Testowo Zorientowane Metodyki Rozwoju Oprogramowania

Marcin Baliński

13 czerwca 2011

Spis treści

Rozdział 1

Wstęp

Metodyka to ustandaryzowane dla wybranego obszaru podejście do rozwiązywania problemów. Praca ta ma na celu szczegółowe omówienie dwóch najbardziej znanych metodyk rozwoju oprogramowania w oparciu o testy: *Test Driven Development* oraz *Behavior Driven Development* wraz ze szczegółowym opisem narzędzi wspomagających rozwój oprogramowania napisanego w języku *Ruby* w zgodzie z tymi metodykami.

Kolejnym nie mniej ważnym celem jest pokazanie jak niezmiernie istotne jest testowanie kodu. Opiszę zarówno problemy wynikające z braku odpowiedniego pakietu testów jak i korzyści płynące z prawidłowego wdrożenia testowo zorientowanych metod rozwoju do projektu informatycznego. Nie zabraknie również wskazówek odnośnie najlepszych praktyk jakie należy stosować przy wdrażaniu testowo zorientowanych metodyk.

We wstępie omówię krótko powszechne problemy i trudności związane z rozwojem oprogramowania oraz bardzo ogólnie opiszę ideę jego testowania. Kolejne dwa rozdziały to szczegółowy opis technik *Test Driven Development* oraz *Behavior Driven Development* oraz narzędzi wspomagających ich wdrożenie w projektach tworzonych przy pomocy języka *Ruby*. Rozdział czwarty to studium przypadku - praktyczny opis tego jak w praktyce wygląda *Behavior Driven Development* na przykładzie rozwoju projektu Open Source. Rozdział czwarty kończy się podsumowaniem zalet testowo zorientowanych metodyk rozwoju oprogramowania oraz czynników determinujących powodzenie ich wdrożenia wraz z krótkimi wnioskami.

1.1 Problemy napotykane w procesie wytwarzania oprogramowania

Na każdy projekt informatyczny można patrzeć z różnych perspektyw, w tej pracy chciałby jednak skupić się na technicznym aspekcie, jakim jest faza jego rozwoju w wybranym języku programowania, zaczynając od opisania częstych problemów podczas tej fazy.

Rozwój oprogramowania nie jest rzeczą trywialną. Po dogłębnej analizie potrzeb, wybraniu narzędzi, które posłużą do budowy systemu następuje faza implementacji, której trudność zależy od wielu czynników takich jak:

Rodzaj wybranych narzędzi Czy wybrane środki techniczne takie jak język programowania lub zestaw zewnętrznych bibliotek nadają się do rozwiązania tego typu problemu?

Stopień skomplikowania systemu

Stopień integracji systemu Czy łatwo oddzielić od siebie poszczególne wewnętrzne funkcje systemu? Czy konieczna jest integracja z zewnętrznym oprogramowaniem?

Wielkość zespołu programistów

Stopień technicznej świadomości uczestników projektu Wpływa na jakość komunikacji z programistami.

Powyższe czynniki wpływają bezpośrednio na problemy, które pojawiają się podczas implementacji systemu. Rozwojowi oprogramowania najczęściej towarzyszą problemy takie jak:

Wzrost stopnia skomplikowania bazy kodu Kod staje się coraz bardziej skomplikowany i trudniejszy w utrzymaniu, wynika to często z braku ustalonych konwencji, polityki włączania do projektu zewnętrznych rozwiązań lub słabej komunikacji w zespole programistów.

Niepotrzebny wzrost stopnia integracji Łamanie zasady modułowego tworzenia oprogramowania, poszczególne części systemu są ze sobą coraz bardziej związane i znacząco na siebie wpływają. Powoduje to sytuację, w której usterka w jednym module powoduję awarię w kilku innych częściach systemu.

Trudność w utrzymaniu systemu zgodnie z dostarczoną specyfikacją Wynikająca z niewłaściwej komunikacji lub z wcześniejszych błędów.

Dokładniejszą analizę przyczyn i skutków problemów napotykanych podczas procesu rozwoju oprogramowania znaleźć można między innymi w takich pozycjach jak [8] oraz [6]. W świetle tych informacji logicznym wydaje się wprowadzenie narzędzia kontroli, dzięki któremu można by upewnić się co do jakości dostarczonych rozwiązań a także zminimalizować ryzyko pojawienia się podobnych problemów w przyszłości. Jednym z takich narzędzi są testy oprogramowania.

1.2 Testowanie oprogramowania

1.2.1 Wady manualnego testowania

Testowanie oprogramowania może odbywać się w sposób manualny lub automatyczny. Testy manualne przeprowadzane są przez żywego testera, który korzystając z oprogramowania, krok po kroku sprawdza jego zgodność ze specyfikacją, następnie wskazuje i opisuje ewentualne braki lub błędy. Każda nowa funkcjonalność lub poprawka wprowadzona do oprogramowania wymaga osobnej sesji z udziałem testera.

Podejście manualne ma wiele wad, wśród których do najważniejszych należą:

- Konieczność dogłębnego zrozumienia założeń projektu przez osobę odpowiedzialną za testowanie
- Trudność związana z koniecznością zidentyfikowania i przetestowania jak największej liczby możliwych przypadków użycia oprogramowania
- Czasochłonność: każda nowa funkcjonalność lub poprawka wymaga osobnej sesji testowania
- Wysokie koszty pracy testera
- Ogromna trudność zastosowania w wysoce specjalistycznych projektach
- Brak możliwości dokładnego przetestowania szczegółów implementacji danej funkcjonalności

Waga powyższych niedogodności rośnie wykładniczo wraz ze wzrostem poziomu skomplikowania oprogramowania, dlatego też manualne testowanie sprawdza się w zasadzie tylko w projektach o małej złożoności. W innych przypadkach istnieje potrzeba uzupełnienia lub zastąpienia żywego testera przez testy automatyczne.

1.2.2 Testy automatyczne

Automatyzacja procesu testowania odbywa się poprzez zastąpienie testera oprogramowaniem, które przejmie jego rolę. Automatyczne metody testowania umożliwiają sprawdzenie działania kodu programu, jak również graficznego interfejsu użytkownika.

Testowanie kodu

W procesie tym testujemy szczegóły implementacji systemu. Oprócz kodu potrzebnego do zrealizowania danej funkcjonalności programiści piszą również testy weryfikujące jej implementację. Testy takie mogą mieć różne funkcje, wymienię tylko niektóre z nich:

Testy jednostkowe Sprawdzają pojedynczy, niepodzielny element implementacji taki jak metoda lub funkcja

Testu integracyjne Sprawdzają interakcję między składowymi elementami systemu

Celem testu może być wynik działania danej części kodu, może być nim również chęć upewnienia się, że wynik działania osiągnięty jest w konkretny sposób. Dla przykładu testując funkcję, której zadaniem jest wyświetlić na ekranie monitora napis *Witaj Świecie!* możliwe jest, że prócz samego faktu pojawienia się treści na ekranie chcemy również upewnić się, że do jej wyświetlenia użyta została jakaś konkretna metoda pochodząca z biblioteki standardowej. Jest to duża przewaga w stosunku to manualnego testowania oprogramowania, które nie daje nam takiej możliwości kontrolowania procesów prowadzących do widzialnych rezultatów.

W chwili obecnej istnieją dziesiątki gotowych narzędzi pozwalających testować kod napisany w każdym szerzej używanym języku programowania. Ich używanie należy do podstaw każdej nowoczesnej metodyki prowadzenia projektów informatycznych.

Testowanie interfejsu użytkownika

Istnieje szereg narzędzi pozwalających testować zachowanie, oraz wygląd interfejsów użytkownika, ich działanie opiera się najczęściej na nagrywaniu i późniejszym odtwarzaniu testowanych interakcji oraz porównywaniu ich rezultatów z naszymi oczekiwaniami. W taki sposób można testować tradycyjne aplikacje, jak również aplikacje www, działające w przeglądarce[1]

[1] w tym przypadku szczegóły działania narzędzi testujących są inne, interfejs użytkownika jest bowiem najczęściej zdefiniowany przez znaczniki HTML.

W kwestii testowania interfejsu użytkownika przewaga automatycznych testów nie jest już tak druzgocąca jak w przypadku testowania kodu, jednak i tutaj jesteśmy w stanie znacząco skorzystać na automatyzacji, zyskiem jest przede wszystkim czas oraz zerowy koszt powtórzenia testu.

Decydując się na automatyzacje procesu testowania oprogramowania należy pamiętać, że nadal kluczowym elementem jest konieczność dogłębnego zrozumienia specyfikacji oprogramowania przez osobę odpowiedzialną za pisanie testów. Równie ważnym wymogiem jest to, że pakiet testów powinien być kompletny, to znaczy pokrywać wszystkie kluczowe elementy systemu. Im większy procent kodu pokryty jest testami tym lepiej. Oznacza to również, że każdy nowy kod musi być dostarczony wraz z odpowiednimi testami.

Jeśli spełnimy te warunki proces utrzymania oprogramowania stanie się dużo łatwiejszy, oto niektóre z korzyści:

- Mamy pewność, że system działa zgodnie z założeniami
- Proces modyfikacji oprogramowania staje się łatwiejszy i bezpieczniejszy: jeśli nowy kod spowoduje defekt w którejś z bieżących funkcjonalności zostaniemy o tym niezwłocznie poinformowani przez nie przechodzący test

Na temat automatyzacji procesu testowania oprogramowania napisano niezliczoną ilość książek oraz artykułów. Dobrym punktem startu dla tych, którzy chcieliby zgłębić temat bardziej będzie strona wikipedii[2] poświęcona temu tematowi.

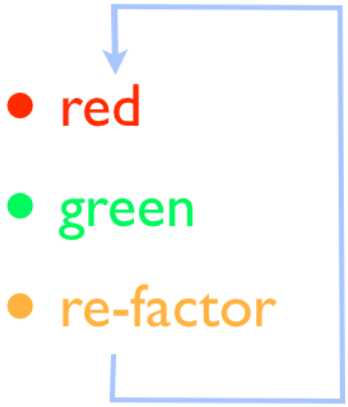
[2]http://en.wikipedia.org/wiki/Test_automation

Rozdział 2

Test Driven Development

2.1 Czym jest TDD?

Test Driven Development jest praktyką, według założeń której każda modyfikacja systemu poprzedzona jest stworzeniem odpowiedniego testu opisującego tą modyfikację. Programista zaczyna od napisania testu, który z naturalnych przyczyn (testowany kod nie istnieje a tym etapie) daje wynik negatywny. Następnie napisany zostaje właściwy kod, którego zachowanie zgodne jest z testowanym. Kiedy testy przechodzą można wprowadzić ewentualne poprawki. Proces rozwoju oprogramowania w zgodzie z filozofią TDD składa się z wielu takich cyklów, które zobrazować można diagramem 2.1



Rysunek 2.1: Red -> Green -> Refactor

Red Pierwszy etap cyklu otrzymał swoją nazwę ze względu na to, że w większości środowisk służących do testowania oprogramowania testy, które zakończyły się niepowodzeniem oznaczane są czerwony kolorem. Etap ten polega na napisaniu testu przed rozpoczęciem implementacji właściwej funkcjonalności oraz na uruchomieniu go. Należy upewnić się, że w tym momencie test zakończy się niepowodzeniem - daje to pewność, że faktycznie testujemy nowe zachowanie, którego w tym momencie system jeszcze nie obsługuje a także że ewentualna przypadkowa modyfikacja tego zachowania zawsze zostanie wykryta przez nieprzechodzący test.

Green Drugi etap polega na zaprogramowaniu zachowania opisanego wcześniejszym testem. Programista piszę tylko tyle kodu, aby spełnić warunki testu po czym uruchamia ponownie cały zestaw testów. Uruchomienie tylko ostatniego testu związanego z napisanym kodem jest nie wystarczające - może okazać się, że nasze ostatnie zmiany modyfikują bezpośrednio lub pośrednio wiele obszarów aplikacji. Jak wskazuje nazwa, etap ten powinien zakończyć się gdy wszystkie do tej pory stworzone testy przechodzą pozytywnie.

Re-factor Ostatni etap polega na jakościowej modyfikacji kody. W tym momencie programista powinien skupić się na usunięciu wszelkich zbędnych powtórzeń, uproszczeniu implementacji czy też dopracowaniu użytego nazewnictwa zmiennych lub metod. Etap ten nie pociąga za sobą żadnych zmian w sposobie działania oprogramowania, jest jednak równie ważny jak poprzednie, dobry jakościowo kod jest łatwiejszy w utrzymaniu i modyfikacji.

Opisane powyżej iterację powinny być jak najprostsze. Oznacza to, że każdą implementowaną funkcjonalność należy podzielić na jak najmniejsze części i wykonywać pełen zestaw powyższych kroków dla każdej z nich. Idealna sytuacja to taka, w której pojedynczy test sprawdza tylko jedną rzecz.

2.1.1 Główne zasady TDD

Zacznij od testu Test powinien być napisany zanim zacznie się implementacja funkcjonalności. Takie podejście gwarantuje, że będziemy mieli pełen zestaw testów opisujących każdą funkcję systemu. Inną zaletą jest konieczność dokładnego przemyślenia szczegółów implementacji jeszcze przed jej rozpoczęciem.

Zaraz po napisaniu nowe testy powinny dawać negatywny wynik Daje to pewność, że testy faktycznie spełniają swoją funkcję, oraz każda degradacja funkcjonalności będzie sygnalizowana nieprzechodzącym testem.

Dużo więcej na temat historii, filozofii oraz zasad Test Driven Development znaleźć można w książce *The RSpec Book*[1]

2.1.2 Przykładowe iteracja TDD

Przypuśćmy, że pracujemy nad oprogramowaniem sportowej tablicy wyników. Naszym aktualnym zadaniem jest napisanie metody, która na wejściu otrzymuje nazwy dwóch drużyn sportowych, zwraca zaś łańcuch składający się z nazw tych drużyn połączonych łańcuchem *vs* . Oprogramowanie napisane jest w języku Ruby, do testowania użyjemy biblioteki RSpec.

Praca rozpoczyna się od napisania testu opisującego pożądane zachowanie. W naszym wypadku może on wyglądać tak:

```
require 'sports_table'

describe SportsTable do
  it "should properly join team names with 'vs.'" ↩
     do
    table = SportsTable.new
    table.header('Chicago Bulls', 'Los Angeles ↩
       Lakers').should == 'Chicago Bulls vs. Los ↩
       Angeles Lakers'
  end
end
```

Dokładny opis budowy testu zawarty jest w podrozdziale *Narzędzia wspierające TDD dostępne dla języka Ruby* i nie będę go tutaj powielał. Chciałbym jednak zwrócić uwagę na fakt, że już na etapie pisania testu programista zmuszony jest przemyśleć szczegóły implementacji. Zaprezentowany przykład jest bardzo prosty, ale na pierwszy rzut oka widać, że oprócz sprawdzenia poprawności zwracanego wyniku test definiuje także pewne szczegóły architektury programu. Po pierwsze zakładamy, że tablica wyników reprezentowana będzie przez klasę o nazwie `SportsTable`, a żądana funkcjonalność zostanie zaimplementowana jako metoda instancyjna `header` a więc aby mieć z niej pożytek użytkownik musi skorzystać z istniejącego obiektu tej klasy. Widać tutaj wyraźnie jedną z głównych zalet testowo zorientowanych metodyk rozwoju oprogramowania - konieczność dokładnego przemyślenia szczegółów implementacji przed jej rozpoczęciem.

