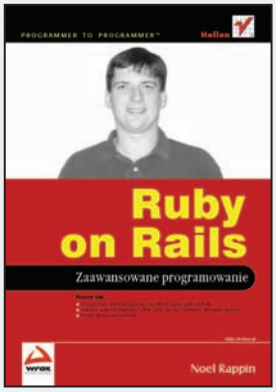
Ruby on Rails. Zaawansowane programowanie



Autor: Noel Rappin
Tłumaczenie: Justyna Walkowska
ISBN: 978-83-246-1844-6
Tytuł oryginału: Professional Ruby On Rails
Format: 172x245, stron: 488

Naucz się:

- tworzyć testy automatyczne dla wszystkich części aplikacji Rails
- wdrażać aplikacje napisane w Ruby przy użyciu Capistrano, Mongrel i Apache
- bronić stron przed atakami

Ruby on Rails (RoR) to sieciowy szkielet open source, pozwalający utrzymać równowagę pomiędzy łatwością programowania a jego produktywnością. To, co odróżnia ten framework od innych, to przewaga konwencji nad konfiguracją, co ułatwia budowę i zrozumienie aplikacji. Prostota i intuicyjność tego środowiska pomagają uniknąć powtórzeń i sprawiają, że programowanie jest łatwiejsze niż kiedykolwiek. W ciągu lat w RoR wprowadzono szereg zmian, związanych z ewolucją technik programistycznych. Poza tym wystarczającą rekomendacją dla tego środowiska wydaje się uznanie wyrażane przez takie osoby, jak James Duncan Davidson (twórca Tomcata i Anta), Bruce Perens (Open Source Luminary), Nathan Torkington (O'Reilly, OSCON) i wiele innych.

Książka „Ruby on Rails. Zaawansowane programowanie" jest przeznaczona dla średnio i bardzo zaawansowanych programistów Rails. Autor zakłada, że Czytelnik zna język Ruby i przeczytał chociaż jedną z dostępnych książek, wprowadzających w świat Rails, lub ma za sobą inną formę podstawowego kursu. Czytelnik tej książki powinien wiedzieć, jak stworzyć prostą aplikację Rails. W tej publikacji znajdzie natomiast szereg odpowiedzi na pytania pojawiające się po napisaniu pierwszej aplikacji. Autor wyjaśnia, jak poradzić sobie z użytkownikami i zabezpieczeniami, opisuje obsługę stref czasowych i problemy związane z użytkowaniem aplikacji w różnych stronach świata oraz podaje sposoby zabezpieczania strony przed atakami. Czytelnik znajdzie tu porady dotyczące zarządzania zespołem programistów Rails i kodem źródłowym, automatyzacji powtarzalnych zadań i wdrażania aplikacji w środowisku produkcyjnym, a także sposobów korzystania z nieustannie powstających rozszerzeń Rails.

- Tworzenie zasobów
- Kontrola kodu przy pomocy Subversion (SVN)
- Budowanie i automatyzacja
- Nawigacja i portale społecznościowe
- Opieka nad bazami danych
- JavaScript w Rails
- Narzędzia do testowania
- Metaprogramowanie
- Tworzenie wtyczek

Poszerz swoją wiedzę na temat środowiska Ruby on Rails

Spis treści

7

Narzędzia do testowania

Znaczną część tej książki poświęcam testom automatycznym. Robię to dlatego, że uważam je za jedną z najważniejszych metod dbania o jakość i stabilność kodu. Do tej pory skupiałem się na standardowej strukturze Test::Unit, czyli podstawowym zestawie narzędzi sprawdzających poprawność kodu w języku Ruby. Test::Unit stanowi bardzo ważną część testowania automatycznego, jednak nie jest jedynym narzędziem, z którego powinien korzystać programista chcący przeprowadzić kompletne testy swojej aplikacji.

W tym rozdziale przedstawię różne narzędzia do testowania aplikacji i postaram się przekonać Czytelnika, że każde z nich wnosi nową jakość do programowania sterowanego testami. Pokażę, jak zmierzyć ilość kodu pokrytego testami oraz jak wykorzystać obiekty-atrapy w celu poprawy jakości kodu i przetestowania jego trudno dostępnych części. Omówię także narzędzia do testów sterowanych zachowaniem, które są techniką intensywnie wykorzystującą atrapy. Na koniec powiem, jak oddzielić testy kontrolera od testów widoku.

Programowanie sterowane testami

Testy automatyczne zostały po raz pierwszy uznane za standardową część procesu wytwarzania oprogramowania, gdy stały się jedną z głównych praktyk programowania ekstremalnego (XP). Ta metodologia jest często źle rozumiana, jednak — niezależnie od XP, a częściowo również dzięki niemu — praktyki wcześniej określane mianem *Test-First Programming* (programowanie „najpierw testy") są teraz znane pod nazwą *Test-Driven Development* (programowanie sterowane testami, w skrócie TDD).

TDD składa się z trzech kroków, które powtarza się w kółko, aż do ukończenia aplikacji. Oto one:

1. Napisz prosty test, który będzie sprawdzał coś, czego Twój program jeszcze nie robi. Jeśli ten krok zajmuje więcej niż kilka minut, oznacza to, że próbujesz zrobić za dużo lub że struktura aplikacji jest zbyt złożona. Test powinien zakończyć się niepowodzeniem, ponieważ nowa funkcjonalność nie została jeszcze dodana do systemu. Uruchom test, aby upewnić się, że aplikacja rzeczywiście go nie przejdzie.

2. Stwórz prosty kod, dzięki któremu test będzie kończył się sukcesem. Istotne jest, by nie zastanawiać się nad tym, czy aplikacja przejdzie następny test — skup się tylko na bieżącym teście.
3. Przeprowadź refaktoryzację. Możliwe, że w wyniku zmian wprowadzonych w aplikacji w poprzednich krokach pojawiły się niepożądane elementy, takie jak duplikacja kodu. Wyczyść je od razu i upewnij się, że aplikacja pomyślnie przechodzi napisane do tej pory testy. Następnie przejdź do kolejnego testu.

Przykłady zamieszczone w tej książce najczęściej przedstawiają pierwszą wersję testu oraz końcowy kod. Raczej nie omawiam procesu refaktoryzacji, który ma na celu wyczyszczenie kodu. Czasami przedstawiam kilka powiązanych ze sobą testów, mimo że w rzeczywistości kod powstawał zgodnie z przedstawionymi powyżej krokami — po jednym teście naraz.

Można spotkać się z różnymi opiniami na temat prostoty kodu dodawanego w drugim kroku. Niektórzy programiści posuwają się do ekstremalnego stosowania stałych. Jeśli test ma postać:

```
assert_equals(7, x.foo)
```

postulują oni rozpoczęcie implementacji od

```
def foo
  return 7
end
```

Jeśli `foo` nie ma być metodą zwracającą stałą wartość, należy dodać inny test, który nie zakończy się sukcesem. W końcu powinno się okazać, że łatwiej jest napisać całą metodę, niż dodawać zwracanie nowych stałych. Moim zdaniem takie rozwiązanie jest pewną przesadą (chociaż w praktyce do pisania właściwej metody dochodzi się bardzo szybko). Sam stosuję podejście polegające na tworzeniu „najprostszego kodu, który można by uznać za implementację tej funkcji", bądź „najprostszego kodu, który zadziałałby, nawet gdybym nie wiedział dokładnie, jakie wartości zostaną podane podczas testów". Warto również napisać osobne przypadki testowe dla sytuacji, w których mogą pojawić się błędy po przekazaniu pustych argumentów.

Pod żadnym pozorem nie wolno pomijać kroku związanego z refaktoryzacją, ponieważ to wtedy odbywa się właściwe projektowanie. Częsta refaktoryzacja jest prosta i mało kosztowna, a na dodatek zmniejsza ona prawdopodobieństwo tego, że na późniejszym etapie konieczne będzie przeprowadzenie wielkich porządków.

Zgodnie z teorią aplikacja wytwarzana zgodnie z tym procesem zawsze jest w całości pokryta testami i jest możliwie prosta. Są to bardzo ważne cechy — aplikacja napisana w oparciu o TDD powinna w przyszłości o wiele łatwiej poddawać się zmianom i być prostsza w utrzymaniu.

TDD to przede wszystkim metodologia wytwarzania oprogramowania. Postulowane przez nią testy automatyczne dają programiście pewność co do jakości i funkcjonalności kodu, ale nie mogą zostać uznane za wystarczające narzędzie testujące. Zwłaszcza wartość testów pisanych przez programistę jest ograniczona przez jego pojmowanie problemu. Innymi słowy, programista napisze testy jednostkowe jedynie dla sytuacji, które jest w stanie przewidzieć. W celu sprawdzenia rzeczywistej wartości oprogramowania należy przeprowadzić dodatkową rundę ręcznych lub automatycznych testów akceptacyjnych.

Pokrycie całości

Ocena jakości testów jednostkowych jest nierozerwalnie związana z odpowiedzią na pytanie, czy testy sprawdzają całość kodu aplikacji. Wprowadzono nawet specjalne pojęcie *stopnia pokrycia* (ang. *code coverage*). Wskazuje on procent kodu aplikacji, który jest uruchamiany podczas testowania. Można go wyznaczać w oparciu o liczbę linii kodu bądź liczbę rozgałęzień w kodzie. Niezależnie od tego, na co się zdecydujemy, najważniejsze jest to, że naszym dążymy do pokrycia testami 100 procent aplikacji Rails (sam rozpocząłem próby osiągnięcia tych 100 procent na dwa sposoby: poprzez 100-procentowe pokrycie modeli samymi testami jednostkowymi oraz 100-procentowe pokrycie całej aplikacji kompletnym zestawem testów). Szczegóły wyliczania stopnia pokrycia nie są aż tak istotne.

Oczywiście samo pokrycie nie zapewnia jakości testów. Ktoś mógłby stworzyć testy, podczas których uruchamiany jest każdy skrawek aplikacji, ale nie jest przeprowadzana żadna asercja. W takim wypadku testy sprawdzałyby jedynie, czy program działa bez „wywrotek" (co w niektórych wypadkach okazuje się całkowicie poprawnym testem). Brak pokrycia prawie zawsze oznacza kłopoty — części aplikacji, do których nie ma testów często zawierają problematyczny kod. Jeśli jednak wiemy, że osoby piszące testy do naszej aplikacji tworzą je regularnie i kompetentnie, wówczas stopień pokrycia kodu może służyć za przyzwoity wyznacznik jakości testów.

Instalacja rcov

rcov to standardowe narzędzie do mierzenia stopnia pokrycia testami aplikacji Rails. Nie jest wbudowane w Rails, ale można zainstalować je na dwa sposoby. Osoby, które nie mają dostępu do kompilatora C, mogą zainstalować rcov jako gem Ruby:

```
$ gem install rcov
```

Podobnie jak w przypadku innych gemów być może trzeba będzie wybrać osobną wersję przeznaczoną dla aktualnie używanego systemu operacyjnego.

Jeśli programista ma dostęp do kompilatora C (czyli, ogólnie rzecz biorąc, korzysta z dowolnej poważnej dystrybucji Linuksa, z Mac OS X z zainstalowanym Xcode lub z Windows z darmowym kompilatorem konsolowym), powinien zainstalować rcov w wersji natywnego rozszerzenia. Twórcy narzędzia zapewniają, że w tej postaci będzie ono działało dwa razy szybciej, co z pewnością równoważy koszt wpisania dodatkowego polecenia.

Przed instalacją należy pobrać spakowany plik ze strony *http://eigenclass.org/hiki.rb?rcov* i rozpakować go do nowego katalogu. Z wnętrza tego katalogu należy następnie wydać polecenie `setup`:

```
$ ruby setup.rb
```

Dla osób, które pracują w systemie Windows pozbawionym kompilatora, możliwe jest jeszcze pobranie z tej samej strony prekompilowanego rozszerzenia dla jednej ze standardowych dystrybucji Ruby. Na stronie znajduje się instrukcja instalacji.

