Technologia i rozwiązania

Biblioteka jQuery
Sprawdzone wzorce projektowe

Thodoris Greasidis

Helion

[PACKT] PUBLISHING open source community experience distilled

Tytuł oryginału: jQuery Design Patterns

Tłumaczenie: Piotr Pilch

ISBN: 978-83-283-2832-7

First published in the English language under the title ‘jQuery Design Patterns – 9781785888687’.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/bjqswp.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/bjqswp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 1

Utrwalenie wiedzy o bibliotece jQuery i wzorcu Kompozyt

Przed pojawieniem się standardu **Web 2.0** serwisy internetowe były jedynie nośnikiem opartym na dokumentach, który oferował tylko wzajemne połączenie różnych stron/dokumentów oraz obsługę skryptową po stronie klienta ograniczoną przeważnie do sprawdzania poprawności formularza. W 2005 roku pojawiły się serwisy Gmail i Google Maps, a JavaScript stał się językiem używanym przez duże przedsiębiorstwa do tworzenia aplikacji o dużej skali i zapewniania rozbudowanych interakcji opartych na interfejsie użytkownika.

Nawet pomimo tego, że od czasu pojawienia się języka JavaScript dokonano w nim bardzo niewielu zmian, znacząco zmieniły się oczekiwania świata przedsiębiorców względem możliwości stron internetowych. Odtąd projektanci witryn internetowych musieli zapewniać złożone interakcje z użytkownikiem. Wreszcie na rynku zaczął funkcjonować termin „aplikacja internetowa". W efekcie oczywiste zaczynało stawać się to, że projektanci powinni tworzyć pewne abstrakcje kodu, definiować sprawdzone procedury i adaptować wszystkie możliwe do zastosowania **wzorce projektowe** (ang. *design patterns*), jakie informatyka mogła zaoferować. Powszechne wykorzystanie języka JavaScript w przypadku aplikacji dla przedsiębiorstw ułatwiło jego rozwój, który został rozszerzony za pomocą specyfikacji **EcmaScript2015/EcmaScript6** (**ES6**) w sposób pozwalający na łatwe użycie jeszcze większej liczby wzorców projektowych.

W sierpniu 2006 roku po raz pierwszy została udostępniona przez Johna Resiga biblioteka jQuery w witrynie o adresie *http://jquery.com/*. Biblioteka została stworzona w celu zapewnienia wygodnego w użyciu interfejsu API służącego do lokalizacji elementów modelu DOM. Od tego czasu biblioteka stanowi integralną część pakietu narzędziowego projektanta aplikacji

internetowych. U swoich podstaw biblioteka jQuery korzysta z kilku wzorców projektowych i próbuje zachęcić projektanta do używania ich za pośrednictwem zapewnianych przez siebie metod. Wzorzec Kompozyt (ang. *Composite Pattern*) to jeden z tych wzorców ujawniany projektantowi za pomocą bardzo podstawowej metody `jQuery()`, która służy do wykonywania operacji przemieszczania się w obrębie modelu DOM i stanowi jeden z najbardziej wyróżniających się elementów biblioteki jQuery.

W tym rozdziale zostały omówione następujące zagadnienia:

- Utrwalenie wiedzy na temat obsługi skryptowej modelu DOM za pomocą biblioteki jQuery.
- Wprowadzenie do wzorca Kompozyt.
- Sposób wykorzystania wzorca Kompozyt przez bibliotekę jQuery.
- Korzyści wynikające z użycia biblioteki jQuery zamiast zwykłych operacji modyfikacji modelu DOM za pomocą kodu w JavaScripcie.
- Wprowadzenie do wzorca Iterator.
- Zastosowanie wzorca Iterator w przykładowej aplikacji.

Biblioteka jQuery i obsługa skryptowa modelu DOM

Dzięki obsłudze skryptowej modelu DOM odwołujemy się do dowolnej procedury zmieniającej lub przetwarzającej elementy strony internetowej po załadowaniu jej przez przeglądarkę. DOM API to interfejs API języka JavaScript poddany standaryzacji w 1998 roku. Zapewnia on projektantom aplikacji internetowych kolekcję metod, które pozwalają na modyfikowanie elementów drzewa modelu DOM tworzonych przez przeglądarkę po załadowaniu i poddaniu analizie składniowej kodu HTML strony internetowej.

> Więcej informacji o modelu DOM (*Document Object Model*) i jego interfejsach API znajdziesz na stronie internetowej o adresie *https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction*.

Wykorzystując interfejs API modelu DOM w swoim kodzie napisanym w JavaScripcie, projektanci aplikacji internetowych mogą modyfikować węzły tego modelu i dodawać nowe elementy lub usuwać elementy istniejące w obrębie strony. Początkowo podstawowym przypadkiem użycia obsługi skryptowej modelu DOM było jedynie sprawdzanie poprawności formularzy po stronie klienta. Jednakże z upływem lat i w miarę jak JavaScript zyskiwał uznanie w świecie przedsiębiorstw zaczęto implementować bardziej złożone interakcje z użytkownikiem.

Początkowa wersja biblioteki jQuery pojawiła się w sierpniu 2006 roku. Jej zadaniem było ułatwienie projektantom aplikacji internetowych modyfikowania drzewa modelu DOM i wykonywania

operacji przechodzenia w jego obrębie. Jednym z głównych celów biblioteki było zapewnienie abstrakcji, które umożliwiały tworzenie krótszego, bardziej czytelnego i mniej podatnego na błędy kodu, a jednocześnie zagwarantowanie zgodności z różnymi przeglądarkami.

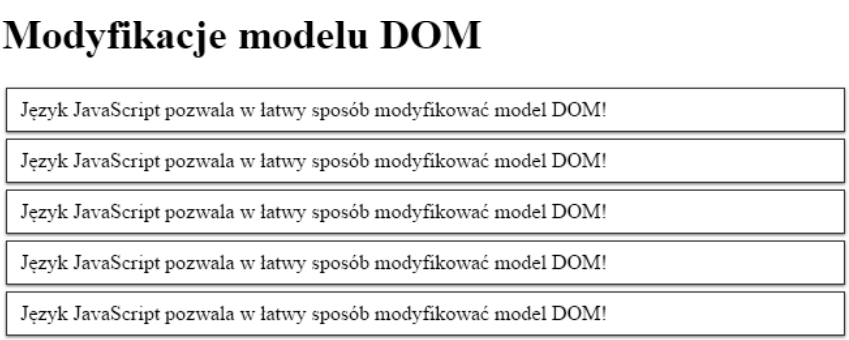
Powyższe podstawowe zasady obowiązujące w przypadku biblioteki jQuery są wyraźnie widoczne na jej stronie internetowej, na której zaprezentowano je w następujący sposób:

> *(…)szybka, niewielka i bogata w możliwości biblioteka języka JavaScript. Sprawia ona, że takie działania, jak modyfikowanie dokumentów HTML i przechodzenie w ich obrębie, obsługa zdarzeń, używanie animacji i technologii Ajax, są znacznie prostsze z wykorzystaniem prostego w użyciu interfejsu API, który współpracuje z wieloma przeglądarkami. Łącząc wszechstronność i rozszerzalność, biblioteka jQuery zmieniła sposób, w jaki miliony osób pisze kod w JavaScripcie.*

Zapewniane od początku przez bibliotekę jQuery interfejsy API z abstrakcjami, a także sposób organizacji różnych wzorców projektowych, przyczyniły się do wysokiego poziomu akceptacji biblioteki przez projektantów aplikacji internetowych. W rezultacie, zgodnie z kilkoma źródłami informacji, takimi jak serwis BuiltWith.com (*http://trends.builtwith.com/javascript/jQuery*), biblioteka jQuery jest wykorzystywana przez ponad 60% najczęściej odwiedzanych witryn internetowych z całego świata.