[1] [6]

Sednem tego konkretnego testu jest jednak upewnienie się, że dla przykładowych danych wejściowych otrzymamy poprawny wynik. W tym wypadku sprawdzamy, czy wywołanie metody

```
header('Chicago Bulls', 'Los Angeles Lakers↩
    ')
```

na obiekcie klasy `SportsTable` zwróci łańcuch znaków:

```
Chicago Bulls vs. Los Angeles Lakers
```

Po napisaniu uruchamiamy nasz zestaw testów, co kończy się porażką. Zakładając, że rozpoczęliśmy pracę z istniejącą, pustą definicją klasy `SportsTable` zdefiniowaną w pliku `sports_table.rb` wynik powinien wyglądać jak na ilustracji 2.2

```
MacBook-Marcin-Balinski:examples marcinbalinski$ rspec -c spec/
F

Failures:

  1) SportsTable should properly join team names with 'vs.'
     Failure/Error: table.header('Chicago Bulls', 'Los Angeles Lakers').should == 'Chicago Bulls vs. Los Angeles Lakers'
     NoMethodError:
       undefined method `header' for #<SportsTable:0x101e8c3d8>
     # ./spec/sports_table_spec.rb:6

Finished in 0.0004 seconds
1 example, 1 failure
MacBook-Marcin-Balinski:examples marcinbalinski$
```

Rysunek 2.2: Nowo napisany test nie przechodzi.

Tym samym zakończyliśmy pierwszy etap: opisaliśmy wymagane zachowanie testem oraz upewniliśmy się, że test nie przechodzi. Następnym krokiem jest napisanie pierwszej wersji metody `header`:

```
class SportsTable
  def header(team1, team2)
    return team1 + " vs. " + team2
  end
end
```

Ponowne uruchomienie zestawu testów kończy się sukcesem, ilustruje to rysunek 2.3

```
MacBook-Marcin-Balinski:examples marcinbalinski$ rspec -c spec/
.

Finished in 0.00039 seconds
1 example, 0 failures
MacBook-Marcin-Balinski:examples marcinbalinski$
```

Rysunek 2.3: Po dostarczeniu wymaganej funkcjonalności test przechodzi.

Nasza metoda spełnia wszystkie założenia opisane przez testy, wciąż jednak jest pole do poprawy jakości kodu. W języku Ruby każda metoda domyślnie zwraca ostatnią zdefiniowaną w swoim ciele wartość, możemy więc zrezygnować ze zbędnego słowa kluczowego `return`. Oprócz tego zmienimy sposób konstrukcji wynikowego łańcucha: zrezygnujemy z operatora + na rzecz metody `join` obiektu klasy `Array`. Po modyfikacji metoda `header` wygląda następująco:

```
class SportsTable
  def header(team1, team2)
    [team1, team2].join(' vs. ')
  end
end
```

Powtórne uruchomienie zestawu testów kończy się sukcesem, a pierwsza iteracja TDD jest zakończona. Zdołaliśmy opisać nową funkcjonalność oraz poprawnie ją zaimplementować. W tym miejscu należy dodać, że w trzecim kroku, po modyfikacji i ulepszeniu kodu nie zmodyfikowaliśmy zestawu testów. W tym konkretnym przypadku najważniejszy dla nas jest wynik działania metody `header`, nie zaś szczegóły implementacji. Nie testujemy np. tego, że konstrukcja wynikowego łańcucha znaków odbywa się z użyciem metody `join`. Czasami jednak szczegóły implementacji są równie ważne jak zwracane wynik i wtedy należy napisać odpowiednie testy.

2.1.3 Zalety TDD

Test Driven Development wymusza na programiście konkretną dyscyplinę pracy. Proces rozwoju oprogramowania jest iteracyjny i bardzo uporządkowany a także wymaga uprzedniego zaplanowania każdej zmiany lub dodatku do istniejącej bazy kodu. Każda iteracja może zostać zakończona jedynie,

gdy wszystkie testy zakończą się sukcesem. Taki sposób pracy niesie ze sobą wiele zalet, między innymi:

- Pewność, że oprogramowanie zawsze działa zgodnie z założeniami
- Wzrost produktywności
- Wzrost jakości kodu
- Minimalizacja liczby defektów
- Możliwość wczesnego wykrycia defektów
- Modularyzacja kodu jako pozytywny skutek uboczny

2.2 Narzędzia wspierające TDD dostępne dla języka Ruby

2.2.1 Test::Unit[2]

Test::Unit należy do bibliotek dołączanych standardowo do każdej dystrybucji języka Ruby. Oprogramowanie testujemy tutaj przy pomocy tak zwanych asercji. Najprostsza asercja ma postać wywołania metody `assert`, która sygnalizuje niepowodzenie testu w momencie, kiedy wyrażenie przekazane jako jej parametr jest fałszem, przykładowo:

```
def test_one_plus_one_equals_two
  assert 1 + 1 == 3
end
```

Konstrukcja zestawu testów

Zestaw testów biblioteki Test::Unit ma postać definicji klasy, która dziedziczy po klasie Test::Unit::TestCase. Pojedyncze testy definiujemy jako metody, rozpoczynające się od ciągu znaków *test*. W ciele każdej z takich metod możemy zdefiniować wiele asercji (aczkolwiek idealnie jest, jeśli jeden test równoznaczny jest z jedną asercją). Test kończy się sukcesem jedynie, kiedy wszystkie należące do niego asercję również zakończą się sukcesem.

Przykładowy prosty zestaw testów opisujących zachowanie aplikacji będącej kalkulatorem może wyglądać tak:

[2] `http://test-unit.rubyforge.org/test-unit/`

```
require 'test/unit'

class CalculatorTest < Test::Unit::TestCase
  def test_adding
    calc = Calculator.new
    assert calc.add(2, 2) == 4
  end

  def test_substracting
    calc = Calculator.new
    assert calc.sub(2, 2) == 0
  end

  def test_multiplying
    calc = Calculator.new
    assert calc.multiply(2, 4) == 8
  end

  def test_dividing
    calc = Calculator.new
    assert calc.div(2, 2) == 1
  end
end
```

Oprócz metody `assert` biblioteka oferuje bardziej wyspecjalizowane typy asercji:

assert_equal(expected, actual) przyjmuje dwa parametry, zwraca prawdę, jeśli parametry są sobie równe.

assert_not_equal(expected, astual) przyjmuje dwa parametry, zwraca prawdę, jeśli parametry różne od siebie.

assert_match(regex, string) przyjmuje dwa parametry w postaci łańcucha znaków lub wyrażenia regularnego, zwraca prawdę jeśli nastąpi dopasowanie wzorca.

assert_no_match(regex, string) przyjmuje dwa parametry w postaci łańcucha znaków lub wyrażenia regularnego, zwraca prawdę jeśli dopasowanie wzorca nie nastąpi.

assert_nil(object) zwraca prawdę jeśli przekazany parametry ma pustą wartość (`nil`).

assert_not_nil(object) zwraca prawdę, jeśli przekazany parametr ma niepustą wartość (różną od `nil`).

assert_instance_of(class, object) przyjmuje na wejściu nazwę klasy oraz parametr, zwraca prawdę, jeśli parametr jest obiektem typu `class`.

Warunki początkowe i końcowe

Czasem grupa testów powinna być uruchamiana przy takich samych warunkach początkowych, albo też (nie tak częsty przypadek) występuje konieczność wykonania jakichś czynności na koniec każdego testu (np. wyczyszczenie bazy danych). Biblioteka Test::Unit pozwala definiować warunki początkowe i końcowe przy pomocy metod `setup` i `teardown`. Metoda `setup` zostanie wykonana przed każdym testem, metoda `teardown` bezpośrednio po każdym teście. Możemy np. znacząco uprościć nasz przykładowy zestaw testów kalkulatora poprzez przeniesienie procesu tworzenia obiektu Calculator do metody setup. zmodyfikowane testy używają również bardziej właściwych asercji:

```
require 'test/unit'

class CalculatorTest < Test::Unit::TestCase
  def setup
    @calc = Calculator.new
  end

  def test_adding
    assert_equal 4, @calc.add(2, 2)
  end

  def test_substracting
    assert_equal 0, @calc.sub(2, 2)
  end

  def test_multiplying
    assert_equal 8, @calc.multiply(2, 4)
  end

  def test_dividing
    assert_equal 1, @calc.div(2, 2)
  end
end
```

Przedrostek `@` przy zmiennej `calc` oznacza, że odnosimy się do zmiennej instancyjnej a więc dzielonej w obrębie obiektu danej klasy, zmienne bez tego przedrostka traktowane są w Ruby jako zmienne lokalne.

Pakiety testów

Testując dużą aplikację będziemy chcieli podzielić testy na kilka plików tak, aby odzwierciedlić modułową budowę aplikacji oraz poprawić czytelność testów. Biblioteka Test::Unit upraszcza proces uruchamiania całego pakietu testów. Jedyne co musimy w tym wypadku zrobić to stworzyć nowy plik i przy pomocy metody `require` załadować wszystkie pliki zawierające interesujące nas testy:

```
# test/full_suite.rb

require 'test/unit'

require 'test_suite1'
require 'test_suite2'
require 'test_suite3'
```

Teraz, jeśli wykonamy polecenie:

```
ruby test/full_suite.rb
```

uruchomione zostaną wszystkie testy zdefiniowane w plikach:

```
test/test_suite1.rb
test/test_suite2.rb
test/test_suite3.rb
```

2.2.2 RSpec[3]

Kolejnym narzędziem świetnie wspierającym testowanie oprogramowania w języku Ruby jest biblioteka RSpec. RSpec jest narzędziem bardziej rozbudowanym niż Test::Unit, w tym podrozdziale skupię się jednak na jego podstawowych funkcjach umożliwiających testowanie oprogramowania w zgodzie z filozofią TDD.

Bibliotekę RSpec najłatwiej zainstalować przy pomocy narzędzia ruby gems, które jest standardowo dostarczane wraz z każdą dystrybucją języka Ruby. Proces instalacji jest trywialny, wystarczy z konsoli systemowej wydać polecenie:

[3] `http://rspec.info/`

```
gem install rspec
```

Przykłady

W RSpec każdy test nazywany jest przykładem (ang. example). RSpec definiuje własny dialekt języka Ruby, dzięki czemu przykłady wyglądają bardziej jak naturalny język. W początkowej części tego rozdziału zdefiniowaliśmy już przykłady dla oprogramowania obsługującego sportową tablicę wyników:

```
require 'sports_table'

describe SportsTable do
  it "should properly join team names with 'vs.'" ↩
     do
    table = SportsTable.new
    table.header('Chicago Bulls', 'Los Angeles ↩
       Lakers').should == 'Chicago Bulls vs. Los ↩
       Angeles Lakers'
  end
end
```

Pierwsze co powyższy kod robi, to dołącza plik z klasą `SportsTable`, którą mamy zamiar przetestować. Zaraz później znajduje się linia:

```
describe SportsTable do
```

Metoda `describe` zwraca tak zwaną grupę przykładów, która w bibliotece RSpec implementowana jest przez klasę `ExampleGroup`. Jako parametry przyjmuje ona testowaną klasę oraz opcjonalny opis. Metoda `describe` przyjmuje też blok kodu, który programista definiuje pomiędzy słowami kluczowymi `do ... end`.

Konkretne przykłady zdefiniowane są w bloku przekazanym do metody `describe`. Przykład definiuje metoda `it`, której parametrem jest łańcuch znaku będący opisem a sam przykład zdefiniowany w przekazywanym do tej metody bloku.

Oczekiwania

Oczekiwanie jest odpowiednikiem asercji znanych z Test::Unit, w bibliotece RSpec mechanizm oczekiwań zaimplementowany jest w module Spec::Expectations w taki sposób, że do każdego obiektu dynamicznie dodawane są metody

`should` oraz `should_not`. Każda z tych metod jako parametr przyjmuje kolejne wyrażenie dopasowujące (ang. Matcher) wyrażenia takie można definiować samemu, albo pozwolić aby RSpec zdefiniował je dynamicznie. Przykładowe oczekiwania, które dynamicznie rozumie RSpec:

```
[].should be_empty  # prawda, [] jest pusta ↩
   tablica

["kot", "pies"].should include("malpa") # falsz, ↩
   testowana tablica nie zawiera elementu "malpa"

[].should be_instance_of(Array) # prawda
```

W języku Ruby metody, które kończą się znakiem zapytania w zgodzie z konwencją powinny być metodami które zwracają prawdę lub fałsz. Metoda taka testuje po prostu jakieś założenie w stosunku do obiektu danej klasy. Przykładowo metoda `empty?` obiektu klasy Array testuje czy dana tablica jest pusta zwracając wartość `true` jeśli tak jest, lub `false` w przeciwnym wypadku. Automatyczne sprawdzanie oczekiwań w bibliotece RSpec korzysta właśnie z tej konwencji w taki sposób, że jeśli nie uda się znaleźć zdefiniowanego wyrażenia dopasowującego o danej nazwie RSpec szuka metody obiektu o tej samej nazwie zakończonej dodatkowo znakiem zapytania. W tym procesie pomijane są pewne początkowe słowa kluczowe takie jak `be_`, `a_` czy `an_` co pozwala to na konstruowanie przykładów tak, aby bardziej przypominały naturalny język. W powyższych przykładach do sprawdzenia prawdziwości testu kolejno wykorzystywane są następujące metody obiektu klasy Array: `empty?`, `include?`, `instance_of?`

Imitacje obiektów

W trakcie testowania oprogramowania zdarzają się sytuacje, kiedy chcemy uniknąć korzystania z prawdziwych instancji jakiejś klasy. Najczęściej jest tak w przypadku, kiedy dany obiekt nie jest bezpośrednio przedmiotem testu, ale inna część oprogramowania polega na nim, lub jeśli wykonuje on kosztowne operacje, które spowalniają testy.

RSpec pozwala w takim przypadku na definiowanie tak zwanych imitacji obiektów. Imitacja (ang. Mock) to w skrócie obiekt-atrapa, programista deklaruje na jakie komunikaty ma odpowiadać. Wyobraźmy sobie, że pewien moduł naszego systemu wykonuje bardzo czasochłonne operacje matematyczne, na wynikach tych operacji polega drugi moduł, który chcemy przetestować. W takiej sytuacji moglibyśmy oczywiście skonstruować test tak, że faktycznie inicjowalibyśmy cały proces obliczeniowy modułu matematycz-

nego, a jego wynik przekazywali do testowanego modułu jednak taka metoda szybko sprawiłaby, że nasze testy byłyby bardzo powolne.

