Ajax w akcji

Autorzy: Dave Crane, Eric Pascarello, Darren James
ISBN: 83-246-0495-2
Tytuł oryginału: Ajax in Action
Format: B5, stron: 712

Praktyczne zastosowania technologii, która zrewolucjonizowała aplikacje sieciowe

- Podstawy Ajaksa
- Metodologie tworzenia aplikacji internetowych
- Optymalizacja i zabezpieczanie aplikacji

Ajax to technologia tworzenia aplikacji i witryn WWW, która zyskuje coraz większe uznanie wśród programistów. Dzięki połączeniu języków JavaScript i XML z asynchroniczną komunikacją między przeglądarką i serwerem twórcom tej technologii udało się wyeliminować podstawową wadę aplikacji bazujących na przeglądarce internetowej, czyli konieczność oczekiwania na „przeładowanie" strony. Ajax sprawia, że niezbędne dane są pobierane w tle. Zastosowanie tej technologii powoduje, że aplikacje sieciowe pod względem obsługi coraz bardziej przypominają tradycyjne programy. Efekty działania Ajaksa można zaobserwować na coraz większej ilości witryn WWW.

„Ajax w akcji" to praktyczny przewodnik po zagadnieniach związanych z projektowaniem witryn i aplikacji WWW w tej technologii. Czytając go, poznasz genezę Ajaksa i podstawy korzystania z niego. Nauczysz się stosować wzorce projektowe, model MVC oraz frameworki i biblioteki wspomagające tworzenie witryn WWW. Przeczytasz o profesjonalnych technikach budowania aplikacji sieciowych i zastosujesz je w praktyce, pisząc własne systemy. Znajdziesz tu również informacje o optymalizowaniu wydajności aplikacji oraz tworzeniu mechanizmów zabezpieczeń. Dzięki przykładom zgromadzonym w kolejnych rozdziałach książki, takim jak dynamiczna lista rozwijana, mechanizm podpowiedzi, rozbudowany portal sieciowy oraz interfejsy użytkownika, poznasz praktyczne aspekty stosowania Ajaksa w projektach.

- Podstawowe zastosowania Ajaksa
- Dostosowywanie istniejących aplikacji do Ajaksa
- Wzorzec MVC w aplikacjach
- Biblioteki i frameworki wspomagające pracę nad aplikacjami
- Oddzielanie logiki od widoku
- Wymiana danych pomiędzy przeglądarką i serwerem
- Zabezpieczanie aplikacji
- Optymalizacja wydajności

Twórz aplikacje, które będą nie tylko funkcjonalne, ale również wygodne w obsłudze

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

3 Porządkowanie aplikacji wykorzystujących technologię Ajax

Zagadnienia opisywane w tym rozdziale:

- Tworzenie i pielęgnacja rozbudowanego kodu wykorzystującego możliwości technologii Ajax.
- Refaktoryzacja kodu JavaScript wykorzystującego technologię Ajax.
- Wzorce projektowe najczęściej używane w aplikacjach wykorzystujących Ajaksa.
- Stosowanie wzorca projektowego MVC w aplikacjach wykonywanych po stronie serwera.
- Przegląd dostępnych bibliotek ułatwiających korzystanie z technologii Ajax.

W rozdziale 2. opisaliśmy wszystkie podstawowe technologie, które ogólnie określamy jako **Ajax**. Dysponując wiedzą, jaką już zdobyliśmy, możemy napisać niesamowitą aplikację wykorzystującą technologię Ajax, o jakiej zawsze marzyliśmy. Równie dobrze możemy jednak wpaść w bardzo poważne kłopoty i skończyć na całej masie złożonego kodu JavaScript, HTML i arkuszy stylów, których utrzymanie i pielęgnacja są praktycznie niemożliwe i który, z jakichś całkowicie niewyjaśnionych przyczyn, pewnego dnia może przestać działać. Albo może być jeszcze gorzej — możemy stworzyć aplikację działającą tak długo, jak długo nie będziemy przy niej nic robić, oddychać ani wydawać żadnych głośniejszych dźwięków. Znalezienie się w takiej sytuacji podczas prac nad naszym własnym, prywatnym projektem może być jedynie zniechęcające. Jeśli jednak znajdziemy się w takiej sytuacji podczas modyfikacji witryny pracodawcy lub klienta, który płaci za jej utrzymanie i zażądał właśnie wprowadzenia kilku zmian tu i tam, to bez wątpienia będzie to przerażające.

Na szczęście takie problemy występowały powszechnie już od samego początku ery komputerów, a można sądzić, że nawet wcześniej! Wiele osób łamało sobie głowy nad znalezieniem sposobu zarządzania złożonością oraz utrzymaniem coraz większych ilości kodu źródłowego w takim porządku, który umożliwiałby jego dalsze rozwijanie i pielęgnację. W tym rozdziale przedstawimy podstawowe narzędzia, które pozwolą nam zachować orientację w tym, co dzieje się w naszym kodzie, pisać aplikacje wykorzystujące technologię Ajax, jakie spełnią oczekiwania naszych klientów, a jednocześnie sprawią, że będziemy mogli wracać do domu o normalnej porze.

Ajax stanowi przełom w porównaniu z wcześniej stosowanymi technologiami DHTML i to nie tylko pod względem sposobu połączenia poszczególnych technologii, lecz także skali, na jaką są one stosowane. Aplikacje wykorzystujące technologię Ajax w znacznie większym stopniu używają skryptów JavaScript, a ich kod jest przechowywany w przeglądarce znacznie dłużej. W efekcie muszą one rozwiązywać znacznie poważniejsze problemy ze złożonością, które nie występowały we wcześniejszych aplikacjach DHTML.

W tym rozdziale przedstawimy narzędzia i techniki, które pozwolą Czytelnikowi zachować przejrzystość i prostotę tworzonego kodu. Z naszego doświadczenia wynika, iż są one najbardziej skuteczne w przypadku tworzenia dużych i złożonych aplikacji wykorzystujących technologię Ajax. W przypadku pisania prostych aplikacji sugerujemy pominąć tę część książki i przejść bezpośrednio do rozdziałów prezentujących konkretne rozwiązania, czyli do rozdziału 9. i kolejnych. Jeśli Czytelnik już doskonale zna zasady refaktoryzacji oraz wszystkie wzorce projektowe, to może pominąć ten rozdział i zająć się lekturą rozdziałów opisujących zastosowanie tych technik w aplikacjach wykorzystujących technologię Ajax (czyli w rozdziałach od 4. do 6.). Niemniej jednak podstawowe informacje, jakie prezentujemy w tym rozdziale, mają duże znaczenie dla zapewniania wysokiej jakości kodu JavaScript, dlatego też przypuszczamy, że wcześniej lub później Czytelnik i tak tu zajrzy. Skorzystamy także z nadarzającej się sposobności i pod koniec rozdziału przedstawimy dostępne biblioteki ułatwiające korzystanie z technologii Ajax; a zatem jeśli Czytelnik szuka biblioteki lub frameworka, który mógłby mu ułatwić pracę nad projektem, to może zajrzeć do podrozdziału 3.5.

3.1. Porządkowanie chaosu

Podstawowym narzędziem, jakie zastosujemy, będzie tzw. **refaktoryzacja** — czyli proces polegający na przepisywaniu kodu lub wprowadzaniu w nim modyfikacji, które nie mają na celu rozbudowy jego możliwości funkcjonalnych, lecz poprawienie jego przejrzystości. Poprawianie przejrzystości samo w sobie może być doskonałym powodem do wprowadzania zmian w kodzie, niemniej jednak ma ono także i inne zalety, które powinny przekonać każdego.

Zazwyczaj znacznie łatwiej jest dodawać nowe możliwości do opracowanego wcześniej kodu, modyfikować już zaimplementowane lub usuwać je. Najprościej rzecz ujmując, takie operacje są zrozumiałe. Jeśli jednak kod aplikacji nie zostanie dobrze opracowany i zaprojektowany, to chociaż będzie on poprawnie funkcjonował i spełniał wymagania zleceniodawcy, to jednak osoby pracujące nad nim mogą nie do końca rozumieć, jak on działa oraz dlaczego działa poprawnie.

Zmiany wymagań aplikacji, często połączone z krótkim okresem realizacji, są przykrą codziennością dla programistów. Refaktoryzacja pozwala na zapewnienie przejrzystości kodu, ułatwia jego utrzymanie i pozwala, by perspektywa implementacji zmian w założeniach projektu nie napawała nas przerażeniem.

Podstawowe sposoby refaktoryzacji poznaliśmy już w rozdziale 2., przy okazji przenoszenia kodu JavaScript, HTML oraz definicji stylów do osobnych plików. Jednak kod JavaScript staje się już długi, gdy objętość pliku przekroczy około 120 wierszy oraz gdy zaczniemy w nim stosować możliwości funkcjonalne niskiego poziomu (takie jak generowanie żądań i przesyłanie ich na zdalny serwer) z kodem operującym na obiektach. Kiedy zaczniemy tworzyć większe projekty, umieszczanie całego kodu JavaScript w jednym pliku przestanie być dobrym rozwiązaniem (to samo zresztą dotyczy arkuszy stylów). Cel, do którego dążymy — tworzenie niewielkich, przejrzystych, zrozumiałych i łatwych do modyfikacji fragmentów kodu obsługujących konkretne zagadnienia — jest często określany jako **separacja obowiązków** (ang. *separation of responsibility*).

Jednak refaktoryzacja jest często przeprowadzana także z innego powodu, pozwala ona bowiem na wskazanie często stosowanych rozwiązań, sposobów realizacji konkretnych zadań i tworzenie kodu na podstawie określonych wzorców. Także te cele są dostateczną motywacją do wprowadzania zmian w kodzie, a dążenie do nich ma bardzo praktyczne konsekwencje. Przedstawimy je w kolejnym punkcie rozdziału.

3.1.1. Wzorce: tworzenie wspólnego słownictwa

Prawdopodobieństwo, że kod spełniający założenia jakiegoś dobrze znanego i powszechnie stosowanego wzorca będzie działać poprawnie, jest bardzo duże. Wynika to z prostego faktu, iż taki kod był już wielokrotnie tworzony i stosowany. Wiele problemów, jakie występują podczas tworzenia konkretnego rozwiązania zostało już dokładnie przemyślanych i, miejmy nadzieję, skutecznie rozwiązanych. A jeśli mamy szczęście, to może się okazać, że ktoś już napisał nadający się do wielokrotnego stosowania kod, realizujący dokładnie te możliwości funkcjonalne, o jakie nam chodzi.

Taki sposób wykonywania konkretnego zadania nazywamy **wzorcem projektowym**. Pojęcie wzorca pojawiło się w latach 70. ubiegłego wieku w odniesieniu do sposobów rozwiązywania problemów występujących w architekturze i planowaniu. Później zapożyczono je także do świata programowania komputerów, gdzie z powodzeniem jest stosowane od blisko dziesięciu lat. Wzorce projektowe są bardzo popularne w środowisku programistów wykorzystujących język Java, a ostatnio firma Microsoft usilnie stara się wprowadzić je także w platformie .NET. Termin „wzorzec projektowy" bardzo często jest kojarzony z czymś akademickim, teoretycznym i odpychającym; równie często nadużywa się go lub stosuje błędnie, by wywrzeć większe wrażenie na rozmówcach. Jednak wzorzec projektowy jest jedynie opisem sposobu rozwiązania pewnego konkretnego problemu, który bardzo często występuje podczas projektowania i tworzenia oprogramowania. Warto zauważyć, iż wzorce projektowe określają i nadają nazwy abstrakcyjnym, technicznym rozwiązaniom, dzięki czemu nie tylko łatwiej można o nich rozmawiać, lecz także łatwiej jest je zrozumieć.

Wzorce projektowe mogą być ważne w kontekście refaktoryzacji kodu, gdyż pozwalają nam na zwięzłe i precyzyjne opisanie celów, do jakich dążymy. Stwierdzenie, iż chodzi nam o „zaimplementowanie, w formie obiektów tych fragmentów kodu, które realizują proces obsługi konkretnej czynności wykonywanej przez użytkownika" jest długie, złożone i raczej trudno się na nim skoncentrować podczas wprowadzania zmian w kodzie. Jeśli jednak stwierdzimy, że chcemy zastosować wzorzec projektowy Command, to nasz cel nie tylko będzie bardziej precyzyjnie określony, lecz także łatwiej będzie nam o nim rozmawiać.

Jeśli Czytelnik zna się na pisaniu w Javie programów wykonywanych po stronie serwera bądź jest architektem oprogramowania, który zna się na rzeczy, to zapewne zacznie się zastanawiać, cóż takiego odkrywczego jest w tym, co przed chwilą napisaliśmy. Z kolei, jeśli Czytelnik wywodzi się z kręgów projektantów stron WWW lub osób zajmujących się nowymi rodzajami mediów, to pewnie pomyśli, że jesteśmy tymi natchnionymi dziwakami, którzy wolą rysowanie śmiesznych schematów i diagramów od tworzenia kodu. Jednak niezależnie od tego, kim jest Czytelnik i jakie ma doświadczenia, to mógł pomyśleć, co to wszystko ma wspólnego z aplikacjami wykorzystującymi technologię Ajax. Otóż ma, i to całkiem sporo. Przekonajmy się zatem, co programista tworzący aplikacje wykorzystujące Ajaksa może zyskać dzięki refaktoryzacji.

3.1.2. Refaktoryzacja a Ajax

Wspominaliśmy już, że aplikacje wykorzystujące technologię Ajax będą zazwyczaj zawierać więcej kodu JavaScript oraz że ten kod będzie dłużej działać w środowisku przeglądarki niż w tradycyjnych aplikacjach internetowych.

W tradycyjnych aplikacjach internetowych najbardziej złożony kod jest wykonywany na serwerze, a wzorce projektowe są używane przy pisaniu skryptów i aplikacji w językach PHP, Java oraz językach używanych na platformie .NET. Jednak w aplikacjach wykorzystujących technologię Ajax te same techniki możemy zastosować podczas tworzenia kodu wykonywanego po stronie przeglądarki.

Można nawet podać argument sugerujący, iż zastosowanie takiego sposobu organizacji kodu jest znacznie bardziej potrzebne w języku JavaScript niż w językach o rygorystycznej kontroli typów, takich jak Java lub C#. Pomimo tego iż składnia języka JavaScript bardzo przypomina składnię C, to jednak sam język jest znacznie bardziej podobny do języków skryptowych, takich jak Ruby, Python, bądź nawet Common Lisp niż do Javy lub C#. JavaScript zapewnia ogromną elastyczność i przestrzeń do opracowywania własnych rozwiązań i metodologii. W rękach doświadczonego programisty jest to wspaniałe narzędzie, jednak przed przeciętnymi programistami stawia trudne i niebezpieczne wyzwania. Języki najczęściej stosowane do tworzenia aplikacji korporacyjnych, takie jak Java lub C# zostały zaprojektowane w taki sposób, by ułatwiały pracę zespołów przeciętnych programistów i dawały możliwość szybkiego zdobywania nowych użytkowników. Jednak JavaScript jest zupełnie innym językiem.

W przypadku stosowania JavaScriptu prawdopodobieństwo tworzenia zawikłanego, niezrozumiałego kodu jest bardzo wysokie; a kiedy zaczniemy używać go nie tylko do tworzenia prostych sztuczek, lecz także do pisania złożonych aplikacji wykorzystujących Ajaksa, to wszystkie te zagrożenia staną się znacznie bardziej realne i zauważalne. Właśnie dlatego gorąco zachęcamy, by znacznie częściej stosować refaktoryzację w aplikacjach wykorzystujących technologię Ajax niż w aplikacjach pisanych w językach Java lub C#, czyli językach bezpiecznych, w przypadku których społeczność programistów już dawno opracowała i zaczęła stosować wzorce projektowe.

3.1.3. Zachowanie właściwych proporcji

Zanim zajmiemy się kolejnymi zagadnieniami, koniecznie należy podkreślić, że zarówno refaktoryzacja, jak i wzorce projektowe są jedynie narzędziami i należy je stosować wyłącznie w przypadku, gdy może to zapewnić wymierne korzyści. Jeśli popadniemy w przesadę, to możemy doprowadzić do sytuacji określanej jako **paraliż przez analizę** (ang. *paralysis by analisys*), w której implementacja aplikacji jest wstrzymywana przez ciągłe zmiany projektu, mające na celu poprawienie elastyczności struktury kodu i uwzględnienie potencjalnych zmian wymagań, jakie mogą, lecz nie muszą, się pojawić.

Errich Gamma, ekspert zajmujący się wzorcami projektowymi, bardzo trafnie podsumował tę sytuację w jednym z ostatnich wywiadów (patrz podrozdział „Zasoby" zamieszczony na samym końcu tego rozdziału) — opowiadał w nim czytelniku, który zwrócił się do niego z prośbą o pomoc, gdyż w tworzonej aplikacji udało mu się zaimplementować jedynie 21 z 23 wzorców projektowych opisanych w książce *Design Patterns*. Należy pamiętać, że wzorce projektowe nadają się do zastosowania wyłącznie w określonych sytuacjach, tak samo jak różne typy danych — przecież nikt przy zdrowych zmysłach nie starałby się używać wszystkich dostępnych typów danych, liczb całkowitych, zmiennoprzecinkowych, łańcuchów znaków, tablic, itd. — w każdym fragmencie kodu.

Gamma stwierdza, że właśnie refaktoryzacja jest najlepszą okazją do wprowadzania wzorców projektowych. Najpierw należy napisać działający kod w możliwie jak najprostszy sposób, a dopiero potem starać się stosować wzorce

do rozwiązywania określonych problemów, jakie udało się nam wyróżnić. Jeśli Czytelnik sam jest autorem kodu lub jeśli musi „opiekować się" kodem, w którym „nabałaganił" ktoś inny, to zapewne aż do tej chwili odczuwa nieprzyjemny skurcz żołądka. Na szczęście wzorce projektowe można także stosować z powodzeniem w już istniejącym kodzie i to bez względu na to, jak został on napisany. W następnym punkcie rozdziału przypomnimy jakiś fragment kodu zapisany w rozdziale 2. i przekonamy się, co będziemy z niego mogli zrobić dzięki zastosowaniu refaktoryzacji.

3.1.4. Refaktoryzacja w działaniu

Być może niektórym z Czytelników refaktoryzacja już się spodobała, jednak bardziej praktyczne osoby, zanim się nią zachwycą, wolałyby zapewne ujrzeć ją w działaniu. Poświęćmy zatem nieco czasu na wprowadzenie zmian w kodzie, który napisaliśmy w poprzednim rozdziale i przedstawiliśmy na listingu 2.11. Kod ten składa się z funkcji `sendRequest()`, która odpowiada za przesyłanie żądań na serwer. Funkcja `sendRequest()` wywołuje z kolei funkcję `initHttpRequest()`, która tworzy odpowiedni obiekt `XMLHttpRequest` i definiuje funkcję obsługującą odpowiedzi — `onReadyState()`. Obiekt `XMLHttpRequest` został zapisany w zmiennej globalnej, dzięki czemu wszystkie używane przez nas funkcje mogły bez trudu pobrać referencję do niego. Funkcja zwrotna obsługująca odpowiedzi sprawdzała stan obiektu żądania i generowała stosowne informacje testowe.

Kod przedstawiony na listingu 2.11 wykonuje dokładnie to, o co nam chodziło, jednak jego ponowne wykorzystanie przysporzyłoby nam sporych problemów. Zazwyczaj jeśli decydujemy się na przesyłanie żądania na serwer, chcemy przetwarzać odpowiedzi i wykonywać pewne operacje charakterystyczne jedynie dla tej konkretnej aplikacji. W celu zastosowania nowej logiki biznesowej w istniejącym już kodzie, będziemy musieli zmodyfikować pewne fragmenty funkcji `onReadyState()`.

