

Zawartość:

- co to są testy
- po co są testy
- co testować a co nie
- jakie założenia musi spełniać kod by dało się go testować
- jak nazywać testy
- jakie cechy powinny mieć testy
- przykład w JS i C#

Testy automatyczne oprogramowania:

Test jest to najprościej ujmując funkcja, która testuje inny kawałek kodu. Co to znaczy testuje? Oznacza to, że sprawdza ona czy wynik działania pewnego kodu (najczęściej metody/funkcji) dla określonych argumentów jest zgodny z tym co przewidział programista.

Przykład:

```
0 references
class Calculator
{
    //some stuff

    0 references
    public double divide(int a, int b)
    {
        if (b == 0)
            throw new DivideByZeroException();

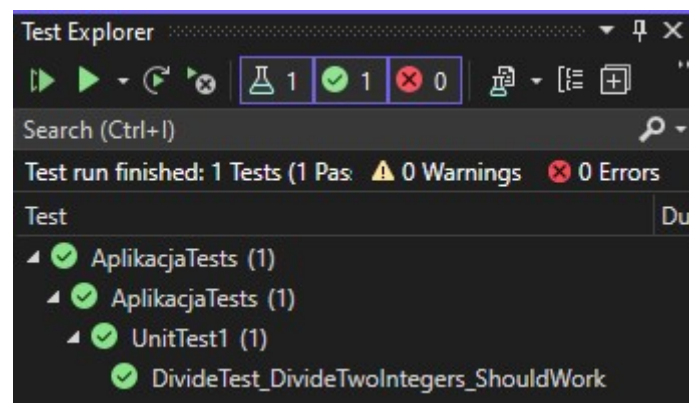
        return a / b;
    }
}
```

Mamy klasę kalkulatora, która implementuje n metod. Celem testu będzie metoda dzieląca dwie liczby:

```
0 references
public class UnitTest1
{
    [Fact]
    0 references
    public void DivideTest_DivideTwoIntegers_ShouldWork()
    {
        //Arrange
        Calculator calculator = new Calculator();

        //Act
        double result = calculator.divide(6,2);

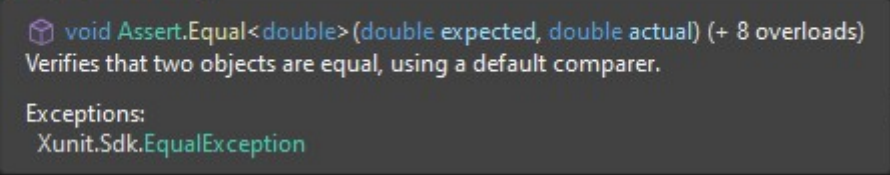
        //Assert
        Assert.Equal(3, result);
    }
}
```



Sukces

Napisaliśmy sobie funkcję dzielącą dwie liczby całkowite. Umiemy policzyć co powinna zwrócić dla konkretnych danych. Dla argumentów 6 i 2 powinna zwrócić liczbę 3. Zgodnie z podpowiedzią dla funkcji `Assert.Equal()` wpisujemy wartość oczekiwaną oraz uzyskany wynik:

```
//Assert
Assert.Equal(3, result);
```



Oczywiście powyższy przykład jest banalny na tyle, że testy nie mają sensu chociaż funkcja `divide` w przeciwieństwie do `add` czy `multiple` ma przypadki użycia, które nie są trywialne (dzielenie przez zero).

Po co komu te testy?

