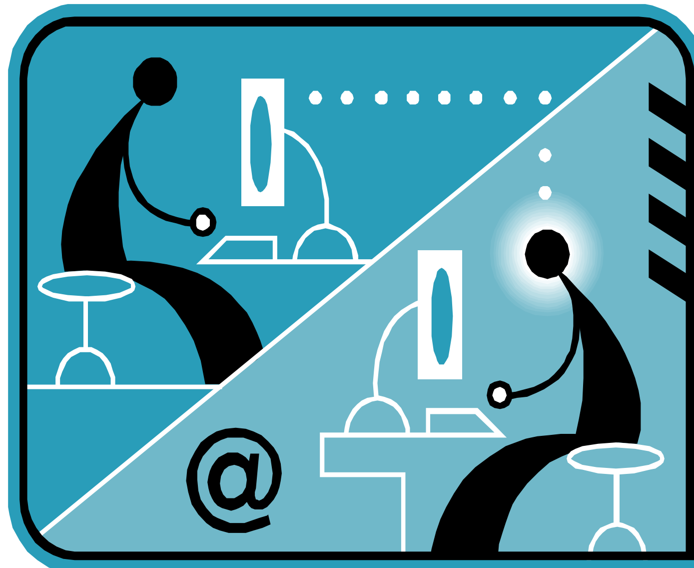
Inżynieria oprogramowania

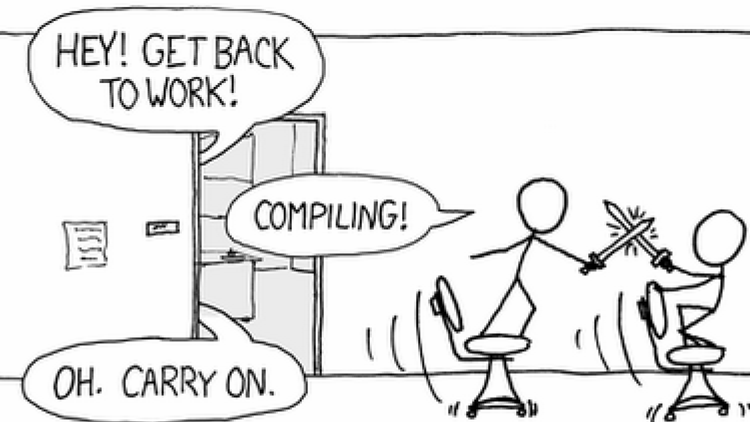
Robert Szmurło

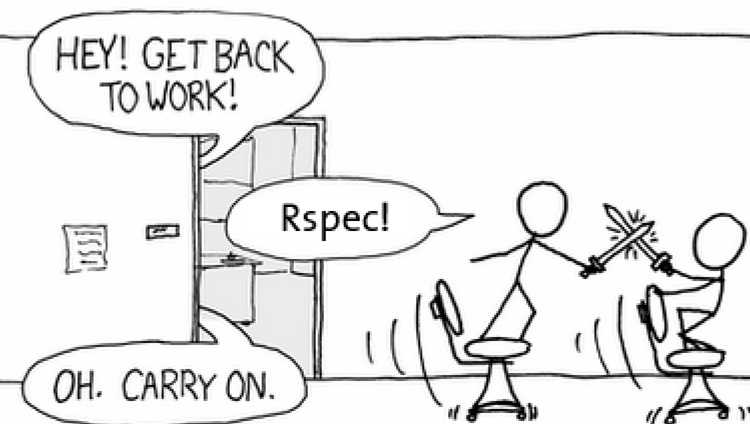


Agile Software Development:

- Test Driven Development
- Behaviour Driven Development

Automatyzacja





Original by XKCD.

Programowanie sterowane przez testy (TDD)

- Programowanie sterowane przez testy (ang. Test Driven Development, w skrócie TDD) jest jedną z najważniejszych technik w programowaniu ekstremalnym. Może być także stosowana osobno, jednak przynosi o wiele większe korzyści, jeśli będzie stosowana w połączeniu z innymi technikami wspomnianymi wcześniej np. w połączeniu z programowaniem w parach.
- Programowanie wymaga precyzyjnego określenia operacji, jakie program ma wykonywać. Niestety programiści nie są doskonali i popełniają błędy. W najlepszym wypadku błędy uniemożliwią kompilację kodu, a w najgorszym ujawnią się w najmniej oczekiwanym momencie, przy bardzo specyficznych warunkach, które nie zostały wcześniej przewidziane.
- Aby uniknąć błędów niezbędne jest testowanie kodu i co najważniejsze, powinno być wykonane w odpowiedni sposób.

Błędy podczas testowania

- Kent Beck wymienia najczęstsze błędy popełniane podczas testowania, które prowadzą do niepoprawnie działających aplikacji.
  - testy są niepełne, nie sprawdzają wszystkich możliwych problematycznych przypadków;
  - testowania systemu dokonuje się najczęściej po jego zaimplementowaniu, przez co programista, który pisał kod, może go mniej rozumieć i musi poświęcić dodatkowy czas na pełne zrozumienie;
  - testy często są pisane przez inne osoby niż te, które pisały kod; ponieważ mogą nie rozumieć wszystkich dokładnych szczegółów, mogą także ominąć ważne przypadki w testach;
  - testy nie są zautomatyzowane, przez co najprawdopodobniej nie są uruchamiane regularnie i w taki sam sposób;
  - naprawienie jednego problemu często powoduje pojawienie się problemu w innym miejscu, a istniejące testy często nie znajdują wtedy problemów w innych miejscach;

Przejście z TDD na BDD

- Oprócz terminologii, BDD określa, w jaki sposób należy definiować zachowanie kodu. W TDD powszechne było myślenie o kodzie w kategoriach jednostek. Jednostka oznacza przeważnie pewną klasę, udostępniającą wybraną funkcjonalność. Testy jednostkowe natomiast polegają na tworzeniu testów, których struktura ściśle odpowiada strukturze kodu, a ilość testów jest równa ilości metod w kodzie.
- Mario Gleichmann twierdzi w, że ścisłe powiązanie testów z metodami klas uniemożliwia dokonywanie refaktoryzacji kodu, ponieważ każda zmiana w implementacji kodu będzie wymagała zmiany testów. Myślenie w kategoriach jednostek powoduje koncentrację na metodach poszczególnych klas, a nie na zachowaniu aplikacji.
- Drugim problemem, który wynika z takiego podejścia jest testowanie stanu. W testach konkretnych metod możliwe jest zweryfikowanie stanu przed i po wykonaniu danej metody. Jednak w większości przypadków funkcjonalność wybranej klasy ma sens w połączeniu z innymi klasami. Zamiast testowania stanu należy skupić się na interakcji między poszczególnymi obiektami i opisywać, w jaki sposób ma wyglądać taka interakcja.

