STUDIA INFORMATICA 2010
Volume 31 Number 2B (90)

Dariusz R. AUGUSTYN
Politechnika Śląska, Instytut Informatyki

ROZSZERZENIA MODUŁU TESTÓW JEDNOSTKOWYCH DLA WIELOWARSTWOWYCH APLIKACJI ZBUDOWANYCH NA BAZIE SZABLONU PUREMVC I TECHNOLOGII ADOBE® FLEX™

Streszczenie. Artykuł prezentuje metody i narzędzia automatyzacji testowania wielowarstwowych systemów informatycznych wykonanych w technologii Adobe-/Flex. W pracy rozważana jest architektura aplikacji zbudowana w oparciu o popularny szablon PureMVC (rozbudowany wariant metawzorca Model-Widok-Kontroler). Artykuł przedstawia sposoby automatyzacji testowania systemu od strony funkcjonalności GUI, jak i serwera aplikacji. W szczególności, artykuł przedstawia nowo stworzone, użyteczne rozszerzenia modułu testującego, pozwalające na sprawdzanie przepływu notyfikacji – specyficznego mechanizmu komunikacji pomiędzy elementami aplikacji, zastosowanego w szablonie PureMVC. Rozszerzenia dotyczą również obsługi notyfikacji przy wykorzystaniu obiektów zastępczych (mock objects) w procesie testowania jednostkowego.

Słowa kluczowe: automatyzacja procesu testowania, testy jednostkowe, wielowarstwowa architektura oprogramowania, technologia Adobe Flex, szablon PureMVC

EXTENSIONS OF THE UNIT TESTING MODULE FOR MULTILAYER SYSTEMS BASED OF PUREMVC FRAMEWORK AND ADOBE® FLEX™ TECHNOLOGY

Summary. The paper presents methods and software tools for testing automation of multilayer systems developed in Adobe/Flex technology. Architecture of tested applications is based on popular PureMVC Framework (Pure Model View Controller). This paper shows approaches to an automation of testing either GUI functionalities or application server ones. In particular a new useful extension of a testing module is presented. It supports testing of notification's flows – the mechanism of communication among elements of tested application, specific for applied PureMVC Framework. The extension supports also notification handling in a unit testing process based on mock objects.

Keywords: automation of testing process, unit tests, multilayer software architecture, Adobe Flex technology, PureMVC Framework

1. Wprowadzenie

Automatyzacja procesu testowania systemów informatycznych jest dziedziną stale rozwijającą się, z uwagi na to, że technologie wytwórcze ulegają ciągłym zmianom. Niniejsza praca dotyczy metod i narzędzi testowania wielowarstwowych systemów wykonanych w nowej, popularnej technologii Adobe Flex. Technologia ta umożliwia tworzenie systemów informatycznych zaliczanych do tzw. klasy RIA (ang. Rich Internet Application), pozwalając na m.in. na tworzenie aplikacji internetowych z „bogatym", efektownym, ale i ergonomicznym interfejsem użytkownika. Architektura takich systemów może być oparta na popularnym szablonie PureMVC (ang. Pure Model-View-Controller, czyli Model-Widok-Kontroler), zapewniającym elastyczność, modułowość, luźne powiązanie elementów składowych oraz ułatwienia w zakresie testowalności.

W rozdziałach 2 i 4 artykuł opisuje ogólnie i na przykładzie architekturę aplikacji RIA, bazujących na szablonie PureMVC. Rozdział 3.1 krótko omawia metody i narzędzia automatycznego testowania systemu od strony funkcjonalności serwera aplikacji. Rozdziały 3.2 i 3.3 opisują metody i narzędzia automatycznego testowania systemu od strony interfejsu użytkownika. Rozdział 5 omawia zaproponowane rozszerzenia, pozwalające na budowę testów jednostkowych, dostosowanych do użytego w testowanych aplikacjach szablonu PureMVC.

Głównymi elementami pracy są opisy zaproponowanych, nowych rozszerzeń modułu testującego pozwalających na testowanie elementów aplikacji wynikających z użytego szablonu PureMVC. Pierwsze z rozszerzeń dotyczy możliwości logowania i sprawdzania wystąpień notyfikacji. Model komunikacji zastosowany w szablonie PureMVC zakłada przepływ notyfikacji pomiędzy elementami aplikacji. Stąd moduł testujący został rozbudowany o wsparcie śledzenia przepływu notyfikacji w ramach testów jednostkowych. Drugie rozszerzenie dotyczy rozbudowy modułu o możliwość testowania aplikacji bazujących na PureMVC w oparciu o obiekty zastępcze – atrapy (ang. mock object). Wykorzystanie obiektów zastępczych (zamiast oryginalnych) sprowadza się do tzw. nagrywania oczekiwań w stosunku do obiektu zastępczego (np. jak obiekt ma odpowiedzieć na wywołanie metody z podanymi parametrami), a następnie użycie (nagranego) obiektu (obiektów) zastępczego w ramach właściwego testu jednostkowego. Takie podejście jest znane i wykorzystywane, ale standardowe moduły wpierające testowanie na atrapach nie wspierają testowania przepływu notyfikacji wynikających z użycia PureMVC. Zaproponowane rozszerzenie pozwala na nagrywanie oczekiwań

w stosunku do obiektów zastępczych również w zakresie generowanych przez nie notyfikacji, pozwalając w ten sposób na pełniejsze testowanie aplikacji bazujących na PureMVC.

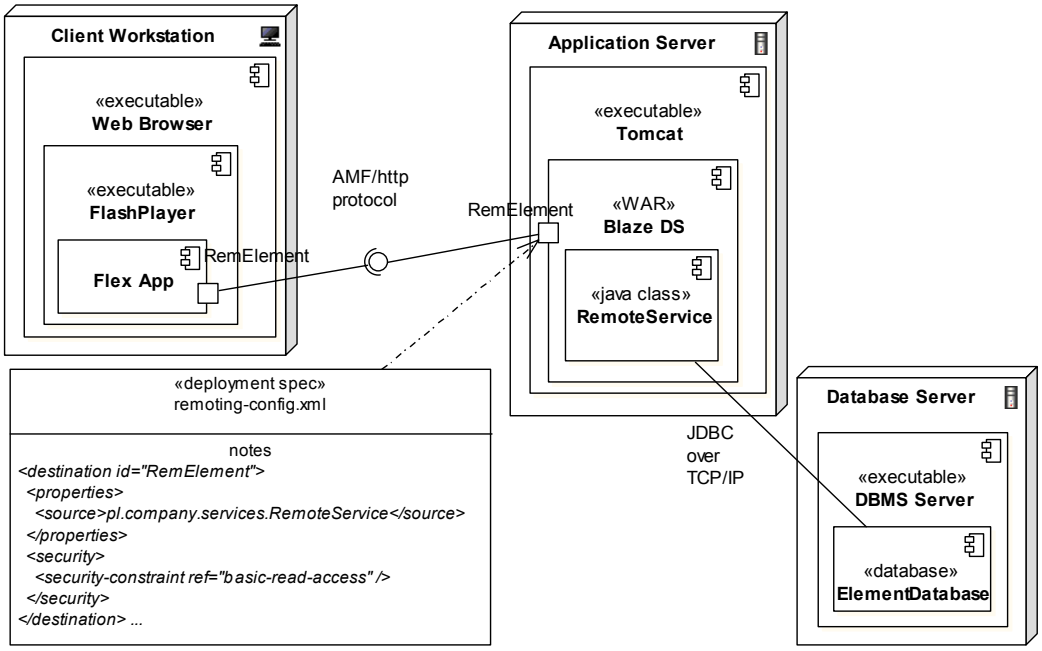
2. Architektura wielowarstwowego systemu opartego na technologii Adobe Flex

Architektura przykładowego systemu informatycznego opartego na technologii firmy Adobe pokazana została na UML-owym diagramie wdrożenia z rys. 1.

W przedstawionej architekturze wielowarstwowego systemu komponent aplikacji klienta zbudowany został w oparciu o technologię Adobe Flex/Flash [14, 15]. Logika przetwarzania wyrażona została w języku Action Script (AS [17]), natomiast interfejs użytkownika opisany jest językiem MXML (dialekt języka bazującego na XML, pozwalającego m.in. na deklaratywne określenie budowy widoków, kontrolek etc. [18]). Architektura aplikacji klienta bazuje na szablonie PureMVC [19] (ang. Pure Model View Controller), pozwalającym na luźne powiązanie elementów aplikacji odpowiedzialnych za prezentację, przetwarzanie, pobranie i utrwalanie danych. Z tego powodu można ją nazwać wielowarstwową mikroarchitekturą aplikacji klienta.

W części serwerowej systemu wykorzystany jest moduł Blaze DS [13] (ang. Blaze Data Services), udostępniany w ramach serwera aplikacji, np. Tomcat/JBoss. Blaze DS udostępnia usługę zdalnych obiektów (ang. remote objects), pozwalając na uruchamianie metod obiektów zaimplementowanych w języku Java, osadzonych w serwerze aplikacji. Blaze DS jest servletem wchodzącym w skład infrastruktury proponowanej przez firmę Adobe, udostępniającym usługi poprzez binarny protokół AMF [16] (ang. Action Message Format). Interfejs udostępnianych, zdalnych obiektów opisywany jest deklaratywnie w pliku *remoting-config.xml*, będącym elementem konfiguracji modułu BlazeDS (rys. 1). Przetwarzanie po stronie serwera aplikacyjnego jest również zorganizowane w oparciu o wielowarstwową koncepcję architektury, zakładającą co najmniej użycie warstwy fasady (spełniającej wymagania wynikające z interfejsu żądań aplikacji klienta, a dokładniej, żądań obiektów Proxy z modelu PureMVC) i warstwy DAO (ang. Data Access Object) dla „uniwersalnego” dostępu metod fasady do danych.

