TDD

Test Driven Development

Tomasz Sadza 2022-10-03 , **2022-10-04**

TDD

Dzień 2

State verification

TDD Patterns

Po wykonaniu testu (fazie *ACT*) dokonywana jest inspekcja stanu testowanego systemu (SUT). Otrzymane wartości porównywane są z wartościami oczekiwanymi dla poprawnego działania systemu.

State Verification powinien być używany, gdy interesuje nas tylko stan końcowy systemu, a nie w jaki sposób system znalazł się w takim stanie.

```
test ('user name can be changed', function() {
    $user = User::factory()->make();
    $user->name = 'John';
    expect($user->name)->toBe('John');
}
```

Guard Assertion

TDD Patterns

Czasami jednak warto zweryfikować nie tylko stan przed, ale również stan po teście, a więc użyć wzorca Guard Assertion. Wzorzec Guard Assertion polega na ujawnieniu założeń przed wywołaniem funkcjonalności, którą chcemy testować.

```
test ('can add item to Order', function() {
    $order = new Order();
    expect(is_empty($order->getItems()))->toBeTrue();
    $order->add(new Item());
    expect(count($order->getItems()))->toBe(1);
}
```

Delta Assertion

TDD Patterns

Jeśli w testach nie możemy stanów początkowych zakodować na sztywno, to nie należy weryfikować stanu absolutnego po wywołaniu kodu w teście. Lepiej sprawdzić, czy różnica (delta) pomiędzy stanem początkowym a końcowym jest zgodna z naszymi oczekiwaniami.

```
test ('if new Item increased Order total', function() {
     $order = Order::get()->random();
     $total = $order->getTotal();
     $order->add($item = Item::get()->random());

     expect($order->getTotal())->toBe($total + $item->getPrice());
}
```

Custom Assertion

TDD Patterns

Zdarza się, że długość kod weryfikującego nasze oczekiwania przekracza długość kodu wymaganą do wywołania kodu w teście. W takim przypadku zaleca się wyodrębnienie metody *Custom Assertion* z testu w celu hermetyzacji złożonej logiki weryfikacji w prostej metodzie, którą możemy wywołać z testu.

```
class Temperature {
    …
    public function isEqual(Temperature
$temperature) {
        return $this->getValue() ===
$temperature->getValue();
    }
}
```

```
test('check if two Temperatures are
equals', function() {
    $t1 = new Temperature(20);
    $t2 = new Temperature(20);

    expect($t1->isEqual($t2))->toBeTrue();
}
```

Behavior Verification

TDD Patterns

Wzorzec *Behavior Verification* nie weryfikuje poprawności zwracanych wartości, lecz sprawdza, czy nasz kod współdziała z obiektami współpracownikami w oczekiwany przez nas sposób. Każdy test weryfikuje jakie metody, i w jaki sposób, są wywoływane na współpracownikach przez testowany obiekt.
W jednym teście możemy używać zarówno stylu asercji opartego na stanie, jak i stylu opartego na interakcji.

```
test('if Manager can process Order to
Jobs', function() {

    $order = new Order();
    $order->add(Drink::get()->random());
    $order->add(Drink::get()->random());
    $order->add(Drink::get()->random());

    Manager::process($order);

    expect($order->getStatus())
    ->toBe('IN PROGRESS');
    expect(Job::get()->count())
    ->toBe(3);
}
```

Object Mother

TDD Patterns

Object Mother jest specjalizowaną fabryką, której rolą jest dostarczenie istotnych dla testu danych.

Dane te są przez fabrykę odpowiednio skonfigurowane.

```
class CandidateJobRequest {
    …
    public static function positive() {
    …
     }
}
```

```
test('if valid Job Request can be
approved', function() {

    $jobRequest =
      CandidateJobRequest::positive();

   $jV = new JobValidator($jobRequest);

   expect($jV->getStatus())
     ->toBe(JobValidator::APPROVED);
}
```

Builder Object

TDD Patterns

Wzorzec *Object Mother* nie sprawdza się w sytuacji, kiedy musimy uwzględnić wiele wariacji danych testowych.

Lepszym rozwiązaniem jest dynamiczne budowanie takich danych na żądanie z wykorzystaniem wzorca Test Builder. Obiekty Test Builder są implementacją klasycznego wzorca Budowniczy, który umożliwia zbudowanie złożonego obiektu poprzez kolejne wywołania metod budowniczego i odebranie finalnego obiektu przez metodę get().

```
class JobRequestBuilder {
    …
    public function withExperience(..) { … }
    public function get() { … }
}
```

```
test('if senior Job Request can be
approved', function() {

    $jobRequest = new JobRequestBuilder();
    $jobRequest->withExperience('senior');

    $jV = new JobValidator(
      $jobRequest->get());

    expect($jV->getStatus())
       ->toBe(JobValidator::APPROVED);
}
```

Wzorce testów

TDD Patterns

https://www.codeproject.com/Articles/5772/Advanced-Unit-Test-Part-V-Unit-Test-Patterns#Patterns1

Identyfikacja i naprawa delikatnych testów

Identifying and repairing fragile tests

- https://phpunit.readthedocs.io/en/9.5/risky-tests.html
  - Useless tests
  - Unintentionally covered code
  - Output during test execution
  - Test execution timeout
  - Global state manipulation

TDD a refaktoryzacja

Refaktoryzacja to, według definicji,
wprowadzenie zmian w kodzie bez zmiany jej funkcjonalności.

W przypadku legacy

- opcja 1 - jeśli działa, nie ruszaj,

- opcja 2 - przelicz czy warto (czas pracy programisty na napisanie testów),

- opcja 3 - piszemy testy do kodu nowego i zmienianego

TDD a refaktoryzacja

Istnieje sporo metod refaktoryzacji, które służą wprowadzeniu poprawek w istniejącym kodzie. W najprostszym uogólnieniu, opierają się one na:

- Grupowaniu i ekstrahowaniu logiki biznesowej do oddzielnych metod lub klas.
- Wydzieleniu nowej funkcjonalności do odrębnej metody lub klasy, a następnie wstrzyknięcie jej do starego kodu.
- Odizolowaniu możliwie małej części starego kodu (np. pętli), napisanie testów jednostkowych a następnie wprowadzenie zmiany.
- Eksponowanie zależności na zewnątrz.

TDD a refaktoryzacja

Testy integracyjne i akceptacyjne

- Mogą okazać się prostszym rozwiązaniem niż pokrywanie testami jednostkowymi
- Dzięki nim nabierzemy pewności przy wprowadzaniu zmian do kodu legacy
- Związanie testów integracyjnych i akceptacyjnych z systemem buildów oraz dążenie do funkcjonalnej regresji przyniesie wymierne korzyści, nie wprowadzając przy tym ryzyka związanego z testami jednostkowymi i niezbędną przy tym refaktoryzacją.