Analiza oraz wymagania funkcjonalne

- Definiowanie zachowania obiektów oraz opisywanie interakcji między nimi pozwala na projektowanie architektury aplikacji. Oprócz tego BDD pozwala na przeprowadzenie pełnej analizy systemu jako całości i łatwe definiowanie wymagań dotyczących tego systemu, a dokładniej jaka funkcjonalność powinna zostać zaimplementowana.
- BDD udostępnia język, a właściwie pewien szablon do opisywania poszczególnych właściwości systemu. Każdy szablon zaczyna się od zdefiniowania trzech elementów:
  - rodzaju użytkownika,
  - funkcjonalności,
  - korzyści jaką uzyska użytkownik dzięki tej funkcjonalności.

Scenariusze

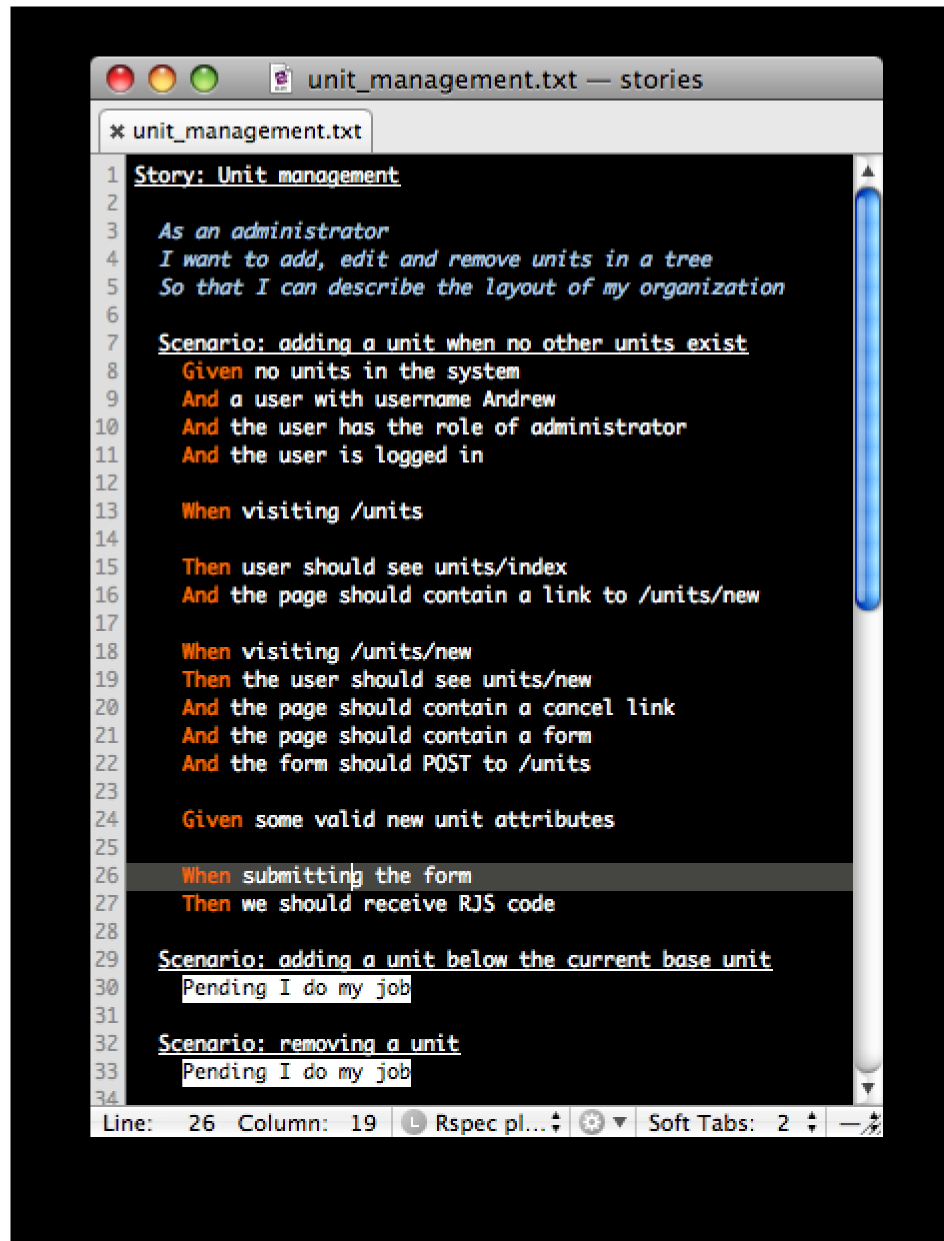
- Następnym etapem jest opisanie, jak ma wyglądać wykorzystanie funkcjonalności w postaci scenariuszy. Składają się one z opisu kryteriów początkowych, akcji jakie użytkownik może wykonać oraz oczekiwań odnośnie zachowania aplikacji po wykonaniu tych akcji. Poniżej znajduje się scenariusz, jak ma wyglądać logowanie do systemu:

```
Właściwość: logowanie do systemu
   Jako użytkownik
   Chcę się zalogować
   Abym mógł rozpocząć pracę z aplikacją

   Scenariusz: udane logowanie
      Mając zarejestrowanego użytkownika o nazwie ’Michał’
      i haśle ’tajne_hasło’
      Kiedy Michał otwiera stronę logowania
      Oraz wpisuje w polu nazwa ’Michał’
      Oraz wpisuje w polu hasło ’tajne_hasło’
      Oraz klika w przycisk Zaloguj

      Wtedy powinien zostać przekierowany do swojej
      spersonalizowanej strony
      Oraz powinien zobaczyć informację ’Witaj Michał’
```

Przykład scenariusza w narzędziu



Weryfikacja scenariuszy

- Każda funkcjonalność może mieć wiele scenariuszy rozpatrujących różne przypadki użycia, natomiast każdy scenariusz opisuje kryteria, jakie aplikacja powinna spełniać aby działała poprawnie. Przy wykorzystaniu odpowiednich narzędzi możliwe jest sprawdzenie, czy aplikacja spełnia wymagane kryteria i jeżeli tak, to znaczy, że działa poprawnie.
- Sprawdzanie czy aplikacja spełnia wymagane kryteria, określane jest jako wykonywanie testów akceptacyjnych. Opisane przez analityków, testerów czy nawet użytkowników systemu funkcjonalności przy pomocy BDD pełnią rolę dokumentacji, definicji wymagań systemu, a także automatycznych testów akceptacyjnych. Zachowujemy dzięki temu **spójność** systemu.
- Dzięki temu zaoszczędza się bardzo dużo czasu. Specyfikacje pozwalają na wygodne projektowanie systemu, a testy akceptacyjne pozwalają zweryfikować czy działa on poprawnie. Dodatkowo nie trzeba tworzyć dokumentacji systemu, ponieważ specyfikacje stworzone przy pomocy metodyki BDD same w sobie stanowią wystarczającą dokumentację.

