Spis treści

Streszczenie

Brak ujednoliconego środowiska dokumentowania wymagań użytkownika to jeden z głównych problemów małych i średnich firm, w trakcie fazy analizy. Proces dokumentowania wymagań użytkownika zdaje się być zaniedbanym aspektem zarządzania projektami na rynku nowoczesnych aplikacji internetowych. Istniejące rozwiązania są drogie i często przystosowane do dużych projektów. W niniejszej pracy zaprezentowano narzędzie ułatwiające proces zbierania i przetwazania wymagań użytkownika z wykorzystaniem technologii internetowych, w niewielkich projektach.

W niniejszej pracy zaprezentowano rozwiązanie dla małych i średnich projektów, pozwalające skupić się na najważniejszych akpektach formułowania i dokumentowania wymagań. Powstały system umożliwia wprowadzanie i kategoryzowanie wymagań zarówno w formie graficznej, jak i tekstowej. Na podstawie wprowadzonych danych, aplikacja umożliwia automatyczną generację dokumentu specyfikacji wymagań.

Dzięki zastosowaniu powszechnie dostępnych technologii internetowych, proponowane narzędzie może zostać uruchomione w wielu różnych środowiskach. Wykorzystanie technologii HTML5 oraz języka javascript umożliwiło stworzenie narzędzia wspomagającego graficzne modelowanie przypadków użycia.

Rozdział 1

Wstęp

Rozwój technologii internetowych w ostatnich latach umożliwił programistom tworzenie wysoce interaktywnych narzędzi przy jednoczenym ograniczeniu wymagań systemowych jakie muszą spełnić klienci, chcący skorzystać z wytworzonego oprogramowania. W szczególności, niedawno wprowadzona na rynek, technologia HTML5 daje nowe, szerokie możliwości przeglądarkom internetowym. Możliwości, które były dotychczas zarezerwowane głównie dla technologii Flash (oraz z trochę gorszymi wynikami MS Silverlight, JavaFX, itp.) stają się wspierane natywnie, w każdej popularnej przeglądarce internetowej.

Z wymaganiami użytkownika stykamy się praktycznie od samego początku pracy nad projektem informatycznym. Z tego powodu potrzebne są skuteczne narzędzia gromadzenia i przetwarzania wymagań użytkownika oraz komunikowania ich wszystkim członkom zespołu. Dzięki wyżej wspomnianej technologii HTML5, istnieje możliwość usprawnienia procesu dokumentowania wymagań przy jednoczesnym zachowaniu wysokiej dostępności aplikacji, włączając do niego nowe narzędzia oparte na metodach graficznych.

Rynek nowych technologii roi się dzisiaj od aplikacji rozwiązujących rozmaite problemy, od prostych list zakupów, po zaawansowane narzędzia wspierające zarządzanie projektami. Pomimo istnienia wielu aplikacji skupiających się na procesach fazy analizy, bardzo ciężko znaleźć rozwiązanie dostosowane do potrzeb i realiów prowadzenia projektów w małych i średnich

przedsiębiorstwach.

W niniejszej pracy zostanie zaprezentowana koncepcja aplikacji wspierającej zarządzanie wymaganiami użytkownika, stworzona z myślą o rzeczywistych problemach jakie wiążą się z tym etapem projektu informatycznego. Zaproponowany system ma za zadanie ułatwić zespołom projektowym komunikację i dostęp do wiedzy przy jednoczesnym skupieniu uwagi na najistotniejszych aspektach definiowania i przetwarzania wymagań.

1.1 Cel pracy

W małych i średnich przedsiębiorstwach, proces gromadzenia i dokumentowania wymagań użytkowników jest najczęściej słabo sformalizowany. Wymagania często są przekazywane werbalnie lub trafiają do zespołu z różnych, heterogenicznych źródeł i w wielu, zwykle niekompatybilnych, formatach. Pomimo faktu, iż na rynku nie brakuje oprogramowania wspierającego zarządzanie wymaganiami, istniejące rozwiązania, często nie są przystosowane do realiów prowadzenia projektów w małych, dynamicznych przedsiębiorstwach. Skomplikowane i trudno dostępne systemy często wymagają instalacji oprogramowania po stronie klienta a rozwiązania dostępne online, w formule SaaS (Sofware As A Service) często powielają tylko funkcjonalnosći potężnych (i drogich) systemów takich jak IBM DOORS czy IBM ReqiusitePro. Ponadto niezwykle trudno jest znaleźć rozwiązanie darmowe lub posiadające ogólno dostępną wersję demonstracyjną.

Celem niniejszej pracy jest konstrukcja prototypu systemu usprawniającego zbieranie i przetwarzanie wymagań użytkownika w postaci tekstowej oraz graficznej, dostarczając narzędzi edycji tekstu oraz graficznego modelowania przypadków użycia. W efekcie, użytkownik, po wprowadzeniu początkowych wymagań, ma możliwość automatycznego wygenerowania dokumentu SRS (Software Requirement Specification) [16].

Podejście zaprezentowane w tej pracy, opiera się na założeniach, że nowoczesne oprogramowanie wspierające zarządzanie wymaganiami, powinno m.in.: stawnowić centralne repozytorium wiedzy o wymaganiach; być łatwo dostępne w jak największej ilości różnych środowisk; umożliwiać łatwą ko-

laborację oraz być łatwe w obsłudze, prowokując do kreatywności, zamiast stawiać bariery w postaci skomplikowanych formularzy i tabel oraz niejasnych funkcjonalności.

1.2 Zarys koncepcji

Proces zbierania wymagań jest trudny, ponieważ często jest procesem niesformalizowanym, rozproszonym i udokumentowanym na wiele sposobów (różne nośniki danych, niekomatybilne oprogramowanie). Ponadto wymagania zwykle pochodzą od wielu interesariuszy: od sponsora projektu, po użytkowników końcowych. Niezależnie od poziomu dojrzałości organizacji i stosowanej metodyki zarządzania projektami, źródła pochodzenia wymagań pozostają rozproszone i niekompatybilne.

Na potrzeby ninejszej pracy, został stworzony prototyp systemu pozwalający w łatwy sposób dokumentować gromadzone wymagania użytkownika. W powstałym systemie, głownymi narzędziami definiowania wymagań są pliki tekstowe oraz diagramy przypadków użycia. Osoby odpowiedzialne w projekcie za zarządzanie wymaganiami, otrzymują proste w obsłudze narzędzia, pozwalające na skupienie się nad sednem problemu, który poszczególne wymaganie ma na celu rozwiązać. Zaimplementowano system zarządzania projektem, w którym użytkownik ma możliwość zdefiniowania podstawowych parametrów takich jak nazwa projektu, planowany czas zakończenia oraz ogólny opis, zawierający kluczowe informacje o projekcie na etapie analizy. Korzystając z narzędzia definiowania wymagań w projekcie, użytkownik ma do dyspozycji edytor tekstowy, w którym dokumentuje zidentyfikowane wymagania użytkownika. Dzięki wykorzystaniu prostego języka znaczników (Markdown [17]), użytkownik skupia się na logicznej strukturze opisu wymagania. Brak narzędzi typu „WYSIWYG" znanych ze standardowych edytorów tekstu jest zabiegiem celowym - ma sprawiać, że użytkownik nie jest rozpraszany potrzebą myślenia o graficznej reprezentacji tekstu opisującego problem, przy jednoczesnym zachowaniu czytelnej struktury dokumentu. System załączników i komenatrzy, umożliwia iteracyjną pracę nad wymaganiami przy wykorzystaniu zewnętrzych źródeł informacji.

Dzięki graficznemu edytorowi przypadków użycia standardu UML, użytkownik ma możliwość definiowania kluczowych funkcjonalności systemu i łączenia ich z wybranymi wymaganiami. Tworzone przez użytkowników diagramy, są dostępne do ponownego wykorzystania w innych miejscach w systemie, dzięki czemu wspierana jest kolaboracja i korzystanie z już istniejących rozwiązań. Zarówno opisy wymagań jak i przypisane im przypadki użycia, są integralną częścią dokumentu specyfikacji wymagań użytkownika. Dlatego na każdym etapie fazy analizy istnieje możliwość automatycznego wygenerowania specyfikacji wymagań na podstawie danych wprowadzonych do systemu. Takie podejście sprzyja modelowi iteracyjnemu tworzenia specyfikacji i umożliwia prezentację prac nad dokumentem już na wczesnym etapie projektu.

Koncepcja systemu bazuje na silnym przekonaniu, że brak ograniczeń w postaci skomplikowanych i przytłaczających fukncjonalności pozwala skupić się na procesie twórczym w fazie definiowania wymagań, umożliwiając rozwiązywanie rzeczywistych problemów.

1.3 Struktura pracy

W rozdziale 2 zostanie przedstawiony stan faktyczny związany z zastosowaniem metod pozyskiwania, gromadzenia i zarządznia wymaganiami w małych i średnich przedsiębiorstwach. Rozdział 3 przedstawia dostępne na rynku narzędzia wspierające zarządzanie wymaganiami. Rodział 4 jest dokładnym opisem dziedziny problemu oraz koncepcji zaimplementowanego prototypu. W rozdziale 5 zaprezentowano najistotniejsze technologie, jakie wykorzystano podczas pracy nad prototypem oraz krótkie omówienie środowiska programistycznego w jakim powstał przedmiot pracy. Rozdział 6 stanowi opis zaimplementowanego prototypu. W rozdziale 7 zawarto podsumowanie wyników pracy oraz propozycje kierunków dlaszego rozwoju.

Rozdział 2

Stan faktyczny w inżynierii oprogramowania

W poprzednim rozdziale nakreślono temat przewodni niniejszej pracy. Poruszono problem dostępności i użyteczności istniejących narzędzi wspomagających zarządzanie wymaganiami. Zaprezentowano także autorską koncepcję rozwiązania, mającego na celu usprawnienie procesu tworzenia specyfikacji wymagań użytkownika.

Ten rozdział zadedykowany jest analizie stanu faktycznego w zakresie inżynierii oprogramowania w różnych organizacjach.

2.1 Źródła wiedzy

Inżynieria wymagań stanowi fundament wiedzy pozwalającej na realizację wysokiej jakości projektów informatycznych w założonym budżecie i czasie.

Istnieje obszerna literatura podejmująca tematy stricte związane z inżynierią wymagań (Gause and Weinberg 1989; Jackson 1995; Kovitz 1999; Robertson and Robertson 1999a; Sommerville and Sawyer 1997), jednak praktyczne doświadczenia oraz liczne dostępne badania wskazują na istotne problemy z wprowadzaniem teorii w życie [5, 20, 26].

Znany raport opublikowany przez Standish Group International [26] prezentuje dramatyczną rzeczywistość, gdzie tylko niewielka część projektów

kończy się częściowym lub całkowitym sukcesem. Wyniki raportu wskazują na najważniejsze powody niepowodzeń projektów jak brak włączenia użytkowników w proces budowy oprogramowania, niekompletne i nieudokumentowane wymagania, wiecznie zmieniający się zakres projektów czy brak wsparcia kadry zarządzającej.

W badaniu Nikula et al. (2000) przeprowadzonym wśród małych i średnich przedsiębiorstw w Finlandii, autorzy wskazują na alarmujący brak wiedzy dotyczącej wdrażania procesów inżynierii wymagań. Jednocześnie, wnioski z badania wskazują na konieczność zajęcia się problemami inżynierii wymagań na poziomie korporacyjnym, poprzez wdrażanie, optymalizację i automatyzację procesów IW w małych i średnich przedsiębiorstwach [20].

Z kolei badanie Quispe et al. (2010), również dotyczące małych i średnich przedsiębiorstw, zwraca szczególną uwagę na brak lub nienależytą komunikację z klientem, adresowanie nieodpowiednich problemów klienta, wadliwe specyfikacje oraz stosowanie praktyk inżynierii wymagań „ad hoc" [22].

Powyższe badania zajmują się głównie czynnikami niepowodzeń projektów. W badaniu Fernández et al. (2012) przeanalizowano 12 projektów zakończonych sukcesem. Wnioski z tego badania sugerują, że wiele ze zidentyfikowanych czynników wpływających na niepowodzenia projektów można z powodzeniem rozwiązać zanim staną się realnym zagrożeniem. Ponadto zauważono, że klienci często mają znaczny wpływ na parametry związane z niepowodzeniem projektu.

Analiza wyników powyższych badań w kontekście tej pracy sugeruje przede wszystkim konieczność zwrócenia szczególnej uwagi na poziom dojrzałości organizacji w aspekcie inżynierii wymagań. Nie wszystkie projekty kończą się porażką, a doświadczenia pokazują, że im bardziej dojrzałe procesy inżynierii wymagań w organizacji i odpowiednia „kultura" firmy, tym większe są szanse na realizację projektu zwieńczoną sukcesem.

Podsumowując, żaden system informatyczny ani narzędzie nie rozwiąże wszystkich problemów związanych z inżynierią oprogramowania. Na organizacji wytwarzającej oprogramowanie ciąży odpowiedzialność usprawniania procesów inżynierii wymagań, edukacji pracowników i zasięgania wiedzy ekspertów. Sukcesy w branży oprpgramowania można osiągnąć tylko poprzez

ciągłą naukę, konsekwentną realizację procesów i ich usprawnianie na poziomie korporacyjnym.

Rozdział 3

Istniejące rozwiązania

W tym rozdziale zostanie przeprowadzona analiza rynku oprogramowania wspierającego fazę analizy. Druga część rozdziału zajmie się identyfikacją najistotniejszych problemów jakie posiadają istniejące rozwiązania.

Ilość aplikacji związanych z procesami inżynierii wymagań dostępnych na rynku jest ogromna. Zestawienie INCOSE Requirements Management Tools Survey [14] zawiera podzbiór tych narzędzi, jednak opublikowana lista nie jest od dawna aktualizowana, oraz nie wprowadza żadnego rodzaju kategoryzacji.

3.1 Analiza rynku systemów wsparcia zbierania wymagań

Jak wspomniano we wstępnie do rozdziału, na rynku nie brakuje systemów wsparcia procesu zbierania wymagań. Dostępne rozwiązania oferowane są w zasadzie w trzech różnych modelach. Najpopularniejszą ostatnio architekturą jest Software As A Service, czyli aplikacja internetowa zainstalowana na serwerach twórcy oprogramowania lub w infrastrukturze zewnętrznej firmy hostingowej.

Również klasyczne aplikacje okienkowe, które należy zainstalować na komputerze klienta nadal cieszą się dużą popularnością. Należy jednak zaznaczyć, iż w często są to starsze systemy, nierzadko implementowane jeszcze przed popularyzacją zaawansowanych aplikacji webowych. Niektóre firmy

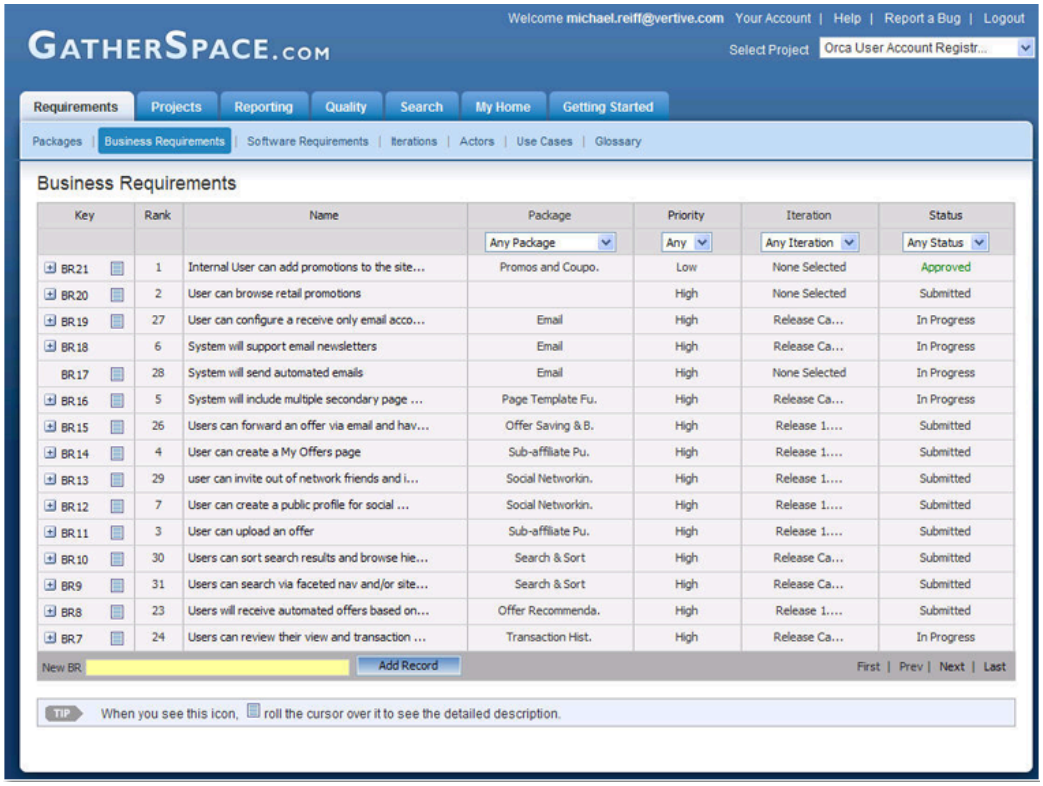
oferują także platformy w architekturze klient-serwer, wymagające istnienia zarówno centralnego serwera - repozytorium w sieci jak i desktopowych aplikacji klienckich. W tej sekcji zostaną przedstawione i porównane wybrane aplikacje z każdej z trzech kategorii.