Sporym problemem może być także zastosowanie zmiennych globalnych. Jeśli jednocześnie będziemy chcieli przesłać na serwer kilka żądań, musimy mieć możliwość określenia odpowiedniej funkcji zwrotnej w każdym z nich. Jeżeli np. będziemy pobierali listę zasobów do aktualizacji oraz drugą listę zasobów przeznaczonych do aktualizacji, to przecież należałoby je w jakiś sposób rozróżnić!

W przypadku zastosowania technik programowania obiektowego rozwiązaniem takiego problemu byłoby zaimplementowanie niezbędnych możliwości funkcjonalnych w formie obiektu. Język JavaScript zapewnia niezbędne możliwości, które z powodzeniem będziemy mogli zastosować. Obiekt, jaki utworzymy w tym rozdziale, nazwiemy `ContentLoader`, gdyż będziemy go używać do pobierania zawartości z serwera. Ale jak ten obiekt powinien wyglądać? Optymalnie byłoby, gdybyśmy mogli utworzyć go, przekazując w wywołaniu konstruktora adres URL, na jaki należy przesłać żądanie. Oprócz tego powinniśmy mieć możliwość przekazania referencji do zdefiniowanej przez nas funkcji zwrotnej, która zostanie wywołana w razie pomyślnego zakończenia obsługi

żądania, oraz drugiej funkcji — wywoływanej w razie wystąpienia jakichś błędów. Instrukcja tworząca ten obiekt mogłaby mieć zatem następującą postać:

```
new loader = new ContentLoader('mydata.xml', parseMyData);
```

Przy czym `parseMyData` jest funkcją, którą należy wywołać w momencie odebrania odpowiedzi na żądanie. Kod obiektu `ContentLoader` przedstawiliśmy na listingu 3.1. Zastosowaliśmy w nim kilka nowych rozwiązań, które wymagają dokładniejszego opisu.

Listing 3.1. Obiekt ContentLoader

```
var net=new Object();     ❶ Obiekt pełniący funkcję przestrzeni nazw
net.READY_STATE_UNINITIALIZED=0;
net.READY_STATE_LOADING=1;
net.READY_STATE_LOADED=2;
net.READY_STATE_INTERACTIVE=3;
net.READY_STATE_COMPLETE=4;
net.ContentLoader=function(url,onload,onerror){      ❷ Funkcja konstruktora
  this.url=url;
  this.req=null;
  this.onload=onload;
  this.onerror=(onerror) ? onerror : this.defaultError;
  this.loadXMLDoc(url);
}
net.ContentLoader.prototype = {
  loadXMLDoc:function(url){     ❸ Funkcja initXMLHttpRequest ze zmienioną nazwą
    if (window.XMLHttpRequest){                                  ❹ Zmodyfikowana
      this.req=new XMLHttpRequest();                               funkcja loadXML
    } else if (window.ActiveXObject){
      this.req=new ActiveXObject("Microsoft.XMLHTTP");
    }
    if (this.req){
      try{
        var loader=this;
        this.req.onreadystatechange=function(){
          net.ContentLoader.onReadyState.call(loader);
        }
        this.req.open('GET',url,true);   ❺ Zmodyfikowana funkcja sendRequest
        this.req.send(null);
      }catch (err){
        this.onerror.call(this);
      }
    }
  },
  onReadyState:function(){     ❻ Zmodyfikowana funkcja zwrotna
    var req=this.req;
    var ready=req.readyState;
    var httpStatus=req.status;
    if (ready==net.READY_STATE_COMPLETE){
      if (httpStatus==200 || httpStatus==0){
        this.onload.call(this);
      }else{
        this.onerror.call(this);
      }
    }
  },
```

```
    defaultError:function(){
      alert("Błąd podczas pobierania danych!"
      +"\n\nreadyState:"+this.req.readyState
      +"\nstatus: "+this.req.status
      +"\nnagłówki: "+this.req.getAllResponseHeaders());
    }
  }
```

Pierwszą rzeczą, na jaką należy zwrócić uwagę w powyższym kodzie, jest zdefiniowanie zmiennej globalnej `net` ❶ i skojarzenie z nią wszystkich pozostałych referencji. Dzięki temu minimalizujemy ryzyko wystąpienia konfliktów nazw z innymi zmiennymi i grupujemy cały kod związany z generacją i obsługą żądań w jednym miejscu.

Następnie zdefiniowaliśmy funkcję konstruktora naszego obiektu ❷. Wymaga ona przekazania trzech argumentów, jednak tylko dwa pierwsze są obowiązkowe. Trzeci argument, określający funkcję wywoływaną w przypadku wystąpienia błędów, jest opcjonalny — jeśli jego wartość będzie równa `null`, to wywołamy funkcję domyślną. Dla osób, które mają doświadczenie w programowaniu obiektowym, możliwość przekazywania zmiennej ilości argumentów w wywołaniach funkcji może się wydawać dziwna, podobnie jak przekazywanie funkcji jako referencji. Jednak są to często stosowane możliwości języka JavaScript. Szczegółowe informacje na ich temat oraz dokładny opis samego języka można znaleźć w Dodatku B.

Znaczne fragmenty kodu funkcji `initXMLHttpRequest()` ❹ oraz `sendRequest()` ❺ z listingu 2.11 przenieśliśmy do metod tworzonego obiektu `ContentLoader`. Przy okazji zmieniliśmy trochę nazwy metod, by lepiej odpowiadały nieco większemu zakresowi ich działania. Teraz kod generujący żądania znajduje się w metodzie o nazwie `loadXMLDoc()`. ❸ Obiekt `XMLHttpRequest` odnajdujemy i tworzymy dokładnie w taki sam sposób jak poprzednio, jednak użytkownik naszego obiektu `ContentLoader` nie musi nic na ten temat wiedzieć. Funkcja zwrotna `onReadyState()` ❻ także powinna wyglądać bardzo podobnie do swego pierwowzoru z listingu 2.11. Wywołania generujące komunikaty na konsoli do testów zastąpiliśmy w tym przypadku wywołaniami funkcji `onload` oraz `onerror`. Składnia używana do wykonania tych wywołań może się wydawać nieco dziwna, zatem przyjrzyjmy się jej dokładniej. We właściwościach `onload` oraz `onerror` są zapisane obiekty `Function` języka JavaScript, a obiekty tego typu udostępniają metodę `call()`. Pierwszym argumentem wywołania metody `Function.call()` jest obiekt, który stanie się kontekstem dla wywoływanej funkcji, czyli wewnątrz niej będzie się można odwoływać do niego przy wykorzystaniu słowa kluczowego `this`.

Napisanie funkcji zwrotnej, którą przekażemy do obiektu `ContentLoader`, także nie przysporzy nam większych problemów. Jeśli w kodzie funkcji zwrotnej będziemy musieli odwołać się do jakiejkolwiek właściwości obiektu `ContentLoader`, to wystarczy, że jej nazwę poprzedzimy słowem kluczowym `this`. Np.:

```
function myCallBack(){
  alert( "Plik "
    +this.url
    +" został wczytany! Oto jego zawartość:\n\n"
```

```
      +this.req.responseText
    );
  }
```

Utworzenie takiego kontekstu wymaga napisania dodatkowego kodu i doskonałej znajomości tajników JavaScriptu, jednak kiedy go już napiszemy i zastosujemy w obiekcie, to użytkownik końcowy nie będzie musiał o niczym wiedzieć.

Taka sytuacja, w której użytkownik obiektu nie musi się interesować szczegółami jego implementacji, bardzo często jest sygnałem dobrze przeprowadzonej refaktoryzacji. Oznacza bowiem, że wszystkie złożone czynności zostały umieszczone wewnątrz obiektu, a użytkownik może się posługiwać łatwym interfejsem. W ten sposób jest on chroniony przed całą złożonością działania obiektu, a ekspert, który go stworzył i odpowiada za jego pielęgnację, umieścił cały złożony kod w jednym miejscu. Wszelkie ewentualne zmiany będą musiały być wykonane tylko jeden raz i w jednym, precyzyjnie określonym miejscu, a ich efekt zostanie zauważony i uwzględniony w całym kodzie aplikacji.

W ten sposób przedstawiliśmy podstawowe założenia refaktoryzacji i pokazaliśmy, jakie korzyści nam ona zapewnia. W następnym podrozdziale przedstawimy problemy, które najczęściej występują podczas pisania aplikacji wykorzystujących technologię Ajax, oraz jak można je rozwiązywać poprzez zastosowanie refaktoryzacji. Jednocześnie zaprezentujemy kilka przydatnych sztuczek, które my z powodzeniem będziemy mogli wykorzystać w następnych rozdziałach książki, a Czytelnik — we własnych projektach.

3.2. Niewielkie przykłady refaktoryzacji

W tym podrozdziale zaprezentujemy kilka problemów występujących podczas wykorzystywania technologii Ajax oraz najczęściej stosowane sposoby ich rozwiązywania. W każdym z przykładów pokażemy, jak zastosowanie refaktoryzacji może nam pomóc w rozwiązaniu problemu, a następnie wskażemy elementy rozwiązania, które będzie można wykorzystać przy innych okazjach.

Nawiązując do zaszczytnych tradycji literatury poświęconej wzorcom projektowym, w pierwszej kolejności opiszemy problem, następnie przedstawimy jego techniczne rozwiązanie, a na końcu opiszemy je w szerszym kontekście.

3.2.1. Niezgodności przeglądarek: wzorce Façade oraz Adapter

Gdyby zapytać osoby zajmujące się tworzeniem witryn i aplikacji internetowych — niezależnie od tego, czy są to programiści, projektanci, graficy, czy osoby zajmujące się każdym z tych aspektów tworzenia stron WWW — co je najbardziej denerwuje w ich pracy, to istnieje bardzo duże prawdopodobieństwo, iż na liście odpowiedzi znajdzie się konieczność zapewniania poprawnego wyglądu stron w różnych przeglądarkach. Na WWW istnieje wiele standardów przeróżnych technologii, a większość nowoczesnych przeglądarek implementuje zazwyczaj

w całości lub prawie w całości znaczną większość tych standardów. Czasami jednak standardy te nie są precyzyjne i zostawiają miejsce dla dowolnej interpretacji, czasami twórcy przeglądarek rozszerzają je w sposób przydatny, choć niespójny, a zdarza się również, że w przeglądarkach pojawiają się stare, wszechobecne błędy programistyczne.

Programiści używający języka JavaScript już od momentu jego pojawienia się nawykli do sprawdzania tego, jaka przeglądarka jest aktualnie używana bądź to testowania oraz czy konkretny obiekt jest dostępny, czy nie. Przeanalizujmy teraz bardzo prosty przykład.

Operowanie na elementach DOM

Zgodnie z informacjami podanymi w rozdziale 2. zawartość strony WWW jest udostępniana w kodzie JavaScript w formie obiektowego modelu dokumentu (ang. *Document Object Model*, w skrócie DOM) — struktury drzewiastej, której węzły odpowiadają elementom HTML umieszczonym w kodzie strony. W przypadku programowego manipulowania drzewem DOM bardzo często pojawia się konieczność określenia położenia elementu na stronie. Niestety, twórcy przeglądarek udostępnili różne, niestandardowe sposoby określania tego położenia, przez co napisanie jednego, ogólnego kodu wykonującego to zadanie prawidłowo w różnych przeglądarkach jest dosyć trudne. Listing 3.2 przedstawia uproszczoną wersję funkcji pochodzącej z biblioteki x Mike'a Fostera (patrz podrozdział 3.5), która rozwiązuje problem określenia położenia wskazanego elementu od lewej krawędzi strony. W tym przypadku element jest przekazywany jako argument wywołania funkcji.

Listing 3.2. Funkcja getLeft()

```
function getLeft(e){
  if(!(e=xGetElementById(e))){
    return 0;
  }
  var css=xDef(e.style);
  if (css && xStr(e.style.left)) {
    iX=parseInt(e.style.left);
    if(isNaN(iX)) iX=0;
  }else if(css && xDef(e.style.pixelLeft)) {
    iX=e.style.pixelLeft;
  }
  return iX;
}
```

Różne przeglądarki udostępniają wiele sposobów określania położenia węzła przy wykorzystaniu tablicy stylów, którą przedstawiliśmy w rozdziale 2. Standard W3C CSS2 udostępnia właściwość `style.left`, zawierającą łańcuch znaków składający się z wartości liczbowej oraz określenia jednostki, np. `100px`. Należy pamiętać, że mogą być stosowane inne jednostki niż piksele. Z kolei właściwość `style.pixelLeft` zawiera wartość numeryczną i to określającą przesunięcie elementu liczone w pikselach. Jednak właściwość ta jest dostępna wyłącznie

w przeglądarce Internet Explorer. Przedstawiona na powyższym listingu funkcja `getLeft()` w pierwszej kolejności sprawdza, czy jest dostępny obiekt stylów CSS — jeśli jest, to najpierw stara się pobrać wartość właściwości `style.left`, zaś jeśli nie jest ona dostępna, to właściwości `style.pixelLeft`. Jeżeli nie uda się pobrać żadnej z tych wartości, funkcja zwraca domyślną wartość `0`. Warto zauważyć, że metoda ta nie sprawdza nazw ani wersji używanej przeglądarki, a zamiast tego korzysta ze znacznie lepszego sposobu polegającego na testowaniu dostępności obiektów (opisaliśmy go w rozdziale 2.).

Pisanie takich funkcji, ukrywających różnice pomiędzy poszczególnymi przeglądarkami, jest zadaniem żmudnym i czasochłonnym; jednak kiedy się już z nim uporamy, będziemy mogli zająć się pisaniem aplikacji bez zwracania uwagi na te drobne, lecz niezwykle denerwujące szczegóły. A dzięki zastosowaniu dobrze napisanych i przetestowanych bibliotek, takich jak biblioteka x, możemy się uchronić przed koniecznością wykonywania większości najtrudniejszej pracy. Możliwość zastosowania godnych zaufania, ogólnych funkcji pozwalających na określanie położenia elementów DOM na stronie może w znaczny sposób przyspieszyć tworzenie interfejsu użytkownika aplikacji wykorzystujących technologię Ajax.

Przesyłanie żądania na serwer

Już w poprzednim rozdziale zetknęliśmy się z podobnym problemem niezgodności pomiędzy różnymi przeglądarkami. Wynikał on z faktu, iż różne przeglądarki udostępniają różne sposoby tworzenia obiektu `XMLHttpRequest`, używanego do asynchronicznego przesyłania żądań na serwer. Chcąc pobrać z serwera dokument XML, musieliśmy sprawdzić, którą z możliwości należy zastosować.

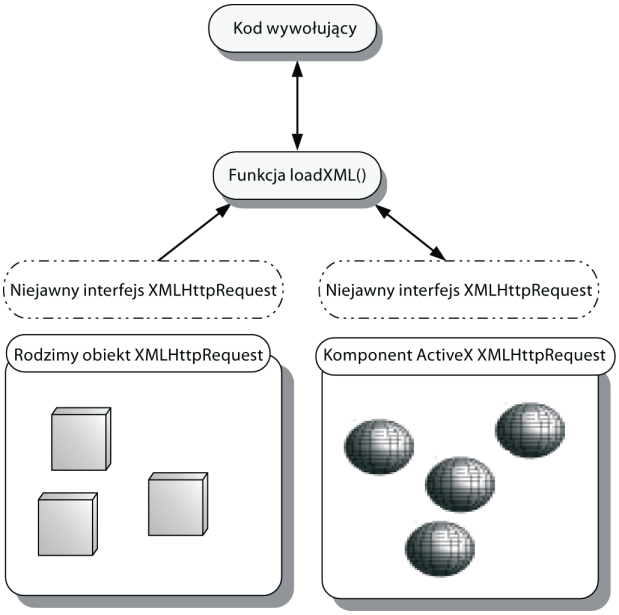
Internet Explorer pozwoli nam na asynchroniczne pobieranie plików wyłącznie w przypadku gdy poprosimy o odpowiedni komponent ActiveX, natomiast przeglądarki fundacji Mozilla i Safari doskonale poradzą sobie, używając wbudowanego obiektu `XMLHttpRequest`. Jednak o tych rozbieżnościach „wiedział" jedynie kod bezpośrednio odpowiedzialny za wczytywanie dokumentów XML. Kiedy obiekt `XMLHttpRequest` został już pobrany, to dalszy sposób korzystania z niego był identyczny we wszystkich przeglądarkach. Jednak kod skryptu, który chce pobrać jakiś dokument XML z serwera, nie musi „wiedzieć" ani o obiektach ActiveX, ani o podsystemie obiektów udostępnianych przez przeglądarkę: wystarczy, że będzie „wiedział", jak wywołać konstruktor `net.ContentLoader()`.

Wzorzec Façade

W obu przedstawionych wcześniej funkcjach, zarówno `getLeft()`, jak i `net.ContentLoader()`, kod służący do wykrywania aktualnie używanej przeglądarki jest paskudny i nużący. Dzięki stworzeniu funkcji, która ukryje go przed resztą naszego kodu, nie tylko poprawimy jego czytelność, lecz także umieścimy wszystkie możliwości funkcjonalne związane z wykrywaniem przeglądarki w jednym, precyzyjnie określonym miejscu. To jedna z podstawowych zasad refaktoryzacji — nie należy się powtarzać. Jeśli odkryjemy jakąś szczególną sytuację, w której

nasz obiektowy kod nie działa zbyt dobrze, wystarczy wprowadzić zmiany w jednym miejscu, a zostaną one uwzględnione we wszystkich wywołaniach określających położenie lewej krawędzi elementu DOM, tworzenia obiektu `XMLHttpRequest` bądź też jakiejkolwiek innej metody.

W języku wzorców projektowych mówimy, iż używamy wzorca o nazwie **Façade** (fasada). Wzorzec ten służy do tworzenia wspólnego punktu dostępu do różnych implementacji pewnej usługi lub możliwości funkcjonalnej. Taką użyteczną usługę udostępnia np. obiekt `XMLHttpRequest`, a naszej aplikacji tak naprawdę nie „obchodzi", w jaki sposób jest on tworzony — ważne, by działał prawidłowo (patrz rysunek 3.1).



Rysunek 3.1. Schemat wzorca Façade oraz jego związków z obiektem XMLHttpRequest w przeglądarkach. Funkcja loadXML() wymaga obiektu XMLHttpRequest, jednak nie „interesuje" jej faktyczna implementacja tego obiektu. Obie faktyczne implementacje mogą udostępniać znacznie bogatsze możliwości obsługi obiektów żądań HTTP, jednak w tym przypadku zostają one ograniczone od minimalnych możliwości wymaganych przez funkcję wywołującą

W wielu przypadkach zdarza się, że będziemy chcieli uprościć dostęp do pewnego podsystemu. Chcąc np. określić położenie lewej krawędzi elementu DOM, możemy to zrobić na wiele różnych sposobów, a zwrócona wartość może być wyrażona w pikselach, punktach lub wielu innych jednostkach. Ta dowolność może być nam zupełnie nieprzydatna. Przedstawiona wcześniej funkcja `getLeft()` będzie działać, jeśli określając układ strony, konsekwentnie będziemy to robić przy wykorzystaniu pikseli. Taki sposób uproszczenia podsystemu jest kolejną cechą wzorca Façade.