RSpec

- RSpec pozwala na stosowanie wszystkich aspektów BDD, zarówno na opisywanie <u>pojedynczych obiektów przy pomocy specyfikacji</u>, jak i opisywanie <u>całej aplikacji za pomocą scenariuszy</u>. Dużą jego zaletą jest zdefiniowany język, specjalnie przystosowany do pisania specyfikacji.
- <u>Specyfikacje w RSpec są w rzeczywistości programami napisanymi w języku Ruby, jednak są bardzo zbliżone do naturalnego języka, przez co mogą być zrozumiałe nie tylko dla programistów, ale także dla innych osób biorących udział w projekcie.</u>
- Oprócz nowego języka RSpec udostępnia narzędzia, służące do automatycznego uruchamiania stworzonych specyfikacji oraz scenariuszy. Za każdym razem kiedy w projekcie zostanie zapisana dowolna zmiana, automatycznie zostaną wykonane odpowiednie testy, powiązane ze zmienionym plikiem. Po każdym takim przebiegu, generowane jest krótkie podsumowanie, które testy zostały zakończone pomyślnie, a które nie i jakie wystąpiły błędy.

Inne narzędzia

- Inne narzędzia podobne do RSpeca dla języka Ruby:
  - Shoulda - http://dev.thoughtbot.com/shoulda/
  - test/spec

Przykład użycia RSpeca - Specyfikacja

- Na początku należy stworzyć plik grade list.rb z pustą definicją klasy GradeList:

```
class GradeList
end
```

- Specyfikacja dla tej klasy wygląda (np. w pliku grade_list_spec.rb) następująco:

```
require 'grade_list'

describe "Lista ocen" do
    it "jest pusta po utworzeniu" do
        list = GradeList.new
        list.size.should eql(0)
    end
    it "oblicza średnią ocen" do
        list = GradeList.new
        list.add(3)
        list.add(4)
        list.add(5)
        list.get_average.should eql(4)
    end
end
```

Uruchomienie RSpeca

- Uruchamiamy z linii komend polecenie
  - spec -c --format specdoc grade_list_spec.rb.

```
ronin@ronin:~/ruby_projects$ spec -c --format specdoc grade_list_spec.rb

Lista ocen
- jest pusta po utworzeniu (ERROR - 1)
- oblicza średnią ocen (ERROR - 2)

1)
NoMethodError in 'Lista ocen jest pusta po utworzeniu'
undefined method `size' for #<GradeList:0xb7ab4c00>
./grade_list_spec.rb:6:

2)
NoMethodError in 'Lista ocen oblicza średnią ocen'
undefined method `add' for #<GradeList:0xb7ab365c>
./grade_list_spec.rb:10:

Finished in 0.008892 seconds

2 examples, 2 failures
ronin@ronin:~/ruby_projects$
```

  - Na początku generowana jest tekstowa dokumentacja, która składa się z napisów zawartych w specyfikacji. Na górze jest opis kontekstu, w tym przypadku jest to nazwa obiektu, a następnie każdy z testów zaczyna się od myślnika. Oba testy wypisane są w kolorze fioletowym, dodatkowo z informacją o wystąpieniu błędu wraz z numerem. Pod dokumentacją znajdują się szczegółowe informacje o błędach, które wystąpiły. W tym przypadku jest to brak metod size oraz add.

Pasek czerwony

- Aby pozbyć się tych błędów trzeba je zdefiniować w pliku grade_list.rb

```
class GradeList
    def size
    end

    def add(value)
    end

    def get_average
    end
end
```

- W tej chwili po ponownym uruchomieniu RSpec pojawi się wynik:

```
ronin@ronin:~/ruby_projects$ spec -c --format specdoc grade_list_spec.rb

Lista ocen
- jest pusta po utworzeniu (FAILED - 1)
- oblicza średnią ocen (FAILED - 2)

1)
'Lista ocen jest pusta po utworzeniu' FAILED
expected 0, got nil (using .eql?)
./grade_list_spec.rb:6:

2)
'Lista ocen oblicza średnią ocen' FAILED
expected 4, got nil (using .eql?)
./grade_list_spec.rb:13:

Finished in 0.009315 seconds

2 examples, 2 failures
ronin@ronin:~/ruby_projects$
```

Pasek zielony

- Kolejny etap to osiągnięcie tzw. zielonego paska, czyli napisanie jak najmniejszej ilości kodu, aby wszystkie przykłady zostały wypisane na zielono. Kod ten wygląda następująco:

```
class GradeList
    def size
        0
    end
    def add(value)
    end
    def get_average
        4
    end
end
```

- Wszystkie testy zostały wyświetlone na zielono, a więc drugi etap został zakończony.

```
ronin@ronin:~/ruby_projects$ spec -c --format specdoc grade_list_spec.rb

Lista ocen
- jest pusta po utworzeniu
- oblicza średnią ocen

Finished in 0.008212 seconds

2 examples, 0 failures
ronin@ronin:~/ruby_projects$
```

Ostateczna wersja kodu po refaktoryzacji

- Ostatni etap to rektoryzacja kodu tak, aby cały czas wszystkie testy wyświetlane były na zielono.

```
class GradeList
    def initialize
        @size = 0
        @sum = 0
    end

    def size
        @size
    end

    def add(value)
        @sum += value
        @size += 1
    end

    def get_average
        @sum / @size
    end
end
```

Definiowanie oczekiwań

- Oczekiwania są odpowiednikiem asercji w TDD i służą do sprawdzania, czy dany fragment kodu zachowuje się poprawnie. Nawiązując do przykładu z listą ocen, jednym z oczekiwań jest to, że po utworzeniu nowej listy powinna ona być pusty lub ilość ocen na liście powinna być równa 0.
- RSpec został stworzony z myślą o tym, aby powyższe oczekiwanie można było jak najprościej przełożyć na język programowania i żeby taki zapis był zrozumiały. Aby osiągnąć ten cel, RSpec definiuje metodę should, która jest dostępna dla każdego obiektu, jaki zostanie zdefiniowany w aplikacji. Metoda ta jako parametr przyjmuje specjalny obiekt, dokonujący właściwego porównania między wartością zadaną, a aktualną.

```
size.should eql(5)
```

- Kod ten sprawdza czy zmienna size jest równa 5. W tym celu zostaje wywołana metoda should, która w tym przypadku jako parametr przyjmuje obiekt klasy Eql, sprawdzający czy zmienna size jest równa 5. Dla prostoty oraz lepszej czytelności proces tworzenia tego obiektu jest ukryty w metodzie eql, czyli zapis eql(5) oznacza Eql.new(5).

