Testowanie w PHP

v.1.3

Coders Lab
SZKOŁA PROGRAMOWANIA

Plan

- Wprowadzenie do testowania
- PHPUnit
- Test Driven Developement
- Fikstury
- Uruchamianie testów
- Organizacja i konfiguracja testów
- Testowanie bazy danych

Wprowadzenie do testowania

Testowanie

Ręczne

- Łatwe na początku.
- Z czasem uciążliwe (a wręcz niemożliwe).
- Jest wolne.
- Podatne na błędy.

Automatyczne

- Wymaga trochę wysiłku.
- Im dłużej trwa projekt, tym bardziej się zwraca.
- Zapewnia nam siatkę bezpieczeństwa.
- Może pomagać w programowaniu.
- Są rzeczy, które testuje się trudno
  (ale praktycznie nie ma rzeczy,
  których nie da się przetestować).
- Jest rzetelne (brak czynnika ludzkiego).

Metody testowania

Testowanie statyczne

Polega na statycznej analizie napisanego kodu, sprawdzeniu jego składni i przepływu.

Najczęściej spotykane metody to:

- code review czyli przeczytanie kodu przez kilku developerów, zanim zostanie on dodany do projektu,
- zastosowanie programów sprawdzających wszystkie możliwości przepływu programu.

Testowanie dynamiczne

Testy przeprowadzane na działającym programie. Ich zadaniem jest sprawdzenie, czy wyniki i zachowanie jest takie jak przewidywane przez nas wcześniej.

Testowanie dynamiczne powinno być rozpoczęte zanim program jest ukończony. (A najlepiej zanim program zacznie być tworzony).

Metody testowania

Testy funkcjonalne

Testy zakładające, że moduł jest „czarnym pudełkiem”, o którym wiadomo tylko, jak ma się zachowywać. Nazywamy je też testami czarnej skrzynki (black-box testing).

Testy strukturalne

Testy skupiające się na wewnętrznej pracy modułu, a nie na jego kooperacji z innymi modułami. Nazywamy je też testami białej skrzynki (white-box testing).

Poziomy testowania

Testy jednostkowe (unit testing)

Najniższy poziom testów. Ich zadaniem jest sprawdzenie poszczególnych jednostek logicznych kodu (zazwyczaj klas i ich funkcji).

Tworzone przez deweloperów podczas pracy nad poszczególnymi funkcjonalnościami.

Testy integracji (integration testing)

Testy sprawdzające integracje poszczególnych interfejsów programu i ich wzajemne oddziaływanie.

- **Metoda od dołu (bottom up)** – metoda sprawdzająca najpierw najniższe części programu i budująca testy wzwyż.
- **Metoda od góry (top down)** – metoda sprawdzająca najbardziej ogólne moduły i budująca test w dół.
- **Big Bang** – metoda grupująca moduły wedle funkcjonalności.

Poziomy testowania

Testy systemowe (end-to-end testing)

- Testy polegające na sprawdzaniu całego gotowego oprogramowania w celu znalezienia potencjalnych błędów wpływających na użytkownika.
- Testy te mają również na celu sprawdzenie, czy program nie wpływa na system, w którym działa.

Typy testów

Testy poczytalności (Sanity tests)
Najmniejsza grupa testów sprawdzająca, czy podstawowa funkcjonalność systemu działa.

Testy dymne (Smoke tests)
Grupa testów, których wykonanie trwa stosunkowo krótko.
Pokazują one, czy można rozwijać dany kod.

Regresja (regression testing)
Niezamierzone zmiany wprowadzone zazwyczaj w komponencie, nad którym nie pracujemy, a który to komponent polega na aktualnie zmienianym.

Testy regresji (Regresiontests)
Testy sprawdzające, czy funkcjonalność pozostaje bez zmian między wersjami. Często pokazują zmiany, które nie są de facto błędami, ale niechcianymi naruszeniami starego standardu.

Słowniczek

Atrapy (dummy)
Obiekt w teście, który jest nam potrzebny jako wypełnienie, a nie spełnia żadnego logicznego celu. Możemy w tym celu wykorzystać nawet pustą klasę.

Fake
Obiekt, który zawiera już logikę, ale nie taką jak prawdziwa implementacja. Na przykład tabelka z danymi zamiast bazy danych.