Jeśli moduł matematyczny sam w sobie jest dobrze przetestowany, to nic nie stoi na przeszkodzie, aby przy testowaniu zależnej od niego części systemu użyć już tylko jego imitacji. W Rspec może to wyglądać w taki sposób:

```
describe CoolPartUsingMathsModule do
  it "should use result of math module computation↩
      when run method triggered" do
    maths = mock(MathsModule, :compute => 1337)
    obj = CoolPartUsingMathsModule.new(maths)
    obj.run
  end
end
```

Zakładamy tutaj, że metoda `run` obiektu klasy `CoolPartUsingMathsModule` korzysta z wyniku jaki zwraca metoda `compute` obiektu typu `MathsModule`. Metoda `mock` zwraca imitacje obiektu. Klasę obiektu definiujemy jako pierwszy argument, następnie występuje opcjonalna lista komunikatów wraz ze zwracaną wartością, na te komunikaty atrapa będzie odpowiadać.

Test ten można skonstruować jeszcze lepiej. W aktualnej formie nie stawiamy w nim bowiem żadnych warunków jakie musi spełniać testowana metoda `run`. Zdefiniowaliśmy wprawdzie imitację obiektu matematycznego, która odpowiada na wywołanie metody `compute` zwracając wartość `1337`, nie sprawdzamy jednak, czy metoda ta kiedykolwiek zostaje wykonana. Chcąc upewnić się, że tak faktycznie jest możemy napisać powyższy test tak:

```
describe CoolPartUsingMathsModule do
  it "should use result of math module computation↩
      when run method triggered" do
    maths = mock(MathsModule)
    obj = CoolPartUsingMathsModule.new(maths)
    maths.should_receive(:compute).and_return↩
       (1337)
    obj.run
  end
end
```

Wywołując na obiekcie metodę `should_receive` definiujemy nowe oczekiwanie: teraz test zakończy się sukcesem tylko wtedy, kiedy nastąpi dokładnie jedno wywołanie metody `compute`. W taki sposób przetestowaliśmy integrację pomiędzy naszymi modułami i jednocześnie sprawiliśmy, że test jest bardzo szybki, nie wykonuje on bowiem żadnych obliczeń modułu mate-

matycznego, a zamiast tego korzysta z bardzo prostej atrapy.

Rozdział 3

Behavior Driven Development

3.1 Czym jest BDD?

Behavior Driven Development jest metodyką rozwoju oprogramowania, w której główny nacisk położony jest na zacieśnienie współpracy między programistami i nie technicznymi uczestnikami projektu. Podobnie jak w przypadku TDD implementację konkretnej funkcjonalności poprzedza zdefiniowanie jej zachowania w teście. Różnica polega na tym, że scenariusze BDD pisane są w naturalnym języku, tak aby były zrozumiałe nie tylko dla programistów. Idealną sytuacją jest kiedy scenariusze takie powstają w wyniku ścisłej współpracy programistów oraz właścicieli projektu. BDD jak sama nazwa wskazuje kładzie nacisk na zdefiniowanie i zrozumienie zachowania aplikacji a szczegóły implementacji są tutaj mniej istotne. W tym rozdziale przybliżę najważniejsze cechy oraz przykładowe narzędzia, jakie można wykorzystać w procesie *Behavior Driven Development*. W celu dokładniejszego zgłębienia tematu jeszcze raz polecam książkę *The RSpec Book*[1].

3.1.1 BDD jako narzędzie dokumentacji. Funkcjonalności i scenariusze

Każdy program tak naprawdę składa się z zestawu funkcjonalności. Każda funkcjonalność wnosi konkretną wartość dodaną z punktu widzenia grupy docelowej, dla której oprogramowanie powstaje. Przykładowo program do obsługi księgowości powinien pozwalać zarządzać rachunkiem zysków i strat oraz sporządzać bilans (to oczywiście tylko niektóre z funkcji). Każda z wyżej wymienionych czynności to pojedyncza funkcjonalność, która w zgodzie z filozofią BDD powinna zostać opisana zestawem scenariuszy.

[1] [6]

Scenariusze opisują zachowanie się programu w kontekście konkretnej funkcjonalności w precyzyjnie zdefiniowanej sytuacji a dla każdej funkcjonalności możemy zdefiniować dowolną ilość scenariuszy użycia. Na przykład dla funkcjonalności Zarządzanie rachunkiem zysków i strat mogą być zdefiniowane następujące scenariusze:

- Wprowadzenie nowej poprawnej pozycji
- Wprowadzenie nowej nie poprawnej pozycji
- Modyfikacja pozycji
- Próba wprowadzenia nowej pozycji przez nieautoryzowanego użytkownika

Narzędzia BDD pozwalają na bardzo dużą swobodę jeśli chodzi o język definiowania scenariuszy, przykładowy plik definiujący powyższą funkcjonalność wraz ze scenariuszem *Wprowadzenie nowej nie poprawnej pozycji* mógłby wyglądać tak:

```
Funkcjonalność: Zarządzanie rachunkiem zysków i ↩
   strat

Scenariusz: Wprowadzenie nowej nie poprawnej ↩
   pozycji
  Jako zalogowany użytkownik systemu
  Kiedy wybieram z menu głównego pozycję "Wprowadź↩
      nową pozycję"
  Oraz wprowadzam do pola "Przychody netto ze ↩
     sprzedaży produktów" wartość "↩
     to_nie_jest_liczba"
  Wtedy powinienem zobaczyć wiadomość "Błąd: ↩
     Niepoprawna wartość. To pole jest polem ↩
     liczbowym."
  Oraz pole "Przychody netto ze sprzedaży produktó↩
     w" powinno być puste
```

Powyższy przykład opisuje zachowanie aplikacji w sposób na tyle naturalny, że nikt nie będzie miał trudności w jego zrozumieniu, a po poznaniu kilku prostych zasad językowych, którymi należy się kierować również nie techniczny uczestnicy projektu mogą opisywać nowe funkcjonalności w ten sposób.

Tworzenie specyfikacji projektu w postaci scenariuszy BDD jest bardzo pożądane. Zmniejsza to ilość niepotrzebnej dokumentacji, powoduje zacieśnienie współpracy w zespole oraz, o czym więcej w następnym podrozdziale, zwiększa pokrycie kodu testami.

3.1.2 BDD jako narzędzie testowania

Prawdziwą mocą bibliotek wspierających Behavior Driven Development jest to, że traktują one specyfikacje dostarczoną w postaci scenariuszy jako zestaw testów. Dołączając więc scenariusze do naszego zestawu testów automatycznych mamy pewność, że oprogramowanie zachowuje się zgodnie z opisanymi w nich założeniami. W tym podrozdziale chciałbym opisać w jaki sposób, z technicznego punktu widzenia przebiega proces translacji specyfikacji (będącej plikiem tekstowym składającym się z pojedynczych scenariuszy) na zestaw automatycznych testów.

Definicje kroków na przykładzie biblioteki Cucumber[2]

Aby skutecznie uruchomić scenariusze w formie testów należy skonstruować plik tekstowy je zawierający w zgodzie z pewnymi zasadami. Przykładowy plik definiujący funkcjonalność *Zarządzanie rachunkiem zysków i strat* rządzi się pewnymi prawami:

- Nazwa opisywanej funkcjonalności zdefiniowana jest po słowie kluczowym *Funkcjonalność:*
- Pojedyncze scenariusze definiowane są po słowie kluczowym *Scenariusz:*. Należą do ostatnio zdefiniowanej funkcjonalności.
- Każda kolejna linia nie będąca definicją nowego scenariusza lub funkcjonalności traktowana jest jako pojedynczy krok ostatnio zdefiniowanego scenariusza.

Sercem biblioteki Cucumber są definicje kroków, które pozwalają powiązać każdy z nich z konkretną akcją. Definicja kroku składa się z wzorca językowego kroku oraz kodu który ma zostać wykonany jeśli wzorzec pasuje do aktualnie przetwarzanego kroku.

Wzorzec językowy najczęściej przybiera postać wyrażenia regularnego. Cucumber przetwarzając nowy krok iteruje po wzorcach, które zostały zdefiniowane i wykonuje kod powiązany z pierwszym, do którego pasuje nazwa kroku (dlatego ważne jest definiowanie wzorców tak, aby były unikalne).

[2] `http://cukes.info/`

Przykładowa definicja dla kroku *Kiedy wybieram z menu głównego pozycję "Wprowadź nową pozycję"* może wyglądać tak:

```
Kiedy /wybieram z menu głównego pozycję "(.*)"/ do
  |pozycja|


  # zmienna 'pozycja' zawiera teraz ciąg znaków,
  # który dopasowany został do wyrażenia "(.*)" a ↩
    więc
  # dla kroku 'Kiedy wybieram z menu głównego ↩
    pozycję
  # "Wprowadź nową pozycję"' będzie miała wartość
  # "Wprowadź nową pozycję"
end
```

Kiedy cucumber spróbuje przetworzyć krok *Kiedy wybieram z menu głównego pozycję "Wprowadź nową pozycję"* dopasuje go do pierwszego odpowiadającego wzorca, który zdefiniowaliśmy oraz uruchomi blok kodu zdefiniowany pomiędzy słowami kluczowymi `do` oraz `end`. Przekaże do tego bloku również wszelkie zmienne zdefiniowane we wzorcu. Powyższy przykład jest na razie bezużyteczny ponieważ jedyne co znajduje się w bloku kodu to komentarz.

Krok scenariusza najczęściej definiuje jedną z trzech rzeczy: założenie co do stanu środowiska w jakim uruchomione jest oprogramowanie, konkretną akcję wykonywaną na oprogramowaniu (taką jak np. kliknięcie przycisku) lub rezultat, jakiego się spodziewamy. Cucumber sam w sobie jest narzędziem które dopasowuje krok do odpowiadającej mu definicji, wszelkie interakcje z działającym programem, modyfikowanie środowiska działania czy też testowanie otrzymywanych wyników muszą zostać wykonywane przez zewnętrzne biblioteki, które należy samodzielnie skonfigurować.

Aby lepiej zobrazować w jaki sposób może wyglądać działające środowisko BDD poczynię kilka założeń co do aplikacji, którą testujemy, oraz bibliotek, których użyjemy. Pominę szczegóły konfiguracji, jest to treścią jednego z kolejnych rozdziałów tej pracy.

- Interfejs naszej aplikacji zdefiniowany jest językiem opisu dokumentów HTML.
- Aplikacja napisana jest w języku Ruby, przy pomocy frameworka Ruby on Rails
- Narzędzie RSpec zostało zainstalowane i skonfigurowane

- Narzędzie Capybara zostało zainstalowane i skonfigurowane

O bibliotece RSpec pisałem już w poprzednim rozdziale, tutaj użyjemy go w podobnym celu, to jest do testowania wyników działania aplikacji. Biblioteka Capybara służy do symulacji interakcji użytkownika z aplikacją używającą jako interfejsu HTML. Używając powyższych narzędzi mogę opisać pełen zestaw definicji kroków dla scenariusza *Wprowadzenie nowej nie poprawnej pozycji*:

```
Jako /zalogowany użytkownik systemu/ do
  # znajdź pierwszego zarejestrowanego użytkownika
  user = User.first

  # otwórz stronę logowania
  visit "/login"

  # wypełnik pola 'email' i 'hasło' poprawnymi ↩
     danymi
  fill_in("email", :with => user.email)
  fill_in("hasło", :with => user.password)
  click_button("Zaloguj")
end

Kiedy /wybieram z menu głównego pozycję "(.*)"/ do
  |pozycja|

  select(pozycja, :from => "menu główne")
end

Oraz /wprowadzam do pola "(.*)" wartość "(.*)"/ do
  |pole, wartosc|

  fill_in(pole, :with => wartosc)
end

Wtedy /powinienem zobaczyć wiadomość "(.*)"/ do
  |wiadomosc|

  response.should contain(wiadomosc)
end
```

```
Oraz /pole "Przychody netto ze sprzedaży produktów↩
   " powinno być puste/ do |pole|
  field_labeled(pole).value.should be_empty
end
```

Rola programisty

Jak widać definiowanie kroków wymaga podstawowej wiedzy z zakresu programowania oraz budowy opisywanego systemu, dlatego też zadanie to najczęściej wykonują programiści pracujący nad projektem. Klient w tym wypadku dostarcza specyfikacji w postaci scenariuszy użycia, programiści zaś definiują brakujące kroki. Na szczęście problem ten jest uciążliwy tylko na początku życia projektu. Wraz z rosnącą ilością scenariuszy rośnie też ilość definicji kroków, a w pewnym momencie życia projektu definicje kroków zaczynają gęsto pokrywać większość funkcjonalności systemu tak, że w większości wypadków nawet do nowych funkcjonalności można bez problemu dopasować istniejące już kroki.

Ważne jest aby scenariusze były jak najbardziej zestandaryzowane, to znaczy, aby w miarę możliwości korzystać z już zdefiniowanych kroków w procesie ich tworzenia. Aby to osiągnąć osoba odpowiedzialna za dostarczenie specyfikacji w postaci scenariuszy powinna poświęcić trochę czasu na zapoznanie się z dostępnymi definicjami kroków.

Ze strony programistów, którzy piszą definicje szczególny nacisk powinien zostać położony na kilka kwestii:

Unikanie powtórzeń Należy upewnić się, że brakujący krok jest na pewno unikalny. Może okazać się, że wcześniej został zdefiniowany bardzo podobny krok, o innej nazwie.

Parametryzacja Jeśli zachodzi taka potrzeba należy przystosować definicję kroku tak, aby obsługiwała więcej niż jeden przypadek użycia.

Klasyfikacja Aby zwiększyć czytelność kodu należy klasyfikować kroki na przykład według funkcjonalności, które obsługują, można w tym celu umieszczać kroki w osobnych plikach.

Przestrzeganie tych zasad nie jest konieczne do poprawnego działania systemu ale na pewno w znaczącym stopniu poprawi czytelność i jakość naszej bazy testów.

3.1.3 Testy akceptacyjne a BDD

Kiedy nowa funkcjonalność jest skończona klient zawsze powinien ją przetestować i zaakceptować jeśli spełnia jego oczekiwania, lub odrzucić w przeciwnym wypadku. W tradycyjnym podejściu do rozwoju oprogramowania testy akceptacyjne wyglądają najczęściej tak, że klient ręcznie sprawdza nową funkcjonalność. Minusy manualnego testowania zostały szczegółowo opisane w rozdziale dotyczącym TDD. Behavior Driven Development pozwala w dużym stopniu wyeliminować czasochłonne manualne testy akceptacyjne. W momencie, kiedy mamy gotowy dobry zestaw scenariuszy użycia, który jednocześnie jest specyfikacją funkcjonalności sam fakt, że wszystkie scenariusze zostają uruchomione z pozytywnym skutkiem jest wystarczającym testem akceptacyjnym.