Klasa oczekiwań

```
class Eql
	def initialize(expected)
		@expected = expected
	end

	def matches?(given)
		@given = given
		@given.eql?(@expected)
	end

	def failure_message
		return "expected #{@expected.inspect},
		got #{@given.inspect} (using .eql?)", @expected, @given
	end
end
```

- Najważniejsza jest metoda matches?, która dokonuje właściwego porównania. Jest ona wywoływana w środku metody should, a jako parametr przekazywany jest badany obiekt, czyli ten, na którym metoda should została wywołana.

Pozostałe klasy oczekiwań

- RSpec udostępnia moduł Matchers, zawierający kilkanaście klas służących do testowania obiektów oraz metody ukrywające przed programistą tworzenie tych obiektów. Możliwe jest także dodanie własnych metod.Wystarczy stworzyć nową klasę w tym module, która będzie zawierała metody matches? oraz failure message
- Analogicznie do metody should można stosować metodę should not, która po prostu sprawdza, czy podany warunek nie jest spełniony. Na listingu poniżej znajduje się kilka przykładowych oczekiwań:

```
# zmienna pi powinna mieć w
# przybliżeniu wartość 3.14,
# błąd może wynieść 0.01
pi.should be_close(3.14, 0.01)

# zmienna foo powinna istnieć
foo.should exist

# zmienna bar nie powinna być równa nil
bar.should_not be_nil

# tablica powinna zawierać wartość 2
[1,2,3].should include(2)

array = [1,2,3]
# kod powinien zmienić rozmiar tablicy o 1
lambda {
array.push(4)
}.should change(array, :size).by(1)

# kod ujęty w instrukcji lambda powinień
# zgłosić wyjątek
lambda {
user.do_something
}.should raise_error
```

Ćwiczenie

- Rozszerzymy klasę GradeList

Definiowanie kontekstów

- Każdy z przykładów użycia zdefiniowany jest w ramach pewnego kon-tekstu. Domyślnie jest to kontekst pewnej klasy, której zachowanie jest opisywane. Taki kontekst tworzy się przy pomocy metody describe, która jako parametr przyjmuje nazwę klasy oraz blok kodu, w którym definiowane są przykłady dla tego kontekstu.
- Bardzo często występuje jednak potrzeba definiowania przykładów dla specyficznych przypadków lub opisanie interakcji kilku powiązanych klas. W takiej sytuacji możliwe jest zdefiniowane dodatkowych kontekstów poprzez podanie ich opisów, a nie nazw klas. Możliwe jest także definiowanie zagnieżdżonych kontekstów.
- Przykładowym zastosowaniem jest definiowanie zachowania w zależności od różnych stanów danego obiektu.

```
describe "Lista ocen" do
    describe "po utworzeniu" do
        it "powinna mieć rozmiar 0" do
        ...
        end
    end
    describe "po dodaniu do niej kilku ocen" do
        it "powinna obliczać ich średnią" do
        ...
        end
    end
end
```

Definiowanie kontekstów - wynik

- Dla specyfikacji z poprzedniego slajdu RSpec wyświetli wynik:

```
ronin@ronin:~/ruby_projects$ spec -c --format specdoc grade_list_spec.rb

Lista ocen po utworzeniu
- powinna być pusta

Lista ocen po dodaniu do niej kilku ocen
- powinna obliczać ich średnią

Finished in 0.010187 seconds

2 examples, 0 failures
ronin@ronin:~/ruby_projects$
```

Tworzenie sztucznych obiektów

- Poprawność działania aplikacji zależy w największej mierze od poprawnej interakcji między poszególnymi obiektami. Dlatego właśnie, metoda BDD kładzie tak duży nacisk na opisywanie interakcji. Pojawia się tutaj pewien problem. Jeżeli przykład opisuje kilka współpracujących ze sobą obiektów i w implementacji jednego z nich zostanie wprowadzony błąd, to wtedy Rspec wyświetli ten przykład na czerwono. Może to zmylić programistę, ponieważ sugeruje, że błąd znajduje się w kodzie implementującym interakcję między obiektami, a nie w kodzie danego obiektu.
- Aby uniknąć takich problemów zaleca się stosowanie specjalnych obiektów, które można skonfigurować w taki sposób, żeby zachowywały się tak jak rzeczywiste obiekty aplikacji. RSpec udostępnia dwa mechanizmy do definiowania takich obiektów: **stub** oraz **mock**.

Metoda Stub

- Stub jest to sztuczna metoda, która przesłania metodę oryginalną. Nie wykonuje ona żadnego kodu, ale potrafi zwrócić z góry zadaną wartość.

```
object.stub!(:new_method).and_return(true)

# wynikie tej metody jest true
object.new_method
```

- Wpowyższym przykładzie przysłonięta została metoda new method, której wywołanie powoduje zawsze zwrócenie wartości true. Możliwe jest także przysłanianie metod przyjmujących parametry.Wywołanie takiej metody zawsze zwróci zadaną wartość.

Obiekt Mock

- Jeżeli zachodzi potrzeba utworzenia wielu sztucznych funkcji na jednym obiekcie, można zamiast tego utworzyć w pełni sztuczny obiekt mock, a następnie zdefiniować, jakie wartości powinny zwracać poszczególne metody.
- W pierwszym przykładzie pokazano przesłanianie metod za pomocą słowa stub:

```
object = Object.new
object.stub!(:method1).and_return(true)
object.stub!(:method2).and_return("hello")
object.stub!(:method3).and_return(10)
```

- Ten sam obiekt zdefiniowany za pomocą słowa mock:

```
object = mock('object', :method1 => true,
:method2 => "hello", :method3 => 10)
```

Mock szczegóły

- Metoda mock przyjmuje jeden obowiązkowy parametr, którym jest dowolny opis sztucznego obiektu. Dodatkowym parametrem jest mapa zawierająca nazwy metod, które mają być przysłonięte oraz wartości jakie mają zwracać. Możliwe jest także stworzenie dwóch obiektów typu mock, a następnie skonfigurowaniu ich tak, że jeden z nich będzie zwracał drugi:

```
msg_1 = mock('msg_1', :body => 'tresc 1')
msg_2 = mock('msg_2', :body => 'tresc 2')
user = mock('user', :messages => [msg_1, msg_2])
# kod zwraca wartość 'tresc 1'
user.messages[0].body
```