3.1.1 Model SaaS (Sowtware as a Service)

W ostatnim czasie, wraz z popularyzacją i rozwojem technologii internetowych znacznie wzrosły techniczne możliwości aplikacji dostępnych z poziomu przeglądarki internetowej. Tendencja ta sprzyja powstawaniu licznych aplikacji, adresujących specyficzne problemy, często w obrębie ściśle określonego segmentu rynku. Jednym z przykładów takiego wąskiego segmentu mogą być np. aplikacje do zarządzania projektami online. Wśród popularnych rozwiązań znajdują się takie produkty jak basecamp.com unfuddle.com czy polski nozbe.com. Innym przykładem mogą być aplikacje wspierające proces rekrutacji (humanway.com, recruiterbox.com, jobvite.com). Na uwagę zasługują także aplikacje wspierające rezerwacje hoteli, pensjonatów i niezależnych pokoi i mieszkań jak airbnb.com, booking.com czy b&b.com.

Naturalnie powyższe przykłady dotyczą tylko skrawka możliwych do zagospodarowania segmentów. W rzeczywistości, bardzo ciężko znaleźć wertykalny rynek, który jest niezagospodarowany serwisami, oferującymi różne podejścia do rozwiązania problemów z danej dziedziny.

Powodem takiego stanu rzeczy jest szeroko pojęta popularyzacja internetu oraz sukcesywne przenoszenie się do sieci firm tworzących opgrogramowanie. Dystrybucja oprogramowania w formule SaaS ma wiele zalet w stosunku do klasycznych aplikacji "desktopowych". W szczególności pominięty jest w tym przypadku proces rozprowadzania aplikacji do klientów za pomocą sieci stacjonarnych sklepów z oprogramowaniem, jak miało to miejsce w ciągu ostatniej dekady. Zaimplementowane rozwiązanie, jest gotowe do użycia z chwilą udostępnienia w internecie. Udostępnienie aplikacji na serwerach twórcy oprogramowania lub takich dostarczycieli usług jak Amazon Web Services czy Heroku, daje nieograniczone możliwości wprowadzania nowych funkcjonalności, poprawek i skalowania systemu. Z kolei monitorowanie



Rysunek 3.1: Gatherspace.com - business requirements

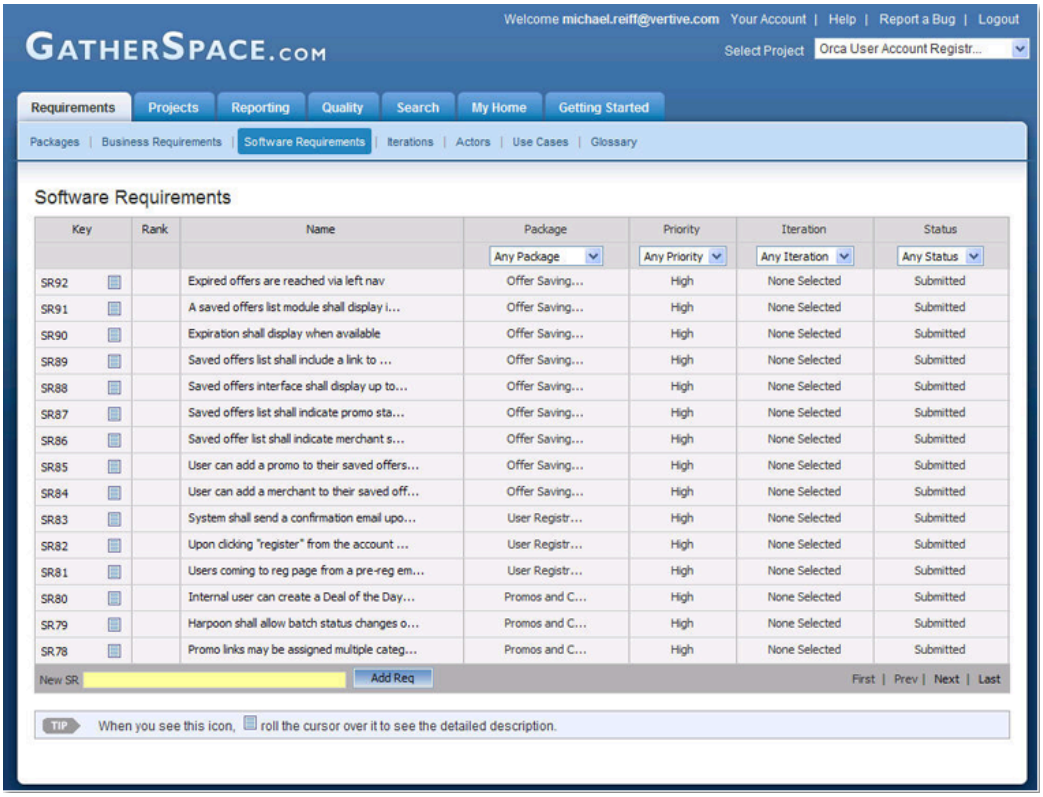
zachowań klientów pozwala niemalże w czasie rzeczywistym odpowiadać na potrzeby użytkowników poprzez modyfikację i wdrażanie nowych funkcjonalności.

W związku z licznymi zaletami modelu SaaS, w internecie powstało wiele aplikacji próbujących zaadresować problemy związane z procesem zbierania i dokumentowania wymagań użytkownika. Do najciekawszych rozwiązań można zaliczyć gatherspace.com, accompa.com, requirementone.com, tracecloud.com czy Jama Contour (http://www.jamasoftware.com/contour/)

Na potrzeby tej pracy zostaną przedstawione funkcjonalności systemów Gatherspace oraz Tracecloud.

Gatherspace

Gatherspace dzieli wymagania na biznesowe (business requirements) oraz funkcjonalne (software requirements). Dostępne są osobne widoki dla obu



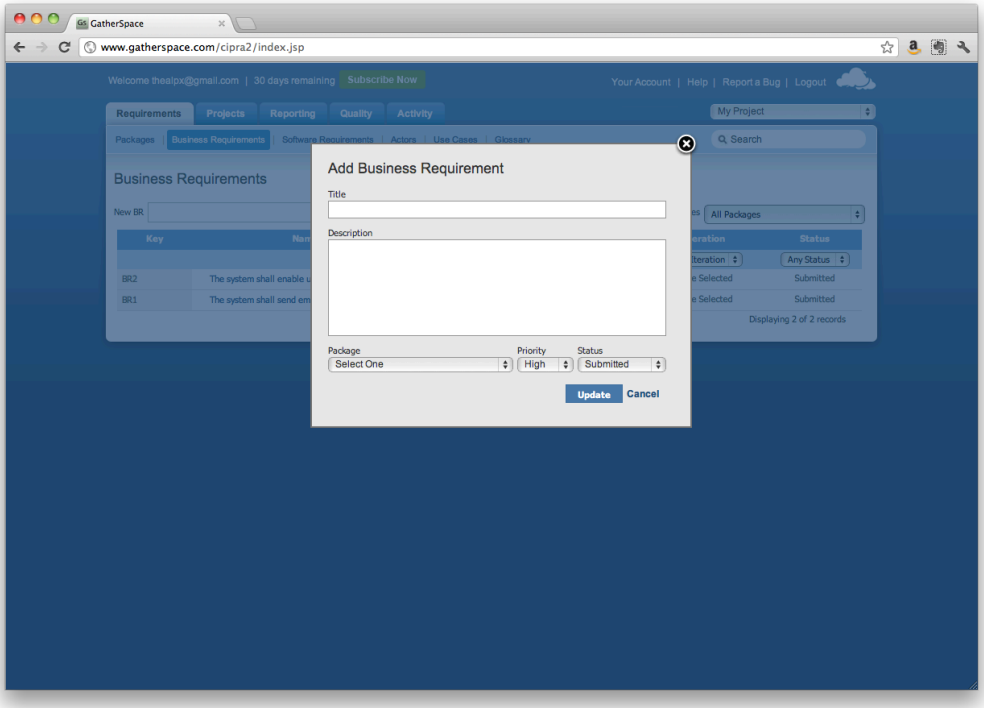
Rysunek 3.2: Gatherspace.com - software requirements

typów wymagań (rys. 3.1, 3.2). Wymagania można katalogować w definiowane przez użytkownika pakiety, podobnie jak pliki w katalogach, w systemach operacyjnych.

Na rysunku 3.2 przedstawiono widok wymagań funkcjonalnych. Dodawanie wymagań przebiega w prosty sposób. Po wpisaniu tytułu wymagania i zatwierdzeniu pola formularza, element zostaje dodany do listy bez przeładowania strony.

Wymaganie może zostać dodane na dwa sposoby - podając jedynie tytuł lub wybierając opcję „Add with detail". Wówczas pojawia się „popup" umożliwiający dodanie zarówno treści jak i opisu wymagania (patrz rys. 3.3).

Wymaganie posiadające opis, zaopatrzone jest w specjalną ikonkę po lewej stronie. Użytkownik wskazujący w to miejsce myszką może podejrzeć treść wymagania, bez konieczności otwierania jego szczegółów. Jest to dość przydatna funkcjonalność, pozwalająca szybko zapoznać się pobieżnie z danym



Rysunek 3.3: Gatherspace.com - dodawanie wymagania ze szczegółami

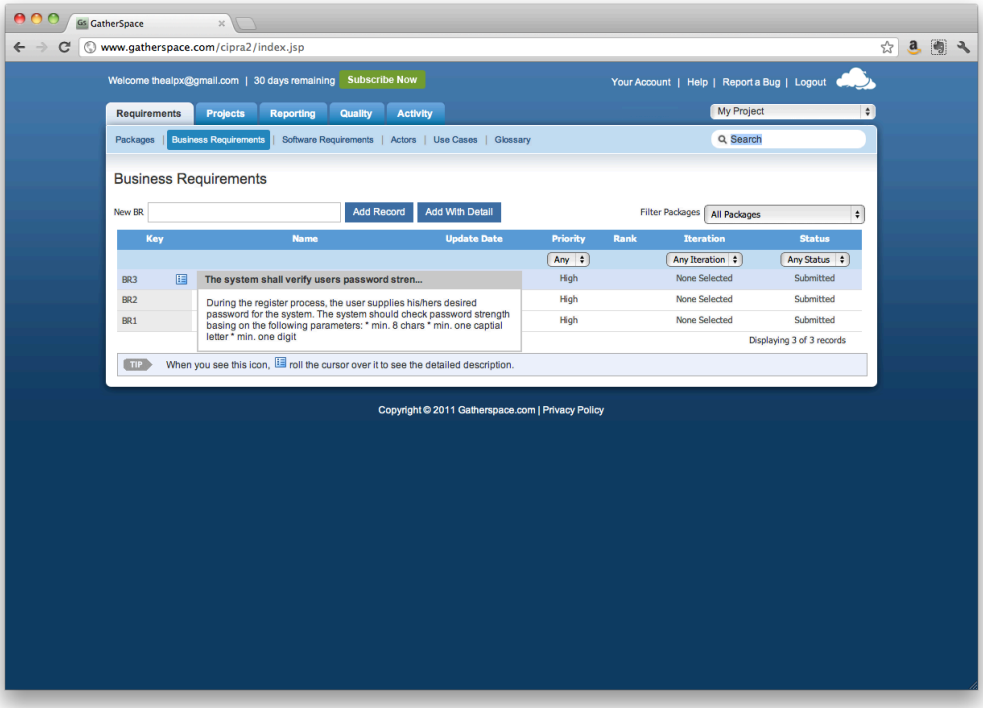
wymaganiem. Rysunek 3.4 zawiera zrzut ekraniu prezentujący tę funkcjonalność.

Widok szczegółów wymagania jest dość prosty i przejrzysty (rys. 3.5). Do dyspozycji użytkownika jest formularz zawierający pola z parametrami wymagania, takimi jak m.in.: priorytet, status, poziom złożoności, pakiet do którego należy, przez kogo wymaganie zostało dodane oraz kto lub co jest źródłem wymagania.

Dodawanie przypadków użycia jest wydzielone do osobnego widoku i funkcjonuje jako osobny zestaw formularzy (rys. 3.6).

Istnieje możliwość łączenia przypadków użycia z wymaganiami poprzez wskazanie konkretnego wymagania w osobnym widoku, jednak system nie umożliwia łączenia tych artefaktów z poziomu edytora graficznego.

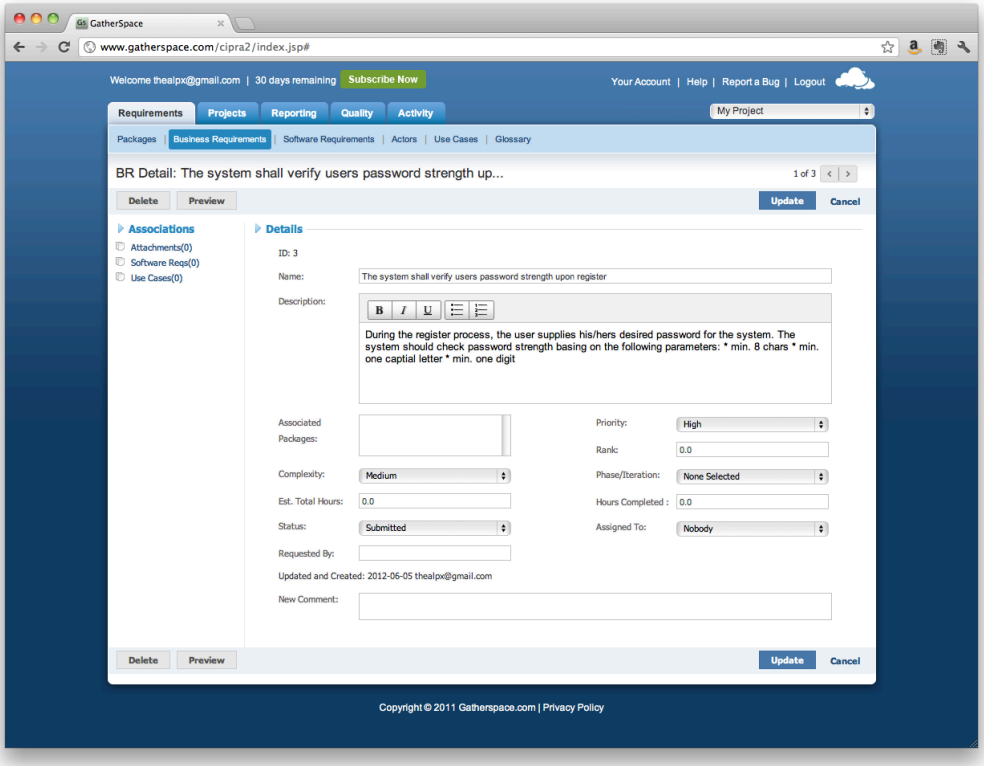
Zarówno przypadki użycia jak i wymagania mogą zostać wyświetlone w formie przyjaznej do druku. Dla przykładu zamieszczono zrzut ekranu obra-



Rysunek 3.4: Gatherspace.com - tooltip ze szczegółami wymagania

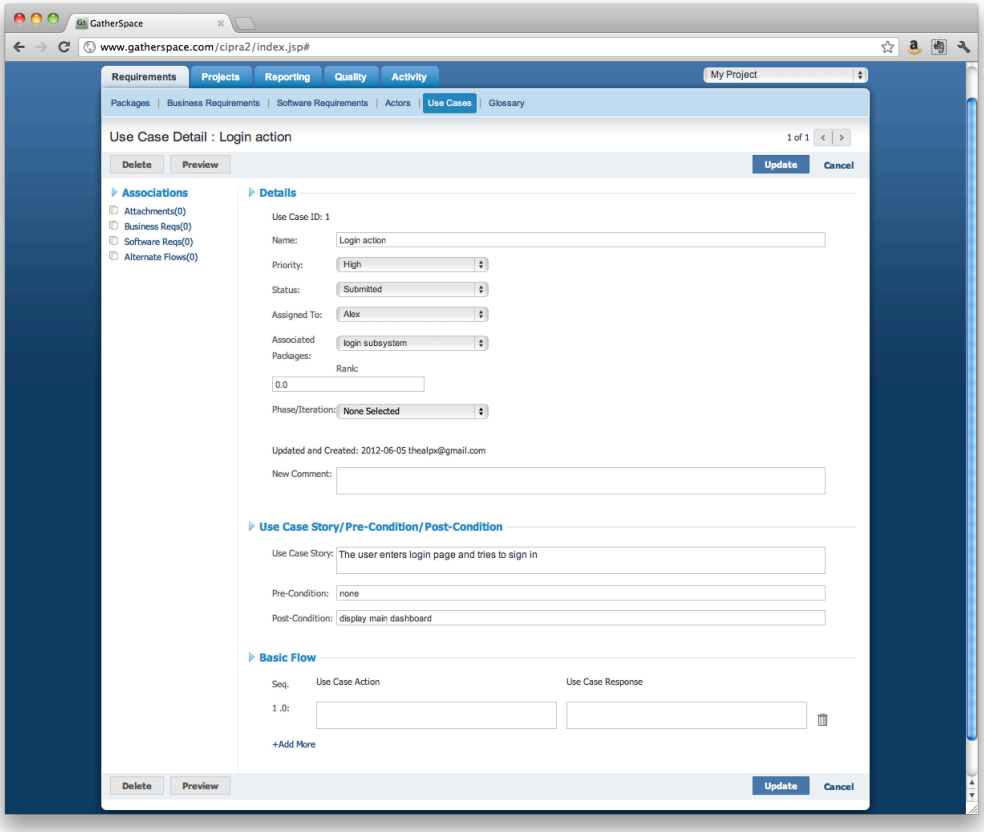
zujący jeden przypadek użycia przygotowany do druku na rys. 3.7

Dostępne w systemie raporty umożliwiają wyświetlenie i wydruk raportów dotyczących wymagań funkcjonalnych i niefunkcjonalnych, raport przypadków użycia oraz raport identyfikowalności (traceability report), umożliwiający przeanalizowanie związków pomiędzy wymaganiami a przypadkami użycia (patrz rysunek 3.8).



