Twórz lepszy kod JavaScript!

JavaScript Wzorce

HELION

O'REILLY® | YAHOO! PRESS

Stoyan Stefanov

Tytuł oryginału: JavaScript Patterns

Tłumaczenie: Rafał Jońca

ISBN: 978-83-246-3821-5

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)

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

Printed in Poland.

Spis treści

ROZDZIAŁ 7.

Wzorce projektowe

Wzorce projektowe opisane w książce tak zwanego gangu czworga oferują rozwiązania typowych problemów związanych z projektowaniem oprogramowania zorientowanego obiektowo. Są dostępne już od jakiegoś czasu i sprawdziły się w wielu różnych sytuacjach, warto więc się z nimi zapoznać i poświęcić im nieco czasu.

Choć same te wzorce projektowe nie są uzależnione od języka programowania i implementacji, były analizowane przez wiele lat głównie z perspektywy języków o silnym sprawdzaniu typów i statycznych (niezmiennych) klasach takich jak Java lub C++.

JavaScript jest językiem o luźnej kontroli typów i bazuje na prototypach (a nie klasach), więc niektóre z tych wzorców okazują się wyjątkowo proste, a czasem wręcz banalne w implementacji.

Zacznijmy od przykładu sytuacji, w której w języku JavaScript rozwiązanie wygląda inaczej niż w przypadku języków statycznych bazujących na klasach, czyli od wzorca singletonu.

Singleton

Wzorzec singletonu ma w założeniu zapewnić tylko jedną instancję danej klasy. Oznacza to, że próba utworzenia obiektu danej klasy po raz drugi powinna zwrócić dokładnie ten sam obiekt, który został zwrócony za pierwszym razem.

Jak zastosować ten wzorzec w języku JavaScript? Nie mamy przecież klas, a jedynie obiekty. Gdy powstaje nowy obiekt, nie ma w zasadzie drugiego identycznego, więc jest on automatycznie singletonem. Utworzenie prostego obiektu za pomocą literału to doskonały przykład utworzenia singletonu.

```
var obj = {
    myprop: 'wartość'
};
```

W JavaScripcie obiekty nie są sobie równe, jeśli nie są dokładnie tym samym obiektem, więc nawet jeśli utworzy się dwa identyczne obiekty z takimi samymi wartościami, nie będą równoważne.

```
var obj2 = {
    myprop: 'wartość'
};
obj === obj2; //false
obj == obj2; //false
```

Można więc stwierdzić, że za każdym razem, gdy powstaje nowy obiekt tworzony za pomocą literału, powstaje nowy singleton, i to bez użycia dodatkowej składni.

Czasem gdy ludzie mówią „singleton" w kontekście języka JavaScript, mają na myśli wzorzec modułu opisany w rozdziale 5.

Użycie słowa kluczowego new

JavaScript jest językiem niestosującym klas, więc dosłowna definicja singletonu nie ma tu zastosowania. Z drugiej strony język posiada słowo kluczowe `new`, które tworzy obiekty na podstawie funkcji konstruujących. Czasem tworzenie ich w ten sposób jako singletonów może być ciekawym podejściem. Ogólny pomysł jest następujący: kilkukrotne wywołanie funkcji konstruującej z użyciem `new` powinno spowodować każdorazowo zwrócenie dokładnie tego samego obiektu.

Przedstawiony poniżej opis nie jest użyteczny w praktyce. Stanowi raczej teoretyczne wyjaśnienie powodów powstania wzorca w językach statycznych o ścisłej kontroli typów, w których to funkcje nie są pełnoprawnymi obiektami.

Poniższy przykład ilustruje oczekiwane zachowanie (pod warunkiem że nie wierzy się w światy równoległe i akceptuje się tylko jeden).

```
var uni = new Universe();
var uni2 = new Universe();
uni === uni2; // true
```

W tym przykładzie `uni` tworzone jest tylko przy pierwszym wywołaniu konstruktora. Drugie i kolejne wywołania zwracają ten sam obiekt. Dzięki temu `uni === uni2` (to dokładnie ten sam obiekt). Jak osiągnąć taki efekt w języku JavaScript?

Konstruktor `Universe` musi zapamiętać instancję obiektu (`this`), gdy zostanie utworzona po raz pierwszy, a następnie zwracać ją przy kolejnych wywołaniach. Istnieje kilka sposobów, by to uzyskać.

- Wykorzystanie zmiennej globalnej do zapamiętania instancji. Nie jest to zalecane podejście, bo zmienne globalne należy tworzyć tylko wtedy, gdy jest to naprawdę niezbędne. Co więcej, każdy może nadpisać taką zmienną, także przez przypadek. Na tym zakończmy rozważania dotyczące tej wersji.
- Wykorzystanie właściwości statycznej konstruktora. Funkcje w języku JavaScript są obiektami, więc mają właściwości. Można by utworzyć właściwość `Universe.instance` i to w niej przechowywać obiekt. To eleganckie rozwiązanie, ale ma jedną wadę: właściwość `instance` byłaby dostępna publicznie i inny kod mógłby ją zmienić.
- Zamknięcie instancji w domknięciu. W ten sposób instancja staje się elementem prywatnym i nie może zostać zmieniona z zewnątrz. Ceną tego rozwiązania jest dodatkowe domknięcie.

Przyjrzyjmy się przykładowym implementacjom drugiej i trzeciej opcji.

Instancja we właściwości statycznej

Poniższy kod zapamiętuje pojedynczą instancję we właściwości statycznej konstruktora `Universe`.

```
function Universe() {

    // czy istnieje już instancja?
    if (typeof Universe.instance === "object") {
        return Universe.instance;
    }

    // standardowe działania
    this.start_time = 0;
    this.bang = "Wielki";

    // zapamiętanie instancji
    Universe.instance = this;

    // niejawna instrukcja return:
    // return this;
}

// test
var uni = new Universe();
var uni2 = new Universe();
uni === uni2; // true
```

To bardzo proste rozwiązanie z jedną wadą, którą jest publiczne udostępnienie `instance`. Choć prawdopodobieństwo zmiany takiej właściwości przez kod jest niewielkie (i na pewno znacząco mniejsze niż w przypadku zmiennej globalnej), to jednak jest to możliwe.

Instancja w domknięciu

Innym sposobem uzyskania singletonu podobnego do rozwiązań klasowych jest użycie domknięcia w celu ochrony instancji. W implementacji można wykorzystać wzorzec prywatnej składowej statycznej omówiony w rozdziale 5. Tajnym składnikiem jest nadpisanie konstruktora.

```
function Universe() {

    // zapamiętanie instancji
    var instance = this;

    // standardowe działania
    this.start_time = 0;
    this.bang = "Wielki";

    // nadpisanie konstruktora
    Universe = function () {
        return instance;
    };
}

// testy
var uni = new Universe();
var uni2 = new Universe();
uni === uni2; // true
```

Za pierwszym razem zostaje wywołany oryginalny konstruktor, który zwraca `this` w sposób standardowy. Drugie i następne wywołania wykonują już zmieniony konstruktor, który ma dostęp do zmiennej prywatnej `instance` dzięki domknięciu i po prostu ją zwraca.

Przedstawiona implementacja jest w zasadzie przykładem wzorca samomodyfikującej się funkcji z rozdziału 4. Wadą tego rozwiązania opisaną we wspomnianym rozdziale jest to, że nadpisana funkcja (w tym przypadku konstruktor `Universe()`) utraci wszystkie właściwości dodane między jej zdefiniowaniem i nadpisaniem. W tej konkretnej sytuacji nic z tego, co zostanie dodane do prototypu `Universe()` po pierwszym obiekcie, nie będzie mogło posiadać referencji do instancji utworzonej przez oryginalną implementację.

Dla uwidocznienia problemu wykonajmy krótki test. Najpierw kilka wierszy przygotowujących:

```
// dodanie właściwości do prototypu
Universe.prototype.nothing = true;

var uni = new Universe();

// ponowne dodanie właściwości do prototypu
// po utworzeniu pierwszego obiektu
Universe.prototype.everything = true;

var uni2 = new Universe();
```

Oto właściwy test:

```
// tylko oryginalny prototyp jest powiązany z obiektami
uni.nothing; // true
uni2.nothing; // true
uni.everything; // undefined
uni2.everything; // undefined

// wygląda prawidłowo:
uni.constructor.name; // "Universe"

// ale to jest dziwne:
uni.constructor === Universe; // false
```

Powodem, dla którego właściwość `uni.constructor` nie jest już taka sama jak konstruktor `Universe()`, jest fakt, iż `uni.constructor` nadal wskazuje na oryginalny konstruktor zamiast przedefiniowanego.

Jeśli prototyp i referencja wskazująca na konstruktor muszą działać prawidłowo, do wcześniejszej implementacji trzeba wprowadzić kilka poprawek.

```
function Universe() {

    // zapamiętanie instancji
    var instance;

    // nadpisanie konstruktora
    Universe = function Universe() {
        return instance;
    };

    // przeniesienie właściwości prototypu
    Universe.prototype = this;

    // instancja
    instance = new Universe();

    // zmiana referencji wskazującej na konstruktor
    instance.constructor = Universe;
```

```
        // właściwa funkcjonalność
        instance.start_time = 0;
        instance.bang = "Wielki";

        return instance;
    }
```

Teraz wszystkie testy powinny działać zgodnie z oczekiwaniami.

```
// aktualizacja prototypu i utworzenie instancji
Universe.prototype.nothing = true; // true
var uni = new Universe();
Universe.prototype.everything = true; // true
var uni2 = new Universe();

// to ta sama pojedyncza instancja
uni === uni2; // true

// wszystkie właściwości prototypu działają prawidłowo
// niezależnie od momentu ich zdefiniowania
uni.nothing && uni.everything && uni2.nothing && uni2.everything; // true
// standardowe właściwości również działają prawidłowo
uni.bang; // "Wielki"
// referencja wskazująca na konstruktor również jest prawidłowa
uni.constructor === Universe; // true
```

Alternatywne rozwiązanie mogłoby polegać na otoczeniu konstruktora oraz instancji funkcją natychmiastową. Pierwsze wywołanie konstruktora tworzy obiekt i zapamiętuje go w prywatnej zmiennej `instance`. Drugie i kolejne wywołania jedynie zwracają zawartość zmiennej. Wszystkie poprzednie testy będą działały również dla implementacji przedstawionej poniżej.

```
var Universe;

(function () {

    var instance;

    Universe = function Universe() {

        if (instance) {
            return instance;
        }

        instance = this;

        // właściwa funkcjonalność
        this.start_time = 0;
        this.bang = "Wielki";

    };

}());
```

Fabryka

Celem wzorca fabryki jest tworzenie obiektów. Najczęściej fabryką jest klasa lub metoda statyczna klasy, której celem jest:

- wykonanie powtarzających się operacji przy tworzeniu podobnych obiektów;
- zapewnienie użytkownikom możliwości tworzenia obiektów bez potrzeby znania konkretnego typu (klasy) na etapie kompilacji.

