System aukcyjny w Ruby on Rails

Paweł Placzyński

26 listopada 2011

Spis treści

Rozdział 1

Wstęp

1.1 Problematyka i zakres pracy

1.1.1 Problematyka

Praca dotyczy zagadnień inżynierii oprogramowania. Zasadniczym problemem pracy jest zaprojektowanie oraz implementacja systemu aukcyjnego przy użyciu aplikacji szkieletowej Ruby on Rails.

1.1.2 Zakres pracy

Praca obejmuje następujące zagadnienia:

1. prezentacja języka Ruby oraz aplikacji szkieletowej Ruby on Rails;
2. zaprojektowanie systemu aukcyjnego;
3. implementacja wyżej wymienionego systemu aukcyjnego przy użyciu aplikacji szkieletowej Ruby on Rails;
4. przedstawienie przebiegu pracy nad projektem przy zastosowaniu się do zasad metodologii Scrum;
5. prezentacja przykładowego procesu wdrożenia aplikacji;
6. stworzenie dokumentacji wykonanego projektu;

1.2 Cele i problematyka pracy

1.2.1 Cele

Przybliżenie metodologii Scrum

Jednym z podstawowych celów pracy inżynierskiej jest przedstawienie (prezentacja) wykorzystanych w niej technologii, narzędzi, metod itp. Ze względu na użycie podczas pisania mojej pracy metodologii Scrum [1] (używanej powszechnie w małych i średnich firmach – nie tylko programistycznych) zamierzam opisać tę metodologię i zaprezentować w prosty sposób jej przebieg.

Przybliżenie (popularyzacja) technologii Ruby on Rails

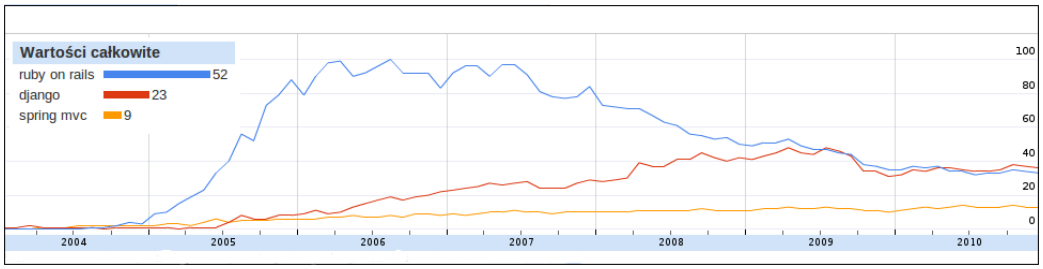
Ruby on Rails (często nazywany RoR lub po prostu Rails) to framework open source do szybkiego tworzenia aplikacji webowych stworzony głównie przez duńskiego programistę Davida Heinemeiera Hanssona w ramach pracy nad oprogramowaniem Basecamp[1]. Rails to w pełni wyposażone środowisko do tworzenia aplikacji internetowych opartych o bazy danych zgodnie ze wzorcem MVC (Model-View-Controller). Ruby on Rails daje programiście środowisko w pełni oparte o język programowania Ruby – od Ajax'a dostępnego w widokach (View), do zapytania i odpowiedzi w kontrolerach i logice biznesowej modeli.

Tuż po pojawieniu się Ruby on Rails na forum publicznym okrzyknięto go sensacyjnym. Tim O'Reilly, Założyciel O'Reilly Media mówił[2] „Ruby on Rails jest przełomem w dziedzinie programowania aplikacji internetowych. Potężne aplikacje, których tworzenie do tej pory zabierało tygodnie czy miesiące, są teraz tworzone dosłownie w kilka dni."

Niestety – w ciągu ostatnich trzech lat spadło zainteresowanie technologią Ruby on Rails (patrz wykres 1.1). Programiści coraz rzadziej sięgają po ten produkt wybierając nowsze rozwiązania takie jak Django[3] napisane w języku Python[4]. Nadzieją na poprawienie tej sytuacji jest nowo wydana – trzecia wersja frameworku Ruby on Rails oraz ciągły rozwój dodatków – wtyczek `gem`. Dlatego też chcę przybliżyć tę technologię i zachęcić do jej używania.

Liczby na wykresie 1.1 wskazują, ile wyszukiwań przeprowadzono na podstawie określonego hasła w porównaniu do łącznej liczby wyszukiwań przeprowa-

[1] Patrz: http://basecamphq.com/
[2] Źródło: http://www.rubyonrails.pl/cytaty
[3] Patrz: http://www.djangoproject.com/
[4] Patrz: www.python.org



Rysunek 1.1: Statystyka wyszukiwarki Google na temat znanych aplikacji szkieletowych (źródło: http://www.google.com/insights/search/#cat=5&q=Ruby%20on%20Rails%2CDjango%2CSpring%20MVC&cmpt=q).

dzonych w Google w tym czasie. Wartości te nie odzwierciedlają bezwzględnej liczby wyszukiwań, ponieważ dane są znormalizowane i przedstawione na skali od 0 do 100. Każdy punkt na wykresie jest dzielony przez wartość najwyższego punktu. Jeśli ilość danych jest za mała, podawana jest wartość 0. Liczby wyświetlane nad wykresem obok wyszukiwanych haseł stanowią podsumowania lub wartości łączne[5].

Opis zastosowania technologii Ruby on Rails w problemie stworzenia systemu aukcyjnego

Technologia Ruby on Rails umożliwia proste tworzenie aplikacji webowych dowolnego typu. Dla zaprezentowania jej możliwości wybrałem system aukcyjny jako przykład aplikacji webowej stworzonej w tym środowisku. Wybór ten nie jest przypadkowy – do tej pory nie znalazłem przykładowego systemu aukcyjnego napisanego przy użyciu aplikacji szkieletowej Ruby on Rails[6].

Pomysł jednak nie jest nowatorski – w sieci oraz w wielu pozycjach książkowych znajdują się przykłady wykonania sklepów internetowych, które są w budowie bardzo podobne do systemów aukcyjnych.

System aukcyjny to niezwykle rozwlekły i obszerny temat. Projekt tego typu zatem może być bardzo rozbudowany. Właśnie z tego względu zakładam, że mój projekt nie będzie „dokończony". Celem nie jest tu wykonanie całego projektu

[5]Źródło: http://www.google.com/support/insights//bin/answer.py?hl=pl&answer=87285

[6]Jedynym możliwym gotowym rozwiązaniem dla wykorzystania aplikacji webowej w celu wystawiania aukcji/prowadzenia licytacji jest zastosowanie wtyczki dla systemu CMS napisanego w Ruby on Rails.

„od początku do końca” a jedynie prezentacja możliwości jakie oferuje Ruby on Rails.

Przedstawienie prototypu systemu aukcyjnego

Wraz ze stworzeniem prototypu systemu aukcyjnego prezentuję podstawowe rozwiązania dla tego rodzaju problemu. Zagadnienie stworzenia systemu aukcyjnego jest problemem typowym dla dziedziny inżynierii oprogramowania. Wymaga wybrania i opracowania rozwiązań technicznych i technologicznych oraz określenia metodyki pracy nad danym zagadnieniem.

Stworzony przeze mnie prototyp jest swego rodzaju prezentacją zastosowanych w nim technologii oraz przykładowych rozwiązań.

Ocena możliwości wdrożenia proponowanych rozwiązań

Do pełnego przedstawienia cyklu pracy nad projektem wykonanym w technologii Ruby on Rails potrzebne jest przedstawienie metod wdrożenia aplikacji webowej oraz zaproponowanie sposobu jej konserwacji. W tym celu mam zamiar przybliżyć jedne z najprostszych znanych mi sposobów wdrożenia aplikacji Ruby on Rails.

