|  |  |  |
| --- | --- | --- |
| GODLO_C | **Politechnika Gdańska**  **WYDZIAŁ ELEKTRONIKI TELEKOMUNIKACJI I INFORMATYKI** | logo_eti |

**Katedra: Katedra Systemów Geoinformatycznych**

**Imię i nazwisko dyplomanta: Mateusz Jamiołkowski**

**Nr albumu: 113668**

**Forma i poziom studiów: Stacjonarne magisterkie**

**Kierunek studiów: Informatyka**

**Praca dyplomowa**

**Temat pracy: Kod JavaScript wysokiej jakości w przenośnej aplikacji mobilnej**

**Opiekun pracy: dr hab. inż Marek Moszyński, prof. nadzw. PG**

**Zakres pracy: Zbadanie zagadnienia wytwarzania przenośnych apliklacji mobilnych z naciskiem na wysoką jakośc kodu.**

**Gdańsk, 2012 rok**

Spis treści

[1 Wprowadzenie 6](#_Toc340935218)

[1.1 Motywacja 6](#_Toc340935219)

[1.2 Cel pracy 6](#_Toc340935220)

[2 Zastosowania języka JavaScript 7](#_Toc340935221)

[2.1 Język JavaScript 7](#_Toc340935222)

[2.1.1 Podstawowe założenia 7](#_Toc340935223)

[2.1.2 Popularność 18](#_Toc340935224)

[2.2 Problematyka mnogości platform mobilnych 22](#_Toc340935225)

[2.2.1 Najważniejsze platformy mobilne 23](#_Toc340935226)

[2.2.2 Problemy wynikające z mnogości platform 24](#_Toc340935227)

[2.3 Język JavaScript w aplikacjach mobilnych 25](#_Toc340935228)

[2.3.1 Aplikacje webowe 25](#_Toc340935229)

[2.3.2 Aplikacje hybrydowe 25](#_Toc340935230)

[2.4 Język JavaScript w aplikacjach serwerowych 27](#_Toc340935231)

[2.4.1 Historia powstania node.js 27](#_Toc340935232)

[2.4.2 Główna idea node.js 28](#_Toc340935233)

[2.4.3 Podstawowe założenia node.js 30](#_Toc340935234)

[2.4.4 Niskopoziomowa architektura node.js 30](#_Toc340935235)

[2.5 Język JavaScript w aplikacjach desktopowych 31](#_Toc340935236)

[3 Jakość kodu JavaScript 32](#_Toc340935237)

[3.1 Definicja jakości kodu 32](#_Toc340935238)

[3.2 Wartość kodu wysokiej jakości 32](#_Toc340935239)

[3.2.1 Ewolucja oprogramowania i prawa Lehmana 32](#_Toc340935240)

[3.2.2 Dług technologiczny 33](#_Toc340935241)

[3.2.3 Wpływ jakości strukturalnej na proces utrzymania oprogramowania 33](#_Toc340935242)

[3.3 Praktyczne sposoby zapewniania jakości kodu JavaScript 34](#_Toc340935243)

[3.3.1 Formatowanie 34](#_Toc340935244)

[3.3.2 Dobre praktyki czystego kodu 36](#_Toc340935245)

[3.3.3 Testy jednostkowe 46](#_Toc340935246)

[3.4 Metryki jakości kodu 50](#_Toc340935247)

[3.4.1 Złożoność cyklomatyczna 51](#_Toc340935248)

[3.4.2 Kohezja 51](#_Toc340935249)

[3.4.3 Zależność 52](#_Toc340935250)

[3.4.4 Poziom pokrycia 53](#_Toc340935251)

[3.5 Dziedzina problemu 54](#_Toc340935252)

[3.5.1 Metodyka GTD 54](#_Toc340935253)

[3.5.2 Analiza metodyki GTD 56](#_Toc340935254)

[4 Projekt aplikacji do zarządzania zadaniami 58](#_Toc340935255)

[4.1 Cel projektu 58](#_Toc340935256)

[4.2 Wymagania 58](#_Toc340935257)

[4.3 Projekt architektury wysokiego poziomu 59](#_Toc340935258)

[4.3.1 Warstwa prezentacji 59](#_Toc340935259)

[4.3.2 Warstwa aplikacji 61](#_Toc340935260)

[4.3.3 Komunikacja między serwerem i klientem 61](#_Toc340935261)

[4.3.4 Warstwa danych 64](#_Toc340935262)

[4.3.5 Schemat danych 67](#_Toc340935263)

[4.4 Narzędzia 68](#_Toc340935264)

[4.5 Testowanie jednostkowe aplikacji 69](#_Toc340935265)

[4.6 Plan badania jakości kodu stworzone aplikacji 73](#_Toc340935266)

[4.6.1 Mierzenieilości naruszeń SOLID 73](#_Toc340935267)

[4.6.2 Mierzenie wartości metryk jakości kodu 74](#_Toc340935268)

[4.6.3 Subiektywna ocena jakości kodu 74](#_Toc340935269)

[5 Implementacja aplikacji do zarządzania zadaniami 75](#_Toc340935270)

[5.1 Implementacja 75](#_Toc340935271)

[5.1.1 Aplikacja serwerowa 75](#_Toc340935272)

[5.1.2 Aplikacja kliencka 80](#_Toc340935273)

[5.2 Rezultat prac 82](#_Toc340935274)

[5.3 Badanie jakości kodu 83](#_Toc340935275)

[5.3.1 Wyniki pomiarów ilości naruszeń zasad SOLID 84](#_Toc340935276)

[5.3.2 Wyniki pomiarów wartości metryk jakości kodu 85](#_Toc340935277)

[6 Podsumowanie 89](#_Toc340935278)

[7 Bibliografia 91](#_Toc340935279)

Kod JavaScript wysokiej jakości w przenośnej aplikacji mobilnej

# Wprowadzenie

## Motywacja

JavaScript, jak zostanie pokazane w rozdziale 2. zdobywa coraz większą popularność. Obecnie jest on już nie tylko językiem, który jedynie poprawia atrakcyjność stron internetowych. Przy jego użyciu buduje się całe, nawet skomplikowane aplikacje.

Jest to też język dosyć nietypowy, zwłaszcza w porównaniu do bardziej powszechnie używanych C, C# czy Java. Specyfika języka zostanie dokładnie wyjaśniona w kolejnym rozdziale. JavaScript nie wymaga wiele nauki, aby rozpocząć z nim pracę, dzięki czemu można tworzyć za jego pomocą aplikacje bez poznania np. obiektowości czy elementów programowania funkcyjnego. Z tego powodu kod JavaScript często pozostawia wiele do życzenia.