Wzorzec Adapter

Wzorzec Adapter jest powiązany ze wzorcem Façade. Także w jego przypadku operujemy na dwóch podsystemach, które wykonują te same zadania, takich jak dwa różne sposoby tworzenia obiektu `XMLHttpRequest` używane w przeglądarkach Internet Explorer i Mozilla. Jednak zamiast tworzyć fasadę dla każdego z tych dwóch podsystemów, to dla jednego z nich tworzymy dodatkową warstwę abstrakcji udostępniającą taki sam interfejs, jakim dysponuje drugi. To właśnie ta warstwa stanowi wzorzec Adapter. Biblioteka Sarissa, którą przedstawimy w punkcie 3.5.1, używa tego wzorca, by upodobnić komponent ActiveX stosowany w Internet Explorerze do wbudowanego obiektu `XMLHttpRequest` stosowanego w przeglądarkach fundacji Mozilla. Oba te wzorce projektowe są dobre i mogą znacząco pomóc w wykorzystaniu istniejącego już kodu w tworzonej aplikacji (jak widać, dotyczy to także rozwiązań udostępnianych przez przeglądarki).

Zajmijmy się kolejnym studium, w którym zastanowimy się nad problemami związanymi z modelem obsługi zdarzeń stosowanym w języku JavaScript.

3.2.2. Zarządzanie procedurami obsługi zdarzeń: wzorzec Observer

Trudno sobie wyobrazić pisanie aplikacji wykorzystujących technologię Ajax bez zastosowania technik programowania bazującego na zdarzeniach. Interfejsy użytkownika tworzone i obsługiwane w języku JavaScript powszechnie korzystają ze zdarzeń, a zastosowanie w aplikacji asynchronicznych żądań wysyłanych i odbieranych przy użyciu technologii Ajax powoduje, że w aplikacji pojawia się kolejna grupa zdarzeń oraz obsługujących je funkcji. W stosunkowo prostych aplikacjach zdarzenia takie, jak kliknięcie przycisku lub odebranie odpowiedzi przesłanej z serwera można obsługiwać przy wykorzystaniu jednej funkcji. Jednak wraz z powiększaniem się aplikacji zarówno pod względem ilości kodu, jak i jego złożoności może się pojawić konieczność, aby informacja o odebraniu jednego zdarzenia trafiła do kilku niezależnych podsystemów. W skrajnym przypadku może się nawet okazać, że konieczne jest udostępnienie mechanizmu pozwalającego zainteresowanym obiektom na rejestrowane się i otrzymywanie informacji o konkretnych zdarzeniach. Przyjrzyjmy się przykładowi, który zilustruje taką sytuację.

Stosowanie wielu procedur obsługi zdarzeń

Podczas programowego operowania na węzłach DOM często stosowana jest technika polegająca na umieszczaniu skryptu w procedurze obsługi zdarzeń `onload` (wykonywanej po załadowaniu całej strony, a co za tym idzie, po zakończeniu tworzenia drzewa DOM). Załóżmy, że na naszej stronie jest umieszczony element, w którym chcemy dynamicznie wyświetlać dane, jakie po załadowaniu strony będą cyklicznie pobierane z serwera. Kod JavaScript koordynujący pobieranie danych oraz ich wyświetlanie musi odwoływać się do węzła DOM, a zatem wszystkie operacje zaczynają się w procedurze obsługi zdarzeń `onload`:

```
window.onload=function() {
  displayDiv=document.getElementById('display');
}
```

Jak na razie wszystko idzie doskonale. A teraz załóżmy, że chcemy umieścić na stronie drugi obszar, prezentujący komunikaty pochodzące z kanałów RSS (jeśli Czytelnik jest zainteresowany mechanizmem implementującym takie możliwości, to może go znaleźć w rozdziale 13.). Kod aktualizujący prezentowane powiadomienia, także podczas uruchamiania strony, musi pobrać referencję do węzła DOM. Zatem także dla niego definiujemy procedurę obsługi zdarzeń `onload`:

```
window.onload=function() {
  feedDiv=document.getElementById('feeds');
}
```

Przetestowaliśmy oba te rozwiązania na osobnych stronach i okazało się, że działają doskonale. A zatem w ramach kolejnego etapu prac połączyliśmy je i wtedy okazało się, że druga procedura obsługi zdarzeń `onload` przesłoniła pierwszą, dane nie wyświetlają się na stronie, a co gorsza zamiast nich na stronie pojawia się cała masa komunikatów o błędach JavaScriptu. W tym przypadku cały problem polegał na tym, iż obiekt `window` pozwala na określenie tylko jednej procedury obsługi zdarzeń `onload`.

Ograniczenia złożonej procedury obsługi zdarzeń

Jak już napisaliśmy, w powyższym przykładzie druga procedura obsługi zdarzeń `onload` przesłania pierwszą. Problem ten możemy w prosty sposób rozwiązać poprzez napisanie jednej, złożonej funkcji:

```
window.onload=function() {
  displayDiv=document.getElementById('display');
  feedDiv=document.getElementById('feeds');
}
```

W naszym przykładzie takie rozwiązanie zda egzamin, choć ma tę wadę, iż łączy w sobie kod związany z prezentacją danych i pobieraniem zawartości kanałów RSS, które nie są ze sobą w żaden sposób powiązane. Jeśli w tej aplikacji będą stosowane nie 2, lecz 20 takich niezależnych podsystemów, i każdy będzie musiał pobrać referencję do jakiegoś węzła DOM, to kod takiej złożonej procedury obsługi zdarzeń stanie się trudny do pielęgnacji. Poza tym takie rozwiązanie znacznie utrudniłoby dodawanie do aplikacji i usuwanie z niej komponentów oraz sprawiłoby, że takie modyfikacje zostaną obciążone dodatkowym ryzykiem wystąpienia błędów. Innymi słowy, w większej aplikacji zastosowanie takiej złożonej procedury obsługi zdarzeń mogłoby doprowadzić do opisanej wcześniej sytuacji, w której nikt nie chciałby zmieniać kodu, obawiając się, iż przestanie on działać. Spróbujmy zatem nieco zmodyfikować nasz kod i zdefiniować osobne funkcje inicjujące dla każdego z podsystemów:

```
window.onload=function(){
  getDisplayElements();
  getFeedElements();
}
```

```
function getDisplayElements(){
  displayDiv=document.getElementById('display');
}
function getFeedElements(){
  feedDiv=document.getElementById('feeds');
}
```

Jak widać, udało nam się nieco poprawić przejrzystość i czytelność kodu oraz zredukować objętość funkcji `window.onload` do jednego wiersza kodu dla każdego podsystemu. Niemniej jednak ta złożona funkcja wciąż jest słabym punktem całego projektu i potencjalną przyczyną problemów. W następnym podpunkcie przedstawimy nieco bardziej złożone, lecz jednocześnie bardziej skalowalne rozwiązanie.

Wzorzec Observer

Czasami warto się zastanowić, na kim spoczywa odpowiedzialność za wykonanie pewnej akcji. W przypadku przedstawionej wcześniej funkcji złożonej odpowiedzialność za pobranie referencji do węzłów DOM spoczywała na obiekcie `window`, co oznacza, że musi on „wiedzieć", które z nich będą używane na danej stronie. W idealnym przypadku każdy z podsystemów sam powinien być odpowiedzialny za pobranie potrzebnej referencji. Dzięki temu jeśli podsystem zostanie użyty na stronie, to pobierze referencję, w przeciwnym razie żadne operacje inicjalizacyjne tego modułu nie zostaną wykonane.

Aby jasno i precyzyjnie określić obowiązki poszczególnych podsystemów, możemy zapewnić im możliwość rejestrowania się i uzyskiwania powiadomień o zgłoszeniu zdarzenia `load`. W tym celu podsystemy będą przekazywały funkcje zwrotne, które należy wywołać w procedurze obsługi zdarzeń `onload`. Poniżej przedstawiliśmy prostą implementację takiego rozwiązania:

```
window.onloadListeners=new Array();
window.addOnLoadListener(listener){
  window.onloadListeners[window.onloadListeners.length]=listener;
}
```

Po wczytaniu całej zawartości strony, podczas obsługi zdarzenia `load`, musimy jedynie przejrzeć całą zawartość tablicy `window.onloadListeners` i wywołać każdą z zapisanych w niej funkcji:

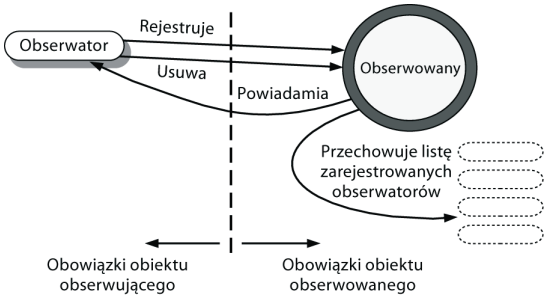
```
window.onload=function(){
  for(var i=0;i<window.onloadListeners.length;i++){
    var func=window.onlloadListeners[i];
    func.call();
  }
}
```

Jeśli założymy, że każdy podsystem wykorzysta to rozwiązanie, będziemy mogli zaoferować znacznie bardziej przejrzysty sposób ich inicjalizacji i to bez konieczności niepotrzebnego kojarzenia wszystkich podsystemów ze sobą w jednym fragmencie kodu. Oczywiście, wystarczy tylko jeden nieostrożny wiersz kodu, by jawnie przesłonić naszą funkcję `window.onload` i sprawić, że całe rozwiązanie

przestanie działać. Jednak w jakimś momencie będziemy musieli zatroszczyć się o nasz kod i zadbać o to, by takie błędy się w nim nie pojawiały.

Przy tej okazji warto zwrócić uwagę Czytelnika na fakt, iż nowy model obsługi zdarzeń W3C także implementuje podobne rozwiązanie, w którym w jednym obiekcie można rejestrować wiele funkcji obsługujących to samo zdarzenie. Zdecydowaliśmy się jednak na zaimplementowanie własnego rozwiązania bazującego na starym modelu obsługi zdarzeń języka JavaScript, gdyż nowy model W3C nie jest spójnie obsługiwany we wszystkich przeglądarkach. Zagadnieniem tym zajmiemy się bardziej szczegółowo w rozdziale 4.

Wzorzec projektowy, który zastosowaliśmy, tworząc powyższe rozwiązanie, nosi nazwę Observer (obserwator). Definiuje on obiekt „obserwowany" (w naszym przypadku jest to wbudowany obiekt `window` przeglądarki) oraz grupę obiektów „obserwatorów" lub „słuchaczy", które mogą się w nim rejestrować (patrz rysunek 3.2).



Rysunek 3.2. Rozdzielenie obowiązków we wzorcu projektowym Observer. Obiekty, które „chcą" być informowane o zdarzeniach, czyli obserwatory, mogą się rejestrować i usuwać ze źródła zdarzeń, które z kolei będzie przekazywać im stosowne informacje w momencie zajścia zdarzenia

W tym wzorcu obowiązki są prawidłowo rozdzielone pomiędzy źródło zdarzeń oraz procedurę obsługi tego zdarzenia. Funkcje obsługujące zdarzenie same powinny się zarejestrować oraz, ewentualnie, poinformować obiekt obserwowany, że nie są już zainteresowane generowanymi przez niego zdarzeniami. Obiekt stanowiący źródło zdarzeń odpowiada za przechowywanie listy wszystkich „zainteresowanych" funkcji oraz generowanie powiadomień w przypadku zgłoszenia zdarzenia. Wzorzec Observer od dawna jest stosowany podczas programowej obsługi interfejsów użytkownika sterowanych zdarzeniami i wrócimy do niego w rozdziale 4., przy okazji szczegółowej analizy obsługi zdarzeń w języku JavaScript. Jak się okaże, wzorzec ten z powodzeniem można zastosować we własnych obiektach, które nie mają nic wspólnego z obsługą zdarzeń generowanych przez klawiaturę lub mysz.

Teraz zajmiemy się kolejnym często występującym problemem, który możemy rozwiązać przy użyciu refaktoryzacji.

3.2.3. Wielokrotne stosowanie funkcji obsługujących działania użytkownika: wzorzec Command

Być może takie stwierdzenie będzie dla Czytelnika oczywiste, jednak w większości przypadków, to użytkownik „informuje" aplikację (przy wykorzystaniu klawiatury lub myszy) o tym, co należy zrobić, a ta wykonuje odpowiednie czynności. W prostych programach możemy udostępnić użytkownikowi tylko jedną możliwość wykonania pewnej czynności, jednak w przypadku tworzenia bardziej złożonych interfejsów możemy chcieć, aby mógł on doprowadzić do wykonania tej samej operacji na kilka różnych sposobów.

Implementacja kontrolki przycisku

Załóżmy, że na naszej stronie znajduje się element DOM, któremu, przy użyciu stylów CSS, nadaliśmy wygląd przypominający przycisk. Kliknięcie tego przycisku powoduje wykonanie pewnych obliczeń i wyświetlenie wyników w tabeli HTML. Procedurę obsługującą zdarzenia kliknięcia takiego przycisku moglibyśmy zdefiniować w następujący sposób:

```
function buttonOnclickHandler(event){
  var data=new Array();
  data[0]=6;
  data[1]=data[0]/3;
  data[2]=data[0]*data[1]+7;
  var newRow=createTableRow(dataTable);
  for (var i=0;i<data.length;i++){
    createTableCell(newRow,data[i]);
  }
}
```

Zakładamy przy tym, że zmienna `dataTable` zawiera referencję do faktycznie istniejącej tabeli, a funkcje `createTabelRow()` oraz `createTableCell()` potrafią prawidłowo operować na węzłach DOM. Najciekawszym fragmentem powyższego kodu jest fragment odpowiadający za wykonanie obliczeń; w rzeczywistej aplikacji może on mieć długość kilkuset wierszy kodu. Powyższą funkcję kojarzymy z przyciskiem w następujący sposób:

```
buttonDiv.onclick=buttonOnclickHandler;
```

Obsługa zdarzeń wielu różnych typów

Załóżmy teraz, że rozbudowaliśmy naszą aplikację o możliwości wykorzystywania technologii Ajax. Pobieramy z serwera aktualizacje danych i chcemy wykonywać powyższe obliczenia jedynie w przypadku gdy pewna wartość została zmodyfikowana, a oprócz tego chcemy aktualizować inną tabelę danych. Tym razem nie będziemy się zajmowali konfiguracją cyklicznego przesyłania żądań na serwer. Załóżmy, że dysponujemy obiektem, który zapewnia odpowiednie możliwości. Używa on obiektu `XMLHttpRequest` i konfiguruje go w taki sposób, by po odebraniu odpowiedzi z serwera wywoływana była funkcja `onload()`. Możemy wydzielić fragmenty kodu związane z dokonywaniem odpowiednich

obliczeń oraz z aktualizacją strony i umieścić je w dwóch niezależnych funkcjach przedstawionych poniżej:

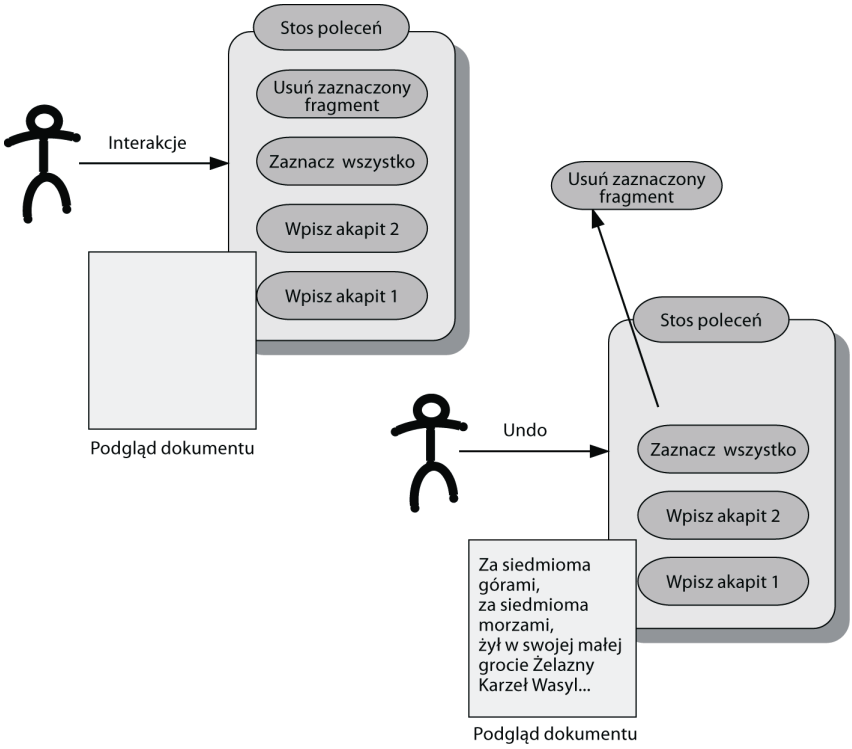
```
function buttonOnclickHandler(event){
  var data=calculate();
  showData(dataTable,data);
}
function ajaxOnloadHandler(){
  var data=calculate();
  showData(otherDataTable,data);
}
function calculate(){
  var data=new Array();
  data[0]=6;
  data[1]=data[0]/3;
  data[2]=data[0]*data[1]+7;
  return data;
}
function showData(table,data){
  var newRow=createTableRow(table);
  for (var i=0;i<data.length;i++){
    createTableCell(newRow,data[i]);
  }
}
buttonDiv.onclick=buttonOnclickHandler;
poller.onload=ajaxOnloadHandler;
```

Cały wielokrotnie używany kod został wydzielony i umieszczony w funkcjach `calculate()` oraz `showData()`; dzięki temu znacznie ograniczyliśmy powielanie się fragmentów kodu w całej aplikacji (aktualnie powtarzają się w niej jedynie dwa wiersze, używane w funkcjach `buttonOnclickHandler()` oraz `ajaxOnloadHandler()`).

Dzięki zastosowanemu rozwiązaniu znacznie lepiej rozdzieliliśmy kod odpowiedzialny za realizację logiki biznesowej oraz za aktualizację interfejsu użytkownika. Także w tym przypadku udało nam się określić przydatne rozwiązanie nadające się do wielokrotnego wykorzystania. Wzorzec projektowy Command definiuje pewną czynność o dowolnym stopniu złożoności, którą bez trudu będzie można przekazywać i używać w różnych miejscach kodu oraz w różnych elementach interfejsu użytkownika. W klasycznym wzorcu Command stosowanym w obiektowych językach programowania wszystkie czynności wykonywane przez użytkownika zostają umieszczone w obiektach Command, które zazwyczaj dziedziczą do pewnej wspólnej klasy bazowej (może to także być interfejs bazowy). W przedstawionym przykładzie ten sam problem rozwiązaliśmy w nieco inny sposób. Ponieważ w języku JavaScript funkcje są obiektami, same funkcje możemy potraktować jako obiekty Command. Jednak nawet takie nieco uproszczone rozwiązanie gwarantuje nam ten sam poziom abstrakcji.

Choć obsługa wszystkich czynności wykonywanych przez użytkownika w obiektach Command może się wydawać rozwiązaniem nieco dziwnym i nieporęcznym, to jednak posiada ono konkretne zalety. Kiedy np. wszystkie czynności wykonywane przez użytkownika zastaną zaimplementowane w formie obiektów Command, bardzo łatwo będziemy mogli z nimi skojarzyć inne, standardowe

możliwości funkcjonalne. Jednym z najczęściej opisywanych rozszerzeń tego typu jest dodanie metody `undo()`. W ten sposób można stworzyć zalążek ogólnego mechanizmu o zasięgu całej aplikacji, pozwalającego na odtwarzanie skutków wcześniej wykonanych czynności. Można też sobie wyobrazić bardziej złożone rozwiązanie, w którym każdy wykonywany obiekt Command będzie także zapisywany na stosie, a użytkownik, klikając odpowiedni przycisk, będzie mógł zdejmować te obiekty ze stosu, przywracając tym samym aplikację do wcześniejszego stanu (patrz rysunek 3.3).



