JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 4 (2017) 112-116
Wysłane: 2017-03-16
Przyjęte: 2017-03-22

Test-Driven Development jako narzędzie optymalizacji procesu wytwarzania oprogramowania na platformie JEE

Grzegorz Sochacki*, Beata Pańczyk

Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. W artykule poruszono temat korzyści płynących z zastosowania metodyki Test-Driven Development. Badania przeprowadzono z wykorzystaniem autorskiej aplikacji na platformę Java Enterprise Edition. Badana metodyka została porównana ze standardowym podejściem pisania testów.

Słowa kluczowe: Test-Driven Development; Java; testy jednostkowe; testowanie

*Autor do korespondencji.

Adres e-mail: g.sochacki91@gmail.com

Test-Driven Development as a tool to optimize the JEE programming

Grzegorz Sochacki*, Beata Pańczyk

Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The article is about the benefits of using the Test-Driven Development methodology. Tests were carried out on the author's application based on Java Enterprise Edition platform. The investigated methodology was compared with the standard approach to writing tests.

Keywords: Test-Driven Development; Java; unit tests; testing

*Corresponding author.

E-mail address: g.sochacki91@gmail.com

1. Wstęp

W procesach wytwarzania oprogramowania stosuje się wiele technik pozwalających na uzyskanie kodu dobrej jakości. Jednym z najpopularniejszych sposobów jest pisanie testów jednostkowych [1, 2]. Testy te mają za zadanie podzielić kod aplikacji na małe fragmenty funkcjonalności, które będą testowane niezależnie od reszty aplikacji. W programowaniu obiektowym, jedna klasa testowa przypada zazwyczaj na jedną klasę aplikacji a każdą metodę publiczną testuje przynajmniej jedna metoda testowa.

Standardowym podejściem w testach jest pisanie ich dla istniejących już funkcjonalności. W takim podejściu nie ma nic złego pod warunkiem, że do napisanego kodu istnieje możliwość napisania testu jednostkowego. Często próba napisania testów kończy się niepowodzeniem, ponieważ logika jest rozproszona pomiędzy wieloma klasami lub sama architektura kodu nie pozwala na poprawne przetestowanie. Wówczas pozostaje jedynie żmudny proces refaktoryzacji kodu (który bez napisanych testów nie zawsze jest bezpieczną operacją), lub zaniechanie testów.

Aby zapobiec tworzeniu kodu, który nie da się przetestować, wprowadzono metodykę programowania sterowaną testami (ang. *Test-Driven Development*) [3]. Metodyka ta odwraca kolejność operacji stosowanych w klasycznym podejściu. Najpierw pisze się testy do programowanej funkcjonalności, a następnie implementację. Proces pisania testów i implementacji powtarzany jest do zakończenia programowania funkcjonalności. W momencie zakończenia programista powinien dokonać refaktoryzacji kodu, co dzięki testom wychwyci ewentualne naruszenie logiki aplikacji.

Używanie Test-Driven Development (TDD) spotyka się zazwyczaj ze sprzecznymi opiniami. Często wdrożenie metodyki do projektu zostaje zaniechane ze względu na obawy związane z dodatkowym narzutem czasu. W przypadku małych projektów brak testów nie stanowi dużego problemu. Ciężko natomiast odpowiednio utrzymać duży projekt, w którym pracuje wielu programistów, bez odpowiednio testowanego kodu.

W niniejszym artykule przedstawiono porównanie standardowego testowania z podejściem Test-Driven Development. Analizy dokonano podczas implementacji nowych funkcjonalności w aplikacji testowej opartej o platformę Java Enterprise Edition [4]. Do porównania obu podejść posłużyły obserwacje zarejestrowane podczas pisania kodu oraz struktura otrzymanego kodu. Testy wykonane zostały przez programistę, który posiadał już niezbędną wiedzę konieczną do realizacji postawionych zadań, zarówno w podejściu tradycyjnym jak i TDD.