Dane przechowywane i zarządzane są przez serwer relacyjnej bazy danych, którego usługi są dostępne dla warstwy DAO poprzez interfejs JDBC.



Rys. 1. Architektura wielowarstwowego systemu informatycznego bazującego na technologii Adobe Flash/Flex
Fig. 1. Architecture of multilayer system based on Adobe Flash/Flex technology

3. Narzędzia automatyzacji testowania systemu wielowarstwowego

Separacja warstw sytemu informatycznego pozwala na niezależne, automatyczne testowanie komponentów składowych. Podnosi to skuteczność testowania i jego efektywność (poprzez możliwość określania zakresu funkcjonalnego testowanego oprogramowania w zależności od warstwy).

3.1. Automatyzacja testowania funkcjonalności serwerowej z użyciem jUnit, jMock

Niewizualna funkcjonalność serwerowa systemu, implementowana w języku Java, może być testowana z wykorzystaniem modułu wspierającego testowanie jednostkowe – jUnit [1]. Niezależnie mogą być przetestowane proste metody klas obsługi DAO, a następnie metody klas fasady. Aby spełnić założenia o niezależności i bezstanowości pojedynczych metod testowych, co jest utrudnione w przypadku testowanej funkcjonalności trwale modyfikującej bazę danych [6], można posłużyć się przy testowaniu fasady modułem jMock [2] (czy EasyMock [3]). Dzięki temu, metody fasady zamiast operować na „rzeczywistych" obiektach DAO, mogą w trakcie testowania funkcjonować w oparciu o ich uproszczone odpowiedniki, tzw. obiekty atrapy (ang. mock object). Obiekty atrapy operują w pamięci operacyjnej i nie korzystają z bazy danych. Opisane w niniejszym podrozdziale metody testowania dotyczą komponentu *RemoteService* z rys. 1.

3.2. Automatyzacja testowania interfejsu użytkownika z użyciem narzędzi Selenium

Testowanie systemu od strony interfejsu użytkownika, tzw. testowanie funkcjonalne, może być zrealizowane np. w oparciu o moduły systemu Selenium [7]. W najprostszej konfiguracji testy GUI (ang. graphical user interface) aplikacji klienta, uruchamianej w środowisku przeglądarki internetowej mogą być automatycznie uruchamiane dzięki programowi Selenium IDE, zaimplementowanemu w postaci wtyczki do przeglądarki Mozilla FireFox (ang. *Plug-in*). Narzędzia Selenium pozwalają na testowanie aplikacji WWW wykonanych w różnych technologiach, w szczególności takich, gdzie aplikacja Flash/Flex wywoływana jest z poziomu przeglądarki. Niestety, dla technologii Flash/Flex nie zaimplementowano opcji „nagrywania" testów – komendy testujące muszą być ręcznie wpisane przez osobę tworząca przypadek testowy. Testy opisujące sekwencje zdarzeń dotyczących elementów GUI mogą być potem uruchamiane na żądanie użytkownika z poziomu okna Selenium IDE. Wykorzystanie Selenium IDE wymaga rozszerzenia projektu aplikacji Flex/Flash o dodatkową bibliotekę SeleniumFlexAPI.swc (na potrzeby wyeksponowania komponentów GUI). Koncepcja sterowania aplikacją Flash/Flex poprzez Selenium IDE opiera się na mechanizmie *ExternalInterface API* [20], stąd wraz z ww. wtyczką powinien być zainstalowany zbiór funkcji javascritpt (na potrzeby komunikacji aplikacji javascript i aplikacji Flash/Flex).

Wykorzystanie dodatkowego modułu Selenium RC (ang. remote control), działającego w tzw. infrastrukturze serwerowej [8], pozwala na testowanie z dowolną przeglądarką (nie tylko Mozilla FireFox). Dodatkowy komponent w postaci niezależnego serwera TCP, (działający jako http proxy) odbiera żądania od klienta (ang. client-driver) w postaci komend opisujących kroki testu. Serwer Selenium RC uruchamia niezależne dwa okna przeglądarek (dowolny typ przeglądarki): dla sterowania i śledzenia przebiegu testów (poprzez kod javasctiptowy modułu Selenium-Core) oraz dla właściwej testowanej aplikacji Flash/Flex.

Testy wyrażone są poprzez pojedyncze przypadki testowe (ang. test case) i zgrupowane w zestawy przypadków (ang. test suite). Testy mogą być wyeksportowane np. do postaci kodu języka Java czy C#, w celu użycia ich wraz z modułami jUnit czy NUnit [4, 5, 6]. Dzięki tej opcji, wykonanie skryptów testujących GUI, wyrażonych w języku Java, może być zautomatyzowane tak samo, jak wykonanie wspomnianych wcześniej testów niewizualnych funkcji przetwarzania serwerowego. Uruchomiony moduł jUnit, „zasilony" testami GUI wyrażonymi w języku Java, pełni rolę sterującego elementu client-driver w omawianej infrastrukturze Selenium RC [8, 9]. Takie podejście (użycie jUnit) pozwala na uruchamianie zarówno testów GUI, jak i testów niewizualnych z poziomu środowiska wytwórczego (np. Eclipse) lub/i w ramach oprogramowania ciągłej integracji (ang. continuous integration).

3.3. Automatyzacja testowania interfejsu użytkownika z użyciem Flex Monkey

Alternatywna do opisanej powyżej, metoda testowania funkcjonalnego aplikacji Flash/Flex zakłada użycie modułu FlexMonkey [12]. Narzędzie FlexMonkey dedykowane jest dla technologii Flex/Flash i pozwala na automatyzację testów GUI zarówno aplikacji uruchamianych samodzielnie (technologia Adobe AIR [10, 11]), jak i tych „zanurzonych" w przeglądarkach webowych. Narzędzie FlexMonkey bazuje na mechanizmie Flex Automation [21].

Możliwe jest zarówno nagrywanie ciągu kroków testowych w ramach interakcji użytkownika z aplikacją, jak i „ręczne" wpisywanie komend-kroków oraz późniejsze odtwarzanie przypadków testowych lub całych zestawów testowych.

Aplikacja nagrywająco-odtwarzająca – FlexMonkey Console – może sterować bezpośrednio testowaną aplikacją AIR. Może też sterować testowaną aplikacją (plik SWF), zanurzoną w stronie HTML, uruchomioną w niezależnym oknie przeglądarki. Oba tryby testowania (w których następuje załadowanie testowanej aplikacji przez program FlexMonkey Console) możliwe są do użycia bez konieczności rekompilacji projektu testowanej aplikacji Flash/Flex.

Za pomocą narzędzia FlexMonkey można wygenerować test w postaci kodu źródłowego w języku AS. Zestaw testowy jest pakietem, przypadek testowy jest klasą, pojedynczy test jest wywołaniem bezparametrowej metody.

Rozszerzenie projektu, w którym znajduje się testowana aplikacja i wygenerowane testujące kody źródłowe o elementy takie jak:

- biblioteka *monkey-ant-task.jar* (implementacja wywołania „silnika testującego" z poziomu skryptu Ant) ,
- aplikacja testująca uruchamiająca zestaw testów z poziomu przeglądarki –*MonkeyTestLauncher* (elementy: *MonkeyTestLauncher.swf* i M*onkeyTestLauncher.html*),

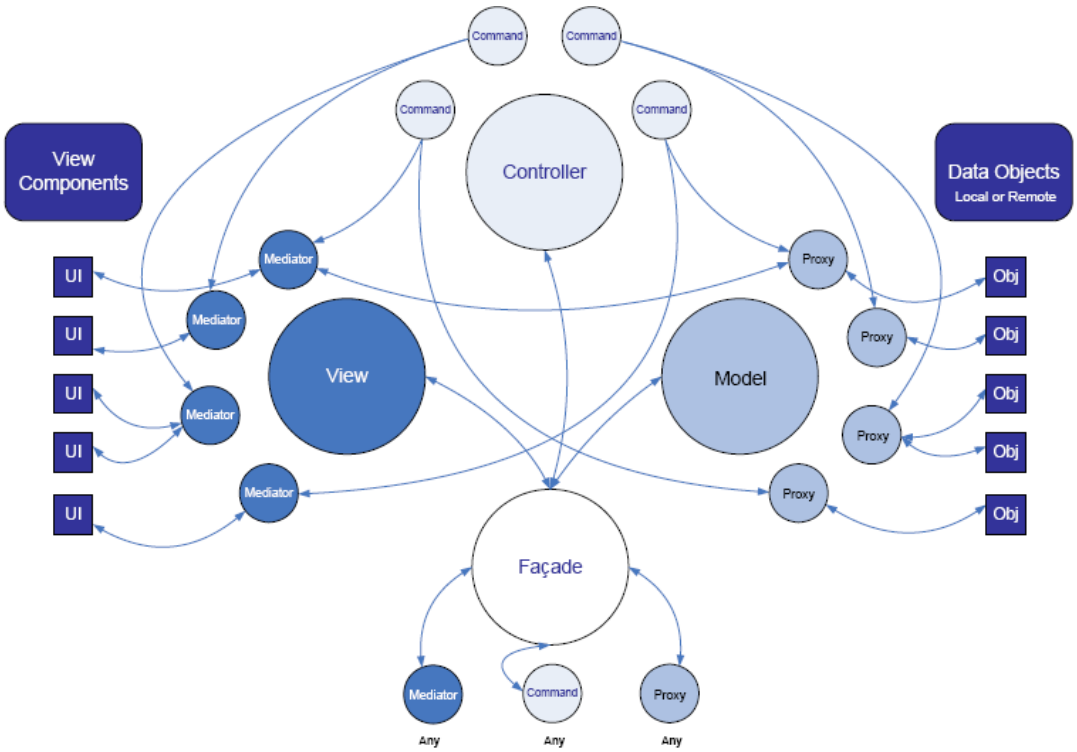
pozwalają na pełną automatyzację wykonania zdefiniowanych testów [22].