W celu integracji rcov z Rails konieczna jest instalacja pluginu `rails_rcov`:

```
$ ruby ./script/plugin install -x http://svn.codahale.com/rails_rcov
```

Jeśli ktoś nie chce korzystać z Subversion, może jak zwykle pominąć opcję `-x`. Ten akurat plugin zawiera wyłącznie pojedynczy plik Rake, zatem Subversion na wiele się nie przyda.

Jak korzystać z rcov w Rails?

Plugin `rails_rcov` rozszerza nasz repertuar o kilka nowych zadań Rake. Dla każdego typu testów (jednostkowych, funkcjonalnych i integracyjnych) plugin dodaje zadanie `rcov`, tworzące dane `rcov`, oraz zadanie `clobber`, które czyści te dane. Nazwy nowych zadań są rozszerzeniami nazw istniejących typów testów, zatem mamy na przykład zadania `test:units:rcov` lub `test:units:clobber`.

Wymienione zadania Rake są przydatne, jednak nie są kompletne. Każde z nich uruchamia odpowiedni typ testów i tworzy raport pokrycia, którego najważniejsze elementy są wysyłane na standardowe wyjście. Wyjście plikowe jest zapisywane w katalogu *<główny katalog Rails>/coverage*. Warto nakazać Subversion ignorowanie plików w tym katalogu, ponieważ nie ma sensu wgrywanie ich do repozytorium (aby Subversion nie wykrywał plików, trzeba będzie najpierw dodać sam katalog).

Informacja o pokryciu aplikacji testami funkcjonalnymi jest oczywiście ciekawa, jednak chcemy również wiedzieć, w jakim stopniu wszystkie testy pokrywają aplikację. Niestety, nie istnieje standardowe zadanie gromadzące te dane w jednym miejscu (w dokumentacji pluginu znajduje się informacja, że takie zadanie istnieje, ale to kłamstwo — a przynajmniej było to kłamstwo, kiedy ostatni raz sprawdzałem). Oczywiście samodzielne skonstruowanie takiego zadania nie jest przesadnie trudne. Poniższy przykład poprawia wersję przedstawioną na stronie rcov. Należy umieścić kod w pliku *lib/tasks/coverage.rake*:

```
require 'rcov/rcovtask'

namespace :test do
  namespace :coverage do
    desc "Usuwa łączne dane na temat pokrycia."
    task(:clean) do
      rm_rf "data/coverage"
      rm_f "data/coverage.data"
    end
  end

  test_types = %w[unit functional integration]

  desc 'Łączy dane na temat pokrycia kodu testami jednostkowymi,
  ↪funkcjonalnymi i integracyjnymi'
  task :coverage => "test:coverage:clean"

  tests_to_run = test_types.select do |type|
    FileList["test/#{type}/**/*_test.rb"].size > 0
  end

  tests_to_run.each do |target|
    namespace :coverage do
```

```
    Rcov::RcovTask.new(target) do |t|
      t.libs << "test"
      t.test_files = FileList["test/#{target}/**/*_test.rb"]
      t.verbose = true
      t.rcov_opts << '--rails --aggregate data/coverage.data'
      if target == tests_to_run[-1]
        t.output_dir = "data/coverage"
      else
        t.rcov_opts << '--no-html'
      end
    end
  end
  task :coverage => "test:coverage:#{target}"
 end
end
```

W pliku znajdują się dwa zadania — `test:coverage_clean` i `test:coverage` — które w programistyczny sposób tworzą zadania rcov, umieszczając ostateczny wynik w katalogu *coverage/*↪*complete*. Najważniejsze są tutaj opcje `--aggregate` i `--no-html`. Pierwsza z nich sprawia, że plik *coverage.data* będzie zawierał wspólne dane wygenerowane przez osobne zadania (działanie rcov będzie nieco spowolnione, dlatego naprawdę warto zainstalować natywne rozszerzenie). W normalnych okolicznościach doprowadziłoby to do sytuacji, w której dane każdego zadania są łączone z danymi wszystkich wykonanych wcześniej zadań. Ponieważ nie wydaje się to szczególnie przydatne, we wszystkich testach z wyjątkiem końcowego, który jest zapisywany w *coverage/complete*, używam opcji `--no-html`.

Zadanie `test:coverage` uruchamia wszystkie trzy zestawy testów jednostkowych, które mają dwa rodzaje wyjścia: ich normalne wyjście oraz tekstowy przegląd danych. Przy okazji powstaje bardzo wiele plików HTML. Przyjrzyjmy się najpierw plikowi *index.html*. Powinien on wyglądać mniej więcej tak jak zawartość rysunku 7.1.

Rysunek 7.1

CO code coverage information

Generated on Wed Sep 03 12:26:11 +0200 2008 with rcov 0.8.1.2

Name	Total lines	Lines of code	Total coverage	Code coverage
TOTAL	1130	903	95.9%	95.1%
app/controllers/application.rb	31	23	100.0%	100.0%
app/controllers/categories_controller.rb	19	16	100.0%	100.0%
app/controllers/ingredients_controller.rb	101	76	100.0%	100.0%
app/controllers/recipes_controller.rb	122	95	91.8%	89.5%
app/controllers/users_controller.rb	137	110	95.6%	94.5%
app/helpers/application_helper.rb	59	47	96.6%	95.7%
app/helpers/categories_helper.rb	2	2	100.0%	100.0%
app/helpers/ingredients_helper.rb	2	2	100.0%	100.0%
app/helpers/recipes_helper.rb	2	2	100.0%	100.0%
app/helpers/tabular_form_builder.rb	37	32	100.0%	100.0%
app/helpers/users_helper.rb	2	2	100.0%	100.0%
app/models/authorization_mailer.rb	11	10	100.0%	100.0%
app/models/extensions.rb	18	13	100.0%	100.0%
app/models/ingredient.rb	69	56	100.0%	100.0%
app/models/ingredient_base.rb	18	16	100.0%	100.0%
app/models/ingredient_data.rb	13	9	100.0%	100.0%
app/models/ingredient_data_proxy.rb	9	4	100.0%	100.0%
app/models/ingredient_parser.rb	67	58	100.0%	100.0%
app/models/math_captcha.rb	71	60	100.0%	100.0%
app/models/menu_sidebar.rb	60	45	100.0%	100.0%
app/models/rating.rb	3	3	100.0%	100.0%
app/models/recipe.rb	71	60	91.5%	90.0%
app/models/remote_data_proxy.rb	33	27	33.3%	25.9%
app/models/tag_cloud.rb	44	36	100.0%	100.0%
app/models/token.rb	60	48	100.0%	100.0%
app/models/user.rb	49	36	100.0%	100.0%
lib/tasks/cleanup.rake	20	15	100.0%	100.0%

Plik zawiera listę wszystkich plików źródłowych projektu Rails. Dla każdego pliku podano liczbę linii w ogóle i liczbę linii kodu. Różnica polega na tym, że wartość przedstawiona w kolumnie `Total` została obliczona z uwzględnieniem komentarzy, a także linii `def` i `end`, podczas gdy kolumna `Lines of code` zawiera tylko wykonywalne linie. W oparciu o te wartości wyliczony został odsetek linii, które zostały chociaż raz wykonane podczas testów. rcov może także podać informację na temat tego, ile razy wykonano daną linię, jednak w tej chwili nie jest to nam specjalnie potrzebne. Pliki, które nie zostały uruchomione podczas testów, nie pojawiają się na liście, dlatego należy sprawdzić, czy widać tam wszystkie oczekiwane pliki.

Każdy wiersz tabeli opisuje jeden plik. Nazwy plików są linkami prowadzącymi do bardziej szczegółowych informacji na temat pokrycia. Na rysunku 7.2 widać fragment pliku *ingredients_controller.rb*, w którym oznaczono jeden z segmentów niepokrytych testami.

Rysunek 7.2

```
42  end
43
44  # POST /ingredients
45  # POST /ingredients.xml
46  def create
47    @ingredient = Ingredient.new(params[:ingredient])
48    respond_to do |format|
49      if @ingredient.save
50        flash[:notice] = 'Tworzenie przepisu zakończone powodzeniem.'
51        format.html { redirect_to([@recipe, @ingredient]) }
52        format.xml  { render :xml => @ingredient, :status => :created, :location => @ingredient }
53      else
54        format.html { render :action => "new" }
55        format.xml  { render :xml => @ingredient.errors, :status => :unprocessable_entity }
56      end
57    end
58  end
59
```

Nie jestem pewien, na ile wyraźnie widać to w odcieniach szarości, ale klauzula `else` została podświetlona na czerwono, co znaczy, że ta gałąź kodu nie została przetestowana. W celu pozyskania tego typu danych rcov przeprowadza bardzo zaawansowaną analizę wyjścia z interpretera Rails, jednak czasami popełnia błędy. Widać to także w naszym przykładzie: przez to, że bloki wewnętrzne `respond_to` znalazły się w jednej linii, rcov nie jest w stanie wykryć, że nie istnieje test generowania XML przez tę metodę. Dość dziwne jest również to, że linie zawierające `end` — zarówno te kończące blok, jak i te kończące metodę — zostały oznaczone jako niepokryte. Być może nie ma to po prostu znaczenia.

Naszym celem powinno być każdorazowe osiąganie wartości 100 procent pokrycia. W Rails jest to możliwe również dzięki elastyczności języka Ruby. Jeśli osiągniemy wartość 100 procent, a testy są napisane rozsądnie, możemy uznać, że jesteśmy bezpieczni podczas dodawania, naprawiania i refaktoryzacji kodu. Za pomocą pojedynczego uruchomienia testów możemy upewnić się, że nasze zmiany nie doprowadziły do powstania nowych błędów.

Jak widać na rysunku 7.1, który przedstawia stan aplikacji Zupy OnLine podczas pisania wstępnej wersji tego rozdziału, projekt jest bardzo blisko 100-procentowego pokrycia kodu testami. Do osiągnięcia tego wyniku wystarczyło automatyczne wygenerowanie kodu przez Rails oraz omówione wcześniej praktyki TDD. Jedynym poważnym brakiem jest obiekt pośredniczący, który został przedstawiony jako możliwa opcja w rozdziale 6. W swojej aktualnej postaci kod nie wywołuje bazowej klasy pośrednika, dlatego nie jest ona uruchamiana podczas testów. W tej chwili nie jest to dla nas żadnym problemem, natomiast w przypadku prawdziwej aplikacji trzeba by się zastanowić nad wyeliminowaniem nieużywanego kodu.

Większość niepokrytych przypadków to niepowodzenia różnych metod tworzących i uaktualniających obiekty, co również widać na rysunku. Najwyraźniej zapomniałem również napisać testy kontrolera dla ajaksowych formularzy edycji tagów. By przetestować niepowodzenie zapisu danych, należałoby stworzyć scenariusz testowy, w którym nie zapis nie byłby możliwy. Najprostszy sposób to próba zachowania obiektu, który nie przejdzie walidacji Rails lub złamie ograniczenia bazy danych.

Jest tylko jeden problem. W obecnej chwili `Recipe` nie posiada żadnych ograniczeń bazodanowych ani walidacji Rails. Chociaż mógłbym podać przynajmniej dwa odpowiednie przykłady obiektów (lub zrobić coś naprawdę dziwnego i podjąć próbę zapisu tytułu o długości 10 000 znaków), wydaje się, że musi istnieć prostszy sposób.

Otóż istnieje.

Testowanie za pomocą atrap

Testowanie za pomocą atrap (ang. *mock testing*) polega na wykorzystaniu „udawanych" obiektów, które zastępują te prawdziwe podczas testów automatycznych. Pierwotnie technika ta była stosowana podczas testowania systemów wymagających baz danych, połączenia z siecią lub innych zasobów, które trudno wiarygodnie skonfigurować w środowisku testowym. Obecnie stosuje się ją także w celu sprawdzenia poprawności działania programów i w celu lepszej orientacji testów.