Drugi punkt ma większe znaczenie w przypadku języków ze statyczną analizą typów, w których to utworzenie instancji klas nieznanych na etapie kompilacji nie jest zadaniem łatwym. Na szczęście w języku JavaScript nie trzeba głowić się nad tym zagadnieniem.

Obiekty tworzone przez metodę fabryczną z reguły dziedziczą po tym samym przodku, ale z drugiej strony są wyspecjalizowanymi wersjami z pewnymi dodatkowymi rozwiązaniami. Czasem wspólny przodek to klasa zawierająca metodę fabryczną.

Przyjrzyjmy się przykładowej implementacji, która ma:

- wspólny konstruktor przodka `CarMaker`;
- metodę statyczną `CarMaker` o nazwie `factory()`, która tworzy obiekty samochodów;
- wyspecjalizowane konstruktory `CarMaker.Compact`, `CarMaker.SUV` i `CarMaker.Convertible`, które dziedziczą po `CarMaker` i wszystkie są statycznymi właściwościami przodka, dzięki czemu globalna przestrzeń nazw pozostaje czysta i łatwo je w razie potrzeby odnaleźć.

Implementacja będzie mogła być wykorzystywana w następujący sposób:

```
var corolla = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');
corolla.drive(); // "Brum, mam 4 drzwi"
solstice.drive(); // "Brum, mam 2 drzwi"
cherokee.drive(); // "Brum, mam 17 drzwi"
```

Fragment

```
var corolla = CarMaker.factory('Compact');
```

to prawdopodobnie najbardziej rozpoznawalna część wzorca fabryki. Metoda przyjmuje typ jako tekst i na jego podstawie tworzy i zwraca obiekty danego typu. Nie pojawiają się konstruktory wykorzystujące `new` lub literały obiektów — użytkownik stosuje funkcję, która tworzy obiekty na podstawie typu wskazanego jako tekst.

Oto przykładowa implementacja wzorca fabryki, która odpowiada wcześniejszemu przykładowi jego użycia:

```
// konstruktor przodka
function CarMaker() {}

// metoda przodka
CarMaker.prototype.drive = function () {
    return "Brum, mam " + this.doors + " drzwi";
};

// statyczna metoda fabryczna
CarMaker.factory = function (type) {
    var constr = type,
        newcar;

    // błąd, jeśli konstruktor nie istnieje
    if (typeof CarMaker[constr] !== "function") {
        throw {
            name: "Error",
            message: constr + " nie istnieje"
        };
    }

    // na tym etapie wiemy, że konstruktor istnieje
    // niech odziedziczy przodka, ale tylko raz
```

```
        if (typeof CarMaker[constr].prototype.drive !== "function") {
            CarMaker[constr].prototype = new CarMaker();
        }
        // utworzenie nowej instancji
        newcar = new CarMaker[constr]();
        // opcjonalne wywołanie dodatkowych metod i zwrócenie obiektu...
        return newcar;
    };

    // definicje konkretnych konstruktorów
    CarMaker.Compact = function () {
        this.doors = 4;
    };
    CarMaker.Convertible = function () {
        this.doors = 2;
    };
    CarMaker.SUV = function () {
        this.doors = 17;
    };
```

W implementacji wzorca fabryki nie ma nic szczególnego. Wystarczy wyszukać odpowiednią funkcję konstruującą, która utworzy obiekt wymaganego typu. W tym przypadku zastosowano bardzo proste odwzorowanie nazw przekazywanych do fabryki na odpowiadające im obiekty. Przykładem powtarzających się zadań, które warto byłoby umieścić w fabryce, zamiast powtarzać osobno dla każdego konstruktora, jest dziedziczenie.

Wbudowane fabryki obiektów

W zasadzie język JavaScript posiada wbudowaną fabrykę, którą jest globalny konstruktor `Object()`. Zachowuje się on jak fabryka, ponieważ zwraca różne typy obiektów w zależności od parametru wejściowego. Przekazanie liczby spowoduje utworzenie obiektu konstruktorem `Number()`. Podobnie dzieje się dla tekstów i wartości logicznych. Wszystkie inne wartości lub brak argumentu spowodują utworzenie zwykłego obiektu.

Oto kilka przykładów i testów tego sposobu działania. Co więcej, `Object` można również wywołać z takim samym efektem bez użycia `new`.

```
    var o = new Object(),
        n = new Object(1),
        s = Object('1'),
        b = Object(true);

    // testy
    o.constructor === Object; // true
    n.constructor === Number; // true
    s.constructor === String; // true
    b.constructor === Boolean; // true
```

To, że `Object()` jest również fabryką, ma małe znaczenie praktyczne, ale warto o tym wspomnieć, by mieć świadomość, iż wzorzec fabryki pojawia się niemal wszędzie.

Iterator

We wzorcu iteratora mamy do czynienia z pewnym obiektem zawierającym zagregowane dane. Dane te mogą być przechowywane wewnętrznie w bardzo złożonej strukturze, ale sekwencyjny dostęp do nich zapewnia bardzo prosta funkcja. Kod korzystający z obiektu nie musi znać całej złożoności struktury danych — wystarczy, że wie, jak korzystać z pojedynczego elementu i pobrać następny.

We wzorcu iteratora kluczową rolę odgrywa metoda `next()`. Każde jej wywołanie powinno zwracać następny element w kolejce. To, jak ułożona jest kolejka i jak posortowane są elementy, zależy od zastosowanej struktury danych.

Przy założeniu, że obiekt znajduje się w zmiennej `agg`, dostęp do wszystkich elementów danych uzyska się dzięki wywoływaniu `next()` w pętli:

```
var element;
while (element = agg.next()) {
    // wykonanie działań na elemencie...
    console.log(element);
}
```

We wzorcu iteratora bardzo często obiekt agregujący zapewnia dodatkową metodę pomocniczą `hasNext()`, która informuje użytkownika, czy został już osiągnięty koniec danych. Inny sposób uzyskania sekwencyjnego dostępu do wszystkich elementów, tym razem z użyciem `hasNext()`, mógłby wyglądać następująco:

```
while (agg.hasNext()) {
    // wykonanie działań na następnym elemencie...
    console.log(agg.next());
}
```

Po przedstawieniu sposobów użycia wzorca czas na implementację obiektu agregującego.

Implementując wzorzec iteratora, warto w zmiennej prywatnej przechowywać dane oraz wskaźnik (indeks) do następnego elementu. W naszym przykładzie załóżmy, że dane to typowa tablica, a „specjalna" logika pobierania tak naprawdę zwraca jej następny element.

```
var agg = (function () {

    var index = 0,
        data = [1, 2, 3, 4, 5],
        length = data.length;

    return {

        next: function () {
            var element;
            if (!this.hasNext()) {
                return null;
            }
            element = data[index];
            index = index + 2;
            return element;
        },

        hasNext: function () {
            return index < length;
        }

    };
}());
```

Aby zapewnić łatwiejszy dostęp do danych i możliwość kilkukrotnej iteracji, obiekt może oferować dodatkowe metody:

- `rewind()` — ustawia wskaźnik na początek kolejki;
- `current()` — zwraca aktualny element, bo nie można tego uczynić za pomocą `next()` bez jednoczesnej zmiany wskaźnika.

Implementacja tych dodatkowych metod nie sprawi żadnych trudności.

```
var agg = (function () {

    // [jak wyżej...]

    return {

        // [jak wyżej...]

        rewind: function () {
            index = 0;
        },
        current: function () {
            return data[index];
        }
    };
}());
```

Oto dodatkowy test iteratora:

```
// pętla wyświetla wartości 1, 3 i 5
while (agg.hasNext()) {
    console.log(agg.next());
}

// powrót na początek
agg.rewind();
console.log(agg.current()); // 1
```

W konsoli pojawią się następujące wartości: 1, 3 i 5 (z pętli), a na końcu ponownie 1 (po przejściu na początek kolejki).

Dekorator

We wzorcu dekoratora dodatkową funkcjonalność można dodawać do obiektu dynamicznie w trakcie działania programu. W przypadku korzystania ze statycznych i niezmiennych klas jest to faktycznie duże wyzwanie. W języku JavaScript obiekty można modyfikować, więc dodanie do nich nowej funkcjonalności nie stanowi wielkiego problemu.

Dodatkową cechą wzorca dekoratora jest łatwość dostosowania i konfiguracji jego oczekiwanego zachowania. Zaczyna się od prostego obiektu z podstawową funkcjonalnością. Następnie wybiera się kilka z zestawu dostępnych dekoratorów, po czym rozszerza się nimi podstawowy obiekt. Czasem istotna jest kolejność tego rozszerzania.

Sposób użycia

Przyjrzyjmy się sposobom użycia tego wzorca. Przypuśćmy, że opracowujemy aplikację, która coś sprzedaje. Każda nowa sprzedaż to nowy obiekt `sale`. Obiekt zna cenę produktu i potrafi ją zwrócić po wywołaniu metody `sale.getPrice()`. W zależności od aktualnych warunków można zacząć „dekorować" obiekt dodatkową funkcjonalnością. Wyobraźmy sobie, że jako amerykański sklep sprzedajemy produkt klientowi z kanadyjskiej prowincji Québec. W takiej sytuacji klient musi zapłacić podatek federalny i dodatkowo podatek lokalny. We wzorcu dekoratora będziemy więc „dekorowali" obiekt dekoratorem podatku federalnego i dekoratorem podatku lokalnego. Po wyliczeniu ceny końcowej można również dodać dekorator do jej formatowania. Scenariusz byłby następujący:

```
var sale = new Sale(100); // cena wynosi 100 dolarów
sale = sale.decorate('fedtax'); // dodaj podatek federalny
sale = sale.decorate('quebec'); // dodaj podatek lokalny
sale = sale.decorate('money'); // formatowanie ceny
sale.getPrice(); // "USD 112.88"
```

W innym scenariuszu kupujący może mieszkać w prowincji, która nie stosuje podatku lokalnego, i dodatkowo możemy chcieć podać cenę w dolarach kanadyjskich.

