ROZDZIAŁ 99

ZAUTOMATYZOWANE TESTY AKCEPTACYJNE DLA APLIKACJI INTERNETOWYCH W PROGRAMOWANIU STEROWANYM TESTAMI

Marian JURECZKO[1,2], Michał MŁYNARSKI[34]

Rozdział prezentuje analizę porównawczą wybranych, darmowych narzędzi umożliwiających tworzenie zautomatyzowanych funkcjonalnych testów akceptacyjnych, czyli Fitnesse, PROVEN!, JFCUnit oraz Selenium. Badana jest możliwość stosowania tych narzędzi do testowania aplikacji internetowych wytwarzanych w procesie opierającym się o podejście programowania przez testy. Aplikacje internetowe są bardzo specyficzną grupą programów. Do ich uruchomienia potrzeba serwera aplikacji. W związku z tym nie każde narzędzie nadaje się do przeprowadzania na nich testów akceptacyjnych. Dodatkowe komplikacje pojawiają się, jeżeli wymagać, aby testy powstały przed napisaniem kodu źródłowego, co ma miejsce w przypadku programowaniu przez testy.

1. WPROWADZENIE

Metodyki zwinne, oraz wywodzące się z nich programowanie przez testy, odgrywają w inżynierii oprogramowania coraz większą rolę. Równocześnie praktyka przemysłowa pokazuje, że automatyzacja testów, która w metodykach zwinnych jest kluczowa [1,3], nie jest zadaniem trywialnym. Automatyzacja testów jest dodatkowo utrudniona jeżeli wymaga się jej wykonania przed implementacją kodu produkcyjnego. Nie sposób używać wtedy narzędzi typu capture&replay, które nagrywają interakcję użytkownika z aplikacją, po to by móc później ją odtworzyć w celu przetestowania aplikacji.

[1] Politechnika Wrocławska, Instytut Informatyki Automatyki i Robotyki; Wybrzeże Wyspiańskiego 27, 50-370 Wrocław; marian.jureczko@pwr.wroc.pl.

[2] Capgemini Polska Sp. z o. o. Software Solution Center, ul. Legnicka 51-53, 54-203 Wrocław

[3] Software Quality Lab, University of Paderborn, Warburger Str. 100, 33098 Paderborn, Niemcy; mmlynarski@s-lab.upb.de

[4] Capgemini sd&m Research, Carl-Wery-Str. 42, 81739 München, Niemcy

Różne typy aplikacji wymagają różnorakiego podejścia do testowania. Bodajże najtrudniejszą w testowaniu grupą programów są aplikacje internetowe. Tego typu aplikacje mają często skomplikowaną architekturę oraz wykorzystują przynajmniej kilka różnych technologii, przez co pisanie testów automatycznych staje się trudniejsze. Testy akceptacyjne powinny być wykonywane w takim środowisku, w jakim docelowo ma działać gotowy system. W przypadku aplikacji internetowych oznacza to, że program musi zostać wdrożony na serwerze aplikacji.

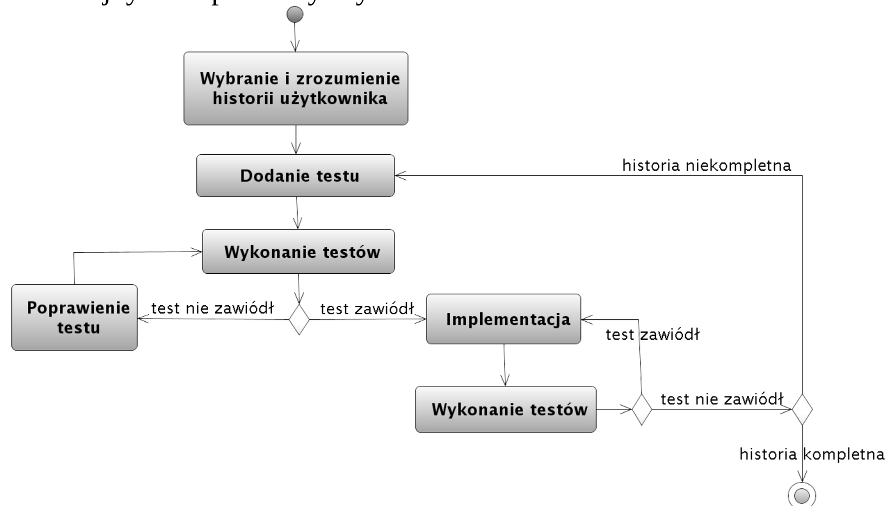
Kluczem do sukcesu testów akceptacyjnych może być wybór odpowiedniego narzędzia. Praca ta analizuje kilka popularnych, darmowych narzędzi do automatyzowania funkcjonalnych testów akceptacyjnych po to, aby udzielić odpowiedzi na pytanie, które z tych narzędzi najlepiej nadaje się do przygotowywania testów akceptacyjnych dla aplikacji internetowej wytwarzanej w podejściu programowania przez testy. Analiza jest przeprowadzana w oparciu o implementację jednej wybranej historii użytkownika realizowanej z wykorzystaniem frameworka Spring. Analizie poddawane są następujące narzędzia: JFCUnit [5], Selenium[9], FitNesse[4], PROVEN![11].