Rysunek 3.5: Gatherspace.com - widok szczegółów wymagania

Koszt zakupu licencji na oprogramowanie Gatherspace to minimum 19$ za miesiąc użytkowania. Co ciekawe, istnieje możliwość zakupu wersji oprogramowania, które zostanie zainstalowane w infrastrukturze klienta. Jest to interesujące rozwiązanie dla klientów korporacyjnych, dla których przechowywanie danych na zewnętrznych serwerach jest wykluczone z powodów bezpieczeństwa (np. banki). Jednak koszt zakupu takiej licencji, jak można przeczytać na stronie gatherspace, to 4900$ za instalację systemu. Dodatkowo firma żąda 295$ miesięcznie za obsługę systemu. Biorąc pod uwagę zakres funkcjonalności oraz użyteczność systemu, są to ogromne kwoty, przewyższające, w ocenie autora, wartość dodaną wniesioną do organizacji wraz z wdrożeniem systemu Gatherspace.



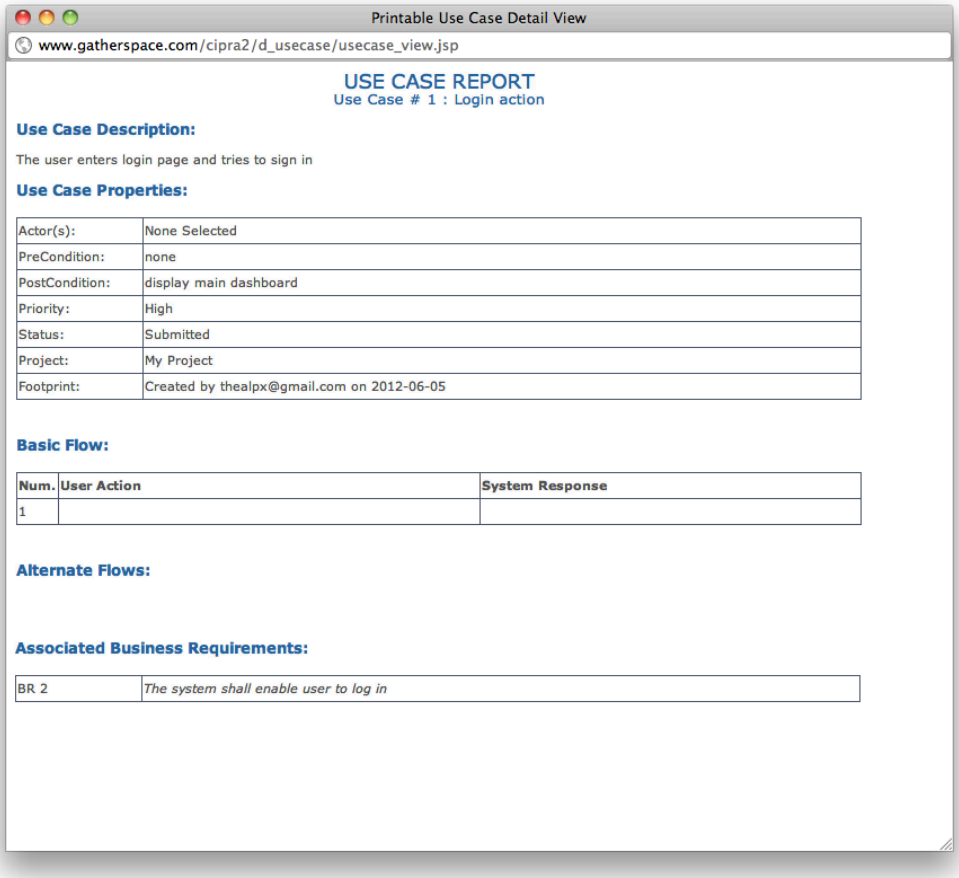
Rysunek 3.6: Gatherspace.com - widok szczegółów wymagania

Tracecloud

Jak widać na rys. 3.9 Tracecloud jest znacznie gorzej przygotowany od strony interfejsu użytkownika. Mimo tego jest znacznie bardziej skompilkowanym systemem, udostępniającym znacznie więcej funkcjonalności niż Gatherspace.

Główny przegląd projektu zawiera tabelę z wyświetlonymi metrykami projektu. Można z niej odczytać ilość zaimplementowanych wymagań, ilościowe wyniki testów, przedląd zgłoszonych błędów i wartości liczbowe wymagań pogrupowane w poszczególne statusy. Po dokładnym przeanalizowaniu tabeli, można odczytać interesujące informacje na temat projektu, jednak użyteczność tego sposobu reprezentacji stanu projektu jest bardzo niska.

Oprócz ilościowego przeglądu stanu projektu, użytkownik ma do dyspo-



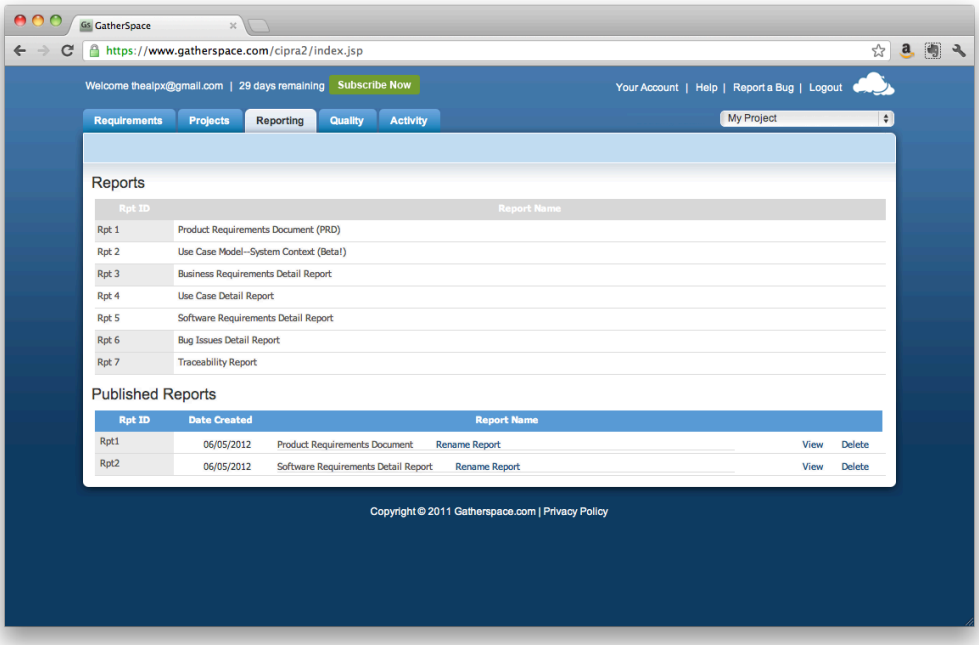
Rysunek 3.7: Gatherspace.com - wymaganie przygotowane do druku

zycji menu i odnośniki do najważniejszych modułów systemu, taki jak wymagania biznesowe, funkcjonalne, scenariusze testowe i błędy.

Przegląd wymagań opiera się o wykres ze statystykami oraz prostą listę (rys. 3.11), z kolei szczegóły wymagania zawierają informację o jego nazwie, priorytecie, właścicielu oraz opis. Dodatkowo twórcy umieścili moduł śledzenia umożliwiający połączenie wymagań biznesowych z funkcjonalnymi (rys. 3.12).

Oprócz licznych raportów wyświetlanych w obrębie aplikacji, nie znaleziono funkcjonalności generowania dokmentu specyfikacji wymagań.

Tracecloud dostępny jest jako wersja Trial przez 6 miesięcy. Koszt wdrożenia waha się od 350$ do 1000$ za miesiąc użytkownania systemu w za-



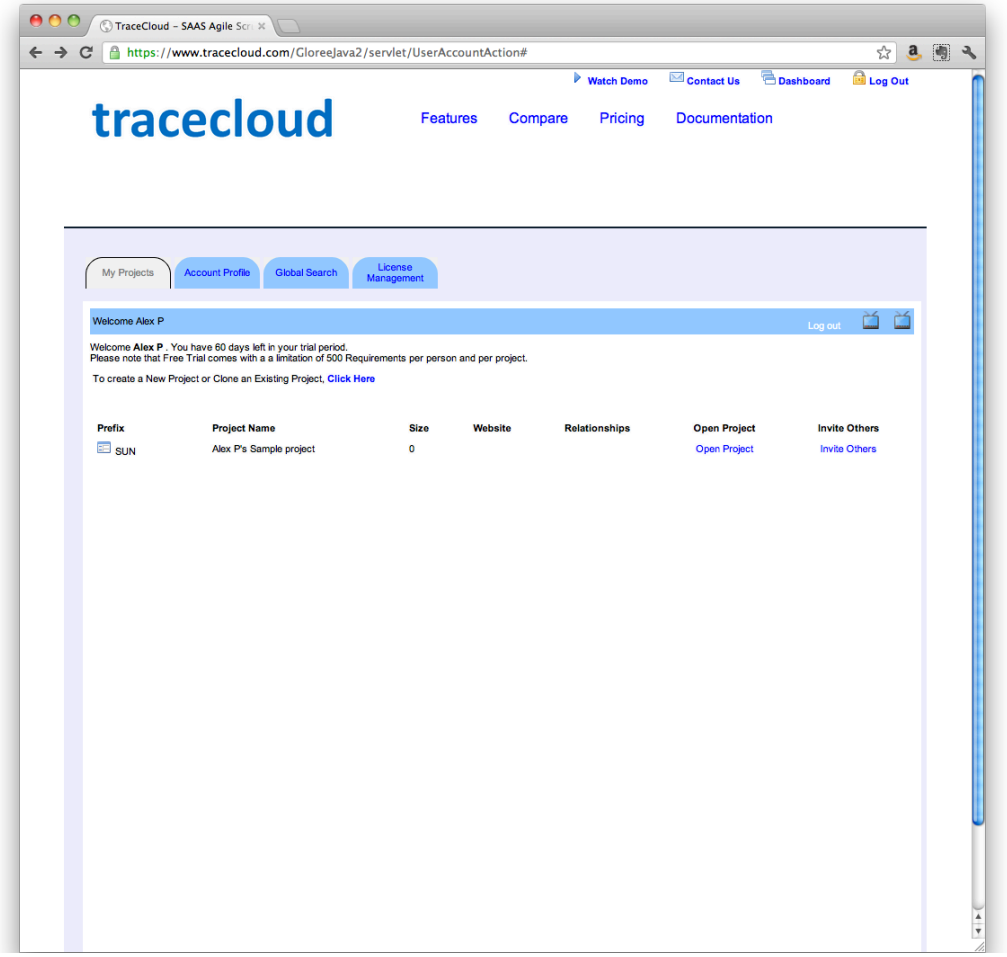
Rysunek 3.8: Gatherspace.com - dostępne raporty

leżności od maksymalnej ilości obsługiwanych wymagań. Najtańsza licencja przewiduje obsługę do 5000 wymagań, podczas gdy wersja najdroższa - do 10tyś.

3.1.2 Aplikacje desktopowe

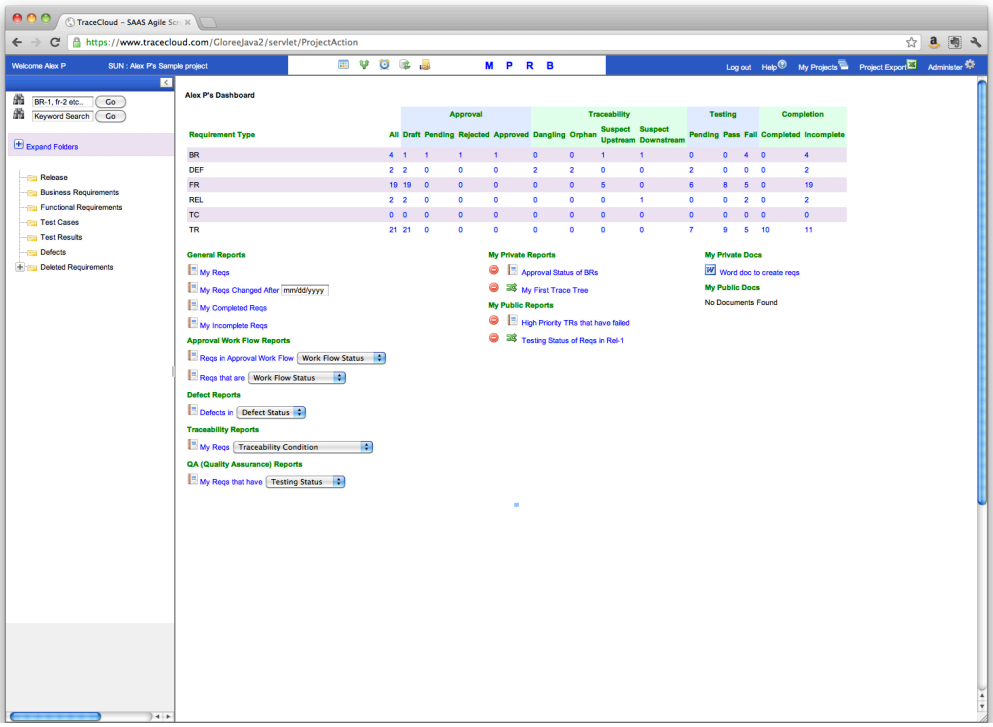
Model SaaS, mimo wszystkich swoich zalet, posiada również wady. Głównym powodem kontrowersji jest konieczność powierzenia danych biznesowych firmie udostępniającej narzędzie na swoich serwerach. Dla dużych korporacji, pilnie strzegących swoich tajemnic handlowych, takie rozwiązanie, może być nie do zaakceptowania ze względów bezpieczeństwa. Mimo potencjalnej możliwośći uniknięcia kosztów budowy i utrzymania infrastruktury sprzętowo-software'owej ryzyko związane z brakiem kontroli nad powierzonymi danymi jest często zbyt wielkie.

W klasycznych aplikacjach okienkowych, odpowiedzialność w zakresie zabezpieczenia danych spoczywa na samej korporacji i jej pracowniku. Dzięki



Rysunek 3.9: Tracecloud - widok listy projektów

temu, nadal istnieje zapotrzebowanie na oprogramowanie instalowane na lokalnym dysku użytkownika. Przykładami takich aplikacji są m.in. IBM DOORS oraz IBM Rational Requirements Composer.



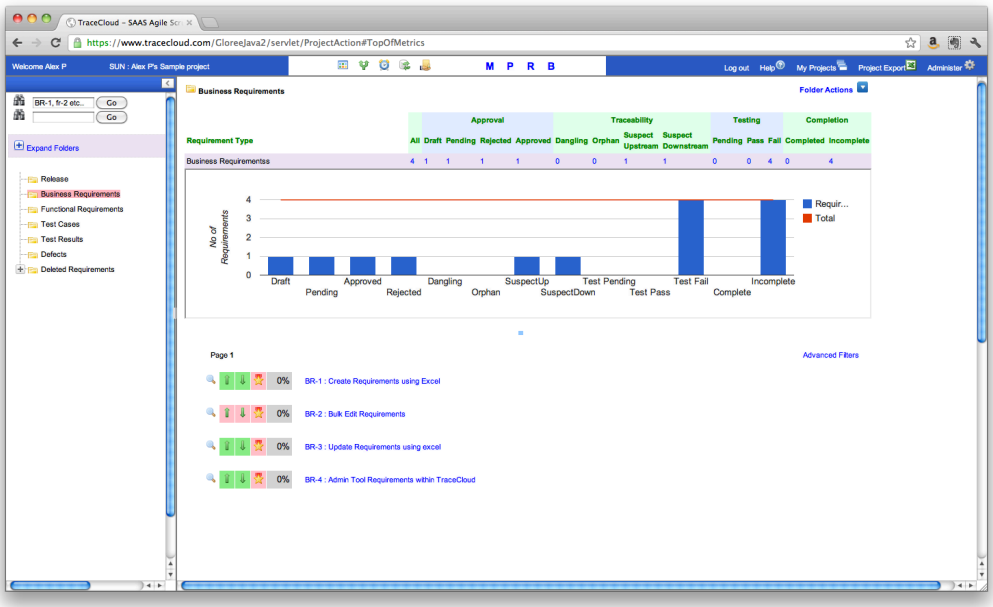
Rysunek 3.10: Tracecloud - ekran główny

RequisitePRO

DOORS

3.1.3 Platforma IBM .jazz

Platforma IBM jazz zasługuje na osobną sekcję, ponieważ jest zintegrowanym, kompleksowym zestawem narzędzi i aplikacji dla przedsiębiorstw tworzących oprogramowanie. Centrum zarządzania platformą jazz jest serwer stanowiący repozytorium danych i ośrodek dowodzenia. Platforma składa się zarówno udostępnionych na serwerze usług sieciowych oraz aplikacji instalowanych lokalnie, łączących się ze zdalnym serwerem (tzw. rich-client applications).

