Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

KATEDRA AUTOMATYKI

PRACA MAGISTERSKA

LESZEK PIĄTEK

SPOŁECZNOŚCIOWY, INTERNETOWY SYSTEM MONITORINGU

PROMOTOR:
dr inż. Sebastian Ernst

Kraków 2012

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

. . . . . .

PODPIS

AGH

University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

DEPARTMENT OF AUTOMATICS

MASTER OF SCIENCE THESIS

LESZEK PIĄTEK

SOCIAL, INTERNET MONITORING SYSTEM

SUPERVISOR:
Sebastian Ernst Ph.D

Krakow 2012

Serdecznie dziękuję promotorowi dr inż. Sebastianowi Ernstowi za wyrozumiałość i cierpliwość, mamie oraz Elizie Tymczuk za słowa, SMS'y i przekazy podprogowe motywujące do „wzięcia się za siebie" i stworzenia niniejszej pracy.

Spis treści

1. Wprowadzenie

W ostatnich latach można zauważyć gwałtowny rozwój rozwój infrastruktury i technologii sieciowych. Coraz więcej gospodarstw domowych dołącza się do „chmury informacyjnej" jaką jest Internet. Komputery bez dostępu do sieci powoli stają się po prostu bezużyteczne. Internet dzięki Wi-Fi oraz technologiom telefonii komórkowej takich jak HSPA lub jej następcy LTE staje się medium bezprzewodowym, dostępnym wszędzie tam gdzie istnieje zasięg sieci komórkowej.

Wszechobecność i dostępność „wielkiej pajęczyny" powoduje, że coraz częściej korzysta się z niej nie tylko za pomocą komputerów czy laptopów. Nowe „gadżety" takie jak tablety, smartphone'y czy netbook'i stają się nowym interfejsem pomiędzy siecią, a użytkownikiem. Szacuje się, że do roku 2015 większość użytkowników Internetu będzie korzystała właśnie z takich urządzeń mobilnych. Wtedy również, 40% światowej populacji (2,7 miliarda osób) będzie już korzystała z dobrodziejstw dostępu do natychmiastowej informacji. [1]

Zwiększenie szybkości dostępu do Internetu umożliwiło stworzenie technologii pozwalających na transmisję strumieniową danych audio/wideo. Takie serwisy jak YouTube, Vimeo czy MetaCafe już od 2005 roku umożliwiają użytkownikom wysyłanie i udostępnianie swoich nagrań online. Niestety żaden z tych serwisów nie umożliwia udostępniania strumienia danych wideo w czasie rzeczywistym. Dopiero w 2007 roku powstał Bambuser oraz Qik. Serwisy te jak i technologie, które wdrożyły pozwalają dzielić się strumieniem audio/video w czasie rzeczywistym, wykorzystując do tego jedynie telefon komórkowy lub kamerę podłączoną do komputera.

Naturalną koleją rzeczy wydaje się wykorzystanie Internetu celem stworzenia gigantycznego systemu monitoringu. Można sobie wyobrazić system do którego podłączono istniejącą infrastrukturę kamer CCTV i udostępniono ich strumień video online, tak aby każdy obywatel mógł monitorować to co się dzieje w jego okolicy. System można by było rozszerzyć o kamery podłączane przez użytkowników lub ich najnowocześniejsze „gadżety" czyli tablety czy smartphone'y. Tak właśnie autor niniejszej pracy wyobraża sobie „Społecznościowy, internetowy system monitoringu".

[1] Dane w oparciu o artykuł [Kim11]

1.1. Cele pracy

Głównym celem pracy jest stworzenie specyfikacji wymagań, analizy systemu określonego jako „Społecznościowy, internetowy system monitoringu".

W niniejszej pracy, na każdym etapie prowadzenia projektu, chciano wykorzystać jak najwięcej metodyk „zwinnych". Autor bazując na pracy starszych kolegów ([JM06]), jak i doświadczeniu w zwinnym specyfikowaniu systemu doktorantów ([MK09]), podpierając się literaturą fachową ([Bec99]) chce zaproponować „zwinną specyfikację i analizę wymagań". W szczególności pokazać jej zalety względem podejścia konwencjonalnego.

