O'REILLY®

Wydanie II

Wzorce projektowe w JavaScripcie

Przewodnik dla programistów JavaScriptu i Reacta

Addy Osmani

Helion

Tytuł oryginału: Learning JavaScript Design Patterns:
A JavaScript and React Developer's Guide, 2nd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-0548-1

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/wzprj2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/wzprj2.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 7.

Wzorce projektowe w JavaScripcie

W poprzednim rozdziale przedstawiłem trzy różne kategorie wzorców projektowych. Część z nich jest ważna bądź wymagana w kontekście tworzenia aplikacji internetowych. Wymieniłem kilka ponadczasowych wzorców projektowych, które mogą okazać się użyteczne po zastosowaniu w JavaScripcie. Natomiast w tym rozdziale omówię implementacje w JavaScripcie różnych wzorców projektowych, zarówno klasycznych, jak i nowoczesnych. Poszczególnym kategoriom wzorców projektowych, czyli konstrukcyjnym, strukturalnym i operacyjnym, zostały poświęcone oddzielne punkty. Omawianie rozpocznę od kategorii konstrukcyjnych wzorców projektowych.

Wybór wzorca projektowego

Programiści dość często zastanawiają się, czy istnieje idealny wzorzec bądź zbiór wzorców, z których należy korzystać w pracy. Nie istnieje jedna dobra odpowiedź na takie pytanie. Każdy skrypt i każda aplikacja internetowa prawdopodobnie będą miały zupełnie odmienne wymagania. Trzeba rozważyć, czy dany wzorzec projektowy może zaoferować jakąkolwiek rzeczywistą wartość dla implementacji.

Na przykład w niektórych projektach korzystne okażą się zalety zastosowania wzorca Obserwator (zmniejsza on poziom wzajemnej zależności poszczególnych elementów aplikacji). Z kolei inne projekty mogą być zbyt małe, aby mogły czerpać korzyści z oddzielenia poszczególnych elementów aplikacji.

Gdy poznasz dostępne wzorce projektowe i konkretne problemy, do których rozwiązywania są one przeznaczone, to znacznie łatwiej będzie Ci zintegrować te wzorce z architekturą tworzonej aplikacji.

Konstrukcyjne wzorce projektowe

Konstrukcyjne wzorce projektowe zapewniają mechanizmy przeznaczone do tworzenia obiektów. W tym rozdziale omówię następujące wzorce z tej kategorii:

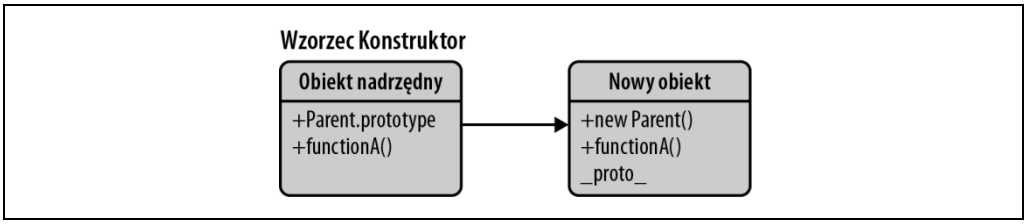
- wzorzec Konstruktor,
- wzorzec Moduł,

- wzorzec Moduł Odkrywający,
- wzorzec Singleton,
- wzorzec Prototyp,
- wzorzec Fabryka.

Wzorzec Konstruktor

Konstruktor to metoda specjalna używana do inicjalizacji nowo utworzonego obiektu tuż po zaalokowaniu dla niego pamięci. Wraz z wydaniem wersji ES2015+ do języka JavaScript została dodana składnia przeznaczona do tworzenia klas (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes*). Pozwala ona tworzyć obiekty jako egzemplarze klasy, wykorzystując do tego konstruktor domyślny (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor*).

W JavaScripcie niemalże wszystko jest obiektem, a klasy to po prostu syntaktyczny cukier dla stosowanego w JavaScripcie prototypowego podejścia do dziedziczenia. W przypadku klasycznego JavaScriptu najbardziej interesującym aspektem jest konstruktor obiektu. Omawiany tutaj wzorzec pokazałem na rysunku 7.1.



Rysunek 7.1. Wzorzec Konstruktor

Konstruktor obiektu jest używany do tworzenia określonego typu obiektów. To oznacza przygotowanie obiektu do użycia i akceptowania argumentów w celu przypisywania wartości metodom i właściwościom składowym podczas pierwszego tworzenia obiektu.

Tworzenie obiektu

W języku JavaScript mamy trzy powszechnie stosowane sposoby na utworzenie nowych obiektów:

```
// Każde z przedstawionych tutaj rozwiązań spowoduje utworzenie nowego, pustego obiektu
const newObject = {};

// Ewentualnie:
const newObject = Object.create(Object.prototype);

// Ewentualnie:
const newObject = new Object();
```

W tym przypadku każdy obiekt został zadeklarowany jako stała, co oznacza utworzenie przeznaczonej tylko do odczytu zmiennej o zasięgu bloku. W ostatnim poleceniu konstruktor `Object` tworzy obiekt opakowania dla konkretnej wartości. Jeżeli nie zostanie przekazana żadna wartość, zostanie utworzony i zwrócony pusty obiekt.

Teraz w następujące sposoby można przypisywać klucze i wartości obiektowi:

```
// Podejścia zgodne z ECMAScript 3

// 1. Składnia z użyciem kropki
// Przypisanie wartości właściwości
newObject.someKey = "Witaj, świecie!";

// Pobranie wartości właściwości
var key = newObject.someKey;

// 2. Składnia z użyciem nawiasu kwadratowego
// Przypisanie wartości właściwości
newObject["someKey"] = "Witaj, świecie!";

// Pobranie wartości właściwości
var key = newObject["someKey"];

// Podejścia zgodne jedynie z ECMAScript 5
// Więcej informacji na ten temat znajdziesz na stronie http://kangax.github.com/es5-compat-table/

// 3. Object.defineProperty
// Przypisanie wartości właściwości
Object.defineProperty( newObject, "someKey", {
    value: "Większa kontrola nad sposobem działania właściwości",
    writable: true,
    enumerable: true,
    configurable: true
});

// 4. Jeżeli masz trudność z odczytaniem przedstawionej składni,
// jej krótsza wersja może mieć następującą postać:
var defineProp = function ( obj, key, value ){
  config.value = value;
  Object.defineProperty( obj, key, config );
};

// W celu użycia zmiennej trzeba utworzyć nowy pusty obiekt "person"
var person = Object.create( null );

// Zdefiniowanie właściwości obiektu
defineProp( person, "car", "Delorean" );
defineProp( person, "dateOfBirth", "1981" );
defineProp( person, "hasBeard", false );

// 5. Object.defineProperties
// Przypisanie wartości właściwości
Object.defineProperties( newObject, {
  "someKey": {
    value: "Witaj, świecie!",
    writable: true
```

```
  },
  "anotherKey": {
    value: "Foo bar",
    writable: false
  }
});

// Pobranie właściwości dla punktów 3. i 4. może się odbyć
// za pomocą dowolnych opcji w 1. i 2.
```

Te metody można również wykorzystać na potrzeby dziedziczenia:

```
// Użyte słowa kluczowe bądź składnia ES2015+: const
// Przykłady użycia:

// Tworzenie nowego obiektu driver, który dziedziczy po obiekcie person
const driver = Object.create(person);

// Zdefiniowanie właściwości dla obiektu driver
defineProp(driver, 'topSpeed', '100mph');

// Pobranie wartości właściwości dziedziczonej (1981)
console.log(driver.dateOfBirth);

// Pobranie wartości właściwości zdefiniowanej w nowym obiekcie (100 mph)
console.log(driver.topSpeed);
```

Prosty konstruktor

Jak już wspomniałem w rozdziale 5., klasy JavaScriptu zostały wprowadzone w wydaniu ES2015. To pozwala definiować szablony obiektów JavaScriptu oraz implementować hermetyzację i dziedziczenie (*https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Classes_in_JavaScript#inheritance_with_class_syntax*) w JavaScripcie.

Warto w tym miejscu przypomnieć, że klasa musi zawierać i deklarować metodę o nazwie `constructor()`, która będzie używana do tworzenia nowego obiektu. Słowo kluczowe `new` umożliwia wywołanie konstruktora. Z kolei słowo kluczowe `this` w konstruktorze odwołuje się do nowo utworzonego obiektu. Oto przykład użycia prostego konstruktora:

```
class Car {
    constructor(model, year, miles) {
        this.model = model;
        this.year = year;
        this.miles = miles;
    }

    toString() {
        return `${this.model} ma przebieg ${this.miles} mil.`;
    }
}

// Przykład użycia:
// Można utworzyć nowe egzemplarze klasy Car
let civic = new Car('Honda Civic', 2009, 20000);
let mondeo = new Car('Ford Mondeo', 2010, 5000);
```

```
// Następnie wystarczy przejść do konsoli przeglądarki WWW, aby wyświetlić
// dane wyjściowe metody toString() wywoływanej w tych obiektach
console.log(civic.toString());
console.log(mondeo.toString());
```

To jest prosty przykład użycia wzorca Konstruktor, choć trzeba dodać, że wiążą się z nim pewne problemy. Jednym z nich jest trudna obsługa dziedziczenia. Inny problem polega na tym, że funkcje takie jak `toString()` są ponownie definiowane dla każdego nowego obiektu, który jest tworzony za pomocą konstruktora klasy `Car`. Takie rozwiązanie nie będzie więc optymalne, ponieważ egzemplarze typu `Car` powinny idealnie współdzielić te same funkcje.

Konstruktory z prototypami

Prototypy w JavaScripcie pozwalają na łatwe definiowanie metod dla wszystkich egzemplarzy określonego obiektu, niezależnie od tego, czy mamy do czynienia z funkcją, czy klasą. Kiedy wywoływany jest konstruktor JavaScriptu w celu utworzenia obiektu, wszystkie właściwości prototypu konstruktora zostają udostępnione nowemu obiektowi. Dzięki temu można mieć wiele obiektów `Car` uzyskujących dostęp do tego samego prototypu. Spójrz na rozszerzoną wersję pierwotnego przykładu:

```
class Car {
    constructor(model, year, miles) {
        this.model = model;
        this.year = year;
        this.miles = miles;
    }
}

// Zwróć tutaj uwagę na użycie składni Object.prototype.newMethod zamiast
// Object.prototype, aby w ten sposób uniknąć ponownego definiowania prototypu obiektu.
// Nadal można używać składni Object.prototype podczas dodawania nowych metod,
// ponieważ wewnętrznie korzystamy wówczas z tej samej struktury.
Car.prototype.toString = function() {
    return `${this.model} ma przebieg ${this.miles} mil.`;
};

// Przykład użycia:
let civic = new Car('Honda Civic', 2009, 20000);
let mondeo = new Car('Ford Mondeo', 2010, 5000);

console.log(civic.toString());
console.log(mondeo.toString());
```

Wszystkie obiekty klasy `Car` współdzielą teraz pojedynczy egzemplarz metody `toString()`.

Wzorzec Moduł

Moduły to integralne części każdej solidnej architektury aplikacji. Zwykle umożliwiają zastosowanie przejrzystej separacji i dobrej organizacji jednostek kodu projektu.

Klasyczny JavaScript oferuje wiele opcji w zakresie implementacji modułów:

- notacja literału obiektu,
- wzorzec Moduł,
- moduły AMD,
- moduły CommonJS.

W rozdziale 5. już wspomniałem o nowoczesnych modułach JavaScriptu. W przykładach zamieszczonych w tym rozdziale skoncentruję się przede wszystkim na modułach ES.

Przed wersją ES2015 popularnymi alternatywami były moduły CommonJS i AMD, ponieważ pozwalały wyeksportować zawartość modułu. Dokładniejsze omówienie tematu eksportu modułów AMD, CommonJS i UMD znajdziesz w dalszej części książki, a dokładnie w rozdziale 10. Trzeba zacząć od poznania wzorca Moduł i jego korzeni.

Wzorzec Moduł bazuje częściowo na literałach obiektu, więc rozsądne wydaje się przypomnienie wiadomości na ten temat.

Literał obiektu

W notacji literału obiektu obiekt jest opisywany jako zbiór rozdzielonych przecinkami par nazwa – wartość umieszczonych w nawiasie klamrowym. Nazwą w obiekcie może być ciąg tekstowy bądź identyfikator zakończony dwukropkiem. Najlepiej jest nie używać przecinka po ostatniej parze w obiekcie, ponieważ to może doprowadzić do błędów.

```
const myObjectLiteral = {
    variableKey: variableValue,
    functionKey() {
        // ...
    }
};
```

Literał obiektu nie wymaga inicjalizacji za pomocą operatora `new`. Nie powinien być używany na początku polecenia, ponieważ otwierający nawias klamrowy, {, może być zinterpretowany jako początek nowego bloku. Spoza obiektu nowe elementy składowe mogą być dodawane za pomocą przypisania w postaci `nazwaModułu.property = "dowolna wartość";`.

Oto pełny przykład modułu zdefiniowanego za pomocą notacji literału obiektu:

```
const myModule = {
    myProperty: 'someValue',
    // Literał obiektu może zawierać właściwości i metody.
    // Na przykład można zdefiniować obiekt przeznaczony na potrzeby konfiguracji modułu.
    myConfig: {
        useCaching: true,
        language: 'en',
    },
    // Oto bardzo prosta metoda
    saySomething() {
        console.log('Co Paul Irish będzie dzisiaj debugował?');
    },
```

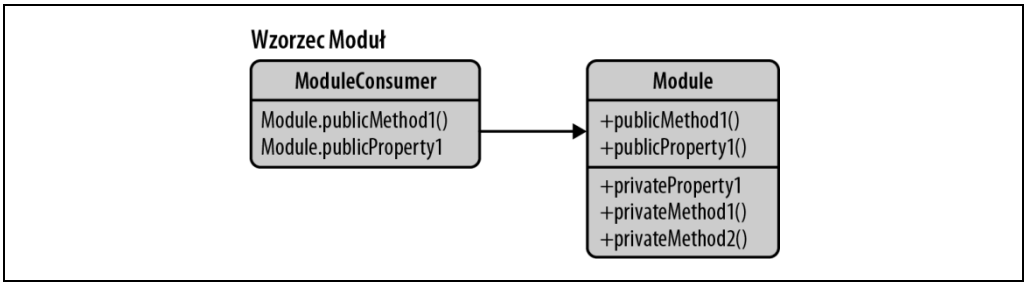
```
    // Wyświetlenie wartości na podstawie bieżącej konfiguracji
    reportMyConfig() {
        console.log(
            `Buforowanie jest: ${this.myConfig.useCaching ? 'włączone' : 'wyłączone'}`
        );
    },
    // Pominięcie bieżącej konfiguracji
    updateMyConfig(newConfig) {
        if (typeof newConfig === 'object') {
            this.myConfig = newConfig;
            console.log(this.myConfig.language);
        }
    },
};

// Dane wyjściowe: Co Paul Irish będzie dzisiaj debugował?
myModule.saySomething();

// Dane wyjściowe: Buforowanie jest: włączone
myModule.reportMyConfig();

// Dane wyjściowe: fr
myModule.updateMyConfig({
    language: 'fr',
    useCaching: false,
});

// Dane wyjściowe: Buforowanie jest: wyłączone
myModule.reportMyConfig();
```

Używanie literału obiektu to sposób na hermetyzację i organizację kodu. Rebecca Murphey dość wnikliwie opisała to zagadnienie (*https://github.com/rmurphey/rmurphey/blob/master/public/blog/using-objects-to-organize-your-code.md*) — warto przeczytać jej publikację, aby jeszcze dokładniej zapoznać się z tematem literałów obiektów.

Wzorzec Moduł

Wzorzec Moduł został opracowany w celu zapewnienia prywatnej i publicznej hermetyzacji dla klas w dziedzinie konwencjonalnego tworzenia oprogramowania.

W pewnym momencie organizowanie pisanych w JavaScripcie aplikacji o każdej rozsądnej wielkości było wyzwaniem. Programiści polegali na oddzielnych skryptach, dzieląc w ten sposób aplikację na rozsądnej wielkości fragmenty logiki. Niczym nadzwyczajnym nie było ręczne importowanie w pliku HTML 10 czy 20 skryptów, aby w ten sposób zapewnić organizację aplikacji. Korzystając z obiektów, wzorzec Moduł był po prostu jednym ze sposobów na hermetyzację logiki w pliku zawierającym metody zarówno publiczne, jak i „prywatne”. Na przestrzeni czasu pojawiło się wiele niestandardowych systemów modułów, które ułatwiały to zadanie. Obecnie programiści mogą używać modułów JavaScriptu do organizowania obiektów, funkcji, klas lub zmiennych w sposób pozwalający na łatwy eksport lub import w innych plikach. To pomaga unikać konfliktów między nazwami klas lub funkcji znajdującymi się w różnych metodach. Na rysunku 7.2 w sposób graficzny pokazałem wzorzec Moduł.



Rysunek 7.2. Wzorzec Moduł

Prywatność

Wzorzec Moduł hermetyzuje stan „prywatności" i organizację za pomocą domknięć. Umożliwia opakowanie grupy zmiennych oraz metod publicznych i prywatnych, chroni elementy przed ich udostępnieniem w przestrzeni globalnej i przed przypadkową kolizją z interfejsem innego programisty. Dzięki temu wzorcowi udostępnia się jedynie API publiczne, wszystko zaś pozostaje prywatne w domknięciu.

Dzięki temu otrzymujemy czyste rozwiązanie, w którym logika ochronna wykonuje najtrudniejsze zadania, udostępniając jedynie interfejs, który ma być używany przez inne fragmenty aplikacji. Ten wzorzec używa natychmiast wywoływanego wyrażenia funkcji (ang. *immediately invoked function expression*, IIFE; *https://benalman.com/news/2010/11/immediately-invoked-function-expression/*), w którym następuje zwrócenie obiektu. Więcej informacji na temat IIFE znajdziesz w rozdziale 11.

Zwróć uwagę na brak prawdziwego znaczenia „prywatności" w JavaScripcie, ponieważ w przeciwieństwie do innych języków programowania nie mamy tutaj modyfikatorów dostępu. Formalnie rzecz biorąc, zmiennej nie można zadeklarować jako publicznej lub prywatnej, więc do symulowania tej koncepcji jest używany zasięg funkcji. We wzorcu Moduł zadeklarowane zmienne lub metody są dostępne jedynie wewnątrz danego modułu, co ma miejsce dzięki użyciu domknięcia. Natomiast zmienne lub metody zdefiniowane w zwracanym obiekcie są dostępne dla każdego.

Obejście umożliwiające implementację prywatności zmiennych w zwracanych obiektach używa wywołania `WeakMap()` (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap*), którego dokładne omówienie znajdziesz w dalszej części rozdziału. To wywołanie pobiera klucze jedynie w postaci obiektów i nie zezwala na iterację. Dlatego jedynym sposobem na uzyskanie dostępu do obiektu w module jest użycie odwołania. Na zewnątrz modułu dostęp może odbywać się jedynie za pomocą metody publicznej zdefiniowanej w module. W ten sposób zapewniona jest prywatność obiektu.

Historia

Z perspektywy historycznej wzorzec Moduł został początkowo opracowany w 2003 roku przez kilka osób, w tym Richarda Cornforda (*https://groups.google.com/g/comp.lang.javascript/c/eTzWVa1W_pE/m/N9lnvRG9WJ8J*). Później Douglas Crockford spopularyzował ją w swoich pracach. Za ciekawostkę można uznać również to, że wybrane jego funkcje mogą wydawać się

znajome osobom, które kiedykolwiek miały styczność z biblioteką *YUI* firmy Yahoo!. Powód jest prosty: wzorzec Moduł miał ogromny wpływ na YUI podczas tworzenia jej komponentów.

Przykłady

Rozpocznę od przedstawienia implementacji wzorca Moduł na przykładzie tworzenia samodzielnego modułu. W tej implementacji zostaną użyte słowa kluczowe `import` i `export`. Dla przypomnienia dodam, że słowo kluczowe `export` zapewnia na zewnątrz modułu dostęp do funkcjonalności zdefiniowanej w tym module. Natomiast słowo kluczowe `import` pozwala importować w skrypcie elementy wyeksportowane przez moduł.

```
let counter = 0;

const testModule = {
  incrementCounter() {
    return counter++;
  },
  resetCounter() {
    console.log(`Wartość licznika przed wyzerowaniem: ${counter}`);
    counter = 0;
  },
};

// Domyślnie eksportowany moduł, bez podawania nazwy
export default testModule;

// Przykład użycia:
// Importowanie modułu z podanej ścieżki dostępu
import testModule from './testModule';

// Inkrementacja wartości licznika
testModule.incrementCounter();

// Sprawdzenie wartości licznika i jej wyzerowanie
// Dane wyjściowe: Wartość licznika przed wyzerowaniem: 1
testModule.resetCounter();
```

W tym przykładzie pozostałe fragmenty kodu nie mogą bezpośrednio odczytywać wartości wywołania `incrementCounter()` lub `resetCounter()`. Zmienna `counter` jest w zasadzie ukryta przez zasięgiem globalnym, więc działa podobnie jak zmienna prywatna — jej istnienie ogranicza się do domknięcia modułu i tylko te dwie funkcje kodu są w stanie uzyskać dostęp do jej zasięgu. Nasze metody stosują przestrzenie nazw, więc w testowej części kodu konieczne jest stosowanie prefiksu przed wszelkimi wywołaniami nazwy modułu (np. `testModule`).

Podczas pracy ze wzorcem Moduł użyteczne może okazać się przygotowanie prostego szablonu, który będzie stosowany w charakterze punktu wyjścia. Oto przykład takiego szablonu, zapewniającego obsługę przestrzeni nazw oraz zmiennych publicznych i prywatnych:

```
// Prywatna zmienna licznika
let myPrivateVar = 0;

// Funkcja prywatna, która rejestruje wszelkie argumenty
const myPrivateMethod = foo => {
```

```
  console.log(foo);
};

const myNamespace = {
  // Zmienna publiczna
  myPublicVar: 'foo',

  // Funkcja publiczna używająca elementów prywatnych
  myPublicFunction(bar) {
    // Inkrementacja wartości licznika prywatnego
    myPrivateVar++;

    // Wywołanie metody prywatnej za pomocą bar
    myPrivateMethod(bar);
  },
};

export default myNamespace;
```

W kolejnym fragmencie kodu zamieściłem przykładową implementację koszyka na zakupy przygotowaną z wykorzystaniem tego wzorca. Sam moduł znajduje się w zmiennej globalnej o nazwie `basketModule`. Tablica `basket` w module jest prywatna, więc inne fragmenty aplikacji nie mogą bezpośrednio jej odczytywać. Tablica istnieje jedynie w domknięciu modułu. Dlatego też jedynymi metodami, które mogą uzyskać do niej dostęp, są metody posiadające dostęp do zasięgu modułu, np. `addItem()`, `getItem()`.

```
// Elementy prywatne
const basket = [];

const doSomethingPrivate = () => {
  //...
};

const doSomethingElsePrivate = () => {
  //...
};

// Utworzenie obiektu, który będzie dostępny publicznie
const basketModule = {
  // Dodanie elementów do koszyka na zakupy
  addItem(values) {
    basket.push(values);
  },

  // Pobranie liczby elementów znajdujących się w koszyku na zakupy
  getItemCount() {
    return basket.length;
  },

  // Publiczny alias do funkcji prywatnej
  doSomething() {
    doSomethingPrivate();
  },

  // Pobranie wartości całkowitej elementów znajdujących się w koszyku na zakupy
  // Metoda reduce() stosuje funkcję dla akumulatora i poszczególnych elementów
  // tablicy (od lewej do prawej) w celu jej zredukowania do pojedynczej wartości.
```

```
  getTotal() {
    return basket.reduce((currentSum, item) => item.price + currentSum, 0);
  },
};

export default basketModule;
```

Jak można było zauważyć, w module jest zwracany obiekt. Jest on automatycznie przypisywany zmiennej `basketModule` i można go używać w następujący sposób:

```
// Importowanie modułu z podanej ścieżki dostępu
import basketModule from './basketModule';

// Moduł basketModule zwraca obiekt razem z publicznym API, którego można używać
basketModule.addItem({
  item: 'chleb',
  price: 0.5,
});

basketModule.addItem({
  item: 'masło',
  price: 0.3,
});

// Dane wyjściowe: 2
console.log(basketModule.getItemCount());

// Dane wyjściowe: 0.8
console.log(basketModule.getTotal());

// Jednak przedstawione tutaj podejście nie działa:

// Dane wyjściowe: undefined
// Tak się dzieje, ponieważ koszyk na zakupy sam w sobie
// nie został udostępniony w postaci publicznego API.
console.log(basketModule.basket);

// To również nie zadziała, ponieważ istnieje jedynie w zasięgu domknięcia
// basketModule, a nie w zwróconym publicznie dostępnym obiekcie.
console.log(basket);
```

Te metody działają w przestrzeni nazw `basketModule`. Wszystkie funkcje zostały opakowane wymienionym modułem, co zapewniło wiele korzyści, m.in.:

- Swobodę w posiadaniu funkcji prywatnych, które mogą być używane jedynie przez moduł. Nie będą udostępnione pozostałej części strony (będą jedynie wyeksportowane API), więc można je faktycznie uznać za prawdziwe.
- Biorąc pod uwagę fakt, że te funkcje są zwykle zadeklarowane i nazwane, znacznie łatwiejsze może być pokazanie stosu wywołań w debugerze podczas próby ustalenia, która funkcja (lub funkcje) zgłosiła wyjątek.