Tezą badawczą jest:

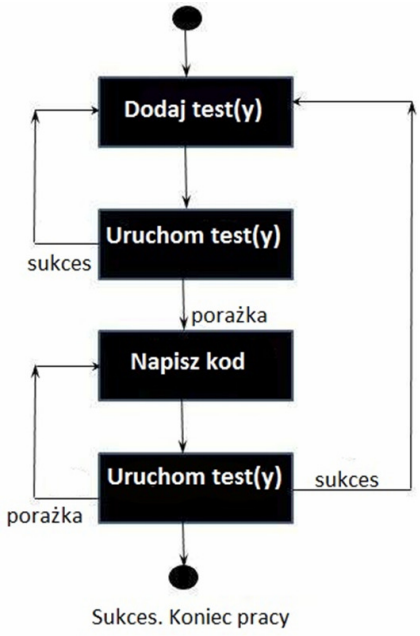
TDD jest narzędziem, które przyspiesza proces wytwarzania dobrej jakości kodu na platformie JEE.

2. Metodyka TDD w aplikacjach JEE

Platforma testowa pozwala na implementację funkcjonalności, tworzonych za pomocą metodyki TDD. Ocena korzyści zastosowania podejścia była oparta o wpływ TDD na proces tworzenia i jakość kodu.

W klasycznym podejściu pisania testów, testy pisane są do istniejącego już kodu. Odwrócenie tej zależności dla wielu programistów wydaje się zaprzeczeniem idei pisania testów jednostkowych. Istnieje obawa, że wdrożenie takiego podejścia może w początkowej fazie przynieść straty w postaci dłuższego czasu wytwarzanego kodu.

Samo w sobie podejście TDD oparte jest na prostym algorytmie pracy, który każdy, nawet początkujący programista powinien bez problemu zrozumieć. TDD wyróżnia 4 główne fazy pisania kodu. Rysunek 1 przedstawia podstawowy schemat pracy w TDD.



Rys. 1. Schemat pracy w metodyce Test-Driven Development

3. Realizacja badań

3.1. Platforma testowa

Do badań stworzono aplikację napisaną w języku Java 8. Jest to aplikacja internetowa, która została oparta o komponenty platformy Java Enterprise Edition. Aplikacja jest prostym systemem pozwalającym na zarządzanie klientami w przedsiębiorstwie. Aplikacja działa na serwerze WildFly w wersji 9.0.1. Dane o klientach przechowywane są w relacyjnej bazie danych uruchomionej na serwerze PostgreSQL. Do implementacji wykorzystano następujące technologie [5]:

- EJB 3,
- JAX-RS,
- Hibernate.

Zestaw technologii nie jest przypadkowy. Jest to podstawowy stos technologii, umożliwiający napisanie API funkcjonalnej aplikacji. Zapewnia on sprawne zarządzanie warstwą danych systemu oraz komunikację z zewnątrz poprzez interfejs REST [6].

W celu napisania odpowiednich testów jednostkowych, wykorzystano technologie TestNG [8] oraz Mockito [7]. Pierwsza z technologii jest frameworkiem wspierającym testowanie w aplikacjach Java. Umożliwia pisanie zarówno testów jednostkowych, jak i integracyjnych. Bibliotekę Mockito wykorzystano do tworzenie niezbędnych atrap obiektów.

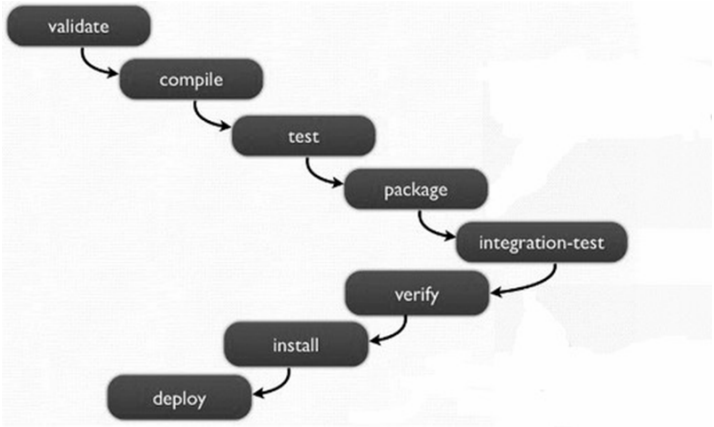
Do sprawnego poruszania się po projekcie zastosowano narzędzie developerskie Eclipse w wersji Neon, obsługujące większość języków programowania. W środowisku zainstalowano wtyczkę JbossTools, potrzebną do sprawnej integracji aplikacji z serwerem WildFly.