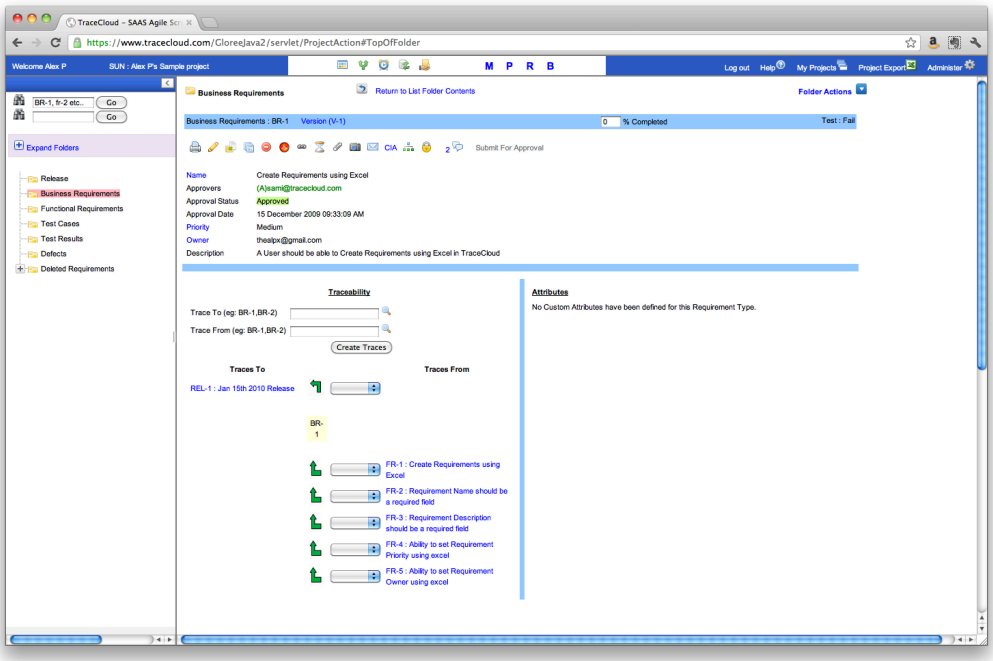


Rysunek 3.11: Tracecloud - przegląd wymagań

3.2 Problemy z istniejącymi rozwiązaniami

Analizując istniejące rozwiązania, nie można oprzeć się wrażeniu, że wszystkie przystosowane są do dużych, korporacyjnych projektów. Większość aplikacji powiela rozwiązania znane i sprawdzone funkcjonalności. W konsekwencji trudnym zadaniem jest znalezienie wyróżniających się elementów wśród oferowanych możliwości tych produktów. Wyróżniają się szczególnie ciężkie aplikacje desktopowe od firmy IBM, jednak są one przystosowane do bardzo dużych projektów. Małe i średnie przedsiębiorstwa potrzebują prostszych i „lżejszych rozwiązań".

Proces definiowania wymagań w niewielkich projektach jest w dużej mierze procesem twórczym. Często pojawienie się nowego wymagania jest inicjowane z kilku heterogenicznych źródeł. Nierzadko zdarza się również, że to samo wymaganie jest różnie komunikowane przez wiele źródeł. W niewielkich projektach, gdzie często jedna osoba, lub mały zespół odpowiedzialny jest za specyfikację wymagań, wymagania przybierają postać wiadomości email, rozmów telefonicznych, notatek ze spotkań i formalnych dokumentów. Za-



Rysunek 3.12: Tracecloud - szczegóły wymagania

daniem osoby lub zespołu odpowiedzialnego za specyfikację wymagań, jest przefiltrowanie cząstkowych informacji oraz ekstrakcja i udokumentowanie wymagań. Żadne narzędzie (przynajmniej na razie) nie zastąpi skutecznie pracy człowieka nad wyłuskaniem wszystkich wymagań.

Rozdział 4

Podstawy teoretyczne i omówienie koncepcji proponowanego rozwiązania

W poprzednim rozdziale opisano przykładowe narzędzia funkcjonujące na rynku oprogramowania do zarządzania wymaganiami. Opisano ich główne wady i zalety oraz przedstawiono najważniejsze problemy związane ze stosowaniem ich w praktyce.

W tym rozdziale, zostaną opisane procesy inżynierii wymagań od strony teoretycznej. Zostanie również zwrócona uwaga na niezbędne elementy nowoczesnego systemu wspierającego pracę z wymaganiami w małych i średnich przedsiębiorstwach. Na koniec rozdziału, przedstawiona zostanie koncepcja nowego narzędzia, będącego przedmiotem tej pracy.

4.1 Inżynieria wymagań

Inżynieria wymagań (IW) jest dziedziną, na którą składają się wszelkie działania związane z odkrywaniem (elicitation), analizą (analysis), weryfikacją (validation) oraz zarządzaniem wymaganiami [25]. Dotyczy ona każdego systemu informatycznego, niezależnie od jego rozmiarów i jest jednocześnie jednym z najistotniejszych procesów w cyklu życia projektu.

IW dostarcza metodologii i narzędzi, służących do precyzyjnego określenia celu i zakresu projektu, poprzez identyfikację interesariuszy i ich potrzeb oraz udokumentowanie wyników w formie umożliwiającej ich analizę, komunikację oraz finalnie - implementację [2].

Zarys historyczny

Kryzys oprogramowania na przełomie lat '60 i '70 XX wieku był katalizatorem do podjęcia prób usystematyzowania młodej wówczas dziedziny tworzenia systemów informatycznych. Był to okres popularyzjacji sprzętu komputerowego, powszechnego, dramatycznie rosnącego zapotrzebowania na programistów oraz nagłośnionych, wielkich porażek systemów informatycznych, jak katastrofa rakiety kosmicznej Mariner 1 [7]. Niedobór wykwalifikowanych programistów, zmusił korporacje do intensywnych poszukiwań nowych rozwiązań. Literatura branżowa szybko przepełniła się głosami nawołującymi do poprawy metod zarządzania w projektach informatycznych. Pojawiły się takie tytuły jak "Controlling Computer Programming", ; "New Power for Management'"; 'Managing the Programming Effort"; "The Management of Computer Programming Efforts". W roku 1968, w drodze ustaleń na konferencji NATO Software Engineering (tu po raz pierwszy w branży zaczęto mówić o „kryzysie oprogramowania"), "zarządzanie oprogramowaniem" stało się fundamentem i jednym z głównych dyskursów w dziedzinie inżynierii oprogramowania [11].

Pionierzy inżynierii oprogramowania podejmowali próby zaczerpnięcia wiedzy z zakresu inżynierii systemów i wykorzystania jej w projektach informatycznych. Ten kierunek rozwoju opierał się na paradygmacie klasycznej inżynierii, zakładającym budowę systemu zgodną z określonym procesem: od identyfikacji problemu, opracowaniu specyfikacji, konstrukcji systemu i jego utrzymania. Takie sekwencyjne podejście do tworzenia systemów informatycznych było fundamentem powstania pierwszych procesów wytwórczych oprogramowania.

Klasycznym podejściem był przedstawiony w pracy Winstona Royce'a

(1970), model kaskadowy (sekwencyjny model liniowy)[1]. W swojej pracy, Royce proponując rozwiązanie sekwencyjne (w którym żaden kolejny etap nie mógł zostać rozpoczęty przed zakończeniem etapu poprzedniego) zastrzegł, że „wierzy w zaproponowaną koncepcję, jednak jej implementacja jest ryzykowna i naraża projekty na niepowodzenie" [27]. Zatem już na samym początku swojego istnienia, model kaskadowy wraz z analogią inżynierii oprogramowania do inżynierii klasycznej, był uważany w branży za nieprzystosowany do dziedziny problemu. Mimo tego, model kaskadowy, z powodzeniem, znalazł zastosowanie w dużych projektach m. in. rządowych, wojskowych i kosmicznych, gdzie zarówno klient jak i wykonawcy bardzo dobrze rozumieli wymagania systemu [19]. W kolejnych latach pojawiały się modyfikacje modelu kaskadowego, a rola podejścia iteracyjnego przybierała na znaczeniu. W roku 1986 Barry Boehm opisał model spiralny, będący połączeniem ustrukturalizowanego procesu kaskadowego z rozwojem przyrostowym, opartym na prototypowaniu [4]. Kolejnym kamieniem milowym w dziedzinie procesów wytwórczych oprogramowania było wydanie książki Kent'a Beck'a „Extreme Programming Explained" [3] w roku 1999 oraz publikacja „Agile Manifesto" w roku 2001, gdzie po raz pierwszy, formalnie zaproponowano termin „zwinnego programowania" [15].

Podobieństwa w procesach wytwórczych

Pomimo znaczących różnic w dostępnych i udokumentowanych metodykach projektowych, dla wszystkich istnieje część wspólna, w postaci nieuniknionej fazy analizy na etapie rozpoczęcia projektu. Nie ma możliwości rozpoczęcia modelowania ani implementacji systemu, bez wcześniejszej wiedzy w zakresie jego przeznaczenia i oczekiwanych funkcji. Zarówno w metodykach zwinnych, jak i w tradycyjnych procesach wytwórczych, w początkowej fazie projektu, kluczową rolę gra przygotowanie systemu od strony wymagań. Zatem inżynieria wymagań dotyczy każdego procesu wytwórczego.

[1]Royce podał w swoim modelu wariant liniowego procesu, gdzie kolejne fazy były zwrotnie sprzężone z fazami poprzednimi [27], jednak w większości organizacji stosujących model kaskadowy, jest to ściśle sekwencyjny proces liniowy [21].

Procesy w inżynierii wymagań

Jak podaje Sommerville [25], do dziedziny inżynierii wymagań możemy zaliczyć następujące pod-procesy:

- studium wykonalności
- gromadzenie i analiza wymagań
- weryfikacja wymagań
- zarządzanie wymaganiami

Studium wykonalności (feasibility study)

Wg. Sommerville'a (2006), przy dużych projektach, w początkowym stadium powinno zostać przeprowadzone studium wykonalności. Podstawowym celem jego przeprowadzania jest udzielenie odpowiedzi na pytanie, czy należy kontynuować projekt. W trakcie tego etapu zwiększa się wiedza o dziedzinie problemu, a koncepcja systemu jest weryfikowana pod kątem celowości zastosowania w organizacji, wszelkich ograniczeń systemowych, a także sprzętowych. W modelu Rational Unified Process, studium wykonalności powinno zostać włączone do fazy rozpoczęcia projektu (inception phase) [18].

W niektórych przypadkach (szczególnie w dużych projektach krytycznych) studium wykonalności może zostać rozbudowane do pełnej analizy oceny ryzyka. Wówczas powinny zostać uwzględnione aspekty nietechniczne i administracyjne, np.: czy projekt ma przydzielony wystarczający budżet, jakie aspekty polityczne należy wziąć pod uwagę w trakcie definiowania wymagań, czy projekt może być wykonany w przyjętych ramach czasowych.

Gromadzenie i analiza wymagań

Gromadzenie wymagań jest procesem, na który składają się działania zmierzające do zgłębienia celu i motywacji przyświecających budowie analizowanego systemu. W szczególności etap ten wiąże się z identyfikacją wszystkich wymagań, jakie musi spełnić system, aby osiągnąć sukces.

Proces gromadzenia wymagań w języku angielskim nosi nazwę „requirements elicitaion". „Elicitaion" w dosłownym tlumaczeniu, oznacza „wywołanie", „wydobycie", „ujawnienie". W szczególności termin „wydobycie" lepiej oddaje naturę tych działań, niż popularne w języku polskim „gromadzenie" czy „definiowanie" wymagań. Innymi słowy jest to proces komunikacji analityków z użytkownikami w celu zdobycia jak największej ilości informacji o budowanym systemie.

Głównymi technikami związanymi z pozyskiwaniem wymagań są: identyfikacja interesariuszy, gromadzenie faktów i informacji przy pomocy wywiadów, kwestionariuszy, list kontrolnych, warsztatów kreatywnych, burzy mózgów, map myśli (mindmapping), modelowania i prototypowania. Znaczna część wymagań stanowi wiedzę zdobytą w procesie przeprowadzania wywiadów z interesariuszami projektu. W związku w powyższym, można wprowadzić uniwersalne uogólnienie, stwierdzając, że głównym narzędziem wykorzystywanym w procesie gromadzenia wymagań jest szeroko rozumiana komunikacja. Wywiady przeprowadzane z interesariuszami mogą mieć formalny lub nieformalny przebieg, a stosowane w tym procesie narzędzia w dużej mierze zależą od poziomu skomplikowania projektu, wypracowanych metod, dostępnych narzędzi i preferencji samych analityków.

Wymagania zwykle pochodzą z wielu heterogenicznych źródeł. Najwyższy priorytet zawsze powinien przysługiwać wymaganiom pochodzącym od klienta i użytkowników systemu. Jednak wpływ na specyfikację systemu mogą mieć również źródła zewnętrzne, takie jak niezależni eksperci z dziedzin obejmowanych przez projekt, uwarunkowania prawne i ekonomiczne, czy ogólnie przyjęte standardy i procedury związane ze szczególnymi aspektami tworzonego systemu.

Dokumentem stanowiącym rezultat pracy nad gromadzeniem wymagań jest specyfikacja wymagań systemu. Standardem określającym proces tworzenia i strukturę tego dokumentu jest IEEE Guide for Developing System Requirements Specifications [10]. Wiele organizacji definiuje również własne standardy i procesy, lepiej odpowiadające specyfice realizowanych projektów.

Do procesu gromadzenia wymagań i narzędzi z nim związanych, autor powróci przy okazji opisu koncepcji proponowanego rozwiązania, w następnym

rozdziale.

Weryfikacja wymagań

Celem procesu weryfikacji wymagań jest dowiedzenie, że koncepcja budowanego systemu odpowiada rzeczywistym potrzebom użytkowników. Jest to kluczowy proces inżynierii wymagań, ponieważ poprawa zidentyfikowanych na tym etapie problemów jest o rząd wielkości mniej kosztowna, niż późniejsze zmiany systemowe. Zdecydowanie łatwiej jest bowiem poprawiać projekt architektury opgrogramowania niż wprowadzać modyfikacje koncepcyjne w istniejącym już systemie.

Weryfikacja wymagań powinna być przeprowadzana na podstawie dokumentu specyfikacji wymagań systemu. Działania podejmowane w trakcie weryfikacji wymagań powinny uwzględniać przede wszystkim zasadność, bezkonfliktowość, kompletność, implementowalność i weryfikowalność wszystkich wymagań. Zasadność pozwala odpowiedzieć na pytanie, czy dane wymaganie rozwiązuje realny problem w systemie, potwierdzając tym samym celowość swojego istnienia. Ponieważ w trakcie gromadzenia wymagań przeprowadzane są wywiady z wieloma interesariuszami projektu, mającymi różne, często sprzeczne wymagania, konieczna jest weryfikacja spójności specyfikacji pod względem konfliktów. Zapewnienie bezkonfliktowości polega na takim sfomułowaniu wymagań w specyfikacji, aby żadne nie były ze sobą w sprzeczności. Kompletność jest weryfikacją dokumentu wymagań z perspektywy realizacji wszystkich porządanych funkcji. Wszystkie zdefiniowane wymagania muszą być realizowalne w określonym czasie, przy użyciu określonego sprzętu i opgrogramowania. W trakcie badania implementowalności, analityk skupia się na odpowiedzi na pytanie czy dane wymaganie jest w pełni realizowalne w rzeczywistych warunkach. Z kolei weryfikowalność ma na celu zapewnienie narzędzi, umożliwiających sprawdzenie i demonstrację poprawności działania zrealizowanego wymagania za pomocą metod empirycznych.

Istnieje wiele narzędzi i metod, jakie można wykorzystać w procesie weryfikacji wymagań. Wyniki badania Boehem et al „Prototyping vs. Specifying" [6] dowodzą, że jedną ze skuteczniejszych metod prewencji przed definiowa-

niem błędnych wymagań jest metoda prototypowania. Badanie to polegało na podziale grupy studentów inżynierii oprogramowania na dwa zespoły. Celem obu zespołów była implementacja systemu o określonym zakresie. Jedna grupa korzystała z metody prototypowania. Ich konkurenci, natomiast, korzystali z podejścia zorientowanego na specyfikację. W rezultacie, grupa stosująca metodę prototypowania osiągnęła lepsze wyniki, w szczególności, w zakresie użyteczności interfejsu użytkownika i prostoty obsługi dostarczonego systemu. W przeciwieństwie do podejścia opartego na samej specyfikacji, prototypowanie umożliwia wizualizację wymagań systemu, uniezależniając interpretację treści wymagania od wyobraźni odbiorcy.

Jednak, jak każda metoda, prototypowanie nie jest wolne od wad. Istnieje ryzyko, że prototyp, kładąc zbyt duży nacisk na pewne szczegóły, sprawi, że ogólna koncepcja systemu przestanie być wyraźnie dostrzegalna. Ponadto, wyzwaniem w przypadku prototypowania jest odpowiednia komunikacja z klientem. Klient musi dokładnie rozumieć specyfikę i cel tworzenia prototypu. Istnieje bowiem pokusa, aby pod naciskami klienta wykorzystać prototyp jako finalny produkt. W związku z powyższym, wśród wszystkich członków zespołu, niezbędna jest pełna świadomość faktu, iż prototyp, po spełnieniu swojej funkcji zostanie odstawiony, a doświadczenia z jego budowy posłużą jako informacje wejściowe do implementacji finalnego rozwiązania. Powstało wiele prac dotyczących prototypowania, jednak szczegółowe aspekty tego podejścia wykraczają poza zakres tej pracy. Osoby zainteresowane metodami prototypowania znajdą kilka ciekawych pozycji w bibliografii [1, 8].

Zarządzanie wymaganiami

Specyfika większości projektów informatycznych, wiąże się z tym, że raz udokumentowane wymagania rzadko pozostają aktualne do momentu zakończenia prac. Wymagania są przedmiotem ciągłych zmian i niezbędne są narzędzia, których zadaniem będzie zarządzanie tymi zmianami. Proces zarządzania wymaganiami należy rozpocząć w momencie powstania pierwszej wersji specyfikacji wymagań i kontynuować aż do zakończenia projektu.

4.2 Motywacja

Mając na uwadze teoretyczne tło dziedziny jaką jest inżynieria wymagań oraz praktyczne problemy z jakimi borykają się dzisiaj małe i średnie przedsiębiorstwa w zakresie zarządzania wymaganiami, powstała koncepcja rozwiązania, adresującego najpilniejsze problemy w rzeczywistych projektach małej i średniej skali.

Jak wykazano w rozdziale 2, pomimo rozwoju technologii oraz świadomości firm w zakresie inżynierii oprogramowania, bardzo duża część małych i średnich przedsiębiorstw nadal ignoruje zagadnienia z zakresu inżynierii wymagań. Dojrzałość tych organizacji pod kątem inżynierii oprogramowania jest bardzo niska. Niewielką część harmonogramów projektów przeznacza się na analizę wymagań, a narzędzia stosowane w trakcie tego procesu najczęściej ograniczają się do pakietów biurowych, klientów poczty elektronicznej i odręcznych notatek. Mikro-firmy osiągające na tym polu sukcesy najczęściej charakteryzują się posiadaniem bardzo małych zespołów ekspertów z ogromym doświadczeniem w branży, posiadającymi techniki wypracowane metodami prób i błędów, nad którymi pracowali przez wiele lat w swojej karierze. Jednak zdecydowana większość firm nadal eksperymentuje z autorskimi podejściami, często integrując różne istniejące narzędzia.