- Wyobraźmy sobie, że napisaliśmy dosyć skomplikowaną funkcję. Chcemy szybko i sprawnie sprawdzić czy działa. Takie sprawdzenie może być czasami łatwiejsze a czasami trudniejsze w zależności od tego co robi testowana funkcja. Czasami trzeba przeklinać parę okienek/przycisków aby dotrzeć do momentu wykorzystującego daną funkcję. Czasami trzeba wprowadzić dużo danych z klawiatury. Ogólnie taki test może być skomplikowany a co gorsze trzeba będzie proces powtarzać wiele razy w czasie pisania aplikacji. Test automatyczny pozwoli nam raz napisać test a później uruchamiać go do woli ile razy chcemy.
- Gdy mamy napisaną funkcję oraz testy do niej możemy ją swobodniej edytować. Chcemy dodać jakiś kawałek kodu. Dodajemy bez stresu, ponieważ wystarczy uruchomić testy by upewnić się, że nie zepsuliśmy nic w istniejącej funkcjonalności. Jeżeli coś zepsujemy, któryś test nie przejdzie. Z jego nazwy lub ciała możemy domyślić się, który przypadek testowy zepsuliśmy.
- Testy to dobry sposób na dokumentowanie projektu. Dzięki testom szybko można zorientować się co jest zrobione a co nie.

Rodzaje testów automatycznych:

- testy jednostkowe
- testy integracyjne

Testy jednostkowe testują małe niezależne kawałki kodu. Zazwyczaj są to funkcje lub metody. Testy integracyjne testują większe elementy aplikacji np. całe klasy. Testy integracyjne mogą korzystać ze specjalnie spreparowanych do tego celu baz danych czy innych API.

Co testujemy a co nie:

- TESTUJEMY logikę biznesową aplikacji (pojęcie to sprecyzuje poniżej)
- NIE TESTUJEMY UI np. widoków (znaczy pewnie istnieją do tego jakieś mechanizmy, ale to nie temat na teraz 😊)
- NIE TESTUJEMY Kontrolerów np. w MVC (kontrolery obsługują połączenie i co najwyżej wywołują logikę w klasach od logiki biznesowej)
- NIE TESTUJEMY metod prywatnych (metody prywatne i tak są zapewne użyte jako składowe metod publicznych, więc zostaną przetestowane razem z nimi, a zresztą wymagało by to ustawienia ich na publiczne)
- NIE TESTUJEMY metod przypinanych pod eventy np. onclick przycisków (jeżeli w metodach tych znajduje się logika to znaczy, że należy ją wyciągnąć do zewnętrznych metod)
- NIE TESTUJEMY metod typu CRUD (dodających do bazy czy też usuwających bo to nie ma sensu)
- NIE TESTUJEMY statycznych metod (o tym niżej)

Oczywiście mogą zdarzyć się wyjątki.

Warunki jakie musi spełniać metoda aby była „testowalna”:

1. Metoda musi być napisana zgodnie z zasadą Single Responsibility Principle

Prosty wniosek. Nie da się przetestować metody, która coś liczy, odświeża UI, zapisuje do bazy dane i pobiera coś z API. Zasada pojedynczej odpowiedzialności ma duże znaczenie:

```
0 references
public class MyCalculator
{
    1 reference
    public void setValueField(int v)
    {
        //set ui
    }

    1 reference
    public void addHistoryEntry(int a, int b, int c)
    {
        //save something in database
    }

    0 references
    public void add(int a, int b)
    {
        int c = a + b;
        setValueField(c);
        addHistoryEntry(a, b, c);
    }
}
```

Powyższego kodu nie da się przetestować. Mowa o metodzie add(). Czemu? Ponieważ wykonuje ona, więcej niż jedną czynność. Oprócz dodawanie wykonuje ona również zapis do bazy danych i odświeżenie UI. Tak, te dwie czynności zostały zapisane jako osobne metody co w teorii spełnia SRP, ale nie do końca, ponieważ ich wywołania są NIEODŁĄCZNĄ częścią metodą add(). Zmodyfikujmy trochę ten kod:

```
2 references
public class DataBase
{
    1 reference
    public void addHistoryEntry(int a, int b, int c)
    {
        //save something in database
    }
}

2 references
public class MyCalculatorUI
{
    1 reference
    public void setValueField(int v)
    {
        //set ui
    }
}
```

```

0 references
public class MyCalculator
{
    private DataBase data = new DataBase();
    private MyCalculatorUI ui = new MyCalculatorUI();

    0 references
    public void add(int a, int b)
    {
        int c = a + b;
        ui.setValueField(c);
        data.addHistoryEntry(a, b, c);
    }
}