Dodatkowo autor pracy chce zaimplementować ograniczony prototyp specyfikowanego systemu wykorzystując przy tym jedynie darmowe technologie z preferencją rozwiązań Open Source.

1.2. Zawartość pracy

Do uzupełnienia/dokończenia później...

1.3. „Zwinna” metodyka tworzenia oprogramowania – „XP”

„Wytwarzając oprogramowanie i pomagając innym w tym zakresie,
odkrywamy lepsze sposoby wykonywania tej pracy.
W wyniku tych doświadczeń przedkładamy:

Ludzi i interakcje ponad procesy i narzędzia.
Działające oprogramowanie ponad obszerną dokumentację.
Współpracę z klientem ponad formalne ustalenia.
Reagowanie na zmiany ponad podążanie za planem.

Doceniamy to, co wymieniono po prawej stronie,
jednak bardziej cenimy to, co po lewej.”

— Kent Beck et al., http://agilemanifesto.org

Przy tworzeniu projektu wchodzącego w skład niniejszej pracy autor poczynił kilka założeń. Większość z nich dotyczy sposobu prowadzenia projektu – dokumentacji, testów, komunikacji w grupie.

W następnym punkcie znajduje się lista tych założeń. Projekt nie spełniający ich, nie będzie mógł być prowadzony „zwinnie”. Listę można traktować jako swojego rodzaju zbiór dobrych praktyk metodyki rozwoju oprogramowania zwanej „Extreme Programming” lub w skrócie „XP” – Programowanie Ekstremalne. Oczywiście ze względu na samodzielny charakter pracy, autor nie mógł sprawdzić wszystkich założeń w praktyce. [2]

1.3.1. Założenia

- klient jest zawsze dostępny – Osoba która wie jak system ma działać od strony użytkownika jest zawsze dostępna dla programistów, forma komunikacji nie jest narzucona, natomiast preferowana jest twarzą w twarz. Osoba ta nie jest potrzebna tylko na początku projektu lecz *przez cały okres* jego tworzenia i rozwoju.
- klient ma zawsze możliwość zmiany – Nie ma rzeczy której nie da się zmienić w systemie, klient może w każdym momencie zmienić wymagania systemu. Grupa projektowa musi reagować na te zmiany.
- projekt jest prowadzony w krótkich iteracjach – jedno do trzy-tygodniowe okresy czasu, po których klient otrzymuje chciane funkcjonalności, wykonania których podjęliśmy się w zadanym okresie czasu.
- testowanie automatyczne – Każdy nowy element systemu musi posiadać napisane testy automatyczne (jednostkowe, funkcjonalne) – system można w każdym momencie automatycznie przetestować. Najlepiej testy pisać przed implementacją nowego elementu.

[2] Głównym źródłem informacji na którym autor pracy bazuje ten rozdział jest książka [Bec99]. Autor tej książki uważany jest za twórcę metodyki „XP”.

- projektujemy, planujemy zawsze *proste minimum* – Nie planujemy na wyrost, określamy minimum, które będzie spełniało wymagania klienta w danej iteracji. Zawsze staramy się trzymać jak najmniejszą złożoność systemu, nawet kosztem dodatkowych zmian.
- ciągła integracja – Poprawki nanoszone są cały czas na istniejący, system, integracja następuje wręcz kilka razy dziennie. Wymagany jest reżim testowy – integrowane jest tylko to co ma napisane testy automatyczne oraz powoduje, że żadne dotychczasowe nie zwracają błędów.
- brak specjalizacji – Żadna osoba z grupy projektowej nie powinna specjalizować się w jakiejś funkcji (programista, architekt, integrator, analityk). Wszyscy biorą czynny udział we wszystkich aspektach projektu.
- grupa prowadząca projekt nie jest zbyt liczna – Grupa w której tworzony jest projekt lub podprojekt nie może być zbyt liczna, ułatwia to komunikację. Każdy większy projekt da się podzielić na szereg mniejszych pod-projektów. Nie ma określonego limitu górnego, jeżeli występują problemy w komunikacji najczęściej grupa projektowa jest zbyt liczna.
- preferowaną formą komunikacji w grupie projektowej jest komunikacja twarzą w twarz – Tylko przy takiej rozmowie uczestnicy projektu nie są rozpraszani przez inne rzeczy i mogą skupić się na zadanym temacie. Taki sposób komunikacji ułatwia wynajdywanie błędów i niejasności we wczesnej fazie projektu.
- wspólne programowanie, projektowanie – Faworyzuje się tworzenie wszystkich elementów systemu w parach. Zapewniając w ten sposób ciągłe badanie jakości kodu, czy też poprawności architektury systemu.
- architektura jest zmienna – Architektura jest czymś co się zmienia wraz z rozwojem projektu i funkcjonalności wymaganej przez klienta. Może się zmienić na każdym etapie projektu.
- kod i testy są dokumentacją – Nie ma prowadzonej dodatkowej dokumentacji implementacyjnej np. dla programistów. Dobrze napisane testy, kod z komentarzami oraz ostatecznie rozmowa z innymi uczestnikami projektu jest najlepszą dokumentacją aktualnego stanu systemu.