Rozdział 2

Metoda

2.1 Opis technologii zastosowanych w pracy

2.1.1 Technologie W3 i poboczne

1. `XHTML5`[5] (z ang. HyperText Markup Language version 5 – język znaczników hipertekstu w wersji piątej). XHTML5 jest językiem określającym strukturę stron internetowych. Składniowo bazuje on na języku XML (jest podzbiorem języka XML). Wersja piąta zapewnia kompatybilność wsteczną względem poprzednich wersji, a przy tym precyzuje niejasności wersji 4 powodujące nieoczekiwane zachowanie w niektorych przeglądarkach.

2. `CSS3`[6] (z ang. Cascade Style Sheet version 3 – kaskadowy arkusz styli w wersji trzeciej). CSS3 jest językiem określającym wygląd elementów języka XML jaki wyświetlany jest w przeglądarce. Wersja 3 zapewnia kilka dodatkowych opcji, jak np. grid layouts (szablony pozycjonowane na bazie siatki)[1], shadows and rounded borders (cienie obiektów, zaokrąglenia obramowania)[2], itp.

3. `JavaScript` – jest małym, lekkim, zorientowanym obiektowo wieloplatformowym językiem skryptowym. JavaScript, mimo że nie jest użyteczny jako samodzielny język, został stworzony z myślą o łatwym zagnieżdżaniu w innych produktach i aplikacjach, jak na przykład przeglądarki internetowe. JavaScript może zostać powiązany z wewnętrzną strukturą danego środowiska dając programiście swobodną kontrolę nad jego elementami.

4. `AJAX` (z ang. Asynchronous JavaScript and XML, asynchroniczny JavaScript i XML) – technologia tworzenia aplikacji internetowych, w której interakcja użytkownika z serwerem odbywa się bez przeładowywania całego dokumentu, w sposób asynchroniczny. Ma to umożliwiać bardziej dynamiczną interakcję z użytkownikiem niż w tradycyjnym modelu, w którym każde żądanie nowych danych wiąże się z przesłaniem całej strony HTML.

5. `jQuery`[3] (z. ang. query – zapytanie) – lekka biblioteka programistyczna dla języka JavaScript, ułatwiająca korzystanie z JavaScript (w tym manipulację

[1] http://www.w3.org/TR/css3-grid/
[2] http://www.w3.org/TR/css3-background/
[3] http://docs.jquery.com/Main_Page

drzewem DOM). Kosztem niewielkiego spadku wydajności w stosunku do profesjonalnie napisanego kodu w niewspomaganym JavaScripcie pozwala osiągnąć interesujące efekty animacji, dodać dynamiczne zmiany strony, wykonać zapytania AJAX. Większość pluginów i skryptów opartych o jQuery działa na stronach nie wymagając zmian w kodzie HTML (np. zamienia klasyczne galerie złożone z miniatur linkujących do obrazków w dynamiczną galerię). Wszystkie efekty osiągnięte z pomocą jQuery można osiągnąć również bez jej użycia. Jednak kod okazuje się nieporównywalnie dłuższy i bardziej skomplikowany.

2.1.2 Ruby oraz Ruby on Rails

1. `Ruby 1.9.2` (z. ang. ruby – rubin) – interpretowany, w pełni obiektowy i dynamicznie typowany język programowania stworzony w 1995 roku przez Yukihiro Matsumoto (pseudonim Matz). Wersja 1.9.2 cechuje się szybszym interpreterem, mniejszą konsumpcją zasobów oraz drobnymi poprawkami w standardowych bibliotekach języka.

2. `Ruby on Rails`[4] – patrz rozdział 1.2.1.

3. `RSpec`[5] + `Cucumber`[6] RSpec to narzędzie do testownia oprogramowania pod względem testów jednostkowych oraz behawioralnych przydatne w realizowaniu projektów Test Driven Developement oraz Behavior Driven Development. Narzędzie Cucumber pozwala na testowanie oprogramowania na podstawie tzw. scenariuszy – dokumentów napisanych w języku naturalnym opisujących krok po kroku funkcjonalności projektu.

2.1.3 Gemy i pluginy

1. `haml`[7] + `sass`[8] – plugin obsługujący język znaczników HAML używany do prostego i przejrzystego opisywania dokumentów XHTML oraz CSS.

[4] http://rubyonrails.pl/
[5] http://rspec.info/
[6] http://cukes.info/
[7] http://haml-lang.com/
[8] http://sass-lang.com/

2. `device`[9] – system obsługi autentyfikacji użytkowników (rejestracja, sesje, zarządzanie hasłami itp.).

3. `thinking-sphinx`[10] – potężny silnik wyszukiwania i dopasywania danych do podanych wzorców w relacyjnych bazach danych.

4. `will_paginate`[11] – plugin obsługujący paginację stron.

5. `tiny_mce`[12] – plugin pozwalający na użycie wewnętrznego edytora HTML.

6. `sqlite3`[13] – adapter bazy danych Sqlite w wersji trzeciej.

[9] https://github.com/plataformatec/devise
[10] http://freelancing-god.github.com/ts/en/
[11] https://github.com/mislav/will_paginate/wiki
[12] http://tinymce.moxiecode.com/
[13] http://www.sqlite.org/

2.2 Narzędzia użyte podczas pisania pracy

2.2.1 Kontrola pracy w Scrum

1. `git`[14] – system kontroli wersji bazujący na przechowywaniu plików różnic – tzw. plików diff.
2. `ticgit`[15] – issue tracker działajacy jako rozszerzenie dla systemu kontroli wersji git; zapisuje zmiany w trackingu w oddzielnej gałęzi repozytorium git projektu.
3. `Evolus Pencil`[16] – narzędzie do tworzenia mockupów oraz szkiców założeń.
4. `Umlet`[17] – narzędzie do rysowania diagramów UML.

2.2.2 Środowisko programistyczne

1. `vim`[18] – edytor tekstu.
2. `RVM`[19] – system kontrolii wersji języka Ruby.
3. `rubygems`[20] – system obsługujący bazę pluginów.
4. `IRB` – interaktywna konsola języka Ruby.
5. `rspec` + `cucumber` – patrz rozdział 2.1.3
6. `Sqliteman`[21] – narzędzie do zarządzania bazami danych Sqlite3.

[14] http://git-scm.com/
[15] https://github.com/schacon/ticgit/wiki/
[16] http://pencil.evolus.vn/en-US/Home.aspx
[17] http://www.umlet.com/
[18] http://www.vim.org/
[19] https://rvm.beginrescueend.com/
[20] http://rubygems.org/
[21] http://sqliteman.com/

2.2.3 Wdrożenie

1. `heroku`[22] – serwis pozwalający na tworzenie deploymentu projektów napisanych w języku Ruby „w chmurze”.

2. `Nginx + thin`[23] – lokalne wdrażanie aplikacji przy użycu serwera HTTP `Nginx` oraz lekkiego serwera `thin`.

[22] http://www.heroku.com/
[23] Nginx: http://wiki.nginx.org/NginxPl, Thin: http://code.macournoyer.com/thin/

2.3 Metodyka Scrum

Zastosowaną przeze mnie metodyką pracy jest Scrum [1] (z ang. scrum – przepychanka, młyn). Jest to jedna z wielu pochodnych metodyki programowania zwinnego [2] (z ang. agile programming).

