Spis treści

I. Cel pracy

Celem pracy jest przedstawienie praktycznego zastosowania *Programowania Metodą Behawioralną* – jednej z metodyk *Programowania Zwinnego*, reprezentującego alternatywne do *Modelu Kaskadowego*, kompleksowe podejście do zarządzania i rozwijania projektów informatycznych.

Na potrzeby prezentacji tego podejścia w kolejnych rozdziałach przedstawię historię ewolucji metodyk programowania, a później, wykorzystując technikę *Behawioralną* krok po kroku, zbuduję w pełni funkcjonalna, modularna aplikacja sieciowa, przedstawiającą podstawowe funkcjonalności nowoczesnej aplikacji typu Web2.0.

W projekcie wykorzystane zostaną:

- Język programowania: *Ruby* (wersja 1.9.2, implementacja *MRI*)
- Platforma do tworzenia aplikacji sieciowych: *Ruby on Rails* (werjsa 3.1)
- Platforma do projektowania behawioralnego: *Rspec*
- Język opisu scenariuszy behawioralnych: *Cucumber*
- System zarządzania uprawnieniami: *Cancan*

W tworzeniu aplikacji pomagać będą inne narzędzia, nie mają one jednak wpływu na kształt programu, ani na sposób prezentacji metodyki BDD.

Skorzystanie z nich wynika tylko z moich osobistych preferencji.

- System wersjonowania oprogramowania: *Git*
- System zarządzania wersjami interpretera języka *Ruby*: *RVM*

II. Wprowadzenie historyczne
Jak zmieniało się myślenie o programowaniu i jak powstała metodyka *BDD*

Programowanie było jeszcze bardzo młodą dziedziną nauki, kiedy pod koniec lat 50ych XX wieku w *Massachusetts Institute of Technology* pojawiły się pierwsze próby wprowadzenia „obiektów" jako jej podstawowego budulca. Cyfrowy obiekt miał reprezentować nowy poziom abstrakcji, stanowić most pomiędzy ludzkim językiem i jego postrzeganiem świata, a pamięcią komputera, w której wszystko musi mieć swoje miejsce, wartość; a przy tym koncepcja ta miała być instynktownie prosta, jak pojęcie podmiotu w zdaniu. Stworzony w 1958 r. przez Johna McCarthy'ego (nagroda Turinga w 1971 r. za prace nad podstawami sztucznej inteligencji) język *LISP* oferował „atomy" – protoplastów dzisiejszych obiektów – i system ich atrybutów na długo przed tym, gdy dr Alan Kay (który w 2003 r. otrzymał nagrodę Turinga za wkład w prace nad językami programowania, jak również interfejsami graficznymi) w 1966 r. zaproponował termin *object-oriented programming*[1] (ang. programowanie zorientowane obiektowo – choć w języku polskim używa się prostszej nazwy „programowanie obiektowe"). Język *LISP* nie był dla Kaya jedyną inspiracją – na przełomie lat 1960-61 Ivan Edward Sutherland (laureat nagrody Turinga w 1988 r.) rozwijał swój autorski projekt *Sketchpad*, kładąc kamień węgielny pod całą rodzinę programów typu *CAD* (*Computer-aided Design*, ang. Projektowanie Wspomagane Komputerowo). Jednak ten pierwszy w historii program z interfejsem graficznym (którego Kay stał się natychmiast gorącym orędownikiem), napisany został w języku, który przeznaczony był tylko dla niego; poza tym miał inną wadę – działał na superkomputerze *Lincoln TX-2*, o którym dr Alan Kay powiedział: „Był to ostatni komputer w Ameryce, który miał własny dach"[2].

Pierwszy język programowania, w którym explicite zdefiniowano obiekty powstał jednak nie na terenie Stanów Zjednoczonych, ale w odległej Norwegii – był to język *Simula I*, autorstwa Kristena Nygaarda i Ole-Johana Dahla (nagroda Turinga przyznana obojgu w 2001 r. za pionierskie prace nad programowaniem obiektowym). Zgodnie z nazwą, rodzina języków *Simula* (później powstał jeszcze *Simula 67*) służyła do przeprowadzania symulacji – nie tylko wspomaganych komputerowo, ale także wykonywanych w całości w pamięci maszyny. Nowatorskie rozwiązania zapewniły norweskim informatykom sławę. W roku 1962 Nygaard został zaproszony do laboratoriów *UNIVAC*, producenta komputerów „biznesowych", czyli maszyn typu mainframe. W czasie swojego pobytu w USA prezentował rozwiązania programowania obiektowego,

1 http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en (stan z dnia 12.07.2011)

2 W 1987r, na potrzeby *University Video Communications*, dr Alan Kay przeprowadził prezentację pionierskich systemów komputerowych, które jako pierwsze w latach 60-tych i 70-tych przedstawiły światu interfejs użytkownika; w połowie piątej minuty filmu rozpoczyna się prezentacja programu *Sketchpad* – http://www.archive.org/details/AlanKeyD1987 (stan z dnia 13.07.2011)

które sam stworzył. Wkrótce *Simula* zaczęła zdobywać uznanie na całym świecie, zaimplementowano ją w amerykańskim komputerze *Burroughs B5500*, i w radzieckim *Ural-16*.

W 1966 r. brytyjski informatyk Sir Charles Antony Richard Hoare (laureat nagrody Turinga w 1980 r.) przedstawił definicję „klasy" – formalnego opisu danego rodzaju obiektu, przy założeniu, że programista może stworzyć podklasę – nowy „gatunek" obiektu, który „dziedziczyć" będzie atrybuty z klasy z której się wywodzi, dokonując na nich własnych zmian, analogicznie jak podczas ewolucji czynią to gatunki potomne w przyrodzie. Rok później powstał *Simula 67* – był pierwszym językiem w historii, który wprowadził klasy i pierwszym wyposażonym w system *garbage collection* (ang. zbieranie śmieci) – proces działający w tle programu, „czyszczący" zwalniane komórki pamięci tak, by łatwo zająć je mogły inne programy. W maju tego roku, w Oslo, przyjęto formalną definicję tego języka, a już w czerwcu Ole-Johan Dahl postulował wprowadzenie paradygmatu, który miał traktować typy prymitywne (liczby, znaki alfanumeryczne, wartości logiczne) jako obiekty określonej klasy, w ten sposób unifikując wszystkie dane programu do postaci obiektu. Jego propozycja została jednak odrzucona.

Dr Alan Kay, zafascynowany biologią, widział przyszłość programistyki w kontynuowaniu prac nad „obiektem" – cyfrowym bytem, opisanym atrybutami, reagującym na komunikaty, wysyłane przez inne byty. Atomy *LISPa* były dosyć prymitywne, obejmowały tylko liczby i łańcuchy znaków (zwane symbolami); program *Sketchpad* traktował geometryczne wierzchołki, krawędzie, a również całe figury, jako obiekty, ale jakkolwiek innowacyjne rozwiązania wprowadzał (jest pierwszym programem wykorzystującym ideę „ekranu dotykowego", chociaż „dotyk" odbywał się tutaj za pomocą świetlnego rysika, więc system nie reagowałby na dotyk ludzkiej dłoni), w swojej implementacji ograniczony był do konkretnego rozwiązania – projektowania graficznego. *Simula I* i *67*, były krokiem w dobrym kierunku, ale choć języki te jako pierwsze wprowadziły w życie koncepcje, o których Kay mówił na konferencjach otwarcie już od roku, chociaż dawały programistom potężne narzędzie w postaci cyfrowego obiektu i jego klasy, to już w 1969r Kay (pracujący wtedy w ośrodku *Xerox* w Palo Alto) poprowadził zespół w pracach nad językiem nowej generacji.

W roku 1972 powstał język *Smalltalk* (wśród członków zespołu Kaya należy wymienić Daniela Ingallsa, który pierwszą wersję nowego języka zaimplementował w ciągu zaledwie kilku dni na podstawie jednej strony notatek, w efekcie wygrywając zakład z Kayem). Na przestrzeni lat 1972-80 powstało kilka wersji rozwojowych tego języka. Zaprezentowany w 1980 r. *Smalltalk-80* z niewielkimi zmianami przetrwał do dziś i wciąż jest wykorzystywany w praktyce. Podstawową ideą tego języka było „traktowanie wszystkiego jako obiekt" – tak, jak to w 1967 r. w Oslo postulował Ole-Johan Dahl. Ujednolicenie wszystkich „podmiotów" biorących udział w konstrukcji programu, poza spełnieniem osobistego marzenia dra Alana Kaya o języku prawdziwie „obiektowo-zorientowanym", miało bardzo konkretny cel – zmienić sposób myślenia programistów.

Pracując jeszcze w *MIT*, dr Kay miał okazję obserwować początki rewolucji

w komunikacji człowieka z maszyną. Wśród prekursorów w tej dziedzinie należy wymienić rzecz jasna wspomnianego wcześniej Ivana Sutherlanda, ale przede wszystkim Douglasa Carla Engelbarta (nagroda Turinga w 1997 r.) – wynalazcę myszki komputerowej i *hipertekstu* – systemu wzajemnie powiązanych dokumentów (leksji), które użytkownik nawiguje nie w ustalonej kolejności, a wg. własnego uznania, wybierając odpowiednie łącza – bez tego wynalazku internet, jakim go znamy, nigdy by nie powstał. Mimo wszystko te koncepcje całkowicie odmiennego podejścia do komunikacji z maszyną przez blisko 20 lat pozostawały tylko w sferze ciekawostek. Doświadczenia wyniesione przez dr Alana Kaya z tych obserwacji umocniły go w przekonaniu, że komputery zmieniły się wystarczająco i nadszedł czas by zmienić „drugą stronę" dialogu: użytkownika i jego podejście. Przyświecała mu przy tym myśl kanadyjskiego filozofa i teoretyka komunikacji Herberta Marshalla McLuhana (jako człowiek niezwiązany z amerykańskimi ośrodkami badań i rozwoju informatyki przewidział powstanie sieci komputerowej i unifikację mediów na 30 lat zanim został stworzony internet; jako pierwszy użył też powiedzenia, że świat dąży do tego, by stać się „globalną wioską"), który żartobliwie mówił, że „nie wiemy kto wymyślił wodę, ale na pewno nie były to ryby" – przekaz był dla Kaya jasny, rewolucji obsługi komputera nie można dokonać tylko w gronie naukowców, którzy od lat stykają się z najnowszymi technologiami. Potrzebni byli użytkownicy „z zewnątrz".

Smalltalk posłużył w serii eksperymentów w drugiej połowie lat 70ych, w których wzięło udział kilkaset dzieci w wieku lat 12-13 z okolicznych szkół w Palo Alto. W ciągu trwających dwa szkolne semestry zajęć nauczono młodych programistów obsługi komend *Smalltalk* poczynając od tych prostych, odziedziczonych po języku *Logo* – czyli tworzenia grafiki na ekranie za pomocą funkcji – aż po naukę pisania własnych programów. Najważniejszym elementem eksperymentu były zajęcia prowadzone dla uczniów przez uczniów ze starszych klas. Okazało się, że dzieci, które nie miały wcześniej żadnej styczności z informatyką, nie będące „zakładnikami" starych koncepcji programowania, były w stanie pisać kompletne programy użytkowe, jak edytory grafiki i animacji, a ich kod był wyjątkowo koherentny w porównaniu z tym jak wyglądałby on w języku proceduralnym. Zespół *Xeroxa* z Palo Alto dowiódł tym samym, że paradygmat „obiektu" jest naturalny, instynktownie rozumiany nawet przez dzieci, mające zaledwie kilkumiesięczne doświadczenie w pracy z komputerem.

Języki z rodziny *Smalltalk*, pierwsze w historii „zorientowane obiektowo", stały się inspiracją dla wielu późniejszych języków wysokiego poziomu, jak chociażby *Java*, dodatkowo część języków przejęła rozwiązania składni i dynamicznego typowania (typ, czyli klasa zmiennych nie jest deklarowana wprost, przez podanie słowa kluczowego, ale przez przypisanie wartości), są to przede wszystkim języki skryptowe i interpretowane, jak *Perl*, *Python* i *Ruby*.

Nie ma przesady w mówieniu, że programowanie obiektowe zrewolucjonizowało informatykę, chociaż na prawdziwe efekty należało czekać do lat 80ych, kiedy komputery osobiste na dobre trafiły do domów indywidualnych użytkowników, a programowaniem

mogli zająć się ludzie nie związani z uczelnianymi, bądź wojskowymi ośrodkami badań. Do dziś jedyną alternatywą dla programowania proceduralnego jest programowanie obiektowe, a najpopularniejsze języki[3] w użyciu są właśnie obiektowymi. Wyjątkiem jest język *C*, któremu historia wyznaczyła specjalnie miejsce: jako język niemal natywny dla tworzonych procesorów obsługuje ich najbardziej podstawowe funkcje, jest przez to najszybciej wykonywanym ze wszystkich języków wysokiego poziomu (choć często mówi się o nim, jako o języku poziomu pośredniego), stanowi przez to jedyny słuszny wybór, gdy liczy się szybkość, a nie wygoda wynikająca z użycia obiektów.

Rok 1958 był bardzo ważny w historii informatyki i telekomunikacji nie tylko ze względu na pionierskie prace nad rozwojem programowania w instytucie *MIT*. Amerykanie od początku lat 50ych opracowywali narzędzie ułatwiające komunikację i współpracę cywilnych i wojskowych ośrodków badawczych rozsianych po terenie całych Stanów. Pomysł wykorzystania sieci komputerowej, przesyłającej dane podzielone na „pakiety" pochodził od Paula Barana, informatyka polskiego pochodzenia (ur. w 1926 r, w Grodnie – obecnie Białoruś) ówczesnego członka RAND Corporation, jednego z pierwszych „think-tanków" w historii, i stanowił odpowiedź na postawione przez Departament wymaganie, aby system taki był w stanie przetrwać atak nuklearny z użyciem wielu głowic, co oznaczało, że musiał być zdecentralizowany. Do roku 1957 prace na ten temat były czysto teoretyczne, gdy nieoczekiwanie świat obiegła wiadomość, że Związek Radziecki umieścił na ziemskiej orbicie pierwszego satelitę w historii – *Sputnika 1*. Podjęto decyzję o natychmiastowym przyspieszeniu badań i już rok później, w 1958 r., administracja prezydenta Dwighta Eisenhowera powołała agencję *ARPA* (*Advanced Research Projects Agency* ang. Agencja Zaawansowanych Projektów Badawczych).

Zadanie stworzenia komputerowej sieci było tylko jednym z wielu, jakie rząd zlecił nowo powstałej agencji, ale mimo wszystko na pierwsze efekty trzeba było czekać zaskakująco długo, bo aż jedenaście lat. Projekt rozwijany był jednocześnie w kilku różnych ośrodkach badawczych, a najlepiej rokujące pomysły tworzyły pewnego rodzaju rdzeń dalszych prac. Pierwsze połączenie dwóch niezależnych ośrodków nastąpiło w Kalifornii w 1969 r. pomiędzy Instytutem Badawczym Stanforda, a kampusem Uniwersytetu Kalifornijskiego w Los Angeles[4] – jest to dystans około 500 kilometrów. Jeszcze w grudniu tego roku w sieci nazywanej *ARPANETem* połączone były cztery ośrodki; trzy w Kalifornii, jeden w Utah. To właśnie możliwość włączenia istniejących, wewnętrznych sieci jako podsieci do większego systemu teleinformatycznego, zapewniła *ARPANETowi* sukces.

W połowie lat 70ych na całym świecie operowało kilka niezależnych sieci komputerowych, jak francuski *CYCLADES* i brytyjska sieć Mark I, a w samych Stanach Zjednoczonych: *Tymnet* w Kaliforni, czy *Merit Network* w stanie Michigan. Zrodził się problem jak połączyć istniejące sieci całego świata w jeden wydajny układ. W 1974 r.