Innym istotnym aspektem związanym z językiem jest przenośność. Obecnie można z niego korzystać nie tylko przy tworzeniu aplikacji internetowych, ale też na większości platform mobilnych czy nawet na serwerach.

Rosnąca popularność i rozmiar aplikacji wykonywanych w JavaScript. Sprawiają, że problematyka jakości kodu opisana w rozdziale trzecim zyskuje na znaczeniu. Właśnie ona stanowiła główną motywację dla stworzenia niniejszej pracy dyplomowej.

## Cel pracy

Celem pracy magisterskiej jest zbadanie zagadnienia wytwarzania przenośnych aplikacji w języku JavaScript ze szczególnym naciskiem na wysoką jakość kodu. W oparciu o kod rzeczywistej aplikacji wykazane zostanie, że dobre praktyki zapewniania jakości kodu takie jak testowanie jednostkowe czy zasady SOLID mają praktyczne zastosowanie w programach napisanych w języku JavaScript.

# Zastosowania języka JavaScript

Poniższy rozdział przedstawia zastosowania języka JavaScript. Najpierw omówiony zostanie sam język, wraz historią powstania oraz rozwoju i stanem, w którym obecnie się znajduje. Następnie przedstawiona zostanie sytuacja na rynku platform mobilnych, których mnogość i zróżnicowanie stanowią istotną przyczynę dla miejsca, w którym obecnie znajduje się język JavaScript. Pod koniec rozdziału nastąpi wprowadzenie do wykorzystania omawianego języka po stronie klienckiej aplikacji mobilnej i po stronie serwera. Rozwinięciem tej tematyki będzie pozostała część pracy magisterskiej.

## Język JavaScript

Historia JavaScript opracowana została na podstawie prezentacji Douglasa Crockforda - [1].

Prace nad nowym językiem rozpoczęły się na fali rosnącej popularności przeglądarek internetowych. W firmie Netscape Communications uznano, że istniejące dotychczas możliwości używania technologii Java po stronie klienta są nieodpowiednie, gdyż jest ona zbyt ciężka.

Autorem JavaScriptu był Brendan Eich. Z początku chciał on umieścić w przeglądarce język Scheme, ale pomysł ten został skrytykowany przez jego managera. Tak powstał LiveScript – pierwszy język skryptowy umieszczony w przeglądarce internetowej, ale też pierwszy język skryptowy umieszczony po stronie serwera. Następnie po zawarciu „przymierza” przeciw Microsoftowi przez Sun i Netscape, język zmienił nazwę na znaną nam obecnie. Stała się ona znakiem firmowym, w którego posiadanie wszedł Sun.

JavaScript został ogłoszony przez sprzymierzone firmy 4 grudnia 1995 roku i został udostępniony wraz z wersją beta przeglądarki Netscape Navigator 2.0 [2].

W 1996 Microsoft w odpowiedzi na ruch Netscape i Oracle metodą inżynierii odwrotnej stworzył własną implementację języka, którą z powodu chronionego znaku towarowego nazwał JScript.

Netscape bojąc się utraty kontroli nad swoim dziełem zgłosiło się do European association for standardizing information and communication systems – ECMA - w celu ogłoszenia JavaScriptu standardem. W praktyce dzięki wpływom w organizacji, Microsoftu przejął kontrolę nad językiem, przez co został on ustandaryzowany ze wszystkimi błędami projektowymi popełnionymi w przeszłości przez Netscape. Większość z nich występuje do dziś i są bardzo trudne do usunięcia w związku z koniecznością zachowania kompatybilności wstecznej.

### Podstawowe założenia

Składnia języka JavaScript wywodzi się z rodziny C. Poza strukturami kontrolującym przebieg programu, język ma niewiele wspólnego z innymi potomkami C - C++-em, C#-em czy Javą. Z jednej strony podobieństwo składni, a z drugiej zupełna różnica w głównych założeniach są przyczyną wielu problemów ze zrozumieniem języka.

#### System typów

JavaScript jest językiem dynamicznym, co oznacza, że jego kod nie jest kompilowany i procesy, które normalnie zachodzą w czasie kompilacji odbywają się w czasie wykonywania programu. Pozwala to na np. zmienianie definicji typów w czasie działania aplikacji.

Ponadto jest to język dynamicznie typowany – zmienne, parametry funkcji i zwracane wartości nie są ograniczone przez typ. Ta sama funkcja może zwracać w jednym przypadku łańcuch znaków w innym przypadku liczbę, co zostało pokazane na Rys. 2.1.

function foo(someNumber) {

    if (someNumber > 0) {

        return "some text";

    }

    else {

        return 10;

    }

}

Rys. . - Typowanie dynamiczne

Kolejną cechą, która wyróżnia JavaScript jest słabe typowanie – typy są automatycznie rzutowane. Pozwala to np. na dodanie łańcucha znaków i liczby bez otrzymania błędu wykonania tak jak jest to widoczne na Rys. 2.2.

var foo = "1" + 2;

console.log(foo);

//12

Rys. . - Słabe typowanie

Inną cechą systemu języka, będącą konsekwencją typowania dynamicznego, jest duck typing. Ta specyficzna nazwa pochodzi od słów amerykańskiego pisarza - Jamesa Whitcomb Riley’a: „Kiedy widzę ptaka, który chodzi jak kaczka, i pływa jak kaczka, i kwacze jak kaczka, to nazywam tego ptaka kaczką.” [3]

W językach takich jak C# czy Java obiekty mają określony typ. Z tego powodu występuje konieczność użycia interfejsu w celu umożliwienia wykonania tej samej akcji na dwóch niedziedziczących po sobie obiektach. Ilustruje to Rys. 2.3.

class Duck : IQuackable

{

    public void Quack()

    {

        Console.WriteLine("Quacking");

    }

}

class Human : IQuackable

{

    public void Quack()

    {

        Console.WriteLine("Emulating quacking");

    }

}

static class Quacker

{

    public static void MakeQuack(IQuackable quackable)

    {

        quackable.Quack();

    }

}

Rys. . - Problem duck typingu w c#

W JavaScriptcie ten problem nie występuje. Dopiero w czasie wykonywania programu sprawdzane jest czy przekazany do funkcji obiekt faktycznie zawiera potrzebną metodę – Rys. 2.4. Nie interesuje nas czy jakiś obiekt faktycznie jest klasy Duck, albo implementuje interfejs IQuackable, jak w przykładzie w języku C#. Zgodnie ze słowami Railey’a zakładamy, że jeśli posiada on metodę quack() (czyli kwacze) to jest to dla nas wystarcza informacja do potraktowania go jako kaczkę.

var duck = {

    quack : function () {

        console.log("Quacking");

    }

}

var human = {

    quack: function () {

        console.log("Emulating quacking");

    }

}

function makeQuack(anyObject) {

    anyObject.quack();

}

Rys. . - Duck typing w JavaScript

#### Wieloparadygmatowość

JavaScript jest językiem wieloparadygmatowym. Łączy w sobie podejście obiektowe, imperatywne i funkcyjne.