Zalety stosowania sztucznych obiektów

- Można definiować metody, które nie zostały jeszcze zaimplementowane na rzeczywistych obiektach. Dzięki temu można opisywać interakcję między obiektami pomimo, że te obiekty nie zostały jeszcze zaimplementowane.
- Można zdefiniować specyficzny stan obiektów, jaki może się zdarzyć w rzeczywistości, np. mając metodę authorized, która zwraca czy użytkownik ma dostęp do strony, można ustalić, żeby w jednym przypadku metoda ta zwracała true. Wtedy testujemy przypadek, kiedy użytkownik ma prawa dostępu. W drugim przypadku można ustawić tą metodę aby zwracała false, wtedy testujemy przypadek, kiedy użytkownik nie ma praw dostępu.
- Kolejną ważną zaletą jest wydajność takiego rozwiązania. Sztuczne funkcję nie wykonują żadnych operacji, jedynie zwracają ustalony wynik, więc ich stosowanie znacznie przyśpiesza czas wykonania specyfikacji przez RSpec. Oprócz tego, jeżeli aplikacja komunikuje się z zewnętrznymi serwisami, to zysk wydajnościowy jest jeszcze większy. Ponadto zewnętrzny serwis może być niedostępny w wyniku awarii sprzętu, a wtedy RSpec wyświetli błędy we wszystkich testach, gdzie ten serwis jest wykorzystywany.

Ćwiczenie

- Doprowadzić GradeList aby wszystkie paski były zielone.

```
require 'grade_list'
describe "Lista ocen" do

(…)

	it "oblicza liczbe poszczegolnych ocen" do
		list = GradeList.new
		list.add(3)
		list.add(3)
		list.add(5)
		list.add(4)
		list.add(4)

		liczba_ocen = list.get_grades_count
		liczba_ocen[3].should eql(2)
		liczba_ocen[4].should eql(2)
		liczba_ocen[5].should eql(2)
	end
end
```

Co potrzeba:
1) zaimplementowac metode
2) dodac tablice do klasy GradeList: @oceny
3) zainicjalizowac tablice w konstruktorze @oceny = [ ]
4) dodac w metodzie add dodawanie oceny do tablicy: @oceny.push(value)
5) Zwrocic tablice asocjacyjna z informacja o ocenach
   a) o_info = {}
   b) wykonac petle po wszystkich elementach tablicy @oceny.each { |k| …}
   c) wypelnic tablice asocjacyjna np. tak: @o_info = (o_info[k] == nil ? 1 : o_info[k] + 1)

RSpec w połączeniu z Ruby on Rails

- Framework Ruby on Rails posiada wbudowane mechanizmy ułatwiające testowanie. Dotyczą one głównie udostępnienia pewnych klas, które pozwalają na symulowanie rzeczywistych działań użytkownika, jednak wszystko odbywa się bez użycia przeglądarki internetowej.
- Wszystkie testy wykonywane są w środowisku testowym i wykorzystują osobną bazę danych, która jest czyszczona przed każdym przebiegiem.
- W celu skorzystania z RSpec-a w Ruby on Rails należy skorzystać z rozszerzenia rspec-rails. Tworzy ono dodatkowy podkatalog spec, w którym będą umieszczane wszystkie specyfikacje. RSpec wykorzystuje wbudowane mechanizmy testujące, jednak udostępnia zestaw metod oraz klas pozwalających na pisanie specyfikacji zgodnie z metodyką BDD.

Opisywanie modeli

- Modele w aplikacji zajmują się przechowywaniem danych. W przypadku Ruby on Rails, najczęściej każdy model posiada powiązaną z nim tabelę w bazie danych. Przykładowa specyfikacja:

```
describe Post do
    it "powinien wymagać podania tytułu" do
        post = Post.new(:title => nil, :body => 'treść wiadomości')
        post.should_not be_valid
    end
    it "powinien wymagać podania treści" do
        post = Post.new(:title => 'tytuł', :body => nil)
        post.should_not be_valid
    end
end
```

- Jest to specyfikacja opisująca model przechowujący dane wiadomości np. na blogu internetowym. Opis pierwszego przykładu określa, że wiadomość powinna wymagać podania tytułu. Oczekiwanie to zostanie spełnione jeśli metoda post.valid? zwróci wartość true. Analogicznie można korzystać z oczekiwań post.should be nil, post.should be_blank czy post.should be_empty. Oczekiwania te zostaną spełnione, jeśli odpowiednie metody post.nil?, post.blank? oraz post.empty? zwrócą wartość true. Korzystając z tego schematu można stworzyć dowolne oczekiwanie, które będzie zaczynało się od be .

Opisywanie modeli

- Aby powyższa specyfikacja zostały spełniona, należy zaimplementować model Post i dodać do niego odpowiednie walidacje:

```
class Post < ActiveRecord::Base
    validates_presence_of :title
    validates_presence_of :body
end
```

- Można także sprawdzić, czy w przypadku poprawnych danych są one rzeczywiście zapisywane w bazie:

```
describe Post do
    it "powinien zapisywać nowy post w bazie danych" do
        post = Post.new(:title => ’tytuł’, :body => ’treść’)
        lambda { post.save }.should change(Post, :count).by(1)
    end
end
```

- W tym przypadku niezbędne jest zastosowanie konstrukcji lambda. Znajduje się w niej wywołanie metody save, która zapisuje rekord w bazie danych. Wykorzystane jest tutaj oczekiwanie should change, które pozwala sprawdzić, czy pewna wartość uległa zmianie. W tym przypadku tą wartością jest ilość rekordów w tabeli posts.

Opisywanie modeli

- Oczekiwanie change jest bardzo elastyczne i eliminuje konieczność zapamiętywania wartości przed wywołaniem kodu. To samo można zapisać następująco:

```
it "should save valid record to database" do
    count = Post.count
    post = Post.new(:title => ’tytuł’, :body => ’treść’)
    post.save
    Post.count.should eql(count+1)
end
```

- Powyższe rozwiązanie jest jednak mniej eleganckie i wymaga zapisania w zmiennej lokalnej wcześniejszej liczby rekordów.

Opisywanie kontrolerów

- Celem kontrolerów jest sterowanie aplikacją, a dokładniej interpretowanie parametrów przesyłanych w zapytaniu, wykonanie pewnych operacji na bazie danych przy pomocy warstwy modeli, wygenerowanie kodu HTML przy pomocy warstwy widoków i na koniec wysłanie odpowiedzi do użytkownika.
- Aby możliwe było sprawdzanie, czy działanie kontrolerów jest zgodne ze specyfikacją, należy w pewien sposób symulować wysyłanie przez użytkownika zapytań, które w efekcie wywołują odpowiednie akcje wybranych kontrolerów.
- Framework Ruby on Rails w klasie Session udostępnia wiele metod umożliwiających interakcję z kontrolerami, jednak najbardziej istotne są: get, post, put oraz delete. Każda z nich odpowiada innemu rodzajowi zapytania w protokole HTTP. Dodatkowo istnieją 4 analogiczne metody: get via redirect, post via redirect itd. Te ostatnie różnią się od pierwszych tym, że jeżeli odpowiedź na dane zapytanie jest przekierowaniem, to wysyłają kolejne zapytanie na przekierowany adres i tak dopóki odpowiedź jest ciągle przekierowaniem.