Rysunek 3.3. Zastosowanie wzorca Command do utworzenia ogólnego stosu poleceń w aplikacji edytora tekstów. Wszystkie czynności wykonywane przez użytkownika są reprezentowane jako obiekty Command, dzięki temu czynności te można nie tylko wykonywać, lecz także odtwarzać

Każde nowe polecenie jest umieszczane na wierzchołku stosu, a następnie można je kolejno z niego zdejmować. Wyobraźmy sobie sytuację, w której użytkownik tworzy dokument, wykonując sekwencję akcji zapisu tekstu, a następnie, przypadkowo, zaznacza całą zawartość dokumentu i usuwa ją. W takim przypadku, jeśli kliknie on przycisk *Cofnij*, zdejmie ze stosu ostatni umieszczony na nim obiekt Command i wywoła jego metodę `undo()`, co sprawi, że usunięty tekst znów pojawi się w dokumencie. Dalsze kliknięcia przycisku będą powodowały usuwanie z dokumentu kolejnych fragmentów tekstu.

Oczywiście, z punktu widzenia programisty, zastosowanie obiektów Command do zaimplementowania stosu operacji, które można odtworzyć, oznacza konieczność wykonania dodatkowej pracy. Jest ona związana z zapewnieniem, by kombinacja wykonania pewnej czynności oraz jej odtworzenia spowodowała przywrócenie systemu do początkowego stanu. Niemniej jednak udostępnienie takiego mechanizmu może być ważnym czynnikiem świadczącym o jakości i możliwościach produktu. W szczególności dotyczy to aplikacji, które są używane długo i intensywnie, a zgodnie z informacjami podanymi w rozdziale 1. są to dwie podstawowe cechy aplikacji wykorzystujących technologię Ajax.

Obiekty Command można także zastosować w sytuacjach, gdy konieczne jest przenoszenie danych przez granice rozdzielające różne podsystemy aplikacji. Oczywiście jedną z takich granic jest sam internet, zatem wrócimy do wzorca projektowego Command w rozdziale 5., poświęconym interakcjom pomiędzy klientem i serwerem.

3.2.4. Przechowywanie tylko jednej referencji do zasobu: wzorzec Singleton

W niektórych sytuacjach bardzo duże znaczenie ma zapewnienie, by w aplikacji istniał tylko jeden punkt kontaktu z konkretnym zasobem. Także to zagadnienie najlepiej będzie przedstawić na konkretnym przykładzie.

Prosty przykład handlowy

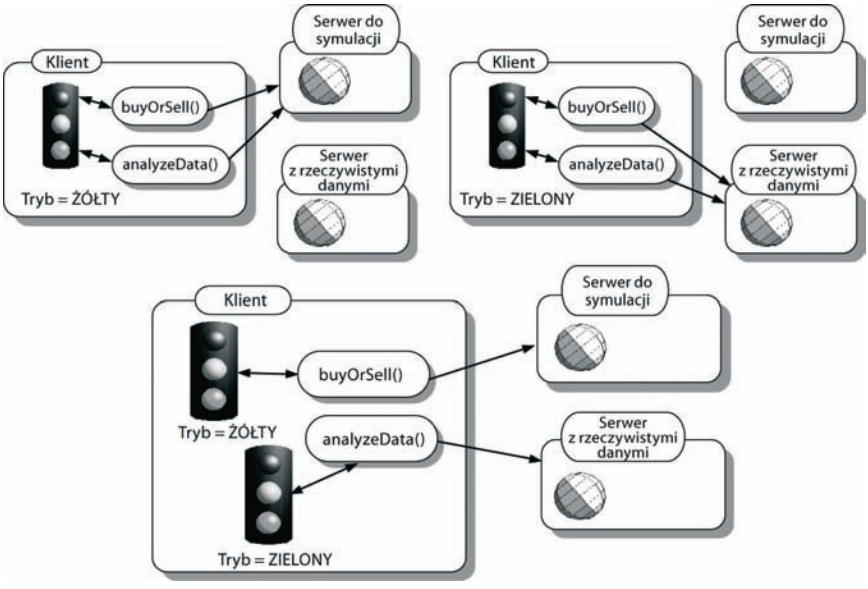
Załóżmy, że nasza aplikacja używająca technologii Ajax operuje na danych giełdowych, pozwalając nam na przeprowadzanie faktycznych operacji na papierach wartościowych, wykonywanie różnych obliczeń typu „co się stanie, jeśli" oraz uczestniczenie w internetowych grach symulacyjnych z innymi użytkownikami. Nasza aplikacja będzie mogła działać w trzech trybach, których nazwy odpowiadają kolorom świateł sygnalizacji ulicznej. I tak, tryb czasu rzeczywistego (tryb zielony) pozwala na wykonywanie faktycznych operacji giełdowych, czyli sprzedawanie i kupowanie papierów wartościowych w czasie gdy giełdy są otwarte oraz przeprowadzenie różnego typu obliczeń na zgromadzonych danych. Kiedy rynki zostaną zamknięte, aplikacja zaczyna działać w trybie analiz (trybie czerwonym), w którym wciąż pozwala na przeprowadzanie obliczeń, lecz nie daje możliwości handlowania papierami wartościowymi. Trzecim dostępnym trybem jest tryb symulacji (czyli żółty); możemy w nim realizować wszystkie czynności dostępne w trybie zielonym, lecz w tym przypadku nie są one wykonywane na faktycznych danych giełdowych, lecz na zestawie danych testowych.

Kod wykonywany w przeglądarce rozróżnia te wszystkie tryby przy wykorzystaniu obiektu JavaScript przedstawionego na poniższym przykładzie:

```
var MODE_RED=1;
var MODE_AMBER=2;
var MODE_GREEN=2;
function TradingMode(){
  this.mode=MODE_RED;
}
```

Możemy zarówno sprawdzać, jak i ustawiać tryb stosowany w tym obiekcie, a kod naszej aplikacji robi to w wielu miejscach. Moglibyśmy zdefiniować takie metody, jak `getMode()` oraz `setMode()`, które sprawdzałyby wiele różnych warunków, takich jak np. czy giełdy są w danej chwili otwarte itd., jak na razie jednak nie będziemy sobie komplikować życia.

Załóżmy, że użytkownicy mogą korzystać z dwóch możliwości aplikacji: sprzedawania i kupowania papierów wartościowych oraz wyliczania, przed wykonaniem operacji, potencjalnych zysków i strat, jakie ona wygeneruje. W zależności od trybu działania aplikacji akcje związane z kupowaniem i sprzedawaniem papierów wartościowych będą odwoływać się do różnych usług sieciowych — wewnętrznych — w trybie żółtym, usług dostępnych na serwerze naszego brokera — w trybie zielonym; natomiast w trybie czerwonym operacje te będą niedostępne. Podobnie analizy będę wykonywane w oparciu o pobierane z serwera dane o wcześniejszych i aktualnych cenach akcji, przy czym w trybie zielonym będą to faktyczne dane giełdowe, a w trybie żółtym — generowane przez nasz system dane testowe. Aby wiedzieć, z jakich źródeł danych skorzystać, aplikacja będzie korzystać z obiektu `TradingMode` (patrz rysunek 3.4).



Rysunek 3.4. W naszej przykładowej aplikacji do handlu akcjami, wykorzystującej technologię Ajax, zarówno funkcje obsługujące operacje kupna i sprzedaży, jak i funkcje analityczne określają, czy należy używać rzeczywistej bazy danych, czy też zestawu danych testowych. Wybór zestawu danych odbywa się na podstawie wartości statusu obiektu TradingMode. Dane testowe używane są w trybie żółtym oraz odnoszą się do serwera z rzeczywistymi danymi w trybie zielonym. Jeśli w systemie pojawiłby się więcej niż jeden obiekt TradingMode, to mógłby on stracić stabilność

Kluczowe znaczenie dla poprawnego działania aplikacji ma zapewnienie, by obie akcje korzystały z tego samego obiektu `TradingMode`. Jeśli użytkownik będzie handlować akacjami w grze symulacyjnej, podejmując decyzje na podstawie faktycznych kursów papierów wartościowych, to można przypuszczać, że przegra. Jeśli natomiast użytkownik będzie handlować prawdziwymi akcjami na podstawie analiz i symulacji wykonywanych na danych testowych, to zapewne szybko straci pracę!

Obiekt, którego w całej aplikacji można utworzyć tylko jeden egzemplarz, nazywamy często **singletonem**. W dalszej części rozdziału w pierwszej kolejności zobaczymy, jak singletony są tworzone w językach obiektowych, a następnie opracujemy sposób ich tworzenia w języku JavaScript.

Singletony w Javie

W językach obiektowych takich jak Java w implementacjach singletonów zazwyczaj ukrywamy konstruktor, a sam obiekt można pobierać przy użyciu odpowiedniej metody pobierającej. Implementację takiego singletonu przedstawiliśmy na listingu 3.3.

Listing 3.3. Obiekt TradingMode jako singleton zaimplementowany w Javie

```
public class TradingMode{
  private static TradingMode instance=null;
  public int mode;
  private TradingMode(){
    mode=MODE_RED;
  }
  public static TradingMode getInstance(){
    if (instance==null){
      instance=new TradingMode();
    }
    return instance;
  }
  public void setMode(int mode){
    ...
  }
}
```

Jak widać, powyższe rozwiązanie napisane w języku Java wymusza odpowiednie działanie singletonu poprzez zastosowanie modyfikatorów dostępu: `public` oraz `private`. Jeśli spróbujemy użyć poniższego fragmentu kodu:

```
new TradingMode().setMode(MODE_AMBER);
```

to okaże się, iż nie można go skompilować, gdyż konstruktor obiektu nie jest publicznie dostępny. Nasz obiekt da się natomiast pobrać przy użyciu poniższego wywołania:

```
TradingMode.getInstance().setMode(MODE_AMBER);
```

Powyższy kod zapewnia, że wszystkie odwołania trafią do tego samego egzemplarza obiektu `TradingMode`. Jednak w powyższym rozwiązaniu zastosowaliśmy kilka możliwości Javy, które nie są dostępne w języku JavaScript. Przekonajmy się zatem, w jaki sposób możemy rozwiązać te problemy.

Singletony w języku JavaScript

Język JavaScript nie udostępnia modyfikatorów dostępu, niemniej jednak możemy „ukryć" konstruktor obiektu — wystarczy w ogóle go nie implementować. JavaScript jest językiem wykorzystującym prototypy, w którym konstruktory są zwyczajnymi obiektami `Function` (jeśli Czytelnik nie wie, co to oznacza, to wszystkie niezbędne informacje można znaleźć w Dodatku B.). Moglibyśmy zaimplementować obiekt `TradingMode` w konwencjonalny sposób:

```
function TradingMode(){
  this.mode=MODE_RED;
}
TradingMode.prototype.setMode=function(){
}
```

i udostępnić zmienną globalną „udającą" singleton:

```
TradingMode.instance = new TradingMode();
```

Jednak takie rozwiązanie nie uchroniłoby nas przed możliwością wywołania konstruktora. Z drugiej strony, cały obiekt możemy także stworzyć ręcznie, bez stosowania prototypu:

```
var TradingMode=new Object();
TradingMode.mode=MODE_RED;
TradingMode.setMode=function(){
  ...
}
```

Ten sam efekt możemy uzyskać, stosując nieco bardziej zwartą formę zapisu:

```
var TradingMode={
  mode:MODE_RED,
  setMode:function(){
    ...
  }
};
```

W obu powyższych przypadkach wygenerowany obiekt będzie identyczny. Pierwsze rozwiązanie będzie zapewne bardziej zrozumiałe dla osób znających języki Java lub C#, natomiast drugi sposób tworzenia obiektów przedstawiliśmy we wcześniejszej części książki, gdyż jest on często stosowany w bibliotece Prototype oraz we wszystkich frameworkach, które z niej korzystają.

Powyższe rozwiązanie działa dobrze w zakresie jednego kontekstu skryptowego. Jeśli jednak skrypt zostanie wyczytany do osobnego elementu `<iframe>`, to spowoduje to utworzenie drugiego egzemplarza singletonu. Problem ten można jednak rozwiązać, jawnie żądając, by obiekt singletonu był pobierany z dokumentu głównego (w języku JavaScript odwołanie do tego dokumentu można pobrać przy wykorzystaniu właściwości `top`). Takie rozwiązanie przedstawiliśmy na listingu 3.4.

Listing 3.4. Obiekt TradingMode jako singleton zaimplementowany w języku JavaScript

```
Function getTradingMode(){
  if (!top.TradingMode){
    top.TradingMode=new Object();
    top.TradingMode.mode=MODE_RED;
    top.TradingMode.setMode=function(){
      ...
    }
  }
  return top.TradingMode;
}
```

Takie rozwiązanie sprawia, że skrypt będzie działał poprawnie nawet w przypadku dołączania go w różnych elementach `<iframe>`, a jednocześnie zapewnia unikalność obiektu singletonu. (Jeśli Czytelnik planuje obsługę singletonów w aplikacji, w której może być kilka okien głównych, to konieczne będzie sprawdzanie właściwości `top.opener`. Jednak ze względu na ograniczoną objętość niniejszej książki nie możemy szczegółowo opisać tych zagadnień i zostawiamy je jako ćwiczenie do samodzielnego wykonania przez Czytelnika).

Zazwyczaj obsługa interfejsu użytkownika nie będzie wymagała od nas stosowania singletonów, niemniej jednak mogą one być niezwykle przydatne podczas modelowania procesów biznesowych w języku JavaScript. W tradycyjnych aplikacjach internetowych logika biznesowa jest modelowana i wykonywana wyłącznie na serwerze; jednak w aplikacjach wykorzystujących technologię Ajax wszystko się zmienia, więc warto pamiętać o czymś takim jak singletony.

Prezentacją wzorca projektowego Singleton kończymy część rozdziału, której zadaniem było bardzo pobieżne przedstawienie praktycznych możliwości, jakie zapewnia nam refaktoryzacja. Oczywiście należy pamiętać, że wszystkie zaprezentowane przykłady były stosunkowo proste; jednak pomimo to zastosowanie refaktoryzacji do poprawienia przejrzystości kodu pomogło nam wyeliminować kilka jego słabych punktów, które w przyszłości, po powiększeniu się aplikacji, mogłyby jeszcze bardzo dać się nam we znaki.

Jednocześnie przedstawiliśmy kilka wzorców projektowych. W następnym podrozdziale zaprezentujemy duży i popularny wzorzec projektowy używany w aplikacjach wykonywanych po stronie serwera i przyjrzymy się, jak za jego pomocą możemy poprawić przejrzystość i elastyczność początkowo bardzo skomplikowanego kodu.

3.3. Model-View-Controller

Niewielkie wzorce projektowe, które przedstawiliśmy w poprzednim podrozdziale, mogą być przydatne podczas realizacji konkretnych zadań programistycznych. Jednak istnieją także wzorce opracowywane z myślą o określaniu sposobu organizacji całych aplikacji. Czasami określa się je jako **wzorce architektoniczne**. W tym podrozdziale przeanalizujemy jeden z takich wzorców architektonicznych, który

pomoże nam uporządkować kod aplikacji wykorzystującej technologię Ajax i sprawi, że zarówno jej tworzenie, jak i późniejsza pielęgnacja, będą łatwiejsze.

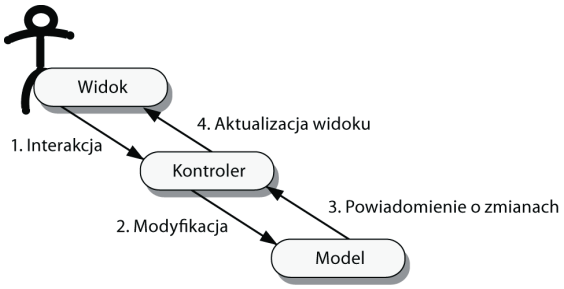
Wzorzec Model-View-Controller (w skrócie: MVC, Model-Widok-Kontroler) jest doskonałym sposobem opisu wzajemnej separacji fragmentów programu odpowiedzialnych za obsługę interakcji z użytkownikiem oraz za wykonywanie różnych poważnych, „biznesowych" operacji.

Wzorze MVC jest zazwyczaj stosowany w dużej skali — obejmuje całe warstwy aplikacji, a czasami nawet kilka warstw. W tym podrozdziale opiszemy go i pokażemy, w jaki sposób można go zastosować do zaimplementowania serwerowej części aplikacji, która będzie zwracać do niej dane w odpowiedzi na żądania przesyłane asynchronicznie przy użyciu technologii Ajax. W rozdziale 4. zajmiemy się nieco trudniejszym zagadnieniem, jakim jest zastosowanie wzorca MVC w kodzie pisanym w języku JavaScript i wykonywanym po stronie przeglądarki.

Wzorzec MVC określa trzy role, jakie mogą pełnić komponenty programu. **Model** jest reprezentacją dziedziny problemu, czyli tym, czym ma się zajmować aplikacja. Np. w edytorze tekstów modelem będzie dokument, a w elektronicznej mapie — punkty na siatce, linie itd.

Drugim elementem wzorca MVC jest **widok**. Jest to ta część programu, która odpowiada za prezentację — to ona wyświetla na ekranie elementy formularzy, obrazki, teksty oraz wszelkie inne kontrolki. Nie należy jednak sądzić, że widoki mogą mieć jedynie charakter graficzny. Np. w aplikacjach sterowanych głosem widokiem będą komunikaty dźwiękowe generowane przez aplikację.

Jedna z podstawowych zasad wzorca MVC zaleca, iż model i widok nie powinny się „komunikować" ze sobą w sposób bezpośredni. Łatwo zauważyć, iż programy składające się wyłącznie z modelu i widoku byłby raczej mało przydatne, dlatego też całość rozwiązania dopełnia trzeci element wzorca MVC — **kontroler**. Kiedy użytkownik kliknie przycisk lub wypełni pola formularza, widok poinformuje o tym fakcie kontroler. Kontroler wykona stosowne operacje na modelu i określi, czy zmiany, jakie zostały wprowadzone w modelu, wymagają aktualizacji widoku. Jeśli taka aktualizacja okaże się konieczna, to kontroler wydaje odpowiednie polecenie widokowi (patrz rysunek 3.5).



Rysunek 3.5. Podstawowe komponenty wzorca Model-View-Controller. Model oraz widok nie wymieniają informacji pomiędzy sobą w sposób bezpośredni, lecz za pośrednictwem kontrolera. Kontroler można sobie wyobrazić jako cienką warstwę pozwalającą na wymianę danych pomiędzy modelem i widokiem, która jednocześnie wymusza precyzyjne rozdzielenie obowiązków pomiędzy poszczególnymi fragmentami kodu, poprawiając w ten sposób jego elastyczność i ułatwiając utrzymanie

Zaleta tego rozwiązania polega na tym, iż widok i model są ze sobą powiązane w bardziej luźny sposób, czyli żaden z tych elementów aplikacji nie musi „wiedzieć" zbyt wiele na temat drugiego. Oczywiście, powinny one „wiedzieć" na tyle dużo, by mogły realizować swoje zadania, niemniej jednak muszą dysponować jedynie bardzo ogólną znajomością modelu.

Przeanalizujmy przykład programu do zarządzania stanem magazynu. W takim programie kontroler może udostępniać widokowi funkcję zwracającą listę wszystkich linii produktów należących do kategorii o podanym identyfikatorze; jednak sam widok nie „wie", w jaki sposób ta lista jest tworzona. Może się okazać, że w pierwszej wersji programu dane niezbędne do wygenerowania takiej listy są gromadzone w pamięci lub odczytywane z pliku. Z kolei w drugiej wersji programu pojawił się wymóg obsługi bardzo dużych zbiorów danych, w związku z tym do architektury systemu został dodany serwer relacyjnej bazy danych. Z punktu widzenia modelu zmiany, jakich trzeba dokonać w celu obsługi nowego źródła danych, są bardzo duże. Niemniej jednak, jeśli założymy, że kontroler wciąż będzie przekazywał widokowi listę linii produktów spełniającą zadane kryteria, to okaże się, że w kodzie widoku nie trzeba wprowadzać żadnych zmian.

