STUDIA INFORMATICA 2011
Volume 32 Number 2B (97)

Dariusz R. AUGUSTYN, Łukasz WARCHAŁ
Politechnika Śląska, Instytut Informatyki

PROJEKTOWANIE I WYTWARZANIE APLIKACJI TYPU RIA NA PLATFORMĘ ADOBE FLEX. TECHNOLOGIE, WZORCE, NARZĘDZIA

Streszczenie. W artykule przedstawiono zaproponowany przez autorów sposób projektowania i wytwarzania zaawansowanych aplikacji internetowych wykorzystujących konkretne technologie związane z platformą Adobe Flex. Omówiono zalecane praktyki, stosowane wzorce projektowe oraz przedstawiono wybrany *framework* aplikacyjny – *PureMVC*. Zaproponowano pewne jego rozszerzenia wspomagające proces wytwarzania oprogramowania.

Słowa kluczowe: Flex, RIA, PureMVC, wzorce projektowe, aplikacja internetowa

ADOBE FLEX-BASED INTERNET APPLICATION DESIGN AND DEVELOPMENT. TECHNOLOGIES, PATTERNS, TOOLS

Summary. In this article authors propose the method of designing and developing advanced internet applications based on Adobe Flex. Best practices and some design patterns are discussed. The paper presents the particular application framework – *PureMVC*. Authors propose some extensions to the framework that support development process.

Keywords: Flex, RIA, PureMVC, design patterns, internet application

1. Wprowadzenie

Nieustanny rozwój sieci Internet doprowadził obecnie do sytuacji, w której większość aplikacji przeznaczonych dla klientów biznesowych czy też dostępnych dla szerokiej rzeszy internautów działa w przeglądarkach internetowych. Jest to korzystne zarówno dla użytkowników – programy tego typu nie wymagają skomplikowanej instalacji czy konfiguracji i są dostępne z każdego komputera podłączonego do sieci – ale także dla dostawców tego typu

rozwiązań – łatwo się nimi zarządza, a wdrożenie nowej wersji jest stosunkowo proste. Nic więc dziwnego, że zapotrzebowanie rynku zaowocowało pojawieniem się dużej liczby technologii umożliwiających tworzenie aplikacji internetowych, często z bardzo rozbudowanym i nowoczesnym interfejsem użytkownika (tzw. RIA – ang. *Rich Internet Application*). Rozwój tych technologii i dedykowanych im narzędzi pozwolił na budowanie coraz bardziej złożonych i zróżnicowanych (technologicznie i architektonicznie) systemów, często silnie ze sobą zintegrowanych. Wymusił także zmianę podejścia do projektowania i wytwarzania tychże systemów. Pojawiło się więc wiele wzorców projektowych czy gotowych zrębów aplikacji (ang. *framework*), niejednokrotnie mających swoje korzenie w klasycznych architekturach (jedno- i dwuwarstwowych) – np. MVC (ang. *Model View Controller*) [4, 11].

Wybór technologii, zalecanych praktyk i architektur aplikacji jest ogromny. Wyłonienie odpowiedniego ich podzbioru jest niemałym wyzwaniem. W dalszej części artykułu zostanie zaprezentowany taki właśnie zbiór technologii związanych z platformą *Adobe Flex,* zapewniający – zdaniem autorów – łatwo rozszerzalną architekturę aplikacji oraz wspomagający programistę w rozwiązywaniu typowych dla aplikacji internetowych problemów realizacyjnych.

2. Rozwój aplikacji internetowych

Swoją popularność aplikacje internetowe zawdzięczają stale rosnącym możliwościom przeglądarek w zakresie obsługi informacji zgromadzonych w bazach danych (w szczególności ich wizualizacji).

Początkowo graficzny interfejs użytkownika (ang. *GUI*) budowany był z dynamicznie generowanych stron WWW, między którymi użytkownik przemieszczał się, klikając w hiperłącza [1, 3]. Same strony były statyczne (pozbawione animacji) i nie mogły reagować na zdarzenia związane z ruchami myszy. W miarę rozwoju technologii, takich jak DHTML, JavaScript czy samych języków HTML i XHTML, możliwe stało się wzbogacanie stron WWW o elementy ruchome, reagujące na poczynania użytkownika. Kolejnym krokiem milowym w rozwoju aplikacji internetowych było wprowadzenie rozwiązań wykorzystujących AJAX (ang. *Asynchronous JavaScript and XML*) [13]. Od tej pory strony WWW mogły asynchronicznie (bez odświeżania całego dokumentu) wymienić dodatkowe dane z serwerem i zmieniać fragment przeglądanego dokumentu. To sprawiło, że wrażenia użytkownika (ang. *user experience*) używającego nowoczesnej aplikacji internetowej czy użytkownika korzystającego ze „zwykłego” programu (aplikacji desktopowej, uruchamianej bezpośrednio w ramach systemu operacyjnego) nie różnią się.