Część winy za taki stan rzeczy leży po stronie twórców oprogramowania. Istnieje niewiele systemów umożliwiających skuteczne zarządzanie wymaganiami w środowsiku dynamicznych, zorientowanych na rynek internetowy, projektów, jakie najczęściej realizują firmy z sektora MŚP. Kolejną ważną obserwacją jest fakt, iż bardzo często mamy do czynienia z projektami, których wymagania wymyślane są przez samych twórców, przez co proces dokumentowania wymagań i tworzenia specyfikacji przebiega ad hoc lub nie istnieje w ogóle. Dodatkowo, brak umiejętności zarządzania zmianą, powoduje, że projekty oddawane są po terminie, z przekroczonym budżetem, często niedostarczając klientowi porządanego efektu. Jest to problem całej branży - zarówno kadry zarządzającej, analityków, programistów, klientów i użytkowników. Problemy z zarządzaniem wymaganiami, powodują, spowolnienie innowacji. Gdyby firmy zaczęły umiejętnie zarządzać wymaganiami i zmianą,

więcej projektów kończyłoby się sukcesem, co pozytywnie wpłynęłoby na prędkość rozwoju oprogramowania. Ewidentnie istnieje niezagospodarowana luka na rynku w tym segmencie.

Małe i średnie przedsiębiorstwa są motorem napędowym dzisiejszej gospodarki opartej w znacznym stopniu o techniki informacyjne. Obowiązkiem inżynierii oprogramowania jest stałe kwestionowanie istniejących rozwiązań i katalizowanie zmian w kierunku usprawniania procesów związanych z wytwarzaniem oprogramowania. Jednak inżynieria wymagań zdaje się być niedocenianym i nadal słabo rozumianym zagadanieniem związanym z wytwarzaniem oprogramowania. Jest to jednak kluczowa, przyszłościowa dziedzina - w miarę postępu technologicznego, zapotrzebowanie na oprogramowanie nadal będzie wzrastało.

4.3 Opis proponowanego rozwiązania

Proces pozyskiwania i przetwarzania wymagań powinien być przede wszystkim łatwy w implementacji dla organizacji oraz dający dużą dozę elastyczności. W rozdziale 'Problemy z istniejącymi rozwiązaniami' opisano stan sztuki i zaadresowano niedociągnięcia dzisiejszych narzędzi.

Motywem przewodnim proponowanego narzędzia jest przystępność i prostota obsługi oraz wniesienie rzeczywistej wartości dodanej poprzez dostarczenie funkcji, mających realne zastosowanie w praktyce.

W trakcie prac nad koncepcją narzędzia postawiono następujące cele:

- Silna koncentracja na praktycznym zastosowaniu systemu wsparcia zarządzania wymaganiami - system musi rozwiązywać rzeczywiste problemy zespołów projektowych w zakresie gromadzenia i przetwarzania wymagań. Wdrożenie systemu musi wiązać się z wniesieniem wartości dodanej do procesów inżynierii wymagań w organizacji;

- Obsługa wielu projektów w obrębie aplikacji - system musi umożliwiać obsługę i zarządzanie wieloma projekami w obrębie tej samej aplikacji;

- Ogólny przegląd stanu projektu - najistotniejsze informacje z perspektywy wymagań systemu powinny być dostępne na jednej stronie podsumowującej projekt;
- Wsparcie gromadzenia wymagań - system powinien wspierać pozyskiwanie, definiowanie, katalogowanie (grupowanie) nieprzetworzonych wymagań oraz nadawanie im odpowiednich priorytetów przy pomocy intuicyjnych narzędzi;
- Uniwersalność formatu przechowywanych danych - system powinien umożliwiać łatwe przetwarzanie treści wymagań przez zewnętrzne systemy;
- Dostarczenie rozwiązań zarządzania zmianą - aplikacja powinna dostarczać wsparcia zarządzania wymaganiami na późniejszych etapach projektu;
- Stworzenie centralnego repozytorium wiedzy na temat wymagań - system powinien stanowić główne źródło wiedzy i komunikacji w zakresie wymagań;
- Wsparcie kolaboracji - system powinien umożliwiać współpracę nad wymaganiami poprzez obsługę wielu użytkowników i dostarczenie im modułu komentarzy;
- Automatyzacja tworzenia specyfikacji - należy zaimplementować algorytm generujący dokument specyfikacji wymagań na podstawie informacji dostępnych w systemie;

W efekcie powstał prototyp systemu, otwierający drogę do dyskusji i rozwoju w kierunku poprawienia procesów inżynierii wymagań w małych i średnich przedsiębiorstwach, skupiający się na praktycznych rozwiązaniach.

4.4 System Reqmanager - opis funkcjonalności

Poniżej opisano poszczególne funkcjonalności systemu i przedstawiono mechanizmy, narzędzia oraz techniczne rowiązania które posłużyły do imple-

mentacji poszczególnych założeń systemu.

4.4.1 Obsługa wielu projektów

System umożliwia tworzenie wielu projektów. Formularz dodawania projektu składa się pól nazwy, krótkiego opisu oraz głównej notatki projektu, będącej jego szczegółowym opisem w formie tzw. briefu. Ostatnią niezbędną informacją, którą należy podać jest planowana data zakończenia prac nad projektem. W założeniu opis i notatka projektu ulegają zmianom w miarę upływu czasu, dlatego na późniejszych etapach projetku istnieje możliwość bardzo łatwej edycji zmian treści tych atrybutów.

4.4.2 Ogólny przegląd stanu projektu

Ekranem głównym w obrębie wybranego projektu jest przegląd stanu projektu z perspektywy wymagań. Użytkownik ma możliwość natychmiastowej edycji głównej notatki opsiującej założenia całego projektu oraz jego krótki opis. Ponadto, dostępna jest lista wymagań zawierająca podstawowe informacje takie jak kod wymagania, jego opis, priorytet oraz ilość przypisanych przypadków użycia. Wszystkie te elementy umieszczone na jednej stronie, umożliwiają użytkownikowi ogólną orientację w zakresie i celach projektu pod kątem wymagań.

4.4.3 Wsparcie gromadzenia wymagań

Praktyczne narzędzie do zastosowania w procesie gromadzenia wymagań musi wspierać najwcześniejszą fazę - powstawania wymagań. W dynamicznych projektach, wymagania pojawiają się niespodziewanie, w różych sytuacjach i pochodzą z wielu różnych źródeł. Narzędzie musi wspierać prostą w obsłudze opcję dodawania wymagań w postaci krótkich notek, dłuższych artykułów lub opisów całych procesów. Jednocześnie nie może obarczać użytkownika obowiązkiem nadawania identyfikatorów oraz innych metainformacji. Powinno również ograniczać potrzebę formatowania treści. Ce-

lem użytkownika, jest spisanie jak najszybciej wszystkich obserwacji i zachowanie ich w formie możliwej do późniejszego przetworzenia.

Proponowane rozwiązanie wspiera ten proces dzięki zintegrowanemu edytorowi tekstu, edytorowi diagramów przypadków użycia oraz systemowi załączników.

Edytor tekstowy

Formatowanie tekstu w języku znaczników Markdown [17] umożliwia nadanie dokumentowi logicznej struktury wykorzystując zwykły tekst (plain text) zaopatrzony w odpowiednie oznaczenia. Dzięki temu użytkownik otrzymuje czytelny dokument, przy jednoczesnym ograniczeniu formy graficznej, która jest w tym konekście drugoplanowa. Przykład składni dokumentu Markdown zaprezentowano na listingu 4.1.

```
# Brief projektu

Ten dokument zawiera tzw "brief" projektu - wstepny opis
systemu, jaki mamy zamiar zbudowac.

## System w musi uwzgledniac
nastepujace funkcjonalnosci:

* logowanie uzytkownika
* dodawanie i zarzadzanie projektami
* dodawanie i zarzadzanie wymaganiami
* _tworzenie diagramow przypadkow uzycia_
```

Listing 4.1: przykład składni języka Markdown

Po konwersji przez Markdown do języka HTML, otrzymamy następujący wynik:

```
<h1>Brief projektu</h1>

<p>Ten dokument zawiera tzw "brief" projektu -
   wstepny opis systemu, jaki mamy zamiar zbudowac.</
   p>
```

```
4
        <h2>System musi uwzgledniac nastepujace
           funkcjonalnosci:</h2>
6
        <ul>
8         <li>logowanie uzytkownika</li>
          <li>dodawanie i zarzadzanie projektami</li>
10        <li>dodawanie i zarzadzanie wymaganiami</li>
          <li><em>tworzenie diagramow przypadkow uzycia</em
             ></li>
12      </ul>
```

Listing 4.2: wynik konwersji powyższego kodu do HTML

Dzięki temu, w łatwy sposób istnieje możliwość tworzenia spójnych dokumentów o logicznej strukturze, przy niewielkim nakładzie pracy.

Edytor diagramów UML

Edytor diagramów UML umożliwia wzbogacenie treści wymagania o graficzną reprezentację powiązanych przypadków użycia. Moduł ten wspiera współdzielenie diagramów pomiędzy wieloma wymaganiami, umożliwiając oznaczanie odpowiednich elementów jako odpowiedzialnych za realizację aktualnie opracowywanego wymagania.

Szczególnie istotnym elementem systemu jest generowanie obiektów aplikacji na podstawie danych odczytywanych bezpośrednio ze struktury diagramu. W efekcie, po utworzeniu elementu na diagramie i odpowiednim powiązaniu go z danym wymaganiem, powstanie reprezentacja tego elementu w postaci obiektu klasy Groovy, który następnie zostanie zapisany do bazy danych dzięki mapowaniu obiektowo - relacyjnemu będącego integralną częścią frameworka Grails. Jednocześnie zostanie zachowana odpowiednia relacja między utworzonym obiektem klasy UseCase, a wymaganiem. Mapowanie elementów utworzonych na diagramie przebiega dzięki możliwości serializacji diagramu do formatu XML. Szczegółowo autor opisuje ten mechanizm w rozdziale dotyczącym rozwiązań implementacyjnych.

W przyszłości, istnieje możliwość rozszerzenia funkcjonalności edytora

o kolejne rodzaje diagramów UML. Ponadto, w trakcie ewentualnej rozbudowy systemu, wygodnym usprawnieniem byłoby automatyczne przypisywanie nowo utworzonych obiektów do aktualnie edytowanego wymagania.

System załączników

System załączników dodatkowo wzbogaca możliwości ekspresji - do wymagania użytkownik może dodać dokumenty lub treści multimedialne powiązane z danym wymaganiem, mające na clu lepsze zrozumienie istoty problemu.

4.4.4 Uniwersalność formatu danych

Narzędzie do zarządzania wymaganiami powinno ułatwiać późniejsze procesowanie wprowadzonych do niego danych. Proponowany system realizuje tę potrzebę przechowując wszystkie wymagania oraz powiązane diagramy w bazie danych. Treść opisu wymagań jest przechowywana w postaci zwykłego tekstu, zaopatrzonego w znaczniki Markdown. Dzięki temu, treść wymagań jest znacznie łatwiejsza w przetwarzaniu, niż dokumenty tworzone przez klasyczne edytory tekstu.

Struktura diagramów powiązanych z wymaganiami jest przechowywana w bazie danych w formacie xml. Takie podejście również w znacznym stopniu ułatwia późniejsze przetwarzanie informacji zawartych na diagramach.

System aktywnie wykorzystuje powyższe właściwości podczas automatycznego generowania specyfikacji wymagań.

4.4.5 Zarządzanie zmianą

Zarządzanie zmianą jest istotne z punktu widzenia ewolucji wymagań i stopniowego uszczegóławiania dokumentacji systemu. Proponowane narzędzie wspiera ten proces zachowując poprzedni stan wymagania w trakcie jego zapisu. Dzięki temu możliwe jest prześledzenie zmian dokonywanych w wymaganiach.

4.4.6 Centralne repozytorium

Proponowany system jest dostępny przez przeglądarkę, dzięki czemu stanowi centralne repozytorium wiedzy na temat wymagań w projekcie. Interfejs www z elementami html5 powoduje, że narzędzie wymaga jedynie nowoczesnej przeglądarki internetowej, bez żadnych dodatkowych wtyczek. Dzięki temu aplikacja cechuje się wysokim poziomem dostępności.

4.4.7 Kolaboracja

Możliwość komentowania wymagań została zaimplementowana z myślą o umożliwieniu kolaboracji nad wymaganiami członkom zespołu.

4.4.8 Generator specyfikacji

Jedną z kluczowych funkcjonalności proponowanego rozwiązania jest automatyczne generowanie dokumentu specyfikacji wymagań na podstawie aktualnych danych w systemie. Użytkownik w każdej chwili jest w stanie wygenerować bieżącą wersję specyfikacji i poddać ją całościowej analizie. Częścią specyfikacji zostają zarówno wymagania jak i powiązane z nimi przypadki użycia.

4.4.9 Praktyczne zastosowanie

Proponowane funkcjonalności są oparte na rzeczywistych doświadczeniach, w rzeczywistych projektach w małych i średnich przedsiębiorstwach. Bazują na silnym przekonaniu, że narzędzie, aby było wykorzystywnane, musi być wygodne i proste w użyciu oraz wnosić wartość dodaną do istniejącej infrastruktury i metodyki pracy. Jednocześnie nie adresuje całego procesu zarządzania projektem - realizuje tylko swoje zadanie związane z przetwarzaniem i zarządzaniem wymaganiami. W dojrzałym przedsiębiorstwie, narzędzie służące do zarządzania wymaganiami powinno być dopełnione pełnoprawnym systemem zarządzania projektami, systemem kontroli wersji oraz systemem zgłaszania i monitorowania błędów (traq, redmine).

Rozdział 5

Technologie wykorzystane w implementacji

W poprzednim rozdziale opisano główne założenia proponowanego systemu i jego poszczególne funkcjonalności. W tym rozdziale zostaną opisane narzędzia, technologie i biblioteki, które zastosowano podczas budowy prototypu.

5.1 Zarys technologii

Implementację systemu oparto o technologie Java/Jee i pochodne. Jako podstawę szkieletu aplikacji wykorzystano framework Grails umożliwiający szybkie prototypowanie aplikacji internetowych. W warstwie bazy danych wykorzystano bazę postgresql oraz technologię mapowania obiektowo-relacyjnego Hibernate ORM, dostarczaną wraz z frameworkiem Grails.

Interfejs użytkownika oparto o Twitter Bootstrap - predefiniowany zestaw styli oraz komponentów do natychmiastowego wykorzystania w aplikacjach internetowych.

Moduł graficznej edycji diagramów UML został stworzony wykorzystując „jsUML2" - bibliotekę przygotowaną przez zespół prof. José Raúl'a Romero, na uniwersytecie w Kordobie.

Edytor tekstu oparty jest o bibliotekę javascript Ace Editor. Jest to wiodąca biblioteka, wykorzystywana m. in. w eksperymencie Cloud9 IDE -

webowym, zintegrowanym środowisku programistycznym zapoczątkowanym przez zespoł Mozilli, pod nazwą Mozilla Skywrite.

Pierwotnie docelowym środowiskiem testowym dla proponowanego systemu był Google App Engine, jednak ze względu na ograniczenia tej usługi, prototyp umieszczono na darmowej infrastrukturze firmy Heroku (heroku.com) znacznie lepiej obsługującej nowoczesne aplikacje oparte o Javę.

5.2 Framework Grails

Framework Grails jest stosunkowo młodą technologią opartą na sprawdzonych rozwiązaniach. Firma która zajmuje się rozwojem tej technologii jest tą samą organizacją, odpowiedzialną za stworzenie Spring Framework - jednego z najpopularniejszych i jednocześnie najbardziej rozbudowanych platform programistycznych dla języka Java dostępnych na rynku. To właśnie Spring Framework leży u podstaw Grails i wraz z Hibernate ORM stanowi trzon technologiczny tego narzędzia.

Grails integruje najlepsze praktyki i narzędzia zarówno ze Spring Framework jak i z Hibernate ORM. Jest poniekąd odpowiedzią środowiska Javy, na rosnącą konkurencyjność frameworków opartych o dynamicznie typowane, skryptowe języki programowania jak Django (język python) czy Ruby On Rails (Ruby). Jednocześnie, ze względu na fakt iż jest swoistą nakładką na technologie oparte na javie, dostarcza znacznie większych możliwości niż konkurenci, wynikających z rozbudowanego ekosystemu javy.

Język Groovy

Podstawowym językiem programowania w platformie Grails, jest język Groovy. Groovy jest dynamicznie kompilowanym językiem o składni i filozofii zbliżonej do takich języków jak Python lub Ruby. Kod bajtowy będący wynikiem kompilacji uruchamiany jest na wirtualnej maszynie Javy, przez co język integruje się niemal w sposób przeźroczysty z technologiami opartymi o JVM. Ponadto, kod programu napisanego w Javie jest poprawnym programem Groovy zarówno pod względem sytnaktycznym jak i semantycznym.

Bardziej zwięzła i przyjazna składnia Groovy wraz z szerokimi możliwościami Javy sprawia, że ten język skryptowy jest solidnym rozwiązaniem o szerokim spektrum zastosowań.