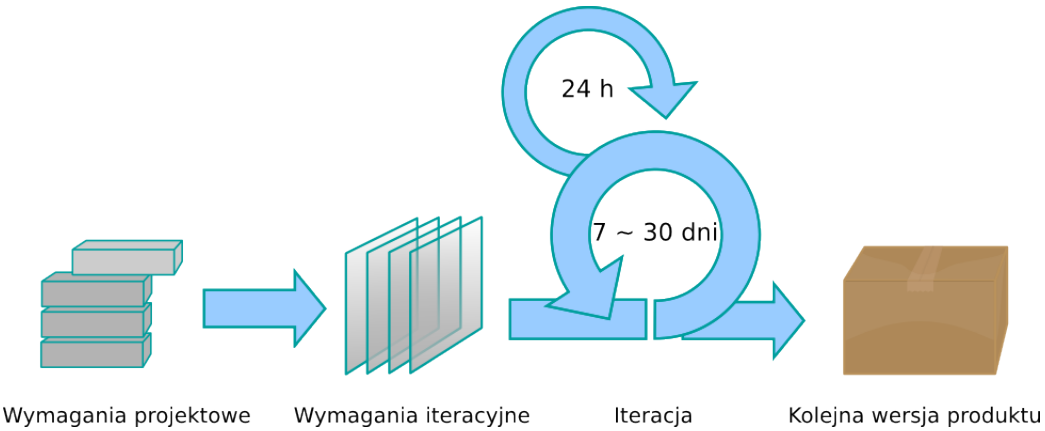
2.3.1 Programowanie zwinne

Programowanie zwinne jest popularną metodą pracy w wielu firmach parających się programowaniem. Dla wyjaśnienia czym tak naprawdę jest wystarczy nam odczytać definicję z Wikipedii [3]:

> Programowanie zwinne (z ang. Agile software development) – grupa metodyk wytwarzania oprogramowania opartego o programowanie iteracyjne (model przyrostowy). Wymagania oraz rozwiązania ewoluują przy współpracy samozarządzalnych zespołów, których celem jest przeprowadzanie procesów wytwarzania oprogramowania. Pojęcie zwinnego programowania zostało zaproponowane w 2001 w Agile Manifesto[24].
>
> Generalnie metodyka oparta jest o zdyscyplinowane zarządzanie projektem, które zakłada częste inspekcje wymagań i rozwiązań wraz z procesami adaptacji (zarówno specyfikacji jak i oprogramowania). Metodyka ta najczęściej znajduje zastosowanie w małych zespołach programistycznych, w których nie występuje problem komunikacji, przez co nie trzeba tworzyć rozbudowanej dokumentacji kodu. Kolejne etapy wytwarzania oprogramowania zamknięte są w iteracjach, w których za każdym razem przeprowadza się testowanie wytworzonego kodu, zebranie wymagań, planowanie rozwiązań itd. Metoda nastawiona jest na szybkie wytwarzanie oprogramowania wysokiej jakości.
>
> Metoda nastawiona jest na bezpośrednią komunikację pomiędzy członkami zespołu, minimalizując potrzebę tworzenia dokumentacji. Jeśli członkowie zespołu są w różnych lokalizacjach, to planuje się codzienne kontakty za pośrednictwem dostępnych kanałów komunikacji (wideokonferencja, e-mail itp.).

Więcej informacji można znaleźć na stronie www.agileprogramming.org [2].

[24] Patrz: agilemanifesto.org/principles.html



Rysunek 2.1: Schemat pracy w kolejnych iteracjach metodyki Scrum (opracowane na podstawie: http://effectiveagiledev.com/Portals/0/800px-Scrum_process_svg.png).

2.3.2 Scrum

Omówię tu po krótce metodykę Scrum wykorzystaną podczas realizacji założeń części praktycznej pracy. Metodyka Scrum opiera się na ścisłych iteracjach, w których realizowane są założenia projektowe. Każda iteracja zaczyna się tzw. Sprint Meeting'iem (z ang. Sprint meeting – „spotkanie w biegu")[25].

Definicje pojęć dla metodyki Scrum

Definicja 1 (Deweloper) *Deweloperem (z ang. developer – konstruktor, inwestor) nazywamy osobę bądź firmę*[26]*, która zaangażowana jest w realizację czynności związanych z procesem wytwarzania oprogramowania. Zwykle deweloperem nazywamy członka zespołu projektowego – programistę.*

Definicja 2 (Iteracja) *Iteracją nazywamy cykl w procesie wytwarzania oprogramowania, który zamknięty zostaje poprzez zrealizowanie pewnego celu.*

Definicja 3 (Ticket) *Zadania przydzielone zespołowi projektowemu określamy mianem ticketu. Ticket (z ang. ticket – bilet) określa jakie obowiązki zostały narzucone na poszczególnych deweloperów. Jest zwykle odzwierciedleniem żądań i zaleceń od klienta.*

[25] Więcej na temat metodyki Scrum można znaleźć na stronie www.scrumalliance.org
[26] tu: osobę

Role

Główne role jakie można wymienić w zespole Scrum to:

1. Zespół programistyczny (z ang. The Team) – wykonawcy projektu, deweloperzy w liczbie do 9 osób. Do ich obowiązków należy ocena trudności zadań – ticketów – realizujących założenia projektowe, realizacja tych zadań oraz zgłaszanie błędów, trudności itp. zaistniałych w projekcie.

2. Właściciel projektu (z ang. Product Owner) – jest to osoba, firma, grupa itp. będąca zleceniodawcą (klientem) zatrudniającym zespół programistyczny do zrealizowania projektu. Zasadniczą rolą właściciela projektu jest przedstawianie założeń, wymagań co do projektu, rewizja wykonanych zadań tegoż zespołu oraz konsultowanie zmian. Gdy właścicielem projektu jest jakaś większa jednostka (firma, spółka, grupa itp.) wtedy wyznaczany jest reprezentant odpowiedzialny za wyżej wymienione czynności.

3. Mistrz młyna (z ang. Scrum Master) – osoba odpowiedzialna za komunikację pomiędzy zespołem programistycznym a właścicielem projektu. Do jej obowiązków należy przede wszystkim: ustalanie sprint meeting'ów, zgłaszanie intencji zespołu programistycznego, wysyłanie powiadomień o zmianach w założeniach itp.

Rozpoczęcie pracy w Scrum

Zanim zostanie otworzona pierwsza iteracja – sprint – należy dokładnie przemyśleć i omówić możliwości grupy projektowej oraz skonfrontować je z wymaganiami klienta. W tym celu organizowane jest spotkanie inicjujące pracę nad projektem. Na takim spotkaniu powinny zostać omówione następujące kwestie:

1. Omówienie metody pracy z klientem – klient powinien wiedzieć jak pracuje zespół czego może się po nim spodziewać.

2. Prezentacja założeń projektowych – tu zespół programistyczny dowiaduje się jakie są postawione wobec niego oczekiwania.

3. Rewizja założeń projektowych – zespół programistyczny ma tu szansę wypowiedzieć się na temat kolejnych założeń – związanych z nimi trudności, konfrontacja z umiejętnościami (czego trzeba się „douczyć" a co jest zagwarantowane poprzez doświadczenie zespołu).

4. Wycena projektu oraz ustalenie licencji jego użytkowania.

Po omówieniu tych zagadnień klient może zdecydować, czy taki sposób pracy mu odpowiada. W tym momencie jest w stanie oszacować jak długo będzie współpracował nad projektem z tą grupą projektową, a co za tym idzie – jaką ilość pieniędzy może przeznaczyć w poszczególnych etapach tworzenia projektu.

Sprint meeting

Sprint meeting zamyka jedną iterację i rozpoczyna następną. Podczas sprint meeting'u realizowane są zatem: sprawdzenie poprawności wykonanych zadań z poprzedniej iteracji oraz przedyskutowanie planu pracy w następnej iteracji. Pozwala to klientowi na pełną kontrolę nad procesem wytwarzania projektu. Do najważniejszych kwestii omawianych na sprint meeting'u należą:

1. Sprawdzenie stanu wykonanych zadań – jeśli zadania zostały dostarczone klientowi jako wykonane, to klient może je zweryfikować pod względem ich poprawności. Każde takie zadanie może zostać zaakceptowane (wtedy oznaczone zostaje jako wykonane) bądź nie (Zadanie odrzucone wymaga poprawki – klient może zdecydować, co powinno być w tej sytuacji wykonane. Jeżeli klient ma wystarczająco funduszy na wykonanie poprawek wtedy może zlecić poprawkę, jeśli nie, to zadanie może zostać „zamrożone" i czekać na pieniądze potrzebne do jego realizacji. Oczywiście klient może także zaniechać realizacji zadania.).