Obiektowość języka różni się od tej, którą można spotkać w C++, C# czy Java. W JavaScriptcie nie występuje pojęcie klasy, ale możliwe jest dziedziczenie i istnieją konstruktory, które są funkcjami budującymi obiekty i inicjalizującymi dane.

Poza paradygmatem obiektowym JavaScript łączy w sobie dwa przeciwstawne podejścia do programowania – funkcyjne i imperatywne. Ich przeciwstawność opiera się na tym, że paradygmat imperatywny zakłada, iż program w czasie swojego działania w kolejnych krokach modyfikuje dane, czyli powoduje efekty uboczne, a paradygmat funkcyjny w swojej najbardziej skrajnej formie zakłada brak istnienia takowych [4].

JavaScript stoi bliżej podejścia imperatywnego oferując również niektóre dobrodziejstwa programowania funkcyjnego takie jak funkcje wyższego rzędu czy domknięcia.

#### Podstawowe typy danych

JavaScript posiada mały zestaw typów podstawowych. Większość z nich pokrywa się z tym, co jest dostępne popularnych językach, niektóre jednak są zupełnie inne.

|  |  |
| --- | --- |
| Nazwa typu | Opis |
| Undefined | Ten typ ma dokładnie jedną wartość czyli **undefined.** Każda zmienna, która nie ma przypisanej wartości, a także nieistniejące pole obiektu mają wartość **undefined** [5]. |
| Null | Ten typ podobnie jak poprzedni ma dokładnie jedną wartość równą **null**. Różnica między **undefined** jest taka, że wartość **null** należy przypisać do zmiennej explicite [5]. |
| Boolean | Typ znany z większości języków programowania. Może przyjąć dwie wartości **true** i **false** [5]. |
| String | Łańcuch znaków, przypomina typ o tej samej nazwie z języków Java i C#. Reprezentuje skończony ciąg 16-bitowych elementów, z których każdy reprezentuje znak. Każdy z nich ma swoje miejsce w ciągu określone przez indeks. Indeksy zaczynają się od 0. Długość łańcucha to ilość 16-bitowych elementów, z których się składa. Pusty string ma długość zero i nie posiada, żadnych elementów. Tekst zakodowany jest przy użyciu UTF-16.  W JavaScript nie występuje typ znakowy znany w językach Java i C# pod nazwą char. W celu przechowania pojedynczego znaku należy użyć stringa o długości 1 [5]. |
| Number | JavaScript w przeciwieństwie do Javy i C# posiada tylko jeden typ numeryczny. Jest on 64-bitową liczbą zmiennoprzecinkową podwójnej precyzji zapisaną zgodnie z formatem IEEE 754 [5].  Number poza wartościami liczbowymi może przyjmować trzy specjalne wartości: **NaN**, **+Infinity** i **–Infinity**. Pierwsza z nich oznacza wynik działania, który w rezultacie daje coś co nie jest liczbą np. wynik 0/0. Druga i trzecia to dodania i ujemna nieskończoność. Można je uzyskać za pomocą działań 0/0, 1/0 i -1/0 [5]. |

#### Objekty

Za pomocą typów przedstawionych powyżej, można tworzyć bardziej złożone konstrukcje zwane obiektami.

W JavaScript obiekt to kolekcja elementów klucz-wartość, czyli tablica asocjacyjna, w której kluczami mogą być jedynie dane typu string, a wartościami dane dowolnego typu włączając w to inne obiekty. Pozwala to na budowanie skomplikowanych struktur danych. Za przykład może posłużyć struktura zaprezentowana na Rys. 2.5.

|  |  |
| --- | --- |
| **Klucz** | **Wartość** |
| name | Kermit |
| surname | The Frog |
| age | 57 |
| address | |  |  | | --- | --- | | **Klucz** | **Wartość** | | city | New York | | street | Sesame Street | |

Rys. .- Struktura danych prezentująca informacje o Kermicie z Ulicy Sezamkowej

Obiekt posiada cztery pary klucz wartość, z czego ostatnia – **address** – jako swoją wartość ma inny obiekt, który posiada dwie pary klucz-wartość. Tą samą strukturę danych można przedstawić w kodzie JavaScript za pomocą literału obiektowego, co zaprezentowano na przykładzie Rys. 2.6.

var kermit = {

    name: "Kermit",

    surname: "The Frog",

    age: 57,

    address: {

        city : "New York",

        street: "Sesame Street"

    }

}

Rys. . - Struktura danych prezentująca informacje o Kermicie z Ulicy Sezamkowej jako literał obiektowy

Inną formą reprezentacji powyższej struktury jest notacja słownikowa, która lepiej obrazuje, to że każdy obiekt jest tak naprawdę tablicą asocjacyjną. Zaprezentowano to na przykładzie Rys. 2.7.

var kermit = {};

kermit["name"] = "Kermit";

kermit["suraname"] = "The Frog";

kermit["age"] = 57;

var address = {};

address["city"] = "New York";

address["street"] = "Sesame Street";

kermit["address"] = address;

Rys. . - Struktura danych prezentująca informacje o Kermicie z Ulicy Sezamkowej jako tablice asocjacyjne

Ta natura obiektów pozwala je dowolnie modyfikować w czasie działania programu. Podobnie jak w przypadku słownika, można z obiektu usuwać pary klucz-wartość i dodawać je w dowolnym czasie.

#### Funkcje

Funkcje to obiekty, które zawierają kod do wykonania. Deklaruje się je za pomocą specjalnej składni, która została przedstawiona na Rys. 2.8.

function (param1, param2) {

    //kod do wykonania

}

Rys. . - Deklaracja funkcji

Najpierw następuje słowo kluczowe **function**, później para nawiasów **()**, w których znajduje się lista parametrów. Dalej w nawiasach **{}** znajduje się ciało funkcji.