Odpowiednio przygotowany skrypt Ant, automatyzujący uruchomienie testów (zarówno w wersji przeglądarkowej, tj. z wykorzystaniem wspomnianego *MonkeyTestLauncher*, jak i aplikacji AIR, tj. FlUnitAirTestRunner.exe), może być uruchomiony z poziomu środowiska wytwórczego (np. Flex Bulider/Flash Builder) lub/i w ramach oprogramowania ciągłej integracji.

4. Mikroarchitektura aplikacji klienta oparta na szablonie PureMVC

4.1. Wprowadzenie do koncepcji szablonu Pure MVC

Szablon PureMVC (ang. Pure Model-View-Controller) jest szablonem tworzenia aplikacji klienta z luźno powiązanymi elementami konstrukcyjnymi (ang. loosely coupled elements) [23, 24]. Szablon bazuje na koncepcji znanych wzorców projektowych [31], m.in. takich jak: *Fasada*, *Metoda fabrykująca*, *Obserwator*, *Pośrednik*, *Mediator*, *Komenda*, *Singleton*, *Multiton* czy metawzorzec projektowy *Model-Widok-Kontroler*. Szablon PureMVC jest implementowany dla wielu platform i języków programowania, w szczególności dla technologii Flash/Flex i języka Action Script.



Rys. 2. Elementy szablonu PureMVC (źródło [23])
Fig. 2. Elements of PureMVC Framework (source [23])

PureMVC zakłada trójwarstwową architekturę aplikacji (nieformalny diagram został pokazany na rys. 2). W wersji PureMVC singlecore (tzw. wersja jednordzeniowa) elementy podstawowe: *Model*, *View* i *Controller* (ang. core actors) występują tylko w jednej instancji (bazują na wzorcu projektowym *singleton*).

View jest elementem szablonu przechowującym referencje do nazwanych elementów typu *Mediator*. Obiekty Mediator zajmują się zarządzaniem tzw. logiką konkretnego elementu interfejsu użytkownika – *ui* (ang. user interface), np. walidacji danych. Mediator separuje przetwarzanie GUI od definicji GUI (elementów *ui* wyrażonych w MXML).

Controller jest elementem szablonu obsługującym „mapowanie" elementów typu *Command*. Bezstanowe obiekty *Command*, tworzone na żądanie, realizują określoną logikę biznesową, udostępnioną poprzez uwspólniony interfejs (poprzez jedyną metodę *execute*).

Model jest elementem szablonu przechowującym referencje do nazwanych elementów typu *Proxy*. Obiekty *Proxy* odpowiedzialne są za realizację dostępu do danych systemu, na ogół wykorzystując serwer aplikacji i bazując na mechanizmie *remote objects*, omawianym w rozdziale 2.

Elementy *Model*, *View*, *Controller* nie są bezpośrednio wykorzystywane przez programistów – ich działanie jest przysłonięte przez obiekt *Fasady* (ang. Facade), również zrealizowany w postaci *singletonu*. *Fasada* udostępnia wszystkie niezbędne usługi MVC. Programiści używają bezpośrednio elementów typu *Mediator*, *Command*, *Proxy*. Poprzez metody *Fasady* rejestrują oni (a potem uzyskują referencje i używają bądź usuwają) nazwane obiekty typu *Mediator* czy *Proxy*.

Luźne powiązanie elementów aplikacji realizowane jest dzięki zastosowaniu w szablonie PureMVC komunikacji pomiędzy elementami za pomocą notyfikacji (powiadomień), czyli obiektów klasy *Notification*. *Notyfikacja* posiada swoją nazwę (identyfikator), typ, ciało (dane, zawartość).

Programiści poprzez *Fasadę* szablonu PureMVC mogą zarejestrować wywołanie komendy (utworzenie elementu *Command* i uruchomienie akcji) na zdarzenie wystąpienia wskazanej, nazwanej notyfikacji. W momencie wysłania notyfikacji (poprzez metodę *SendNotificaton* dostępną z *Fasady*) uruchamiana jest odpowiednia *Komenda*, a ciało notyfikacji jest przekazywane jako parametr wywołania komendy (parametr metody *execute*).

Mediator może wysyłać notyfikacje (np. powodując uruchomienie odpowiednich komend). Mediator może też być uwrażliwiony na wystąpienie notyfikacji (np. w odpowiedzi na zdarzenie dostarczenia danych z elementu *Proxy*).

Komenda, na potrzeby operowania na danych, może wywołać metody elementu *Proxy*. Element *Proxy* (np. po uzyskaniu odpowiedzi od zdalnego obiektu) może wysyłać notyfikacje, przesyłając tym samym żądane wartości (np. do *Mediatora*).

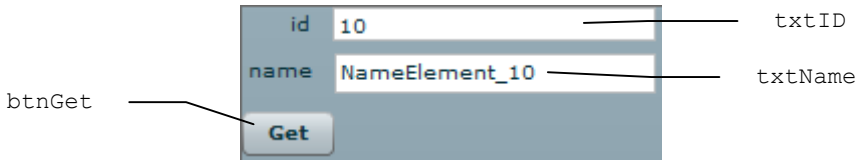
Jednym z elementów służących do ewentualnego rozszerzania funkcjonalności szablonu PureMVC jest *Interceptor* [27]. Wykorzystanie elementu *Interceptor* pozwala na zmianę standardowego przepływu notyfikacji. Zanim notyfikacja faktycznie uaktywni element *Command* lub wywoła metodę elementu *Mediator*, może zostać przechwycona w celu: usunięcia tej notyfikacji, zmiany zwartości ciała notyfikacji (informacji przesyłanej z notyfikacją), podmiany notyfikacji na inną itp.

Mechanizm *Inteceptors* będzie wykorzystany do rozszerzenia modułu testów jednostkowych dla szablonu PureMVC, pozwalając na testowanie wystąpień oczekiwanych notyfikacji. Zaproponowane nowe rozwiązanie będzie opisane w rozdziale 5.

4.2. Budowa przykładowej prostej aplikacji klienta wykonanej w technologii Flex bazującej na szablonie PureMVC

Na potrzeby prezentacji funkcjonalności rozszerzonego modułu testów o możliwość sprawdzania wystąpień notyfikacji (omówionego w rozdziale 5), w niniejszym rozdziale pokazano architekturę pewnej bardzo prostej aplikacji klienta. Prostota GUI nie przekłada się na prostotę omawianej architektury. Na bazie tej aplikacji zostanie prześledzony przepływ sterowania, podlegający automatyzacji testowania w kolejnym rozdziale.

GUI przykładowej aplikacji pozwala na zadanie wartości identyfikatora pewnego elementu i wysłanie żądania uzyskania elementu (element posiada składowe: identyfikator id: int i nazwa – name: String). Żądanie przechodzi poprzez kolejne warstwy systemu i w wyniku jego realizacji po stronie serwerowej, wygenerowana jest odpowiedź, której widocznym rezultatem jest wyświetlenie właściwej wartości nazwy żądanego elementu (rys. 3).



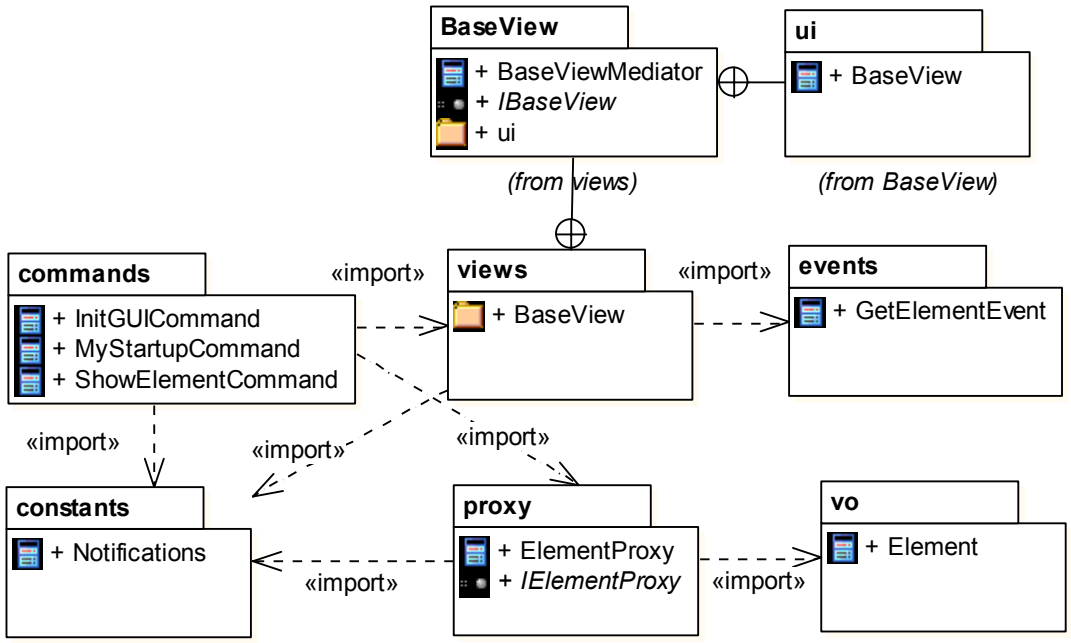
Rys. 3. Komponent widoku (zawartość okna przeglądarki) z przykładowej aplikacji
Fig. 3. Sample view component (content of web browser window) from sample application

Na rysunku 3 przedstawiona jest instancja komponentu widoku po realizacji żądania. Pokazany komponent widoku w kodzie programu Flex odpowiada klasie *BaseView* (rys. 5), opisanej za pomocą MXML.

