Technologia i rozwiązania

Mistrzowski JavaScript

Programowanie zorientowane obiektowo

Andrea Chiarelli

Helion

[PACKT] PUBLISHING open source community experience distilled

Tytuł oryginału: Mastering JavaScript Object-Oriented Programming

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-3198-3

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

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 7

Prezentowanie danych użytkownikowi

Jedną z najbardziej widocznych części aplikacji z oczywistych względów jest warstwa prezentacji danych użytkownikowi. Nieważne, czy aplikacja ma graficzny interfejs użytkownika, czy działa w wierszu poleceń — interakcja użytkownika z programem oraz dane, na których ten program operuje, zawsze mają kluczowe znaczenie. Zapewnienie użytkownikowi efektywnego dostępu do danych i zwracanie wyników lub informacji zwrotnych wymaga wykonywania skomplikowanych operacji, które często prowadzą do powstania nieczytelnego i trudnego w obsłudze kodu.

Istnieją jednak wzorce projektowe, które opisują zasady projektowania strukturalnego kodu w taki sposób, aby był jednocześnie elastyczny i łatwy w utrzymaniu. W tym rozdziale skupiam się właśnie na takich wzorcach, zwanych **prezentacyjnymi**, których głównym celem jest oddzielenie prezentacji od modelu danych. Mówiąc konkretnie, w rozdziale zostały opisane następujące najbardziej znane wzorce prezentacyjne:

- Model-Widok-Kontroler;
- Model-Widok-Prezenter;
- Model-Widok-ModelWidoku.

Interfejsy użytkownika

Jednym z największych wyzwań podczas tworzenia każdej aplikacji jest implementacja mechanizmów interakcji programu z użytkownikiem. Bez względu na sposób interakcji użytkownika z aplikacją obsługujący ją kod prawie zawsze jest skomplikowany i trudny w utrzymaniu. Dotyczy to zarówno interfejsów graficznych, jak i konsolowych.

Problemy dotyczące implementacji interfejsu użytkownika

Główną przyczyną trudności jest konieczność opanowania trzech aspektów interakcji między użytkownikiem i aplikacją: **stanu**, **logiki** i **synchronizacji**.

Stan to zbiór informacji reprezentujących aktualny obraz interfejsu użytkownika. Decyduje o tym, co użytkownik widzi w danym czasie i jak to może współpracować z aplikacją.

Logika to zbiór operacji, które można wykonać na elementach interfejsu w celu pokazania lub ukrycia albo sprawdzenia poprawności danych. Logika może być bardzo skomplikowana, ale wszystko zależy od rodzaju operacji wykonywanych na danych przedstawianych użytkownikowi.

Synchronizacja to działania dotyczące wiązania danych pokazywanych użytkownikowi z danymi reprezentowanymi przez obiekty biznesowe używane przez aplikację.

To właśnie kombinacja tych składników jest najbardziej złożonym elementem mechanizmu prezentacji danych użytkownikowi i interakcji z aplikacją.

Interfejsy użytkownika i JavaScript

Kwestie dotyczące interakcji z użytkownikiem zazwyczaj nie mają ścisłego związku z samym językiem programowania i dlatego także w aplikacjach JavaScriptu trzeba się borykać z wieloma problemami.

Jak wiadomo, interfejsy użytkownika przy użyciu języka JavaScript zwykle tworzy się z wykorzystaniem znaczników HTML-a. Podczas gdy HTML służy do opisywania interfejsu graficznego, JavaScript służy do manipulowania elementami tego języka za pomocą logiki, która może zmieniać stan i zapewniać synchronizację. JavaScript uzyskuje dostęp do elementów HTML-a poprzez **obiektowy model dokumentu** (ang. *Document Object Model* — DOM), czyli obiektową reprezentację kodu znacznikowego stanowiącego interfejs użytkownika.

Przyjrzymy się typowemu przykładowi realizacji prostego interfejsu użytkownika za pomocą JavaScriptu i HTML-a. Najpierw spójrz na poniższy kod języka HTML:

```
<label>Imię: <input type="text" id="txtName"></label><br/>
<label>Nazwisko: <input type="text" id="txtSurname"></label><br/>
<div id="divMessage"></div>

<button id="btnSave">Zapisz</button>
<button id="btnReset">Resetuj</button>
```

Jest to definicja prostego formularza, w którym użytkownik może podać imię i nazwisko oraz kliknąć przycisk, aby zapisać te informacje w bazie danych. Dodatkowo zostawiono pusty element na ewentualne wiadomości dla użytkownika. W przeglądarce internetowej formularz ten wyglądałby mniej więcej tak jak na poniższym rysunku:

Poniżej natomiast znajduje się typowy kod JavaScriptu obsługujący interakcję takiego formularza z użytkownikiem:

```
var person;
var txtName;
var txtSurname;
var btnSave;
var divMessage;

function Person(name, surname) {
  this.name = name;
  this.surname = surname;
}

function savePerson(person) {
  // zapisanie danych lokalnie lub wysłanie ich na serwer
  console.log("Zapisano!");
}

window.onload = function() {
  txtName = document.getElementById("txtName");
  txtSurname = document.getElementById("txtSurname");
  btnSave = document.getElementById("btnSave");
  btnReset = document.getElementById("btnReset");
  divMessage = document.getElementById("divMessage");

  person = new Person("Jan", "Kowalski");

  txtName.value = person.name;
  txtSurname.value = person.surname;

  btnSave.onclick = function() {
    if (txtName.value && txtSurname.value) {
      person.name = txtName.value;
      person.surname = txtSurname.value;
      savePerson(person);
      divMessage.innerHTML = "Zapisano!";
    } else {
      divMessage.innerHTML = "Wpisz imię i nazwisko!";
    }
  };
```

```
    btnReset.onclick = function() {
      txtName.value = "";
      txtSurname.value = "";
      divMessage.innerHTML = "";
      person.name = "";
      person.surname = "";
    };
  };
```

Zdefiniowaliśmy kilka zmiennych globalnych, konstruktor `Person()` do tworzenia obiektu biznesowego oraz funkcję `savePerson()`, która będzie zapisywała dane w odpowiednim miejscu. Następnie powiązaliśmy logikę ze zdarzeniem ładowania aktualnego okna przeglądarki. Logika ta umożliwia wyświetlanie w polach tekstowych aktualnych wartości własności obiektu reprezentującego osobę oraz obsługuje działanie przycisków *Zapisz* i *Resetuj*. Kliknięcie przycisku *Zapisz* powoduje wywołanie funkcji `savePerson()`, jeśli wprowadzone przez użytkownika dane są prawidłowe, lub wyświetla komunikat o braku wymaganych informacji w przeciwnym przypadku. Przycisk *Resetuj* służy do kasowania zawartości pól formularza.

Jest to w pełni funkcjonalny kod, choć pozostawia wiele do życzenia pod względem elastyczności i łatwości obsługi. Stan, logika i synchronizacja są ze sobą tak ściśle powiązane, że nawet pozornie prosta zmiana może być kłopotliwa. Zastanówmy się np., co musielibyśmy zrobić, gdybyśmy zdecydowali, że zamiast w elemencie `div` wiadomości dla użytkownika wolimy wyświetlać w elemencie `textarea`. Co by było, gdybyśmy chcieli przenieść całą logikę biznesową i synchronizację do innego rodzaju interfejsu użytkownika, np. wiersza poleceń?

Musielibyśmy wprowadzić liczne zmiany w kodzie, co nie byłoby takie łatwe nawet mimo prostoty tego przykładu. W takich sytuacjach pomocne są właśnie wzorce prezentacyjne.

Wzorce prezentacyjne

Wzorce prezentacyjne to kategoria wzorców projektowych dotyczących sposobów prezentacji danych użytkownikowi. Są szeroko wykorzystywane przy programowaniu interfejsów użytkownika, a ich podstawową zasadą jest rozdzielenie niezwiązanych ze sobą spraw. Stan, logikę i synchronizację rozdziela się między komponenty, które wspólnie tworzą określony rodzaj architektury zapewniającej elastyczność i łatwość utrzymania. W rozdziale spraw promowanym przez wzorce prezentacyjne chodzi głównie o dokładne wyodrębnienie obiektów biznesowych, które opisują świat realny, i obiektów prezentacyjnych, które są elementami graficznego interfejsu użytkownika widocznego na ekranie. Obiekty biznesowe powinny być całkowicie samodzielne i działać bez odnoszenia się do warstwy prezentacji. Ponadto powinny jednocześnie obsługiwać różne rodzaje prezentacji, takie jak np. interfejs graficzny i wiersz poleceń.

Model, widok i kontroler

Tradycyjnie wzorzec prezentacyjny obejmuje trzy składniki.

Pierwszy z nich to **model** — składają się nań obiekty biznesowe zawierające informacje, które mają zostać przedstawione użytkownikowi. Model można pobrać z usługi albo może być przechowywany w bazie danych. Nie posiada on żadnych informacji na temat sposobu prezentowania danych użytkownikowi i nie zawiera żadnej logiki. To tylko dane.

Widok to komponent odpowiedzialny za prezentację danych użytkownikowi i przechwytywanie zdarzeń interakcji z użytkownikiem. Stanowi on interfejs użytkownika i reprezentuje aktualny stan modelu. W aplikacjach sieciowych widok zazwyczaj obejmuje kod HTML-a, który opisuje użytkownikowi aktualny model i umożliwia wystąpienie interakcji.

Trzeci komponent może mieć różne nazwy w zależności od wzorca. W najczęściej używanych wzorcach bywa nazywany **kontrolerem**, **prezenterem** albo **widokiem modelu**. Zazwyczaj zawiera logikę architektury prezentacji i może koordynować przepływ informacji między widokiem a modelem. Innymi słowy: sprawuje nadzór nad synchronizacją bieżącego stanu i modelu danych.

