

Bielska Wyższa Szkoła  
im. Józefa Tyszkiewicza  
w Bielsku-Białej



Praca Dyplomowa

Michał Czyż

**Temat Pracy: Zwinne zarządzanie wymaganiami w procesie produkcji  
oprogramowania**

**Opiekun pracy:** dr. M. Smółka

**Ocena pracy:**

**Numer albumu:** 2288

**Numer ewidencyjny pracy:**

Bielsko-Biała, 2009



Bielsko-Biała dn. 15 lutego 2009

**Imię i nazwisko:** Michał Czyż

**Nr albumu:** 2288

**Wydział:** Zarządzania i Informatyki

**Kierunek:** Inżynieria Oprogramowania

### OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, że złożona praca inżynierska pt.: **Zwinne zarządzanie wymaganiami w procesie produkcji oprogramowania** została napisana przeze mnie samodzielnie.

Równocześnie oświadczam, że praca ta nie narusza prawa autorskiego w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U.1994 nr 24 poz. 83) oraz dóbr osobistych chronionych prawem cywilnym.

Ponadto praca nie zawiera informacji i danych uzyskanych w sposób nielegalny i nie była wcześniej przedmiotem innych procedur urzędowych związanych z uzyskaniem dyplomów lub tytułów zawodowych uczelni wyższej.

# Spis treści

## 1 Wstęp

7

## 2 Streszczenie 9

## 3 Zarządzanie projektem 10

- 3.1 Metody tradycyjne . . . . . 10
  - 3.1.1 Model Kaskadowy . . . . . 10
  - 3.1.2 Model Spiralny . . . . . 11
- 3.2 Filozofia Agile . . . . . 11
  - 3.2.1 Reguły usprawniające proces . . . . . 13
    - Nie powtarzaj się . . . . . 13
    - Najprostsza rzecz, która działa . . . . . 13
    - Naprawdę tego nie potrzebujesz . . . . . 14
    - Iteracyjny proces wytwarzania . . . . . 14
    - Scrum . . . . . 14
    - Behavior Driven Development . . . . . 15
- 3.3 Dłaczego Agile . . . . . 18

## 4 Technologie i wzorce projektowe 20

- 4.1 Model-Widok-Kontroler . . . . . 20
  - Model . . . . . 20
  - Widok . . . . . 21
  - Kontroler . . . . . 21
- 4.2 Representational state transfer . . . . . 21
- 4.3 Narzędzia wykorzystane w pracy . . . . . 22
  - 4.3.1 Ruby . . . . . 23
    - Główne cechy języka . . . . . 23
  - 4.3.2 Ruby on Rails . . . . . 23
    - Zalety środowiska . . . . . 23
  - 4.3.3 Rspec . . . . . 24
  - 4.3.4 Cucumber . . . . . 26

## 5 Wspomaganie procesu produkcji oprogramowania 29

- 5.1 Specyfikacja funkcjonalna . . . . . 29
  - 5.1.1 Opowieści użytkownika . . . . . 29
  - 5.1.2 Śledzenie zmian . . . . . 30
  - 5.1.3 Eksport opowieści użytkownika . . . . . 30
  - Cucumber . . . . . 30

PDF . . . . .	30
5.2 Implementacja narzędzia . . . . .	30
5.2.1 Diagram Modeli . . . . .	31
5.2.2 Diagram Kontrolerów . . . . .	32
5.2.3 Diagram Bazy Danych . . . . .	33
<b>6 Prezentacja aplikacji</b>	<b>35</b>
<b>7 Podsumowanie</b>	<b>41</b>



# 1 Wstęp

Właściwe pozyskiwanie informacji dotyczących wymagań od klienta, oraz ich dalsze przetwarzanie ma kluczowe znaczenie w procesie produkcji oprogramowania. Na to niełatwe zadanie wpływ ma wiele czynników. Jednym z powodów jest niejednolity język, którym posługują się klienci i deweloperzy. Zamawiający mają problem z artykułowaniem swoich oczekiwań, a deweloperzy często im w tym nie pomagają. Powstają w ten sposób systemy, które nie spełniają w pełni oczekiwań funkcjonalnych, bądź są trudne w obsłudze, albo wręcz nie nadają się do jakiegokolwiek zastosowania i łądują, mówiąc obrazowo, w koszu.

Problemy tego typu, stara się rozwiązać Inżynieria Oprogramowania. Na przełomie ostatnich 30 lat pojawiły się różne metodologie, czy też teorie, które porządkowały proces produkcji oprogramowania. Jednak, jak uargumentował Brooks w artykule „*No Silver Bullet - Essence and Accidents of Software Engineering*”[55] z 1986 roku, nie ma i nie będzie srebrnej kuli, jeśli chodzi o metodologię zarządzania produkcją oprogramowania. Zwraca to naszą uwagę na fakt, że tworzenie oprogramowania jest trudne i wymaga czegoś więcej, niż formalnej struktury pracy, czy też listy kroków za którymi wystarczy podążać.

Pewne metodologie - np.: model kaskadowy - przez swoją niską skuteczność zostały już uznane przez wielu jako nieskuteczne[1, 56]. Jeśli coś nie spełnia swojej roli, zaczynają się poszukiwania i eksperymenty nad nowymi koncepcjami. Od lat siedemdziesiątych wiele się zmieniło, deweloperzy zrozumieli że narzędzia nie rozwiążą tego problemu, potrzebna jest natomiast zmiana w podejściu do klienta, która wymusza, aby był on członkiem zespołu projektowego. Dzięki temu oprogramowanie będzie efektem silnej współpracy, której celem jest jak najlepszy produkt. Jednak aby to było możliwe, potrzeba wskazówek, którymi należy się kierować. Trzeba umieć uargumentować klientowi, że taki model współpracy da najlepsze efekty.

Pomiędzy 11 a 14 Lutego 2001 przedstawiciele[3] wielu idei, takich jak *extreme programming* - Kent Beck, *pragmatic programing* - Dave Thomas, Andrew Hunt, *Scrum* - Jeff Sutterland, *crystal clear* - Alistair Cockburn, *adaptive software development* - Jim Highsmith, oraz 11 innych doświadczonych programistów, którzy podpisali się pod „*Agile Manifesto*”[2]. Zwrócili uwagę na cztery największe wartości, o których należy pamiętać w trakcie produkcji oprogramowania, a brzmią one następująco:

Poprzez wytwarzanie oprogramowania, oraz pomaganie innym w tym zakresie, odkrywamy lepsze sposoby realizowania tego zadania. W wyniku tych doświadczeń zaczęliśmy przedkładać:

*Jednostki i współdziałania między nimi nad procesy i narzędzia.*

*Działające oprogramowanie nad dokładną dokumentacją.*

*Współpracę z klientem nad negocjację umów.*

*Reagowanie na zmiany nad realizowanie planu.*

Oznacza to, że wprawdzie doceniamy to, co wymieniono po prawej stronie, jednak bardziej cenimy to, co znajduje się po lewej.[5]

Sam manifest przykuwa uwagę. Wskazuje, że bycie zwinnym<sup>1</sup> jest nacelowane na proces oraz ludzi, a nie na narzędzia i technologie. Aczkolwiek zrozumienie tego procesu, oraz wdrożenie go efektywnie, wymaga o wiele większej wiedzy. Prezentacja tej filozofii, wraz z przykładem użycia jej przy produkcji - zwłaszcza aplikacji webowych - jest jednym z celów tej pracy.

Drugim celem jest prezentacja narzędzia, które ma za zadanie wspierać proces pozyskiwania wymagań od klienta, udostępniając zaledwie trzy główne funkcjonalności. A są to:

- przechowywać opowieści użytkownika
- udostępniać historie zmian dla tych opowieści użytkownika
- wspierać pisanie automatycznych testów integracyjnych

Dzięki tym trzem funkcjonalnościom, jest możliwe jasne zarządzaniem wymaganiami klienta, oraz wykazywanie jakie zmiany nastąpiły. Daje to obraz ilości pracy, która została wykonana dodatkowo, co umożliwi lepsze rozliczanie pracy. Kolejnym celem jest wykazanie, o ile lepszy jest efekt końcowy od pierwotnego - co wpłynie na zadowolenie klienta.

---

<sup>1</sup>Agile (zwinny), w pracy na przemian będzie używany oryginał oraz polskie tłumaczenie



## 2 Streszczenie

W pierwszej części zostanie zwrócona uwaga na podstawy teoretyczne związane z zarządzaniem projektów. Krótko zostaną przedstawione metody tradycyjne, zwane też ciężkimi, takie jak model kaskadowy, czy też spiralny. Następnie zwrócona zostanie uwaga na metody zgodne z filozofią agile - potocznie zwane lekkimi. Kolejną sprawą będzie przedstawienie czterech wartości z manifestu agile, oraz uzupełniających ich zasad, wraz z praktycznymi przykładami. Zostaną wydzielone fazy produkcji oprogramowania - które są agile, a które fragile. Rozważone zostaną trzy główne metodologie związane z filozofią agile. Na koniec tej części nastąpi porównanie metod lekkich z ciężkimi.

Aby móc być zwinnyymi podczas procesu produkcji oprogramowania, warto wybierać odpowiednie narzędzia, które się wybiera. Nie jest to niezbędne, aczkolwiek daje nam to większą efektywność w czasie pracy. Opisane zostaną wzorce projektowe, które stymulują podejmowanie właściwych decyzji, przy projektowaniu architektury systemu. Zwrócona zostanie uwaga, na dynamiczny i funkcyjny język jakim jest Ruby[6]. Zostanie również omówione środowisko do budowy aplikacji webowych Ruby on Rails[7]. Następnie zwrócona będzie uwaga, na biblioteki wspierające pisanie automatycznych testów, wraz z narzędziami do masowego tworzenia danych testowych.

Trzecią częścią będzie prezentacja koncepcji związanych z narzędziem wspomnianym na wstępie. Zaprezentowane zostaną wymagania dotyczące tego projektu, opisane za pomocą opowieści użytkownika - technika ta będzie opisana w sekcjach poświęconych BDD i Cucumber. Następnie zostanie zwrócona uwaga, na rozwiązania od strony technicznej, związane z tą aplikacją.

Ostatnim etapem będzie prezentacja samego narzędzia.

## 3 Zarządzanie projektem

Każda metodologia ma na celu ulepszenie procesu produkcji oprogramowania. Celem jest dostarczanie produktu, który spełni oczekiwania użytkownika końcowego. Model zarządzania powinien ułatwiać szacowanie czasu, potrzebnego na powstanie systemu, oraz kosztów, lub umożliwić wykonanie, jak największej ilości pracy w danym budżecie. Wymaga to pewnej formalizacji wymagań, oraz rejestrowania postępów, dzięki któremu deweloperzy wiedzą kiedy wszystko jest już zrobione. Zwrócona zostanie teraz uwaga na kilka takich modeli.

### 3.1 Metody tradycyjne

Metody tradycyjne, zwane też ciężkimi, zawdzięczają taki przydomek ilości dokumentacji, która jest tworzona przed rozpoczęciem prac związanych z implementacją, często określane jako BDUF<sup>2</sup>. Argumentacją takiego podejścia, jest przeświadczenie, że solidna dokumentacja niesie ze sobą sukces projektu, oraz pozwala też uniknąć kosztów zmiany, gdyż zmniejsza prawdopodobieństwo pojawienia się zmian w trakcie implementacji. Zwróćmy uwagę na dwa najpopularniejsze modele.