1.3.2. Zalety

Metodyka „Extreme Programming" zmienia całkowicie podejście do sposobu tworzenia oprogramowania. Głównie przez umieszczenie klienta, w centrum prowadzonego projektu i jego ciągłe zaangażowanie na całym etapie tworzenia i rozwoju systemu. To on może w dowolnym momencie wprowadzić dowolne zmiany. Zyski jakie daje metodyka zwinna jaką jest „XP" można rozpatrywać na wielu płaszczyznach projektu:

- koszty – Metodyka daje nowe możliwości związane z liczeniem kosztów (per iteracja, per funkcjonalność). Przez krótkie iteracje i działające oprogramowanie łatwiej rozliczać się z klientem, a sam klient chętniej płaci. Długoterminowo unikamy kosztów wynikających z błędów początkowego projektowania, czy też zmian w projekcie, których nie jesteśmy w stanie uniknąć.

- czas – Zarządzanie czasem w krótkich terminach iteracyjnych jest dużo bardziej wydajne i bardziej przewidywalne niż planowanie całego projektu od początku do końca. Brak konkretnej daty zakończenia projektu umożliwia uniknięcie „marszów śmierci", czy przesuwania terminów. Klient sam zarządza tym co ma być zrobione i jakim kosztem czasowym jest to obarczone. Reżim testowy i ciągła integracja powoduje, że oprogramowanie dostarczane jest szybko, i co bardzo ważne dla klienta – jest to oprogramowanie spełniające jego aktualne wymagania.

- jakość – Ciągła integracja i obecność testów powoduje, że pomimo możliwych ciągłych zmian kod w większości przypadków działa, czyli jest wysokiej jakości. Jakość kodu gwarantowana jest również przez programowanie w parach. Zastępuje ono cykliczne sprawdzanie kodu oraz umożliwia uniknięcie problemów związanych z błędną architekturą systemu, które najczęściej pojawiają się bardzo późno.

- zakres – Bardzo łatwo zmienić zakres systemu w trakcie jego tworzenia, wymaga się jedynie ustalenia w której iteracji ma być on zwiększony lub pomniejszony. Jako, że nie mamy ustalonych terminów całości projektu, zmiana zakresu nie jest dla nas

- brak specjalizacji – Ciągła komunikacja werbalna w projekcie zapobiega specjalizacji poszczególnych członków zespołu. Każdy powinien wiedzieć jak najwięcej o systemie. Takie podejście gwarantuje niezależność w przypadku odejścia członka „specjalisty".

- zmniejszanie możliwości porażki projektu – Dostarczanie działającego, funkcjonalnego oprogramowania bardzo szybko powoduje najczęściej zadowolenie klienta. Przekłada się to bezpośrednio na poczucie satysfakcji oraz pozytywną atmosferę wśród członków zespołu. Dzięki temu ludzie są mniej skłonni do zmiany miejsca zatrudnienia ze względu na stres, ciągłe przekraczanie terminów czy brak satysfakcji z wykonywanej pracy.

1.3.3. Wady

Jak każda metodyka, także i ta „zwinna" posiada swoje wady. Warto znać je przed rozpoczęciem projektu, w którym planuje się ją wykorzystać. Główne z nich to:

- ciągła dostępność klienta – Klient lub osoba wyznaczona przez niego, która zna potrzeby systemu musi być dostępna programistom „na zawołanie" i ściśle z nimi współpracować.