Opisane wyżej technologie, zdaniem autorów, są obecnie u szczytu popularności. Większość serwisów internetowych korzysta z co najmniej kilku z nich. Jednak nie są one pozbawione wad i niedogodności, które wynikają z ograniczeń samego języka HTML, który został zaprojektowany do prezentacji jedynie statycznych treści. Dlatego coraz częściej mamy do czynienia z aplikacjami internetowymi uruchamianymi w przeglądarkach, ale osadzonymi dodatkowo w środowisku uruchomieniowym (ang. *runtime environment*)[1]. Przykładami takich rozwiązań są platformy *Adobe Flash Player* oraz *Microsoft Silverlight*. Ich głównymi zaletami są duże możliwości wizualizacji danych (2D i 3D) oraz zapewnienie tego samego wyglądu i zachowania aplikacji niezależnie od systemu operacyjnego czy samej przeglądarki. Rozwiązanie firmy *Adobe* jest zdecydowanie bardziej popularne, starsze, lepiej dopracowane (odpowiednia wtyczka zainstalowana jest w ok. 97% przeglądarek) niż odpowiednik firmy Microsoft (ok. 56%) [6, 7] i dlatego jest chętnie wybierane jako platforma do tworzenia nowoczesnych aplikacji internetowych.

3. Architektura aplikacji opartej na Adobe Flex

Zaawansowany interfejs graficzny aplikacji internetowej jest widziany przez użytkownika jako całość. Jednak od strony twórcy i projektanta jest to skomplikowana „układanka" widoków, komponentów, zasobów graficznych itp. Odpowiednia organizacja tych elementów zapewnia łatwość rozwijania aplikacji, ale także jej niezawodność i spójność, zarówno jeśli chodzi o zachowanie, jak i wygląd. Osiągnięcie tego celu ułatwia stosowanie odpowiednich zrębów aplikacji (ang. *frameworks*) czy wzorców projektowych (ang. *design patterns*).

Firma *Adobe* dla swojej platformy *Adobe Flash Player* stworzyła zestaw narzędzi i technologii do tworzenia zaawansowanych aplikacji internetowych pod nazwą *Adobe Flex*. W jej skład wchodzą: język programowania *Action Script 3*, biblioteka standardowych klas i komponentów (*Flex SDK*) oraz zintegrowane środowisko programistyczne (ang. *IDE*) *Adobe Flex Builder* [1, 9].

Adobe Flex dostarcza klasy i komponenty najniższego poziomu. Ich organizacja w bardziej złożone struktury jest zadaniem odpowiedniego *frameworka*. Dla tej platformy powstało kilka takich rozwiązań: *Cairngorm* [8]*, Mate* [14]*, PureMVC, Swiz* [15].

3.1. Framework PureMVC

Do tworzenia zaawansowanych aplikacji biznesowych bardzo dobrze nadaje się *PureMVC* [5]. *Framework* ten jest nieskomplikowany, łatwy w adaptacji, a ponadto nie wymaga

[1] Środowisko uruchomieniowe instalowane jest jako wtyczka (ang. *plug-in*) do przeglądarki internetowej.

pisania nadmiarowego kodu. Wykorzystano w nim ugruntowany wśród projektantów systemów informatycznych wzorzec architektoniczny Model – Widok – Kontroler (ang. *Model – View – Controller*) oraz wybrane wzorce projektowe GoF: fasada, komenda, *proxy* (pośrednik), *singleton*, mediator [11].

W tym *frameworku* Model, Widok i Kontroler implementują wzorzec *singleton* i przechowują instancje odpowiednio: obiektów *proxy*, mediatorów oraz komend.

Obiekty *proxy* są odpowiedzialne za pobieranie i przechowywanie danych ze zdalnych źródeł, tj. baz danych, usług sieciowych itp. Pozyskane w ten sposób dane prezentowane są na widokach tworzących graficzny interfejs użytkownika. Widoki są zarządzane przez obiekty mediatorów. Za komunikację pomiędzy poszczególnymi obiektami a instancjami Modelu, Widoku i Kontrolera odpowiada obiekt implementujący wzorzec fasady (który także jest *singletonem*). Zastosowanie wzorca *obserwator* umożliwia przesyłanie komunikatów pomiędzy obiektami aplikacji, w sposób, który zapewnia ich luźne powiązanie (ang. *loose coupling*).

