( x: 1, y: 1), maxFuel: 100,

    Cargo cargo1 = new Cargo( name: "Fish", weight: 50);
    Cargo cargo2 = new Cargo( name: "Ham", weight: 30);

    unit1.loadCargo(cargo1);
    unit1.loadCargo(cargo2);

    unit1.unloadAllCargo();

    assertThat(unit1.getCurrentCargoWeight(), is(equalTo( operand: 0)))
    assertThat(unit1.getCargo().size(), is(equalTo( operand: 0)));
}
```

1 usage

```
public void unloadAllCargo() {
    this.cargo.clear();
    this.currentCargoWeight = 0;
}
```

JUnit 5

```
1 usage
private static Stream<Arguments> createMealsWithNameAndPrice() {
    return Stream.of(
        Arguments.of( ...arguments: "Baconburger", 20),
        Arguments.of( ...arguments: "Hamburger", 10),
        Arguments.of( ...arguments: "Cheeseburger", 15)
    );
}
```

```
@ParameterizedTest
```

```
@MethodSource("createMealsWithNameAndPrice")
```

```
void foodNameShouldBeEqualToValuePassedAsParameter(String name, int price) {
    assertThat(name, containsString( substring: "burger"));
    assertThat(price, greaterThanOrEqualTo( value: 10));
}
```

JUnit 5

1	Fabryczna, 10, 30-301, Kraków
2	Armii Krajowej, 57, 30-150, Kraków

```
@ParameterizedTest
@CsvFileSource(resources = "/address.csv")
void givenAddressesShouldNotBeEmptyFromCsv(String street, String number, String postalCode, String city) {
    assertThat(street, notNullValue());
    assertThat(number, notNullValue());
    assertThat(postalCode, notNullValue());
    assertThat(postalCode, containsString( substring: "-"));
    assertThat(city, notNullValue());
}
```

JUnit 5

```
@BeforeEach
void initializeOrder() {
    // method that will be executed before each test
}

@AfterEach
void cleanUp() {
    // method that will be executed after each test
}
```

```
@Test
void mealListShouldBeEmptyAfterCreationOfOrder() {
    //(...)
    assertThat(order.getMeals(), is(empty()));
    assertThat(order.getMeals(), emptyCollectionOf(Meal.class));
}
```

JUnit 5

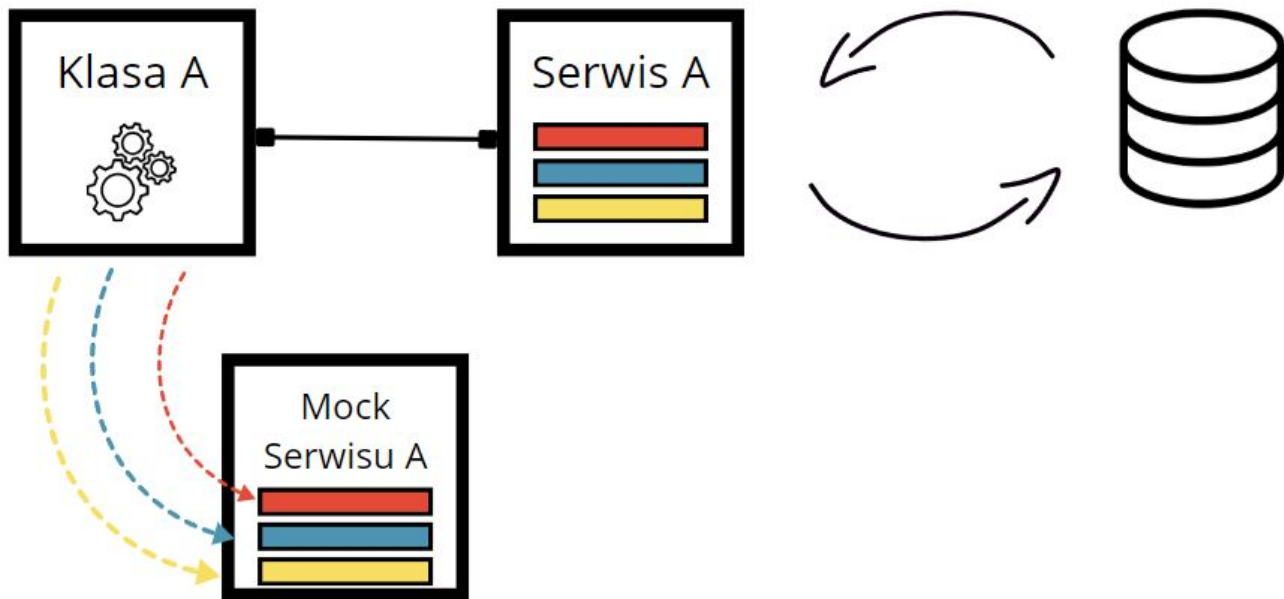
```
assertThat(cart.getOrders(), allOf(  
    notNullValue(),  
    hasSize(1),  
    is(not(emptyCollectionOf(Order.class))),  
    is(not(empty()))  
));
```