2. Określenie wymagań projektowych – klient wymienia swoje oczekiwania względem projektu. Wymagania te zostają skonfrontowane z możliwościami zespołu programistycznego („tego nie da się zrobić", „to jest za trudne", „to wymaga lepszego sprzętu", „realizacja tego zadania zajmie . . . ", „koszta tego zadania wyniosą około . . . ", albo: „to jest proste", „znamy się na tym" itp.). Zwykle zaistniałe trudności w realizacji zadań kończą się kompromisem. Po określeniu możliwości realizacji wymagań tworzone są zadania.

3. Określenie zadań dla grupy projektowej – po „wycenie" możliwości deweloperów względem wymagań tworzone są zadania. Zwykle rozbija się je na prostsze, wymagające mniej czasu (według zasady „dziel i zdobywaj") – uzyskuje się wtedy płynność w realizacji zadań, a praca nad konkretnym wymaganiem podzielona jest pomiędzy członkami zespołu.

4. Przydzielenie zadań – klient po określeniu i sprecyzowaniu zadań określa ich priorytet. Niektóre zadania są ważniejsze – te oznaczane są jako zadania przypisane nadchodzącej iteracji – nad tymi zadaniami grupa projektowa będzie w najbliższym czasie pracować. Zadania te zostają następnie wybierane przez konkretnych deweloperów jako te, które będą przez nich realizowane. Jeżeli po zakończonej iteracji zostaną zadania nie wykonane przez nikogo oznacza to, że grupa nie nadąża z tempem pracy narzuconym przez klienta.

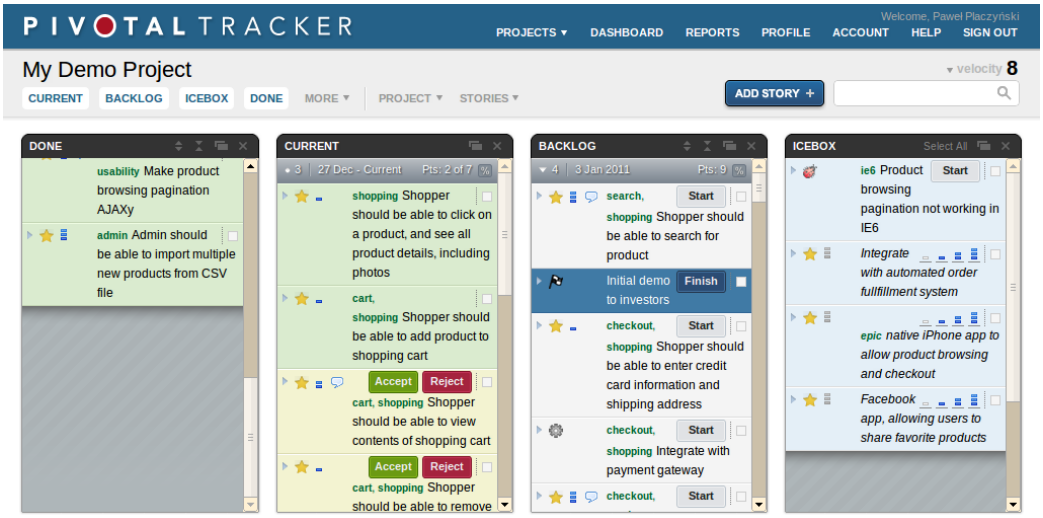
5. Ustalenie „dostępności" deweloperów w najbliższej iteracji. Członkowie zespołu mogą z różnych powodów nie móc pracować w pewnym okresie czasu. Z tego względu pod koniec sprint meeting'u należy ustalić ile godzin pracy każdy deweloper może przeznaczyć na najbliższą iterację. Pozwala to określić „siłę" zespołu oraz długość iteracji. W szczególnych przypadkach iteracje mogą zostać przeniesione bądź zawieszone „do odwołania".

2.3.3 Narzędzia Scrum

Narzędzi, które wspomagają pracę w metodyce Scrum jest wiele. Przykładem mogą być tzw. Issue Tracker'y – środowiska do zarządzania ticketami. Dobrym przykładem takiego Issue Trackera jest PivotalTracker[27]. Pozwala on na kontrolowanie zadaniami poprzez oznaczanie ich jako wykonanych, bieżących itp. Przykładowe zastosowanie tego narzędzia w pracy nad projektem przedstawia rysunek 2.2.

Innym przykładem narzędzia wspomagającego pracę w projekcie Scrum są testy jednostkowe i funkcjonalne aplikacji (patrz rozdział 2.4.3). Służą one tutaj przede wszystkim rewizji zaimplementowanej funkcjonalności a zatem sprawdzeniu poprawności wykonania zadania. Testy takie są zatem udokumentowaniem wykonanej pracy przez dewelopera.

[27] www.pivotaltracker.com



Rysunek 2.2: Zastosowanie narzędzia PivotalTracker w pracy nad projektem (źródło: www.pivotaltracker.com).

2.4 Dokumentacja projektu

Projekty systemów informatycznych mają to do siebie, że rozwijają się niezwykle szybko. Rozwój projektu związany jest niestety z powiększaniem objętości projektu – zarówno merytorycznej, jak i czysto fizycznej (np. ilość linijek kodu, ilość plików, itp.). Problem pojawia się, gdy projekt jest zbyt „duży" żeby programista mógł rozumieć wszystko to, co się w nim dzieje. Rozwiązaniem dla tego problemu jest tworzenie dokumentacji.

W projektach typu OpenSource dokumentacja jest niezwykle ważnym czynnikiem usprawniającym pracę programistów w nich pracujących. Dobrze napisana dokumentacja sprawia, że nowe osoby zaczynające pracę w projekcie mogą łatwiej i szybciej zapoznać się ze strukturą, działaniem oraz organizacją projektu. Jest to szczególnie przydatne, gdy istnieje potrzeba edycji jedynie niewielkiego fragmentu projektu. Co więcej – osoby zajmujące się już od dłuższego czasu pracą w takim projekcie nie muszą pamiętać wszelkich zagadnień z nim związanych. Pozwala to programiście na pracę w wielu projektach na raz.

2.4.1 Kod aplikacji

Najlepszą dokumentacją dla systemu informatycznego jest kod aplikacji. To właśnie on realizuje wszystkie założenia projektowe, algorytmy czy cykle pracy naszego projektu. Prawidłowo napisany kod może powiedzieć więcej niż nie jedna dokumentacja – tu dowiadujemy się jak naprawdę działa interesujący nas moduł i mamy pewność, że nie padniemy ofiarą nieporozumień. Pod pojęciem „prawidłowo napisany kod" mam na myśli spełnienie następujących założeń:

1. trzymanie się konwencji określającej osnowę dokumentu zawierającego kod (np. konsekwencję w stosowaniu wcięć),
2. stosowanie zrozumiałych nazw dla wszelkich struktur (nazwy klas, metod, zmiennych, itp.),
3. tworzenie krótkich i treściwych fragmentów kodu oraz rozbijanie większych partii na mniejsze.

Wszystkie te kroki mają na celu sprawienie, że kod stanie się bardziej czytelny. Omówię teraz te trzy kroki nieco dokładniej w zastosowaniu dla języka Ruby.

Konwencja

Język Ruby ze względu na swoją składnię jest niezwykle czytelny, a co za tym idzie, ułatwia dokumentację projektu w nim napisanego. Składnia Rubiego przypomina pseudokod:

```
1  def dfs node, value, queue
2    return false if node.nil?
3    return true if node.data == value
4    queue.push node.right_neighbor unless node.right_neighbor.nil?
5    queue.push node.left_neighbor unless node.left_neighbor.nil?
6    dfs queue.pop, value, queue
7  end
```

Listing 2.1: Przykład prostej składni języka Ruby – algorytm DFS.