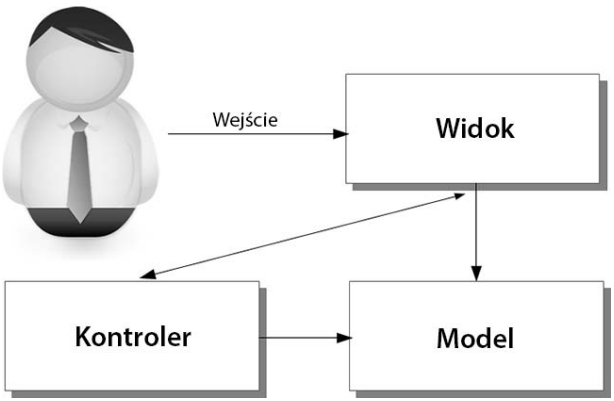
W kilku następnych podrozdziałach objaśniam trzy najczęściej używane wzorce prezentacyjne: **Model-Widok-Kontroler** (ang. *Model-View-Controller* — **MVC**), **Model-Widok-Prezenter** (ang. *Model-View-Presenter* — **MVP**) oraz **Model-Widok-ModelWidoku** (ang. *Model-View-ViewModel* — **MVVM**). Te wzorce i ich różne warianty czasami określa się mianem **MV***, co oznacza, że model i widok to elementy stałe, choć ich rola może być różna. Sposób realizacji zasady rozdziału spraw przez każdy z tych trzech wzorców prześledzimy na podstawie implementacji według ich zasad przykładu opisanego powyżej.

> Opisanych w tym rozdziale wzorców prezentacyjnych nie ma w książce Bandy Czworga. Większość programistów uważa je za wzorce złożone, czyli takie, które można zbudować z innych wzorców. Na przykład widok i kontroler mają implementację strategii, sam widok może być implementacją złożoną, a widok i model mogą być synchronizowane przez wzorzec Obserwator. Jednak ze względu na ich rozpowszechnienie wzorce te opisuję jako odrębne jednostki.

Wzorzec Model-Widok-Kontroler

Wzorzec Model-Widok Kontroler (MVC) jest jednym z pierwszych prezentacyjnych wzorców projektowych, który opracowano w latach 70. jako zalecaną metodę implementacji graficznych interfejsów użytkownika. Z biegiem lat wraz z postępem technologii powstały liczne wersje tego wzorca, ale jego podstawowa struktura nie zmieniła się do dziś. Jak wskazuje nazwa, oprócz modelu i widoku, we wzorcu tym ważną rolę odgrywa kontroler.

Każdy element pełni istotną funkcję, a wszystkie te komponenty razem obsługują interakcję z użytkownikiem, jak pokazano na poniższym rysunku:



Zadaniem widoku jest obsługa interakcji z użytkownikiem. Wyświetla on dane dostarczone przez model i odbiera dane od użytkownika. Kontroler współpracuje z modelem w wyniku reakcji na dane wprowadzane przez użytkownika. Kiedy użytkownik wprowadza dane do widoku, kontroler przejmuje je i dokonuje zmian w modelu. Rozdział zadań między te trzy komponenty ułatwia otrzymanie bardziej elastycznego i łatwiejszego w utrzymaniu kodu.

Aby nie opierać się tylko na abstrakcyjnych opisach, przepiszemy kod z poprzedniej sekcji zgodnie z założeniami architektury MVC. Zaczniemy od zdefiniowania modelu:

```
var Model = (function () {
  function Model(name, surname) {
    this.name = name;
    this.surname = surname;
  }
  return Model;
}());
```

Konstruktor ten definiuje osobę jako obiekt `Model` posiadający własności `name` i `surname`. Oczywiście model może też być klasą, jak w poniższym przykładzie:

```
class Model {
  constructor(name, surname) {
    this.name = name;
    this.surname = surname;
  }
}
```

Widok będzie obsługiwał interakcję z użytkownikiem i wizualnie przedstawiał dane modelu. Oto jego implementacja:

```
var View = (function () {
  function View(model, controller) {
    var self = this;
```

```
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var btnSave = document.getElementById("btnSave");
    var btnReset = document.getElementById("btnReset");

    self.controller = controller;
    txtName.value = model.name;
    txtSurname.value = model.surname;

    btnSave.onclick = function () {
      self.save();
    };

    btnReset.onclick = function () {
      self.clear();
    };
  }

  View.prototype.clear = function () {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var divMessage = document.getElementById("divMessage");

    txtName.value = "";
    txtSurname.value = "";
    divMessage.innerHTML = "";
  };

  View.prototype.save = function () {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var data = {
      name: txtName.value,
      surname: txtSurname.value
    };

    this.controller.save(data);
  };

  Object.defineProperty(View.prototype, "message", {
    set: function (message) {
      var divMessage = document.getElementById("divMessage");
      divMessage.innerHTML = message;
    },
    enumerable: true,
    configurable: true
  });

  return View;
}());
```

Jak widać, konstruktor pobiera argumenty `model` i `controller`. Odnosi się on do elementów HTML-a strony przez DOM i wstawia do nich wartości z modelu. Do konstruktora zostały dołączone metody `clear()` i `save()` powiązane ze zdarzeniem kliknięcia przycisków. Metoda `clear()` resetuje pola tekstowe formularza internetowego, a metoda `save()` pobiera aktualne wartości z tych pól i przekazuje je do metody `save()` kontrolera. Na koniec następuje dołączenie do widoku własności `message`. Wartość tej własności jest przekazywana do elementu `div` przeznaczonego do wyświetlania wiadomości.