Warianty wzorca Moduł

Na przestrzeni czasu projektanci wprowadzili różne warianty wzorca Moduł dopasowane do ich potrzeb.

Importowanie domieszki

Ten wariant wzorca pokazuje, w jaki sposób można przekazywać elementy globalne (np. funkcje narzędziowe lub biblioteki zewnętrzne) jako argumenty dla funkcji wyższego rzędu w module. W praktyce to pozwala na ich importowanie i stosowanie dla nich lokalnego aliasu, wedle potrzeb.

```
// W pliku utils.js
export const min = (arr) => Math.min(...arr);

// W pliku privateMethods.js
import { min } from "./utils";

export const privateMethod = () => {
  console.log(min([10, 5, 100, 2, 1000]));
};

// W pliku myModule.js
import { privateMethod } from "./privateMethods";

const myModule = () => ({
  publicMethod() {
    privateMethod();
  },
});

export default myModule;

// W pliku main.js
import myModule from "./myModule";

const moduleInstance = myModule();
moduleInstance.publicMethod();
```

Eksportowanie

Następny wariant pozwala zadeklarować elementy globalne bez ich używania. W podobny sposób ten wariant mógłby obsługiwać przedstawioną w poprzednim podpunkcie koncepcję globalnego importowania.

```
// W pliku module.js
const privateVariable = "Witaj, świecie!";

const privateMethod = () => {
  // ...
};

const module = {
  publicProperty: "Foobar",
  publicMethod: () => {
    console.log(privateVariable);
  },
};

export default module;
```

Zalety

Wcześniej wyjaśniłem, dlaczego wzorzec Konstruktor może być użyteczny. Zastanawiasz się, dlaczego wzorzec Moduł może być dobrym wyborem? Przede wszystkim pozwala programistom posiadającym doświadczenie w stosowaniu stylu programowania zorientowanego obiektowo na łatwiejszą pracę niż w przypadku idei prawdziwej hermetyzacji, przynajmniej z perspektywy JavaScriptu. Dzięki zaimportowaniu domieszek programista może zarządzać zależnościami między modułami oraz przekazywać elementy globalne wedle potrzeb. W ten sposób kod stanie się modułowy i łatwiejszy w późniejszej obsłudze technicznej.

Ponadto zapewniona jest obsługa danych prywatnych — we wzorcu Moduł mamy dostęp jedynie do wartości, które zostały wyraźnie wyeksportowane za pomocą słowa kluczowego `export`. Z kolei wartości niewyeksportowane pozostają prywatne i dostępne tylko w danym module. To minimalizuje ryzyko przypadkowego zaśmiecenia przestrzeni globalnej. Nie trzeba się obawiać przypadkowego nadpisania wartości utworzonych przez programistów używających modułu, które mogły mieć taką samą nazwę jak wartość prywatna: wzorzec chroni przed kolizjami nazw i zaśmiecaniem przestrzeni globalnej.

Dzięki wzorcowi Moduł można hermetyzować fragmenty kodu, aby nie były dostępne publicznie. Mogą one w znacznie mniej ryzykowny sposób działać z wieloma zależnościami i przestrzeniami nazw. Warto zwrócić uwagę, że transpilator taki jak Babel jest potrzebny do zapewnienia możliwości użycia modułów ES2015 we wszystkich środowiskach uruchomieniowych JavaScriptu.

Wady

Wadą wzorca Moduł jest to, że dostęp do publicznych i prywatnych elementów składowych odbywa się odmiennie. Kiedy chcesz zmienić widoczność, musisz wprowadzić zmiany w każdym miejscu, w którym jest używany element składowy.

Ponadto nie można uzyskać dostępu do prywatnych elementów składowych dodanych później do obiektu. Dlatego w wielu przypadkach wzorzec Moduł wciąż okazuje się użyteczny i jeśli zostanie zastosowany poprawnie, zdecydowanie ma potencjał, aby usprawnić strukturę aplikacji.

Inną wadą jest brak możliwości tworzenia zautomatyzowanych testów jednostkowych dla prywatnych elementów składowych oraz większy poziom złożoności, gdy wykryte błędy wymagają natychmiastowego poprawienia. Po prostu nie ma możliwości poprawiania prywatnych elementów składowych. Zamiast tego konieczne jest nadpisanie wszystkich metod publicznych współdziałających z zawierającymi błędy prywatnymi elementami składowymi. Programiści również nie mogą łatwo rozbudowywać prywatnych elementów składowych, więc warto pamiętać, że nie są one tak elastyczne, jak może się początkowo wydawać.

Jeżeli chcesz dowiedzieć się więcej na temat wzorca Moduł, zajrzyj do doskonałego artykułu, w którym Bena Cherry dokładnie przedstawia ten wzorzec (*http://www.adequatelygood.com/JavaScript-Module-↪Pattern-In-Depth.html*).

Nowoczesny wzorzec Moduł i obiekt WeakMap

Wprowadzony w wydaniu ES6 języka JavaScript obiekt `WeakMap` (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap*) składa się z kolekcji par klucz – wartość, w których odwołania do kluczy są słabe. Klucz musi być obiektem, wartość zaś może być dowolna. Obiekt `WeakMap` to w zasadzie mapa, w której klucze są słabo przechowywane. To oznacza, że klucze mogą podlegać działaniu mechanizmu usuwania nieużytków (ang. *garbage collection*, GC), jeżeli nie ma aktywnego odwołania do obiektu. Na listingach 7.1, 7.2 i 7.3 zamieściłem przykład implementacji wzorca Moduł, w której został użyty obiekt `WeakMap`.

Listing 7.1. Definicja prostego modułu

```
let _counter = new WeakMap();

class Module {
    constructor() {
        _counter.set(this, 0);
    }
    incrementCounter() {
        let counter = _counter.get(this);
        counter++;
        _counter.set(this, counter);

        return _counter.get(this);
    }
    resetCounter() {
        console.log(`Wartość licznika przed wyzerowaniem: ${_counter.get(this)}`);
        _counter.set(this, 0);
    }
}

const testModule = new Module();

// Przykład użycia:

// Inkrementacja wartości licznika
testModule.incrementCounter();
// Sprawdzenie wartości licznika i jej wyzerowanie
// Dane wyjściowe: Wartość licznika przed wyzerowaniem: 1
testModule.resetCounter();
```

Listing 7.2. Przestrzenie nazw razem ze zmiennymi publicznymi i prywatnymi

```
const myPrivateVar = new WeakMap();
const myPrivateMethod = new WeakMap();

class MyNamespace {
    constructor() {
        // Prywatna zmienna licznika
        myPrivateVar.set(this, 0);
        // Funkcja prywatna, która rejestruje wszelkie argumenty
        myPrivateMethod.set(this, foo => console.log(foo));
        // Zmienna publiczna
        this.myPublicVar = 'foo';
    }
```

```
        // Funkcja publiczna używająca elementów prywatnych
        myPublicFunction(bar) {
            let privateVar = myPrivateVar.get(this);
            const privateMethod = myPrivateMethod.get(this);
            // Inkrementacja wartości licznika prywatnego
            privateVar++;
            myPrivateVar.set(this, privateVar);
            // Wywołanie metody prywatnej za pomocą bar
            privateMethod(bar);
        }
    }
```

Listing 7.3. Implementacja koszyka na zakupy

```
    const basket = new WeakMap();
    const doSomethingPrivate = new WeakMap();
    const doSomethingElsePrivate = new WeakMap();

    class BasketModule {
        constructor() {
            // Elementy prywatne
            basket.set(this, []);
            doSomethingPrivate.set(this, () => {
                //...
            });
            doSomethingElsePrivate.set(this, () => {
                //...
            });
        }
        // Publiczne aliasy do funkcji prywatnej
        doSomething() {
            doSomethingPrivate.get(this)();
        }
        doSomethingElse() {
            doSomethingElsePrivate.get(this)();
        }
        // Dodanie elementów do koszyka na zakupy
        addItem(values) {
            const basketData = basket.get(this);
            basketData.push(values);
            basket.set(this, basketData);
        }
        // Pobranie liczby elementów znajdujących się w koszyku na zakupy
        getItemCount() {
            return basket.get(this).length;
        }
        // Pobranie wartości całkowitej elementów znajdujących się w koszyku na zakupy
        getTotal() {
            return basket
                .get(this)
                .reduce((currentSum, item) => item.price + currentSum, 0);
        }
    }
```

Moduły we współpracy z nowoczesnymi bibliotekami

Wzorzec Moduł można wykorzystać podczas budowania aplikacji z użyciem bibliotek JavaScriptu takich jak React. Załóżmy, że masz ogromną liczbę komponentów niestandardowych, które zostały utworzone przez Twój zespół. W takim przypadku poszczególne komponenty można umieścić w oddzielnych plikach, praktycznie tworząc moduł dla każdego komponentu. W kolejnym fragmencie kodu przedstawiłem dostosowany do własnych potrzeb komponent przycisku na podstawie komponentu przycisku *material-ui* (*https://mui.com/core/*) i wyeksportowany jako moduł:

```
import React from "react";
import Button from "@material-ui/core/Button";

const style = {
  root: {
    borderRadius: 3,
    border: 0,
    color: "white",
    margin: "0 20px"
  },
  primary: {
    background: "linear-gradient(45deg, #FE6B8B 30%, #FF8E53 90%)"
  },
  secondary: {
    background: "linear-gradient(45deg, #2196f3 30%, #21cbf3 90%)"
  }
};

export default function CustomButton(props) {
  return (
    <Button {...props} style={{ ...style.root, ...style[props.color] }}>
      {props.children}
    </Button>
  );
}
```

Wzorzec Moduł Odkrywający

Skoro wiesz już więcej na temat wzorca Moduł, możesz poznać jego nieco bardziej usprawnioną wersję, czyli opracowany przez Christiana Heilmanna wzorzec Moduł Odkrywający.

Wzorzec Moduł Odkrywający został opracowany przez Heilmanna na skutek frustracji, która mu towarzyszyła, gdy musiał powtarzać nazwę obiektu głównego podczas wywoływania metody publicznej z poziomu innej bądź w trakcie uzyskiwania dostępu do zmiennych publicznych. Nie podobała mu się również notacja literału obiektu dla elementów, które miały być dostępne publicznie.

Poczynione przez niego wysiłki doprowadziły do uaktualnienia wzorca, za pomocą którego można po prostu zdefiniować wszystkie funkcje i zmienne w zasięgu prywatnym oraz zwrócić obiekt anonimowy razem ze wskaźnikami prowadzącymi do funkcjonalności prywatnej przeznaczonej do udostępnienia jako publiczna.

Dzięki nowoczesnym sposobom implementacji modułów (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules*) w wydaniach ES2015+ JavaScriptu zasięg funkcji i zmiennych zdefiniowanych w module jest już prywatny. Ponadto można wykorzystać słowa kluczowe `export` i `import`, gdy zachodzi potrzeba udostępnienia któregokolwiek elementu.

Oto przykład użycia wzorca Moduł Odkrywający zaimplementowanego w kodzie zgodnym z wydaniami ES2015+:

```
let privateVar = 'Rob Dodson';
const publicVar = 'Witaj!';

const privateFunction = () => {
  console.log(`Imię:${privateVar}`);
};

const publicSetName = strName => {
  privateVar = strName;
};

const publicGetName = () => {
  privateFunction();
};

// Udostępnienie wskaźników publicznych do
// prywatnych funkcji i właściwości
const myRevealingModule = {
  setName: publicSetName,
  greeting: publicVar,
  getName: publicGetName,
};

export default myRevealingModule;

// Przykład użycia:
import myRevealingModule from './myRevealingModule';
myRevealingModule.setName('Matt Gaunt');
```

W tym przykładzie zmienna prywatna `privateVar` została udostępniona za pomocą publicznych metod typu getter i setter, `publicSetName()` i `publicGetName()`.

Ten wzorzec można również wykorzystać w celu udostępniania prywatnych funkcji i właściwości o określonym schemacie nazw:

```
let privateCounter = 0;

const privateFunction = () => {
    privateCounter++;
}

const publicFunction = () => {
    publicIncrement();
}

const publicIncrement = () => {
    privateFunction();
}
```

```
const publicGetCount = () => privateCounter;

// Udostępnienie wskaźników publicznych do
// prywatnych funkcji i właściwości
const myRevealingModule = {
    start: publicFunction,
    increment: publicIncrement,
    count: publicGetCount
};

export default myRevealingModule;

// Przykład użycia:
import myRevealingModule from './myRevealingModule';

myRevealingModule.start();
```

Zalety

Dzięki temu wzorcowi składnia skryptów może być znacznie spójniejsza. Ponadto bardzo ułatwia zrozumienie na końcu modułu, które funkcje i zmienne mogą być dostępne publicznie, co z kolei przekłada się na większą czytelność.

Wady

Wadą tego wzorca jest to, że jeśli funkcja prywatna odwołuje się do funkcji publicznej, wówczas funkcja publiczna nie będzie mogła zostać nadpisana, gdy zachodzi konieczność zainstalowania poprawki. To wynika z tego, że funkcja prywatna będzie kontynuowała odwoływanie się do implementacji prywatnej, wzorzec zaś nie ma zastosowania dla publicznych elementów składowych, lecz jedynie do funkcji.

Reguła braku możliwości instalowania poprawek odnosi się również do publicznych elementów składowych obiektu odwołujących się do zmiennych prywatnych.

W efekcie moduły utworzone za pomocą wzorca Moduł Odkrywający mogą być znacznie bardziej zawodne niż moduły opracowane z wykorzystaniem oryginalnego wzorca Moduł. Dlatego używając wzorca Moduł Odkrywający, należy zachować ostrożność.

Wzorzec Singleton

Singleton to wzorzec projektowy ograniczający do jednego liczbę utworzonych egzemplarzy danej klasy. Takie rozwiązanie jest użyteczne, gdy wymagany jest dokładnie jeden obiekt w celu koordynacji działań w systemie. Klasycznie implementacja wzorca Singleton odbywa się przez zdefiniowanie klasy razem z metodą tworzącą nowy egzemplarz klasy tylko wtedy, gdy taki egzemplarz jeszcze nie istnieje. Natomiast w przypadku istnienia egzemplarza wartością zwrotną metody będzie odniesienie do tego egzemplarza.

Singleton różni się od klas statycznych (lub obiektów) możliwością opóźnienia inicjalizacji, ponieważ wymagane są informacje, które mogą być niedostępne podczas inicjalizacji. Fragment kodu nieposiadający informacji na temat poprzedniego odwołania do klasy singleton nie może go łatwo

otrzymać. To wynika z tego, że nie jest obiektem ani „klasą” zwracaną przez singleton, lecz jest strukturą. Zastanów się nad tym, jak zmienne domknięcia nie są w rzeczywistości domknięciem — zasięg funkcji zapewniającej domknięcie jest tym domknięciem.

ES2015+ pozwala na implementację wzorca Singleton w celu utworzenia globalnego egzemplarza klasy JavaScript, który będzie tworzony tylko raz. W celu udostępnienia egzemplarza singleton można skorzystać z możliwości eksportu modułu. Dzięki temu dostęp jest jawniejszy, kontrolowany i odmienny niż w przypadku innych zmiennych globalnych. Nie można utworzyć nowego egzemplarza obiektu, natomiast można odczytywać i modyfikować egzemplarz, używając do tego publicznych metod typu setter i getter zdefiniowanych w klasie.

Oto przykład implementacji wzorca Singleton:

```
// Egzemplarz przechowuje odwołanie do obiektu klasy typu singleton
let instance;

// Prywatne metody i zmienne
const privateMethod = () => {
    console.log('To jest element prywatny.');
  };
const privateVariable = 'To również jest element prywatny.';
const randomNumber = Math.random();

// Singleton
class MySingleton {
  // Pobranie egzemplarza singleton, o ile istnieje
  // Ewentualnie: utworzenie egzemplarza, jeśli jeszcze nie istnieje
  constructor() {
    if (!instance) {
      // Właściwość publiczna
      this.publicProperty = 'To również jest element publiczny.';
      instance = this;
    }

    return instance;
  }

  // Metody publiczne
  publicMethod() {
    console.log('Metoda publicza może mnie zobaczyć!');
  }

  getRandomNumber() {
    return randomNumber;
  }
}
// [ES2015+] Domyślnie eksportowany moduł, bez podawania nazwy
export default MySingleton;

// Egzemplarz przechowuje odwołanie do obiektu klasy typu singleton
let instance;

// Singleton
class MyBadSingleton {
    // Zawsze następuje utworzenie nowego egzemplarza typu singleton
```

```
    constructor() {
        this.randomNumber = Math.random();
        instance = this;
        return instance;
    }

    getRandomNumber() {
        return this.randomNumber;
    }
}

export default MyBadSingleton;

// Przykład użycia:
import MySingleton from './MySingleton';
import MyBadSingleton from './MyBadSingleton';

const singleA = new MySingleton();
const singleB = new MySingleton();
console.log(singleA.getRandomNumber() === singleB.getRandomNumber());
// Wartość true

const badSingleA = new MyBadSingleton();
const badSingleB = new MyBadSingleton();
console.log(badSingleA.getRandomNumber() !== badSingleB.getRandomNumber());
// Wartość true

// Uwaga: w tym przykładzie są używane losowo wybrane liczby, więc istnieje matematyczne
// prawdopodobieństwo, choć znikome, że obie liczby będą takie same.
// Pomijając to, przedstawiony przykład powinien być jak najbardziej poprawny.
```

Cechą obiektu typu singleton jest dostęp globalny do egzemplarza. We wspomnianej już kilkukrotnie książce napisanej przez Bandę Czterech *możliwość zastosowania* wzorca singleton została przedstawiona następująco:

- Musi istnieć dokładnie jeden egzemplarz klasy i musi być on dostępny dla klientów z doskonale znanym punktem dostępu.
- Powinna istnieć możliwość rozszerzenia pojedynczego egzemplarza za pomocą podklas, a klienty powinny mieć możliwość pracy z rozbudowanym egzemplarzem bez konieczności modyfikowania ich kodu.

Drugi z wymienionych punktów odwołuje się do sytuacji, w której może istnieć fragment kodu podobny do następującego:

```
constructor() {
    if (this._instance == null) {
        if (isFoo()) {
            this._instance = new FooSingleton();
        } else {
            this._instance = new BasicSingleton();
        }
    }

    return this._instance;
}
```

W tym przypadku metoda `constructor()` przypomina metodę wzorca Fabryka. Nie ma konieczności uaktualniania każdego punktu w kodzie, aby można było uzyskać do niego dostęp. W omawianym przykładzie `FooSingleton` będzie podklasą klasy `BasicSingleton` i implementuje dokładnie ten sam interfejs.

Dlaczego opóźnione wykonywanie jest uznawane za ważną cechę wzorca Singleton? W języku C++ służy on jako ochrona przed nieprzewidywalnością dynamicznej kolejności inicjalizacji, zwracając kontrolę programiście.

Trzeba koniecznie zwrócić uwagę na różnice między statycznym egzemplarzem klasy (obiektu) i egzemplarzem typu singleton. Wprawdzie istnieje możliwość implementacji singletona jako egzemplarza statycznego, ale może zostać on utworzony także w celu wczytywania z opóźnieniem, bez konieczności używania jakichkolwiek zasobów, np. pamięci, dopóki to nie będzie konieczne.

Przypuśćmy, że istnieje obiekt statyczny, który można zainicjalizować bezpośrednio. W takim przypadku trzeba się upewnić, że kod zawsze będzie wykonywany w tej samej kolejności (tutaj obiekt `objCar` wymaga podczas inicjalizacji obiektu `objWheel`). To nie będzie się skalowało, gdy masz ogromną liczbę plików kodu źródłowego.

Wprawdzie zarówno obiekty typu singleton, jak i statyczne są przydatne, ale nie należy ich nadużywać — podobnie jak innych wzorców.

W praktyce pomocne będzie stosowanie wzorca Singleton tam, gdzie konieczny jest dokładnie jeden obiekt do koordynacji w systemie. Oto przykład pokazujący użycie wzorca Singleton właśnie w takim kontekście:

```
// Podejście opcjonalne: obiekt zawierający opcje konfiguracyjne dla obiektu singleton,
// np. const options = { name: "test", pointX: 5};
class Singleton {
    constructor(options = {}) {
      // Przypisanie właściwości dla obiektu typu singleton
      this.name = 'SingletonTester';
      this.pointX = options.pointX || 6;
      this.pointY = options.pointY || 10;
    }
  }

  // Zmienna przechowująca nasz egzemplarz
  let instance;

  // Emulacja statycznych zmiennych i metod
  const SingletonTester = {
    name: 'SingletonTester',
    // Metoda służąca do pobierania egzemplarza, jej wartością zwrotną
    // jest egzemplarz obiektu klasy typu singleton
    getInstance(options) {
      if (instance === undefined) {
        instance = new Singleton(options);
      }

      return instance;
    },
  };
```

```
const singletonTest = SingletonTester.getInstance({
  pointX: 5,
});

// Zarejestrowanie danych wyjściowych pointX w celu sprawdzenia ich poprawności
// Dane wyjściowe: 5
console.log(singletonTest.pointX);
```

Wprawdzie wzorzec Singleton ma na pewno zastosowania uzasadniające jego użycie, ale jeśli pojawia się konieczność jego użycia w kodzie pisanym w JavaScripcie, może to wskazywać na potrzebę przemyślenia projektu. W przeciwieństwie do języków typu C++ lub Java, w których trzeba zdefiniować klasę w celu utworzenia obiektu, JavaScript pozwala na bezpośrednie tworzenie obiektów. Dlatego obiekt można utworzyć bezpośrednio zamiast definiować klasę typu singleton. Korzystanie z tego rodzaju klas w JavaScripcie wiąże się z pewnymi wadami:

Identyfikacja obiektu typu singleton może być trudna

Jeżeli importujesz ogromny moduł, nie będziesz w stanie rozpoznać, że dana klasa jest typu singleton. W efekcie możesz przypadkowo wykorzystać ją jako zwykłą klasę do utworzenia wielu obiektów i niepoprawnie je uaktualnić.

Przetestowanie obiektu typu singleton będzie wyzwaniem

Klasa typu singleton jest znacznie trudniejsza do przetestowania z wielu różnych powodów, takich jak m.in. ukryte zależności, trudności podczas tworzenia wielu egzemplarzy, trudności podczas obsługi zależności.

Konieczne jest przeprowadzenie ostrożnej koordynacji

Typowym przykładem zastosowania klasy typu singleton będzie przechowywanie danych niezbędnych w zasięgu globalnym, np. danych uwierzytelniających bądź danych ciasteczek, które mogą być zdefiniowane tylko raz, a następnie używane przez wiele komponentów. Implementacja poprawnej kolejności wykonywania zaczyna nabierać znaczenia krytycznego, aby dane zawsze były używane po ich udostępnieniu, a nie na odwrót. To może okazać się wyzwaniem, gdy aplikacja stanie się większa i bardziej skomplikowana.