3.2. Wspomaganie współdzielenia usług między modułami aplikacji

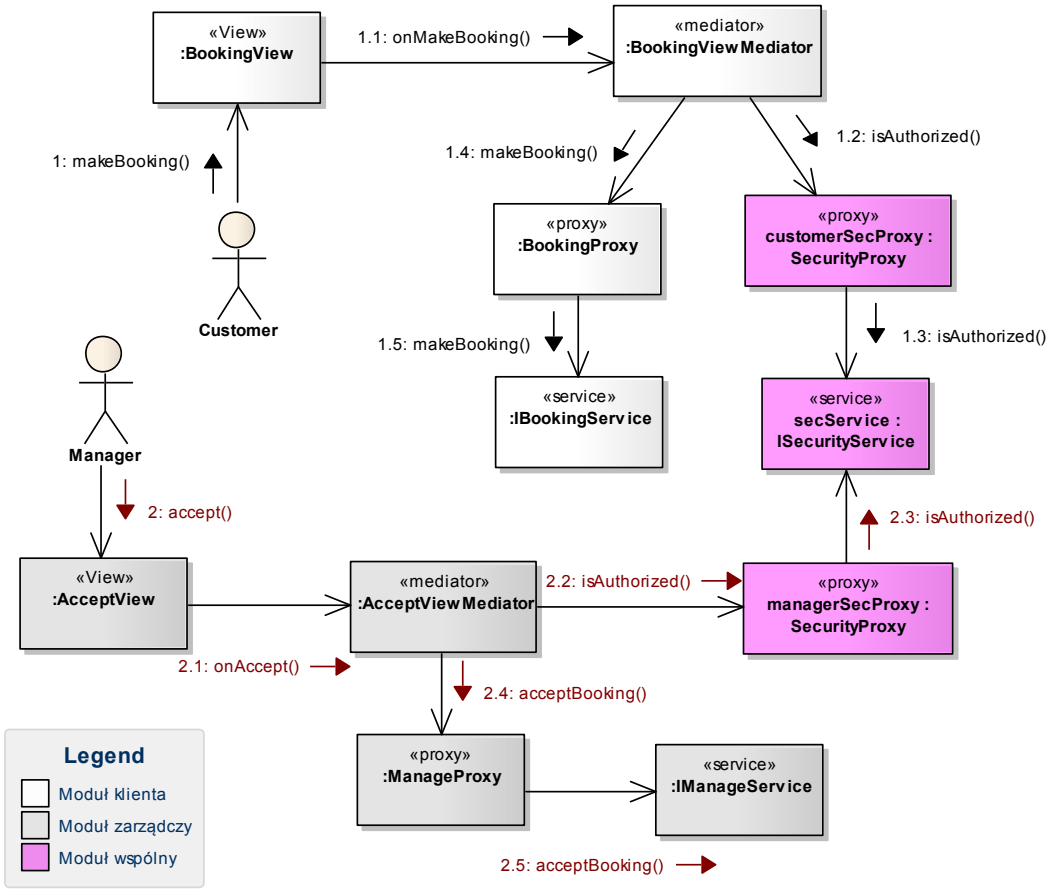
Mechanizmy zaimplementowane w tym *frameworku* pomagają stworzyć zaawansowaną aplikację, jednak nie wspomagają programisty w rozwiązywaniu często spotykanych problemów, jak np. komunikacja między modułami, obsługa stanów aplikacji itp. Użycie wybranych, dodatkowych komponentów może tę sytuację zmienić. *PureMVC* jest dostępny w zmodyfikowanej wersji, dedykowanej dla aplikacji składających się z wielu modułów (ładowanych dynamicznie, tzn. w trakcie działania aplikacji).

W koncepcji twórców tego rozwiązania, moduły stanowią odseparowane, niezależne części aplikacji, które mogą się ze sobą komunikować (np. przez zdarzenia czy komunikaty), ale nie mogą współdzielić swoich usług. Każdy moduł ma swoją instancję Modelu, Widoku i Kontrolera, w których przechowywane są obiekty *proxy*, mediatory czy komendy – nie można jednak w module A pobrać instancji obiektu *proxy* z modułu B. W typowych aplikacjach taka sytuacja jest niedopuszczalna – np. kod odpowiedzialny za komunikację ze zdalnymi usługami na serwerze aplikacji powinien być współdzielony, dzięki czemu można w prosty sposób kontrolować zabezpieczenia czy przeprowadzać audyt. Istnieje więc potrzeba rozszerzenia wspomnianych rozwiązań o własne elementy.