3 Wykresy popularności języków programowania używanych w projektach http://www.langpop.com/ (stan z dnia 12.07.2011)

4 http://www.netvalley.com/cgi-bin/intval/net_history.pl (stan z dnia 17.07.2011)

w efekcie współpracy Roberta Kahna z *DARPA* (Agencja została przemianowana na *Defense Advanced Research Projects Agency* w 1972 r.) i Vintona Cerfa z Uniwersytetu Stanforda, powstała „specyfikacja *RFC-675*[5]", w tytule której po raz pierwszy pada nazwa „internet". Najważniejszym punktem dokumentu było postulowanie o umniejszenie roli samej sieci do zaledwie medium, przenoszącego pakiety, całą resztę pracy miały wykonywać przede wszystkim serwery i w mniejszym stopniu maszyny klienckie; był to pomysł zaczerpnięty od twórcy *CYCLADESa* – Louisa Pouzina. W roku 1982 formalnie zdefiniowano i opatentowano protokół *TCP/IP*.

Jeszcze do końca lat 80ych na terenie USA powstawały odrębne sieci naukowe jak *CSNET* (*Computer Science Network*) i *NSFNET (National Science Foundation*), ale już z początkiem lat 90ych stanowiły wraz z *ARPANETem* jeden nierozróżnialny system, pozbawiony elementów centralnych, tak, jak przewidywały to wymagania Departamentu Obrony sprzed 40 lat. Termin „internet" który wcześniej był zaledwie skrótem od wyrażenia *internetworking* (ang. praca wewnątrz sieci) – zaczął być używany jako nazwa własna nowego medium teleinformatycznego.

Ośrodek *CERN* w Europie kuszony perspektywami połączenia z uczelniami zza oceanu zaadaptował rozwiązania *TCP/IP* na własne potrzeby. To właśnie tam już w 1989 r. Sir Timothy Berners-Lee przedstawił propozycję utworzenia systemu, który umożliwi magazynowanie i przeglądanie danych wewnątrz sieci, wykorzystując do tego programy zwane „przeglądarkami", działające za zasadzie obsługi *hipertkestu* – pomysłu Douglasa Engelbarta sprzed trzydziestu lat, który jeszcze nie doczekał się realizacji na skalę światową. Rok później do Bernersa-Lee dołączył Belg Robert Cailliau; razem dopracowali projekt, stworzyli pierwszą przeglądarkę, pierwsze strony hipertekstowe i serwer na stacji roboczej *NeXT*, do którego można było się połączyć i czytać znajdujące się na nim dokumenty. Swój prototypowy system nazwali *World Wide Web* (ang. Sieć Ogólnoświatowa), jakby przewidując, że istotnie stanie się on narzędziem używanym na całym świecie; Berners-Lee tworzył go od podstaw z zamierzeniem włączenia *WWW* do Internetu. I tak, jak przewidział w latach 60tych Marshall McLuhan, a dziesięć lat po nim jeden z ojców fantastyki naukowej Sir Arthur C. Clarke – powstała teleinformatyczna sieć, dająca ludziom dostęp do wiedzy pochodzącej z całego świata na wyciągnięcie ręki. Optymizm jej twórców tylko raz zderzył się z rzeczywistością – zakładali oni, że rozbudowanie systemu do funkcjonalności, w której użytkownicy sami będą mogli zarządzać jego treścią, to kwestia sześciu miesięcy od czasu zaprezentowania pierwszej wersji. Minęło 10 lat zanim świat internetu wkroczył w epokę *Web 2.0* – epokę blogów, forum, filmu, muzyki i telewizji w internecie. Statyczne internetowe „witryny" i „strony" miały zostać zastąpione funkcjonalnymi aplikacjami, hipertekstowe dokumenty – interaktywnymi prezentacjami z dźwiękiem i animacją.

Czasy gdy *MIT*, czy jakakolwiek inna instytucja, wytyczały kierunek i dyktowały trendy w rozwijaniu nowych metodyk pracy, czy technologii, minęły bezpowrotnie. *Web 2.0* uczynił z internautów współautorów nowego medium; każdy mógł publikować swoje

5 http://tools.ietf.org/html/rfc675 (stan z dnia 17.07.2011)

pomysły i opinie, ale nowe czasy wymagały nowego „przywództwa". To właśnie wtedy, w 2001 roku, w Snowbird, w stanie Utah, odbyła się konferencja, w której udział wzięli entuzjaści nowych technologii. Jej głównym zadaniem było wypracowanie porozumienia w sprawie metodyki pracy alternatywnej do używanego od lat 70ych „modelu kaskadowego". Opracowano *Manifest Zwinnego Wytwarzania Oprogramowania*, zwany potocznie *Manifestem Agile*, od jego angielskiej nazwy *Manifesto for Agile Software Development*. Jego główną ideą było przedkładanie[6]:

- **Ludzi i ich wzajemne interakcje (współdziałanie)** ponad procedury i narzędzia.
- **Działające oprogramowanie** nad wyczerpującą dokumentację.
- **Współpracę z klientem** nad negocjację umów.
- **Reagowanie na zmiany** nad realizowanie planu.

Najciekawszym, moim zdaniem, aspektem *Manifestu*, jest fakt, że był pierwszym formalnym dokumentem opisującym metody, którymi przez całe lata nieoficjalnie posługiwali się najwięksi programiści, jak wspomniany wcześniej współtwórca *Smalltalka* – Dan Ingalls, czy Donald Knuth, autor wielotomowej „Sztuki Programowania" i twórca systemu składania tekstu *TEX*. W gruncie rzeczy wszystkie metody, objęte wspólną nazwą *Zwinnych*, używane były wcześniej, *Manifest* wpisał je tylko w nowe ramy, zidentyfikował je. Na przykład metodyki typu *test-first* znane były już pod koniec lat 90ych, odwracały one „standardową" i – można by pomyśleć – naturalną kolejność pisania oprogramowania, zakładając, że środowisko testowe programu nie powstaje w odpowiedzi na potrzebę sprawdzenia poprawności jego działania, lecz to kod programu powstaje tak, by być w pełni zgodnym z uprzednio narzuconymi ramami testu. W związku z rosnącą rolą testów w tym okresie powstawały również pierwsze automatyczne systemy ich wykonywania na kodzie, wraz z nimi – terminologia określająca m.in. testy jednostkowe i integracyjne. Pierwsze z nich, jak sama nazwa wskazuje, dotyczą testowania wyodrębnionego, atomowego elementu w kodzie – najczęściej pojedynczej metody konkretnej klasy obiektu. Testy integracyjne mają zakres szerszy i sprawdzają współpracę między kilkoma obiektami.

W 2003 r. jeden z sygnatariuszy *Manifestu*, Kent Beck, uporządkował metodyki *test-first* i zaproponował rozwiązanie o nazwie *TDD* (*Test-Driven Development*, ang. dosł. Programowanie Testowo-Sterowane). W tym samym roku Dan North zaproponował metodykę będącą de facto rozwinięciem *TDD* – Programowanie Metodą Behawioralną (*BDD – Behaviour-Driven Development*). Dan North uznał, że *TDD* nie jest dostatecznie intuicyjne[7], nie istnieją wskazówki co testować, jak szczegółowo i kiedy uznać, że test zakończył się pełnym sukcesem. *BDD* miało rozwiązać ten problem – jego głównym

6 http://pl.wikipedia.org/wiki/Manifest_Agile (stan z dnia 12.07.2011)

7 http://dannorth.net/introducing-bdd/ (stan z dnia 12.07.2011)

założeniem jest tworzenie oprogramowania w oparciu nie o suche techniczne testy, ale „scenariusze” przewidujące możliwie jak najdokładniej wszelkie zachowania użytkownika aplikacji, zwłaszcza jego pomyłki i próby uzyskania niedozwolonego dostępu. Takie postępowanie wykracza poza ramy technicznego i technologicznego zarządzania aplikacją – duży nacisk kładzie na komunikację między osobami odpowiedzialnymi za projektowanie funkcjonalności (kierownikami projektu), a tymi odpowiadającymi za implementację (programistami). *BDD* w istocie jest filozofią, której celem jest kierowanie całością rozwoju programu, umożliwienie zleceniodawcom współpracy ze zleceniobiorcą przy całkowitym odrzuceniu sztywnych ram dokumentacji – jest więc kompleksowym rozwiązaniem, wpisującym się w wizję wytwarzania oprogramowania *Zwinnego*. Postuluje, wobec tego, o utworzenie języka pośredniego, między językiem biznesowym (dokumentacji, którą całkowicie odrzucono) i technicznym (który dla filozofii nie jest istotny).

W tym miejscu należy przywołać ponownie osobę Alana Kaya, naukowca odpowiedzialnego nie tylko za podstawy Programowania Obiektowego i pionierskie prace nad graficznymi interfejsami użytkownika. Kay znany jest również w kulturze popularnej jako autor powiedzenia: „Jedyny sposób by przewidzieć przyszłość, to ją wymyślić”, chociaż słowa te wypowiedziane zostały w odniesieniu do całej dziedziny nauki jaką jest informatyka, znakomicie wpisują się w ideę metodyki *BDD* – najlepszy sposób na przewidzenie zachowań użytkownika, odpowiedzi programu, ale również wymagań jakie w przyszłości użytkownik postawi aplikacji, to stworzenie scenariuszy dla każdej możliwości.

Pierwszym frameworkiem Dana Northa był stworzony dla języka Java *JBehave*, prace rozpoczął już w 2003 r[8]. Mimo sukcesu (*Jbehave* rozwijany jest do dzisiaj) North musiał uznać, że język Java nie jest najwygodniejszym do implementacji – dowodem na to jest fakt, że wkrótce porzucił projekt, ale nie filozofię *BDD*. W 2005 r. zainspirowany projektem *RSpec*, platformą *BDD* w języku Ruby autorstwa Dave'a Astelsa, pracował już nad *RBehave*[9]. Chcąc uczynić z *RSpec* kompleksowe rozwiązanie połączył siły z Astelsem, Davidem Chelimskim i Aslakiem Hellesøyem – zgodnie z jego planami *RBehave* przestał istnieć jako osobny projekt i został wcielony do *Rspec*, powoli stającym się standardem w zakresie środowisk testowych platformy *Ruby on Rails* – mimo iż ten popularny framework domyślnie wyposażony jest we własne narzędzia testujące.

Norweg Hellesøy podjął później prace nad maksymalnym uproszczeniem procesu pisania testów i stworzeniem narzędzi, które umożliwiłyby tworzenie bardzo szczegółowych scenariuszy zachowania programu osobom, które nie mają wykształcenia programistycznego. Tak ziścił się pomysł Northa o zaprojektowaniu języka pośredniego między biznesowym a implementacyjnym; powstał *Cucumber* (ang. ogórek), platforma, której głównym przeznaczeniem było generowanie odpowiedniej struktury testów, w odpowiedzi na scenariusze pisane w języku o nazwie *Gherkin* (ang. korniszon), który

8 http://edgibbs.com/2007/12/02/jbehave-and-rspec-history/

9 http://dannorth.net/2007/06/17/introducing-rbehave/

łudząco przypomina potoczny język ludzki, nie tylko angielski – reagować ma on na słowa kluczowe 40 języków, wspieranych przez *Cucumber*.

Metodyka *BDD*, choć zaproponowana w 2003 r. jest nadal bardzo młoda, dopiero od 2010 r. można mówić o dojrzewaniu tej idei, edytory tekstowe zaczynają wspierać rozwiązania *RSpec* i *Cucumber*, powstają też narzędzia dedykowane[10]. Technologia stabilizuje się, przestaje tak dynamicznie zmieniać, co owocuje powstaniem wyczerpujących dokumentacji i publikacji[11] poświęconych zarówno metodyce jak i narzędziom. Nie ma już żadnych technologicznych barier, które uniemożliwiałyby tworzenie testów i scenariuszy osobom nie biorącym udziału w pisaniu kodu.

Nie można jednak mówić o kolejnej nadchodzącej globalnej rewolucji, przede wszystkim dlatego, że *BDD* reprezentuje tylko jeden z wielu sposobów wdrożenia idei Programowania Zwinnego, a samo w sobie jest rozwinięciem pomysłu TDD. Inaczej niż w połowie lat 60ych XX wieku nie istnieje obecnie jeden ośrodek decyzyjny nadający kierunek całej nauce programowania. Z drugiej jednak strony trudno nazywać nowe techniki odłamami „lokalnymi" w dobie internetu, zwłaszcza w dziedzinie, która z tym medium jest tak silnie i naturalnie związana. Przyjęcie danej metodyki jest więc dyktowane tylko osobistymi upodobaniami zarządzających projektem.

10 http://gherkineditor.codeplex.com/

11 http://pragprog.com/book/achbd/the-rspec-book

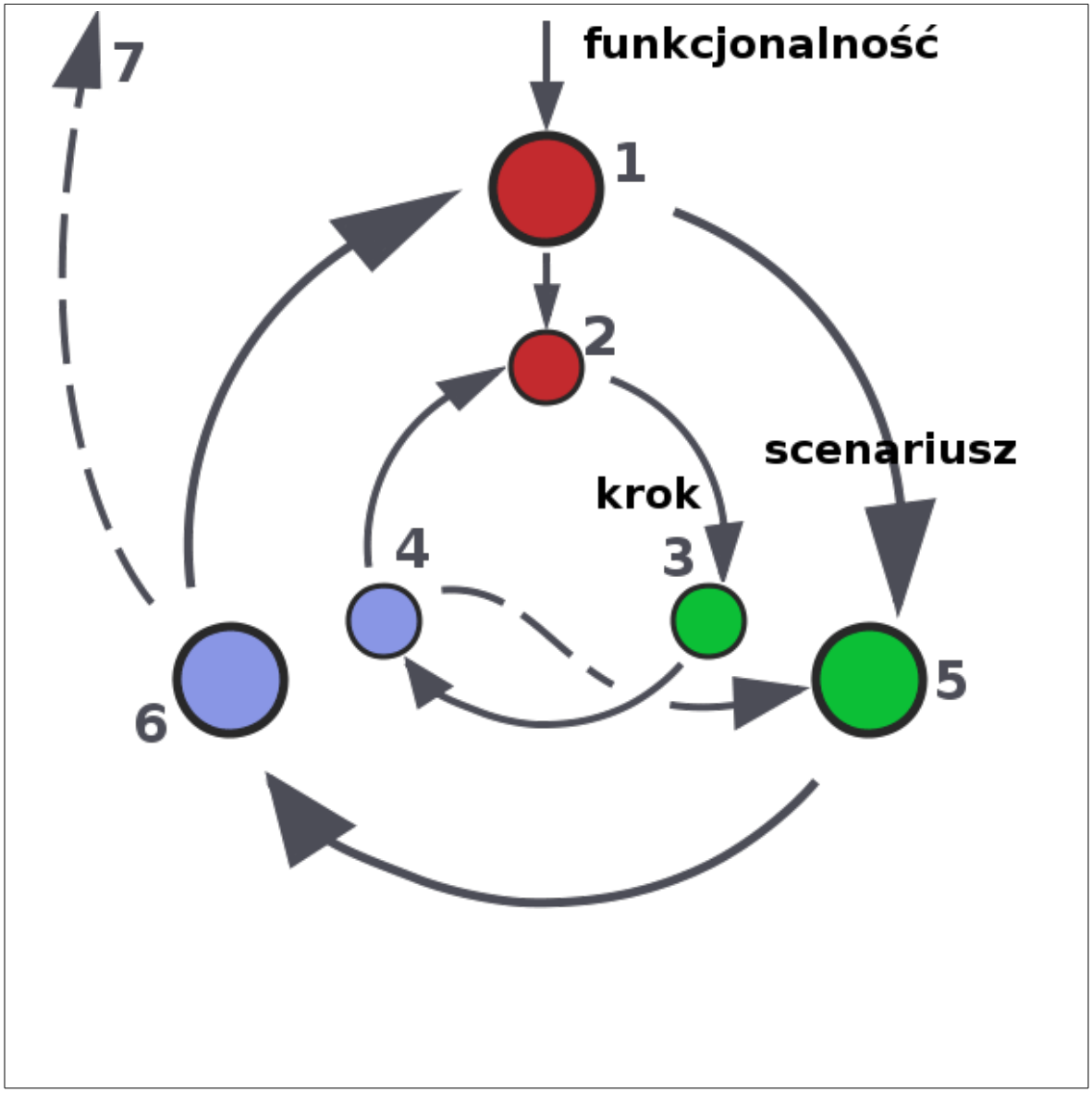
III. Wprowadzenie do *BDD*
Na przykładzie tandemu systemów *Rspec* i *Cucumber*