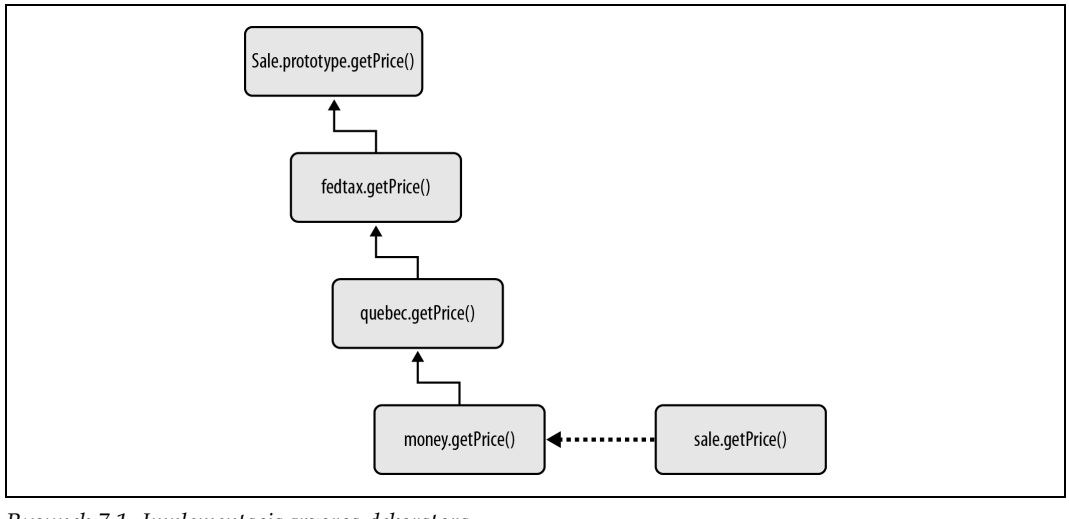
```
var sale = new Sale(100); // cena wynosi 100 dolarów
sale = sale.decorate('fedtax'); // dodaj podatek federalny
sale = sale.decorate('cdn'); // sformatuj jako dolary kanadyjskie
sale.getPrice(); // "CAD 105.00"
```

Nietrudno zauważyć, że jest to wygodny i elastyczny sposób dodawania lub modyfikowania funkcjonalności utworzonych już obiektów. Czas na implementację wzorca.

Implementacja

Jednym ze sposobów implementacji wzorca dekoratora jest utworzenie dekoratorów jako obiektów zawierających metody do nadpisania. Każdy dekorator dziedziczy wówczas tak naprawdę po obiekcie rozszerzonym przez poprzedni dekorator. Każda dekorowana metoda wywołuje swoją poprzedniczkę za pomocą `uber` (odziedziczony obiekt), pobiera wartość i przetwarza ją, dodając coś nowego.

Efekt jest taki, że wywołanie metody `sale.getPrice()` z pierwszego z przedstawionych przykładów powoduje tak naprawdę wywołanie metody dekoratora `money` (patrz rysunek 7.1). Ponieważ jednak każdy dekorator wywołuje najpierw odpowiadającą mu metodę ze swego poprzednika, `getPrice()` z `money` wywołuje `getPrice()` z `quebec`, a ta metodę `getPrice()` z `fedtax` i tak dalej. Łańcuch może być dłuższy, ale kończy się oryginalną metodą `getPrice()` zaimplementowaną przez konstruktor `Sale()`.



Rysunek 7.1. Implementacja wzorca dekoratora

Implementacja rozpoczyna się od konstruktora i metody prototypu.

```
function Sale(price) {
    this.price = price || 100;
}
Sale.prototype.getPrice = function () {
    return this.price;
};
```

Wszystkie obiekty dekoratorów znajdą się we właściwości konstruktora:

```
Sale.decorators = {};
```

Przyjrzyjmy się przykładowemu dekoratorowi. To obiekt implementujący zmodyfikowaną wersję metody `getPrice()`. Metoda najpierw pobiera zwróconą przez metodę przodka wartość, a następnie ją modyfikuje.

```
Sale.decorators.fedtax = {
    getPrice: function () {
        var price = this.uber.getPrice();
        price += price * 5 / 100;
        return price;
    }
};
```

W podobny sposób można zaimplementować dowolną liczbę innych dekoratorów. Mogą one stanowić rozszerzenie podstawowej funkcjonalności `Sale()`, czyli działać jak dodatki. Co więcej, nic nie stoi na przeszkodzie, by znajdowały się w dodatkowych plikach i były implementowane przez innych, niezależnych programistów.

```
Sale.decorators.quebec = {
    getPrice: function () {
        var price = this.uber.getPrice();
        price += price * 7.5 / 100;
        return price;
    }
};

Sale.decorators.money = {
    getPrice: function () {
        return "USD " + this.uber.getPrice().toFixed(2);
    }
};

Sale.decorators.cdn = {
    getPrice: function () {
        return "CAD " + this.uber.getPrice().toFixed(2);
    }
};
```

Na koniec przyjrzyjmy się „magicznej” metodzie o nazwie `decorate()`, która łączy ze sobą wszystkie elementy. Sposób jej użycia jest następujący:

```
sale = sale.decorate('fedtax');
```

Tekst `'fedtax'` odpowiada obiektowi zaimplementowanemu w `Sale.decorators.fedtax`. Nowy obiekt `newobj` dziedziczy obiekt aktualny (oryginał lub już udekorowaną wersję), który jest zawarty w `this`. Do zapewnienia dziedziczenia wykorzystajmy wzorzec konstruktora tymczasowego z poprzedniego rozdziału. Dodatkowo ustawmy właściwość `uber` obiektu `newobj`, by potomek miał dostęp do przodka. Następnie niech kod kopiuje wszystkie właściwości z dekoratora do nowego obiektu i zwraca `newobj` jako wynik całej operacji, co spowoduje, że stanie się on nowym obiektem `sale`.

```
Sale.prototype.decorate = function (decorator) {
    var F = function () {},
        overrides = this.constructor.decorators[decorator],
        i, newobj;
    F.prototype = this;
    newobj = new F();
    newobj.uber = F.prototype;
    for (i in overrides) {
        if (overrides.hasOwnProperty(i)) {
            newobj[i] = overrides[i];
        }
    }
    return newobj;
};
```

Implementacja wykorzystująca listę

Przeanalizujmy inną implementację, która korzysta z dynamicznej natury języka JavaScript i w ogóle nie stosuje dziedziczenia. Dodatkowo, zamiast wymuszać na każdej metodzie dekorującej, by wywoływała swoją poprzedniczkę, przekazujemy tu wynik poprzedniej metody jako parametr następnej.

Taka implementacja znacząco ułatwia **wycofanie** udekorowania, czyli usunięcie jednego z elementów z listy dekoratorów.

Sposób użycia nowej implementacji będzie prostszy, bo nie wymaga ona przypisywania wartości zwróconej przez `decorate()`. W tym przypadku `decorate()` jedynie dodaje nowy element do listy:

```
var sale = new Sale(100); // cena wynosi 100 dolarów
sale.decorate('fedtax'); // dodaj podatek federalny
sale.decorate('quebec'); // dodaj podatek lokalny
sale.decorate('money'); // formatowanie ceny
sale.getPrice(); // "USD 112.88"
```

Tym razem konstruktor `Sale()` zawiera listę dekoratorów jako własną właściwość.

```
function Sale(price) {
    this.price = (price > 0) || 100;
    this.decorators_list = [];
}
```

Dostępne dekoratory są ponownie implementowane jako właściwości `Sale.decorators`. Są prostsze, bo nie muszą już wywoływać poprzedniej wersji metody `getPrice()`, by uzyskać wartość pośrednią. Teraz trafia ona do systemu jako parametr.

```
Sale.decorators = {};

Sale.decorators.fedtax = {
    getPrice: function (price) {
        return price + price * 5 / 100;
    }
};

Sale.decorators.quebec = {
    getPrice: function (price) {
        return price + price * 7.5 / 100;
    }
};
```

```
Sale.decorators.money = {
    getPrice: function (price) {
        return "USD " + price.toFixed(2);
    }
};
```

Interesujące konstrukcje pojawiają się w metodach `decorate()` i `getPrice()` oryginalnego obiektu. W poprzedniej implementacji metoda `decorate()` była w miarę złożona, a `getPrice()` niezwykle prosta. W nowej jest dokładnie odwrotnie — `decorate()` po prostu dodaje nowy element do listy, a `getPrice()` wykonuje całą istotną pracę. Pracą tą jest przejście przez listę wszystkich dodanych dekoratorów i wywołanie dla każdego z nich metody `getPrice()` z poprzednią wartością podaną jako argument metody.

```
Sale.prototype.decorate = function (decorator) {
    this.decorators_list.push(decorator);
};

Sale.prototype.getPrice = function () {
    var price = this.price,
        i,
        max = this.decorators_list.length,
        name;
    for (i = 0; i < max; i += 1) {
        name = this.decorators_list[i];
        price = Sale.decorators[name].getPrice(price);
    }
    return price;
};
```

Druga implementacja jest prostsza i nie korzysta z dziedziczenia. Prostsze są również metody dekorujące. Całą rzeczywistą pracę wykonuje metoda, która „zgadza" się na dekorację. W tej prostej implementacji dekorację dopuszcza jedynie metoda `getPrice()`. Jeśli dekoracja miałaby dotyczyć większej liczby metod, każda z nich musiałaby przejść przez listę dekoratorów i wywołać odpowiednie metody. Oczywiście taki kod stosunkowo łatwo jest umieścić w osobnej metodzie pomocniczej i uogólnić. Umożliwiałby on dodanie dekorowalności do dowolnej metody. Co więcej, w takiej implementacji właściwość `decorators_list` byłaby obiektem z właściwościami o nazwach metod i z tablicami dekorowanych obiektów jako wartościami.

Strategia

Wzorzec strategii umożliwia wybór odpowiedniego algorytmu na etapie działania aplikacji. Użytkownicy kodu mogą stosować ten sam interfejs zewnętrzny, ale wybierać spośród kilku dostępnych algorytmów, by lepiej dopasować implementację do aktualnego **kontekstu**.