Rysunek 4 przedstawia architekturę systemu w kontekście budowy kodu źródłowego aplikacji (struktura pakietów).

W pakiecie *constans* umieszczone są stałe identyfikujące, w szczególności identyfikatory notyfikacji (zgrupowane w ramach klasy *Notifications*).

Pakiet *commands* grupuje klasy komend, w szczególności komendy inicjujące aplikacje *MyStartupCommand, InitGUICommand* oraz komendę *ShowElementCommand*, realizującą zasadniczą logikę przetwarzania aplikacji. W ramach komendy *MyStartupCommand* następuje zarejestrowanie w infrastrukturze odpowiednich obiektów *Proxy* i *Command*, poprzez wywołanie odpowiednich metod fasady szablonu PureMVC (*registerCommand*, *registerProxy*). Rejestracja komendy *ShowElementCommand* polega na zadeklarowaniu uruchomienia tej komendy na zdarzenie wystąpienia notyfikacji o identyfikatorze zadeklarowanym w klasie *Notifications* pakietu *constans* (stała *Notifications.SHOW_ ELE-MENT* z rys. 5). W ramach komendy *InitGUICommand* następuje utworzenie obiektu i zarejestrowanie w infrastrukturze mediatora widoku (klasa *BaseViewMediator*), inicjalizo-wanego odpowiednim widokiem (wstrzyknięcie w konstruktorze obiektu widoku z klasy *BaseView*).



Rys. 4. Diagram pakietów pokazujący w strukturę kodu źródłowego fleksowej aplikacji klienta bazującej na szablonie PureMVC

Fig. 4. Package diagram for source code of the sample flex client application based of PureMVC framework

Pakiet *vo* (ang. Value Objects) zawiera klasy definiujące obiekty wykorzystywane do przekazywania danych w komunikacji pomiędzy warstwami aplikacji, odpowiadające encjom biznesowym (tutaj tylko klasa *Element*).

Pakiet *proxy* zawiera interfejsy (tutaj *IElementProxy*) oraz implementacje obiektów pośredniczących (w tym wypadku jeden element pośredniczący – *ElementProxy*). Zastosowanie interfejsu pozwala na wykorzystywanie w aplikacji (w zależności od konfiguracji) różnych implementacji elementu pośredniczącego. W szczególności w tym opracowaniu, na potrzeby testowania, będą wykorzystywane dwie implementacje *ElementProxy* – dwa rodzaje atrap elementów *Proxy*. W pierwszym rodzaju (rys. 7) przepływ sterowania ograniczy się w całości do aplikacji flexowej (wartości utworzonego obiektu klasy *Element* są w całości wyznaczane w ramach metody obiektu klasy *ElementProxy*). W drugim rodzaju (rys. 8) przypływ sterowania jest „szerszy” – obejmuje również przetwarzanie po stronie serwera (wartość elementu jest wyznaczana przez akcje po stronie serwera aplikacji, w metodzie zdalnego obiektu, zaimplementowanego w Javie).

Pakiet *views* zawiera definicję pakietów opisujących działanie każdego z widoków aplikacji (w tym wypadku jeden pakiet *BaseView*). Każdy pakiet opisujący działanie widoku zawiera: definicję interfejsu widoku (tutaj *IBaseView*), mediator widoku (tutaj *IBaseViewMediator*) oraz implementację widoku (tutaj klasa *BaseView* w podpakiecie *ui*). Wykorzystywanie interfejsu widoku (tutaj *IBaseView*) pozwala na ewentualne zastosowanie przez

mediator innych (nie pokazanych tutaj) implementacji widoków. Takie podejście zwiększa potencjalnie możliwości testowania aplikacji od strony GUI.

Pakiet *events* definiuje klasy zdarzeń generowanych po stronie GUI w ramach widoków (tutaj zdarzenie *GetElementEvent*, generowane w ramach *BaseView* w sytuacji żądania elementu o podanym identyfikatorze). Zdarzenia (ang. events) są generowane i obsługiwane na styku widok-mediator i są charakterystyczne dla środowiska, w którym została wykonana aplikacja, tzn. flash/flex/action script. Notyfikacje pełniąc podobne funkcje, ale w zakresie komunikacji pomiędzy mediatorami, komendami i pośrednikami, są charakterystyczne dla szablonu PureMVC. Wykorzystanie tego innego, uniwersalnego sposobu komunikacji (nie poprzez obiekty klasy *flash.events.Event*) wynika z wieloplatformowości szablonu PureMVC [25] (szablon może być używany również dla .NET/C# czy J2EE/J2SE/Java, a nie tylko w technologii Flex/Flash).

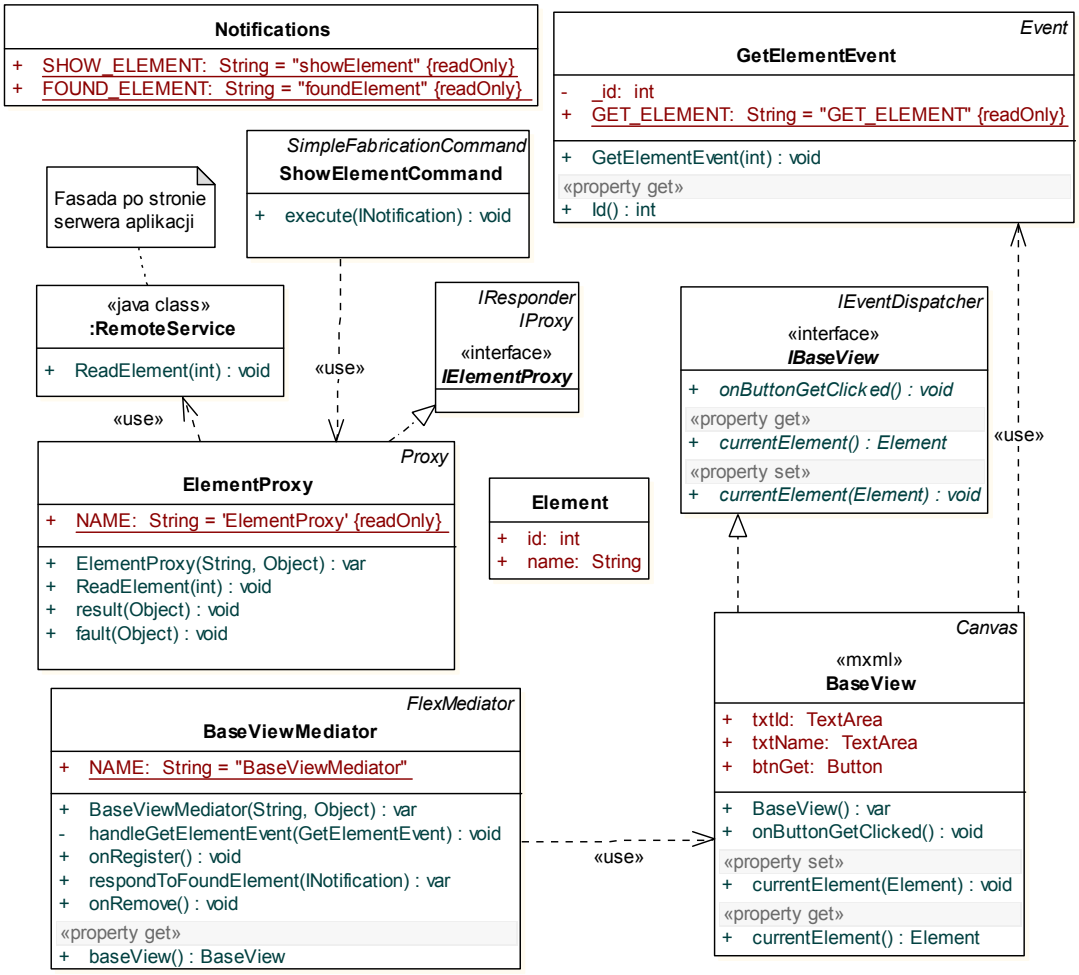
Rysunek 5 przedstawia model ważniejszych klas systemu (bez uwzględnienia zanurzenia w strukturze pakietów).