By móc mówić o metodyce BDD należy najpierw zrozumieć i zdefiniować jej podstawowe terminy, określające trzy kroki konstrukcji całego testu behawioralnego:

- Nadrzędną strukturą testu jest **funkcjonalność** (ang. *feature*). Określa ona ogólne zadanie postawione przed testowanymi elementami. Funkcjonalność powinno się móc nazwać jednym słowem bądź wyrażeniem, jak „uwierzytelnianie", bądź „zarządzanie obrazami w profilu".
- Wewnątrz funkcjonalności definiujemy **scenariusze** (ang. *scenario*). Powinny one przewidywać każdą praktycznie możliwą sytuację użycia funkcjonalności, zarówno poprawną, jak i wynikającą z ludzkiego błędu, czy chęci wymuszenia nieuprawnionego dostępu do zasobów aplikacji (np. atak typu *SQL injection*).
- Scenariusze składają się z **kroków** (ang *step*). Każdy krok powinien być w miarę możliwości atomowy (tzn. odnosić się do jednej logicznie wyodrębnionej cechy aplikacji w danym momencie). Nieoficjalnie te trzy kroki (czy bardziej trzy „zestawy" kroków) określa się mianem *Given-When-Then* (ang. mając [mając dane lub mając do dyspozycji – przyp. autora]; kiedy [podejmuję akcję – przyp. autora]; wtedy [spodziewaj się odpowiedzi – przyp. autora]), połączenia słów kluczowych, używanych do ich opisu w językach domenowych BDD, jak *Gherkin*. Gdybyśmy chcieli tym krokom nadać bardziej oficjalne nazwy, odwzorowujące ich obowiązki, moglibyśmy rozpisać je tak:
  - **Warunki (początkowe)** – określają z jakiej perspektywy podchodzimy do scenariusza, kto jest podmiotem (nazywanym raczej „aktorem") operacji i jakie ma uprawnienia. Jaki jest stan początkowy aplikacji, jakie dane są dostępne przed rozpoczęciem działania. Aktorem scenariusza może być użytkownik, bądź administrator aplikacji, ale również dowolny obiekt w programie, próbujący uzyskać dostęp do innego obiektu.
  - **Akcja** – jakie działanie jest podejmowane przez aktora w scenariuszu. Akcja może być bardzo szczególna, jak np. naciśnięcie konkretnego klawisza w konsoli bankomatu, może być też ogólna, jak np. próba dokonania dowolnego zapisu w bazie danych.
  - **Reakcja** – jaka jest spodziewana odpowiedź systemu na podjętą akcję, jakie działania temu towarzyszą: czy odpowiedź jest sygnalizowana potwierdzeniem, milczącą akceptacją działania aktora, zwróceniem informacji o błędzie, czy wreszcie zwykłym cofnięciem akcji.

Jakakolwiek rozbieżność między odpowiedzią spodziewaną, a otrzymaną w wyniku przeprowadzonej symulacji scenariusza, jest oznaczana jako test niezaliczony. Tylko w wypadku stwierdzenia błędu w formule scenariusza możliwe jest jego poprawienie. Zgodnie z ideą BDD scenariusz niejako zastępuje dokumentację, nie powinien więc być modyfikowany w trakcie tworzenia i rozwijania aplikacji, konieczna jest zatem zmiana (refaktoryzacja) kodu źródłowego, na którym symulacje scenariusza bazują, tak, aby przy założonych warunkach początkowych, określona akcja zakończyła się odpowiednią reakcją systemu.

Ogólny schemat postępowania podczas wytwarzania oprogramowania zgodnego z zadaną funkcjonalnością behawioralną przedstawia się za pomocą poniższego grafu:



rys. 1: Schemat refaktoryzacji kodu zgodnie z opisem funkcjonalności w technologii BDD.

Testujemy całość zadanej funkcjonalności, poprzez sprawdzanie poszczególnych scenariuszy:

1. Scenariusz poddajemy testowi; **jeżeli test wypadnie negatywnie** – a tak będzie w przypadku gdy kod aplikacji nie obsługuje, lub jest niezgodny z chociaż jednym krokiem tego scenariusza – rozpoczynamy testowanie poszczególnych kroków.
2. Zakładamy, że **aplikacja nie jest zgodna z danym krokiem scenariusza**. Przystępujemy do refaktoryzacji kodu.
3. Kod należy modyfikować tak długo, aż dany **krok scenariusza zostanie spełniony, a test wypadnie pomyślnie**.
4. Jeżeli nie jest to ostatni krok scenariusza, **kontynuujemy refaktoryzację w następnym kroku**.
5. Jeżeli był to ostatni krok scenariusza, to **całość jest zgodna z założeniami, a test wypadnie pomyślnie dla całego scenariusza**.
6. Jeżeli nie był to ostatni scenariusz funkcjonalności **kontynuujemy refaktoryzację w następnym scenariuszu**.
7. Jeżeli był to ostatni krok scenariusza, praca nad funkcjonalnością jest **zakończona**.

Testy zwyczajowo przeprowadzane są w testowym środowisku uruchomieniowym, co oznacza, że wszystkie zmiany i operacje są tymczasowe, bazy danych i pamięć programu są czyszczone po zakończeniu wszystkich symulacji, tak, by były przygotowane do przeprowadzenia kolejnych symulacji w takich samych, „sterylnych” warunkach.
W przypadku aplikacji internetowej napisanej z użyciem frameworka *Ruby on Rails* mamy domyślnie do dyspozycji trzy środowiska uruchomieniowe: testowe, rozwojowe i produkcyjne.

Rozumienie terminologii i stosowanie się do odpowiedniej składni jest wymogiem, do którego dostosować się muszą wszyscy uczestniczący w tworzeniu scenariuszy, to główny powód dla którego praktyka stojąca za metodologią BDD została uproszczona do niezbędnego minimum. W przypadku zastosowania platformy *Ruby on Rails* i systemów *Rspec i Cucumber* istnieje oczywiście konieczność stosowania składni języka *Gherkin*:

```
# language: en
Feature: Addition
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers

  Scenario Outline: Add two numbers
    Given I have entered <input_1> into the calculator
    And I have entered <input_2> into the calculator
    When I press <button>
    Then the result should be <output> on the screen

  Examples:
    | input_1 | input_2 | button | output |
    | 20      | 30      | add    | 50     |
    | 2       | 5       | add    | 7      |
    | 0       | 40      | add    | 40     |
```

rys. 2: Przykładowy kod funkcjonalności w języku Gherkin, w angielskiej lokalizacji

IV. Aplikacja webowa.
Funkcjonalność i bezpieczeństwo

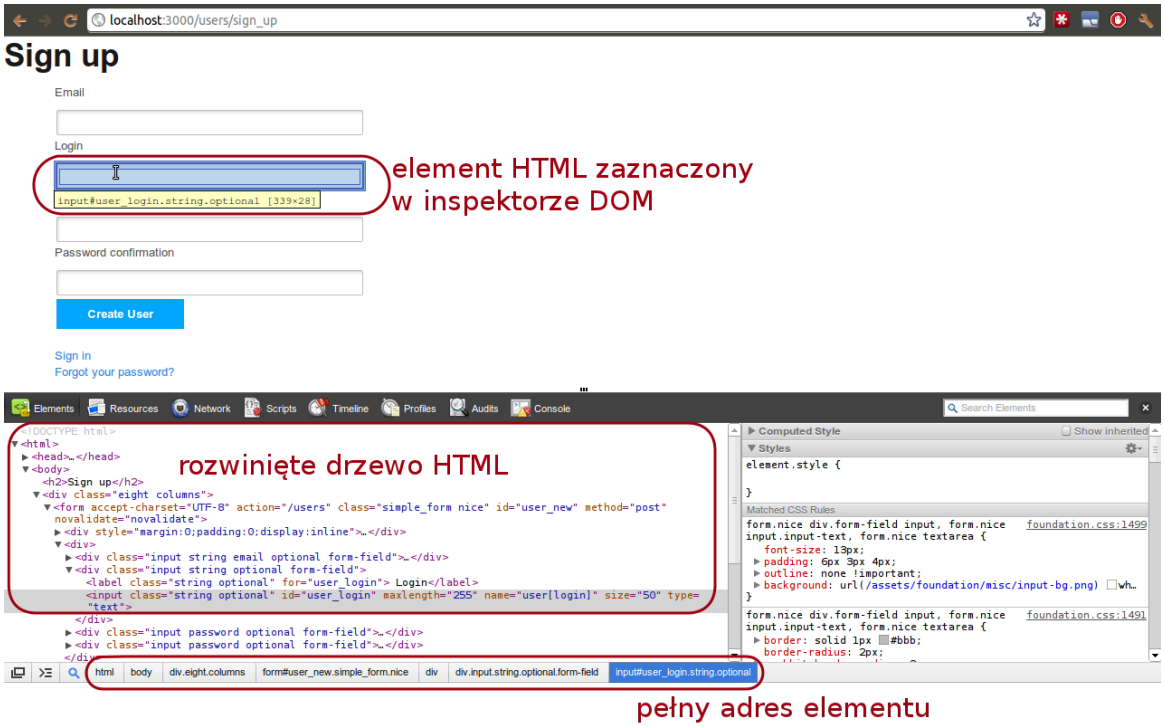
Aplikacja webowa (sieciowa) w erze Web2.0 jest zupełnie innym tworem niż statyczna „witryna” internetowa. Mimo udostępnienia webowych baz danych poprzez wprowadzenie i ciągły rozwój takich technologii jak *PHP*, *asp.net*, *Python Django* i *Ruby on Rails*, wciąż najważniejszym narzędziem w budowaniu czy to statycznych dokumentów, czy rozbudowanych, interaktywnych aplikacji, jest język znaczników *HTML*, oparty na omawianej wcześniej technologii hipertekstowej; podstawowym protokołem komunikacji serwera z maszyną kliencką jest HTTP, a programem klienckim jest przeglądarka internetowa. Bazą wyjściową wszystkich tych technologii i frameworków sieciowych jest relacyjna baza danych typu SQL, stanowiąca serce aplikacji.

Wraz z ewolucją dokumentów do aplikacji rosło zagrożenie ze strony internetu dla przeglądarki, a przez nią – dla systemu operacyjnego użytkownika. Zagrożenie istniało też po drugiej stronie: w postaci użytkowników internetu szukających nieuprawnionego dostępu do danych poufnych – hakerów. Pierwszym „naturalnym odruchem” w ewolucji aplikacji webowych było stworzenie narzędzi do autoryzacji i autentykacji użytkownika. Protokół HTTP udostępnia dwa podstawowe rodzaje „żądań” jakie przeglądarka może wysyłać do serwera: GET - służące do pobierania danych z serwera wysyłane jest w celu pobrania każdego składnika dokumentu HTML, w tym pobrania samego dokumentu i zawartych w nim obrazów, oraz POST – używanie kiedy kliencka przeglądarka wysyła dane (w postaci parametrów HTTP) do serwera.

Teoretycznie żądanie POST nie podlega żadnym ograniczeniom; jest to łańcuch znaków wysyłany do serwera, który otrzymane dane parsuje i wykonuje na nich zadane operacje. Ta swoboda w dobieraniu parametrów oznacza jednak, że w praktyce poświęca się POSTowi wiele uwagi i programuje się serwer tak, by rozpoznawał i odrzucał szkodliwe parametry-tokeny, zanim przejdzie do wykonywania jakichkolwiek akcji na bazie danych. W ten sposób serwer broni się przed atakami iniekcyjnymi, wśród nich przed bardzo popularnymi, acz prymitywnymi atakami SQL Injection, które polegają na podaniu jako parametru fragmentu zapytania do bazy SQL do której serwer ma dostęp. Jeżeli warstwa aplikacyjna serwera nie potrafi rozpoznać zagrożenia i odrzucić takiego parametru, możliwe jest pośrednie wydawanie poleceń bazie danych, uzyskiwanie dostępu do danych wrażliwych, jak również usuwanie danych i modyfikowanie całych struktur bazy.

Wraz z rozwojem technologii przeglądarkowych pojawił się język Javascript, w skrócie zwany po prostu JS. Został stworzony jako ECMAScript przez programistę Brendana Eicha pod koniec 1995 r. przeszedł drobne zmiany i jako nowa implementacja, pod nową nazwą, został wprowadzony przez firmę Netscape w ich przeglądarkach w roku

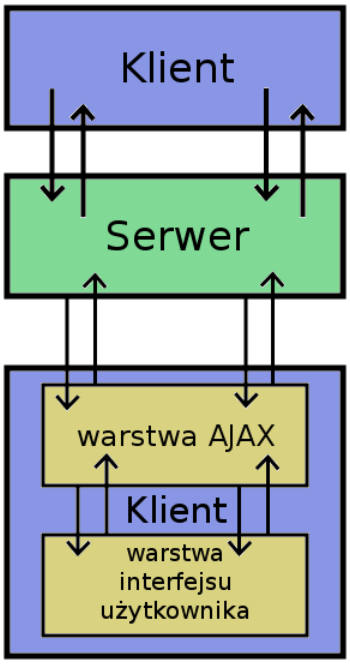
1996. Również wtedy opracowano pierwszy standard *DOM* (ang. *Document Object Model*), technologii która umożliwia językowi Javascript manipulowanie obiektami składowymi (elementami) dokumentu HTML w przeglądarce użytkownika w czasie rzeczywistym.



rys. 3: Struktura dokumentu HTML widziana z perspektywy inspektora DOM (w przeglądarce Google Chrome)

Javascript to interpretowany język obiektowy, o dynamicznym typowaniu, który jako „zasób" dokumentu HTML importowany jest przez przeglądarkę w postaci pełnych skryptów w odpowiedzi na żądanie GET. Umożliwia autorom aplikacji tworzenie bardziej dynamicznych efektów i zachowań dokumentu HTML, uzyskując dostęp do przeglądarki klienckiej i traktując ją jako swoje środowisko. Może też być dodatkowym narzędziem do komunikacji klienta z serwerem – co prawda jego działanie również opiera się na żądaniach GET i POST, ale żądania te mogą być wysyłane w odpowiedzi na niemal dowolną akcję użytkownika przeglądarki, a nie tylko wywołanie hipertekstowego linku, jak to ma miejsce w statycznym dokumencie HTML.

W 2000 r. JavaScript jako narzędzie web-developerskie pojawił się w nowym wcieleniu: technologia AJAX sprawiła, że JavaScript mógł wysyłać żądania do serwera i odbierać jego odpowiedzi asynchronicznie, czyli niezależnie od żądań POST/GET wysyłanych przez użytkownika z poziomu głównego dokumentu HTML. W praktyce każdy element dokumentu może komunikować się z serwerem dwustronnie, bez potrzeby korzystania z HTML-owych formularzy gdy coś wysyła i bez potrzeby odświeżania całego dokumentu, gdy coś odbiera.



rys. 4: Schemat komunikacji z użyciem technologii AJAX.