Zaślepka (stub)
Obiekt mający minimalną implementację potrzebnego przez nas interfejsu. Zazwyczaj funkcje zwracają dla jakiejś danej wejściowej predefiniowaną daną wyjściową.

Mock
Obiekt, który poza predefiniowaną daną wyjściową śledzi jeszcze wszystkie interakcje z interfejsem. Zazwyczaj bardziej skomplikowane klasy. Najlepiej skorzystać już z dostępnych wchodzących w skład bibliotek.


PHPUnit

Instalacja PHPUnit przez Composera

Najlepiej zainstalować PHPUnit przez Composera.

- **Systemy uniksowe**
  - w katalogu swojego projektu wykonaj polecenie: **curl -s https://getcomposer.org/installer | php**

- **System Windows**
  - pobierz i uruchom instalator http://getcomposer.org/Composer-Setup.exe

- W katalogu swojego projektu utwórz plik composer.json – będzie on zawierał wpisy, które poinformują Composera, jakich bibliotek wymaga nasz skrypt.

- Z listy dostępnej na tej stronie: http://packagist.org/explore – wybieramy bibliotekę PHP Unit: http://packagist.org/packages/phpunit/phpunit

Instalacja PHPUnit przez Composera

composer.json

- **require** – jest słowem kluczowym oznaczającym, jakie biblioteki są wymagane w naszym projekcie.
- **monolog/monolog – (autor/nazwa)** to skrócona nazwa biblioteki.
- **3.7.*** – wersja danej biblioteki. Gwiazdka oznacza, że Composer ma zainstalować zawsze najnowszą wersję biblioteki.

```
{
    "require-dev": {
        "phpunit/phpunit": "5.5.*"
    }
}
```

Instalacja PHPUnit przez Composera

Instalacja bibliotek

- Po zdefiniowaniu zależności naszego skryptu możemy pozwolić Composerowi działać – pobierze on biblioteki, od których jest zależny nasz skrypt.

- **Systemu uniksowe**
  W katalogu swojego projektu wykonaj polecenie:

  php composer.phar install

- **System Windows**
  W katalogu swojego projektu wykonaj polecenie:
  composer install

- Pobrane biblioteki Composer umieści w katalogu **vendor**, który zostanie utworzony w katalogu projektu.

- Zostanie także utworzony plik **vendor/autoload.php**, który musimy dołączyć do naszego skryptu, umieszczając na jego początku polecenie:
  require 'vendor/autoload.php';

Sprawdzamy, czy wszystko działa

Komenda

```
./vendor/bin/phpunit
```

Wynik

```
PHPUnit 3.7.21 by Sebastian Bergmann.

Usage: phpunit [switches] UnitTest [UnitTest.php]
       phpunit [switches] <directory>

  --log-junit <file>       Log test execution in JUnit XML format to file.
  --log-tap <file>         Log test execution in TAP format to file.
  --log-json <file>        Log test execution in JSON format.
...
```

Nasz pierwszy test

W pliku **/test/SampleTest.php** wpisz następujący kod:

```
class sampleTest extends
PHPUnit_Framework_TestCase {
    public function testTrue()
    {
        $this->assertTrue(false);
    }
}
```

Komenda uruchamiająca test

./vendor/bin/phpunit test/sampleTest.php

Wynik

```
PHPUnit 3.7.21 by Sebastian Bergmann.
F
Time: 0 seconds, Memory: 1.75Mb
There was 1 failure:

1) sampleTest::testTrue
Failed asserting that false is true.

C:\xampp\htdocs\phpunit\test\SampleTest.php:5
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Nasz pierwszy test

Każdy nasz test ma spełniać następujące założenia:

- Mieć nazwę klasy taką samą jak nazwa pliku,
- dziedziczyć po jednej z klas testujących **(np. PHPUnit_Framework_TestCase)**,
- mieć publiczne metody zaczynające się od słowa **test**.

Asercje

- **Asercje** to funkcje, które przerwą wykonywanie testu, jeżeli nie zajdzie podany warunek.
- **Asercje** to podstawa pisania testów.
- Do **asercji** zawsze jako ostatni argument możemy przekazać string, który zostanie wyświetlony w przypadku niespełnienia wymagań.

Nasza pierwsza asercja

Kod

```
class sampleTest extends
PHPUnit_Framework_TestCase {
    public function testTrue()
    {
        $this->assertTrue(false, "First assertion
        failed");
            }
        }
```

Wynik

```
PHPUnit 3.7.21 by Sebastian Bergmann.
F
Time: 0 seconds, Memory: 1.75Mb
There was 1 failure:

1) sampleTest::testTrue
First assertion failed
Failed asserting that false is true.
…
```

Poprawiamy test

Kod

```
class sampleTest extends
PHPUnit_Framework_TestCase {
    public function testTrue()
    {
        $this->assertTrue(true,
                                    First assertion
        failed");
            }
        }
```

Wynik

```
PHPUnit 3.7.21 by Sebastian Bergmann.