Jeśli chcemy zastąpić testy akceptacyjne z żywym klientem na scenariusze BDD należy pamiętać, że środowisko testowe (w tym definicje kroków) muszą zostać zaprojektowane tak, aby działały w identycznej konfiguracji jak konfiguracja produkcyjna (czyli taka, w jakiej działa nasze oprogramowanie po dostarczeniu do klienta). Szczególnie nie należy używać imitacji (ang. mock) obiektów, których bardzo często używa się w testach jednostkowych, scenariusze powinny testować zachowanie się aplikacji w naturalnym środowisku.

3.2 Narzędzia BDD dostępne dla języka Ruby[3]

3.2.1 Cucumber

W środowisku programistów języka Ruby najpopularniejszym narzędziem BDD jest bez wątpienia biblioteka Cucumber. Cucumber powstał w 2008 roku jako następca biblioteki Story Runner będącej częścią narzędzia RSpec. Biblioteka miała umożliwić programistom opisywanie naturalnym językiem sposobu w jaki powinno zachowywać się oprogramowanie. Scenariusze służą jednocześnie jako dokumentacja oraz zestaw automatycznych testów.

Niewątpliwą zaletą biblioteki Cucumber jest to, że możne ona współpracować również z innymi niż Ruby platformami programistycznymi takimi jak Java, .NET, Adobe Flex czy Python. Więcej informacji na ten temat znaleźć można pod adresem: http://github.com/aslakhellesoy/cucumber/wiki.

Instalacja jest równie prosta jak w przypadku omawianej wcześniej biblioteki RSpec:

```
gem install cucumber
```

[3] `http://cukes.info/`

Jako, że techniki prezentowane w tym rozdziale opisane i zaprezentowane zostały na przykładzie biblioteki Cucumber powtórny szczegółowy opis tego narzędzia jest w tym miejscu zbędny. Szczegółowa dokumentacja dostępna jest pod adresem: http://cukes.info

3.2.2 Steak[4]

Steak podobnie jak Cucumber jest narzędziem, które służy do opisywania zachowania aplikacji w formie automatycznych testów, główny nacisk jest tutaj jednak kładziony na wygodę programisty oraz szybkość powstawania testów. Ograniczenia w stosunku do tego co oferuje Cucumber są następujące:

- Brak wsparcia dla naturalnego języka
- Brak podziału na scenariusze oraz kroki
- Konieczność pisania testów w Ruby

Steak rezygnuje z filozofii angażowania nie technicznych uczestników projektu w cykl testowania oprogramowania wychodząc z założenia, że opcja ta i tak nie jest wykorzystywana w większości projektów informatycznych. Programista sam konstruuje testy w języku Ruby, nie jest to tak czytelne jak w przypadku Cucumbera, jednak ich tworzenie i utrzymanie jest prostsze:

- Definicje kroków są zbędne
- Nie trzeba ustalać i trzymać się konwencji językowych
- Bezpośredni dostęp do wszystkich funkcji języka Ruby

Przykładowa specyfikacja napisana przy pomocy biblioteki Steak wygląda następująco:

```
feature "Main page" do

    background do
      create_user :login => "jdoe"
      login_as "jdoe"
    end

    scenario "should show existing quotes" do
```

[4] http://github.com/cavalle/steak/wiki

```
    create_quote :text => "The language of ↩
       friendship is not words, but meanings",
      :author => "Henry David Thoreau"

    visit "/"

    page.should have_css(".quote", :count => 1)
    within(:css, ".quote") do
      page.should have_css(".text",
        :text => "The language of friendship is ↩
           not words, but meanings")

      page.should have_css(".author",
        :text => "Henry David Thoreau")
    end
  end

end
```

Instalacja jak zwykle w przypadku (ogromnej większości) bibliotek Ruby sprowadza się do

```
gem install steak
```

W przypadku, kiedy nasza aplikacja oparta jest o framework Rails, Steak oferuje dodatkowe udogodnienia, ale wymaga również minimalnie więcej konfiguracji. Wszelkie kroki są jasno opisane pod adresem http://github.com/cavalle/steak

Steak jest dobrym rozwiązaniem jeśli projekt prowadzony jest przez stricte techniczny zespół lub zarząd projektu nie jest przekonany do uczestniczenia w procesie specyfikowania oprogramowania w formie testów. Minusem rezygnacji z takiej formy współpracy jest niestety spadek znaczenia tworzonej w Steaku specyfikacji jako testów akceptacyjnych: jeśli właściciele projektu nie będą rozumieli specyfikacji w formie jaką oferuje to najprawdopodobniej testy akceptacyjne będą chcieli przeprowadzić osobiście.

3.2.3 Capybara[5]

Capybara ułatwia integracyjne testowanie aplikacji działających w przeglądarce www symulując akcję jakie wykonywałby żywy użytkownik. Nie jest to narzędzie związane stricte z BDD, jednak świetnie uzupełnia i wspomaga biblioteki takie jak Cucumber czy Steak.

[5] `http://github.com/jnicklas/capybara`

Chcąc opisać krok po kroku zachowanie aplikacji internetowej, naturalnym jest łączenie akcji, które wykonuje w przeglądarce użytkownik z ich konsekwencjami. Przykładowo w celu zalogowania się należy najpierw wypełnić pole tekstowe *Login*, następnie wpisać hasło do pola *Hasło* oraz ostatecznie kliknąć przycisk *Zaloguj*. Capybara pozwala symulować interakcje użytkownika z naszą aplikacją w przeglądarce i wykonywać na niej takie akcje jak kliknięcie w link, wypełnienie pola czy odświeżenie strony. Informacją zwrotną, która często służy do określenia efektu naszych działań jest obiekt `page`, który przechowuje ostatnią wyświetloną użytkownikowi stronę. Przykładowo kod:

```
feature "signing up" do
  background do
    User.create(:email => 'user@example.com',
      :password => 'caplin')
  end

  scenario "signing in with correct credentials" ↩
     do
    within("#session") do
      fill_in 'Login', :with => 'user@example.com'
      fill_in 'Password', :with => 'caplin'
    end
      click_link 'Sign in'
      assert_true page.has_content?('Successfuly ↩
         logged in!')
    end
  end
end
```

Wypełni wszelkie konieczne do zalogowania się pola, zasymuluje kliknięcie przycisku z etykietą *Sign in* oraz sprawdzi, czy w wyniku tego, na nowo wyświetlonej stronie znajduje się treść *Successfuly logged in!*.

Instalacja podstawowej biblioteki odbywa się poprzez:

```
gem install capybara
```

Tak jak w przypadku Cucumbera, funkcje Capybary wykorzystywane były w tym rozdziale w celu zobrazowania założeń BDD. Dokładna dokumentacja bardzo bogatych możliwości biblioteki wykracza poza ramy tej pracy, można ją jednak odnaleźć wraz ze szczegółami konfiguracji dla różnych środowisk na stronie http://github.com/jnicklas/capybara.

Rozdział 4

Studium przypadku: Dynamicznie generowany panel administracyjny

4.1 Wstęp

Świat technologii informatycznych zmienia się bardzo szybko. Ogromnemu skokowi mocy obliczeniowej sprzętu komputerowego towarzyszył w ostatnich kilku latach znaczący spadek cen związanych z jego wykorzystaniem. Stało się to katalizatorem rozwoju nowych trendów między innymi w dziedzinie wytwarzania oprogramowania. Gwałtowny wzrost zainteresowania dynamicznymi językami programowania takimi jak Ruby, Python czy JavaScript oraz pojawienie się nowych metodyk rozwoju produktów informatycznych, które coraz częściej ujmują ten proces bardziej z filozoficznego aniżeli technicznego lub biznesowego punktu widzenia jest bez wątpienia jednym z owoców tego procesu.

Spadek kosztów mocy obliczeniowej skutecznie rozwiązał problem wyboru technologii realizacji projektu informatycznego: wydajność narzędzi, których użyjemy do realizacji celu stała się w większości przypadków pomijalnym lub przynajmniej drugorzędnym problemem. Dziś najważniejszym kryterium wyboru jest stopień dopasowania możliwości oraz charakterystyki rozważanej technologii do potrzeb zespołu odpowiedzialnego za rozwój projektu. Oczywiście istnieją również skrajne przypadki, w których to oprogramowanie, z różnych przyczyn, musi zostać napisane w konkretnej technologii. Te smutne przypadki stanowią jednak kroplę w morzu wykraczającą daleko poza ramy niniejszej pracy.

W czasach, kiedy o wiele bardziej opłaca się dokupić nowy serwer niż po-

krywać koszty optymalizacji ogromną furorę robi termin *Przedwczesna optymalizacja*. Te dwa słowa mają dzisiaj znaczenie negatywne, które jest jednak jak najbardziej uzasadnione z ekonomicznego oraz użytkowego punktu widzenia. Dopóki niedostatki w wydajności oprogramowania można skompensować inwestycją w nowe zasoby sprzętowe zespół programistów powinien skupiać wszystkie swoje wysiłki na rozwój funkcjonalności. Optymalizacja kodu następuje dopiero w momencie, kiedy koszty inwestycji w sprzęt przewyższają koszt związane z optymalizacją albo w momencie kiedy oprogramowanie ze względu na swoją nie optymalność przestaje się skalować na nowe zasoby sprzętowe.

Na pierwszy rzut oka może wydawać się, że ten wstęp niewiele ma wspólnego z tematem pracy. Należy jednak uświadomić sobie, że to właśnie opisane powyżej zmiany w sposobie myślenia o metodach prowadzenia projektów IT stoją u podstaw rozwoju nowoczesnych narzędzi takich jak wymienione wcześniej dynamiczne języki, wysokopoziomowe frameworki programistyczne na nich oparte czy metodologie pokroju Behaviour Driven Development. Środki, które służą osiągnięciu założonego celu są jedną z najważniejszych zmiennych od których zależy sukces projektu, w przeszłości istniało wiele barier ograniczających ich wybór, dziś większość z nich została usunięta.

4.2 Założenia projektu

Celem niniejszego rozdziału jest pokazanie czytelnikowi w jaki sposób rozwija się konkretny projekt prowadzony w zgodzie z metodologią BDD oraz jakie płyną z tego korzyści. Wybrany temat projektu to dynamiczny panel administracyjny dla aplikacji internetowych opartych na bibliotece *Ruby on Rails*[1], jego główne założenia to:

Uniwersalność Panel powinien współpracować z większością aplikacji napisanych w Ruby on Rails. W praktyce oznacza to, że modele biznesowe powinny być klasami pochodnymi klasy `ActiveRecord` a sposób budowy aplikacji jest zgodny z konwencjami przyjętymi dla aplikacji Ruby on Rails.

CRUD W teorii baz danych istnieją cztery podstawowe operacje jakie możemy wykonać na zasobie: Tworzenie, odczytanie, aktualizacja, usunięcie (ang. Create, read, update and delete). Biblioteka ma pozwalać na zarządzanie modelami biznesowymi aplikacji przy użyciu jedynie tych czterech standardowych metod.

[1] `http://rubyonrails.org/`

Dynamiczność Po instalacji panel powinien sam wykryć rodzaje zasobów na jakich operuje aplikacja oraz wygenerować odpowiednie widoki i formularze do zarządzania nimi. Zmiany w budowie modeli biznesowych, które wymuszają konieczność zmian w zarządzaniu nimi, jak również pojawienie się nowych modeli również powinno być odzwierciedlone w zachowaniu panelu automatycznie, bez konieczności jakiejkolwiek ingerencji.

Proste wdrożenie Podstawowe wdrożenie rozwiązania wymaga jedynie aby aplikacja kliencka korzystała z biblioteki Ruby on Rails w wersji co najmniej 3.0.3. Po dodaniu biblioteki panelu do listy *gemów*[2] wykorzystywanych przez aplikację możliwe jest natychmiastowe korzystanie.

Wygoda użytkowania Proces zarządzania aplikacją powinien być jak najwygodniejszy. Oznacza to między innymi, że pola formularzy służących do tworzenia lub edycji rekordów powinny być dostosowane do rodzaju danych jakie przechowują a próby wprowadzenia nieprawidłowych wartości powinny być sygnalizowane czytelną informacją o błędzie. Jeśli istnieją powiązania pomiędzy kilkoma modelami biznesowymi, to powinna istnieć bardzo szybka możliwość zarządzania każdym z powiązanych rekordów.

4.2.1 Open Source

Biblioteka jest dostępna za darmo na zasadach licencji MIT.[3] Filozofia rozwoju oprogramowania na zasadach open source jest bardzo bliska środowisku programistów Ruby. Sam język udostępniony jest na licencji GPL, biblioteka Ruby on Rails korzysta z licencji MIT. Użycie licencji MIT oznacza, że każdy otrzymuje prawo do nielimitowanego wykorzystania kopii oprogramowania w dowolny sposób, sprawia to, że jest to najchętniej wykorzystywana przez programistów Ruby licencja.

4.2.2 Sposób prowadzenia projektu

Projekt prowadzony jest według bardzo uproszczonych zasad metodologii SCRUM.[4] Rozwój projektu podzielony jest na tygodniowe sprinty, przed każdym z nich następuje spotkanie zespołu, podczas którego wybierane i

[2] `http://docs.rubygems.org/`
[3] `http://en.wikipedia.org/wiki/MIT_License`
[4] `http://en.wikipedia.org/wiki/Scrum_(development)`

przydzielane są konkretne zadania do wykonania w następnej iteracji. Spotkania te służą również omówieniu bieżących spraw związanych z projektem.

Zespół zaangażowany w projekt składa się z trzech osób: dwóch programistów, oraz osoby dzielącej rolę Scrum Mastera, który odpowiedzialny jest za przygotowanie i prowadzenie spotkań oraz Product Ownera, który reprezentuje oczekiwania końcowego użytkownika dotyczące kwestii funkcjonalności oraz ekonomicznych kwestii związanych z rozwojem projektu.

Duży nacisk kładziony jest na testowanie oprogramowania, testy akceptacyjne istnieją w formie zautomatyzowanych scenariuszy BDD. Każda funkcjonalność lub modyfikacja oprogramowania akceptowana jest jedynie jeśli dostarczona jest wraz z pełnym zestawem testów ją dokumentujących.

4.2.3 Dodatkowe narzędzia

Repozytorium projektu zarządzane jest przez system kontroli wersji GIT[5] a hostowane jest przez serwis GitHub.[6] Źródła projektu dostępne są publicznie pod adresem https://github.com/piotrj/administer.