```
assertAll(  
    () -> assertThat(cart.getOrders(), notNullValue()),  
    () -> assertThat(cart.getOrders(), hasSize(1)),  
    () -> assertThat(cart.getOrders(), is(not(emptyCollectionOf(Order.class)))),  
    () -> assertThat(cart.getOrders(), is(not(empty()))),  
    () -> assertThat(cart.getOrders().get(0).getMeals(), empty())  
);
```



- Druga najczęściej pobierana biblioteka Javy (po JUnit).
- Ułatwia testowanie kodu w izolacji od jego zależności.
- Główną ideą jest tworzenie tzw. **mock'ów** - czyli obiektów których celem jest zastąpienie oryginalnych w celu określenia ich zachowania podczas różnych sytuacji.

Do czego przydaje się tworzenie *Mocków* ?





10 usages 1 implementation

```
public interface AccountRepo {  
    5 usages 1 implementation  
    List<Account> getAllAccounts();  
}
```

@Test

```
void getAllActiveAccounts() {  
    // given  
    AccountRepo accountRepoStub = new AccountRepoStub();  
    AccountService accountService = new AccountService(accountRepoStub);  
    //when  
    List<Account> accountList = accountService.getAllActiveAccounts();  
    //then  
    assertThat(accountList.size(), is(equalTo(operand: 2)));  
}
```

1 usage

```
public class AccountRepoStub implements AccountRepo {  
    5 usages  
    @Override  
    public List<Account> getAllAccounts() {  
        Address address1 = new Address( street: "Krakowska", city: "Kraków",  
        Address address2 = new Address( street: "Warszawska", city: "Warszawa"  
  
        Account account1 = new Account(address1);  
        Account account2 = new Account(address2);  
        Account account3 = new Account();  
  
        return Arrays.asList(account1, account2, account3);  
    }  
}
```

```
public class AccountService {  
    3 usages  
    private AccountRepo accountRepo;  
    4 usages  
    public AccountService(AccountRepo accountRepo) {  
        this.accountRepo = accountRepo;  
    }  
    2 usages  
    List<Account> getAllActiveAccounts() {  
        return accountRepo.getAllAccounts().stream()  
            .filter(Account::isActive)  
            .collect(Collectors.toList());  
    }  
}
```




10 usages 1 implementation

```
public interface AccountRepo {  
    5 usages 1 implementation  
    List<Account> getAllAccounts();  
}
```

```
public class AccountService {  
    3 usages  
    private AccountRepo accountRepo;  
    4 usages  
    public AccountService(AccountRepo accountRepo) {  
        this.accountRepo = accountRepo;  
    }  
    2 usages  
    List<Account> getAllActiveAccounts() {  
        return accountRepo.getAllAccounts().stream()  
            .filter(Account::isActive)  
            .collect(Collectors.toList());  
    }  
    2 usages  
    List<Account> getAllInactiveAccounts() {  
        return accountRepo.getAllAccounts().stream()  
            .filter(account -> !account.isActive())  
            .collect(Collectors.toList());  
    }  
}
```

