

Testowo Zorientowane Metodyki Wytwarzania Oprogramowania

Marcin Baliński

9 stycznia 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	11
3	Behavior Driven Development	12
3.1	Czym jest BDD?	12
3.2	Narzędzia BDD dostępne dla języka Ruby	12
4	Studium przypadku: Dynamicznie generowany panel administracyjny	13
4.1	Założenia projektu	13
4.2	Proces implementacji	13
4.3	Wnioski	13

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. Jej 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 konkretne problemy, jakie 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 go 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 oczekiwanymi. 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

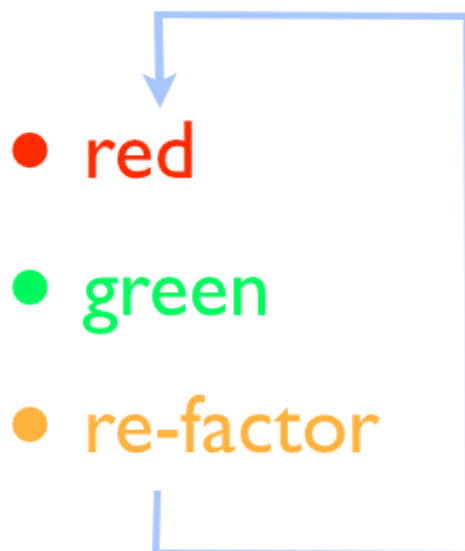
¹w tym przypadku szczegóły działania narzędzi testujących są inne, interfejs użytkownika jest bowiem najczęściej zdefiniowany przez znaczniki HTML.

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

Zaczynaj 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, że 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:

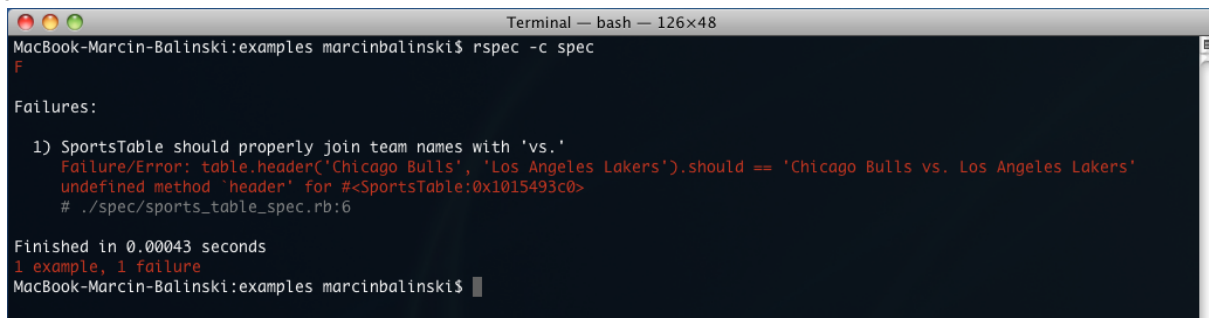
```
require 'sports_table'

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 prompt is "MacBook-Marcin-Balinski:examples marcinbalinski\$". The command "rspec -c spec" is entered, followed by a red "F" indicating failure. The output shows "Failures:" followed by a list item "1) SportsTable should properly join team names with 'vs.'" with a red error message: "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". It then shows "Finished in 0.00043 seconds" and "1 example, 1 failure". The prompt returns to "MacBook-Marcin-Balinski:examples marcinbalinski\$".

```
Terminal — bash — 126x48
MacBook-Marcin-Balinski:examples marcinbalinski$ rspec -c spec
F

Failures:

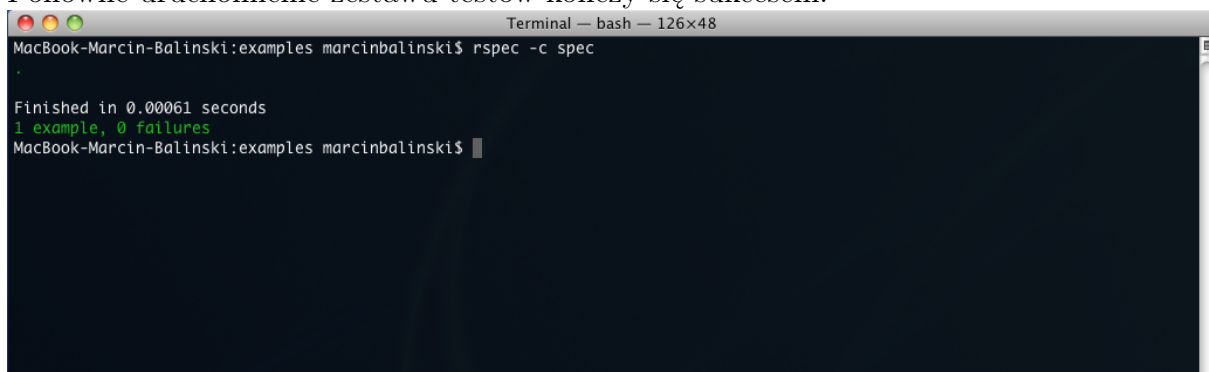
  1) SportsTable should properly join team names with 'vs.'
     Failure/Error: table.header('Chicago Bulls', 'Los Angeles Lakers').should == 'Chicago Bulls vs. Los Angeles Lakers'
     undefined method 'header' for #<SportsTable:0x1015493c0>
     # ./spec/sports_table_spec.rb:6

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 prompt is "MacBook-Marcin-Balinski:examples marcinbalinski\$". The command "rspec -c spec" is entered, followed by a green "." indicating success. The output shows "Finished in 0.00061 seconds" and "1 example, 0 failures". The prompt returns to "MacBook-Marcin-Balinski:examples marcinbalinski\$".

```
Terminal — bash — 126x48
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

2.2.2 RSpec

Rozdział 3

Behavior Driven Development

3.1 Czym jest BDD?

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