Modyfikowanie modelu DOM za pomocą biblioteki jQuery

Aby utrwalić informacje na temat biblioteki jQuery, omówimy przykładową stronę internetową, w której przypadku są wykonywane proste operacje modyfikowania modelu DOM. W przykładzie zostanie załadowana prosta strona ze strukturą, która początkowo wygląda podobnie jak na poniższym rysunku.



Do zmiany treści i układu strony zostanie użyty kod jQuery. Aby efekty uruchomienia kodu były wyraźnie widoczne, zostanie on tak skonfigurowany, żeby działał przez około 700 milisekund od momentu załadowania strony. Wynik operacji modyfikowania zaprezentowano na rysunku na następnej stronie.

Przejrzyjmy kod HTML użyty w poprzednim przykładzie:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Modyfikacje modelu DOM</title>
    <link rel="stylesheet" type="text/css"
      href="dom-manipulations.css">
  </head>
  <body>
    <h1 id="pageHeader">Modyfikacje modelu DOM</h1>
    <div class="boxContainer">
      <div>
        <p class="box">
          Język JavaScript pozwala w łatwy sposób modyfikować model DOM!
        </p>
      </div>
      <div>
        <p class="box">
          Język JavaScript pozwala w łatwy sposób modyfikować model DOM!
        </p>
      </div>
      <div>
        <p class="box">
          Język JavaScript pozwala w łatwy sposób modyfikować model DOM!
        </p>
      </div>
    </div>
    <p class="box">
      Język JavaScript pozwala w łatwy sposób modyfikować model DOM!
    </p>
    <p class="box">
      Język JavaScript pozwala w łatwy sposób modyfikować model DOM!
    </p>
    <script type="text/javascript"
      src="https://code.jquery.com/jquery-2.2.0.min.js"></script>
```

```
    <script type="text/javascript"
      src="jquery-dom-manipulations.js"></script>
  </body>
</html>
```

Użyty kod CSS jest dość prosty. Zawiera on tylko trzy następujące klasy arkuszy stylów CSS:

```
.box {
    padding: 7px 10px;
    border: solid 1px #333;
    margin: 5px 3px;
    box-shadow: 0 1px 2px #777;
}
.boxsizer {
    float: left;
    width: 33.33%;
}
.clear { clear: both; }
```

Powyższy kod generuje stronę podobną do pokazanej na pierwszym rysunku (po wyświetleniu jej w przeglądarce i przed wykonaniem kodu napisanego w JavaScripcie). W kodzie CSS zdefiniowano najpierw kilka stylów podstawowych dla klas CSS `box`, `boxsizer` i `clear`. Aby elementy przypominały pole, klasa `box` określa style powiązanych elementów znajdujących się na stronie przy użyciu dopełniania, cienkiej ramki, marginesu oraz niewielkiego cienia poniżej elementów. Klasa `boxsizer` sprawi, że korzystające z niej elementy będą mieć tylko jedną trzecią szerokości ich elementu nadrzędnego, a ponadto utworzą układ złożony z trzech kolumn. I wreszcie klasa `clear` będzie stosowana w elemencie w roli punktu przerwania dla układu kolumnowego. W efekcie wszystkie elementy następujące po tym elemencie zostaną umiejscowione poniżej niego. Klasy `boxsizer` i `clear` nie są początkowo wykorzystywane przez żaden element zdefiniowany w kodzie HTML, ale będą używane po zmodyfikowaniu modelu DOM za pomocą kodu w JavaScripcie.

W elemencie `<body>` kodu HTML na początku definiowany jest element nagłówka `<h1>` z identyfikatorem `pageHeader`, aby z łatwością element ten mógł być wybierany w obrębie kodu JavaScript. Tuż poniżej elementu `<h1>` definiujemy pięć elementów akapitu (`<p>`) z klasą `box`. Spośród tych elementów pierwsze trzy opakowano za pomocą trzech elementów `<div>`, a następnie przy użyciu kolejnego elementu `<div>` z klasą `boxContainer`.

Po osiągnięciu dwóch znaczników `<script>` dołączane jest najpierw odwołanie do biblioteki jQuery zapewnianej w ramach jej sieci CDN. Więcej informacji możesz znaleźć pod adresem *http://code.jquery.com/*. W drugim znaczniku `<script>` ma miejsce odwołanie do pliku JavaScriptu z kodem wymaganym w omawianym przykładzie. Ma on następującą postać:

```
setTimeout(function() {
    $('#pageHeader').css('font-size', '3em');
    var $boxes = $('.boxContainer .box');
    $boxes.append(
      '<br /><br /><i>W sytuacji, gdy niezbędne są proste rozwiązania</i>.');
```