Niestety – sama składnia nie jest najważniejsza w dokumentowaniu projektów. Powyższy przykład można przecież przepisać w następujący sposób:

```
1  def dfs node, value, queue
2  return false if node.nil?; if node.data == value
3  return true
4  end; queue.push node.right_neighbor unless \\
5    node.right_neighbor.nil?
6  unless node.left_neighbor.nil?
7  queue.push node.left_neighbor; end
8  dfs queue.pop, value, queue
9  end
```

Listing 2.2: Algorytm DFS z zastosowaniem braku konwencji formatu.

Jak widać przykład ten jest mniej czytelny, a co za tym idzie, trudniej jest dowiedzieć się, za co dany fragment kodu jest odpowiedzialny. W celu uzyskania „przejrzystości" kodu stosuje się konwencje zapisu – ogólnie przyjęte zasady mówiące o tym jak powinien wyglądać kod i jak ten kod formatować. Jest to swego rodzaju etykieta – zbiór zasad obowiązujących dla języka. Każdy język ma swoją konwencję (a czasem nawet kilka). Język Ruby także „dorobił się" swojej. Konwencja ta nosi nazwę *The Ruby Style*[28] i określona jest następująco (tłumaczenie własne):

1. Formatowanie:

   (a) Używaj zawsze kodowania ASCII lub UTF8.

[28] Patrz: https://github.com/chneukirchen/styleguide/blob/master/RUBY-STYLE

(b) Używaj dwóch spacji jako wcięć (nigdy tabulatur).

(c) Staraj się kończyć wiersz w stylu używanym w systemach Unix (LF – 0x0A).

(d) Używaj spacji przed i po operatorach, po przecinkach, po dwukropkach, po średnikach, przed i po { oraz po }.

(e) Nie używaj spacji po (oraz [. Nie używaj spacji przed] oraz).

(f) Używaj dwóch spacji przed modyfikatorem warunku (`if`/`unless`/`while`/`until`/`resque`).

(g) Wcięcie dla słowa kluczowego `when` powinno być tak głębokie jak dla `case`.

(h) Użyj pustego wiersza przed zwracaną wartością w metodzie (chyba, że ma ona tylko jeden wiersz), a także przed słowem kluczowym `def`

(i) Używaj RDoc do dokumentacji API. Nie wstawiaj pustego wiersza pomiędzy komentarz a komentowany blok.

(j) Użyj pustego wiersza do podzielenia dużych metod na logiczne fragmenty.

(k) Staraj się by każdy wiersz miał mniej niż 80 znaków.

(l) Unikaj białych znaków na końcu wiersza.

2. Składnia:

(a) Użyj `def` z nawiasami, gdy są podane argumenty.

(b) Nie używaj `for`, chyba, że robisz to celowo.

(c) Nie używaj `then`.

(d) Użyj „`when x; ...`” dla jednolinijkowego wyrażenia `case`.

(e) Użyj `&&`/`||` dla wyrażeń boolowskich, `and`/`or` do kontroli przepływu. (Ogólna zasada: jeśli używasz nawiasów, to znaczy, że używasz złych operatorów).

(f) Unikaj wielolinikowej instrukcji `?:`, użyj `if`.

(g) Zaniechaj użycia nawiasów przy wywołaniu metod, ale użyj ich podczas wywołania „funkcji” (np. gdy używasz zwracanej wartości w tym samym wierszu)

(h) Używaj raczej `{ ... }` niż `do ... end`. Wielolinijkowe bloki `{ ... }` są w porządku: używając `}` na końcu bloku wiemy, że kończy się blok a nie instrukcja `if`/`while`/.... Używaj `do ... end` do kontroli przepływu (np. zadania `rake`, bloki `sinatra`).

(i) Unikaj używania słowa kluczowego `return` jeśli nie jest potrzebne.

(j) Unikaj kontynuacji linii (\) jeśli nie musisz.

(k) Używanie zwracanej wartości przez operator = jest na miejscu.

(l) Używaj operatora `||=`.

(m) Używaj wyrażeń regularnych typu „non-OO". Nie bój się używać `=` , `$0-9`, `$ `, `$‘` oraz `$’` jeśli potrzebujesz.

3. Nazewnictwo:

(a) Używaj snake_case jako stylu nazywania metod.

(b) Używaj CamelCase jako stylu nazywania klas i modułów (Pozostaw akronimy takie jak HTTP, RFC, XML z wielkich liter).

(c) Używaj SCREAMING_SNAKE_CASE jako stylu nazywania stałych.

(d) Długość nazwy zwykle określa kontekst wykorzystania. Używaj jednoliterowych zmiennych jako parametrów bloków/metod według tego schematu:
`a`, `b`, `c`, `o`: dowolny obiekt;
`d`: katalog;
`e`: element (klasy Emumerable oraz pochodnych);
`ex`: wyjątek;
`f`: plik:
`i`, `j`: indeksy;
`k`: klucz tablicy asocjacyjnej;
`m`: metoda;
`r`: zwracana wartość krótkich metod;
`s`: tekst;
`v`: wartość elementu tablicy asocjacyjnej;
`x`, `y`, `z`: liczby;
Ponadto pierwsza litera klasy obiektu może posłużyć za nazwę takiej zmiennej.

(e) Używaj nazw zaczynających się od _ dla nieużywanych zmiennych.

(f) Używając `inject` dla krótkich bloków nazywaj argumenty `|a, e|` (od: akumulator, element).

(g) Definiując operatory dwuargumentowe, nazywaj argument jako „other”.

(h) Używaj raczej `map` niż `collect`, `find` niż `detect`, `find_all` niż `select` oraz `size` niż `length`.

4. Komentarze:

(a) Komentarze dłuższe niż słowo rozpoczynają się z wielkiej litery i używane są zasady interpunkcji.

(b) Używaj dwóch spacji po każdej kropce w komentarzu.

(c) unikaj nie potrzebnych komentarzy.

5. Pozostałe:

(a) Pisz kod przyjazny dla opcji `ruby -w`.

(b) Unikaj tablic asocjacyjnych jako opcjonalnych parametrów. Być może metoda, którą piszesz robi zbyt wiele?

(c) Unikaj długich metod.

(d) Unikaj długich list parametrów.

(e) Używaj konstrukcji `def self.`*metoda* dla zdefiniowania metod singletonów.

(f) Staraj się rozwijać funkcjonalność standardowych metod.

(g) Unikaj `alias` – używaj `alias_method`.

(h) Używaj `OptionParser` do parsowania skomplikowanych opcji wejścia konsoli a `ruby -s` dla rozwiązań trywialnych.

(i) Staraj się zachować zgodność wielu wersji interpretera.

(j) Unikaj zbędnego metaprogramowania.

6. Ogólne zasady:

(a) Programuj w sposób funkcjonalny.

(b) Nie wydziwiaj z używaniem argumentów metod – chyba, że wiesz co robisz.

(c) Nie zmieniaj funkcjonalności standardowych bibliotek pisząc własne.

(d) Nie programuj zachowawczo[29].

(e) Postaraj się zachować prostotę kodu.

(f) Nie przesadź z projektowaniem.

(g) Ale także nie pozostaw swojej pracy niezaprojektowanej.

(h) Unikaj błędów.

(i) Poczytaj o innych konwencjach, aby móc rozwinąć tę.

(j) Bądź konsekwentny.

(k) Używaj prostych rozwiązań.

Stosowanie tej konwencji zapewni, że kod napisany przez nas będzie zgodny z ogólnym standardem, którego używają Rubyści na całym świecie.