W celach porównawczych, poniżej zamieszczono znany, trywialny problem programistyczny „FizzBuzz", zarówno w języku Groovy, jak i Java. Program „FizzBuzz" otrzymując na wejściu ciąg liczb, wypisuje na konsolę słowo „Fizz" jeśli dana liczba jest podzielna przez 3, słowo „Buzz", jeśli dana liczba jest podzielna przez 5, natomiast słowo „FizzBuzz" w przypadku podzielności przez 15.

```
// Groovy:
for (i in 1..100) {
  println "${i%3?'':'Fizz'}${i%5?'':'Buzz'}" ?: i
}

//Java:
public class FizzBuzz{
  public static void main(String[] args){
    for(int i= 1; i <= 100; i++){
      if(i % 15 == 0){
        System.out.println("FizzBuzz");
      } else if(i % 3 == 0){
        System.out.println("Fizz");
      } else if(i % 5 == 0){
        System.out.println("Buzz");
      } else{
        System.out.println(i);
      }
    }
  }
}
```

Listing 5.1: program FizzBuzz

Grails

W szczególności filozofia Grails jest bardzo zbliżona do popularnego frameworka Ruby On Rails (RoR) umożliwiającego ekspresowe prototypowanie aplikacji obsługujących komunikację z bazą danych w zakresie (1) tworzenia, (2) odczytu, (3) aktualizacji i (4) usuwania danych za pomocą formularzy. Tego typu aplikacje popularnie nazywane są aplikacjami „CRUD" - create, retrieve, update, delete). Wiele przyjętych rozwiązań w Grails zostało zaczerpniętych z RoR, jednak oba frameworki znacznie różnią się pod względem technologicznym. Szczegółowe porównanie tych technologii leży poza zakresem tej pracy, jednak osoby zainteresowane, powinny zapoznać się z doskonałym wątkiem w serwisie Stackoverflow, przytoczonym w bibliografii [24].

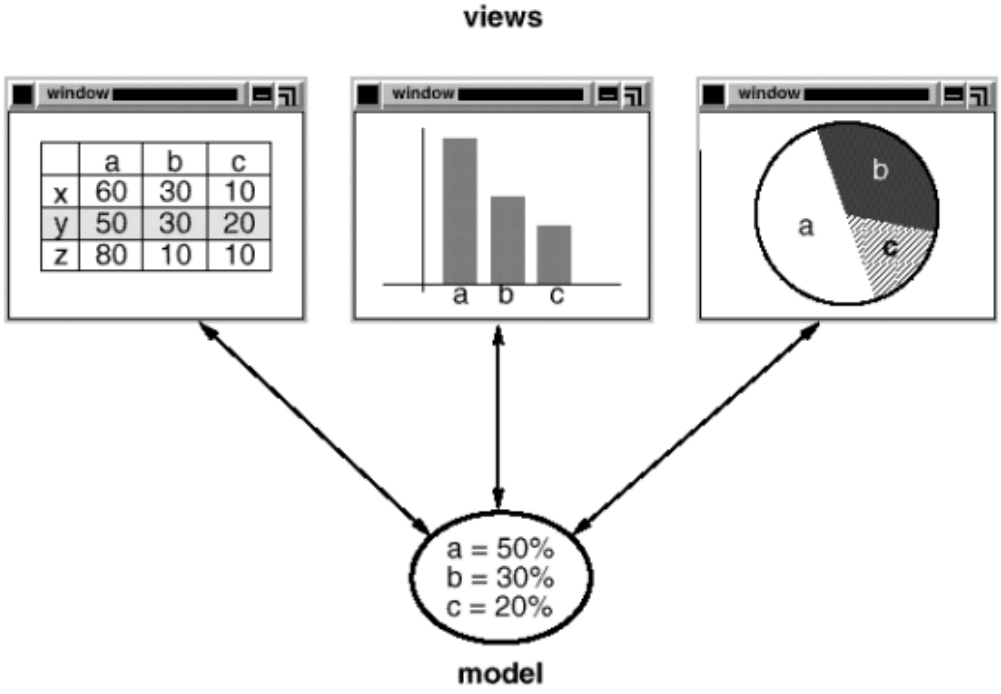
Grails jest silnie oparte o paradygmat „convention over configuration" - architekturze zakładającej minimalizację potrzeby konfiguracji aplikacji przez programistę na rzecz przyjętych konwencji, do których należy się stosować, aby wydajnie dostarczać działające rozwiązania. Podejście to dotyczy wszelkich aspektów związanych z tworzeniem aplikacji w Grails - od struktury katalogów, przez umiejscowienie i zawartość plików konfiguracyjnych, po nazewnictwo klas, zmiennych i zastosowanie odpowiednich struktur danych.

Model - View - Controller

Oparcie warstwy webowej w Grails na wzorcu Model-View-Controller (MVC) jest naturalnym podejściem w nowoczesnych frameworkach ułatwiających tworzenie aplikacji internetowych. Wzorzec MVC [13] pierwotnie dotyczył tworzenia aplikacji okienkowych w Smalltalk'u-80 [9]. Podstawowym założeniem MVC jest separacja danych i logiki od ich reprezentacji graficznej. MVC składa się z trzech rodzajów obiektów.

Model jest reprezentacją danych, wiedzy. Model może być odpowiednikiem jakiegoś obiektu lub strukturą wielu obiektów.

Widok jest wizualną reprezentacją modelu, „pobiera" dane na temat obiektu i wyświetla je w odpowiedni sposób, często uwydatniając pewne elementy modelu, inne z kolei odpowiednio ukrywając. Funkcja ta czyni z widoku



Rysunek 5.1: Model-View-Controller - różna reprezentacja graficzna tych samych danych. Źródło: [13].

pewnego rodzaju filtr prezentacji danych.

Kontroler stanowi łącznik pomiędzy użytkownikiem a systemem, z którym użytkownik wchodzi w interakcję. Dostarcza użytkownikowi odpowiednich widoków, adekwatnych do stanu systemu [23].

Praca z Grails

Struktura katalogów nowo utworzonej aplikacji Grails wygląda następująco:

```
+ grails-app
  + conf           ---> lokalizacja artefaktów konfiguracji
    + hibernate    ---> opcjonalne pliki konfiguracyjne hibernate
    + spring       ---> opcjonalne pliki konfiguracyjne spring
  + controllers    ---> kontrolery aplikacji
```

```
+ domain          ---> klasy reprezentujące modele
+ i18n            ---> zlokalizowane komunikaty il8n
+ services        ---> warstwa serwisowa
+ taglib          ---> biblioteki znaczników
+ util            ---> klasy pomocnicze special utility classes
+ views           ---> lokalizacja widoków
  + layouts       ---> lokalizacja
+ lib
+ scripts         ---> skrypty gant
+ src
+ groovy          ---> opcjonalnie, źródła groovy inne niż
                       w grails-app/*
+ java            ---> opcjonalnie, źródła java
+ test            ---> generowane testy aplikacji
+ web-app
  + WEB-INF
```

GORM - Grails Object Relational Mapping

Po wstępnej analizie, mając wiedzę na temat dziedziny problemu, w prosty sposób można utworzyć podstawową funkcjonalność przykładowej aplikacji. Proces tworzenia elementów budowanego systemu rozpoczyna się od stworzenia klasy Groovy reprezentującej model (na przykładzie implementacji z systemu Reqmanager):

```
1  package pl.edu.pjwstk.reqmanager.examples
   class Project {
3      String name
       String description
5      String mainNote
       java.sql.Timestamp timestamp
7      java.util.Date deadline

9      static hasMany = [requirements:Requirement]
```

```
}
```

Listing 5.2: klasa modelu Project.groovy

Jeśli nie skonfigurujemy aplikacji w niestandardowy sposób, po stworzeniu powyższej klasy Groovy w katalogu grails-app/domain, Framework Grails przy pomocy Hibernate utworzy następującą strukturę w bazie danych:

```
                    Table "public.project"
    Column    |            Type             | Modifiers
--------------+-----------------------------+-----------
 id           | bigint                      | not null
 version      | bigint                      | not null
 deadline     | timestamp without time zone |
 description  | character varying(255)      | not null
 name         | character varying(255)      | not null
 timestamp    | timestamp without time zone | not null
 main_note    | character varying(255)      |
Indexes:
    "project_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "requirement" CONSTRAINT "fk15a8dc43ffd118f2"
    FOREIGN KEY (project_id) REFERENCES project(id)
```

Listing 5.3: struktura bazy danych dla modelu Project

Powyższy przykład przedstawia, jak w prosty sposób, przy niskich nakładach pracy, programista jest w stanie zbudować szkielet aplikacji zasilanej bazą danych. Jak widać, pola klasy Groovy, zostały odpowiednio odwzorowane w strukturze danych wraz z odpowiednimi typami danych po stronie bazy.

Na szczególną uwagę zasługuje konstrukcja w klasie Project:

```
    static hasmany = [requirements:requirement]
```

Listing 5.4: relacja jeden-do-wielu

W języku Groovy konstrukcja „[:]” jest reprezentacją pustej tablicy asocjacyjnej - struktury danych umożliwiającej przechowywanie kolekcji par „klucz - wartość”. W Groovy, domyślną kolekcją inicjalizowaną przy pomocy literału „[:]” jest obiekt klasy java.util.LinkedHashMap. Zatem powyższa

zmienna statyczna jest referencją do obietku klasy LinkedHashMap, zawierającego jedną parę, gdzie kluczem jest ciąg znaków „requirements" a wartością, jest klasa Requirement. W ten sposób Grails obsługuje relacje jeden-do-wielu. W analogiczny sposób, istnieje możliwość definiowania pozostałych rodzajów relacji, tj. jeden-do-jeden, wiele-do-wielu, etc). Konstrukcja taka, umożliwia uzyskanie listy wymagań w projekcie:

```
1       //pobiera z bazy projekt o id = 1
        def project = Project.get(1)
3
        //zwraca liste wymagan podlaczonych do projektu
5       def reqs = project.requirements
```

Listing 5.5: wymagania przypisane do projektu

Kontrolery i widoki

W celu połączenia konkretnych widoków z akcjami kontrolerów, konwencja Grails wymaga odpowiedniego nazewnictwa metod w klasach kontrolerów. Generalizując, należy stosować się do zasady, aby nazwa metody odpowiedzialnej za realizację wybranej akcji kontrolera, posiadała plik o tej samej nazwie, z rozszerzeniem .gsp, w odpowiednim katalogu w ścieżce grails-app/views. Poniżej zamieszczono przykład realizacji akcji „index" w kontrolerze „ProjectController". Wbudowany w Grails mechanizm odwzorowywania adresów URL i przekierowywania do odpowiednich akcji kontrolerów automatycznie wykrywa którą metodę należy wykonać. Zatem, po wskazaniu przeglądarce internetowej adresu http://reqmanager.heroku.com/project/index system, przez konwencję, wykryje konieczność wywołania metody „index" w kontrolerze „ProjectController". Z kolei po wykonaniu tej metody, (o ile programista nie zdecyduje inaczej) automatycznie wyświetlany jest widok index.gsp umieszczony w katalogu grails-app/views/project. W widoku, do wykorzystania dostępne są zmienne usatwione w tabeli asocjacyjnej, zwracanej przez kontroler (w tym przypadku, zmienna „project" zawierająca referencję do listy wszystkich projektów w systemie).

```
1       // klasa kontrolera:
```

```
package pl.edu.pjwstk.reqmanager

class ProjectController {
  def index = {
    return [projects : Project.list()]
  }
}

//widok w pliku o nazwie index.gsp
//umieszczony w katalogu grails-app/views/project/
<!DOCTYPE html>
<html>
  <div>
  <h1>Project list</h1>
  <g:each in='${projects}' var='project'>
    Nazwa: ${project.name}<br />
    Deadline: ${project.deadline.format("dd-MM yy")}<br
        />
    Opis: ${project.description}
  </g:each>
  </div>
</html>
```

Listing 5.6: widok i kontroler

Należy jeszcze zwrócić uwagę na pewien detal w strukturze kodu źródłowego kontrolera. Grails dopuszcza dwa podejścia do deklarowania akcji - (1) za pomocą klasycznych metod oraz (2) przy użyciu dokmnięć (closures). Formalnie domknięcia są obiektami wiążącymi funkcję oraz środowisko w jakim ta funkcja ma działać. Jest to specyficzna kontrukcja, będącą zdefiniowanym blokiem kodu, który można przekazać jako parametr innej metodzie lub funkcji. Poniższy przykład wyjaśnia zasadę na jakiej działają domknięcia w Groovy:

```
def closeAllProjects() {
  def projects = Project.list()
  projects.collect { it.open = false }
}
```

Listing 5.7: domknięcie w Groovy

Powyższy kod pobiera do zmiennej „projects” listę wszystkich projektów w systemie. Następnie, na liście projektów wywoływana jest metoda „collect”. Metoda collect (zdefiniowana w interfejsie Collection) przyjmuje jako parametr blok kodu, domknięcie. Metoda collect iteruje po wszystkich elementach zbioru na rzecz którego została wykonana i wykonuje blok kodu przekazany jej jako parametr na kolejnych elementach kolekcji, następnie zwracaja listę zmodyfikowanych przez domknięcie elementów.

5.2.1 Spring Framework i Hibernate ORM

Sercem Grails jest Spring Framework oraz Hibernate ORM. Obie technologie są wiodące na rynku i szeroko stosowane w komercyjnych projektach dowolnych rozmiarów. Spring jest de facto zestawem narzędzi, wzorców projektowych i bibliotek realizujących ogromną ilość funkcjonalności, mających za zadanie przyśpieszyć proces wytwarzania oprogramowania oraz w pewnym zakresie zapewnić wysoką jakość tworzonych rozwiązań. Przede wszystkim jednak, jest tzw. kontenerem Inversion Of Control (odwrócenie sterowania) [12]. Prócz IoC Spring Framework to potężna platforma integrująca wiele rozwiązań, takich jak webowy framework Spring MVC, biblioteki i wzorce dotyczące bezpieczeństwa (Spring Security), wrapper jdbc, transakcje (Spring JDBC Templates), etc.

Hibernate ORM jest technologią pozwalającą na mapowanie rekordów z relacyjnych baz danych, na obiekty w systemie przy pomocy plikow konfiguracyjnych xml. W Grails, potrzeba konfiguracji warstwy ORM ogranicza się do minimum, a większość standardowych zadań można zrealizować konfigurując odpowiednio blok *mapping* w klasie modelu.

5.3 Pozostałe technologie

5.3.1 Postgresql

Trwałość danych w aplikacjach internetowych standardowo realizowana jest przez bazę danych. Na potrzeby prezentowanego prototypu, wykorzystano

bazę danych Postgresql. Jest to obiektowo-relacyjna baza danych udostępniona na licencji Wolnego i Otwartego Oprogramowania.

Baza danych PostgreSQL implementuje znaczną część standardu SQL:2008 [!ref - http://www.postgresql.org/docs/9.1/static/features.html]

5.3.2 Javascript, jQuery i biblioteka jsUML2

Wiele funkcjonlaności zrealizowano po stronie klienta, wykorzystując język javascript. W niewielkim zakresie wykorzystano popularną bibliotekę jQuery, wspierającą realizację najczęściej pojawiających się problemów w trakcie prac nad aplikacjami intenretowymi. Jednak wykorzystanie jQuery w prototypie systemu jest marginalne i ogranicza się jedynie do zastosowania metody *ajax()* ułatwiającej wykonywanie asynchronicznych wywołań serwera.

jsUML2

jsUML2 jest biblioteką stworzoną przez środowisko naukowe, na uniwerystecie w Kordobie. Jest to zaimplementowany w całości w języku javascript, rozbudowany zestaw klas umożliwiający tworzenie wielu rodzajów diagramów UML w przeglądarce internetowej. jsUML wykorzystuje najnowszą specyfikację języka znaczników HTML5. Obiekt *canvas* oferuje szerokie spektrum możliwości programistom aplikacji internetowych. Element *canvas* umożliwia programistyczne tworzenie kształtów i obrazów bitmapowych w przeglądarce internetowej. Canvas udostępnia programistom interfejs HTMLCanvasElement umożliwiający modyfikację zawartości elementów canvas w htmlu. Metoda interfejsu *getContext()* zwraca tzw. „drawing context" - w zależności od parametru przekazanego metodzie *getContext()*, zwracany jest obiekt klasy (1) CanvasRenderingContext2D w przypadku przekazania paramteru '2d' lub (2) WebGLRenderingContext w przypadku przekazania paramteru 'experimental-webgl'.

Poniżej zamieszczono przykładowy kod, umożliwiający rozpoczęcie pracy obiektem klasy CanvasRenderingContext2D. Wykonanie kodu spowoduje narysowanie prostokąta:

```
var canvas = document.getElementById('example');
```

```
2       var drawingContext = canvas.getContext('2d')

4       drawingContext.fillStyle = "rgb(200,0,0)";
        drawingContext.fillRect(10, 10, 55, 50);
```

Listing 5.8: przykład HTML5 canvas

Biblioteka jsUML2 składa się z dwóch pakietów klas: (1) *UDCore* - zawierający bazowe klasy umożliwiające tworzenie i obsługę graficzną diagramów i komponentów pomocniczych; (2) *UDModules* - zawierający faktyczną implementację diagramów standard UML, zbudowaną na bazie klas pierwotnych z pakietu *core*.

