TDD

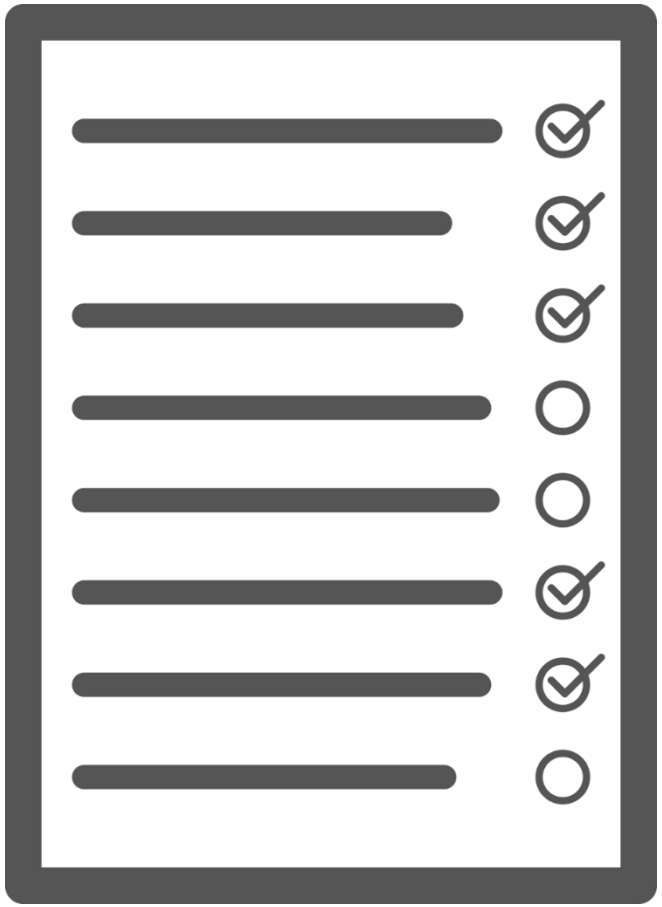
Test-Driven Development

Mateusz Turzyński, Michał Michalczuk
08.11.2017

goyello

Testy?

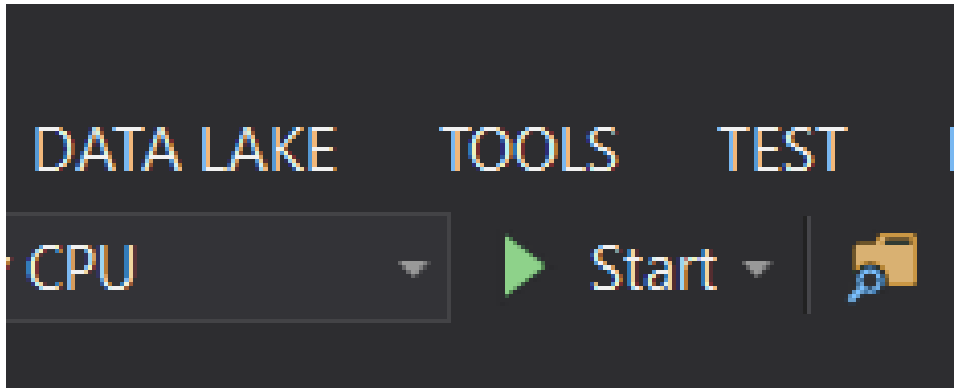
Czym jest testowanie oprogramowania?



Upewnienie się, że program działa **zgodnie z wymaganiami**

Testy?

Uruchom aplikację – sprawdź czy działa poprawnie



I tak wykonujemy testy.

Zautomatyzujmy ten process.

Testy?

Na co wpływają testy

```
[TestFixture]
public class Password
{
    [Test]
    public void IsVal
    {
        // Arrange
        var target =
```

- Prostszy kod
- Architektura
- Luźne powiązania
- Bezpieczniejsza modyfikacja kodu