Pojawienie się *AJAXa* umożliwiło autorom budowanie aplikacji w pełni interaktywnych, zadziwiających dynamiką i ilością elementów które można obsłużyć „pozostając na tej samej stronie", czy – jak nazwalibyśmy to już zgodnie z terminologią ery Web2.0 – „pozostając w tym samym widoku". Jest oczywiście druga strona medalu takiego rozwiązania – nowy sposób pisania i obsługiwania aplikacji wymusił budowę szybszych i silniejszych serwerów, ponieważ ogromnie wzrosła ilość żądań wysyłanych przez takie aplikacje. Logicznym następstwem zwiększenia ilości połączeń klient-serwer jest wzrost zagrożenia jakie niesie ze sobą niezabezpieczenie tej komunikacji.

Kwestie bezpieczeństwa potraktuję szerzej w rozdziale o podstawach praktycznego bezpieczeństwa przykładowej aplikacji.

Kiedy w 1995 roku powstał język *PHP*, pierwsze narzędzie do tworzenia aplikacji internetowych godnych miana Web2.0, historia webowych narzędzi dopiero się rozpoczynała. W 2002 roku Microsoft wprowadził na internetowy rynek swoje rozwiązanie: *ASP.NET*, część większego frameworka *.NET* (wymawiane: „*dot-net*"), kompletne narzędzie programistyczne, oparte na uzupełniających się wzajemnie językach, sprawdzonym i zaufanym *Visual Basic* oraz nowoczesnym *C#* (czytane „*C sharp*").

Dla niniejszej pracy najważniejszą technologią jest powstała w 2004 roku platforma *Ruby on Rails*, autorstwa duńskiego programisty Davida Heinemeiera Hanssona. Hansson jako podstawę swojego rozwiązania wykorzystał interpretowany język obiektowy *Ruby*. Hansson wcale nie próbował stworzyć nowej technologii, a zaledwie napisać jedną aplikację na własny użytek; doszedł jednak do wniosku, że zastosowanie Ruby sprawiło,

że aplikację rozwija się bardzo szybko i przyjemnie, twierdził, że pisanie kolejnych narzędzi ułatwiających dalszą pracę po prostu sprawiało radość[12], przejrzystość kodu umożliwiała łatwe tworzenie dokumentacji. Z tego powodu Hansson zdecydował się na udostępnienie swojego rozwiązania w internecie pod nazwą *Ruby on Rails*.

12 Wywiad z Davidem Heinemeierem Hanssonem: http://bigthink.com/ideas/21596

V. Charakterystyka języka *Ruby*
oraz platformy *Ruby on Rails*

Stworzony w 1995 roku przez japońskiego programistę, Yukihiro Matsumoto, język programowania *Ruby*, to interpretowany, dynamicznie typowany, w pełni obiektowy język skryptowy. Oparty na doświadczeniach programisty wyniesionych z pracy z innymi językami, jak np. *LISP*, *Perl* czy *Smalltalk*. posiada kilka unikalnych cech, które sprawiają że dla niektórych programistów Ruby stanowi osobną klasę w kategorii ergonomii rozwijania kodu:

Czytelna i przyjazna składnia – (ang. *syntax*) już na pierwszy rzut oka wyróżnia język *Ruby*, zwłaszcza wśród języków opierających swoją składnię na *C* (jak *C++*, *C#*, *Java* i *ActionScript*). Wyraźnie widać, że zaproponowane rozwiązanie ma przede wszystkim służyć programiście w swobodnym wyrażaniu swoich myśli:

```
exit unless "restaurant".include? "aura"
```

Wystarczy przeczytać na głos powyższy kod źródłowy, żeby – mając najbardziej podstawową znajomość języka angielskiego – zrozumieć jego znaczenie dla programu: algorytm zakończy działanie, chyba że w wyrazie „restaurant” znajduje się ciąg znaków „aura”. Wyrażenie warunkowe `unless` jest odwrotnością warunku `if`, dostępnego praktycznie w każdym języku wysokiego poziomu. Nawet początkujący programista powinien potrafić napisać dowolne wyrażenie warunkowe i jego zaprzeczenie, korzystając z praw DeMorgana. *Ruby* udostępnia jednak wyrażenie `unless` jako dogodność dla użytkownika, jego znaczenie dla realizacji kodu jest identyczne z zaprzeczeniem `if`, więc może wydawać się zbyteczne, ale w języku *Ruby* wygoda programisty jest stawiana na pierwszym miejscu.

Filozofia programowania, w której najważniejszy jest użytkownik języka, czyli autor programu jest obecna i ma wpływ na wszystkie rozwiązania przyjęte w *Ruby*.

Iteratory i bloki kodu – pozwalają na łatwe pisanie powtarzalnego i rekurencyjnego kodu. Tylko z pozoru są podobne do pętli znanych z innych języków, są jednak metodami, które przyjmują fragment kodu, otoczony nawiasami klamrowymi lub słowami kluczowymi `do` / `end`, jako argument:

```
10.times { puts "a"}
```

Wypisuje znak „a” dziesięciokrotnie, ponownie zwraca na siebie uwagę czytelna

i komfortowa nazwa metody `times` sugerująca wykonanie czynności wielokrotnie.

Kod wywołujący metodę iteratora może do niej przekazać dane i tak:

```
10.times { |x| puts x }
```

...będzie przy każdym iteracyjnym kroku przekazywać licznik do bloku, jako zmienną *x*, każdy krok iteracyjny natomiast wypisuje od nowej linii wartość tej zmiennej, oczywiście zaczynając od zera.

W zależności od klasy obiektu dostępny jest odpowiedni zestaw iteratorów, dla liczb całkowitych i ich zasięgów[13] dostępne są prostsze wyliczenia podobne do pętli for z innych języków, dla łańcuchów znaków dostępne są metody iterujące po każdym znaku, bądź każdym bajcie, tablice i tablice asocjacyjne (hasze) można przeszukiwać po każdym elemencie, po parach klucz/wartość, tylko po elementach, które spełniają podany w bloku kodu warunek, niektóre iteratory wykonują kod na pożądanych elementach, inne zwracają wynikową tablicę itd. Praktycznie dowolne powtarzalne zadanie, jakie chcielibyśmy wykonać przy pomocy pętli, w języku *Ruby* można zapisać prościej i krócej za pomocą odpowiedniego iteratora.

System zarządzania pakietami *RubyGems* – integrujący się z powłoką systemu operacyjnego program ułatwiający instalowanie, usuwanie, uaktualnianie i przede wszystkim wykorzystywanie z poziomu kodu, bądź z poziomu systemu operacyjnego pakietów *Ruby*, zwanych *gemami* (ang. *gem* – klejnot). Pod szerokim pojęciem pakietu rozumiemy przede wszystkim rozliczne biblioteki i wtyczki, które można dołączyć i wykorzystać we własnym kodzie, ale niektóre pakiety są kompletnymi programami, interfejsami typu API, a nawet frameworkami – jak w przypadku *Ruby on Rails*, które integrują się z systemem operacyjnym i mogą pracować jako samodzielne aplikacje.

13 Klasa `Range` może przechowywać liczby całkowite bądź znaki w ich „naturalnej” kolejności, np. `(12..24)` to wszystkie liczby całkowite od 12 do 24 włącznie, natomiast `(„a”..”n”)` przechowuje wszystkie małe litery alfabetu w podanym zakresie.

Ruby on Rails – platforma ułatwiającą pisanie aplikacji webowych przy pomocy języka *Ruby*, jest całym systemem wzajemnie powiązanych gemów. Poza własnym pakietem, zapewniającym integrację ze środowiskiem i powłoką, najważniejszymi jej częściami są:

- *Rack* – api serwera sieciowego, które od czasu powstania stało się standardowym rozwiązaniem wykorzystywanym w pisaniu programów serwerowych i klienckich (jest np. podstawą innego frameworka aplikacji webowych o nazwie *Sinatra*),
- Trzy biblioteki ułatwiające pracę z silnikami bazodanowymi *SQLite*, *MySQL* i *PostGreSQL* (platforma *Rails* wspiera te trzy silniki domyślnie, ale istnieją biblioteki umożliwiające pracę z innymi bazami, jak np. *Oracle*).
- *ActiveRecord* – gem udostępniający funkcjonalność ORM (ang. *Object-Relational Mapping* – mapowanie obiektowo-relacyjne), czyli mapowanie relacji z bazy danych do postaci obiektu języka *Ruby;* dzięki temu zaciągnięta z bazy danych „krotka" – bądź ich tablica – wraz z całą zawartością, czyli wartością każdej kolumny, jest przez program traktowana jako obiekt odpowiedniej klasy, udostępniający metody wspólne dla wszystkich obiektów bazodanowych (np. czytanie danych, porównywanie, nadpisywanie) jak i metody napisane implicite dla danego rodzaju krotki (np. wiersz zawierający dane użytkownika, zaciągnięty z bazy danych i zmapowany wewnątrz aplikacji jako obiekt klasy *User*, może nabyć metody odpowiedzialne za logowanie i wylogowywanie danego użytkownika do aplikacji i zapisywanie niepoufnych danych w sesji przeglądarki).
- *Rake* – program automatyzujący procesy, będący w dużym uproszczeniu wersją programu *Make* dla języka *Ruby*. Jego najważniejszą funkcją jest wykonywanie predefiniowanych zadań (zwanych *Raketask*-ami) w przestrzeni roboczej aplikacji. Najczęściej kojarzony jest z wykonywaniem operacji na bazie danych, które nie są wynikiem działań użytkowników aplikacji (np. wykonywanie okresowych kopii zapasowych).

Poza cechami technologicznymi, istotna jest też architektura systemu oparta o schemat MVC (ang. *Model-View-Controller* – model-widok-kontroler). Ponieważ szczegółowy opis wszystkich cech tej architektury wykracza poza temat pracy, skupimy się na najważniejszych, które zakładają wyizolowanie logiki domenowej od części interfejsowej użytkownika, co – przy założeniu że każdy z trzech tytułowych elementów wykorzystywany jest zgodnie z przeznaczeniem (czego *Ruby on Rails* odgórnie nie narzuca) – pozwala na rozwijanie każdej części niezależnie, ułatwia proces testowania i usuwania błędów (debugowania), jak również stanowi podstawę tworzenia całych rodzin aplikacji, których rdzeń jest wspólny, a zmiany dokonuje się tylko w logice biznesowej.

`<SCHEMAT ARCHITEKTURY MVC>` **`//TODO`**

W zależności od implementacji architektury obowiązki jej elementów mogą rozkładać się bardzo różnie, dla Ruby on Rails przedstawiają się następująco:

Model – to klasa obiektu reprezentującego relacyjną krotkę bazodanowej. Każdy model w aplikacji *Rails* dziedziczy z klasy podstawowej udostępnionej przez omówiony wcześniej interfejs *ORM ActiveRecord*. Dzięki dziedziczeniu mamy dostęp do wszystkich najważniejszych metod związanych z manipulowaniem krotką w bazie danych za pomocą jej obiektu, ale również możemy opisać całą logikę jaką obiekt musi posiadać wewnątrz aplikacji.

Widok – reprezentacja interfejsu użytkownika. W aplikacji *Ruby on Rails* są to dokumenty html i ich szablony w formacie *eRuby* (ang. *embed Ruby* – zagnieżdzony *Ruby*) rozpoznawane po rozszerzeniu `*html.erb` bądź starszym `*.rhtml`. Pliki *eRuby* pozwalają na dopełnianie dokumentu html logiką języka *Ruby*, analogicznie do szablonów wykorzystywanych w technologiach *PHP*, *ASP* i *JSP*.

Widok w momencie wczytywania ma dostęp do zmiennych zdefiniowanych dla niego przez kontroler.

Kontroler – klasa zarządzająca „ruchem” w aplikacji, obsługuje żądania GET/POST wysyłane przez kliencką przeglądarkę. Spośród wymienionych elementów jest najważniejszym pośrednikiem między klientem a serwerem. Ze względów wydajnościowych, jak i ergonomii rozwijania aplikacji, każda funkcjonalność, dająca się logicznie wydzielić od pozostałych, powinna być reprezentowana w aplikacji przez własny kontroler. Standardem jest grupowanie metod obsługujących wszystkie zapytania manipulujące konkretnym modelem (często określane skrótem *CRUD*; ang. *Create-Read-Update-Destroy* – twórz-czytaj-nadpisuj-usuń), w obrębie jednego kontrolera. Jest to zgodne ze wzorcem architektury *REST* (ang. *Represnetational State Transfer* – transfer stanu reprezentacyjnego), której głównym aktorem jest „zasób” (ang. *resource*), reprezentujący obecny stan aplikacji w interfejsie użytkownika. Biorąc pod uwagę bezstanową naturę protokołu http, zasób staje się ważnym medium biorącym udział w przenoszeniu przeglądowego stanu aplikacji z widoku do widoku, bądź w utrwalaniu stanu w bazie danych.

Jak wspomniałem powyżej *Ruby on Rails* integruje się z powłoką środowiska w którym pracuje. Efektem takiej integracji jest udostępnianie zestawu komendy zwanych generatorami, które ułatwiają żmudne procesy zarządzania elementami składowymi aplikacji, takie jak budowanie nowego modelu czy kontrolera, Część gemów i wtyczek, przeznaczonych do pracy z aplikacją Rails dodaje swoje własne generatory.

VI. *Model Iteracyjno-Przyrostowy* a metodyka *BDD*

Metodyka behawioralna, jako jedna z metodyk potomnych Zwinnego Modelu Projektowania, wykazuje analogie do innych metod używanych w ramach Projektowania Zwinnego. Przede wszystkim dobrze daje się zastosować w Zunifikowanym Procesie Projektowania (ang. *Unified Process*) oraz w metodyce *Scrum*, najczęściej stosowanymi procesami Modelu Iteracyjno-Przyrostowego (ang. *Iterative and incremental development*) a jej elementy można sprowadzić do szeroko stosowanych notacji języka UML jak Przypadki Użycia, czy Diagramy Interakcji / Komunikacji.

Ideą Modelu przyrostowego jest projektowanie i rozwijanie aplikacji w autonomicznych krokach, zwanych „iteracjami", z których każdy ma za zadanie wprowadzić określoną na jego początku funkcjonalność w ścisłych ramach czasowych. Powstanie stabilnej wersji programu jest efektem przeprowadzenia określonej z góry ilości iteracji.

Każda z iteracji musi spełniać następujące wymagania:

- równy czas trwania pojedynczej iteracji; zazwyczaj jest to okres od jednego tygodnia do miesiąca.
- jasno określone wymagania i cele wprowadzanych zmian funkcjonalności i poprawek
- rozpoczęta iteracja jest tematem zamkniętym; wprowadzanie zmian – za wyjątkiem całkowitego zaniechania wykonania jednego lub części ze składających się na nią zadań – jest niemożliwe
- musi być kompletna; planowanie, budowa i poparcie testami danej funkcjonalności muszą zamknąć się w tej samej iteracji
- efektem jest stabilne wdrożenie; po przeprowadzeniu iteracji klient otrzymuje działającą aplikację, wzbogaconą o założone funkcjonalności

oraz najważniejsze:

- termin wdrożenia jest nieprzekraczalny; w przypadku napotkania problemów jedyną możliwością jest zaniechanie części zmian wymaganych przez iterację i ewentualne przełożenie ich na iterację następną, wobec czego wdrożenie będzie uboższe o pominiętą zmianę, ale termin zostanie dotrzymany.