W przypadku *PureMVC* w wersji *MultiCore* wspomniany wyżej problem można rozwiązać, wprowadzając wzorzec *ServiceLocator* [10] i jawnie specyfikując obiekty usług (ang. *services*). Przedstawiony na rys. 1 diagram kooperacji pokazuje schemat wywołania zdalnych metod i współdzielenie kodu usług przez kilka modułów.

Rysunek pokazuje, w jaki sposób współpracują z sobą poszczególne komponenty *PureMVC*. Mediatory (obiekty oznaczone stereotypem <<*mediator*>>) reagują na akcje użytkownika podejmowane na widokach (obiekty oznaczone stereotypem <<*view*>>)

i wykorzystują obiekty *proxy* (oznaczone stereotypem <<*proxy*>>) do komunikacji z serwerem aplikacji. Obiekty *proxy* wewnętrznie korzystają z instancji obiektów usług (obiekty oznaczone stereotypem <<*service*>>), które pobierają z obiektu implementującego wzorzec *ServiceLocator*. Na rysunku widać, że obiekty klasy *SecurityProxy* są tworzone w każdym z modułów aplikacji, ale korzystają ze wspólnej instancji obiektu *SecurityService* (wzorzec *Singleton*, [10, 11]). W tym obiekcie znajduje się kod odpowiedzialny za komunikację z serwerem, stosowanie zabezpieczeń, sprawdzanie uprawnień itp. Taka architektura pozwala na korzystanie w różnych modułach aplikacji ze wspólnych usług, zgodnie z zasadami działania *frameworka PureMVC*.



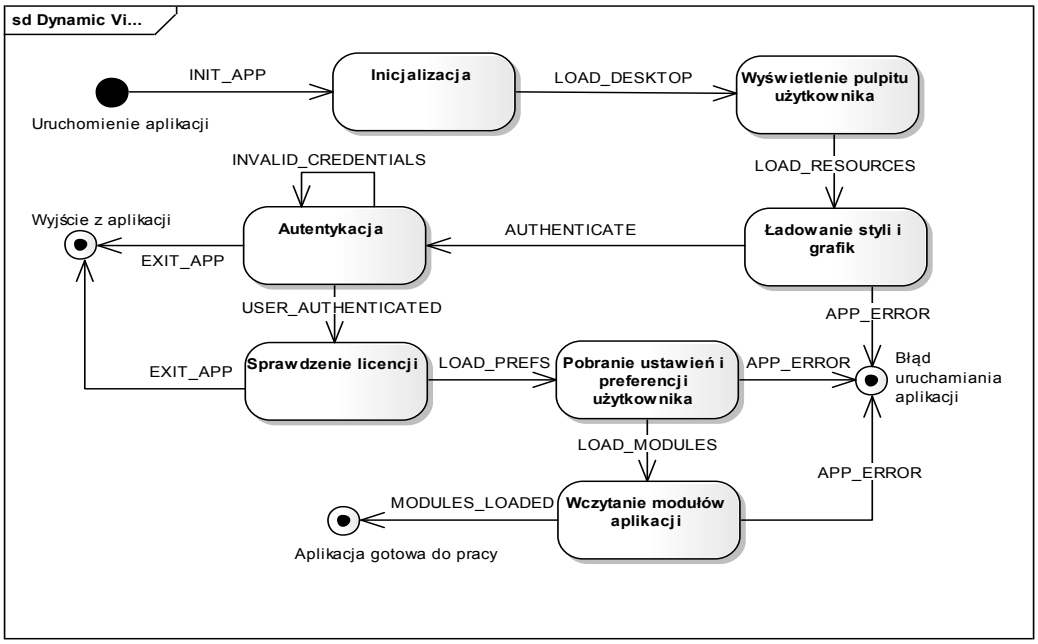
Rys. 1. Kooperacja obiektów podczas wywoływania zdalnych metod. Współdzielenie obiektów usług

Fig. 1. Cooperation between objects during remote method call. Common services are shared

3.3. Wybrane rozszerzenia

Proces uruchamiania złożonej aplikacji biznesowej składa się zazwyczaj z wielu następujących po sobie kroków. Kolejność ich wykonywania jest istotna, a niepowodzenie na którymkolwiek etapie skutkuje albo przerwaniem całego procesu, albo wykonaniem dodatko-

wych zadań. Na etapie analizy i projektowania modeluje się ten aspekt działania aplikacji za pomocą np. diagramu stanów. Rysunek 2 pokazuje taki przykładowy diagram.