Testy Jednostkowe

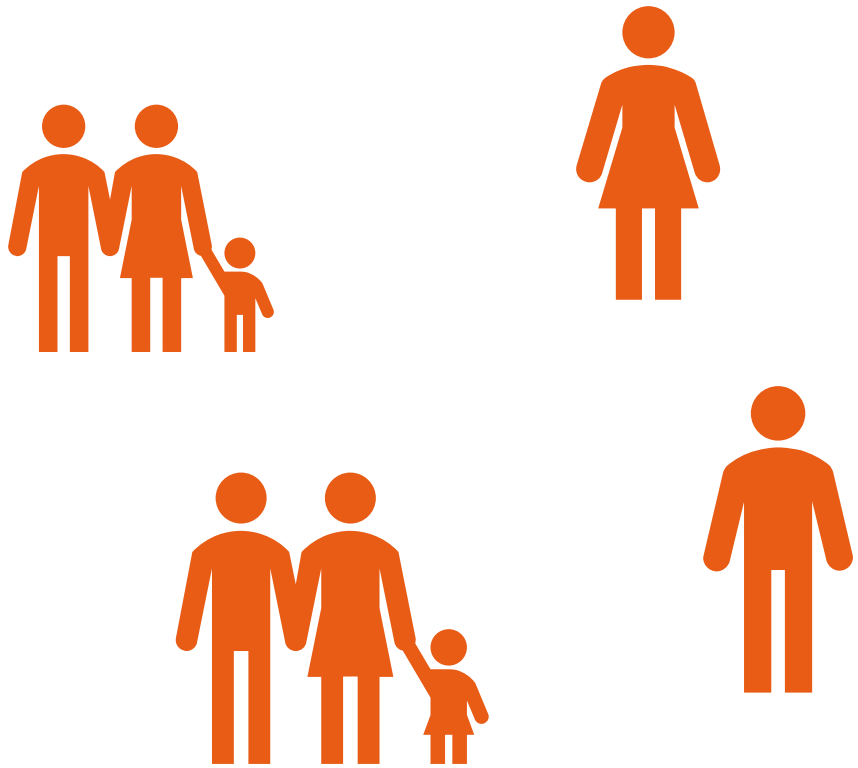
Sprawdzamy jedną rzecz

- Tylko jedna klasa
- Tylko jedna metoda
- Tylko unit

Izolujemy dokładnie jedną rzecz.

Testy Integracyjne

Sprawdzamy jak rzeczy ze sobą współpracują



- Interakcja wielu klas
- Czy klasy wspólnie poprawnie działają

Możemy sprawdzić czy całe moduły system działają wspólnie.

Testy Jednostkowe

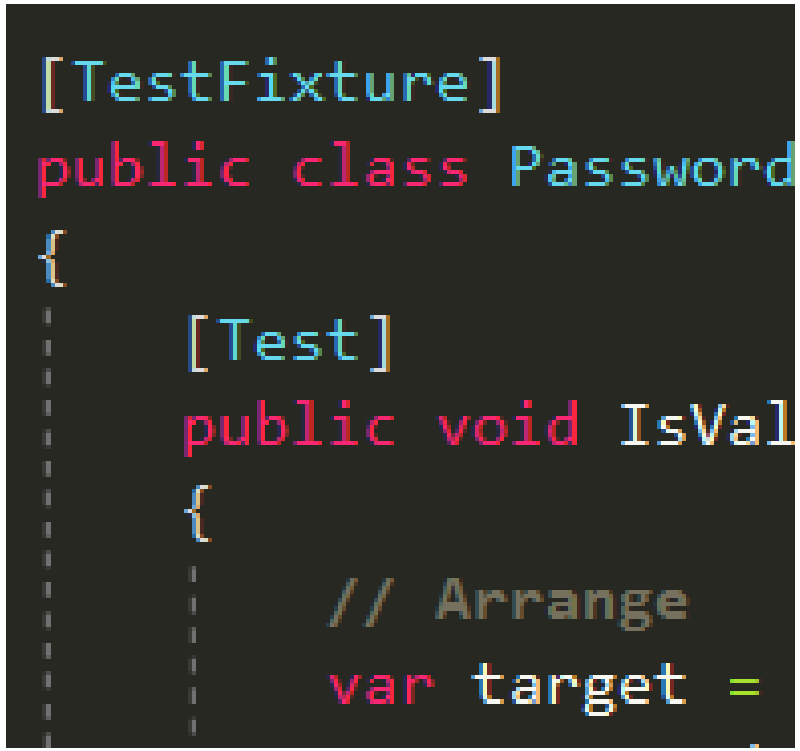
Sprawdzamy jedną rzecz

- Tylko jedna klasa
- Tylko jedna metoda
- Tylko unit

Izolujemy dokładnie jedną rzecz.

Testy jednostkowe w C#

Wsparcie framworków



- **nUnit**
- xUnit
- MSTest
- …

- Pierwotnie port z *Junit*
- Wszystkie języki .net
- Aktualnie: v.3
- Open Source

http://nunit.org/

Struktura testu

```
[TestFixture]
public class AwesomeFeatureTests
{
    [Test]
    public void ShouldReturnPointsSum()
    {
        // Arrange
        var target = new AwesomeFeature();

        var awesomePoints = new List<int>
        {
            1, 2, 10, 20
        };

        var expected = awesomePoints.Sum();

        // Act
        var actual = target.CountAwesome(awesomePoints);

        // Assert
        Assert.That(actual, Is.EqualTo(expected));
    }
}
```

Klasa testowa

Metoda testowa (jeden przypadek)

Klasa testowana.

Tworzymy instancję.