Metody UP powstały w odpowiedzi na wprowadzony jeszcze w latach 60tych

Model Kaskadowy, który zakładał przystąpienie do pracy tylko po uprzednim skompletowaniu dokumentacji i rozplanowaniu wszystkich przypadków użycia. Analizy projektowania potwierdzały, że Model Kaskadowy z czasem przestał być wydajny[14]. Trudno określić co dokładnie przesądziło o jego niepowodzeniu, ale na pewno wpływ na to miało pojawienie się nowych, dynamicznych narzędzi programistycznych, języków obiektowych i związanych z nimi metodyk projektowania obiektowego, które nie tylko radziły sobie z rozwijaniem programu w oparciu o ciągle zmieniające się wymagania, ale nawet zakładały, że taki właśnie sposób współpracy z klientem – formalniej: interesariuszem. Dodatkowo tysiące przeprowadzonych badań wykazały, że duża część funkcjonalności, zaprojektowanych w oparciu o Model Kaskadowy, jest nieużywana. Testy mówią o 45% funkcjonalności, które nie są wykorzystywane nigdy i o 19%, które wykorzystywane są rzadko[15].

```
<diagram z książki Larmana, str. 86> //TODO
```

Prosty wniosek jest taki, że nie jest możliwe określenie wszystkich wymagań systemu zanim rozpoczną się prace i zanim chociaż jego część zostanie oddana interesariuszom i użytkownikom – co za tym idzie, czas wykorzystany na projektowanie i budowanie tych funkcjonalności można wykorzystać inaczej.

Ważnym elementem Modelu Iteracyjno-Przyrostowego jest rozróżnienie poszczególnych faz iteracyjnych, z których najłatwiejsza do wydzielenia, a zarazem najbardziej potrzebna jest faza „opracowywania", podczas której określa się wymagania postawione projektowi, cel do którego projekt ma dążyć oraz serię kilku pierwszych iteracji, których efektem ma być dość podstawowa, ale funkcjonalna wersja programu. Ta grupa iteracji zwanych iteracjami podstawowymi bądź początkowymi, to jedyny formalny przypadek, kiedy iteracje odbiegają od zasady, mówiącej, że każda iteracja kończy się wdrożeniem. Iteracje początkowe, zanim zespół przystąpi do budowania zaplanowanej aplikacji, muszą jeszcze określić technologię i wymagania sprzętowe, wyznaczając tym samym punkt wyjścia dla wszystkich następnych iteracji, są więc iteracjami najważniejszymi z punktu widzenia kształtu projektu zarządzanego zwinnie.

Nawiązując do tej teorii w następnych rozdziałach zaprojektuję aplikację korzystając z Modelu Iteracyjno-Przyrostowego i zbuduję ją wykorzystując technologie związane z metodyką BDD.

14 Craig Larman „UML i Wzorce Projektowe"

15 Craig Larman „UML i Wzorce Prjektowe", rozdz 5 – Ewoluujące Wymagania

VII. Iteracja 0:
Cel projektu i wymagania aplikacji

Opis aplikacji, nawet w iteracjach początkowych, powinien być tworzony zgodnie z paradygmatem zwinnego projektowania, co oznacza, że musi być dość zwięzły, konkretny co do celów projektu, ale bardzo ogólny co do środków, a nawet nie mówiący o nich nic.

Celem projektu jest utworzenie demonstracyjnej aplikacji sieciowej, której głównym elementem ma być forum internetowe. Konieczne jest utworzenie modułu uwierzytelniania i autoryzowania użytkowników, tak, aby można ich podzielić według ról: Administrator, Moderator, użytkownik.

Zadania:

1. *Zidentyfikować potrzebne technologie*
2. *Przygotować system operacyjny serwera pod obsługę aplikacji*
3. *Zadbać o niezmienność technologii będących podstawą działania aplikacji*

Ten ogólny opis projektu jest jak najbardziej dozwolony, daje już asumpt do wybrania odpowiedniej technologii. Ponieważ wszystkie zainteresowane strony projektu: interesariusze, menadżerzy projektu i programiści zdają sobie doskonale sprawę, że to dopiero pierwszy krok. Wykonawcom projektu logika i doświadczenie każe spodziewać się kolejnych bardzo „standardowych" funkcjonalności zlecanych przez interesariusza, jak możliwość edycji i usuwania postów, załączania plików do postów, formatowania wypowiedzi za pomocą kodu HTML – wszystko to zostanie rozpisane na kilka początkowych iteracji projektu i dołączone do pierwszego funkcjonalnego wdrożenia.

Iteracja zerowa ma za zadanie wyłonić odpowiednią technologię i narzędzia spełniające postawione wymagania. Wszystkie prace przygotowawcze w metodykach zwinnych mają służyć jak najszybszemu przejściu do programowania aplikacji. O kształcie dodawanych funkcjonalności i technologii w nich wykorzystanych decyduje się na bieżąco, co oznacza, że wybór dominującej technologii i sposobu jej wersjonowania ma ogromne znaczenie. W mojej aplikacji korzystam, jak wielokrotnie wspominałem, z frameworka *Ruby on Rails* konkretnie w wersji 3.1. W wersji tej standardem jest zintegrowanie ze środowiskiem Rails interpretera języka *Coffescript*. Jest to język transkompilowany do czystej postaci języka JavaScript. Powstał w celu zwiększenia czytelności kodu i możliwości przetwarzania obiektów. Ponieważ efektem wynikowym

kompilowania plików *Coffee* jest poprawny *JavaScript*, jest to technologia w pełni zgodna z najczęściej wykorzystywanymi bibliotekami Javascript jak *jQuery* czy *Backbone*. Inną nowością wprowadzoną w wersji 3.1. platformy „RoR", jest domyślne wykorzystywanie *Sass*, domenowego języka prezentacji dokumentów HTML. *Sass* jest dla CSS tym, czym *Coffeescript* dla *JavaScriptu* – metajęzykiem kompilowanym do innego języka, stworzonym w celu zwiększenia czytelności i użyteczności technologii zaufanej i rozwijanej od wielu lat.

Uzbrojeni w taką wiedzę możemy być spokojni, że wybór dynamicznie rozwijającej się platformy opartej o nowoczesny język *Ruby*, w pełni kompatybilnej ze standardowymi technologiami, a nawet rozszerzającej ich możliwości, jest wyborem trafionym i nie przysporzy problemów w następnych iteracjach.

Należy też pamiętać o tym, że siła platformy *Rails* wynika ze współpracy wielu różnych technologii, z których każda może być rozwijana w innym tempie. Najlepszym rozwiązaniem zapewniających stabilność wersji wszystkich użytych gemów jest wspieranie się programami *RVM* i *Bundler*. *RVM* (*Ruby Version Manager*) umożliwia instalację wielu wersji interpretera języka *Ruby* i używania wybranej w zależności od projektu. Każda osobno zdefiniowana wersja ma też własny system *Rubygems*, a co za tym idzie – własny zestaw dodatkowych narzędzi. Na poziomie pojedynczego projektu natomiast, to właśnie wymieniony wcześniej *Bundler* dba o korzystanie z określonych w pliku konfiguracyjnym odpowiednich wersji gemów. W przypadku nowszych wersji aplikacji *Ruby on Rails* (od wersji 3.0 w górę) korzystanie z tego systemu jest wymagane, musi być więc on dostępny dla danej wersji interpretera przed wywołaniem generatorów *Rails*.

Ostatecznie dobór najbardziej potrzebnych, startowych technologii, przedstawia się następująco:

- *Interpreter języka Ruby – **MRI Ruby 1.9.2***
- *System zarządzania wersjami interpretera języka Ruby – **RVM 1.6.31***
- *System zarządzania dodatkami języka Ruby – **RubyGems 1.8.6***
- *Platforma do tworzenia aplikacji sieciowych – **Ruby on Rails 3.1***
- *System zarządzania dodatkami projektów Ruby – **Bundler 1.0.21***
- *Platforma testów behawioralnych – **RSpec 2.7.0***
- *Generator testów behawioralnych – **Cucumber 1.1.1***
- *Biblioteka skryptów Javascript – **jQuery 1.3.2***

Ważnym elementem projektu jest też dobór systemu wersjonowania oprogramowania. O wyborze technologii zadecyduje przede wszystkim model repozytorium: scentralizowany lub rozproszony. W ostatnich latach dużą popularność zdobywają systemy rozproszone, a zwłaszcza program *Git*, autorstwa Linusa Torvaldsa. Rozproszenie repozytorium zwiększa bezpieczeństwo projektu poprzez pozbycie się

problemu jednego, centralnego repozytorium, którego utrata bądź awaria, oznaczać może całkowity upadek projektu; ze względu na to repozytoria centralne rutynowo zabezpiecza się kopiami zapasowymi. W przypadku repozytorium rozproszonego każdy pracujący z repozytorium przechowuje lokalną historię całego projektu, na podstawie której można odtworzyć mniejszą, bądź większą część oprogramowania w razie wypadku. Inną ważną zaletą jest możliwość pracy i kontynuowania wersjonowania własnych postępów na poziomie lokalnym nawet w sytuacji utraty połączenia sieciowego ze współpracownikami. Pracę można normalnie wykonywać na maszynie lokalnej, a lokalnymi zmianami wymienić się z resztą zespołu w dowolnej chwili gdy połączenie zostanie nawiązane. System *Git* posiada tę właściwość, że zamiast zapisywać kolejne wersje plików, zapisuje skompresowaną historię zmian, wraz ze wszystkimi rozgałęzieniami repozytorium, dzięki czemu jego repozytoria są mniej obszerne, a zaciąganie i wysyłanie zmian trwa krócej niż w przypadku innego popularnego systemu *SubVersion*, głównego konkurenta Git. Na koniec należy wspomnieć o serwisie *Github*[16], który umożliwia darmowe tworzenie publicznych repozytoriów, które umieszcza się w sieci, udostępnia do tego wiele niezwykle przydatnych narzędzi. Takie rozwiązanie ogromnie ułatwia zespołową pracę nad małymi, jawnymi projektami.

Wobec tego do wersjonowania niniejszego projektu będę używał programu Git. W przypadku powiększenia zespołu wprowadzenie nowego członka w historię rozwoju aplikacji jest bardzo ułatwione właśnie dzięki aplikacji *Github*, mogę też dzielić się problemami i rozwiązaniami ze społecznością użytkowników serwisu, jak i spoza niego. Tak długo jak repozytorium jest publiczne, dostęp do jego odczytu ma każdy.

W związku z postawionymi zadaniami, wynikiem przeprowadzenia tej iteracji powinien być serwer z zainstalowanym programem *RVM* (wraz z odpowiednią wersją interpretera *Ruby* i gemem *Bundler* dla tej wersji) oraz systemem wersjonującym Git.

16 www.github.com

VIII. Iteracja 1:
Przygotowanie platformy

Zadania:

1. *Przygotować platformę języka Ruby pod aplikację Ruby on Rails w wersji 3.1*
2. *Przygotować repozytorium GIT*
3. *Zadbać o niezmienność wersji użytych technologii (zamrożenie za pomocą systemu „Bundler")*

Mając do dyspozycji przygotowany system operacyjny serwera, obsługujący określoną wersję interpretera języka Ruby przystępujemy do budowy faktycznej aplikacji. Pierwszą rzeczą jest instalacja platformy Ruby on Rails i dołączenie jej do repozytorium dodatków (gemów) związanego z konkretną wersją interpretera:

```
$ rvm use ruby-1.9.2-p290-patched
$ gem install rails -v=3.1.0
```

Jak pisałem wcześniej platforma Rails integruje się z systemem operacyjnym, udostępniając komendy zwane generatorami, jeden z nich zajmuje się automatycznym procesem przygotowania nowej aplikacji Ruby on Rails:

```
$ rails new mgr
```

Generator stworzy pustą aplikację przygotowaną pod działanie najnowszej, zainstalowanej wersji frameworka. Warto od razu w katalogu projektu przygotować plik `.rvmrc`, w którym wstawiamy linijkę `rvm use ruby-1.9.2-p290-patched` dzięki czemu wchodząc do katalogu aplikacji zawsze przełączymy się na odpowiednią wersję interpretera i otrzymamy dostęp do odpowiednich gemów.

Następnym ważnym krokiem jest zadbanie o repozytorium gemów, wykorzystywanych przez daną aplikację. Do tego służy wygodny *Bundler*, a jego punktem odniesienia, zawierającym informacje o bibliotekach które ma dla danego projektu obsługiwać, jest plik `Gemfile` w głównym katalogu aplikacji. W przypadku świeżego projektu jego zawartość przedstawia się następująco:

```
source 'http://rubygems.org'

gem 'rails', '3.1.0'

# Bundle edge Rails instead:
# gem 'rails',     :git => 'git://github.com/rails/rails.git'

gem 'sqlite3'


# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails', "  ~> 3.1.0"
  gem 'coffee-rails', "~> 3.1.0"
  gem 'uglifier'
end

gem 'jquery-rails'

# Use unicorn as the web server
# gem 'unicorn'

# Deploy with Capistrano
# gem 'capistrano'

# To use debugger
# gem 'ruby-debug19', :require => 'ruby-debug'

group :test do
  # Pretty printed test output
  gem 'turn', :require => false
end
```

Definiowanie odpowiedniego gema jest banalne i zawiera się w jednej linii, co widać na przykładzie:

```
gem 'rails', '3.1.0'
```

Linijka zawiera informację o nazwie biblioteki z której korzystamy i konkretnej wersji – jeżeli chcemy taką wymusić. W przypadku dodania nowego wpisu bez podawania wersji oprogramowania, uruchomienie *Bundlera* automatycznie ściągnie najnowszą wersję gema, doda ją do repozytorium używanego obecnie interpretera i podłączy do niej aplikację. Dodatkowo definiować można grupy użycia gemów, dzięki czemu biblioteki służące do testowania możemy wykorzystywać tylko w środowiskach deweloperskich i testowych, przez to wyraźnie „odchudzając" środowisko produkcyjne.

Lista `Gemfile` zawiera tylko „docelowe" gemy; jeśli te technologie wymagają wcześniej instalacji innych gemów, od których zależą, *Bundler* zajmie się tym samoczynnie. Można wyświetlić listę aktualnie używanych gemów wraz z ich wersjami wywołując komendę `bundle list` wewnątrz katalogu projektu.

Po uzupełnieniu pliku `Gemfile` należy wykonać polecenie `bundle install`, co

sprawi, że *Bundler* porówna listę obecnie używanych gemów z tym plikiem, zsynchronizuje projekt, a jednym z efektów jego działania będzie stworzenie, bądź odświeżenie, pliku `Gemfile.lock` – który zawiera dokładniejsze informacje o gemach i ich zależnościach. To właśnie ten plik jest dla serwera istotniejszy i służy jako referencja co i gdzie znaleźć.

Na potrzeby naszego projektu podamy *Bundlerowi* nowe gemy, po czym nasz plik będzie wyglądał tak:

```
source 'http://rubygems.org'

gem 'rails', '3.1.0'
gem 'jquery-rails'
gem 'mysql2'

gem 'simple_form', '1.5.2'
gem 'haml-rails', '0.3.4'
gem 'pry', '0.9.7.3'
gem 'devise', '1.4.9'
gem 'cancan', '1.6.7'
gem 'zurb-foundation', '2.0.2'

group :development, :test do
  gem 'rspec-rails' , '2.7.0'
  gem 'factory_girl_rails' , '1.3.0'
  gem 'cucumber-rails' , '1.0.2'
  gem 'capybara' , '1.1.1'
end

group :assets do
  gem 'sass-rails', "  ~> 3.1.0"
  gem 'coffee-rails', "~> 3.1.0"
  gem 'uglifier'
end

group :test do
  gem 'turn', :require => false
end
```

Podawanie konkretnych wersji nie jest wymagane, podałem je powyżej, na wypadek gdyby ktoś chciał prześledzić całą budowę aplikacji w przyszłości, gdy wiele z tych gemów będzie miało nowsze wersje, niekoniecznie kompatybilne.