Rys. 2. Diagram stanów, opisujący proces uruchamiania aplikacji
Fig. 2. State diagram describing application startup process

Na etapie implementacji diagram znajduje swoje odzwierciedlenie w klasach programu i powiązaniach między nimi. Klasyczny kod programu – niezależnie od języka programowania – nie wyraża bezpośrednio zaprojektowanej „stanowości" całego procesu uruchamiania. Użycie *PureMVC* oraz dodatku *AS3 StateMachine*, który dostarcza implementację maszyny stanów skończonych, umożliwia wprost odwzorowanie diagramu stanów w kodzie aplikacji. Odbywa się to przez zdefiniowanie odpowiedniego dokumentu XML, którego fragment przedstawiono poniżej[2].

```
var fsm:XML =
   <fsm initial={StartupStates.INIT}>
      <state name={StartupStates.INIT}
         entering={Notifications.LOAD_DESKTOP}>
         <transition action={ShellNotifications.USER_DESKTOP_READY}
            target={StartupStates.RESOURCES}/>
      </state>
      <state name={StartupStates.RESOURCES}
         entering={Notifications.LOAD_RESOURCES}>
         <transition action={Notifications.RESOURCE_LOADED}
            target={StartupStates.AUTH}/>
         <transition action={Notifications.LOADING_RESOURCES_FAILED}
            target={StartupStates.STARTUP_FAILURE}/>
      </state>
      <state name={StartupStates.AUTH}
         entering={Notifications.SHOW_LOGIN_VIEW}>
         <transition action={Notifications.USER_AUTHENTICATED}
            target={StartupStates. LICENCE}/>
         <transition action={Notifications.EXIT_APP}
```

[2] Język *ActionScript 3* dopuszcza definiowanie dokumentów XML wprost w kodzie programu.

```
            target={StartupStates.EXIT}/>
        </state>
        <!-- ... -->
        <state name={StartupStates.STARTUP_FAILURE}
           entering={Notifications.SHOW_STARTUP_FAILED}/>
</fsm>;
```

Nazwy przejść między stanami odpowiadają wprost nazwom komunikatów (zwanych w nomenklaturze *PureMVC* notyfikacjami) zgłaszanych przy wejściu do każdego ze stanów. Z każdą notyfikacją związana jest odpowiednia komenda, odpowiedzialna za wykonanie zadań wymaganych w danym momencie uruchamiania aplikacji.

Dodatek *AS3 State Machine* może być oczywiście użyty do sterowania realizacją innego procesu w aplikacji (w szczególności np. do uruchamiania konkretnego modułu aplikacji).

Framework PureMVC jest rozwiązaniem prostym i efektywnym, ale mimo jego użycia istnieją miejsca w tworzonym oprogramowaniu, gdzie wymagane jest pisanie wielokrotnie tego samego lub bardzo podobnego kodu. Problem ten jest typowy dla większości *frameworków* – stosując te same wzorce w wielu miejscach powtarzamy tylko nieznacznie zmodyfikowane bloki kodu.

Dobrze znanym rozwiązaniem tego zagadnienia jest zastosowanie podejścia: konwencja ponad konfigurację (ang. *convention over configuration*). Polega ono na zachowaniu dodatkowych konwencji (np. nazewniczych), dzięki czemu pewne czynności mogą zostać zautomatyzowane przez wewnętrzne mechanizmy *frameworka*, co w efekcie pozwala uniknąć powtarzania tego samego kodu. Na bazie tego podejścia powstało rozszerzenie *Fabrication* dla *PureMVC*. Wymaga ono stosowania odpowiednich konwencji nazewniczych dla metod czy zmiennych, ale dzięki temu automatyzuje np. rejestrowanie metod nasłuchujących na wybrane notyfikacje (zapewnia, że w mediatorze wywoła się odpowiednia metoda, której sygnatura jest zgodna z nazwą notyfikacji). Przykład stosowania ww. konwencji przedstawiono poniżej.

```
// w klasie proxy
//…
public function result(event:Object) : void
{
    //…
    sendNotification(„dataLoaded”, event.result);
}

//w klasie mediatora
//…
public function respondToDataLoaded(note:INotification):void
{
    //…
    view.data = note.getBody();
}
```

