

Testowo Zorientowane Metodyki Wytwarzania Oprogramowania

Marcin Baliński

25 lutego 2011

Spis treści

1	Wstęp	2
1.1	Problemy napotykane w procesie wytwarzania oprogramowania	2
1.2	Testowanie oprogramowania	3
1.2.1	Wady manualnego testowania	3
1.2.2	Testy automatyczne	4
2	Test Driven Development	6
2.1	Czym jest TDD?	6
2.1.1	Główne zasady TDD	7
2.1.2	Przykładowe iteracja TDD	8
2.1.3	Zalety TDD	10
2.2	Narzędzia wspierające TDD dostępne dla języka Ruby	11
2.2.1	Test::Unit	11
2.2.2	RSpec	14
3	Behavior Driven Development	17
3.1	Czym jest BDD?	17
3.1.1	BDD jako narzędzie dokumentacji. Funkcjonalności i scenariusze	17
3.1.2	BDD jako narzędzie testowania	18
3.1.3	Testy akceptacyjne a BDD	22
3.2	Narzędzia BDD dostępne dla języka Ruby	22
4	Studium przypadku: Dynamicznie generowany panel administracyjny	23
4.1	Założenia projektu	23
4.2	Proces implementacji	23
4.3	Wnioski	23

Rozdział 1

Wstęp

1.1 Problemy napotymane 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 opisanie 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 ludzi odpowiedzialnych za prowadzenie 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 powoduje awarię w kilku innych częściach systemu.

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

Te i wiele innych problemów sprawiają, że koniecznym staje 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

Testy 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 do 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 przeglądarce¹

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 automatyzację 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

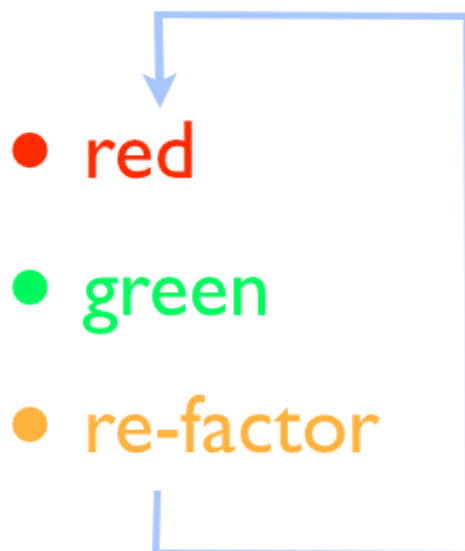
¹w tym przypadku szczegóły działania narzędzi testujących są inne, interfejs użytkownika jest bowiem najczęściej zdefiniowany przez znaczniki HTML.

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 cykli, które zobrazować można diagramem:



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 pisze 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 kodu. 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

Zaczynij 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.

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:

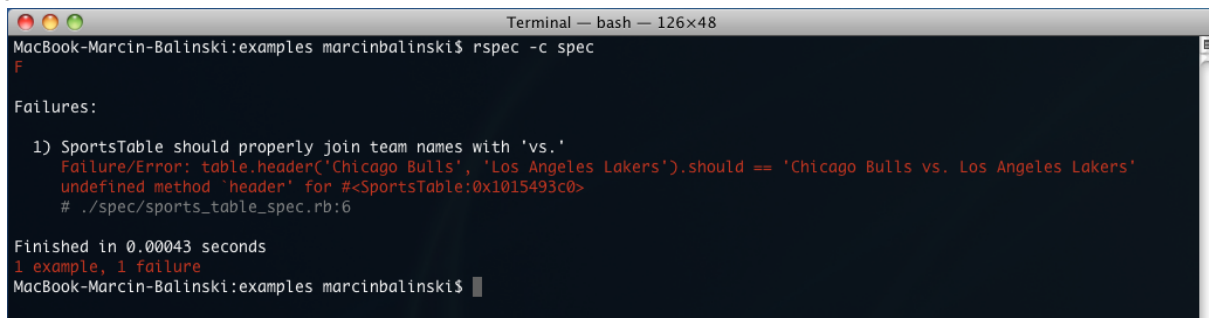
```
require 'sports_table'

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

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 następująco:

A terminal window titled "Terminal — bash — 126x48" showing the execution of RSpec tests. The command "rspec -c spec" is entered. The output shows a failure: "1) SportsTable should properly join team names with 'vs.'". The error message is "Failure/Error: table.header('Chicago Bulls', 'Los Angeles Lakers').should == 'Chicago Bulls vs. Los Angeles Lakers' undefined method 'header' for #<SportsTable:0x1015493c0>". The test finished in 0.00043 seconds with 1 example and 1 failure.

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

Failures:

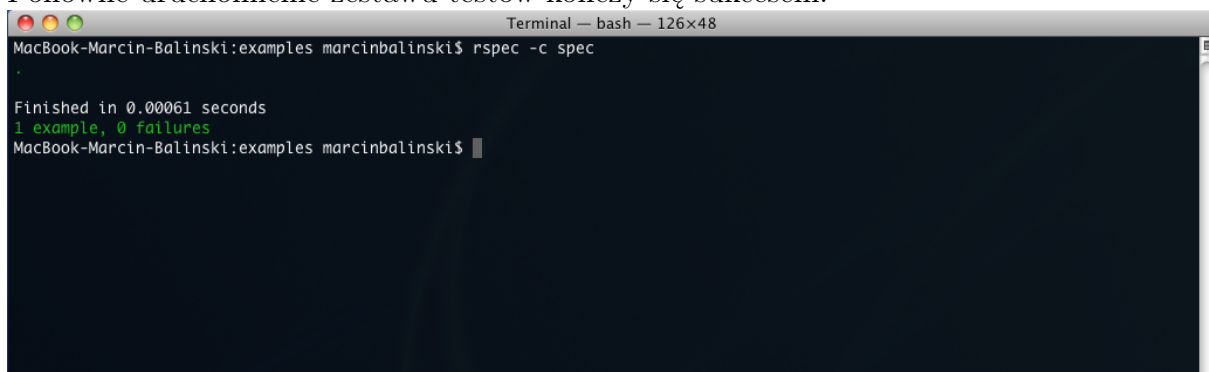
  1) SportsTable should properly join team names with 'vs.'
     Failure/Error: table.header('Chicago Bulls', 'Los Angeles Lakers').should == 'Chicago Bulls vs. Los Angeles Lakers'
     undefined method 'header' for #<SportsTable:0x1015493c0>
     # ./spec/sports_table_spec.rb:6

Finished in 0.00043 seconds
1 example, 1 failure
MacBook-Marcin-Balinski:examples marcinbalinski$
```

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:

A terminal window titled "Terminal — bash — 126x48" showing the execution of RSpec tests. The command "rspec -c spec" is entered. The output shows success: "Finished in 0.00061 seconds", "1 example, 0 failures".

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

Finished in 0.00061 seconds
1 example, 0 failures
MacBook-Marcin-Balinski:examples marcinbalinski$
```

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 wyniki 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

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:

```
require 'test/unit'

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

  def test_subtracting
    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

```

metoda `assert` nie jest jedynym typem asercji, z którego możemy korzystać, istnieje ich dużo więcej a najbardziej przydatne z nich to:

— 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, actual)` 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 nie-pustą 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_subtracting
    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` oraz `test/test_suite3.rb`

2.2.2 RSpec

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:

```
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
  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.

Oczekiwanie

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 pustą tablicą
```



```
["kot", "pies"].should include("małpa") # fałsz, testowana tablica nie zawiera
[].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 matematycznego, 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 trigger"
```

```

    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 imitację 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 w prawdzie 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 trigger"
  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 matematycznego, 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, a nietechnicznymi 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 ponieważ, jak sama nazwa wskazuje, BDD kładzie nacisk na zdefiniowanie i zrozumienie zachowania aplikacji a szczegóły implementacji są tutaj mniej istotne.

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 wyżej wymienionych czynności to pojedyncza funkcjonalność, która w zgodzie z filozofią BDD powinna zostać opisana zestawem scenariuszy.

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 zdefi-

niowane 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

Wtedy powinienem zobaczyć wiadomość "Błąd: Niepoprawna wartość. To pole jest

Oraz pole "Przychody netto ze sprzedaży produktów" 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 techniczni 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

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). 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  
  # 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`. Przekaze 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
    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 właściwie 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

Rozdział 4

Studium przypadku: Dynamicznie generowany panel administracyjny

4.1 Założenia projektu

4.2 Proces implementacji

4.3 Wnioski