Uzasadniając wybór gemów, należy przedstawić za jakie elementy działania aplikacji odpowiadają:

- `simple_form` – umożliwia prostsze i bardziej eleganckie tworzenie formularzy *HTML* w widokach aplikacji, jest w pełni kompatybilny z systemem zasobów (ang. *resources*) platformy Ruby on Rails.
- `haml-rails` – umożliwia korzystanie z języka znaczników *Haml* w miejsce *HTML*. Osobiście uważam, że *Haml* jest czytelniejszy, zwłaszcza kiedy kiedy kod

HTML wymieszany jest z zagnieżdżonym kodem *Ruby* (znaczniki *erb*).

- `pry` – alternatywa dla programu *IRB* (interaktywna powłoka interpretera Ruby), która pozwala na dokładniejsze testowanie kodu i udostępnia czytelniejszy wydruk wyniku działań interpretera.
- `devise` – biblioteka upraszczająca zarządzanie procesem uwierzytelniania użytkownika, udostępnia własny generator i dość ogólne widoki zajmujące się rejestrowaniem, logowaniem i przypominaniem hasła użytkownika.
- `cancan` – biblioteka upraszczająca zarządzanie procesem autoryzacji, napisana z myślą o współpracy z gemem *devise*. Cały system zarządzania uprawnieniami można dzięki niej sprowadzić do jednej prostej klasy.
- `zurb-foundation` – biblioteka udostępniająca zestaw wizualnych stylów, ułatwiający tym samym tworzenie zadowalającej wizualnie kompozycji widoku.

Pozostałe gemy, wymienione w grupie przeznaczonej tylko do rozwoju i testów, są niezbędnymi elementami (chociaż istnieje możliwość skorzystania z innych technologii w ich miejsce) wykorzystywanymi w metodyce BDD:

- `rspec-rails` – to implementacja technologii *RSpec* dla środowiska *Ruby on Rails*.
- `factory_girl_rails` – gem umożliwiający tworzenie testowych obiektów (zwanych w j.ang. *stubs* – pniaki, zalążki) wykorzystywanych w testach automatycznych aplikacji
- `cucumber-rails` – implementacja systemu *Cucumber* dla platformy *Ruby on Rails*
- `capybara` - „symulator” przeglądarki internetowej

Mając tak uporządkowany plik `Gemfile` możemy przystąpić do instalacji i podłączenia wymaganych gemów poprzez polecenie `bundle install`. Następnie trzeba sprawdzić, czy gemy te nie wymagają dodatkowych kroków w celu lepszej integracji ze środowiskiem. W tym wypadku wszystkie gemy wymagają tylko skorzystania z wygodnych generatorów Rails. Polecenie `rails g` wyświetli listę dostępnych generatorów, wśród których zobaczyć można:

```
(…)

Cucumber:
  cucumber:install
  cucumber:feature

Devise:
  devise
  devise:form_for
  devise:install
  devise:shared_views
  devise:simple_form_for
  devise:views
(…)
```

```
  FactoryGirl:
    factory_girl:model

  Foundation:
    foundation:install
    foundation:layout

  Jquery:
    jquery:install

  (…)

  Rspec:
    rspec:install

  SimpleForm:
    simple_form:install

  (…)
```

Należy po kolei odpalić generatory dla tych gemów (korzystając z polecenia `rails g cucumber:install` i analogicznie `rails g nazwa-gema:install` dla wszystkich pozostałych).

Ważnym krokiem będzie skonfigurowanie bazy danych w pliku `database.yml`, jest to niezwykle prosta konfiguracja, na potrzeby tego projektu wygląda ona następująco:

```
development:
  adapter: mysql2
  encoding: utf8
  database: mgr_dev
  username: root

test: &test
  adapter: mysql2
  encoding: utf8
  database: mgr_test
  username: root

cucumber:
  <<: *test
```

Polecenie `rake db:create` utworzy w bazie danych odpowiednie bazy.

Ostatnim ruchem powinno być utworzenie strony startowej (root) w aplikacji, np. poprzez wykonanie `rails g controller home index`, dodanie wiersza

```
root :to => 'home#index'
```

w pliku `config/routes.rb`, a na koniec usunięcie pliku `public/index.html`

Jeżeli aplikacja bezbłędnie uruchamia serwer (polecenie `rails s`) oraz konsolę (`rails c`), należy utworzyć pierwszą rewizję kodu w repozytorium systemu *Git*.

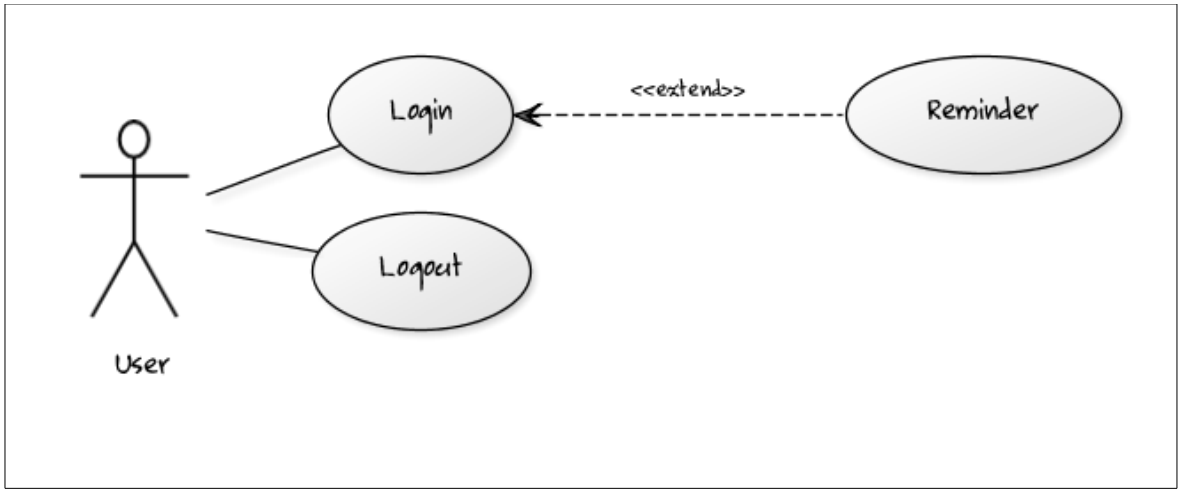
IX. Iteracja 2:
Uwierzytelnianie

Zadania:

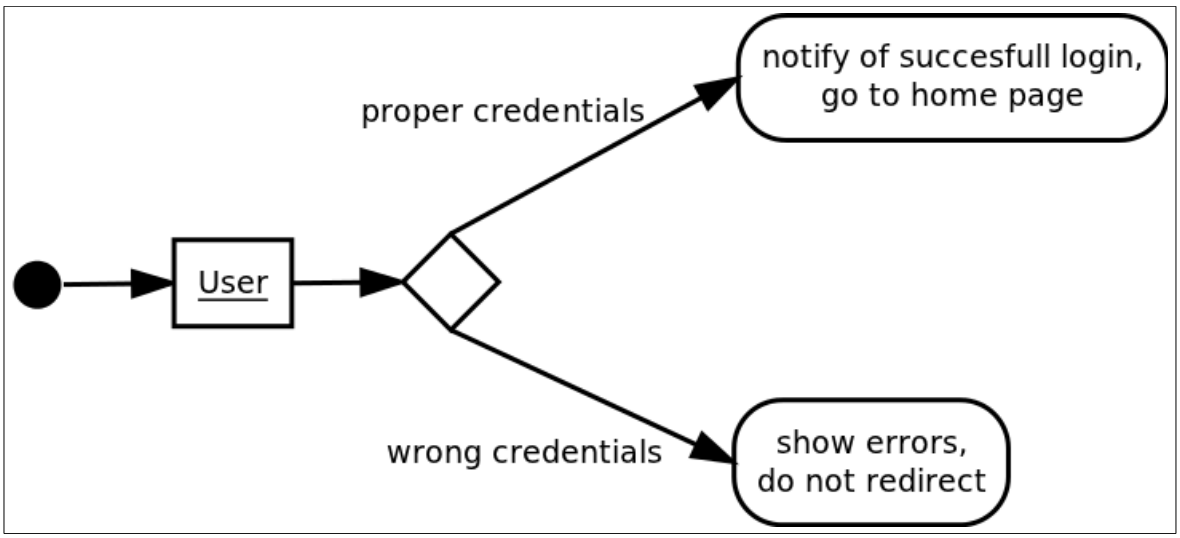
1. *Stworzyć system uwierzytelniania użytkownika za pomocą gema **Devise***

Podstawowym elementem – a na obecnym poziomie rozwoju aplikacji sieciowych wymaganym – bezpieczeństwa aplikacji sieciowej jest identyfikacja użytkowników. W przypadku kiedy umożliwiamy użytkownikom umieszczanie różnego typu treści w bazie danych zagrożenie ze strony szkodliwej zawartości wzrasta, zwłaszcza jeśli użytkownik może w tekście zawierać linki lub fragmenty kodu – aplikacja staje się podatna na ataki typu *inject*. Zabezpieczanie się przed atakami tego typu nie jest wyjątkowo trudne, należy do „kanonu" bezpieczeństwa aplikacji, co nie zmienia faktu, że administrator powinien mieć możliwość zidentyfikowania użytkownika, który ataku dokonał (lub w inny sposób naruszył regulamin użytkowania aplikacji). Wreszcie umieszczanie danych bez podawania autora, od których dane te pochodzą, jest nieodpowiedzialne i nie napawa współużytkowników serwisu zaufaniem. Wobec tego chodzi nie tylko o identyfikację użytkownika, ale o jego uwierzytelnianie, a logicznym rozwinięciem jest uwierzytelniania jest autoryzacja użytkowników. Obie te funkcjonalności obsłużymy łatwo za pomocą gemów `Devise` oraz `Cancan`.

Ze względu na to, że są to gotowe rozwiązania, nie mamy możliwości dokładnego prześledzenia procesu wytwarzania kodu zgodnie z etapami testów behawioralnych, możemy jednak w prosty sposób utworzyć test sprawdzający zgodność z założeniami.



rys. 5: Przypadki użycia prostego systemu logowania



rys. 6: Prosty diagram przepływu funkcjonalności logowania

Zasada działania jest prosta, ale dodatkowo chcielibyśmy wprowadzić możliwość logowania się za pomocą nazwy użytkownika lub adresu e-mail, tak by system przyjmował dowolną z tych wartości i działał poprawnie.

Następnie działania należy wykonywać w tej kolejności:

1. *Utworzyć model użytkownika*
2. *Utworzyć generator testowych obiektów (tzw. fabrykę) dla modelu użytkownika*
3. *Utworzyć test funkcjonalności*
4. *Refaktoryzować kod aż to całkowitej akceptacji*

Modele w aplikacji Rails buduje się korzystając z domyślnych generatorów, ale w naszym przypadku chcemy skorzystać z gema `Devise`, który udostępnia własny generator, automatycznie zajmując się stworzeniem nie tylko samego modelu, ale też przypisania mu odpowiednich akcji w kontrolerze i spreparowaniem odpowiednich ścieżek do tych akcji. Zajmie się też indeksowaniem kluczy w bazie danych, haszowaniem hasła, a do wszystkiego tego dołączy jeszcze możliwość odzyskiwania hasła poprzez e-mail. Słowem – naprawdę nie wypada nie skorzystać.

```
rails g devise user
```

Powyższa komenda utworzy jeszcze migrację, czyli plik konstruujący odpowiednie zapytanie do bazy danych, w celu utworzenia na niej tabeli odpowiadającej modelowi. Jednak domyślnie utworzona migracja nie odpowiada w pełni naszym założeniom, ponieważ udostępnia tylko i wyłącznie identyfikację użytkownika za pomocą adresu e-mail. W trosce o ergonomię serwisu chcielibyśmy dać użytkownikom możliwość logowania za pomocą *loginu*. Musimy zmodyfikować skrypt migracji do następującej postaci (dodaną linijkę wyróżniono poprzez pogrubienie):

```
class DeviseCreateUsers < ActiveRecord::Migration
  def self.up
    create_table(:users) do |t|
      t.database_authenticatable :null => false
      t.recoverable
      t.rememberable
      t.trackable
     t.timestamps
    end

    add_index :users, :email,                :unique => true
    add_index :users, :login,                :unique => true
    add_index :users, :reset_password_token, :unique => true

  end

  def self.down
    drop_table :users
  end
end
```

Migrowanie bazy danych to zadanie dla narzędzia *Rake*:

```
rake db:migrate
```

To polecenie porównuje obecny stan bazy danych z plikami migracji zgromadzonymi w katalogu aplikacji i upewnia się, że baza danych odpowiada ostatnim zmianom. Nie jest narzędziem specjalnie wyrozumiałym i bardzo łatwo można sobie zaszkodzić, jeśli nie przestrzega się zasady, że raz stworzone migracje, o ile nie są migracjami końcowymi, nie mogą być modyfikowane. W skrócie – raz wprowadzoną migrację „poprawia się" tylko za pomocą innej, nowej migracji.

Migracja nie tworzy elementów w testowej bazie danych. Dzieje się tak dlatego, że wszystkie dane w tej bazie traktowane są jak jednorazowego użytku. Niszczenie obiektów, a nawet całych tablic nie jest wykluczone w środowisku testowym, może prowadzić bowiem do odkrywania przyczyn błędów w aplikacji. Dodatkowo, choć wszystkie narzędzia działają w tym kierunku, nie zawsze mamy pewność, że po zakończeniu sesji testowej baza danych wróci do pierwotnego, „sterylnego" stanu. Aplikacja *Rails* dysponuje zadaniem *Rake* właśnie na taką okazję; należy też z niego skorzystać przed pierwszym odpaleniem testu.

```
rake db:test:prepare
```

`Factory_girl` to gem zajmujący się tworzeniem „fabryk" obiektów testowych. Fabryki tworzą instancje modeli aplikacji z konkretnymi, testowymi danymi, które można wykorzystywać w określonych scenariuszach. Zarówno standardowe testy automatyczne, jak i testy behawioralne *Cucumber*, polegają na istnieniu fabryki danego modelu. Można skorzystać z generatora:

```
rails g factory_girl:user
```

Lub samemu utworzyć plik w katalogu aplikacji: `mgr/spec/factories/user.rb` i wypełnić go następująco:

```
Factory.define :user do |usr|
     usr.sequence(:login) { |n| "user_#{n}" }
     usr.sequence(:email) {|n| "user_#{n}@example.com"}
     usr.password "jfhsdk2332"
     usr.password_confirmation {|u| u.password}
end
```

Powyższy kod definiuje prostą „fabrykę użytkowników". Ilekroć podczas wykonywania testów w środowisku testowym, które potrzebują pełnowartościowego obiektu danej klasy (instancji modelu), można łatwo odwołać się do danej fabryki, która zbuduje tymczasowy obiekt. Metoda `sequence` pozwala dodatkowo utworzyć prosty klucz wg którego wypełniane będą atrybuty kolejnych instancji modelu, jeżeli do testu użyjemy więcej niż jednej. W naszym wypadku pierwszy użytkownik będzie miał login `„user_#1"`, kolejny: `„user_#2"`. Ponadto utworzenie użytkownika wymaga potwierdzenia hasła. Można oczywiście wewnątrz fabryki wpisać je dwukrotnie ręcznie, ale znacznie bardziej eleganckim rozwiązaniem jest użyć do wypełnienia atrybutu `password_confirmation` wartości pola `password`, stąd przekazanie bloku `{|u| u.password}` .