#### 3.1.1 Model Kaskadowy

Model Kaskadowy<sup>3</sup> został opisany przez Winstona W. Royce w roku 1970 roku, w artykule „Zarządzanie produkcją dużych systemów informatycznych”[43]. Choć on sam nie użył tej nazwy, jest to jednak najbardziej znany dokument opisujący ten model. Royce w żadnym wypadku nie chwali takiego podejścia, a raczej je krytykuje, jako nieskutecznie i nieefektywne. Głównym założeniem tej metodologii, jest podział produkcji oprogramowania na sześć niepodzielnych faz:

- Specyfikacja wymagań funkcjonalnych
- Analiza systemu
- Modelowanie systemu, rozwiązań
- Implementacja
- Testy jednostkowe, oraz integracyjne
- Wdrożenie i pielęgnacja systemu

Przejście do kolejnej fazy, jest możliwe tylko i wyłącznie po ukończeniu aktualnej. Jeżeli na kolejnym etapie pojawiają się problemy związane z niedopatrzzeniami

---

<sup>2</sup>Big Design Up Front

<sup>3</sup>Waterfall model - z języka angielskiego

w którejś z poprzednich faz, należy wrócić do tego momentu, i wykonać jeszcze raz wszystkie etapy w kolejności, aż do skutku. Oprogramowanie jest wdrażane, gdy ostatnia faza da zadowalający efekt, jest to zazwyczaj długi okres czasu, około jednego roku.

### 3.1.2 Model Spiralny

Model spiralny<sup>4</sup>, zdefiniował Barry Boehm, w artykule „Model Spiralny w produkcji oprogramowania i usprawnienia”[44] z 1988 roku. Boehm podzielił proces produkcji oprogramowania na cztery iteracyjne fazy:

- Specyfikowanie systemu wraz z detalami, ustalenie celów
- Analiza ryzyka oraz prototypy
- Implementacja, zawierająca
  - przygotowanie szczegółowych koncepcji
  - programowanie
  - integrowanie z istniejącą częścią systemu
  - testowanie
  - odbiór
- Analiza zakończonej iteracji i planowanie następnej

Specyfikacja modelu iteracyjnego, nie narzuca jedynie takich etapów w fazie trzeciej, można tutaj użyć również modelu kaskadowego. Każda iteracja jest swego rodzaju samodzielna, może rozpocząć się na już prowadzonym projekcie, a po jej zakończeniu wszystkie zmiany, które miały zostać osiągnięte, są wdrożone w działający system. Zazwyczaj iteracja trwa od jednego do trzech miesięcy, a więc jest krótsza niż w modelu kaskadowym

## 3.2 Filozofia Agile

Jak wykazano na wstępie filozofia agile jest inna. Jako pierwsza, kieruje uwagę na dostosowywanie się do konkretnego projektu klienta, i bycie elastycznym. Bardzo klarowny jest przekaz manifestu agile (wartości które są najistotniejsze, wymienione są jeszcze raz poniżej). Poniżej 12 zasad które uzupełniają sam manifest.

### Największe wartości Agile

---

<sup>4</sup>Spiral Model - z języka angielskiego

*Jednostki i współdziałania między nimi*

*Działające oprogramowanie*

*Współpraca z klientem*

*Reagowanie na zmiany*

## **Zasady Agile[4]**

Najważniejsze jest zadowolenie klienta poprzez wczesne i ciągle dostarczanie wartościowego oprogramowania.

Zmiany w wymaganiach są mile widziane, nawet na późnym etapie prac. Proces agile radzi sobie ze zmianą przez wzgląd na korzyść dla klienta.

Dostarczać działający system często, czy to co kilka tygodni, czy też miesiący, z naciskiem na krótsze okresy.

Ludzie biznesu i deweloperzy muszą codziennie pracować razem, przez cały okres trwania projektu.

Prace nad projektem należy połączyć z motywacją jednostek. Zapewnij im odpowiednie środowisko pracy, oraz wsparcie jakiego potrzebują.

Obdarz ich zaufaniem, że wykonają pracę jak należy.

Najbardziej wydajną i efektowną metodą przekazu informacji dla zespołu oraz dla klienta jest rozmowa twarzą w twarz.

Główną miarą postępu jest działające oprogramowanie.

Proces Agile promuje ciągły niczym nieprzerwany proces rozwoju systemu. Inwestorzy, deweloperzy oraz użytkownicy powinni zabiegać o ciągle utrzymywanie stałego tempa pracy.

Zwracanie ciągłej uwagi na techniczny kunszt oraz dobre projektowanie, wspiera bycie zwinnym.

Prostota - czyli sztuka maksymalizacji ilości pracy,  
niewykonanej - to podstawa.

Najlepsze architektury, wymagania oraz projekty  
pochodzą od zespołów które same sobą zarządzają.

W równych odstępach czasowych, zespół analizuje w jaki  
sposób mógłby stać się bardziej efektywny, a następnie dostosowuje się  
do własnych spostrzeżeń.

Powyższe zasady i wartości warto mieć zawsze w pamięci, jeśli chcemy pozostać  
w zgodzie z filozofią agile. Na podstawie tych reguł, powstało wiele zasad i metodologii,  
które wdrażane, wraz z dostosowaniem do możliwości zespołu i charakteru projektu,  
dają dobre efekty.

### 3.2.1 Reguły usprawniające proces

Reguły wymienione poniżej są bardzo proste, ale tylko gdy dobrze je zrozumiemy

**Nie powtarzaj się** Druga zasada Agile, traktuje zmiany, jako mile widziane na każdym etapie pracy, nawet na tym bardzo zaawansowanym. Wymaga to częstych zmian w kodzie źródłowym. Jedną z zasad, która ułatwia taki cykl obróbki kodu - czasem wręcz tylko dzięki stosowaniu się do niej jest to możliwe - jest unikanie duplikacji kodu. Pomocna przy tym jest reguła DRY<sup>5</sup>, przez wielu odczytywana jako unikanie powtórzeń w kodzie. Jak zwrócił uwagę Dave Thomas[10] to nie jest prawdziwy cel tej reguły - jej sednem jest, aby każda część wiedzy - zachowania - systemu miała jedną autorytatywną i jednoznaczną reprezentację. Wiedza na temat systemu, jak zwrócił uwagę w wywiadzie, wychodzi ponad sam kod aplikacji, dotyczy ona schematu baz danych, kwestii testów, a nawet dokumentacji. Czasem, aby móc osiągnąć taki stan, potrzebne są generatory kodu wiążące się z zaawansowanymi technikami meta-programowania, lecz właśnie dzięki stosowaniu się to tej reguły jesteśmy w stanie uzyskać elastyczny i dający się pielęgnować system.

**Najprostsza rzecz, która działa** Istotą tej reguły jest jak najszybsze pokazanie efektu pracy klientowi. Jeżeli uprościmy pierwszą wersję projektu, uzyskanie efektu jest szybsze. Jest to jak najbardziej zgodne z pierwszą, trzecią oraz siódmą zasadą manifestu agile. Pokazywanie klientowi często tego, co zostało zrobione, stymuluje potrzebne zmiany w systemie, które bez wizualnej prezentacji zostałyby pominięte, czy też zepchnięte na dalszy plan.

---

<sup>5</sup>Don't Repeat Yourself, znana również jako Single Point of Truth. Popularny akronim DRY

**Naprawdę tego nie potrzebujesz** Czasem zdarza się, że tworzymy nowe klasy, metody nie zastanawiając się, czy aby na pewno tego naprawdę potrzebujemy w tej chwili. Bądź też piszemy kod, z myślą o przyszłych ewentualnych udogodnieniach, które mogłyby mieć miejsce, jeśli trzeba będzie zrobić to, lub owo. Związane jest z tym wiele wątpliwości. Reguła[57] silnie nawołuje, aby kodować tylko funkcjonalności rzeczywiste, te które są ważne, i dlatego warto zadać sobie samemu pytanie "Czy aby na pewno tego potrzebuję?", zamiast mechanicznie klepać kod. Drugą sprawą, na którą zwraca ona uwagę, to fakt, że w momencie gdy zaczynamy rozwodzić się nad przyszłymi udogodnieniami, tracimy z oczu główny cel.

**Iteracyjny proces wytwarzania** Aby często dostarczać działający system, potrzeba podzielić proces tworzenia systemu, na małe kawałki, z których każdy dostarcza biznesową wartość dla systemu. XP<sup>6</sup> preferuje przyjąć jako jednostkę tydzień, Scrum dwa. Lecz nie jest to sztywną regułą, warto dostosować jednostkę do swoich możliwości. Jednakże nacisk jest kładziony na krótkie okresy.

**Scrum** W 1986 roku Hirotaka Takeuchi oraz Ikujiro Nonaka, opisali całkowicie nową koncepcję zarządzania projektami informatycznymi. Położyli w nim nacisk, na postęp całej drużyny, współdziałającej razem nad wspólnymi celami. W 1991 roku DeGrace i Stahl, jako pierwsi odnieśli się to tej metodologii, która funkcjonuje pod nazwą Scrum.

Idea Scrum'a[58] wychodzi z założenia, że na początku realizacji projektu, mamy ogólny obraz właściwości, które musi posiadać dany system. Zestawienie tych cech, nazywane jest listą właściwości produktu<sup>7</sup>. Lista ta, powinna być posortowana według biznesowej wartości. Powinna być też wydzielona część, na właściwości krytyczne, oraz tak zwaną listę życzeń. Kolejnym etapem jest pogrupowanie - biorąc pod uwagę ich priorytety - celów, i wydzielenie ram czasowych, dla każdej z grup - okresy te nazywamy sprintami.

Każdy sprint posiada własną listę właściwości, która jest już bardziej szczegółowa. Każdy element listy posiada swój priorytet, dzięki czemu wiadomo jaka jest kolejność wykonywania zadań. Może ona zawierać także właściwości opcjonalne, choć nie jest to konieczne. Sprint trwa od dwóch tygodni do miesiąca. Zadania, które estymowane są na więcej niż 16 godzin, powinny być podzielone.

Najpóźniej - jeśli to możliwe, można wcześniej - na koniec sprintu, wszystkie właściwości muszą być gotowe do użycia. Dzięki temu możliwe jest zebranie reakcji od użytkowników. Ich uwagi są brane pod uwagę przy następnym sprincie, lub jeśli czas pozwala - w tym samym.

---

<sup>6</sup>skrót - Extreme Programming

<sup>7</sup>produkt backlog(ang.)

W ramach sprintu każdy dzień rozpoczyna się od krótkiego - około piętnastominutowego - spotkania, w którym uczestniczą wszyscy członkowie zespołu. Każdy odpowiada na trzy pytania:

- Co udało mi się zrobić w dniu wczorajszym?
- Co planuje zrobić dziś?
- Na jakie przeszkody natrafiłem?

Dzięki temu cały zespół wie, jakie zmiany następują w systemie, oraz daje to możliwość szybszego rozwiązywania przeszkód, dzięki spojrzeniu na problem z wielu perspektyw.

Rozwój projektu jest mierzony za pomocą „tablic postępu”<sup>8</sup>. Jest ona osobna dla całego projektu (poszczególnych właściwości), oraz dla poszczególnych sprintów. Jako skali używa się czasem liczb pierwszych, lub liczb z ciągu Fibonacciego.

**Behavior Driven Development** BDD jest techniką organizacji produkcji oprogramowania, która kładzie duży nacisk na współpracę, pomiędzy programistami, a ludźmi związanymi z biznesem - tymi, którzy zamawiają system, jak również z tymi, którzy będą go używać. Jest to zgodne z filozofią agile. Dan North<sup>9</sup> zaprezentował po raz pierwszy tą koncepcję, w 2003 roku[13]. Jak sama nazwa wskazuje, zachowanie systemu jest najważniejszą kwestią w trakcie produkcji. Autor zwrócił szczególną uwagę na kwestię wartości biznesowej, którą wniesie implementacja konkretnego zachowanie w systemie - gdyż to jest najważniejsze z punktu widzenia użytkowników systemu. Ciekawe jest też jego podejście jak determinować, którą rzecz powinniśmy implementować. Sugeruje, że powinniśmy sobie zadać pytanie „Jaka jest kolejna najważniejsza cecha, której system nie posiada?”. Wydaje się to banalne, aczkolwiek przynosi najlepsze rezultaty dla inwestorów, a zwłaszcza końcowych użytkowników. W przypadku korzystania ze Scrum’a jest to dobra reguła dla ustawiania priorytetów w backlog’u.

Zgodnie z tym, Dan North przedstawił[14] trzy główne założenia BDD, którym przyjrzymy się teraz dokładnie.