Przykładem wzorca strategii może być rozwiązywanie problemu walidacji formularzy. Można utworzyć jeden obiekt sprawdzania z metodą `validate()`. Metoda zostanie wywołana niezależnie od rodzaju formularza i zawsze zwróci ten sam wynik — listę danych, które nie są poprawne, wraz z komunikatami o błędach.

W zależności od sprawdzanych danych i typu formularza użytkownik kodu może wybrać różne rodzaje sprawdzeń. Walidator wybiera najlepszą **strategię** wykonania zadania i deleguje konkretne czynności sprawdzeń do odpowiednich algorytmów.

Przykład walidacji danych

Przypuśćmy, że mamy do czynienia z następującym zestawem danych pochodzącym najprawdopodobniej z formularza i że chcemy go sprawdzić pod kątem poprawności:

```
var data = {
    first_name: "Super",
    last_name: "Man",
    age: "unknown",
    username: "o_O"
};
```

Aby walidator znał najlepszą strategię do zastosowania w tym konkretnym przykładzie, trzeba najpierw go skonfigurować, określając zestaw reguł i wartości uznawanych za prawidłowe.

Przypuśćmy, że nie wymagamy podania nazwiska i zaakceptujemy dowolną wartość imienia, ale wymagamy podania wieku jako liczby i nazwy użytkownika, która składa się tylko z liczb i liter bez znaków specjalnych. Konfiguracja mogłaby wyglądać następująco:

```
validator.config = {
    first_name: 'isNonEmpty',
    age: 'isNumber',
    username: 'isAlphaNum'
};
```

Po skonfigurowaniu obiektu `validator` jest on gotowy do przyjęcia danych. Wywołujemy jego metodę `validate()` i wyświetlamy błędy walidacji w konsoli.

```
validator.validate(data);
if (validator.hasErrors()) {
    console.log(validator.messages.join("\n"));
}
```

Efektem wykonania kodu mógłby być następujący komunikat:

```
Niepoprawna wartość *age*; wartość musi być liczbą, na przykład 1, 3.14 lub 2010
Niepoprawna wartość *username*; wartość musi zawierać jedynie litery i cyfry bez
żadnych znaków specjalnych
```

Przyjrzyjmy się implementacji walidatora. Poszczególne algorytmy są obiektami o z góry ustalonym interfejsie — zawierają metodę `validate()` i jednowierszową informację wykorzystywaną jako komunikat o błędzie.

```
// sprawdzenie, czy podano jakąś wartość
validator.types.isNonEmpty = {
    validate: function (value) {
        return value !== "";
    },
    instructions: "wartość nie może być pusta"
};

// sprawdzenie, czy wartość jest liczbą
validator.types.isNumber = {
    validate: function (value) {
        return !isNaN(value);
    },
    instructions: "wartość musi być liczbą, na przykład 1, 3.14 lub 2010"
};

// sprawdzenie, czy wartość zawiera jedynie litery i cyfry
validator.types.isAlphaNum = {
    validate: function (value) {
```

```
            return !/[^a-zO-9]/i.test(value);
        },
        instructions: "wartość musi zawierać jedynie litery i cyfry bez żadnych znaków
        ↪specjalnych"
    };
```

Najwyższy czas na obiekt `validator`:

```
    var validator = {

        // wszystkie dostępne sprawdzenia
        types: {},

        // komunikaty o błędach
        // z aktualnej sesji walidacyjnej
        messages: [],

        // aktualna konfiguracja walidacji
        // nazwa => rodzaj testu
        config: {},

        // metoda interfejsu
        // data to pary klucz-wartość
        validate: function (data) {

            var i, msg, type, checker, result_ok;

            // usunięcie wszystkich komunikatów
            this.messages = [];

            for (i in data) {

                if (data.hasOwnProperty(i)) {

                    type = this.config[i];
                    checker = this.types[type];

                    if (!type) {
                        continue; // nie trzeba sprawdzać
                    }
                    if (!checker) { // ojej
                        throw {
                            name: "ValidationError",
                            message: "Brak obsługi dla klucza " + type
                        };
                    }

                    result_ok = checker.validate(data[i]);
                    if (!result_ok) {
                        msg = "Niepoprawna wartość *" + i + "*; " + checker.instructions;
                    this.messages.push(msg);
                    }
                }
            }
            return this.hasErrors();
        },

        // metoda pomocnicza
        hasErrors: function () {
            return this.messages.length !== 0;
        }
    };
```

Obiekt `validator` jest uniwersalny i będzie działał prawidłowo dla różnych rodzajów sprawdzeń. Jednym z usprawnień mogłoby być dodanie kilku nowych testów. Po wykonaniu kilku różnych formularzy z walidacją Twoja lista dostępnych sprawdzeń z pewnością się wydłuży. Każdy kolejny formularz będzie wymagał jedynie skonfigurowania walidatora i uruchomienia metody `validate()`.

Fasada

Wzorzec fasady jest bardzo prosty i ma za zadanie zapewnić alternatywny interfejs obiektu. Dobrą praktyką jest stosowanie krótkich metod, które nie wykonują zbyt wielu zadań. Stosując to podejście, uzyskuje się znacznie więcej metod niż w przypadku tworzenia **supermetod** z wieloma parametrami. W większości sytuacji dwie lub więcej metod wykonuje się jednocześnie. Warto wtedy utworzyć jeszcze jedną metodę, która stanowi otoczkę dla takich połączeń.

W trakcie obsługi zdarzeń przeglądarki bardzo często korzysta się z następujących metod:

- `stopPropagation()` — zapobiega wykonywaniu obsługi zdarzenia w węzłach nadrzędnych;
- `preventDefault()` — zapobiega wykonaniu przez przeglądarkę domyślnej akcji dla zdarzenia (na przykład kliknięcia łącza lub wysłania formularza).

To dwie osobne metody wykonujące odmienne zadania, więc nie stanowią jednej całości, ale z drugiej strony w zdecydowanej większości sytuacji są one wykonywane jednocześnie. Zamiast więc powielać wywołania obu metod w całej aplikacji, można utworzyć fasadę, która je obie wykona.

```
var myevent = {
    //...
    stop: function (e) {
        e.preventDefault();
        e.stopPropagation();
    }
    //...
};
```

Wzorzec fasady przydaje się również w sytuacjach, w których za fasadą warto ukryć różnice pomiędzy przeglądarkami internetowymi. Nic nie stoi na przeszkodzie, by rozbudować poprzedni przykład o inny sposób obsługi anulowania zdarzeń przez przeglądarkę IE.

```
var myevent = {
    //...
    stop: function (e) {
        // inne
        if (typeof e.preventDefault === "function") {
            e.preventDefault();
        }
        if (typeof e.stopPropagation === "function") {
            e.stopPropagation();
        }
        // IE
        if (typeof e.returnValue === "boolean") {
            e.returnValue = false;
        }
        if (typeof e.cancelBubble === "boolean") {
            e.cancelBubble = true;
        }
    }
    //...
};
```

Wzorzec fasady bywa pomocny w przypadku zmiany interfejsów związanej na przykład z refaktoryzacją. Gdy chce się zamienić obiekt na inną implementację, najczęściej może to zająć sporo czasu (jeśli jest on naprawdę rozbudowany). Załóżmy też, że już powstaje kod dla nowego interfejsu. W takiej sytuacji można utworzyć przed starym obiektem fasadę, która imituje nowy interfejs. W ten sposób po dokonaniu rzeczywistej zamiany i pozbyciu się starego obiektu ilość zmian w najnowszym kodzie zostanie ograniczona do minimum.

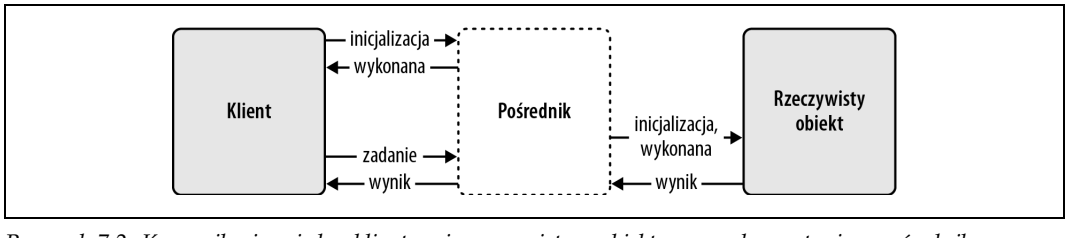
Pośrednik

We wzorcu projektowym pośrednika jeden obiekt stanowi interfejs dla innego obiektu. Różni się to od wzorca fasady, w którym po prostu istnieją pewne metody dodatkowe łączące w sobie wywołania kilku innych metod. Pośrednik znajduje się między użytkownikiem a obiektem i broni dostępu do niego.

Choć wzorzec wygląda jak dodatkowy narzut, w rzeczywistości często służy do poprawy wydajności. Pośrednik staje się strażnikiem rzeczywistego obiektu i stara się, by ten wykonał jak najmniej pracy.

Jednym z przykładów zastosowania pośrednika jest tak zwana **leniwa inicjalizacja**. Stosuje się ją w sytuacjach, w których inicjalizacja rzeczywistego obiektu jest kosztowna, a istnieje spora szansa, że klient po jego zainicjalizowaniu tak naprawdę nigdy go nie użyje. Pośrednik może wtedy stanowić interfejs dla rzeczywistego obiektu. Otrzymuje polecenie inicjalizacji, ale nie przekazuje go dalej aż do momentu, gdy rzeczywisty obiekt naprawdę zostanie użyty.

Rysunek 7.2 ilustruje sytuację, w której klient wysyła polecenie inicjalizujące, a pośrednik odpowiada, że wszystko jest w porządku, choć tak naprawdę nie przekazuje polecenia dalej. Czeka z inicjalizacją właściwego obiektu do czasu, gdy klient rzeczywiście będzie wykonywał na nim pracę — wówczas przekazuje obydwa komunikaty.



Rysunek 7.2. Komunikacja między klientem i rzeczywistym obiektem z wykorzystaniem pośrednika

Przykład

Wzorzec pośrednika bywa przydatny, gdy rzeczywisty obiekt docelowy wykonuje kosztowne zadanie. W aplikacjach internetowych jedną z kosztownych sytuacji jest żądanie sieciowe, więc w miarę możliwości warto zebrać kilka operacji i wykonać je jednym żądaniem. Prześledźmy praktyczne zastosowanie wzorca właśnie w takiej sytuacji.