Jako narzędzie wspomagające proces zarządzania projektem użyta została darmowa wersja Pivotal Tracker[7], który został zaprojektowany aby wspomagać zarządzanie projektem prowadzonym według zasad SCRUM.

4.3 Proces implementacji

Jak zostało to wspomniane wcześniej implementacja podzielona jest na tygodniowe iteracje zwane również sprintami. Każda planowana funkcjonalność zanim zostanie zaimplementowana i stanie się częścią projektu musi zostać krytycznie oceniona pod względem użyteczności oraz opisana w formie krótkiej specyfikacji.

W wypadku kiedy mamy do czynienia z rozbudowaną funkcjonalnością należy podzielić jej implementację na jak najmniejsze, spójne kawałki a każdy z tych fragmentów powinien zostać jasno opisany. Idealnie jest, kiedy fragmenty danej funkcjonalności są od siebie niezależne, to znaczy, jeśli nad każdym z nich można pracować osobno w tym samym czasie. Taka sytuacja niestety zdarza się rzadko. Jeśli dany fragment zależy od innych, należy załączyć taką informację w jego opisie.

Każde nowe zachowanie opisane jest testem w postaci odpowiedniego scenariusza Cucumber. Jest to warunek konieczny, scenariusze powstają przed

[5] `http://git-scm.com`
[6] `http://github.com`
[7] `http://www.pivotaltracker.com`

rozpoczęciem procesu właściwej implementacji, po zaakceptowaniu przez Product Ownera pełnią one rolę testów akceptacyjnych. Testy jednostkowe w postaci specyfikacji RSpec powstają dla części systemu, których szczegóły implementacji są istotne dla prawidłowego jego działania.

4.3.1 Aplikacja testowa

Specyfika projektu dynamicznego panelu administracyjnego utrudnia nieco pisanie scenariuszy jego użycia. Myśląc o panelu administracyjnym nigdy nie myślimy o nim jako o odrębnym bycie, jest raczej nieodłącznie związany z aplikacją, którą administruje. Aby proces testowania i specyfikowania systemu był jak najbardziej naturalny biblioteka *administer* rozwijana jest wraz z małą aplikacją testową.

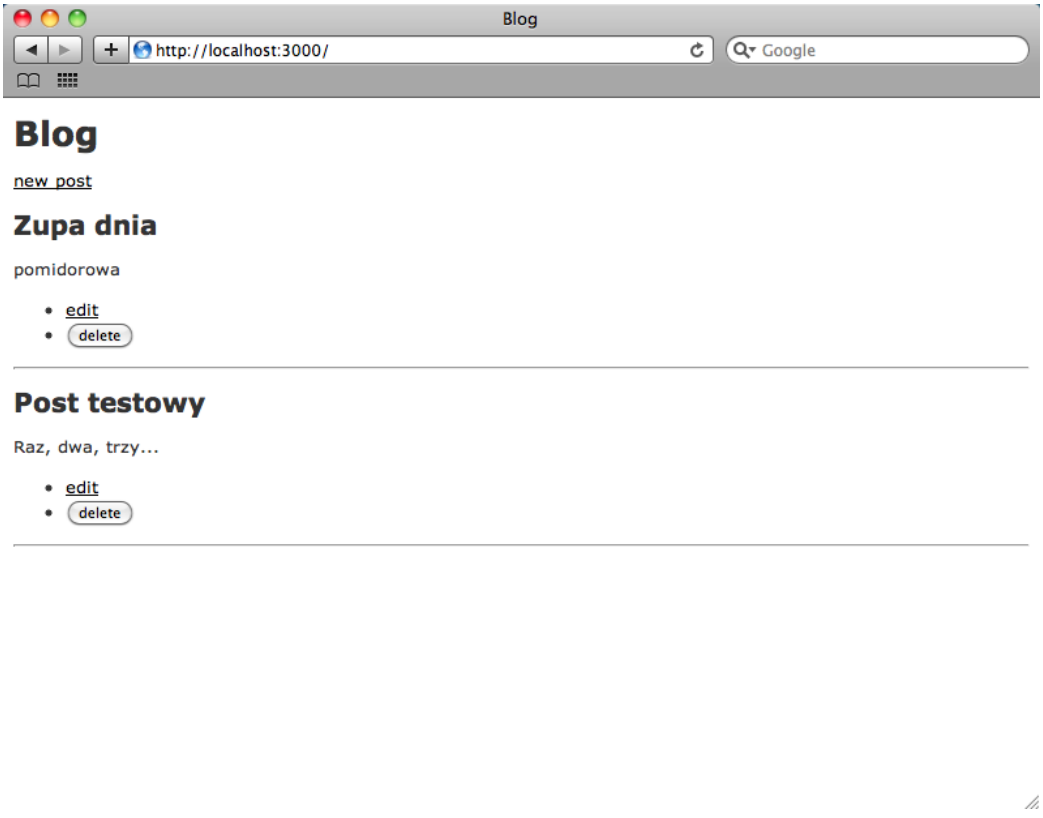
Prosty system blogowy został stworzony, aby opisać i przetestować zachowanie właściwego oprogramowania. Jedynym celem istnienia aplikacji blogowej jest przetestowanie integracji biblioteki *administer* z zewnętrzną aplikacją. Z technicznego punktu widzenia wszystkie testy behawioralne opisują zachowanie aplikacji testowej, która dołącza bibliotekę *administer* do puli wykorzystywanych przez siebie gemów. W przypadku oprogramowania, które działa na zasadzie integracji z zewnętrznym systemem taka konfiguracja środowiska testowego jest najlepsza, odpowiada bowiem faktycznemu scenariuszowi użycia. Sam blog jest niezwykle prostą aplikacją, w założeniach ma pozwalać na:

- Wyświetlanie artykułów
- Tworzenie artykułów
- Edycję artykułów
- Usuwanie artykułów
- Kategoryzację artykułów
- Tworzenie kategorii
- Edycję kategorii
- Usuwanie kategorii

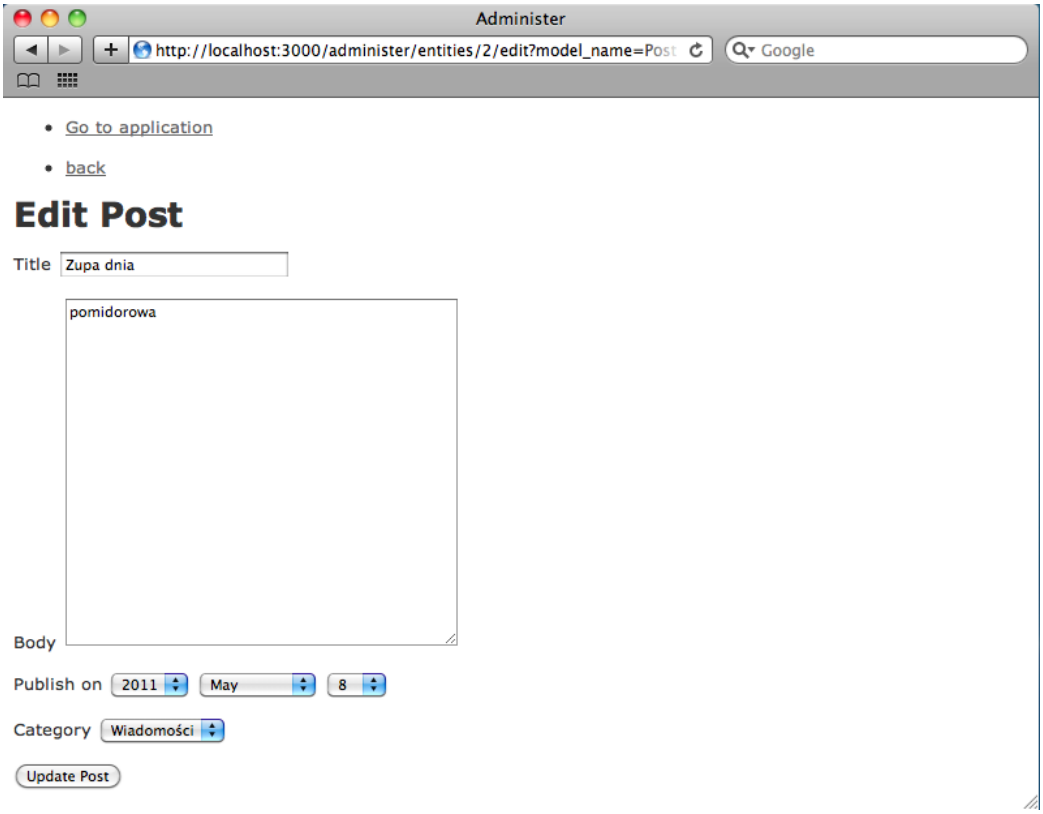
Blog świetnie nadaje się do testowania biblioteki typu *administer* - wszystkie powyższe funkcje, prócz pierwszej, są tak naprawdę funkcjami, które wykonywać ma panel administracyjny. W kontekście tego przykładu cel biblioteki *administer* jest jasny: Twórca bloga ze swojej strony musi jedynie

dołączyć gem *administer* do projektu, zdefiniować modele biznesowe typu `Post` czy `Category`, oraz dopisać akcję odpowiedzialną za wyświetlanie artykułów, cała część odpowiedzialna za akcję zarządzania modelami ma być dynamicznie wykonywana przez gem *administer*.

Ilustracja 4.1 przedstawia stronę z listą artykułów, jest to widok pochodzący z aplikacji testowej. Ilustracja 4.2 przedstawia widok edycji rekordu konkretnego artykułu, funkcja ta implementowana jest przez bibliotekę administer.



Rysunek 4.1: Aplikacja testowa - wyświetlanie artykułów

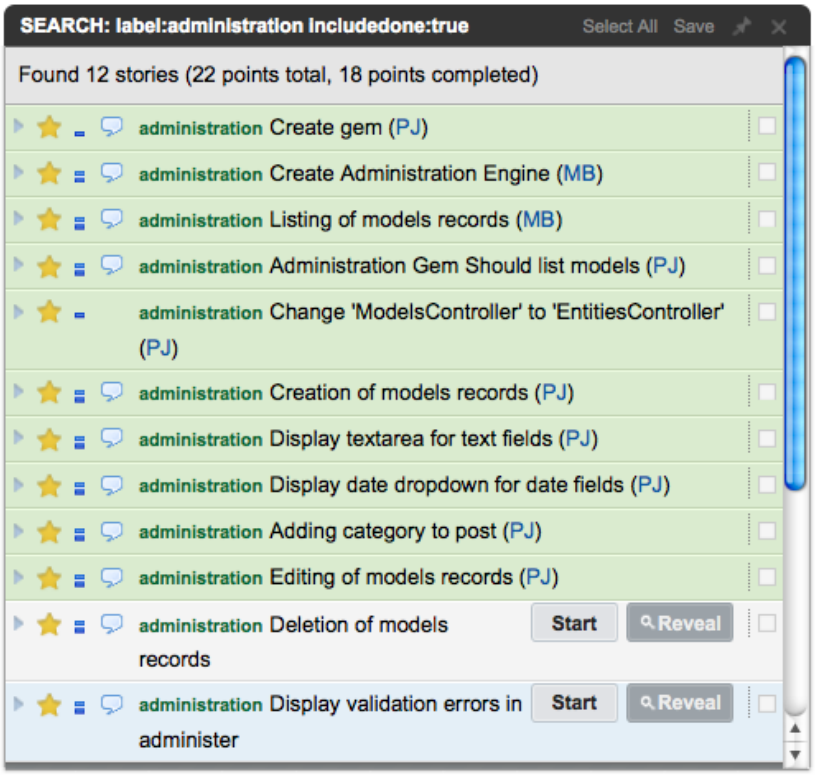


Rysunek 4.2: administer - edycja rekordu

4.3.2 Konfiguracja zależności testowych

4.3.3 Implementacja funkcji usuwania rekordu

Ilustracja 4.3 przedstawia wycinek listy zadań związanych z pracą nad projektem *administer* a konkretnie są to zadania związane z rozwojem panelu administracyjnego.



Rysunek 4.3: Lista zadań do wykonania

Jako, że projekt jest rozwijany już od jakiegoś czasu, większość zadań z tej listy jest już wykonana, znajdziemy jednak dwa nadal nie rozwiązane problemy:

- Deletion of model records
- Display validation errors in administer

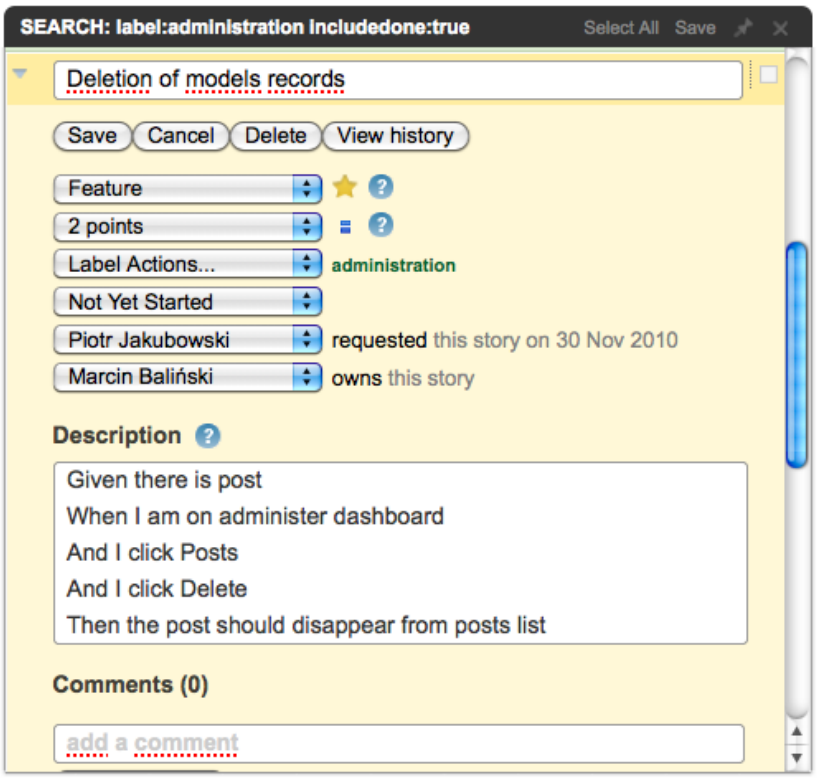
Skupmy się na rozwiązaniu pierwszego z nich. W momencie pisania tego rozdziału gem *administer* potrafi już tworzyć, modyfikować oraz wyświetlać informacje o rekordach modeli biznesowych aplikacji, która z niego korzysta. Jedną z brakujących funkcji jest usuwanie istniejących rekordów. W tym podrozdziale przeanalizujemy krok po kroku cały cykl wdrażania tej nowej funkcji do istniejącego systemu. Przeanalizowanie tego procesu pomoże nam lepiej zrozumieć sposób w jaki rozwijane jest oprogramowanie w zgodzie z zasadami BDD oraz jakie są korzyści z tego płynące.