Zarządzanie stanem w aplikacji tworzonych z użyciem biblioteki React

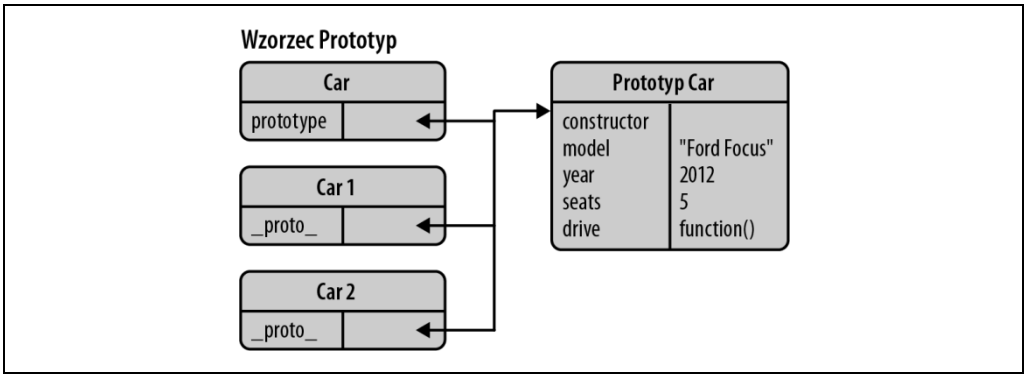
Programiści używający biblioteki React podczas tworzenia aplikacji internetowych mogą bazować na stanie globalnym, wykorzystując do tego narzędzia zarządzania stanem takie jak Redux lub React Context zamiast klas typu singleton. W przeciwieństwie do klas typu singleton wymienione narzędzia dostarczają informacje o stanie dostępne w trybie tylko do odczytu, a nie jako modyfikowalne.

Wady związane z istnieniem stanu globalnego nie znikają magicznie podczas używania wspomnianych narzędzi, ale przynajmniej można mieć pewność, że ten stan globalny zostanie zmieniony w oczekiwany sposób, ponieważ komponenty nie mają możliwości jego bezpośredniego uaktualniania.

Wzorzec Prototyp

Banda Czterech odwołuje się do wzorca Prototyp jako tego, który tworzy obiekty na podstawie szablonu istniejącego obiektu poprzez wykorzystanie klonowania.

Wzorzec Prototyp można potraktować jako bazujący na dziedziczeniu prototypowym, w którym to mamy do czynienia z tworzeniem obiektów będących prototypami dla innych obiektów. Obiekt `prototype` jest praktycznie używany jako matryca dla poszczególnych obiektów tworzonych przez konstruktor. Na przykład jeśli prototyp użytej funkcji konstruktora zawiera właściwość o nazwie `name` (jak zobaczysz w kolejnym fragmencie kodu), wówczas każdy obiekt tworzony przez ten konstruktor również będzie miał tę właściwość. Zapoznaj się ze schematem pokazanym na rysunku 7.3.



Rysunek 7.3. Wzorzec Prototyp

Przeglądając definicje tego wzorca istniejące w (niepoświęconej JavaScriptowi) literaturze, *można* ponownie znaleźć odwołania do klas. Rzeczywistość jest jednak taka, że dziedziczenie prototypowe unika używania klas. Nie ma „definicji" obiektu bądź jądra obiektu, po prostu następuje utworzenia kopii istniejących obiektów funkcjonalnych.

Jedną z zalet stosowanie wzorca Prototyp jest to, że mamy do czynienia z mocnymi stronami prototypu, jakie natywnie ma do zaoferowania JavaScript, a nie z próbą imitowania funkcjonalności znanej z innych języków. Nie zawsze tak bywa w przypadku innych wzorców projektowych.

Omawiany tutaj wzorzec to nie tylko łatwy sposób na implementację dziedziczenia, ale również umożliwia on poprawę wydajności działania kodu. Podczas definiowania funkcji w obiekcie są one tworzone przez odwołanie (więc wszystkie obiekty potomne wskazują tę samą funkcję) zamiast przez utworzenie oddzielnych kopii.

Wydania ES2015+ JavaScriptu pozwalają stosować klasy i konstruktory podczas tworzenia obiektów. Dzięki temu kod będzie bardziej przejrzysty, stosuje reguły projektu i analizy zorientowanej obiektowo, a klasy i konstruktory są wewnętrznie kompilowane do funkcji i prototypów. To gwarantuje możliwość wykorzystania prototypowych mocnych stron JavaScriptu i zwiększenie wydajności działania kodu.

Dla zainteresowanych warto w tym miejscu dodać, że dziedziczenie prototypowe, zgodne ze zdefiniowanym w standardzie ECMAScript 5, wymaga użycia wywołania `Object.create()`, o którym już wspomniałem we wcześniejszej części rozdziału. Przypominam, że metoda `Object.create()` tworzy obiekt o określonym prototypie, który opcjonalnie będzie zawierał określone właściwości, np. `Object.create( prototype, optionalDescriptorObjects )`.

Oto przykład takiego podejścia:

```
const myCar = {
    name: 'Ford Escort',

    drive() {
        console.log("Uwaga. Prowadzę samochód!");
    },

    panic() {
        console.log('Poczekaj. Jak to można zatrzymać?');
    },
};

// Użycie wywołania Object.create() do utworzenia nowego egzemplarza
const yourCar = Object.create(myCar);

// Teraz można zobaczyć, że jeden obiekt jest prototypem tego drugiego
console.log(yourCar.name);
```

Wywołanie `Object.create()` pozwala na łatwe zaimplementowanie koncepcji zaawansowanych, takich jak dziedziczenie różnicowe, w którym to obiekty mogą dziedziczyć bezpośrednio po innych obiektach. We wcześniejszej części rozdziału pokazałem, jak wywołanie `Object.create()` umożliwia zainicjalizowanie właściwości obiektu za pomocą drugiego dostarczonego argumentu. Spójrz na kolejny fragment kodu:

```
const vehicle = {
    getModel() {
        console.log(`Model tego pojazdu to... ${this.model}`);
    },
};

const car = Object.create(vehicle, {
    id: {
        value: MY_GLOBAL.nextId(),
        // writable:false, configurable:false (domyślnie)
        enumerable: true,
    },

    model: {
        value: 'Ford',
        enumerable: true,
    },
});
```

Tutaj właściwości można zainicjalizować za pomocą drugiego argumentu wywołania `Object.create()`, używając do tego składni podobnej do wykorzystywanej przez omówione wcześniej metody `Object.defineProperties()` i `Object.defineProperty()`.

Warto w tym miejscu dodać, że relacje prototypowe mogą spowodować problemy podczas wymieniania właściwości i obiektów oraz (zgodnie z zaleceniem Crockforda) opakowywania zawartości pętlą w sprawdzeniu `hasOwnProperty()`.

Jeżeli chcesz zaimplementować wzorzec prototypu bez uciekania się do bezpośredniego użycia wywołania `Object.create()`, możesz zasymulować wzorzec w sposób pokazany w poprzednim przykładzie:

```
class VehiclePrototype {
  constructor(model) {
    this.model = model;
  }

  getModel() {
    console.log(`Model tego pojazdu to... ${this.model}`);
  }

  clone() {}
}

class Vehicle extends VehiclePrototype {
  constructor(model) {
    super(model);
  }

  clone() {
    return new Vehicle(this.model);
  }
}

const car = new Vehicle('Ford Escort');
const car2 = car.clone();
car2.getModel();
```

To podejście alternatywne nie pozwala użytkownikowi na zdefiniowanie w taki sam sposób właściwości przeznaczonej tylko do odczytu (ponieważ jeśli nie będzie zachowana ostrożność, może dojść do przypadkowego zmodyfikowania `vehiclePrototype`).

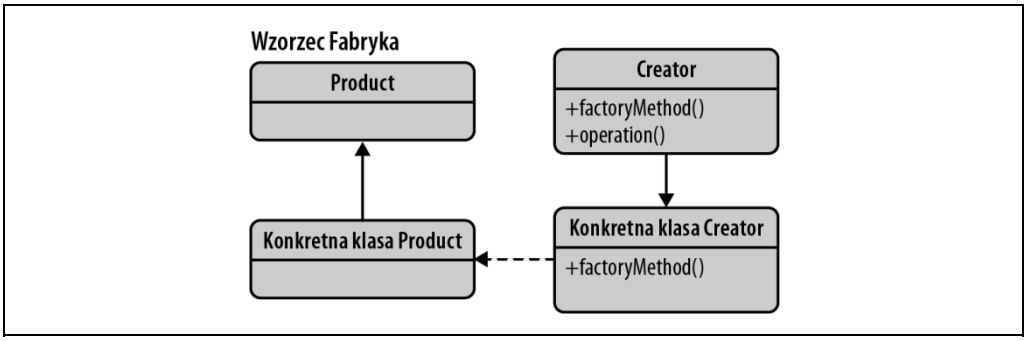
Ostatnia alternatywna implementacja wzorca prototypu może mieć następującą postać:

```
const beget = (() => {
    class F {
        constructor() {}
    }

    return proto => {
        F.prototype = proto;
        return new F();
    };
})();
```

Do tej metody można się odnieść z poziomu funkcji `vehicle`. Jednak zwróć uwagę, że tutaj `vehicle` emuluje konstruktor, ponieważ wzorzec prototypu nie obejmuje żadnej notacji inicjalizacji poza dołączeniem obiektu do prototypu.

Wzorzec Fabryka

Wzorzec Fabryka to kolejny wzorzec konstrukcyjny przeznaczony do tworzenia obiektów. Różni się od pozostałych wzorców w tej kategorii, ponieważ nie wymaga wyraźnego użycia konstruktora. Zamiast tego Fabryka może zapewnić przeznaczony do tworzenia obiektów ogólny interfejs, w którym można określić typ obiektu Fabryka przeznaczony do utworzenia (zobacz rysunek 7.4).



Rysunek 7.4. Wzorzec Fabryka

Wyobraź sobie fabrykę interfejsu użytkownika, w której ma zostać utworzony pewien typ komponentu interfejsu użytkownika. Zamiast tworzyć go bezpośrednio za pomocą operatora `new` lub innego konstruktora, do tego celu zostaje użyty obiekt fabryki. Trzeba mu podać typ nowego komponentu, np. panel lub przycisk, a następnie utworzy on obiekt gotowy do użycia.

Jest to szczególnie użyteczne, jeśli proces tworzenia obiektu jest względnie skomplikowany, np. w ogromnym stopniu zależy od czynników dynamicznych bądź konfiguracji aplikacji.

Kolejny przykład został zbudowany na podstawie poprzedniego fragmentu kodu, używającego logiki wzorca Konstruktor do definiowania obiektów reprezentujących samochody. Pokazuje, jak klasę `VehicleFactory` można zaimplementować za pomocą wzorca Fabryka:

```
// Plik Types.js – klasy używanej w tle
// Klasa przeznaczona do definiowania obiektu reprezentującego nowy samochód osobowy
class Car {
  constructor({ doors = 4, state = 'fabrycznie nowy', color = 'srebrny' } = {}) {
    this.doors = doors;
    this.state = state;
    this.color = color;
  }
}

// Klasa przeznaczona do definiowania obiektu reprezentującego nowy samochód ciężarowy
class Truck {
  constructor({ state = 'używany', wheelSize = 'duże', color = 'niebieski' } = {}) {
    this.state = state;
    this.wheelSize = wheelSize;
    this.color = color;
  }
}
```

```
// Plik FactoryExample.js
// Zdefiniowanie klasy fabryki pojazdów
class VehicleFactory {
  constructor() {
    this.vehicleClass = Car;
  }

  // Metoda fabryki przeznaczona do tworzenia nowych egzemplarzy obiektów reprezentujących pojazdy
  createVehicle(options) {
    const { vehicleType, ...rest } = options;

    switch (vehicleType) {
      case 'car':
        this.vehicleClass = Car;
        break;
      case 'truck':
        this.vehicleClass = Truck;
        break;
      // Domyślne wywołanie to VehicleFactory.prototype.vehicleClass (Car)
    }

    return new this.vehicleClass(rest);
  }
}

// Utworzenie egzemplarza fabryki przeznaczonego do tworzenia obiektów samochodów osobowych
const carFactory = new VehicleFactory();
const car = carFactory.createVehicle({
  vehicleType: 'car',
  color: 'żółty',
  doors: 6,
});

// Test potwierdzający, że obiekt samochodu osobowego został utworzony za pomocą prototypu
vehicleClass/prototype Car
// Dane wyjściowe: true
console.log(car instanceof Car);
// Dane wyjściowe: Car object of color "żółty", doors: 6 in a "fabrycznie nowy" state
console.log(car);
```

Zostały zdefiniowane klasy dla samochodów osobowego i ciężarowego z użyciem konstruktorów definiujących właściwości odpowiednie dla poszczególnych pojazdów. Klasa `VehicleFactory` potrafi utworzyć nowy obiekt pojazdu typu `Car` lub `Truck` na podstawie przekazanej wartości `vehicleType`.

Istnieją dwa możliwe podejścia w zakresie tworzenia ciężarówek z wykorzystaniem klasy `VehicleFactory`.

W podejściu pierwszym trzeba zmodyfikować egzemplarz `VehicleFactory`, aby używał klasy `Truck`:

```
const movingTruck = carFactory.createVehicle({
    vehicleType: 'truck',
    state: 'jak nowy',
    color: 'red',
    wheelSize: 'małe',
});
```

```
// Test potwierdzający, że obiekt samochodu ciężarowego został utworzony za pomocą prototypu
vehicleClass/prototype Truck
// Dane wyjściowe: true
console.log(movingTruck instanceof Truck);

// Dane wyjściowe: Truck object of color "czerwony", a "jak nowy" state
// and a "małe" wheelSize
console.log(movingTruck);
```

W podejściu drugim trzeba utworzyć podklasę `VehicleFactory` przeznaczoną do tworzenia klasy fabryki, która będzie odpowiedzialna za budowanie ciężarówek:

```
class TruckFactory extends VehicleFactory {
    constructor() {
        super();
        this.vehicleClass = Truck;
    }
}
const truckFactory = new TruckFactory();
const myBigTruck = truckFactory.createVehicle({
    state: 'O Boże, naprawdę kiepski.',
    color: 'różowy',
    wheelSize: 'tak duże',
});

// Potwierdzenie, że obiekt myBigTruck został utworzony z użyciem prototypu Truck
// Dane wyjściowe: true
console.log(myBigTruck instanceof Truck);

// Dane wyjściowe: Truck object with the color "różowy", wheelSize "tak duże"
// and state "O Boże, naprawdę kiepski"
console.log(myBigTruck);
```

Kiedy używać wzorca Fabryka?

Zastosowanie wzorca Fabryka może okazać się korzystne w następujących sytuacjach:

- Gdy konfiguracja obiektu bądź komponentu jest naprawdę skomplikowana.
- Gdy potrzebny jest wygodny sposób na wygenerowanie różnych egzemplarzy obiektów w zależności od aktualnego środowiska.
- Gdy pracujesz nad wieloma małymi obiektami lub komponentami, które współdzielą te same właściwości.
- Gdy łączysz obiekty z egzemplarzami innych obiektów, które do działania wymagają jedynie spełnienia kontraktu API (tzw. podejście *duck typing*). Jest to użyteczne rozwiązanie pozwalające wyeliminować powiązanie między obiektami.

Kiedy nie używać wzorca Fabryka?

Zastosowany do niewłaściwego typu problemu, wzorzec ten może ogromnie — i niepotrzebnie — skomplikować aplikację. O ile zapewnienie interfejsu przeznaczonego do tworzenia obiektu nie jest celem projektowym dla budowanej biblioteki lub frameworka, wówczas sugeruję pozostanie przy jawnie używanych konstruktorach, aby w ten sposób uniknąć nadmiernego obciążenia.

Skoro proces tworzenia obiektu jest praktycznie oddzielony przez interfejs, może to wprowadzić także problemy związane z testami jednostkowymi, w zależności od stopnia skomplikowania procesu.

Fabryki abstrakcyjne

Warto również wiedzieć o istnieniu wzorca Fabryka Abstrakcyjna, którego zadaniem jest hermetyzacja grupy poszczególnych fabryk mających wspólny cel. Szczegóły związane z implementacją zbioru obiektów oddziela on od ogólnego sposobu użycia obiektów.

Wzorzec Fabryka Abstrakcyjna można wykorzystać, gdy system musi być niezależny pod względem sposobu generowania tworzonych przez niego obiektów bądź zachodzi potrzeba pracy z wieloma typami obiektów.

Przykład, który będzie jednocześnie prosty i łatwy do zrozumienia, to fabryka pojazdów definiująca sposoby pobierania lub rejestrowania typów pojazdów. Nazwą wzorca Fabryka Abstrakcyjna może być `AbstractVehicleFactory`. Wzorzec Fabryka Abstrakcyjna będzie pozwalał definiować typy pojazdów takie jak `car` lub `truck`, a konkretne fabryki będą implementowały jedynie klasy spełniające kontrakt danego pojazdu, np. `Vehicle.prototype.drive` i `Vehicle.prototype.breakDown`:

```
class AbstractVehicleFactory {
  constructor() {
    // Miejsce na przechowywanie typów pojazdów
    this.types = {};
  }

  getVehicle(type, customizations) {
    const Vehicle = this.types[type];
    return Vehicle ? new Vehicle(customizations) : null;
  }

  registerVehicle(type, Vehicle) {
    const proto = Vehicle.prototype;
    // Zarejestrowane będą jedynie te klasy, które spełniają kontrakt pojazdu
    if (proto.drive && proto.breakDown) {
      this.types[type] = Vehicle;
    }
    return this;
  }
}

// Przykład użycia:
const abstractVehicleFactory = new AbstractVehicleFactory();
abstractVehicleFactory.registerVehicle('car', Car);
abstractVehicleFactory.registerVehicle('truck', Truck);

// Utworzenie nowego obiektu samochodu osobowego na podstawie typu pojazdu abstrakcyjnego
const car = abstractVehicleFactory.getVehicle('car', {
  color: 'oliwkowy',
  state: 'jak nowy',
});
```

```
// Utworzenie w podobny sposób nowego obiektu samochodu ciężarowego
const truck = abstractVehicleFactory.getVehicle('truck', {
  wheelSize: 'średnie',
  color: 'neonowo żółty',
});
```

Wzorce strukturalne

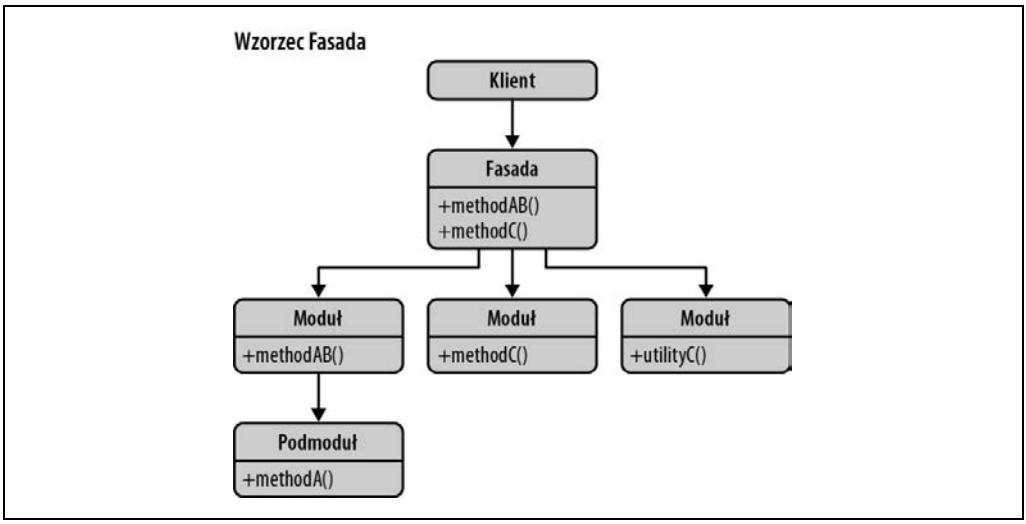
Wzorce strukturalne dotyczą łączenia klas i obiektów. Na przykład koncepcja dziedziczenia pozwala łączyć interfejsy i obiekty w taki sposób, aby mogły zapewnić nową funkcjonalność. Wzorce strukturalne zapewniają najlepsze metody i praktyki w zakresie organizacji klas i obiektów.

W tym rozdziale omówię następujące wzorce z tej kategorii:

- wzorzec Fasada,
- wzorzec Domieszka,
- wzorzec Dekorator,
- wzorzec Pyłek.

Wzorzec Fasada

Gdy stawiasz fasadę, przedstawiasz światu wygląd zewnętrzny, który może skrywać zupełnie inną rzeczywistość. To stanowiło inspirację nazwy następnego wzorca projektowego, który zostanie tutaj omówiony — wzorca Fasada. Dostarcza on wygodny interfejs wysokiego poziomu dla większego fragmentu kodu, ukrywając jego prawdziwy poziom złożoności. Potraktuj to jako uproszczenie API przedstawianego innym programistom, co stanowi jakość, która niemalże zawsze poprawia użyteczność (zobacz rysunek 7.5).



Rysunek 7.5. Wzorzec Fasada

Fasada to wzorzec strukturalny, który często był spotykany w bibliotekach JavaScriptu, takich jak jQuery. Wprawdzie implementacja mogła obsługiwać metody o szerokim spektrum działania, ale tylko „fasada", czyli ograniczona abstrakcja tych metod, była przeznaczona do publicznego użycia.

Takie rozwiązanie umożliwia pracę bezpośrednio z fasadą zamiast ze znajdującym się w tle podsystemem. Gdy używasz metod jQuery takich jak `$(el).css()` lub `$(el).animate()`, tak naprawdę korzystasz z wzorca Fasada, czyli z prostszego interfejsu publicznego, który pozwala uniknąć konieczności ręcznego wywoływania wielu wewnętrznych metod jQuery wymaganych do wykonywania pewnych operacji. Ponadto dzięki temu unika się ręcznej pracy z API modelu DOM i obsługi technicznej zmiennych stanu.

Metody jQuery powinny być uznawane za pośrednie abstrakcje. Większe bezpośrednie znacznie dla programistów ma to, że API modelu DOM i wzorzec Fasada niezwykle ułatwiają korzystanie z biblioteki jQuery.

Opierając się na dotychczasowych informacjach, można stwierdzić, że wzorzec Fasada upraszcza interfejs klasy oraz oddziela klasę od używającego jej kodu. To pozwala na pośrednią pracę z podsystemami w sposób, który czasami jest mniej podatny na błędy niż w przypadku uzyskiwania bezpośredniego dostępu do podsystemu. Zaletą wzorca Fasada jest łatwość w użyciu oraz często mniejsza wielkość rozwiązania, w którym został zaimplementowany ten wzorzec.

Warto zobaczyć ten wzorzec w działaniu. Poniżej zamieściłem niezoptymalizowany fragment kodu, w którym wzorzec Fasada został wykorzystany do uproszczenia interfejsu nasłuchującego zdarzeń w przeglądarkach WWW. Odbywa się to przez utworzenie metody odpowiedzialnej za sprawdzenie pod kątem istnienia pewnych funkcjonalności, aby mogła zapewnić rozwiązanie zarówno bezpieczne, jak i niezależne od przeglądarki WWW.

```
const addMyEvent = (el, ev, fn) => {
    if (el.addEventListener) {
      el.addEventListener(ev, fn, false);
    } else if (el.attachEvent) {
      el.attachEvent(`on${ev}`, fn);
    } else {
      el[`on${ev}`] = fn;
    }
  };
```

W podobny sposób działa doskonale znana funkcja biblioteki jQuery `$(document).ready(...)`. Wewnętrznie jest wspomagana metodą o nazwie `bindReady()`, która wykonuje następujące zadania:

```
function bindReady() {
  // Użycie przydatnego wywołania zwrotnego zdarzenia
  document.addEventListener('DOMContentLoaded', DOMContentLoaded, false);
  // Rozwiązanie awaryjne polega na użyciu wywołania window.onload, które zawsze działa
  window.addEventListener('load', jQuery.ready, false);
}
```

To jest kolejny przykład zastosowania wzorca Fasada, gdy reszta świata używa ograniczonego interfejsu udostępnionego przez wywołanie `$(document).ready(...)`, a znacznie bardziej skomplikowana implementacja jest ukryta przed programistami.