Nazewnictwo w kodzie

Aby zrozumieć istotę działania projektu należy zrozumieć mechanizmy i algorytmy rządzące jego logiką. Zakładając, że chcemy dobrze odokumentować nasz projekt musimy tak napisać kod, by był zrozumiały – jak pseudokod opisujący algorytm. W tym celu wystarczy, że wszelkie obiekty, jakie tylko używamy w naszym kodzie, nazwiemy tak, by ich użycie w kontekscie było zrozumiałe a ich rola nie pozostawiała miejsca na zastanowienie. W trakcie pisania kodu najtrudniejszą kwestią jest nazywanie obiektów a nie – jak to się powszechnie uznaje – rozwiązanie logiki systemu. Szczęśliwie dla programistów języka Ruby można stosować nawet bardzo długie nazwy obiektów (oczywiście nie wolno przesadzać) pozwalające opisać zastosowanie danego obiektu w kodzie.

Najlepszy do wyjaśnienia tej sytuacji będzie przykład:

```
1  def wwd a, w, ko
2    return false if a.nil?
3    return true if a.d == v
4    ko.push a.r unless a.r.nil?
5    ko.push a.l unless a.l.nil?
6    dfs ko.pop, w, ko
7  end
```

Listing 2.3: Algorytm DFS z nazwami zmiennych o niejasnym znaczeniu.

Taki kod nie jest czytelny. Potencjalny „czytelnik” musi przemyśleć działanie kodu a i tak nie ma pewności czy zrozumie przeznaczenie kodu.

[29] Patrz: http://www.erlang.se/doc/programming_rules.shtml#HDR11

```
1  def dfs element, value, list
2    return false if element.nil?
3    return true if element.data == value
4    list.push element.right unless element.right.nil?
5    list.push element.left unless element.left.nil?
6    dfs list.pop, value, list
7  end
```

Listing 2.4: Algorytm DFS z nazwami zmiennych nie mówiących o przeznaczeniu kodu.

Tu kod został opatrzony lepszymi nazwami. Co prawda „czytelnik" jest w stanie domyśleć się (i to dość szybko) co ten kod „robi", ale przeznaczenie kodu nadal nie jest znane.

Nasz pierwszy przykład stanowi kod napisany w sposób przyjazny potencajnemu „czytelnikowi". W tym przykładzie widać już „na pierwszy rzut oka" jaką rolę spełnia kod, jakie jest jego przeznaczenie, a w razie potrzeby można taki kod rozbudować (np. w celu dodania funkcjonalności).

Aby całkowicie opisać problem nazewnictwa obiektów w kodzie przedstawiam ostatni przykład:

```
1  def st element, value, stack
2    return false if element.nil?
3    return true if element.value == value
4    stack.push element.predecessor unless element.predecessor.nil?
5    stack.push element.follower unless element.follower.nil?
6    dfs stack.pop, value, stack
7  end
```

Listing 2.5: Algorytm DFS – mylące nazwy zmiennych.

W tym przykładzie natomiast nazwy zostały specjalnie zmienione tak by ukryć przeznaczenie przed oczyma potencjalnego „czytelnika". W poprzednim przykładzie widzieliśmy jaki jest cel tego fragmentu kodu. Tutaj jesteśmy zakłopotani, gdyż nie rozumiemy przeznaczenia tego algorytmu.

Zasada dziel i zdobywaj

W *The Ruby Style* czytamy, że powinniśmy unikać długich metod. Dlaczego długie metody są złe? Ponieważ utrudniają czytanie i zniechęcają do zapoznania się z ich treścią. Podam tutaj przykład takiej metody:

```
1 def levenshtein s, t
2   cost = 0
```

```
  d = Array.new
  m = s.length
  n = t.length
  0.upto(m) do |i|
    d[i] = Array.new
  end
  0.upto(m) do |i|
    d[i][0] = i
  end
  1.upto(n) do |j|
    d[0][j] = j
  end
  1.upto(m) do |i|
    1.upto(n) do |j|
      cost = (s[i-1,1] == t[j-1,1]) ? 0 : 1
      d[i][j] = [(d[i - 1][j] + 1), (d[i][j - 1] + 1), \\
        (d[i - 1][j - 1] + cost)].min
    end
  end
  return d[m][n]
end
```

Listing 2.6: Obszerna metoda wykonująca algorytm Levenshtein'a.

Metoda ta może zostać zapisana w prostszy sposób. Wystarczy logiczne fragmenty metody zamienić na inne metody:

```
def levenshtein s, t
  a = initiate s.length, t.length
  calculate a
end

def initiate m, n
  array = Array.new
  0.upto(m) do |i|
    array[i] = [i]
  end
  1.upto(n) do |j|
    array.first[j] = j
  end
  array
end

def calculate array
  cost = 0
  1.upto(array.size) do |i|
    1.upto(array.first.size) do |j|
```

```
21        cost = (s[i - 1][1] == t[j - 1][1]) ? 0 : 1
22        d[i][j] = [(d[i - 1][j] + 1), (d[i][j - 1] + 1), \\
23          (d[i - 1][j - 1] + cost)].min
24      end
25    end
26    array.last.last
27  end
```

Listing 2.7: Metoda z poprzedniego przykładu podzielona na mniejsze metody.

Co przy tym zyskujemy? Metoda nas interesująca jest krótsza – łatwiej się ją czyta, łatwiej znaleźć w niej potencjalne błędy. Można także opisać kod stosując odpowiednie nazwy metod wywoływanych przez naszą metodę – dzięki temu nie musimy zagłębiać się dokładnie w działanie metody ale poznajemy kolejność wykonywanych w niej operacji. Podział taki może nawet paradoksalnie zmniejszyć objętość naszego kodu – jedna z metod może wykonać się kilkukrotnie w danej części kodu. W przyszłości może to pomóc nawet w optymalizacji kodu.

Zen programisty języka Ruby

„Właściwie napisany kod staje się dobrą dokumentacją samego siebie". Programiści języka Python byli jednymi z pierwszych i najbardziej zagorzałych zwolenników tej koncepcji. Aby to zaakcentować powstał krótki dokument o nazwie „The Zen of Python" autorstwa Tima Petersa [4]. Dokument ten można przeczytać wpisując w konsoli języka Python polecenie: `import this`. Oto treść tego dokumentu [30]:

1. Piękne jest lepsze niż brzydkie.
2. Wyrażone wprost jest lepsze niż domniemane.
3. Proste jest lepsze niż złożone.
4. Złożone jest lepsze niż skomplikowane.
5. Płaskie jest lepsze niż wielopoziomowe.
6. Rzadkie jest lepsze niż gęste.
7. Czytelność się liczy.

[30] Tłumaczenie wzięte z http://pl.python.org/forum/index.php?topic=392.0

8. Sytuacje wyjątkowe nie są na tyle wyjątkowe, aby łamać reguły.
9. Choć praktyczność przeważa nad konsekwencją.
10. Błędy zawsze powinny być sygnalizowane.
11. Chyba że zostaną celowo ukryte.
12. W razie niejasności powstrzymaj pokusę zgadywania.
13. Powinien być jeden – i najlepiej tylko jeden – oczywisty sposób na zrobienie danej rzeczy.
14. Choć ten sposób może nie być oczywisty jeśli nie jest się Holendrem.
15. Teraz jest lepsze niż nigdy.
16. Chociaż nigdy jest często lepsze niż natychmiast.
17. Jeśli rozwiązanie jest trudno wyjaśnić, to jest ono złym pomysłem.
18. Jeśli rozwiązanie jest łatwo wyjaśnić, to może ono być dobrym pomysłem.
19. Przestrzenie nazw to jeden z niesamowicie genialnych pomysłów – miejmy ich więcej!

Ze względu na to, że język Ruby rządzi się nieco innymi zasadami niż język Python, pokusiłem się o napisanie *The Zen of Ruby* [31]:

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.

[31] https://github.com/placek/ruby_tao/blob/master/zen_of_ruby.txt

8. Special cases aren't special enough to break the rules.
9. Errors should never pass silently.
10. Unless explicitly silenced.
11. In the face of ambiguity, refuse the temptation to guess.
12. There's more than one way to do it.
13. Although all of them may be inappropriate.
14. Now is better than never.
15. Although never is often better than right now.
16. If the implementation is hard to explain, it's a bad idea.
17. If the implementation is easy to explain, it may be a good idea.
18. Divide and abbreviate the code.