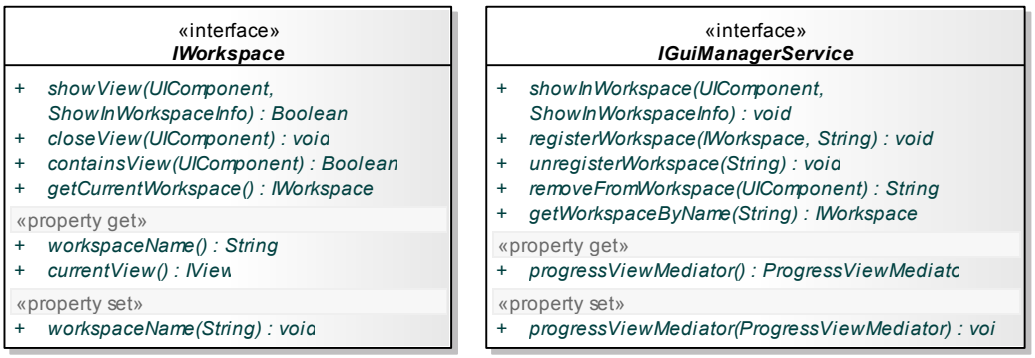
Fabrication umożliwia także wykorzystanie koncepcji tzw. wstrzykiwania do komponentów ich zależności (ang. *dependency injection*), czyli zewnętrznego (poza kodem programu) tworzenia powiązań obiektów, co w konsekwencji także zmniejsza ilość nadmiarowego kodu.

3.4. Warstwa abstrakcji GUI

W klasycznej aplikacji uruchamianej w systemie operacyjnym użytkownik otwiera kolejne okna, za pomocą których realizuje swoje cele. Bogate aplikacje internetowe (ang. *RIA*) różnią się w tym zakresie. Nie otwierają nowych okien przeglądarki, ale dynamicznie przebudowują swój interfejs (wewnątrz tego samego okna przeglądarki), dając użytkownikowi wrażenie, że otworzył kolejne okno czy przeszedł do następnej strony.

Środowisko *Adobe Flex* wspiera programistę/projektanta w tworzeniu takich interfejsów, dając mu do dyspozycji bogaty zestaw kontrolek czy komponentów, dbających o ich prawidłowe rozmieszczenie (tzw. *layout containers*). Możliwe jest także wprowadzanie stanów danego elementu wizualnego, co ułatwia np. kontekstową zmianę widoczności wybranych jego fragmentów. W przypadku prostych aplikacji, opisane wyżej mechanizmy są w zupełności wystarczające. Jednak, gdy mamy do czynienia ze złożonym systemem, składającym się z większej liczby modułów, a co za tym idzie i z dziesiątek czy setek widoków, konieczne jest wprowadzenie dodatkowej warstwy abstrakcji, pozwalającej skutecznie zarządzać organizacją graficznego interfejsu użytkownika. Niestety ani środowisko *Adobe Flex*, ani *framework PureMVC* nie dostarczają takich mechanizmów. Konieczne jest więc zaproponowanie własnych. Poniżej opisano proponowane przez autorów rozwiązanie tego problemu, bazujące na koncepcji przedstawionej w [12].

