

# Testy jednostkowe i behawioralne

## Projektowanie i programowanie systemów internetowych II

---

mgr inż. Krzysztof Rewak

18 października 2018

Wydział Nauk Technicznych i Ekonomicznych

Państwowa Wyższa Szkoła Zawodowa im. Witelona w Legnicy

# Plan prezentacji

1. Testy jednostkowe
2. Testy behawioralne
3. Selenium
4. Test Driven Development
5. Podsumowanie

# Jak ugryźć testowanie?

Testy są **niezbędną** częścią procesu wytwarzania oprogramowania.

# Jak ugryźć testowanie?

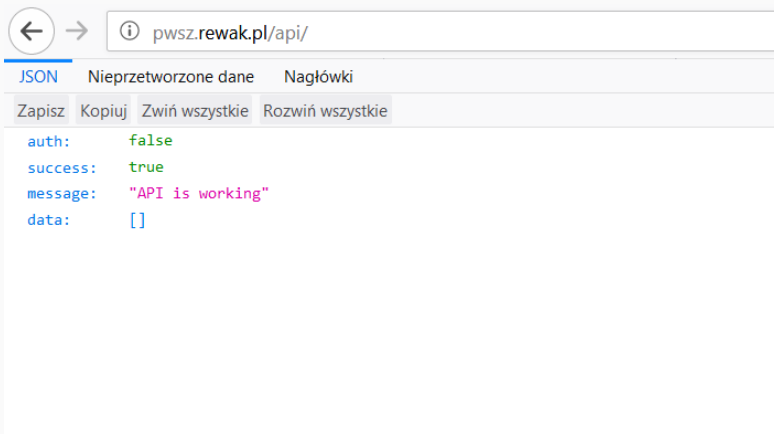
Trudno sobie wyobrazić programowanie *na ślepo*: tworzenie nowych funkcjonalności bez sprawdzania czy faktycznie działają.

Za niedziałający program pracodawca mógłby rozwiązać umowę, klient - nie zapłacić pieniędzy, a nauczyciel - wystawić negatywną ocenę.

# Jak ugryźć testowanie?

Testy są **niezbędną** częścią procesu wytwarzania oprogramowania i wykonujemy je bardzo często bezwiednie.

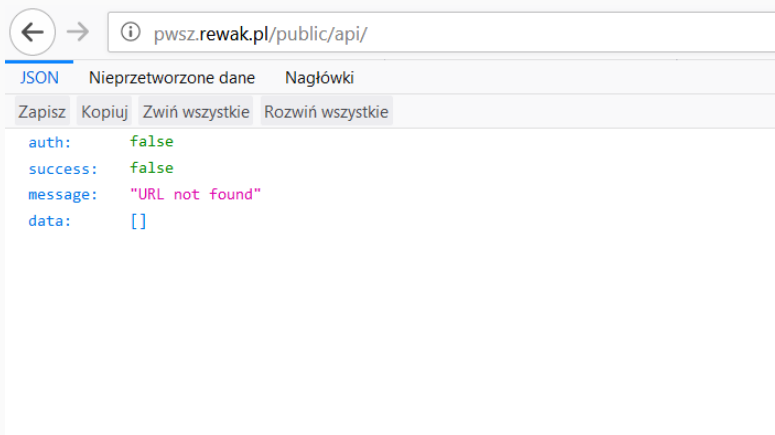
# Jak ugryźć testowanie?



# Testowanie manualne

Testowanie manualne jest *najprostszym* sposobem testowania oprogramowania i polega na ręcznym przeklikaniu aplikacji w celu znalezienia błędów.

# Testowanie manualne



The screenshot shows a web browser window with a REST client interface. The address bar displays the URL `pwsz.rewak.pl/public/api/`. Below the address bar, there are tabs for `JSON`, `Nieprzetworzone dane`, and `Nagłówki`. The `JSON` tab is selected. Below the tabs, there are buttons for `Zapisz`, `Kopiuj`, `Zwiń wszystkie`, and `Rozwiń wszystkie`. The main area displays the JSON response of the API call:

```
auth: false
success: false
message: "URL not found"
data: []
```



Problem pojawi się oczywiście w momencie, w którym zwiększy się liczba testów potrzebnych do sprawdzenia stanu aplikacji.