Oczywiście jak zawsze widok można zdefiniować też przy użyciu składni klasowej:

```
class View {
  constructor(model, controller) {
    var self = this;
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var btnSave = document.getElementById("btnSave");
    var btnReset = document.getElementById("btnReset");

    self.controller = controller;

    txtName.value = model.name;
    txtSurname.value = model.surname;

    btnSave.onclick = function() {
      self.save();
    };

    btnReset.onclick = function() {
      self.clear();
    };
  }

  clear() {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var divMessage = document.getElementById("divMessage");

    txtName.value = "";
    txtSurname.value = "";
    divMessage.innerHTML = "";
  }

  set message(message) {
    var divMessage = document.getElementById("divMessage");

    divMessage.innerHTML = message;
  }
```

```
  save() {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");

    var data = {
      name: txtName.value,
      surname: txtSurname.value
    };

    this.controller.save(data);
  }
}
```

Konstruktor `Controller()` zawiera metodę `initialize()` ustawiającą referencję do modelu i widoku. Ponadto zawiera metodę `save()`, która sprawdza poprawność danych i wprowadza potrzebne zmiany w modelu. Dodatkowym zadaniem tej metody jest wyprowadzanie do widoku wiadomości dla użytkownika:

```
var Controller = (function () {
  function Controller() {
  }

  Controller.prototype.initialize = function (model, view) {
    this.model = model;
    this.view = view;
  };

  Controller.prototype.save = function (data) {
    if (data.name && data.surname) {
      this.model.name = data.name;
      this.model.surname = data.surname;

      this.view.message = "Zapisano!";
    } else {
      this.view.message = "Podaj imię i nazwisko!";
    }
  };

  return Controller;
}());
```

Poniżej znajduje się ekwiwalentna definicja kontrolera w postaci klasy:

```
class Controller {
  initialize(model, view) {
    this.model = model;
    this.view = view;
  }
```

```
    save(data) {
      if (data.name && data.surname) {
        this.model.name = data.name;
        this.model.surname = data.surname;

        this.view.message = "Zapisano!";
      } else {
        this.view.message = "Podaj imię i nazwisko!";
      }
    }
}
```

Po zdefiniowaniu tych trzech składników wzorca Model-Widok-Kontroler należy je jeszcze połączyć, aby ze sobą współpracowały. W związku z tym wiążemy poniższą funkcję ze zdarzeniem ładowania okna przeglądarki:

```
window.onload = function() {
  var model = new Model("Jan", "Kowalski");
  var controller = new Controller();
  var view = new View(model, controller);

  controller.initialize(model, view);
};
```

Funkcja ta tworzy nowy model i kontroler. Następnie tworzy widok przy użyciu modelu i kontrolera, przekazuje go do konstruktora i na koniec wywołuje metodę `initialize()` kontrolera.

Ten kod działa dokładnie tak samo jak poprzednia wersja, ale jest tak zorganizowany, że o wiele łatwiej można go zmieniać i przystosowywać do innych sytuacji.

Gdybyśmy np. zechcieli zmienić element `div` wyświetlający wiadomość dla użytkownika na `textarea`, musielibyśmy wprowadzić zmiany tylko w widoku. Inne komponenty pozostałyby niezmienione i modyfikacja ta nie miałaby dla nich znaczenia. To samo byłoby, gdybyśmy chcieli zmienić cały interfejs, np. zamiast strony internetowej postanowilibyśmy użyć wiersza poleceń. W takim przypadku musielibyśmy napisać nowy widok, ale nie musielibyśmy niczego zmieniać w modelu ani w kontrolerze.

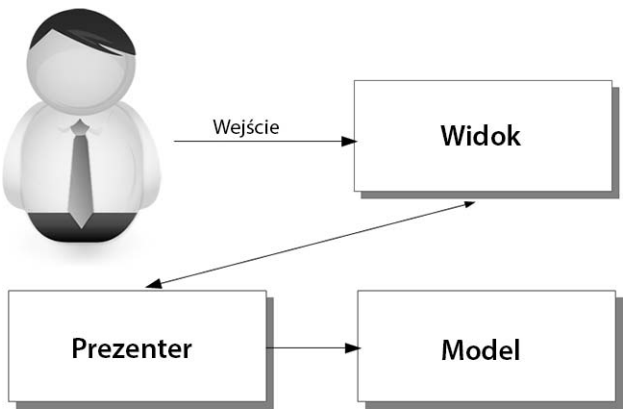
Gdybyśmy chcieli zmienić kryteria weryfikacji danych wprowadzanych przez użytkownika, musielibyśmy tylko zmienić kod kontrolera.

Struktura kodu zbudowana według założeń wzorca MVC umożliwia łatwe znajdowanie komponentu do modyfikacji na podstawie rodzaju potrzebnej zmiany.

Wzorzec Model-Widok-Prezenter

Wzorzec Model-Widok-Kontroler zapewnia lepszą architekturę do prezentacji danych użytkownikowi. Każdemu komponentowi przydziela konkretne zadanie, dzięki czemu w razie konieczności wprowadzenia zmian można ograniczyć się do modyfikacji tylko jednego komponentu. Mimo to te trzy komponenty pozostają wzajemnie powiązane: widok zna swój kontroler i model, a działanie kontrolera zależy od widoku i modelu. Na przykład zmiana modelu może wymagać zmian zarówno w widoku, jak i w kontrolerze.