Oraz w wersji po polsku:

1. Piękne jest lepsze niż brzydkie.
2. Wyrażone wprost jest lepsze niż domniemane.
3. Proste jest lepsze niż złożone.
4. Złożone jest lepsze niż skomplikowane.
5. Płaskie jest lepsze niż wielopoziomowe.
6. Rzadkie jest lepsze niż gęste.
7. Czytelność się liczy.
8. Sytuacje wyjątkowe nie są na tyle wyjątkowe, aby łamać reguły.
9. Błędy zawsze powinny być sygnalizowane.
10. Chyba że zostaną celowo ukryte.
11. W razie niejasności powstrzymaj pokusę zgadywania.

12. Istnieje wiele sposobów na rozwiązanie danego problemu.
13. Aczkolwiek każdy może być nie najlepszym.
14. Teraz jest lepsze niż nigdy.
15. Chociaż nigdy jest często lepsze niż natychmiast.
16. Jeśli rozwiązanie jest trudno wyjaśnić, to jest ono złym pomysłem.
17. Jeśli rozwiązanie jest łatwo wyjaśnić, to może ono być dobrym pomysłem.
18. Dziel i zdobywaj!

2.4.2 Komentarze

Komentarze w kodzie to dobrzy sposób na wyjaśnienie zasadności użycia algorytmów, bądź struktury API poszczególnych elementów kodu. Stosowanie komentarzy pozwala także wyróżnić co ważniejsze dla „czytelnika" elementy kodu.

RDoc

Istnieje wiele konwencji co do formy komentarzy. Format komentarzy ma znaczenie nie tylko podczas czytania kodu – może on posłużyć pośrednio do wygenerowania pełnej dokumentacji API przy użyciu odpowiednich narzędzi.

Komentarze w języku Ruby mogą być dwojakiego formatu:

```
1   # Komentarz jednolinijkowy rozpoczyna sie
2   # od znaku krzyzyka.
3
4   =begin
5     Blok komentarza.
6     Taki blok rozpoczyna sie od znacznika "=begin"
7     a konczy na znaczniku "=end".
8     Komentarze jednolinijkowe sa jednak czesciej
9     uzywane.
10  =end
```

Listing 2.8: Komentarze w języku Ruby.

Narzędzie *RDoc*[32] to narzędzie konsolowe generujące dokumentację API w zadanym formacie (standardowo jest to HTML) na podstawie kodu oraz komentarzy w nim zawartych. Aby wygenerować taką dokumentację wystarczy wpisać w konsoli:

```
$ rdoc <opcje> [plik...]
```

RDoc Wygeneruje dla nas czytelną dokumentację nawet jeśli nie opatrzymy kodu komentarzami. Mimo to lepiej jest napisać takie komentarze. *RDoc* oferuje specyficzny format treści komentarzy:

1. Listy (wypunktowanie) tworzymy poprzez ustawienie na początku linii gwiazdki * bądź minusa -.
   ```
   - Pierwszy punkt.
   - Drugi punkt.
   - ...
   ```

2. Listy numerowane tworzymy poprzez ustawienie numeru oraz kropki na początku linii.
   ```
   1.  Pierwszy punkt.
   2.  Drugi punkt.
   3.  ...
   ```

3. Słowniki tworzymy poprzez wstawienie dwóch dwukropków po tłumaczonym słowie.
   ```
   kot::  Małe domowe zwierzątko.
   tygrys::  Nieco większe dzikie zwierzątko.
   ```

4. Nagłówki generowane są poprzez wstawienie znaku = na początku linii. Mogą istnieć trzy poziomy nagłówków.
   ```
   = Nagłówek pierwszego rzędu
   == Nagłówek drugiego rzędu
   === Nagłowek trzeciego rzędu
   ```

5. Poziomą linię wstawiamy poprzez podanie sekwencji co najmniej trzech minusów -.
   ```
   ----
   ```

6. Kursywę, pogrubienie i podkreślenie możemy zastosować na słowie otaczając słowo podkreślnikiem _, gwiazdką *, bądź plusem +.
   ```
   _italic_ *bold* +underline+
   ```

[32] Patrz: rdoc.sourceforge.net

Ponadto możemy użyć dodatkowych znaczników by kontrolować wygląd naszej dokumentacji w docelowym formacie (np. w HTML).

1. Nazwy klas, metod występujące w komentarzach zamieniane są automatycznie na odnośniki do konkretnych miejsc w dokumentacji.

2. Odnośniki zaczynające się od: `http:`, `mailto:` albo `www.` są rozpoznawane jako linki. Odnośnik HTTP, który wskazuje na obrazek wstawia obrazek w linijce. Adresy zaczynające się od `link:` traktowane są jako odnośniki do lokalnych plików.

3. Hiperłącza mogą być także tworzone według schematu:
   nazwa[*url*]

4. Słowo kluczowe `:title:` pozwala ustawić tytuł strony dokumentacji.

2.4.3 Testy

Testy to dodatkowe aplikacje sprawdzające poprawność logiki naszego projektu. Zasadniczą ich funkcją jest dowiedzenie, że dana funkcjonalność została zaimplementowana.

Testować w naszej aplikacji możemy wszystko, jednak dobrze jest ustalić co chcemy uzyskać poprzez napisanie testów. Biorąc pod uwagę powyższe kryterium testy dzielimy na:

1. testy jednostkowe. Testami jednostkowymi nazywamy testy, które sprawdzają poprawność modułów „silnika" aplikacji. Za pomocą testów jednostkowych testuje się zwykle klasy, metody, stany maszyny, poprawność algorytmów, wejścia i wyjścia strumieni itp.

2. testy behawioralne. Testy takie odpowiadają na pytanie: „co się stanie, gdy zrobimy . . . ". Testują one zachowanie aplikacji – reakcję na żądania, efekty działania zdarzeń (takich jak wypełnienie formularza) itp.

Testy jednostkowe RSpec

RSpec[33] jest narzędziem do tworzenia testów jednostkowych dla aplikacji napisanych w języku Ruby. Jego składnia jest prosta, a testy napisane w nim – czytelne:

[33] Patrz: rspec.info

```
describe Array do
  describe "#push" do
    it "puts a value at the end of array" do
      array = Array.new
      value = "Some value"
      array.push value
      array.last.should == value
    end
  end
end
```

Listing 2.9: Test RSpec testujący klasę `Array`.

W pierwszej kolejności podajemy opis testu – krótką informację o tym co testujemy i jakie mamy oczekiwania względem testowanego obiektu. Wewnątrz bloku testu realizujemy przypadek użycia naszego obiektu. W każdym momencie możemy sprawdzić, czy interesujący nas stan obiektu jest zgodny z naszymi oczekiwaniami. W tym celu używamy metod `should` oraz `should_not`.

Testy uruchamiamy podając w konsoli polecenie `rspec` oraz plik z napisanymi przez nas testami:

```
$ rspec nazwa_pliku_testu.rb
```

Trzeba pamiętać, że testy powinny mieć dostęp do logiki naszego projektu – np. poprzez użycie instrukcji `require`. Efektem działania naszego testu (z przykładu 2.4.3) będzie:

```
.

Finished in 0.00258 seconds
1 example, 0 failures
```

Listing 2.10: Efekt uruchomienia testu jednostkowego z poprzedniego przykładu.

Aby zrozumieć, co stało się podczas testowania, możemy zmienić format wyjścia na bardziej przyjazny człowiekowi (użyjemy do tego opcji `--format d`):

```
Array
  #push
    puts a value at the end of array

Finished in 0.0029 seconds
1 example, 0 failures
```

Listing 2.11: Efekt uruchomienia testu jednostkowego z opcją `--format d`.