Obiekt klasy *BaseView* pozwala na pobranie z GUI lub ustawienie i wyświetlenie w ramach GUI wartości elementu oraz zasymulowanie zdarzenia naciśnięcia klawisza. Naciśniecie klawisza ma wygenerować obiekt zdarzenia *GetElementEvent,* inicjowany identyfikatorem żądanego elementu (własność *Id* klasy *GetElementEvent*).

BaseViewMediator, w reakcji na wystąpienie zdarzenia *GetElementEvent* (w metodzie *handleGetElementEvent*), dokonuje walidacji danych i „zleca" (wywołanie *SendNotification*) wyznaczenie elementu komendzie *ShowElementCommand*, generując *Notyfikację* o identyfikatorze *SHOW_ELEMENT*. Identyfikator żądanego elementu przekazywany jest w ciele *Notyfikacji*.

ShowElementCommand, realizując zlecenie, wykonuje akcję pobrania referencji do zarejestrowanego wcześniej obiektu *Proxy* (nazwa tego obiektu – 'ElementProxy'), a następnie wywołuje metodę *ReadElement* z elementu *Proxy*.

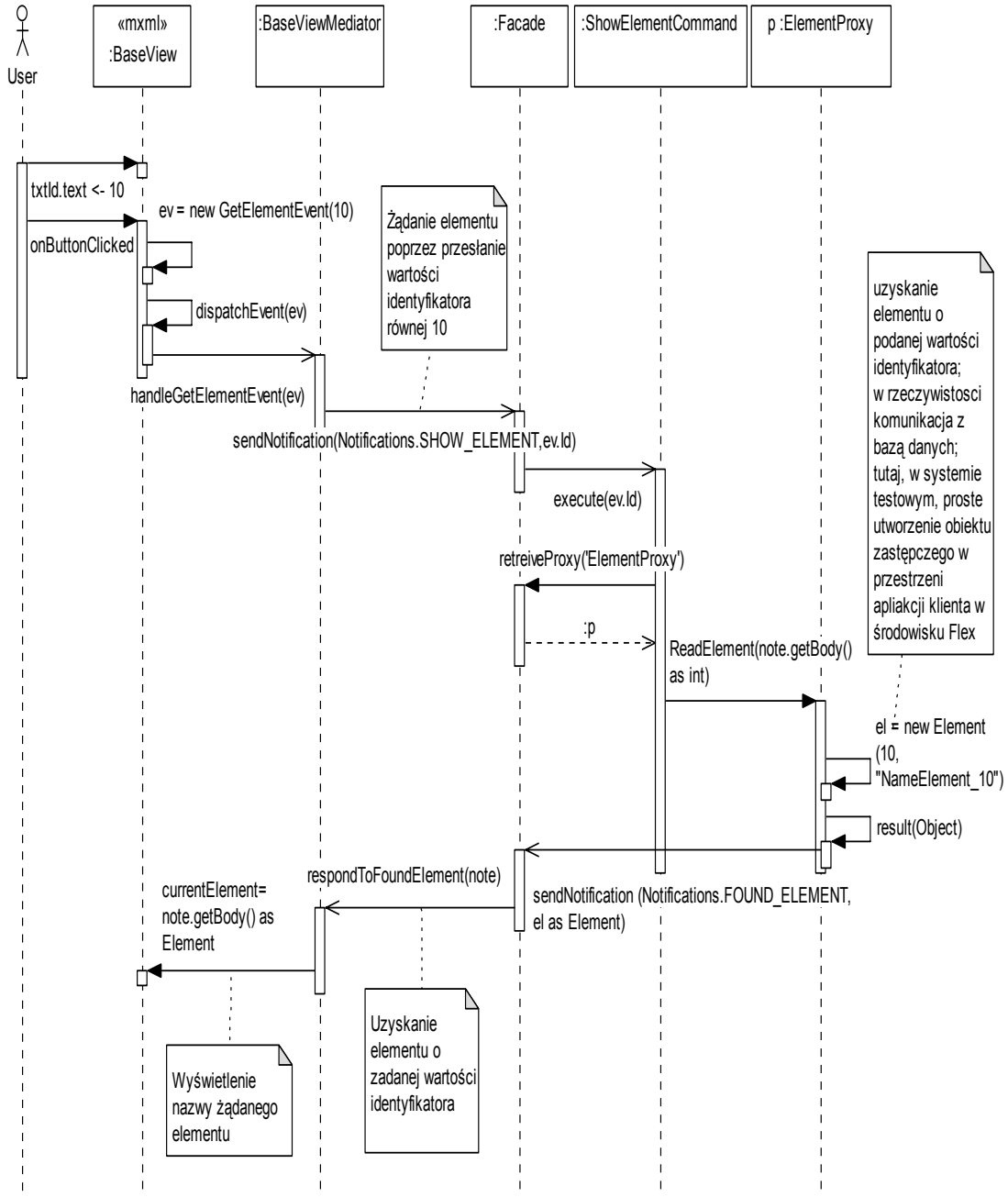
ElementProxy, implementując interfejs *mx.rpc.IResponder,* umożliwia uzyskanie żądanego elementu w ramach metody *result*. W trakcie normalnej eksploatacji systemu (nie w trybie testowania) metoda *result* (lub *fault* w przypadku błędu) jest wywoływana asynchronicznie w stosunku do inicjującego wywołania *ReadElement*. Jest ona uruchamiania w odpowiedzi na poprawne (lub błędne) działanie metody zdalnego obiektu po stronie serwera aplikacji (klasa *RemoteService* na rys.5). Taki tryb pracy pokazany jest na rys. 7. W trakcie testowania opisywanego poniżej, zastosowano synchroniczny sposób wywołania *result* – metoda *ReadElement* z obiektu atrapy *ElementProxy* wprost wywołuje *result*, gdzie tworzony jest odpowiedni żądany element. Taki wariant – polegający na testowaniu warstw aplikacji bez komunikacji z serwerem aplikacji – został pokazany na rys. 6.



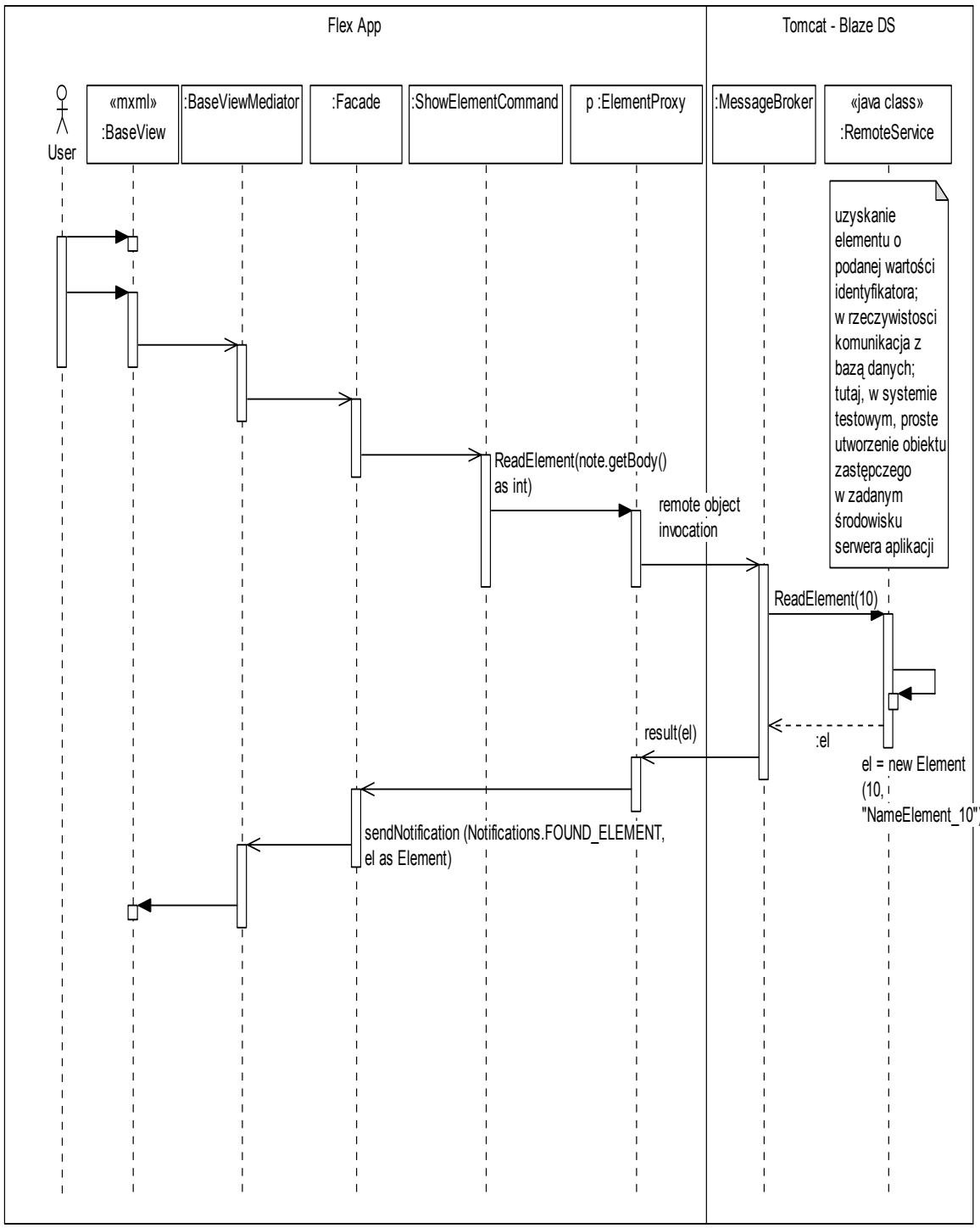
Rys. 5. Model klas aplikacji przykładowej aplikacji
Fig. 5. Class model of sample application

Wywołanie *result*, oprócz uzyskania (lub utworzenia) żądanego elementu, powoduje wysłanie *Notyfikacji* o identyfikatorze *FOUND_ELEMENT*, której ciało stanowi obiekt żądany element.