1.1. PROGRAMOWANIE PRZEZ TESTY

Programowanie przez testy (TDD), szczegółowo opisane w [2] bardzo zyskuje na popularności wraz z rozwojem metodyki Programowania Ekstremalnego [1,3]. Jest to podejście diametralnie różne od klasycznego, wywodzącego się z modelu kaskadowego. Odwraca ono bowiem kolejność aktywności w procesie wytwarzania oprogramowania. W TDD zaczyna się od napisania testu, odzwierciedlającego przykład używania systemu lub komponentu. Następnie powstaje kod produkcyjny, który pozwala na zaliczenie tego testu. A dopiero na koniec dokonuje się analizy struktury tego, co powstało w dwóch poprzednich krokach i jeżeli jest taka potrzeba, to strukturę tą poprawia się refaktoryzując.

TDD można stosować na dwóch poziomach. Pierwszy to poziom testów jednostkowych, gdzie testy konstruuje się zazwyczaj przy pomocy narzędzia z rodziny xUnit. Drugi, to poziom testów akceptacyjnych, który to właśnie jest przedmiotem zainteresowania tej pracy. Przykładem stosowania TDD na poziomie testów akceptacyjnych jest metodyka Programowania Ekstremalnego. W oparciu o modele zaproponowane w [7,6] opracowano sekwencję aktywności istotnych z punktu widzenia tej pracy, a pomijając np. refaktoryzację. Zgodnie z tą sekwencją należy najpierw wybrać i zrozumieć jedną z przygotowanych przez klienta historii użytkownika. Następnie należy przygotować i dodać do puli zautomatyzowanych testów akceptacyjnych test sprawdzający scenariusz interakcji użytkownika z systemem. Wykonanie testów po tej operacji nie powinno zakończyć się sukcesem, jeżeli test został przygotowany poprawnie to powinien zawieść, ponieważ implementacja funkcjonalności, którą sprawdza nie została jeszcze przygotowana (red-green bar patterns [2]). Jeżeli test zawiódł

to można przystąpić do implementacji, w ramach której należy dostarczyć funkcjonalności, która pozwoli na zaliczenie testu. Po poprawnej implementacji należy rozstrzygnąć czy historia użytkownika została już w pełni zaimplementowana. Jeżeli istnieją jeszcze jakieś alternatywne scenariusze, których implementacja nie obsługuje, to należy dodać kolejny test i powtórzyć cykl.



Rys. 1. Aktywności w TDD na poziomie testów akceptacyjnych (na podstawie [7,6])

Analizowane dalej narzędzia zostały przebadane między innymi pod kątem możliwości zrealizowania przedstawionej powyżej sekwencji

2. BADANY PRZYPADEK

Narzędzia zostały przebadane na przykładzie historii użytkownika dotyczącej systemu wspomagającego zarządzanie magazynem (na podstawie [10]). System jest realizowany z wykorzystaniem frameworka Spring. W eksperymencie implementowano następującą historię:

„Użytkownik powinien mieć możliwość zwiększenia ceny wybranego towaru przez podanie w procentach wysokości podwyżki. Po udanym przeprowadzeniu podwyżki użytkownikowi powinna zostać wyświetlona informacja o nowej cenie produktu. Jeżeli użytkownik poda ujemną lub równą zero wysokość podwyżki powinna zostać wyświetlona informacja o błędzie.”

2.1. ANALIZOWANE NARZĘDZIA

JFCUnit [5] to rozszerzenie JUnit, które jest dedykowane do testów przeprowadzanych przez interfejs graficzny. Udostępnia silnik, działający w oparciu o pliki XML, który umożliwia nagrywanie akcji użytkownika, a następnie ich odtwarzanie. Oprócz nagrywania JFCUnit udostępnia również ręczne tworzenie nowych testów. Można je zapisywać w języku XML lub bezpośrednio w javie. JFCUnit umożliwia testowanie tylko aplikacji mających swingowy interfejs graficzny.

