TDD, Testowanie oprogramowania,
Testy jednostkowe, Unit Tests,
F.I.R.S.T, asercja, refraktoryzacja, programowanie w parach

Mariusz NAJWER

PROGRAMOWANIE Z TECHNIKĄ TEST-DRIVEN DEVELOPMENT

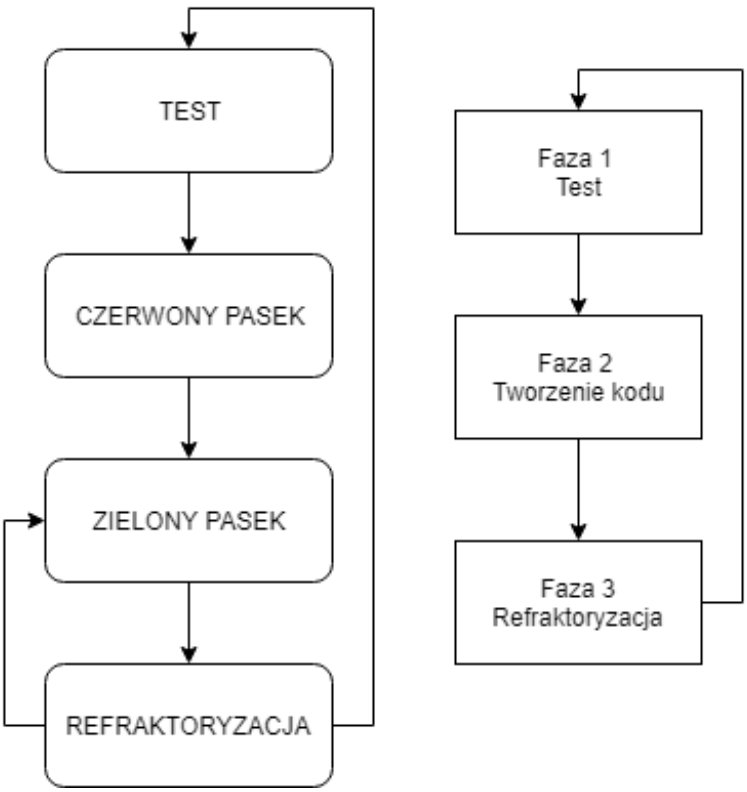
Artykuł prezentuje jeden ze sposobów tworzenia aplikacji informatycznych w oparciu o technikę Test-Driven Drivelopment. Treść pracy została podzielona na trzy rozdziały. W pierwszym rozdziale przedstawiono definicję techniki. Zaprezentowano algorytm postępowania przy wytwarzaniu oprogramowania. Omówiono jego zalety i wady odnośnie tradycyjnego podejścia do tworzenia kodu. Treść w tym rozdziale oparto na pozycjach [1], [2]. W rozdziale następnym zatytułowanym F.I.R.S.T. pokazano pięć głównych zasad, którymi programista powinien się kierować korzystając z techniki TDD. Akronim jest programistyczny formalizmem gwarantującym sukces podczas współpracy z testami jednostkowymi. Pojęcie zostało szerzej zaprezentowane w pozycji [3]. W rozdziale trzecim przedstawiono koncepcje testów wykorzystywane przez programistów. Istotą tego fragmentu jest pokazanie, że tworzenie czytelnych funkcji testujących jest tak samo ważne, jak sama idea sprawdzania jakości oprogramowania. Koncepcja została szerzej zaprezentowana w pozycjach [1] [2]. Główną motywacją stworzenia tego artykułu było ukazanie, że podejście TDD niesie ze sobą zdecydowanie więcej korzyści przy budowie konkretnych programów. Technika jest idealna dla dużych aplikacji internetowych, które są tworzone przez zespoły programistyczne. Nie mniej jednak należy mieć na uwadze, że istnieją problemy nie nadające się, żeby rozwiązywać je przy pomocy Test-Driven Development. Jednym z przykładów może być tworzenie kompilatorów lub interpreterów.

1. TEST-DRIVEN DEVELOPMENT

1.1. ALGORYTM

Test-driven development lub TDD jest szybkim cyklem narzuconych procedur, które musi wykonać programista, żeby stworzyć nowy fragment aplikacji. Technika definiuje architekturę aplikacji zamykając je w ściśle narzuconych prawach i regułach. Możemy wyróżnić trzy główne stany przejściowe zależne między sobą: faza pisania testu, faza

tworzenia kodu, faza refraktoryzacji. Literatura podaję, że czas jednego cyklu powinien wynosić od 20 sekund do kilku minut. Ideą tego podejścia jest skupienie się na poprawności działania aplikacji, a nie na jej implementacji. Zależności pomiędzy stanami zaprezentowano na rysunku 1.