Uwrażliwienie mediatora na wystąpienie konkretnej notyfikacji obywa się poprzez implementację metody, której dwuczłonowa nazwa składa się z prefiksu ‘respondTo’ oraz nazwy notyfikacji. Dzięki temu metoda *respondToFoundElement* z *BaseViewMediator* zostanie wywołana pośrednio w wyniku działania obiektu *ElementProxy*. Przekazany w parametrze metody *respondToFoundElement* żądany obiekt klasy *Element* zostaje następnie przekazany przez mediator do widoku *BaseView* (wywołanie *currentElement* z interfejsu *IBaseView*), a tym samym wyświetlony na ekranie.



Rys. 6. Sekwencja przepływu sterowania w przykładowej aplikacji
Fig. 6. A control flow in sample application



Rys. 7. Sekwencja przepływu sterowania w przykładowej aplikacji klienta z uwzględnieniem uproszonego przetwarzania po stronie serwera aplikacji

Fig. 7. A control flow in sample client application with simple processing on application server side

5. Rozszerzenie biblioteki wspierającej testy jednostkowe w zakresie obsługi notyfikacji

W uproszczeniu testy jednostkowe polegają na użyciu zbioru bezparametrowych metod, w których następuje albo wywołanie metod albo użycie własności obiektów testowanych. W pojedynczej metodzie testującej sprawdza się, czy metoda/własność zwraca oczekiwane wartości.

Taki model testów jednostkowych jest jednak niewystarczający dla testowania aplikacji bazujących na szablonie PureMVC, który opiera się na przepływie notyfikacji. Metody obiektów testowanych nie tylko powinny zwracać oczekiwane rezultaty, ale również generować oczekiwane notyfikacje (pożądany efekt uboczny działania metody).

Funkcjonalność modułu testującego pozwalająca na sprawdzenie, czy wystąpiła określona notyfikacja (np. oczekiwana liczba określonych notyfikacji, oczekiwana notyfikacja z oczekiwaną zawartością ciała), została zaproponowana, zaimplementowana i opisana w niniejszym opracowaniu.

5.1. Mikroarchitektura rozszerzonego modułu testującego wspierającego testowanie notyfikacji

Zaproponowane rozszerzenie modułu wspierającego testowanie jednostkowe aplikacji bazujących na szablonie PureMVC wykorzystuje mechanizm *Interceptors* [27]. Najważniejsze klasy rozszerzenia tego modułu zostały pokazane na rys. 8.

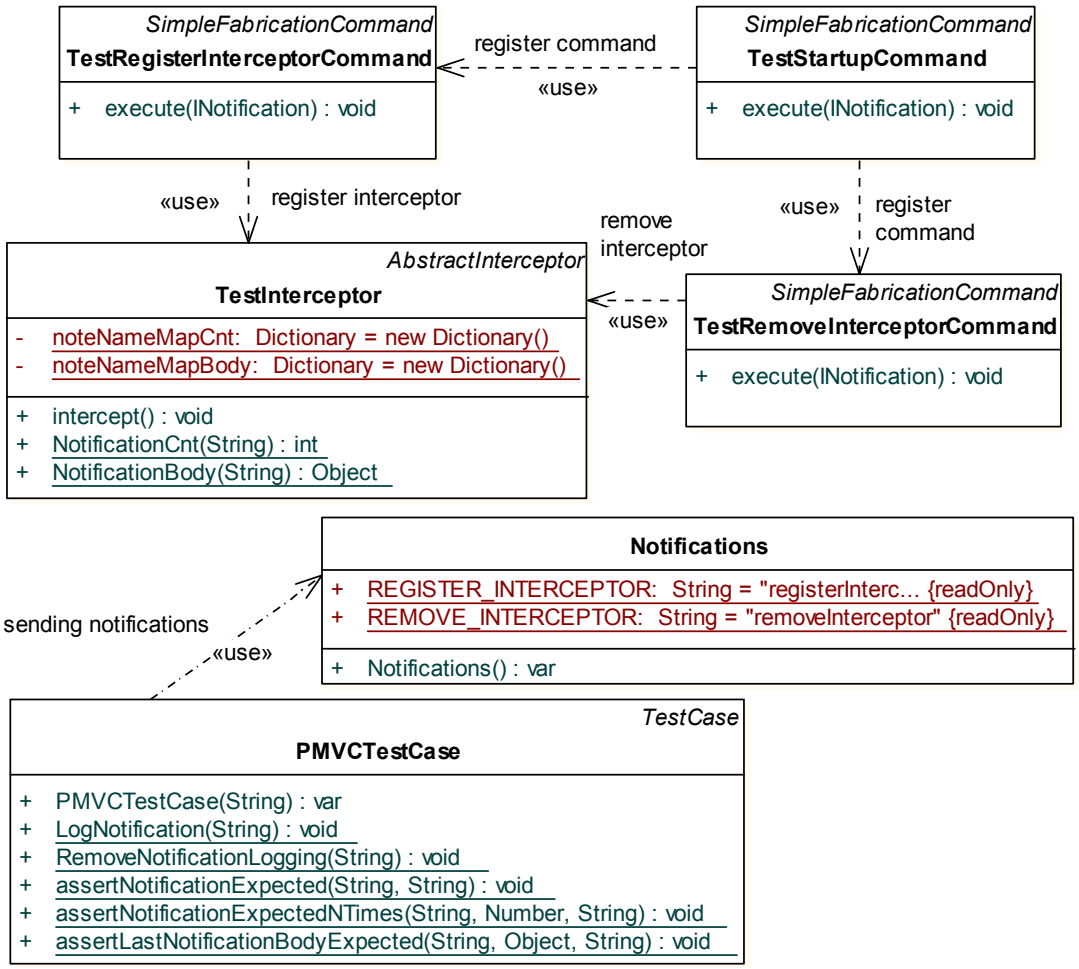
Klasa *TestInterceptor* w dwóch kolekcjach notyfikacji: *noteNameMapCnt* oraz *noteNameMapBody* może przechowywać odpowiednio: liczbę notyfikacji o zadanym identyfikatorze oraz kolejkę wszystkich notyfikacji wraz z zwartością ich ciał. Publiczne metody *NotificationCnt* oraz *NotificationBody* mogą dostarczyć odpowiednio:

- informację o liczbie notyfikacji o zadanym identyfikatorze od momentu rozpoczęcia monitorowania,
- ciało (zawartość) ostatniej notyfikacji o zadanym identyfikatorze.

Metody te wykorzystane są do budowy specyficznych asercji dostarczanych w ramach rozszerzonego modułu testującego (metody *assert*Notification** klasy *PMVCTestCase*). Przeciążona metoda *intercept* powala na zachowanie normalnego przepływu notyfikacji (wewnętrzne wywołanie *proceed* [27]), ale również uaktualnia zawartości omówionych wcześniej kolekcji notyfikacji.

Komendy *TestRegisterInterceptorCommand* oraz *TestRemoveInterceptorCommand* pozwalają na zrejestrowanie oraz wyrejestrowanie *TestInterceptor*-a dla notyfikacji o określonym identyfikatorze (poprzez wywołanie odpowiednich metod z fasady). Identyfikator po-

dawany jest w parametrze komendy (parametr metody *execute*). Uruchomienie ww. komend włącza/wyłącza monitowanie notyfikacji o zadanym identyfikatorze.



Rys. 8. Model klas modułu testującego rozszerzonego o obsługę notyfikacji
Fig. 8. Class model of testing module extended for notification handling

Komenda *TestStartupCommand*, uruchamiana automatycznie na starcie modułu testującego, rejestruje komendy *TestRegisterInterceptorCommand* oraz *TestRemoveInterceptorCommand* na notyfikacje o identyfikatorach *REGISTER_INTERCEPTOR* i *REMOVE_INTERCEPTOR*.

Podstawową klasą rozszerzonego modułu testującego jest *PMVCTestCase*. Rozszerza ona standardową klasę *flexunit.framework.TestCase* o obsługę notyfikacji w procesie testowania.

Standardowe podejście, zgodne z koncepcją xUnit [26] (zgodne np. z jUnit, nUnit, etc.), zakłada, że użytkownik/programista definiuje swoją klasę testującą na bazie *TestCase*. Metody klasy wywiedzionej z *TestCase* mogą występować w rolach:

- pojedynczych metod testujących (publiczne, bezparametrowe metody *void*, których nazwa zaczyna się od słowa „test”),
- metody wykonywanej przed każdym pojedynczym testem (publiczna, przeciążona, bezparametrowa metoda *void* o nazwie *setUp*),

- metody wykonywanej po każdym pojedynczym teście (publiczna, przeciążona, bezparametrowa metoda *void* o nazwie *tearDown*).

Klasa *TestCase* dostarcza 12 wariantów funkcji asercji na potrzeby weryfikacji zgodności wyniku z wartościami oczekiwanymi.