Zrozumienie problemu

Pierwszy etap pracy to rozpoznanie, z czym tak właściwie mamy do czynienia. Jeśli problem, który mamy rozwiązać został wcześniej udokumentowany, tak jak miało to miejsce w tym wypadku, należy zacząć od dokładnego zaznajomienia się z jego opisem.

Ilustracja 4.4 przedstawia szczegółowy widok naszego zadania prezentowany przez aplikację Pivotal Tracker, która używana jest jako narzędzie wspomagające proces zarządzania projektem *administer*.

Podgląd ten informuje nas o wielu rzeczach związanych z zadaniem takich jak status ukończenia, estymacja pracochłonności czy informacja o osobach odpowiedzialnych za zadanie. W kontekście naszych rozważań najważniejszą informacją jaką możemy odczytać z niniejszego podglądu jest jednak opis funkcjonalności.



Rysunek 4.4: Szczegółowy opis zadania

Administer rozwijany jest razem z prostą aplikacją testową - systemem

blogowym, dlatego wszystkie funkcjonalności opisywane są w kontekście wykorzystania biblioteki przez ten właśnie system. Opis naszego zadania sformułowany jest następująco:

```
Given there is post
When I am on administer dashboard
And I click Posts
And I click Delete
Then the post should disappear from posts list
```

Na pierwszy rzut oka widać, że opis dostarczony został w postaci, której bez modyfikacji można użyć jako scenariusza akceptacyjnego biblioteki cucumber. Należy także zwrócić uwagę na inne ważne cechy tego opisu:

Niepodzielność Dostarczone kroki stanowią minimum, jakie trzeba zaimplementować aby funkcjonalność usuwania rekordów była kompletna.

Osadzenie w kontekście Zachowanie zostało opisane w konkretnych warunkach. Mamy informację o stanie początkowym w jakim funkcjonalność powinna działać (Istnieje rekord należący do modelu `Post`), miejscu oraz zachowaniu które wywoła usunięcie rekordu (Kliknięcie przycisku `Delete` na stronie z postami) oraz o warunkach końcowych, jakie muszą zostać spełnione w wyniku działania naszej nowej funkcji (Post powinien zniknąć z listy).

Czynności wstępne

Po zapoznaniu się i zrozumieniu istoty problemu, z którym będziemy się zmagać należy przygotować środowisko pracy. W naszym wypadku składają się na to dwie ważne czynności:

Upewnienie się, że posiadamy aktualną wersję oprogramowania Zawsze powinniśmy rozpocząć pracę z najnowszą rozwojową wersją kodu.

Uruchomienie pełnego zestawu testów Przed rozpoczęciem pracy należy upewnić się, że wszystkie testy kończą się sukcesem.

Kod biblioteki *administer* przechowywany jest w rozproszonym systemie kontroli wersji GIT, spełnienie pierwszego punktu wymaga więc od programisty jedynie synchronizacji swojego lokalnego repozytorium z najnowszą wersją rozwojowej gałęzi kodu znajdującej się w głównym repozytorium. Szczegóły dotyczące operacji na systemie kontroli wersji wykraczają poza ramy tej pracy i zostaną tutaj pominięte. Przyjmijmy jednak, że synchronizacja

została wykonana i mamy u siebie najnowszą wersję kodu biblioteki *administer*.

Uruchomienie pełnego zestawu testów daje nam pewność, że wersja oprogramowania, na której będziemy pracować spełnia warunki specyfikacji. Nie wolno rozpoczynać pracy jeśli testy z jakiegoś powodu nie przechodzą. W takim wypadku należy najpierw znaleźć przyczynę problemu i zlikwidować go a pracę nad nową funkcjonalnością rozpocząć dopiero kiedy wszystkie testy automatyczne sygnalizują sukces.

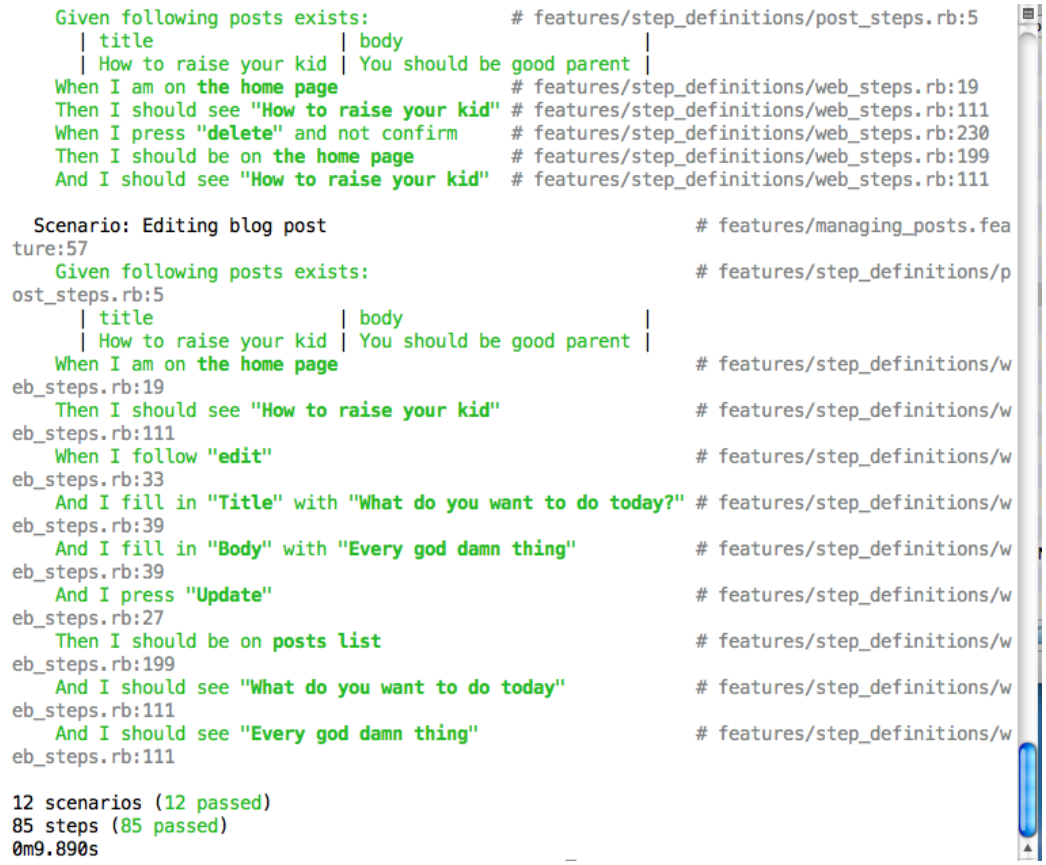
Aby uruchomić pełen zestaw testów biblioteki *administer* wydajemy z głównego katalogu projektu polecenie:

```
rake
```

Projekt został tak skonfigurowany, że komenda `rake` wywołana bez dodatkowych parametrów uruchamia zestaw scenariuszy Cucumber oraz testów RSpec opisujących zachowanie biblioteki.

```
MacBook-Marcin-Balinski:administer marcinbalinski$ rake
(in /Users/marcinbalinski/projects/administer)
(in /Users/marcinbalinski/projects/administer/spec/rails_root)
/System/Library/Frameworks/Ruby.framework/Versions/1.8/usr/bin/ruby -S bundle exec rspec "./
spec/controllers/administer/entities_controller_spec.rb" "./spec/controllers/posts_controlle
r_spec.rb" "./spec/models/administer/model_spec.rb" "./spec/routing/posts_routing_spec.rb" "
./spec/views/posts/edit.html.haml_spec.rb" "./spec/views/posts/new.html.haml_spec.rb" "./spe
c/views/posts/show.html.haml_spec.rb"
.................*..........

Pending:
  Administer::Model should return all models defined within ROOT/app/models/, using lookup
    # test for cases when @@models is empty and not empty
    # ./spec/models/administer/model_spec.rb:8

Finished in 0.79521 seconds
27 examples, 0 failures, 1 pending
```

Rysunek 4.5: Zestaw testów RSpec - raport.



Rysunek 4.6: Zestaw scenariuszy Cucumber - raport.

Raport po pozytywnym przejściu wszystkich testów powinien wyglądać podobnie jak na ilustracji 4.5 oraz 4.6. Oznaczenie `pending` przy jednym z testów RSpec sygnalizuje, że został tymczasowo wyłączony z zestawu testów przez jednego z programistów. Przyczyny wyłączenia testów mogą być różne, ale proceder ten należy stosować w ostateczności i w pełni świadomie, niedopuszczalne jest wyłączanie testu bez wyraźnego powodu a najlepiej jest, żeby takie sytuacje w ogóle się nie zdarzały.

Więcej testów

Proces implementacji zaczyna się od stworzenia scenariusza opisującego nową funkcjonalność. W naszym przypadku scenariusze umieszczone są w podkatalogu `spec/rails_root/features/`. Nas szczególnie interesuje plik `spec/rails_root/features/administer/posts.feature` w którym opisane jest zachowanie wymagane od panelu administrującego blogiem. W momen-

cie pisania tego rozdziału plik `posts.feature` wygląda następująco:

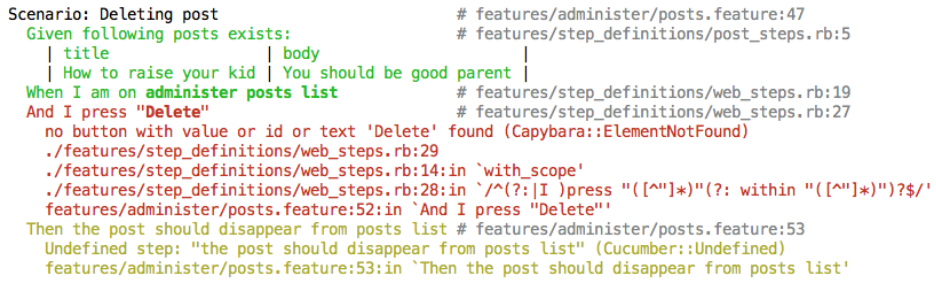
```
Feature: Managing posts via administer
  In order to make sure administer works
  I want to be able to manage posts via administer

  Scenario: Listing posts
    Given following posts exists:
      | title | body |
      | How to raise your kid | You should be good↩
          parent |
    When I am on the administer dashboard page
    And I follow "Posts"
    Then I should see "How to raise your kid"
    And I should see "You should be good parent"

  Scenario: Creating posts
    Given following categories exist:
      | name          |
      | Awesome Posts |
    When I am on administer posts list
    And I follow "New Post"
    Then I should see textfield with label "Title"
    And I should see textarea with label "Body"
    And I should see dateselect with label "↩
       Publish on"
    And I should see select with label "Category"
    And I fill in "Title" with "How to raise your ↩
       kid"
    And I fill in "Body" with "You should be good ↩
       parent."
    And I select "Awesome Posts" from "Category"
    And I press "Create Post"
    Then I should be on administer posts list
    And following posts should exist:
      | title | body | category_name |
      | How to raise your kid | You should be good↩
          parent. | Awesome Posts |

  Scenario: Editing Posts
    Given following posts exists:
```

```
    | title | body |
    | How to raise your kid | You should be good↩
        parent |
  When I am on administer posts list
  And I follow "Edit"
  Then I fill in "Title" with "Can we?"
  And I fill in "Body" with "Yes, we can"
  And I press "Update Post"
  Then I should be on administer posts list
  And following posts should exist:
    | title | body |
    | Can we? | Yes, we can |
```

Powyższe scenariusze opisują zachowanie biblioteki w tym momencie życia projektu. Funkcja usuwania rekordu nie jest jeszcze opisana, nie pozostaje nam nic innego jak dopisać odpowiedni scenariusz:

```
Scenario: Deleting post
  Given following posts exists:
    | title | body |
    | How to raise your kid | You should be good ↩
       parent |
  When I am on administer posts list
  And I press "Delete"
  Then the post should disappear from posts list
```

Oryginalny opis zadania (Ilustracja 4.4) został lekko zmodyfikowany w celu lepszego dopasowania do istniejącej infrastruktury i definicji kroków. Powtórne uruchomienie zestawu testów powinno zakończyć się niepowodzeniem - dodany przez nas scenariusz specyfikuje funkcjonalność, której jeszcze nie ma. Sytuację ilustruje zrzut ekranu 4.7.



Rysunek 4.7: Nowo dodany scenariusz nie przechodzi

Oprócz informacji o błędzie, dostajemy również informacje o tym, że jeden z kroków scenariusza jest nie zdefiniowany:

```
Undefined step: "the post should disappear ↩
   from posts list" (Cucumber::Undefined)
features/administer/posts.feature:53:in 'Then ↩
   the post should disappear from posts list'
```

Z problemem tym można poradzić sobie na dwa sposoby: dodać nową definicję dla kroku *the post should disappear from posts list* lub zmodyfikować scenariusz tak, aby korzystał z istniejących kroków nie zmieniając jednocześnie jego sensu. Generalnie rzecz biorąc niepotrzebne tworzenie nowych bytów nie jest dobrym pomysłem, należy przyjrzeć się więc czy możliwe jest wykorzystanie już istniejących definicji kroków, żeby opisać interesujące nas zachowanie. Plik

```
spec/rails_root/features/step_definitions/↩
   post_steps.rb
```

definiuje następujące kroki:

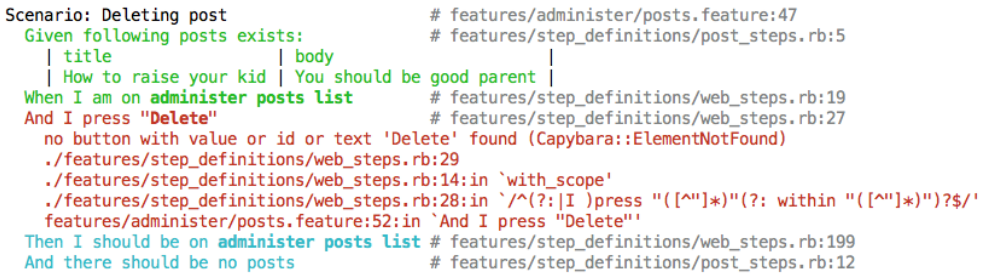
```
Given /^no posts exist$/ do
  Post.destroy_all
end

Given /^following posts exists:$/ do |posts_table|
  Post.destroy_all
  posts_table.hashes.each do |attributes|
    Factory(:post, attributes)
  end
end
```

```
Then /^there should be no posts$/ do
  Post.all.should be_empty
end

Then /^following posts should exist:$/ do |↩
   posts_table|
  posts_table.hashes.each do |attributes|
    category = attributes.delete("category_name")
    post = Post.first(:conditions => attributes)
    post.should_not be_nil
    post.category.name.should == category if ↩
       category
  end
end
```

Krok opisujący zniknięcie rekordu z listy można tak naprawdę opisać przy pomocy dwóch innych kroków, po pierwsze po kliknięciu przycisku `Delete` powinniśmy upewnić się, że nadal jesteśmy na liście Postów, po drugie wiedząc, że istniał tylko jeden post, po wciśnięciu `Delete` nie powinien istnieć żaden. Wykorzystamy więc dwa już zdefiniowane kroki: *I should be on administer posts list* oraz *there should be no posts*. Zmodyfikowany scenariusz wygląda następująco:

```
Scenario: Deleting post
  Given following posts exists:
    | title | body |
    | How to raise your kid | You should be good ↩
       parent |
  When I am on administer posts list
  And I press "Delete"
  Then I should be on administer posts list
  And there should be no posts
```

Powtórne uruchomienie testów nadal sygnalizuje niepowodzenie, jednak wszystkie kroki są zdefiniowane. Jedyne nad czym musimy się teraz skupić to dostarczenie funkcjonalności, która spełni warunki nowego scenariusza.