- konsekwencja – Decydując się na „XP" nie możemy zrezygnować z założeń (wspomniane szczegółowo w rozdziale `1.3.1 Założenia`). Brak konsekwencji spowoduje w większości przypadków porażkę projektu.

- brak konkretnej daty zakończenia projektu – Podejście iteracyjno-wymaganiowe powoduje, że znany jest jedynie czas zakończenia aktualnej iteracji w skład której wchodzą konkretne karty wymagań i scenariusze. Ciągła możliwość zmiany zakresu i funkcjonalności systemu powoduje, że nie można określić daty zakończenia projektu.

Czasami opisywanej metodyki nie można zastosować ze względu na charakter prowadzonego projektu lub przyzwyczajenia klienta/zleceniodawcy. Główne wskazówki kiedy „XP" **nie nadaje się** jako metodyka prowadzenia projektu:

- duża grupa projektowa – Grupa projektowa „XP" nie powinna być większa niż 10 osób. Więcej osób to większe problemy w komunikacji.
- długie wykonywanie testów – Kompilacja i wykonanie testów automatycznych trwa dłużej niż dzień (24 godziny). Co praktycznie uniemożliwia systematyczną „ciągłą integrację".
- brak środowiska testowego innego niż produkcyjny – W sytuacji kiedy koszty środowiska testowego są bardzo wysokie, klient nie będzie chciał wydawać drugi raz dużych pieniędzy. Może się też zdarzyć, że z innych względów nie będziemy w stanie zagwarantować środowiska testowego, co automatycznie wyklucza testy automatyczne i „ciągłą integrację".

1.3.4. Cykl życia idealnego projektu

Każda metodyka ma swój cykl życia. W tym rozdziale autor chciał przybliżyć idealny cykl życia projektu „XP".

1. **Badanie (Exploration)** Faza w której pobiera się główne wymaganie systemu od klienta, testuje możliwe do wykorzystania technologie, bada się ich wydajność i użyteczność w projekcie. Jeżeli wykorzystywana technologia jest nowa dla zespołu, testuje się jej wykorzystanie na jakimś małym projekcie powiązanym z docelowym systemem.
2. **Planowanie (Planning)** Faza w której grupa projektowa ustala z klientem szczegóły głównych wymagań systemu. Klient określa priorytet, a grupa projektowa czas potrzebny na wykonanie poszczególnych kart wymagań.
3. **Iteracje do pierwszego wydania (Iterations to First Release)** Faza w której w iteracjach jedno do trzy-tygodniowych przygotowuje się system zgodnie z głównymi wymaganiami klienta. Koniec każdej iteracji powinien być „małym świętem" – darmowa pizza, wolny dzień w pracy etc.
4. **Wdrażanie do produkcji (Productionizing)** Faza w której system jest wdrażany do produkcji. Pod koniec tego etapu należy zrobić dużą, huczną imprezę.
5. **Utrzymanie i konserwacja (Maintenance)** Jest to najdłuższa faza projektu „XP". Można by właściwie powiedzieć jego naturalny stan. W tej fazie iteracyjne, wdrażane są szczegółowe wymagania klienta, czy też nowe funkcjonalności odpowiadające jego aktualnym potrzebom.
6. **Zakończenie (Death)** Jest to faza w której projekt uznawany jest za zakończony. Może to być sytuacja idealna, w której klient nie ma potrzeby dalszej zmiany oprogramowania – dostarczono mu produkt idealny. Może to być również mniej przyjemna sytuacja – system nie jest w stanie spełnić nowych wymagań stawianych przez klienta. Tak czy inaczej, dobrym pomysłem jest zorganizowanie „stypy" na której trzeba poruszyć problem, tego: „Co można było zrobić lepiej?" i wyciągnąć na tej podstawie wnioski mogące się przydać w nowym projekcie.

2. Specyfikacja wymagań

Jak wspomniano w rozdziale 1.1 Cele pracy autor chce wykorzystać jak najwięcej metodyk „zwinnych" przy każdym aspekcie tworzenia oprogramowania czyli także w przypadku specyfikacji wymagań. Nie znaleziono do tej pory idealnego rozwiązania uniwersalnego pobierania wymagań od klienta. W większości przypadków trzeba pewnego czasu, aby wypracować wspólny język z klientem i nauczyć się razem z nim współpracować przy akwizycji wymagań.