Wzorzec **Model-Widok-Prezenter** (MVP) opisuje architekturę warstwową z mniejszą liczbą zależności. W tym wzorcu widok przejmuje zdarzenia generowane przez użytkownika i prosi prezenter o dokonanie zmian w modelu. Oznacza to, że widok nie współpracuje bezpośrednio z modelem, ale działa na nim za pośrednictwem prezentera. W ten sposób zostają zlikwidowane wszelkie zależności między widokiem i modelem. Na poniższym schemacie przedstawione są relacje między poszczególnymi składnikami omawianego wzorca:



Zobaczmy, jak będzie wyglądała implementacja na bazie wzorca MVP naszej przykładowej aplikacji.

Implementacja modelu pozostaje taka sama jak w wersji MVC.

Natomiast widok nieco się zmieni, co widać poniżej:

```
var View = (function () {
  function View(presenter) {
    var self = this;
    var btnSave = document.getElementById("btnSave");
    var btnReset = document.getElementById("btnReset");

    self.presenter = presenter;

    btnSave.onclick = function () {
      self.save();
```

```
    };

    btnReset.onclick = function () {
      self.clear();
    };
  }

  View.prototype.clear = function () {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var divMessage = document.getElementById("divMessage");

    txtName.value = "";
    txtSurname.value = "";
    divMessage.innerHTML = "";
  };

  Object.defineProperty(View.prototype, "message", {
    set: function (message) {
      var divMessage = document.getElementById("divMessage");
      divMessage.innerHTML = message;
    }
  });

  Object.defineProperty(View.prototype, "name", {
    set: function (value) {
      var txtName = document.getElementById("txtName");
      txtName.value = value;
    }
  });

  Object.defineProperty(View.prototype, "surname", {
    set: function (value) {
      var txtSurname = document.getElementById("txtSurname");
      txtSurname.value = value;
    },
    enumerable: true,
    configurable: true
  });
  View.prototype.save = function () {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var data = {
      name: txtName.value,
      surname: txtSurname.value
    };

    this.presenter.save(data);
  };
  return View;
}());
```

Główna różnica w stosunku do implementacji widoku według wzorca MVC dotyczy definicji własności name i surname. Wprowadziliśmy je po to, by zlikwidować zależność między widokiem i modelem. Ponieważ widok nie zna modelu, udostępnia te własności, które prezenter wiąże z modelem.

Widok możemy też zdefiniować jako klasę, tak jak pokazano poniżej:

```
class View {
  constructor(presenter) {
    var self = this;
    var btnSave = document.getElementById("btnSave");
    var btnReset = document.getElementById("btnReset");

    self.presenter = presenter;

    btnSave.onclick = function() {
      self.save();
    };

    btnReset.onclick = function() {
      self.clear();
    };
  }

  clear() {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var divMessage = document.getElementById("divMessage");

    txtName.value = "";
    txtSurname.value = "";
    divMessage.innerHTML = "";
  }

  set message(message) {
    var divMessage = document.getElementById("divMessage");
    divMessage.innerHTML = message;
  }

  set name(value) {
    var txtName = document.getElementById("txtName");
    txtName.value = value;
  }
  set surname(value) {
    var txtSurname = document.getElementById("txtSurname");
    txtSurname.value = value;
  }
```

```
    save() {
      var txtName = document.getElementById("txtName");
      var txtSurname = document.getElementById("txtSurname");
      var data = {
        name: txtName.value,
        surname: txtSurname.value
      };
      this.presenter.save(data);
    }
  }
```

Implementacja prezentera jest dość podobna do implementacji kontrolera we wzorcu MVC. Jedyna różnica w stosunku do wersji MVC dotyczy przypisania wartości modelu do własności widoku, co zaznaczono pogrubionym drukiem w poniższym kodzie:

```
var Presenter = (function () {
  function Presenter() {
  }

  Presenter.prototype.initialize = function (model, view) {
    this.model = model;
    this.view = view;
    this.view.name = this.model.name; this.view.surname =
    this.model.surname;
  };

  Presenter.prototype.save = function (data) {
    if (data.name && data.surname) {
      this.model.name = data.name;
      this.model.surname = data.surname;
      this.view.message = "Zapisano!";
    }
    else {
      this.view.message = "Podaj imię i nazwisko!";
    }
  };

  return Presenter;
}());
```

Klasowa wersja prezentera może wyglądać tak:

```
class Presenter {
  initialize(model,view) {
    this.model = model;
    this.view = view;

    this.view.name = this.model.name;
    this.view.surname = this.model.surname;
  }
```

```
  save(data) {
    if (data.name && data.surname) {
      this.model.name = data.name;
      this.model.surname = data.surname;

      this.view.message = "Zapisano!";
    } else {
      this.view.message = "Podaj imię i nazwisko!";
    }
  }
}
```

Teraz możemy utworzyć egzemplarze komponentów wzorca, jak w poniższym przykładzie:

```
window.onload = function() {
  var model = new Model("Jan", "Kowalski");
  var presenter = new Presenter();
  var view = new View(presenter);

  presenter.initialize(model, view);
};
```

Wzorzec MVP posuwa zasadę rozdziału spraw między trzy komponenty o krok dalej niż MVC. Tylko prezenter „wie" o istnieniu zarówno modelu, jak i widoku. Dzięki temu można wprowadzać zmiany w modelu bez ruszania widoku. Zadaniem prezentera jest synchronizacja widoku i modelu.