Aplikacja wideo

Załóżmy istnienie prostej aplikacji odtwarzającej materiał wideo wybranego artysty (patrz rysunek 7.3). W zasadzie możesz nawet przetestować kod, wpisując w przeglądarce internetowej adres *http://www.jspatterns.com/book/7/proxy.html*.

Rysunek 7.3. Aplikacja wideo w akcji

Strona zawiera listę tytułów materiałów wideo. Gdy użytkownik kliknie tytuł, obszar poniżej rozszerzy się, by przedstawić dodatkowe informacje i umożliwić odtworzenie filmu. Szczegóły dotyczące materiałów oraz adres URL treści wideo nie stanowią części strony — są pobierane poprzez osobne wywołania serwera. Serwer przyjmuje jako parametr kilka identyfikatorów materiałów wideo, więc aplikację można przyspieszyć, wykonując mniej żądań HTTP i pobierając za każdym razem dane kilku filmów.

Aplikacja umożliwia jednoczesne rozwinięcie szczegółów kilku (a nawet wszystkich) materiałów, co stanowi doskonałą okazję do połączenia kilku żądań w jedno.

Bez użycia pośrednika

Głównymi elementami aplikacji są dwa obiekty:

- `videos` — jest odpowiedzialny za rozwijanie i zwijanie obszarów informacyjnych (metoda `videos.getInfo()`) oraz za odtwarzanie materiałów wideo (metoda `videos.getPlayer()`).
- `http` — jest odpowiedzialny za komunikację z serwerem za pomocą metody `http.make↪Request()`.

Bez stosowania pośrednika `videos.getInfo()` wywoła `http.makeRequest()` dla każdego materiału wideo. Po dodaniu pośrednika pojawi się nowy obiekt o nazwie `proxy` znajdujący się między `videos` oraz `http` i delegujący wszystkie wywołania `makeRequest()`, a także łączący je ze sobą.

Najpierw pojawi się kod, w którym nie zastosowano wzorca pośrednika. Druga wersja, stosująca obiekt pośrednika, poprawi ogólną płynność działania aplikacji.

Kod HTML

Kod HTML to po prostu zbiór łączy.

```
<p><span id="toggle-all">Przełącz zaznaczone</span></p>
<ol id="vids">
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--2158073">Gravedigger</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--4472739">Save Me</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--45286339">Crush</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--2144530">Don't Drink The Water
        ↪</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--21724180O">Funny the Way It Is
        ↪</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--2144532">What Would You Say</a></li>
</ol>
```

Obsługa zdarzeń

Zanim pojawi się właściwa obsługa zdarzeń, warto dodać funkcję pomocniczą `$` do pobierania elementów DOM na podstawie ich identyfikatorów.

```
var $ = function (id) {
    return document.getElementById(id);
};
```

Stosując delegację zdarzeń (więcej na ten temat w rozdziale 8.), można obsłużyć wszystkie kliknięcia dotyczące listy uporządkowanej `id="vids"` za pomocą jednej funkcji.

```
$('vids').onclick = function (e) {
    var src, id;

    e = e || window.event;
    src = e.target || e.srcElement;

    if (src.nodeName !== "A") {
        return;
    }

    if (typeof e.preventDefault === "function") {
        e.preventDefault();
    }
    e.returnValue = false;

    id = src.href.split('--')[1];

    if (src.className === "play") {
```

```
            src.parentNode.innerHTML = videos.getPlayer(id);
            return;
        }

        src.parentNode.id = "v" + id;
        videos.getInfo(id);
    };
```

Obsługa kliknięcia zainteresowana jest tak naprawdę dwoma sytuacjami: pierwszą dotyczącą rozwinięcia lub zamknięcia części informacyjnej (wywołanie `getInfo()`) i drugą związaną z odtworzeniem materiału wideo (gdy kliknięcie dotyczyło obiektu z klasą `play`), co oznacza, że rozwinięcie już nastąpiło i można bezpiecznie wywołać metodę `getPlayer()`. Identyfikatory materiałów wideo wydobywa się z atrybutów `href` łączy.

Druga z funkcji obsługujących kliknięcia dotyczy sytuacji, w której użytkownik chce przełączyć wszystkie części informacyjne. W zasadzie sprowadza się ona do wywoływania w pętli metody `getInfo()`.

```
$('toggle-all').onclick = function (e) {

    var hrefs,
        i,
        max,
        id;

    hrefs = $('vids').getElementsByTagName('a');
    for (i = 0, max = hrefs.length; i < max; i += 1) {
        // pomiń łącza odtwarzania
        if (hrefs[i].className === "play") {
            continue;
        }
        // pomiń niezaznaczone
        if (!hrefs[i].parentNode.firstChild.checked) {
            continue;
        }

        id = hrefs[i].href.split('--')[1];
        hrefs[i].parentNode.id = "v" + id;
        videos.getInfo(id);
    }
};
```

Obiekt videos

Obiekt `videos` zawiera trzy metody:

- `getPlayer()` — zwraca kod HTML niezbędny do odtworzenia materiału wideo (nieistotny w rozważaniach na temat obiektu pośrednika).
- `updateList()` — wywołanie zwrotne otrzymujące wszystkie dane z serwera i generujące kod HTML do wykorzystania przy rozwijaniu szczegółów filmów (w tej metodzie również nie dzieje się nic interesującego).
- `getInfo()` — metoda przełączająca widoczność części informacyjnych i wykonująca metody obiektu `http` przez przekazanie `updateList()` jako funkcji wywołania zwrotnego.

Oto istotny fragment obiektu `videos`:

```
var videos = {

    getPlayer: function (id) {...},
    updateList: function (data) {...},

    getInfo: function (id) {

        var info = $('info' + id);

        if (!info) {
            http.makeRequest([id], "videos.updateList");
            return;
        }

        if (info.style.display === "none") {
            info.style.display = '';
        } else {
            info.style.display = 'none';
        }
    }
};
```

Obiekt http

Obiekt `http` ma tylko jedną metodę, która wykonuje żądanie JSONP do usługi YQL firmy Yahoo.

```
var http = {
    makeRequest: function (ids, callback) {
        var url = 'http://query.yahooapis.com/v1/public/yql?q=',
            sql = 'select * from music.video.id where ids IN ("%ID%")',
            format = "format=json",
            handler = "callback=" + callback,
            script = document.createElement('script');

        sql = sql.replace('%ID%', ids.join('","'));
        sql = encodeURIComponent(sql);

        url += sql + '&' + format + '&' + handler;
        script.src = url;

        document.body.appendChild(script);
    }
};
```

YQL (*Yahoo! Query Language*) to **uogólniona** usługa internetowa, która oferuje możliwość korzystania ze składni przypominającej SQL do pobierania danych z innych usług. W ten sposób nie trzeba poznawać szczegółów ich API.

Gdy jednocześnie przełączone zostaną wszystkie materiały wideo, do serwera trafi sześć osobnych żądań; każde będzie podobne do następującego żądania YQL:

```
select * from music.video.id where ids IN ("2158073")
```

Obiekt proxy

Zaprezentowany wcześniej kod działa prawidłowo, ale można go zoptymalizować. Na scenę wkracza obiekt `proxy`, który przejmuje komunikację między `http` i `videos`. Obiekt stara się połączyć ze sobą kilka żądań, czekając na ich zebranie 50 ms. Obiekt `videos` nie wywołuje

usługi HTTP bezpośrednio, ale przez pośrednika. Ten czeka krótką chwilę z wysłaniem żądania. Jeśli wywołania z `videos` będą przychodziły w odstępach krótszych niż 50 ms, zostaną połączone w jedno żądanie. Takie opóźnienie nie jest szczególnie widoczne, ale pomaga znacząco przyspieszyć działanie aplikacji w przypadku jednoczesnego odsłaniania więcej niż jednego materiału wideo. Co więcej, jest również przyjazne dla serwera, który nie musi obsługiwać sporej liczby żądań.

Zapytanie YQL dla dwóch materiałów wideo może mieć postać:

```
select * from music.video.id where ids IN ("2158073", "123456")
```

W istniejącym kodzie zachodzi tylko jedna zmiana: metoda `videos.getInfo()` wywołuje metodę `proxy.makeRequest()` zamiast metody `http.makeRequest()`.

```
proxy.makeRequest(id, videos.updateList, videos);
```

Obiekt pośrednika korzysta z kolejki, w której gromadzi identyfikatory materiałów wideo przekazane w ostatnich 50 ms. Następnie przekazuje wszystkie identyfikatory, wywołując metodę obiektu `http` i przekazując własną funkcję wywołania zwrotnego, ponieważ `videos.updateList()` potrafi przetworzyć tylko pojedynczy rekord danych.

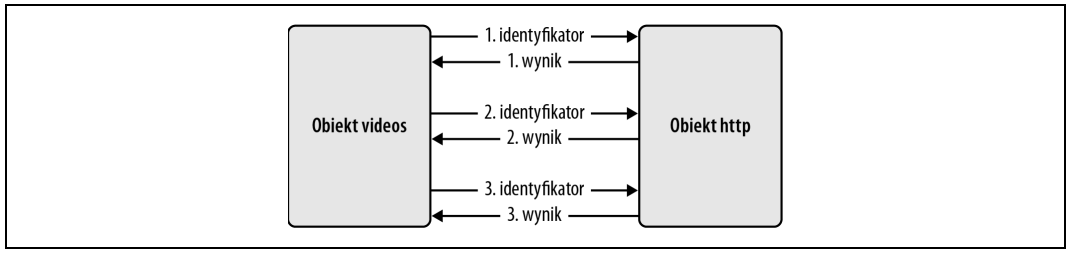
Oto kod obiektu pośredniczącego `proxy`:

```
var proxy = {
    ids: [],
    delay: 50,
    timeout: null,
    callback: null,
    context: null,
    makeRequest: function (id, callback, context) {

        // dodanie do kolejki
        this.ids.push(id);

        this.callback = callback;
        this.context = context;

        // ustawienie funkcji czasowej
        if (!this.timeout) {
            this.timeout = setTimeout(function () {
                proxy.flush();
            }, this.delay);
        }
    },
    flush: function () {

        http.makeRequest(this.ids, "proxy.handler");

        // wyczyszczenie kolejki i funkcji czasowej
        this.timeout = null;
        this.ids = [];
    },
    handler: function (data) {
        var i, max;

        // pojedynczy materiał wideo
        if (parseInt(data.query.count, 10) === 1) {
            proxy.callback.call(proxy.context, data.query.results.Video);
            return;
        }
```

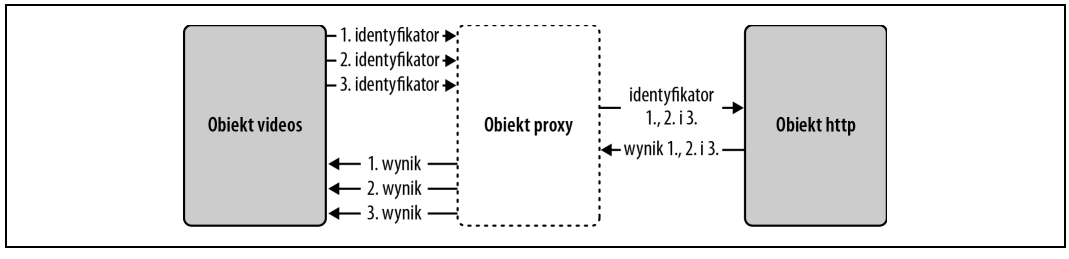
```
            // kilka materiałów wideo
            for (i = 0, max = data.query.results.Video.length; i < max; i += 1) {
                proxy.callback.call(proxy.context, data.query.results.Video[i]);
            }
        }
    };
```

Wprowadzenie pośrednika umożliwiło połączenie kilku żądań pobrania danych w jedno poprzez zmianę tylko jednego wiersza oryginalnego kodu.