2.1. „Zwinna" specyfikacja wymagań

Podchodząc „zwinnie" do projektowania systemu od początku należy pobrać od klienta tylko te wymagania, które są konieczne do osiągnięcia celów biznesowych. Wszystkie sprawy poboczne, dodatkowe, należy zostawić na później, aż projekt przejdzie do etapu „Utrzymanie i konserwacja" (więcej w rozdziale 1.3.4 Cykl życia idealnego projektu). Celem „zwinnej" specyfikacji wymagań jest stworzenie wymagań *minimalnego systemu, spełniającego wszystkie podstawowe funkcje biznesowe zdefiniowane przez klienta.*

2.1.1. Narzędzia

Głównymi narzędziami wykorzystywanymi przy „zwinnej" specyfikacji wymagań są:

- user stories (w [MK09] określane jako karty wymagań) – krótkie, ogólnikowe historyjki opisujące jakąś funkcjonalność systemu
- screens (w [MK09] określane jako ekrany) – szkice obrazujące ogólny układ przycisków, formularzy, tabel mogących pojawić się w karcie wymagań
- acceptance tests (w [MK09] określane jako testy akceptacyjne/scenariusze) – opisy słowne prawidłowego zachowania systemu dotyczące kart wymagań, swojego rodzaju scenariusz karty wymagań na bazie których programiści mogą stworzyć testy (o testach akceptacyjnych w „XP" można więcej przeczytać w [Jef00] pod hasłem „Acceptance tests")

Dokumenty te nie mają konkretnej formy, „zwinna" metodyka pozwala dostosować ją do własnych wymagań, które będą działały najlepiej dla wykorzystujących metodykę programistów oraz ich klienta. Karty wymagań, scenariusze oraz ekrany należy spisywać na małych karteczkach jednego formatu. Przygotowując powyższe „dokumenty" należy pamiętać o zasadach:

- prostota – Złożone scenariusze czy karty wymagań należy rozbić na kilka mniejszych, dzielenie musi być zrobione z klientem w formie rozmowy – nigdy przez samego programistę. Karty wymagań powinny być do wykonania w ciągu jednego do dwóch tygodni czasu programisty.

- klient jest autorem kart wymagań jak i testów akceptacyjnych/scenariuszy – Programista może jedynie pokazać jak należy pisać kartę wymagań, tak żeby wypracować z klientem wspólny język, może pomagać w podzieleniu złożonych wymagań na mniejsze. Dopuszcza się sugerowanie możliwych scenariuszy, natomiast sama reakcja systemu musi być już określona przez klienta.

- ekrany powinny mieć swoją sekwencję – Jak istnieje sekwencja ekranów, należy pamiętać o nadaniu numerów odpowiadających kolejności ich występowania.

2.1.2. Cyfryzacja

Na potrzeby tej pracy autor musiał wymyślić sposób na cyfryzację kart wymagań, związanych z nimi testów akceptacyjnych oraz ekranów, tak aby w łatwy sposób dało się je dołączyć do pracy. Dodatkową zaletą formy elektronicznej jest fakt, że ułatwi ona wymianę informacji pomiędzy uczestnikami projektu, pomoże uniknąć sytuacji zagubienia/zniszczenia jakiegoś elementu kart wymagań, ułatwi ich zarządzaniem i przetrzymywaniem np. w wykorzystywanym systemie kontroli wersji.

Wykorzystywanie LaTeX u do składu pracy nakierowało autora pracy na pomysł stworzenia szablonu LaTeX, który można by było wykorzystać celem konwersji kart wymagań i ich elementów analogowych do dokumentu cyfrowego. Łatwe oddzielenie warstwy prezentacji od samej treści oraz możliwość eksportu do formatu PDF umożliwiałoby również tworzenie w prosty sposób ładnego dokumentu dla klienta po spotkaniach na których pobierane są od niego wymagania.

Tak powstał tzw. „package" udostępniający środowisko `userstory`. Szczegółowy opis tego środowiska znajduje się w rozdziale `A LaTeX, środowisko userstory`.