**Najważniejszy jest opis zachowania systemu[15]** Ta reguła zwraca uwagę, iż najważniejsze jest to, aby wiedzieć jak system ma się zachować i jakie funkcjonalności ma udostępniać dla użytkownika. Zapis funkcjonalności w ten sposób - opis, zachowanie - jest uwarunkowany potrzebą jednoznacznego rozumienia słów - efektu jaki mają mieć na system - tak, aby programiści, oraz ludzie związani z biznesem

---

<sup>8</sup>burnout chart(ang.)

<sup>9</sup><http://dannorth.net/> - oficjalny blog

wzajemnie się rozumieli. Wykorzystuje się do tego celu opowieści użytkownika, które są narracją interakcji użytkownika z systemem, a które dokładniej zostaną omówione później.

**System udostępnia pewne zachowanie, ponieważ wnosi ono wartość dla biznesu[16]** Dosyć częstym błędem jest występowanie w systemie nowych funkcjonalności, bądź użycie nowych technologii, tylko dlatego, że sprawiają wrażenie interesujących. Jest to pewnego rodzaju patologia, występująca w produkcji oprogramowania. Zwrócenie uwagi na wartość biznesową, ma ogromny wpływ na efektywność, oraz koszt związany z systemem, choć zazwyczaj nie jest to sprawa oczywista ani, na co zwraca uwagę North, nie jest to postrzegane jako rzecz niezbędna.

**Opierać się na wiedzy, a nie przewidywaniach[17]** Domena metodologii BDUF zakłada, że jesteśmy w stanie zapisać wszystkie informacje na temat systemu, przed rozpoczęciem jego realizacji. Innymi słowy, jesteśmy w stanie przewidzieć wszystko, na temat systemu - jednak bardzo często tak nie jest. Warto zwrócić uwagę na fakt, że im dłuższy okres chcemy przewidzieć, oraz im bardziej złożony jest problem, tym większe prawdopodobieństwo, iż nasze założenia będą błędne. Taka postawa - posiadania wiedzy na początku - wynika również z obawy przed potencjalnym kosztem zmiany, a menadżerowie są w tym wypadku najlepszym przykładem. BDUF się nie sprawdziło, gdyż zmiany zawsze się pojawiają. BDD z założenia zabrania modelowania systemu przez zgadywanie, a minimalizacja kosztu zmian jest uzyskiwana poprzez utrzymanie technicznie systemu w taki sposób, aby łatwo można było w nim zmiany wprowadzać. W sekcji Filozofia Agile zwrócono uwagę na metodologie, które to umożliwiają, takie jak iteracyjny proces produkcji - najprostsza rzecz, która działa, oraz nie implementowanie funkcjonalności, które nie są wymagane.

**Opowieści użytkownika** North zwraca uwagę na trzy części, które występują w opowieści

- tytuł, który jednoznacznie determinuje zakres serwowanych funkcjonalności w systemie
- krótki opis tego, co jest celem danej funkcjonalności i dlaczego ją wprowadzamy
- warunki akceptacji, które definiują kiedy funkcjonalność w systemie jest w pełni zaimplementowana są w formie scenariuszy



## Właściwość

### Dodawanie opisu właściwości

Aby w przejrzysty sposób opisywać funkcjonalności w systemie, aby jak najwięcej osób - zwłaszcza nie informatycznych - je rozumiało, by łatwiej było wspólnie tworzyć systemy informatyczne klient, system architect, menadżer projektu chce mieć możliwość opisanie funkcjonalności poprzez wypełnianie intuicyjnych formularzy

### Scenariusz

### Przejdź na stronę

### formularza tworzenia

**Dane** jestem na stronie listy właściwości  
**Oraz** jestem zalogowany  
**Jeżeli** kliknę na link 'Utwórz nową właściwość'  
**Wtedy** zobacz formularz polami do wprowadzania opisu

### Scenariusz

### utworzenie nowej

### właściwości

**Dane** jestem na stronie z formularzem tworzenia właściwości  
**Jeżeli** poprawnie wypełnię pola  
**Oraz** kliknę przycisk utwórz  
**Wtedy** powinienem zostać przekierowany na stronę właściwości  
**Oraz** powinienem zobaczyć tekst który wprowadziłem w formularzu  
**Ale** ładnie sformatowany

**Automatyczne testowanie zachowania systemu** Celem BDD jest wymuszenie współpracy z klientem, oraz polepszenie z nim komunikacji, poprzez stosowanie języka, które obie strony rozumieją. Efektem takiej pracy jest specyfikacja w postaci opowieści użytkownika, które to umożliwiają pisanie automatycznych testów akceptacji, dzięki czemu jest możliwa lepsza weryfikacja poprawności zachowania systemu. Dużym błędem[19], który popełniano podczas TDD<sup>10</sup> było zbyt dokładne testowanie kodu, a nie zachowania systemu, co utrudniało zmiany, zamiast je stymulować<sup>11</sup>. Dlatego tak ważne jest, aby pisać testy, które weryfikują zachowanie systemu, bądź poszczególnych komponentów. Tak też narodził się pomysł, aby przygotować narzędzia wspierające ten proces. W 2003 roku Dan postanowił zastąpić JUnit, własną biblioteką, która ułatwiała by pisanie testów zachowania i nazwał ją JBehave. Na przełomie ostatnich pięciu lat powstało wiele odpowiedników, bądź też rozszerzeń dla poszczególnych języków. Poniżej lista.

**.NET** NBehave[45], NSpec[46], NSpecify[47], NUnit[48], Cucumber[36]

**Java** Instinct[49], JBehave[50], JDave[51], Cucumber

<sup>10</sup>Test-Driven-Development

<sup>11</sup>Ludzie z kręgu BDD, nie zalecają pisanie testów dla metod prywatnych, i chronionych

**JavaScript** JSSpec, JSpec[52]

**C** CSpec[53]

**Scala** Specs[54]

**Ruby** Rspec[8], Cucumber

W kolejnej sekcji zostanie omówiona biblioteka Rspec, ze względu na użycie języka Ruby w projekcie, oraz biblioteka Cucumber, która daje możliwość uruchamiania opowieści użytkownika jako wykonywalnego kodu, przez co jest niezwykle użyteczna. Narzędzie które jest opisane w sekcji piątej, wspiera prezentację informacji w tym formacie.

### 3.3 Dlaczego Agile

Metodologie zwinne nie są wymysłem ostatnich lat[56], a jednak popularność ich zaczęła wzrastać w połowie lat dziewięćdziesiątych. Co więcej, model kaskadowy jest ciągle w użyciu. Powodem takiej sytuacji, jest zapewne zatwierdzenie modelu kaskadowego, jako standardu dla projektów informatycznych przez DoD<sup>12</sup>, a następnie przyłączyło się do tego NATO. Royce - który jako jeden z pierwszych dokonał analizy tej metodologii - opisał dokładnie ten proces w latach siedemdziesiątych - w artykule „Zarządzanie produkcją dużych systemów informatycznych”, i podał go jako przykład negatywny. Zwrócił uwagę na jego niską skuteczność, oraz ogromne koszty.

Jeff Sutherland w artykule „Czego mnie nauczył pierwszy Scrum”[60], zapisał cztery spostrzeżenia na temat metodologii kaskadowych względem Scrum’a:

- nie jest w stanie przewidzieć wszystkiego na początku
- nie jest w stanie dostarczyć projektu na czas
- jest mniej efektywna - mniejsza ilość funkcjonalności na jednostkę czasu programisty
- wrażenie użytkownika jest niezadowolające

Metodologia Agile zamiast walczyć z problem uzyskania całkowitej wiedzy na początku, stara się radzić z tą niewiedzą - bo jest to naturalne, że nie da się na początku wszystkiego ustalić. To samo tyczy się zmian, które wynikają w okresie produkcji - są one nawet traktowane, jako coś pożądanego. A skoro użytkownicy potrafią sprecyzować czego oczekują od systemu, gdy zobaczą jego początkową wersję, warto ją pokazać jak najwcześniej, i jak najczęściej. Zwinne zarządzanie projektem stara

---

<sup>12</sup>Departament Obrony Narodowej Stanów Zjednoczonych

się wkomponować w proces, który daje klientowi dużą kontrolę nad kształtem systemu, i to na każdym etapie prac, a nie tylko na etapie wstępnej specjalizacji. Jest to ważny element, który przyczynia się do zadowolenia z wyprodukowanego oprogramowania - co potwierdza skuteczność tej metody.

## 4 Technologie i wzorce projektowe

Choć metodologia agile nie jest związana z konkretnymi technologiami, to jednak języki, oraz środowiska dające się łatwo re-faktorować są korzystniejsze. Stosowanie się do odpowiednich wzorców, stymuluje pisanie dobrego kodu, dzięki czemu zmiany w takim systemie są efektywniejsze, i łatwiej taki system sprzężyć z innym. Jest to efekt filozofii jaką agile ma do zmian, i musimy być na nie ciągle otwarci. Zmiany ze strony klienta są nieuniknione, a mówiąc bardziej precyzyjnie są niezbędne aby produkt był wartościowy. Wpływ na to ma również użytkownik końcowego produktu, którego uwagi mogą podnieść znacznie używalność systemu, niestety informacje takie docierają późno. Czasem sami deweloperzy zauważają pewne nieprawidłowości i decydują się na zmiany. Wymaga to, aby kod był dobrze utrzymany, bez redundantności, przejrzysty i jednoznaczny. Poniżej zwrócę uwagę na sprawdzone wzorce, które pomagają osiągnąć ten cel.

### 4.1 Model-Widok-Kontroler

Wzorzec MVC<sup>13</sup> ma już za sobą prawie 30 lat, a został opracowany przez norwega Trygve Renskauga na Uniwersytecie w Oslo na przełomie lat 1978-79. Pierwotna nazwa brzmiała THING-MODEL-VIEW-EDITOR, końcową nazwę zawdzięczamy dyskusji autora z Adele Goldberg, dzięki której 10 grudnia 1979, przyjęto ostateczną nazwę. O samej przyczynie zaistnienia tego wzorca, sam autor mówi: „MVC powstało jako naturalne rozwiązanie dla powszechnego problemu dostarczenia użytkownikom kontroli nad ich własnymi danymi z wielu perspektyw”[11]. Wskazuje to na idee oddzielenia samych danych i sposobu ich prezentacji oraz modyfikacji, co jest logiczne. Zwróćmy uwagę dokładniej na poszczególne elementy oraz interakcje między nimi.

**Model** Model jako abstrakcyjna forma, reprezentacja encji, jest ważnym aspektem izolacji. Część systemu kwalifikowaną jako model, można przedstawić jako zestaw atrybutów, które ona przechowuje, oraz logikę biznesową, która jednoznacznie określa zachowanie. Zgodnie z definicją wzorca, model nie ma bezpośredniego dostępu do kontrolera czy widoku. Jeśli próby przechowywanie atrybutów poza modelem wydają się dziwne i rzadko spotykane, to umieszczanie logiki w kontrolerach, czy też widokach jest dosyć powszechną patologią, która powoduje duże problemy w momencie prób optymalizacji, czy też pielęgnacji kodu. Doświadczenie pokazuje, że trzymanie się ściśle tych prostych dwóch reguł, wymusza większy porządek w implementacji systemu. Łatwiej wtedy jest uniknąć redundancji kodu,

---

<sup>13</sup>Model-View-Controller

a optymalizacja kodu jest mniej ryzykowna. Ma to duże znaczenie zwłaszcza w aplikacjach webowych, gdzie mnogość form prezentacji danych, wymaga jednoznacznie określonej logiki na poziomie modelu.

**Widok** Widok jest odpowiedzialny za reprezentację wizualną danych, oraz interfejsów edycyjnych. Pierwotna definicja[11] zakładała, że widok jest bezpośrednio powiązany z modelem. The Passive View Pattern zakłada, że widok ma dostęp do modelu, tylko za pomocą kontrolera - nie występuje bezpośrednie odwołanie do modelu. Widok, który operuje tylko na danych przygotowanych przez kontroler jest dużo prostszy, a ponieważ wszelkie dane przechodzą przez kontroler, programista ma nad nimi większą kontrolę. Ta odmiana wzorca, jest promowana w aplikacjach webowych. Jedyna logika jaka może występować w widoku, to ta związana z kontrolkami, których zachowanie nie jest bezpośrednio związane z logiką biznesową.