3.2. Struktura projektu

Automatyzacja budowania projektu użytego do badań jest zapewniona dzięki narzędziu Maven. Narzędzie to służy do budowania wielomodułowych projektów w języku Java. W pliku *pom.xml* definiowane są cele, które mają zostać osiągnięte po kompletnym zbudowaniu projektu [8]. Do przykładowych celi zalicza się kompilacje plików źródłowych, uruchomienie testów, zbudowanie pliku wdrożeniowego.

Proces budowania projektu został podzielony na cykle, które odpowiadają za konkretne zadania (Rysunek 2):

- validate – weryfikacja poprawności projektu,
- compile – kompilacja kodu źródłowego,
- test – uruchomienie testów jednostkowych,
- package – zbudowanie pliku wdrożeniowego,
- integration-test – uruchomienie projektu w środowisku testowym,
- verify – weryfikacja paczki wdrożeniowej,
- install – umieszczenie paczki w lokalnym repozytorium,
- deploy – umieszczenie paczki w zdalnym repozytorium.



Rys. 2. Proces budowania aplikacji za pomocą narzędzia Maven [9]

3.3. Badane obiekty

Badanie przeprowadzono podczas implementacji nowych funkcjonalności w obu podejściach. Pojawiło się ryzyko, że programowanie dwa razy tych samych funkcjonalności może niekorzystnie wpłynąć na wyniki. Aby temu zapobiec, ustalono tygodniowy odstęp pomiędzy badaniami. W każdym przypadku wykorzystywano osobne klasy. Pomiędzy kodem obu podejść współdzielone są tylko klasy niezależne od testowanej w danym momencie metody. Dla wiarygodności testów, kolejność badanych technik była stosowana na przemian. Testy zostały przeprowadzone na implementacji trzech przykładowych metod biznesowych:

- obliczenie salda klienta,
- naliczanie rabatu dla klienta,
- naliczenia prowizji pracownika.

Na wyniki badań złożyły się dwa czynniki:

- informacje na temat parametrów pracy zarejesrtowane podczas implementacji w danym podejściu,
- otrzymany kod.

Głównym elementem, na który należało zwrócić uwagę, był czas. Dla każdego z podejść rejestrowano czas poświęcony na pisanie kodu i pisanie testów. Rejestrowano także liczbę błędów w kodzie, jakie wykryły testy.

Otrzymany kod został poddany analizie w celu określenia jego jakości. Posłużyły do tego metryki CK. Metryki te pozwalają na ocenianie wykorzystanych mechanizmów programowania obiektowego.

4. Wyniki badań

4.1. Analiza implementacji

Pierwszą z badanych metod była metoda służąca do naliczania salda klienta. Logika metody nie była skomplikowana, lecz testy wymagały uwzględnienie wielu szczególnych przypadków (Przykład 1). Na potrzeby testów operacje prowadzone na datach zostały przeniesione z poziomu bazy danych na poziom kodu. Czas poświęcony na napisanie kodu nie różnił się bardzo pomiędzy dwoma podejściami. W przypadku TDD skrócił się natomiast czas pisania testów jednostkowych. W przypadku podejścia standardowego można zauważyć większą liczbę błędów popełnionych w kodzie. W przypadku TDD błędów jest znacznie mniej ze względu na implementację logiki w mniejszych fragmentach i ciągłe jej testowanie. Wyniki przedstawiono w tabeli 1.