Jednak wzorzec Fasada nie musi być używany w pojedynkę. Istnieje możliwość jego połączenia z innymi wzorcami, np. wzorcem Moduł. Jak możesz zobaczyć w kolejnym przykładzie, egzemplarz wzorca Moduł składa się z pewnej liczby metod, które zostały zdefiniowane jako prywatne. Następnie wzorzec Fasada jest używany w celu zapewnienia znacznie prostszego API, które pozwala uzyskać dostęp do tych metod:

```
// Plik privateMethods.js
const _private = {
  i: 5,
  get() {
    console.log(`Wartość bieżąca: ${this.i}`);
  },
  set(val) {
    this.i = val;
  },
  run() {
    console.log('bieganie');
  },
  jump() {
    console.log('skakanie');
  },
};

export default _private;

// Plik module.js
import _private from './privateMethods.js';

const module = {
  facade({ val, run }) {
    _private.set(val);
    _private.get();
    if (run) {
      _private.run();
    }
  },
};

export default module;

// Plik index.js
import module from './module.js';

// Dane wyjściowe: "Wartość bieżąca: 10" i "bieganie"
module.facade({
  run: true,
  val: 10,
});
```

W tym przykładzie wywołanie `module.facade()` spowodowało zdefiniowanie działania prywatnego w module, ale nie zostało ono ujawnione użytkownikom. Znacznie łatwiejsze dla nich jest wykorzystanie funkcjonalności bez przejmowania się szczegółami na poziomie implementacji.

Wzorzec Domieszka

W tradycyjnych językach programowania, takich jak C++ i Lisp, domieszki są klasami oferującymi funkcjonalność, którą podklasa lub grupa podklas może łatwo dziedziczyć i tym samym zapewnić sobie możliwość wielokrotnego używania funkcji.

Tworzenie podklasy

Wspomniałem już o udostępnionych w wydaniach ES2015+ JavaScriptu funkcjach, które umożliwiły rozszerzenie bazy, czyli klasy nadrzędnej (superklasy), oraz wywoływanie metod w klasie nadrzędnej. Klasa potomna rozszerzająca superklasę to podklasa.

Tworzenie podklasy odnosi się do dziedziczenia po obiekcie klasy bazowej właściwości dla nowego obiektu. Podklasa wciąż może definiować metody, w tym także te, które nadpisują metody pierwotnie zdefiniowane w superklasie. Metoda podklasy może wywoływać nadpisaną metodę superklasy, co jest znane pod nazwą łączenia metod. Podobnie może wywoływać także konstruktor superklasy, co jest określane mianem łączenia konstruktora.

Aby pokazać tworzenie podklasy w działaniu, potrzebna jest klasa bazowa, dla której będzie można utworzyć kilka nowych egzemplarzy. Załóżmy, że modelowany będzie koncept reprezentujący człowieka:

```
class Person{
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.gender = "male";
    }
}
// Nowy egzemplarz klasy Person można łatwo utworzyć w następujący sposób:
const clark = new Person( 'Clark', 'Kent' );
```

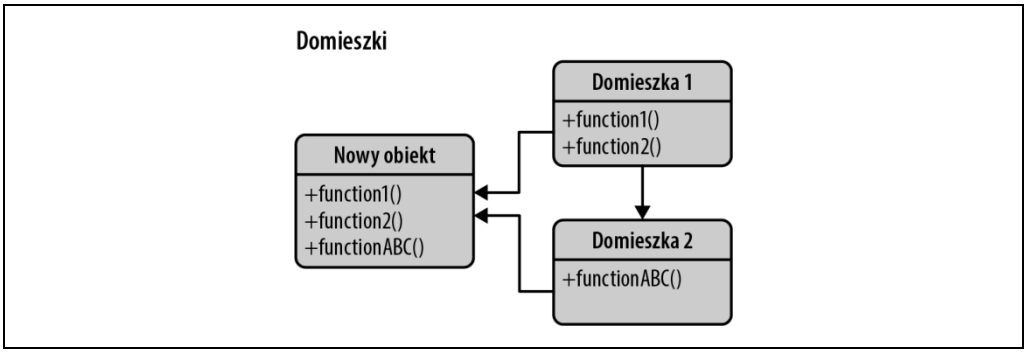
Następnie określamy, że nowa klasa będzie podklasą istniejącej klasy o nazwie `Person`. Załóżmy, że chcemy dodać właściwości pozwalające odróżnić egzemplarz klasy `Person` od egzemplarza klasy `Superhero`, przy czym ten drugi ma dziedziczyć właściwości po superklasie `Person`. Superbohater (ang. *superhero*) będzie miał wiele cech wspólnych ze zwykłą osobą (ang. *person*), np. imię i płeć, co powinno pomóc w idealnym pokazaniu sposobu działania podklasy:

```
class Superhero extends Person {
    constructor(firstName, lastName, powers) {
        // Wywołanie konstruktora superklasy
        super(firstName, lastName);
        this.powers = powers;
    }
}

// Nowy egzemplarz klasy Superhero można utworzyć w następujący sposób:

const SuperMan = new Superhero('Clark','Kent', ['flight','heat-vision']);
console.log(SuperMan);

// Dane wyjściowe to atrybuty obiektu typu Person, a także atrybut powers
```

Konstruktor `Superhero` tworzy egzemplarz klasy `Superhero`, która jest rozszerzeniem klasy `Person`. Obiekty tego typu mają atrybuty klas znajdujących się wyżej w łańcuchu klas. Jeżeli zostaną zdefiniowane wartości domyślne w klasie `Person`, wówczas egzemplarz klasy `Superhero` może nadpisać wszelkie dziedziczone wartości wartościami przeznaczonymi dla tej właśnie klasy.

Domieszka

W JavaScripcie można dziedziczyć po domieszce w celu otrzymania za pomocą rozszerzenia dostępu do jej funkcjonalności. Każda nowo definiowana klasa może mieć superklasę, po której dziedziczy metody i właściwości. Klasy mogą również definiować własne metody i właściwości. Ten fakt można wykorzystać do promowania wielokrotnego użycia funkcji, jak pokazałem na rysunku 7.6.



Rysunek 7.6. Domieszki

Domieszka pozwala obiektom pożyczać (lub dziedziczyć) funkcjonalności po niej, co odbywa się przy jedynie minimalnym poziomie złożoności. Dlatego domieszka to klasa razem z atrybutami i metodami, które mogą być łatwo współdzielone z wieloma innymi klasami.

Wprawdzie klasy JavaScriptu nie mogą dziedziczyć po wielu superklasach, ale nadal można łączyć funkcjonalność pochodzącą z wielu klas. Klasa w JavaScripcie może być używana jako wyrażenie, a także jako polecenie. W przypadku wyrażenia zwraca nową klasę za każdym razem, gdy dane wyrażenie jest wykonywane. Klauzula `extends` może również akceptować dowolne wyrażenie zwracające klasy lub konstruktory. Te funkcje pozwalają zdefiniować domieszkę jako funkcję akceptującą superklasę i tworzącą na jej podstawie nową podklasę.

Wyobraź sobie zdefiniowanie domieszki zawierającej funkcje narzędziowe w standardowej klasie JavaScriptu, jak pokazałem w kolejnym fragmencie kodu:

```
const MyMixins = superclass =>
    class extends superclass {
        moveUp() {
            console.log('Przejdź w górę.');
        }
        moveDown() {
            console.log('Przejdź w dół.');
        }
```

```
        stop() {
            console.log('Stop! W imię miłości!');
        }
    };
```

Tutaj została zdefiniowana funkcja `MyMixins`, która może rozszerzyć dynamiczną superklasę. Następnie zostaną utworzone dwie klasy, `CarAnimator` i `PersonAnimator`, które `MyMixins` może rozszerzyć i zwrócić podklasę razem z metodami zdefiniowanymi w `MyMixins`, a także w rozszerzonych klasach:

```
// Szkielet konstruktora klasy carAnimator
class CarAnimator {
    moveLeft() {
        console.log('Przejdź w lewo.');
    }
}
// Szkielet konstruktora klasy personAnimator
class PersonAnimator {
    moveRandomly() {
        /*...*/
    }
}

// Rozszerzenie MyMixins za pomocą CarAnimator
class MyAnimator extends MyMixins(CarAnimator) {}

// Utworzenie nowego egzemplarza klasy carAnimator
const myAnimator = new MyAnimator();
myAnimator.moveLeft();
myAnimator.moveDown();
myAnimator.stop();

// Dane wyjściowe:
// Przejdź w lewo.
// Przejdź w dół.
// Stop! W imię miłości!
```

Jak widać, dość łatwo można łączyć podobny sposób działania w klasach.

W kolejnym fragmencie kodu zostały zdefiniowane dwie klasy, `Car` i `Mixin`. Zadanie polega na zmodyfikowaniu (jest to tutaj synonim słowa „rozszerzyć”) klasy `Car` w taki sposób, aby dziedziczyła po klasie `Mixin` pewne metody, a dokładnie `driveForward()` i `driveBackward()`.

Ten przykład pokazuje, jak można zmodyfikować konstruktor, aby zawierał żądaną funkcjonalność, bez konieczności powielania tego procesu dla każdej funkcji konstruktora:

```
// Plik Car.js
class Car {
  constructor({ model = 'nie podano modelu', color = 'nie podano koloru' }) {
    this.model = model;
    this.color = color;
  }
}

export default Car;

// Pliki Mixin.js i index.js pozostają niezmienione
```

```
// Plik index.js
import Car from './Car.js';
import Mixin from './Mixin.js';

class MyCar extends Mixin(Car) {}

// Utworzenie nowego egzemplarza Car
const myCar = new MyCar({});

// Upewnienie się o posiadaniu dostępu do żądanych metod
myCar.driveForward();
myCar.driveBackward();

// Dane wyjściowe:
// Jazda do przodu.
// Jazda wstecz.

const mySportsCar = new MyCar({
  model: 'Porsche',
  color: 'czerwony',
});

mySportsCar.driveSideways();

// Dane wyjściowe:
// Jazda w bok.
```

Wady i zalety

Domieszka pomaga zmniejszyć powtarzalność funkcjonalnego kodu i zwiększyć możliwość wielokrotnego używania funkcji w rozwiązaniu. Gdy istnieje prawdopodobieństwo, że aplikacja będzie wymagała współdzielonego sposobu działania w wielu egzemplarzach obiektów, wówczas bardzo łatwo można uniknąć powielającego się kodu przez zdefiniowanie w domieszce tej funkcjonalności współdzielonej. Dzięki temu można się skoncentrować na implementacji w systemie jedynie funkcjonalności, która jest naprawdę wyjątkowa.

Mając to na uwadze, wady domieszki są nieco bardziej dyskusyjne. Według niektórych programistów wstrzykiwanie funkcjonalności do klasy bądź prototypu obiektu jest kiepskim pomysłem, ponieważ prowadzi zarówno do zaśmiecenia prototypu, jak i do niepewności dotyczącej pochodzenia funkcji. W ogromnych systemach może to stanowić problem.

Nawet w przypadku biblioteki React domieszki były często używane w celu dodania funkcjonalności do komponentów przed wprowadzeniem klas ES6. Zespół tworzący tę bibliotekę odradza używanie domieszek (*https://legacy.reactjs.org/blog/2016/07/13/mixins-considered-harmful.html*), ponieważ prowadzą one do niepotrzebnego zwiększenia złożoności komponentu, który w efekcie staje się trudny w późniejszej obsłudze technicznej oraz podczas wielokrotnego używania. Zamiast tego zespół Reacta zachęca do stosowania wzorców Komponenty Wyższego Rzędu i Zaczepy (*https://medium.com/@dan_abramov/mixins-are-dead-long-live-higher-order-components-↪94a0d2f9e750*).

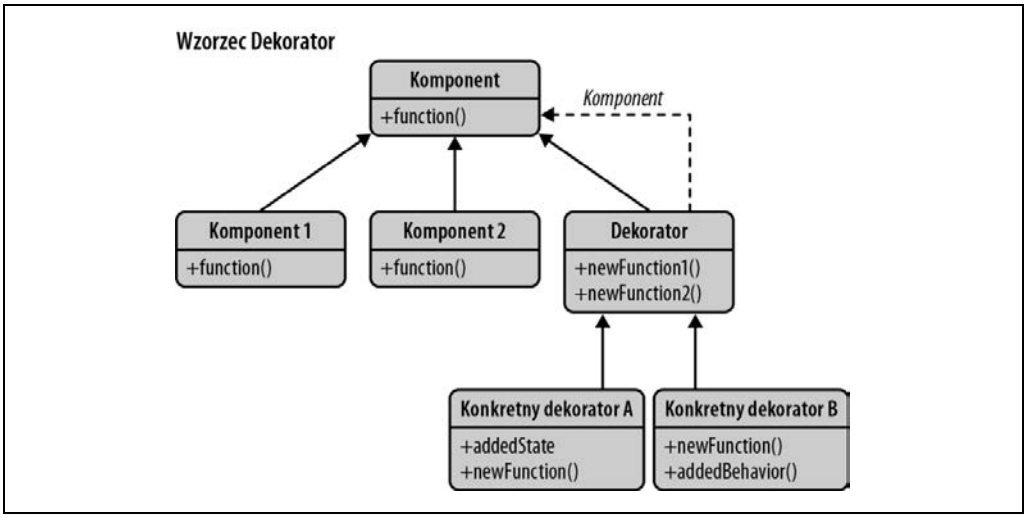
Uważam, że solidna dokumentacja mogłaby pomóc w zminimalizowaniu dezorientacji co do pochodzenia funkcji w domieszkach. Podobnie jak w przypadku innych wzorców, także ten nie powinien powodować problemów, o ile zachowasz ostrożność podczas jego implementacji.

Wzorzec Dekorator

Dekorator to wzorzec strukturalny, którego celem jest promowanie wielokrotnego używania kodu. Podobnie jak domieszki także wzorzec Dekorator można uznać za kolejną realną alternatywę dla tworzenia podklas.

Klasycznie dekoratory umożliwiały dynamiczne dodanie funkcjonalności do istniejących klas w systemie. Idea polegała na tym, że samo *dekorowanie* nie było istotnym czynnikiem dla podstawowej funkcjonalności klasy. W przeciwnym razie można by je było zastosować dla samej *superklasy*.

Wzorca Dekorator można użyć do modyfikowania istniejącego systemu, w którym do obiektów ma zostać dodana kolejna funkcjonalność bez wprowadzania poważnych zmian w kodzie wykorzystującym te obiekty. Częstym powodem, dla którego programiści używają dekoratorów, jest to, że aplikacja może zawierać funkcjonalności wymagające wielu odmiennych typów obiektów. Wyobraź sobie definiowanie setek różnych obiektów konstruktorów na przykład dla gry utworzonej w JavaScripcie (zobacz rysunek 7.7).



Rysunek 7.7. Wzorzec Dekorator

Konstruktory obiektów mogą reprezentować różne typy graczy, każdy z odmiennymi możliwościami. Na przykład gra *Władca Pierścieni* wymagałaby konstruktorów dla obiektów reprezentujących postacie — `Hobbit`, `Elf`, `Orc`, `Wizard`, `Mountain Giant`, `Stone Giant` itd. — których bez problemu mogą być setki. Jeżeli do tego trzeba będzie uwzględnić możliwości, wyobraź sobie konieczność tworzenia podklas dla poszczególnych połączeń postaci i typów możliwości — np. `HobbitWithRing`, `HobbitWithSword`, `HobbitWithRingAndSword`. To nie jest praktyczne i zdecydowanie trudne w zarządzaniu, jeśli uwzględni się coraz większą liczbę poszczególnych umiejętności postaci.

Wzorzec Dekorator nie ma zbyt dużego związku ze sposobem tworzenia obiektów, lecz koncentruje się na problemie rozszerzenia ich funkcjonalności. Zamiast polegać na dziedziczeniu prototypowym mamy pojedynczą klasę bazową i stopniowo dodajemy do obiektów dekoratory zapewniające kolejne możliwości. Pomysł polega na tym, że zamiast tworzyć podklasy dodajemy (dekorujemy) właściwości lub metody do obiektu bazowego, aby w ten sposób go usprawnić.

Klasy JavaScriptu można wykorzystać do definiowania klas bazowych, które następnie mogą być dekorowane. Dodanie nowych atrybutów lub metod do egzemplarzy obiektów klasy JavaScriptu jest prostym procesem. Na listingach 7.4 i 7.5 pokazałem, jak można zaimplementować prosty dekorator.

Listing 7.4. Dekorowanie konstruktora za pomocą nowej funkcjonalności

```
// Konstruktor klasy Vehicle
class Vehicle {
    constructor(vehicleType) {
        // Wartości domyślne
        this.vehicleType = vehicleType || 'car';
        this.model = 'default';
        this.license = '00000-000';
    }
}

// Sprawdzenie egzemplarza prostego pojazdu
const testInstance = new Vehicle('car');
console.log(testInstance);

// Dane wyjściowe:
// vehicle: car, model:default, license: 00000-000

// Przystępujemy do utworzenia nowego egzemplarza klasy Vehicle, który zostanie udekorowany
const truck = new Vehicle('truck');

// Nowa funkcjonalność, którą zostanie udekorowany egzemplarz
truck.setModel = function(modelName) {
    this.model = modelName;
};

truck.setColor = function(color) {
    this.color = color;
};

// Sprawdzenie poprawności działania metody typu setter i przypisywania wartości
truck.setModel('CAT');
truck.setColor('blue');

console.log(truck);

// Dane wyjściowe:
// vehicle:truck, model:CAT, color: blue

// Pokazanie, że "pojazd" pozostaje niezmieniony
const secondInstance = new Vehicle('car');
console.log(secondInstance);

// Dane wyjściowe:
// vehicle: car, model:default, license: 00000-000
```

W tym przykładzie `truck` jest egzemplarzem klasy `Vehicle`. Został udekorowany metodami dodatkowymi, `setColor()` i `setModel()`.

Wprawdzie taki rodzaj uproszczonej implementacji jest funkcjonalny, ale nie pokazuje wszystkich zalet dekoratorów. Dlatego przedstawię teraz mój wariant pochodzącego z doskonałej książki pod tytułem *Wzorce projektowe. Rusz głową! Tworzenie rozszerzalnego i łatwego w utrzymaniu oprogramowania obiektowego. Wydanie II* (Freeman i in.) przykładu modelującego zakup MacBooka.

Listing 7.5. Dekorowanie obiektów wieloma dekoratorami

```
// Konstruktor przeznaczony do udekorowania
class MacBook {
    constructor() {
        this.cost = 997;
        this.screenSize = 11.6;
    }
    getCost() {
        return this.cost;
    }
    getScreenSize() {
        return this.screenSize;
    }
}

// Dekorator 1
class Memory extends MacBook {
    constructor(macBook) {
        super();
        this.macBook = macBook;
    }
    getCost() {
        return this.macBook.getCost() + 75;
    }
}

// Dekorator 2
class Engraving extends MacBook {
    constructor(macBook) {
        super();
        this.macBook = macBook;
    }
    getCost() {
        return this.macBook.getCost() + 200;
    }
}

// Dekorator 3
class Insurance extends MacBook {
    constructor(macBook) {
        super();
        this.macBook = macBook;
    }
    getCost() {
        return this.macBook.getCost() + 250;
    }
}
```

```
// Inicjalizacja obiektu głównego
let mb = new MacBook();

// Inicjalizacja dekoratorów
mb = new Memory(mb);
mb = new Engraving(mb);
mb = new Insurance(mb);

// Dane wyjściowe: 1522
console.log(mb.getCost());

// Dane wyjściowe: 11.6
console.log(mb.getScreenSize());
```

W tym przykładzie nasze dekoratory nadpisują funkcję `.cost()` superklasy `MacBook`, aby zwracała bieżącą cenę MacBooka razem z kosztem wybranego uaktualnienia.

Jest to uznawane za dekorowanie, ponieważ metody oryginalnego konstruktora obiektów typu `MacBook` nie zostały nadpisane, np. `screenSize()`, podobnie jak wszelkie inne właściwości, które mogły zostać zdefiniowane jako część klasy `MacBook` — one również pozostały nietknięte.

W poprzednim przykładzie nie mamy zdefiniowanego interfejsu. Podczas przechodzenia od twórcy do odbiorcy następuje odejście od odpowiedzialności za zapewnienie zgodności obiektu z interfejsem.

Dekoratory pseudoklasyczne

Przedstawię teraz wariant wzorca Dekorator przedstawiony po raz pierwszy w książce pod tytułem *Pro JavaScript Design Patterns* Dustina Diaza i Rossa Harmesa.

Inaczej niż w wybranych wcześniej przykładach, Diaz i Harmes ściślej trzymali się sposobu implementacji dekoratorów w innych językach programowania (takich jak Java i C++), używając koncepcji „interfejsu", która wkrótce zostanie nieco dokładniej zdefiniowana.

Ta konkretna wersja wzorca Dekorator została tutaj przedstawiona w celach informacyjnych. Jeżeli uważasz ją za zbyt skomplikowaną, zachęcam do stosowania jednej z prostszych implementacji, które przedstawiłem we wcześniejszej części rozdziału.

Interfejs

W książce pod tytułem *Pro JavaScript Design Patterns* wzorzec Dekorator został użyty do przejrzystego opakowywania obiektów innymi obiektami tego samego interfejsu. Wspomniany interfejs to sposób zdefiniowania metod, które obiekt *powinien* posiadać. Jednak interfejs nie określa bezpośrednio tego, jak te metody mają być zaimplementowane. Interfejs może również opcjonalnie wskazywać parametry pobierane przez metody.

Dlaczego programista miałby używać interfejsu w JavaScripcie? Idea polega na tym, że są one samodokumentujące się i promują wielokrotne używanie tego samego kodu. Teoretycznie dzięki

interfejsowi kod jest znacznie stabilniejszy, ponieważ wszelkie zmiany interfejsu muszą być propagowane do implementujących je obiektów.

W kolejnym fragmencie kodu przedstawiłem przykład implementacji interfejsu w JavaScripcie z wykorzystaniem podejścia o nazwie *duck typing*. Takie podejście pomaga ustalić, czy obiekt jest egzemplarzem konstruktora bądź obiektu, bazując na implementowanych przez niego metodach.

```
// Utworzenie interfejsów z użyciem wcześniej przygotowanego
// konstruktora Interface(), który pobiera nazwę interfejsu
// i szkielety metod przeznaczonych do udostępnienia

// W pozostałej części przykładu summary() i placeOrder()
// reprezentują funkcjonalność, która powinna być
// obsługiwana przez interfejs
const reminder = new Interface('List', ['summary', 'placeOrder']);

const properties = {
    name: 'Pamiętaj, aby kupić mleko.',
    date: '05/06/2040',
    actions: {
        summary() {
            return 'Pamiętaj, aby kupić mleko, prawie się skończyło!';
        },
        placeOrder() {
            return 'Kup mleko w lokalnym sklepie spożywczym';
        },
    },
};

// Teraz zostanie utworzony konstruktor implementujący
// te właściwości i metody
class Todo {
    constructor({ actions, name }) {
        // Należy wymienić metody, które mają zostać zaimplementowane.
        // Trzeba również podać egzemplarz interfejsu, względem którego
        // odbędzie się sprawdzenie.

        Interface.ensureImplements(actions, reminder);

        this.name = name;
        this.methods = actions;
    }
}

// Utworzenie nowego egzemplarza konstruktora Todo
const todoItem = new Todo(properties);

// Sprawdzenie poprawnego działania tych funkcji
console.log(todoItem.methods.summary());
console.log(todoItem.methods.placeOrder());

// Dane wyjściowe:
// Pamiętaj, aby kupić mleko, prawie się skończyło!
// Kup mleko w lokalnym sklepie spożywczym.
```

Zarówno klasyczny JavaScript, jak i jego wydania ES2015+ nie obsługują interfejsów. Jednak istnieje możliwość utworzenia klasy interfejsu. W omówionym przykładzie `Interface.ensureImplements` zapewnia ścisłe sprawdzenie funkcjonalności. Na stronie *https://gist.github.com/addyosmani/89827e5c72273354e8c48b501ce208eb* znajduje się kod źródłowy zarówno dla wymienionej metody, jak i dla konstruktora `Interface`.

Największy problem związany z interfejsami polega na tym, że JavaScript nie ma wbudowanej ich obsługi. To może prowadzić do prób emulowania tej funkcjonalności w postaci znanej z innych języków programowania, co nie musi być najlepszym rozwiązaniem. Jednak jeżeli naprawdę chcesz używać interfejsów, możesz wykorzystać język TypeScript, mający wbudowaną ich obsługę. Lekkiego interfejsu można w JavaScripcie używać bez widocznego spadku wydajności działania. W następnym punkcie omówię dekoratory *abstrakcyjne*, wykorzystując tę samą koncepcję.