Jak już zostało wcześniej wspomniane funkcje są obiektami, w związku z czym można przypisywać je do zmiennych oraz pól. Funkcje przypisane do pola obiektu zwane są metodami i posiadają specjalny wskaźnik **this**, który wskazuje na obiekt będący ich właścicielem. Ilustruje to Rys. 2.9.

var kermit = {

    name: "Kermit",

    surname: "The Frog",

    age: 57,

    address: {

        city: "New York",

        street: "Sesame Street"

    },

    sayMyName: function () {

        console.log("Hi my name is %s", this.name);

    }

}

Rys. . - Metody

Funkcje można też przekazywać jako parametry do innych funkcji, tworząc w ten sposób istotne dla programowania funkcyjnego funkcje wyższego rzędu. Dobrym przykładem jest użycie operatora map, który modyfikuje każdy element tablicy na podstawie funkcji przekazanej jako parametr. W przypadku kodu Rys. 2.10 podnosi on każdy element do kwadratu.

[1, 2, 3].map(function(x) { return x\*x });

//[ 1, 4, 9 ]

Rys. . - Funkcje wyższego rzędu - funkcja map

JavaScript oferuje charakterystyczne dla programowania funkcyjnego domknięcia. Jest to bardzo ważna cecha języka, gdyż umożliwia wygodne programowanie asynchroniczne, co omówione zostanie w następnym podrozdziale.

Domknięcia pozwalają funkcji wewnętrznej na dostęp do wszystkich zmiennych zadeklarowanych w funkcji zewnętrznej, nawet wtedy gdy ta już się wykonała. Ten mechanizm ilustruje Rys. 2.11.

function startCounting() {

    var counter = 1;

    var timer = setInterval(function () {

        console.log("Counting: " + counter);

        counter++;

        if (counter > 10) {

            clearInterval(timer);

        }

    }, 1000);

}

startCounting();

console.log("Counting started!");

//Counting started!

//Counting: 1

//Counting: 2

//Counting: 3

//Counting: 4

//Counting: 5

//Counting: 6

//Counting: 7

//Counting: 8

//Counting: 9

//Counting: 10

Rys. . - Domknięcia – przykład

W pierwszej kolejności deklarowana jest funkcja startCounting(), która natychmiast zostaje wywołana. Sterowanie przechodzi do pierwszej linii statCounting(), w której deklarowana jest zmienna counter o wartości jeden. Następnie za pomocą wbudowanej funkcji setInterval(), tworzony jest timer, który co 1000 milisekund będzie wywoływał podaną mu funkcję. Timer jest przypisywany do zmiennej timer, po czym sterowanie opuszcza ciało startCounting().

Kolejnym krokiem jest wypisanie tekstu "Counting started!" do konsoli. Program czeka przez kolejną sekundę nie robiąc nic, aby po upływie tego czasu otrzymać zdarzenie z timera, które przekaże sterowanie do ciała anonimowej funkcji przekazanej jako pierwszy argument do setInterval().

Funkcja ta za każdym razem wypisze wartość zmiennej counter na ekran, a w przypadku gdy counter osiągnie 10 przerwie dalsze obliczanie.

Najważniejszym elementem przedstawionego wyżej programu jest to, że funkcja anonimowa ma dostęp do zmiennej counter i timer zadeklarowanych w funkcji w której została zagnieżdżona, nawet gdy ta już dawno się wykonała.

#### Tablice

W JavaScript tablice są obiektami, które posiadają kilka specjalnych właściwości. Pierwszą z nich jest składnia ich tworzenia, która została już zaprezentowana na przykładzie Rys. 2.10. Drugą jest wbudowane pole length, które jest równe liczbowej wartości największego indeksu + 1. To dziwne zachowanie spowodowane jest faktem, że JavaScript nie posiada tablic znanych z większości języków programowania, a jedynie tablice asocjacyjne, w których kluczami są łańcuchy znaków utworzone z numerów indeksów, co zaprezentowano na Rys. 2.12.

var array = [1, 2, 3];

console.log(array[2]);

//3

console.log(array["2"]);

//3

Rys. . - Tablica i obiekt jako jedno

Odwołując się do elementu pod indeksem 2 tak naprawdę odwołujemy się do wartości tablicy asocjacyjnej o kluczu będącego łańcuchem znaków „2”.

#### Instrukcje sterujące

JavaScript posiada zestaw instrukcji sterujących typowy dla języków z rodziny C. Do najważniejszych należą instrukcja if else, while i for. Ze względu na powszechną znajomość języków z rodziny C, nie zostaną one tutaj opisane.

#### Konstruktory i tworzenie obiektów

Konstruktory w JavaScript przypominają te znane z języków Java czy C#, ale poza zadaniem ustawienia początkowych wartości pól, muszą jeszcze te pola stworzyć.

var Character = function (name, surname) {

    this.name = name;

    this.surname = surname;

    this.sayMyName = function () {

        console.log("My name is " + name + " " + surname);

    }

}

var character = new Character("Kermit", "the Frog");

character.sayMyName();

//My name is Kermit the Frog

Rys. . - Konstruktor w JavaScript

Działanie konstruktora pokazuje Rys. 2.13. W pierwszej linii deklarowana jest zmienna Character, do której przypisana zostaje funkcja będąca konstruktorem. Przyjmuje ona dwa parametry, którymi zostanie zainicjalizowany nowy obiekt.

Specjalna zmienna this wskazuje na aktualnie tworzony obiekt, do którego dodawane są kolejno elementy – pole name, surname i metoda sayMyName().

Użycie konstruktora wygląda podobnie do użycia zwyczajnej funkcji z tym, że wywołanie jest poprzedzone słowem kluczowym new.

Jak już zostało wspomniane JavaScript technicznie rzecz biorąc nie posiada klas. Pojęcie klasy jest jednak bardzo przydatne w omawianiu wzorców oraz zasad programowania obiektowego, dlatego na potrzeby tej pracy dyplomowej zostanie sformułowana definicja klasy w JavaScript.