Podobnie projektanci pracujący nad widokiem powinni być w stanie poprawiać i rozszerzać jego możliwości funkcjonalne bez obawy, że ich zmiany spowodują wystąpienia jakichś problemów w modelu, oczywiście o ile tylko będą stosowali prosty interfejs, jaki udostępnia im kontroler. A zatem, dzieląc cały system na trzy elementy, model MVC jest swoistą „polisą ubezpieczeniową", która zabezpiecza nas przed sytuacją, gdy jakaś drobna zmiana zmusi nas do wprowadzania wielu innych zmian w całym kodzie aplikacji. Oprócz tego wzorzec ten zapewnia możliwość szybkiego i sprawnego działania zespołów pracujących nad poszczególnymi elementami aplikacji i minimalizuje niebezpieczeństwo, że będą one sobie wzajemnie przeszkadzać.

Pewien szczególny sposób wykorzystania wzorca MVC jest bardzo często stosowany we frameworkach obsługujących klasyczne aplikacje internetowe. Otóż frameworki te generują statyczne strony WWW stanowiące interfejs użytkownika. W aplikacjach wykorzystujących technologię Ajax, działających i przesyłających żądania na serwer, proces zwracania danych przypomina nieco działanie klasycznych aplikacji internetowych. Oznacza to, że także aplikacje wykorzystujące technologię Ajax mogą skorzystać na zastosowaniu tej specyficznej wersji wzorca MVC. Ponieważ wzorzec ten jest powszechnie znany, nasze rozważania zaczniemy właśnie do niego, by później przejść do innych jego zastosowań, bardziej charakterystycznych dla aplikacji wykorzystujących technologię Ajax.

Jeśli Czytelnik nie zetknął się jeszcze z frameworkami używanymi do tworzenia aplikacji internetowych, to w tej części rozdziału może znaleźć informacje, które pomogą mu zrozumieć, dlaczego narzędzia te poprawiają solidność i skalowalność aplikacji. Jeśli jednak Czytelnik zna narzędzia i technologie używane do tworzenia wielowarstwowych aplikacji internetowych, takie jak mechanizmy obsługi szablonów, mechanizmy odwzorowań obiektowo-relacyjnych (określane skrótowo jako mechanizm **ORM**), czy też frameworki, takie jak Struts, Spring lub Tapestry, to zapewne będzie doskonale wiedział, o czym będziemy tu mówić.

W takim przypadku Czytelnik może pominąć dalszą część rozdziału i kontynuować lekturę od rozdziału 4., w którym zajmiemy się całkowicie innymi sposobami wykorzystania wzorca MVC.

3.4. Wzorzec MVC stosowany na serwerze

Wzorzec MVC jest od dawna stosowany w aplikacjach internetowych i to nawet w tych starych, bazujących na „normalnych" stronach, które tak mocno krytykujemy w niniejszej książce. Sam charakter tych aplikacji wymusza istnienie pewnej separacji pomiędzy widokiem i modelem, gdyż są one obsługiwane na dwóch różnych komputerach. A zatem czy aplikacje internetowe same w sobie są zgodne ze wzorcem MVC? Albo, ujmując to w inny sposób: czy można napisać aplikację internetową, w której role widoku i modelu zostaną połączone?

Niestety, taka możliwość istnieje. Co więcej, można to zrobić bardzo łatwo i większość programistów tworzących takie aplikacje, włącznie z autorami niniejszej książki, kiedyś tak postąpiła.

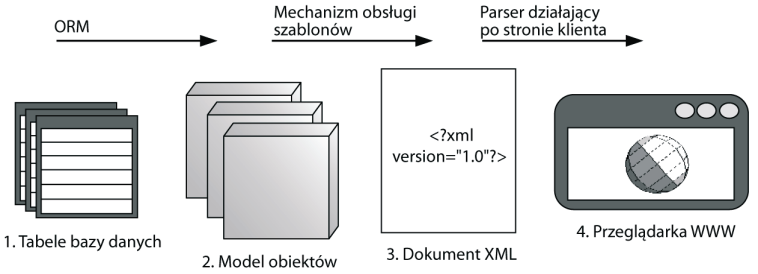
Przeważająca większość zwolenników stosowania wzorca MVC w aplikacjach internetowych traktuje wygenerowaną stronę HTML, jak również kod, który ją generuje, jako widok. Nie uważają oni, że widokiem jest jedynie to, co zobaczą na ekranie po wygenerowaniu strony. W przypadku aplikacji wykorzystujących technologię Ajax, przekazujących dane do kodu JavaScript wykonywanego w przeglądarce, widokiem jest dokument XML generowany przez serwer i zwracany w odpowiedzi na żądanie HTTP. Oznacza to, że rozdzielenie wygenerowanego kodu od logiki biznesowej będzie wymagać pewnej dyscypliny.

3.4.1. Serwerowa część aplikacji tworzona bez użycia wzorców

Aby zilustrować analizowane zagadnienia na przykładzie, napiszemy fragment aplikacji wykorzystującej technologię Ajax działający po stronie serwera. Już wcześniej, rozdziale 2. oraz w podpunkcie 3.1.4, przedstawiliśmy proste przykłady kodu JavaScript wykorzystującego możliwości Ajaksa; kolejne przykłady przedstawimy w rozdziale 4. Na razie jednak skoncentrujemy się na tym, co się dzieje po stronie serwera WWW. Kod serwerowej części aplikacji napiszemy najpierw w najprostszy możliwy sposób, a potem będziemy go stopniowo przekształcać w postać zgodną ze wzorcem MVC. W ten sposób pokażemy, jakie zalety — pod względem łatwości wprowadzania różnego typu zmian — posiada ten wzorzec. Zacznijmy zatem od przedstawienia aplikacji.

Otóż dysponujemy listą ubrań oferowanych w sklepie odzieżowym; lista ta jest przechowywana w bazie danych, a my chcemy przeszukiwać tę bazę i wyświetlać użytkownikom listę dostępnych produktów, prezentując dla każdego z nich obrazek, nazwę, krótki opis oraz cenę. Jeśli konkretny produkt będzie dostępny w kilku różnych kolorach, to chcemy, aby użytkownik mógł także określić kolor zamawianego produktu. Podstawowe komponenty tego systemu

zostały przedstawione na rysunku 3.6; są to: baza danych, struktura danych reprezentujących konkretny produkt, dokument XML jaki ma być przesłany do przeglądarki oraz lista wszystkich produktów spełniających podane kryteria wyszukiwania.



Rysunek 3.6. Podstawowe komponenty systemu używane w naszej przykładowej aplikacji sklepu internetowego do generacji danych XML. W ramach procesu generacji widoku pobieramy zbiór wyników z bazy danych, następnie na jego podstawie wypełniamy struktury danych reprezentujące poszczególne produkty, a w końcu przesyłamy wszystkie dane do klienta w postaci dokumentu XML

Załóżmy, że użytkownik właśnie wszedł na witrynę naszego sklepu i może wejść do jednego z trzech głównych działów: odzieży męskiej, kobiecej oraz dziecięcej. Każdy produkt w bazie jest przypisany do jednej z tych trzech kategorii; odpowiada za to kolumna `Category` umieszczona w tabeli `Garments`. Poniżej przedstawiliśmy proste polecenie SQL pobierające wszystkie produkty dostępne w kategorii odzieży męskiej:

```
SELECT * FROM garments WHERE CATEGORY = 'Menswear';
```

Naszym zadaniem jest pobranie wyników tego zapytania i przesłanie ich w postaci kodu XML do przeglądarki. Zobaczmy zatem, jak można to zrobić.

Generacja danych XML w celu ich przesłania do przeglądarki

Listing 3.5 ilustruje szybkie i niezbyt „eleganckie" rozwiązanie przedstawionego wcześniej wymagania. Skrypt został napisany w języku PHP i korzysta z bazy danych MySQL. Jednak akurat ten jego aspekt nie ma większego znaczenia — w przypadku tego skryptu ważna jest jego ogólna struktura. Skrypty pisane w innych językach: ASP, JSP czy też Ruby, mogłyby zostać napisane w bardzo podobny sposób.

Listing 3.5. Szybkie rozwiązanie problemu generacji danych XML na podstawie wyników zapytania

```
<?php
header("Content-type: application/xml"); ← Informujemy, że zwracany będzie
                                           kod XML
echo "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n";
```

```
$db=mysql_connect("my_db_server","mysql_user");                    Pobranie
mysql_select_db("mydb",$db);                                       danych z bazy
$sql="SELECT id,title,description,price,colors,sizes"
  ."FROM garments WHERE category=\"{$cat}\"";
$result=mysql_query($sql,$db);
echo "<garments>\n";
while ($myrow = mysql_fetch_row($result)) {    ← Przejrzenie zwróconych wyników
printf("<garment id=\"%s\" title=\"%s\">\n"
."<description>%s</description>\n<price>%s</price>\n",
$myrow["id"],
$myrow["title"],
$myrow["description"],
$myrow["price"]);
if (!is_null($myrow["colors"])){
echo "<colors>{$myrow['colors']}</colors>\n";
    }
if (!is_null($myrow["sizes"])){
echo "<sizes>{$myrow['sizes']}</sizes>\n";
    }
    echo "</garment>\n";
}
echo "</garments>\n";
?>
```

Powyższa strona PHP będzie generować kod XML przypominający ten przedstawiony na listingu 3.6, przy czym w tym przypadku widać, że z bazy zostały pobrane dwa produkty. Dla poprawienia czytelności listingu dodaliśmy do niego odpowiednie wcięcia. Zdecydowaliśmy się przekazywać dane do klienta w formie dokumentu XML, ponieważ jest to bardzo często stosowane rozwiązanie, oraz dlatego, że w poprzednim rozdziale pokazaliśmy, jak można pobierać takie dane po stronie klienta, używając do tego celu obiektu `XMLHttpRequest`. W rozdziale 5. przedstawimy inne możliwości wymiany danych pomiędzy serwerem i klientem.

Listing 3.6. Przykładowe wyniki wygenerowane przez skrypt z listingu 3.5

```
<garments>
  <garment id="SCK001" title="Golfers' Socks">
<description>Garish diamond patterned socks. Real wool.
Real itchy.</description>
<price>$5.99</price>
<colors>heather combo,hawaiian medley,wild turkey</colors>
</garment>
<garment id="HAT056" title="Deerstalker Cap">
<description>Complete with big flappy bits.
As worn by the great detective Sherlock Holmes.
Pipe is model's own.</description>
<price>$79.99</price>
<sizes>S, M, L, XL, egghead</sizes>
  </garment>
</garments>
```

A zatem udało nam się stworzyć serwerową część aplikacji internetowej, przy czym założyliśmy, że istnieje już „śliczna" aplikacja działająca po stronie przeglądarki, która, wykorzystując technologię Ajax, pobiera dane XML. Załóżmy, że wraz z powiększaniem się asortymentu naszego sklepu chcemy dodać do niego kilka nowych kategorii (takich jak: Na szczególne okazje, Codzienna, Dla aktywnych), opcję wyszukiwania odzieży „na wybraną porę roku" oraz ewentualnie opcję wyszukiwania po słowie kluczowym i łącze pozwalające na usunięcie wybranych wcześniej produktów. Wszystkie te opcje można by z powodzeniem obsłużyć przy użyciu podobnych dokumentów XML. Zobaczmy zatem, w jaki sposób moglibyśmy użyć istniejącego kodu do obsługi nowych możliwości oraz jakie problemy możemy przy tym napotkać.

Problemy wielokrotnego stosowania kodu

Istnieje kilka problemów utrudniających ponowne zastosowanie naszego kodu PHP w jego aktualnej postaci. Przede wszystkim zauważmy, że wykonywane przez skrypt zapytanie SQL zostało podane na stałe. Oznacza to, że chcąc ponownie przeszukać bazę na podstawie kategorii lub słowa kluczowego, będziemy musieli zmienić sposób generacji zapytania SQL. W efekcie możemy uzyskać koszmarną sekwencję instrukcji warunkowych, która stopniowo, wraz z pojawianiem się coraz to nowych opcji przeszukiwania, będzie stawać się coraz dłuższa.

Można sobie jednak wyobrazić jeszcze gorsze rozwiązanie: umieszczanie w klauzuli `WHERE` zapytania parametru przekazywanego w żądaniu:

```
$sql="SELECT id,title,description,price,colors,sizes"
  ."FROM garments
WHERE ".$sqlWhere;
```

W takim przypadku adres URL używany do wywołania skryptu mógłby mieć następującą postać:

```
garments.php?sqlWhere=CATEGORY="Menswear"
```

Takie rozwiązanie może wprowadzić jeszcze większe zamieszanie w naszym modelu i widoku, gdyż kod prezentacji uzyskuje bezpośrednio dostęp do zapytania SQL. Oprócz tego stanowi ono jawne zaproszenie do przeprowadzania ataków typu „SQL injection"[1] i choć nowe wersje języka PHP posiadają wbudowane zabezpieczenia przeciwko takim atakom, to jednak bezkrytyczne poleganie na nich byłoby co najmniej nierozważne.

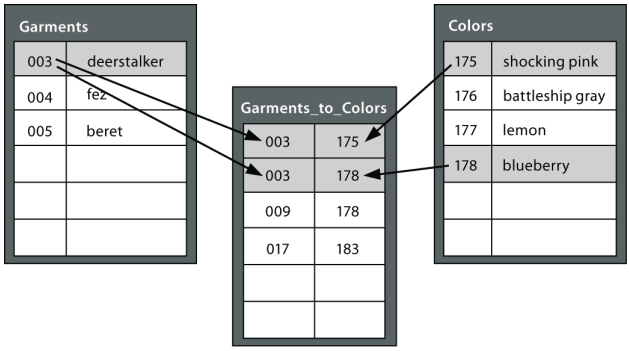
Co więcej, także struktura generowanego dokumentu XML została określona na stałe w kodzie PHP — gdzie jest generowana przy użyciu instrukcji `printf` oraz `echo`. Istnieje jednak wiele potencjalnych powodów, które mogą nas skłonić do zmiany formatu danych. Może się np. zdarzyć, że oprócz ceny sprzedaży będziemy chcieli wyświetlić także oryginalną cenę produktu, by przekonać wszystkich niedowiarków, że zakup oferowanych przez nas golfów jest naprawdę świetnym interesem!

[1] Ataki tego typu polegają na przekazywaniu tak spreparowanych danych, że po zastosowaniu ich w poleceniu SQL atakujący uzyska dostęp do tych, których autor skryptu nie planował ujawniać. — *przyp. tłum.*

W końcu ostatni problem wynika z faktu, iż dane XML są generowane bezpośrednio na podstawie informacji pobranych z bazy. Można by sądzić, iż jest to wydajne rozwiązanie, jednak w rzeczywistości jest ono potencjalnym źródłem dwóch problemów. Przede wszystkim podczas generowania kodu XML połączenie z bazą danych jest cały czas otwarte. W tym konkretnym przypadku wewnątrz pętli `while` nie wykonujemy żadnych bardzo skomplikowanych operacji, zatem połączenie z bazą nie będzie otwarte zbyt długo. Niemniej jednak w przyszłości takie przetwarzanie wyników przy otwartym połączeniu z bazą może się stać wąskim gardłem całego systemu. Poza tym rozwiązanie to będzie działać wyłącznie w przypadku gdy będziemy traktować naszą bazę jako płaską strukturę danych.

3.4.2. Refaktoryzacja modelu dziedziny

Używany przez nas obecnie sposób przechowywania listy kolorów i rozmiarów odzieży nie jest szczególnie wydajny — są one przechowywane w polach tabeli `Garments` jako łańcuchy znaków zawierające wartości oddzielone od siebie przecinkami. Jeśli znormalizujemy naszą bazę danych i zastosujemy się do zaleceń związanych z tworzeniem dobrych, relacyjnych modeli danych, to okaże się, że konieczne będzie utworzenie dwóch dodatkowych tabel — pierwszej, zawierającej listę wszystkich dostępnych kolorów oraz drugiej, kojarzącej konkretne produkty z kolorami (definiującej coś, co specjaliści od baz danych nazywają relacją „wiele-do-wielu"). Zastosowanie takiej relacji zostało przedstawione na rysunku 3.7.



Rysunek 3.7. Relacje wiele-do-wielu w tabelach bazy danych. W tabeli Colors przechowywane są wszystkie kolory, jakie mogą mieć produkty, dzięki temu w tabeli Garments nie muszą być już przechowywane żadne informacje o kolorach

Aby określić listę kolorów, w jakich jest dostępna oferowana w sklepie czapeczka z dwoma daszkami, przeglądamy tabelę `Garments_to_Colors`, używając przy tym klucza zewnętrznego `garment_id`. Kolejne porównanie, tym razem kolumny `color_id` z kluczem głównym tabeli `Colors`, pozwoli nam dowiedzieć się, że czapeczka jest dostępna w kolorach różowym i ciemnofiletowym, lecz nie w kolorze ciemnoszarym. Wykonując oba polecenia w odwrotnej kolejności, możemy użyć tabeli `Garments_to_Colors` do sporządzenia listy wszystkich produktów dostępnych w konkretnym kolorze.

Obecnie używamy już bazy danych w nieco lepszy sposób, jednak kod SQL konieczny do obsłużenia wszystkich zapytań i ich opcji powoli staje się dosyć długi. Dlatego też zamiast samodzielnego tworzenia długich, łączonych zapytań, dobrze by było, gdybyśmy mogli potraktować nasze produkty jako obiekty, które zawierałyby tablice dostępnych kolorów i rozmiarów.

Narzędzia obsługi odwzorowań obiektowo-relacyjnych

Na szczęście istnieją biblioteki i narzędzia, które są w stanie przekształcić zbiór danych w obiekty; są to tzw. mechanizmy odwzorowań obiektowo-relacyjnych (ang. *Object-Relational Mappig*, w skrócie ORM). Narzędzia ORM automatycznie przekształcają obiekty bazy danych w obiekty dostępne dla skryptu PHP i są w stanie „stworzyć" za programistę cały niezbędny kod SQL. Programiści używający języka PHP mogą się przyjrzeć takim rozwiązaniom jak obiekt `DB_DataObject`, pakiet Easy PHP Data Objects (EZPDO) czy też Metastorage. Programiści używający języka Java mają raczej prosty wybór — świadczy to tym fakt, iż ich przeważająca większość wybiera pakiet `Hibernate` (aktualnie trwają prace nad udostępnieniem go w wersji dla środowiska .NET). Narzędzia ORM stanowią bardzo obszerne zagadnienie, którym, niestety, nie możemy się w niniejszej książce zająć.

Analizując naszą aplikacje pod kątem zastosowania w niej wzorca MVC, można zauważyć, iż użycie mechanizmu ORM daje tę dodatkową zaletę, że dostajemy do dyspozycji doskonały obiektowy model. Teraz możemy zmienić kod generujący dokument XML w taki sposób, by korzystał z obiektu `Garment` i w ogóle nie „przejmował" się operacjami na bazie danych, które w całości będą realizowane przez mechanizm ORM. Co więcej, dzięki takiemu rozwiązaniu nie jesteśmy już uzależnienie od funkcji służących do obsługi konkretnej bazy danych. Listing 3.7 przedstawia zmodyfikowaną wersję poprzedniego kodu, wykorzystującą mechanizm ORM.