Opisywanie kontrolerów

- Obowiązkowym parametrem każdej z tych metod jest nazwa akcji kontrolera przekazana jako symbol np. :index. Dodatkowo możliwe jest przekazanie dodatkowych parametrów oraz ustawienie wybranych nagłówkóch HTTP:

```
get :index
post :create, {:title => ’Tytuł’, :body => ’Opis’}
get :index, {}, {:referer => ’http://google.pl’}
```

- Pierwszy przykład wywołuje akcję index przy pomocy zapytania HTTP GET. Drugi symuluje wysyłanie formularza przez użytkownika. W formularzu zostały wypełnione dwa pola o nazwach title oraz body, a następnie formularz został wysłany do akcji create przy pomocy zapytania HTTP POST. Ostatni przykład ma takie samo działanie, jak pierwszy, tylko dodatkowo wysyłany jest dodatkowy nagłówek HTTP REFERER.

Opisywanie kontrolerów

- Po wysłaniu symulowanego zapytania, generowana jest odpowiedź aplikacji, do której można się odwoływać przez zmienną **response**. Pod zmienną **assigns** dostępne są natomiast wszystkie zmienne utworzone w kontrolerze, które zostały przekazane do widoku.
- RSpec udostępnia kilka specyficznych oczekiwań, w których można wykorzystywać te dwa obiekty:

```
# spełnione jeśli odpowiedź ma kod 200
response.should be_success

# spełnione jeśli odpowiedź jest przekierowaniem
response.should be_redirect

# spełnione jeśli odpowiedź jest przekierowaniem
# pod adres http://google.pl
response.should redirect_to('http://google.pl')

# spełnione jeśli odpowiedź zawiera kod HTML
# wygenerowany przy pomocy podanego widoku
# posts/new.html.erb
response.should render_template('posts/new')

# spełnione jeśli kontroler zdefiniował zmienną @user
assigns[:user].should_not be_nil
```

Prosta specyfikacja kontrolera

```
describe UsersController do
	describe "podczas przetwarzania akcji new" do
		it "powinien utworzyć nowego użytkownika" do
			User.should_receive(:new)
			get :new
		end
		it "powinien przekazać do widoku zmienną @user" do
			get :new
			assigns[:user].should_not be_nil
		end
		it "powinien zwrócić kod HTTP 200" do
			get :new
			response.should be_success
		end
		it "powinien wygenerować widok z pliku users/new.html.erb" do
			get :new
			response.should render_template('users/new')
		end
	end
end
```

- Prosta specyfikacja kontrolera dla akcji **new**. Jej celem jest wyświetlenie formularza, za pomocą którego użytkownik może dokonać Rejestracji.
- Pojawia się tu specjalne oczekiwanie should receive, które sprawdza czy podana metoda została wywołana na podanym obiekcie. W tym przypadku oczekiwane jest wywołanie metody new na klasie User. Kolejność zapisania tego oczekiwania jest celowa. Musi ono zostać zdefiniowane przed użyciem testowanego kodu.

Implementacja takiej specyfikacji

```
class UsersController < ApplicationController
    def new
        @user = User.new
    end
end
```

- Kod akcji **new** jest bardzo prosty. Tworzony jest nowy obiekt klasy User i przypisywany do zmiennej @user. Framework na podstawie nazwy kontrolera oraz akcji domyślnie generuje widok z pliku users/new.html.erb i wysyła go do przeglądarki z kodem HTTP 200.
- Można łatwo zauważyć, że specyfikacja zajmuje 4 razy więcej kodu niż implementacja. Sytuacja wygląda jeszcze gorzej, kiedy weźmie się pod uwagę pełną specyfikację dla kontrolera implementującego wszystkie akcje REST, których jest 7. Przeważnie każda z tych akcji wymaga rozpatrzenia 2 lub więcej przypadków. Pierwszego kiedy wszystkie operacje wykonują się pomyślnie i drugiego kiedy występuje jakiś błąd np. nie udało się zapisać użytkownika do bazy danych. Pełna specyfikacja może zawierać setki linii kodu, podczas gdy implementacja tylko 50.
- Taki sposób pisania specyfikacji był stosowany w wielu projektach komercyjnych oraz w początkowej fazie budowy opisywanej aplikacji, jednak okazało się to bardzo niepraktyczne, ze względu na konieczność pisania ogromnej ilości kodu w stosunku do właściwej implementacji.

Opisywanie widoków

- Specyfikacje widoków służą do określenia, jakie dane powinny być w nich wyświetlane oraz sprawdzania ogólnej struktury kodu HTML. W Ruby on Rails kontroler odpowiada za wywołanie widoku i przekazanie do niego da nych. W RSpec natomiast specyfikacje widoków są uruchamiane osobno, bez udziału kontrolera, a więc trzeba wyraźnie określić, jaki widok ma zostać wygenerowany oraz jakie dane powinno zostać do niego przekazane. Do generowania widoku służy metoda render, której trzeba przekazać nazwę widoku.
- Do przekazywania danych natomiast służy zmienna assigns.

Opisywanie widoków

```
describe "Strona powitalna" do
    before do
        mocked_user = mock('mocked user', :name => 'Michał')
        assigns[:user] = mocked_user
        render "pages/welcome"
    end

    it "powinna wyświetlać nazwę użytkownika" do
        response.should have_text(/Michał/)
    end
    it "powinna wyświetlać link do wyszukiwarki" do
        response.should have_tag("a[href=?]",
            "http://www.google.pl", "Wyszukiwarka")
    end
end
```

- Specyfikacja wykorzystuje blok kodu before, który jest wykonywany przed każdym przykładem. Pozwala to uniknąć duplikacji kodu. W bloku tym najpierw tworzony jest sztuczny obiekt, zawierający metodę name. Następnie obiekt ten zostaje przypisany do zmiennej assigns pod kluczem :user, dzięki czemu obiekt ten będzie dostępny w widoku pod zmienną @user. Na koniec wywoływana jest metoda render, która generuje widok z pliku pages/welcome.html.erb i sprawdzane są oczekiwania.