**Kontroler** Kontroler ma spełniać dwa proste zadania, przyjąć komunikat od użytkownika i przygotować dane zwrotne. W kwestii pierwszej funkcjonalności, rolę kontrolera jest, aby przyjąć informacje od użytkownika, przekazać komunikat razem z danymi do odpowiedniego modelu, przyjąć wynik który model zwrócił, a przygotowane dane przesłać do odpowiedniego widoku. Dodatkowo dochodzą kwestie takie jak uwierzytelnienie użytkownika, oraz kwestie własności do obiektów.

Pilnowanie, aby nie wychodzić poza kompetencje, owocuje dużą niezależnością poszczególnych komponentów, oraz możliwością rozwoju aplikacji bez naruszania wcześniejszych elementów.

## 4.2 Representational state transfer

REST jest stylem projektowania interfejsów w systemach rozproszonych takich jak globalna sieć. Samo pojęcie zostało wprowadzone w 2000 roku przez Roy T. Fieldinga[40]. Warto wspomnieć, że jest on jednocześnie jednym z głównych autorów protokołu HTTP, oraz współtwórcą Apacha[42].

Najważniejszą regułą jest projektowanie systemu z zewnątrz, (z publicznych interfejsów), poprzez wydzielenie zasobów<sup>14</sup>. Zasób jest to źródło informacji o danym charakterze i musi być określone przez rzeczownik nigdy przez czasownik. Przykładem zasobu może być profil użytkownika w danym serwisie, daje on najczęściej możliwość pobierania informacji oraz manipulowania. Każdy zasób musi posiadać unikalny identyfikator, dla protokołu HTTP jest to URI<sup>15</sup>. Bardzo często w aplikacjach webowych, jeden zasób potrafi prezentować te same dane w różnych formatach.

---

<sup>14</sup>resource, w języku angielskim

<sup>15</sup>Uniform Resource Identifier

W postaci HTML lub w formacie XML lub JSON dla różnego typu kontrolek zbudowanych w technologii javascript, czy też RSS, ATOM dla czytników wiadomości.

Kolejną istotną sprawą jest kwestia minimalizacji metod w zasobie, czyli tak naprawdę ilości komunikatów, na które odpowiadają. Z założenia należy się ograniczyć do maksymalnie siedmiu metod. Cztery z nich są związane z operacjami na pamięci stałej<sup>16</sup>, a są to metody:

- Utwórz
- Odczytaj
- Uaktualnij
- Usuń

Kolejne dwie metody związane są z ułatwieniem operacji edycyjnych:

- Nowy - wspiera tworzenie interfejsu, oraz tworzenie nowych instancji zasobu
- Edycja - przygotowuje interfejs z wczytanymi danymi istniejącej instancji zasobu

Siódma metoda służy do pobierania z zasobu kolekcji instancji, często jest określana mianem indeks lub lista.

Wyjście ponad siedem akcji jest dopuszczalne tylko w szczególnych przypadkach. Jeśli w systemie występuje średnio więcej niż półtora akcji na zasób, sugeruje to możliwy błąd związany ze sposobem ich zaprojektowaniem.

Wielką zaletą projektowania systemu zgodnie z tymi zasadami, jest przejrzysta możliwość interakcji między różnymi aplikacjami. Platforma taka jak Rails posiada osobny moduł - `ActiveResource` - wspierający podpięcie zewnętrznych zasobów jako wewnętrzne w systemie, ale i bez tego komunikacja z innymi systemami jest przejrzysta. Ponadto przybliży to marzenie Suna o globalnej sieci, jako jednym wielkim komputerze.  
`hyphenationopo-wie-ścihyphenationdo-sto-so-wa-ne`

### 4.3 Narzędzia wykorzystane w pracy

Poniżej zamieszczony jest krótki opis poszczególnych technologii, użytych w części praktycznej, czyli języka Ruby, oraz środowiska rozwoju aplikacji Ruby on Rails. Następnie dokładnie zostanie omówiona kwestia środowiska wspierającego przygotowanie testów - `Rspec` i biblioteka uruchamiają opowieści użytkownika w formacie tekstowym `Cucumber`.

---

<sup>16</sup>często określane akronimem `CRUD` (*Create, Read, Update, Delete*)

### 4.3.1 Ruby

Jest to język opracowany w połowie lat dziewięćdziesiątych, przez Yukihiro Matsumoto<sup>17</sup>. Chciał stworzyć język w którym znalazłoby się miejsce dla wybranych cech takich języków jak Perl, Smalltalk, Eiffel, Ada i Lisp. Miał to być balans, pomiędzy programowaniem funkcyjnymi, a programowaniem imperatywnym. Pierwsze publiczne wydanie nastąpiło w 1995 roku. Ogromny wzrost popularności w ostatnich latach, zawdzięcza on, rozwojowi aplikacji Ruby on Rails. To zaś zaowocowało dużą społecznością, która znalazła jeszcze inne zastosowania tego języka.<sup>18</sup>

#### Główne cechy języka

- wszystko jest obiektem (liczba, napis tekstowy)<sup>19</sup>
- elastyczna składnia
- wsparcie dla bloków kodu
- *mixins*, które w naturalny sposób rozwiązują problem wielodziedziczenia
- obsługa błędów przez wyjątki
- garbage collector

### 4.3.2 Ruby on Rails

Jest to środowisko, które umożliwia szybki rozwój aplikacji webowych opartych na bazach danych. Środowisko to, bardzo szybko się rozwija, dzięki dużej społeczności zaangażowanej w projekt. Ruby on Rails[7] powstał poprzez wyciągnięcie go z projektu BaseCamp w 2004 roku przez Davida Heinemeiera Hanssona. Od tego czasu rozwija się bardzo prężnie, zwłaszcza na przełomie ostatnich dwóch lat.

#### Zalety środowiska

- narzucony wzorzec MVC
- bardzo dobre wsparcie dla REST'a
- abstrakcyjny *mapper* bazy danych ActiveRecord
- rozbudowane wsparcie dla testów (Test Unit, Rspec-on-Rails, Cucumber)
- mnóstwo pluginów[59], wraz ze wsparciem do wydzielania części systemu

---

<sup>17</sup>Yukihiro Matsumoto jest często nazywany wśród ludzi związanych z Ruby jako „matz”.

<sup>18</sup>Jako przykład może posłużyć powłoka systemowa - która może być użyta jako zamiennik dla basha - w której posługujemy się językiem Ruby, nazywa się rush(<http://rush.heroku.com>). Warto nadmienić że wiele skryptów dla systemów \*nix, również jest już pisane w tym języku.

<sup>19</sup>Wyjątek stanowi tu referencja do obiektu

### 4.3.3 Rspec

Jedno z założeń BDD wiąże się z automatyzacją testów w aplikacji. Aby osiągnąć to w efektywny sposób, potrzebny jest dedykowany język<sup>20</sup>, dla tego konkretnego zadania. Pierwsze wzmianki na temat powstania języka Ruby taki DSL, przedstawił Dave Astel[22] w artykule „Nowe Spojrzenie na TDD”[24] z 2005 roku. Zwrócił on uwagę na zbudowanie takiego DSL’a, który będzie wspierał opis zachowania wraz z weryfikacją tego zachowania. Zaproponował kilka zmian związanych z nazewnictwem, zamiast zaczynać nazwy metod od słowa „test”, zacząć od słowa „should”, używać słowa „Context” zamiast „TestCase” oraz „shouldBe” w zamian „assert”. Celem takich zabiegów, było położenie nacisku na testowanie zachowania. Poniżej jest umieszczony przykład zaczerpnięty z artykułu „Wstęp do BDD”[25]

```
1  require 'spec'
2  require 'movie'
3  require 'movie_list'
4  class EmptyMovieList < Spec::Context
5    def setup
6      @list = MovieList.new
7    end
8    def should_have_size_of_0
9      @list.size.should_equal 0
10   end
11   def should_not_include_star_wars
12     @list.should_not_include "Star Wars"
13   end
14 end
15 class OneMovieList < Spec::Context
16   def setup
17     @list = MovieList.new
18     star_wars = Movie.new "Star Wars"
19     @list.add star_wars
20   end
21   def should_have_size_of_1
22     @list.size.should_equal 1
23   end
24   def should_include_star_wars
25     @list.should_include "Star Wars"
26   end
27 end
```

Przez okres ostatnich 3 lat, *framework* przeszedł wiele modyfikacji i usprawnień. Dzięki pracy wielu osób[26, 27], formalny zapis testów jest jeszcze bardziej przejrzysty i intuicyjny. Gdyby uaktualnić przykład zamieszczony powyżej, biorąc pod uwagę aktualną specyfikację *framework*’a, będzie on miał postać zbliżoną do zamieszczonej poniżej.

---

<sup>20</sup>Domain specific language (DSL)



```

1  require 'spec'
2  require 'movie'
3  require 'movie_list'
4
5  describe MovieList do
6    context "empty movie list" do
7      before(:all) do
8        @list = MovieList.new
9      end
10     it "should have size of 0" do
11       @list.size.should == 0
12     end
13     it "should not include 'Star Wars'" do
14       @list.should_not include('Star Wars')
15     end
16   end
17   context "'Star Wars' item on movie list" do
18     before(:all) do
19       @list = MovieList.new
20       star_wars = Movie.new "Star Wars"
21       @list.add star_wars
22     end
23     it "should have size of 1" do
24       @list.size.should == 1
25     end
26     it "should include 'Star Wars'" do
27       @list.should include("Star Wars")
28     end
29   end
30 end

```

Tak napisany spec<sup>21</sup> ma dużo większy wpływ na postrzeganie zachowania systemu, który chcemy weryfikować. Jest to również wielce pomocne przy generowaniu raportów, które notabene są pewną formą dokumentacji systemu - opisują jego pożądane zachowanie.

RSpec Results		4 examples, 0 failures Finished in 0.035787 seconds
MovieList empty movie list		
	should have size of 0	
	should not include 'Star Wars'	
MovieList 'Star Wars' item on movie list		
	should have size of 1	
	should include 'Star Wars'	

Aktualnie liderem projektu jest David Chelimsky[23]. Astel w dalszym ciągu jest aktywnym uczestnikiem w projekcie, zwłaszcza w promowaniu metodologii BDD.

<sup>21</sup>Zamiast używać słowa test, Astel zaczął nazywać taką weryfikację „spec”, prawdopodobnie od angielskiego słowa specification

Sam Rspec nie doczekał się jeszcze żadnej książki, która zebrałaby informacje na temat BDD, Rspec i Cucumber. W planach jest wydanie takiej książki[32] w kwietniu 2009 roku. Będzie to praca zbiorowa Chelimskiego, Astela, Northa, Dennisa[29], Hellesoya[30] i Helmkampa[31]. Biorąc pod uwagę autorów, będzie to z pewnością pozycja, z którą warto się zapoznać.

#### 4.3.4 Cucumber

Idealnym rozwiązaniem byłaby możliwość pisania testów integracyjnych przez osoby, które nie są programistami, a które mogłyby powtarzalnie weryfikować system. Niestety nie jest to w pełni osiągalne. Pewnym postępowaniem w tym kierunku, jest tworzenie tekstowych opowieści użytkownika, z wydzielonymi scenariuszami, które są zrozumiałe dla ludzi biznesu. Jedną z najciekawszych implementacji jest Cucumber.