Testy za pomocą atrap są jedną z tych dziedzin, w których każdy zestaw narzędzi wprowadza strukturę nazw odmienną od pozostałych. Zamierzam oprzeć się na konwencji postulowanej przez Martina Fowlera w artykule *Mocks Aren't Stubs* o różnicy pomiędzy obiektami *mock* a innym typem obiektów o nazwie *stub* (szczegóły w sekcji „Źródła" na końcu rozdziału). Jedne i drugie są dublerami prawdziwych obiektów w sposób analogiczny do tego, w jaki kaskaderzy zastępują aktorów podczas niebezpiecznych ujęć. *Stub* to „udawany" obiekt lub metoda, które podczas testów zwracają ustaloną wartość bez dokonywania rzeczywistych obliczeń. *Mock* (atrapa) także zwraca ustaloną wartość, jednak powinna posiadać dodatkową funkcjonalność śledzenia wywołań do obiektu oraz, co ważniejsze, powinna sprawdzać, czy wywołania te są zgodne z oczekiwaniami określonymi podczas tworzenia testu.

Jedną ze wspaniałych cech testów za pomocą atrap w Rails jest to, że narzędzia do tworzenia takich testów potrafią wykorzystać szerokie możliwości metaprogramowania w Rails do zamiany istniejących obiektów i klas w obiekty *stub* lub *mock*. Odróżnia to Rails od, na przykład, Javy, gdzie narzędzia do tworzenia atrap za pomocą interfejsów i ładowania klas produkują obiekty, które zastępują te oryginalne, lecz różnią się od nich w ryzykowny sposób.

Jako że tworzenie „udawanych" obiektów w Ruby jest bardzo proste, zyskujemy nowe możliwości testowania. Przykładem dobrym i na czasie może być stworzenie modelu ActiveRecord, który zachowuje się dokładnie tak jak każdy inny model ActiveRecord w systemie z tą różnicą, że wychwytuje próby zapisu do bazy danych i zwraca określoną wartość bez łączenia się z bazą.

FlexMock

Istnieją trzy lub cztery różne pakiety Rails, które umożliwiają przeprowadzanie testów za pomocą atrap. Możliwości pakietów są zbliżone, dlatego omówię tylko jeden z nich — FlexMock. Został on napisany przez Jima Weiricha (który jest także osobą odpowiedzialną za Rake). FlexMock jest dostępny w postaci gema Ruby, czyli instalujemy go za pomocą standardowego polecenia `gem`:

```
$ sudo[1] gem install flexmock
```

By testy mogły korzystać z FlexMock, konieczne jest dodanie linii `require` na początku każdego skryptu testowego, który będzie używał atrap (a jeśli będą one wykorzystywane często, linię warto dodać do *test_helper.rb*):

```
require 'flexmock/test_unit'
```

Pora zacząć udawanie.

Zamierzam przedstawić konkretny test, który stworzyłem w celu dotarcia do wcześniej niepokrytej części kodu, a następnie opowiedzieć, jak można rozszerzyć ten test i jak stworzyć inne obiekty *mock* i *stub*.

Zajmiemy się niepokrytą klauzulą `else` z pliku *recipes_controller.rb*:

```
def update
  @recipe = Recipe.find(params[:id])

  respond_to do |format|
    if @recipe.update_attributes(params[:recipe])
      flash[:notice] = 'Uaktualniono przepis.'
      format.html { redirect_to(@recipe) }
      format.xml  { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml  { render :xml => @recipe.errors, :status => :unprocessable_entity }
    end
  end
end
```

Niepokryta część kodu zostanie osiągnięta, jeśli `update_attributes` zakończy się niepowodzeniem. Jak już wspomniałem, stworzenie obiektu, którego nie da się zapisać, nie jest takie proste, ponieważ klasa `Recipe` nie jest w żaden sposób walidowana. Zamiast tracić czas, łamiąc sobie głowę nad tym zadaniem, możemy zmusić system do błędu zapisu. Poniższy kod należy umieścić w pliku *test/fixture/recipes_controller_test.rb*:

```
def test_should_fail_update
  flexmock(Recipe).new_instances.should_receive(:update_attributes).
  ↪at_most.once.and_return(false)
  put :update, :id => 1, :recipe => {:title => "Rosołek babci"}
  assert_template('edit')
  actual = Recipe.find(1)
  assert_not_equal("Rosołek babci ", actual.title)
  assert_equal("1", actual.servings)
end
```

[1] `sudo`, oczywiście, dotyczy tylko pewnych odmian Linuksa i Mac OS X — *przyp. tłum.*

Najważniejsza jest pierwsza linia. Przeanalizujmy ją po kawałku:

- `flexmock(Recipe)` — tworzy nowy obiekt zastępczy (tym razem jest on typu *stub*). Metoda `flexmock` pobiera wiele różnych opcji, do których przejdę za moment. W tym wypadku jest wywoływana z argumentem będącym obiektem Ruby z krwi i kości, czyli z klasą `Recipe`. Chodzi nam o osiągnięcie omówionych wcześniej funkcjonalności atrapy wokół istniejącego obiektu.
- `new_instances` — jest to metoda pośrednika obiektu-atrapy. Działa tylko wtedy, gdy obiekt, którego atrapę tworzymy, jest obiektem klasy. Po wywołaniu tej metody FlexMock zastosuje podaną specyfikację do wszystkich instancji klasy, czyli zmieni zachowanie wywołań `new`. W rezultacie każdy obiekt ActiveRecord przepisu zostanie rozszerzony o zachowanie atrapy. Oznacza to, że obiekty już istniejące i przechowywane jako dane do testów nie zostaną obdarzone tym zachowaniem, ale te same obiekty na nowo załadowane z bazy do modeli ActiveRecord — już tak. Wywołanie tej metody, podobnie jak większości innych metod we FlexMock, zwraca specjalny obiekt, który zapamiętuje różne oczekiwania, by móc stworzyć łańcuch ograniczeń taki jak w przykładzie.
- `should_receive(:update_attributes)` — tutaj zaczynamy określać zachowanie obiektów *stub*. Wywołanie tej metody informuje FlexMock o tym, że będziemy chcieli zrobić coś z wywołaniem `update_attributes`, ale nie określa jeszcze konkretnego zachowania. Metoda `should_receive` może pobrać dowolną liczbę argumentów w postaci symboli, z których każdy reprezentuje metodę, która będzie zastępowana zgodnie z tą samą specyfikacją.
- `and_return(:false)` — tą linią kończymy określanie zachowania. FlexMock zwróci wartość `false` dla każdego wywołania `update_attributes`, bez odwoływania się do bazy danych ani wykonywania żadnych innych zbędnych operacji. Metoda ta jest bardzo elastyczna. Można przekazać jej kilka wartości — wówczas będzie je zwracała jedną po drugiej podczas kolejnych wywołań. Jeśli klas będzie więcej niż wartości, zwrócone zostaną ponownie wartości z początku listy. Można jeszcze przekazać blok pobierający wszystkie argumenty dublowanej metody, na których można wykonać dowolne operacje obliczające zwracaną wartość.

Test kończy się sukcesem, ponieważ został dodany w celu zwiększenia pokrycia i koncentruje się na określonej gałęzi kodu. Polecenie `flexmock` wyprowadza z testu, po którym uruchamiany jest normalny test funkcjonalny metody kontrolera. Różnica polega na tym, że gdy metoda dochodzi do `update_attributes`, FlexMock przechwytuje wywołanie i zwraca `false` zgodnie ze specyfikacją. Kontroler interpretuje to jako niepowodzenie i wchodzi w odgałęzienie kodu związane z błędem zapisu, co pozwala sprawdzić, czy kontroler w przypadku błędu zachowuje się zgodnie z oczekiwaniami.

Przedstawione rozwiązanie jest eleganckim sposobem testowania obsługi błędów lub innych sytuacji, które trudno wprowadzić do modelu. Po dodaniu podobnych testów dla innych klauzul obsługi błędów w aplikacji pokrycie powinno zbliżyć się do 100 procent. (Powiem tylko, że osiągnięcie 100-procentowego pokrycia zajęło mi około półtorej godziny i objęło między innymi: znalezienie jednego prawdziwego błędu w kodzie, próbę ponownego wprowadzenia obiektu pośredniczącego oraz niepotrzebne zamieszanie, gdy okazało się, że posiadam dwa testy o tej samej nazwie.)

Specyfikacja obiektów i metod typu stub

FlexMock daje ogromną swobodę tworzenia różnego typu obiektów pośredniczących. Omówiony wcześniej test z użyciem atrapy korzystał z obiektu klasy, na podstawie którego tworzone były wszystkie instancje. Inna możliwość to stworzenie rzeczywistej instancji i dodanie do obiektu metod typu *stub* w następujący sposób:

```
soup = Recipe.new
flexmock(soups).should_receive(:ingredients).and_return([])
```

Nazwa metody i zwracana wartość są określane jako prosta para klucz/wartość. FlexMock oferuje następującą skróconą składnię, która może zastąpić drugą linię:

```
flexmock(soup, :ingredients => [])
```

Jest ona krótsza, ale nie można już, jak w przypadku pierwszej wersji, odczytać jej jako poprawnego zdania w języku angielskim. Można, oczywiście, podać różne pary klucz/wartość i umieścić je w różnych liniach. Na przykład:

```
flexmock(soup)
flexmock(soup).should_receive(:ingredients).and_return([])
```

Można również podać metody i wartości za pomocą bloku:

```
flexmock(soup) do |soup|
  soup.should_receive(:ingredients).and_return([])
end
```

Kolejna opcja to stworzenie poprzez przekazanie `flexmock` symbolu lub napisu całkowicie sztucznego obiektu, który nie zostanie powiązany z żadną istniejącą klasą. Na przykład:

```
flexmock("banan")
flexmock("banan").should_receive(:zrob_cos).and_return(3)
flexmock("banan", :zrob_cos => 3)
flexmock(:zrob_cos => 3)
```

Linie druga i trzecia mają dokładnie ten sam efekt. Ostania linia jest niemalże identyczna, jednak obiekt *stub* nie otrzymuje w niej własnej nazwy.

Jeśli, z jakiegoś powodu, ktoś zechce dodać metody typu *stub* lub *mock* do obiektu napisu, będzie musiał zastosować pewną sztuczkę, ponieważ domyślnie napis zostanie potraktowany jako nazwa nowej metody, tak jak w pierwszej linii przykładu. By dodać metodę do napisu lub symbolu, należy zastosować opcję `:base`, co pokazano poniżej:

```
flexmock(:base, "string_to_mock")
```

Można jeszcze jako pierwszy argument przekazać symbol `:safe`. Obiekt FlexMock w normalnym trybie działania dodaje kilka metod do przestrzeni nazw prawdziwego obiektu. Są to przede wszystkim metody `should_receive` i `new_instance`. Jeśli metody o tych nazwach już istnieją w projekcie, pojawi się problem. W trybie `safe` FlexMock nie doda tych metod. W takiej sytuacji nasze oczekiwania określamy wewnątrz bloku postaci:

```
flexmock(:safe, soup) do |mock|
  mock.should_receive(:ingredients).and_return([])
end
```

Rozwiązanie to zadziała, ponieważ obiekt pośredniczący użyty wewnątrz bloku będzie posiadał metodę FlexMock `should_receive`, natomiast nie będzie jej posiadał poza blokiem.

Wszystkie elementy łańcucha zwracają wewnętrzny obiekt oczekiwań FlexMock, a nie samą atrapę (w celu wsparcia łańcuchowego łączenia wywołań). Aby odzyskać atrapę, łańcuch należy zakończyć wywołaniem `mock`:

```
mock = flexmock(Recipe).should_receive(:save).mock
```

Specjalny mechanizm FlexMock jest odpowiedzialny za nienaganną współpracę z ActiveRecord:

```
require 'flexmock/activerecord'
mock = flexmodel(Recipe) do |mock|
  Dowolne działanie atrapy.
end
```

`flexmodel` tworzy obiekt typu *stub* z kilkoma predefiniowanymi metodami w stylu ActiveRecord: `class`, `id`, `is_a?`, `errors`, `new_record?` oraz `to_params`, które zwracają proste wersje *stub* prawdziwych metod. `id` zwraca unikalny identyfikator, `is_a?` i `class` odpowiadają prawdziwej klasie itd.