```

W tym momencie jest już trochę lepiej. Całe klasy są już całkiem zgodne z SRP. Jednak przetestowanie tego jest wciąż niemożliwe, ponieważ zarówno wywołanie setValueField() jest stałe jak i addHistoryEntry(). Co można zrobić? Zastosować wstrzykiwanie zależności:

```

2 references
public interface IDatabase
{
    2 references
    void addHistoryEntry(int a, int b, int c);
}

1 reference
public class DataBase : IDatabase
{
    2 references
    public void addHistoryEntry(int a, int b, int c)
    {
        //save something in database
    }
}

1 reference
public interface IMyCalculatorUI
{
    2 references
    void setValueField(int v);
}

2 references
public class MyCalculatorUI : IMyCalculatorUI
{
    2 references
    public void setValueField(int v)
    {
        //set ui
    }
}

```

Po pierwsze dodałem interfejsy. Po co? By zastosować bardziej efektywne wstrzykiwanie zależności.

```
1 reference
public class MyCalculator
{
    private IDatabase data;
    private IMyCalculatorUI ui;

    0 references
    public MyCalculator(IDatabase data, IMyCalculatorUI ui)
    {
        this.data = data;
        this.ui = ui;
    }

    0 references
    public void add(int a, int b)
    {
        int c = a + b;
        ui.setValueField(c);
        data.addHistoryEntry(a, b, c);
    }
}
```

W czasie normalnej pracy aplikacji poprzez konstruktor podamy normalne instancje klas DataBase oraz MyCalculatorUI, które normalnie będą wykonywać swoje zadanie.

W czasie testu w fazie Arrange podamy do konstruktora specjalne Mocki, które będą udawały, że coś robią albo będą mieć puste ciała. Ważne by spełniały interfejsy.

```
0 references
public class DataBase_Mock : IDatabase
{
    2 references
    public void addHistoryEntry(int a, int b, int c)
    {
        //nothing
    }
}
```

```
0 references
public class MyCalculatorUI_Mock : IMyCalculatorUI
{
    2 references
    public void setValueField(int v)
    {
        //nothing or some fake stuff
    }
}
```

Wykorzystaliśmy DI, polimorfizm oraz mocki do ułatwienia testowania metody add. Oczywiście powyższy kod nie jest za dobrze zaprojektowany. Normalnie nikt by tego tak nie napisał (ta sytuacja początkowa). To tylko przykład.

2. Dependency Injection zawsze i wszędzie.

DI to bardzo ważna zasada. W testach jako pokazał poprzedni przykład także ma ogromne znaczenie. Po prostu jeżeli klasa ma podmienianą zależność możemy ją podmienić na MOCK w trakcie testu co spowoduje pominięcie pewnych czynności np. zapisu do bazy.

Istnieją specjalne biblioteki, które generują obiekty MOCKi na bazie interfejsu. Podaje im się interfejs oraz informacje co ma zwracać każda metoda (z tych, które coś zwracają).

Wyobraźmy sobie MOCK klasy walidującej maila. Ma ona jedną metodę ValidateMail(string mail) zwracającą boola. Możemy kazać bibliotece wykonać atrapę, która z powyższej metody zawsze będzie zwracać true. Dzięki temu w samym teście nie musimy przejmować się walidacją maila a za to możemy się skupić na realnym celu testu.