Dekoratory abstrakcyjne

Aby przyjrzeć się strukturze tej wersji wzorca Dekorator, załóżmy, że mamy superklasę `MacBook`, która ponownie modeluje MacBooka, i sklep pozwalający „udekorować" laptop wieloma usprawnieniami za dodatkową opłatą.

Usprawnienia obejmują zwiększenie ilości pamięci RAM do 4 GB lub 8 GB (oczywiście może być jej więcej), grawerowanie, dodanie oprogramowania Parallels lub pokrowca. Jeżeli takie rozwiązanie miałoby być modelowane za pomocą oddzielnych podklas dla poszczególnych wariantów opcji, wówczas możemy otrzymać kod podobny do następującego:

```
const MacBook = class {
    //...
};

const MacBookWith4GBRam = class {};
const MacBookWith8GBRam = class {};
const MacBookWith4GBRamAndEngraving = class {};
const MacBookWith8GBRamAndEngraving = class {};
const MacBookWith8GBRamAndParallels = class {};
const MacBookWith4GBRamAndParallels = class {};
const MacBookWith8GBRamAndParallelsAndCase = class {};
const MacBookWith4GBRamAndParallelsAndCase = class {};
const MacBookWith8GBRamAndParallelsAndCaseAndInsurance = class {};
const MacBookWith4GBRamAndParallelsAndCaseAndInsurance = class {};
…itd.
```

Takie rozwiązanie będzie niepraktyczne, gdyż nowa podklasa okaże się wymagana dla każdego możliwego wariantu dostępnych rozszerzeń MacBooka. Ponieważ lepszym rozwiązaniem będzie zachowanie prostoty, bez konieczności zapewnienia obsługi ogromnego zbioru podklas, warto zobaczyć, jak można wykorzystać dekoratory, by przygotować lepsze podejście do tego problemu.

Zamiast wymagać klas dla wszystkich wariantów, jak pokazałem wcześniej, zostanie utworzonych tylko pięć nowych klas dekoratorów. Metody wywoływane w tych klasach rozszerzeń będą przekazywane do klasy `MacBook`.

W kolejnym fragmencie kodu dekoratory przejrzyście opakowują komponenty i mogą być wymieniane, ponieważ używają dokładnie tego samego interfejsu. Spójrz na interfejs, za pomocą którego będzie zdefiniowany MacBook:

```
const MacBook = new Interface('MacBook', [
    'addEngraving',
    'addParallels',
    'add4GBRam',
    'add8GBRam',
    'addCase',
]);

// Laptop MacBook Pro może więc być reprezentowany następująco:
class MacBookPro {
    // Implementacja interfejsu MacBook
}

// ES2015+: W celu dodania nowych metod nadal można użyć Object.prototype,
// ponieważ wewnętrznie jest wykorzystywana dokładnie ta sama struktura
MacBookPro.prototype = {
    addEngraving() {},
    addParallels() {},
    add4GBRam() {},
    add8GBRam() {},
    addCase() {},
    getPrice() {
        // Cena bazowa.
        return 900.0;
    },
};
```

Aby ułatwić możliwość późniejszego dodawania kolejnych opcji, została zdefiniowana klasa dekoratora abstrakcyjnego razem z metodami domyślnymi wymaganymi do implementacji interfejsu `MacBook`, który zapewni podklasy dla pozostałych opcji. Wzorzec Dekorator Abstrakcyjny gwarantuje możliwość niezależnego dekorowania klasy bazowej z użyciem dowolnej liczby dekoratorów, jaka jest niezbędna dla różnych wariantów rozszerzeń (pamiętasz wcześniejszy przykład?), bez konieczności tworzenia nowej klasy dla każdego możliwego wariantu opcji:

```
// Klasa dekoratora abstrakcyjnego MacBook
class MacBookDecorator {
    constructor(macbook) {
        Interface.ensureImplements(macbook, MacBook);
        this.macbook = macbook;
    }

    addEngraving() {
        return this.macbook.addEngraving();
    }

    addParallels() {
        return this.macbook.addParallels();
    }

    add4GBRam() {
        return this.macbook.add4GBRam();
    }
```

```
    add8GBRam() {
        return this.macbook.add8GBRam();
    }

    addCase() {
        return this.macbook.addCase();
    }

    getPrice() {
        return this.macbook.getPrice();
    }
}
```

W tym przykładzie Dekorator `MacBook` akceptuje obiekt (`MacBook`) przeznaczony do użycia jako komponent bazowy. Wykorzystany zostaje zdefiniowany wcześniej interfejs `MacBook`, a poszczególne metody po prostu wywołują tę samą metodę komponentu. Teraz można utworzyć klasy opcji, które zostaną dodane za pomocą Dekoratora `MacBook`:

```
// Teraz można rozszerzyć (udekorować) CaseDecorator
// za pomocą MacBookDecorator
class CaseDecorator extends MacBookDecorator {
    constructor(macbook) {
        super(macbook);
    }

    addCase() {
        return `${this.macbook.addCase()} Dodanie pokrowca dla MacBooka`;
    }

    getPrice() {
        return this.macbook.getPrice() + 45.0;
    }
}
```

Zostają nadpisane metody `addCase()` i `getPrice()`, które mają zostać udekorowane. Najpierw odbywa się wywołanie tych metod zdefiniowanych w pierwotnej klasie `MacBook`, a następnie do wyniku zostaje po prostu dołączona wartość w postaci ciągu tekstowego lub liczbowa (np. `45.00`).

W tym punkcie przedstawiłem całkiem sporo informacji. Warto spróbować połączyć wszystko w całość w jednym przykładzie, który mam nadzieję zawiera wszystko, co omówiłem w tym punkcie:

```
// Utworzenie nowego egzemplarza reprezentującego MacBooka
const myMacBookPro = new MacBookPro();
// Dane wyjściowe: 900.00
console.log(myMacBookPro.getPrice());
// Udekorowanie egzemplarza
const decoratedMacBookPro = new CaseDecorator(myMacBookPro);
// Wartością zwrotną będzie 945.00
console.log(decoratedMacBookPro.getPrice());
```

Dekoratory mogą dynamiczne modyfikować obiekty i stanowią doskonały wzorzec do zmiany istniejących systemów. W przypadku ich rzadszego używania znacznie prostsze jest utworzenie dekoratorów dla obiektu zamiast poszczególnych podklas dla każdego typu obiektu. Dzięki temu znacznie łatwiejsza staje się obsługa techniczna aplikacji, które mogą wymagać wielu obiektów podklas.

Funkcjonalną wersję tego przykładu znajdziesz w serwisie JSBin (*http://jsbin.com/UMEJaXu/1/edit?html,js,output*).

Wady i zalety

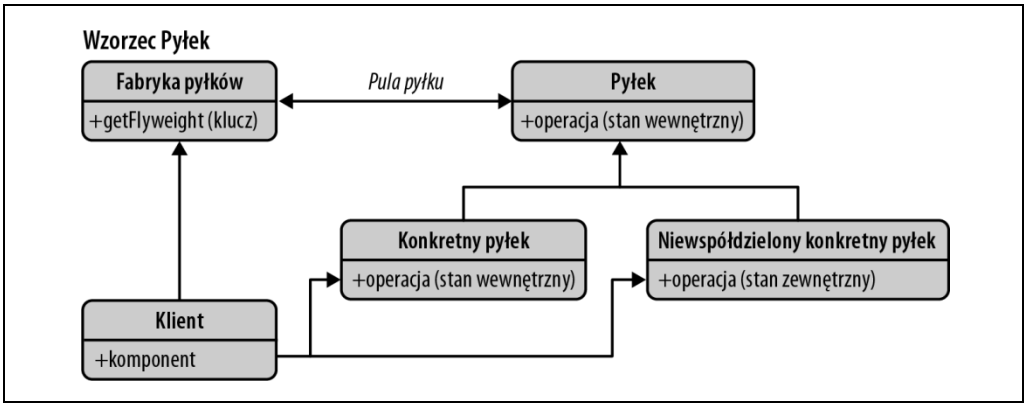
Programiści lubią używać tego wzorca, ponieważ można z niego korzystać w sposób przejrzysty i pod pewnymi względami elastyczny. Jak można było zobaczyć, obiekty mogą być opakowane, inaczej „udekorowane", nową funkcjonalnością i używane bez obaw dotyczących modyfikacji obiektu bazowego. W szerszym kontekście ten wzorzec pomaga również uniknąć konieczności opierania się na ogromnej liczbie podklas, które mogłyby zapewnić te same zalety.

Jednak omówiony wzorzec ma również pewne wady, o których istnieniu trzeba wiedzieć podczas jego implementowania. Przede wszystkim jeśli będzie kiepsko zarządzany, to może znacznie skomplikować architekturę aplikacji, ponieważ powoduje pojawienie się w przestrzeni nazw wielu małych, choć podobnych obiektów. Problem polega na tym, że inni programiści, nieświadomi użycia tego wzorca, mogą mieć trudność w określeniu powodu jego zastosowania, a tym samym zarządzanie aplikacją stanie się dla nich trudniejsze.

W tym drugim przypadku rozwiązaniem może być wystarczająco obszerne użycie komentarzy bądź też wyszukanie informacji na temat tego wzorca. Jednak jeśli masz wpływ na częstotliwość stosowania wzorca Dekorator w aplikacjach, wymienione wcześniej problemy nie powinny występować.

Wzorzec Pyłek

Wzorzec Pyłek to klasyczne rozwiązanie strukturalne przeznaczone do optymalizacji kodu, który się powtarza, jest wolny oraz w sposób nieefektywny współdzieli dane. Celem tego wzorca jest minimalizacja użycia pamięci przez aplikację, co staje się możliwe dzięki współdzieleniu maksymalnej ilości danych z powiązanymi obiektami, np. konfiguracja aplikacji, informacje o stanie (zobacz rysunek 7.8).



Rysunek 7.8. Wzorzec Pyłek

Ten wzorzec został opisany po raz pierwszy w 1990 roku przez Paula Caldera i Marka Lintona. Nazwa wzorca wynika z tego, że pomaga on zminimalizować ilość pamięci wymaganą przez aplikację.

W praktyce współdzielenie danych we wzorcu Pyłek może obejmować wiele podobnych obiektów lub konstrukcji danych, używanych przez wiele obiektów i umieszczających dane w pojedynczym obiekcie zewnętrznym. Taki obiekt może następnie zostać przekazany obiektom zależnym od znajdujących się w nim danych. To znacznie lepsze podejście niż sytuacja, w której poszczególne obiekty przechowują identyczne dane.

Stosowanie wzorca Pyłek

Wzorzec Pyłek można zastosować na dwa sposoby. Pierwszy dotyczy warstwy danych, na której mamy do czynienia z koncepcją współdzielenia danych przez ogromną liczbę podobnych obiektów przechowywanych w pamięci. Drugi dotyczy warstwy modelu DOM, jako centralnego menedżera zdarzeń. To pozwala uniknąć dołączania procedur obsługi zdarzeń do każdego elementu potomnego w kontenerze nadrzędnym o podobnym sposobie działania.

Tradycyjnie wzorzec Pyłek jest najczęściej używany na warstwie danych, więc to podejście omówię jako pierwsze.

Wzorzec Pyłek i współdzielenie danych

W przypadku tego podejścia trzeba poznać kilka dodatkowych koncepcji związanych z klasycznym wzorcem Pyłek. W tym wzorcu istnieje koncepcja dwóch stanów: wewnętrznego i zewnętrznego. Informacje dotyczące stanu wewnętrznego mogą być wymagane przez wewnętrzne metody obiektów, bez których to danych absolutnie nie będą mogły funkcjonować. Z kolei informacje zewnętrzne mogą zostać usunięte i być przechowywane zewnętrznie.

Obiekty można zastępować tymi samymi danymi wewnętrznymi dzięki użyciu pojedynczego obiektu współdzielonego, utworzonego przez metodę fabryki. To pozwala znacznie ograniczyć ogólną ilość niejawnie przechowywanych danych.

Zaletą jest możliwość obserwowania już zainicjowanych obiektów, aby nowe kopie były tworzone tylko wtedy, gdy stan wewnętrzny różni się od już istniejących obiektów.

Do obsługi stanu zewnętrznego jest używany menedżer. Jego implementację można przeprowadzić na wiele sposobów. Jedna z nich bazuje na obiekcie menedżera zawierającym centralną bazę danych informacji o stanach zewnętrznych i obiektach pyłków, do których one należą.

Implementacja klasycznego wzorca Pyłek

Wzorzec Pyłek nie jest ostatnio zbyt często używany w JavaScripcie, więc wiele implementacji, które można wykorzystać jako inspirację, pochodzi z języków Java i C++.

Pierwszy przykład wzorca Pyłek w kodzie to moja implementacja w JavaScripcie przykładu tego wzorca w Javie pochodzącego z artykułu Wikipedii (*https://en.wikipedia.org/wiki/Flyweight_pattern*).

W tej implementacji zostały użyte trzy typy komponentów:

Pyłek

Odpowiada interfejsowi, za pomocą którego pyłki są w stanie otrzymywać informacje o stanie i reagować na nie.

Konkretny pyłek

Zawiera rzeczywistą implementację interfejsu pyłku i przechowuje informacje o stanach wewnętrznych. Konkretne pyłki muszą być współdzielone i muszą mieć możliwość przeprowadzania operacji na danych stanu zewnętrznego.

Fabryka pyłków

Zarządza obiektami pyłków i ich tworzeniem. Gwarantuje współdzielenie pyłków i zarządzanie nimi jako grupą obiektów, do których można wykonywać zapytania, gdy będą potrzebne poszczególne egzemplarze. Jeżeli obiekt został już utworzony w grupie, zostanie po prostu zwrócony. W przeciwnym razie fabryka doda nowy obiekt do puli i go zwróci.

To odpowiada następującym definicjom w omawianej tutaj implementacji:

- `CoffeeOrder`: pyłek,
- `CoffeeFlavor`: konkretny pyłek,
- `CoffeeOrderContext`: element pomocniczy,
- `CoffeeFlavorFactory`: fabryka pyłków,
- `testFlyweight`: elementy wykorzystujące pyłki.

Implementacja bazująca na dynamicznym modyfikowaniu metod

Dynamiczne modyfikowanie metod pozwala na rozszerzenie możliwości języka bądź rozwiązania bez konieczności modyfikowania kodu źródłowego środowiska uruchomieniowego. Do zaimplementowania interfejsów to rozwiązanie wymaga niedostępnego natywnie w JavaScripcie słowa kluczowego Javy (`implements`), więc pracę trzeba zacząć od rozwiązania tego problemu.

`Function.prototype.implementsFor` działa w konstruktorze obiektu i akceptuje klasę nadrzędną (funkcję) lub obiekt oraz dziedziczy po niej za pomocą zwykłego mechanizmu dziedziczenia (dla funkcji) bądź też z użyciem dziedziczenia wirtualnego (dla obiektów):

```
// Klasa narzędziowa symulująca implementację interfejsu
class InterfaceImplementation {
  static implementsFor(superclassOrInterface) {
    if (superclassOrInterface instanceof Function) {
      this.prototype = Object.create(superclassOrInterface.prototype);
      this.prototype.constructor = this;
      this.prototype.parent = superclassOrInterface.prototype;
    } else {
      this.prototype = Object.create(superclassOrInterface);
      this.prototype.constructor = this;
      this.prototype.parent = superclassOrInterface;
    }
    return this;
  }
}
```

Tę klasę można wykorzystać do poradzenia sobie z brakiem w JavaScripcie słowa kluczowego `implements` — rozwiązanie polega na jawnym dziedziczeniu interfejsu przez funkcję. Następnie `CoffeeFlavor` implementuje interfejs `CoffeeOrder` i musi zawierać jego metody, aby pozwolić na przypisanie obiektowi funkcjonalności obsługującej tę implementację:

```
// Interfejs CoffeeOrder
const CoffeeOrder = {
  serveCoffee(context) {},
  getFlavor() {},
};

class CoffeeFlavor extends InterfaceImplementation {
  constructor(newFlavor) {
    super();
    this.flavor = newFlavor;
  }

  getFlavor() {
    return this.flavor;
  }

  serveCoffee(context) {
    console.log(`Podanie kawy ${this.flavor} do
       stolika ${context.getTable()}`);
  }
}

// Implementacja interfejsu dla CoffeeOrder
CoffeeFlavor.implementsFor(CoffeeOrder);
const CoffeeOrderContext = (tableNumber) => ({
  getTable() {
    return tableNumber;
  },
});

class CoffeeFlavorFactory {
  constructor() {
    this.flavors = {};
    this.length = 0;
  }

  getCoffeeFlavor(flavorName) {
    let flavor = this.flavors[flavorName];
    if (!flavor) {
      flavor = new CoffeeFlavor(flavorName);
      this.flavors[flavorName] = flavor;
      this.length++;
    }
    return flavor;
  }

  getTotalCoffeeFlavorsMade() {
    return this.length;
  }
}
```

```
// Przykładowy sposób użycia:
const testFlyweight = () => {
  const flavors = [];
  const tables = [];
  let ordersMade = 0;
  const flavorFactory = new CoffeeFlavorFactory();

  function takeOrders(flavorIn, table) {
    flavors.push(flavorFactory.getCoffeeFlavor(flavorIn));
    tables.push(CoffeeOrderContext(table));
    ordersMade++;
  }

  // Złożenie zamówienia
  takeOrders('Cappuccino', 2);
  // ...

  // Realizacja zamówienia
  for (let i = 0; i < ordersMade; ++i) {
    flavors[i].serveCoffee(tables[i]);
  }

  console.log(' ');
  console.log(`Całkowita liczba utworzonych obiektów CoffeeFlavor:
    ${flavorFactory.getTotalCoffeeFlavorsMade()}`);
};

testFlyweight();
```

Konwersja kodu na postać używającą wzorca Pyłek

Kontynuuję omawianie wzorca Pyłek przez implementację systemu przeznaczonego do zarządzania wszystkimi książkami w bibliotece. Oto lista niezbędnych metadanych dla każdej książki:

- identyfikator,
- tytuł,
- autor,
- dziedzina,
- liczba stron,
- identyfikator wydawcy,
- ISBN.

Potrzebne będą również następujące właściwości, które umożliwiają śledzenie wypożyczeń poszczególnych książek, daty wypożyczenia i oczekiwanej daty zwrotu:

- `checkoutDate`,
- `checkoutMember`,
- `dueReturnDate`,
- `availability`.

Kolejny fragment kodu pokazuje reprezentującą książkę klasę `Book`, jeszcze przed wprowadzeniem jakichkolwiek optymalizacji wynikających z użycia wzorca Pyłek. Konstruktor pobiera wszystkie właściwości związane bezpośrednio z książką oraz te, które są niezbędne do śledzenia jej wypożyczeń:

```
class Book {
  constructor(
    id,
    title,
    author,
    genre,
    pageCount,
    publisherID,
    ISBN,
    checkoutDate,
    checkoutMember,
    dueReturnDate,
    availability
  ) {
    this.id = id;
    this.title = title;
    this.author = author;
    this.genre = genre;
    this.pageCount = pageCount;
    this.publisherID = publisherID;
    this.ISBN = ISBN;
    this.checkoutDate = checkoutDate;
    this.checkoutMember = checkoutMember;
    this.dueReturnDate = dueReturnDate;
    this.availability = availability;
  }

  getTitle() {
    return this.title;
  }

  getAuthor() {
    return this.author;
  }

  getISBN() {
    return this.ISBN;
  }

  // Dla zachowania zwięzłości pozostałe metody typu getters nie zostały zamieszczone
  updateCheckoutStatus(
    bookID,
    newStatus,
    checkoutDate,
    checkoutMember,
    newReturnDate
  ) {
    this.id = bookID;
    this.availability = newStatus;
    this.checkoutDate = checkoutDate;
    this.checkoutMember = checkoutMember;
    this.dueReturnDate = newReturnDate;
  }
```

```
  extendCheckoutPeriod(bookID, newReturnDate) {
    this.id = bookID;
    this.dueReturnDate = newReturnDate;
  }

  isPastDue(bookID) {
    const currentDate = new Date();
    return currentDate.getTime() > Date.parse(this.dueReturnDate);
  }
}
```

Takie rozwiązanie prawdopodobnie będzie sprawdzało się dobrze w przypadku małej kolekcji książek. Jednak wraz z rozbudową biblioteki o coraz więcej tytułów z ich wieloma wydaniami i egzemplarzami taki system zarządzania okaże się bardzo wolny. Używanie tysięcy obiektów książek może być ogromnym obciążeniem dla pamięci. Na szczęście system można zoptymalizować za pomocą wzorca Pyłek, który pozwoli poprawić to rozwiązanie.

Dane mogą być przechowywane oddzielnie w stanach wewnętrznym i zewnętrznym — dane dotyczące obiektu książki (np. `title`, `author`) są uznawane za wewnętrzne, podczas gdy dane dotyczące wypożyczeń (np. `checkoutMember`, `dueReturnDate`) są uznawane za zewnętrzne. W praktyce to oznacza, że potrzebny jest tylko jeden obiekt `Book` dla wszystkich wariantów właściwości książek. Wprawdzie to wciąż znaczna liczba obiektów, ale i tak dużo mniejsza niż przed optymalizacją.

Egzemplarz przedstawionej tutaj klasy ze zbiorem metadanych zostanie utworzony dla wszystkich wymaganych kopii obiektu książki o określonym tytule i numerze ISBN:

```
// Wersja zoptymalizowana za pomocą wzorca Pyłek
class Book {
  constructor({ title, author, genre, pageCount, publisherID, ISBN }) {
    this.title = title;
    this.author = author;
    this.genre = genre;
    this.pageCount = pageCount;
    this.publisherID = publisherID;
    this.ISBN = ISBN;
  }
}
```

Jak widać, z klasy zostały usunięte dane przedstawiające informacje o stanie zewnętrznym. Wszystko, co jest związane z wypożyczeniami, zostało przeniesione do menedżera. Skoro dane obiektu są obecnie segmentowane, do inicjalizacji można wykorzystać fabrykę.

Prosta fabryka

Zdefiniujemy bardzo prostą fabrykę. Trzeba sprawdzić, czy obiekt książki o danym tytule został już wcześniej utworzony w systemie. Jeżeli tak, ten obiekt zostanie po prostu zwrócony. Natomiast jeżeli nie, obiekt dla nowej książki zostanie utworzony i będzie przechowywany w celu późniejszego użycia. To daje pewność, że tworzona jest tylko jedna kopia każdych unikatowych danych wewnętrznych.

```
// Fabryki książki to egzemplarz typu singleton
const existingBooks = {};
```

```
class BookFactory {
  createBook({ title, author, genre, pageCount, publisherID, ISBN }) {
    // Sprawdzenie, czy istnieje już wariant składający się z określonej książki + metadanych
    // Zapis z użyciem !! wymusza zwrot wartości boolowskiej
    const existingBook = existingBooks[ISBN];
    if (!!existingBook) {
      return existingBook;
    } else {
      // Jeżeli nie, obiekt dla nowej książki zostanie utworzony i będzie przechowywany w celu późniejszego użycia
      const book = new Book({ title, author, genre, pageCount, publisherID, ISBN });
      existingBooks[ISBN] = book;
      return book;
    }
  }
}
```

Zarządzanie danymi stanu zewnętrznego

Kolejnym krokiem jest przechowywanie gdzieś danych stanu, które zostały usunięte z obiektów typu `Book`. Na szczęście do ich hermetyzacji można wykorzystać menedżer (zdefiniowany jako egzemplarz typu singleton). Połączenie obiektu `Book` i informacji o czytelniku, który wypożyczył tę książkę, będzie nazywane rekordem `Book`. Menedżer będzie przechowywał oba te typy danych oraz zawierał logikę dotyczącą wypożyczeń, która została usunięta z klasy `Book` podczas jej optymalizacji z użyciem wzorca Pyłek:

```
// BookRecordManager to egzemplarz typu singleton
const bookRecordDatabase = {};

class BookRecordManager {
  // Dodanie do systemu obiektu reprezentującego nową książkę
  addBookRecord({ id, title, author, genre, pageCount, publisherID, ISBN,
       checkoutDate, checkoutMember, dueReturnDate, availability }) {
    const bookFactory = new BookFactory();
    const book = bookFactory.createBook({ title, author, genre, pageCount,
       publisherID, ISBN });
    bookRecordDatabase[id] = {
      checkoutMember,
      checkoutDate,
      dueReturnDate,
      availability,
      book,
    };
  }

  updateCheckoutStatus({ bookID, newStatus, checkoutDate, checkoutMember,
     newReturnDate }) {
    const record = bookRecordDatabase[bookID];
    record.availability = newStatus;
    record.checkoutDate = checkoutDate;
    record.checkoutMember = checkoutMember;
    record.dueReturnDate = newReturnDate;
  }

  extendCheckoutPeriod(bookID, newReturnDate) {
    bookRecordDatabase[bookID].dueReturnDate = newReturnDate;
  }
```

```
    isPastDue(bookID) {
      const currentDate = new Date();
      return currentDate.getTime() >
         Date.parse(bookRecordDatabase[bookID].dueReturnDate);
    }
  }
```

W wyniku tych zmian wszystkie dane wyodrębnione z *klasy* `Book` są teraz przechowywane w atrybucie egzemplarza `BookManager` typu singleton (`BookDatabase`). Takie rozwiązanie jest znacznie efektywniejsze niż ogromna liczba obiektów, która była używana w poprzednim podejściu. Metody związane z wypożyczeniami książki są teraz w tym egzemplarzu, ponieważ operują na danych zewnętrznych, a nie wewnętrznych.