Rysunki 7.4 i 7.5 przedstawiają scenariusze z trzema osobnymi żądaniami (bez pośrednika) i z jednym połączonym żądaniem (po użyciu pośrednika).



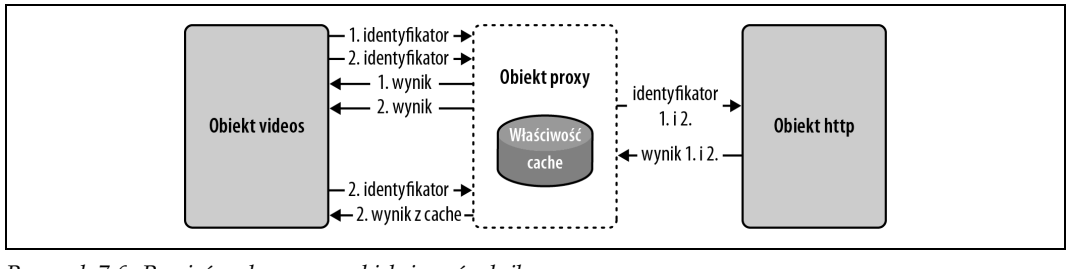
Rysunek 7.4. Trzy osobne żądania do serwera



Rysunek 7.5. Wykorzystanie pośrednika do zmniejszenia liczby żądań wysyłanych do serwera

Pośrednik jako pamięć podręczna

W prezentowanym przykładzie obiekt `videos` żądający danych jest na tyle inteligentny, że nie żąda tych samych informacji dwukrotnie. Nie zawsze jednak musi tak być. Pośrednik może pójść o krok dalej i chronić rzeczywisty obiekt `http` przed powielaniem tych samych żądań, zapamiętując je w nowej właściwości `cache` (patrz rysunek 7.6). Gdyby obiekt `videos` ponownie poprosił o informacje o tym samym materiale (ten sam identyfikator), pośrednik wydobyłby dane z pamięci podręcznej i uniknął komunikacji z serwerem.

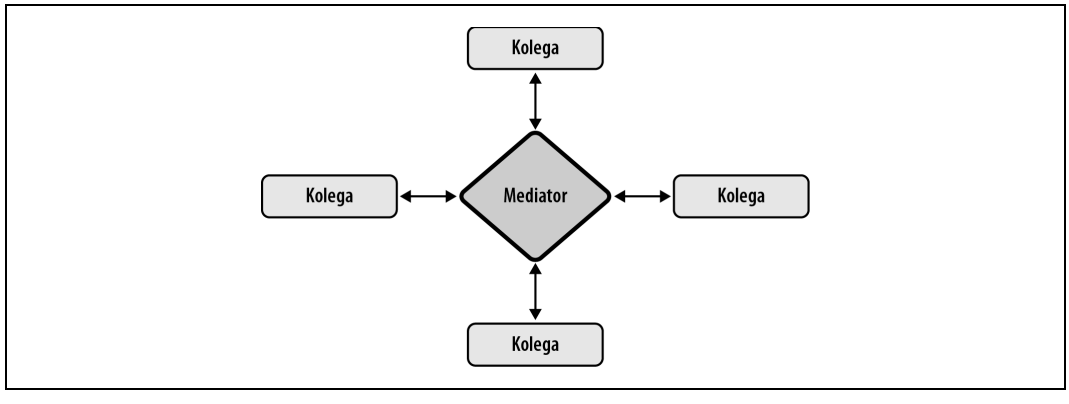


Rysunek 7.6. Pamięć podręczna w obiekcie pośrednika

Mediator

Aplikacje — duże czy małe — składają się z wielu obiektów. Obiekty muszą się ze sobą komunikować w sposób, który nie uczyni przyszłej konserwacji kodu prawdziwą drogą przez mękę i umożliwi bezpieczną zmianę jednego fragmentu bez potrzeby przepisywania wszystkich innych. Gdy aplikacja się rozrasta, pojawiają się coraz to nowe obiekty. W trakcie refaktoryzacji obiekty usuwa się lub przerabia. Gdy wiedzą o sobie za dużo i komunikują się bezpośrednio (wywołują się wzajemnie i modyfikują właściwości), powstaje między nimi niepożądany **ścisły związek**. Jeśli obiekty są ze sobą powiązane zbyt mocno, niełatwo zmienić jeden z nich bez modyfikacji pozostałych. Wtedy nawet najprostsza zmiana w aplikacji nie jest dłużej trywialna i bardzo trudno oszacować, ile tak naprawdę czasu trzeba będzie na nią poświęcić.

Wzorzec mediatora ma za zadanie promować **luźne powiązania** obiektów i wspomóc przyszłą konserwację kodu (patrz rysunek 7.7). W tym wzorcu niezależne obiekty (**koledzy**) nie komunikują się ze sobą bezpośrednio, ale korzystają z obiektu **mediatora**. Gdy jeden z kolegów zmieni stan, informuje o tym mediator, a ten przekazuje tę informację wszystkim innym zainteresowanym kolegom.



Rysunek 7.7. Uczestnicy wzorca mediatora

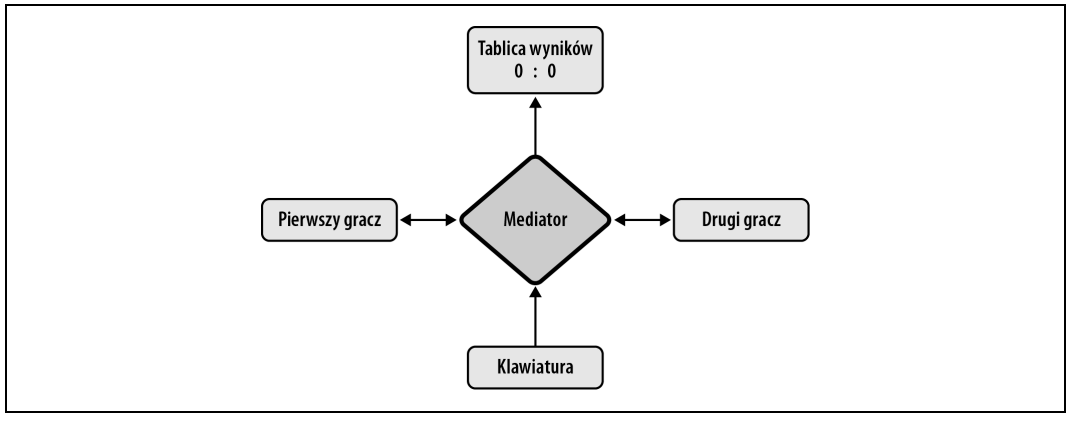
Przykład mediatora

Prześledźmy przykład użycia wzorca mediatora. Aplikacja będzie grą, w której dwóch graczy przez pół minuty stara się jak najczęściej klikać w przycisk. Pierwszy gracz naciska klawisz nr 1, a drugi klawisz 0 (spory odstęp między klawiszami zapewnia, że nie pobiją się o klawiaturę). Tablica wyników pokazuje aktualny stan rywalizacji.

Obiektami uczestniczącymi w wymianie informacji są:

- pierwszy gracz,
- drugi gracz,
- tablica,
- mediator.

Mediator wie o wszystkich obiektach. Komunikuje się z urządzeniem wejściowym (klawiaturą), obsługuje naciśnięcia klawiszy, określa, który gracz jest aktywny, i informuje o zmianach wyników (patrz rysunek 7.8). Gracz jedynie gra (czyli aktualizuje swój własny wynik) i informuje mediator o tym zdarzeniu. Mediator informuje tablicę o zmianie wyniku, a ta aktualizuje wyświetlaną wartość.



Rysunek 7.8. Uczestnicy w grze na szybkość naciskania klawiszy

Poza mediatorem żaden inny obiekt nie wie nic o pozostałych. Dzięki temu bardzo łatwo zaktualizować grę, na przykład dodać nowego gracza lub zmienić tablicę wyników na wersję wyświetlającą pozostały czas.

Pełna wersja gry wraz z kodem źródłowym jest dostępna pod adresem *http://www.jspatterns.com/book/7/mediator.html*.