3. NIE testujemy metod statycznych

Pola w klasie, które nie są statyczne są wspólne w obrębie każdego obiektu. **Jeżeli zrobimy w klasie prywatnego inta to każdy obiekt ma swojego. Jeżeli int ten jest statyczny to każdy obiekt widzi tego samego inta.**

Metody statyczne podobnie jak pola statyczne są wspólne dla całej klasy (całego programu).

```
0 references
class PointS //statyczna klasa
{
    public static int x = 10;
    public static int y = 2;

    0 references
    public static void SymetriaOY()
    {
        x *= (-1);
    }
}

0 references
class PointM //zwykła klasa
{
    public int x = 10;
    public int y = 2;

    0 references
    public void SymetriaOY()
    {
        x *= (-1);
    }
}
```

Mamy dwie klasy. Jedna jest statyczna a druga zwykła.


```

0 references
static void Main(string[] args)
{
    PointM o1 = new PointM(); //x = 10;
    o1.SymetriaOY(); //x = -10;
    PointM o2 = new PointM(); //x = 10;
    o2.SymetriaOY(); //x = -10;
}

```

W przypadku zwykłej klasy tworzymy obiekty. Każdy obiekt ma swojego X. Metoda wywołana na obiekcie wykonuje operacje na X tego konkretnego obiektu.

```

PointS.SymetriaOY(); // x = -10;

//somewhere else in code
PointS.SymetriaOY(); // x = ???

```

W przypadku statycznego podejścia wszystkie pola są przypisane do klasy i występują w aplikacji RAZ. Jeżeli raz wywołamy SymetriaOY() to wykonujemy operację na tym samym X w przypadku pozostałych wywołań (które mogą znajdować się w różnych miejscach projektu).

Tak samo jest z testami. Załóżmy, że mamy 10 metod testujących funkcję SymetriaOY(). W przypadku obiektowego podejścia każda z nich dostaje własny obiekt i po problemie. W przypadku statyczności każdy test modyfikuje tego samego X. W C/C++ sytuację taką nazwaliby „undefined behaviour”. Nie mamy żadnej pewności, który test wykonuje się pierwszy, który ostatni. Nie możliwe jest określenie w takim przypadku na początku testu jaką wartość ma X.

Metody statyczne można testować jeżeli są na tyle hermetyczne, że nie korzystają z zewnętrznych zmiennych (korzystają tylko z lokalnych zmiennych i argumentów).

Bardzo ważną zasadą w testowaniu jest to, że każdy test MUSI być niezależny od pozostałych.

```

0 references
public class UnitTest1
{
    [Fact]
    0 references
    public void DivideTest_DivideTwoIntegers_ShouldWork()
    {
        //Arrange
        Calculator calculator = new Calculator();

        //Act
        double result = calculator.divide(6,2);

        //Assert
        Assert.Equal(3, result);
    }
}

```

Tutaj wszystko jest ok. Metoda testująca dostała własny obiekt, na którym wykonuje test. Jeżeli metoda divide() była by statyczna, oznaczałoby to, że każda metoda posiada tę samą sztukę. Jeden test wpływałby na drugi. Oczywiście w przypadku kalkulatora nie ma

problemu bo `divide()` nie korzysta z pól oraz nie zapisuje w nich danych. Ale przypadków gdzie ma to znaczenie jest cała masa.

Jakie powinny być testy:

- test musi zawsze dać ten sam wynik, jeżeli testowany kod się nie zmienił
- testy muszą być między sobą niezależne (jeden nie może wpływać na drugi)
- testy muszą być szybkie (nie piszmy testu, który wykonuje się bardzo długo bo logika testowana trwa dużo czasu)
- każdy test powinien być jasno napisany (powinien testować konkretną rzecz i rzecz ta powinna być łatwa do określenia patrząc na ciało testu)

Jak nazywać testy:

Na pewno nie „test1” itd.

Testy nazwa się zazwyczaj `snake_case` według wzoru:

`CoTestuje_CoSprawdzam_JakiegoWynikuSieSpodziewamy();`

Np.