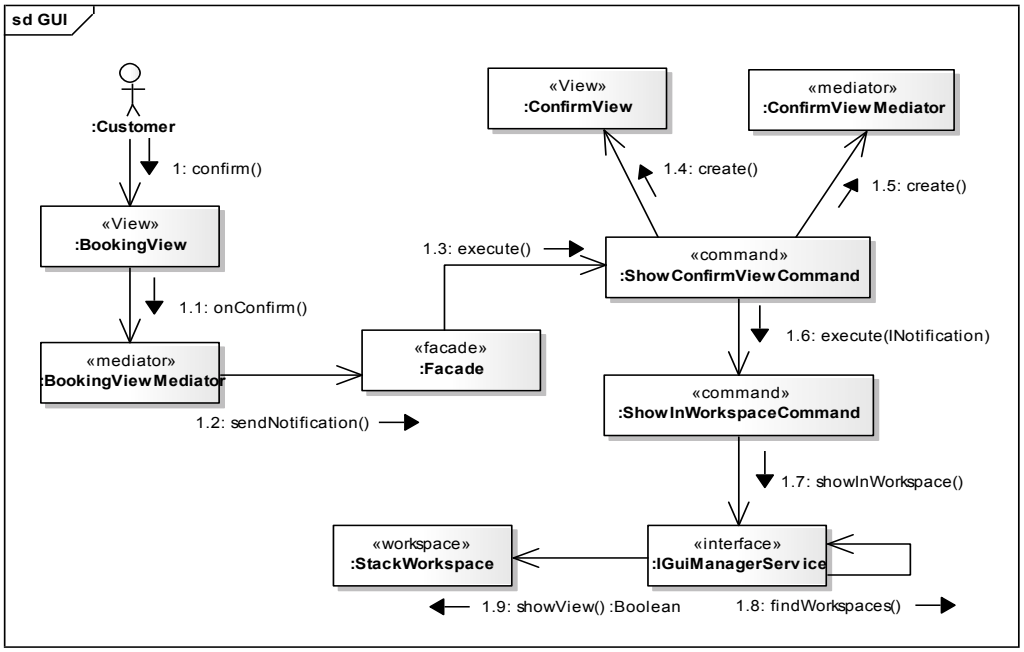
W większości aplikacji ekran podzielony jest na logiczne fragmenty, które pełnią specyficzną rolę, np. po lewej stronie znajduje się lista przycisków uruchamiających funkcje programu, a po prawej pojawiają się kolejne widoki, realizujące wybrany przypadek użycia. Fragmenty te, zwane dalej obszarami roboczymi (ang. *workspace*), służą do zbudowania szkieletu GUI aplikacji. Każdy obszar roboczy odpowiedzialny jest za wyświetlenie jednego lub kilku widoków. Od konkretnej jego implementacji zależy, w jaki sposób to zrobi (np. w postaci stosu, kolejnych zakładek czy obracalnego sześcianu). Z każdym obszarem związana jest jego unikatowa nazwa. Rysunek 3a przedstawia interfejs *IWorkspace*, specyfikujący zachowanie obszaru roboczego.



Rys. 3. Specyfikacja interfejsów *IWorkspace* i *IGuiManager*
Fig. 3. *IWorkspace* and *IGuiManager* interfaces specification

Utrzymywaniem listy dostępnych obszarów roboczych zajmuje się obiekt implementujący interfejs *IGuiManager* (rys. 3b). Jego zadaniem jest także umieszczanie widoków w obszarach roboczych i ich usuwanie, gdy nie są już więcej potrzebne. Każdy widok może zawierać w sobie jeden lub kilka obszarów roboczych, tak więc podczas ich wyświetlania *IGuiManager* wyszukuje je i udostępnia innym komponentom aplikacji, a podczas zamykania widoku niszczy także zawarte w nich obszary robocze i wyświetlone w nich widoki.

Opisane wyżej rozwiązanie można łatwo zintegrować z komponentami *PureMVC*, co pokazano na rys. 4. Kiedy w reakcji na akcję użytkownika ma zostać wyświetlony nowy widok, w mediatorze wysyłana jest odpowiednia notyfikacja. Z nią związana jest komenda, której zadaniem jest utworzenie instancji nowego widoku (oraz dedykowanego mu mediatora), a następnie wywołanie komendy *ShowInWorkspaceCommand* z parametrem określającym nazwę obszaru roboczego, w którym ma pojawić się nowy widok. Wyżej wymieniona komenda pobiera instancję usługi *IGuiManager* i zleca jej wyświetlenie widoku. Ta z kolei odnajduje żądany obszar roboczy (obiekt klasy *StackWorkspace*) i umieszcza w nim widok.



Rys. 4. Kooperacja komponentów *PureMVC* podczas wyświetlania widoku
Fig. 4. PureMVC component collaboration when showing view

Stosowanie opisanego wyżej podejścia ma wiele zalet. Po pierwsze umożliwia wielokrotne wykorzystywanie tych samych widoków w różnych miejscach aplikacji (ang. *reusability*), co skraca czas potrzebny na wytworzenie aplikacji i zapewnia spójność jej interfejsu graficznego. Po drugie otrzymujemy jasny podział odpowiedzialności – separujemy od siebie kod obsługujący akcje użytkownika, od kodu tworzącego widoki i wreszcie od sposobu ich wyświetlania. Dzięki takiemu podziałowi łatwiej wytwarza się złożone aplikacje, nad którymi pracuje kilka zespołów programistów. Podział interfejsu na mniejsze, niezależne elementy, ułatwia implementację testów jednostkowych.