Wykonaj test

Sprawdź wyniki

Struktura testu

```
[TestFixture]
public class AwesomeFeatureTests
{
    private AwesomeFeature _target;

    [SetUp]
    public void SetUp()
    {
        _target = new AwesomeFeature();
    }

    [Test]
    public void ShouldReturnPointsSum()
    {
        // Arrange
        var awesomePoints = new List<int>
        {
            1, 2, 10, 20
        };

        var expected = awesomePoints.Sum();
```

SetUp

Wykona się przed każdym testem

Metod testowych może być dużo.

Po jednej na przypadek.

Jak sprawdzicie działanie tego programu?

Setup projektów

Kiedy pisać testy w projekcie

Projekt powstający bez testów

Projekt z testami napisanymi na koniec

Test Driven Development

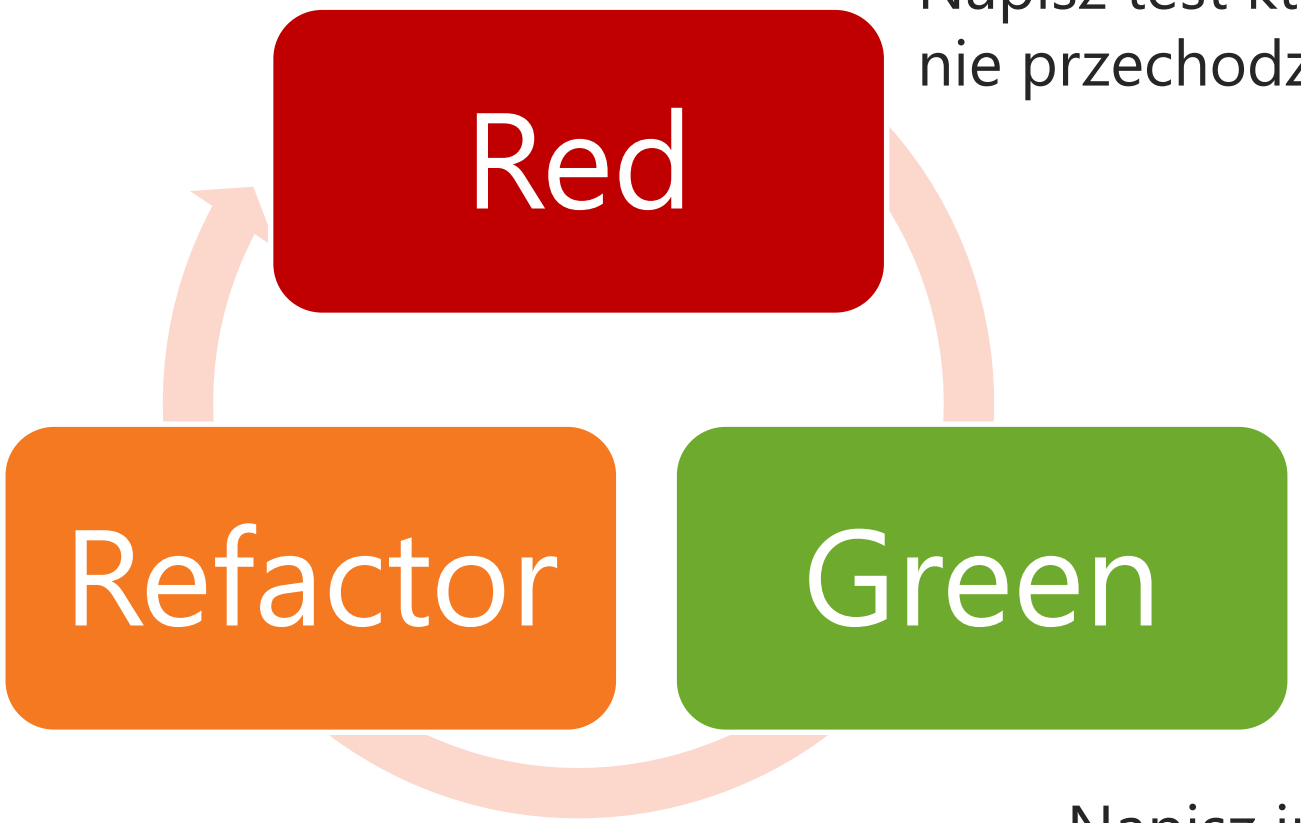
Idź za testem

- Testy jako **element** procesu
- Najpierw testy
- Testy wpływają na architecture
- Testy przyśpieszają dalszą pracę

Red Green Refactor

Cykl pracy

Napisz test który nie przechodzi.



Refactoring.
Poprawiamy.

Napisz implementację.
Test przechodzi.

Co testować?