`Calculator_DivideMethod_DivideByZero_ShouldThrowDivideByZeroE();`

- testujemy metodę `Divide` w klasie `Calculator`
- podajemy jej dzielnik 0
- spodziewamy się, że rzuci wyjątek `DivideByZeroException`

Co testować:

- testujemy jeden dwa zwykłe przypadki
- testujemy skrajne/mniej oczywiste przypadki
- testujemy przypadki trudniejsze do sprawdzenia manualnie

Assertcje:

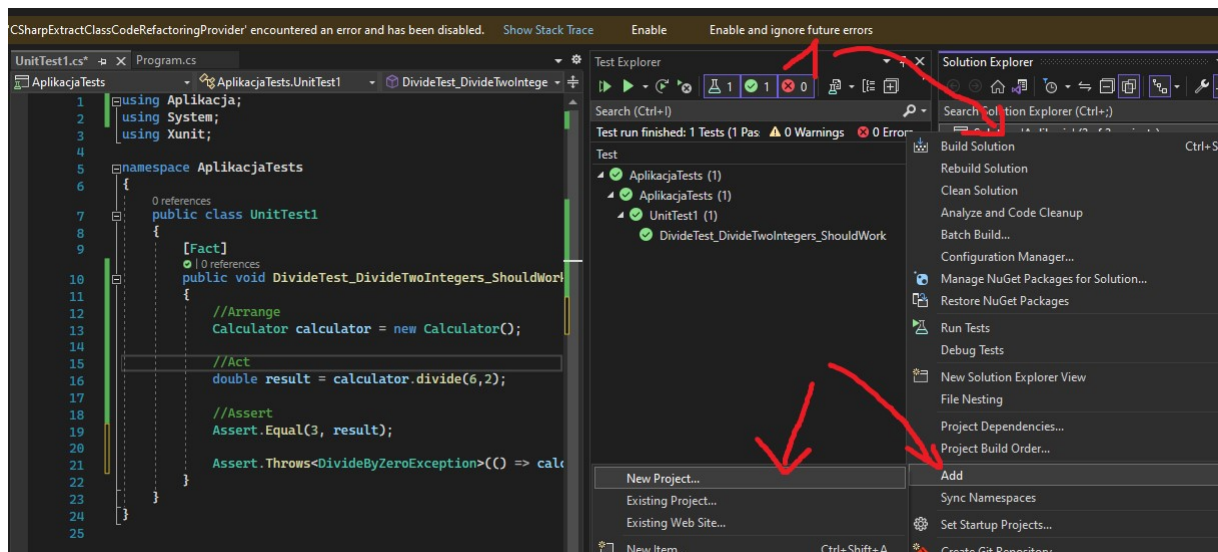
Istnieje wiele asercji np. `isEqual()`, `isTrue()`, `isFalse()`, `isEmpty()`, `isNull()`, `contains()` i wiele innych. Istnieją też asercje Sprawdzające czy poleciał wyjątek:

```
Assert.Throws<DivideByZeroException>(() => calculator.divide(6, 2));
```