Przykład 1. Fragment kodu pierwszej metody testowej

```
@Override
public BigDecimal calculateSaldo(Customer customer) {
BigDecimal saldo  = BigDecimal.ZERO;
List<Obligation> obligations =
    obligationService.getObligationForClient(customer);
List<Payment> payments = new ArrayList<Payment>();
if(obligations == null || obligations.isEmpty()){
  return null;
}
for(Obligation obligation : obligations){
  Calendar calendar = prepareFirstPenaltyDate(obligation);
  Date firstCashPenaltyDate = calendar.getTime();
  calendar = prepareSecondPenaltyDate(obligation);
  Date secondCashPenaltyDate = calendar.getTime();
  Date today = new Date();

if(obligation.getPayments() != null){
  payments.addAll(obligation.getPayments());
}else if
(TimeUnit.DAYS.convert(today.getTime() -
firstCashPenaltyDate.getTime(), TimeUnit.MILLISECONDS)
<= 0){
saldo = saldo.subtract(obligation.getAmountToPay().
multiply(BigDecimal.valueOf(0.1)));
}else if(TimeUnit.DAYS.convert(today.getTime() -
secondCashPenaltyDate.getTime(),
TimeUnit.MILLISECONDS) <= 0){
saldo =   saldo.subtract(obligation.getAmountToPay().
multiply(BigDecimal.valueOf(0.3)));
}
  saldo = saldo.subtract(obligation.getAmountToPay());
}

  saldo = calculatePayments(saldo, payments);
  return saldo;
}
```

Tabela 1. Wyniki badania pierwszej metody

	Standardowe podejście	Podejście TDD
Kolejność przeprowadzonego badania	1	2
Czas pisania kodu potrzebnego do implementacji metody [min]	50	45
Czas pisanie testów [min]	40	28
Liczba wykrytych błędów przez testy	5	1

Drugą metodą (Przykład 2) była implementacja logiki odpowiedzialnej za naliczenie rabatu dla klienta. Ze względu na dużo większą złożoność zadania niż w pierwszym przypadku, czas poświęcony na badanie znacznie się wydłużył. Złożona logika i wiele przypadków testowych wymagały większej liczby testów. Ponownie czas potrzebny na implementacje logiki był zbliżony do siebie w obu przypadkach. Mniej czasu poświecono na pisanie testów w podejściu TDD. Testy w obu przypadkach wykryły kilka błędów krytycznych. Większą liczbę błędów można zauważyć w standardowym podejściu. Wyniki zawarte w tabeli 2 przemawiają nieznacznie na korzyść TDD.

Przykład 2. Nagłówek drugiej metody testowej

```
public Integer calculateDiscount(Customer customer) {}
```

Badanie metody trzeciej (Przykład 3) polegało na naliczenie prowizji dla pracownika. Złożoność metody była nieco mniejsza niż w przypadku poprzedniej. Testy wykryły błędy w operacjach matematycznych przeprowadzanych na kwotach prowizji. Czas poświęcony na napisanie logiki był znacznie dłuższy w przypadku standardowego podejścia.

Pisanie testów w obu podejściach zajęło podobną ilość czasu. Zestawienie wyników przedstawiono w tabeli 3.

Przykład 3. Nagłówek trzeciej metody testowej

```
public Provision calculateProvision(Contract contract){}
```