Oczekiwania atrap

Przedstawione powyżej przykłady wykorzystania FlexMock w rzeczywistości tworzą jedynie obiekty *stub*, czyli obiekty, które mogą odbierać wiadomości i zwracać wartości, ale nie posiadają żadnych wytycznych pozwalających na ocenę poprawności zachowania. FlexMock posiada szereg metod, które można łańcuchowo połączyć z deklaracją oczekiwań w celu dodania walidacji.

Istnieją trzy ogólne typy oczekiwań, które można dodać do metody w postaci *stub* w celu zmiany jej na atrapę (*mock*). Można określić przewidywaną liczbę wywołań metody za pomocą następujących modyfikatorów:

- `never` (nigdy),
- `zero_or_more_times` (zero lub więcej razy),
- `once` (raz),
- `twice` (dwa razy),
- `times(n)` (n razy).

Domyślnie test polega na sprawdzeniu, czy metoda została wywołana dokładnie oczekiwaną liczbę razy (z wyjątkiem, rzecz jasna, `zero_or_more_times`). Każdą z tych metod można poprzedzić przedrostkiem `at_least` (co najmniej) lub `at_most` (co najwyżej), na przykład:

```
should_receive(:update_attributes).at_most.once.and_return(false)
```

Modyfikatory pojawiają się po wywołaniu `should_receive` i przed wywołaniem `and_return`. Można je połączyć tak jak w przypadku `at_least.once.at_most.time(3)`.

Po dodaniu przedrostka test nadal będzie kończył się sukcesem. Wywołanie `at_most` jest tu konieczne: chociaż na obiekcie ActiveRecord utworzonym w kontrolerze wywoływane jest `update_attributes`, to jednak nie dzieje się tak w przypadku obiektu stworzonego w czwartej linii testu. Wszystkie obiekty `Recipe` otrzymują tę samą specyfikację. Test zakończy się niepowodzeniem z następującą wiadomością:

```
in mock 'flexmock(Recipe)': method "update_attributes(*args)' called incorrect
➥number of times
```

Warto zwrócić uwagę, że wiadomość nie zawiera informacji na temat tego, która z instancji `Recipe` doprowadziła do błędu. Nieco łatwiej jest zdiagnozować niepowodzenia walidacji, jeśli obiekt atrapy został utworzony w oparciu o pojedynczą instancję, a nie o klasę.

Można określić, jakie argumenty mają zostać przekazane do wywołania atrapy za pomocą metody specyfikującej `with(`*`Argument1`*`,`*`Argument2`*`...)`. Dodana do oczekiwań lista argumentów jest dopasowywana do listy argumentów przekazanej podczas wywołania. Obiekty są porównywane za pomocą `eq` — z dwoma wyjątkami. Jeśli przekażemy nazwę klasy, za poprawną wartość zostanie uznana dowolna jej instancja, jak poniżej:

```
foo.should_receive(:cos).with(String, Integer) #Poprawną wartością będzie cos("cześć", 2).
```

Jako argument można podać także wyrażenie regularne (ang. *regex*). Argumenty zostaną uznane za poprawne, jeśli zostaną dopasowane do wyrażenia (FlexMock zamieni wszystkie argumenty w napisy przed porównaniem). Jeśli ktoś naprawdę chce skorzystać z walidacji w oparciu o nazwę klasy lub wyrażenie regularne, może zrobić to za pomocą deklaracji postaci `with(eq(ClassName))`.

Ostatni typ walidacji to sprawdzenie kolejności wywołania metod. Należy użyć dekoratora `ordered` — wówczas FlexMock sprawdzi, czy kolejność wywołania metod odpowiada oczekiwanej. Następująca specyfikacja:

```
flexmock(tost) do |mock|
  mock.should_receive(:przypiecz_chlebek).ordered
  mock.should_receive(:posmaruj_maselkiem).ordered
end
```

za poprawne uzna:

```
tost.przypiecz_chlebek
tost.skocz_do_lodowki
tost.posmaruj_maselkiem
```

Obecność metody `skocz_do_lodowki`, dla której nie określono kolejności, nie wpływa na walidację.

Metoda `ordered` pobiera jeden argument, który jest symbolem reprezentującym grupę. Tą samą nazwą grupy można opisać kilka kolejnych metod. Zmienia to nieco przebieg walidacji. Metody należące do jednej grupy mogą zostać wywołane w dowolnej kolejności. Wszystkie metody określone jako wcześniejsze muszą zostać wywołane w pierwszej kolejności, a wszystkie późniejsze — w drugiej. Oto przykład:

```
flexmock(tost) do |mock|
  mock.should_receive(:przypiecz_chlebek).ordered
  mock.should_receive(:skocz_do_lodowki).ordered(:czas_uplywa)
```

```
    mock.should_receive(:spojrz_na_zegarek).ordered(:czas_uplywa)
    mock.should_receive(:pospiesz_sie).ordered(:czas_uplywa)
    mock.should_receive(:posmaruj_maselkiem).ordered
  end
```

Metody `skocz_do_lodowki`, `spojrz_na_zegarek` i `pospiesz_sie` mogą zostać wywołane w dowolnej kolejności, o ile tylko `przypiecz_chlebek` zostanie wywołane wcześniej, a `posmaruj_maselkiem` później.

Atrapy są przydatne podczas walidacji trudno osiągalnych części kodu, przydają się także wtedy, gdy testy jednostkowe mają się koncentrować na określonej części aplikacji. W konsekwencji z niebytu wyłania się nieco odmienny paradygmat testów jednostkowych.

Projektowanie w oparciu o zachowanie

Wprowadzenie walidacji w oparciu o obiekty-atrapy zmienia naturę testów jednostkowych. Tradycyjne testy jednostkowe, które przeprowadzają walidację głównie w oparciu o asercje, testują *stan* aplikacji. Testy korzystające z atrap, sprawdzające zgodność listy wywołań z określonymi wcześniej oczekiwaniami, badają *zachowanie*. Test zachowania, przynajmniej w teorii, pozwala na łatwiejsze odizolowanie zamierzonego działania aplikacji od konkretnej implementacji.

Zwolennicy odmiany testów automatycznych określanych mianem *Behavior-Driven Design* (projektowanie w oparciu o zachowanie, w skrócie BDD) postulują projektowanie testów w sposób bliższy problemowi, który ma rozwiązywać aplikacja, niż konkretnej implementacji. Starają się to uzyskać między innymi poprzez projektowanie zestawów narzędzi, które pozwalają na definiowanie testów w języku bardziej zbliżonym do naturalnego. Narzędzia BDD intensywnie wykorzystują obiekty *mock* w celu określenia dziedziny problemu oraz oddzielenia od siebie różnych testów jednostkowych. Podczas testów TDD może się okazać, że zmiana w niskopoziomowej metodzie powoduje niepowodzenie wielu testów jednostkowych. Zwolennicy BDD będą próbowali przekonać nas, że fakt, iż jedna metoda psuje kilka testów jednostkowych, dowodzi tego, że nie są one wcale testami jednostkowymi, tylko testami integracyjnymi na bardzo małą skalę. Testy TDD korzystają z atrap tylko wtedy, gdy prawdziwe obiekty są niedostępne lub ich stosowanie byłoby bardzo niewygodne. Testy BDD bardziej agresywnie korzystają z atrap w celu odizolowania testowanej metody od reszty systemu.

Opowiem teraz o RSpec, najpopularniejszym pakiecie testowym BDD dla języka Ruby. RSpec daje się łatwo zintegrować z Rails, pozwalając na przykład na osobne testowanie kontrolerów, widoków i metod pomocniczych.

Instalacja RSpec

RSpec jest dostępny zarówno w postaci gema Ruby (`gem install rspec`), jak i pluginu Rails. Jeśli ma być wykorzystywany w Rails, należy pobrać dwa pluginy: `RSpec` i `RSpec Rails`:

```
$ ruby script/plugin install -x svn://rubyforge.org/var/svn/rspec/tags/CURRENT/rspec
$ ruby script/plugin install -x svn://rubyforge.org/var/svn/rspec/tags/CURRENT/rspec_on_rails
```

Oczywiście jeśli nie chcemy instalować RSpec w postaci zewnętrznego repozytorium Subversion, pomijamy opcję `-x`.

Użytkownicy systemu Windows będą jeszcze musieli zainstalować gem o nazwie `win32console`.

Po instalacji RSpec należy uruchomić następujący generator w celu utworzenia katalogów i plików RSpec:

```
$ ruby script/generate rspec
```

Najważniejsze zadanie tego polecenia to utworzenie podkatalogu *spec* w głównym katalogu Rails. W katalogu tym jest umieszczany plik *spec_helper.rb*, odpowiadający standardowemu plikowi *test_helper.rb*. Polecenie tworzy jeszcze kilka skryptów i plików, którymi nie będziemy się teraz zajmować. Podkatalogi RSpec nie są, niestety, zgodne z konwencją nazw oczekiwaną przez Rails — trzeba je przekształcić ręcznie.

Konwencja nazw stosowana przez RSpec jest podobna do rozpoznawanych przez Rails. Poniższa tabela przedstawia tę konwencję poprzez przykładowe nazwy plików:

Typ testu	Przykładowa nazwa
Test kontrolera	*spec/controllers/recipe_controller_spec.rb*
Test pomocnika	*spec/helpers/recipes_helper_spec.rb*
Test modelu	*spec/models/recipe_spec.rb*
Test widoku	*spec/views/recipe/new_spec.rb*

Ponieważ pliki modelu, widoku, kontrolera oraz metod pomocniczych zostały już wygenerowane, będziemy musieli ręcznie stworzyć pliki RSpec. Jeśli ktoś zaczyna programowanie od zera z użyciem RSpec, ma dostęp do kilku generatorów: `rspec_controller`, `rspec_model` i `rspec_scaffold`. Ich zachowanie będzie identyczne z tym istniejących kontrolerów — wszystkie pobierają te same argumenty. Jedyna różnica polega na utworzeniu zalążków plików testowych RSpec w katalogu *spec*, a nie plików Test::Unit w katalogu *test*.

Specyfikacje RSpec można uruchomić za pomocą polecenia `rake spec`. Jeśli uruchomiony ma zostać podzbiór testów, trzeba skorzystać z bardziej szczegółowego polecenia, na przykład `rake spec:models`. Istnieją podzadania: `controllers`, `helpers`, `models`, `plugins` i `views`. Jeśli w systemie zainstalowany jest rcov, do każdego z poleceń można dodać `:rcov`, dzięki czemu wygenerowany zostanie raport o pokryciu. Należy jednak pamiętać, że chociaż RSpec posiada testy szablonów widoków, rcov nie utworzy raportu pokrycia dla plików ERB. W Rails 2.0 domyślne zadanie Rake uruchomi zarówno testy jednostkowe z katalogu *test*, jak i specyfikacje RSpec z katalogu *spec*.

Pisanie specyfikacji RSpec

Plik ze specyfikacją RSpec zawiera opis jednego lub więcej zachowań, z których każde obejmuje co najmniej jeden przykład. Sama konwencja nazewnicza mówi już coś o różnicach metodologicznych pomiędzy RSpec a Test::Unit. „Zachowanie" (ang. *behavior*) i „przykład" (ang. *example*) odnoszą się do testów od strony funkcjonalności i zamiarów, podczas gdy „test" i „asercja" są związane z implementacją. Pojedyncze zachowanie RSpec w przybliżeniu odpowiada jednej klasie Test::Unit, chociaż o wiele bardziej prawdopodobne jest napotkanie wielu zachowań w jednym pliku RSpec niż wielu klas w pojedynczym pliku Test::Unit.