# Testowanie automatyczne

Stąd wziął się pomysł na **testowanie automatyczne**.

# Testy jednostkowe

---

# Testy jednostkowe

Test jednostkowy sprawdza czy metoda lub funkcja wykonuje się w oczekiwany sposób. Sprawdzenie warunku nazywa się **asercją**.

# Co testować?

Oto prosta metoda dzieląca liczbę przez liczbę:

```
public static function
divide(float $dividend, float $divisor): float {
    if($divisor == 0) {
        throw new DoNotDivideByZeroException();
    }

    return $dividend / $divisor;
}
```

# Jak testować?

Czy to jest wystarczający test?

```
public function testDividingFunction(): void {  
    $this->assertEquals(Math::divide(1.0, 2.0), 0.5);  
}
```

# Jak testować?

A to?

```
public function testDividingFunction(): void {  
    $this->assertEquals(Math::divide(1.0, 2.0), 0.5);  
    $this->assertEquals(Math::divide(2.0, 4.0), 2.0);  
}
```

# Jak testować?

A może to?

```
public function testDividingFunction(): void {  
    $this->assertEquals(Math::divide(1.0, 2.0), 0.5);  
    $this->assertEquals(Math::divide(2.0, 4.0), 2.0);  
    $this->expectException(DoNotDivideByZeroException::class);  
    Math::divide(1.0, 0.0);  
}
```



# Jak testować?

A gdyby tak...?

```
public function testDividingFunction(): void {  
    $this->assertEquals(Math::divide(1.0, 2.0), 0.5);  
    $this->assertEquals(Math::divide(2.0, 4.0), 2.0);  
    $this->assertNotEquals(Math::divide(2.0, 4.0), 1.0);  
    $this->expectException(DoNotDivideByZeroException::class);  
    Math::divide(1.0, 0.0);  
}
```

# Jak testować?

A co, jeżeli przekazemy zmianną typu `int`?

A co, jeżeli podzielimy coś, co powinno zwrócić na przykład 3.(3)?

A co, jeżeli...?

# Testy jednostkowe w praktyce

Jedna asercja na test czy wiele na funkcjonalność? Dobrą praktyką jest testowanie osobno, jednakże wybór może zostać wymuszony (w obie strony) przez specyfikę projektu.

# Testy jednostkowe w praktyce

Ogólnie przyjęte zasady mówią, że testować jednostkowo powinno się wszystkie publiczne metody klas w projekcie.

Oczywiście stuprocentowe pokrycie testami może być czymś do czego warto dążyć, ale niektóre metody (choćby proste settery i gettery) zwyczajnie nie potrzebują być testowane.

# Kiedy testować?

Testy jednostkowe najlepiej uruchamiać... zawsze.

Przy commitach, przy merge requestach, przy wdrożeniu. Najlepiej zautomatyzować proces wdrożeniowy w taki sposób, aby przejście przez wszystkie testy było obowiązkowe przed wprowadzeniem zmian na serwerze.

# Mockowanie

Czasami chcemy przetestować coś, co jest uwikłane w sieć zależności. Wyobraźmy sobie serwis, w którym używana jest klasa pobierająca listę użytkowników z bazy danych i wysyłają im newsletter poprzez mejla.

Karygodnym błędem byłoby przetestowanie tego na danych produkcyjnych i wysłanie każdemu mejla przy każdym teście, prawda?

# Mockowanie

Dlatego do testowania wygodne jest wykorzystanie wzorca projektowego odwracania zależności. Wówczas można z poziomu testu wywołać potrzebny serwis z *mockiem* klasy pobierającej dane czy wysyłającej mejle.