Jego historia zaczyna się od pierwszej próby - dla Rspec'a - którą był RBehave[28], napisany przez Northa w 2006 roku. Niestety nie było to jeszcze czymś akceptowalnym dla klienta biznesowego. Kolejnym krokiem był Story Runner[33], nad którym pracował Astel, we współpracy z Northem - pierwsza wersja była dostępna w drugiej połowie 2007 roku. Projekt bazował na wcześniejszym RBehave, jednak jego największym sukcesem, była możliwość uruchamiania, z pliku tekstowego, opowieści użytkownika napisanych po angielsku. Początkiem kwietnia 2008 roku, Aslak Hellesoy rozpoczął prace nad nowym środowiskiem do uruchamiania opowieści użytkownika z czystego pliku, a projekt został ostatecznie nazwany Cucumber (pierwotnie nazywał się Stories). Po zaledwie pięciu miesiącach prac, został on zamiennikiem dla Story Runner[37]. Chelimsky zmianę tę uargumentował tym, że Cucumber posiada większą ilość funkcjonalności, i jest prostszy w konfiguracji. Poniżej wymienione są jego cechy.

- gramatyczny parser opowieści użytkownika w postaci tekstowej
- pozwala pisać opowieści użytkownika w ponad 20 językach (również po polsku)
- lepsze śledzenie błędów (od story runnera)
- prosta konfiguracja (względem Story Runner)
- warunkowa kontynuacja kroku

Opowieści są pisane w języku dedykowanym, którego struktura umożliwia jednoznaczne zrozumienie w świecie biznesu - język ten nazywa się Gherkin[35]<sup>22</sup>. To właśnie

---

<sup>22</sup>Gherkin to po angielsku mały ogórek

ten moduł odpowiada za wsparcie dla wielu języków[34]. Możliwe jest również dodawanie nowych modułów, lub adaptacji dla własnych potrzeb, już istniejących. Sam język posiada niewielką gamę słów kluczowych<sup>23</sup>:

- Właściwość (*Feature*) - określa jaką właściwość opisujemy
- Scenariusz (*Scenario*) - opisuje konkretny scenariusz działania użytkownika
- Dane (*Given*) - określa dane wejściowe (np: użytkownika znajduje się w widoku systemu; użytkownik jest zalogowany)
- Jeżeli (*When*) - zwraca uwagę na czynności które podejmuje użytkownik
- Wtedy (*Then*) - określa pożądany efekt końcowy
- Oraz (*And*) - łączy kroki w sekcjach (Dane, Jeżeli, Wtedy)

Poniżej zamieszczony jest przykład.

---

<sup>23</sup>w nawiasach podane angielskie odpowiedniki

### *Właściwość* Zarządzanie funkcjonalnościami

W celu umożliwienia wprowadzania opisów funkcjonalności

Jako klient lub programista

Chcę mieć możliwość zarządzania(dodawanie, edytowanie, usuwanie, przeglądania) funkcjonalności

#### *Scenariusz*      Dodanie nowego opisu funkcjonalności

*Dane*            jestem na stronie tworzenia opisu funkcjonalności

*Jeżeli*          wpisze "Zarządzanie Scenariuszami" w pole 'tytuł'

*Oraz*            "umożliwić zarządzanie opisami funkcjonalności" w pole 'cel'

*Oraz*            "klient lub programista" w pole 'jako'

*Oraz*            "mieć możliwość zarządzania(dodawanie, edytowanie, usuwanie, przeglądania) funkcjonalności" w polę 'chcę'

*Oraz*            nacisnę przycisk utwórz

*Wtedy*          powinien zostać przeniesiony na stronę funkcjonalności

#### *Scenariusz*      Przeglądania funkcjonalności ze względu na date dodania

*Dane*            Jestem na stronie z listy funkcjonalności

*Jeżeli*          wybiorę zakres 'od' z kalendarza '01.01.2009'

*Oraz*            wybiorę zakres 'do' z kalendarza '10.01.2009'

*Wtedy*          powinienem zobaczyć funkcjonalności dodane w okresie od '01.01.2009' do '10.01.2009'

## 5 Wspomaganie procesu produkcji oprogramowania

Jak zostało wspomniane na początku, metodologie agile nie są związane z konkretną technologią, czy też narzędziem, jednakże bardzo proste narzędzia potrafią usprawnić ten proces. Taki też cel jest postawiony przed tym narzędziem - ma wspomagać proces agregowania wymagań od klienta, ułatwiać pisanie dokumentacji dla projektów, oraz analizy wykonanej pracy, względem tej wyznaczonej. Od strony deweloperów ma ponadto ułatwiać pisanie automatycznych testów.

Potrzeba powstania takiego narzędzia, była efektem współpracy z klientami, którzy często zmieniali wymagania dotyczące projektów, czy też dokładali mnóstwo pracy w trakcie realizacji zadań. Było to powodem wielu problemów i komplikacji względem rozliczania pracy wykonanej, gdyż nie było dobrego sposobu śledzenia zmian.

Kwestią drugą, był problem związanym z brakiem automatycznych testów. Powodowało to częste problemy, które pojawiały się najczęściej tuż po przekazaniu kodu do produkcji. To uzmysłowiło jak ważne jest posiadanie testów integracyjnych.

Środowisko Cucumber do uruchamiania opowieści tekstowych, na podstawie których weryfikowane jest zachowanie systemu, jest w tym przypadku idealnym rozwiązaniem. Dodając do tego intuicyjny interfejs tworzenia takich opowieści, dzięki któremu klienci sami by mogli wprowadzać te informacje, zaczyna to wyglądać coraz bardziej interesująco. Uzupełnieniem powyższych zalet jest możliwość śledzenia zmian. Gdy weźmiemy te trzy na pozór proste właściwości, otrzymamy wartościowe narzędzie dla developerów.

### 5.1 Specyfikacja funkcjonalna

Poniżej zostanie zwrócona uwaga na główne właściwości projektu, oraz cele, które są ważne dla realizacji tego projektu

#### 5.1.1 Opowieści użytkownika

Jest to w zasadzie trzon całej aplikacji. O samej wartości opowieści użytkownika dużo powiedziano w sekcjach poświęconych BDD i Cucumber. Zalety opowieści w postaci czystego tekstu są ogromne, jednakże czasem potrzeba jest manipulacja prowadzona na tych danych, jak na przykład eksport do dokumentacji ładnie sformatowanych opowieści. Kolejną istotną kwestią jest wygoda wprowadzania tych opowieści. deweloperzy lubią specyficzne<sup>24</sup> edytory tekstowe, jednakże inni preferują dostosowane interfejsy, i właśnie dla nich to narzędzie będzie użyteczne.

---

<sup>24</sup>Przykładem może być vi(m) czy też emacs.

### 5.1.2 Śledzenie zmian

Wielokrotnie była już zwrócona uwaga na sposób traktowania zmiany w metodologiach zwinnych, które są mile widziane na każdym etapie prac. Chociaż klienci bardzo chętnie te zmiany wprowadzają, to już trudniej im wszystkie modyfikacje spamiętać. Nie zawsze są świadomi jak bardzo ich pierwotna wizja różni się od wersji końcowej. Możliwość wykazania tych zmian w przejrzysty sposób, daje dwie korzyści. Po pierwsze, klient może zobaczyć jak proces, w którym bierze aktywny udział wpływa na poprawę jakości produktu. Drugą sprawą są kwestie finansowe: taki raport, to solidny argument dla negocjatorów.

Ważne jest tutaj śledzenie zmian, ze względu na konkretną właściwość systemu, oraz ze względu na wydzielony okres czasu.

### 5.1.3 Eksport opowieści użytkownika

Przechowując dane o odpowiedniej strukturze, mamy możliwość zautomatyzowania procesów monottonnych, poprzez prezentację danych w innych formatach.

**Cucumber** Ten format jest sztandarowy dla tego projektu. Dzięki eksportowaniu do niego danych, zautomatyzowany zostaje proces tworzenia plików dla poszczególnych właściwości. Dzięki temu, zmiany w specyfikacji, można szybko aktualizować dla systemu automatycznych testów.

**PDF** Potrzeba tworzenia dokumentacji technicznej systemu implementowanego jest bardzo ważna. Gdy weźmiemy pod uwagę fakt, że opowieści użytkownika to świetna dokumentacja, to automatyczne utworzenie dokumentu pdf ze wszystkimi właściwościami, docenią wszyscy - nawet ci, którzy lubią pisać dokumentacje.

## 5.2 Implementacja narzędzia

Opowieści użytkownika zostały zapisane za pomocą trzech modeli: „Feature”, „Scenario”, „ScenarioStep”. Pokróćce zostaną teraz opisane. Ponadto poniżej znajduje się również diagram modeli, oraz diagram klas.

**Feature** - właściwość przechowuje tytuł opowieści, powód wprowadzenia, kto będzie z niego korzystał, oraz krótkie sformułowanie co dana funkcjonalność wnosi do systemu (co użytkownik może). Agreguje on również scenariusze.

**Scenario** - scenariusz przechowuje tylko tytuł, oraz agreguje główne typy kroków - są kroki takie jak „Dane”, „Jeżeli”, „Wtedy”.

**ScenarioStep** - krok scenariusza, posiada typ, oraz instrukcje. Typy główne zostały wymienione wcześniej, możliwe podtypy które są agregowane przez krok główny to: „ale”, „oraz”.

Ważnym elementem systemu jest funkcjonalność dzięki której możliwy jest monitoring zmian które są dokonywane na obiektach typu „Feature”, „Scenario” oraz „ScenarioStep”. Dzięki czemu nie jest problemem określenie kto danej zmiany dokonał, a jest to ważne jak zostało to już wcześniej wyjaśnione.