Dokumentacja języka Java oferuje prostą definicję klasy: „klasa to schemat na podstawie, którego tworzone są indywidualne obiekty” [6]. W języku JavaScript taką rolę pełnią funkcje. Wobec tego zdefiniujmy klasę następująco:

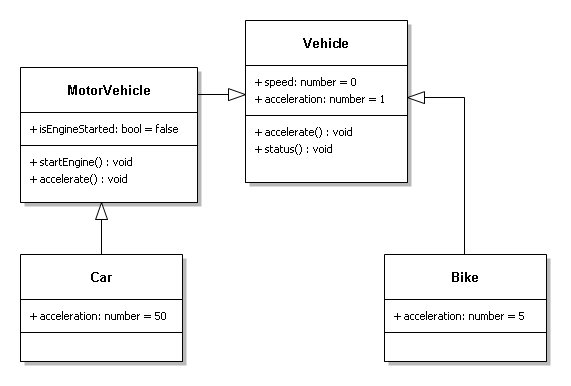
**Klasą jest schemat, zapisany w funkcji budującej za każdym wywołaniem obiekty o identycznej strukturze, na podstawie którego tworzone są indywidualne obiekty.**

W JavaScript funkcja może zbudować dwa obiekty o zupełnie różnym zestawie metod i pól, dlatego do definicji został dodany fragment: „budującej za każdym wywołaniem obiekty o identycznej strukturze”, który nakłada na klasy znane z języków takich jak Java czy C# ograniczenia.

#### Dziedziczenie prototypowe

Jak zostało to wyjaśnione wyżej JavaScript nie posiada prawdziwych klas, nie zmienia to jednak faktu, że możliwe jest wykorzystanie w nim dziedziczenia. Podczas omawiania tego mechanizmu należy pamiętać jaki jest jego cel – wielokrotne użycie kodu. Można go osiągnąć na różne sposoby i często dziedziczenie nie jest najlepszym rozwiązaniem.

W języku JavaScript obiekty dziedziczą bezpośrednio po innych obiektach – nazywa się to dziedziczeniem prototypowym. W celu ilustracji tego zagadnienia zostanie stworzona prosta hierarchia pojazdów, którą przedstawia diagram Rys. 2.14.



Rys. 2.14 - Hierarchia pojazdów - diagram UML

var Vehicle = function () {

    this.speed = 0;

    this.acceleration = 1.0;

    this.accelerate = function () {

        this.speed += this.acceleration;

    }

    this.status = function(){

        if (this.speed > 0) {

            console.log("Moving with %s speed", this.speed);

        }

        else {

            console.log("Not moving");

        }

    }

}

var MotorVehicle = function () {

    this.isEngineStarted = false;

    this.startEngine = function () {

        this.isEngineStarted = true;

    }

    this.accelerate = function () {

        if (this.isEngineStarted) {

            this.speed += this.acceleration;

        }

    }

}

MotorVehicle.prototype = new Vehicle();

var Car = function () {

    this.acceleration = 50.0;

}

Car.prototype = new MotorVehicle();

var Bike = function () {

    this.acceleration = 5.0;

}

Bike.prototype = new Vehicle();

var car = new Car();

var bike = new Bike();

car.accelerate();

car.status();

//Not moving

car.startEngine();

car.accelerate();

car.status();

//Moving with 50 speed

bike.accelerate();

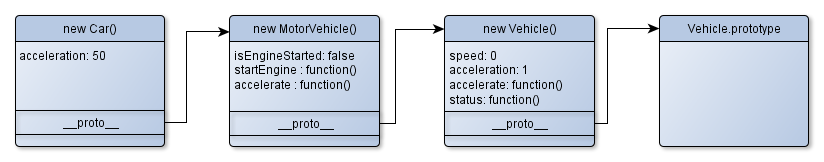
bike.status();

//Moving with 5 speed

Rys. 2.15 - Dziedziczenie prototypowe

Rys. 2.15 pokazuje prezentuje implementację hierarchii pojazdów. Najpierw tworzony jest konstruktor Vehicle. Posiada on dwie proste metody i dwa pola. Wyspecjalizowaną wersją pojazdu jest pojazd zmotoryzowany – MotorVehicle. Jedyną różnicą między nim i zwykłym pojazdem jest to, że przed jazdą trzeba uruchomić silnik, za pomocą metody startEngine(). Informacja o tym, że MotorVehicle dziedziczy po Vehicle, zapisana jest poprzez przypisanie MotorVehicle.prototype = new Vehicle(). Właściwość prototype konstruktora pełni specjalną rolę – wskazuje na obiekt, w którym JavaScript będzie szukał pól i metod w przypadku nieznalezienia ich w obiekcie utworzonym za pomocą konstruktora.

W celu wyjaśnienia tego mechanizmu najlepiej spojrzeć na to jak obiekty wyglądają w pamięci i jak wygląda dostęp do pól oraz wywołanie funkcji. Diagram Rys. 2.16 przedstawia strukturę obiektu stworzonego za pomocą konstruktora Car.



Rys. . - Obiekt car - struktura w pamięci

Teraz kiedy następuje wywołanie metody car.status(), interpreter JavaScript najpierw szuka jej w obiekcie new Car(). Jak widać z diagramu new Car() nie zawiera wspomnianej metody, dlatego interpreter próbuje znaleźć ją w prototypie. Pole prototype występuje w konstruktorach, a nie instancjach obiektów, konkretne obiekty mają jednak ukryte pole, które wskazują na ten sam obiekt co prototype z konstruktora. Na diagramie oznaczono je jako \_\_proto\_\_ co odpowiada nazewnictwu stosowanemu przez przeglądarkę Mozilla [7]. Interpreter zagląda do obiektu new MotorVehicle(), w którym również nie znajduje poszukiwanej metody, patrzy więc do obiektu new Vehicle(), w którym się ona znajduje. Właśnie ta metoda zostanie wywołana.

W metodzie status() znajduje się odwołanie do this.speed. Cały proces powtarza się – interpreter szuka odpowiedniego pola zaczynając od new Car(), na który wskazuje zmienna this, w końcu dociera do właściwej wartości występującej w new Vehicle().

Ciekawiej wygląda odwołanie do car.accelerate. Obiekt new Car() nie posiada żadanej metody w związku z czym interpreter zagląda do new MotorVehicle(), który ją ma. Szukanie zostaje przerwane i w rezultacie zostaje wywołana metoda accelerate z new MotorVehicle(), a nie z new Vehicle(). Wszystkie składowe obiektu potomnego przesłaniają te zadeklarowane w obiekcie rodzicu.

Ograniczeniem dziedziczenia prototypowego jest to, że nie ma możliwości stworzenia składowych klasy, do których dostęp będą miały jedynie obiekty klasy potomnej. Takie ograniczenie dostępu znane jest w C# i Java pod nazwą protected.

#### JSON

JSON (JavaScript Object Notation) jest podzbiorem języka JavaScript. Podobnie jak XML jest to format tekstowy. Opiera się on na składni tworzenia obiektów i tablic JavaScript. Użycie JSON-a ilustruje to Rys. 2.17. Widoczny na nim jest obiekt przedstawiony na Rys. 2.5 zserializowany do formatu JSON.