Ten proces powoduje jedynie niewielkie zwiększenie skomplikowania ostatecznego rozwiązania. Jednak to naprawdę małe zmartwienie w porównaniu do pojawiających się wcześniej problemów z wydajnością działania. Z perspektywy danych, jeżeli mamy 30 egzemplarzy tej samej książki, informacje o niej będą przechowywane tylko jednokrotnie. Pamiętaj, że każda funkcja wymaga pewnej ilości pamięci. Dzięki wykorzystaniu wzorca Pyłek te funkcje istnieją w jednym miejscu (menedżer), a nie w każdym obiekcie, co przekłada się na mniejsze zużycie pamięci. W przypadku wspomnianej wcześniej wersji niezoptymalizowanej przechowywane jest odniesienie do obiektu funkcji, ponieważ został wykorzystany prototyp konstruktora klasy `Book`. Jeżeli zostanie użyta inna implementacja, funkcje będą tworzone dla każdego egzemplarza książki.

Wzorzec Pyłek i model DOM

Model DOM obsługuje dwa podejścia umożliwiające obiektom wykrywanie zdarzeń — od początku do końca (przechwytywanie zdarzeń) lub od dołu do góry (propagacja zdarzeń).

W pierwszym przypadku zdarzenie jest przechwytywane przez wysunięty najbardziej na zewnątrz element i propagowane do najbardziej zagnieżdżonego elementu. Natomiast w drugim zdarzenie jest przechwytywane i przekazywane do najbardziej zagnieżdżonego elementu, a dopiero później trafia do elementów zewnętrznych.

Gary Chisholm jest autorem jednej z najlepszych metafor opisujących wzorzec Pyłek w tym kontekście i brzmi ona tak:

> O pyłku spróbuj pomyśleć z perspektywy stawu. Ryba otwiera otwór gębowy (zdarzenie) i pojawiają się bąbelki unoszące się w kierunku powierzchni stawu (propagowanie zdarzeń). Gdy bąbelek powietrza dotrze do powierzchni, siedząca na tafli wody mucha odlatuje (akcja). W tym przykładzie rybę otwierającą otwór gębowy można porównać do kliknięcia przycisku, bąbelki do przekazywania zdarzeń, a odlatującą muchę do wykonywanej funkcji.

Propagowanie zostało opracowane do obsługi sytuacji, w których pojedyncze zdarzenie (np. `click`) może zostać obsłużone przez wiele procedur obsługi zdarzeń zdefiniowanych na różnych poziomach hierarchii modelu DOM. Gdy tak się dzieje, przekazywanie zdarzeń powoduje wykonanie procedury obsługi zdarzeń zdefiniowanej dla określonego elementu, na najniższym poziomie. Od tego momentu zdarzenie jest propagowane w górę do elementów nadrzędnych, a następnie do znajdujących się jeszcze wyżej.

Wzorzec Pyłek można wykorzystać do dalszego dopracowania procesu propagowania zdarzeń, jak to wyjaśnię w następnym punkcie.

Przykład: scentralizowana obsługa zdarzeń

W pierwszym przykładzie praktycznym zakładamy, że w dokumencie mamy wiele podobnych elementów, które działają w podobny sposób, gdy użytkownik podejmuje względem nich działanie, np. klika element lub umieszcza na nim kursor myszy.

Podczas tworzenia komponentu typu accordion, menu lub innego widżetu bazującego na liście zdarzenie `click` jest zwykle łączone z każdym elementem kontenera nadrzędnego, np. `$('ul li a').on(...)`. Zamiast dołączać zdarzenie do wielu elementów, na górze kontenera bardzo łatwo można umieścić obiekt pyłku, który będzie nasłuchiwał zdarzeń nadchodzących z dołu. Następnie mogą one zostać obsłużone za pomocą logiki tak prostej lub tak skomplikowanej, jak wymaga tego sytuacja.

Skoro wspomniane wcześniej typy komponentów często zawierają powtarzający się kod znaczników dla poszczególnych sekcji, np. wszystkich sekcji komponentu typu accordion, to istnieje duże prawdopodobieństwo, że sposób działania każdego klikniętego elementu będzie podobny i względny wobec podobnych klas w pobliżu. Te informacje można wykorzystać do opracowania prostego komponentu typu accordion, jak pokazałem na listingu 7.6.

Listing 7.6. Przykład scentralizowanej procedury obsługi zdarzeń

```
<div id="container">
    <div class="toggle">Więcej informacji (adres)
      <span class="info">
        Tutaj znajdziesz więcej informacji.
      </span>
    </div>

    <div class="toggle">Jeszcze więcej informacji (mapa)
      <span class="info">
        <iframe src="MAPS_URL"></iframe>
      </span>
    </div>
  </div>

  <script>
    (function() {
      const stateManager = {
        fly() {
          const self = this;
          $('#container')
            .off()
            .on('click', 'div.toggle', function() {
              self.handleClick(this);
            });
        },
        handleClick(elem) {
          $(elem)
            .find('span')
            .toggle('slow');
        },
      };
```

```
    // Inicjalizacja nasłuchiwania zdarzeń
    stateManager.fly();
  })();
</script>
```

Przestrzeń nazw `stateManager` została tutaj użyta do hermetyzacji logiki wzorca Pyłek, natomiast biblioteka *jQuery* posłużyła do powiązania początkowego kliknięcia i elementu `<div>` kontenera. Zdarzenie `unbind` zostaje zastosowane jako pierwsze, aby zagwarantować, że żadna inna logika na stronie nie dołączy do kontenera podobnej procedury obsługi zdarzeń.

W celu ustalenia, który dokładnie element potomny w kontenerze został kliknięty, sprawdzana jest wartość `target`, zapewniająca odniesienie do klikniętego elementu, niezależnie od jego elementu nadrzędnego. Te informacje są następnie używane do obsługi zdarzenia `click` bez konieczności rzeczywistego dołączania zdarzenia do konkretnego elementu potomnego podczas wczytywania strony.

Zaletą przedstawionego rozwiązania jest konwersja wielu niezależnych akcji na współdzielone, co potencjalnie pozwala zmniejszyć poziom użycia pamięci.

Wzorce operacyjne

Wzorce operacyjne pomagają w zdefiniowaniu komunikacji między obiektami. Usprawniają również komunikację między poszczególnymi obiektami w systemie.

W tym rozdziale zostaną omówione następujące wzorce z tej kategorii:

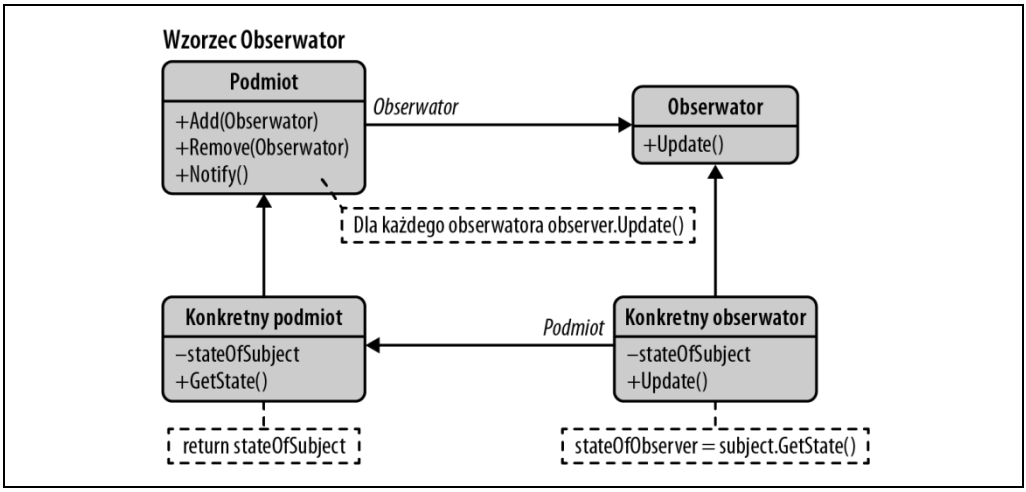
- wzorzec Obserwator,
- wzorzec Mediator,
- wzorzec Polecenie.

Wzorzec Obserwator

Wzorzec Obserwator pozwala powiadomić jeden obiekt o zmianie wprowadzonej w drugim obiekcie, przy czym obiekt nie musi mieć żadnych informacji o swoich obiektach zależnych. Bardzo często jest to wzorzec, gdzie obiekt (nazywany podmiotem) zawiera listę jego obiektów zależnych (nazywanych obserwatorami), które automatycznie informuje o wszelkich zmianach swojego stanu. W nowoczesnych frameworkach wzorzec Obserwator jest używany do informowania komponentów o zmianie stanu. W sposób graficzny ten wzorzec pokazałem na rysunku 7.9.

Kiedy podmiot musi poinformować obserwatory o czymś interesującym, przekazuje im powiadomienie (może ono obejmować konkretne dane dotyczące tematu). Gdy obserwator nie chce już być informowany o zmianach w danym temacie, może zostać usunięty z listy obserwatorów.

Dobrze jest odwołać się do opublikowanych definicji wzorców projektowych, niezależnych od języka programowania, aby w ten sposób dowiedzieć się nieco więcej na ich temat i oferowanych przez nie korzyści. Definicja wzorca Obserwator została podana w książce napisanej przez Bandę Czterech pod tytułem *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*:



Rysunek 7.9. Wzorzec Obserwator

> Jeden lub więcej obserwatorów jest zainteresowany stanem podmiotu, więc rejestruje się w podmiocie i dołącza do grupy obserwatorów. Gdy w obiekcie podmiotu zmieni się coś, co może interesować obserwatora, podmiot przekazuje obserwatorom komunikat powiadomienia, który wywołuje u nich metodę uaktualnienia. Gdy obserwator nie jest już zainteresowany stanem podmiotu, może po prostu usunąć się z listy obserwatorów.

Teraz zdobyte przed chwilą informacje można wykorzystać do implementacji wzorca Obserwator za pomocą następujących komponentów:

Podmiot
: Zawiera listę obserwatorów oraz zapewnia obsługę dodawania i usuwania obserwatorów na liście.

Obserwator
: Dostarcza interfejs `update` dla obiektów wymagających powiadamiania o zmianach zachodzących w stanie podmiotu.

`ConcreteSubject`
: Przekazuje obserwatorom powiadomienia w przypadku zmiany stanu podmiotu i przechowuje stan `ConcreteSubject`.

`ConcreteObserver`
: Przechowuje odwołanie do `ConcreteSubject` i implementuje interfejs update dla obserwatora w celu zagwarantowania spójności stanu ze stanem podmiotu.

Począwszy od wydań ES2015+, JavaScript umożliwia implementację wzorca Obserwator za pomocą klas przeznaczonych dla obserwatorów i podmiotów, razem z metodami `notify()` i `update()`.

Trzeba zacząć od modelowania listy obserwatorów, które może mieć podmiot. W tym celu zostanie wykorzystana klasa `ObserverList`:

```
class ObserverList {
    constructor() {
        this.observerList = [];
```

```
    }

    add(obj) {
        return this.observerList.push(obj);
    }

    count() {
        return this.observerList.length;
    }

    get(index) {
        if (index > -1 && index < this.observerList.length) {
            return this.observerList[index];
        }
    }

    indexOf(obj, startIndex) {
        let i = startIndex;

        while (i < this.observerList.length) {
            if (this.observerList[i] === obj) {
                return i;
            }
            i++;
        }

        return -1;
    }

    removeAt(index) {
        this.observerList.splice(index, 1);
    }
}
```

Następnie trzeba przygotować klasę podmiotu (`Subject`) pozwalającą na dodawanie, usuwanie lub powiadamianie obserwatorów znajdujących się na utworzonej wcześniej liście:

```
class Subject {
  constructor() {
    this.observers = new ObserverList();
  }

  addObserver(observer) {
    this.observers.add(observer);
  }

  removeObserver(observer) {
    this.observers.removeAt(this.observers.indexOf(observer, 0));
  }

  notify(context) {
    const observerCount = this.observers.count();
    for (let i = 0; i < observerCount; i++) {
    this.observers.get(i).update(context);
    }
  }
}
```

Kolejnym krokiem jest zdefiniowanie szkieletu przeznaczonego do tworzenia nowych obserwatorów. Funkcjonalność `update()` zostanie później nadpisana.

```
// Obserwator
class Observer {
    constructor() {}
    update() {
        // ...
    }
}
```

Z użyciem wymienionych wcześniej komponentów obserwatora w tej przykładowej aplikacji zostaną teraz zdefiniowane następujące komponenty:

- Przycisk umożliwiający dodanie do strony nowego obserwatora (element pola wyboru).
- Kontrolka pól wyboru działająca w charakterze podmiotu i powiadamiająca pola wyboru o konieczności uaktualnienia informacji o stanie.
- Kontener dla nowododanego pola wyboru.

Następnie zostaną zdefiniowane procedury obsługi `ConcreteSubject` i `ConcreteObserver` przeznaczone do dodawania nowych obserwatorów na stronie oraz implementacji uaktualnienia interfejsu. W tym rozwiązaniu wykorzystam dziedziczenie do rozszerzenia klas podmiotu i obserwatorów. Klasa `ConcreteSubject` hermetyzuje pole wyboru i generuje powiadomienie w przypadku kliknięcia głównego pola wyboru. Z kolei klasa `ConcreteObserver` hermetyzuje wszystkie pola wyboru będące obserwatorami i implementuje interfejs `Update` przez zmianę wartości pola wyboru. W kolejnym fragmencie kodu znajdują są komentarze wyjaśniające współpracę tych komponentów w kontekście omawianego przykładu.

Kod w HTML-u przedstawia się następująco:

```
<button id="addNewObserver">Dodaj nowe pole wyboru obserwatora</button>
<input id="mainCheckbox" type="checkbox"/>
<div id="observersContainer"></div>
```

Kod w JavaScripcie przedstawia się następująco:

```
// Odwołania do naszych elementów w modelu DOM
// Klasa ConcreteSubject
class ConcreteSubject extends Subject {
    constructor(element) {
      // Wywołanie konstruktora klasy nadrzędnej
      super();
      this.element = element;

      // Kliknięcie pola wyboru spowoduje wysłanie powiadomienia do obserwatorów tego elementu
      this.element.onclick = () => {
        this.notify(this.element.checked);
      };
    }
  }
  // Klasa ConcreteObserver
  class ConcreteObserver extends Observer {
    constructor(element) {
      super();
```

```
      this.element = element;
    }

    // Nadpisanie metody i zdefiniowanie w niej innego sposobu działania
    update(value) {
      this.element.checked = value;
    }
  }
  // Odwołania do naszych elementów w modelu DOM
  const addBtn = document.getElementById('addNewObserver');
  const container = document.getElementById('observersContainer');
  const controlCheckbox = new ConcreteSubject(
    document.getElementById('mainCheckbox')
  );

  const addNewObserver = () => {
    // Utworzenie nowego pola wyboru, które ma być dodane
    const check = document.createElement('input');
    check.type = 'checkbox';
    const checkObserver = new ConcreteObserver(check);

    // Dodanie nowego obserwatora do listy obserwatorów
    // naszego podmiotu głównego
    controlCheckbox.addObserver(checkObserver);

    // Dołączenie elementu do kontenera
    container.appendChild(check);
  };

  addBtn.onclick = addNewObserver;
}
```

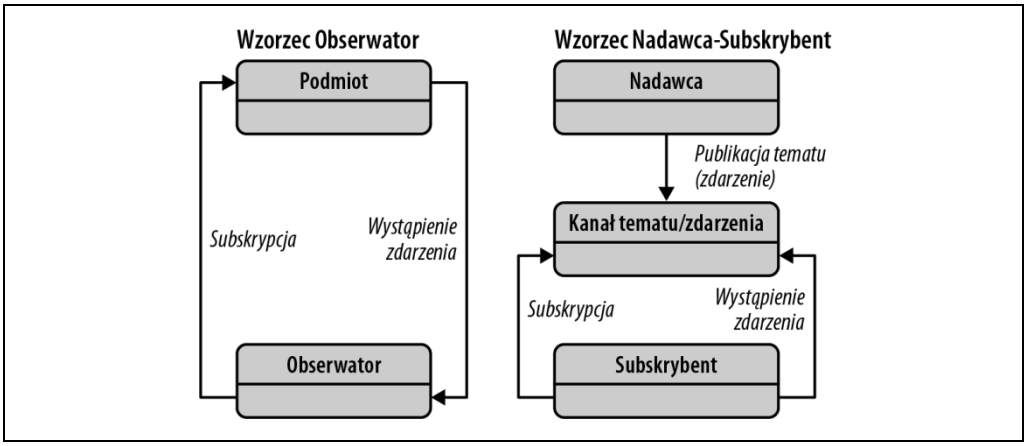
W tym przykładzie pokazałem, jak można zaimplementować i wykorzystać wzorzec Obserwator. Omówiłem przy tym koncepcje podmiotu, obserwatora, `ConcreteSubject` i `ConcreteObserver`.

Różnice między wzorcem Obserwator i wzorcem Nadawca-Subskrybent

Wprawdzie dobrze jest znać wzorzec Obserwator, ale w świecie JavaScriptu często można się spotkać z implementacją jego wariantu znaną jako wzorzec Nadawca-Subskrybent. Te dwa wzorce są do siebie bardzo podobne, choć między nimi występują też różnice, o których warto wiedzieć.

Wzorzec Obserwator wymaga, aby obserwator (obiektu), który chce otrzymywać powiadomienia na dany temat, został zarejestrowany w obiekcie wysyłającym te powiadomienia (podmiot), jak widać na rysunku 7.10.

Jednak wzorzec Nadawca-Subskrybent używa kanału tematu/zdarzenia znajdującego się między obiektami, które chcą otrzymywać powiadomienia (subskrybenty), i obiektem publikującym zdarzenie (nadawca). Taki system zdarzeń pozwala zdefiniować w kodzie zdarzenia typowe dla danej aplikacji, które mogą przekazywać niestandardowe argumenty, zawierające z kolei wartości wymagane przez subskrybenty. Pomysł polega tutaj na unikaniu zależności między nadawcą i subskrybentem.



Rysunek 7.10. Wzorzec Nadawca-Subskrybent

Mamy tu różnicę względem wzorca Obserwator, ponieważ dowolny subskrybent może implementować odpowiednią procedurę obsługi zdarzeń do rejestracji i otrzymywania powiadomień o zdarzeniach wyemitowanych przez nadawcę.

Oto przykład użycia wzorca Nadawca-Subskrybent w połączeniu z funkcjonalną implementacją w tle `publish()`, `subscribe()` i `unsubscribe()`:

```
<!-- Ten kod w HTML-u umieść na stronie -->
<div class="messageSender"></div>
<div class="messagePreview"></div>
<div class="newMessageCounter"></div>
// Prosta implementacja wzorca Nadawca-Subskrybent
const events = (function () {
  const topics = {};
  const hOP = topics.hasOwnProperty;

  return {
    subscribe: function (topic, listener) {
      if (!hOP.call(topics, topic)) topics[topic] = [];
      const index = topics[topic].push(listener) - 1;

      return {
        remove: function () {
          delete topics[topic][index];
        },
      };
    },

    publish: function (topic, info) {
      if (!hOP.call(topics, topic)) return;
      topics[topic].forEach(function (item) {
        item(info !== undefined ? info : {});
      });
    },
  };
})();
```

```
// Bardzo prosta procedura obsługi nowej poczty elektronicznej
// Otrzymana zostaje liczba wiadomości
let mailCounter = 0;

// Inicjalizacja subskrybentów, którzy będą nasłuchiwali naszego
// tematu o nazwie "inbox/newMessage"
// Wygenerowanie podglądu nowej wiadomości
const subscriber1 = events.subscribe('inbox/newMessage', (data) => {
  // Zarejestrowanie tematu na potrzeby procesu debugowania
  console.log('Otrzymano nową wiadomość:', data);

  // Wykorzystanie danych otrzymanych z podmiotu do
  // wyświetlenia użytkownikowi podglądu wiadomości
  document.querySelector('.messageSender').innerHTML = data.sender;
  document.querySelector('.messagePreview').innerHTML = data.body;
});

// Oto inny subskrybent używający tych samych danych
// do wykonania zupełnie innego zadania
// Uaktualnienie wartości licznika wyświetlającego liczbę nowych
// wiadomości otrzymanych poprzez nadawcę
const subscriber2 = events.subscribe('inbox/newMessage', (data) => {
  document.querySelector('.newMessageCounter').innerHTML = ++mailCounter;
});

events.publish('inbox/newMessage', {
  sender: 'hello@google.com',
  body: 'Witaj! Jak się dzisiaj masz?',
});

// Subskrybent może później zrezygnować z otrzymywania
// powiadomień, wystarczy następujące wywołanie:
// subscriber1.remove();
// subscriber2.remove();
```

Ogólna idea polega tutaj na promowaniu luźnego powiązania. Zamiast bezpośrednio wywoływać metody w innych obiektach, pojedyncze obiekty subskrybują informacje na temat określonych zadań lub aktywności innych obiektów i są o tym powiadamiane.

Zalety

Wzorce Obserwator i Nadawca-Subskrybent zachęcają do głębszego zastanowienia się na związkami zachodzącymi między różnymi fragmentami aplikacji. Ponadto pomagają w identyfikacji warstw zawierających bezpośrednie związki, które mogą być zastąpione zbiorami podmiotów i obserwatorów. To w praktyce można wykorzystać w celu podziału aplikacji na mniejsze i luźniej powiązane ze sobą bloki, co z kolei usprawnia zarządzanie kodem i zwiększa możliwość jego wielokrotnego używania.

Kolejną motywację do stosowania wzorca Obserwator może stanowić sytuacja, w której zachodzi potrzeba zapewnienia spójności między powiązanymi ze sobą obiektami bez powodowania zbyt silnego powiązania klas ze sobą. Na przykład obiekt musi otrzymywać powiadomienia dotyczące innych obiektów, ale nie powinien przyjmować żadnych założeń dotyczących tych obiektów.

Gdy używany jest dowolny z dwóch omawianych tutaj wzorców, między obserwatorami i podmiotami mogą istnieć relacje dynamiczne. To zapewnia doskonałą elastyczność, której implementacja nie musi być wcale taka łatwa, gdy odmienne części aplikacji są ze sobą ściśle powiązane.

Wprawdzie to nie zawsze będzie najlepsze rozwiązanie dla każdego problemu, ale omówione tutaj wzorce pozostają jednymi z najlepszych narzędzi do projektowania niepowiązanych ściśle ze sobą systemów. Powinny stanowić niezbędne narzędzia w zestawie każdego programisty JavaScriptu.

Wady

W związku z informacjami zamieszczonymi w poprzednim punkcie można stwierdzić, że niektóre problemy powodowane przez omawiane tutaj wzorce tak naprawdę wynikają w ich najważniejszych zalet. W podejściu nadawca-subskrybent oddzielenie nadawców od subskrybentów może czasami skutkować trudnościami w zagwarantowaniu, że poszczególne fragmenty aplikacji będą funkcjonowały zgodnie z oczekiwaniami.