A. LaTeX, środowisko `userstory`

Na potrzeby niniejszej pracy autor musiał wymyślić sposób na cyfryzację kart wymagań, związanych z nimi testów akceptacyjnych oraz ekranów, tak aby w łatwy sposób dało się je dołączyć do pracy. Tak powstał tzw. „package" udostępniający środowisko `userstory`. Więcej o dodatkowych zaletach cyfryzacji można przeczytać w rozdziale `2.1.2 Cyfryzacja`.

A.1. Sposób użycia

Aby skorzystać ze środowiska należy wykonać dwie rzeczy:

- do katalogu dokumentu LaTeXwgrać plik `userstory.sty`
- do generowanego dokumentu dołączyć nagłówek:

Listing A.1:

```
1 \usepackage{userstory}
```

Po takim zabiegu w kodzie dostępne jest środowisko `userstory`. W jego wnętrzu zapisujemy treść karty wymagań, dodatkowo mamy do dyspozycji:

- Środowisko `tests` do definiowania nowych testów akceptacyjnych. Za pomocą komendy `item` możemy zdefiniować w nim kolejny test akceptacyjny.
- Środowisko `questions` do definiowania pytań do klienta, które pojawiły się w „międzyczasie". Za pomocą komendy `item` możemy zdefiniować w nim kolejne pytanie.
- Dwuargumentową komendę `src$file$caption` do umieszczania ekranów w dowolnym miejscu środowiska (przy pytaniach, przy samej karcie wymagań albo przy konkretnym teście akceptacyjnym). Argument **$file** to ścieżka do pliku graficznego, a **$caption** to opis który będzie dodany pod ekranem.

A.2. Przykład użycia

Poniżej przygotowany przykład wykorzystania środowiska `userstory` celem przygotowania karty wymagań „Logowania użytkownika".

Listing A.2: userstory–latex/logowanie–uzytkownika.tex

```
% domyślne formatowania dokumentu, można zmienić
\documentclass[a4paper]{article}
\usepackage{fullpage}
\usepackage[utf8]{inputenc}
\usepackage{polski}
\usepackage[polish]{babel}

% dodanie wymaganego nagłówka środowiska userstory
\usepackage{userstory}

\begin{document}
    \begin{userstory}{Logowanie użytkownika}
        Użytkownik za pomocą formularza może zalogować się do serwisu,
        uzyskując w ten sposób dostęp do jego dodatkowych funkcjonalności.
        \scr{img/us1/1.png}{Formularz logowania.}

        \begin{tests}
            \item{
                Użytkownik tylko po podaniu podaniu loginu oraz pasującego do niego
                    hasła
                zostaje zalogowany.
            }
            \item{
                Użytkownik po podaniu błędnego hasła lub nieistniejącego loginu,
                zostaje o tym poinformowany oraz oferuje mu się od razu formularz
                    przypomnienia hasła.
                \scr{img/us1/2.jpg}{Ekran informacji o błędym haśle lub loginie.}
            }
            \item{
                Użytkownik po zalogowaniu
                widzi zmodyfikowane menu na wszystkich stronach serwisu.
                \scr{img/us1/2.jpg}{Zakładki menu dla zalogowanego użytkownika.}
                \scr{img/us1/3.jpg}{Zakładki menu dla niezalogowanego użytkownika.}
            }
            \item{
                Użytkownik po zalogowaniu
                ma dostęp do panelu.
            }
            \item{
                Użytkownik po zalogowaniu\\*
                może przeglądać profile innych użytkowników.
            }
        \end{tests}
        \begin{questions}
            \item{
                Czy użytkownik po wykonaniu błędnego logowania\\*
                ma być przekierowany na nową stronę, czy pozostawać na tej samej?
            }
```

```
47          \end{questions}
        \end{userstory}
49  \end{document}
```

A.3. Kod środowiska

Poniżej znajduje się kod wykorzystywany w celu definicji środowiska.