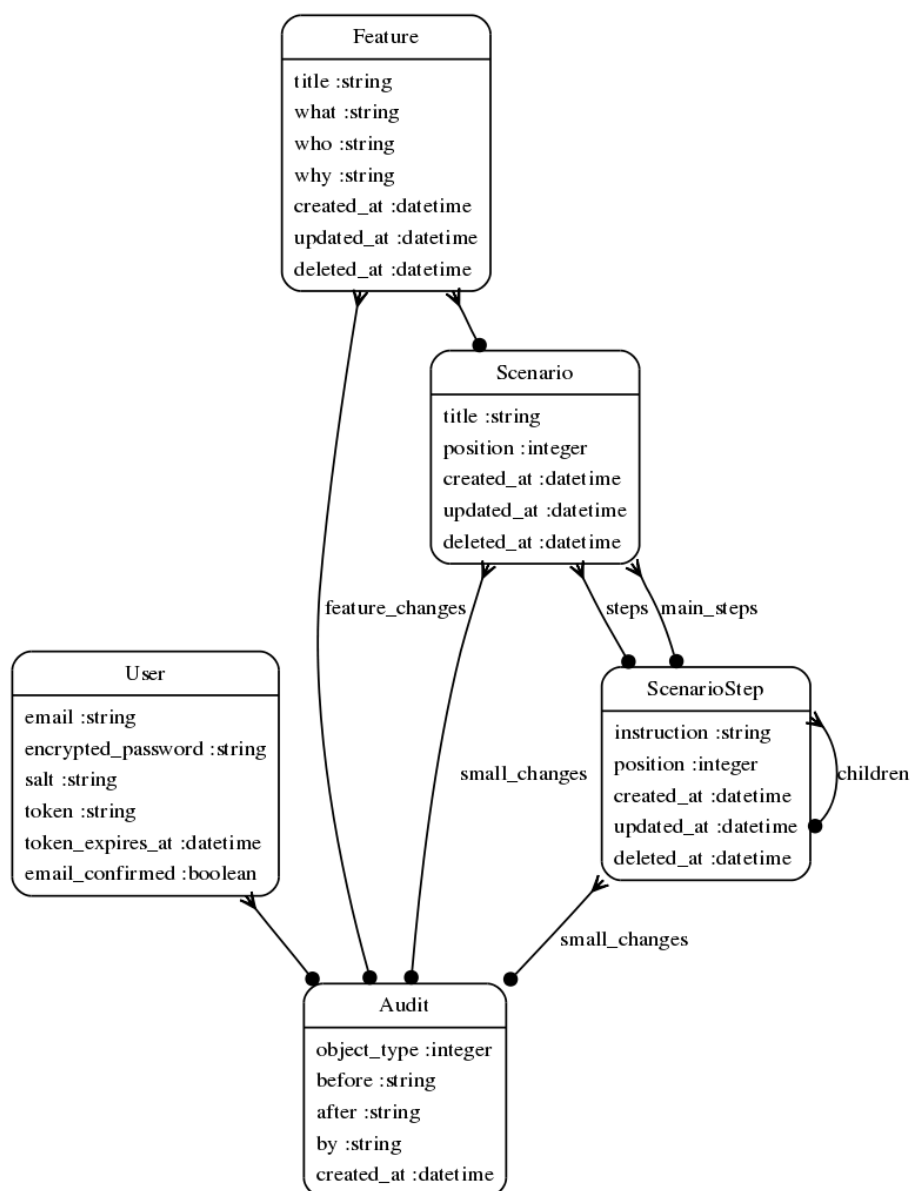
Wprowadzenie zmiany polega, tak jak zawsze, na nadpisaniu atrybutów, jednak zanim nastąpi zapis na medium trwałe - w tym przypadku bazę danych - następuje kopiowanie atrybutów, które uległy zmianie. Jest to możliwe dzięki pobraniu aktualnych wartości z bazy. Dla każdego zmienionego atrybutu tworzony jest obiekt typu Audit. Zapisywana jest w nim wartość atrybut przed zmianą oraz nowa wartość, wraz z informacją o użytkowniku który tej zmiany dokonał. Przechowywana jest również informacja o obiekcie oraz atrybucie dla którego audyt jest wykonywany. Powiązanie między audytem a różnymi modelami jest możliwe dzięki polimorficznemu powiązaniu. Wszystkie klucze które są reprezentowane przez ciąg znaków - nazwa modelu, atrybutu - są mapowane na wartości liczbowe całkowite, dzięki czemu osiągany jest zysk na ładowaniu danych z bazy i przesyłaniu tych danych za pośrednictwem sieci. Dodatkowym atutem jest mniejsza ilość wykorzystywanej pamięci operacyjnej która jest niezbędna przy operacjach na dużych zbiorach danych. Przykładowy fragment kodu który triggeruje, zapis zmian dla czterech atrybutów klasy Feature, jest ukazany poniżej.

```
1  before_save :audit_changes
2  before_destroy :audit_deletion
3
4  audit_only :title, :who, :what, :why
5
6  def audit_changes
7    changed.each do |attribute|
8      small_changes.build(:before => send("#{attribute}_was"),
9                          :after => send(attribute),
10                         :attribute => attribute.to_sym,
11                         :committed_by => committed_by) if audit_only.include?(attribute)
12    end
13  end
14
15  def audit_deletion
16    audit_only.each do |attribute|
17      small_changes.build(:before => send("#{attribute}_was"),
18                          :after => nil, :attribute => attribute,
19                          :committed_by => committed_by).save
20    end
21  end
```

### 5.2.1 Diagram Modeli

Poniższy diagram przedstawia modele użyte w aplikacji, oraz relacje jakimi są połączone. U góry każdego prostokąta znajduje się nazwa klasy, poniżej wymienione są

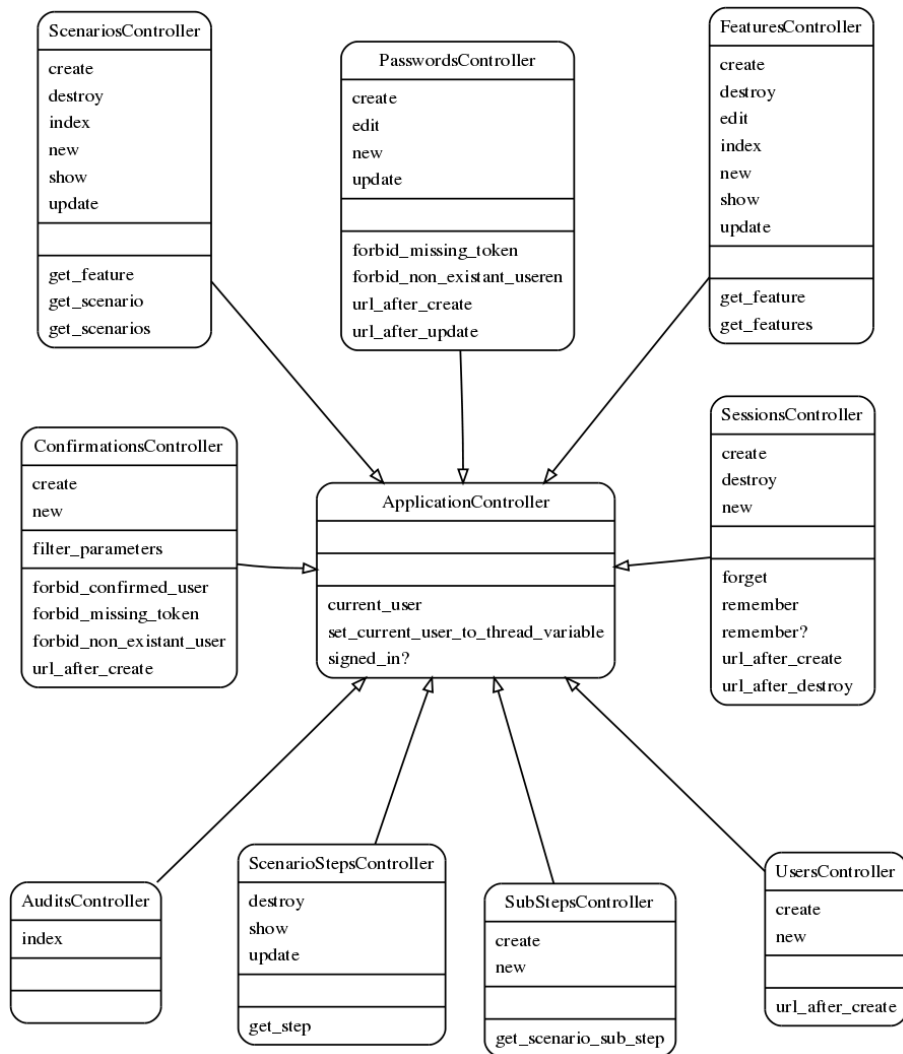
wszystkie zmienne instancyjne które są przechowywane w bazie wraz z typem.



## 5.2.2 Diagram Kontrolerów

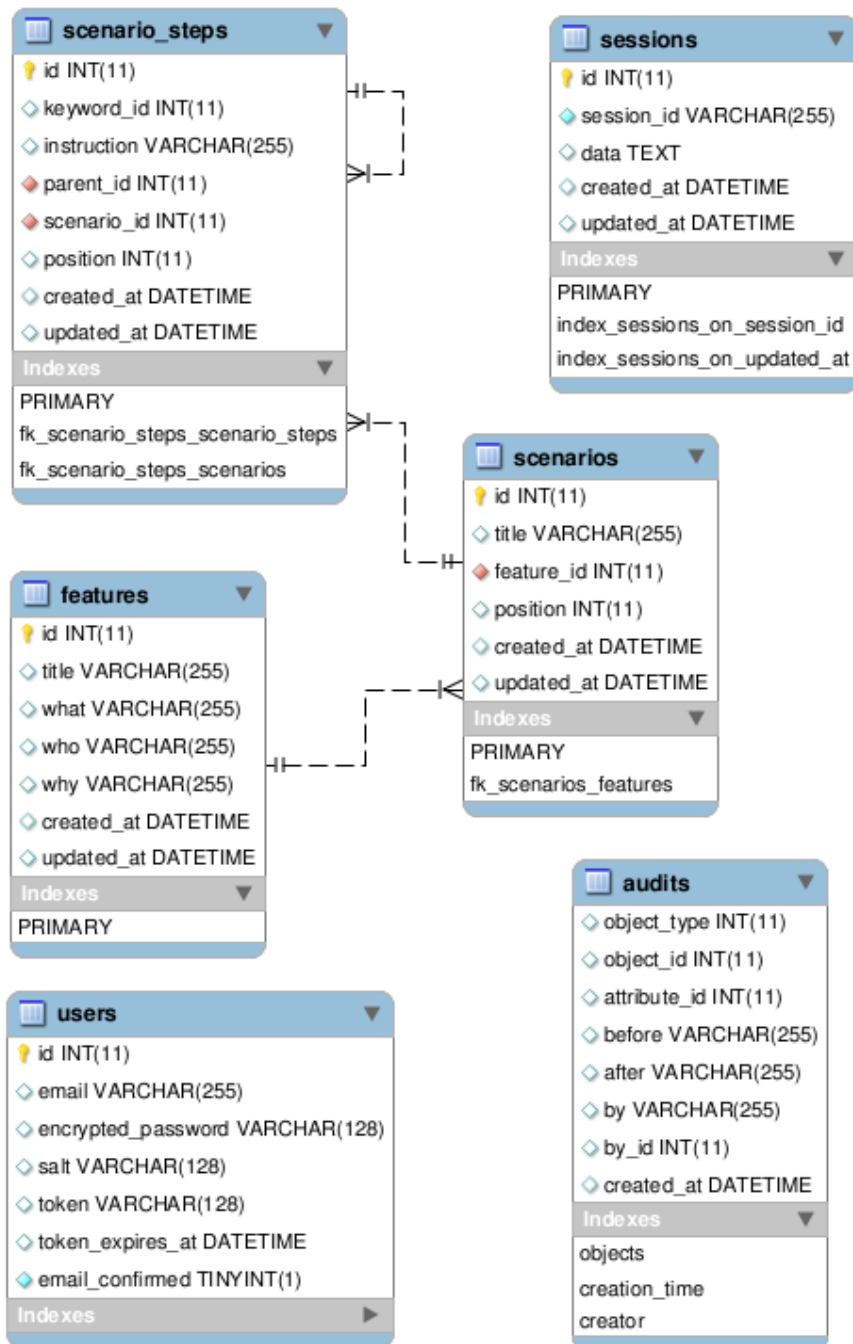
Struktura kontrolerów jest przedstawiona na poniższym rysunku. Wszystkie kontrolery dziedziczą z głównego kontrolera ApplicationController. Każdy prostokąt reprezentujący jeden kontroler jest jednocześnie zasobem - poza kontrolerem głównym. Wszystkie są zaprojektowane zgodnie z ideą REST, ilość metod publicznych jest minimalna, wychodzą poza standardowe cztery metody CRUD, o dwie metody pomocnicze *new* - przygotowanie formularza tworzenia, *index* - prezentacja kolekcji obiektów. U góry każdej reprezentacji kontrolera jest jego nazwa, następna jest lista metod publicznych, a na samym dole metody prywatne.





### 5.2.3 Diagram Bazy Danych

Schemat bazy danych jest narysowany za pomocą aplikacji MySQL Workbench dzięki możliwości wstecznej inżynierii. Zaprezentowana jest struktura poszczególnych tabel, relacji kluczy, oraz indeksy które przyspieszają działanie bazy



## 6 Prezentacja aplikacji

W tej sekcji zostaną zaprezentowane zrzuty ekranu wraz z krótkimi opisemami.

**Ekran powitalny oraz logowanie** Jako login używany jest adres email, w prawym górnym rogu widać menu pozwalające przejść na stronę tworzenia nowego konta i odzyskiwania konta, wraz z możliwością powrotu na stronę logowania.

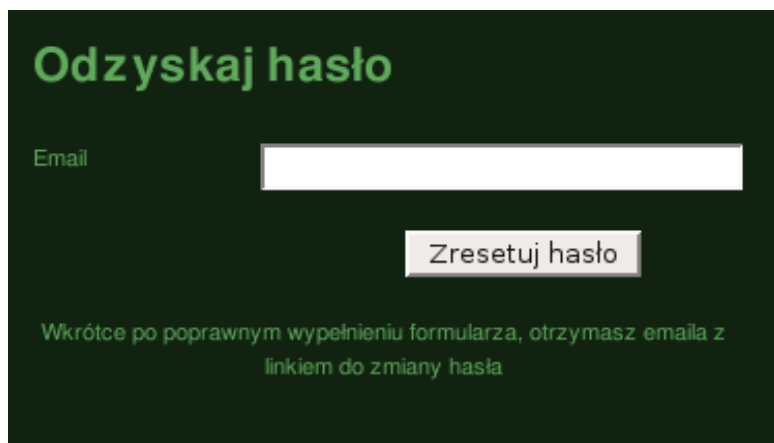


**Tworzenie nowego konta** Utworzenie nowego konta wymaga podania adresu email oraz zdefiniowania hasła. Następnie użytkownik otrzymuje wiadomość na swoją skrzynkę pocztową wraz z linkiem dzięki któremu może aktywować konto, przed aktywacją nie może się zalogować do systemu. W ten sposób następuje weryfikacja prawdziwości adresu email podanego w formularzu.