Do deklarowania zachowań służy metoda `describe`, a do deklarowania przykładów — metoda `it`. Szkielet pliku ze specyfikacją wygląda tak[2]:

```
describe Foo do
  it "should not crash when I call it" do #Po polsku: "nie powinno się wywrócić, kiedy to uruchomię".
    [...] Treść testu.
  end
end
```

Jak widać, przykład jest opisywany za pomocą napisu w języku naturalnym, a nie za pomocą nazwy metody.

Wewnątrz zachowania można określić warunki początkowe oraz zasady czyszczenia danych za pomocą metod `before` i `after`. Każda z tych metod pobiera jeden z dwóch modyfikatorów. Domyślnym jest `:each`, który oznacza, że blok związany z metodą powinien zostać uruchomiony przed i po każdym przykładzie, podobnie jak metody `setup` i `teardown` w przypadku Test::Unit. Druga opcja to `:all`, która spowoduje, że blok zostanie wykonany raz dla danego zachowania — przed uruchomieniem wszystkich przykładów lub po ich zakończeniu. Można zadeklarować wiele bloków `before` i `after` z tym samym modyfikatorem. Wszystkie zostaną w odpowiedniej chwili wykonane.

Każda metoda zadeklarowana wewnątrz zachowania przy użyciu standardowej składni `def` języka Ruby jest dostępna dla wszystkich przykładów wewnątrz tego zachowania, zatem nadal możemy pisać własne metody sprawdzające wartości i własnych pomocników.

Chociaż przykłady z reguły pobierają blok, możliwe jest tymczasowe utworzenie pozbawionego bloku przykładu z samym napisem. Na przykład:

```
it "should do something that hasn't been implemented yet" #Po polsku: "powinno robić coś, co
                                                          ↪jeszcze nie zostało zaimplementowane".
```

RSpec zinterpretuje test jako zawieszony i poda liczbę takich testów obok liczby testów, które kończą się sukcesem lub niepowodzeniem.

Można posunąć się o krok dalej i zliczać przypadki, w których wiemy, że test kończy się niepowodzeniem i nie przeszkadza nam to, ale chcielibyśmy otrzymać informację, gdy test zacznie kończyć się pomyślnie. W ogólności wygląda to tak:

---

[2] Zachowania opisujemy za pomocą języka naturalnego. Można stosować język polski, jednak składnia z `it` na początku po polsku wygląda nieco gorzej, dlatego zachowana została wersja oryginalna — *przyp. tłum.*

```
it "should fix this silly bug" do #Po polsku: "powinno załatwić sprawę tego głupiego błędu".
  pending("this is Bob's problem") do #Po polsku, mniej więcej: "to broszka Roberta".
    [...] Treść testu kończącego się niepowodzeniem.
  end
end
```

W takim przypadku Rails uruchomi kod wewnątrz bloku `pending` (zawieszony). Jeśli zakończy się niepowodzeniem, test zostanie zgłoszony jako zawieszony. Jeśli się powiedzie, zgłoszone zostanie złamanie oczekiwań, co oznacza, że test działa i nie musi już być opisany jako zawieszony.

Jak tworzyć testy modelu

Na najbliższych stronach postaram się przedstawić przykłady specyfikacji RSpec oparte na napisanych już przez nas testach jednostkowych Test::Unit. Celem nie jest całkowita rezygnacja z testów Test::Unit, tylko prezentacja przykładów działania testów RSpec i różnic pomiędzy jednymi a drugimi. Zacznijmy od modeli.

Aby testy działały, niezbędne jest skopiowanie plików YAML składników i przepisów z *test/fixtures* do *spec/fixtures*. Poniższą specyfikację umieściłem w pliku *spec/models/recipe_spec.rb*:

```
require File.dirname(__FILE__) + '/../spec_helper'

describe Recipe, "basic test suite" do #Po polsku: "podstawowy zestaw testów".
  fixtures :recipes
  fixtures :ingredients

  it "should have ingredients in order" do #Po polsku: "powinien przechowywać składniki
                                           w określonej kolejności".
    subject = Recipe.find(1)
    subject.ingredients.collect { |i| i.order_of }.should == [1, 2, 3]
  end
```

Jest to niemalże bezpośredni przekład z testu jednostkowego z rozdziału 1. Blok `describes` określa zachowanie (na razie nie ma w nim żadnych bloków `before` ani `after`). Blok `it` określa konkretne oczekiwanie — w tym wypadku polega ono na tym, że napisy odpowiadające składnikom mają być zawsze uporządkowane w odpowiedniej kolejności.

Metoda `should` (oraz jej siostrzana metoda `should_not`) umożliwia testowanie stanu w RSpec. W tym wypadku została użyta z modyfikatorem ==, co oznacza, że przeprowadzona zostanie asercja. Wartość po prawej stronie to wartość oczekiwana, wartość po lewej (obiekt przekazany przez metodę `should`) jest wartością sprawdzaną.

Po metodzie `should` można umieścić także inne modyfikatory. Zwłaszcza każda wiadomość postaci `be_<cos>` jest automatycznie tłumaczona przez RSpec na predykat `<cos>?`. Jako że `nil?` jest zdefiniowane dla wszystkich obiektów, zawsze możemy testować poprzez `should be_nil` lub `should_not be_nil`. W projektach Rails można testować poprzez `should be_blank`. Tablice można testować za pomocą `should be_empty`, itd. Jeśli komuś wydaje się to czytelniejsze[3], zawsze może korzystać z przedrostka `be_a` lub `be_an`. RSpec odmienia

[3] Oczywiście podczas stosowania wyłącznie języka angielskiego — *przyp. tłum.*

także angielski czasownik *to have*, zatem `should have_key` wykorzysta predykat `has_key?`. Należy pamiętać, że tego typu sztuczki działają tylko na metodach, które posiadają w nazwie znak zapytania, chociaż nie pisze się go w RSpec.

Pierwszy test nie różni się zbytnio od oryginalnego testu jednostkowego, jednak kolejny rzuci więcej światła na różnicę pomiędzy dwiema metodologiami. W rozdziale 1. napisaliśmy mnóstwo testów sprawdzających parsowanie napisów, takich jak `"2 szklanki marchewki, kostka"`, w celu zapisania danych w obiekcie `Ingredient`. Napisaliśmy wówczas test, który upewniał się, że przepis potrafi pobrać pewną liczbę takich napisów i zamienić je na składniki. Oto jak wygląda wersja RSpec testu przepisu:

```
it "should split ingedient strings into separate lines" do #Po polsku: "powinien rozdzielić
                                                          ↪opisy składników na osobne linie".
  Ingredient.should_receive(:parse).exactly(3).times.and_return do |str, recipe, order|
    Ingredient.new(:recipe_id => recipe.id, :order_of => order, :amount => 2,
        :ingredient => order.to_s)
  end
  subject = Recipe.find(2)
  subject.ingredient_string = "2 szklanki marchewki, kostka\n\n1/2 łyżki soli\n\n1 1/3
  ↪szklanki bulionu"
  subject.ingredients.count.should == 3
  subject.ingredients.collect { |i| i.order_of }.should == [1, 2, 3]
  subject.ingredients.collect { |i| i.ingredient }.should == %w[1 2 3]
end
```

Należy przetestować dwa komponenty. Obiekt przepisu otrzymuje napis z trzema składnikami i dwiema pustymi liniami. Przepis powinien zignorować puste linie i sparsować inne.

W wersji Test::Unit tego testu wywoływany jest parser składników, a konkretne obiekty w przepisie są sprawdzane w oparciu o oczekiwany wynik pracy parsera. W wersji RSpec nie wywołujemy parsera składników, ponieważ to nie on jest tutaj poddawany testom. Zamiast tego pierwsza linia testu zamienia obiekt `Ingredient` w częściową atrapę, która zwraca udawane obiekty składników, i sprawdza, czy obiekt ten zostanie wywołany dokładnie trzy razy. Udawane obiekty składników nie są zgodne z tym, co zwróciłby parser (mają tak mało danych, jak to możliwe bez powodowania wyjątku `nil`), ale nie ma to znaczenia — poprawność działania parsera powinna zostać sprawdzona w specjalnie do tego przeznaczonym teście. Tutaj chcemy się tylko upewnić, że obiekt poradzi sobie z pustymi liniami. W testach RSpec chodzi o maksymalne skoncentrowanie się na testowanej metodzie i otoczenie jej murem, który forsować będą tylko metody *mock*.

RSpec korzysta z własnej specyfikacji obiektów-atrap, która jest zbliżona do tej z FlexMock (jeśli ktoś wolałby nadal korzystać z FlexMock lub innego narzędzia do tworzenia testów atrapowych, może w dość prosty sposób odpowiednio skonfigurować RSpec). Specyfikacja atrapowego `should_receive` jest bardzo podobna do specyfikacji walidacji atrap we FlexMock. Do wartości można dodawać `once`, `twice`, `times(n)`, `at_least` i `at_most`. Oprócz znanego nam `and_return` można korzystać z `and_raise`, które pobiera klasę wyjątku i sprawdza, czy w określonych okolicznościach rzucany jest wyjątek danego typu. Jest jeszcze `and_yield`, które działa podobnie do `end_return`, ale zamiast zwracania wartości przekazuje je do argumentu blokowego.

W omawianym teście obiekt-atrapa posiada jedną zmienną część. Ponieważ do `end_return` przekazano argument blokowy, identyfikator składnika i wartości związane z kolejnością mogą pochodzić z argumentów przesłanych przez obiekt przepisu. Z reguły lepiej jest unikać

dynamicznych atrap i upewnić się, że zwracane wartości są całkowicie statyczne, aby nie dopuścić do powstania wzajemnych zależności — w tym akurat wypadku istnieje zależność, którą powinniśmy sprawdzić. Kolejność składników nie zależy od parsera, tylko jest przekazywana jako argument obiektu `Recipe`. Gdyby kolejność składników była ustalana statycznie przez atrapę, nie byłoby możliwe sprawdzenie jej poprawności. Dlatego test pozwala na dynamiczne ustawianie składników w określonym porządku, oznaczając każdy z nich za pomocą przypisanego mu miejsca. Dzięki temu kolejność składników w przepisie może zostać sprawdzona.

Zaprezentowany test jest dobrym przykładem tego, jak RSpec pozwala na łączenie testowania zachowania z testowaniem stanu. Zachęca także do zabawy z nazwami metod, aby przykłady były możliwie przejrzyste. Gdy zacząłem pisanie testów parsera składników, postanowiłem w pełni wykorzystać tę funkcjonalność. W pliku *spec/model/ingredient_spec.rb* umieściłem następującą treść:

```
require File.dirname(__FILE__) + '/../spec_helper'

class String
  def parsing_to?(hash)
    expected = Ingredient.new(hash)
    actual = Ingredient.parse(self, Recipe.find(1), 1)
    actual == expected
  end
end

describe Ingredient, "basic test suite" do
  fixtures :ingredients, :recipes

  it "should parse a basic string" do #Po polsku: " powinien sparsować prosty napis".
    "2 szklanki marchewki, kostka".should be_parsing_to(:recipe_id => 1,
        :order_of => 1, :amount => 2, :unit => "szklanki",
        :ingredient => "marchewki", :instruction => "kostka")
  end
end
```

Pewnym ewenementem jest tutaj dodatkowa metoda, którą dołożyłem do `String` — jest to dokładnie to, przed czym z niepokojem ostrzegają co bardziej zasadniczy inżynierowie oprogramowania, kiedy słyszą, że Ruby pozwala na dodawanie dowolnych metod do istniejących klas.

Skoro jednak metoda będzie istniała tylko podczas testów, sądzę, że nie zachwiałem stabilnością programu, a bez wątpienia `"2 szklanki marchewki, kostka".should be_parsing_to`[4] to dość dobitny sposób sformułowania warunku testowego. Moim zdaniem narzuca się tu pytanie, czy sam program nie powinien mieć metody `String#parse_to_ingredient`, ale na razie powstrzymam się od odpowiedzi.