Na przykład nadawca może przyjmować założenie, że jego powiadomienia są nasłuchiwane przez jednego lub więcej subskrybentów. Powiedzmy, że takie założenie jest używane do rejestrowania bądź wyświetlania błędów dotyczących procesów aplikacji. Jeżeli subskrybent przeprowadzający taką rejestrację danych ulegnie awarii (lub z jakiegokolwiek powodu nie będzie w stanie funkcjonować), wówczas nadawca nie będzie mógł wiedzieć o tych problemach, co wynika z luźnego połączenia komponentów systemu.

Inną wadą tego wzorca jest to, że subskrybenty zupełnie nic nie wiedzą o sobie nawzajem i nie mają świadomości kosztu zmiany nadawcy. Ze względu na dynamiczną naturę relacji między subskrybentami i nadawcami śledzenie zależności uaktualnień może być trudne.

Implementacje

Wzorzec Nadawca-Subskrybent świetnie wpisuje się w ekosystem JavaScriptu, przede wszystkim dlatego, że implementacje ECMAScript bazują na zdarzeniach. To dotyczy w szczególności środowisk przeglądarek WWW, ponieważ model DOM używa zdarzeń jako podstawowego API współpracy ze skryptami.

Mając to na uwadze, należy pamiętać, że ani ECMAScript, ani model DOM nie dostarczają podstawowych obiektów lub metod przeznaczonych do tworzenia własnych systemów zdarzeń w kodzie implementacji. (Wyjątkiem może być tutaj interfejs `CustomEvent` w DOM3, który jest dołączony do modelu DOM i tym samym nie ma ogólnego zastosowania).

Przykład implementacji wzorca Nadawca-Subskrybent

Aby lepiej docenić, jak wiele zwykłych implementacji wzorca Obserwator może działać w JavaScripcie, zapoznaj się z minimalistyczną wersją wzorca Nadawca-Subskrybent, którą opublikowałem w serwisie GitHub w postaci projektu o nazwie pubsubz (*https://github.com/addyosmani/pubsubz*). Celem tego projektu jest pokazanie podstawowych koncepcji nadawcy i subskrybenta oraz idei rezygnacji ze subskrypcji.

Zdecydowałem się przedstawić przykłady, bazując na tym kodzie, ponieważ pozostaje on ściśle zgodny z sygnaturami metod i podejściem do implementacji, które spodziewam się spotkać w klasycznym wzorcu Obserwator w wersji dla JavaScriptu.

```
class PubSub {
    constructor() {
        // Pamięć masowa dla tematów, które będą publikowane
        // lub nasłuchiwane
        this.topics = {};

        // Identyfikator tematu
        this.subUid = -1;
    }

    publish(topic, args) {
        if (!this.topics[topic]) {
            return false;
        }

        const subscribers = this.topics[topic];
        let len = subscribers ? subscribers.length : 0;

        while (len--) {
            subscribers[len].func(topic, args);
        }

        return this;
    }

    subscribe(topic, func) {
        if (!this.topics[topic]) {
            this.topics[topic] = [];
        }

        const token = (++this.subUid).toString();
        this.topics[topic].push({
            token,
            func,
        });

        return token;
    }

    unsubscribe(token) {
        for (const m in this.topics) {
            if (this.topics[m]) {
                for (let i = 0, j = this.topics[m].length; i < j; i++) {
                    if (this.topics[m][i].token === token) {
                        this.topics[m].splice(i, 1);

                        return token;
                    }
                }
            }
        }
        return this;
    }
}
```

```
const pubsub = new PubSub();

pubsub.publish('/addFavorite', ['test']);
pubsub.subscribe('/addFavorite', (topic, args) => {
    console.log('test', topic, args);
});
```

W tym kodzie została zdefiniowana prosta klasa `PubSub`, która zawiera:

- Listę tematów, które mogą być subskrybowane przez subskrybenty.
- Metodę `Subscribe`, która umożliwia utworzenie nowego subskrybenta dla tematu z użyciem funkcji wywoływanej podczas publikowania tematu i unikatowego tokenu.
- Metodę `Unsubscribe`, która usuwa subskrybenta z listy na podstawie przekazanej wartości `token`. Metoda `Publish` publikuje dotyczącą danego tematu treść dla wszystkich subskrybentów przez wywołanie funkcji `registered`.

Stosowanie przedstawionej implementacji

Tę implementację można teraz wykorzystać do publikowania i subskrybowania zdarzeń, jak pokazałem na listingu 7.7.

Listing 7.7. Przykład użycia implementacji

```
// Inny przykład prostej procedury obsługi komunikatów

// Prosty komponent rejestrowania komunikatów, zapisujący wszelkie
// tematy i dane otrzymane przez subskrybenta
const messageLogger = (topics, data) => {
    console.log(`Rejestracja danych: ${topics}: ${data}`);
};

// Subskrybent nasłuchuje interesujących go tematów oraz wykonuje
// funkcję wywołania zwrotnego (np. messageLogger) po otrzymaniu
// powiadomienia na dany temat
const subscription = pubsub.subscribe('inbox/newMessage', messageLogger);

// Nadawca jest odpowiedzialny za publikowanie tematów lub powiadomień
// interesujących aplikację, np.:

pubsub.publish('inbox/newMessage', 'Witaj, świecie!');

// Ewentualnie:
pubsub.publish('inbox/newMessage', ['test', 'a', 'b', 'c']);

// Ewentualnie:
pubsub.publish('inbox/newMessage', {
    sender: 'hello@google.com',
    body: 'Witaj ponownie!',
});

// Jeżeli subskrybent nie chce już otrzymywać powiadomień,
// może zrezygnować ze subskrypcji
pubsub.unsubscribe(subscription);
```

```
// Po zrezygnowaniu z subskrypcji ten przykład nie będzie powodował
// wykonywania messageLogger, ponieważ subskrybent już nie
// nasłuchuje zdarzeń
pubsub.publish('inbox/newMessage', 'Witaj! Wciąż tu jesteś?');
```

Interfejs użytkownika powiadomień

Załóżmy że mamy aplikację internetową, której zadaniem jest wyświetlanie w czasie rzeczywistym informacji o notowaniach giełdowych.

Taka aplikacja może mieć siatkę przeznaczoną do wyświetlania giełdowych danych statystycznych oraz komunikat (licznik) wskazujący datę i godzinę ostatniego uaktualnienia danych. Aplikacja musi uaktualnić siatkę i licznik po każdej zmianie modelu danych. W takim scenariuszu nasz podmiot (odpowiedzialny za publikowanie tematów i powiadomień) jest modelem danych, natomiast subskrybentami są siatka i licznik.

Kiedy subskrybenty zostają powiadomione o zmianie w modelu, odpowiednio się uaktualniają.

W przedstawionej tutaj implementacji subskrybent będzie nasłuchiwał tematu `newDataAvailable`, aby dowiedzieć się o dostępności nowych informacji giełdowych. Jeżeli zostanie opublikowane nowe powiadomienie w tym temacie, spowoduje, że `gridUpdate` dołączy nowy wiersz do siatki zawierającej informacje. Uaktualniony zostanie także licznik *Ostania aktualizacja*, aby zarejestrować datę i godzinę dodania najnowszych danych (zobacz listing 7.8).

Listing 7.8. Interfejs użytkownika powiadomień

```
// Zwrot bieżącego czasu lokalnego, który później zostanie użyty w interfejsie użytkownika
getCurrentTime = () => {
    const date = new Date();
    const m = date.getMonth() + 1;
    const d = date.getDate();
    const y = date.getFullYear();
    const t = date.toLocaleTimeString().toLowerCase();

    return `${m}/${d}/${y} ${t}`;
  };

  // Dodanie nowego wiersza danych do fikcyjnego komponentu siatki
  const addGridRow = data => {
    // ui.grid.addRow( data );
    console.log(`updated grid component with:${data}`);
  };

  // Uaktualnienie fikcyjnego komponentu siatki, aby wyświetlił
  // datę i godzinę jego ostatniej aktualizacji
  const updateCounter = data => {
    // ui.grid.updateLastChanged( getCurrentTime() );
    console.log(`data last updated at: ${getCurrentTime()} with ${data}`);
  };

  // Uaktualnienie siatki za pomocą danych przekazanych subskrybentom
  const gridUpdate = (topic, data) => {
    if (data !== undefined) {
      addGridRow(data);
```

```
      updateCounter(data);
    }
  };

// Utworzenie subskrypcji nowego tematu newDataAvailable
const subscriber = pubsub.subscribe('newDataAvailable', gridUpdate);

// Poniższy kod przedstawia uaktualnienia warstwy danych. Można tutaj wykorzystać
// żądania wykonywane w technologii AJAX, które będą pozostałą część aplikacji
// powiadamiały o dostępności nowych danych.

// Publikowanie zmian do tematu gridUpdated przedstawiającego nowe wpisy
pubsub.publish('newDataAvailable', {
  summary: 'Firma Apple zarobiła 5 miliardów dolarów.',
  identifier: 'APPL',
  stockPrice: 570.91,
});

pubsub.publish('newDataAvailable', {
  summary: 'Firma Microsoft zarobiła 20 milionów dolarów.',
  identifier: 'MSFT',
  stockPrice: 30.85,
});
```

Ograniczenie powiązania komponentów aplikacji za pomocą implementacji Pub/Sub opracowanej przez Bena Almana

W kolejnym przykładzie, przeznaczonym do oceny filmów, wykorzystamy implementację wzorca Nadawca-Subskrybent opracowaną w jQuery przez Bena Almana (*https://gist.github.com/cowboy/661855*). Ten kod pokazuje, jak można uniknąć powiązania komponentów w interfejsie użytkownika. Zwróć uwagę, że przekazanie oceny ma jedynie skutek w postaci opublikowania powiadomienia informującego o dostępności nowych danych.

Subskrybentom tych tematów pozostawiono ustalenie, co będzie się działo z danymi. W omawianym przykładzie nowe dane zostają przekazane do istniejących tablic, a następnie wygenerowane za pomocą metody `.template()` biblioteki *Lodash* w celu użycia w szablonie.

Kod szablonu w HTML-u przedstawiłem na listingu 7.9.

Listing 7.9. Kod szablonu w HTML-u dla omawianego przykładu wzorca Nadawca-Subskrybent

```
<script id="userTemplate" type="text/html">
   <li><%- name %></li>
</script>

<script id="ratingsTemplate" type="text/html">
   <li><strong><%- title %></strong> został oceniony na <%- rating %>/5</li>
</script>

<div id="container">
   <div class="sampleForm">
       <p>
           <label for="twitter_handle">Twitter:</label>
           <input type="text" id="twitter_handle" />
       </p>
```

```
        <p>
            <label for="movie_seen">Podaj tytuł filmu, który chcesz ocenić:</label>
            <input type="text" id="movie_seen" />
        </p>
        <p>
            <label for="movie_rating">Oceń ten film:</label>
            <select id="movie_rating">
                  <option value="1">1</option>
                   <option value="2">2</option>
                   <option value="3">3</option>
                   <option value="4">4</option>
                   <option value="5" selected>5</option>
              </select>
           </p>
           <p>
               <button id="add">Wyślij ocenę</button>
           </p>
      </div>

      <div class="summaryTable">
          <div id="users"><h3>Najnowsi użytkownicy</h3></div>
          <div id="ratings"><h3>Najnowsze oceny filmów</h3></div>
      </div>
  </div>
```

Kod w JavaScripcie dla tego przykładu został zamieszczony na listingu 7.10.

Listing 7.10. Kod w JavaScripcie dla omawianego przykładu wzorca Nadawca-Subskrybent

```
;($ => {
    // Przygotowane szablony i ich "buforowanie" za pomocą domknięcia
    const userTemplate = _.template($('#userTemplate').html());

    const ratingsTemplate = _.template($('#ratingsTemplate').html());

    // Subskrypcja nowego tematu użytkownika powodująca dodanie użytkownika
    // do listy użytkowników, którzy wysłali ocenę filmu
    $.subscribe('/new/user', (e, data) => {
      if (data) {
        $('#users').append(userTemplate(data));
      }
    });

    // Subskrypcja nowego tematu oceny, składa się z tytułu i oceny filmu
    // Nowe oceny są dołączane do listy ocen dodanych przez użytkowników
    $.subscribe('/new/rating', (e, data) => {
      if (data) {
        $('#ratings').append(ratingsTemplate(data));
      }
    });

    // Możliwość dodania nowego użytkownika
    $('#add').on('click', e => {
      e.preventDefault();

      const strUser = $('#twitter_handle').val();
      const strMovie = $('#movie_seen').val();
      const strRating = $('#movie_rating').val();
```

```
    // Powiadomienie aplikacji o dostępności nowego użytkownika
    $.publish('/new/user', {
      name: strUser,
    });

    // Powiadomienie aplikacji o dostępności nowej oceny
    $.publish('/new/rating', {
      title: strMovie,
      rating: strRating,
    });
  });
})(jQuery);
```

Ograniczenie powiązania komponentów aplikacji jQuery bazującej na technologii AJAX

W ostatnim przykładzie pokażę, jak uniknąć powiązania komponentów kodu, używając wzorca Nadawca-Subskrybent już na wczesnym etapie tworzenia kodu źródłowego. Dzięki temu będzie można uniknąć konieczności późniejszej refaktoryzacji, która może być bolesnym procesem.

W aplikacjach intensywnie wykorzystujących technologię AJAX często chcemy osiągnąć więcej niż jedną unikatową akcję po otrzymaniu odpowiedzi na żądanie. Wprawdzie całą logikę obsługi po żądaniu można dodać do wywołania zwrotnego wykonywanego w przypadku sukcesu operacji, ale takie podejście ma pewne wady.

W przypadku aplikacji o ściśle powiązanych komponentach czasami zwiększa się wysiłek konieczny do wielokrotnego wykorzystania funkcji, co ma związek z większym poziomem zależności w kodzie. Umieszczenie w wywołaniu zwrotnym logiki wykonywanej po żądaniu może wydawać się dobrym pomysłem, o ile próba pobrania zbioru wynikowego odbędzie się tylko jednokrotnie. Jednak nie wydaje się odpowiednia, gdy mają być wykonane kolejne wywołania AJAX do tego samego źródła danych (inny sposób działania) bez konieczności wielokrotnego przepisywania fragmentów kodu. Zamiast przechodzić przez poszczególne warstwy wywołujące to samo źródło danych i uogólnić je później, już od samego początku można wykorzystać wzorzec Nadawca-Subskrybent i tym samym oszczędzić sobie nieco czasu.

Używając obserwatorów, można również łatwo oddzielić dotyczące różnych zdarzeń powiadomienia o zasięgu aplikacji do niezbędnego poziomu szczegółowości — jeżeli do tego celu użyjesz innych wzorców, rozwiązanie może być mniej eleganckie.

W kolejnym omawianym przykładzie zwróć uwagę na to, jak jedno powiadomienie o temacie zostaje wygenerowane, gdy użytkownik wskazuje, że chce wykonać żądanie wyszukiwania. Inne powiadomienie jest wykonywane po zakończeniu żądania i pojawieniu się danych, które można do czegoś wykorzystać. Subskrybentom pozostawiono decyzję o sposobach wykorzystania wiedzy dotyczącej tych zdarzeń (lub zwróconych danych). Zalety są takie, że jeśli chcemy, to możemy mieć 10 różnych subskrybentów, które na różne sposoby będą używać zwróconych danych. Z perspektywy warstwy technologii AJAX to nie ma żadnego znaczenia. Jej celem jest po prostu wykonanie żądania, zwrócenie danych i ich przekazanie tam, gdzie mają być używane. Taki podział obowiązków powoduje, że ogólny projekt kodu staje się nieco bardziej przejrzysty.

Kod szablonu w HTML-u został przedstawiony na listingu 7.11.

Listing 7.11. Kod szablonu w HTML-u w przykładzie wykorzystującym technologię AJAX

```
<form id="flickrSearch">
   <input type="text" name="tag" id="query"/>
   <input type="submit" name="submit" value="submit"/>
</form>

<div id="lastQuery"></div>
<ol id="searchResults"></ol>

<script id="resultTemplate" type="text/html">
    <% _.each(items, function( item ){ %>
        <li><img src="<%= item.media.m %>"/></li>
    <% });%>
</script>
```

Kod w JavaScripcie omawianego przykładu znajduje się na listingu 7.12.

Listing 7.12. Kod w JavaScripcie przykładu wykorzystującego technologię AJAX

```
($ => {
    // Przygotowany szablon i jego "buforowanie" za pomocą domknięcia
    const resultTemplate = _.template($('#resultTemplate').html());

    // Subskrypcja tematu nowych tagów wyszukiwania
    $.subscribe('/search/tags', (e, tags) => {
      $('#lastQuery').html(`Szukano: ${tags}`);
    });

    // Subskrypcja tematu nowych wyników wyszukiwania
    $.subscribe('/search/resultSet', (e, results) => {
      $('#searchResults')
        .empty()
        .append(resultTemplate(results));
    });

    // Wysłanie zapytania wyszukiwania i publikowanie tagów w tematach dotyczących wyszukiwania i tagów
    $('#flickrSearch').submit(function(e) {
      e.preventDefault();
      const tags = $(this)
        .find('#query')
        .val();
      if (!tags) {
        return;
      }

      $.publish('/search/tags', [$.trim(tags)]);
    });

    // Subskrypcja publikacji nowych tagów i wykonanie zapytania wyszukiwania,
    // które je wykorzystuje. Po zwróceniu danych są one publikowane dla pozostałej
    // części aplikacji w celu dalszego ich użycia. Wykorzystano składnię przypisania
    // destrukturalnego, która pozwala rozpakować wartości ze struktur danych
    // i umieścić je w oddzielnych zmiennych.
    $.subscribe('/search/tags', (e, tags) => {
      $.getJSON(
        'http://api.flickr.com/services/feeds/photos_public.gne?jsoncallback=?',
        {
```

```
        tags,
        tagmode: 'any',
        format: 'json',
      },
      // Przypisanie destrukturalne jako parametr funkcji
      ({ items }) => {
        if (!items.length) {
          return;
        }
        // Skrót nazw właściwości podczas tworzenia obiektu,
        // jeżeli nazwa zmiennej odpowiada kluczowi obiektu
        $.publish('/search/resultSet', { items });
      }
    );
  });
})(jQuery);
```

Wzorzec Obserwator w ekosystemie Reacta

Popularną biblioteką używającą wzorca obserwacji jest *RxJS*. W jej dokumentacji (*https://rxjs.dev/guide/overview*) można znaleźć następujące stwierdzenie:

> ReactiveX łączy wzorce obserwatora i iteratora oraz programowanie funkcyjne z kolekcjami, aby w ten sposób wypełnić lukę dla idealnego sposobu na zarządzanie sekwencjami zdarzeń.

Dzięki bibliotece *RxJS* można tworzyć obserwatory i subskrybować zdarzenia. Spójrz na pochodzący ze wspomnianej dokumentacji przykład, w którym zostaje zarejestrowane, który użytkownik przeciągnął dany dokument:

```
import ReactDOM from "react-dom";
import { fromEvent, merge } from "rxjs";
import { sample, mapTo } from "rxjs/operators";

import "./styles.css";

merge(
  fromEvent(document, "mousedown").pipe(mapTo(false)),
  fromEvent(document, "mousemove").pipe(mapTo(true))
)
  .pipe(sample(fromEvent(document, "mouseup")))
  .subscribe(isDragging => {
    console.log("Czy przeciągasz element?", isDragging);
  });

ReactDOM.render(
  <div className="App">Kliknij lub przeciągnij gdziekolwiek, a następnie sprawdź w konsoli!</div>,
  document.getElementById("root")
);
```

Wzorzec Obserwator pozwala w wielu różnych sytuacjach ograniczyć zależność między komponentami aplikacji. Jeżeli jeszcze z niego nie korzystasz, zachęcam do sięgnięcia po jedną z gotowych implementacji, o których tutaj wspomniałem, i jej wypróbowania. Jest to jeden z wzorców projektowych, z którymi łatwo rozpocząć pracę i który jednocześnie oferuje największe możliwości.

Wzorzec Mediator

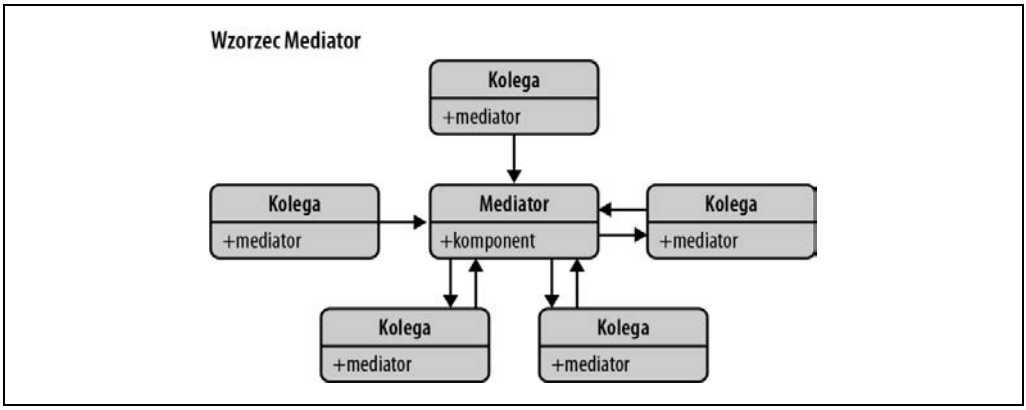
Mediator to wzorzec projektowy pozwalający obiektowi powiadomić zbiór innych obiektów o wystąpieniu zdarzenia. Różnica między wzorcami Mediator i Obserwator polega na tym, że w tym pierwszym jeden obiekt może być powiadomiony o zdarzeniach występujących w innych obiektach, w drugim zaś jeden obiekt może subskrybować wiele zdarzeń występujących w innych obiektach.

Omawiając wzorzec Obserwator, wspomniałem o sposobie kierowania wielu źródeł zdarzeń przez pojedynczy obiekt. Jest to znane pod nazwą podejścia typu nadawca-subskrybent lub agregacji zdarzeń. Bardzo często zdarza się, że programiści myślą o mediatorach, gdy stają przed tym problemem. Warto więc wyjaśnić różnice między wymienionymi wzorcami.

Zgodnie z definicją w słowniku mediator to bezstronny podmiot biorący udział w negocjacjach i rozwiązywaniu konfliktów[1]. W świecie programowania mediator to operacyjny wzorzec projektowy umożliwiający udostępnienie ujednoliconego interfejsu, za pomocą którego różne części systemu mogą się komunikować.

Jeżeli wydaje się, że system ma zbyt wiele bezpośrednich związków między komponentami, być może właściwym podejściem będzie utworzenie centralnego punktu kontroli umożliwiającego komponentom komunikowanie się. Mediator promuje luźne połączenie przez zagwarantowanie centralnego zarządzania interakcjami między komponentami. Komponenty nie muszą się bezpośrednio do siebie odwoływać. To pomaga ograniczyć zależność systemu oraz zwiększyć możliwości wielokrotnego używania komponentów.

Analogią ze świata rzeczywistego może być typowy system kontroli ruchu na lotnisku. Wieża (mediator) pomaga samolotom w startach i lądowaniach, ponieważ cała komunikacja (nasłuchiwane powiadomień i ich rozgłaszanie) odbywa się z samolotu do wieży kontrolnej, a nie między samolotami. Scentralizowany kontroler ma znaczenie kluczowe dla osiągnięcia sukcesu w takim systemie. Dokładnie taką rolę odgrywa mediator w dziedzinie tworzenia oprogramowania (zobacz rysunek 7.11).



Rysunek 7.11. Wzorzec Mediator

[1] Zobacz w Wikipedii (*https://pl.wikipedia.org/wiki/Mediacja*) i Dictionary.com (*https://www.dictionary.com/browse/mediator*).

Inną analogią będzie propagowanie zdarzeń modelu DOM i delegowanie zdarzeń. Jeżeli wszystkie subskrypcje w systemie dotyczą dokumentu zamiast poszczególnych węzłów, wówczas dokument jest w zasadzie mediatorem. Zamiast dołączać do zdarzeń poszczególnych węzłów, wysokiego poziomu obiekt staje się odpowiedzialny za powiadamianie subskrybentów o zdarzeniach.

Czasami może się wydawać, że wzorców Mediator i Agregacja Zdarzeń można używać wymiennie ze względu na podobieństwa w implementacji. Jednak ich semantyka i przeznaczenie całkowicie się różnią.

Nawet jeśli implementacje obu w pewnym stopniu używają tych samych podstawowych konstrukcji, jestem przekonany o istnieniu ważnych różnic między nimi. Nie powinny być stosowane zamiennie ani mylone podczas komunikacji — właśnie z powodu tych różnic.