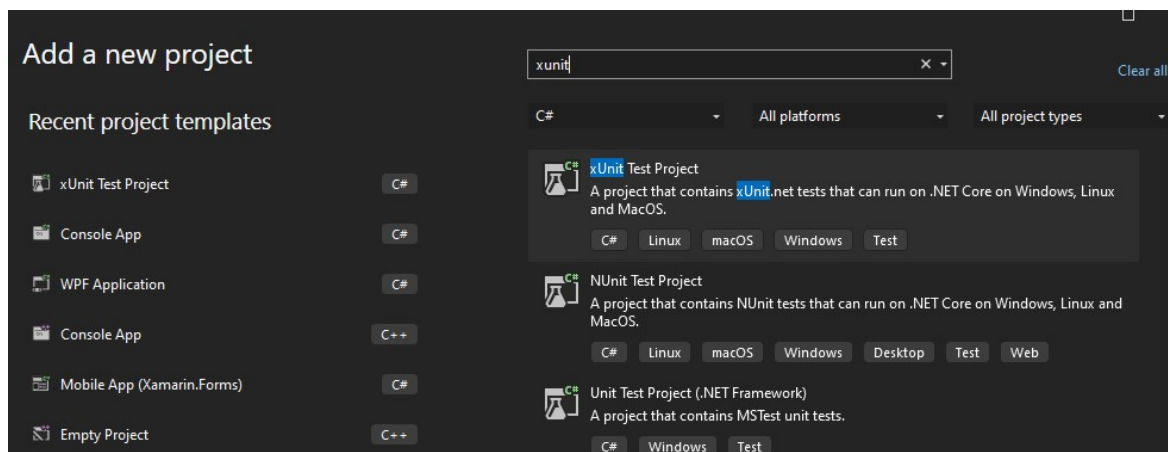
Wszystko zależy od języka i frameworka testującego.

PRZYKŁADY:

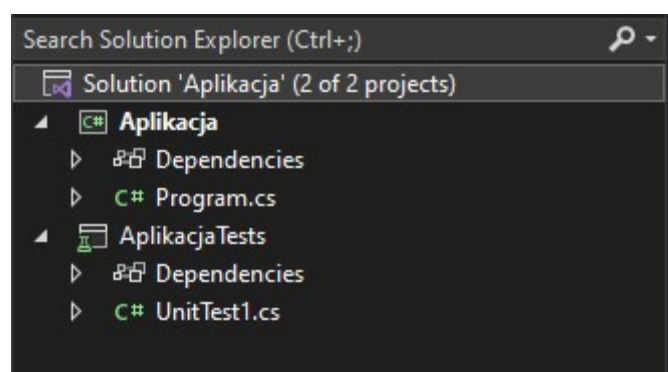
Przykłady testu w C# oraz JS:



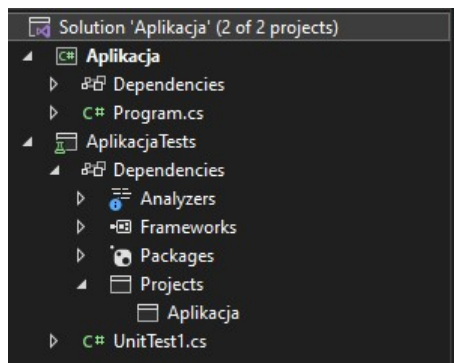
Jak dodać testy w C#? Klikamy prawym na solucji (1) i dodajemy nowy projekt do solucji (2).



Projekt można nazwać tak samo jak projekt docelowy z dopiskiem Tests. Ja wybieram zawsze Xunit, bo wole go względem NUnita.



Projektowi z testami trzeba dodać zależność w powyższym oknie do projektu docelowego:



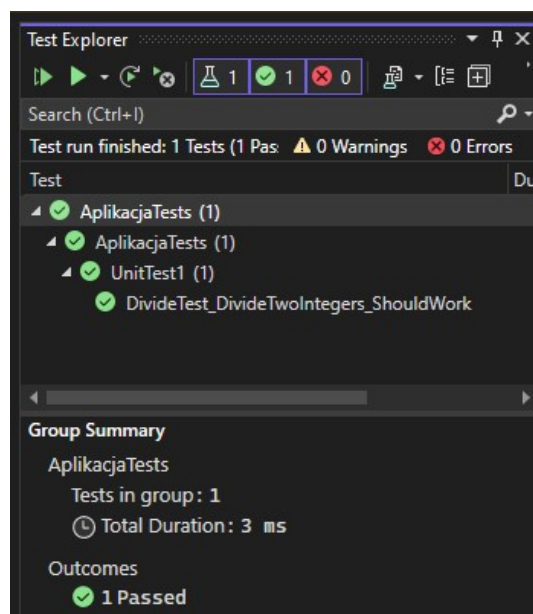
Polecam zmienić nazwę klasy testującej na coś znaczącego. Każda metoda w tej klasie to osobny test oznaczony [Fact]

```
0 references
public class UnitTest1
{
    [Fact]
    public void DivideTest_DivideTwoIntegers_ShouldWork()
    {
        //Arrange
        Calculator calculator = new Calculator();

        //Act
        double result = calculator.divide(6,2);

        //Assert
        Assert.Equal(3, result);
    }
}
```