Najważniejszymi różnicami między notacją obiektową JavaScript, a JSON-em jest konieczność umieszczania nazw pól w cudzysłowach (w JavaScript jest to opcjonalne) oraz brak wsparcia dla funkcji i dla obiektu Date.

{

    "name": "Kermit",

    "surname": "The Frog",

    "age": 57,

    "address": {

        "city": "New York",

        "street": "Sesame Street"

    }

}

Rys. . - Obiekt zserializowany do formatu JSON

### Popularność

Pomimo tego, że JavaScript istnieje już od siedemnastu lat, dopiero ostatnimi czasy zdobył on popularność i zainteresowanie społeczności programistów.

Mierzenie zainteresowania językiem nie jest rzeczą prostą. Istnieje kilka podejść do tego zagadnienia, a każde z nich daje inne rezultaty. Z tego powodu nie można traktować wyników zbyt ściśle, należy patrzeć na nie przez pryzmat metodologii za pomocą, której zostały uzyskane.

Najbardziej znanym indeksem popularności języków jest TIOBE Index. Porównuje on popularność ponad dwustu różnych języków programowania. Opiera się na ilości wyników uzyskanych w wyszukiwarce google. Według wyników z sierpnia 2012 roku JavaScript ma jedenaste miejsce [8]. Wyniki języka na tle innych zaprezentowano na wykresie Rys. 2.17, widać z niego że JavaScript jest dość daleko za najpopularniejszymi językami takimi jak C czy Java.

Rys. .-Popularność języków według TIOBE Index (sierpień 2012)

Innym indeksem popularności języków jest stosujący podobną metodologię co TIOBE, The Transparent Language Popularity Index. Opiera się on na tych samych założeniach, z tym że jest otwarty – można pobrać zarówno dane na bazie, których jest tworzony jak i narzędzia, które są do tego celu używane [9]. W tym zestawieniu JavaScript osiąga identyczny rezultat, chociaż wyniki konkurencji są nieznacznie inne – Rys. 2.18.

Rys. . Popularność języków według The Transparent Language Popularity Index (sierpień 2012)

Badanie ilości wyników wyszukiwania nie jest jedyną mierzenia popularności. Kilka innych podejść proponują Klint Finley [10] i Joe Brockmeier [11] w artykułach na portalu ReadWriteWeb.

Jednym z nich jest badanie ilości projektów na portalu GitHub, który jest obecnie najpopularniejszym miejscem hostowania kodu open source [12]. Wyniki takiego porównania [13] prezentują się zaskakująco. Z wykresu Rys. 2.19 widać, że w tym zestawieniu dominuje JavaScript, dalej jest Ruby, a wiodące prym w zestawieniach bazujących na wynikach wyszukiwania Java i C są zdecydowanie z tyłu.

Rys. . Popularność języków według GitHub (sierpień 2012)

Prawdopodobnym wyjaśnieniem tego zjawiska jest to, że GitHub skupia się na społeczności OpenSource, a więc w dużej mierze hobbystach programujących w domu dla przyjemności, a nie tych tworzących komercyjne rozwiązania w pracy. W związku z tym, można wyciągnąć wniosek, iż język jest obecnie bardzo modny i posiada dynamicznie rozwijający się ekosystem, który w przyszłości może stanowić bazę dla wielu rozwiązań komercyjnych.

Wnioski te potwierdza inne zestawienie – wielkość sprzedaży książek za rok 2011 wydawnictwa O’Reilly [14]. Dane zostały zestawione na wykresie Rys. 2.20.

Rys. . Popularność języków według ilości sprzedanych tytułów wydawnictwa O'Reilly (rok 2011)

Wysoka pozycja JavaScriptu potwierdza teorię, iż jest to obecnie „gorący temat" w środowisku programistycznym.

Kolejnym interesującym zestawieniem jest ranking języków i technologii opracowany na podstawie ofert pracy z serwisu dice.com [15]. Tutaj JavaScript zajmuje również wysoką pozycję – Rys. 2.21.

Rys. . Popularność języków na bazie ofert pracy z portalu dice.com

Jedną z przyczyn jest z pewnością to, że pracodawcy często wymagają znajomości JavaScriptu razem z innymi językami np. HTML, Ruby czy PHP, które są wykorzystywane głównie do tworzenia serwisów internetowych.

Przedstawione wyżej indeksy pokazują, iż JavaScript jest językiem o stale rosnącej popularności. Wynika to z przede wszystkim z rosnącego znaczenia aplikacji webowych, w których przeglądarka ma do zrobienia dużo więcej niż tylko wyświetlanie HTML-a wyrenderowanego przez serwer www.

Poza programowaniem klienckiej strony aplikacji internetowych, język ostatnio znajduje zastosowanie w tworzeniu kodu znajdującego się po stronie serwera oraz aplikacji mobilnych przeznaczonych na różne platformy. Ta tematyka zostanie szerzej omówiona w kolejnych podrozdziałach.

## Problematyka mnogości platform mobilnych

Wraz z premierą pierwszego iPhone’a, która miała miejsce w 2007 roku na rynku urządzeń mobilnych rozpoczął się boom. Dzięki doskonałej stylistyce, intuicyjnemu interfejsowi i intensywnej kampanii reklamowej telefon szybko zdobył zainteresowanie szerokiej gamy ludzi, rozpoczynając ekspansję smartphone’ów poza rynek klientów biznesowych.

Do Apple zaczęli dołączać kolejni producenci sprzętu i oprogramowania, oferując bogate w funkcjonalności urządzenia w różnych przedziałach cenowych. Obecnie na rynku dostępnych jest kjilka istotnych systemów operacyjnych. Zapoznanie się nimi i zrozumienie poziomu ich zróżnicowania jest niezwykle istotne dla zrozumienia roli jaką zaczyna odgrywać JavaScript na rynku aplikacji mobilnych.

### Najważniejsze platformy mobilne

#### iOS

iOS jest obecnie drugim najpopularniejszym systemem operacyjnym obecnym w Smartphone’ach pod względem ilości sprzedanych egzemplarzy [16]. Wywodzi się on z systemu OS X, który to należy do rodziny Unix. System występuje na telefonach iPhone, tabletach iPad oraz odtwarzaczach mp3 iPod Touch.

iOS jest zamknięty - źródła nie są wolno dostępne, ponadto użytkownik nie ma pełnej kontroli nad systemem, a aplikacje mają nałożony szereg ograniczeń. Jednym z nich jest konieczność przejścia rygorystycznego systemu weryfikacji przed umieszczeniem w sklepie. Taki zabieg z jednej strony utrudnia tworzenie oprogramowania na platformę i umożliwia cenzurowanie dostępnej treści, ale z drugiej strony zapewnia wyższą jakość.