Rys. 1. Algorytm postępowania przy wytwarzaniu oprogramowania techniką TDD

Warto zauważyć, że stosując algorytm z rysunku 1 tworzymy tak naprawdę prosty kod, który działa i nie narusza całej struktury aplikacji. Programista piszą testy jednostkowe nie musi się obawiać, że w momencie kiedy chce dodać kolejną funkcje do aplikacji, zmieni elementy dodane przez innego partnera z zespołu, powodując niestabilność programu. Inną ważna zaletą tego podejścia jest zmniejszenie stresu twórcy kodu. Ukryta i wymuszona metoda „dziel i zwyciężaj" przez TDD powoduje, że programista nie musi się przejmować wszystkimi zleconymi zadaniami związanymi z aplikacją, lecz tylko z poszczególnymi fragmentami. Dekompozycja problemu pozwala wydajniej pracować przy projekcie oraz gwarantuje jakość napisanego kodu.

Według autora książki [2] projekty wykorzystujące technikę TDD są realizowane szybciej, niż przy tradycyjnym podejściu.

1.2. TEST

Test jest pierwszym krokiem od którego należy rozpocząć algorytm. W tym kroku programista zastanawia się co chce przetestować w programie, jaką funkcje chce dodać do aplikacji. Tworząc wyobrażenie o swoim teście równocześnie stara się myśleć o jak najkrótszym teście. Im test jest krótszy tym lepiej. Test musi być czytelny, a także dobrą częścią dokumentacji.

1.3. CZERWONY PASEK

W drugim kroku programista pisze swój wyobrażony test, ale musi się on załamać w trakcie sprawdzenia. Jeśli w pierwszym kroku wymyliśmy sobie, że chcemy utworzyć nową klasę, to w tym kroku dobrym pomysłem byłoby sprawdzenie, czy klasa już istnieje. Nasz program testujący zwróci nam informację o błędzie, że podana klasa nie istniej, równocześnie wyświetlając nam charakterystyczny czerwony pasek.

1.4. ZIELONY PASEK

W trzecim kroku należy napisać kod, który pozwoli nam zaliczyć test. Pozytywny test wyświetla komunikat o udanym teście w postaci zielonego paska. Kod produkcyjny należy pisać w jak najszybszym czasie oraz w jak najkrótszy. Pisząc kod komputerowy nie rozwlekamy się nad dobrymi praktykami, naszym jedynym celem jest pomyślny test nawet kiedy nie jest on czytelny.

1.5. REFRAKTORYZACJA

W czwartym kroku, kiedy wszystkie nasze testy zostały zaliczone do pomyślnych, następuje ostatnia faza refraktoryzacji. Część ta polega na ulepszaniu napisanego kodu z części trzeciej. Dla przykładu wykonujemy następujące czynność o ile są możliwe: zmienimy nazwy zmiennych na bardziej znaczące, redukujemy pętle, redukujemy liczbę instrukcji warunkowych, usuwamy niepotrzebne zmienne itd. Faza trwa tak długo, aż po wprowadzonych poprawkach uruchamiając nasze oprogramowanie testowe na naszym ekranie znów pojawi się zielony pasek.

Po skończonej pracy, rozpoczynamy algorytm od nowa dodając kolejną funkcje do aplikacji. Ten tryb pracy pozwala zminimalizować czas na myślenie i pozwala nam zacząć działać. Po godzinie pracy w tym trybie jesteśmy w stanie zrealizować dwadzieścia cykli algorytmu, co daje nam dwadzieścia nowych i sprawdzonych funkcji, które nie naruszają naszego głównego programu. Również łatwo możemy zauważyć, że czytając same testy, wiemy co oferuje nasz program. Jesteśmy bardziej zadowoleni z naszego oprogramowani, ponieważ mamy świadomość, iż nasz kod jest czytelny. W przyszłości szybciej zlikwidujemy pojawiające się błędy.

1.6 TRZY PRAWA TDD

TDD charakteryzuje się hermetycznymi zasadami, ponieważ definiuje szkielet aplikacji. Często wymuszając pewne rzeczy nie wprost, a nawet powodując, że programista pisze zbyt dużo testów, niż powinien. Dlatego stworzono kilka nienaruszalnych praw ułatwiające życie programistom.

W szczególności możemy wyróżnić trzy najważniejsze wskazówki. Należy zauważyć, że mimowolne naruszenie któregokolwiek prawa często kończy się komplikacjami, które ciężko jest naprawić w dalszej części pracy przy projekcie. Chciałbym teraz przedstawić te zasady gwarantujące sukces przy korzystaniu z tej techniki.

Pierwsze prawo mówi, że nie można zacząć pisać kodu produkcyjnego do momentu napisania niespełnionego testu jednostkowego. Czasami test, który wydaje nam się banalny i traktujemy go lekceważąco, przez to nie tworząc testowej instancji w programie powoduje to, że możemy nie odkryć niespodziewanego błędu. Zdecydowanie lepiej opłaca sprawdzić nasze domysły.

Drugie prawo. Nie można napisać więcej testów jednostkowych, które są wystarczające do niespełnienia testu, a brak kompilacji jest jednocześnie nieudanym testem. Nadgorliwość powoduje, że spędzamy czas na rozwiązywanie problemów, których nie da się rozwiązać lub nie potrzebujemy ich rozwiązywać. Konsekwencją jest marnowanie czasu na rzeczy nie mające znaczenia w naszym oprogramowaniu.