```
@Test  
void getAllActiveAccounts() {  
    // given  
    List<Account> accounts = prepareAccountData();  
    AccountRepo accountRepo = mock(AccountRepo.class);  
    AccountService accountService = new AccountService(accountRepo);  
  
    given(accountRepo.getAllAccounts()).willReturn(accounts);  
    //when  
    List<Account> accountList = accountService.getAllActiveAccounts();  
    //then  
    assertThat(accountList.size(), is(equalTo(operand: 2)));  
}
```

Unittest

- Wzorowany na frameworku JUnit
- Zawarty jest w bibliotece standardowej Pythona - co czyni go łatwo dostępnym oraz pewnym rozwiązaniem
- Pozwala zdefiniować testy, które mogą być wykonywane automatycznie przy każdej zmianie w kodzie
- oferuje przypadki testowe, zestawy, moduły uruchamiające i raporty.

```
import unittest

class Test(unittest.TestCase):
    def setUp(self):    #prepare test environment
        pass

    def test_addition(self):
        self.assertEqual(1 + 2, second: 3)

    def tearDown(self): # cleanup after test
        pass

if __name__ == "__main__":
    unittest.main()
```

Struktura



```
===== test session starts =====
collecting ... collected 1 item


test_unit_1.py::Test::test_addition PASSED [100%]

===== 1 passed in 0.03s =====
```

```
import sys
import unittest

class Test(unittest.TestCase):
    @unittest.skipIf(sys.platform == "win32", reason="Can't run on Windows")
    def test_addition_2(self):
        self.assertEqual(1+2, second: 4)

    @unittest.expectedFailure
    def test_addition_3(self):
        raise RuntimeError("ERROR HAPPENED")
```



```
===== test session starts =====
collecting ... collected 2 items

test_unit_2.py::Test::test_addition_2
test_unit_2.py::Test::test_addition_3

===== 1 skipped, 1 xfailed in 0.29s =====
```

PyTest

- Jeden z najbardziej popularnych open-sourcowych frameworków na rynku
- Daje użytkownikowi więcej swobody w przeciwieństwie do Unittest
- Oferuje prostą składnię i bogatą funkcjonalność, ułatwiającą pisanie testów jednostkowych, integracyjnych i funkcjonalnych.
- Posiada również wiele dodatkowych funkcji takich jak parametryzacja testów, fixture czy też wsparcie dla testowania asynchronicznego

Struktura



```
import pytest

def test_addition():
    assert 1 + 2 == 3

class TestClass:
    def test_addition(self):
        assert 1 + 2 == 3
```

```
===== test session starts =====
platform win32 -- Python 3.12.0, pytest-8.1.1, pluggy-1.4.0 -- C:\Users\Thinkpad\Desktop\JPWP projekt\projekt\.venv
cachedir: .pytest_cache
rootdir: C:\Users\Thinkpad\Desktop\JPWP projekt\projekt
collected 2 items

testy.py::test_addition PASSED
testy.py::TestClass::test_addition PASSED

===== 2 passed in 0.07s =====
```

Fikstura

```
import pytest

1 usage

class Connector:
    def __init__(self, login, passwd):
        self.login = login
        self.password = passwd

1 usage

    def send_msg(self, msg):
        return f"Message '{msg}' sent successfully"

2 usages

@pytest.fixture(scope="function")
def connector():
    conn = Connector(login="login", passwd="passwd")
    yield conn
    del conn

def test_connection(connector):
    assert connector.send_msg("Hello") == "Message 'Hello' sent successfully"
```

1 usage