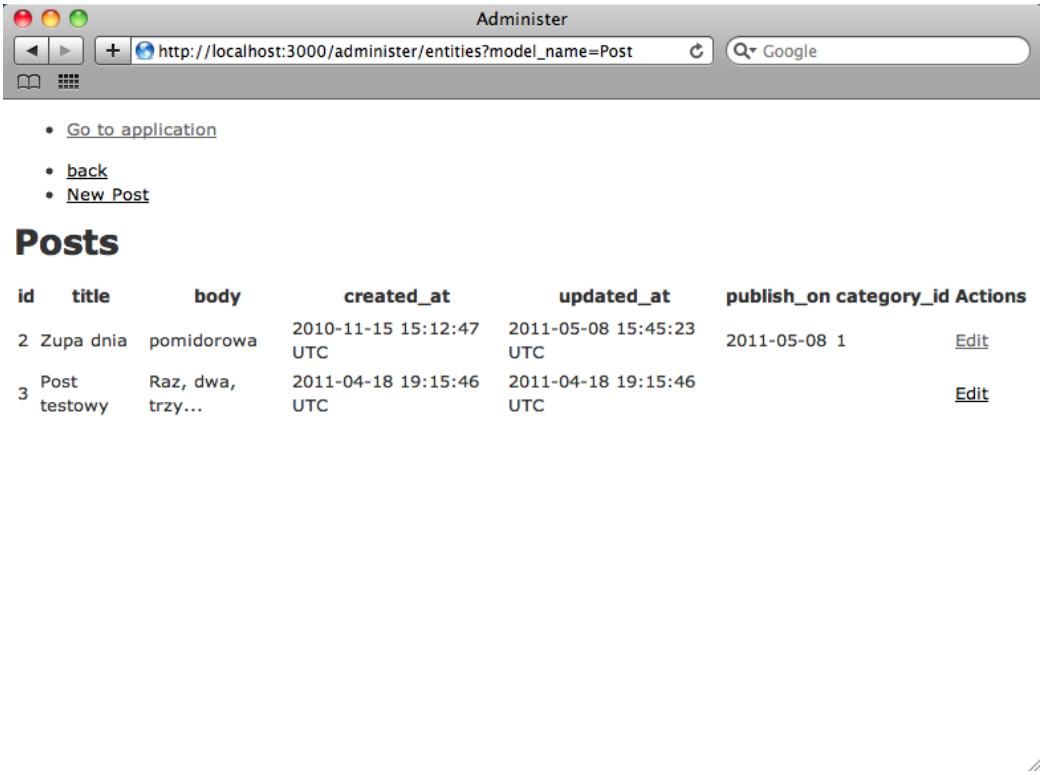
Rysunek 4.8: Nowo dodany scenariusz nie przechodzi, wszystkie kroki są zdefiniowane.

Dostarczenie wymaganej funkcjonalności

Nasze następne działania polegać będą na zaspokajaniu oczekiwań kolejnych kroków zdefiniowanego przez nas scenariusza aż do momentu ich wyczerpania i tym samym dostarczenia wymaganej funkcjonalności. Jak widać na ilustracji 4.7 dwa pierwsze kroki są zielone, skupmy się więc na trzecim z nich. Wszystko co musimy wiedzieć o przyczynach jego niepowodzenia zawarte jest w raporcie z przebiegu scenariusza:

```
And I press "Delete"   # features/↩
   step_definitions/web_steps.rb:27

no button with value or id or text 'Delete' ↩
   found (Capybara::ElementNotFound)
./features/step_definitions/web_steps.rb:29
./features/step_definitions/web_steps.rb:14:in ↩
   `with_scope'
./features/step_definitions/web_steps.rb:28:in ↩
   `/^(?:|I )press "([^"]*)"(?: within "([^"]*)↩
   ")?$/'
features/administer/posts.feature:52:in `And I ↩
   press "Delete"'
```

Przycisk `Delete` nie istnieje, dodajmy go do odpowiedniego widoku. Plik, którego szukamy to `app/views/administer/entities/index.html.haml` - definiuje on widok panelu administracyjnego zobrazowany na ilustracji 4.9



Rysunek 4.9: Lista rekordów wybranego modelu

Jest to ten sam widok, o którym mowa w scenariuszu. Naszym zadaniem będzie dodać przycisk `Delete` w kolumnie `Actions` dla każdego wyświetlanego rekordu. Przyjrzyjmy się naszemu widokowi od środka:

```
1  %ul.actions
2    %li= link_to('back', ↩
       administer_dashboard_index_path)
3    %li= link_to("New #{model_class.model_name.human↩
       }", new_administer_entity_path(:model_name => ↩
       model_class.model_name))
4  %h1= model_class.model_name.human.pluralize
5  %table.items_list
6    %thead
7      %tr
8      - model_class.column_names.each do |↩
         column_name|
9        %th= column_name
10     %th Actions
```

```
11
12    %tbody
13      - @collection.each do |object|
14        %tr{:class => cycle('odd', 'even')}
15          - model_class.column_names.each do |↩
              column_name|
16            %td= object.send(column_name).to_s
17          %td= link_to "Edit", ↩
              edit_administer_entity_path(object, :↩
              model_name => model_class.model_name)
```

Powyższy kod to prosty język szablonów HAML, który jest dynamicznie przetwarzany do postaci HTML zrozumiałej dla przeglądarki. Jeśli zlokalizujemy w nim kod odpowiedzialny za wyświetlenie linka `Edit` to prawdopodobnie bardzo łatwo będzie nam dodać pod nim przycisk `Delete`. Interesujący nas fragment znajdziemy w 17 linii:

```
          %td= link_to "Edit", ↩
              edit_administer_entity_path(object, :↩
              model_name => model_class.model_name)
```

Modyfikujemy kod szablonu dodając brakujący przycisk, który połączony jest z akcją odpowiedzialną za usunięcie odpowiedniego rekordu. Po modyfikacji interesujący nas fragment kodu wygląda następująco:

```
%td
  %ul.actions
    %li= link_to("Edit", ↩
        edit_administer_entity_path(object, :↩
        model_name => model_class.model_name))
    %li= button_to("Delete", ↩
        administer_entity_path(object, :model_name ↩
        => model_class.model_name), :method => :↩
        delete)
```

Dla zachowania poprawnej semantyki dokumentu dodanych zostało kilka tagów HTML, jednak najważniejsza deklaracja to:

```
    %li= button_to("Delete", ↩
        administer_entity_path(object, :model_name ↩
        => model_class.model_name), :method => :↩
        delete)
```

Kod ten składa się z metod implementowanych przez framework Rails oraz metod specyficznych dla samego projektu *administer*. W wyniku jego

działania wygenerowany zostanie przycisk o etykiecie `Delete` powiązany z metodą `destroy` kontrolera, który odpowiedzialny jest za zarządzanie rekordami modeli biznesowych przez bibliotekę *administer*. W kontekście naszych rozważań na temat testowo zorientowanych metod rozwoju oprogramowania istotne jest, że modyfikacja ta zaspokaja wymaganie kroku scenariusza, nad którym aktualnie pracujemy. Uruchommy ponownie zestaw testów, aby przekonać się, że mimo iż przycisk `Delete` w tej chwili istnieje, to jednak ciągle są problemy z tym samym krokiem, informacja o niepowodzeniu jest jednak inna niż poprzednio:

```
And I press "Delete"  # features/↩
   step_definitions/web_steps.rb:27

The action 'destroy' could not be found for ↩
   Administer::EntitiesController (↩
   AbstractController::ActionNotFound)
./features/step_definitions/web_steps.rb:29
./features/step_definitions/web_steps.rb:14:in `↩
   with_scope'
./features/step_definitions/web_steps.rb:28:in ↩
   `/^(?:|I )press "([^"]*)"(?: within "([^"]*)")↩
   ?$/'
features/administer/posts.feature:52:in `And I ↩
   press "Delete"'
```

Wyświetlamy przycisk `Delete` oraz jest on prawidłowo skojarzony z akcją `destroy` kontrolera `Administer::EntitiesController`. Problem w tym, że akcja `destroy` nie została jeszcze zaimplementowana co skutkuje błędem - próbą uruchomienia nieistniejącej akcji w momencie kiedy użytkownik (W tym wypadku symulowany przez bibliotekę Capybara) klika w przycisk.

Przejdźmy więc do wyżej wymienionego kontrolera `EntitiesController`. Jego głównym zadaniem jest obsługa żądań związanych z modelami biznesowymi istniejącymi w aplikacji, która korzysta z biblioteki *administer* do zadań tych zaliczają się m.in dodawanie, modyfikowanie i usuwanie rekordów tych modeli. Pełen kod kontrolera wygląda w tym momencie następująco:

```
module Administer
  class Administer::EntitiesController < ↩
     ApplicationController
    unloadable
    before_filter :set_model
    before_filter :collection, :only => :index
```

```
before_filter :fields, :only => [:new, :create↩
   , :edit, :update]

def new
  @object = model_class.new
end

def create
  @object = model_class.new(params[model_class↩
     .model_name.underscore])
  if @object.save
    redirect_to administer_entities_path(:↩
       model_name => model_class.model_name)
  else
    render :new
  end
end

def edit
  @object = model_class.find(params[:id])
end

def update
  @object = model_class.find(params[:id])
  if @object.update_attributes(params[↩
     model_class.model_name.underscore])
    redirect_to administer_entities_path(:↩
       model_name => model_class.model_name)
  else
    render :edit
  end
end

protected
def model_class
  @model.entity
end
helper_method :model_class

def set_model
  @model = Model.for(params[:model_name])
```

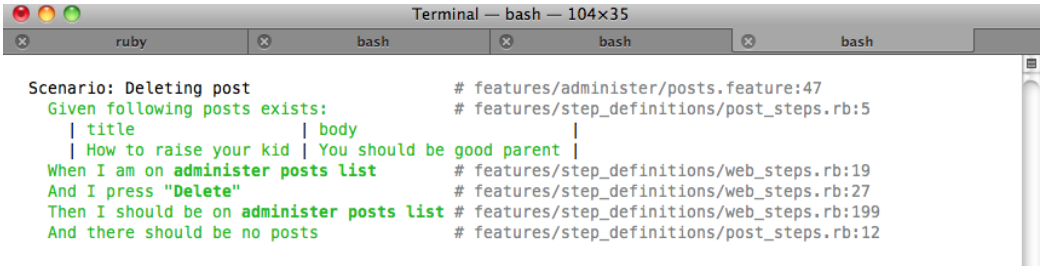
```
    end

    def collection
      @collection = model_class.all
    end

    # TODO: move this to model
    def fields
      @fields = @model.fields
    end
  end
end
```

Kontroler ten definiuje akcje, które użytkownik może za jego pomocą wykonać, oprócz tego robi kilka innych ciekawych rzeczy. Przede wszystkim ustala informację o rodzaju modelu, na którym ma przeprowadzać operacje. Nazwa modelu jest przekazywana jako jeden z parametrów żądania (`params[:model_name]`) i to na jej podstawie znajdowana jest właściwa klasa reprezentująca ten model, konkretne jego rekordy, lub cała ich kolekcja. W tej chwili istnieją cztery akcje: `new`, `create`, `edit` oraz `update` brakuje (o czym zostaliśmy już poinformowani przez nieprzechodzący scenariusz) akcji `destroy`, której implementacją zajmiemy się w tej chwili.

Z treści naszego scenariusza wynika, że zadaniem akcji `destroy` jest usunięcie konkretnego rekordu z bazy danych oraz przekierowanie użytkownika z powrotem na listę wszystkich rekordów. Na wejściu akcja musi więc otrzymać dwa parametry: nazwę modelu oraz identyfikator rekordu a na podstawie tych danych wykonać polecenie które usunie rekord należący do tego modelu oraz noszący taki właśnie identyfikator. Kod spełniający taką specyfikacje może wyglądać następująco:

```
1 def destroy
2   @object = model_class.find(params[:id])
3   unless @object.destroy
4     flash[:error] = "Could not destroy object."
5   end
6   redirect_to administer_entities_path(:model_name↩
        => model_class.model_name)
7 end
```

Po dodaniu metody `destroy` do kontrolera uruchamiamy testy raz jeszcze przy pomocy polecenia `rake`. Tym razem wszystkie scenariusze przechodzą.



Rysunek 4.10: W tym momencie funkcjonalność spełnia wymagania scenariusza.

Teoretycznie moglibyśmy w tym momencie uznać zadanie za zakończone, jednak pisząc metodę `destroy` wykroczyliśmy nieco poza to, co opisane zostało w scenariuszu. Linie 3-5 definiują następujący blok kodu:

```
unless @object.destroy
  flash[:error] = "Could not destroy object."
end
```

Jeśli usunięcie rekordu z jakiejś przyczyny nie zakończy się powodzeniem wyświetlony zostanie komunikat o treści *Could not destroy object*. Zachowanie to jest jak najbardziej sensowne, zostało ono jednak dodane samowolnie i nie jest nigdzie przetestowane. Jako odpowiedzialni programiści nie powinniśmy dopuścić do takiej sytuacji i dopisać odpowiedni test.