```
        $boxes.parent().addClass('boxsizer');
        $('.boxContainer').append('<div class="clear">');
    }, 700);
```

W celu opóźnienia jego wykonania, zgodnie z wcześniej opisanym przypadkiem użycia, cały kod opakowano za pomocą wywołania funkcji `setTimeout`. Pierwszy parametr tego wywołania to funkcja anonimowa, która zostanie wykonana po upływie 700 milisekund (określonych w drugim parametrze) mierzonych przez licznik.

W pierwszym wierszu funkcji anonimowej wywołania zwrotnego użyto funkcji `$()` biblioteki jQuery, aby dokonać przejścia w obrębie modelu DOM i zlokalizować element o identyfikatorze `pageHeader`, a następnie zastosować metodę `css()` do zwiększenia rozmiaru czcionki dla tego elementu przez ustawienie wartości `3em` dla właściwości `font-size`. W dalszej kolejności funkcji `$()` zapewniany jest bardziej złożony selektor CSS, co ma na celu zlokalizowanie wszystkich elementów z klasą `box`, które są elementami potomnymi elementu z klasą `boxContainer`. Wynik operacji jest zapisywany w zmiennej o nazwie `$boxes`.

> **Konwencje nazewnicze dotyczące zmiennych**
>
> Wśród projektantów powszechną praktyką jest stosowanie konwencji nazewniczych w wypadku zmiennych, które przechowują obiekty określonego typu. Korzystanie z takich konwencji nie tylko ułatwia zapamiętywanie tego, co przechowuje zmienna, ale też sprawia, że napisany kod jest łatwiejszy do zrozumienia przez innych programistów wchodzących w skład zespołu. Projektanci biblioteki jQuery często używają nazw zmiennych zaczynających się od znaku $, gdy zmienna przechowuje wynik funkcji `$()` (znana również jako obiekt kolekcji biblioteki jQuery).

Po zajęciu się interesującymi nas elementami z klasą `box` na końcu każdego z nich dołączamy dwie spacje łamiące i dodatkowy tekst w postaci kursywy. Dalej używana jest zmienna `$boxes` i za pomocą metody `parent()` wykonywana jest operacja przejścia o jeden poziom w górę drzewa modelu DOM. Metoda zwraca inny obiekt biblioteki jQuery przechowujący elementy nadrzędne `<div>` początkowo wybranych pól. W ramach łańcucha dołączane jest następnie wywołanie metody `addClass()`, która przypisuje tym polom klasę CSS `boxsizer`.

> Jeśli konieczne jest przejście wszystkich węzłów nadrzędnych wybranego elementu, możesz skorzystać z metody `$.fn.parents()`. Aby jedynie znaleźć pierwszy element nadrzędny zgodny z danym selektorem CSS, rozważ użycie metody `$.fn.closest()`.

Ponieważ do uzyskania trójkolumnowego układu klasa `boxsizer` używa elementów unoszących się, muszą one zostać wyczyszczone w klasie `boxContainer`. I tym razem przechodzenie między elementami modelu DOM umożliwia prosty selektor CSS `.boxContainer` i funkcja `$()`. Dalej wywoływana jest metoda `.append()` w celu utworzenia nowego elementu `<div>` z klasą CSS `.clear`. Element wstawiany jest na końcu kodu elementu `boxContainer`.

Po upływie 700 milisekund wykonywanie kodu jQuery zostanie ukończone. Uzyskamy zaprezentowany wcześniej trójkolumnowy układ. W swoim końcowym stanie kod HTML elementu `boxContainer` będzie mieć następującą postać:

```
<div class="boxContainer">
  <div class="boxsizer">
    <p class="box">
      Język JavaScript pozwala w łatwy sposób modyfikować model DOM!
      <br><br><i>W sytuacji, gdy niezbędne są proste rozwiązania</i>.
    </p>
  </div>
  <div class="boxsizer">
    <p class="box">
      Język JavaScript pozwala w łatwy sposób modyfikować model DOM!
      <br><br><i>W sytuacji, gdy niezbędne są proste rozwiązania</i>.
    </p>
  </div>
  <div class="boxsizer">
    <p class="box">
      Język JavaScript pozwala w łatwy sposób modyfikować model DOM!
      <br><br><i>W sytuacji, gdy niezbędne są proste rozwiązania</i>.
    </p>
  </div>
  <div class="clear"></div>
</div>
```

Tworzenie łańcucha metod i interfejsy „płynne”

W poprzednim przykładzie właściwie wykonano też jeden dodatkowy krok i połączono w jedną wszystkie trzy instrukcje kodu powiązane z polami. Instrukcja ma następującą postać:

```
$('.boxContainer .box')
  .append('<br /><br /><i>W sytuacji, gdy niezbędne są proste
rozwiązania</i>.')
  .parent()
  .addClass('boxsizer');
```

Ten wzorzec składni nosi nazwę **wzorca tworzenia łańcucha metod** i jest szczególnie zalecany przez twórców biblioteki jQuery oraz społeczność języka JavaScript. Tworzenie łańcucha metod jest częścią wzorca implementacji obiektowej interfejsów „płynnych”, w którego przypadku każda metoda przekazuje swój kontekst wykonywania kolejnej metodzie.

Większość metod biblioteki jQuery, które dotyczą obiektu jQuery, zwraca też ten sam lub nowy obiekt kolekcji elementów jQuery. Pozwala to utworzyć łańcuch złożony z kilku metod. Dzięki temu uzyskuje się kod nie tylko bardziej czytelny i wyrazisty, ale też z mniejszą liczbą niezbędnych deklaracji zmiennych.

Wzorzec Kompozyt

W przypadku wzorca Kompozyt kluczowym zagadnieniem jest możliwość traktowania kolekcji obiektów w taki sam sposób, w jaki traktujemy instancję pojedynczego obiektu. Zmodyfikowanie kompozycji za pomocą metody kolekcji spowoduje uwzględnienie modyfikacji dla każdej części kompozycji. Metody te mogą być pomyślnie stosowane, niezależnie od liczby elementów należących do kolekcji kompozytu, nawet wtedy, gdy kolekcja nie zawiera elementów.

Ponadto obiekty kolekcji kompozytu niekoniecznie muszą zapewniać dokładnie te same metody. Obiekt kompozytu może ujawniać tylko metody wspólne w przypadku obiektów kolekcji lub udostępniać interfejs API z abstrakcją i odpowiednio obsługiwać odróżnianie metod każdego obiektu.

Wyjaśnijmy dalej, w jaki sposób intuicyjny interfejs API zapewniany przez bibliotekę jQuery w dużym stopniu zależny jest od wzorca Kompozyt.

Sposób wykorzystania wzorca Kompozyt przez bibliotekę jQuery

Wzorzec Kompozyt stanowi integralną część architektury biblioteki jQuery i stosowany jest z poziomu samej, bardzo podstawowej funkcji `$()`. Każde wywołanie tej funkcji powoduje utworzenie i zwrócenie obiektu kolekcji elementów, który często jest po prostu nazywany obiektem jQuery. Właśnie w tym przypadku widoczna jest pierwsza zasada wzorca Kompozyt. Okazuje się, że zamiast pojedynczego elementu funkcja `$()` zwraca kolekcję elementów.

Zwracany obiekt jQuery to obiekt podobny do tablicy, który pełni rolę obiektu opakowującego i przechowuje kolekcję pobranych elementów. Obiekt ten zapewnia też kilka następujących dodatkowych właściwości:

- właściwość `length` określająca długość uzyskanej kolekcji elementów;
- właściwość `context` określająca kontekst, w jakim obiekt został utworzony;
- właściwość CSS `selector` określająca selektor użyty w wywołaniu funkcji `$()`;
- właściwość `prevObject` używana w sytuacji, gdy konieczne jest uzyskanie dostępu do poprzedniej kolekcji elementów po dodaniu wywołania metody do łańcucha metod.

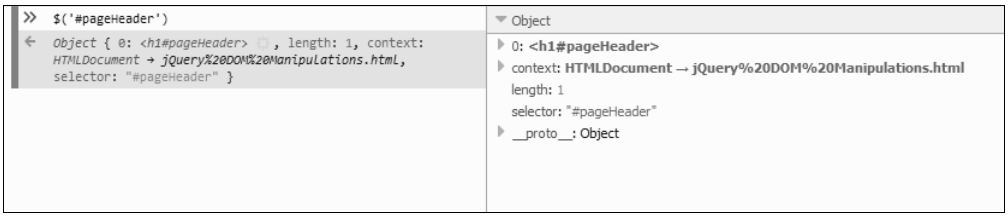
Korzystając z narzędzi programistycznych naszej ulubionej przeglądarki, z łatwością możemy poeksperymentować z obiektami jQuery zwracanymi przez funkcję `$()` i sprawdzić opisane wcześniej właściwości. Aby uruchomić te narzędzia w większości przeglądarek, niezbędne jest jedynie naciśnięcie klawisza *F12* w systemach Windows i Linux lub użycie kombinacji klawiszy *Cmd+Opt+I* w systemie Mac. Bezpośrednio po wykonaniu tej czynności można zastosować wywołania funkcji `$()` w konsoli i kliknąć zwrócone obiekty w celu sprawdzenia ich właściwości.

Prosta definicja obiektu przypominającego tablicę

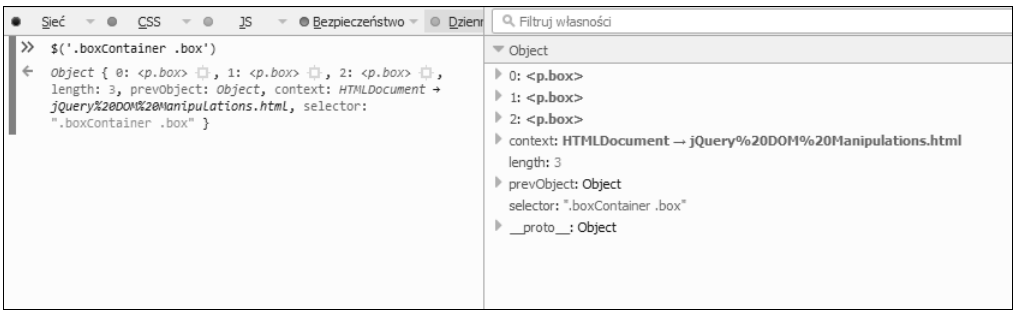
Obiekt podobny do tablicy to obiekt { } języka JavaScript z właściwością numeryczną `length` i odpowiednią liczbą właściwości o nazwach w postaci kolejnych liczb. Inaczej mówiąc, obiekt przypominający tablicę z właściwością `length` o wartości 2 (`length == 2`) powinien też mieć zdefiniowane dwie właściwości `"0"` i `"1"`. W wypadku tych właściwości obiekty podobne do tablicy umożliwiają uzyskanie dostępu do ich zawartości za pomocą prostych pętli `for` z wykorzystaniem składni akcesorów właściwości z nawiasami klamrowymi języka JavaScript:

```
for (var i = 0; i < obj.length; i++) {
  console.log(obj[i]);
}
```

Na poniższym rysunku widoczny jest wynik wywołania `$('#pageHeader')` użytego we wcześniejszym przykładzie w przypadku narzędzi programistycznych przeglądarki Firefox.



Wynik wywołania `$('.boxContainer .box')` został przedstawiony na kolejnym rysunku.



To, że w bibliotece jQuery obiekty przypominające tablicę pełnią rolę obiektów opakowujących zwracane elementy, pozwala tej bibliotece zapewnić dodatkowe metody, które są stosowane względem zwróconej kolekcji. Jest to osiągane dzięki dziedziczeniu prototypowemu obiektu `jQuery.fn`. W efekcie każdy obiekt jQuery ma też dostęp do wszystkich metod zapewnianych przez bibliotekę jQuery. W ten sposób tworzony jest cały wzorzec Kompozyt udostępniający metody, które po zastosowaniu dla kolekcji są odpowiednio uwzględniane dla każdego jej elementu składowego. Ponieważ biblioteka jQuery korzysta z obiektów przypominających tablicę z dziedziczeniem prototypowym, metody te z łatwością mogą zostać użyte jako właściwości w każdym obiekcie jQuery. Zostało to zaprezentowane w przykładzie na początku rozdziału:

`$('#pageHeader').css('font-size', '3em');`. Co więcej, biblioteka jQuery dodaje do swojego kodu modyfikującego model DOM kilka dodatkowych elementów, co ma na celu uzyskanie bardziej zwięzłego kodu, który jest mniej podatny na błędy. Gdy na przykład metoda `jQuery.fn.html()` używana jest do zmiany wewnętrznego kodu HTML węzła modelu DOM, który zawiera już elementy podrzędne, biblioteka jQuery próbuje najpierw usunąć wszelkie dane i procedury obsługi zdarzeń powiązane z tymi elementami, zanim usunie je ze strony i dołączy zapewniony kod HTML.

Przyjrzyjmy się temu, jak biblioteka jQuery implementuje takie metody stosowane względem kolekcji. W tym celu możemy pobrać i wyświetlić kod źródłowy ze strony witryny GitHub powiązanej z biblioteką jQuery (*https://github.com/jquery/jquery/releases*) lub skorzystać z narzędzia takiego jak jQuery Source Viewer, które jest dostępne pod adresem *http://james.padolsey.com/jquery*.

> Zależnie od używanej wersji uzyskane wyniki mogą się w pewnym stopniu różnić. Najnowsza stabilna wersja biblioteki jQuery, która była dostępna i wykorzystywana, gdy pisano tę książkę, to wersja 2.2.0.

Metoda `jQuery.fn.empty()` to jedna z najprostszych metod demonstrujących, jak implementowane są metody stosowane względem kolekcji. Implementację tej metody z łatwością możesz zlokalizować w kodzie źródłowym biblioteki jQuery, wyszukując łańcuch `empty:` lub używając narzędzia jQuery Source Viewer i szukając łańcucha `jQuery.fn.empty`. Skorzystanie z dowolnego z tych sposobów pozwoli uzyskać następujący kod:

```
empty: function() {
  var elem, i = 0;
  for ( ; ( elem = this[ i ] ) != null; i++ ) {
    if ( elem.nodeType === 1 ) {
      // Zapobiegnięcie „wyciekowi" pamięci
      jQuery.cleanData( getAll( elem, false ) );
      // Usunięcie wszystkich pozostałych węzłów
      elem.textContent = "";
      }
  }
  return this;
}
```

Jak widać, kod wcale nie jest złożony. Biblioteka jQuery dokonuje iteracji wszystkich elementów obiektu kolekcji (przywoływany w postaci słowa kluczowego `this`, ponieważ ma to miejsce wewnątrz implementacji metody) przy użyciu zwykłej pętli `for`. Dla każdego elementu kolekcji, czyli węzła elementu, kod jQuery czyści wartości wszystkich właściwości `data-*` za pomocą funkcji pomocniczej `jQuery.cleanData()`. Bezpośrednio po tej operacji kod usuwa zawartość elementów, ustawiając dla nich łańcuch pusty.

Więcej informacji o różnych określanych typach węzłów możesz znaleźć pod adresem *https://developer.mozilla.org/en-US/docs/Web/API/Node/nodeType*.

Porównanie korzyści uzyskiwanych w przypadku rezygnacji ze zwykłego interfejsu API modelu DOM

Aby wyraźnie zademonstrować korzyści zapewniane przez wzorzec Kompozyt, zmodyfikujemy przykład z początku rozdziału bez używania abstrakcji oferowanych przez bibliotekę jQuery. Używając tylko zwykłego kodu napisanego w JavaScripcie i interfejsu API modelu DOM, możemy utworzyć równorzędny kod o następującej postaci:

```
setTimeout(function() {
  var headerElement = document.getElementById('pageHeader');
  if (headerElement) {
    headerElement.style.fontSize = '3em';
  }
  var boxContainerElement =
      document.getElementsByClassName('boxContainer')[0];
  if (boxContainerElement) {
    var innerBoxElements =
        boxContainerElement.getElementsByClassName('box');
    for (var i = 0; i < innerBoxElements.length; i++) {
      var boxElement = innerBoxElements[i];
      boxElement.innerHTML +=
          '<br /><br /><i>W sytuacji, gdy niezbędne są proste rozwiązania</i>.';
      boxElement.parentNode.className += ' boxsizer';
    }
    var clearFloatDiv = document.createElement('div');
    clearFloatDiv.className = 'clear';
    boxContainerElement.appendChild(clearFloatDiv);
  }
}, 700);
```

I tym razem używana jest funkcja `setTimeout` z funkcją anonimową. Jako drugi parametr ustawiono czas `700` milisekund. Wewnątrz samej funkcji zastosowano metodę `document.getElementById` do pobrania elementów, w których przypadku wiadomo, że mają na stronie unikatowy identyfikator. W dalszej kolejności używana jest metoda `document.getElementsByClassName`, gdy niezbędne jest uzyskanie wszystkich elementów zawierających konkretną klasę. Zastosowano też metodę `boxContainerElement.getElementsByClassName('box')` do pobrania wszystkich elementów z klasą `box`, które są elementami podrzędnymi elementu z klasą `boxContainer`.

W tym wypadku najbardziej oczywistą obserwacją jest to, że w celu osiągnięcia identycznych wyników niezbędnych było 18 wierszy kodu. Dla porównania: w przypadku korzystania z biblioteki jQuery wymaganych było tylko 9 wierszy kodu, czyli połowa tego, co zostało użyte w powyższej implementacji. Zastosowanie funkcji `$()` biblioteki jQuery z selektorem CSS było prostszym sposobem uzyskania wymaganych elementów. Sposób ten zapewnia ponadto zgodność z przeglądarkami, które nie obsługują metody `getElementsByClassName()`. Istnieje jednak więcej korzyści niż tylko mniejsza liczba wierszy kodu i zwiększona czytelność. Implementując wzorzec Kompozyt, funkcja `$()` zawsze pobiera kolekcje elementów, dzięki czemu kod staje się bardziej jednolity

w porównaniu z różniącą się obsługą każdej z zastosowanych metod `getElement*`. Funkcja `$()` używana jest dokładnie w taki sam sposób, niezależnie od tego, czy pożądane jest uzyskanie jedynie elementu z unikatowym identyfikatorem, czy liczby elementów z konkretną klasą.

W ramach dodatkowej korzyści wynikającej ze zwracania obiektów przypominających tablicę biblioteka jQuery może też zapewniać wygodniejsze metody przechodzenia w obrębie modelu DOM i modyfikowania go. Są to na przykład metody `.css()`, `.append()` i `.parent()` zaprezentowane w pierwszym przykładzie, dostępne jako właściwości zwracanego obiektu. Ponadto biblioteka jQuery oferuje metody dokonujące abstrakcji bardziej złożonych przypadków użycia, takie jak `.addClass()` i `.wrap()`. W wypadku tych metod nie istnieją odpowiadające im metody, które są częścią interfejsu API modelu DOM.

Ponieważ zwracane obiekty kolekcji biblioteki jQuery nie różnią się niczym innym niż opakowywane przez nie elementy, w identyczny sposób możemy użyć dowolnej metody interfejsu API biblioteki jQuery. Jak wcześniej pokazano, metody te, niezależnie od liczby elementów, stosowane są dla każdego elementu pobranej kolekcji. W efekcie nie ma potrzeby używania osobnej pętli `for` w celu przeprowadzenia iteracji dla każdego pobranego elementu, a także wykonywania osobno operacji modyfikowania (np. przy użyciu metody `.addClass()`), które są stosowane bezpośrednio względem obiektu kolekcji.

Aby w zamieszczonym dalej przykładzie nadal zapewnić takie same gwarancje dotyczące bezpiecznego wykonania, konieczne jest również dodanie kilku dodatkowych instrukcji `if` sprawdzających wartości `null`. Jest to wymagane, ponieważ jeśli na przykład nie znaleziono zmiennej `headerElement`, wystąpi błąd, a reszta wierszy kodu nigdy nie zostanie wykonana. Ktoś mógłby uznać, że tego rodzaju sprawdzenia, takie jak `if (headerElement)` i `if (boxContainerElement)`, nie są wymagane w tym przykładzie i mogą zostać pominięte. W tym przypadku może się to wydawać poprawne, ale w rzeczywistości jest to jeden z głównych powodów pojawiania się błędów podczas projektowania aplikacji o dużej skali, w których nieustannie elementy są tworzone, wstawiane i usuwane z drzewa modelu DOM. Niestety, w wypadku wszystkich języków i platform docelowych programiści tworzą zwykle najpierw logikę implementacji, a dopiero później dodają takie sprawdzenia. Często ma to miejsce po pojawieniu się błędu podczas testowania implementacji.

Zgodnie z wzorcem Kompozyt nawet pusty obiekt kolekcji biblioteki jQuery (nie zawierający żadnych pobranych elementów) w dalszym ciągu jest poprawnym obiektem kolekcji, w którym można bezpiecznie zastosować dowolną metodę zapewnianą przez bibliotekę jQuery. W rezultacie nie ma potrzeby używania dodatkowych instrukcji `if` do sprawdzenia przed zastosowaniem metody takiej jak `.css()` (tylko po to, aby uniknąć błędu środowiska wykonawczego kodu JavaScript), czy kolekcja faktycznie zawiera jakikolwiek element.

Ogólnie rzecz biorąc, abstrakcje oferowane przez bibliotekę jQuery z wykorzystaniem wzorca Kompozyt zapewniają mniejszą liczbę wierszy kodu, który jest bardziej czytelny i jednolity, a ponadto zawiera mniej wierszy podatnych na literówki (porównaj wpisywanie instrukcji `$('#elementID')` z instrukcją `document.getElementById('elementID')`).

Użycie wzorca Kompozyt do projektowania aplikacji

Gdy już wyjaśniono, jak biblioteka jQuery korzysta z wzorca Kompozyt w swojej architekturze, a także porównano korzyści zapewniane przez wzorzec, spróbujmy samodzielnie utworzyć przykładowy przypadek użycia. Postaramy się uwzględnić wszystkie zagadnienia zaprezentowane wcześniej w tym rozdziale. Kompozyt będzie mieć strukturę obiektu przypominającego tablicę. Ponadto wzorzec będzie przetwarzać obiekty o całkowicie innej strukturze, zapewniać „płynny" interfejs API umożliwiający tworzenie łańcucha metod oraz udostępniać metody, które stosowane są dla wszystkich elementów kolekcji.

Przykładowy przypadek użycia

Załóżmy, że istnieje aplikacja, która w pewnym momencie wymaga wykonania operacji na liczbach. Z kolei elementy wymagane przez aplikację do przetwarzania pochodzą z różnych źródeł i zupełnie nie są jednolite. Aby przykład był interesujący, przyjmijmy, że jedno źródło danych zapewnia zwykłe liczby, a inne — obiekty z określoną właściwością, która zawiera interesującą nas liczbę.