Listing A.3: userstory–latex/userstory.sty

```
1   \ProvidesPackage{userstory}

3   % requirements
    \NeedsTeXFormat{LaTeX2e}
5   \RequirePackage{graphicx}
    \RequirePackage{capt-of}
7
    \newenvironment{userstory}[1]{
9       \newcommand{\scr}[2]{
            \begin{center}
11              \includegraphics[width=0.80\textwidth]{##1}
                \captionof{figure}{##2}
13          \end{center}
        }
15      \newenvironment{tests} {
            \subsection[Testy akceptacyjne] {Testy akceptacyjne\footnote{Testy powinny
                być pisane \emph{przez klienta}, jedynie \emph{pod nadzorem} programisty
                !}}
17          \begin{itemize}
                \let\itemi\item
19              \renewcommand{\item}[1]{
                    \begin{itemi}
21                      \textit{\noindent\ignorespaces ####1}
                    \end{itemi}
23              }
        }
25      {
                \let\item\itemi
27          \end{itemize}
        }
29      \newenvironment{questions} {
            \subsection[Pytania do klienta]{Pytania do klienta\footnote{Pytania powinny
                pojawiać się od razu przy rozmowie z klientem tak aby dało się w
                rozmowie wyjaśnić o co dokładnie chodzi. Każde z pytań niedotyczące głó
                wnej karty wymagań systemu powinno wygenerować test akceptacyjny lub
                zmodyfikować istniejący!}}
31          \begin{itemize}
                \let\itemi\item
33              \renewcommand{\item}[1]{
```

```
                \begin{itemi}
35                  \textit{\noindent\ignorespaces ####1}
                \end{itemi}
37          }
    }
39  {
            \let\item\itemi
41      \end{itemize}
    }
43
    \section{Karta wymagań ,,\emph{#1}''}
45  \begin{textit}
        \noindent\ignorespaces
47  }
    {
49  \end{textit}
    \newpage
51  }
```

A.4. Szablon „Główne zadanie systemu”

Poniżej znajduje się kod odpowiedzialny za wygenerowanie szablonu głównego zadania systemu.

Listing A.4: userstory–latex/szablon–glowne–zadanie–systemu.tex

```
1   \documentclass[a4paper]{article}
    \usepackage{fullpage}
3   \usepackage[utf8]{inputenc}
    \usepackage{polski}
5   \usepackage[polish]{babel}

7   \usepackage{userstory}

9   \begin{document}
        \begin{userstory}{Główne zadanie systemu}
11          Tutaj opisać ogólnie główne zadanie systemu, bez realizacji którego nie bę
                dzie można powiedzieć, że system działa. Jak system składa się z kilku
                podsystemów, dla każdego systemu przygotować podobną kartę wymagań.
            \begin{questions}
13              \item{
                    Jakie jest główne zadanie systemu?
15              }
                \item{
17                  Kto jest użytkownikiem systemu?
                }
19              \item{
                    Jakie urządzenia muszą współpracować z systemem?
21              }
                \item{
```

```
23                  Czy jest wymóg użycia konkretnej technologii?
                }
25              \item{
                    Jaka jest wymagana skalowalność systemu?
27              }
                \item{
29                  Czy są jakieś wymagania dotyczące środowiska produkcyjnego w jakim
                        ma działać system?
                }
31              \item{
                    Czy system ma współpracować z innymi systemami lub udostępniać coś
                        innym systemom?
33              }
                \item{
35                  Czy istnieją systemy podobne do tworzonego?
                }
37              \item{
                    Czy są jakieś inne specjalne wymagania, które nie wynikają z funkcji
                         jakie powinien posiadać system (wymagania niefunkcjonalne)?
39              }
            \end{questions}
41      \end{userstory}
\end{document}
```

Bibliografia

[Bec99] K. Beck. *Extreme Programming Explained*. Addison Wesley, 1999.

[Jef00] R. Jeffries. *Extreme Programming Installed*. 2000.

[JM06] M. Jakała and M. Michno. Projekt i implementacja internetowego systemu obsługi konferencji. *AGH University of Science and Technology*, 2006. `http://bit.ly/JakMich06`.

[Kim11] R. Kim. Mobile internet user to eclipse wireline users by 2015, 2011. `http://bit.ly/Kim11`.

[MK09] L. Madeyski and M. Kubiasiak. Zwinna specyfikacja wymagań. *e-Informatyka.pl*, 2009. `http://bit.ly/Mad09`.