4. Podsumowanie

Możliwości współczesnych przeglądarek internetowych powodują przenoszenie coraz bardziej skomplikowanych aplikacji do sieci WWW. Technologie takie jak AJAX, zmieniające zwykłą stronę HTML w pełni funkcjonalną aplikację, niczym nieróżniącą się od tej uruchamianej w systemie operacyjnym (ang. *desktop application*), są u szczytu popularności. Pojawiają się także i nowe technologie, o większych możliwościach wizualizacji prezentowanych danych (np. w 3D). Przykładem jest platforma *Adobe Flex*, bazująca na popularnym rozszerzeniu przeglądarek internetowych – *Adobe Flash Player*. Osiągnęła ona już poziom dojrzałości, którego cechą charakterystyczną jest pojawienie się dużej liczby w pełni profesjonalnych szablonów, zwanych *frameworkami*, określających architekturę tworzonych za pomocą tej platformy aplikacji.

W niniejszym artykule zaprezentowano jeden z nich – *PureMVC*. Przedstawiono jego wewnętrzną strukturę oraz pokazano sposób komunikacji między składnikami *frameworka*. Zaproponowano także wybrane rozszerzenia, zwiększające jego użyteczność i ułatwiające rozwiązanie typowych problemów, występujących przy projektowaniu i wytwarzaniu złożonych systemów. Na przykładzie zaproponowanej warstwy abstrakcji GUI pokazano, w jaki sposób można go zintegrować z własnym rozwiązaniem.

Możliwość wykorzystania dobrze znanych i ugruntowanych wzorców projektowych oraz zrębów architektury ułatwia proces projektowania i wprowadzania zmian w aplikacji. Usprawnia także organizację pracy zespołu wytwarzającego.

W artykule, na podstawie doświadczeń projektowych i implementacyjnych, autorzy wskazują na wybór *frameworka PureMVC* w wersji *Mulicore* jako optymalnego dla platformy *Adobe Flex*. Dodatkowo w pracy omówiono zaproponowane rozszerzenia, związane ze współdzieleniem usług między moduły aplikacji (rozdział 3.2) oraz sposobem obsługi GUI (rozdział 3.4).

Platforma *Adobe Flex* w połączeniu z *frameworkiem PureMVC* oraz omówionymi rozszerzeniami może z powodzeniem stanowić narzędzie do wytworzenia nawet najbardziej zaawansowanej aplikacji typu *RIA*.

BIBLIOGRAFIA

1. Tapper J., Labriola M., Boles M., Talbot J.: Adobe® Flex 3. Oficjalny Podręcznik, Helion, Gliwice 2010.
2. Fain Y., Rasputnis V., Tartakovsky A.: Enterprise Development with Flex. O`Reilly Media, USA 2010.

3. Shklar L., Rosen R.: Web Application Architecture: Principles, Protocols and Practices. Wiley, Anglia 2003.
4. Burbeck S.: Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC), http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html.
5. PureMVC Homepage, http://puremvc.org/content/view/67/178/.
6. Flash Player Version Support, http://www.statowl.com/flash.php.
7. Silverlight Market Penetration, http://www.statowl.com/silverlight.php.
8. Wischusen J.: Professional Cairngorm. Wiley Publishing, USA 2010.
9. Gassner D.: Flash Builder 4 and Flex 4 Bible. Wiley Publishing, USA 2010.
10. Alur D., Crupi J., Malks D.: J2EE. Wzorce projektowe. Wydanie 2. Helion, Gliwice 2004.
11. Gamma E., Helm R., Johnson R., Vlissides J.: Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku. WNT, Warszawa 2005.
12. Smart Client – Composite UI Application Block, http://msdn.microsoft.com/en-us/library-/aa480450.aspx.
13. Garrett J. J.: Ajax: A New Approach to Web Applications, http://www.adaptivepath.com/ideas/essays/archives/000385.php.
14. Mate Framework Homepage, http://mate.asfusion.com/.
15. Swiz Framework Homepage, http://swizframework.org/.

Recenzent: Prof. dr hab. inż. Andrzej Grzywak

Wpłynęło do Redakcji 31 stycznia 2011 r.

Abstract

In this paper authors present how *Adobe Flex* platform with *PureMVC* application framework and some of its extensions can be used to design and develop rich internet applications.

In chapter 2 authors describe evolution of rich internet applications in recent years. In the next chapter, the architecture of a rich internet application is presented. Fig. 1 shows how components interact with each other to obtain data from a remote server in reaction to a user action from GUI. It also presents some concepts and design patterns used in *PureMVC*.

Next, authors show how to avoid problems of sharing common services in a multimodule application when the *Multicore* version of this framework is used. Last chapter introduces the custom solution for organizing and managing user interface in a RIA application. This proposed solution is fully integrated with *PureMVC* components.

Adresy

Dariusz R. AUGUSTYN: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, dariusz.augustyn@polsl.pl.

Łukasz WARCHAŁ: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, lukasz.warchal@polsl.pl.