Selenium [9] działa jako wtyczka do przeglądarki internetowej, która umożliwia nagrywanie akcji użytkownika i utworzenie na ich podstawie zautomatyzowanego testu. Dostępne jest API umożliwiające ręczne pisanie testów w wybranym języku programowania. Można również konwertować do wybranego języka uprzednio nagrany test, dzięki czemu Selenium bardzo dobrze integruje się z niskopoziomowymi środowiskami testowymi, takimi jak JUnit.

FitNesse [4] to narzędzie oparte o Fit, rozbudowany framework szczegółowo opisany w [8]. FitNesse udostępnia kompletną funkcjonalność Fit, zachowując jego rozszerzalną architekturę i dostarcza ulepszonej warstwy prezentacji w postaci edytowalnych stron www – wiki. Typowe testy FitNesse działają z pominięciem warstwy graficznego interfejsu użytkownika odwołując się bezpośrednio do logiki biznesowej testowanego systemu. Testy są zapisywane przy pomocy tabel reprezentujących sekwencje akcji użytkownika. Dzięki licznym rozszerzeniom (nazywanym fixtures) można konstruować również testy odbiegające swym charakterem od powyższego opisu, np. testy interfejsu graficznego czy bazy danych.

PROVEN! [11] to inne narzędzie oparte o Fit. Jego podstawową zaletą jest zestaw rozszerzeń, które uzupełniają braki FitNesse. Między innymi umożliwia tworzenie i sterowanie testami wykorzystującymi silnik Selenium. Rozszerzenia wchodzące w skład PROVEN! mogą być wykorzystywane również w FitNesse. Testy zapisywane są w tabeli, która posiada pięć kolumn opisujących: rozkaz do wykonania (np. enter, press, select albo check), jego kontekst, rodzaj konektora, który ma zostać użyty, potrzebne argumenty oraz ewentualne komentarze. Bardzo ważną rolę odgrywają tutaj konektory, za pomocą których PROVEN! komunikuje się z bazą danych, przeglądarką albo dowolnymi innymi lokalnymi lub rozproszonymi systemami. Jeżeli zestaw standardowych konektorów nie wystarcza do testowania, istnieje możliwość zaimplementowania własnego konektora. Czynność ta nie jest skomplikowana, a potrzebna dokumentacja jest dołączona do narzędzia. PROVEN! został opracowany przez Capgemini sd&m i jest dostępne na licencji CPL.

3. EKSPERYMENT

Przeprowadzony eksperyment polegał na wykonaniu implementacji historii użytkownika opisanej w podrozdziale 2. Implementacja była wykonywana zgodnie z paradygmatami TDD. Implementację wykonywano w oparciu o testy przygotowane w każdym z analizowanych narzędzi.

3.1. KRYTERIA OCENY BADANYCH NARZĘDZI

W celu oceny przydatności narzędzia sprawdzano czy za jego pomocą można zrealizować następujące cele:

- pisanie testów przed implementacją, czyli w procesie zgodnym z przedstawionym na rys. 1,
- testowanie aplikacji internetowej,
- uruchamianie testów w środowisku jak najbliższym temu, w którym systemu będzie używać klient,
- sprawdzanie i modyfikowanie stanu bazy danych,
- sprawdzanie komunikatów o błędach,
- tworzenie testów czytelnych dla klienta,
- udział klienta w pisaniu automatycznych testów.

3.2. TESTY AKCEPTACYJNE W WYBRANYCH NARZĘDZIACH

Poniżej przedstawiono testy sprawdzające poprawność implementacji opisanej powyżej historii użytkownika. Przy pomocy każdego z badanych narzędzi przygotowano testy sprawdzające tą samą funkcjonalność.