Obiekty graczy są tworzone przy użyciu konstruktora `Player()` i zawierają własne właściwości `points` i `name`. Metoda `play()` z prototypu zwiększa liczbę punktów o jeden i informuje o tym fakcie mediator.

```
function Player(name) {
    this.points = 0;
    this.name = name;
}
Player.prototype.play = function () {
    this.points += 1;
    mediator.played();
};
```

Obiekt `scoreboard` zawiera metodę `update()` wywoływaną przez mediator po zdobyciu punktu przez jednego z graczy. Tablica nie wie nic o graczach i nie przechowuje wyniku — po prostu wyświetla informacje przekazane przez mediator.

```
var scoreboard = {

    // aktualizowany element HTML
    element: document.getElementById('results'),

    // aktualizacja wyświetlacza
    update: function (score) {

        var i, msg = '';
        for (i in score) {
            if (score.hasOwnProperty(i)) {
```

```
                msg += '<p><strong>' + i + '<\/strong>: ';
                msg += score[i];
                msg += '<\/p>';
            }
        }
        this.element.innerHTML = msg;
    }
};
```

Czas na obiekt mediatora. Odpowiada on za inicjalizację gry oraz utworzenie obiektów graczy w metodzie `setup()` i śledzenie ich poczynań dzięki umieszczeniu ich we właściwości `players`. Metoda `played()` zostaje wywołana przez każdego z graczy po wykonaniu akcji. Aktualizuje ona wynik (`score`) i przesyła go do tablicy (`scoreboard`). Ostatnia metoda, `keypress()`, obsługuje zdarzenia klawiatury, określa, który gracz jest aktywny, i powiadamia go o wykonanej akcji.

```
var mediator = {

    // wszyscy gracze
    players: {},

    // inicjalizacja
    setup: function () {
        var players = this.players;
        players.home = new Player('Gospodarze');
        players.guest = new Player('Goście');
    },

    // ktoś zagrał, uaktualnij wynik
    played: function () {
        var players = this.players,
            score = {
                "Gospodarze": players.home.points,
                "Goście": players.guest.points
            };

        scoreboard.update(score);
    },

    // obsługa interakcji z użytkownikiem
    keypress: function (e) {
        e = e || window.event; // IE
        if (e.which === 49) { // klawisz "1"
            mediator.players.home.play();
            return;
        }
        if (e.which === 48) { // klawisz "0"
            mediator.players.guest.play();
            return;
        }
    }
};
```

Ostatni element to uruchomienie i zakończenie gry.

```
// start!
mediator.setup();
window.onkeypress = mediator.keypress;

// gra kończy się po 30 sekundach
setTimeout(function () {
    window.onkeypress = null;
    alert('Koniec gry!');
}, 30000);
```

Obserwator

Wzorzec **obserwatora** jest niezwykle często wykorzystywany w programowaniu po stronie klienta w języku JavaScript. Wszystkie zdarzenia przeglądarki (poruszenie myszą, naciśnięcie klawisza itp.) to przykłady jego użycia. Inną często pojawiającą się nazwą tego wzorca są **zdarzenia własne**, czyli zdarzenia tworzone przez programistę, a nie przeglądarkę. Jeszcze inna nazwa to wzorzec **subskrybenta-dostawcy**.

Głównym celem używania wzorca jest promowanie luźnego powiązania elementów. Zamiast sytuacji, w której jeden obiekt wywołuje metodę drugiego, mamy sytuację, w której drugi z obiektów zgłasza chęć otrzymywania powiadomień o zmianie w pierwszym obiekcie. Subskrybenta nazywa się często obserwatorem, a obiekt obserwowany obiektem publikującym lub źródłem. Obiekt publikujący wywołuje subskrybentów po zajściu istotnego zdarzenia i bardzo często przekazuje informację o nim w postaci obiektu zdarzenia.

Pierwszy przykład — subskrypcja magazynu

Aby dowiedzieć się, jak zaimplementować wzorzec, posłużmy się konkretnym przykładem. Przypuśćmy, że mamy wydawcę `paper`, który publikuje gazetę codzienną i miesięcznik. Subskrybent `joe` zostanie powiadomiony o wydaniu nowego periodyku.

Obiekt `paper` musi zawierać właściwość `subscribers`, która jest tablicą przechowującą wszystkich subskrybentów. Zgłoszenie się do subskrypcji polega jedynie na dodaniu nowego wpisu do tablicy. Gdy zajdzie istotne zdarzenie, obiekt `paper` przejdzie w pętli przez wszystkich subskrybentów, by ich o nim powiadomić. Notyfikacja polega na wywołaniu metody obiektu subskrybenta. Oznacza to, że w momencie zgłoszenia chęci otrzymywania powiadomień subskrybent musi przekazać obiektowi `paper` jedną ze swoich metod w wywołaniu metody `subscribe()`.

Obiekt `paper` może dodatkowo umożliwić anulowanie subskrypcji, czyli usunięcie wpisu z tablicy subskrybentów. Ostatnią istotną metodą obiektu `paper` jest `publish()`, która wywołuje metody subskrybentów. Podsumowując, obiekt publikujący musi zawierać następujące składowe:

- `subscribers` — tablica;
- `subscribe()` — dodaje wpis do tablicy;
- `unsubscribe()` — usuwa wpis z tablicy;
- `publish()` — przechodzi w pętli przez subskrybentów i wywołuje przekazane przez nich metody.

Wszystkie trzy metody potrzebują parametru `type`, ponieważ wydawca może zgłosić kilka różnych zdarzeń (publikację gazety lub magazynu), a subskrybenci mogą zdecydować się na otrzymywanie powiadomień tylko o jednym z nich.

Ponieważ powyższe składowe są bardzo ogólne i mogą być stosowane przez dowolnego wydawcę, warto zaimplementować je jako część osobnego obiektu. W ten sposób będzie je można w przyszłości skopiować do dowolnego obiektu, zamieniając go w wydawcę (obiekt publikujący).

Oto przykładowa implementacja ogólnej funkcjonalności obiektu publikującego, która definiuje wszystkie wymagane składowe oraz metodę pomocniczą `visitSubscribers()`:

```
var publisher = {
    subscribers: {
        any: [] // typ zdarzenia
    },
    subscribe: function (fn, type) {
        type = type || 'any';
        if (typeof this.subscribers[type] === "undefined") {
            this.subscribers[type] = [];
        }
        this.subscribers[type].push(fn);
    },
    unsubscribe: function (fn, type) {
        this.visitSubscribers('unsubscribe', fn, type);
    },
    publish: function (publication, type) {
        this.visitSubscribers('publish', publication, type);
    },
    visitSubscribers: function (action, arg, type) {
        var pubtype = type || 'any',
            subscribers = this.subscribers[pubtype],
            i,
            max = subscribers.length;

        for (i = 0; i < max; i += 1) {
            if (action === 'publish') {
                subscribers[i](arg);
            } else {
                if (subscribers[i] === arg) {
                    subscribers.splice(i, 1);
                }
            }
        }
    }
};
```

Poniżej znajduje się kod funkcji, która przyjmuje obiekt i zamienia go w obiekt publikujący przez proste skopiowanie wszystkich ogólnych metod dotyczących publikacji.

```
function makePublisher(o) {
    var i;
    for (i in publisher) {
        if (publisher.hasOwnProperty(i) && typeof publisher[i] === "function") {
            o[i] = publisher[i];
        }
    }
    o.subscribers = {any: []};
}
```

Czas na implementację obiektu `paper`, który będzie publikował gazetę i magazyn.

```
var paper = {
    daily: function () {
        this.publish("ciekawy news");
    },
    monthly: function () {
        this.publish("interesującą analizę", "magazyn");
    }
};
```

Trzeba jeszcze uczynić z obiektu wydawcę.

```
makePublisher(paper);
```

Po utworzeniu wydawcy możemy utworzyć obiekt subskrybenta o nazwie `joe`, który ma dwie metody.

```
var joe = {
    drinkCoffee: function (paper) {
        console.log('Właśnie przeczytałem ' + paper);
    },
    sundayPreNap: function (monthly) {
        console.log('Chyba zasnę, czytając ' + monthly);
    }
};
```

Następny krok to obiekt `paper` subskrybujący `joe` (tak naprawdę to `joe` zgłasza się jako subskrybent **do** `paper`).

```
paper.subscribe(joe.drinkCoffee);
paper.subscribe(joe.sundayPreNap, 'magazyn');
```

Obiekt `joe` udostępnił dwie metody. Pierwsza z nich powinna być wywoływana dla domyślnego zdarzenia „wszystko", a druga jedynie dla zdarzeń „magazyn". Oto kilka zgłoszeń zdarzeń:

```
paper.daily();
paper.daily();
paper.daily();
paper.monthly();
```

Wszystkie metody publikujące wywołały odpowiednie metody z obiektu `joe`, co spowodowało wyświetlenie w konsoli następującego wyniku:

```
Właśnie przeczytałem ciekawy news
Właśnie przeczytałem ciekawy news
Właśnie przeczytałem ciekawy news
Chyba zasnę, czytając interesującą analizę
```

Bardzo ważnym elementem całego systemu jest to, że `paper` nie zawiera w sobie informacji o `joe` i odwrotnie. Co więcej, nie istnieje obiekt mediatora, który wiedziałby o wszystkich obiektach. Obiekty uczestniczące w interakcjach są ze sobą powiązane bardzo luźno i bez jakichkolwiek modyfikacji można dodać jeszcze kilku subskrybentów. Co ważne, `joe` może w dowolnym momencie anulować subskrypcję.

Nic też nie stoi na przeszkodzie, by `joe` również został wydawcą (przecież to nic trudnego dzięki systemom blogowym i mikroblogowym). Jako wydawca `joe` wysyła aktualizację swojego statusu do serwisu Twitter:

```
makePublisher(joe);
joe.tweet = function (msg) {
    this.publish(msg);
};
```

Wyobraźmy sobie, że dział relacji z klientami wydawcy gazety decyduje się czytać, co o gazecie sądzi jej subskrybent `joe`, i dodaje w tym celu metodę `readTweets()`.

```
paper.readTweets = function (tweet) {
    alert('Zwołajmy duże zebranie! Ktoś napisał: ' + tweet);
};
joe.subscribe(paper.readTweets);
```

Gdy `joe` zamieści swój wpis, wydawca (`paper`) go otrzyma.

```
joe.tweet("nie lubię tej gazety");
```

Wykonanie kodu spowoduje wyświetlenie w konsoli tekstu „Zwołajmy duże zebranie! Ktoś napisał: nie lubię tej gazety".

Pełny kod źródłowy przykładu oraz możliwość sprawdzenia wyników jego działania w konsoli zapewnia strona HTML dostępna pod adresem *http://www.jspatterns.com/book/7/observer.html*.

Drugi przykład — gra w naciskanie klawiszy

Przyjrzymy się jeszcze jednemu przykładowi. Zaimplementujemy tę samą grę w naciskanie klawiszy co przy wzorcu mediatora, ale tym razem użyjemy wzorca obserwatora. Aby nieco utrudnić zadanie, zapewnijmy obsługę dowolnej liczby graczy, a nie tylko dwóch. Ponownie skorzystamy z konstruktora `Player()`, który tworzy obiekty graczy, i z obiektu `scoreboard`. Jedynie obiekt `mediator` zamieni się w obiekt `game`.