Teraz w końcu możemy wypróbować system *Cucumber* i jego język domenowy *Gherkin*. *Cucumber*, poza generatorem instalacyjnym, udostępnia też polecenie do tworzenia testów funkcjonalności:

```
rake g cucumber:feature authenticate
```

Generator stworzy plik `features/manage_authenticates.feature` jak również wypełni odpowiednimi plikami katalog step_definitions, za pierwszym razem utworzyć może również pliki konfiguracyjne środowiska (`env.rb`), ścieżek (`paths.rb`) i selektorów (`selectors.rb`). Pliki `*.feature` interesują nas najbardziej. To właśnie pliki testu behawioralnego, napisane w języku *Gherkin* – najwyższa powłoka systemu *Cucumber*. Pliki definicji kroków to skrypty Ruby, wykonujące testy za pomocą zdefiniowanego systemu (w naszym wypadku jest to program *RSpec*). Warto zwłaszcza zwrócić uwagę na zawartość pliku `web_steps.rb` – dość kontrowersyjnego skryptu, który w nowszych wersjach systemu *Cucumber* został całkowicie usunięty. Jego zadaniem jest interpretowanie słów kluczowych *Gherkina* dotyczących stanu przeglądarki, lub wyglądu strony. Takie testy nie mogą być przeprowadzane bez udziału symulatora przeglądarki, programu, który potrafi kalkulować jak wyglądać będzie odpowiedź serwera na dane żądanie; w naszej aplikacji tę niezwykle ważną rolę spełnia system *Capybara* i w wypadku gdy *Cucumber,* wykonując testy, będzie musiał sprawdzić odpowiedź serwera, zawartość formularza, czy dowolny inny warunek związany z funkcjonowaniem przeglądarki,

przekaże odpowiednie testowe obiekty nie do skryptów definicji kroków obsługiwanych przez *RSpec*, ale do tych wywołujących symulator *Capybary*.

Osobiście uważam, że generator `rake g cucumber:feature` powinno się uruchamiać co najwyżej raz, na początku budowania systemu testowego. Kiedy zbudowana zostanie struktura plików, można bezpiecznie kontynuować dalej tworząc pliki `*.feature` ręcznie. Nie jest chyba wielkim zaskoczeniem, że takie nazwy plików: `features/manage_authenticates.feature` nie cieszą się popularnością, przedrostek `manage_` jest całkowicie zbędny. Przemianuję plik na `features/authenticate.rb` a następnie opiszę funkcjonalność uwierzytelniania tak, jak mógłby to zrobić klient, albo lider projektu. Dla celów demonstracyjnych użyję polskiego języka.

UWAGA: Jeśli podczas wykonywania któregokolwiek z testów pojawi się błąd `invalid multibyte char (US-ASCII) (SyntaxError)`, oznacza to, że w składni *Ruby* użyto znaków spoza zbioru znaków *ASCII*. Jedynym rozwiązaniem, które pozwoli wykonywać testy pisane zestawem znaków UTF-8 jest umieszczanie wiersza

```
#encoding utf-8
```

na początku każdego pliku testu kroków *RSpec*, które z takich znaków korzystają.

```
# language: pl
Właściwość: Autoryzacja
  W celu zapewnienia elementarnego bezpieczeństwa
  Jako użytkownik
  Chcę mieć możliwość uwierzytelniania i autoryzacji moich poczynań

  Scenariusz: Poprawne Logowanie
    Zakładając że mamy danych użytkowników
      | email                   | login | password |
      | jkb.zuchowski@gmail.com | ellmo | haslo1   |
    Oraz że jestem na stronie "root"
    Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "haslo1"
    Wtedy trafię na stronę "root"
    Oraz będę widział informację "Zalogowałeś się jako ellmo"
```

Możemy zacząć się przyzwyczajać, że praca z testami automatycznymi w dużym stopniu polega na wypełnianiu warunków jednej z powłok testu – w tym wypadku mówimy o poprawnym zdefiniowaniu testu behawioralnego, wykonywaniu testów i reagowaniu na ich wyniki. Jakim zakończy się test, który składa się tylko i wyłącznie z najwyższej powłoki?

```
$ bundle exec cucumber

Właściwość: Autoryzacja
  W celu zapewnienia elementarnego bezpieczeństwa
  Jako użytkownik
  Chcę mieć możliwość uwierzytelniania i autoryzacji moich poczynań

  Scenariusz: Poprawne Logowanie
    Zakładając że mamy danych użytkowników
      | email                   | login | password |
      | jkb.zuchowski@gmail.com | ellmo | haslo1   |
      Undefined step: "że mamy danych użytkowników"
    Oraz że jestem na stronie "root"
      Undefined step: "że jestem na stronie "root""
    Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "haslo1"
      Undefined step: "niezalogowany użytkownik loguje się z widoku za
pomocą kredencjałów "ellmo" oraz "haslo1""
    Wtedy trafię na stronę "root"
      Undefined step: "trafię na stronę "root""
    Oraz będę widział informację "Zalogowałeś się jako ellmo"
      Undefined step: "będę widział informację "Zalogowałeś się jako
ellmo""

1 scenario (1 undefined)
5 steps (5 undefined)
0m0.724s
```

(zawartość wydruku edytowana dla większej czytelności)

Powyżej przedstawiono tylko pierwszą część wydruku. Pokazuje ona wyraźnie, że test nie może zostać wykonany ponieważ jego kroki nie mogą być poprawnie sparsowane do postaci testów *RSpec*. *Cucumber* wciela się w rolę drogowskazu, który kieruje wątek programu do konkretnych testów, sam zaś za testy nie odpowiada. W sytuacji powyżej *Cucumber* nie potrafi skierować wątków do odpowiednich kroków testowych, ponieważ te zwyczajnie nie istnieją. Dalsza część wydruku wygląda następująco:

```
You can implement step definitions for undefined steps with these
snippets:

Zakładając /^że mamy danych użytkowników$/ do |table|
  # table is a Cucumber::Ast::Table
  pending # express the regexp above with the code you wish you had
end

Zakładając /^że jestem na stronie "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Jeżeli /^niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "([^"]*)" oraz "([^"]*)"$/ do |arg1, arg2|
  pending # express the regexp above with the code you wish you had
end
```

```
Wtedy /^trafię na stronę "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Wtedy /^będę widział informację "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Cucumber wie, że testy nie istnieją, dlatego pomaga skonstruować chociaż ich zalążki (ang. *stub*). Nie potrafi rozpoznać logiki, jaka stoi czy też stać powinna za danymi testami, ale wie czego szuka i gdzie; podpowiedzi – zwracane przez system zarówno w początkowej fazie planowania, podczas rozwijania funkcjonalności, jak i usuwania błędów – są bezcenne.

Zgodnie z zaleceniem w pierwszej linijce drugiej części wydruku, *stuby* należy umieścić w pliku definicji kroków, w tym przypadku w pliku `authenticate_steps.rb`.

Metoda `pending` mówi platformie *RSpec*, że ten test istnieje, ale na razie powinien być traktowany jako zalążek, fragment niekompletnego testu. *RSpec* rozpoznaje takie kroki i w trakcie wykonywania testu wyraźnie zaznacza w którym miejscu w kodzie znajduje się nieobsłużony test – tak samo jak test, który zakończył się niepowodzeniem.

```
Using the default profile...
# language: pl
Właściwość: Autoryzacja
  W celu zapewnienia elementarnego bezpieczeństwa
  Jako użytkownik
  Chcę mieć możliwość uwierzytelniania i autoryzacji moich poczynań

  Scenariusz: Poprawne Logowanie
    Zakładając że mamy danych użytkowników
      | email                  | login | password |
      | jkb.zuchowski@gmail.com | ellmo | haslo1   |
      TODO (Cucumber::Pending)
      ./features/step_definitions/authenticate_steps.rb:5:in `/^że
mamy danych użytkowników$/'
      features/authenticate.feature:8:in `Zakładając że mamy danych
użytkowników'
    Oraz że jestem na stronie "root"
    Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "haslo1"
    Wtedy trafię na stronę "root"
    Oraz będę widział informację "Zalogowałeś się jako ellmo"

1 scenario (1 pending)
5 steps (4 skipped, 1 pending)
0m0.235s
```

Cucumber reaguje inaczej, kiedy widzi, że utworzono odpowiednie kroki *RSpec* odpowiadające jego krokom wewnątrz scenariusza behawioralnego, nawet jeśli mają status nierozstrzygniętych. Jeśli trafi na taki krok, pomija wszystkie następne wewnątrz testu scenariusza.

Pierwszy krok testu przedstawia bardzo prostą tabelę, zawierającą dane

użytkownika (tworzenie takich tabel, rozdzielonych kreską pionową – ang. *pipe'em*, pozwala na utworzenie kilku różnych obiektów testowych z fabryki w jednym kroku), które zakładamy że znajdują się w bazie danych. Kolejne kroki będą na tym polegały. Ten krok obsłużony po stronie *RSpec* musi więc upewnić się, że tak będzie – nie wolno zapomnieć o fakcie, że przed każdą sesją testów baza danych jest „sterylna", nie zawiera żadnych danych – zawiera tylko odpowiednie tabele. Użytkownicy muszą więc być dodani do testowej bazy zanim wykonamy jakiekolwiek testy z ich udziałem:

```
Zakładając /^że mamy danych użytkowników$/ do |table|
  table.hashes.each do |hash|
    Factory(:user, hash)
  end
end
```

Ta metoda jest wywoływana za każdym razem gdy Cucumber wykona krok założeń, określony tekstem „`że mamy danych użytkowników`", jako argument przyjmuje tablicę i pozwala po niej iterować. `Factory(:user, hash)` – tworzy testowy obiekt za pomocą zdefiniowanej wcześniej fabryki User, jako drugi parametr przyjmuje tablicę asocjacyjną z elementami, które mogą zastąpić domyślne atrybuty fabryki.

Tym razem wykonanie `bundle exec cucumber` zwróci wynik:

```
Scenariusz: Poprawne Logowanie
    Zakładając że mamy danych użytkowników
      | email                 | login | password |
      | jkb.zuchowski@gmail.com | ellmo | haslo1   |
    Oraz że jestem na stronie "root"
      TODO (Cucumber::Pending)
      ./features/step_definitions/authenticate_steps.rb:11:in `(...)`
      features/authenticate.feature:11:in `Oraz że jestem na stronie
"root"'
    Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "haslo1"
    Wtedy trafię na stronę "root"
    Oraz będę widział informację "Zalogowałeś się jako ellmo"

1 scenario (1 pending)
5 steps (3 skipped, 1 pending, 1 passed)
```

Test funkcjonalności nadal traktowany jest jako nierozstrzygnięty, ponieważ przynajmniej jeden z jego kroków jest tak oznaczony. Widać jednak, że pierwszy krok wykonywany jest poprawnie, a test zakończony jest sukcesem.

Ciekawostka: nazwa systemu *Cucumber* (ang. ogórek) pochodzi właśnie od zielonego koloru, oznaczającego że wszystkie testy zakończyły się powodzeniem.

Wiemy już teraz jak zmienia się status testu funkcjonalności w zależności od podejmowanych działań i uzupełniania testów, możemy przystąpić do utworzenia pełnego testu, bez nierozstrzygniętych kroków.

```
#encoding: utf-8

Zakładając /^że mamy danych użytkowników$/ do |table|
  table.hashes.each do |hash|
    Factory(:user, hash)
  end
end

Zakładając /^że jestem na stronie "(.+)"$/ do |page_name|
  steps "Given I am on #{page_name}"
end

Jeżeli /^niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "(.+)" oraz "(.+)"$/ do |login, pass|
  visit new_user_session_path
  fill_in "user_login", :with => login
  fill_in "user_password", :with => pass
  click_button "Sign in"
end

Wtedy /^trafię na stronę "(.+)"$/ do |page_name|
  assert_equal path_to(page_name), current_path
end

Wtedy /^będę widział informację "(.+)"$/ do |msg|
  page.should have_content(msg)
end
```

Na uwagę zasługuje drugi krok, który przechwytuje tytuł strony i kieruje go na inny krok *RSpec* – krok zdefiniowany wewnątrz pliku `web_steps.rb`:

```
Given /^(?:|I )am on (.+)$/ do |page_name|
  visit path_to(page_name)
end
```

`web_steps.rb` obsługuje typowe kroki scenariuszy odwołujące się do stanu przeglądarki i przekazuje je do wskazanego symulatora przeglądarki – w naszym przypadku *Capybary*. Metoda `visit` symuluje odwiedzenie danego widoku strony ze wszystkimi konsekwencjami. Ścieżka do widoku musi być uprzednio zdefiniowana i znana *Capybarze*, w innym wypadku uruchomienie tego testu zakończy się niepowodzeniem:

```
  Oraz że jestem na stronie "root"
# features/step_definitions/authenticate_steps.rb:10
       Can't find mapping from "root" to a path.
       Now, go and add a mapping in features/support/paths.rb
(RuntimeError)
       ./features/support/paths.rb:30:in `rescue in path_to'
       ./features/support/paths.rb:24:in `path_to'
       ./features/step_definitions/web_steps.rb:45:in `/^(...)'
       features/authenticate.feature:11:in `Oraz że jestem na stronie
"root"'
```

Wydruk edytowany dla czytelności

Cucumber informuje na bieżąco jakie kroki należy podejmować w celu wykonania poprawnego testu. Zaglądamy więc do pliku `features/support/paths.rb` do metody `path_to`:

```
def path_to(page_name)
  case page_name
 when /^the home\s?page$/
    '/'
  when /the new authenticate page/
    new_authenticate_path
 else
    begin
      page_name =~ /^the (.*) page$/
      path_components = $1.split(/\s+/)
      self.send(path_components.push('path').join('_').to_sym)
    rescue NoMethodError, ArgumentError
      raise "Can't find mapping from \"#{page_name}\" to a path.\n" +
        "Now, go and add a mapping in #{__FILE__}"
    end
  end
end
```

Metoda nie obsługuje widoku „`root`”, stronę startową woli nazywać „`the home page`”, co w przypadku terminologii używanej do definiowania ścieżek aplikacji Rails jest dużą niekonsekwencją. Możemy oczywiście zamienić wyrażenie „`root`” na „`the home page`” w testach funkcjonalności, ja osobiście uważam, że powinno się trzymać jednego terminu. Razi też użyte niżej wyrażenie `page_name =~ /^the (.*) page$/` , które obsłuży symulację otwarcia widoku tylko w momencie, kiedy użyjemy angielskiego określenia „`the (dowolna_nazwa) page`” do nazwania strony – w przypadku użycia języka polskiego do opisu funkcjonalności, staje się to poważnym ograniczeniem. Proponuję zatem dokonać następujące zmiany:

```
def path_to(page_name)
  case page_name
 when /^root$/
    '/'
(...)
 else
    begin
      page_name =~ /^(.*)$/
(...)
  end
end
```

To rozwiąże pierwszy „czerwony” krok testu zakończonego niepowodzeniem. Przyjrzyjmy się drugiemu. Ponieważ nie sposób przedstawić wszystkie możliwości niepowodzenia każdego z kroków, mogę tylko stwierdzić, że należy kroki wykonywać do skutku, czyli do przeprowadzenia udanego testu. Jedynym wartym wspomnienia problemem, było rozwiązanie problemu logowania za pomocą loginu **lub** adresu e-mail.

W obecnej sytuacji Cucumber najprawdopodobniej nie zaliczy testu:

```
 Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "haslo1" #
features/step_definitions/authenticate_steps.rb:14
       cannot fill in, no text field, text area or password field with
id, name, or label 'user_login' found (Capybara::ElementNotFound)
       (eval):2:in `fill_in'
```

Przyjrzyjmy się testowi:

```
 Jeżeli /^niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "(.+)" oraz "(.+)"$/ do |login, pass|
   visit new_user_session_path
   fill_in "user_login", :with => login
   fill_in "user_password", :with => pass
   click_button "Sign in"
 end
```

Wykonywanie tego kroku kończy się wysłaniem formularza, w odpowiedzi na który użytkownik ma zostać poprawnie zalogowany. Tak się jednak nie dzieje. Można się o tym przekonać uruchamiając serwer środowiska rozwojowego (komenda `rails s`) i wchodząc poprzez przeglądarkę na domyślny adres aplikacji w środowisku deweloperskim (`http://localhost:3000`). Przejdźmy na adres `/users/sign_in`. Wyświetlając kod strony (za pomocą wbudowanych inspektorów, lub opcji przeglądania kodu źródłowego w przeglądarce), zobaczymy, że formularz nie zawiera pola `user_login`, więc *Capybara* nie może go odpowiednio wypełnić.

Instalowanie gema devise w poprzedniej iteracji utworzyło w katalogach aplikacji zarówno kopię kontrolera jak i widoków związanych z uwierzytelnianiem użytkownika.

W pierwszej kolejności zajmiemy się plikiem `app/views/devise/session/…new.html.haml`, który zawiera wspomniany formularz logowania. Uprościmy widok (korzystając z *helperów* udostępnianych przez gem `simple_form`) i zamienimy pole `email` na pole `login`:

```
 %h2 Sign in
 %p.notice= notice
 %p.alert= alert
 .eight.columns
   = simple_form_for(resource, :as => resource_name, :url =>
session_path(resource_name)) do |f|
     .inputs
       = devise_error_messages!
       = f.input :login, :required => false, :autofocus => true
       = f.input :password, :required => false
       = f.input :remember_me, :as => :boolean if
devise_mapping.rememberable?
     .actions
       = f.button :submit, "Sign in"

   = render :partial => "devise/shared/links"
```

To nie wystarczy; jak się przekonamy *Cucumber* przejdzie wspomniany test, ale już następnego nie uda mu się zaliczyć:

```
  Wtedy trafię na stronę "root"
        <"/"> expected but was
        <"/users/sign_in">. (MiniTest::Assertion)
        ./features/step_definitions/authenticate_steps.rb:22:in `(...)'
        features/authenticate.feature:13:in `Wtedy trafię na stronę
 "root"'
```

Niepowodzenie nie dotyczy tego kroku – jest on tylko testem poprawności poprzednich kroków, więc prawdziwym winowajcą znowu jest metoda logowania. Po wysłaniu formularza z poprawnymi danymi użytkownik nie został zalogowany, dlatego aplikacja zamiast utworzyć sesję i przekierować przeglądarkę na stronę startową, została w widoku formularza oczekując aż użytkownik poprawi błędy – to bardzo czytelny dowód na to, że system *Capybara* bardzo dokładnie symuluje działanie przeglądarki.

Skąd bierze się problem? Jak spełnić warunki testu? Rozwiązaniem jest poprawna konfiguracja gema `devise`, który domyślnie reaguje tylko na próby logowania za pomocą adresu e-mail. Odpowiada za to metoda:

```
  def self.find_for_database_authentication(conditions={})
    self.find_by_email conditions[:email]
  end
```

Wystarczy przesłonić ją w klasie modelu użytkownika, nie zapominając o udostępnieniu atrybutu `login` za pomocą metody `attr_accessible`:

```
  class User < ActiveRecord::Base
    devise :database_authenticatable, :registerable,
           :recoverable, :rememberable, :trackable, :validatable

    attr_accessible :email, :login, :password,
           :password_confirmation, :remember_me

    def self.find_for_database_authentication(conditions={})
      self.find_by_login conditions[:login] or
      self.find_by_email conditions[:login]
    end

  end
```

Ten zabieg sam w sobie nie wystarczy, ponieważ `devise` nie przetworzy parametru `login` wysłanego z formularza, dopóki nie pozwolimy na to, modyfikując wiesz kodu w pliku inicjującym `config/initializers/devise.rb`:

```
  # config.authentication_keys = [ :email ]
```

Odkomentujemy tę linijkę, zmienimy wartość na `login` i dodatkowo przyjrzymy się innym opcjom obsługi tego parametru:

```
config.authentication_keys = [ :login ]
config.case_insensitive_keys = [ :login ]
config.strip_whitespace_keys = [ :login ]
```

To zapewni poprawne ukończenie kolejnego kroku testu i zostawi nas z krokiem ostatnim, najprostszym ze wszystkich:

```
Zakładając że mamy danych użytkowników
      | email                   | login | password |
      | jkb.zuchowski@gmail.com | ellmo | haslo1   |
    Oraz że jestem na stronie "root"
    Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "haslo1"
    Wtedy trafię na stronę "root"
    Oraz będę widział informację "Zalogowałeś się jako ellmo"
      (Rspec::Expectations::ExpectationNotMetError)

1 scenario (1 failed)
5 steps (1 failed, 4 passed)
```

Capybara sprawdza, czy po poprawnym zalogowaniu i przekierowaniu do strony startowej, widoczne jest powiadomienie. Jesteśmy tylko o krok od spełnienia ostatniego kroku w całym teście funkcjonalności. Wystarczy tylko zmienić widok strony startowej, znajdujący się w pliku `app/views/home/index.html.haml`:

```
%h1 Home#index

-unless current_user
  = link_to "Sign in", :new_user_session
-else
  %p
    = "Zalogowałeś się jako #{current_user.login}"
    = link_to "Sign out", :destroy_user_session, :method => :delete
```

Po tych wszystkich zmianach wykonanie `bundle exec cucumber` zaliczy cały test funkcjonalności:

```
Scenariusz: Poprawne Logowanie
    Zakładając że mamy danych użytkowników
      | email                   | login | password |
      | jkb.zuchowski@gmail.com | ellmo | haslo1   |
    Oraz że jestem na stronie "root"
    Jeżeli niezalogowany użytkownik loguje się z widoku (...)
    Wtedy trafię na stronę "root"
    Oraz będę widział informację "Zalogowałeś się jako ellmo"

1 scenario (1 passed)
5 steps (5 passed)
```

X. Iteracja 3
Rejestracja i odzyskiwanie hasła

Zadania:

1. *Usprawnić system rejestrowania*
2. *Założyć system resetowania hasła poprzez e-mail*

W poprzedniej iteracji omówiliśmy zautomatyzowany system uwierzytelniania użytkowników *Devise* i bardzo szczegółowo opisaliśmy wszystkie kroki refaktorowania kodu pod kątem testów *Cucumbera*. Nie mówiliśmy jednak o poziomie bezpieczeństwa, jaki on oferuje, czemu zapobiega i jak dalej wykorzystywać jego możliwości pod kątem utrzymania, lub zwiększenia poziomu bezpieczeństwa.

Utworzony przez generator *Devise* model użytkownika domyślnie zawiera pole hasła haszowane metodą `<sprawdzić metodę> //TODO`, co oznacza, że nawet w przypadku nieuprawnionego dostępu do bazy danych, intruz odczyta tylko poprawnie zahaszowane hasła. Ma to też wpływ na sposób odzyskiwania dostępu do konta przez zarejestrowanych użytkowników, ponieważ nie można hasła zwyczajni przypomnieć – system nie ma fizycznych możliwości odtworzenia oryginalnego hasła, może co najwyżej nadpisać je innym, tymczasowym i poinformować użytkownika. W tym celu wykorzystamy zautomatyzowane systemy powiadomień, tzw. *Mailery*.

Przyjrzyjmy się fragmentowi migracji użytkownika, którą stworzył *Devise*:

```
t.database_authenticatable :null => false
t.recoverable
```

Te dwie linijki odpowiadają za dodawanie do tabeli użytkowników kolumn, które pozwalają na uwierzytelnianie i odzyskiwanie konta. Na pierwszy rzut oka są dość enigmatyczne, ale wiemy, że dodają kolumnę na hasła i kolumnę na jednorazowe tokeny, służące jako jednorazowe uwierzytelnienie użytkownika, który chce zresetować hasło.

Funkcjonalność rejestracji zawiera się w kontekście pierwszego planu behawioralnego, przemianowałem zatem plik authenticate.feature na login_register.feature (analogicznie przemianować również należy plik kroków w katalogu step_definitions). Pracując z więcej niż jednym scenariuszem wewnątrz funkcjonalności, można wykorzystywać przełącznik `@wip` (od ang. *Work In Progress* – praca w toku), słowo kluczowe dodawane linijkę przed scenariuszem, aby system mógł wykonywać testy *Cucumber* tylko dla scenariuszy roboczych.

Domyślnie *Cucumber* obsługuje maksymalnie trzy scenariusze robocze, ale modyfikowanie tych własności jest bardzo łatwe, wystarczy odnaleźć plik `cucumber.yml` w katalogu konfiguracji aplikacji i modyfikować linijkę

```
wip: --tags @wip:3 --wip features
```

Moim zdaniem bardziej elastycznym rozwiązaniem jest utworzenie kilku poziomów profili roboczych, dzięki czemu możemy osobno uruchamiać testy dla scenariuszy oznaczonych np. jako `@wip`, `@wip2` i `@wip3`. Wystarczy dodać takie linijki:

```
wip: --tags @wip:3 --wip features
wip2: --tags @wip2:3 --wip features
wip3: --tags @wip3:3 --wip features
```

Następnie należy uzupełnić ścieżki w pliku `features/support/paths.rb`, w sposób w pełni zgodny ze schematami routingu aplikacji *Ruby on Rails*.

```
def path_to(page_name)
  case page_name

  when "root"
    root_path
  when "sign_in"
    new_user_session_path
  when "sign_up"
    new_user_registration_path
(…)
  end
end
```

Dzięki takim ustawieniom nasz test jest odporny na zmiany „miękkich” adresów url danych widoków, ponieważ do nawigacji wykorzystuje zmienne aplikacji, zamiast bezpośrednich adresów.

Mając już trochę doświadczenia w pracy z *Cucumberem* możemy pokusić się nie tylko o dopisanie nowego scenariusza, ale też o refaktorowanie poprzedniego, poprzez oddzielenie testów dotyczących nawigacji do formularza, od testów przesyłania formularza i przetwarzania jego parametrów, jak również o utworzenie dwóch różnych testów na dowód, że system poprawnie zaloguje użytkownika niezależnie od tego, czy wykorzystał do identyfikacji adres e-mail, czy osobisty login. Jednak budowanie dwóch bliźniaczo podobnych scenariuszy jest nielogiczne, a przede wszystkim kłóci się z podstawową zasadą programowania zwinnego: *DRY* (*Don't Repeat Yourself* – ang. nie powtarzaj się). To dobrzy moment, by zastosować wygodne rozwiązanie, zwane „szablonem scenariusza” (ang. *Scenario Outline*).

```
# language: pl
Właściwość: Uwierzytelnianie i Rejestracja
  W celu zapewnienia elementarnego bezpieczeństwa
  Jako użytkownik
  Chcę mieć możliwość uwierzytelniania i autoryzacji moich poczynań

  @wip
  Scenariusz: Widoczność formularza logowania
    Zakładając że jestem na stronie "root"
    Oraz że widzę odnośnik "Sign in"
    Jeżeli kliknę odnośnik "Sign in"
    Wtedy trafię na stronę "sign_in"
    Oraz będę widział formularz logowania

  @wip
  Szablon scenariusza: logowanie emailem LUB loginem
    Zakładając że mamy danych użytkowników
      | email                  | login | password   |
      | jkb.zuchowski@gmail.com | ellmo | haslo123   |
    Oraz że jestem na stronie "sign_in"
    Jeżeli wypełnię formularz logowania danymi
      | user_login       | user_password   |
      | <login_or_email> | <pass>          |
    Oraz kliknę przycisk "Sign in"
    Wtedy trafię na stronę "root"
    Oraz będę widział informację "Zalogowałeś się jako ellmo"

    Przykłady:
      | login_or_email          | pass     |
      | ellmo                   | haslo123 |
      | jkb.zuchowski@gmail.com | haslo123 |

  @wip2
  Scenariusz: Widoczność formularza rejestracji
    Zakładając że jestem na stronie "root"
    Oraz że widzę odnośnik "Sign up"
    Jeżeli kliknę odnośnik "Sign up"
    Wtedy trafię na stronę "sign_up"
    Oraz będę widział formularz rejestracji

  @wip3
  Scenariusz: Poprawne rejestrowanie
    Zakładając że jestem na stronie "sign_up"
    Jeżeli wypełnię formularz rejestracji danymi
      | email               | login | password | password_confirm |
      | ellmunzai@gmail.com | ellmo | haslo123 | haslo123         |
    Oraz kliknę odnośnik //TODO
    Oraz będę widział informację "Zalogowałeś się jako ellmo"
```

Scenariusze oznaczyłem stosownie przełącznikami `@wip` – dla dwóch pierwszych, które powstały z rozbicia napisanego w poprzedniej iteracji działającego testu i które muszą być jak najszybciej poprawione, oraz `@wip2` i `@wip3` dla nowych scenariuszy.

Żeby testować tylko scenariusze z odpowiednim przełącznikiem, wykorzystamy polecenie: `bundle exec cucumber -p wip`

Wynik komendy wskaże, że *Cucumber* nie wie jak rozumieć część kroków, co jest oczywiste, zważywszy na to, że dokonaliśmy poważnych zmian i przemianowaliśmy dane kroki. Kiedy naprawimy definicje kroków, okaże się, że jeden krok może być wykorzystywany przez wiele scenariuszy. Na przykład poniższe kroki są uniwersalne:

```
Zakładając że widzę odnośnik "Sign in"
Jeżeli kliknę odnośnik "Sign in"
Jeżeli kliknę przycisk "Sign in"
```

A przede wszystkim są bardzo proste, powiedziałbym nawet „atomowe" – nie daje się danego kroku rozbić na mniejsze części, ani opisać go zwięźlej.

Odpowiednie wywołania *RSpec* dla tych kroków wyglądają następująco:

```
Zakładając /^że widzę odnośnik "(.+)"$/ do |link|
  page.should have_link(link)
end

Jeżeli /^kliknę odnośnik "(.+)"$/ do |link|
  click_link(link)
end

Jeżeli /^kliknę przycisk "(.+)"$/ do |button|
  click_button(button)
end
```

Brakuje tylko ostatniego kroku, by pierwszy scenariusz kończył się pomyślnie, ponieważ *Cucumber* nie potrafi zinterpretować „`Wtedy będę widział formularz logowania`". Wystarczy zdefiniować krok *RSpec* w następujący sposób:

```
Wtedy /^będę widział formularz logowania$/ do
  page.should have_selector("form", :id => "user_new")
end
```

W drugim scenariuszu mamy po raz pierwszy do czynienia z ciekawą konstrukcją schematu scenariusza:

```
Szablon scenariusza: tytuł
  Zakładając że …
  Jeżeli …
  Wtedy …

  Przykłady:
    | parametr1 | parametr2 |
    | wartosć1  | wartosć2  |
```

Podobna do konstrukcji tablicy, która pozwalała wysyłać do kroków *RSpec*, tylko tym razem kolekcja przypisana jest do całego scenariusza i dla każdego jej elementu, wykonywane są wszystkie kroki.

Aby oba scenariusze danego schematu kończyły się pomyślnie, wystarczy napisać następującą definicję kroku *RSpec*:

```
Jeżeli /^wypełnię formularz logowania danymi$/ do |table|
  table.hashes.each_with_index do |hash, index|
    fill_in("user_login", :with => hash[:user_login] )
    fill_in("user_password", :with => hash[:user_password])
  end
end
```

Zwróćmy uwagę, że wydruk wyniku testu, będzie odbiegał (zwłaszcza kolorystycznie) od naszych oczekiwań:

```
Szablon scenariusza: logowanie emailem LUB loginem
    Zakładając że mamy danych użytkowników
      | email                  | login | password |
      | jkb.zuchowski@gmail.com | ellmo | haslo123 |
    Oraz że jestem na stronie "sign_in"
    Jeżeli wypełnię formularz logowania danymi
      | user_login       | user_password |
      | <login_or_email> | <pass>        |
    Oraz kliknę przycisk "Sign in"
    Wtedy trafię na stronę "root"
    Oraz będę widział informację "Zalogowałeś się jako ellmo"

    Przykłady:
      | login_or_email          | pass     |
      | ellmo                   | haslo123 |
      | jkb.zuchowski@gmail.com | haslo123 |
```

W tym wydruku jest logika – schemat nie może zawieść, zawieść mogą konkretne scenariusze, podpięte do schematu.