Aplikacje na iOS pisze się w języku Objective-C, którego składnie to połączenie C i Smalltalka. Głównym środowiskiem w wytwarza się aplikacje jest IDE X-Code [17].

#### Android

Android to system operacyjny stworzony i udostępniony w 2009 roku przez firmę Google. Obecnie jest on głównym przeciwnikiem iOS-a na rynku konsumenckich urządzeń mobilnych. Według danych z drugiego kwartału 2012 urządzenia z systemem Android sprzedają się najlepiej [16].

W przeciwieństwie do iOS-a Android jest systemem otwartym, kod źródłowy dostępny jest na licencji Apache [18]. Dzięki temu zdobył on duże zainteresowanie społeczności open source, która tworzy wiele modyfikacji systemu operacyjnego w tym CyanogenMod, oferującą zestawem ulepszeń związanych z prywatnością i wydajnością.

Głównym językiem programowania na platformie Android jest Java. Kod źródłowy w niej napisany jest kompilowany do byte kodu i uruchamiany na wirtualnej maszynie Dalvik. Poza Javą można tworzyć również natywne aplikacje przy użyciu języków C i C++.

Omawiany system operacyjny występuje na urządzeniach z różnych zakresów cenowych. Można go znaleźć na tanich telefonach jak Samsung Galaxy Mini, a także na urządzeniach które mają możliwości podobne co iPhone jak Samsung Galaxy SIII.

#### Windows Phone 7

W odpowiedzi na sukces iOS i Androida, Microsoft stworzył własną platformę, która miała zastąpić wysłużonego już, przeznaczonego głównie dla biznesu Windowsa Mobile. Windows Phone 7 jest systemem młodym, swoją premierę miał w 2010 roku. W przeciwieństwie do poprzednika, posiada on interfejs w całości przystosowany do obsługi za pomocą palca bez użycia rysika [19].

Podobnie ja iOS WP7 jest systemem zamkniętym. Użytkownik nie ma dostępu do kodu źródłowego, nie może też samodzielnie wgrywać aplikacji na swój telefon. Są one dostępne jedynie za pomocą sklepu Marketplace, na którym obowiązuje selekcja.

Aplikacje na WP7 można pisać jedynie przy użyciu technologii .NET i Silverlight. Oficjalnie wspieranym językiem jest C#, ale jest też możliwość użycia innych np. Visual Basic .NET czy F# [20].

System operacyjny występuje na urządzeniach wielu producentów. Wśród nich należy koniecznie wymienić Nokię, która produktami s serii Lumia stara się zastąpić telefony z umierającym systemem Symbian.

#### Symbian

Symbian to system frimy Nokia, który swoją premierę miał w 1997 roku. Od tamtego czasu system stracił większość swojej świetności i obecnie jest w etapie wygaszania i ma zostać zastąpiony przez WP7. Głównym językiem programowania na Symbianie jest C++ wraz z technologią Qt [21].

#### Bada OS

Samsung poza wytwarzaniem telefonów działających w oparciu o Androida sam próbuje sił w tworzeniu własnego systemu operacyjnego. W 2010 firma wypuściła przeznaczony dla urządzeń mobilnych system Bada OS. Podobnie jak w przypadku Androida, iOS-a i WP7 aplikacje na Bada OS są udostępnianie w jednym sklepie zwanym Samsung Apps. Głównymi językami programowania na Bada są C i C++ [22].

#### Blackberry OS

Blackberry OS to system operacyjny przeznaczony dla telefonów Blackberry kanadyjskiego producenta firmy Resaearch in Motion. Miał on swoją premierę w 1999 roku. Telefony Blackberry są głównie popularne wśród klientów biznesowych i rządowych. Istotnym cechą jest szyfrowanie danych pomiędzy urządzeniem i serwerem Blackberry Enterprise Server. Głównym językiem wytwarzania aplikacji na Blackberry OS jest Java [23].

### Problemy wynikające z mnogości platform

W poprzednim podrozdziale przedstawiono przegląd najpopularniejszych platform mobilnych. Na rynku występuje duże zróżnicowanie, co jest generalnie rzeczą dobrą, bo zapewnia klientom szeroki wybór i konkurencję zapewniającą ciągły wzrost jakości oferowanych rozwiązań.

Niestety to zróżnicowanie stanowi ogromny problem dla twórców aplikacji. Bardzo ciężko jest wykorzystać elementy przygotowane dla jednej platformy na innej, a stworzenie całkowicie przenośnego programu posługując się dedykowanymi dla danej platformy technologiami jest niemożliwe.

Jest to duży problem dla firm tworzących oprogramowanie, które z założenia ma być szeroko dostępne, gdyż wymaga to znacznie większych nakładów pracy i zatrudniania specjalistów od różnych technologii.

## Język JavaScript w aplikacjach mobilnych

Z powodu przedstawionych wyżej problemów popularność zdobywają dwa inne sposoby wytwarzania mobilnych aplikacji.

### Aplikacje webowe

Pierwszym jest tworzenie aplikacji webowych. Jak się okazuje większość współczesnych smarphone’ów oferuje dobre wsparcie dla HTML5, dzięki czemu można bez większych obaw wykorzystać tą technologię do tworzenia mobilnych aplikacji [24].

Mobilna aplikacja webowa to po prostu interaktywna strona www. Od zwykłej strony internetowej różni ją to, że jest przystosowana do obsługi za pomocą interfejsu dotykowego na małym ekranie telefonu.

Niektóre mobilne aplikacje webowe oferują interfejs typowy dla danej platformy np. aplikacja webowa portalu LinkedIn, która ma interfejs iOS. Czasem zawierają też wsparcie dla obsługi zdarzeń dotykowych, które nie są jeszcze częścią standardu HTML5 [25].

W przypadku aplikacji webowych proces dystrybucji jest znacznie uproszczony. Nie ma potrzeby spełniania surowych wymagań sklepu z aplikacjami, ani czekania na akceptację. Nie ma też konieczności dzielenia się zyskami z producentem systemu. Ponadto użytkownik zawsze korzysta z najnowszej wersji programu.