Jak pisać specyfikacje kontrolera

RSpec ma tę przewagę nad standardowymi narzędziami testowymi Rails, że umożliwia tworzenie niezależnych od siebie testów kontrolerów, widoków i pomocników. Testy kontrolera umieszcza się w *spec/controllers*. Specyfikację `RecipesController` zacząłem od przetłumaczenia

[4] *Should be parsing* można tu przetłumaczyć jako „powinno zostać sparsowane do postaci" — *przyp. tłum.*

na RSpec testu, który sprawdza, czy działa metoda `index` i czy wywołanie HTML `GET` dla metody `new` zwróci `CAPTCHA` jak w rozdziale 3. Pierwsza część to metoda `before:(each)`, która tworzy atrapę przepisów. W celu całkowitej enkapsulacji testu kontrolera i powstrzymania go od interakcji z modelami i bazą, należy użyć metod RSpec typu *stub*, które przechwycą wywołania metod klas ActiveRecord takich jak `new` czy `find` i zwrócą modele-atrapy zamiast prawdziwych modeli ActiveRecord. W tym celu musimy umieścić następujący kod w *spec/controllers/recipes_controller_spec.rb*:

```
require File.dirname(__FILE__) + '/../spec_helper'

describe RecipesController do

  before(:each) do
    @recipe = mock_model(Recipe)
    @recipe.stub!(:new_record?).and_return(false)
    Recipe.stub!(:new).and_return(@recipe)
    Recipe.stub!(:find).and_return(@recipe)
  end
```

Tworzenie obiektów i metod typu *stub* w przypadku RSpec wygląda nieco inaczej niż we FlexMock. Powyższy fragment kodu zmienia `Recipe#new` i `Recipe#find` w taki sposób, by zwracały udawaną instancję przepisu utworzoną w dwóch pierwszych liniach metody.

Pierwsza ze specyfikacji formułuje następujące wymagania wobec wywołania metody `index`:

```
it "powinno na żądanie pobrać listę przepisów"
  get "index"
  response.should be_success
  assigns[:recipes].should_not be_nil
end
```

W specyfikacji pojawiło się kilka metod RSpec zaprojektowanych specjalnie w celu testowania odpowiedzi kontrolera. Specyfikacja `should be_success` zwraca `true`, jeśli status odpowiedzi to 200, a związana z nią `response.should be_redirect` sprawdza status przekierowania. Należy zwrócić uwagę na to, że jeśli w testach nie tworzymy widoków, `should be_` ➥`success` nigdy nie zawiedzie.

Dla kontrolerów, które wykorzystują szablony, `should render_template` pobierze ścieżkę do pliku z szablonem i sprawdzi, czy załadowano odpowiedni szablon. Jeżeli kontroler zwraca tekst pozbawiony szablonu, można sprawdzić oprawność tego tekstu za pomocą specyfikacji `should have_text`, która pobiera napis lub wyrażenie regularne i weryfikuje, czy możliwe jest jego dopasowanie do tekstu zwróconego przez kontroler. Jeśli kontroler wywołuje `redirect_to`, można skorzystać z `should redirect_to`, która pobiera pełen adres URL, odpowiadającą mu lokalną ścieżkę oraz tablicę opcji, która normalnie zostałaby przesłana do `url_for`.

Ostatnia linia metody korzysta z tablicy asocjacyjnej `assigns`, która jest podobna do metody `assigns` znanej nam ze standardowych testów funkcjonalnych. Reprezentuje ona zmienne instancji stworzone przez kontroler. Mamy jeszcze dostęp do tablic `flash` i `session`, które pozwalają na przypisanie wartości zmiennym kontrolera.

Drugi test kontrolera w następujący sposób sprawdza metodę `new`:

```
it "should respond to GET new with a captcha" do #Po polsku: "powinien odpowiedzieć na GET
                                                  ↪new za pomocą captcha".
  @token = mock_model(Token)
  captcha = mock(MathCaptcha)
  MathCaptcha.should_receive(:create).with(3).and_return(captcha)
  get "new"
  assigns[:captcha].should == captcha
end
```

Jeśli porównać tę metodę z odpowiadającym jej tradycyjnym testem jednostkowym napisanym w rozdziale 1., a w rozdziale 3. rozszerzonym o `CAPTCHA`, okaże się, że różnią w istotny sposób. Przede wszystkim oryginalny test jednostkowy posiadał wiele wywołań `assert_select`, które sprawdzały widok generowany przez tę metodę. W RSpec taka weryfikacja należy do testów widoku, a w kontrolerze testujemy tylko, czy tworzy on oczekiwane zmienne i w przewidziany sposób odwołuje się do bazy danych.

W oryginalnej wersji sprawdzaliśmy jeszcze, czy obiekt `CAPTCHA` tworzy żeton. Podobnie jak w przypadku widoku, z perspektywy `CAPTCHA` takie posunięcie uważane jest za nadmiarowe lub źle umiejscowione — obiekt `CAPTCHA` powinien być testowany przez test przeznaczony specjalnie dla niego. W RSpec wystarczy upewnić się, że kontroler prosi klasę `MathCaptcha` o utworzenie nowej instancji i umieszcza ją w zmiennej instancji w celu późniejszego wykorzystania. Tworzenie jest sprawdzane przez `should_receive` w trzeciej linii przykładu, a przypisanie sprawdzane jest na końcu.

Przedstawiony poniżej test metody `update_attributes` związanej z metodą HTTP `PUT` ujawnia kolejne różnice pomiędzy specyfikacjami RSpec a testami jednostkowymi:

```
it "should respond to a PUT with an update" do #Po polsku: "powinien zaktualizować atrybuty
                                               ↪w odpowiedzi na PUT".
  @recipe.should_receive(:update_attributes).with(
      {"title" => "Rosołek babci"}).and_return(@recipe)
  put "update", :id => 1, :recipe => {:title => "Rosołek babci"}
  response.should redirect_to("http://test.host/recipes/#{@recipe.id}")
end
```

Przykład jest dość prosty. Pierwsza linia określa oczekiwania: na jedynej istniejącej atrapie przepisu ma zostać wywołana metoda `update_attributes`. Druga linia przygotowuje wywołanie, które zapoczątkuje proces uaktualniania, a trzecia sprawdza, czy nastąpiło przekierowanie w odpowiednie miejsce (warto zwrócić uwagę na serwer testowy umieszczony w URL). Ważne jest, czego nie testuje specyfikacja — nie sprawdza, czy klasa `Recipe` w odpowiedni sposób postępuje z atrybutami po wywołaniu `update_attributes` — określa jedynie zachowanie kontrolera.

Określanie zachowania widoku

Od samego początku staram się wyraźnie pokazać wyjątkową cechę RSpec: nacisk na oddzielanie metody poddawanej testom od reszty systemu. Po tym wstępie nikogo nie zdziwi fakt, że testy widoku powinny być odizolowane zarówno od związanego z nimi kontrolera, jak i od bazy danych. Jako przykładu użyję widoku *new.html.erb* odpowiadającego za generowanie formularza. Co prawda w rzeczywistości większy fragment pracy jest wykonywany

przez widok częściowy, więc tak naprawdę to jego właśnie powinniśmy testować. Załóżmy jednak, że piszemy specyfikacje na etapie kodowania i nie zdążyliśmy jeszcze dokonać refaktoryzacji, w wyniku której pojawił się formularz częściowy, zatem będziemy testować główny widok.

Formularz został obdarzony następującą logiką: wyświetla obiekt `MathCaptcha`, jeśli został on określony. Niezależnie od tego, musimy utworzyć atrapę użytkownika i atrapę przepisu, by móc rozpocząć test.

Plik specyfikacji powinien zostać zapisany jako *specs/views/recipes/new_spec.rb*. Tworzy on atrapy w następujący sposób:

```
require File.dirname(__FILE__) + '/../../spec_helper'

describe 'recipe/new' do

  before(:each) do
    @recipe = mock_model(Recipe)
    @recipe.should_receive(:title).and_return("Rosołek babci")
    @recipe.should_receive(:servings).and_return("2")
    @recipe.should_receive(:ingredient_string).and_return("marchewki")
    @recipe.should_receive(:description).and_return("opis")
    @recipe.should_receive(:directions).and_return("wskazówki")
    @recipe.should_receive(:tag_list).and_return("pyszne")
    @user = mock_model(User)
    assigns[:recipe] = @recipe
    assigns[:user] = @user
  end
```

Powinno to już wyglądać znajomo. Najpierw tworzymy udawane obiekty instancji i określamy ich parametry. Następnie obiekty te trafiają bezpośrednio do tablicy `assigns`. W odróżnieniu od testów kontrolera testy widoku z reguły nie muszą tworzyć atrapy klasy ActiveRecord działającej jako fabryka atrap. Widoki, w przeciwieństwie do kontrolerów, nie tworzą obiektów, zatem wystarczy wstawić do nich odpowiednie obiekty. Test widoku ma dostęp do tych samych tablic `assigns`, `flash` i `session` co test kontrolera. Dodatkowo ma dostęp do tablicy `params`.

Poniższy przykład określa zachowanie formularza bez obiektu `CAPTCHA`:

```
it "should display an entire form" do #Po polsku: "powinien wyświetlić cały formularz".
  render "/recipes/new"
  response.should have_tag("form") do
    with_tag "input[name *= title]"
    with_tag "input[name *= servings]"
    with_tag "textarea[name *= ingredient_string]"
    with_tag "textarea[name *= description]"
    with_tag "textarea[name *= directions]"
    with_tag "input[name *= tag_list]"
  end
end
```

Łatwo odgadnąć, że metoda `render` udaje generowanie widoku na potrzeby testu. Wyrażenia `should have_tag` i `with_tag` są jedynie synonimami naszego starego przyjaciela `assert_select`, więc możemy stosować znaną nam składnię do walidacji istnienia różnych struktur HTML w zwracanym widoku.

Oprócz obiektu odpowiedzi, które możemy badać za pomocą `should have_tag`, jest jeszcze obiekt szablonu, który może zostać wykorzystany do zamiany wywołań metod pomocniczych na metody typu *mock* lub *stub*. Warto to zrobić, ponieważ metody pomocnicze powinny być testowane osobno. W tym celu skorzystamy ze standardowej składni tworzenia atrap RSync:

```
template.stub!(:helper_method).and_return("krzemień")
template.should_receive(:helper_method).once.and_return("tłuczeń")
```

Okaże się to przydatne podczas oddzielania się od zachowań związanych z logowaniem, które w naszej aplikacji są kontrolowane przez kod w pomocniku aplikacji.

Oczywiście należy zamienić w atrapy wszystkie obiekty, które mają swoje własne testy. Walidację zachowania formularza, który posiada obiekt `CAPTCHA`, można przeprowadzić w następujący sposób:

```
it "should display captcha" do #Po polsku: "powinien wyświetlić captcha".
  @token = mock_model(Token)
  @token.should_receive(:token).and_return("żeton")
  captcha = mock(MathCaptcha)
  captcha.should_receive(:display_string).and_return("jakiś napis")
  captcha.should_receive(:token).and_return(@token)
  assigns[:captcha] = captcha
  render "/recipes/new"
  response.should have_tag("form") do
    with_tag "input[name *= captcha_value]"
    with_tag "input[name *= token]"
  end
end
```

Przykład tworzy atrapę podobną do tej z testu kontrolera, jednak tym razem na `CAPTCHA` nałożono dodatkowe walidacje — sprawdzamy, czy widok poprosi o napis do wyświetlenia i o żeton. `CAPTCHA` jest umieszczane w tablicy `assigns`, następnie tworzony jest widok (uwaga: należy wstawić wszystkie wartości przed rozpoczęciem generowania widoku, w przeciwnym razie nie będą widoczne dla testu). W tym wypadku sprawdzam tylko istnienie nowych właściwości formularza, głównie dla zwięzłości. W rzeczywistym teście warto jednak sprawdzić, czy istniejące wcześniej elementy nie zniknęły.