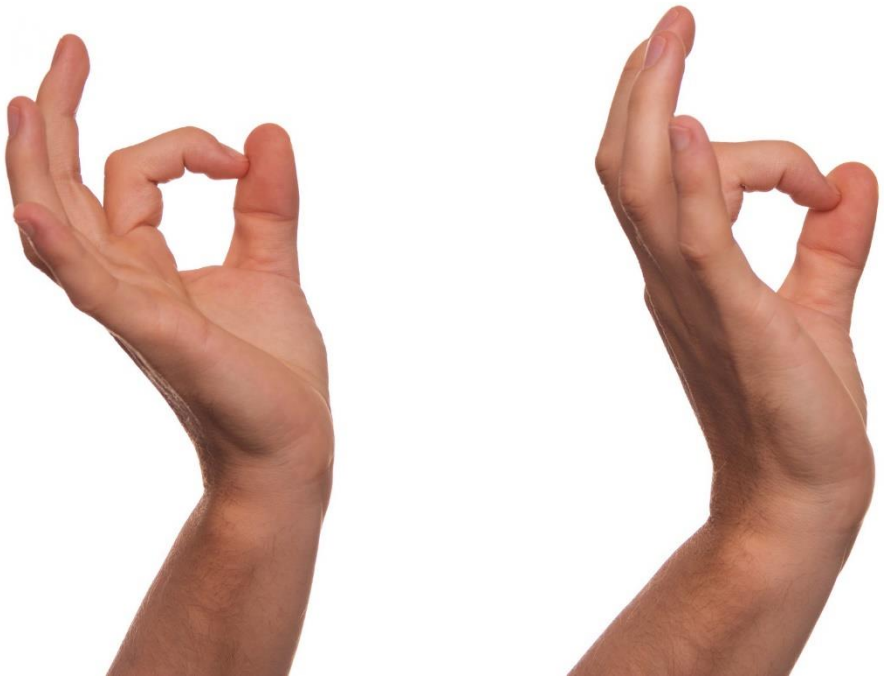
Testuj

- Logika
- Zmiana stanu
- Przepływ danych

Nie ma sensu

- Konstruktory
- Kod deklaratywny

Dobry test jednostkowy to ...



- Jeden przypadek
- Jedna asercja
- Dla jednej klasy
- Niezależny od innych klas
- Niezależny od **innych testów**

Testy mogą wykonywać się w **dowolnej** kolejności.

Popracujmy w parach

TDD, pair programming

Tworzymy kod przez kompozycję.

Większość naszych klas wykorzystuje inne klasy do działania.

- Jak je testować?
- Czy to dalej testy jednostkowe?

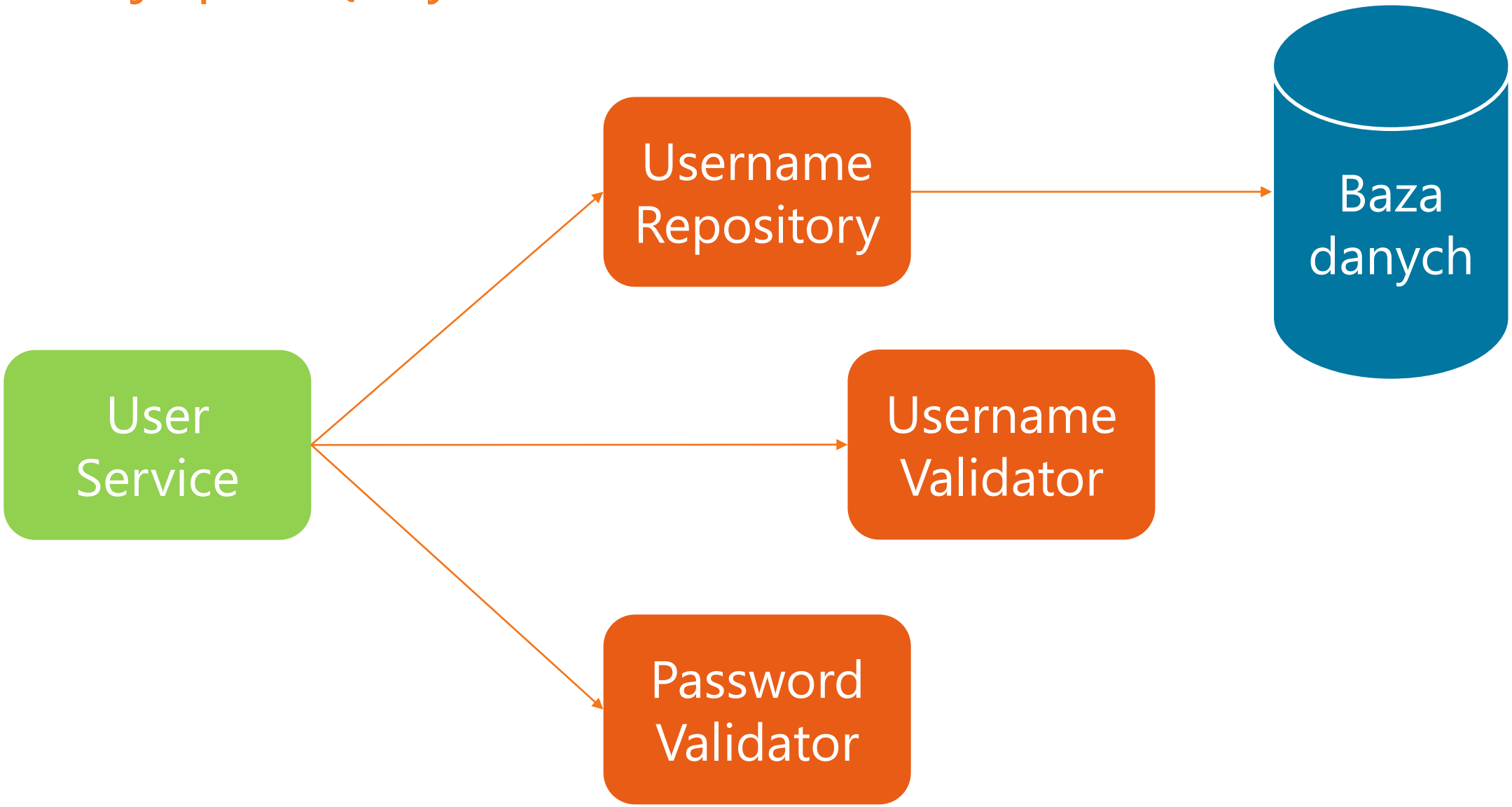
Jeśli nie użyjemy implementacji klas zależnych