Taki *mock* (ang. atrapa) będzie symulowanym odzwiedleniem wymaganej funkcjonalności. Przykładowo gdy `users.all()` odwołuje się do bazy danych i zwraca tablicę użytkowników, mockowany obiekt `users` klasy `MockUsers` może dla metody `all()` wypisać zahardkodowane dane.

# Mockowanie

Które rozwiązanie będzie lepsze do mockowania obiektów?

```
public function send(): void {  
    $mailer = new Mailer;  
    $mailer->send();  
}
```

czy może:

```
public function send(Mailer $mailer): void {  
    $mailer->send();  
}
```

czy może:

```
public function send(MailerInterface $mailer): void {  
    $mailer->send();  
}
```



Ostrzeżenie: młodych programistów kusi czasami testowanie losowanych danych:

```
public function testDividingFunction(): void {  
    for($i = 0; $i < 100; $++) {  
        $a = rand(); $b = rand();  
        $this->assertEquals(Math::divide($a, $b), $a / $b);  
    }  
}
```

Istotą testowania jest sprawdzenie jak program zadziała w kontrolowanych warunkach. Jeżeli programista chce sprawdzić inne warunki, musi stworzyć nowy test z konkretnymi wartościami.

# Czym testować?

- PHP: PHPUnit, PHP Unit Testing Framework, Behat
- JavaScript: Mocha, Unit.js
- .NET: csUnit, NUnit, Visual Studio Unit Testing Framework
- Python: Doctest, pytest
- Java: JUnit
- go: go test
- Ruby: RSpec
- C++: CppUnit

# Testy behawioralne

---

# Testy behawioralne

Czasami zdarza się, że klient aktywnie bierze udział przy tworzeniu projektu. Oczywiście nie programuje, ale przecież to od niego zależy co będzie wykonywała opracowywana aplikacja.

Klient powinien znać domenę projektu, a więc powinien znać wszystkie realia środowiskowe.

# Testy behawioralne

O ileż prostsze byłoby programowanie, gdyby klient zamiast opowiadania o wymaganiach, mógłby rozpisać scenariusze zachowań aplikacji?

# Testy behawioralne

Problem ten pomaga rozwiązać idea testów behawioralnych.

Przykładowy test może wyglądać następująco:

Scenario: Checking if retrieving newsreel returns correct result:

When a client requests "/api/news" with "GET" method

Then "200" status code should be received

And proper response array should be received

And response array should have success status

And response array should have empty message

And response array should not have empty data array

And there should be "3" news entries

And received news should be arranged in chronological order

Przykładowo realizacja zdania:

```
Then "200" status code should be received
```

może wyglądać następująco:

```
public function  
statusCodeShouldBeReceived(string $statusCode): void {  
    $responseStatusCode = (string) $this->response->getStatusCode();  
    PHPUnit::assertEquals($responseStatusCode, $statusCode);  
}
```



# Testy behawioralne

Korzystając ze słów kluczowych Given, When i Then można stworzyć *historyjki* opisujące każdy przypadek omawianej aplikacji.

# Testy behawioralne

Testy behawioralne *pod spodem* korzystają z asercji znanych z testów jednostkowych, jednakże są wygodniejsze do planowania i czytania dla osób nietechnicznych lub nie będących programistami. Język Gherkin jest uniwersalny i może być stosowany do kontekstów napisanych w Ruby (Cucumber), PHP (Behat), Java (YatSpec/Concordion), .NET (SpecFlow) i innych.

# Selenium

---

# Testy regresyjne

Przedstawione przykłady świetnie się wpisują do testowania backendu.

A co jeżeli chcemy przeklikać frontend naszej aplikacji?

Z pomocą przychodzi Selenium, czyli narzędzie do automatyzacji testów. Pozwala ono na *nagranie* interakcji z interfejsem aplikacji i następnie powtarzanie go w ramach testowania funkcjonalności.