Jak widać `rspec` poinformował nas o tym, które testy „przeszły” – czyli, które z testowanych przez nas funkcjonalności spełniły nasze oczekiwania. Dla jasności podam jeszcze jeden przykład testów jednostkowych:

```
describe Array do
  describe "#push" do
    it "puts␣a␣value␣at␣the␣end␣of␣an␣array" do
      array = Array.new
      value = "Some␣value"
      array.push value
      array.last.should == value
    end
    it "increases␣a␣size␣of␣an␣array␣of␣one" do
      array = Array.new
      array_size = array.size
      array.push "Some␣value"
      array.size.should == (array_size + 1)
    end
  end
  describe "#pop" do
    it "gets␣a␣value␣from␣the␣end␣of␣an␣array" do
      array = Array.new
      value = "Some␣value"
      array.push value
      array.pop.should == value
    end
    it "increases␣a␣size␣of␣an␣array␣of␣one" do
      array = Array.new
      array.push "Some␣value"
      array_size = array.size
      array.pop
      array.size.should == (array_size + 1)
    end
  end
end
```

Listing 2.12: Efekt uruchomienia testu jednostkowego z opcją `--format d`.

Uruchamiając testy poprzez podanie `rspec` bez opcji `--format d` otrzymujemy:

```

...F

Failures:
  1) Array#pop increases a size of an array of one
     Failure/Error: array.size.should == (array_size + 1)
     expected: 2,
          got: 0 (using ==)
     # ./array_spec.rb:28

```

```
Finished in 0.00951 seconds
4 examples, 1 failure
```

Listing 2.13: Efekt uruchomienia testu jednostkowego dla przykładu 2.4.3.

Jak widać pewne testy nie przechodzą. Oznacza to, że albo funkcjonalność nas interesująca nie jest zgodna z naszym założeniem, albo oczekujemy od niej zbyt wiele. Po uruchomieniu testów z opcją `--format d` otrzymujemy w wyniku:

```
Array
  #push
    puts a value at the end of an array
    increases a size of an array of one
  #pop
    gets a value from the end of an array
    increases a size of an array of one (FAILED - 1)

Failures:
  1) Array#pop increases a size of an array of one
     Failure/Error: array.size.should == (array_size + 1)
     expected: 2,
          got: 0 (using ==)
     # ./array_spec.rb:28

Finished in 0.00692 seconds
4 examples, 1 failure
```

Listing 2.14: Efekt uruchomienia testu jednostkowego dla przykładu 2.4.3 z opcją `--format d`.

RSpec jako dokumentacja

Przykład 2.4.3 mówi nam dość dużo o sposobie użycia elementów logiki naszego projektu. Testy jednostkowe są tak naprawdę (jak już wspomniałem) przypadkami użycia tychże elementów. Co więcej – przypadki te realizują się zgodnie z założeniami twórcy takiego projektu (uruchomione testy „przechodzą"). Oznacza to, że są one dobrą dokumentacją działania i sposobu użycia poszczególnych obiektów logiki projektu.

Dokumentacja poprzez testy behawioralne

Skoro testy jednostkowe są swego rodzaju dokumentacją, to także testy behawioralne mogą nią być. Oczywiście ze względu na inny cel testy behawioralne będą

dokumentować projekt z innej perspektywy. Testy te określają zachowanie aplikacji, a zatem kwestii ściśle związanej z użytkowaniem. W praktyce testy behawioralne są stosowane dla określenia wymagań ze strony klienta – zleceniodawcy projektu.

Istnieje kilka narzędzi, przy pomocy których można napisać i przeprowadzić testy behawioralne. Można do tego użyć `rspec` oraz `Test::Unit` (czyli narzędzi wykorzystywanych przy pisaniu testów jednostkowych). Dla dobrej czytelności oraz eleganckiego formatu używam narzędzia *Cucumber*[34].

2.4.4 System kontroli wersji

Systemy kontroli wersji pozwalają na współdzielenie kodu projektu oraz jednoczesną pracę nad nim wielu programistom. Systemy takie przechowują także pełną historię zmian zachodzących w projekcie, w tym: informacje o autorze zmian, informacje o czasie modyfikacji, wersje plików przed i po modyfikacji itp. Informacje te są niezwykle cenne jako dokumentacja opisująca pracę zespołu. Na podstawie logów można określić np. trudność realizacji danego zagadnienia (na podstawie czasu wykonania bądź dodanych linii kodu), aktywność programistów oraz temu podobne.

Przykładową dokumentacją przebiegu pracy nad projektem może być log narzędzia *Git*[35] wygenerowany za pomocą polecenia:

```
$ git log
```

```
1
2  commit aaf288534af2ed15da3ca1b200025b8be4c3526b
3  Author: Pawel Placzynski <placek@example.com>
4  Date:   Fri Dec 10 14:47:45 2010 +0100
5
6      Updated mongodb
7
8  commit 06fdd81b0104186f9f449487a9a1ceb92d4101f6
9  Author: Kamil Zygmuntowicz <kamil.zygmuntowicz@example.com>
10 Date:   Fri Dec 3 11:10:25 2010 +0100
11
12     Added rspec on [#6412487]
13
14 commit d081a370bf60f8ef0b8678602b4745b0b5e00d39
15 Author: andst4@example.com <andrzej@andrzej-laptop.(none)>
16 Date:   Fri Dec 3 09:54:04 2010 +0100
```

[34] Patrz: cukes.info
[35] Patrz: git-scm.com

```
17
18      Copying query with name.
19
20  commit 65d7d3daa13eabba81ed23cad8117c59e6a44e74
21  Author: Sebastian Wojtczak <wojtczaksebastian@example.com>
22  Date:   Tue Nov 30 17:11:39 2010 +0100
23
24      [6672367] Removed unnecessary border on app dashboard.
25
26  commit 818a69e07cf665021a4bb01cb5ed533020c8ada5
27  Author: Sebastian Wojtczak <wojtczaksebastian@example.com>
28  Date:   Sat Nov 27 21:32:03 2010 +0100
29
30      [#6760943] Sort helper
```

Listing 2.15: Log narzędzia *Git* dla pewnego projektu.

2.4.5 UML

Ostatnią, chyba najlepszą dla tak zwanych „wzrokowców", formą dokumentacji jest opisywanie struktur oraz zachowań projektu przy pomocy diagramów UML. UML (z ang. Unified Modeling Language, czyli Zunifikowany Język Modelowania) jest formalnym językiem pozwalającym na modelowanie różnego rodzaju systemów (w tym systemów informatycznych). Język ten jest używany w większości przypadków raczej jako model przyszłego (mającego powstać) systemu a rzadziej jako dokumentacja istniejącego już systemu. Jednak w myśl zasad inżynierii wstecznej[36] każda forma opisywania systemu jest jak najbardziej poprawną formą dokumentacji tegoż systemu.

Okazuje się, że UML jest jedną z bardziej czytelnych form dokumentacji, a co za tym idzie – szybką i praktyczną w użytkowaniu. Diagramy mają to do siebie, że szybko wpadają w pamięć, łatwiej przyswoić noszoną przez nie treść a także zrozumieć co bardziej skomplikowane aspekty tej treści.

[36] Patrz: http://www.npd-solutions.com/reoverview.html

Rozdział 3

Postępy pracy

Bibliografia

[1] *The Scrum Framework in 30 seconds*, www.scrumalliance.org/pages/what_is_scrum

[2] *Agile and Scrum programming*, www.agileprogramming.org

[3] *Programowanie zwinne*, http://pl.wikipedia.org/wiki/Programowanie_zwinne

[4] *The Zen of Python*, www.python.org/dev/peps/pep-0020

[5] *Dokumentacja standardu HTML5*, http://dev.w3.org/html5/spec/Overview.html

[6] *Dokumentacja standardu HTML5*, http://www.w3.org/TR/CSS/