To wciąż testy jednostkowe.

Dalej testujemy tylko jedną klasę.

Oraz kontrakty pomiędzy klasami.

Relacja pomiędzy klasami



Relacja pomiędzy klasami

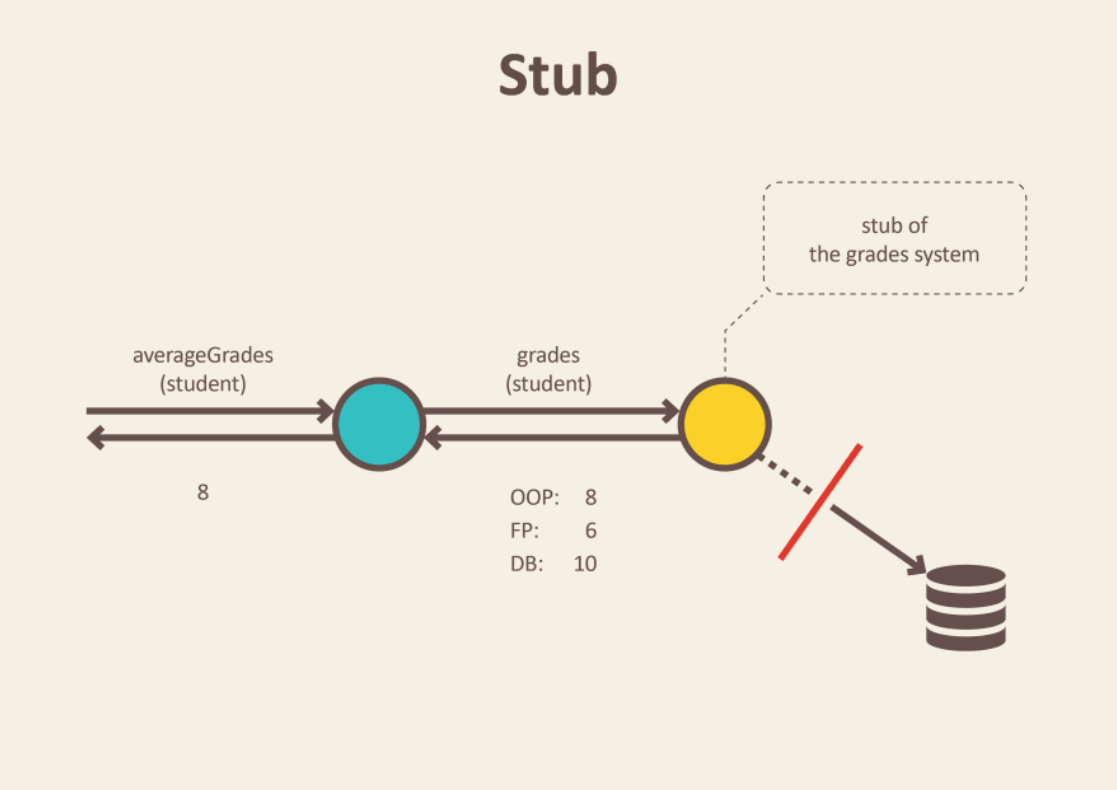
```
public class UserService
{
    private readonly IUsernameValidator _usernameValidator;
    private readonly IPasswordValidator _passwordValidator;
    private readonly IUsernameRepository _usernameRepository;

    public UserService(IUsernameValidator usernameValidator, IPasswordValidator passwordValidator,
        IUsernameRepository usernameRepository)
    {
        _usernameValidator = usernameValidator;
        _passwordValidator = passwordValidator;
        _usernameRepository = usernameRepository;
    }
```

...