```
var numberValues = [2, 5, 8];
var objectsWithValues = [
    { value: 7 },
    { value: 4 },
    { value: 6 },
    { value: 9 }
];
```

Obiekty zwracane przez drugie źródło występujące w przykładowym przypadku użycia mogą mieć bardziej złożoną strukturę i prawdopodobnie kilka dodatkowych właściwości. Takie zmiany w żaden sposób nie spowodują zróżnicowania przykładowej implementacji, gdyż podczas tworzenia wzorca Kompozyt interesuje nas jedynie zapewnienie jednolitej obsługi wspólnych części elementów docelowych.

Implementacja kolekcji kompozytu

Zdefiniujmy funkcję konstruktora i prototyp opisujące przykładowy obiekt kolekcji kompozytu:

```
function ValuesComposite() {
    this.length = 0;
}
ValuesComposite.prototype.append = function(item) {
    if ((typeof item === 'object' && 'value' in item) ||
        typeof item === 'number') {
        this[this.length] = item;
        this.length++;
    }
    return this;
};
```

```
ValuesComposite.prototype.increment = function(number) {
    for (var i = 0; i < this.length; i++) {
        var item = this[i];
    if (typeof item === 'object' && 'value' in item) {
            item.value += number;
        } else if (typeof item === 'number') {
            this[i] += number;
        }
    }
    return this;
};
ValuesComposite.prototype.getValues = function() {
    var result = [];
    for (var i = 0; i < this.length; i++) {
        var item = this[i];
        if (typeof item === 'object' && 'value' in item) {
            result.push(item.value);
        } else if (typeof item === 'number') {
            result.push(item);
        }
    }
    return result;
};
```

Funkcja konstruktora `ValuesComposite()` w przykładzie jest dość prosta. W razie wywołania za pomocą operatora `new` zwraca ona pusty obiekt z właściwością `length` równą zeru, która wskazuje, że kolekcja opakowywana przez funkcję jest pusta.

> Więcej informacji o opartym na prototypie modelu programowania języka JavaScript dostępnych jest na stronie internetowej pod adresem *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript*.

Konieczne jest najpierw zdefiniowanie sposobu umożliwiającego wypełnienie obiektów kolekcji kompozytu. Zdefiniowano metodę `append`, która sprawdza, czy zapewniony parametr jest jednego z obsługiwanych przez nią typów. Metoda dołącza parametr w obiekcie kompozytu w wypadku następnej dostępnej właściwości numerycznej i inkrementuje wartość właściwości `length`. Na przykład pierwszy dołączony element, niezależnie od tego, czy jest obiektem z właściwością w postaci wartości czy zwykłej liczby, zostanie udostępniony właściwości `"0"` obiektu kompozytu i będzie dostępny jako `myValuesComposition[0]`w ramach składni akcesorów właściwości z nawiasami klamrowymi.

Metoda `increment` jest prezentowana jako prosta, przykładowa metoda, która może modyfikować takie kolekcje przez przetwarzanie wszystkich ich elementów. Jako parametr metoda akceptuje wartość liczbową, a następnie odpowiednio ją obsługuje, dodając ją do każdego elementu kolekcji (zależnie od ich typu). Ponieważ przykładowy kompozyt to obiekt podobny do tablicy,

metoda `increment` korzysta z pętli `for`, aby dokonać iteracji wszystkich elementów kolekcji, i zwiększa albo wartość `item.value` (w sytuacji, gdy element jest obiektem), albo faktyczną przechowywaną wartość liczbową (gdy zapisany element kolekcji to liczba). W ten sam sposób możemy kontynuować działania i implementować inne metody, które na przykład umożliwią pomnożenie elementów kolekcji przez konkretną liczbę.

Aby umożliwić utworzenie łańcucha metod przykładowego obiektu kompozytu, wszystkie metody prototypu muszą zwracać odwołanie do instancji obiektu. Cel ten osiągamy po prostu przez dodanie instrukcji `return this;` jako ostatniego wiersza kodu w wypadku wszystkich metod modyfikujących kolekcję, takich jak `append` i `increment`. Miej świadomość tego, że takie metody jak `getValues`, które nie modyfikują kolekcji, lecz służą do zwrócenia wyniku, z założenia nie mogą być dodawane do łańcucha metod, aby przekazać instancję obiektu kolekcji kolejnym wywołaniom metody.

I wreszcie metoda `getValues` implementowana jest jako wygodny sposób uzyskiwania rzeczywistych wartości liczbowych wszystkich elementów kolekcji. Podobnie do metody `increment` metoda `getValues` dokonuje abstrakcji obsługi dotyczącej różnych typów elementów kolekcji. Metoda przeprowadza iterację elementów kolekcji, wyodrębnia każdą wartość liczbową i dołącza ją do tablicy `result`, która jest zwracana elementowi wywołującemu metodę.

Przykład wykonania kodu

Pora zaznajomić się z faktycznym przykładem korzystającym z właśnie zaimplementowanego obiektu kompozytu:

```
var valuesComposition = new ValuesComposite();
for (var i = 0; i < numberValues.length; i++) {
    valuesComposition.append(numberValues[i]);
}
for (var i = 0; i < objectsWithValues.length; i++) {
    valuesComposition.append(objectsWithValues[i]);
}
valuesComposition.increment(2)
    .append(1)
    .append(2)
    .append({ value: 3 });
console.log(valuesComposition.getValues());
```

Wykonanie w przeglądarce powyższego kodu po umieszczeniu go w obrębie istniejącej strony lub bezpośrednio w konsoli przeglądarki zarejestruje ona następujący wynik:

```
▸ Array [ 4, 7, 10, 9, 6, 8, 11, 1, 2, 3 ].
```

Korzystamy ze źródeł danych, takich jak zmienne `numberValues` i `objectsWithValues`, które zaprezentowano wcześniej. W powyższym kodzie iterowane są obie zmienne, a ich elementy są dołączane do nowo utworzonej instancji obiektu kompozytu. Dalej inkrementowane są o 2 wartości kolekcji kompozytu. Bezpośrednio po tej operacji tworzony jest łańcuch trzech metod

`append` wstawiających element. Pierwsze dwie metody dołączają wartości liczbowe, a trzecia dodaje obiekt z właściwością w postaci wartości. Aby uzyskać tablicę z wszystkimi wartościami liczbowymi kolekcji i zarejestrować ją w konsoli przeglądarki, na końcu używana jest metoda `getValues`.

Implementacje alternatywne

Miej świadomość tego, że kompozyt nie musi być obiektem przypominającym tablicę, ale jest to ogólnie preferowane, ponieważ JavaScript ułatwia utworzenie takiej implementacji. Ponadto implementacje podobne do tablicy zapewniają też korzyść w postaci umożliwiania iteracji elementów kolekcji przy użyciu prostej pętli `for`.

Z kolei gdy obiekt przypominający tablicę nie jest preferowany, z łatwością można skorzystać z właściwości w obiekcie kompozytu, aby przechowywać elementy kolekcji. Taka właściwość może na przykład mieć nazwę `items` i służyć do przechowywania (za pomocą instrukcji `this.items.push(item)`) oraz udostępniania (przy użyciu instrukcji `this.items[i]`) elementów kolekcji w obrębie metod.

Wzorzec Iterator

Kluczowym pojęciem związanym z wzorcem Iterator jest zastosowanie funkcji jednej odpowiedzialności w celu przechodzenia w obrębie kolekcji i zapewniania dostępu do jej elementów. Funkcja ta jest znana jako iterator i umożliwia uzyskanie dostępu do elementów kolekcji bez ujawniania szczegółów implementacji oraz bazowej struktury danych używanej przez obiekt kolekcji.

Iteratory zapewniają poziom hermetyzacji dotyczący sposobu przeprowadzania iteracji elementów kolekcji, oddzielając ten proces od implementacji logiki konsumentów tych elementów.

> Więcej informacji o zasadzie jednej odpowiedzialności znajdziesz pod adresem *http://www.oodesign.com/single-responsibility-principle.html*.

Sposób wykorzystania wzorca Iterator przez bibliotekę jQuery

Jak wspomniano wcześniej w tym rozdziale, podstawowa funkcja `$()` biblioteki jQuery zwraca obiekt przypominający tablicę, który opakowuje kolekcję elementów strony, a także zapewnia funkcję iteratora umożliwiającą przechodzenie w obrębie kolekcji i uzyskiwanie dostępu do poszczególnych elementów. W przypadku biblioteki jQuery wykonano właściwie jeden dodatkowy krok, udostępniając ogólną funkcję pomocniczą `jQuery.each()`, która może dokonywać iteracji tablic, obiektów podobnych do tablicy oraz właściwości obiektu.

Bardziej techniczny opis dostępny jest na stronie dokumentacji interfejsu API biblioteki jQuery pod adresem *http://api.jquery.com/jQuery.each/*, gdzie zamieszczono następującą informację dotyczącą metody `jQuery.each()`:

> *Ogólna funkcja iteratora, która może posłużyć do ciągłej iteracji zarówno obiektów, jak i tablic. Tablice i obiekty przypominające tablicę z właściwością długości (np. obiekt argumentu funkcji) są iterowane przy użyciu indeksu liczbowego, od zera do wartości długość-1. Inne obiekty są iterowane z wykorzystaniem ich właściwości z nazwami.*

Funkcja pomocnicza `jQuery.each()` używana jest wewnętrznie w kilku miejscach kodu źródłowego biblioteki jQuery. Jednym z zastosowań tej funkcji jest iteracja elementów obiektu jQuery i, zgodnie z tym, co sugeruje wzorzec Kompozyt, wykonywanie operacji modyfikowania dla każdego z nich. Proste wyszukiwanie słowa kluczowego `.each(` zapewnia 56 dopasowań.

> Gdy pisano książkę, najnowszą stabilną wersją biblioteki była wersja 2.2.0, której użyto do uzyskania powyższych statystyk.

Z łatwością można prześledzić implementację funkcji w kodzie źródłowym biblioteki jQuery, wyszukując łańcuch `each:` (zauważ, że są dwa wystąpienia) lub używając narzędzia jQuery Source Viewer i szukając łańcucha `jQuery.each()` (jak to miało miejsce wcześniej w tym rozdziale):

```
each: function( obj, callback ) {
  var length, i = 0;
  if ( isArrayLike( obj ) ) {
    length = obj.length;
    for ( ; i < length; i++ ) {
      if ( callback.call( obj[ i ], i, obj[ i ] ) === false ) {
        break;
      }
    }
  } else {
    for ( i in obj ) {
      if ( callback.call( obj[ i ], i, obj[ i ] ) === false ) {
        break;
      }
    }
   }
  return obj;
}
```

Ta funkcja pomocnicza jest też dostępna w dowolnym obiekcie jQuery dzięki użyciu tego samego dziedziczenia prototypowego, które zaprezentowano wcześniej w przypadku metod takich jak `.append()`. Z łatwością możesz znaleźć kod, który realizuje dokładnie coś takiego. W tym celu za pomocą narzędzia jQuery Source Viewer wyszukaj łańcuch `jQuery.fn.each()` lub w kodzie źródłowym biblioteki jQuery bezpośrednio poszukaj łańcucha `each:` (zauważ, że są dwa wystąpienia):

```
each: function( callback ) {
  return jQuery.each( this, callback );
}
```

Zastosowanie wariantu z wyszukiwaniem łańcucha `.each()` pozwala na bezpośrednią iterację elementów obiektu kolekcji biblioteki jQuery z wykorzystaniem wygodniejszej w użyciu składni.

Widoczny poniżej przykładowy kod prezentuje, jak w kodzie mogą zostać użyte dwa warianty funkcji `.each()`:

```
// Użycie funkcji pomocniczej dla tablicy
$.each([3, 5, 7], function(index){
    console.log(this + 1);
});
// Użycie metody dla obiektu jQuery
$('.boxContainer .box').each(function(index) {
    console.log('To jest pole o numerze ' + (index + 1)); // Indeks rozpoczyna się
                                                          // od zera
});
```

Po wykonaniu powyższy kod spowoduje zarejestrowanie w konsoli przeglądarki wyniku widocznego na poniższym rysunku.