Przetestowanie tego zachowania scenariuszem Cucumber jest trudne, trudno bowiem odtworzyć sytuację, w której usunięcie rekordu zakończy się niepowodzeniem - może się tak stać np. kiedy silnik bazy danych w której jest przechowywany nałożył na ten rekord blokadę, która trwa w momencie próby jego usunięcia. W tej sytuacji do testowania wykorzystamy bibliotekę RSpec i jej mechanizm oczekiwań oraz sztucznych obiektów. Odpowiedni zestaw testów zdefiniowany jest w pliku:

```
spec/rails_root/spec/controllers/↩
   administer/entities_controller_spec.rb
```

W którym następująco opiszemy interesujące nas zachowanie:

```
describe "DELETE /:id" do
  it "should display error message when unable to ↩
     delete" do
    controller.should_receive(:model_class).twice.↩
       and_return(Post)
    post = mock_model(Post, :id => 1, :destroy => ↩
       false)
```

```
    Post.should_receive(:find).and_return(post)
    delete :destroy, :model_name => 'post', :id =>↩
        1
    flash[:error].should == "Could not destroy ↩
       object."
  end
end
```

Upewniliśmy się, że informacja o błędzie została ustawiona w kontrolerze. To jednak nie wszystko, musimy sprawdzić, czy jest ona wyświetlona w odpowiednim widoku. *Administer* wyświetla informacje o błędach w głównym widoku zdefiniowanym w pliku:

`app/views/layouts/administer.html.haml`

Odpowiedni test ma sprawdzać, czy ustawiony komunikat zostanie wyświetlony:

```
describe "layouts/administer.html.haml" do
  it "renders error message when assigned" do
    view.stub!(:flash).and_return({:error => '↩
       Could not destroy object'})
    render
    rendered.should match(/Could not destroy ↩
       object/)
  end
end
```

4.4 Podsumowanie

W niniejszym rozdziale prześledziliśmy krok po kroku proces tworzenia oprogramowania w zgodzie z zasadami *Behavior Driven Development*. Nadszedł czas aby wyciągnąć wnioski oraz podsumować ten sposób prowadzenia projektów informatycznych.

4.4.1 Zalety zastosowania testowo zorientowanych metodyk rozwoju oprogramowania

Oprogramowanie odporne na błędy

Dobre pokrycie kodu testami zapewnia, że system informatyczny jest dużo bardziej odporny na błędy, szczególnie podczas procesu jego modyfikacji. Jeśli

programista przez przypadek zmieni charakter już istniejącej funkcjonalności zostanie o tym natychmiast poinformowany przez jeden lub więcej nieprzechodzących testów. Jest to szczególnie istotne kiedy system jest skomplikowany i zmiana w jednym jego module może spowodować niepożądane zmiany w modułach współzależnych. Nieprzechodzące testy uświadamiają programiście jakie dokładnie części systemu zostały dotknięte wprowadzonymi przez niego zmianami, pomagają mu rozeznać się w sytuacji oraz podjąć decyzję na temat następnych kroków.

Funkcjonalność jest najważniejsza

Zasady *BDD* nakazują, żeby każde nowe zadanie rozpoczynać od napisania odpowiedniego scenariusza użycia. Scenariusz ten od tej chwili staje się częścią specyfikacji systemu. Odpowiedzialnością programisty jest doprowadzić system do stanu, który spełnia specyfikację, a więc do momentu kiedy wszystkie testy i scenariusze przechodzą. Można powiedzieć, że w testowo zorientowanych metodykach testy automatyczne wyznaczają rytm i kierunek działań programistów.

W poprzednich podrozdziałach pokazałem, w jaki sposób scenariusz zdefiniowany dla danej funkcji determinował kolejne kroki, jakie musiał podjąć programista, zawsze skupiając się na pierwszym nieprzechodzącym kroku i robiąc tylko tyle, ile trzeba, aby spełnić jego wymagania. Taki rytm pracy ma dwie ważne zalety: po pierwsze wymaga przemyślenia przyszłych zadań już na etapie pisania testów, po drugie bardzo ułatwia skupienie się tylko na tym co ważne, a więc jedynie na kolejnej czynności, która przybliży nas do celu.

Oszczędność czasu i pieniędzy

Co prawda na samym początku życia projektu pisanie zestawu testów do każdej nowej funkcjonalności może w istocie wydłużać czas pracy, jednak zmienia się to wraz z dojrzewaniem systemu. Przy dobrym pokryciu kodu testami większość usterek wynikających z modyfikacji zostanie wykryta i naprawiona od razu. Procent ukrytych usterek w przypadku oprogramowania dobrze pokrytego testami również jest znikomy, tak samo jest w przypadku nowych funkcjonalności - jeśli są dostarczane z odpowiednim zestawem testów, oraz nie psują już istniejących. Wszystko to sprawia, że z czasem oszczędność w porównaniu z systemami nie pokrytymi testami zaczyna być widoczna.

Pewność, że system zawsze działa według specyfikacji

Wynika to z tego, że testy pełnią jednocześnie rolę specyfikacji i testów automatycznych. Każda istotna zmiana w systemie musi zostać opisana testem, oraz żadna modyfikacja nie zostanie przyjęta, jeśli nie jest dostarczona z pełnym zestawem przechodzących testów. W praktyce oznacza to, że jeśli mamy dobry zestaw testów, możemy być spokojni, że nasza aplikacja działa prawidłowo.

Eliminacja czynnika ludzkiego z procesu testowania

Testy automatycznie mogą sprawdzać zarówno szczegóły implementacji, interfejs użytkownika jak i integrację poszczególnych modułów systemu. Czynnik ludzki jest zbędny w procesie testowania - wystarczą istniejące testy automatyczne. Oszczędza to czas, pieniądze oraz redukuje ilość niedopatrzeń i błędów jakie mógłby popełnić człowiek.

4.4.2 Czynniki determinujące skuteczność testowo zorientowanych metodyk rozwoju oprogramowania

Mechaniczne dopisywanie testów do każdej nowej linijki kodu nie ma dużego sensu. Istnieje wiele czynników, na które należy zwrócić uwagę jeśli chcemy rozwijać oprogramowanie w oparciu o testy i czerpać z tego maksymalne korzyści. Część z wymienionych przeze mnie poniżej punktów ma ogromne znaczenie dla prawidłowego wdrożenia metod testowania w ogóle, część z nich wpływa na jakość oprogramowania lub komfort jego rozwoju.

Wybór odpowiednich narzędzi

Sposobów na testowanie kodu jest bardzo wiele. Na upartego działanie metody można przetestować nawet wewnątrz jej ciała, a istnieją również języki oraz techniki programowania, które z definicji sprzężone są z pewnymi mechanizmami testowania - przykładem jest język *Eiffel*[8], który natywnie wspiera ideę *Programowania Kontraktowego*[9].

Z drugiej strony dla każdego znanego języka programowania istnieją wyspecjalizowane biblioteki i narzędzia, z których możemy skorzystać. W przypadku języka *Ruby* wybór narzędzi jest bardzo duży, jak więc wybrać najodpowiedniejsze? Odpowiedź na to pytanie nie jest jednoznaczna, często wynika

[8] `http://en.wikipedia.org/wiki/Eiffel_(programming_language)`
[9] `http://en.wikipedia.org/wiki/Design_by_contract`

również z indywidualnych upodobań członków zespołu jednak istnieje kilka wskazówek, które mogą pomóc nam podjąć decyzję.

Narzędzia służące do testowania różnią się od siebie stopniem specjalizacji. Przykładowo biblioteka *Test::Unit* dostarczana standardowo wraz z językiem *Ruby* jest narzędziem bardzo ogólnym, które nadaje się do zastosowania w każdego rodzaju projekcie. Z drugiej strony barykady stoi biblioteka *Cucumber*, która podchodzi do tematu testowania w szczególny sposób, który w założeniach wymaga ścisłego dostosowania stylu prowadzenia projektu do zasad *BDD*. Należy dokładnie przeanalizować sytuację w jakiej się znajdujemy, rodzaj oprogramowania, które rozwijamy i skonfrontować tą informację z charakterem dostępnych narzędzi.

Przykładowo jeśli pracujemy nad projektem sklepu internetowego dla klienta, który bardzo chętnie włącza się w proces planowania i świetnie współpracuje z programistami a do jest skłonny wziąć na siebie obowiązek specyfikowania funkcjonalności *Cucumber* będzie idealnym rozwiązaniem. Jeśli pracujemy nad projektem wewnętrznym, lub z ludźmi o dużej świadomości technicznej, a chcemy tworzyć system w zgodzie z zasadami *BDD* lepszym rozwiązaniem okaże się *Steak*, który nie wymaga od nas definiowania wzorców językowych. Nic też nie stoi na przeszkodzie aby pozostać w duchu *Test Driven Development* korzystając z jednej z bibliotek wspierających testowanie jednostkowe jak wspomniany wcześniej *Test::Unit*.

Przykładem niezbyt dobrego doboru narzędzi do testowania może być również biblioteka *administer*. *Cucumber* został tutaj użyty jako główne narzędzie służące do specyfikowania i testowania funkcjonalności, jednak po pewnym czasie okazało się, że w przypadku biblioteki *administer* korzyści z użycia scenariuszy *Cucumbera* w porównaniu na przykład ze scenariuszami definiowanymi przy pomocy biblioteki *Steak* są żadne. *Administer* rozwijany jest przez profesjonalnych programistów na ich własne potrzeby, nie ma tutaj osoby klienta, który mógłby wyciągnąć korzyści z istnienia scenariuszy pisanych naturalnym językiem. Adresatami biblioteki również są programiści, którzy mogą wdrożyć go do swoich własnych aplikacji. W tym wypadku dużo lepszym rozwiązaniem byłoby sformułowanie testów behawioralnych przy użyciu bardziej ogólnego narzędzia jak *Steak* lub nawet ograniczenie się jedynie do wykorzystania biblioteki *RSpec* - oszczędziłoby to wysiłku niepotrzebnie zmarnowanego na definiowanie oraz utrzymanie kroków i wzorców językowych.

Stopień pokrycia kodu testami

Żadne testy nie będą skuteczne, jeśli stopień pokrycia nimi kodu nie będzie wystarczająco wysoki. Jeśli prowadzimy projekt na zasadach *BDD* to należy pisać scenariusze dla każdej kluczowej funkcjonalności systemu oraz testy jednostkowe dla części systemu, których nie da się w prosty sposób opisać scenariuszami, lub z innych przyczyn użycie testów jednostkowych wydaje się bardziej naturalne. Testy jednostkowe piszemy również zawsze wtedy, kiedy szczegóły implementacji danej części systemu są ważne z punktu widzenia końcowego odbiorcy.

W przypadku projektów, które nie są testowane behawioralnie testy jednostkowe powinny jak najdokładniej pokrywać publiczną część implementacji, do testowania całości systemu oraz symulowania różnych scenariuszy użycia powinniśmy również napisać odpowiednie testy integracyjne.

Wysoki stopień pokrycia kodu testami daje nam pewność, że wyłapią one więcej usterek, które na pewno będą pojawiać się w procesie implementacji, dlatego też bardzo ważne jest aby **nigdy nie odkładać pisania testów na później** - zawsze należy dostarczać je wraz z testowanym kodem.

Sposób testowania, albo *Pisanie kodu jest łatwe, pisanie testów jest trudne*

Pisząc test czy scenariusz musimy przede wszystkim wiedzieć co naprawdę chcemy przetestować. Na tym etapie pojawić się może wiele pytań.

- Czy szczegóły implementacji są ważne?
- Czy interesuje nas sam wynik?
- Czy skorzystamy z testowych danych czy danych rzeczywistych?
- A może wykorzystamy imitację obiektów?

Od tego w jaki sposób przetestujemy kod zależy czy nasze testy będą spełniać prawidłowo swoją funkcję jak również czy będą łatwe w utrzymaniu. Temat ten jest bardzo rozległy a pisanie dobrych testów to trudna sztuka, której programista uczy się wraz z doświadczeniem. W niektórych wypadkach wystarczy, że test sprawdza tylko wartość zwracaną przez metodę. Nawet jeśli jest ona dość skomplikowana, jednak szczegóły implementacji nie są istotne, to taki test spełni swoją rolę.

W innym wypadku może okazać się, że nawet najmniejsze szczegóły implementacji są dla nas bardzo ważne i będziemy chcieli je dokładnie przetestować. Takie testy są z reguły trudniejsze w utrzymaniu - będą kończyć się

niepowodzeniem nawet po bardzo małych modyfikacjach testowanego kodu sygnalizując jego niezgodność ze specyfikacją.

Tego typu decyzję programista podejmuje za każdym razem, kiedy pisze nowy test. Podobnie jest w przypadku pytań o rodzaj danych, jakich użyjemy do testów, wykorzystanie rzeczywistych składników systemu, bądź imitacji obiektów itd. Nie ma tutaj niestety żadnej złotej reguły a każdy przypadek należy rozpatrywać osobno. Sekcja ta ma na celu zwrócenie uwagi na to, że przemyślenie wszystkich tych szczegółów jest niezmiernie ważne, zależy od nich czy test będzie dobry, czy nie.

Testuj usterki

Każda wykryta usterka w oprogramowaniu oznacza, że powinna pojawić się odpowiednia modyfikacja w istniejących testach bądź powinien zostać napisany zupełnie nowy test, który ją udokumentuje. Jeśli mamy usterkę w kodzie, której nie wykryły testy oznacza to, że nie są one kompletne i powinniśmy natychmiast je poprawić w taki sposób żeby nie przechodziły przed usunięciem problemu, a przechodziły po jego likwidacji. Da nam to pewność, że jeśli w przyszłości zdarzy się podobny problem zostanie on natychmiast wykryty.

4.4.3 Wnioski

Testowanie kodu nie jest dziś opcjonalną praktyką, a koniecznością. Automatyzacja testów pozwala znacząco oszczędzić czas oraz zminimalizować margines błędów oraz niedopatrzeń jakie mógłby popełnić człowiek. Metodyki takie jak *TDD* czy *BDD* pozwalają nie tylko pisać dobre testy, narzucają również sposób w jaki zespół myśli o projekcie oraz w jaki sposób go rozwija. Sprawia to, że odpowiednie stosowanie tych technik ma pozytywny wpływ nie tylko na pokrycie kodu testami ale również na wiele innych aspektów, które na pierwszy rzut oka mogą nie mieć z testowaniem nic wspólnego. Każdy programista powinien testować swój kod ponieważ kod nie przetestowany to kod, któremu nie można ufać. Opisane w tej pracy metodyki są tak naprawdę zbiorem zasad, narzędzi i dobrych praktyk, których skuteczność została wielokrotnie potwierdzona. Rozpoczynając nowy projekt należy zadać sobie nie pytanie o to, czy je wdrożyć, a raczej czy stać nas na to, aby tego nie zrobić.

Bibliografia

[1] Polskie forum ruby. http://rubyonrails.pl/forum/.

[2] Cucumber documentation. http://github.com/cucumber/cucumber/wiki/, 2011.

[3] Rspec documentation. http://rspec.info/, 2011.

[4] Ruby on rails guides. http://guides.rubyonrails.org/, 2011.

[5] Test::unit documentation. http://test-unit.rubyforge.org/test-unit/, 2011.

[6] David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, and Dan North. *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, 2009.

[7] Hal Fulton. *Ruby Tao Programowania w 400 Przykładach*. Helion, 2008.

[8] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison Wesley, 1999.

[9] Wikipedia. Scrum (development). http://en.wikipedia.org/wiki/Scrum_(development)/, 2011.

[10] Wikipedia. Test automation. http://en.wikipedia.org/wiki/Test_automation/, 2011.