Testowanie pomocników

RSpec umożliwia testowanie metod pomocniczych w oderwaniu od widoków, które z nich korzystają. Testy plików pomocniczych umieszcza się w katalogu *spec/helpers*. Nazwa pliku i obiektu opisu powinna odpowiadać konkretnej testowanej metodzie. W chwili, gdy to piszę, wszystkie metody pomocnicze aplikacji Zupy OnLine znajdują się w pliku *application_*
↪helper.rb. Oto przykładowa metoda:

```
def inflect(singular, count, plural = nil)
  plural ||= singular.pluralize
  if count == 1 then singular else plural end
end
```

Test tej metody powinien znaleźć się w pliku *spec/helpers/application_helper_spec.rb* i wyglądać mniej więcej tak:

```
require File.dirname(__FILE__) = '/../spec_helper'

describe ApplicationHelper do

  it "powinien odmienić słowo" do
    inflect("banan", 3).should == "banany"
    inflect("banan", 1).should == "banan"
  end
end
```

Ponieważ argumentem `describe` jest nazwa klasy z metodami pomocniczymi, będą one dostępne w przestrzeni nazw zachowania. Oznacza to, że można je wywoływać bez żadnych przedrostków, czego przykładem jest metoda `inflect` w powyższym teście.

Z wnętrza testu pomocnika nie można pobrać obiektów kontrolera ani szablonu, co może okazać się problemem podczas testowania metody pomocniczej, która korzysta z `concat` w celu umieszczenia obiektu w szablonie. Sugeruję oddzielenie metod generujących tekst od tych, które go wstawiają, lub testowanie takich metod równocześnie z widokiem w sposób, w jaki testuje się widoki częściowe.

Niektóre z zainstalowanych przez nas pluginów Test::Unit nie potrafią poprawnie współpracować z RSpec, dlatego może być konieczne usunięcie katalogu spec przed rozpoczęciem dalszej części kodowania.

Jak uzyskać funkcjonalności RSpec bez RSpec?

RSpec posiada wiele przydatnych funkcji, do których nie mamy dostępu w Test::Unit. Jednak przejście na RSpec nie jest takie proste, a jednoczesne korzystanie z obu tych środowisk nie jest zalecane (chociaż najnowsza wersja RSpec pozwala na uruchamianie testów Test::Unit wewnątrz RSpec). Trudno zmusić narzędzia takie jak rcov, Rake czy Autotest do uruchamiania obu zestawów testów (to także zostało poprawione w RSpec 1.1). O wiele ważniejsze jest to, że członkom zespołów programistycznych sprawia problem zapamiętanie, kiedy korzystać z których testów, jeśli system pozwala na stosowaniu obu wersji.

Istnieje jednak kilka pluginów i narzędzi, które udostępniają podobne lub takie same funkcjonalności standardowym testom Rails. Ten podrozdział krótko przedstawia kilka z nich.

Testowanie widoków

Są co najmniej dwa różne pakiety pozwalające rozszerzyć funkcjonalności testów automatycznych Rails w taki sposób, by móc tworzyć osobne testy kontrolerów i widoków. Dobra wiadomość jest taka, że jeden z nich już zainstalowaliśmy — pakiet ZenTest, który umożliwił nam korzystanie z Autotest w rozdziale 4. ZenTest posiada jeszcze inne oblicze o nazwie Test::Rails. Jego celem jest rozszerzenie istniejących testów Rails o testy przeznaczone wyłącznie dla kontrolera i wyłącznie dla widoku. Wydaje się to dobrym przybliżeniem funkcjonalności RSpec.

Test::Rails

Jak już wspomniałem, osoby, które programują razem z książką, zainstalowały już ZenTest. Jeśli ktoś jeszcze tego nie zrobił, może zrobić to za pomocą `gem install ZenTest`.

Do korzystania z Test::Rails nie trzeba instalować nic więcej, ale konieczne jest wprowadzenie pewnych zmian w istniejących plikach, począwszy od *test_helper.rb*. Pierwsze jego linie należy zmienić w następujący sposób:

```
ENV["RAILS_ENV"] = "test"
require File.expand_path(File.dirname(__FILE__) + "/../config/environment")
require 'test/rails'
require 'test_help'
class Test::Rails::TestCase
```

Dodajemy `require 'test/rails'` w postaci nowej linii i zmieniamy nazwę klasy z `Test::Unit::TestCase` na `Test::Rails::TestCase`. Należy także we wszystkich klasach testowych zmienić nazwę modułu-rodzica z `Test::Unit` na `Test::Rails` (nazwy klas pozostają bez zmian).

Następnie na końcu pliku *Rakefile* umieszczamy linię:

```
require 'test/rails/rake_tasks'
```

Dzięki temu zadania Rake będą uruchamiały nowe testy stworzone przez Test::Rails. Nazywają się one `rake test:controllers` i `rake test:views`.

Po wprowadzeniu wszystkich tych zmian w końcu możemy pisać testy przeznaczone wyłącznie dla kontrolera. Powinny się one znaleźć *test/controllers* i zostać nazwane zgodnie ze standardową konwencją. Klasa testująca `RecipesController` powinna otrzymać nazwę `RecipesControllerTest` i trafić do pliku *recipes_controller_test.rb*. Oto przykładowy test:

```
require 'test/test_helper'
class RecipesControllerTest < Test::Rails::ControllerTestCase
  def test_should_get_an_index
    get :index
    assert_response :success
    assert_not_nil assigns(:recipes)
  end
end
```

Różnica pomiędzy testem kontrolera a standardowym testem funkcjonalnym polega na tym, że kontroler nie stara się wygenerować widoku.

Testy widoków Test::Rails działają nieco inaczej. Testy wszystkich widoków danego kontrolera trafiają do jednego pliku. Plik jest umieszczany w *test/views*, a nazwa jest oparta na nazwie kontrolera. Testy widoków `RecipesController` znajdują się w pliku *test/views/recipes_view_test.rb*:

```
require 'test/test_helper'

class RecipesViewTest < Test::Rails::ViewTestCase
  fixtures :recipes, :users

  def test_new
    assigns[:recipe] = Recipe.find(1)
```

```
    assigns[:users] = User.find(1)
    render
    assert_form("/recipes/1") do
      assert_input(:text, "recipe[title]")
      assert_input(:text, "recipe[servings]")
      assert_textarea("recipe[ingredient_string]")
      [...] I tak dalej.
    end
  end
end
```

Wygląda to podobnie do istniejących standardowych testów Rails. Metoda `render` generuje widok, próbując odgadnąć jego nazwę na podstawie nazwy testu; jest nawet w stanie wywnioskować na podstawie nazwy w stylu `test_new_with_captcha`, że chodzi o szablon `new`. Wewnątrz testu widoku `Test::Rails` definiuje asercje, które są skrótami do często stosowanych wersji `assert_select`. Dotyczy to przede wszystkim wyszukiwania konkretnych elementów formularza, co można zobaczyć we wcześniejszym przykładzie.

view_test

Inny mechanizm testowania widoków zapewnia plugin o nazwie `view_test`, który instalujemy w następujący sposób:

```
$ ruby script/plugin install http://continuous.rubyforge.org/svn/tags/view_test-0.10.0
```

Warto pilnować numeru wersji — 0.10 jest aktualna w chwili pisania tego tekstu, ale nowe wersje pojawiają się dość szybko. By uruchomić `view_test` potrzebne są jeszcze dwa gemy: `mocha` (kolejne narzędzie tworzące atrapy) oraz `metaid`, które dostarcza pewnych skrótów związanych z metaprogramowaniem.

`view_test` jest bardzo przyjaznym narzędziem, które pozwala stopniowo, po jednym, przekształcać testy funkcjonalne. Do każdego z nich, przed wywołaniem kontrolera, można wstawić metodę `stub_render`, która powstrzyma generowanie widoku. Zamiast niej można użyć metody `expect_render`, za pomocą której określa się pewne oczekiwania. Metoda ta pobiera tablicę asocjacyjną podobną do tej z `url_for` i sprawdza, czy pobrany szablon jest zgodny z oczekiwanym.

Testy widoku znajdują się w *test/views* i korzystają z odmiennej konwencji nazewniczej. Jeden plik z testami odpowiada jednemu szablonowi. Na przykład testy szablonu *new.html.erb* kontrolera przepisu powinny zostać umieszczone w pliku *test/views/recipes/new.html.erb_test.rb*. Wewnątrz testu widoku można korzystać z atrapy metod pomocniczych, korzystając z `expect_helper`, którą można stosować w połączeniu z dekoratorami podobnymi do omawianych wcześniej:

```
expect_helper(:my_helper).with("fred").returns(100)
```

Charakterystyczną cechą `view_test` jest to, że każdy widok częściowy wywoływany przez widok poddawany testom musi zostać zamieniony w atrapę. Podczas testów widoki częściowe nie są dostępne dla widoku-rodzica. Muszą zostać przetestowane przez specjalnie dla nich przeznaczoną klasę testową.

Plugin `view_test` istnieje od niedawna i nadal jest aktywnie rozwijany. Najświeższą wersję można znaleźć na stronie *http://www.continuousthinking.com*.

Bardziej naturalna składnia testowania

Jeśli komuś spodobała się oparta na języku naturalnym (angielskim) składnia RSpec, istnieją dwa pakiety RubyGem i jeden plugin, które pozwolą na stosowanie podobnej konwencji w Test::Unit.

Pierwszy gem nazywa się Behaviors. Instalujemy go za pomocą `gem install behaviors`. Zadanie pakietu jest proste — pozwala zmienić test postaci:

```
test should_do_something
  [...] Treść testu.
end
```

na:

```
should "do something" do
  [...] Treść testu.
end
```

By móc korzystać z tej funkcjonalności w przypadkach testowych, należy dodać następującą linię nad definicjami klas z testami:

```
require 'behaviors'
```

Do samych klas testowych należy dopisać linię:

```
extend 'behaviors'
```

Dodatkowo gem daje możliwość zawieszania testów, zbliżoną do oferowanej przez RSpec. Jeśli po `should` nie pojawi się blok, podczas przeprowadzania testów test zostanie oznaczony jako niezaimplementowany.

Drugi gem nazywa się Dust (`gem install dust`). Pozwala on tworzyć testy w następujący sposób:

```
unit_tests do
  test "should do something" do #Po polsku: "powinien coś robić".
  [...] Treść testu.
  end
end
```

Pojedynczy blok może zawierać wiele testów. Testy funkcjonalne powinny znaleźć się wewnątrz bloku `functional_tests`, a nie `unit_tests`.

Poza samą konwencją nazw ani Behaviors, ani Dust nie zmieniają funkcjonalności Test::Unit.

Istnieje jeszcze ambitny plugin do testowania o nazwie Shoulda, który dodajemy do projektu w następujący sposób[5]:

```
$ svn export https://svn.thoughtbot.com/plugins/shoulda/tags/rel-3.0.4 vendor/plugins/shoulda
```

[5] Inny sposób instalacji, podany na *http://www.thoughtbot.com/projects/shoulda*, to `script/plugin install git://github.com/thoughtbot/shoulda.git` — *przyp. tłum.*

Plugin pozwala podzielić testy na konteksty i dopiero potem na osobne testy. Kontekst odpowiada zachowaniu RSpec. Testy wewnątrz jednego kontekstu mogą posiadać wspólny początkowy blok konfiguracyjny. Testy Shoulda nie zakłócają działania napisanych wcześniej testów — mogą być wykonywane razem z nimi. Na przykład w *test/unit/recipe_test.rb* można umieścić następujący test:

```
context "a recipe" do #Po polsku: "przepis".
  setup do
    @recipe = Recipe.find(:first)
  end

  should "have an ingredient string" #Po polsku: (powinien) "posiadać napis reprezentujący składnik".
    assert_not_nil @recipe.ingredient_string
  end
end
```

Same testy definiuje się za pomocą metody `should`. Powyższy przykład zostanie opisany na wyjściu z testu jako `test: with a recipe should have an ingredient string`. Wszystkie testy z tego samego kontekstu posiadają wspólną metodę `setup`, która jest wywoływana oprócz zwykłej metody `setup` definiowanej przez Test::Unit.