.

Time: 0 seconds, Memory: 1.75Mb

OK (1 test, 1 assertion)
```

Typy asercji

W PHPUnit jest ponad 30 asercji, które testują różne założenia.

- Np. użyta przez nas wcześniej funkcja **assertTrue** sprawdza, czy podany do niej parametr konwertuje się na wartość true.

```
$this->assertTrue(true, "First assertion failed");
//pass
$this->assertTrue(false, "First assertion failed");
//fail


$this->assertTrue(1<5, "First assertion failed");
//pass
$this->assertTrue("foo"==="bar",
                  "First assertion failed"); //fail
```

Najważniejsze asercje

- **assertArrayHasKey(key, array)**
  Sprawdza, czy w tablicy znajduje się podany klucz.
- **assertCount(expected, array)**
  Sprawdza wielkość podanej tablicy.
- **assertEmpty(array)**
  Sprawdza, czy tablica jest pusta.
- **assertEquals(expected, actual)**
  Sprawdza, czy podane zmienne są takie same.
- **assertFalse(value)**
  Sprawdza, czy podana zmienna to false.
- **assertNull(value)**:
  Sprawdza, czy podana zmienna to null.
- **assertSame(value1, value2)**:
  Sprawdza, czy podane zmienne to ten sam obiekt.
- **assertTrue(value)**:
  Sprawdza, czy podana zmienna to true.

Asercje

Stringi

Sprawdzają, czy podane w napisie założenia są spełnione.

- **assertStringEndsWith(suffix, string)**
- **assertStringStartsWith(prefix, string)**

Liczby

Sprawdzają, czy podane w liczbach założenia są spełnione.

- **assertGreaterThan(expected, actual)**
- **assertGreaterThanOrEqual(expected, actual)**
- **assertLessThan(expected, actual)**
- **assertLessThanOrEqual(expected, actual)**

Opis asercji możecie znaleźć na stronie:
https://phpunit.de/manual/current/en/appendixes.assertions.html

Czas na zadania

- Przeróbcie ćwiczenia z dnia pierwszego i katalogu 1_PHPUnit.

Test Driven Development

Co to jest TDD?

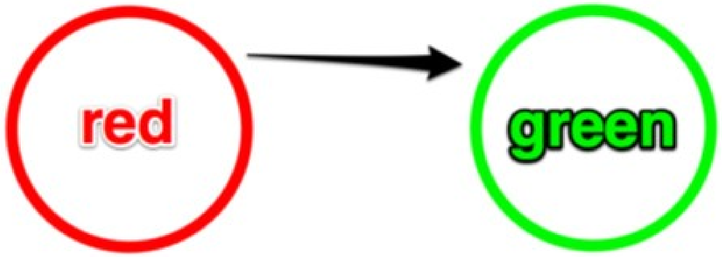
Test driven development (TDD) jest techniką tworzenia oprogramowania zaliczaną do metodyk zwinnych **(Agile)**.