Prosty mediator

Mediator to obiekt koordynujący interakcje (logikę i sposób działania) między wieloma obiektami. Decyduje o tym, kiedy wywoływać określone obiekty, na podstawie akcji (bądź ich braku) innych obiektów i danych wejściowych.

Do zapisania mediatora wystarczy pojedynczy wiersz kodu:

```
const mediator = {};
```

Oczywiście to jest tylko literał obiektu w JavaScripcie. Warto podkreślić, że mówimy tutaj o semantyce. Celem mediatora jest kontrolowanie sposobu komunikacji między obiektami. Do tego naprawdę nie potrzeba niczego więcej poza literałem obiektu.

W kolejnym fragmencie kodu przedstawiłem prostą implementację obiektu `mediator` razem z metodami narzędziowymi, które mogą wywoływać i subskrybować zdarzenia. Obiekt `orgChart` jest tutaj mediatorem przypisującym akcje, jakie powinny zostać podjęte w przypadku wystąpienia określonego zdarzenia. W omawianym przykładzie menedżer zostaje przypisany pracownikowi w celu wypełnienia szczegółów dotyczących nowego pracownika, a następnie rekord informacji o pracowniku zostaje zapisany:

```
const orgChart = {
    addNewEmployee() {
        // Metoda getEmployeeDetail() dostarcza widok, z którym będą pracować użytkownicy
        const employeeDetail = this.getEmployeeDetail();

        // Po wypełnieniu szczegółów dotyczących pracownika mediator (tutaj obiekt 'orgchart')
        // decyduje, co powinno później nastąpić
        employeeDetail.on('complete', employee => {
            // Skonfigurowanie obiektów dodatkowych, które mają kolejne zdarzenia, które z kolei
            // są używane przez mediatora do wykonywania jeszcze innych zadań
            const managerSelector = this.selectManager(employee);
            managerSelector.on('save', employee => {
                employee.save();
            });
        });
    },

    // ...
};
```

W przeszłości ten typ obiektu często określałem mianem obiektu „sposobu pracy”, choć tak naprawdę to jest mediator. Ten obiekt zajmuje się obsługą sposobu pracy między innymi obiektami oraz agregacją w pojedynczym obiekcie odpowiedzialności za tę wiedzę. W efekcie otrzymujemy sposób pracy, który jest łatwiejszy do zrozumienia i późniejszej obsługi technicznej.

Podobieństwa i różnice

Nie ulega wątpliwości, że istnieją podobieństwa między przedstawionymi w tym rozdziale przykładami wzorców Agregator Zdarzeń i Mediator. Sprowadzają się one do dwóch podstawowych aspektów: zdarzeń i obiektów podmiotów zewnętrznych. Jednak te różnice są co najwyżej niewielkie. Gdy zagłębimy się w cel wzorców i zobaczymy, że implementacje mogą być bardzo odmienne, wówczas natura tych wzorców stanie się bardziej oczywista.

Zdarzenia

Zarówno wzorzec Agregator Zdarzeń, jak i Mediator używają zdarzeń w przedstawionych tutaj przykładach. Jest oczywiste, że Agregator Zdarzeń używa zdarzeń — na to przecież wskazuje jego nazwa. Z kolei Mediator używa zdarzeń tylko dlatego, że ułatwiają one zadanie podczas pracy z nowoczesnymi frameworkami JavaScriptu przeznaczonymi do tworzenia aplikacji internetowych. Nie ma żadnej reguły, zgodnie z którą mediator musiałby być zbudowany w oparciu o zdarzenia. Mediatora można zbudować z użyciem metod wywołania zwrotnego przez przekazanie odwołania do mediatora obiektowi potomnemu bądź też wykorzystać inne rozwiązania.

Zatem różnica polega na tym, dlaczego oba te wzorce używają zdarzeń. Agregator Zdarzeń, jako wzorzec, został opracowany do pracy ze zdarzeniami. Natomiast Mediator używa zdarzeń tylko dlatego, że takie rozwiązanie okazuje się wygodne.

Obiekty podmiotów zewnętrznych

Zgodnie z projektem agregator zdarzeń i mediator wykorzystują obiekt podmiotu zewnętrznego do usprawnienia interakcji. Agregator zdarzeń sam w sobie jest podmiotem zewnętrznym dla nadawcy zdarzeń i jego subskrybentów. Działa w charakterze centralnego huba przekazującego zdarzenia. Mediator jest również podmiotem zewnętrznym dla innych obiektów. Na czym więc polega różnica? Dlaczego agregatora zdarzeń nie określamy mianem mediatora? Odpowiedź na to pytanie zależy przede wszystkim od logiki aplikacji i zastosowanego w niej sposobu działania.

W przypadku agregatora zdarzeń obiekt podmiotu zewnętrznego ma na celu jedynie ułatwienie przekazywania zdarzeń z nieznanej liczby źródeł do nieznanej liczby procedur obsługi. Cały sposób pracy i logika konieczna do wykonania będą umieszczone bezpośrednio w obiekcie wywołującym zdarzenia oraz w obiektach obsługujących te zdarzenia.

W przypadku mediatora logika biznesowa i sposób pracy są zagregowane w samym mediatorze. To mediator decyduje, czy obiekt powinien wywołać swoje metody i zaktualizować atrybuty na podstawie czynników znanych mediatorowi. Hermetyzuje sposób pracy i procesy, koordynuje wiele obiektów w celu zapewnienia oczekiwanego działania systemu. Poszczególne obiekty zaangażowane w pracę wiedzą, jak mają wykonywać swoje zadania. Jednak to mediator wskazuje tym

obiektom, kiedy mają wykonywać zadania, a tym samym decyzje są podejmowane na wyższym poziomie niż poszczególne obiekty.

Agregator zdarzeń ułatwia model komunikacji „uruchom i zapomnij". Dla obiektu wywołującego zdarzenie nie ma znaczenia, czy istnieją jakiekolwiek subskrybenty tego zdarzenia. Obiekt po prostu wywołuje zdarzenie i kontynuuje działanie. Mediator może używać zdarzeń do podejmowania decyzji, przy czym zdecydowanie to nie jest model typu „uruchom i zapomnij". Mediator zwraca uwagę na znany zbiór danych wejściowych lub aktywności, aby móc ułatwić i skoordynować inne działania z użyciem znanego zbioru aktorów (obiektów).

Relacje — kiedy nie używać agregatora zdarzeń lub mediatora?

Zrozumienie podobieństw i różnic między agregatorem zdarzeń i mediatorem jest ważne z powodów semantycznych. Naprawdę dobrze jest wiedzieć, kiedy należy używać tych wzorców. Podstawowa semantyka i cele tych wzorców prowadzą do pytania typu „kiedy?". Doświadczenie w stosowaniu tych wzorców pomoże zrozumieć subtelniejsze aspekty i niuanse decyzji, jakie należy podjąć.

Stosowanie agregatora zdarzeń

Ogólnie rzecz biorąc, agregator zdarzeń jest używany w sytuacji, gdy istnieje zbyt wiele obiektów do bezpośredniego nasłuchiwania bądź obiekty są zupełnie ze sobą niezwiązane.

Gdy między dwoma obiektami istnieje bezpośrednia relacja — np. widoki nadrzędny i potomny — wówczas korzystne może być użycie agregatora zdarzeń. Jeżeli widok potomny wygeneruje zdarzenie, widok nadrzędny będzie mógł je obsłużyć. Takie podejście jest najczęściej spotykane w kontekście kolekcji i modelu frameworka Backbone w JavaScripcie, w którym wszystkie zdarzenia modelu są przekazywane w górę aż do jego nadrzędnego elementu kolekcji. Z kolei kolekcja często używa zdarzeń modelu do modyfikowania stanu swojego bądź innych modeli. Doskonałym przykładem będzie tutaj obsługa „wybranych" elementów w kolekcji.

Metoda `on()` w bibliotece *jQuery* jako agregator zdarzeń to doskonały przykład sytuacji, w której istnieje zbyt wiele obiektów do nasłuchiwania. Jeżeli masz 10, 20 lub nawet 200 elementów modelu DOM, które mogą wywołać zdarzenie „kliknięcia", to kiepskim pomysłem będzie zdefiniowanie komponentów oddzielnie nasłuchujących każdego z nich. To może szybko drastycznie zmniejszyć wydajność działania aplikacji i zrujnować wrażenia jej użytkowników. Zamiast tego metoda `on()` biblioteki *jQuery* pozwala na agregację wszystkich zdarzeń i zmniejszenie obciążenia z 10, 20 lub nawet 200 procedur obsługi zdarzeń do zaledwie jednej.

Relacje pośrednie to również doskonały przykład stosowania agregatorów zdarzeń. W nowoczesnej aplikacji wszechobecne jest posiadanie wielu obiektów widoku, które muszą się komunikować, ale nie mają bezpośredniej relacji. Na przykład menu systemowe może mieć widok odpowiedzialny za obsługę kliknięcia elementu menu. Jednak nie ma potrzeby, aby menu było bezpośrednio powiązane z widokiem treści wyświetlającym wszystkie szczegóły i informacje po kliknięciu elementu menu — połącznie treści i menu spowodowałoby, że w dłuższej perspektywie czasu taki kod byłby niezwykle trudny w obsłudze technicznej. Zamiast tego można więc użyć agregatora zdarzeń, wywoływać `menu:click:foo` i mieć obiekt „foo" obsługujący zdarzenie `click` i wyświetlający na ekranie związaną z nim treść.

Stosowanie mediatora

Najlepszym zastosowaniem mediatora jest sytuacja, gdy co najmniej dwa obiekty mają działającą relację pośrednią, a logika biznesowa lub sposób pracy wymagają dyktowania interakcji i koordynacji tych obiektów. Doskonałym przykładem będzie tutaj interfejs kreatora, który przedstawiłem w przykładzie `orgChart`. Wiele widoków ułatwia pracę kreatora. Zamiast ścisłego powiązania widoków przez bezpośrednie odwoływania między nimi, można je oddzielić, a wprowadzenie mediatora pozwala na jawniejszy model ich działania.

Mediator wyodrębnia ze szczegółów implementacji sposób działania i na wyższym poziomie tworzy bardziej naturalną abstrakcję, znacznie szybciej pokazując, na czym polega dany sposób pracy. Nie trzeba dłużej zagłębiać się w szczegóły poszczególnych widoków, aby odkryć sposób działania.

Połączenie mediatora i agregatora zdarzeń (nadawca-subskrybent)

Sedno różnicy między agregatorem zdarzeń i mediatorem oraz powód, dla którego nazw tych wzorców nie należy używać wymiennie, można najlepiej zilustrować przez pokazanie przykładu ich zastosowania razem. Przykład menu dla agregatora zdarzeń to doskonałe miejsce na wprowadzenie mediatora.

Kliknięcie elementu menu może wywołać serię zmian w aplikacji. Część z nich pozostanie niezależna od reszty, a użycie agregatora zdarzeń ma sens. Natomiast część zmian może być wewnętrznie powiązana ze sobą i do ich wprowadzenia może być wykorzystany mediator.

Mediator może być skonfigurowany do nasłuchiwania agregatora zdarzeń. Może wykonać swoją logikę i proces w celu ułatwienia i skoordynowania wielu powiązanych ze sobą obiektów, które jednak pozostają zupełnie bez związku z pierwotnym źródłem zdarzenia:

```
const MenuItem = MyFrameworkView.extend({
    events: {
        'click .thatThing': 'clickedIt',
    },

    clickedIt(e) {
        e.preventDefault();
        // Przyjęto założenie, że to spowoduje wywołanie "menu:click:foo"
        MyFramework.trigger(`menu:click:${this.model.get('name')}`);
    },
});

// Gdzieś w innym miejscu aplikacji

class MyWorkflow {
    constructor() {
        MyFramework.on('menu:click:foo', this.doStuff, this);
    }

    static doStuff() {
        // W tym miejscu następuje utworzenie wielu obiektów.
        // Zdefiniowane będą również procedury obsługi dla tych obiektów.
        // Ponadto odbywa się koordynacja wszystkich obiektów w sensowne rozwiązanie.
    }
}
```

W tym przykładzie po kliknięciu `MenuItem` za pomocą właściwego modelu zostanie wywołane zdarzenie `menu:click:foo`. Egzemplarz klasy `MyWorkFlow` obsłuży to konkretne zdarzenie i skoordynuje wszystkie obiekty, o których wie, że odpowiadają za utworzenie oczekiwanego przez użytkownika środowiska.

W ten sposób udało się połączyć agregatora zdarzeń i mediatora w celu przygotowania oczekiwanego środowiska zarówno w kodzie, jak i w aplikacji. Agregator zdarzeń był w stanie zapewnić czystą separację między menu i sposobem pracy, natomiast mediator zapewnił przejrzystość sposobu pracy i łatwość późniejszej obsługi technicznej rozwiązania.

Mediator i oprogramowanie pośredniczące w nowoczesnym JavaScripcie

Express.js (*https://expressjs.com/*) to popularny framework serwerowy przeznaczony do tworzenia aplikacji internetowych. Pozwala dodawać wywołania zwrotne do określonych tras, do których użytkownik może mieć dostęp.

Załóżmy, że ma być dodany nagłówek do żądania, gdy użytkownik dotrze do katalogu głównego (*/*). Taki nagłówek można dodać w wywołaniu zwrotnym oprogramowania pośredniczącego:

```
const app = require("express")();

app.use("/", (req, res, next) => {
  req.headers["test-header"] = 1234;
  next();
});
```

Metoda `next()` wywołuje następne wywołanie zwrotne w cyklu żądanie – odpowiedź. Można utworzyć łańcuch funkcji oprogramowania pośredniczącego znajdujący się między żądaniem i odpowiedzią na nie bądź na odwrót. Istnieje możliwość śledzenia i modyfikowania obiektu żądania przez całą drogę odpowiedzi przez jedną lub wiele funkcji oprogramowania pośredniczącego.

Wywołanie zwrotne oprogramowania pośredniczącego zostanie wywołane za każdym razem, gdy użytkownik dotrze do katalogu głównego (*/*):

```
const app = require("express")();
const html = require("./data");

  app.use(
    "/",
    (req, res, next) => {
      req.headers["test-header"] = 1234;
      next();
    },
    (req, res, next) => {
      console.log(`Żądanie ma nagłówek testowy: ${!!req.headers["test-header"]}`);
      next();
    }
  );

  app.get("/", (req, res) => {
    res.set("Content-Type", "text/html");
    res.send(Buffer.from(html));
  });
```

```
app.listen(8080, function() {
  console.log("Serwer nasłuchuje na porcie 8080");
});
```

Mediator kontra fasada

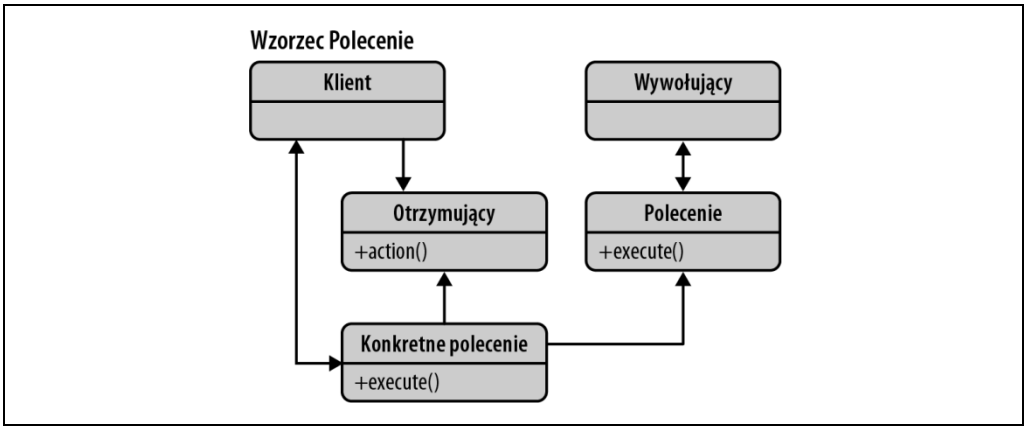
Niektórzy programiści mogą zastanawiać się nad podobieństwami między wzorcami Mediator i Fasada (wzorzec Fasada został omówiony we wcześniejszej części rozdziału). Oba służą do przygotowania abstrakcji dla istniejących modułów, choć między nimi istnieją drobne różnice.

Mediator centralizuje komunikację między modułami, gdy te moduły mają wyraźne odwołania do siebie. W takim sensie mamy do czynienia z wielokierunkowością. Z kolei Fasada definiuje znacznie prostszy interfejs do modułu lub systemu, ale nie dodaje żadnej dodatkowej funkcjonalności. Inne moduły w systemie nie wiedzą o koncepcji fasady i mogą być uznawane za jednokierunkowe.

Wzorzec Polecenie

Celem wzorca Polecenie jest hermetyzacja w pojedynczym obiekcie wywołań metod, żądań i operacji oraz umożliwienie zarówno parametryzacji, jak i przekazywania wywołań metod, które będą wykonywane według uznania programisty. Ponadto omawiany wzorzec pomaga oddzielić obiekty wywołujące akcje od implementujących je obiektów, a tym samym zapewnić większą elastyczność w zakresie zamiany *klas* konkretnych (obiektów).

Klasę *konkretną* można najlepiej wyjaśnić w kategoriach języka programowania opartego na klasach. Jest ona powiązana z ideą klasy abstrakcyjnej. Klasa *abstrakcyjna* definiuje interfejs, ale niekoniecznie dostarcza implementacje dla jej wszystkich funkcji składowych. Działa w charakterze klasy bazowej, po której dziedziczą inne klasy. Klasa pochodna implementująca brakującą funkcjonalność jest nazywana klasą *konkretną* (zobacz rysunek 7.12). Klasy bazowe i konkretne mogą być zaimplementowane w wydaniach ES2015+ JavaScriptu za pomocą słowa kluczowego `extends`, dostępnego dla wszystkich klas JavaScriptu.



Rysunek 7.12. Wzorzec Polecenie

Ogólna idea kryjąca się za wzorcem Polecenie polega na tym, że umożliwia on oddzielenie odpowiedzialności za wykonywanie poleceń od wszelkich komponentów wydających te polecenia, delegując odpowiedzialność do różnych obiektów.

Z perspektywy implementacji proste obiekty poleceń wiążą akcję i obiekt, który ma wywoływać tę akcję. Konsekwentnie są używane operacje wykonywania, takie jak `run()` i `execute()`. Wszystkie obiekty poleceń o takim samym interfejsie bardzo łatwo mogą być zastępowane, co jest jedną z największych zalet tego wzorca.

Aby zobaczyć wzorzec Polecenie w działaniu, spójrz na prostą usługę dotyczącą zakupu samochodu:

```
const CarManager = {
        // Żądanie informacji
        requestInfo(model, id) {
            return `Informacje dotyczące ${model} o identyfikatorze ${id} to foobar.`;
        },

        // Zakup samochodu
        buyVehicle(model, id) {
            return `Udało się dokonać zakupu samochodu o identyfikatorze ${id}, ${model}`;
        },

        // Umówienie się na jazdę próbną
        arrangeViewing(model, id) {
            return `Umówiono jazdę próbną samochodem ${model} ( ${id} ) `;
        },
    };
```

W omawianym przykładzie `CarManager` to obiekt polecenia odpowiedzialny za wykonywanie poleceń w celu żądania informacji o samochodzie, zakupu samochodu bądź umówienia się na jazdę próbną. Bardzo łatwe będzie wywoływanie metod `CarManager` dzięki uzyskaniu bezpośredniego dostępu do obiektu. Można wybaczyć założenie, że nie ma tutaj nic złego — formalnie rzecz biorąc, jest to poprawny kod w JavaScripcie. Jednak zdarzają się sytuacje, w których takie podejście może się okazać szkodliwe.

Na przykład wyobraź sobie, że zmianie uległo jedno z ważniejszych API kryjących się za `CarManager`. To wymagałoby modyfikacji wszystkich obiektów, które w aplikacji uzyskują bezpośredni dostęp do tego API. Taki rodzaj powiązania jest sprzeczny z metodologią programowania zorientowanego obiektowo, w której obiekty powinny być luźno ze sobą powiązane. Zamiast tego rozwiązaniem problemu będzie zastosowanie abstrakcji API.

Zobacz, jak można rozbudować klasę `CarManager`, aby aplikacja używająca wzorca Polecenie działała w następujący sposób: akceptowała dowolną nazwaną metodę przeznaczoną do wykonania w obiekcie `CarManager`, przekazując jednocześnie dane przeznaczone do użycia, np. model samochodu i jego identyfikator.

Oto efekt, jaki chcielibyśmy osiągnąć:

```
CarManager.execute('buyVehicle', 'Ford Escort', '453543');
```

Zgodnie z przedstawioną strukturą teraz trzeba dodać definicję metody `carManager.execute()`:

```
carManager.execute = function(name) {
    return (
        carManager[name] &&
        carManager[name].apply(carManager, [].slice.call(arguments, 1))
    );
};
```

Ostatecznie próbka wywołań będzie miała następującą postać:

```
carManager.execute('arrangeViewing', 'Ferrari', '14523');
carManager.execute('requestInfo', 'Ford Mondeo', '54323');
carManager.execute('requestInfo', 'Ford Escort', '34232');
carManager.execute('buyVehicle', 'Ford Escort', '34232');
```

Podsumowanie

W ten sposób zakończyłem omawianie tradycyjnych wzorców projektowych, które można wykorzystać podczas opracowywania klas, obiektów i modułów. Starałem się zastosować idealne połączenie wzorców projektowych różnych typów: konstrukcyjnych, strukturalnych i operacyjnych. Przedstawiłem wzorce projektowe przeznaczone dla klasycznych języków programowania zorientowanego obiektowo, takich jak Java i C++, oraz ich adaptacji na potrzeby JavaScriptu.

Wzorce te będą pomocne w opracowaniu wielu obiektów charakterystycznych dla domeny (np. koszyk na zakupy, pojazd, książka) pojawiających się w modelach biznesowych aplikacji. W następnym rozdziale spojrzysz z szerszej perspektywy na to, jak można nadać strukturę aplikacji, aby taki model zapewnił inne warstwy aplikacji, np. widoku lub prezentacyjną.

To obowiązkowa pozycja dla programistów myślących systemowo.

Sarah Drasner, dyrektor zespołu inżynierów w Google

JavaScript należy do najpopularniejszych i najwszechstronniejszych języków programowania na świecie. Rozwój JavaScriptu w ogromnym stopniu oddziałuje na sposoby tworzenia aplikacji internetowych. Z kolei na kwestię ich obsługi technicznej znaczący wpływ mają rozwiązania wybierane przez programistów. Okazuje się, że stosowanie nowoczesnych wzorców projektowych zdecydowanie poprawia komfort pracy z aplikacją na poszczególnych etapach cyklu jej rozwoju.

W tej książce omówiono ponad 20 najprzydatniejszych wzorców projektowych, dzięki którym tworzone aplikacje internetowe będą łatwe w późniejszej obsłudze technicznej i w trakcie skalowania. Poza wzorcami projektowymi przedstawiono wzorce generowania i wydajności działania, których znaczenie dla użytkowania aplikacji jest ogromne. Opisano również nowoczesne wzorce Reacta, między innymi Zaczepy, Komponenty Wyższego Rzędu i Właściwości Generowania. Sporo miejsca poświęcono najlepszym praktykom związanym z organizacją kodu, wydajnością działania czy generowaniem, a także innym zagadnieniom, które pozwalają na podniesienie jakości aplikacji internetowych.

W książce między innymi:

- wzorce architekturalne i struktura aplikacji
- omówienie ponad 20 wzorców projektowych w języku JavaScript i bibliotece React
- kategorie wzorców projektowych i ich zastosowanie
- wzorce związane z wydajnością działania kodu
- wzorce generowania

Addy Osmani od lat pracuje jako programista JavaScriptu, jest autorem kilku popularnych książek na temat tego języka. Często występuje na konferencjach i innych wydarzeniach branżowych. Chętnie angażuje się w pomaganie innym programistom w rozwoju ich umiejętności.

Oto wyczekiwana aktualizacja klasycznej książki dotyczącej wzorców projektowych w JavaScripcie.

Stoyan Stefanov, autor książki *JavaScript. Wzorce*

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-0548-1
9 788328 905481
Cena: 69,00 zł