W nowym, proponowanym rozwiązaniu zakłada się, że definiowana przez użytkownika/programistę klasa testującą powinna dziedziczyć z *PMVCTestCase*.

Metody *LogNotification(noteName:String)* oraz *RemoveNotificationLogging (noteName:String)* z *PMVCTestCase* pozwalają na włączanie/wyłączenie w *interceptorze* logowania wystąpień notyfikacji o podanym identyfikatorze.

Metoda *LogNotification* wysyła notyfikację *REGISTER_INTERCEPTOR,* a jako ciało tej notyfikacji wysyłana jest, przekazana w parametrze *noteName*, nazwa identyfikatora notyfikacji wskazanej do śledzenia. To, oczywiście, spowoduje uruchomienie komendy *TestRegisterInterceptorCommand* z odpowiednim parametrem (nazwą notyfikacji do śledzenia), a tym samym rozpoczęcie logowania potencjalnych wystąpień notyfikacji o wskazanym identyfikatorze.

Analogicznie metoda *RemoveNotificationLogging* wysyła notyfikację *REMOVE_INTERCEPTOR*, etc..., co w konsekwencji spowoduje zaprzestanie logowania wystąpień notyfikacji o wskazanym identyfikatorze.

Klasa *PMVCTestCase* dostarcza też użyteczne asercje związane z obsługą notyfikacji, np.:

- `assertNotificationExpected(noneName:String,text:String)` – sprawdzenie, czy notyfikacja o podanym identyfikatorze *noteName* wystąpiła co najmniej raz,
- `assertNotificationExpectedNTimes(noneName:String,occuredExpected:Number=1,text:String)` – sprawdzenie, czy notyfikacja o podanym identyfikatorze *noteName* wystąpiła dokładnie *occuredExpected* razy,
- `assertLastNotificationBodyExpected(noneName:String, bodyExpected:Object, text:String)` – sprawdzenie, czy ostatnia notyfikacja o podanym identyfikatorze *noteName* zawiera ciało o wartości równej *bodyExpected*.

5.2. Wykorzystanie rozszerzonego modułu testującego w testowaniu przykładowej aplikacji

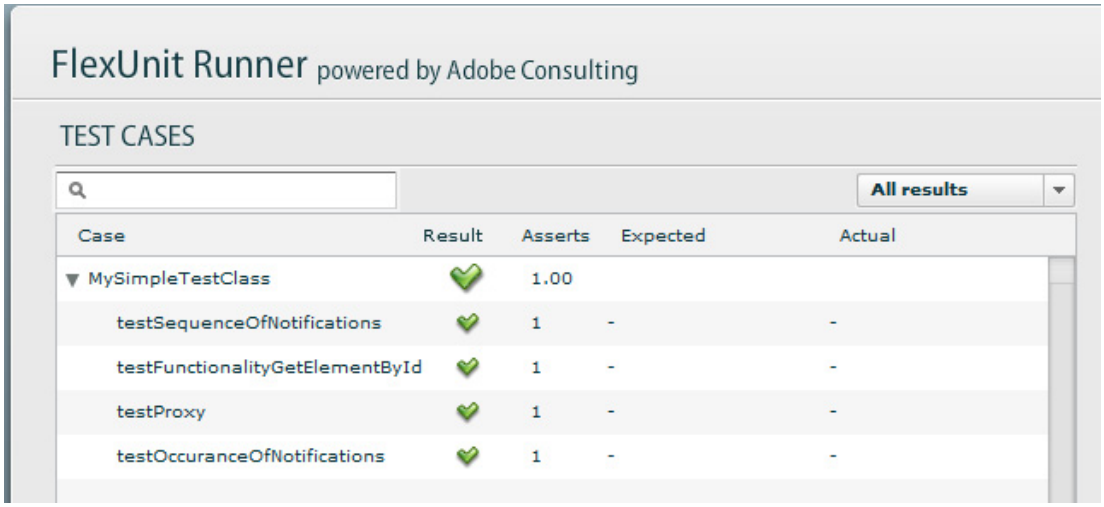
Poniżej pokazano kod źródłowy w języku Action Script klasy *MySimpleTestCase*, testującej elementy przykładowej aplikacji omówionej w rozdziale 4.

```
01  public class MySimpleTestCase extends PMVCTestCase {

02  private var _fa:Facade ;
03  private var _bv:BaseView;
```

```
04  public  override function setUp(): void {
     // inicjalizacja;
     // uzyskanie uchwytu do fasady szablonu PuremMVC
05   _fa = Facade.getInstance(key) as Facade;
     // utworzenie widoku
06   _bv = new BaseView();
     //  inicjalizacja elementów infrastruktury szablonu PureMVC
07   _fa.registerMediator(new BaseViewMediator(BaseViewMediator.NAME,_bv));
08   _fa.registerCommand(Notifications.SHOW_ELEMENT, ShowElementCommand);
09   _fa.registerProxy(new ElementProxy(ElementProxy.NAME));
    }

10  public function testOccuranceOfNotifications () : void {
      // sprawdzenie dokładnej liczby wystąpień powiadomień
      // o zadanej nazwie.
11    LogNotification(Notifications.SHOW_ELEMENT);
      // wielokrotne powiadomienie
13    _fa.sendNotification(Notifications.SHOW_ELEMENT,19) ;
14    _fa.sendNotification(Notifications.SHOW_ELEMENT, 20) ;
      // oczekiwanie dokładnie dwóch powiadomień
16    assertNotificationExpectedNTimes(Notifications.SHOW_ELEMENT,2);
17    RemoveNotificationLogging(Notifications.SHOW_ELEMENT);
    }

18  public function testSequenceOfNotifications () : void {
      // pośrednie testowanie działania BaseViewMediator -
      // sprawdzenie następstwa powiadomień SHOW_ELEMENT -> FOUND_ELEMENT
19    LogNotification(Notifications.FOUND_ELEMENT);
20    _fa.sendNotification(Notifications.SHOW_ELEMENT, 10) ;
21    assertNotificationExpected(Notifications.FOUND_ELEMENT);
22    RemoveNotificationLogging (Notifications.FOUND_ELEMENT);
   }

23  public function testProxy () : void {
      // Testowanie elementu Proxy -
      // sprawdzenie czy wykonanie metody obiektu pośredniczącego
      // powoduje wygenerowanie powiadomienia FOUND_ELEMENT
24     LogNotification(Notifications.FOUND_ELEMENT);
25    var _pr:ElementProxy
           = _fa.retrieveProxy(ElementProxy.NAME) as ElementProxy;
26     _pr.ReadElement(10) ;
27    assertNotificationExpected(Notifications.FOUND_ELEMENT);
28    RemoveNotificationLogging (Notifications.FOUND_ELEMENT);
   }
29   public function testFunctionalityGetElementById () : void {
       // inicjalizacja widoku, powołanie kontrolek zagnieżdżonych
30     _bv.initialize();
       // ustawienie wartości pola tekstowego txtID
31     _bv.txtId.text = "10";
       // symulacja naciśnięcia klawisza btnGet
32     _bv.onButtonGetClicked();
       // sprawdzenie GUI - czy w polu txtName test oczekiwana wartość
33    assertEquals
     ("Wartość 'name' pobranego elementu nie jest zgodna z oczekiwaniem",
      "NameElement_10", _bv.txtName.text );
   }

32  public  override function tearDown(): void {
     // deinicjalizacja . . .
    }

  } // end of MySimpleTestCase
```

Metody testujące *testProxy*, *testOccuranceOfNotifications*, *testSequenceOfNotifications* wykorzystują bezpośrednio nowe rozszerzenia modułu testującego. Metody specyficzne dla opisywanego rozszerzenia zostały zaznaczone pogrubioną czcionką. Metoda testująca

testFunctionalityGetElementById realizuje dokładnie scenariusz opisany diagramem sekwencji pokazanym na rys. 6.



Rys. 9. Wynik uruchomienia testów jednostkowych obejmujących elementy przykładowej aplikacji
Fig. 9. Result of running of unit tests for sample application's elements

Na rys. 9 pokazano wynik uruchomienia zestawu testów jednostkowych, zdefiniowanych w ramach opisywanej klasy *MySimpleTestClass*. Rys. 9 pokazuje wycinek ekranu utworzonego przez komponent FlexUnit Runner, który został wykorzystany w roli „silnika" testów jednostkowych.

5.3. Rozszerzenie modułu obsługi obiektów zastępczych o obsługę notyfikacji

Kolejne rozwinięcie modułu wspierającego testowanie dotyczy rozbudowy modułu obsługi atrap (np. mock4as [28]) o obsługę notyfikacji. Moduł wspierający tworzenie atrap pozwala na nagrywanie oczekiwań w stosunku do obiektu zastępczego (atrapy) po to, by potem odtworzyć zachowanie obiektu zastępczego na etapie testu właściwego jakieś złożonej funkcjonalności, działającej na tym obiekcie zastępczym [3, 5, 29].