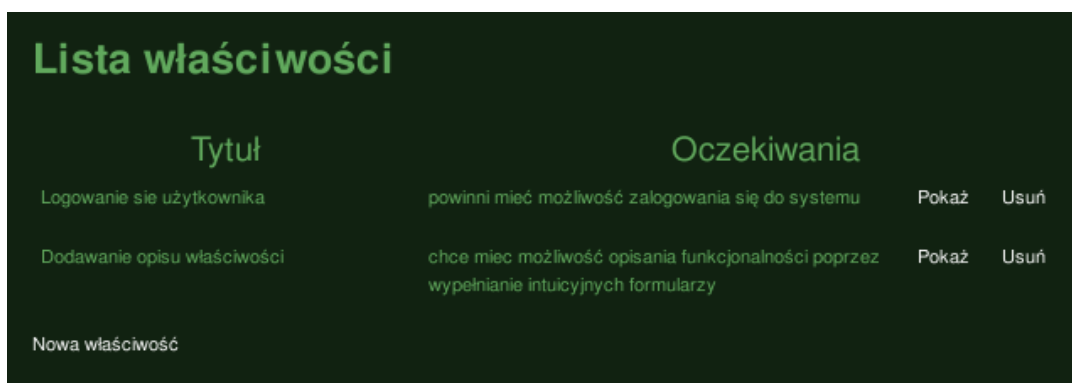


**Odzyskiwanie hasła** Jest to wartościowa funkcjonalność, dzięki której użytkownik który zapomni jakie hasło ustawił, a który ma dostęp do swojej skrzynki pocztowej, ma możliwość zmiany hasła. Gdy wpisze w formularzu swój adres email,

oraz ten adres email znajduje się w systemie - w bazie użytkowników - wtedy zostanie wysłany na skrzynkę pocztową email z linkiem do strony na której jest możliwa zmiana hasła za pomocą formularza. Dzięki takiemu rozwiązaniu hasła nie muszą być przechowywane w systemie jako tekst, aby możliwe było przypomnienie hasła, nigdy też nie są wysyłane pocztą email - co ma wpływ na bezpieczeństwo systemu.



**Lista właściwości** Jest to centralny punkt systemu, w tym widoku jest możliwe przeglądanie wszystkich właściwości które już zostały wprowadzane oraz przejście do szczegółowego widoku. Ponadto możliwe jest usunięcie właściwości czy też przejście do formularza tworzenia właściwości.



Tytuł	Oczekiwania		
Logowanie się użytkownika	powinni mieć możliwość zalogowania się do systemu	Pokaż	Usuń
Dodawanie opisu właściwości	chce mieć możliwość opisania funkcjonalności poprzez wypełnianie intuicyjnych formularzy	Pokaż	Usuń

Nowa właściwość

**Tworzenie właściwości** Formularz „Nowa właściwość” jest pierwszym krokiem do utworzenia opowieści użytkownika. Na początku jest definiowana część związana z dokumentacją - nie podlega ona testom. Przy ustawieniu focusu - ustawienie kursora w polu edycyjnym - pojawia się chmurka z podpowiedzią i przykładem aby użytkownik mógł lepiej zrozumieć jaką formę powinien przyjąć tekst w danym polu edycyjnym.

## Nowa właściwość

Tytuł

Powód wprowadzenia

Kto będzie użytkownikiem

Oczekiwania

[Powrót](#)

DLaczego ta funkcjonalność jest potrzebna? Jaką biznesową wartość wnosi?, przykład: "W celu determinowania autorstwa zawartości dodawanej do systemu"

**Widok właściwości** Ten widok daje najpełniejsze informacje na temat danej właściwości. Widać część która dokumentuje oraz scenariusze. Z tego widoku można przejść do tworzenia nowego scenariusza, widoku historia zmian, oraz wyeksportować opowieść.

[Dodaj scenariusz](#)
[Pokaż Historie Zmian](#)
[Format cucumber](#)
[Format PDF](#)

**Właściwość** Logowanie sie użytkownika

W celu determinowania autorstwa zmian  
wszyscy posiadający konto  
powinni mieć możliwość zalogowania się do systemu

**Scenariusz** ⊖ Wejście na stronę logowania

- + **Dane** jestem na stronie logowania
- + **Jeżeli** kliknę link załoguj
- + **Wtedy** zostanę przekierowany do strony logowania
- ⊖ **Oraz** zobaczę formularz z polami login i hasło

[Powrót](#)

**Dodanie scenariusza** Formularz ten został zbudowany na zakładkach aby zmotywować użytkownika do wprowadzania w pierwszej kolejności informacji efekcie

końcowym danego scenariusza, a w kolejnych stan wejściowy systemu, oraz sekwencje kroków. Podobnie jak w formularzu tworzenia właściwości, jest zaimplementowana pomoc w postaci chmurek.

**Dodanie nowego scenariusza**

Tytuł scenariusza

Wtedy **Jeżeli** Dane

+

-

Oraz ▾

element warunków definiujący efekt końcowy dla danego scenariusza. przykład: "zostanę przekierowany do strony logowania", /oraz/ "zobaczę formularz z polami login i hasło ]"

Powrót

---

**Dodanie nowego scenariusza**

Tytuł scenariusza

Wtedy **Jeżeli** Dane

+

krok w sekwencja czynności które muszą zostać wykonane aby uzyskać efekt końcowy, przykład: "kliknę link zaloguj"

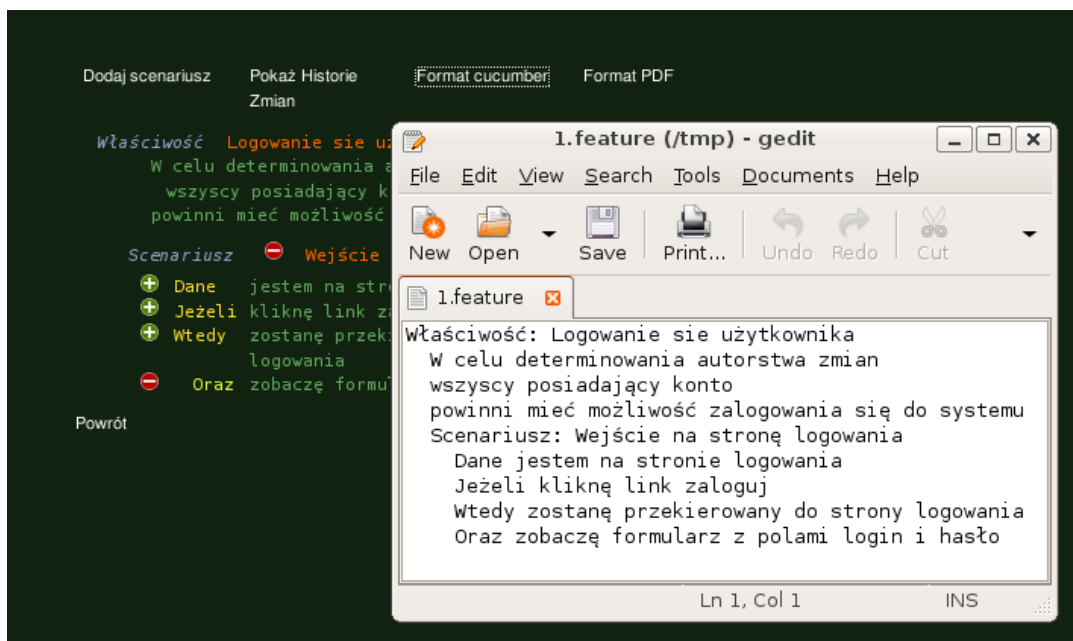
Powrót

**Edycja** Aby maksymalnie ułatwić edycja danych wprowadzonych, zastosowany został edytor w miejscu wyświetlania. W momencie kliknięcia na wpis następuje podmiana zawartości na pole edytora, w momencie kliknięcia klawisza enter następuje zbudowania formularza i wysłanie go do serwera, w odpowiedzi dostaje obiekt który edytował, i wyświetla jego zawartość na podstawie danych otrzymanych od serwera. Dzięki czemu nie ma problemu z wyświetlaniem niepoprawnych danych. Sekwencja całego procesu jest przedstawiona na trzech poniższych zrzutach ekranu.

<b>Właściwość</b> Dodawanie opisu właściwości Aby w przejrzysty sposób opisywać funkcjonalności w systemie, aby jak najwięcej osób - zwłaszcza nie informatycznych - je rozumiało, by łatwiej było wspólnie tworzyć systemy informatyczne klient, system architect, menadżer projektu chce mieć możliwość opisanie funkcjonalności poprzez wypełnianie intuicyjnych formularzy	
<b>Scenariusz</b>	<b>➖</b> Przejście na stronę formularza tworzenia + Dane    jestem na stronie listy właściwości - Oraz    jestem zalogowany + Jeżeli   kliknę na link 'Utwórz nową właściwość' + Wtedy   zobaczę formularz polami do wprowadzania opisu
<b>Właściwość</b> Dodawanie opisu właściwości Aby w przejrzysty sposób opisywać funkcjonalności w systemie, aby jak najwięcej osób - zwłaszcza nie informatycznych - je rozumiało, by łatwiej było wspólnie tworzyć systemy informatyczne klient, system architect, menadżer projektu chce mieć możliwość opisanie funkcjonalności poprzez wypełnianie intuicyjnych formularzy	
<b>Scenariusz</b>	<b>➖</b> Przejście na stronę formularza tworzenia + Dane    jestem na stronie listy właściwości - Oraz    saving... + Jeżeli   kliknę na link 'Utwórz nową właściwość' + Wtedy   zobaczę formularz polami do wprowadzania opisu
<b>Właściwość</b> Dodawanie opisu właściwości Aby w przejrzysty sposób opisywać funkcjonalności w systemie, aby jak najwięcej osób - zwłaszcza nie informatycznych - je rozumiało, by łatwiej było wspólnie tworzyć systemy informatyczne klient, system architect, menadżer projektu chce mieć możliwość opisanie funkcjonalności poprzez wypełnianie intuicyjnych formularzy	
<b>Scenariusz</b>	<b>➖</b> Przejście na stronę formularza tworzenia + Dane    jestem na stronie listy właściwości - Oraz    jestem zalogowany jako + Jeżeli   kliknę na link 'Utwórz nową właściwość' + Wtedy   zobaczę formularz polami do wprowadzania opisu

**Eksport**   Jednym z założeń systemu było wspomaganie pisania testów integracyjnych, cel ten jest osiągnięty dzięki eksportu danych do formatu języka Gherkin, czyli do formatu czystego tekstu z wcięciami. Wystarczy go zapisać do odpowied-

nego katalogu, a zaraz po tym możemy zacząć definiować kroki które wcześniej zapisaliśmy w scenariuszach.