3.2.1. Testy JFCUnit

```
public class PriceIncreaseTest extends JFCTestCase {
…
@Override
protected void setUp() throws Exception {
 super.setUp();
 setHelper( new JFCTestHelper() );
 SwingApp są = SwingApp.getApplication();
 swingView = new SwingView( sa );
 sa.show(swingView);
```

```
 getComponents();
}
protected void getComponents() {
 finder=new NamedComponentFinder(JComponent.class,"PrcIn",false);
 mPriceIncreaseButton = (JButton) finder.find();
 finder.setName("Percentage");
 mPercentageField = (JTextField)finder.find();
 finder.setName("PriceField");
 mPriceField = (JTextField)finder.find();
 finder.setName("MasterTable");
 mMasterTable = (JTable)finder.find();
 finder.setName("ErrorLabel");
 mErrorLabel = (JLabel)finder.find();
}
public void testPositive() {
 getHelper().sendKeyAction( new KeyEventData( this,
mPercentageField, KeyEvent.VK_A, KeyEvent.CTRL_DOWN_MASK, 100));
 getHelper().sendKeyAction( new KeyEventData( this,
mPercentageField, KeyEvent.VK_DELETE ));
 getHelper().sendString( new StringEventData( this,
mPercentageField, "10"));
 getHelper().enterClickAndLeave( new
JTableMouseEventData(this, mMasterTable, 0, 1, 1) );
 getHelper().enterClickAndLeave( new MouseEventData( this,
mPriceIncreaseButton ) );
 assertEquals( "6.36", mPriceField.getText() );
}
public void testNegative() {
 getHelper().sendKeyAction( new KeyEventData( this,
mPercentageField, KeyEvent.VK_A, KeyEvent.CTRL_DOWN_MASK, 100 ));
 getHelper().sendKeyAction( new KeyEventData( this,
mPercentageField, KeyEvent.VK_DELETE ));
 getHelper().sendString( new StringEventData( this,
mPercentageField, "-15" ) );
 getHelper().enterClickAndLeave( new
JTableMouseEventData(this, mMasterTable, 0, 1, 1) );
 getHelper().enterClickAndLeave( new
JTableMouseEventData(this, mMasterTable, 0, 2, 1) );
 getHelper().enterClickAndLeave( new MouseEventData( this,
mPriceIncreaseButton ) );
 assertEquals("Must be greater than 0!", mErrorLabel.getText());
```

```
} }
```

Niestety JFCUnit jest narzędziem o bardzo wąskiej arenie potencjalnych zastosowań. W szczególności nie umożliwia testowania aplikacji internetowych o interfejsie w postaci strony www. Na potrzeby eksperymentu trzeba było stworzyć testowaną aplikacji z interfejsem „swingowym”. Sytuacja taka uniemożliwia testowanie aplikacji w środowisku klienckim. Testy można było napisać przed przystąpieniem do implementacji, aczkolwiek było to bardzo niewygodne z uwagi na mnogość szczegółów jakie trzeba było w teście ustalić (np. nazwy kontrolek interfejsu graficznego). Podejście TDD wymusiło również rezygnację z „nagrywania – odtwarzania” testów. JFCUnit nie wspiera komunikacji z bazą danych. Nic natomiast nie stoi na przeszkodzie, aby w testach osadzić kod JDBC (testy są pisane w javie). Testy JFCUnit są nieczytelne dla klienta bez wykształcenia informatycznego, co wyraźnie widać na zamieszczonym powyżej fragmencie kodu (zapisanie testów w XML'u zamiast w javie nie poprawia ich czytelności). Wykluczyć również należy potencjalny udział klienta w tworzeniu tych testów.

3.2.2. Testy Selenium

```
public class PriceIncreaseTest extends SeleneseTestCase{
 @Override
 public void setUp() throws Exception {
  setUp("http://localhost:8080/springapp", "*firefox3");
 }
 public void testPossitive() throws Exception {
  selenium.open("/springapp");
  selenium.click("link=Increase Prices");
  selenium.waitForPageToLoad("30000");
  selenium.type("percentage", "10");
  selenium.select("prod", "label=Lamp");
  selenium.click("//input[@value='Execute']");
  selenium.waitForPageToLoad("30000");
  if( !selenium.isTextPresent("Lamp $6.36") )
     throw new SeleniumException("AssertError!");
  }
 public void testNegative() throws Exception {
  selenium.open("/springapp");
  selenium.click("link=Increase Prices");
  selenium.waitForPageToLoad("30000");
  selenium.type("percentage", "-15");
  selenium.select("prod", "label=Table");
  selenium.click("//input[@value='Execute']");
```

```
        selenium.waitForPageToLoad("30000");
        if( !selenium.isTextPresent("Must be higher than 0!") )
           throw new SeleniumException("Error!");
        } } }
```