```
@pytest.fixture()
```

```
def setup1():
```

```
    print("\nSetup 1")
```

```
    yield
```

```
    print("\nTearDown 1")
```

1 usage

```
@pytest.fixture()
```

```
def setup2(request):
```

```
    print("\nSetup 2")
```

```
    def teardown_a():
```

```
        print("\nTearDown A")
```

```
    def teardown_b():
```

```
        print("\nTearDown B")
```

```
    request.addfinalizer(teardown_a)
```

```
    request.addfinalizer(teardown_b)
```

```
def test1(setup1):
```

```
    print("\nExecuting test 1")
```

```
    assert True
```

```
def test2(setup2):
```

```
    print("\nExecuting Test2")
```

```
    assert True
```

test_fixture_yield.py::test1

Setup 1

Executing test 1

PASSED

TearDown 1

test_fixture_yield.py::test2

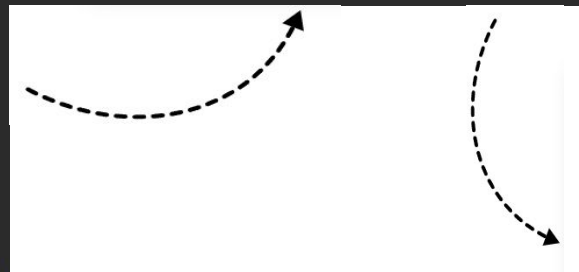
Setup 2

Executing Test2

PASSED

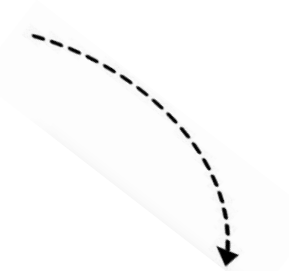
TearDown B

TearDown A



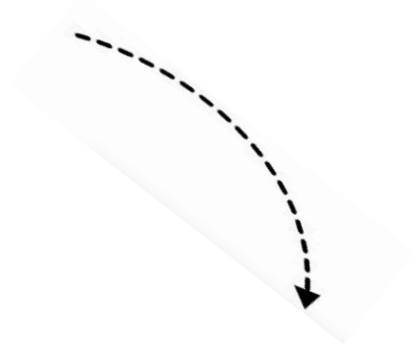
Parametryzacja

```
@pytest.mark.parametrize("a,b", [(1,2), (3,4)])  
def test_addition(a,b):  
    print(a + b)
```



```
collecting ... collected 2 items  
|  
test_pytest3.py::test_addition[1-2] PASSED [ 50%]  
  
test_pytest3.py::test_addition[3-4] PASSED [100%]
```

```
@pytest.mark.parametrize("a", [1, 2])
@pytest.mark.parametrize("b", [3, 4])
def test_addition(a, b):
    print(a + b)
```



```
collecting ... collected 4 items

test_pytest3.py::test_addition[3-1] PASSED [ 25%]

test_pytest3.py::test_addition[3-2] PASSED [ 50%]

test_pytest3.py::test_addition[4-1] PASSED [ 75%]

test_pytest3.py::test_addition[4-2] PASSED [100%]
```

Zarządzanie testami

```
import pytest
import sys

@pytest.mark.skipif(condition=sys.platform == 'win32', reason="Can't run tests on Windows")
def test_addition(self):
    self.assertEqual(1+2, 4)

@pytest.mark.xfail
def test_addition_3(self):
    raise RuntimeError("ERROR HAPPENED")
```

Bibliografia

- https://pl.wikipedia.org/wiki/Test_jednostkowy
- <https://junit.org/junit5/docs/current/user-guide/>
- <https://site.mockito.org/>
- <https://www.baeldung.com/java-unit-testing-best-practices>
- <https://docs.python.org/3/library/unittest.html>
- <https://docs.pytest.org/en/stable/deprecations.html#applying-a-mark-to-a-fixture-function>
- <https://www.browserstack.com/guide/what-is-test-driven-development>