```
4
6
8
To jest pole o numerze 1
To jest pole o numerze 2
To jest pole o numerze 3
```

Powiązanie wzorca Iterator z wzorcem Kompozyt

Ponieważ wzorzec Kompozyt hermetyzuje kolekcję elementów w postaci jednego obiektu, a wzorzec Iterator może zostać użyty do iteracji struktury danych z abstrakcją, z łatwością możemy te dwa wzorce opisać jako uzupełniające się.

Zastosowanie wzorca Iterator

Wzorzec Iterator może być stosowany w tworzonych aplikacjach w celu definiowania abstrakcji sposobu uzyskiwania dostępu do elementów struktury danych. Dla przykładu załóżmy, że konieczne jest uzyskanie z następującej struktury drzewa wszystkich elementów o wartości większej niż 4:

```
var collection = {
    nodeValue: 7,
    left: {
        nodeValue: 4,
        left: 2,
        right: {
            nodeValue: 6,
            left: 5,
            right: 9
        }
    },
    right: {
        nodeValue: 9,
        left: 8
    }
};
```

Zaimplementujmy teraz funkcję iteratora. Ponieważ drzewiaste struktury danych mogą zawierać zagnieżdżanie, uzyskujemy następującą implementację rekurencyjną:

```
function iterateTreeValues(node, callback) {
    if (node === null || node === undefined) {
        return;
    }
    if (typeof node === 'object') {
        if ('left' in node) {
            iterateTreeValues(node.left, callback);
        }
        if ('nodeValue' in node) {
            callback(node.nodeValue);
        }
        if ('right' in node) {
            iterateTreeValues(node.right, callback);
        }
    } else {
        // Ponieważ jest to liść, węzeł jest wartością
        callback(node);
    }
}
```

Ostatecznie otrzymujemy następującą implementację:

```
var valuesArray = [];
iterateTreeValues(collection, function(value) {
    if (value > 4) {
        valuesArray.push(value);
    }
});
console.log(valuesArray);
```

Po wykonaniu powyższego kodu w konsoli przeglądarki zostanie zarejestrowany następujący wynik:

- `Array [ 5, 6, 9, 7, 8, 9 ]`.

Wyraźnie widać, że iterator uprościł kod. Nie ma potrzeby zajmowania się już szczegółami implementacji struktury danych używanej każdorazowo, gdy konieczne jest uzyskanie dostępu do wybranych elementów spełniających określone kryteria. Implementacja bazuje na ogólnym interfejsie API udostępnianym przez iterator. Z kolei logika implementacji pojawia się w wywołaniu zwrotnym zapewnianym iteratorowi.

Taka hermetyzacja umożliwia odłączenie implementacji od używanej struktury danych, przy założeniu, że będzie dostępny iterator z takim samym interfejsem API. W omawianym przykładzie z łatwością można zmienić stosowaną strukturę danych na sortowane drzewo binarne lub prostą tablicę, a ponadto zachować bez zmian logikę implementacji.

Podsumowanie

W tym rozdziale utrwaliliśmy wiedzę dotyczącą biblioteki jQuery i skryptowego interfejsu API modelu DOM języka JavaScript. Zaprezentowałem wzorzec Kompozyt i pokazałem, jak jest on wykorzystywany przez bibliotekę jQuery. Wyjaśniłem, w jaki sposób wzorzec Kompozyt upraszcza przepływ pracy po przebudowaniu przykładowej strony bez użycia kodu jQuery. Dalej przedstawiłem przykład zastosowania wzorca Kompozyt w aplikacjach. I wreszcie dokonałem wprowadzenia do wzorca Iterator, a także pokazałem, jak dobrze może współdziałać z wzorcem Kompozyt.

Po ukończeniu niniejszego wprowadzenia na temat tego, jak ważną rolę odgrywa wzorzec Kompozyt w sposobie codziennego użytkowania metod biblioteki jQuery, możemy przejść do następnego rozdziału. Zaprezentujemy w nim wzorzec Obserwator oraz wygodny sposób wykorzystania go w obrębie stron bazujących na kodzie jQuery.

Skorowidz

I

J

K

L

Ł

M

N

O

P

R

S

T

U

W

Z

Biblioteka jQuery
Sprawdzone wzorce projektowe

jQuery jest lekką, wszechstronną i rozszerzalną biblioteką języka JavaScript o ogromnych możliwościach. Zapewnia prosty w użyciu API, który jest bezproblemowo obsługiwany przez różne przeglądarki i znacząco ułatwia wykonywanie modyfikacji i operacji przechodzenia w obrębie modelu DOM dokumentów HTML, obsługę zdarzeń, uruchamianie animacji, a także korzystanie z technologii Ajax. Pozwala to na coraz bardziej złożone interakcje z użytkownikiem oraz na obsługę dużych implementacji aplikacji.

Niniejsza książka jest przeznaczona dla projektantów, którzy pracując z biblioteką jQuery, chcą wykorzystywać różne standardowe wzorce branżowe. Docenią ją zespoły programistów, którym ułatwi pisanie dobrze zorganizowanych i rozszerzalnych implementacji. Zaprezentowano tu różne wzorce projektowe, takie jak Fasada, Obserwator czy wzorce optymalizacji. Omówiono również techniki i biblioteki związane z tworzeniem szablonów klienckich, a także przedstawiono kilka wzorców projektowania dodatków. Nie zabrakło opisu sprawdzonych procedur i wskazówek dotyczących wydajności, dzięki którym maksymalnie wykorzystasz możliwości biblioteki jQuery!

jQuery — maksymalna efektywność wdrożeń o dużej skali!

W książce między innymi:

- przypomnienie najważniejszych informacji o jQuery
- struktura aplikacji: podział na niezależne moduły i rozdzielanie kodu
- emitowanie i odbieranie zdarzeń
- tworzenie abstrakcji złożonych interfejsów API
- interfejsy API Deferred i Promises
- tworzenie szablonów klienckich

Thodoris Greasidis — pochodzi z Grecji. Jest projektantem, implementuje aplikacje internetowe o dużej skali z intuicyjnymi interfejsami oraz usługi sieciowe o dużej dostępności. Jest jednym z twórców biblioteki AngularUI. Brał udział w wielu projektach *open source*, zwłaszcza dotyczących Mozilli. Jest entuzjastą języka JavaScript, a jego pasją są operacje bitowe.

[PACKT] open source PUBLISHING community experience distilled

Helion

32960 numer katalogowy

księgarnia internetowa
http://helion.pl

zamówienia telefoniczne
0 801 339900
0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-283-2832-7

cena: 44,90 zł

Informatyka w najlepszym wydaniu