Głównymi elementami w poszczególnych pakietach są:

- UDCore
  - Diagram
  - Node (węzeł)
  - Relation (relacja)
- Modules
  - Use case diagrams
  - Class diagrams
  - Component diagrams
  - Message sequence diagrams
  - State charts

Na listingu 5.9 zawarto wykaz klas dostępnych w pakiecie *core*:

```
1       AttributeFields.js
        AttributeItem.js
3       CircleSymbol.js
        CollapsibleFields.js
5       Component.js
        ComponentSymbol.js
7       ConnectorItem.js
```

```
        DataStoreItem.js
        Diagram.js
        Dialog.js
        Element.js
        Elliptical.js
        GuardItem.js
        JSFun.js
        JSGraphic.js
        LoopItem.js
        NodeFigure.js
        Node.js
        ObjectItem.js
        OperationFields.js
        OperationItem.js
        Point.js
        Rectangular.js
        RegionItem.js
        Region.js
        RegionLine.js
        RelationEnd.js
        Relation.js
        RelationLine.js
        Rhombus.js
        RoleItem.js
        Rombo.js
        Separator.js
        Space.js
        SpecificationItem.js
        StereotypeFields.js
        StereotypeItem.js
        SuperComponent.js
        SuperNode.js
        Tab.js
        TextArea.js
        TextBox.js
        TextFields.js
        Text.js
        TransitionItem.js
```

Listing 5.9: klasy bazowe jsUML2

W pakiecie modules, do dyspozycji programisty znajdują się następujące podpakiety klas:

```
activity
class
component
generic
profile
sequence
stateMachine
usecase
```

Listing 5.10: pakiety modułów jsUML2

W celu inicjalizacji biblioteki jsUML2, należy stworzyć odpowiednią strukturę dokumentu html:

```
<div id="ud_diagram_div" style="position: relative">
  <canvas id="c1" class="ud_diagram_canvas" style="
     position: absolute;" width="500")
  <canvas id="c2" class="ud_diagram_canvas" style="
     position: absolute;" width="500")
</div>
```

Listing 5.11: HTML canvas element

Następnie, niezbędna jest inicjalizacja odpowiedniego diagramu (na przykładzie diagramu przypadków użycia):

```
var mainEl = document.getElementById('c1');
var motionEl = document.getElementById('c2');

var mainCtx = mainEl.getContext('2d');
var motionCtx = motionEl.getContext('2d');

var div = document.getElementById('ud_diagram_div')

var diagram = new UMLUseCaseDiagram()

//metoda initialize() przyjmuje kolejno parametry:
//(1) unikalny identyfikator diagramu
//(2) element blokowy html bedacy kontenerem diagramu
```

```
14          //(3) obiekt 'context' stanowiacy 'gorna' warstwe
            //     diagramu
16          //(4,5) rozmiary w pikslach

18          diagram.initialize(1, div, motionCtx, 600, 600)
```

Listing 5.12: HTML canvas element - JS

5.3.3 System kontroli wersji git i serwer heroku

W trakcie pracy wykorzystano system kontroli wersji *git* oraz serwer heroku.

Git jest systemem kontroli wersji udostępnionym na licencji GPL, stworzonym przez Linusa Torvaldsa w 2005 roku. Głównymi założeniami projektu były szybkość działania, bezpieczeństwo oraz rozwiązanie problemów jakie twórca uważał są najbardziej rażące w systemach CSV oraz SVN.

Heroku udostępnia platformę hostingową obsługującą wiele języków programowania. Zasada działa serwerów Heroku, oparta jest na koncepcji utrzymania aplikacji w „chmurze" serwerów - infrastruktura sprzętowa jest niewidoczna dla użytkowników. Użytkownicy tej usługi wykupują moc obliczeniową serwerów odpowiadającą potrzebom utrzymywanych aplikacji, płacąc za czas dostępu do serwerów o wybranej mocy obliczeniowej. Główną zaletą tego rozwiązania jest możliwość natychmiastowego skalowania aplikacji w zależności od zapotrzebowania na moc obliczeniową. W przypadku znacznego wzrostu ruchu na serwerze, użytkownik może wykupić dodatkową moc obliczeniową, bez konieczności przerywania ciągłości działania usługi w sieci. Heroku udostępnia swoją infrastrukturę w podstawowym zakresie, bez konieczności uiszczania opłat, dlatego jest to rewelacyjne rozwiązania na potrzeby testów i prezentacji aplikacji internetowych. Dzięki natywnemu wsparciu systemu kontroli wersji Git, instalacja aplikacji w chmurze, polega na przesłaniu na serwer aktualizacji kodu źródłowego i wykonania polecenia inicjalizującego instalację kodu na serwerze heroku:

```
            ~$ git add .
2           ~$ git commit -m 'update'
            ~$ git push origin master
```

5.3.4 Środowisko programistyczne (Linux, vim)

Do stworzenia zarówno prorotypu aplikacji, jak i do edycji i składu części tekstowej niniejszej pracy, wykorzystano zasadniczo narzędzia dostępne w systemie operacyjnym Linux (Ubuntu Linux v11.10).

Do edycji kodu źródłowego wykorzystano klasyczny edytor tekstowy vim z zainstalowanym dodatkiem ułatwiającym poruszanie się w strukturze katalogów grails (vim grails plugin z zainstalowanym dodatkiem ułatwiającym poruszanie się w strukturze katalogów grails (vim grails plugin).

Część opisowa niniejszej pracy również powstała w edytorze tekstu vim. Do składu tekstu zastosowano popularne oprogramowanie LaTeX.

Rozdział 6

Rozwiązania implementacyjne

W poprzednim rozdziale opisano narzędzia jakie wykorzystano w trakcie implementacji prototypu proponowanego rozwiązania. W niniejszym rozdziale, zostaną zaprezentowane konkretne rozwiązania implementacyjne zastosowane w prototypie. Szczególną uwagę zwrócono na moduł edycji diagramów przypadków użycia oraz generatora specyfikacji wymagań.

6.1 Architektura systemu

Jak wspomniano w ogólnym opisie prototypu w rozdziale 4, definiowanie wymagań w systemie Reqmanager polega na jednoczesnym dostępie do edytora tekstu oraz edytora diagramów przypadków użycia. Użytkownik, mając otwartą aplikację, oprócz tekstowego opisu wymagania, ma możliwość dodawania oraz edycji przypadków użycia w formie graficznej.

Aplikacja Reqmanager jest typową aplikacją internetową typu klient-serwer, gdzie klientem jest przeglądarka użytkownika korzystającego z aplikacji zainstalowanej na zdalnym serwerze, odpowiednio reagującej na żądania klienta na zasadzie żądanie - odpowiedź (request - response). Przeglądarka internetowa jest tak zwanym „cienkim klientem" z racji silnego uzależnienia działania od serwera aplikacji.

Framework Grails narzuca architekturę kodu aplikacji według wzorca Model-View-Controller, opisanego w rozdziale 5.

Na potrzeby aplikacji Reqmanager, utworzono następującą strukturę modeli aplikacji:

```
Diagram
Project
Requirement
UseCase
```

Struktura bazy danych jest generowana automatycznie, na podstawie analizy klas modeli:

```
                List of relations
Schema |          Name           |   Type   |  Owner
--------+-------------------------+----------+----------
public | diagram                 | table    | postgres
public | hibernate_sequence      | sequence | postgres
public | project                 | table    | postgres
public | requirement             | table    | postgres
public | requirement_use_cases   | table    | postgres
public | use_case                | table    | postgres
```

6.1.1 Tworzenie diagramów

Diagramy przypadków użycia w systemie edytowane są po stronie klienta, dzięki wykorzystaniu elementu *canvas* wprowadzonego niedawno do specyfikacji języka HTML5. Obsługa tej funkcjonalności po stronie przeglądarki, stanowi problem związany z zachowaniem stanu diagramu w systemie. W celu rozwiązania tego problemu, zaimplementowano mechanizm serializacji całego diagramu w formacie xml do bazy danych. Cel ten osiągnięto, wykorzystując metodę *getXMLString()* udostępnioną przez wszystkie obiekty dziedziczące po klasie *Diagram* w bibliotece jsUML2. Przykładową strukturę

zserializowanego diagramu, umieszczno na Listingu 6.3. Dodanie nowego diagramu wiąże się zatem z inicjalizacją i wyświetleniem użytkownikowi obszaru roboczego diagramu bez komunikacji z serwerem. Diagram tworzony jest po załadowaniu widoku edycji wymagania:

```
window.onload = function() {
  var app = new AppUseCase("ud_diagram_div");
}
```

Listing 6.1: widok requriement/edit.gsp

AppUseCase jest obiektem javascript typu singleton, którego skrócona implementacja została przedstawiona na poniższym listingu:

```
var AppUseCase = function(elementId) {
  var xmlstr = document.getElementById('diagramXml').
      value
  var ucDiag = new UMLUseCaseDiagram()
  if(xmlstr) {
    ucDiag.setXMLString(xmlstr);
  }

  var c1 = document.getElementById('c1');
  var c = c1.getContext('2d');
  var c2 = document.getElementById('c2');
  c2.onmousedown = function() { return false; };
  var mc = c2.getContext('2d')}

  var div = document.getElementById('ud_diagram_div');

  ucDiag.initialize(11, div, c, mc, 600, 1000);
  ucDiag.draw();
}
```

Listing 6.2: implementacja obiektu JS AppUseCase

Powyższy kod w javascript, pobiera wartość z elementu formularza i identyfikatorze „diagramXml" oraz tworzy nowy obiekt UMLUseCaseDiagram, zdefiniowany w pakiecie *modules* biblioteki jsUML2. Następnie weryfikuje czy pole *diagramXml* jest niepuste. Jeśli warunek jest prawdziwy, wywołuje metodę *setXMLString()* na obiekcie diagramu, dzięki czemu diagram

inicjalizowany jest z wartościami wcześniej zserializowanymi do formatu xml i zapisanymi w bazie danych. Kolejne wiersze zajmują się utworzeniem kontekstów na bazie elementów *canvas*, aby na końcu zainicjować i narysować odpowiedni diagram w oknie przeglądarki użytkownika. Należy jednocześnie zaznaczyć, iż w przypadku gdy nie istnieje zserializowana wersja diagramu w bazie (pole *diagramXml* jest puste), wówczas zostanie zainicjalizowany nowy, pusty diagram, nieposiadających żadnych elementów. Pole *diagramXml* jest wcześniej ustawiane na podstawie danych z bazy.

Zapis danych xml

Dopiero gdy użytkownik wybierze opcję zapisu aktualnego wymagania, zserializowany stan diagramu w formacie xml, zostanie przesłany metodą POST do serwera. Następnie otrzymany po stronie serwera ciąg znaków xml zostanie przetworzony, utworzony zostanie obiekt klasy Diagram i na końcu nastąpi zapis obiektu do bazy danych. Za ten proces odpowiedzialna jest akcja *save* kontrolera *Requirement*. Jest to dość ograniczone rozwiązanie, które można byłoby usprawnić wprowadzając asynchroniczny, automatyczny zapis stanu diagramu co kilka sekund lub po każdej aktualizacji po stronie klienta.

```
<UMLUseCaseDiagram name="Use case diagram">
  <UMLSystem id="UMLSystem_4" x="165" y="97" ... >
    <superitem id="stereotypes" visibleSubComponents="
       true"/>
    <item id="name" value="System name"/>
    <UMLUseCase id="UMLUseCase_1" x="232" y="142" ... >
      <superitem id="stereotypes" visibleSubComponents=
         "true"/>
      <item id="name" value="test test"/>
    </UMLUseCase>
    <UMLUseCase id="UMLUseCase_3" x="222" y="209" ... >
      <superitem id="stereotypes" visibleSubComponents=
         "true"/>
      <item id="name" value="for req 74"/>
    </UMLUseCase>
  </UMLSystem>
```

```
</UMLUseCaseDiagram>
```

Listing 6.3: struktura xml diagramu

Baza danych postgresql obsługuje specjalny typ danych „xml". Dzięki obsłudze tego typu danych, po stronie bazy dokonywana jest weryfikacja poprawnosci dokumentu jaki programista ma zamiar zapisać. Framework Grails nie współpracuje dobrze z typem danych xml w bazie, dlatego niezbędna była implementacja rozwiązania rzutującego typ String na typ SQLXMLType.W tym celu została stworzona specjalna klasa w Javie, implementująca interfejs *org.hibernate.usertype.UserType*. Hibernate udostępnia ten interfejs umożliwiając programistom tworzenie własnych typów danych i zdefiniowanie ich zachowania w stosunku do bazy danych. Dodatkowo, niezbędne było odpowiednie przygotowanie modelu, podając typ pola przechowującego xmla jako String, a następnie sworzyć odpowiedni blok „mapping" w modelu. Na Listingu 6.4, dla przykładu zaprezentowano implementację klasy Diagram z systemu Reqmanager:

```
package pl.edu.pjwstk.reqmanager
public class Diagram {
  String name
  String xmlString

  static belongsTo = Requirement
  static mapping = {
    xmlString type: SQLXMLType
  }
  static constraints = {
    name(nullable:true)
  }
}
```

Listing 6.4: implementacja modelu Diagram

```
package pl.edu.pjwstk.reqmanager

import java.io.Serializable;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
```

```
import java.sql.SQLException;
import java.sql.Types;
import org.hibernate.HibernateException;

/**
 * Store and retrieve a PostgreSQL "xml" column as a
    Java string.
 */
public class SQLXMLType implements org.hibernate.
   usertype.UserType {

    private final int[] sqlTypesSupported = new int[] {
        Types.VARCHAR };

    @Override
    public int[] sqlTypes() {
        return sqlTypesSupported;
    }

    @Override
    public Class returnedClass() {
        return String.class;
    }

    @Override
    public boolean equals(Object x, Object y) throws
       HibernateException {
        if (x == null) {
            return y == null;
        } else {
            return x.equals(y);
        }
    }

    @Override
    public int hashCode(Object x) throws
       HibernateException {
        return x == null ? null : x.hashCode();
    }
```

```
41          @Override
            public Object nullSafeGet(ResultSet rs, String[]
                names, Object owner) throws HibernateException,
                SQLException {
43           assert(names.length == 1);
             String xmldoc = rs.getString( names[0] );
45           return rs.wasNull() ? null : xmldoc;
            }
47
            @Override
49          public void nullSafeSet(PreparedStatement st,
                Object value, int index) throws
                HibernateException, SQLException {
             if (value == null) {
51               st.setNull(index, Types.OTHER);
             } else {
53               st.setObject(index, value, Types.OTHER);
             }
55          }

57          @Override
            public Object deepCopy(Object value) throws
                HibernateException {
59           if (value == null)
                 return null;
61           return new String( (String)value );
            }
63
            @Override
65          public boolean isMutable() {
             return false;
67          }

69          @Override
            public Serializable disassemble(Object value)
                throws HibernateException {
71           return (String) value;
            }
```

```
73
            @Override
75          public Object assemble(Serializable cached, Object
                owner) throws HibernateException {
                 return (String) cached;
77          }

79          @Override
            public Object replace(Object original, Object
                target, Object owner) throws HibernateException
                {
81               return original;
            }
83      }
```

Listing 6.5: implementacja niestandardowego typu SQLXMLType

6.1.2 Współdzielenie diagramów

Istotnym elementem systemu jest współdzielenie diagramów między wieloma wymaganiami. Każdy diagram może zostać przypisany do kilku wymagań, dzięki czemu wiele wymagań może wykorzystywać i rozszerzać już istniejące diagramy. Modyfikacja diagramu przypadku użycia w trakcie edycji konkretnego wymagania, wiąże się z aktualizacją stanu diagramu we wszystkich korzystających z niego wymaganiach.

6.1.3 Mapowanie przypadków użycia

W związku z możliwością współdzielenia diagramów pomiędzu wieloma wymaganiami, w aktualnej wersji prototypu, przypadki użycia są łączone z wymaganiem, tylko po wykonaniu przez użytkownika akcji wskazania przypadku użycia jako realizującego aktualnie modyfikowane wymaganie.

Na etapie edycji diagramu, użytkownik ma możliwość wskazania konkretnych use case'ów znajdujących się na diagramie, jako odpowiedzialnych za realizację aktualnie edytowanego wymagania. Powiązanie przypadku użycia z wymaganiem polega na wyborze opcji *mark for requirement* z paska narzędzi

diagramu, a następnie kliknięcie w dany przypadek użycia na diagramie. Po zatwierdzeniu operacji, zostaje wysłane asynchroniczne żądanie do serwera, pod adres http://reqmanager.herokuapp.com/requirement/addUseCase/id_wymagania. W żądaniu przekazywane są do serwera parametry niezbędne dla zestawienia przypadku użycia z wymaganiem. Poniżej zamieszczono listę tych parametrów:

- *id* - identyfikator wymagania do którego dodawany jest przypadek użycia
- *clickedUCName* - nazwa przypadku użycia wprowadzona przez użytkownika
- *clickedUCId* - identyfikator przypadku użycia wygenerowany automatycznie przez bibliotekę jsUML2

Asynchronicznie wysłane do serwera żądanie zostaje przekazane wraz z paramterami do akcji *addUseCase* kontrolera *Requirement*. Kod odpowiedzialny za połączenie przypadku użycia z wymaganiem, został załączony na Listingu 6.6.

```
def addUseCase = {
  def requirement = Requirement.get(params.id)
  def useCase = UseCase.findByTitle(params.clickedUCName)

  if(useCase == null) {
    useCase = new UseCase(title: params.clickedUCName,
        code: params.clickedUCId)
      requirement.addToUseCases(useCase)
      render(text: "ok")
  } else {
    if(!requirement.useCases.contains(useCase)) {
      requirement.addToUseCases(useCase)
      render(text: "ok")
    } else {
      render(text: "wymaganie zawiera ten przypadek")
    }
  }
```

```
17      }
```

Listing 6.6: addUseCase

Jak można odczytać z powyższego kodu źródłowego, akcja ta, pobiera odpowiednie wymaganie oraz podejmuje próbę znalezienia przypadku użycia po nazwie. Jeśli dany przypadek użycia nie istnieje jeszcze w bazie danych (nowo utworzony przypadek użycia), zostaje swtorzony nowy obiekt klasy UseCase z odpowiednimi parametrami (nazwa nadana przez użytkownika oraz identyfikator wygenerowany po stronie klienta). W przeciwnym razie, zostaje on dodany do listy przypadków użycia połączonych asocjacją z danym wymaganiem, o ile nie istnieje już na tej liście. Cała operacja wykonywana jest asynchronicznie, po stronie serwera.