We wzorcu mediatora obiekt `mediator` wiedział o wszystkich uczestnikach gry i wywoływał ich metody. Obiekt `game` ze wzorca obserwatora nie będzie tego robił — to same obiekty będą zgłaszały chęć otrzymywania informacji o zajściu wybranych zdarzeń. Przykładowo, obiekt `scoreboard` zgłosi chęć bycia informowanym o zajściu zdarzenia `scorechange` w obiekcie `game`.

Oryginalny obiekt `publisher` należy nieco zmienić, by upodobnić go do rozwiązań znanych z przeglądarek internetowych.

- Zamiast metod `publish()`, `subscribe()` i `unsubscribe()` pojawią się metody `fire()`, `on()` i `remove()`.
- Typ zdarzenia (`type`) będzie używany cały czas, więc stanie się pierwszym parametrem wszystkich trzech funkcji.
- Dodatkowy parametr `context` przekazywany wraz z funkcją powiadomienia umożliwi wywołanie funkcji zwrotnej z odpowiednio ustawioną wartością `this`.

Nowy obiekt `publisher` ma następującą postać:

```
var publisher = {
    subscribers: {
        any: []
    },
    on: function (type, fn, context) {
        type = type || 'any';
        fn = typeof fn === "function" ? fn : context[fn];

        if (typeof this.subscribers[type] === "undefined") {
            this.subscribers[type] = [];
        }
        this.subscribers[type].push({fn: fn, context: context || this});
    },
    remove: function (type, fn, context) {
        this.visitSubscribers('unsubscribe', type, fn, context);
    },
    fire: function (type, publication) {
        this.visitSubscribers('publish', type, publication);
    },
    visitSubscribers: function (action, type, arg, context) {
        var pubtype = type || 'any',
            subscribers = this.subscribers[pubtype],
            i,
            max = subscribers ? subscribers.length : 0;

        for (i = 0; i < max; i += 1) {
            if (action === 'publish') {
                subscribers[i].fn.call(subscribers[i].context, arg);
            } else {
```

```
                if (subscribers[i].fn === arg && subscribers[i].context === context) {
                    subscribers.splice(i, 1);
                }
            }
        }
    }
};
```

Nowy konstruktor `Player()` wygląda następująco:

```
function Player(name, key) {
    this.points = 0;
    this.name = name;
    this.key = key;
    this.fire('newplayer', this);
}

Player.prototype.play = function () {
    this.points += 1;
    this.fire('play', this);
};
```

Nowym parametrem przyjmowanym przez konstruktor jest `key` — określa on klawisz na klawiaturze, który gracz będzie naciskał, by uzyskiwać punkty (we wcześniejszej wersji kodu klawisze były zapisane na sztywno). Dodatkowo utworzenie nowego obiektu gracza powoduje zgłoszenie zdarzenia `newplayer`, a każde naciśnięcie klawisza przez gracza skutkuje zgłoszeniem zdarzenia `play`.

Obiekt `scoreboard` pozostaje bez zmian — nadal aktualizuje tablicę wyników, korzystając z bieżących wartości.

Nowy obiekt `game` potrafi śledzić poczynania wszystkich graczy, by mógł zliczać wyniki i zgłaszać zdarzenie `scorechange`. Dodatkowo zgłasza się on jako subskrybent wszystkich zdarzeń `keypress` przeglądarki, by wiedzieć o wszystkich klawiszach przypisanych poszczególnym graczom.

```
var game = {

    keys: {},

    addPlayer: function (player) {
        var key = player.key.toString().charCodeAt(0);
        this.keys[key] = player;
    },

    handleKeypress: function (e) {
        e = e || window.event; //IE
        if (game.keys[e.which]) {
            game.keys[e.which].play();
        }
    },

    handlePlay: function (player) {
        var i,
            players = this.keys,
            score = {};

        for (i in players) {
            if (players.hasOwnProperty(i)) {
                score[players[i].name] = players[i].points;
            }
```

```
            }
            this.fire('scorechange', score);
        }
    };
```

Funkcja `makePublisher()`, która zamieniała dowolny obiekt w obiekt publikujący zdarzenia, jest identyczna jak w przykładzie z wydawcą i gazetą. Obiekt `game` będzie zgłaszał zdarzenia takie jak `scorechange`. Obiektem publikującym stanie się również `Player.prototype`, by możliwe było zgłaszanie zdarzeń `play` i `newplayer` wszystkim zainteresowanym.

```
makePublisher(Player.prototype);
makePublisher(game);
```

Obiekt `game` zgłasza się jako subskrybent zdarzeń `play` i `newplayer` (a także zdarzenia `keypress` przeglądarki), natomiast obiekt `scoreboard` chce być powiadamiany o zdarzeniach `scorechange`.

```
Player.prototype.on("newplayer", "addPlayer", game);
Player.prototype.on("play", "handlePlay", game);
game.on("scorechange", scoreboard.update, scoreboard);
window.onkeypress = game.handleKeypress;
```

Metoda `on()` umożliwia subskrybentom określenie funkcji zwrotnej jako referencji (`score` ↪`board.update`) lub jako tekstu (`"addPlayer"`). Wersja tekstowa działa prawidłowo tylko w przypadku przekazania jako trzeciego parametru kontekstu (na przykład `game`).

Ostatni element to dynamiczne tworzenie tylu obiektów graczy (po naciśnięciu klawiszy), ile zostanie zażądanych przez grających.

```
var playername, key;
while (1) {
    playername = prompt("Dodaj gracza (imię)");
    if (!playername) {
        break;
    }
    while (1) {
        key = prompt("Klawisz dla gracza " + playername + "?");
        if (key) {
            break;
        }
    }
    new Player(playername, key);
}
```

To już wszystko w temacie gry. Pełny kod źródłowy wraz z możliwością zagrania znajduje się pod adresem *http://www.jspatterns.com/book/7/observer-game.html*.

W implementacji wzorca mediatora obiekt `mediator` musiał wiedzieć o wszystkich obiektach, by móc w odpowiednim czasie wywoływać właściwe metody i koordynować całą grę. W nowej implementacji obiekt `game` jest nieco głupszy i wykorzystuje fakt, iż obiekty zgłaszają zdarzenia i obserwują się nawzajem (na przykład obiekt `scoreboard` nasłuchuje zdarzenia `scorechange`). Zapewnia to jeszcze luźniejsze powiązanie obiektów (im mniej z nich wie o innych, tym lepiej), choć za cenę utrudnionej analizy, kto tak naprawdę nasłuchuje kogo. W przykładowej grze wszystkie subskrypcje są na razie w jednym miejscu, ale gdyby stała się ona bardziej rozbudowana, wywołania `on()` mogłyby się znaleźć w wielu różnych miejscach (niekoniecznie w kodzie inicjalizującym). Taki kod trudniej jest testować, gdyż trudno od razu zrozumieć, co tak naprawdę się w nim dzieje. Wzorzec obserwatora zrywa ze standardowym, proceduralnym wykonywaniem kodu od początku do końca.

Podsumowanie

W rozdziale pojawiły się opisy kilku popularnych wzorców projektowych i przykłady ich implementacji w języku JavaScript. Omawiane były następujące wzorce:

- **Singleton** — tworzenie tylko jednego obiektu danej „klasy". Pojawiło się kilka rozwiązań, w tym takie, które starały się zachować składnię znaną z języka Java przez zastosowanie funkcji konstruujących. Trzeba jednak pamiętać, że z technicznego punktu widzenia w języku JavaScript wszystkie obiekty są singletonami. Nie należy też zapominać, że czasem programiści stosują słowo „singleton", a mają na myśli obiekty utworzone przy użyciu wzorca modułu.
- **Fabryka** — metoda tworząca obiekty o typie przekazanym jako wartość tekstowa.
- **Iterator** — interfejs umożliwiający łatwe przetwarzanie elementów umieszczonych w złożonej strukturze danych.
- **Dekorator** — modyfikacja obiektów w trakcie działania programu przez dodawanie do nich funkcjonalności zdefiniowanych w dekoratorach.
- **Strategia** — zachowanie tego samego interfejsu przy jednoczesnym wyborze najlepszej strategii jego implementacji (uzależnionej od kontekstu).
- **Fasada** — zapewnienie bardziej przyjaznego API przez otoczenie typowych (lub źle zaprojektowanych) metod ich nowymi wersjami.
- **Pośrednik** — otoczenie obiektu zapewniające kontrolę dostępu do niego, gdy celem jest uniknięcie wykonywania kosztownych operacji przez ich grupowanie lub opóźnianie do momentu, gdy będą naprawdę konieczne.
- **Mediator** — promowanie luźnego powiązania obiektów przez unikanie bezpośredniej komunikacji między nimi i zastąpienie jej komunikacją poprzez obiekt mediatora.
- **Obserwator** — luźne powiązanie między obiektami osiągane przez tworzenie obiektów, których zmiany można obserwować, jawnie zgłaszając się jako subskrybent (często mówi się również o własnych zdarzeniach lub wzorcu **subskrybenta-dostawcy**).

Skorowidz

F

G

H

I

J

K

Ś

T

V

W

X

Y

Z

Programowanie

O'REILLY®

JavaScript. Wzorce

Jakie jest najlepsze podejście do tworzenia aplikacji w języku JavaScript? Z tą książką, zawierającą najlepsze praktyki i wiele wzorców kodowania, znajdziesz odpowiedź na to pytanie. Jeśli jesteś doświadczonym programistą szukającym sposobów rozwiązania problemów związanych z obiektami, funkcjami, dziedziczeniem i innymi aspektami języka, przedstawione tu abstrakcje i szablony sprawdzą się idealnie.

Książka napisana przez eksperta języka JavaScript Stoyana Stefanova, starszego inżyniera Yahoo! i architekta narzędzia do optymalizacji stron WWW YSlow 2.0, zawiera wiele praktycznych wskazówek w zakresie implementacji opisywanych wzorców, a także kilka całościowych przykładów. Znajdziesz w niej również opis antywzorców, czyli podejść, które w rzeczywistości powodują więcej problemów, niż przynoszą korzyści.

Dowiedz się, jak:

- tworzyć łatwy w utrzymaniu kod,
- wybierać właściwe nazwy dla zmiennych, metod i funkcji,
- wykorzystać klasyczne wzorce programowania,
- skorzystać ze wzorców specyficznych dla środowiska przeglądarki internetowej.

Poznaj tajniki tworzenia łatwego w utrzymaniu kodu źródłowego!

helion.pl
księgarnia internetowa

Nr katalogowy: 7907

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-3821-5

Cena 39,00 zł

9788324638215