Przykładowo, wykorzystując moduł mock4as, pewien obiekt zastępczy *MockMyElement* może zostać utworzony na podstawie interfejsu *IMyElement*:

```
interface IMyElement{
  MyFunction (in: String):String;
}

class MockMyElement extends Mock implements IMyElement { … }.
```

Wtedy nagrywanie oczekiwania: ma być wywołana funkcja *MyFunction* z parametrem *in*=”InVal”, która powinna zwrócić wartość “OutVal” będzie wyrażone następującym kodem:

```
MockMyElement.expects("MyFunction").withArgs("InVal").willReturn("OutVal").
```

Proponowane w niniejszej pracy rozszerzenie pozwala na uwzględnienie faktu, że przy wykorzystaniu szablonu PureMVC pewne funkcje wykonują istotne z punktu widzenia sterowania akcje uboczne, tzn. mogą wysyłać notyfikacje. Stąd proponowane rozszerzenie – metoda *willSend(noteName:String, noteBody:Object)*.

Przykładowo, dzięki temu można rozwinąć poprzednio omówione ustawianie oczekiwań w stosunku do obiektu zastępczego o wysłanie notyfikacji (w ramach wywołania *MyFunction*) o nazwie „SOME_NOTIFY" z zawartością, którą jest pewna wartość typu *integer*, tj.102:

```
MockMyElement.expects("MyFunction").withArgs… .willSend("SOME_NOTIFY",102).
```

Frazy *willSend* można użyć wielokrotnie, jeśli istnieje potrzeba zasymulowania wysyłania wielu notyfikacji w ramach jednego wywołania funkcji. Przykładowo, poniżej pokazano kod nagrywający żądania dwukrotnego wysłania notyfikacji:

```
MockMyElement.expects("MyFunction").withArgs… .willSend("SOME_NOTIFY1",1)
   .willSend("SOME_NOTIFY2", obj2).
```

6. Podsumowanie

Artykuł prezentuje metody i narzędzia wpierające automatyzację procesu testowania wielowarstwowych systemów, wytworzonych z wykorzystaniem technologii Adobe Flex, których architektura opiera się na wykorzystaniu popularnego szablonu PureMVC.

W opracowaniu omówione są sposoby automatycznego testowania systemu od strony interfejsu użytkownika. Omówione są dwa podejścia – jedno z użyciem narzędzia Selenium, drugie FlexMonkey. Chociaż podejście z wykorzystaniem Selenium jest uniwersalne (pozwala na testowanie aplikacji WWW wykonanych również w technologii innej niż Flex), to jednak inne ograniczenia (np. brak możliwości „nagrywania interakcji" z GUI w postaci skryptów testowych, zorientowanie na przeglądarkę FireFox) skłaniają raczej do wyboru drugiej z opcji – modułu FlexMonkey (przeznaczonego jedynie dla technologii Flash/Flex).

W pracy zaprezentowano też skrótowo sposoby testowania funkcjonalności udostępnianej przez serwer aplikacji, polegające na wykorzystaniu standardowego podejścia z użyciem jUnit, jMock.

Głównym elementem opracowania jest prezentacja nowego elementu – rozszerzenia wspomagającego tworzenie testów jednostkowych w zakresie elementów aplikacji klienta, wynikających z użycia szablonu PureMVC. Zaproponowane rozszerzenia stanowią uzupełnienie wcześniej opisanych metod testowania (testowania funkcjonalności GUI i serwera aplikacji).

Wykorzystanie tzw. notyfikacji, czyli specyficznego sposobu komunikacji pomiędzy elementami aplikacji bazującej na PureMVC, wymaga, zdaniem autora, odpowiedniego wsparcia ze strony modułu testującego. W proponowanym rozszerzeniu modułu testującego zaimplementowano obsługę notyfikacji. W szczególności umożliwiono śledzenie i logowanie notyfikacji oraz sprawdzenie wystąpienia określonych notyfikacji poprzez zestaw asercji. Dodatkowo rozszerzono moduł wspierający testowanie w oparciu o mechanizm obiektów zastępczych – atrap (ang. mock object). Dzięki temu rozszerzeniu, obiekty zastępcze nie tylko mogą symulować wywołania metod, ale również mogą generować żądane notyfikacje. Takie podejście pozwala w pełni na testowanie aplikacji bazujących na szablonie PureMVC, gdzie sterowanie opiera się na przepływnie notyfikacji.

Opisywane metody i narzędzia oraz proponowane rozszerzenia zostały pozytywnie zweryfikowane w praktyce.

BIBLIOGRAFIA

1. JUnit.org Resources for Test Driven Development Page, http://www.junit.org.
2. jMock – A Lightweight Mock Object Library for Java Page, http://www.jmock.org.
3. EasyMock Home Page, http://easymock.org.
4. NUnit Home Page, http://www.nunit.org.
5. Augustyn D. R.: Rozwój narzędzi programowych wspierających automatyzację testów jednostkowych dla technologii .NET. Bazy danych. Rozwój metod i technologii. Bezpieczeństwo, wybrane technologie i zastosowania. WKŁ, Warszawa 2008.
6. Augustyn D.R., Stanek I.: „Zastosowanie modułu NUnit w testowaniu jednostkowym aplikacji bazodanowych". Studia Informatica Vol. 30 No. 2B, Gliwice 2009.
7. SeleniumHQ Web application testing tool Page: http://seleniumhq.org/projects.
8. Selenium-RC Page, http://seleniumhq.org/docs/05_selenium_rc.html#selenium-rc-architecture.
9. Caroli P., Lindahl H.: Writing and running functional tests for Flash with Selenium RC, http://www.adobe.com/devnet/flash/articles/flash_selenium.html.
10. Rich Internet Applications, Adobe AIR Page, http://www.adobe.com/products/air.
11. Adobe Labs – Adobe AIR 2 Page, http://labs.adobe.com/technologies/air2.
12. Flex Monkey Home Page, http://flexmonkey.gorillalogic.com/gl/stuff.flexmonkey.html.
13. BlazeDS – Adobe Open Source Page, http://opensource.adobe.com/wiki/dis-play/blazeds-/Developer+Documentation.
14. Adobe Flex Page, http://www.adobe.com/products/flex.

15. Adobe Labs – Adobe Flex Framework Technologies Page, http://labs.ado-be.com-/technologies/flex.
16. AMF specification Page, http://download.macromedia.com/pub/labs/amf/amf3_spec-_121207.pdf.
17. Adobe® Flex™ 3.5 Language Reference Page, http://livedocs.adobe.com/flex/3/ lan-gref/index.html.
18. Adobe – Coding with MXML and ActionScript Page, http://www.adobe.com/devnet/ flex/quickstart/coding_with_mxml_and_actionscript.
19. PureMVC Framework Page, http://puremvc.org/content/view/98/189.
20. External Interface Page, http://livedocs.adobe.com/flex/201/langref/flash/exter-nal/Exter-nalInterface.html.
21. Adobe – Flex Samples – Automation API sample applications Page, http://www.adobe-.com/devnet/flex/samples/custom_automated.
22. Monkey Code Generation Page, http://www.gorillalogic.com/flexmonkey/Mon-keyCode-GenerationR1b1.pdf.
23. Hall C.: PureMVC Implementation Idioms and Best Practices, http://puremvc.org/pa-ges/docs/current/PureMVC_Implementation_Idioms_and_Best_Practices.pdf
24. Hall C.: PureMVC Framework Overview with UML Diagrams, http://puremvc.org/pa-ges/docs/current/PureMVC_Framework_Overview_with_UML.pdf.
25. Hall C.: PureMVC Framework Goals & Benefits, http://puremvc.org/pages/docs/cur-rent/PureMVC_Framework_Goals_and_Benefits.pdf.
26. Webb N.: Unit testing and Test Driven Development (TDD) for Flex and ActionScript 3.0, http://www.adobe.com/devnet/flex/articles/unit_testing.html.
27. Using Interceptors, http://code.google.com/p/fabrication/wiki/Interceptors.
28. Peters J., Caroli P.: Unit testing with mock objects in ActionScript 3.0, http://www.adobe.com/devnet/actionscript/articles/unit_test_mock_objects.html.
29. NMock: A Dynamic Mock Object Library for .NET, http://www.nmock.org.
30. Augustyn D. R.: Rozwój narzędzi programowych wspierających automatyzację testów jednostkowych dla technologii .NET. Bazy danych. Rozwój metod i technologii. Bezpie-czeństwo, wybrane technologie i zastosowania. WKŁ, Warszawa 2008.
31. Gamma E., Helm R., Johnson R., Vlissides J.: Wzorce projektowe. Elementy oprogramo-wania obiektowego wielokrotnego użytku. WNT, Warszawa 2000.

Recenzent: Prof. dr hab. inż. Stanisław Wrycza

Wpłynęło do Redakcji 31 stycznia 2010 r.

Abstract

The paper presents methods and software tools supporting test automation of multilayer information system. Considered systems are based on Adobe Flex technology. Architecture of application layer of such systems is based on PureMVC Framework (Pure Model-View-Controller). The architecture of such systems is described in details.

Methods of GUI functionality testing using Selenium or FlexMonkey is presented. Some methods of testing application server layer using standard approach based on jUnit, JMock are mentioned too.

The main contribution of this work is an extension of the testing module adopted to the conception of PureMVC Framework. In applications based on PureMVC, loosely coupled application elements (commands, mediator, proxies) communicates using the mechanism of notifications. The extension of notification handling by the testing module is required. The functionality of tracing and logging notifications and checking the occurrence of a required notification (notification assertion) was proposed, implemented and described. The extended testing module supports mock mechanism with notification handling. In proposed testing module a mock object can generate a required notification (we can "record in mock object " not only property using or method invoking, but sending required notifications too).

Adres

Dariusz R. AUGUSTYN: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-101 Gliwice, Polska, draugustyn@polsl.pl.