Selenium umożliwia pisanie testów przed implementacją, jednak trzeba wtedy zrezygnować z będącej najmocniejszą stroną narzędzia funkcjonalności polegającej na nagrywaniu i odtwarzaniu testów. Selenium jest narzędziem dedykowanym do testowania aplikacji internetowych, więc bardzo dobrze sobie radzi z aplikacjami tego typu. W szczególności pozwala na uruchamianie testowanej aplikacji w dowolnym środowisku, więc również w zbliżonym do klienckiego. Selenium nie wspiera komunikacji z bazą danych jednak, podobnie jak w JFCUnit, nie jest problemem osadzenie kodu JDBC w kodzie testów. Testy są, dla klienta, na granicy czytelności. Są zapisane w języku programowania (w podanym powyżej przykładzie w javie), ale są na tyle czytelne, że można je zrozumieć bez posiadania umiejętności programowania. Tworzenie nowych testów wydaje się jednak zadaniem zbyt trudnym dla osoby nie potrafiącej programować.

3.2.3. Testy FitNesse

```
public class PriceIncreaseFixture extends ColumnFixture {
 public String product;
 public int percentage;

 public Double increasePrice(){
  ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("web/appContext.xml");
  BeanFactory bFctry = ctx.getAutowireCapableBeanFactory();
  ProductManager pm =
(ProductManager)beanFactory.getBean("productManager");
  pm.increasePrice(percentage, product);
  List<Product> products = pm.getProducts();
  Iterator<Product> itr = products.iterator();
  while( itr.hasNext() ){
     Product prod = itr.next();
     if( prod.getDescription().equalsIgnoreCase(product) )
```

```
            return new Double(prod.getPrice());
        }
        return null;
       } }
```

Pisanie testów w FitNesse przed implementacją kodu produkcyjnego nie jest problematyczne. Można również bez przeszkód testować aplikacje internetowe, są one jednak uruchamiane w słabo konfigurowalnym środowisku testowym, czyli poza kontenerem aplikacji, z którego korzystałby klient. Fitnesse wspiera testowanie baz danych rozszerzeniem JdbcFixture, które niestety nie jest już rozwijane, a do jego obsługi wymagana jest znajomość SQL'a, co może stanowić problem w komunikacji z klientem. W testach Fitnesse można również bez przeszkód osadzać kod JDBC i tą drogą komunikować się z bazą danych. Testy Fitnesse składają się z dwóch części. Pierwsza, zawierająca specyfikację właściwego testu, pokazana na rys. 2, jest czytelna dla klienta i może również być z jego udziałem tworzona. Część ta jednak musi zostać uzupełniona przez programistę kodem, który pozwoli na wykonanie testu (przykładowy kod znajduje się powyżej). Jest to rozsądny kompromis, który pozwala na udział klienta w specyfikowaniu scenariuszy testowych, a równocześnie umożliwia testowanie dowolnie skomplikowanych technicznie aspektów. Przedstawiony powyżej test ilustruje typowe dla Fitnesse podejście, czyli bezpośrednie testowanie logiki biznesowej z pominięciem warstwy prezentacji. Tym sposobem można uniezależnić testy od modyfikacji interfejsu graficznego, ale jak pokazał omawiany przypadek, nie można przez to m. in. zweryfikować poprawności komunikatu z błędem. Istnieją rozszerzenia pozwalające na testowanie w FitNesse interfejsu graficznego aplikacji internetowych, takie jak konektory z omawianego dalej PROVEN!. Nie zostały one tu zaprezentowane, ponieważ uzyskano by dla nich wyniki podobne jak dla PROVEN! też czy Selenium.

PriceIncreaseFixture		
product	percentage	increasePrice?
Lamp	10	6.36

New price should be 10% greater.

PriceIncreaseFixture		
product	percentage	increasePrice?
Table	-15	64.0

The price shouldn't be changed.

Rys. 2. Testy FitNesse

3.2.4. Testy PROVEN!

Testy zostały wyspecyfikowane za pomocą pseudojęzyka w tabeli HTML. Do ich specyfikowania nie było potrzeba implementacji, co więcej scenariusze testowe eksponują logikę aplikacji, a tym samym ułatwiają pracę w podejściu TDD. Testy można wykonywać w dowolnym środowisku, a w szczególności w takim, w jakim testowanej aplikacji będzie używał klient. Testy mogą być specyfikowane nie tylko przez programistę, ale także klienta. Dodatkowym atutem PROVEN! jest rozwiązanie dla typowych problemów znanych z narzędzi typu capture&replay, jak np. dynamicznie zmieniający się layout strony oraz identyfikatory obiektów. Możliwe jest także testowanie aplikacji asynchronicznych typu AJAX. PROVEN! wspiera również testowanie baz danych. Można w specjalnie przygotowanym języku, który powinien być zrozumiały dla klienta, komunikować się z bazą danych. Modyfikowanie i sprawdzanie stanu bazy danych jest możliwe za pomocą konektora DBKonnektor i standardowych rozkazów (select, enter, check).