Trzecie prawo. Nie można pisać większej ilości kodu produkcyjnego, niż wystarczy do spełnienia obecnie niespełnionego testu. Redundancja kodu produkcyjnego rozrasta nasz program, choć wcale tego nie potrzebujemy.

2. F.I.R.S.T

2.1. SZYBKOŚĆ (FAST)

Pisane przez nas testy powinny być szybkie. Muszą działać szybko. Gdy testy działają powoli, nie chcemy ich uruchamiać zbyt często. Jeżeli nie uruchamiamy ich zbyt często, nie znajdziemy problemów wystarczająco szybko, aby można było łatwo je poprawić. Nie czujemy się wystarczająco pewnie, aby czyścić nasz kod. W końcu kod zaczyna się psuć.

2.2. NIEZALEŻNE (INDEPENDENT)

Testy nie powinny zależeć od siebie. Jeden test nie powinien konfigurować warunków do następnego testu. Powinniśmy być w stanie uruchamiać każdy test nieza-

leżnie i uruchamiać testy w dowolnie wybranym porządku. Jeśli testy zależą od siebie, to gdy nie uda się pierwszy test, powstaje kaskada awarii, co utrudnia diagnozę i ukrywa awarie na niższym poziomie.

2.3. POWTARZALNE (REPEATABLE)

Testy powinny być powtarzalne w każdym środowisku. Powinniśmy być w stanie uruchomić testy w środowisku produkcyjnym, środowisku zapewnienia jakości oraz na naszym laptopie w trakcie podróży pociągiem do domu. Jeżeli nasze testy nie są powtarzalne w każdym środowisku, to zawsze będziemy mieli wymówkę, gdy się nie powiodą. Okazuje się również, że możemy mieć kłopoty z uruchomieniem testów, gdy nie jest dostępne dane środowisko.

2.4. SAMOKONTROLUJĄCE SIĘ (SELF-VALIDATING)

Testy powinny mieć jeden parametr wyjściowy typu logicznego. Mogą one się powieść lub nie. Nie powinniśmy musieć czytać plików dzienników w celu sprawdzenia, czy testy się powiodły. Nie powinniśmy ręcznie porównywać dwóch plików testowych w celu sprawdzenia, czy test się powiódł. Jeżeli testy nie kontrolują się samodzielnie, to awaria może stać się subiektywna i uruchomienie testów będzie wymagać długiego procesu ręcznej analizy.

2.5. O CZASIE (TIMELY)

Testy powinny być pisane w odpowiednim momencie. Testy jednostkowe powinny być pisane bezpośrednio przed tworzeniem testowanego kodu produkcyjnego. Jeżeli piszemy testy po kodzie produkcyjnym, może się okazać, że jest on trudny do przetestowania. Można zdecydować, że pewna część kodu produkcyjnego jest zbyt trudna do przetestowania. Nie można zaprojektować kodu produkcyjnego tak, aby nie dał się przetestować.

3. KONCEPCJA TESTÓW

3.1. JEDNA ASERCJA NA TEST

Koncepcja ta zakłada, że piszmy jeden test, czyli używamy tylko jednej asercji do sprawdzenia naszego modułu testowego. Autor książki [] nazywa tę zasadę drakońską, ponieważ wymusza na nas pisaniu bardzo dużej liczby funkcji testujący, sprawdzających ten sam moduł. Powoduje to duży nakład pracy programisty i zmniejsza czytel-

ność testów. Mimo wszystko są niezależne i lepiej jest korygować błędy.

```
/** @test */
public function testTddController()
{
        $this->assertFileExists('src/Controller/TddControllerRegister.php');
}

public function testUserEntity()
{
        $this->assertFileExists('src/Entity/User.php');
}
```

Rys. 2. Przykład funkcji testujących z jedną asercją w PHPUnit

3.2 JEDNA KONCEPCJA NA TEST

Drugą koncepcją jest użycie kilku asercji w jednej funkcji testującej. Ten sposób pozwala nam zachować jednolitość. Nasza funkcja testująca testuje pojedynczą koncepcję, co daje nam lepszą kontrolę nad pisanym kodem.

```
public function testcheckEmail()
{
        //check no .
        $result = $this->register->checkEmail("najwer@gmailcom", "ok");
        $this->assertEquals($result, 'Niepoprawny adres email');

        //check no @
        $result = $this->register->checkEmail("najwergmail.com", "ok");
        $this->assertEquals($result, 'Niepoprawny adres email');

        //good test
        $result = $this->register->checkEmail("najwer23@gmail.com", "ok");
        $this->assertEquals($result, 'ok');
}
```

Rys. 3. Przykład funkcji testujących z trzema asercjami w PHPUnit

LITERATURA

[1] Beck K. *TDD. Sztuka tworzenia dobrego kodu*, Helion 2014.
[2] Martin R.-C., *Clean Code: A Handbook of Agile Software Craftsmanship*, First Edition, Pearson Education 2008.
[3] Martin R.-C., *Professionalism and Test-Driven Development*, Object Mentor, IEEE Software, maj – czerwiec 2007 (Vol. 24, No. 3), s. 32 – 36.
[4] https://phpunit.de/, 5.01.2019.