> Jeśli chodzi o inne wzorce prezentacyjne, nawet MVP ma kilka różnych wariantów implementacyjnych. W prawdziwych zastosowaniach niektórzy nadal chcą umieszczać podstawową logikę wewnątrz widoku, a bardziej skomplikowaną w prezenterze, natomiast inni wolą całą logikę wstawić do prezentera. W efekcie powstają przynajmniej dwa wzorce pochodne: **MVP z widokiem pasywnym**, w którym logika dodawana do widoku jest ograniczona do minimum, oraz **kontroler nadzorczy** (ang. *Supervising Controller*), w którym część logiki dotycząca prostych deklaratywnych operacji jest pozostawiona w widoku.

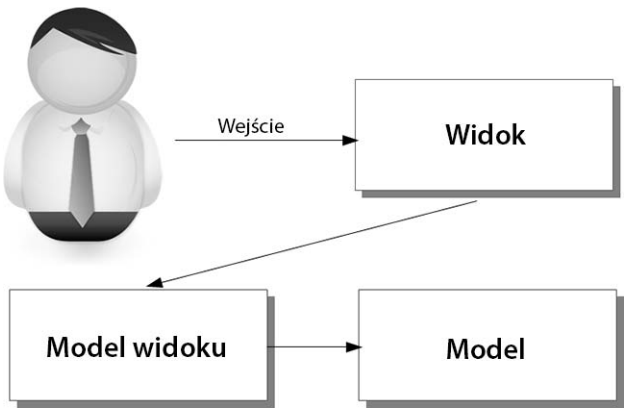
Wzorzec Model-Widok-ModelWidoku

Wzorzec **Model-Widok-ModelWidoku** (ang. *Model-View-ViewModel* — **MVVM**) jest próbą dalszej redukcji zależności między komponentami wzorca prezentacyjnego. Nowością w nim jest komponent *model widoku*, który zastępuje prezenter. Oczywiście zmiana nie dotyczy tylko wprowadzenia nowej nazwy. Zasady tego wzorca spróbujemy zrozumieć dzięki porównaniu go ze wzorcem MVP.

Podobnie jak we wzorcu MVP, widok nie posiada żadnych informacji o istnieniu modelu. Jednak we wzorcu MVP widok „wiedział" o istnieniu jakiegoś komponentu pośredniego. We wzorcu MVVM modelem dla widoku jest „model widoku". Zamiast prosić prezenter o związanie danych

i modyfikować model, widok posługuje się własnym modelem reprezentowanym przez „model widoku". Pełni on więc funkcję opakowania prawdziwego modelu oraz przeprowadza pewne testy spójności i wykonuje inne działania związane z zarządzaniem danymi.

Na poniższym schemacie ukazane są relacje między poszczególnymi komponentami wzorca MVVM:



Zobaczmy, jak zaimplementować widok we wzorcu MVVM:

```
var View = (function () {
  function View(modelView) {
    var self = this;
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var btnSave = document.getElementById("btnSave");
    var btnReset = document.getElementById("btnReset");

    self.modelView = modelView;
    txtName.value = modelView.name;
    txtSurname.value = modelView.surname;

    btnSave.onclick = function () {
      self.save();
    };

    btnReset.onclick = function () {
      self.clear();
    };
  }

  View.prototype.clear = function () {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var divMessage = document.getElementById("divMessage");
```

```
    txtName.value = "";
    txtSurname.value = "";
    divMessage.innerHTML = "";
  };

  View.prototype.setMessage = function (message) {
    var divMessage = document.getElementById("divMessage");
    divMessage.innerHTML = message;
  };

  View.prototype.save = function () {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var data = {
      name: txtName.value,
      surname: txtSurname.value
    };

    this.modelView.save(data, this.setMessage);
  };
  return View;
}());
```

Jak widać, widok wykorzystuje przekazany w konstruktorze model widoku tak, jakby to był zwykły model. Wiąże własności modelu widoku z elementami interfejsu użytkownika i przetwarza go za pomocą metod modelu widoku.