Test case for Use case „Increase prices“

Testfall	Increase Prices		
	1. Positive test case		
start	Springapp	seleniumkonnektor	http://localhost:8080/springapp
press	Springapp	Increase Prices	wait=30000
press	Springapp	Prod	
select	Springapp	Prod[Lamp]	
enter	Springapp	percentage	10
press	Springapp	Execute	
contains	Springapp		Lamp $6.36
	2. Negative test case		
Start	Springapp	seleniumkonnektor	http://localhost:8080/springapp
press	Springapp	Increase Prices	wait=30000
press	Springapp	prod	
select	Springapp	prod[Table]	
enter	Springapp	percentage	-15
press	Springapp	Execute	
contains	Springapp		You have to specify a percentage higher than 0!

Rys. 3. Testy PROVEN!

4. WNIOSKI

Przeprowadzony eksperyment wykazał, że podejście TDD nie wymusza stosowania konkretnego narzędzia do automatyzowania testów akceptacyjnych, aczkolwiek można sobie pracę nieco utrudnić stosując narzędzia testujące przez interfejs graficzny. Wybór narzędzia implikują natomiast pozostałe, brane pod uwagę kryteria. JFCUnit nie nadaje się do testowania aplikacji internetowych. W FitNesse można natrafić na problemy z uruchamianiem aplikacji w środowisku klienckim, aczkolwiek możemy wykorzystać w nim rozszerzenie dostarczane przez PROVEN!, a umożliwiające integrację z Selenium i tym samym pozwalające na testowanie przez interfejs graficzny aplikacji uruchomionej w dowolnym środowisku. Problematyczna może być również komunikacja z bazą danych. W funkcjonalność taką wyposażone są jedynie FitNesse i PROVEN!, a jeżeli wymagać, aby ta komunikacja była czytelna dla klienta, to użyć można tylko tego drugiego. Dostarczany przez FitNesse JDBCFixture wymaga znajomości języka SQL. Czytelność testów dla osoby nie posiadającej wykształcenia informatycznego, w przypadku niektórych narzędzi, również nie jest doskonała. W przy-

padku PROVEN! oraz FitNesse nie powinno to stanowić problemu, nieco gorzej jest z testami przygotowanymi w Selenium, a zupełnie fatalnie z tymi z JFCUnit.

W rozdziale przeanalizowano cztery, istotnie różniące się od siebie, narzędzia do automatyzowania testów akceptacyjnych. Przeprowadzona analiza pozwoliła na wskazanie wad i zalet tych narzędzi w procesie wytwarzania aplikacji internetowych z wykorzystaniem podejścia TDD. Rozdział ten jest również pierwszą opublikowaną w Polsce pracą, w której przedstawione zostało, opracowane w laboratoriach Capgemini-sd&m, narzędzie do automatyzowania testów akceptacyjnych – PROVEN!.

LITERATURA DO ROZDZIAŁU

[1] Astels, D., Miller, G., Novak, M.: Practical guide to extreme programing. Prentice Hall PTR, 2000.

[2] Beck K.: Test Driven Development by Example. Addison-Wesley Professional, 2002.

[3] Beck, K. Anders, C.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 2004.

[4] FitNesse: http://fitnesse.org

[5] JFCUnit: http://jfcunit.sourceforge.net

[6] Jureczko, M., Magott J.: High-level Petri net model for XP methodology. In: Software Engineering in Progress, Nakom, 2007.

[7] Madeyski L.: Test-First Programming Experimentation and Meta-Analysis, jeszcze nie opublikowana.

[8] Mugridge R., Cunningham W.: Fit for Developing Software: Framework for Integrated Tests. Prentice Hall PTR, 2005.

[9] Selenium: http://seleniumhq.org

[10] SpringMVConNetBeansGlassFish: http://wiki.netbeans.org/SpringMVConNetBeansGlassFish

[11] PROVEN!: http://sww.sdm.de/org/sr/produkte/proven (Intranet Capgemini sd&m)