Plugin Shoulda dostarcza użytecznych skrótów, takich jak bardzo skomplikowana metoda `should_be_restful`, która wykonuje 40 różnych testów sprawdzających standardowe zachowania REST. Plugin definiuje również pewną liczbę testów makro dla walidacji ActiveRecord, a także kilka dodatkowych asercji związanych z testowaniem list.

Lepsze dane do testów

Wiemy już, że RSpec pozwala na określenie odmiennych ustawień obiektów *mock* dla różnych zachowań oraz na wykorzystanie specjalnej składni atrap do definiowania danych. Te funkcjonalności mogą się jednak okazać kłopotliwe podczas tradycyjnych testów ze względu na ograniczenia sposobu obsługi danych do testów przez Rails. Składnia danych testowych jest dość dziwaczna, a sama ich natura powoduje, że trudno jest stosować inne grupy danych w różnych testach.

Można obejść ten problem stosując dwa pluginy: FixtureScenario oraz FixtureScenarioBuilder. Instalujemy je w następujący sposób:

```
$ script/plugin install http://fixture-scenarios.googlecode.com/svn/trunk/fixture_scenarios
$ script/plugin install svn://errtheblog.com/svn/plugins/fixture_scenarios_builder
```

FixtureScenario umożliwia swobodne tworzenie podkatalogów w katalogu *test/fixtures*. Każdy z podkatalogów może zawierać jeden lub więcej plików YAML stosujących standardową składnię definiowania danych do testów Rails. Do dowolnej klasy testowej można wgrać naraz wszystkie pliki YAML znajdujące się w jednym katalogu za pomocą polecenia:

```
scenario :<nazwa_katalogu>
```

FixtureScenarioBuilder pozwala na pominięcie pliku YAML i określenie danych w języku Ruby przy wykorzystaniu normalnych modeli ActiveRecord. Należy stworzyć plik *test/fixtures/*
↪scenarios.rb. To w nim generuje się dane dla określonych scenariuszy testowych, które plugin przekształci do postaci plików YAML. Na przykład:

```
scenario :my_favourite_recipe do
  Recipe.create! :title => "rosół", :ingredient_string => "2 szklanki marchewki"
end
```

Po wykonaniu testów i wgraniu scenariusza pojawi się katalog reprezentujący scenariusz *my_favourite_recipe*, w którym znajdą się odpowiednie pliki YAML.

Testowanie pomocników

Pomocnicy Rails często przypominają zagracony strych, który wypełnia się rzeczami zbyt dziwnymi lub brzydkimi, by mogły trafić do głównej części programu. Jako że nie istnieją oczywiste sposoby ich testowania, z reguły można w nich znaleźć kruchy i trudny do utrzymania kod.

W rzeczywistości testowanie pomocników nie jest takie trudne. Należy przygotować środowisko zawierające wszystkie standardowe zmienne i metody, których istnienia oczekuje pomocnik. W wyniku tej czynności otrzymamy abstrakcyjną klasę, którą można umieścić w *test/helper_test_class.rb* i wykorzystać jako rodzica wszystkich testów pomocników:

```
require File.dirname(__FILE__) + '/test_helper'
class HelperTestClass < Test::Unit::TestCase

  include ActionView::Helpers::CaptureHelper
  include ActionView::Helpers::DateHelper
  include ActionView::Helpers::FormHelper
  include ActionView::Helpers::NumberHelper
  include ActionView::Helpers::RecordIdentificationsHelper
  include ActionView::Helpers::RecordTagHelper
  include ActionView::Helpers::TagHelper
  include ActionView::Helpers::TextHelper
  include ActionView::Helpers::UrlHelper

attr_accessor :text
```

Ta część definicji klasy załącza wszystkie klasy pomocnicze, które prawdopodobnie będą często wykorzystywane. Nie jest to pełna lista, zatem w szczególnych przypadkach może być konieczne dopisanie dodatkowej klasy do listy.

Do definicji tej samej klasy dodajemy teraz następujący kod:

```
class_inheritable_accessor :controller_class

def self.set_controller_class(controller_class)
  self.controller_class = controller_class
end
```

Pozwoli to konkretnej, zdefiniowanej później, klasie pomocniczej na ustawienie klasy kontrolera za pomocą deklaracji na poziomie klasy, a nie poprzez nadpisywanie metody `setup` czy jeszcze inną dziwną czynność. Do `HelperTestClass` musimy dodać jeszcze to:

```
def setup
  @text = []
  return unless controller_class
```

```
    @controller = controller_class.new
    request = ActionController::TestRequest.new
    @controller.send(:params=, {})
    @controller.send(:request=, request)
    @controller.send(:initialize_current_url)
  end

  def teardown
    @text = []
  end
```

Metoda `setup` deklaruje obiekt kontrolera należący do wcześniej zadeklarowanej klasy kontrolera, a następnie tworzy żądanie testowe, które jest wysyłane do kontrolera. Dzięki temu wszystkie elementy potrzebne pomocnikowi staną się dla niego widoczne. Zmienna `@text` i odpowiadający jej atrybut zostały wprowadzone, aby umożliwić testowanie metod pomocniczych pobierających wejście w postaci bloku, a także pomocników, którzy wywołują metodę pomocniczą `concat`. By wszystko działało, definicję `HelperTestClass` trzeba zakończyć za pomocą:

```
    def _erbout
      @text
    end
    def test_text
      assert_equal([], @text)
    end

  end
```

W normalnych okolicznościach wejście blokowe powoduje wywołanie ukrytego atrybutu `_erbout`. Jest to część procesu konwersji zawartości ERB przekazanego bloku do postaci HTML. Jednak test pomocnika nie bierze udziału w procesie przetwarzania ERB, zatem można oszukać test za pomocą lokalnej tablicy, w której można umieścić tekst, co pozwoli testom na walidację wyjścia ERB z metody. Podczas testowania metody blokowej należy w bloku jawnie dodać tekst do `_erbout`, ponieważ nie jest uruchamiana rzeczywista obsługa ERB.

Poniżej zamieszczam przykład, w którym rozwiązanie to zostało wykorzystane do testowania metod `inflect` i `span_for` z pomocnika aplikacji (inny przykład testowania pomocników zostanie przedstawiony w następnym rozdziale). Plik powinien zostać zapisany jako *test/units/*
↪application_helper_test.rb[6]:

```
require File.dirname(__FILE__) + '/../test_helper'
require File.dirname(__FILE__) + '/../helper_test_class'

class ApplicationHelperTest < HelperTestClass
  fixtures :recipes

  set_controller_class RecipesController

  def test_inflect
    assert_equal("banany", inflect("banan", 2))
    assert_equal("banan", inflect("banany", 1))
  end
```

[6] Autor umieścił plik dopiero w kodzie w rozdziale 8. — *przyp. tłum.*

W ramach konfiguracji musimy załączyć utworzony przed chwilą plik *helper_test_class.rb*. Klasa musi dziedziczyć z klasy `HelperTestClass`. Deklarujemy (dość przypadkowo), że kontrolerem odpowiadającym testowi jest `RecipeController`. Następujący po tych ustawieniach test metody `inflect` jest zwykłym testem jednostkowym. Do tego samego pliku dodajmy jeszcze:

```
  def test_span_for
    span_for(Recipe.find(1)) do
      text << "banan"
    end
    assert_equal("<span class=\"recipe\" id=\"recipe_1\">banan</span>", text[-1])
  end

end
```

Test metody `span_for` pokazuje, jak testować `concat` i pomocników blokowych. Każdy fragment tekstu z wnętrza bloku, który ma się pojawić na wyjściu, musi w jawny sposób zostać włożony do tablicy z tekstem. Po wywołaniu pomocnika można sprawdzić zawartość tablicy, która zawiera zarówno tekst dodany do strumienia w pomocniku za pomocą `concat`, jak i wszystko, co zostało dodane wewnątrz bloku.

Źródła

Jednym z najlepszych wczesnych opisów tworzenia testów automatycznych jest tekst Kenta Becka *JUnit Test Infected: Programmers Love Writing Tests* („Skażeni testami JUnit: programiści kochają pisanie testów"), dotyczący pisania testów JUnit, dostępny na stronie *http://junit.sourceforge.net/doc/testinfected/testing.htm*. Również wczesne książki tego autora na temat programowania ekstremalnego (XP) dobrze opisują proces testowania.

Strona domowa rcov to *http://eigenclass.org/hiki.rb?rcov*. Można tam znaleźć dokumentację i informacje o dodatkowych opcjach. Adres FlexMock to *http://flexmock.rubyforge.org*. Drugi z pluginów o zbliżonym zakresie funkcjonalności, Mocha, jest opisany na stronie *http://mocha.rubyforge.org*. Być może w chwili czytania tego tekstu Mocha jest już w pełni zintegrowane z RSpec.

Stroną domową RSpec jest *http://rspec.rubyforge.org*, który zawiera także sporą ilość dokumentacji na temat korzystania z BDD. Klasyczny tekst Martina Fowlera na temat testów za pomocą atrap znajduje się pod adresem *http://martinfowler.com/articles/mocksArentStubs.html*.

Więcej informacji na temat Shoulda można znaleźć na stronie *http://thoughtbot.com/projects/*
↪*shoulda*. Poznałem ten plugin lepiej na końcowym etapie pisania książki i bardzo go polubiłem.

Istnieje jeszcze kilka innych narzędzi do testowania stworzonych przez grupę doradczą określającą się mianem Ruby Sadists. Narzędzie o nazwie flog (*http://ruby.sadi.st/Flog.html*) bada złożoność kodu i odnajduje najbardziej skomplikowane metody, które są kandydatami do refaktoryzacji. Inne narzędzie, heckle, (*http://ruby.sadi.st/Heckle.html*) przeprowadza testy mutacyjne: zmienia losowo fragment kodu i sprawdza, czy powoduje to niepowodzenie któregoś z testów.

Jay Fields prowadzi blog *http://blog.jayfields.com*, na którym często pojawiają się ciekawe wskazówki na temat testowania w Rails. Niektóre z pomysłów związanych z testowaniem metod pomocniczych zaczerpnąłem z wspominanej już książki Chada Fowlera *Rails. Przepisy*.

Podsumowanie

Programowanie sterowane testami jest bardzo ważną częścią procesu wytwarzania oprogramowania, a Rails wspiera je wyjątkowo mocno. Można jeszcze uprościć tworzenie testów przy użyciu dodatkowych narzędzi. rcov służy do obliczania ilości kodu, która została pokryta testami. Duży stopień pokrycia jest warunkiem koniecznym, ale nie wystarczającym, posiadania dobrych testów.

Obiekty-atrapy są standardowym mechanizmem symulowania zachowań obiektów, które trudno utworzyć wewnątrz testu jednostkowego. FlexMock to zestaw narzędzi do tworzenia atrap w języku Ruby pozwalający zwiększyć zasięg testów.

RSpec to narzędzie posiadające istotną przewagę nad standardowymi testami jednostkowymi i funkcjonalnymi. Pozwala ono na osobne testy kontrolerów, widoków i pomocników, które umożliwiają pełniejsze testowanie zachowania programu. Konwencja nazewnicza i struktura testów ułatwiają czytanie i rozumienie testów także osobom spoza zespołu programistycznego.

Możliwe jest odtworzenie niektórych zdolności RSpec bez konieczności przenoszenia całości testów do tego środowiska. Narzędzia Test::Rails i `view_test` pozwalają w lepszy sposób testować widoki. Pluginy takie jak Dust czy Shoulda naśladują składnię RSpec. FixtureScenario umożliwia definiowanie różnych danych testowych dla testów różnych części systemu. Można testować również metody pomocnicze.