Opisywanie widoków

- Widok spełniający wcześniejszą specyfikację znajduje się poniżej.

```
<html>
<body>
    Witaj <%= @user.name %><br />
    <a href="http://www.google.pl">Wyszukiwarka</a>
</body>
</html>
```

- Podsumowując można stwierdzić, że nie warto stosować specyfikacji widoków stosując przedstawiony sposób, ponieważ ograniczają one możliwości późniejszej zmiany struktury HTML oraz wymagają napisania dużej ilości kodu tworzącego sztuczne obiekty.
- RSpec udostępnia jednak inny, dużo lepszy sposób na opisywanie widoków. Są to scenariusze użycia.

Scenariusze użycia

- Wykorzystywane są do definiowania wymagań funkcjonalnych aplikacji.
- RSpec zawiera narzędzie **cucumber**, które pozwala na uruchamianie takich scenariuszy i sprawdzanie, czy aplikacja działa poprawnie.
- Wszystkie opisy funkcjonalności umieszczane są w podkatalogu projektu o nazwie features (właściwosci). Aby wygenerować ten katalog razem z plikami konfiguracyjnymi należy użyć polecenia **script/generate cucumber**. Każdą funkcjonalność należy umieszczać w osobnym pliku z rozszerzeniem feature. Poniżej znajduje się opis, co powinno się stać, jeżeli użytkownik odwiedzi stronę główną. Opis ten znajduje się w pliku features/example.feature.

```
Właściwość: powitanie użytkownika

    Aby użytkownik był zadowolony
    Musimy go ładnie przywitać

    Scenariusz: powitanie uzytkownika
        Jeżeli użytkownik wchodzi na stronę główną
        Wtedy powinien zobaczyć tekst "Witaj drogi użytkowniku"
```

Scenariusze użycia

- Słowa Właściwość, Scenariusz, Jeżeli i Wtedy są słowami kluczowymi i muszą być napisane w takiej postaci (w przypadku języka polskiego). Dostępne są jeszcze słowa Dane, Oraz i Ale. Pozostała treść jest dowolna. Po opisaniu funkcjonalności można ją uruchomić poleceniem cucumber –language features/example.feature.

```
ronin@ronin:~/rails_apps/portal$ cucumber --language pl features/example.feature
Właściwość: powitanie użytkownika  # features/example.feature

  Aby użytkownik był zadowolony
  Musimy go ładnie przywitać
  Scenariusz: powitanie uzytkownika                          # features/example.feature:6
    Jeżeli użytkownik wchodzi na stronę główną               # features/example.feature:8
    Wtedy powinien zobaczyć tekst "Witaj drogi użytkowniku"  # features/example.feature:10


2 steps pending

You can use these snippets to implement pending steps:

Jeżeli /^użytkownik wchodzi na stronę główną$/ do
end

Wtedy /^powinien zobaczyć tekst "Witaj drogi użytkowniku"$/ do
end

ronin@ronin:~/rails_apps/portal$
```

Scenariusze użycia

- Cucumber wyświetla pełną treść pliku z tym, że wszystkie zdania w scenariuszu są w kolorze żółtym. Każde zdanie scenariusza określane jest jako krok (ang. step). Cucumber wykonuje kolejno wszystkie kroki w scenariuszu, jednak przy pierwszym wykonaniu narzędzie to nie wie, jakie operacje musi wykonać.
- Implementacje kroków należy umieszczać w katalogu features/step definitions, w plikach o nazwie zakończonej steps.rb np. all steps.rb. Na rysunku 5.6 widać, że Cucumber podpowiada, jakie kroki dokładnie należy zaimplementować pod treścią scenariusza. Wystarczy skopiować wypisany kod do pliku all steps.rb i zamienić słowa kluczowe Jeżeli, Wtedy odpowiednio na When oraz Then. W treści scenariusza słowa kluczowe występują w wybranym języku, jednak w plikach implementujących kroki słowa te są nazwami metod i mogą występować tylko w języku angielskim. Plik all steps.rb wygląda następująco:

```
When /^użytkownik wchodzi na stronę główną$/ do
end
Then /^powinien zobaczyć tekst "Witaj drogi użytkowniku"$/ do
end
```

- Wynik:

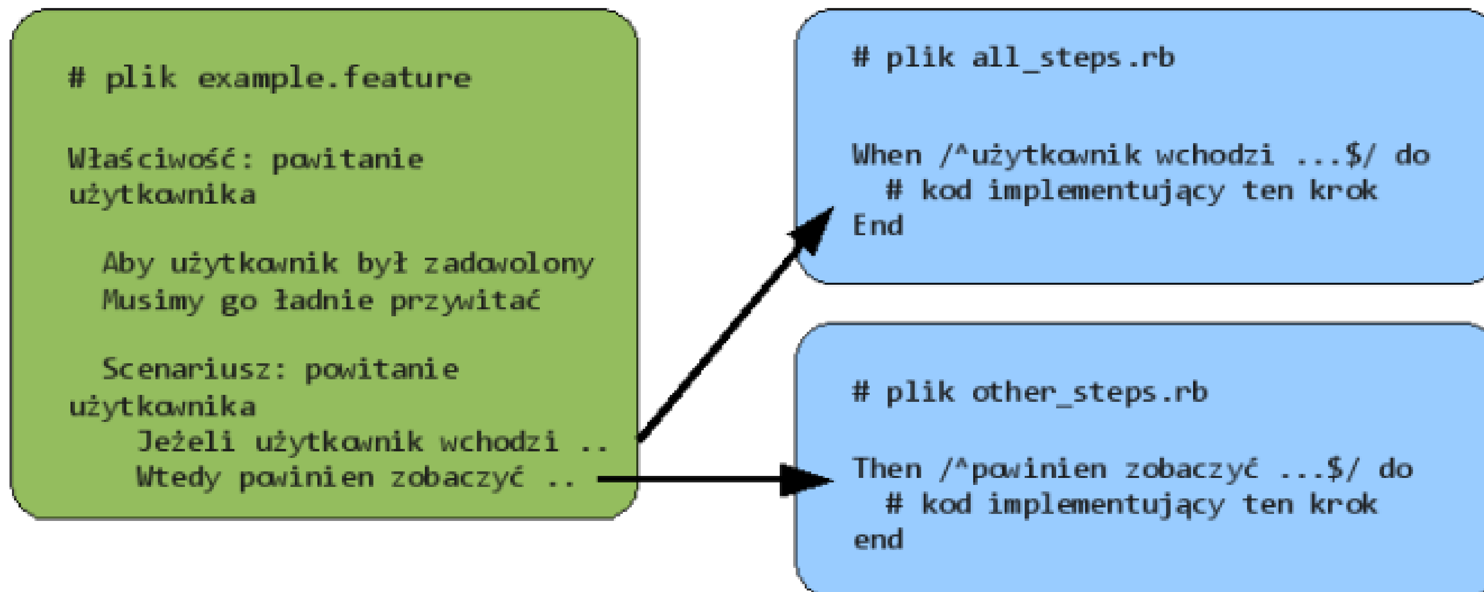
```
ronin@ronin:~/rails_apps/portal$ cucumber --language pl features/example.feature
Właściwość: powitanie użytkownika  # features/example.feature

  Aby użytkownik był zadowolony
  Musimy go ładnie przywitać
  Scenariusz: powitanie uzytkownika                          # features/example.feature:6
    Jeżeli użytkownik wchodzi na stronę główną               # features/step_definitions/all_steps.rb:1
    Wtedy powinien zobaczyć tekst "Witaj drogi użytkowniku"  # features/step_definitions/all_steps.rb:4


2 steps passed
ronin@ronin:~/rails_apps/portal$
```