# Testy regresyjne

moz-extension://998e2cee-c46e-44b9-9769-c353f0ff5c8a - Selenium IDE - PWSZ\* - Mozilla Firefox

Project: PWSZ\*

Tests +

Search tests...

http://pwsz.revak.pl

	Command	Target	Value
1	open	/	
2	click	linkText=Oceny	
3	click	css= basic:nth-child(1)	
4	click	css=tr:nth-child(2) ui:nth-child(1)	
5	click	css=tr:nth-child(3) ui	
6	click	css=input.student.number	
7	type	css=input.student.number	12345
8	click	css= circular	
9	close	win_ser_local	

Command

click

#

Target

linkText=Oceny

Value

Description

Log Reference

Kiedy przyda się taka możliwość testowania?

Na pewno w momencie, w którym wprowadzamy zmiany, które mogą rzutować na inne części aplikacji. Nawet stuprocentowe pokrycie testami jednostkowymi nie zagwarantuje tego, że wszystko od strony użytkownika będzie działało poprawnie.

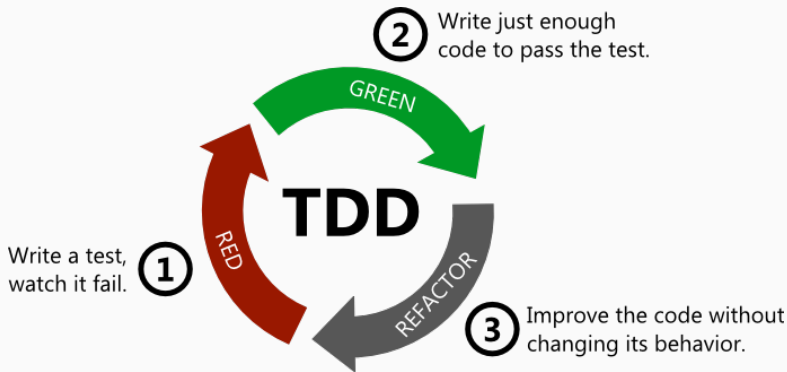
# Test Driven Development

---



TDD to jeden z elementów tzw. *extreme programming*, czyli metodologii wytwarzania oprogramowania z którą każdy powinien się zapoznać w którymś momencie kariery.

**Test Driven Development**, czyli mniej więcej *programowanie prowadzone testami*, to idea wedle której testy powinny zostać napisane przed rozpoczęciem pracy nad daną funkcjonalnością.



źródło: <https://medium.com/pacroy/test-driven-development-tdd-resource-site-tdd-pacroy-com-a02e02396f32>

# TDD w trzech krokach

Najpierw piszemy test opsiujący funkcjonalność którą chcemy dodać.  
Uruchamiamy go i otrzymujemy serię błędów, ponieważ nie stworzyliśmy jeszcze kodu, który miałby się uruchomić poprawnie.

# TDD w trzech krokach

Następnie dopisujemy funkcjonalność. Pod koniec tego kroku uprzednio napisany test powinien zakończyć się sukcesem.

# TDD w trzech krokach

Wówczas dokonujemy *refactoru*, czyli poprawiamy wszystko, co nie wiąże się bezpośrednio z działaniem funkcjonalnym, ale może poprawić szeroko rozumianą jakość kodu.

# TDD w trzech krokach

Wówczas dokonujemy *refactoru*, czyli poprawiamy wszystko, co nie wiąże się bezpośrednio z działaniem funkcjonalnym, ale może poprawić szeroko rozumianą jakość kodu.

# Podsumowanie

---

# Bibliografia i ciekawe źródła



Krzysztof Rewak, *Projektowanie i programowanie obiektowe*,  
materiały do zajęć laboratoryjnych



**Pytania?**

Kod prezentacji dostępny jest w repozytorium git pod adresem  
<https://bitbucket.org/krewak/pwsz-ppsi2>



Wszystkie informacje dot. kursu dostępne są pod adresem  
<http://pwsz.rewak.pl/kursy/6>