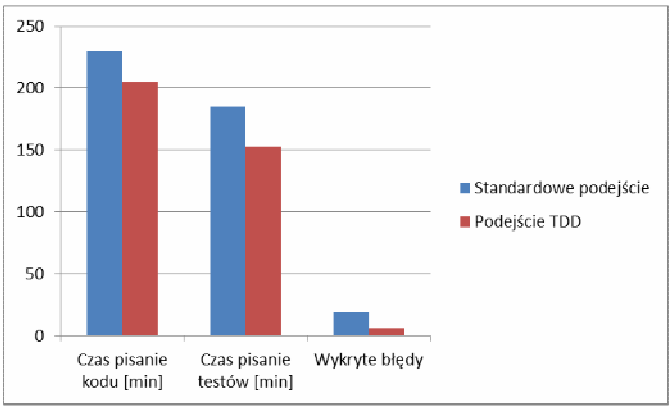
Tabela 2. Wyniki badania drugiej metody

	Standardowe podejście	**Podejście TDD**
Kolejność przeprowadzonego badania	2	1
Czas pisania kodu potrzebnego do implementacji metody [min]	110	105
Czas pisanie testów [min]	100	85
Liczba wykrytych błędów przez testy	8	3

Tabela 3. Wyniki badań trzeciej metody

	Standardowe podejście	**Podejście TDD**
Kolejność przeprowadzonego badania	1	2
Czas pisania kodu potrzebnego do implementacji metody [min]	70	55
Czas pisanie testów [min]	45	40
Liczba wykrytych błędów przez testy	6	2

Zebrane z trzech badań wyniki wskazują jako faworyta podejście TDD. Nie jest to jednak znacząca przewaga. Można uznać jednak, że wdrożenie TDD na pewno nie wpływa negatywnie na czas realizacji projektów. Na rysunku 3 przedstawiono porównanie wyników z trzech badań.



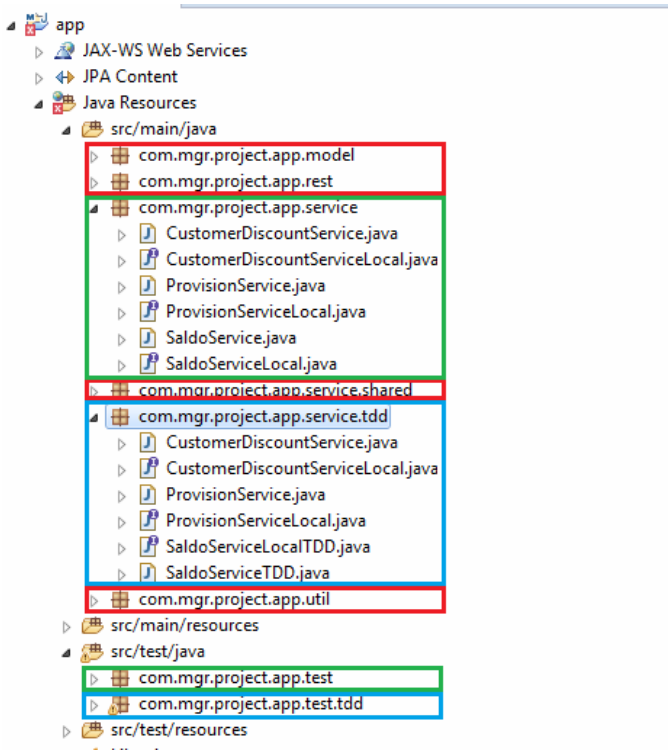
Rys. 3. Zestawienie wyników analizy implementacji

4.2. Analiza jakości kodu

Po oddzieleniu od siebie kodu obu podejść możliwa była jego analiza. Z analizy wykluczono metody i klasy wbudowane w język Java.

Na rysunku 4 przedstawiono strukturę projektu z podziałem na zasoby wykorzystane w badanych podejściach:

- kolor czerwony – zasoby współdzielone pomiędzy dwoma metodami,
- kolor zielony – standardowe podejście,
- kolor niebieski – podejście TDD.



Rys. 4. Podział projektu ze względu na kod wykorzystywany w badanych podejściach

Metryki CK, które miały za zadanie ocenić jakość kodu mierzyły następujące parametry:

- RFC – metryka mierząca odpowiedzialność klasy. Na jej wartość składa się liczba publicznych metod i jej podklas;
- CBO – metryka określająca powiązania pomiędzy obiektami. Uwzględnia liczbę odwołań pomiędzy klasami np. poprzez parametry metod;
- WMC – miara złożoności w klasie, składa się z ważonej liczby metod w klasie;
- LCOM3 – miara (wartość z przedziału <0,1>) spójności metod wewnątrz klasy.