Scenariusze użycia

- Ostatnim etapem jest implementacja kroków, tak aby podczas ich przetwarzania cucumber rzeczywiście wykonywał opisane operacje. I tak dla kroku Jeżeli użytkownik wchodzi na stronę główną powinien zasymulować akcję wejścia użytkownika na stronę główną. Dla drugiego kroku Wtedy powinien zobaczyć tekst "Witaj drogi użytkowniku", natomiast powinien spraw dzać, czy na głównej stronie znajduję się podany tekst.



Moduł webrat

- Wcześniej opisany został sposób na symulowanie akcji użytkownika, jednak dotyczyło to tylko jednego kontrolera. Cucumber został zintegrowany z biblioteką webrat, która nie ma tego ograniczenia oraz pozwala na imitowanie akcji użytkownika, takich jak: klikanie na odnośniki, wypełnianie formularzy czy naciskanie przycisków
- Przedstawione wcześniej kroki można zaimplementować następująco:

```
When /^użytkownik wchodzi na stronę główną$/ do
    visits "/"
end
Then /^powinien zobaczyć tekst "Witaj drogi użytkowniku"$/ do
    response.should have_text(/Witaj drogi użytkowniku/)
end
```

Przykłady implementacji kroków

- Cucumber zawiera plik step definitions/webrat_steps.rb w którym zdefiniowane są powszechnie wykorzystywane akcje użytkownika.
- Przykłady:

```
When /^odwiedzam stronę główną$/ do
    visits "/"
end

When /^odwiedzam stronę "(.+)"$/ do |url|
    visits url
end

When /^w polu "(.+)" wpisuję "(.+)"$/ do |field, value|
    fills_in(field, :with => value)
end

When /^klikam na "(.+)"$/ do |link|
    clicks_link(link)
end

When /^naciskam przycisk "(.+)"$/ do |button|
    clicks_button(button)
end

Then /^powinienem zobaczyć tekst "(.+)"$/ do |text|
    response.should have_text(/#{text}/m)
end
```

Tablice FIT

- Tablice FIT są nową funkcją zaimplementowaną w cucumber. Służą one do łatwego zapisywania w scenariuszach powtarzających się kroków.
- Jeżeli przykładowo w scenariuszu należy stworzyć 3 użytkowników, można stworzyć jeden krok Dane jest użytkownik o imieniu ‘‘Imię’’ oraz nazwisku ‘‘Nazwisko’’ i powtórzyć go trzy razy:

```
Dane jest użytkownik o imieniu "Jan" oraz nazwisku "Kowalski"
Oraz jest użytkownik o imieniu "Zbigniew" oraz nazwisku "Nowak"
Oraz jest użytkownik o imieniu "Michał" oraz nazwisku "Młoźniak"
```

- Implementacja tego kroku może wyglądać tak jak na listingu poniżej:

```
Given /jest uzytkownik o imieniu "(.+)" oraz nazwisku "(.+)"/
      do |first_name, last_name|
      User.create(:first_name => first_name,
                  :last_name => last_name)
end
```

- Jak widać mimo, że zmieniają się tylko parametry, trzeba pisać pełne zdania. Jeżeli byłaby potrzeba opisania większej ilości użytkowników albo podania więcej parametrów, to wtedy taki scenariusz stałby się bardzo nieczytelny.

Tablice FIT

- Za pomocą tablic FIT ten sam scenariusz można zapisać o wiele prościej i czytelniej.

```
Dane są użytkownicy:
    | imię | nazwisko |
    | Jan | Kowalski |
    | Zbigniew | Nowak |
    | Michał | Młoźniak |
```

- Pierwszy wiersz tej tabeli określa nazwy poszczególnych kolumn, a pozostałe zawierają odpowiednie wartości. Implementacja tego kroku:

```
Given /są użytkownicy:/ do |table|
    table.hashes.each do |hash|
        User.create(:first_name => hash['imię'],
                    :last_name => hash['nazwisko'])
    end
end
```

- Do kroku przekazywany jest obiekt table, którego metoda hashes zwraca tablicę zawierającą wartości dla poszczególnych wierszy. Każdy element tej tablicy jest mapą przechowującą wartości dla poszczególnych kolumn.

Tablice FIT – Pełny przykład

Właściwość: logowanie użytkownika
Aby zarejestrowani użytkownicy mogli skorzystać z funkcji portalu
Powinni mieć możliwość zalogowania

Scenariusz: udane logowanie
Dane jest, że nie jestem zalogowany
Oraz są następujący użytkownicy w bazie danych:

login	hasło	email
michał	test	michal-testowy@test.pl
tomek	test	tomek-testowy@test.pl
rafał	test	rafal-testowy@test.pl

Jeżeli odwiedzam stronę "/zaloguj"
Oraz w polu "Login" wpisuję "michał"
Oraz w polu "Hasło" wpisuję "test"
Oraz naciskam przycisk "Zaloguj"

Wtedy powinienem zobaczyć tekst "Witaj michał"

Kroki dla przykładu

```
Given /^jest, że nie jestem zalogowany$/ do
end

Given /^są następujący użytkownicy w bazie danych:$/ do |table|
    table.hashes.each do |hash|
            options = {
                :login => hash[’login’],
                :email => hash[’email’],
                :password => hash[’hasło’],
                :password_confirmation => hash[’hasło’]
            }
            options[:id] = hash[’id’] unless hash[’id’].nil?
            Factory.create_valid_user(options)
    end
end
```

Dziękuję za uwagę.

Chcemy być coraz lepsi!

Jeżeli coś cię zainteresowało napisz e-maila:

- robert@iem.pw.edu.pl

Jeżeli coś cię bardzo znudziło napisz e-maila:

- robert@iem.pw.edu.pl

Jeżeli zauważyłeś błąd napisz e-maila:

- robert@iem.pw.edu.pl