Polega na wielokrotnym powtarzaniu trzech kroków:

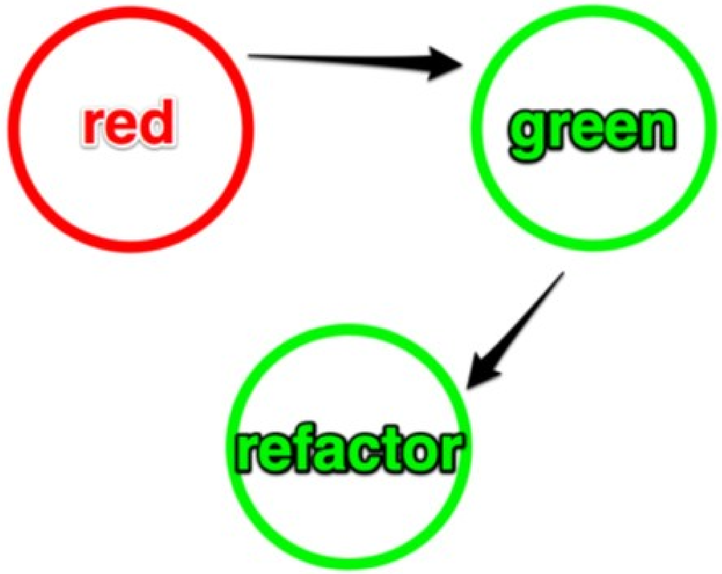
- napisaniu testu automatycznego,
- implementacji funkcjonalności do chwili, gdy wszystkie testy przejdą,
- poprawiania kodu do momentu, w którym spełnia wszystkie wymagania (a testy nadal przechodzą).

Trzy kroki TDD

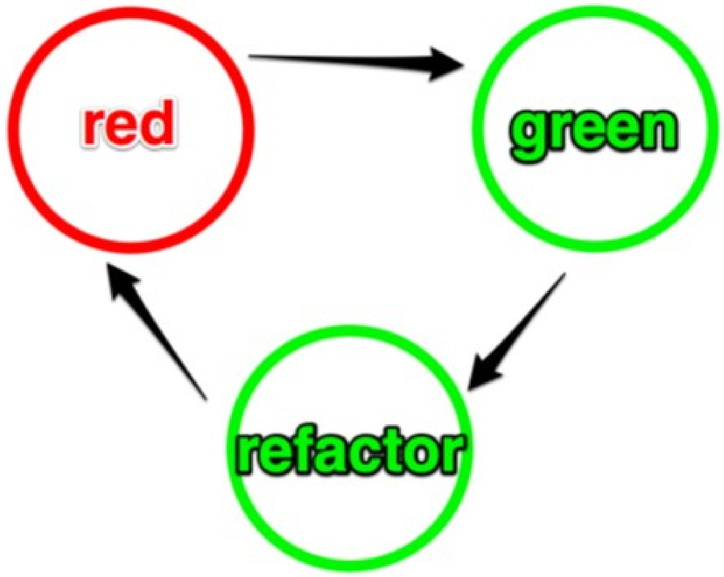
Trzy kroki TDD



Trzy kroki TDD



Trzy kroki TDD



Najważniejsza zasada TDD

Nie chodzi o często bezmyślne pisanie testów.

Chodzi o przemyślany proces pisania testów.

Zalety i wady TDD

Zalety

- Szybkie wychwytywanie błędów (już na etapie początkowej implementacji logiki).
- Błędy wykrywane i naprawiane przez autora są tańsze w naprawie.
- Kod tworzony za pomocą TDD jest bardziej przejrzysty i łatwiejszy w utrzymaniu.
- Możliwość testowania aplikacji bez potrzeby uruchamiania całego programu.

Wady

- Deweloper potrzebuje dodatkowego czasu przeznaczonego na tworzenie testów.
- Rozpoczęcie pracy nad kodem jest przesunięte w czasie.
- Testy muszą być odpowiednio zarządzane i uaktualniane wraz ze zmianą logiki. Inaczej TDD traci sens.

Cztery złote zasady TDD

Dodawaj kod, gdy czerwone.

Usuwaj kod, gdy zielone.

Usuwaj duplikaty.

Poprawiaj złe nazwy.

Nie wiem, jak to napisać (i przetestować)

Metoda Spike

- Brzydko napisz kod metodą prób i błędów.
- Gdy już wiesz, jak rozwiązać problem,
  usuń kod i zrób to porządnie.

Kiedy nasze testy są dobre?

Testy można uznać za dobrze napisane, gdy są:

- **Niezależne**
  (od środowiska i innych testów).
- **Szybkie**
  (dzięki temu mogą być wykorzystane w Continuous Integration).
- **Powtarzalne**
  (za każdym razem dają takie same wyniki).
- **Na bieżąco z kodem**
  (zawsze, gdy zmienia się funkcjonalność, muszą zmienić się testy).
- **Krótkie**
- **Odporne na zmiany**
  (innych części naszej aplikacji)

Czas na zadania

- Przeróbcie ćwiczenia z katalogu 2_TDD.