Poniżej znajduje się definicja widoku jako klasy:

```
class View {
  constructor(modelView) {
    var self = this;
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var btnSave = document.getElementById("btnSave");
    var btnReset = document.getElementById("btnReset");

    self.modelView = modelView;

    txtName.value = modelView.name;
    txtSurname.value = modelView.surname;

    btnSave.onclick = function() {
      self.save();
    };
    btnReset.onclick = function() {
      self.clear();
    };
  }
```

```
  clear() {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");
    var divMessage = document.getElementById("divMessage");

    txtName.value = "";
    txtSurname.value = "";
    divMessage.innerHTML = "";
  }

  setMessage(message) {
    var divMessage = document.getElementById("divMessage");
    divMessage.innerHTML = message;
  }

  save() {
    var txtName = document.getElementById("txtName");
    var txtSurname = document.getElementById("txtSurname");

    var data = {
      name: txtName.value,
      surname: txtSurname.value
    };

    this.modelView.save(data, this.setMessage);
  }
}
```

Jak napisałem wcześniej, model widoku opakowuje model i dodaje metody umożliwiające kontrolowane modyfikowanie danych. Poniżej znajduje się implementacja modelu widoku dla naszego przykładu:

```
var ViewModel = (function () {
  function ViewModel(model) {
    this.model = model;
  }

  Object.defineProperty(ViewModel.prototype, "name", {
    get: function () {
      return this.model.name;
    }
  });

  Object.defineProperty(ViewModel.prototype, "surname", {
    get: function () {
      return this.model.surname;
    }
  });
```

```
  ViewModel.prototype.save = function (data, callback) {
    if (data.name && data.surname) {
      this.model.name = data.name;
      this.model.surname = data.surname;

      if (callback) {
        callback("Zapisano!");
      }
    }
    else {
      if (callback) {
        callback("Podaj imię i nazwisko!");
      }
    }
  };

  return ViewModel;
}());
```

Definicja modelu widoku udostępnia własności `name` i `surname` modelu przez swoje własności tylko do odczytu. Ponadto zawiera metodę `save()`, która umożliwia aktualizację modelu po sprawdzeniu poprawności danych. W tej metodzie ciekawym elementem jest obecność argumentu `callback`. W tym argumencie wywołujący może przekazać funkcję, która ma zostać wywołana po aktualizacji modelu z informacjami o wyniku operacji. W naszym przypadku model widoku przekazuje jako parametr funkcji zwrotnej wiadomość do wyświetlenia. Użycie funkcji zwrotnej sprawia, że model widoku jest niezależny od widoku. Oczywiście istnieją też inne rozwiązania, np. metoda `save()` może zwracać wiadomość albo, jeszcze lepiej, kod. Cel jest taki, aby model widoku nie „wiedział" o używającym go widoku.

Model widoku w postaci klasy może wyglądać następująco:

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

  get name() {
    return this.model.name;
  }
  get surname() {
    return this.model.surname;
  }

  save(data, callback) {
    if (data.name && data.surname) {
    this.model.name = data.name;
    this.model.surname = data.surname;
```

```
      if (callback) {
        callback("Zapisano!");
      }

      } else {
        if (callback) {
          callback("Podaj imię i nazwisko!");
        }
      }
    }
  }
```

Teraz możemy połączyć wszystkie komponenty w jedną działającą całość:

```
window.onload = function() {
  var model = new Model("Jan", "Kowalski");
  var viewModel = new ViewModel(model);
  var view = new View(viewModel);
};
```

W tym kodzie utworzyliśmy po kolei każdy komponent, przekazując jego konstruktorowi jedyny komponent, od którego jest zależny.

Porównanie wzorców MV*

Trzy opisane w tym rozdziale wzorce prezentacyjne mają wiele podobieństw. Wszystkie opierają się na trzech komponentach, między którymi występują podobne interakcje. Jednak każdy wzorzec ma pewne cechy szczególne, które sprawiają, że jest bardziej odpowiedni do określonych zastosowań. Podsumujemy najważniejsze cechy poszczególnych wzorców i zwrócimy szczególną uwagę na te, które odróżniają każdy z nich od pozostałych.

Wzorzec MVC opisuje współpracę między trzema komponentami: modelem, widokiem i kontrolerem. Każdy z nich pełni ściśle określoną funkcję i w jakiś sposób współpracuje z innymi. Widok używa modelu do początkowego wiązania, a kontroler zarządza żądaniami zmian modelu i przekazuje informacje zwrotne do widoku. Wzorzec ten jest pierwszą próbą dokonania rozdziału spraw, ale pewne zmiany w jednym komponencie mogą wymagać modyfikacji także w innych komponentach. MVC jest najstarszy ze wszystkich wzorców prezentacyjnych. Został opracowany w latach 70., czyli w czasach, gdy pierwsze graficzne interfejsy użytkownika były jeszcze bardzo proste.

Wzorzec MVP zrywa zależność między widokiem i modelem, zlecając prezenterowi rolę pośrednika. Widok pozostaje jedynym komponentem odpowiedzialnym za obsługę interakcji z użytkownikiem, podczas gdy prezenter jest jedynym komponentem uprawnionym do pracy z modelem i reagowania na widok. Taka architektura, dzięki strukturze warstwowej, zapewnia większą niezależność komponentów.

Wzorzec MVVM posuwa się jeszcze dalej, przypisując pośredniemu komponentowi, zwanemu modelem widoku, rolę specjalnego modelu dla widoku. Model widoku współpracuje z użytkownikiem i bezpośrednio przekazuje dane do tego, co dla niego jest modelem. W rzeczywistości model ten jest opakowaniem prawdziwego modelu, a jego nazwa „model widoku" wskazuje, że reprezentuje model dla widoku. Wzorzec ten podobnie jak MVP opisuje strukturę warstwową, ale każdy komponent zależy tylko od komponentu znajdującego się bezpośrednio pod nim. Zatem widok zależy od modelu widoku, ale nie odwrotnie. Zastosowanie wzorca MVVM wymaga, aby widok mógł wiązać dane i implementował trochę logiki. Dlatego wzorzec ten najlepiej sprawdza się na platformach obsługujących dwukierunkowe wiązanie i mających elementy graficzne o zaawansowanych możliwościach.

Podsumowując: wzorzec MVC przydziela określone role każdemu komponentowi, ale nie rozwiązuje kwestii wzajemnego sprzężenia komponentów. Widok i kontroler mogą współpracować ze sobą nawzajem i z modelem. Architektura ta może być efektywna pod względem wydajności, ale może również powodować powstawanie problemów dotyczących bezpieczeństwa, ponieważ widok ma dostęp do całego modelu.

Wzorzec MVP lepiej zabezpiecza model, gdyż dostęp do niego ma tylko prezenter, ale sam prezenter może powodować problemy wydajnościowe w skomplikowanych aplikacjach.

Te same wątpliwości natury wydajnościowej dotyczą wzorca MVVM, w którym część logiki jest przeniesiona z warstwy pośredniej do warstwy widoku.

Podsumowanie

W tym rozdziale omówiłem sposoby prezentacji danych użytkownikowi z punktu widzenia modelu obiektowego. Pokazałem, z jakimi trudnościami dotyczącymi rozszerzalności i obsługi serwisowej kodu musi się zmierzyć programista stosujący podejście niestrukturalne. Wzorce prezentacyjne pomagają w lepszym organizowaniu kodu oraz otrzymywaniu przejrzystszej i elastyczniejszej architektury. Opisane w tym rozdziale najbardziej znane wzorce prezentacyjne definiują trzy współpracujące ze sobą komponenty.

Na początku opisałem wzorzec Model-Widok-Kontroler, którego komponenty mają wyraźnie określone role i wspólnie obsługują interakcje między użytkownikiem a modelem danych. Następnie omówiłem wzorzec Model-Widok-Prezenter, który jest wariantem wzorca MVC opartym na systemie warstwowym. Założenia tego wzorca wykluczają bezpośredni dostęp widoku do modelu. Dostęp ten zawsze odbywa się za pośrednictwem prezentera. Ostatnim z opisanych wzorców jest Model-Widok-ModelWidoku, który charakteryzuje się warstwowym podejściem, jak w MVP, z dodatkiem nowego sposobu interpretacji roli komponenta pośredniego między widokiem i modelem. Jest to specjalny model dla widoku o nazwie „model widoku". Na zakończenie rozdziału przedstawiłem porównanie tych trzech wzorców.

Temat następnego rozdziału łączy się z wzorcami prezentacyjnymi. Omawiam w nim techniki wiązania danych i synchronizacji oraz metody optymalnego prezentowania danych użytkownikowi.

Skorowidz

H

I

K

L

Ł

M

N

O

P

R

S

T

U

W

Z

Mistrzowski JavaScript
Programowanie zorientowane obiektowo

JavaScript jest popularnym, rozwiniętym i dojrzałym językiem programowania, a jego zastosowanie znacząco wykracza dziś poza WWW, HTML i CSS: skrypty można uruchamiać na serwerze, komputerach PC, w urządzeniach przenośnych i układach wbudowanych. Dzięki tym możliwościom JavaScript stał się potężnym i wszechstronnym narzędziem. Co więcej, ten język świetnie nadaje się do programowania zorientowanego obiektowo i pozwala na pisanie solidnego kodu, a w efekcie na tworzenie nawet bardzo złożonych, skalowalnych i łatwych w utrzymaniu aplikacji.

Niniejsza książka jest przeznaczona dla osób, które mniej więcej znają JavaScript, ale chcą się nauczyć programować obiektowo w tym języku. Można tu znaleźć informacje o definiowaniu obiektów za pomocą klas ES6, metodach hermetyzacji oraz różnych sposobach dziedziczenia. Są tu niektóre zaawansowane wzorce projektowe i opis wykorzystania mechanizmu obietnic do pracy z procesami asynchronicznymi. Nie zabrakło również wyjaśnień zasad SOLID, dzięki którym tworzony kod staje się efektywny i niezawodny.

JavaScript — pisz skrypty do zadań specjalnych!

W książce znajdziesz:

- podstawy programowania obiektowego w JavaScripcie
- techniki imitacji klasycznych interfejsów obiektowych
- modele prezentacyjne: MVC i MVVM
- programowanie asynchroniczne
- porządkowanie kodu źródłowego — moduły ECMAScript 6

Andrea Chiarelli jest ekspertem programowania w różnych technologiach (C#, JavaScript, ASP.NET czy AngularJS, REST i PhoneGap/Cordova), a także autorem i współautorem licznych książek o programowaniu. Poza tym regularnie pisuje do takich magazynów jak „Computer Programming" czy „ASP Today". Pracuje na stanowisku starszego inżyniera oprogramowania we włoskim oddziale Apparound Inc., firmy, która urodziła się w samym sercu Doliny Krzemowej.

[PACKT] open source PUBLISHING community experience distilled

Helion

księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

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

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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-3198-3
9 788328 331983
cena: 49,00 zł