Listing 3.7. Model obiektów stosowany w sklepie odzieżowym

```
require_once "DB/DataObject.php";
class GarmentColor extends DB_DataObject {
  var $id;
  var $garment_id;
  var $color_id;
}
class Color extends DB_DataObject {
  var $id;
  var $name;
}
class Garment extends DB_DataObject {
  var $id;
  var $title;
  var $description;
  var $price;
  var $colors;
  var $category;
  function getColors(){
```

```
    if (!isset($this->colors)){
      $linkObject=new GarmentColor();
      $linkObject->garment_id = $this->id;
      $linkObject->find();
      $colors=array();
      while ($linkObject->fetch()){
        $colorObject=new Color();
        $colorObject->id=$linkObject->color_id;
        $colorObject->find();
        while ($colorObject->fetch()){
          $colors[] = clone($colorObject);
        }
      }
    }
    return $colors;
  }
}
```

W tym przypadku obiekty biznesowe (czyli model), które będą używane w naszym przykładowym sklepie, definiujemy w języku PHP, używając do tego klasy `PEAR::DB_DataObject`. Wymaga ona, żeby klasy definiowane przez programistę dziedziczyły po klasie `DB_DataObject`. W innych mechanizmach ORM mogą być stosowane zupełnie inne rozwiązania, jednak podstawowa zasada ich działania zawsze jest taka sama i polega na tym, że tworzymy grupę obiektów, które są używane tak samo jak wszystkie inne obiekty i ukrywają przed nami wszystkie problemy i uciążliwości związane ze stosowaniem języka SQL.

Oprócz głównej klasy `Grament` zdefiniowaliśmy także klasę `Color` oraz metodę pozwalającą na pobieranie kolorów (obiektów `Color`), w jakich jest dostępny dany produkt (obiekt `Garment`). W bardzo podobny sposób można by zaimplementować obsługę rozmiarów, jednak dla uproszczenia przykładu pominęliśmy ją tutaj. Ponieważ zastosowana biblioteka nie obsługuje bezpośrednio relacji „wiele-do-wielu", musimy zdefiniować klasę dla tabeli pośredniej i korzystać z niej w metodzie `getColors()`. Pomimo to utworzony tak model obiektów jest stosunkowo kompletny i zrozumiały. A teraz przekonajmy się, w jaki sposób możemy wykorzystać ten model obiektów w naszym skrypcie.

Zastosowanie poprawionego modelu

Wygenerowaliśmy nowy model obiektów, bazujący na nowej, bardziej przejrzystej strukturze bazy danych. Teraz musimy go użyć w kodzie skryptu PHP. Listing 3.8 przedstawia poprawiony kod strony, w której zastosowaliśmy nasze nowe klasy.

Listing 3.8. Zmodyfikowana strona obsługująca bazę danych przy wykorzystaniu mechanizmu ORM

```
<?php
header("Content-type: application/xml");
echo "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n";
include "garment_business_objects.inc";
$garment=new Garment;
```

```
$garment->category = $_GET["cat"];
$number_of_rows = $garment->find();
echo "<garments>\n";
while ($garment->fetch()) {
  printf("<garment id=\"%s\" title=\"%s\">\n"
    ."<description>%s</description>\n<price>%s</price>\n",
    $garment->id,
    $garment->title,
    $garment->description,
    $garment->price);
  $colors=$garment->getColors();
  if (count($colors)>0){
    echo "<colors>\n";
    for($i=0;$i<count($colors);$i++){
      echo "<color>{$colors[$i]}</color>\n";
    }
    echo "</colors>\n";
  }
  echo "</garment>\n";
}
echo "</garments>\n";
?>
```

Na początku kodu skryptu dołączamy definicje klas, a w kolejnych wykonywanych operacjach posługujemy się już wyłącznie obiektami naszego modelu. Zamiast tworzenia jakiegoś doraźnego kodu SQL tworzymy pusty obiekt `Garment` i zapisujemy w nim kryteria wyszukiwania, jakimi dysponujemy. Ponieważ model obiektów jest dołączany w formie pliku zewnętrznego, będziemy go mogli używać także na innych stronach, obsługujących inne zapytania. Także kod XML stanowiący widok naszej aplikacji jest generowany na podstawie obiektów. Kolejną czynnością związaną z refaktoryzacją naszej aplikacji będzie oddzielenie formatu kodu XML od procesu, który ten kod generuje.

3.4.3. Separacja zawartości od prezentacji

Widok naszej aplikacji wciąż jest powiązany z obiektem, gdyż z nim powiązany jest generowany kod XML. Jeśli aplikacja ta będzie się składać z kilku stron, to nie będziemy mogli zmienić formatu kodu XML w jednym, centralnym miejscu i liczyć na to, że wszystkie strony „zauważą" i uwzględnią te zmiany. W bardziej złożonych rozwiązaniach wykorzystujących kilka różnych formatów dokumentów XML, np. osobny format dla listy informacji skróconych, listy informacji pełnych oraz listy stanu magazynu, zapewne chcielibyśmy zdefiniować każdy z nich tylko jeden raz i dla każdego udostępnić jakieś scentralizowane rozwiązanie pozwalające na generację dokumentu XML w konkretnym formacie.

Systemy wykorzystujące szablony

Jednym z bardzo popularnych rozwiązań problemów tego typu jest użycie systemu pozwalającego na tworzenie dokumentów tekstowych zawierających pewne znaczniki, które podczas przetwarzania są zastępowane wartościami odpowiednich

zmiennych. Skrypty PHP, ASP oraz JSP same są językami przetwarzającymi szablony, gdyż można je zapisać w formie strony HTML zawierającej znaczniki, które w odpowiednich miejscach wstawiają podane wartości. Ten sposób działania odróżnia szablony od serwletów Javy oraz tradycyjnych skryptów CGI, które zawierają sekwencje instrukcji generujących kod HTML. Języki te pozwalają na stosowanie w szablonach wszystkich swoich możliwości, dzięki czemu tak łatwo można połączyć logikę biznesową z prezentacją.

Z kolei w specjalistycznych językach przetwarzania szablonów, takich jak Smarty używany języku PHP lub Apache Velocity (system napisany oryginalnie w języku Java, a później przeniesiony także na platformę .NET pod nazwą NVelocity), możliwości tworzonego kodu są zazwyczaj znacząco mniejsze i ograniczają się do kontroli przepływu sterowania (np. instrukcji warunkowych `if`) oraz pętli (takich jak `for` i `while`). Listing 3.9 przedstawia szablon Smarty generujący dokument XML używany w naszej przykładowej aplikacji.

Listing 3.9. Szablon Smarty służący do generacji kodu XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<garments>
{section name=garment loop=$garments}
  <garment id="{$garment.id}" title="{$garment.title}">
<description>{$garment.description}</description>
<price>{$garment.price}</price>
{if count($garment.getColors())>0}
<colors>
{section name=color loop=$garment.getColors()}
<color>$color->name</color>
{/section}
</colors>
{/if}
  </garment>
{/section}
</garments>
```

Powyższy szablon „oczekuje", że jako dane wejściowe zostanie do niego przekazana tablica o nazwie `garments`, zawierająca obiekty `Garment`. Przeważająca część szablonu jest generowana w oryginalnej postaci, jednak jego fragmenty umieszczone w nawiasach klamrowych są interpretowane jako instrukcje i zastępowane odpowiednimi zmiennymi bądź traktowane jako pętle lub instrukcje warunkowe. Struktura generowanego kodu XML jest znacznie wyraźniejsza w szablonie niż w kodzie skryptowym, o czym łatwo można się przekonać, porównując powyższy kod z listingiem 3.7. A teraz zobaczmy, w jaki sposób można wykorzystać taki szablon w naszym skrypcie.

Poprawiony widok

Udało nam się przenieść definicje formatu XML z głównego skryptu do niezależnego pliku szablonu. W efekcie, musimy teraz w skrypcie jedynie zainicjować mechanizm obsługi szablonów i przekazać do niego odpowiednie dane. Zmiany, jakie należy w tym celu wprowadzić, przedstawiliśmy na listingu 3.10.

Listing 3.10. Zastosowanie mechanizmu Smarty do generacji kodu XML

```
<?php
header("Content-type: application/xml");
include "garment_business_objects.inc";
include "smarty.class.php";
$garment=new DataObjects_Garment;
$garment->category = $_GET["cat"];
$number_of_rows = $garment->find();
$smarty=new Smarty;
$smarty->assign('garments',$garments);
$smarty->display('garments_xml.tpl');
?>
```

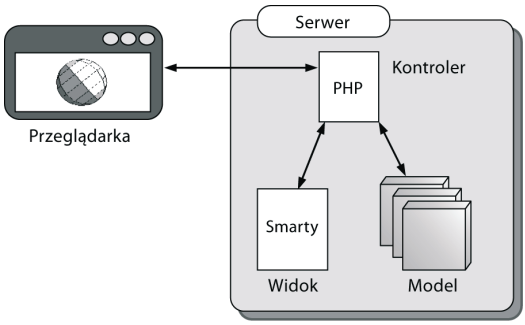
Sposób stosowania mechanizm Smarty jest bardzo zwięzły i składa się z trzech etapów. Pierwszy z nich polega na utworzeniu samego mechanizmu. W ramach drugiego etapu należy określić wartości zmiennych, które będą potrzebne do wygenerowania szablonu. W naszym przypadku będzie to tylko jedna zmienna, jednak możemy ich określić dowolnie wiele — gdyby w sesji były przechowywane jakieś dane dotyczące użytkownika, to moglibyśmy je także przekazywać, by wyświetlić np. spersonalizowane powitanie. W końcu wywołujemy metodę `display()`, przekazując do niej nazwę pliku szablonu.

Jak widać, udało się nam szczęśliwie odseparować widok od samej strony pobierającej dane z bazy. Postać dokumentu XML jest definiowana tylko jeden raz, a wykonanie jego generacji zajmuje nam kilka wierszy kodu. Strona z wynikami wyszukiwania jest ścisła i precyzyjna — zawiera tylko te informacje, które są jej potrzebne, konkretnie rzecz biorąc, są to wartości parametrów wyszukiwania oraz dane określające format wyników. Być może Czytelnik pamięta, że wcześniej wspominaliśmy, iż byłoby cudownie, gdybyśmy mogli w dowolnej chwili zmieniać format XML używany do generacji danych wynikowych. Dzięki zastosowaniu mechanizmu Smarty nie przysparza nam to najmniejszych problemów — wystarczy go zdefiniować. Mechanizm Smarty zapewnia nawet możliwość wstawiania jednych szablonów wewnątrz innych, z której możemy skorzystać, jeśli zechcemy generować dane wynikowe w sposób bardzo strukturalny.

Jeśli przyjrzymy się informacjom podanym na samym początku analizy wzorca MVC, przekonamy się, iż implementujemy go we właściwy sposób. Na rysunku 3.8 przedstawiliśmy graficznie to, co do tej pory udało się nam osiągnąć.

Nasz model jest kolekcją obiektów dziedziny, które są automatycznie zapisywane w bazie danych przy wykorzystaniu mechanizmu ORM. Rolę widoku pełni szablon określający postać generowanego kodu XML. I w końcu kontrolerem jest strona PHP służąca do wyszukiwania produktów według kategorii, jak również wszystkie inne strony, które zdecydujemy się zdefiniować i które będą łączyć model oraz widok.

W taki sposób wygląda klasyczne zastosowanie wzorca MVC w aplikacjach internetowych. W prezentowanym przykładzie zastosowaliśmy ten wzorzec w serwerowej części aplikacji wykorzystującej technologię Ajax, która generuje



Rysunek 3.8. Wzorzec MVC w postaci często stosowanej w aplikacjach internetowych. Strona WWW lub serwlet działają jako kontroler i w pierwszej kolejności pobierają odpowiednie dane z obiektów modelu. Następnie przekazują te dane do pliku szablonu (widoku), który z kolei generuje zawartość przesyłaną do przeglądarki użytkownika. Warto zauważyć, że jest to rozwiązanie obsługujące jedynie operacje odczytu. W przypadku modyfikacji modelu sekwencja wykonywanych czynności i zdarzeń byłaby nieco inna, niemniej jednak role poszczególnych elementów pozostałyby takie same

dokumenty XML. Niemniej jednak bez trudu można sobie wyobrazić, w jaki sposób można by zastosować ten sam wzorzec w klasycznej aplikacji internetowej, generującej dokumenty HTML.

W zależności od używanych technologii Czytelnik będzie mógł się spotkać z różnymi wersjami przedstawionego wzorca MVC; jednak zawsze podstawowe zasady jego działania i konstrukcji będą identyczne. Np. komponenty EJB (ang. *Enterprise JavaBean*) używane w technologii J2EE definiują model oraz kontroler w taki sposób, by można je było przechowywać na różnych serwerach. Z kolei na platformie .NET rolę kontrolerów pełnią specjalne „ukryte" obiekty definiowane osobno dla poszczególnych stron. Jeszcze inaczej sytuacja wygląda we frameworkach takich jak Struts, które definiują tak zwany „kontroler frontonu" (ang. *front controller*) przechwytujący wszystkie żądania kierowane do aplikacji i przekazujący je w odpowiednie miejsca. W takich frameworkach działanie kontrolera zostało zmodyfikowane tak, by nie tylko obsługiwał konkretne strony, lecz także zarządzał przemieszczaniem się użytkownika pomiędzy poszczególnymi stronami aplikacji. (W aplikacji wykorzystującej technologię Ajax podobne możliwości możemy zaimplementować przy użyciu języka JavaScript). Jednak we wszystkich tych przypadkach postać wzorca MVC jest w zasadzie taka sama i zakłada się, że właśnie tak powinno się ją rozmieć w kontekście aplikacji internetowych.

Opisanie architektury naszej aplikacji przy użyciu wzorca MVC jest przydatnym rozwiązaniem, a co więcej, będziemy mogli z niego korzystać, nawet jeśli zdecydujemy się wzbogacić klasyczną aplikację internetową o wykorzystanie technologii Ajax. Warto jednak zwrócić uwagę, iż nie jest to jedyna możliwość zastosowania wzorca MVC w aplikacjach używających Ajaksa. W rozdziale 4. przedstawimy inne wersje tego wzorca, dzięki którym jego zalety będziemy mogli wykorzystać także w innych miejscach aplikacji. Zanim to jednak zrobimy, przyjrzyjmy się innym sposobom zapewniania porządku i organizacji w aplikacjach wykorzystujących Ajaksa.

Oprócz modyfikowania i poprawiania własnego kodu całość kodu aplikacji możemy często uprościć i skrócić poprzez zastosowanie różnego typu frameworków i bibliotek. Wraz z ciągłym wzrostem zainteresowania i popularności technologii Ajax rośnie także ilość wykorzystujących ją frameworków i bibliotek. Kilka najpopularniejszych rozwiązań tego typu przedstawiliśmy na końcu niniejszego rozdziału.

3.5. Biblioteki i frameworki

Jednym z podstawowych celów refaktoryzacji jest usuwanie powielającego się kodu poprzez umieszczanie go w funkcjach i obiektach, których będzie można używać w wielu miejscach aplikacji. Rozwijając tę logikę działania, takie wielokrotnie używane możliwości funkcjonalne można umieścić w bibliotece (lub frameworku), której następnie będzie można używać także w innych projektach. W ten sposób można zmniejszyć ilość unikalnego kodu, jaki należy napisać podczas prac nad projektem, a tym samym poprawić wydajność pracy. Co więcej, ponieważ taka biblioteka została przetestowana podczas tworzenia wcześniejszych projektów, można oczekiwać, że jej jakość będzie wysoka.

W dalszej części niniejszej książki stworzymy kilka niewielkich frameworków, które Czytelnik będzie mógł zastosować we własnych projektach: w rozdziałach 4. i 5. stworzymy obiekt `ObjectBrowser`, w rozdziale 5. zaprezentujemy także obiekt `CommandQueue`, w rozdziale 6. — framework służący do prezentacji powiadomień, w rozdziale 8. — mechanizm profilujący `StopWatch`, a w Dodatku A — konsolę testującą. Będziemy także modyfikować aplikacje przykładowe, tworzone w rozdziałach od 9. do 13., przekształcając je w postać komponentów nadających się do wielokrotnego stosowania.

Oczywiście nie tylko my zajmujemy się tworzeniem takich rozwiązań — obecnie na WWW dostępnych jest bardzo wiele frameworków ułatwiających pisanie aplikacji obsługiwanych przy użyciu skryptów JavaScript i wykorzystujących technologię Ajax. Najpopularniejsze z nich mają tę ogromną zaletę, iż zostały bardzo dokładnie przetestowane przez wielu programistów.

W tym podrozdziale przedstawimy kilka bibliotek i frameworków, po które mogą sięgnąć osoby tworzące aplikacje wykorzystujące technologię Ajax. Obecnie frameworki do takich aplikacji rozwijają się bardzo dynamicznie, dlatego też nie jesteśmy w stanie szczegółowo opisać każdego z rozwiązań dostępnych na rynku; spróbujemy jednak pokazać, jakie rodzaje frameworków są dostępne, oraz w jaki sposób zastosowanie ich w tworzonych aplikacjach może ułatwić wprowadzanie porządku w pisanym przez nas kodzie.

3.5.1. Biblioteki zapewniające poprawne działanie skryptów w różnych przeglądarkach

W punkcie 3.2.1 wspominaliśmy o tym, iż aplikacje wykorzystujące technologię Ajax muszą być pisane w nieco inny sposób — w zależności od przeglądarki, w jakiej mają działać. Wiele bibliotek ułatwia programistom zdanie, udostępniając

wspólną fasadę, która niweluje rozbieżności pomiędzy poszczególnymi przeglądarkami i zapewnia możliwość korzystania z technologii Ajax w jeden spójny sposób. Niektóre z nich koncentrują się na ściśle określonych możliwościach funkcjonalnych, natomiast inne starają się stworzyć środowiska programistyczne o znacznie większym zakresie. W dalszej części rozdziału opisaliśmy te spośród dostępnych bibliotek, które uznaliśmy na za najbardziej przydatne i użyteczne podczas pisania aplikacji wykorzystujących technologię Ajax.

Biblioteka x

Biblioteka x jest dojrzałą biblioteką ogólnego przeznaczenia, służącą do tworzenia aplikacji DHTML. Opublikowana po raz pierwszy w 2001 roku, zastąpiła wcześniejszą bibliotekę autora o nazwie CBE (ang. *Cross-Browser Extensions*) i wykorzystuje znacznie prostszy styl programowania. Biblioteka ta udostępnia działające w wielu przeglądarkach funkcje pozwalające na manipulację oraz określanie stylów elementów DOM, obsługę modelu zdarzeń przeglądarki oraz gotowe funkcje do tworzenia animacji oraz obsługi operacji typu „przeciągnij-i-upuść". Można ją z powodzeniem stosować w przeglądarce Internet Explorer 4 (i nowszych), jak również w nowych wersjach Opery i przeglądarek fundacji Mozilla.

Biblioteka ta używa prostego stylu programowania opierającego się na wykorzystaniu funkcji i korzysta z dostępnej w języku JavaScript możliwości przekazywania do funkcji różnej ilości argumentów oraz braku rygorystycznej kontroli typów. Przesłania ona np. standardową metodę `document.getElementById()` wymagającą przekazywania łańcucha znaków bardziej elastyczną funkcją, która pozwala na przekazywanie zarówno łańcuchów znaków, jak i elementów DOM. W razie przekazania łańcucha znaków funkcja działa normalnie, czyli odnajduje element o podanym identyfikatorze, jednak w przypadku przekazania elementu DOM funkcja nie wykonuje żadnych operacji i zwraca element przekazany w jej wywołaniu. Możliwość użycia elementu DOM zamiast jego identyfikatora jest szczególnie przydatna i użyteczna podczas stosowania dynamicznie generowanego kodu, np. kodu przekazywanego w wywołaniu metody `setTimeout()` lub podczas określania funkcji zwrotnych.

Podobny, zwięzły styl jest używany także w metodach służących do określania stylów elementów DOM. W tym przypadku dokładnie ta sama funkcja służy zarówno do odczytu wartości stylu, jak i jej zapisu. Np. wywołanie o postaci:

```
xWidth(myElement);
```

zwróci szerokość elementu DOM określonego przez zmienną `myElement`; przy czym zmienna ta może zawierać zarówno sam element DOM, jak i jego identyfikator. Jednak umieszczając w powyższym wywołaniu dodatkowy argument, np.:

```
xWidth(myElement, 420);
```

jesteśmy w stanie określić szerokość wskazanego elementu. A zatem, aby nadać jednemu elementowi taką samą szerokość, jaką ma inny, możemy użyć wywołania:

```
xWidth(element2, xWidth(element1));
```

Biblioteka x nie zawiera żadnego kodu służącego do tworzenia i obsługi połączeń sieciowych, niemniej jednak z powodzeniem można jej użyć do tworzenia interfejsu użytkownika aplikacji wykorzystujących technologię Ajax. Warto podkreślić, iż została ona napisana w bardzo przejrzysty i zrozumiały sposób.

Sarissa

W porównaniu z biblioteką x Sarissa jest biblioteką o znacznie węższym obszarze zastosowań — służy ona głównie do obsługi danych XML w skryptach JavaScript. Biblioteka ta obsługuje komponenty ActiveX MSXML (w wersji 3. i wyższych) oraz zapewnia swoje podstawowe możliwości funkcjonalne także w przeglądarkach fundacji Mozilla, Opera, Konqueror oraz Safari; choć niektóre, bardziej zaawansowane możliwości, takie jak obsługa wyrażeń XPath oraz przekształceń XSLT, nie są dostępne we wszystkich przeglądarkach.

Z punktu widzenia aplikacji wykorzystujących technologię Ajax najważniejszą cechą biblioteki Sarissa jest działająca we wszystkich przeglądarkach możliwość tworzenia obiektu `XMLHttpRequest`. Jednak zamiast stosowania swojej własnej implementacji wzorca Façade, Sarissa używa wzorca Adapter do utworzenia obiektu, który symuluje obiekt `XMLHttpRequest` w przeglądarkach, jakie go nie udostępniają (głównie chodzi tu o Internet Explorer). Tworzony obiekt będzie oczywiście używał odpowiedniego obiektu ActiveX (który przedstawiliśmy w rozdziale 2.), jednak, z punktu widzenia programisty, po zaimportowaniu biblioteki Sarissa poniższy kod będzie działać na wszystkich przeglądarkach:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "myData.xml");
xhr.onreadystatechange = function(){
  if(xhr.readyState == 4){
    alert(xhr.responseXML);
  }
}
xhr.send(null);
```

Jeśli porównamy powyższy kod z listingiem 2.11, to zauważymy, że używane wywołania są identyczne do tych, używanych w przypadku stosowania wbudowanego obiektu `XMLHttpRequest` dostępnego w przeglądarkach fundacji Mozilla oraz w przeglądarce Opera.

Jak już wcześniej wspominaliśmy, Sarissa udostępnia także grupę ogólnych mechanizmów pozwalających na wykonywanie różnego typu operacji na dokumentach XML. Pozwala ona np. na zapisanie dowolnego obiektu JavaScript w postaci kodu XML. Mechanizmy te mogą się przydać podczas przetwarzania danych XML zwracanych przez serwer, zakładając oczywiście, że zdecydujemy się na przesyłanie kodu XML w odpowiedziach na nasze asynchroniczne żądania. (Szczegółowe informacje dotyczące możliwych sposobów prowadzenia wymiany danych pomiędzy klientem i serwerem zostały opisane w rozdziale 5.).

Prototype

Prototype jest biblioteką ogólnego przeznaczenia wspomagającą pisanie skryptów w języku JavaScript. Jej podstawowym celem jest rozszerzanie samego języka w celu umożliwienia stosowania bardziej obiektowego stylu programowania. Biblioteka ta wykorzystuje unikalny styl tworzenia kodu JavaScript, bazujący na wprowadzanych przez nią rozszerzeniach. Sam kod tworzony dzięki możliwościom tej biblioteki może być trudny do analizy, gdyż znacznie odbiega on od kodu tworzonego w językach Java i C#, jednak stosowanie zarówno samej biblioteki Prototype, jak i innych bibliotek stworzonych w oparciu o nią jest dosyć proste. Prototype można traktować jako bibliotekę przeznaczoną dla twórców innych bibliotek. Programiści tworzący aplikacje wykorzystujące technologię Ajax nie będą zapewne korzystali bezpośrednio z biblioteki Prototype, lecz z innych bibliotek wykorzystujących jej możliwości. Biblioteki te przedstawimy w dalszej części rozdziału. Na razie jednak zaprezentujemy podstawowe możliwości, jakie daje Prototype, by na ich przykładzie pokazać stosowany w niej styl programowania. Informacje te przydadzą nam się później, podczas opisywania takich rozwiązań, jak Scriptaculous, Rico oraz Ruby on Rails.

Prototype pozwala, aby jeden obiekt rozszerzał drugi — w tym celu wszystkie właściwości i metody obiektu bazowego są kopiowane do obiektu potomnego. Możliwość tę najlepiej jest przedstawić na przykładzie. Załóżmy, że zdefiniowaliśmy klasę bazową o nazwie `Vehicle`:

```
function Vehicle(numWheels,maxSpeed){
  this.numWheels=numWheels;
  this.maxSpeed=maxSpeed;
}
```

Teraz chcemy utworzyć konkretny obiekt tej klasy, który będzie reprezentować pociąg pasażerski. W naszym nowym obiekcie chcemy określić ilość wagonów oraz udostępnić mechanizm ich dodawania i usuwania. W kodzie JavaScript możliwości te można by zaimplementować w następujący sposób:

```
var passTrain=new Vehicle(24,100);
passTrain.carriageCount=12;
passTrain.addCarriage=function(){
this.carriageCount++;
}
passTrain.removeCarriage=function(){
  this.carriageCount--;
}
```

Jak widać, udało nam się zaimplementować w nowym obiekcie wszystkie potrzebne możliwości funkcjonalne. Jednak analizując ten kod pod względem architektury, można łatwo zauważyć, iż te dodatkowe możliwości funkcjonalne nie zostały umieszczone w jakiejś ogólnie pojętej „jednostce". Biblioteka Prototype może nam pomóc w rozwiązaniu tego problemu, pozwala ona bowiem na zdefiniowanie rozszerzonych możliwości w postaci obiektu, a następnie na skopiowanie ich do obiektu bazowego. W pierwszej kolejności musimy zdefiniować nowy obiekt, posiadający rozszerzone możliwości:

```
function CarriagePuller(carriageCount){
this.carriageCount=carriageCount;
this.addCarriage=function(){
    this.carriageCount++;
  }
  this.removeCarriage=function(){
    this.carriageCount--;
  }
}
```

Teraz możemy połączyć oba obiekty w jeden, dysponujący wszystkimi oczekiwanymi możliwościami:

```
var parent=new Vehicle(24,100);
var extension=new CarriagePuller(12);
var passTrain=Object.extend(parent,extension);
```

Warto zauważyć, że w pierwszej kolejności definiujemy obiekt bazowy oraz obiekt rozszerzający, a dopiero później „scalamy" je ze sobą. W tym przypadku relacja występuje pomiędzy konkretnymi obiektami, a nie pomiędzy klasami `Vehicle` oraz `CarrigePuller`. Choć nie jest to klasyczne rozwiązanie obiektowe, niemniej jednak pozwala ono na umieszczenie kodu związanego z pewną funkcją (taką jak ciągnięcie wagonów) w jednym, precyzyjnie określonym miejscu i bardzo upraszcza jego wielokrotnie stosowanie. Choć wykorzystywanie tych możliwości funkcjonalnych w niewielkim i prostym przykładzie, takim jak ten, może się wydawać raczej niepotrzebne, to jednak w dużych projektach taka hermetyzacja możliwości funkcjonalnych jest niezwykle przydatna.

Biblioteka Prototype wspomaga także stosowanie technologii Ajax — udostępnia ona klasę `Ajax` oraz grupę klas potomnych „ukrywających" przed programistą wszelkie rozbieżności związane z tworzeniem obiektu `XMLHttpRequest` w różnych przeglądarkach i ułatwiających obsługę żądań. Jedną z takich klas jest klasa `Ajax.Request`, która pozwala na generowanie żądań przy użyciu obiektu `XMLHttpRequest`. Poniżej przedstawiliśmy przykład jej użycia:

```
var req=new Ajax.Request('myData.xml');
```

Powyższy konstruktor używa stylu, który często można zobaczyć także w innych bibliotekach korzystających z Prototype. Otóż pozwala on na przekazanie dodatkowego argumentu, będącego tablicą asocjacyjną. W ten sposób, w razie konieczności, można określić bardzo wiele aspektów działania tworzonego obiektu. Dla każdej z takich opcji konfiguracyjnych została określona sensowna wartość domyślna, dzięki czemu przekazywanie takiego dodatkowego obiektu jest konieczne wyłącznie gdy chcemy przesłonić jakieś domyślne ustawienia. W przypadku konstruktora klasy `Ajax.Request` w tej dodatkowej tablicy można podać: informacje, jakie zostaną przesłane na serwer metodą POST, parametry żądania, metodę, jaka zostanie użyta do przesłania żądania na serwer, oraz funkcje zwrotne służące do obsługi odpowiedzi oraz ewentualnych błędów. Poniżej przedstawiliśmy znacznie bardziej rozbudowane wywołanie konstruktora `Ajax.Request()`:

```
var req=new Ajax.Request(
  'myData.xml',
```

```
    {
      method: 'get',
      parameters: { name:'dawid',likes:'czekoladę,rabarbar' },
      onLoaded: function(){ alert('plik wczytany!'); },
      onComplete: function(){
        alert('gotowe!\n\n'+req.transport.responseText);
      }
    }
  );
```

W powyższym przykładzie w tablicy zostały określone cztery parametry. Najpierw określiliśmy, że żądanie ma zostać przesłane metodą `get`; to ważne, gdyż Prototype domyślnie używa metody `post`. Ponieważ użyjemy metody `get`, parametry określone w kolejnym elemencie tablic (`parameters`) zostaną umieszczone w łańcuchu zapytania. W przypadku przesłania żądania metodą `post` parametry te zostałyby zapisane w treści żądania. I w końcu określamy funkcje zwrotne `onLoad` oraz `onComplete`, które zostaną wywołane po zmianie stanu właściwości `readyState` obiektu `XMLHttpRequest`. Zmienna `req.transport` zastosowana w kodzie funkcji `onComplete` zawiera referencję do obiektu `XMLHttpRequest`.

Kolejną klasą związaną z obsługą żądań, udostępnianą przez bibliotekę Prototype, jest klasa `Ajax.Updater`. Obiekty tej klasy pobierają fragmenty kodu JavaScript przesyłane z serwera i przetwarzają je. Rozwiązanie to stanowi przykład „interakcji operujących na skryptach", które opiszemy w rozdziale 5.; dokładniejsza prezentacja tego zagadnienia wykracza poza ramy tematyczne niniejszego rozdziału.

Na tym zakończymy prezentację bibliotek ułatwiających tworzenie skryptów działających w wielu różnych przeglądarkach. Przedstawiliśmy jedynie kilka wybranych bibliotek, a zamieszczone tu informacje na pewno nie są wyczerpujące. Należy także pamiętać, iż biblioteki, frameworki oraz wszelkie innego typu narzędzia do tworzenia aplikacji wykorzystujących technologię Ajax rozwijają się obecnie bardzo dynamicznie, dlatego też musieliśmy ograniczyć zamieszczone tu informacje jedynie do rozwiązań, które cieszą się największą popularnością i są najlepsze. W następnym punkcie rozdziału przedstawimy szkielety bazujące na kontrolkach, utworzone w oparciu o inne biblioteki (w tym także biblioteki, o których wspominaliśmy wcześniej).

3.5.2. Kontrolki oraz bazujące na nich szkielety

Biblioteki, które przedstawiliśmy w poprzedniej części rozdziału, zapewniały bezpieczne i pewne możliwości wykonywania operacji stosunkowo niskiego poziomu, takich jak operowanie na elementach DOM bądź pobieranie dokumentów z serwera. Dzięki zastosowaniu takich narzędzi na pewno można sobie znacznie ułatwić tworzenie funkcjonalnych interfejsów użytkownika, jednak i tak zmuszają nas one do wykonania znacznej pracy — w porównaniu z takimi bibliotekami, jak Swing, MFC bądź Qt.

Powoli zaczynają się jednak pojawiać gotowe kontrolki lub nawet całe pakiety kontrolek, których mogą używać programiści tworzący aplikacje w języku JavaScript. W typu punkcie rozdziału przedstawimy kilka wybranych rozwiązań tego

typu; chodzi nam raczej o ogólne zaprezentowanie możliwości, jakie stwarzają takie narzędzia, niż o przedstawienie ich wyczerpującego opisu.

Scriptaculous

Biblioteki Scriptaculous zawierają komponenty graficznego interfejsu użytkownika napisane w oparciu o bibliotekę Prototype (patrz poprzedni punkt rozdziału). Koncentrują się one na dwóch podstawowych grupach możliwości funkcjonalnych, choć są cały czas aktywnie rozwijane.

Biblioteka Effects definiuje wiele efektów wizualnych, których można używać do modyfikowania postaci i wyglądu elementów DOM, np. można zmieniać ich położenie, wielkość oraz przezroczystość. Poszczególne efekty może ze sobą łatwo łączyć, a dodatkowo przygotowano grupę gotowych efektów złożonych, takich jak `Puff()`, który powoduje, że wskazany element staje się coraz większy i bardziej przezroczysty, aż w końcu całkowicie znika. Kolejnym bardzo ważnym i przydatnym efektem jest efekt o nazwie `Parallel()` — pozwala on na równoczesne wykonywanie innych, wskazanych efektów. Mogą one stanowić doskonały sposób na szybkie dodanie do interfejsu użytkownika aplikacji wizualnych sygnałów zwrotnych; zagadnieniem tym zajmiemy się bardziej szczegółowo w rozdziale 6.

Wywołanie jednego z predefiniowanych efektów wizualnych jest niezwykle proste — sprowadza się do wywołania odpowiedniego konstruktora i przekazania do niego identyfikatora elementu DOM. Oto przykład:

```
new Effect.SlideDown(myDOMElement);
```

Podstawą tych wszystkich efektów jest obiekt przejścia, którego działanie można parametryzować, np. określając czas trwania przejścia oraz funkcje zwrotne, jakie mają być wykonana na jego początku i końcu. Dostępnych jest kilka podstawowych typów przejść, takich jak: linearne, sinusoidalne, pulsujące oraz drgające. Tworzenie własnych efektów sprowadza się do wybrania odpowiedniego efektu podstawowego (lub ich kombinacji) i określenia odpowiednich parametrów. Szczegółowy opis sposobu tworzenia takich efektów wykracza jednak poza ramy tematyczne tej krótkiej prezentacji. Przykład praktycznego zastosowania biblioteki Scriptaculous można znaleźć w rozdziale 6., w którym użyliśmy jej do tworzenia systemu prezentacji powiadomień.

Kolejną możliwością, jaką zapewnia biblioteka Scriptaculous jest obsługa operacji typu „przeciągnij-i-upuść". Do tego celu służy klasa `Sortable`. Wymaga ona określenia elementu rodzica i umożliwia przeciąganie i upuszczanie wszystkich jego elementów potomnych. Przy użyciu opcji podawanych w wywołaniu konstruktora można określić funkcje zwrotne wywoływane w momencie rozpoczynania i kończenia przeciągania, typy elementów, jakie można przeciągać, oraz listę „celów" przeciągania (czyli elementów, w których można upuszczać przeciągane elementy). Istnieje także możliwość przekazywania jako opcji obiektów efektów wizualnych określających, jak element będzie się zachowywać w momencie rozpoczynania przeciągania, podczas samego przeciągania oraz w chwili upuszczania.

Rico

Pakiet Rico, podobnie jak biblioteki Scriptaculous, korzysta z możliwości, jakie daje biblioteka Prototype. Rico pozwala na stosowanie operacji „przeciągnij-i-upuść" oraz różnego rodzaju efektów wizualnych, stwarzając przy tym bardzo duże możliwości ich konfigurowania. Oprócz tego Rico udostępnia pojęcie obiektu zachowania — `Behavior` — czyli fragmentu kodu, który można dodać do fragmentu drzewa DOM, aby później rozszerzać jego działanie. Pakiet zawiera kilka przykładów zastosowania tych obiektów; jednym z nich jest komponent `Accordion`, który umieszcza kilka elementów DOM w jednym obszarze, a następnie pozwala na wyświetlanie zawartości wybranego z nich. (Komponenty o takim działaniu często określa się jako **pasek programu Outlook**, gdyż zostały one spopularyzowane w programie Microsoft Outlook).