Eksplorator testów znajduje się w zakładce Tests. Jego zawartość odświeża się po wykonaniu kompilacji obu projektów.



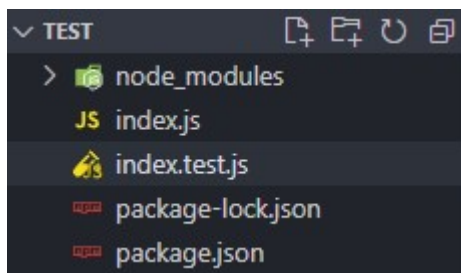
W przypadku niepowodzenia w oknie summary pokazywane są liczne informacje diagnostyczne.

W jsie sytuacja wygląda tak:

npm i jest --save-dev

npm i jest-cli --g --save-dev

Polecenie jest uruchamia testy, zawarte w plikach x.test.js gdzie x to nazwa celu:



Plik z kodem:

```
JS index.js > ...
1  const divide = (a, b) => {
2      if(b == 0){
3          throw 'DivideByZero';
4      }
5
6      return a / b;
7  }
8
9  module.exports = divide;
```

Plik z testami:

```
index.test.js > ...
1  const divide = require('./index.js')
2  test("Calculator_Divide_TwoIntegers_ShouldWork", () => {
3      expect(divide(6,2)).toBe(3);
4  });
```

Funkcja test() rejestruje test o określonej nazwie. Ciało testu definiuje lambda. Funkcja expect() pobiera argument w postaci wartości aktualnej (zwróconej z testowanego kodu). Po kropce wykonać można odpowiednią asercję Equal a mianowicie toBe().

Odpalenie testów wykonujemy poleceniem **jest**.

```
C:\Users\siedl\Desktop\PR\test>jest
PASS ./index.test.js
  ✓ Calculator_Divide_DivideTwoIntegers_ShouldWork (1 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.664 s
Ran all test suites.
```

Jest pozwala grupować testy:

```
describe("NormalCases_ShouldWork", ()=>{
  it("Divide6by2", () =>{ //pierwszy test w grupie
    expect(divide(6,2)).toBe(3);
  }),
  it("Divide12by8", () =>{ //drugi test w grupie
    expect(divide(12,8)).toBe(1.5);
  })
})

describe("DivideByZero_ShouldThrow", ()=>{
  it("Divide6by0", () =>{ //pierwszy test w grupie
    expect(() => {divide(6,0)}).toThrow('DivideByZero');
    //lamda w expect() jest bardzo ważna w przypadku testu typu throw
  })
})
```

Zarejestrowałem dwie grupy. Pierwsza posiada dwa standardowe testy (ciężko coś wymyślić pod kalkulator).

Grupa druga sprawdza czy poleci wyjątek w przypadku podania do testu dzielnika równego 0. **Tutaj ważne by ubrać testowaną funkcję w lambde, ponieważ inaczej jest uzna, że wyjątek leci w samym teście i się wyrzyczy.**

```
C:\Users\siedl\Desktop\PR\test>jest
PASS ./index.test.js
  NormalCases_ShouldWork
    ✓ Divide6by2 (2 ms)
    ✓ Divide12by8
  DivideByZero_ShouldThrow
    ✓ Divide6by0 (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.679 s, estimated 1 s
Ran all test suites.
```