Zwróćcie uwagę na to że mamy interfejsy.

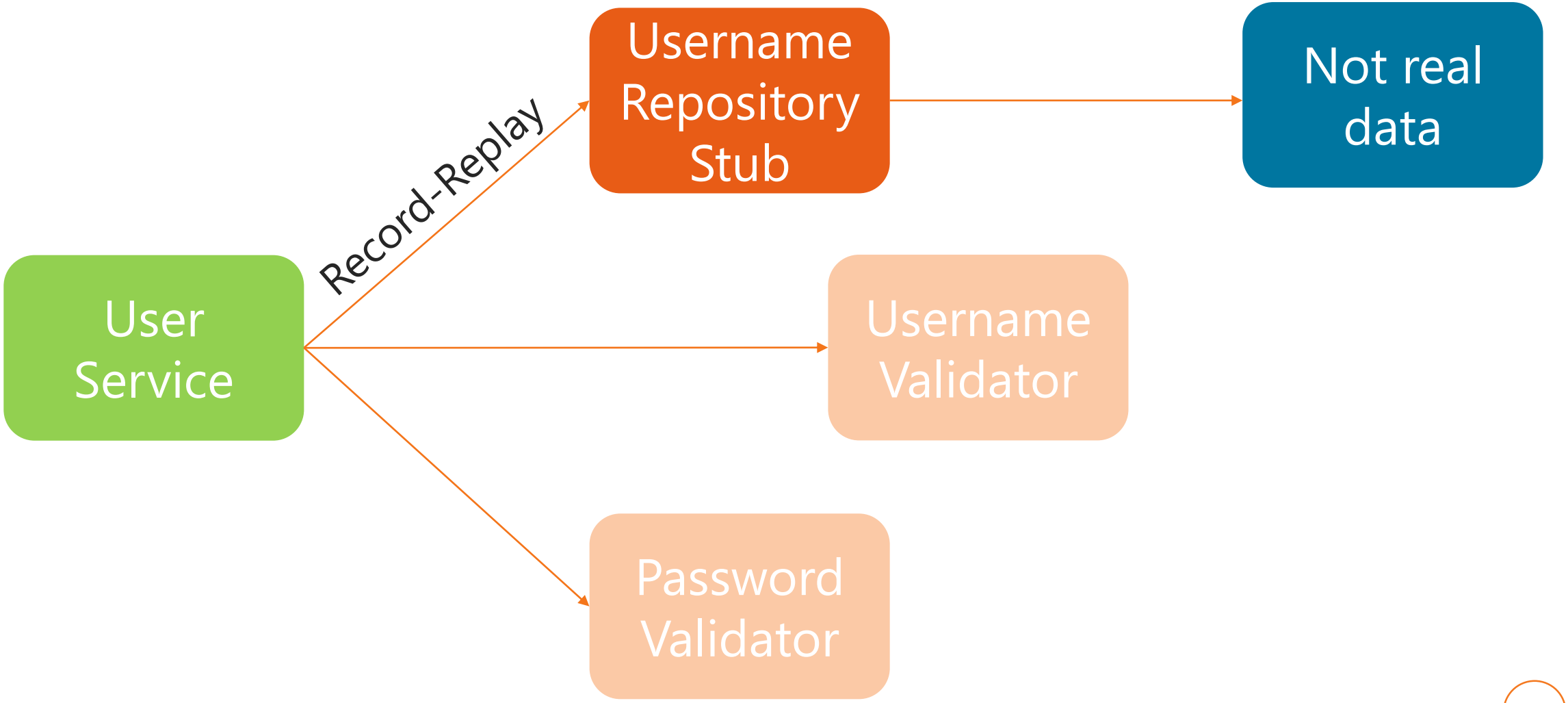
Stub



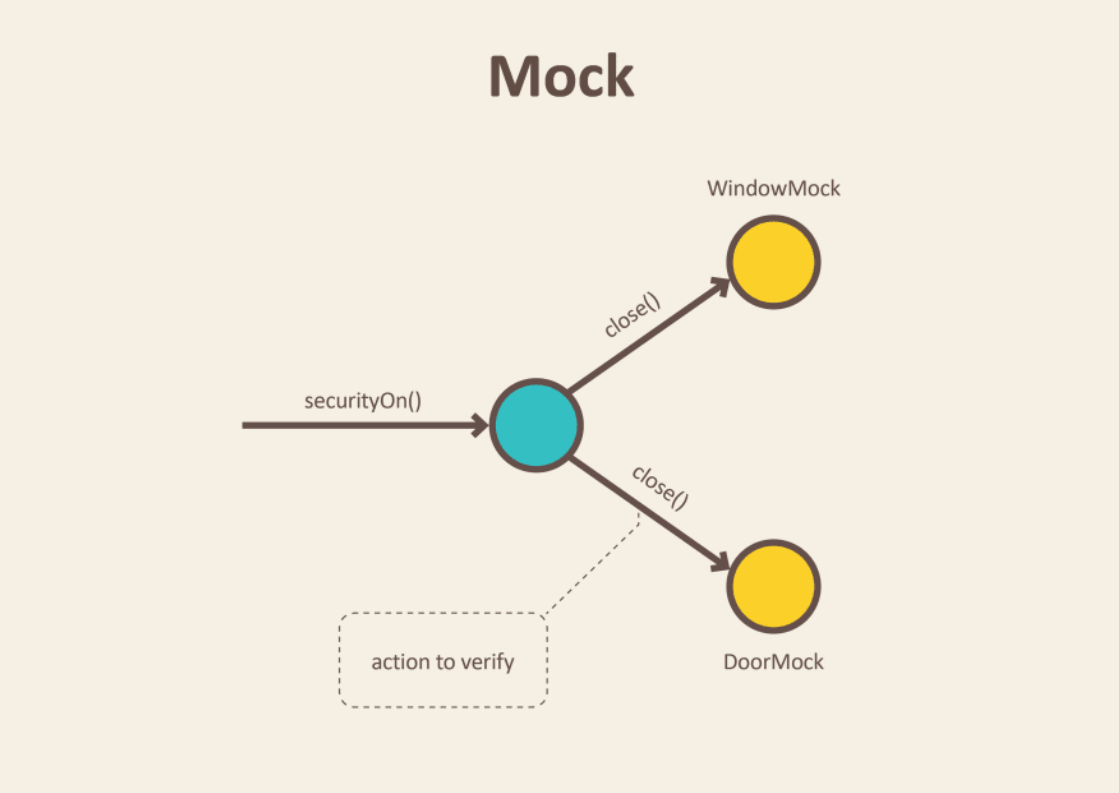
https://dev.to/milipski/test-doubles---fakes-mocks-and-stubs

- Nie chcemy pobierać prawdziwych danych
- Nie chcemy korzystać z implementacji naszych zależności
- Sami „nagrajmy” co mają nam zwracać

Stub


Username Repository Stub
Not real data
Record-Replay
User Service
Username Validator
Password Validator