**Historia zmian** Jest to lista ułożona w porządku chronologicznym odwrotnym. Czerwone tło i przekreślenie jest ustawiane na wyrazach które zostały usunięte, natomiast na jasnozielonym z białą czcionką wyrazy które zostały dodane. Wyświetlana jest również informacja kto tej zmiany dokonał oraz do jakiego obiektu należy to pole - właściwość składa się z trzech różnych modeli.

michalczyz@gmail.com	Scenariusz	Tytuł	Wejście na stronę logowania	2009-02-23T18:25:27Z
michalczyz@gmail.com	Krok scenariusza	Instrukcja kroku	Jeżeli kliknę link zaloguj	2009-02-23T18:25:27Z
michalczyz@gmail.com	Krok scenariusza	Instrukcja kroku	Wtedy zostaną przekierowani do strony logowania	2009-02-23T18:25:27Z
michalczyz@gmail.com	Krok scenariusza	Instrukcja kroku	Oraz zobaczę formularz z polami login i hasło	2009-02-23T18:25:27Z
michalczyz@gmail.com	Krok scenariusza	Instrukcja kroku	Dane jestem na stronie logowania	2009-02-23T18:25:26Z
michalczyz@gmail.com	Scenariusz	Tytuł	<del>Wejście na stronę logowania</del>	2009-02-23T18:24:15Z
michalczyz@gmail.com	Krok scenariusza	Instrukcja kroku	<del>Wtedy zostaną przekierowani na stronę logowania</del>	2009-02-23T18:24:14Z
michalczyz@gmail.com	Krok scenariusza	Instrukcja kroku	<del>Jeżeli</del>	2009-02-23T18:24:14Z
michalczyz@gmail.com	Krok scenariusza	Instrukcja kroku	<del>Dane</del>	2009-02-23T18:24:14Z
michalczyz@gmail.com	Scenariusz	Tytuł	Wejście na stronę logowania	2009-02-23T18:23:28Z
michalczyz@gmail.com	Krok scenariusza	Instrukcja kroku	Jeżeli	2009-02-23T18:23:28Z
michalczyz@gmail.com	Krok scenariusza	Instrukcja kroku	Wtedy zostaną przekierowani na stronę logowania	2009-02-23T18:23:28Z
michalczyz@gmail.com	Krok scenariusza	Instrukcja kroku	Dane	2009-02-23T18:23:28Z
michalczyz@gmail.com	Właściwość	Tytuł	Logowanie się <del>użytkownika</del> użytkownika	2009-02-22T16:54:06Z
michalczyz@gmail.com	Właściwość	Oczekiwania	<del>Użytkownicy</del> powinni mieć możliwość zalogowania się do systemu	2009-02-21T20:38:00Z
michalczyz@gmail.com	Właściwość	Tytuł	Logowanie się <del>użytkownika</del> użytkowników	2009-02-21T20:37:35Z
michalczyz@gmail.com	Właściwość	Powód	W celu determinowania autorstwa zmian	2009-02-21T19:59:02Z
michalczyz@gmail.com	Właściwość	Wprowadzenia		
michalczyz@gmail.com	Właściwość	Kto będzie użytkownikiem	wszyscy posiadający konto	2009-02-21T19:59:02Z
michalczyz@gmail.com	Właściwość	Oczekiwania	<del>Użytkownicy</del> mogą się zalogować	2009-02-21T19:59:02Z
michalczyz@gmail.com	Właściwość	Tytuł	Logowanie się użytkowników	2009-02-21T19:59:02Z



## 7 Podsumowanie

Pierwszy cel pracy - przybliżenie metodologii zwinnych - został osiągnięty poprzez zaprezentowanie manifestu, oraz dwunastu zasad agile. Została również zwrócona uwaga na stosowaną od ponad dwudziestu lat metodologię organizacji pracy jakim jest Scrum. Wsparciem dla samego procesu budowania systemu jest BDD. Jak już zostało wspomniane to co różni metodologie zwinne to nacisk na współpracę, nie na narzędzia.

Kolejnym etapem pracy było zbudowanie aplikacji która wspierała by pisanie testów. Cel został osiągnięty, dzięki właściwemu układowi formularzy, oraz użyciu podpowiedzi tworzenie opowieści użytkownika jest łatwiejsze. Dzięki temu osoby nie muszą znać dokładnie składni języka Gherkin, potrafią dobrze napisać testy akceptacyjne do swojego systemu, które po drobnych korektach dają wstęp do pisania automatyzacji testowania.

Testowanie aplikacji ma ogromne znaczenie w systemach informatycznych, jego automatyzacja ma ogromny wpływ na szybkość, oraz jakość tworzonego systemu. Dzięki temu narzędziu będzie to znacznie ułatwione zadanie.

## Literatura

- [1] Alan Cooper - The Wisdom of Experience  
<http://www.cooper.com/journal/agile2008> - 10.01.2009
- [2] Agile Manifesto - official webpage  
<http://agilemanifesto.org/> - 10.01.2009
- [3] Agile Manifesto - Authors  
<http://agilemanifesto.org/authors.html> - 10.01.2009
- [4] Principles behind the Agile Manifesto  
<http://agilemanifesto.org/principles.html> - 10.01.2009
- [5] Tłumaczenie Agile Manifest  
[http://pl.wikipedia.org/wiki/Manifest\\_Agile](http://pl.wikipedia.org/wiki/Manifest_Agile) - 10.01.2009
- [6] Oficjalna strona języka Ruby  
<http://www.ruby-lang.org/pl/> - 10.01.2009
- [7] Oficjalna strona frameworku Ruby on Rails  
<http://www.rubyonrails.pl/> - 10.01.2009
- [8] Oficjalna strona frameworku Rspec  
<http://rspec.info> - 10.01.2009
- [9] Wiki biblioteki Remarkable  
<http://wiki.github.com/carlosbrando/remarkable> - 10.01.2009
- [10] Bill Venners - Orthogonality and the DRY Principle  
<http://www.artima.com/intv/dry.html> - 10.01.2009
- [11] Trygve Reenskaug - The original MVC reports Trygve Reenskaug Dept. of Informatics University of Oslo  
[http://heim.ifi.uio.no/~trygver/2007/MVC\\_Originals.pdf](http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf) - 13.01.2009
- [12] Interactive Application Architecture Patterns  
<http://ctrl-shift-b.blogspot.com/2007/08/interactive-application-architecture.html> - 10.01.2009
- [13] IDan North - ntroducing BDD  
<http://dannorth.net/introducing-bdd> - 17.01.2009
- [14] Dan North - Oficjalna strona o BDD  
<http://behaviour-driven.org/> - 10.01.2009

- [15] Dan North - BDD - It's All Behaviour  
<http://behaviour-driven.org/ItsAllBehaviour> - 17.01.2009
- [16] Dan North - BDD - Where's The Business Value  
<http://behaviour-driven.org/WheresTheBusinessValue> - 17.01.2009
- [17] Dan North - BDD - Enough Is Enough  
<http://behaviour-driven.org/EnoughIsEnough> - 17.01.2009
- [18] Dan North - BDD - Stories  
<http://behaviour-driven.org/Story> - 18.01.2009
- [19] Dan North - BDD - Behaviour Driven Programming  
<http://behaviour-driven.org/BehaviourDrivenProgramming>
- [20] Dan North - BDD - Getting The Words Right  
<http://behaviour-driven.org/GettingTheWordsRight> - 17.01.2009
- [21] Dan North - BDD - Cost Of Change  
<http://behaviour-driven.org/CostOfChange> - 17.01.2009
- [22] Dave Astel - Blog  
<http://blog.daveastels.com> - 23.01.2009
- [23] David Chelimsky - Blog  
<http://blog.davidchelimsky.net> - 23.01.2009
- [24] Dave Astel - A New Look at Test Driven Development  
<http://techblog.daveastels.com/2005/07/05/a-new-look-at-test-driven-development> - 23.01.2009
- [25] Dave Astel - BDD Intro  
[http://techblog.daveastels.com/files/BDD\\_Intro.pdf](http://techblog.daveastels.com/files/BDD_Intro.pdf) - 23.01.2009
- [26] Rspec Core Team - Contribute  
<http://rspec.info/community> - 23.01.2009
- [27] github - Rspec Commits History  
<http://github.com/dchelimsky/rspec/commits/master> - 23.01.2009
- [28] Dan North - Introducing rbehave  
<http://dannorth.net/2007/06/introducing-rbehave> - 23.01.2009
- [29] Zach Denis - Blog  
<http://continuousthinking.com> - 23.01.2009

- [30] Aslak Hellesoy - Blog  
<http://blog.aslakhellesoy.com> - 23.01.2009
- [31] Bryan Helmkamp - Blog  
<http://www.brynary.com/>
- [32] The Pragmatic Programmer -The RSpec Book: Behaviour Driven Development with Ruby  
<http://www.pragprog.com/titles/achbd/the-rspec-book> - 23.01.2009
- [33] David Chelimsky - Story Runner in Plain English  
<http://blog.davidchelimsky.net/articles/2007/10/21/story-runner-in-plain-english> - 23.01.2009
- [34] Aslak Hellesoy - Cucumber spoken languages  
<http://github.com/aslakhellesoy/cucumber/blob/master/lib/cucumber/languages.yml>  
- 23.01.2009
- [35] Aslak Hellesoy - Cucumber: Gherkin  
<http://wiki.github.com/aslakhellesoy/cucumber/gherkin> - 23.01.2009
- [36] Oficjalna strona Cucumber  
<http://cukes.info/> - 10.01.2009
- [37] David Chelimsky - Cucumber  
<http://blog.davidchelimsky.net/2008/9/22/cucumber> - 23.01.2009
- [38] Martin Fowler - Will DSLs allow business people to write software rules without involving programmers?  
<http://martinfowler.com/bliki/BusinessReadableDSL.html> - 23.01.2009
- [39] Społeczność ruby - o Języku Ruby  
<http://www.ruby-lang.org/pl/about> - 23.01.2009
- [40] Roy T. Fielding - Strona domowa  
<http://roy.gbiv.com/>
- [41] Roy Thomas Fielding - Architectural Styles and the Design of Network-based Software Architectures - Roy Thomas Fielding  
CHAPTER 5 Representational State Transfer (REST)  
[http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) -  
10.01.2009
- [42] The Apache Software Foundation  
<http://www.apache.org/>

- [43] Royce W.W. - Managing the Development of Large Software Systems  
<http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf> - 10.01.2009
- [44] Boehm B. - A Spiral Model of Software Development and Enhancement  
<http://www.cs.usu.edu/~supratik/CS%205370/r5061.pdf> - 10.01.2009
- [45] NBehave  
<http://code.google.com/p/nbehave/> - 10.01.2009
- [46] NSpec  
<http://nspec.tigris.org/> - 10.01.2009
- [47] NSpecify  
<http://nspecify.sourceforge.net/> - 10.01.2009
- [48] Nunit  
<http://www.nunit.org> - 10.01.2009
- [49] Instinct  
<http://code.google.com/p/instinct/> - 10.01.2009
- [50] JBehave  
<http://jbehave.org/> - 10.01.2009
- [51] JDave  
<http://www.jdave.org/> - 10.01.2009
- [52] JSpec  
<http://wiki.github.com/visionmedia/jspec> - 10.01.2009
- [53] CSpec  
<http://wiki.github.com/arnaudbrejeon/cspec/home> - 10.01.2009
- [54] Specs  
<http://code.google.com/p/specs/> - 10.01.2009
- [55] Fred Brooks - No Silver Bullet - Essence and Accidents of Software Engineering  
<http://www.lips.utexas.edu/ee382c-15005/Readings/Readings1/05-Brook87.pdf> - 24.01.2009
- [56] Tarmo Toikkanen - Don't draw diagrams of wrong practices - or: Why people still believe in the Waterfall model  
<http://tarmo.fi/blog/2005/09/09/dont-draw-diagrams-of-wrong-practices-or-why-people-still-believe-in-the-waterfall-model/> - 24.01.2009

- [57] Ronald E Jeffries - You're NOT gonna need it!  
<http://www.xprogramming.com/Practices/PracNotNeed.html> - 24.01.2009
- [58] Agile Software - How To Implement Scrum In 10 Easy Steps  
<http://www.agile-software-development.com/2007/09/how-to-implement-scrum-in-10-easy-steps.html>
- [59] Agile Web Development - Zbiór pluginów dla Railsów  
<http://agilewebdevelopment.com/plugins> - 24.01.2009
- [60] Jeff Sutherland - What i have learned from first Scrum  
<http://jeffsutherland.com/scrum/FirstScrum2004.pdf>