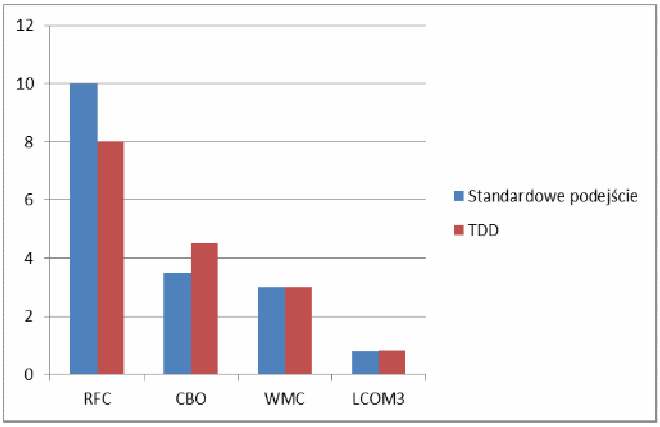
Analizę kodu przeprowadzono na klasach zawartych w katalogu *src/main/java*. To w tym katalogu znajduje się główna implementacja logiki. Do analizy nie wykorzystano klas z testami, ponieważ głównym celem badan był kod odpowiedzialny za logikę aplikacji.

Wartości metryk dla otrzymanego kodu przedstawione zostały w tabeli 4.

Tabela 4. Wartości metryk CK

	Standardowe podejście	**TDD**
RFC	10	8
CBO	3,5	4,5
WMC	3	3
LCOM3	0,8	0,82

Wyniki obu podejść są bardzo zbliżone do siebie i trudno jednoznacznie określić przewagę któregoś z nich. Graficzne zestawienie wyników przedstawiono na rysunku 5.



Rys. 5. Wyniki porównania wartości metryk

5. Wnioski

Wyniki badań wskazują na to, że zastosowanie techniki Test-Driven Development wpłynęło korzystanie na proces tworzenia kodu. Wbrew obawom, czas jaki należy przeznaczyć na zaprogramowanie funkcjonalności, nie wydłużył się. Fakt ten korzystnie wpływa na ocenę z biznesowego punktu widzenia.

Jakość kodu otrzymanego w badaniach utrzymuje się na wysokim poziomie. Na jakość kodu metodyka TDD nie ma bezpośredniego wpływu. Główną rolę odgrywają tutaj testy jednostkowe. To one zapewniają bezpieczeństwo oraz zachowanie odpowiednich struktur w pisanym kodzie. Dzięki zastosowaniu badanej techniki można uniknąć sytuacji, w której napisany kod nie jest możliwy do przetestowania. Zastosowanie Test-Driven Development samo w sobie wymusza pokrycie testami praktycznie całej aplikacji, dzięki czemu z góry można założyć, że otrzymany kod będzie dobrej jakości.

Oczywiście największą rolę w programowaniu odgrywa programista. Wiele zależy od jego osobistych upodobań i przyzwyczajeń. Sama metodyka nie zastąpi doświadczenia i kreatywności w programowaniu.

Literatura

[1] Pressman R.S., Software Engineering: A Practitioner's Approach 8th Edition, McGraw-Hill Higher Education, 2015.

[2] http://www.testerzy.pl [07.03.2017]

[3] Farcic V, Garcia A., *Test-Driven Java Development*, Packt Publishing, 2015.

[4] Mil M., Java EE 7 Development with Wildfly, Packt Publishing, 2014.

[5] https://docs.oracle.com/javaee/7/ [07.03.2017]

[6] Burke B., RESTful Java with JAX-RS 2.0. 2nd Edition, O'Reilly Media, 2013.

[7] http://site.mockito.org/ [07.03.2017]

[8] http://junit.org [03.01.2017]

[9] https://cooldevstuff.wordpress.com/tag/maven-jetty-plugin-2/, [10.02.207]