6.1.4 Przypadki użycia a wymagania

W celu odpowiedniego oznaczenia przypadków użycia przypisanych do wybranego wymagania, zaimplementowano algorytm dwustopniowej inicjalizacji diagramu. Pierwszym krokiem jest ekstrakcja z bazy danych odpowiedniego wymagania, weryfikacja czy do danego wymagania przypisany jest diagram oraz czy istnieją jakieś przypadki użycia dla wymagania. Na podstawie informacji z bazy danych przekazanych przez kontroler, ładowany jest widok *requirement/show.gsp*. Następnie, po załadowaniu większej części strony uruchamiany jest skrypt po stronie klienta, odwołujący się asynchronicznie z powrotem do serwera z „pytaniem” o nazwy przypadków użycia przypisanych do danego wymagania. Jednocześnie, do pamięci ładowana jest struktura xml diagramu. Program iteruje po wszyskich węzłach w poszukiwaniu pokrywających się przypadków użycia. Jeśli znajdzie choć jeden, zmienia kolor jego tła na żółty.

```
1
    var reqUseCases = $.ajax({
3     type: "GET",
      url: "http://reqmanager.herokuapp.com/requirement/
          getUseCases/" + reqId,
5     cache: false
```

```
}).done(function(xht){
  var dom = (new DOMParser()).parseFromString(diag.
     diagram.getXMLString(), "text/xml");
  var documentEle = dom.documentElement;
  var usecaseArray = documentEle.getElementsByTagName('
     UMLUseCase');

  for(var i = 0; i < usecaseArray.length; i++) {
    for(var j = 0; j < usecaseArray[i].childNodes.length;
        j++) {
      var itemValue = usecaseArray[i].childNodes[j].
         attributes.value;
      if(itemValue != undefined) {
        if(xht.indexOf(itemValue.value.toString()) != -1)
            {
          var divv = document.getElementById("
             ud_diagram_div");
          var foundUseCase = null;

          for(var k = 0; k < diag.diagram._nodes.length;
             k++) {
            if(diag.diagram._nodes[k].getName().toString
               ()
             === xht[xht.indexOf(itemValue.value.
                toString())]) {
              foundUseCase = diag.diagram._nodes[k];
            }
          };
          if(foundUseCase) foundUseCase.
             setBackgroundColor("#F3F5BC");
          diag.diagram.draw();
        }
      }
    }
  }
});
```

Listing 6.7: Kolorowanie tła przypadków użycia

6.1.5 Edytor tekstu

W celu ułatwienia użytkownikowi pracy z tekstem, zaimplementowano edytor, oparty na bibliotece Ace Editor. Kod inicjujący edytor tekstu, znajduje się poniżej:

```
var editor = ace.edit("description");
var MarkdownMode = require("ace/mode/markdown").Mode;
editor.setTheme("ace/theme/clouds_midnight");
editor.getSession().setMode(new MarkdownMode());
editor.getSession().setUseWrapMode(true);
```

Listing 6.8: Inicjalizacja edytora tekstu

Powyższy kod inicjuje edytor, załączając go do wskazanego węzła w dokumencie html (w tym przypadku jest to warstwa „description"). Następnie ustawia wygląd graficzny edytora oraz przestawia go w tryb pracy z formatem Markdown. Zawartość edytora tekstu, przekazywana jest do serwera po wysłaniu formularza przez użytkownika. Ponieważ zawartość formularza znajduje się w elemencie *div* w strukturze DOM strony, w celu przekazania jego wartości, jest ona kopiowana do specjalnego pola typu textarea:

```
$("#submitbtn").click(function() {
  textarea.val(editor.getSession().getValue());
  $("#diagramXml").val(app.diagram.getXMLString());
});
```

Listing 6.9: Inicjalizacja edytora tekstu

6.1.6 Generowanie dokumentacji

6.2 Zalety i wady proponowanego rozwiązania

Proponowane rozwiązanie jest podjęciem próby uproszczenia procesów inżynierii wymagań. Głównym celem stworzenia prototypu było dostarczenie narzędzia umożliwiającego efektywne zbieranie i przetwarzanie wymagań przy jednoczesnym uproszczeniu poziomu jego skomplikowania. Powstał prototyp realizujący te cele, jednak siłą rzeczy jest to narzędzie przystosowane jedynie

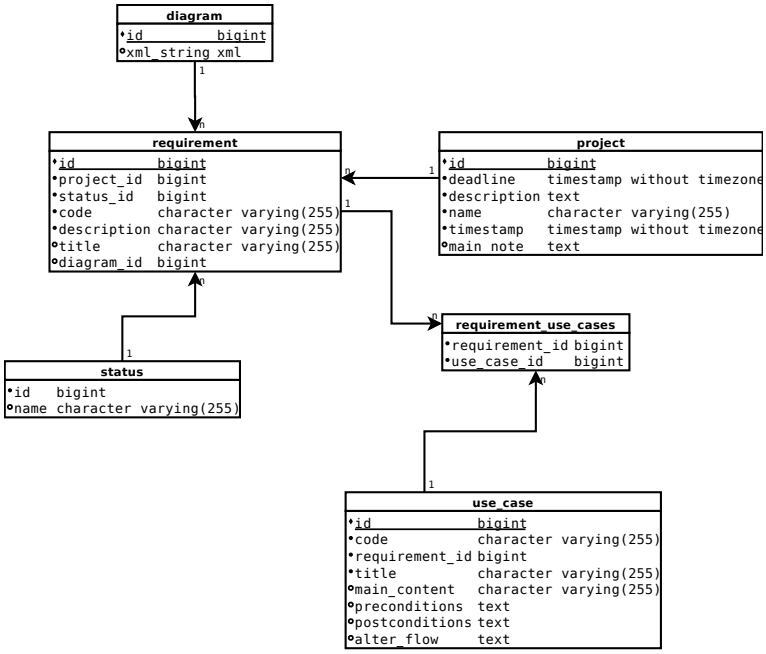
do małych projektów. W przeciwieństwie do istniejących rozwiązań, filozofia proponowanego systemu inspirowana była filozofią systemów Unix - programy i aplikacje powinny wykonywać jedną czynność i powinny ją wykonywać możliwie najlepiej. Jednocześnie należy tworzyć aplikacje tak, aby mogły w łatwy sposób współpracować z innymi systemami. Dlatego proponowane rozwiązanie, aby rozwiązywało skutecznie problemy ogólnie pojętej inżynierii oprogramowania, powinno być stosowane komplementarnie z innymi narzędziami, w szczególności systemem zarządzania projektami i zespołem. Od świadomości i dojrzałości organizacji zależy, jak taka infrastruktura zostanie zbudowana oraz jak wybrane narzędzia będą funkcjonowały w organizacji.

Dużą zaletą systemu jest dostępność przez przeglądarkę internetową przy jednoczesnej realizacji funkcjonalności manipulacji obiektami graficznymi. W szczególności zastosowanie technologii HTML5 daje przewagę w dostępności nad istniejącymi rozwiązaniami, najczęściej implementowanymi w technologii Flash. Rzadko też można spotkać podobny poziom elastyczności aplikacji do zarządzania wymaganiami. Koncepcja systemu celowo nie narzuca wielu ograniczeń związanych ze strukturą wymagań w wierze, że brak restrykcji projektanta wpłynie pozytywnie na proces definiowania i formułowania wymagań.

Sam wybór technologii jaką jest Grails, jest zaletą stworzonego prototypu. Zastosowanie tej technologii umożliwia łatwą rozbudowę i skalowanie systemu. System wtyczek platformy Grails sprzyja modułowej naturze aplikacji stworzonych w tej technologii. Z kolei zarządzający całą aplikacją „silnik" oparty na String Framework, Hibernate ORM oraz filozofii „convention over configuration" sprzyjają stosowaniu najlepszych praktyk programistycznych i ponowne wykorzystanie kodu.

Z pewnością istotną wadą jest brak obsługi „śledzenia" wymagań (traceability). Jednak w istniejących rozwiązaniach śledzenie wymagań nie rozwiązuje w pełni problemu. Nadal należy dbać o zgodność kodów i identyfikatorów wymagań w komunikacji z wieloma systemami. W dobrze zorganizowanej firmie, te problemy można wyeliminować przez narzucenie odpowiedniej kultury pracy, dbałość o szczegóły i skrupulatność w raportowaniu implementowanych rozwiązań do różnych systemów, jednak dopóki istnieje wiele

narzędzi wspomagających różne etapy w procesie wytwórczym, odpowiednia komunikacja i wymiana danych jest jedyną metodą umożliwiającą skuteczne śledzenie wymagań.



Rysunek 6.1: Diagram struktury bazy danych systemu Reqmanager

Rozdział 7

Podsumowanie

W poprzednim rozdziale omówiono szczegóły implementacyjne przyjętego rozwiązania oraz krótko omówiono jego wady i zalety z perspektywy autora. Ten rozdział zawiera proponowane ścieżki rozwoju stworzonego oprogramowania oraz podsumowanie całości pracy.

7.1 Plan rozwoju

Proponowane narzędzie wsparcia gromadzenia wymagań, jest jedynie próbą nadania kierunku rozwojowi oprogramowania z tego segmentu. W pełni funkcjonalne narzędzie powinno wspierać obsługę wielu kont użytkowników, śledzenie wymagań, pełną obsługę diagramów (aktorzy, połączenia przypadków użycia, dziedziczenie), wsparcie importu/eksportu danych oraz automatyczny, asynchroniczny zapis obszaru roboczego diagramu.

Na szczególną uwagę zasługuje koncepcja integracji narzędzia z systemem kontroli wersji. Programista pracujący nad kodem źródłowym dotyczącym wybranego wymagania, mógłby w komentarzu zamykającym zadanie umieszczać identyfikator zrealizowanego wymagania. W systemie Reqmanager należałoby zaimplementować podsystem monitorujący wszystkie operacje zapisu do repozytorium, ekstrahować z komentarzy identyfikatory wymagań i odpowiednio oznaczać status ich realizacji na podstawie treści komentarza, umieszczając przy wymaganiu informacje z systemu kontroli wersji (progra-

mista, komentarz, data, status, etc).

Kolejną możliwością rozwoju jest lepsze wsparcie dokumentowania przypadków użycia poprzez wzbogacenie możliwości edycji warunków zajścia a priori i a posteriori. Przykładowo warunki takie mogłyby być zapisywane w postaci kodu źródłowego Groovy lub Java i wykonywane po stronie serwera w odpowiednim momencie. Taka funkcjonalność umożliwiałaby nowatorskie podejście do testowania systemów oraz mogłaby dawać początek kierunku rozwoju bardziej zaawansowanego systemu umożliwiającego dynamiczne generowanie kodu poprzez definicję samych wymagań.

Interesującą koncepcją jest również procesowanie języka naturalnego i automatyczna ekstrakcja wymagań z dokumentów tekstowych. Takie rozwiązanie musiałoby opierać się o zaawansowane biblioteki analizujące tekst oraz implementację inteligentnych algorytmów radzących sobie z ekstrakcją najistotniejszych fragmentów tekstu w celu sformułowania logicznego wymagania.

7.2 Zakończenie

Analiza istniejących rozwiązań, doświadczenia wyniesione z licznych, małych projektów komercyjnych realizowanych przez sektor MŚP oraz badania naukowe w tej dziedzinie, pokazują, że inżynieria wymagań jest w większości organizacji bardzo słabo rozwinięta. Jednak poziom skomplikowania procesorow i sprzętu komputerowego będzie rozwijał się w zatrważającym tempie jeszcze przez wiele lat, a wraz z nimi, potrzeba tworzenia coraz bardziej złożonego, niezawodnego oprogramowania. Dlatego organizacje chcące przetrwać na tak konkurencyjnym rynku jakim jest wytwarzanie oprogramowania, muszą pracować nad ciągłym usprawnianiem procesów, budową odpowiedniej infrastruktury i świadomości wśród pracowników oraz pracować blisko ze środowiskiem naukowym w poszukiwaniu coraz lepszych rozwiązań inżynieryjnych. Jedno narzędzie czy koncepcja nie rozwiąże problemów z jakimi mamy do czynienia w inżynierii wymagań. Jest to dziedzina integrująca bardzo zróżnicowane środowiska - świat naukowy i techniczny, ze światem biznesowym, opartym często na bardzo ciężko mierzalnych parametrach. Z tego

względu niezwykle trudno jest wyznaczyć jasne granice inżynierii wymagań i odpowiednio sformalizować procesy. Być może w przyszłości, zostaną rozwinięte zaawansowane metody formalne pozyskiwania wymagań lub powstaną systemy procesowania języka naturalnego, które dadzą początek znacznyn przemianom i zniwelują zależność od czynnika ludzkiego w procesach wytwórzczych.

Od formalnego wprowadzenia dziedziny inżynierii oprogramowania, jako dziedziny nauki, w roku 1968, na konferencji NATO, wypracowano wiele skutecznych metod wspierającyych procesy związane z pozyskiwaniem, przetwarzaniem i zarządzaniem wymaganiami. Dojrzałość organizacji tworzących oprogramowanie zależy między innymi od poziomu wiedzy związanej z procesami inżynierii wymagań oraz umiejętności ich skutecznego wdrożenia. Narzędzia zastosowane w tych procesach grają istotną, jednak nie pierwszoplanową rolę. Nadal na dzień dzisiejszy kluczem do sukcesu projektu, oprócz odpowiedniego przygotowania merytorycznego jest umiejętność zbudowania odpowiedniego zespołu specjalistów z doświadczeniem i wyposażenie ich w odpowiednie mechanizmy pozwalające na swobodne i efektywne działanie. Organizacje nadal starające się przełożyć bezpośrednio dziedzinę wytwarzania oprogramowania na klasyczne zasady inżynierii systemów zdają się niedostrzegać lub ignorować drastyczne różnice jakie dzielą te dziedziny. Kluczem do sukcesu projektów informatycznych, w pierwszej kolejności, są doskonali, doświadczeni projektanci.

■

Bibliography

[1] J. Arnowitz, M. Arent, and N. Berger. *Effective Prototyping for Software Makers*. Morgan Kaufmann series in interactive technologies. Elsevier Science, 2006. ISBN: 9780080468969.

[2] S. Easterbrook B. Nuseibeh. *Requirements Engineering: A Roadmap*. 2000.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. The XP Series. Addison-Wesley, 2000. ISBN: 9780201616415.

[4] B Boehm. "A spiral model of software development and enhancement". In: *SIGSOFT Softw. Eng. Notes* 11.4 (Aug. 1986), pp. 14–24. ISSN: 0163-5948. DOI: 10.1145/12944.12948.

[5] Barry Boehm. "A view of 20th and 21st century software engineering". In: *Proceedings of the 28th international conference on Software engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 12–29. ISBN: 1-59593-375-1.

[6] Barry W. Boehm, Terence E. Gray, and Thomas Seewaldt. "Prototyping vs. specifying: A multi-project experiment". In: *Proceedings of the 7th international conference on Software engineering*. ICSE '84. Orlando, Florida, United States: IEEE Press, 1984, pp. 473–484. ISBN: 0-8186-0528-6.

[7] F.P. Brooks. *The mythical man-month: essays on software engineering*. Addison-Wesley Pub. Co., 1995. ISBN: 9780201835953. URL: http://books.google.pl/books?id=fUYPAQAAMAAJ.

[8] R. Budde and P. Bacon. *Prototyping: an approach to evolutionary system development*. Springer-Verlag, 1992. ISBN: 9783540543527.

[9] P. Coad and J. Nicola. *Object-Oriented Programming*. Yourdon Press Computing Series v. 1. Yourdon Press, 1993. URL: http://books.google.pl/books?id=nd2WQgAACAAJ.

[10] Institute of Electrical et al. *IEEE Guide to Software Requirements Specifications*. American national standard. IEEE, 1984. URL: http://books.google.pl/books?id=F0A1RAAACAAJ.

[11] Nathan L. Ensmenger. "Letting the "Computer Boys" Take Over: Technology and the Politics of Organizational Transformation". In: *International Review of Social History* 48 (2003), pp. 153–180.

[12] M. Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004.

[13] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[14] Tools Database Working Group. *INCOSE Requirements Management Tools Survey*. International Council on Systems Engineering. 2010. URL: http://www.incose.org/productspubs/products/rmsurvey.aspx.

[15] Jim Highsmith and Martin Fowler. "The Agile Manifesto". In: *Software Development Magazine* 9.8 (2001), pp. 29–30.

[16] USA) IEEE (IEEE. "IEEE Recommended Practice for Software Requirements Specifications: Std 830". In: (1998). internal report.

[17] A. Swartz J. Gruber. *Markdown*. 2004. URL: http://daringfireball.net/projects/markdown/.

[18] Philippe Kruchten. *The Rational Unified Process: An Introduction*. 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321197704.

[19] NASA. *NASA Software Engineering Requirements*. 2009. URL: http://nodis3.gsfc.nasa.gov/npg_img/N_PR_7150_002A_/N_PR_7150_002A_.pdf.

[20] U. Nikkula, J. Sajaniemi, and H. Kälviäinen. "A State-of-the-Practice Survey on Requirements Engineering in Small- and Medium-Sized Enterprises". In: (2000).

[21] R.S. Pressman. *Software engineering: a practitioner's approach*. 5th ed. McGraw-Hill Higher Education, 2010. ISBN: 9780073375977. URL: http://books.google.pl/books?id=y4k_AQAAIAAJ.

[22] Alcides Quispe et al. "Requirements Engineering Practices in Very Small Software Enterprises: A Diagnostic Study". In: *Proceedings of the 2010 XXIX International Conference of the Chilean Computer Science Society*. SCCC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 81–87.

[23] Trygve Reenskaug. *Models - Views - Controllers*. 1979. URL: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf.

[24] *Ruby on Rails vs Grails vs. Spring ROO vs. Spring App*. Stackoverflow. 2010. URL: http://stackoverflow.com/questions/2840890/ruby-on-rails-vs-grails-vs-spring-roo-vs-spring-app.

[25] I. Sommerville. *Software Engineering*. 8th ed. Pearson, 2006.

[26] *The CHAOS Report*. Tech. rep. Standish Group International, 1994.

[27] R. Winston. "Managing the Development of Large Software Systems". In: *Proceedings* (), pp. 1–9.