Spróbujmy zatem teraz utworzyć prosty przykład komponentu Accordion. W pierwszej kolejności potrzebujemy nadrzędnego elementu DOM — każdy z elementów umieszczonych wewnątrz niego stanie się jednym panelem tworzonego komponentu. Każdy panel definiujemy jako element `<div>`, wewnątrz którego zostają umieszczone dwa kolejne elementy `<div>` reprezentujące odpowiednio: nagłówek oraz treść panelu:

```
<div id='myAccordion'>
  <div>
    <div>Definicja</div>
    <div>
      <ul>
        <li><b>n.</b>instrument muzyczny z grupy idiofonów -
          instrymentów perkusyjnych. Rodzaj harmonii, oparty na
          stroikach przelotowych. Prąd powietrza, zagęszczonego
          przez ręcznie poruszany miech, wywołuje dźwięk, wprawiając
          w drganie stroik (źródłem dźwięku jest zatem drgająca
          blaszka, a nie słup powietrza - dlatego
          instrument perkusyjny, nie aerofon).</li>
      </ul>
    </div>
  </div>
  <div>
    <div>Rysunek</div>
    <div>
     <img src='monkey-accordion.jpg'></img>
    </div>
  </div>
</div>
```

Pierwszy panel zawiera fragment definicji opisującej, czym jest **akordeon**, natomiast drugi przedstawia jego zdjęcie (patrz rysunek 3.9). Powyższy kod HTML wyświetlony w swojej oryginalnej postaci będzie przedstawiał oba panele — jeden nad drugim. Jednak w głównym elemencie naszego kodu określiliśmy identyfikator (jako wartość atrybutu `id`), co pozwala nam przekazać referencję do tego elementu w wywołaniu konstruktora obiektu `Accordion`. Obiekt ten tworzymy w następujący sposób:

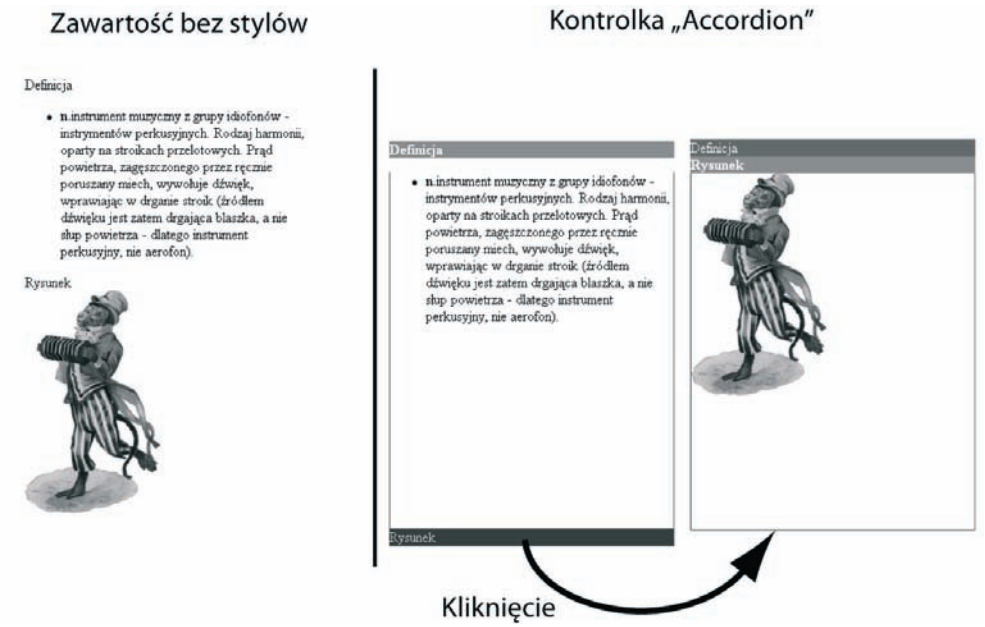
```
var outer=$('myAccordion');
outer.style.width='320px';
new Rico.Accordion(
  outer,
  { panelHeight:400,
    expandedBg:'#909090',
    collapsedBg:'#404040',
  }
);
```

Pierwszy wiersz powyższego fragmentu kodu wygląda dosyć ciekawie. Okazuje się jednak, że znak `$` jest normalną, dopuszczalną nazwą zmiennej w języku JavaScript, a biblioteka Prototype zastosowała go jako nazwę funkcji. A zatem funkcja `$()` przypomina nieco funkcję `xGetElementById()`, którą poznaliśmy przy okazji prezentacji biblioteki x. Pobraną referencję do elementu DOM przekazujemy w wywołaniu konstruktora obiektu `Accordion` wraz z dodatkową tablicą opcji. W naszym przypadku opcje określają jedynie style, jakie należy zastosować podczas prezentowania komponentu na stronie; jednak w tablicy opcji można także określić funkcje zwrotne, wywoływane w momencie otwierania i zamykania paneli. Efekty zastosowania obiektu `Accordion` do określenia postaci zwyczajnych elementów `<div>` zostały przedstawione na rysunku 3.9. Klasa `Behavior` pakietu Rico pozwala na bardzo łatwe tworzenie komponentów na podstawie kodu HTML, a oprócz tego oddziela ich zawartość od kodu określającego sposób ich działania. Zagadnienia związane ze stosowaniem dobrych i sprawdzonych rozwiązań podczas tworzenia interfejsów użytkownika w języku JavaScript przedstawimy w rozdziale 4.

Ostatnią cechą pakietu Rico, o jakiej chcieliśmy tu wspomnieć, są doskonałe narzędzia do generowania i obsługi asynchronicznych żądań. Do tego celu używany jest globalny obiekt `AjaxEngine`. Jest on czymś znacznie więcej niż jedynie „pojemnikiem" ukrywającym w sobie zwyczajny obiekt `XMLHttpRequest` i pozwalającym na jego stosowanie w różnych przeglądarkach. Obiekt ten definiuje format odpowiedzi XML, który powinien się składać z sekwencji elementów `<response>`. Mechanizm udostępniany przez pakiet Rico potrafi automatycznie dekodować takie odpowiedzi, przy czym rozróżnia dwa ich rodzaje: odpowiedzi modyfikujące bezpośrednio zawartość elementów DOM oraz modyfikujące wartości przechowywane w obiektach JavaScript. Podobne rozwiązania przedstawimy szczegółowo w punkcie 5.5.3 poświęconym zagadnieniom interakcji pomiędzy klientem a serwerem. Na razie zajmiemy się jednak frameworkami, które spajają części aplikacji działającej na serwerze oraz w kliencie.

3.5.3. Frameworki aplikacji

Wszystkie frameworki, jakie przedstawiliśmy wcześniej, działają jedynie w przeglądarce i można je stosować jako statyczne pliki JavaScript pobierane z serwera. Ostatnią grupą rozwiązań, jakie chcielibyśmy tu przedstawić, są frameworki, które są przechowywane na serwerze, lecz dynamicznie generują fragmenty kodu JavaScript i HTML.



Rysunek 3.9. Obiekty Behavior dostępne we frameworku Rico pozwalają na przekształcanie zwyczajnych węzłów DOM w interaktywne kontrolki. W tym celu wystarczy przekazać do konstruktora odpowiedniego obiektu Behavior identyfikator głównego węzła. W tym przypadku grupa elementów (widoczna po lewej stronie) została przekształcona w interaktywną kontrolkę menu (pokazaną z prawej strony), w której kliknięcia otwierają i zamykają poszczególne panele

Są to zdecydowanie najbardziej złożone ze wszystkich frameworków, o jakich wspominamy w tej książce, i niestety, nie będziemy w stanie przedstawić ich wyczerpująco i szczegółowo. Pomimo to chcielibyśmy zamieścić krótki opis cech i możliwości takich rozwiązań. Do zagadnień związanych z takimi frameworkami wrócimy jednak w rozdziale 5.

DWR, JSON-RPC oraz SAJAX

Zaczniemy od przedstawienia trzech niewielkich frameworków działających po stronie serwera. Opiszemy je wspólnie, gdyż posiadają wiele tych samych cech, choć zostały napisane w różnych językach. Framework może współpracować z wieloma różnymi językami, takimi jak: PHP, Python, Perl oraz Ruby. DWR (ang. *Direct Web Remoting*) został napisany w języku Java, przy czym wykorzystuje to samo podejście, polegające na udostępnianiu metod obiektów, a nie samych funkcji. Trzeci z wymienionych frameworków, JSON-RPC (ang. *JavaScript Object Notation-base Remote Procedure Calls*), został zaprojektowany dokładnie w taki sam sposób. Także i on umożliwia korzystanie z języków JavaScript, Python, Ruby, Perl oraz Java.

Wszystkie te frameworki pozwalają, by metody obiektów działających na serwerze były udostępniane jako żądania Ajaksa. Często zdarza się, że dysponujemy wykonywaną na serwerze funkcją generującą przydatne informacje,

które następnie muszą być określone na serwerze, gdyż np. bazują na informacjach pobieranych z bazy danych. Opisywane tu frameworki zapewniają wygodny sposób dostępu do tych funkcji lub metod z poziomu przeglądarki i z powodzeniem można je zastosować do udostępnienia obiektów modelu dziedziny używanego na serwerze w kodzie wykonywanym w przeglądarce.

Przyjrzyjmy się przykładowi zastosowania frameworka SAJAX, w którym udostępnimy funkcję używaną na serwerze i napisaną w języku PHP. Nasza funkcja jest wyjątkowo prosta, gdyż jej działanie ogranicza się do zwrócenia łańcucha znaków:

```
<?php
function sayHello(name){
  return("Hello! {$name} Ajax in Action!!!!");
}
?>
```

Aby udostępnić tę funkcję dla kodu JavaScript wykonywanego w przeglądarce, należy zaimportować mechanizm SAJAX i wywołać funkcję `sajax_export()`:

```
<?php
require('Sajax.php');
sajax_init();
sajax_export("sayHello");
?>
```

Następnie, pisząc kod dynamicznie generowanej strony WWW, używany kolejnych funkcji frameworka do wygenerowania kodu pozwalającego na odwoływanie się do wyeksportowanej wcześniej funkcji. Ten wygenerowany kod definiuje w przeglądarce lokalną funkcję, której sygnatura odpowiada sygnaturze funkcji działającej na serwerze:

```
<script type='text/javascript'>
<?
sajax_show_javascript();
?>
...
alert(sayHello("Dawid"));
...
</script>
```

Wywołanie funkcji `sayHello("Dawid")` w przeglądarce spowoduje, że wygenerowany kod JavaScript prześle na serwer żądanie, które spowoduje wykonanie odpowiedniej funkcji i przekazanie zwróconych przez nią wyników w odpowiedzi HTTP. Następnie, po stronie przeglądarki, odebrane wyniki zostaną przetworzone i udostępnione dla kodu JavaScript. Programista nie musi zajmować się jakimikolwiek aspektami korzystania z technologii Ajax — wszystkie niezbędne operacje są wykonywane w niewidoczny sposób przez biblioteki SAJAX.

Prezentowane tu frameworki kojarzą funkcje i obiekty dostępne na serwerze z kodem JavaScript wykonywanym w przeglądarce — na stosunkowo niskim poziomie abstrakcji. Automatyzują one czynności, których wykonanie byłoby żmudne i pracochłonne, jednak stosując je, ryzykujemy, że udostępnimy zbyt wiele możliwości funkcjonalnych zaimplementowanych na serwerze. Zagadnienia te opiszemy bardziej szczegółowo w rozdziale 5.

Pozostałe frameworki, które opiszemy w dalszej części rozdziału, prezentują znacznie bardziej złożone podejście, gdyż pozwalają na generację całych warstw interfejsu użytkownika na podstawie modelu zadeklarowanego na serwerze. Choć wewnętrznie frameworki te wykorzystują standardowe możliwości technologii Ajax, jednak z punktu widzenia twórców aplikacji udostępniają swój własny model programowania. W efekcie korzystanie z nich raczej w niewielkim stopniu przypomina stosowanie ogólnej technologii Ajax, dlatego też w niniejszej książce będziemy w stanie przedstawić je jedynie bardzo krótko i pobieżnie.

Backbase

Backbase Presentation Server udostępnia bardzo bogaty zbiór kontrolek kojarzących wykonawczą część aplikacji ze znacznikami XML umieszczonymi w dokumencie HTML generowanym przez serwer. Zasada działania tego frameworka przypomina nieco działanie komponentów `Behavior` pakietu Rico, z tym że Backbase używa własnego zbioru elementów XHTML reprezentujących komponenty interfejsu użytkownika, a nie zwyczajnych znaczników HTML.

Część frameworka działająca po stronie serwera jest dostępna w wersji napisanej w języku Java oraz w wersji przeznaczonej dla platformy .NET. Backbase jest produktem komercyjnym, jednak dostępna jest także wersja, z której można korzystać bezpłatnie.

Echo2

Echo2 firmy NextApp's jest napisanym w Javie frameworkiem wykonywanym po stronie serwera, który generuje zaawansowane kontrolki na podstawie modelu interfejsu użytkownika zadeklarowanego na serwerze. Po uruchomieniu w przeglądarce kontrolki interfejsu użytkownika są raczej niezależne i będą obsługiwały czynności wykonywane przez użytkownika lokalnie, używając odpowiedniego kodu JavaScript, bądź też będą przesyłać żądania na serwer, używając przy tym kolejek, podobnie jak ma to miejsce w przypadku pakietu Rico.

Echo2 jest reklamowany jako rozwiązanie wykorzystujące technologię Ajax, które od swych użytkowników nie wymaga żadnej znajomości języka HTML, JavaScript ani CSS, oczywiście zakładając, że nie będziemy chcieli rozszerzać lub zmieniać domyślnego zbioru dostępnych komponentów. W większości przypadków tworzenie interfejsu użytkownika odbywa się wyłącznie w języku Java. Echo2 jest frameworkiem typu „open source", rozpowszechnianym na warunkach licencji Mozilla, która daje możliwość stosowania go w aplikacjach komercyjnych.

Ruby on Rails

Ruby on Rails jest frameworkiem służącym do tworzenia aplikacji internetowych w języku Ruby. Łączy on w sobie rozwiązania pozwalające na kojarzenie obiektów wykonywanych na serwerze z bazą danych, prezentowanie zawartości przy użyciu szablonów i działa w sposób bardzo przypominający wzorzec MVC opisany w podrozdziale 3.4. Ruby on Rails zapewnia bardzo szybki proces

tworzenia niewielkich i średnich aplikacji internetowych, gdyż pozwala on na automatyczne generowanie często powtarzającego się kodu. Oprócz tego twórcy frameworka starali się zminimalizować ilość czynności konfiguracyjnych koniecznych do zapewnienia poprawnego działania aplikacji.

W nowszych wersjach Ruby on Rails zapewnia możliwość wykorzystywania technologii Ajax poprzez udostępnienie biblioteki Prototype. Oba te narzędzia współpracują ze sobą w bardzo ścisły i naturalny sposób, gdyż kod JavaScript korzystający z biblioteki Prototype jest generowany automatycznie. Co więcej, zarówno Ruby, jak i Prototype wykorzystują podobny styl programowania. Podobnie framework Echo2, także Ruby on Rails pozwala na stosowanie technologii Ajax bez konieczności dobrej znajomości technologii podstawowych, takich jak JavaScript. Jednak osoby znające JavaScript mogą rozbudowywać mechanizmy wykorzystania Ajaksa dostępne w tym frameworku.

Informacjami o frameworku Ruby on Rails zakończyliśmy prezentację narzędzi i rozwiązań wspomagających wykorzystywanie technologii Ajax. Jak zaznaczyliśmy na samym początku, narzędzia te rozwijają się obecnie bardzo dynamicznie, dotyczy to także znacznej części prezentowanych tu frameworków i bibliotek.

Warto zauważyć, że większość z prezentowanych tu bibliotek i frameworków charakteryzuje się swoimi własnymi rozwiązaniami i unikalnym stylem programowania. Pisząc przykłady do niniejszej książki, staraliśmy się przedstawiać charakter poszczególnych technologii i technik oraz unikać zbyt częstego stosowania jednego konkretnego frameworka. Niemniej jednak Czytelnik na pewno zauważy, że wzmianki o niektórych frameworkach i bibliotekach zawarte w tym rozdziale będą się pojawiały także w innych miejscach książki.

3.6. Podsumowanie

W tym rozdziale przedstawiliśmy pojęcie refaktoryzacji jako sposobu na poprawę jakości i elastyczności kodu. Nasze pierwsze spotkanie z refaktoryzacją polegało na zapewnieniu łatwych możliwości wielokrotnego stosowania obiektu `XMLHttpRequest` w wielu różnych przeglądarkach.

Przedstawiliśmy także kilka wzorców projektowych przydatnych do rozwiązywania problemów, które najczęściej występują podczas tworzenia kodu aplikacji wykorzystujących technologię Ajax. Wzorce projektowe stanowią nieformalny sposób pozyskiwania wiedzy zdobytej przez innych programistów, którzy wcześniej rozwiązywali podobne problemy, i pozwalają nam dążyć do określonego celu.

Wzorce Façade oraz Adapter pozwalają ukrywać różnice występujące pomiędzy różnymi używanymi implementacjami. W aplikacjach wykorzystujących technologię Ajax oba te wzorce są szczególnie przydatne, gdyż pozwalają nam stworzyć warstwę chroniącą nasz kod przed rozbieżnościami występującymi w poszczególnych przeglądarkach. A właśnie te rozbieżności są podstawowym utrapieniem programistów tworzących aplikacje w języku JavaScript.

Kolejnym przedstawionym przez nas wzorcem projektowym był Observer. Jest to elastyczny wzorzec wykorzystywany w systemach sterowanych zdarzeniami. Wrócimy do niego w rozdziale 4., gdzie będziemy zajmowali się warstwami interfejsu użytkownika występującymi w aplikacjach internetowych. Observer w połączeniu z wzorcem projektowym Command, stanowiącym doskonały sposób hermetyzacji czynności wykonywanych przez użytkownika, umożliwia utworzenie solidnego rozwiązania służącego do obsługi poczynań użytkownika na witrynie i zapewniającego możliwość odtwarzania ich skutków. Wzorzec Command można także stosować podczas określania sposobu organizacji interakcji pomiędzy klientem a serwerem; tymi zagadnieniami zajmiemy się w rozdziale 5.

Wzorzec Singleton stanowi bardzo prosty i skuteczny sposób kontrolowania dostępu do konkretnego zasobu. W aplikacjach wykorzystujących technologię Ajax można go z powodzeniem używać do kontroli dostępu do sieci. Stosowne przykłady zamieściliśmy w rozdziale 5.

W końcu przedstawiliśmy także wzorzec projektowy MVC (ang. *Model-View-Controller*). Jest to wzorzec architektoniczny, który od wielu lat (przynajmniej w krótkiej historii Sieci) jest stosowany do tworzenia aplikacji internetowych. W niniejszym rozdziale opisaliśmy, jak zastosowanie tego wzorca może poprawić elastyczność aplikacji. Przedstawiliśmy też przykład, w którym zastosowaliśmy model obiektowy stworzony w oparciu o mechanizm ORM oraz system obsługi szablonów.

Na przykładzie aplikacji obsługującej sklep odzieżowy zademonstrowaliśmy, w jaki sposób można jednocześnie używać refaktoryzacji oraz wzorców projektowych. Bardzo trudno jest już za pierwszym razem stworzyć idealnie zaprojektowany kod, jednak z powodzeniem można zastosować refaktoryzację oraz „nieeleganckie", lecz użyteczne fragmenty kodu (takie jak kod z listingu 3.4), by powoli, etapami, wprowadzać wzorce projektowe do naszej aplikacji.

Pod koniec rozdziału przedstawiliśmy grupę bibliotek oraz frameworków. Stosowanie takich narzędzi nie tylko ułatwia nam pracę, lecz także pozwala na wprowadzenie porządku w tworzonym kodzie. Obecnie pojawia się coraz więcej bibliotek i frameworków przeznaczonych do użycia w aplikacjach wykorzystujących technologię Ajax, zaczynając od zbiorów kontrolek działających we wszystkich przeglądarkach, a kończąc na kompletnych rozwiązaniach wykorzystywanych zarówno po stronie klienta, jak i serwera. Pokrótce przedstawiliśmy kilka najpopularniejszych frameworków, a nieco więcej informacji o niektórych spośród nich będzie można znaleźć w dalszej części książki.

W dwóch kolejnych rozdziałach spróbujemy zastosować zdobytą wiedzę o refaktoryzacji oraz wzorcach projektowych podczas tworzenia kodu JavaScript działającego po stronie przeglądarki, a następnie do zaimplementowania systemu komunikacji pomiędzy klientem a serwerem. Przykłady te pomogą nam w opracowaniu niezbędnej terminologii oraz zespołu praktycznych umiejętności przydatnych podczas tworzenia solidnych, wielofunkcyjnych aplikacji internetowych.

3.7. Zasoby

Martin Fowler (oraz współautorzy: Kent Beck, John Brant, William Opdyke oraz Don Roberts) napisał bardzo cenioną i popularną książkę dotyczącą refaktoryzacji: *Refactoring: Improving the Design of Existing Code* (Addison-Wesley Professional 1999).

Erich Gamma, Richard Helm, Ralph Johnson oraz Johm Vlissides (określani jako „The Gang of Four", czyli „Gang Czworga"), napisali książkę *Design Patterns* (Addison-Wesley Professional, 1995), która wywarła ogromny wpływ na środowisko projektantów oprogramowania oraz programistów.

Erich Gamma został później architektem zajmującym się tworzeniem zintegrowanego środowiska programistycznego Eclipse (patrz dodatek A). Informacje o Eclipse oraz wzorcach projektowych można znaleźć w wywiadzie opublikowanym na stronie *http:/www.artima.com/lejava/articles/gammadp.html*.

Michael Mahemoff stworzył ostatnio witrynę, *http://www.ajaxpatterns.org/*, poświęconą katalogowaniu wzorców projektowych stosowanych w aplikacjach wykorzystujących technologię Ajax.