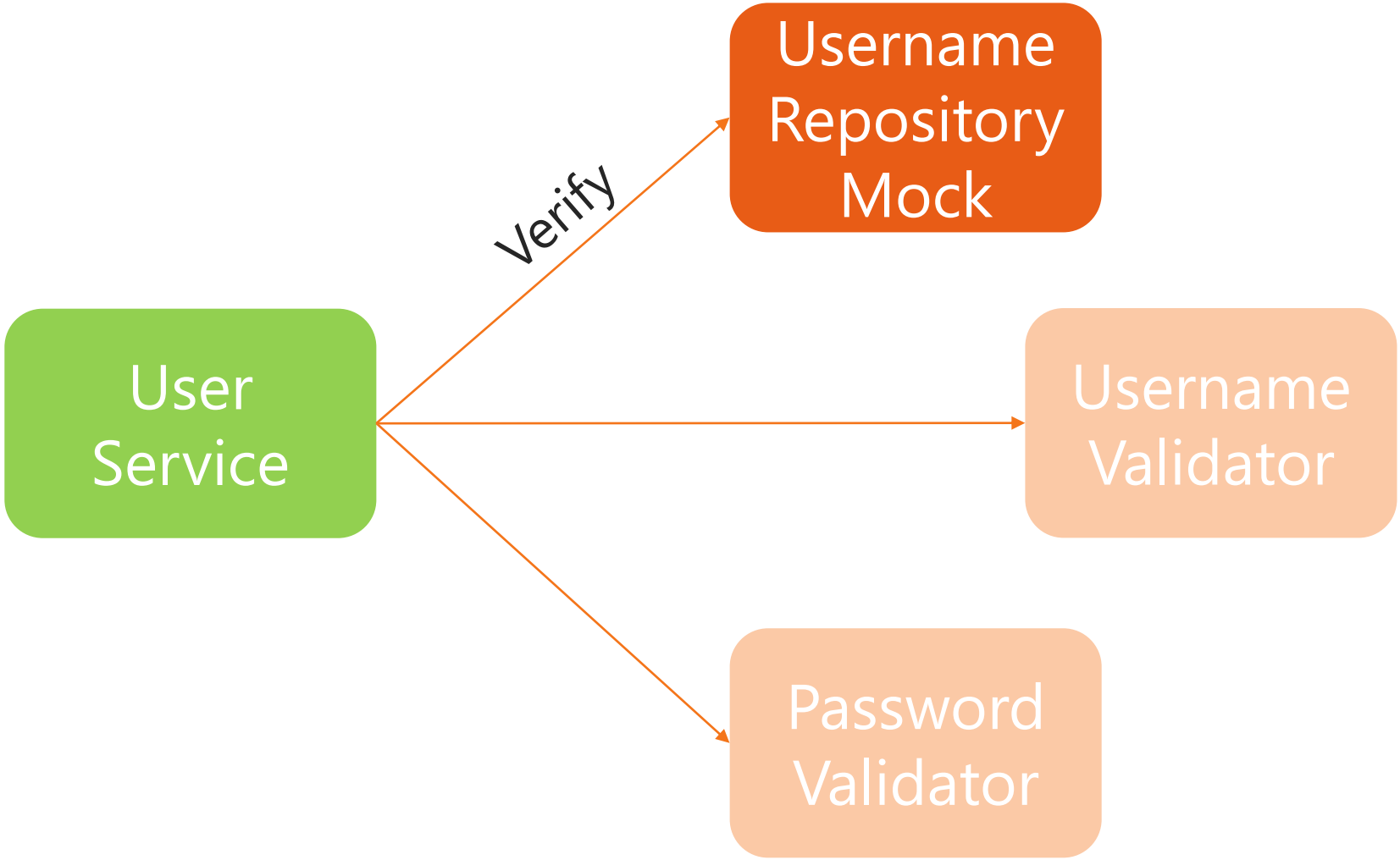
Mock



https://dev.to/milipski/test-doubles---fakes-mocks-and-stubs

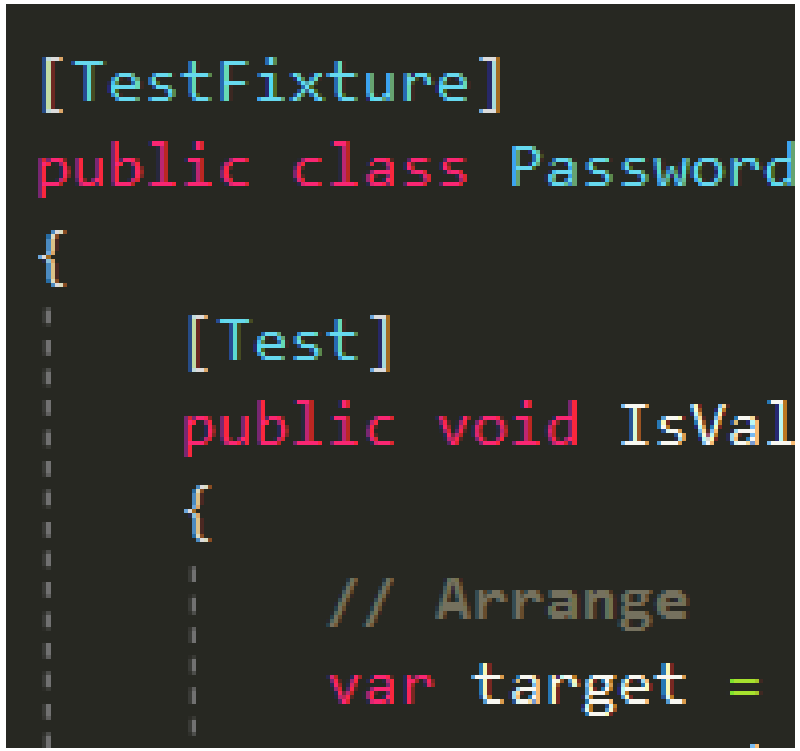
- Czasem wystarczy wytestować czy wykonaliśmy metodę
- Np. zapis – sprawdź czy został wykonany … lub czy nie został

Mock


Username
Repository
Mock
Verify
User
Service
Username
Validator
Password
Validator

Testy jednostkowe w C#

Biblioteki do “izolacji” wspierające framworki



- **Moq**
- FakeItEasy
- Microsoft Fakes
- NMock
- …

Przydadne linki

MS Academy: Test-Driven Development
https://mva.microsoft.com/en-US/training-courses/testdriven-development-16458

Unit testing frameworks in C# compartion
https://raygun.com/blog/unit-testing-frameworks-c/

The Art. Of Unit Testing
http://artofunittesting.com/

Nasze repozytorium:

https://github.com/mturzynskigoyello/tdd

Dziękujemy za uwagę

Mateusz Turzyński

mturzynskigoyello

mateusz.turzynski@goyello.com

Michał Michalczuk

michalczukm

michal.michalczuk@goyello.com