Z aplikacjami webowymi wiążą się dwa problemy. Pierwszym jest konieczność ciągłego połączenia do Internetu. Częściowym rozwiązaniem jest zastosowanie funkcjonalności HTML5 o nazwie application cache, która pozwala poinformować przeglądarkę, o tym jakie zasoby ma umieścić w cache’u na wypadek utraty połączenia z siecią. W połączeniu z Web Storage czyli małą bazą danych po stronie przeglądarki, pozwala to na utrzymanie funkcjonowania aplikacji bez dostępu do Internetu.

Drugim z problemów jest dostęp do specjalnych funkcji telefonu. Aplikacja webowa nie ma możliwości odczytania danych z akcelerometru, czy użycia kamery w celu zrobienia zdjęcia. Aktualnie nie istnieje rozwiązanie tego problemu, co jest zrozumiałe gdyż danie większej kontroli stronom www nad telefonem byłoby zagrożeniem prywatności użytkownika.

### Aplikacje hybrydowe

Próbą znalezienia złotego środka między aplikacjami natywnymi i webowymi są aplikacje hybrydowe. Założenie jest proste – pozwolić programistom wykorzystywać przenośność i prostotę HTML-a, ale też dać dostęp do ważnych funkcji telefonu.

Aplikacje hybrydowe to aplikacje natywne zawierające kontrolkę przeglądarki www (UIWebView dla iOS, android.webkit.WebView dla Androida, WebBrowser dla WP7 itd.), rozciągniętą na cały ekran. Taka kontrolka w przeciwieństwie do zwyczajnej przeglądarki nie posiada elementów służących do nawigacji w tym paska adresu - Rys. 2.22.

http://app.com

Rys. . Po lewej stronie zwyczajna przeglądarka www, po prawej kontrolka przeglądarki

W kontrolce przeglądarki wyświetlany jest interfejs utworzony w technologii HTML, a także wykonywana jest większość kodu aplikacji napisanego w języku JavaScript. Kontrolka przeglądarki wraz z otaczającym ją kodem aplikacji natywnej zwane są kontenerem.

Podstawową praktyczną różnicą między aplikacją webową i hybrydową jest to, że kontener udostępnia API pozwalające na dostęp do natywnych funkcjonalności np. listy kontaktów, kamery czy akcelerometru. Jest ono widoczne po stronie części webowej jako zestaw obiektów JavaScript.

Architekturę aplikacji hybrydowej prezentuje diagram Rys. 2.23. Kolorem niebieskim zaznaczono aplikację natywną, w której działa część webowa, zaznaczona kolorem czerwonym. Część webowa „widzi” API wystawiane przez kontener, w tym przypadku np. obiekt camera. Kiedy kod JavaScript korzysta z metod API kontenera, ten wywołuje odpowiednie metody natywnego API telefonu umożliwiając tym samym dostęp do jego funkcji.

Poza udostępnianiem zaawansowanych funkcji urządzenia, aplikacje hybrydowe mają inną ważną cechę – są instalowane na telefonie tak samo jak aplikacje natywne, gdyż kontener jest aplikacją natywną. Wiąże się to z koniecznością przejścia procesu certyfikacji w sklepie danego producenta, ale z drugiej strony oferuje możliwość uruchomienia aplikacji bez dostępu do Internetu.

Najbardziej istotnym aspektem aplikacji hybrydowych jest jednak ich przenośność. Dzięki temu, że większość kodu jest napisana przy użyciu JavaScript i HTML5, uruchomienie tego samego programu na wielu urządzeniach jest rzeczą bardzo prostą.

Część webowa

Aplikacja natywna - kontener

Telefon

Komunikacja przez natywne API

camera.getPicture()

contacts.find()

contacts.create()

...

Rys. . Architektura aplikacji hybrydowej

## Język JavaScript w aplikacjach serwerowych

Niniejszy podrozdział został opracowany głównie w oparciu o dwie prezentacje autora technologii Node.js [26] i [27]**.**

### Historia powstania node.js

Przed rokiem 2009 JavaScript występował głównie w przeglądarkach internetowych. Poza krótkim epizodem na początku swojego życia język nie gościł na serwerach www. Sytuacja zmieniła się za sprawą Ryana Dahla - amerykańskiego doktoranta matematyki.

Na pewnym etapie swojego życia postanowił on porzucić uczelnię oraz matematykę i wyjechać do Chile. Próbował tam zarabiać na życie nauczyciel angielskiego, jednak nie osiągnął w tym zawodzie sukcesów. Wtedy poznał programistę PHP z USA, który zainteresował go wytwarzanie aplikacji internetowych.

Z PHP zainteresowania Ryana rozwinęły się w kierunku języka Ruby i frameworka Rails. Podczas pracy z Railsami zauważył problemy wydajnościowe. Kiedy zaczął się im przyglądać zrozumiał, że dużym problemem Railsów jest to, iż potrafią przetwarzać tylko jedno żądanie w danym czasie – nie obsługują wielowątkowości. Dahl podjął próbę naprawienia frameworka, ale głębsze spojrzenie na problem pokazało, ze nie jest to wada samych Railsów tylko Rubiego, który nie posiada dobrego wsparcia dla systemowych wątków.

Matematyk zaczął, więc pisać własny serwer. Zainspirowany projektem Mongrel postanowił umieścić go w bibliotece, a nie osobnej aplikacji. Najpierw do swych celów próbował użyć języka C, ale okazało się, że takie rozwiązanie jest nie do przyjęcia w środowisku programistów aplikacji internetowych. W międzyczasie wpadł mu do głowy pomysł, że jednowątkowość wcale nie jest zła, pod warunkiem, że nie występują operacje blokujące np. długie oczekiwanie na odczyt z pliku na dysku.

Analizując języki, które można by zastosować do implementacji koncepcji jednowątkowego serwara www zrozumiał, że idealnie do tego celu nadawał się JavaScript. Język nie posiadał żadnych bibliotek do operacji na systemie plików czy socketach. Oznaczało to, że można wszystko napisać od zera i od razu zrobić to dobrze, wymuszając na programiście podejście asynchroniczne.

W 2009 roku Ryhan Dahl pracował w Niemczech, zajmując się tworzeniem małych, szybkich serwerów http. Postanowił rzucić pracę i przez sześć miesięcy pracować nad nowym projektem o nazwie Node.js.

Efekty swoich prac zaprezentował w listopadzie tego samego roku, na konferencji JSConf w Berlinie. Przedstawił tam projekt serwera IRC napisanego w 400 linijkach w języku JavaScript. Prezentacja okazała się sukcesem. Zaraz po niej skonatkował się z Ryhanem przedstawiciel firmy Joyent, w której również próbowano używać JavaScriptu po stronie serwera. W rezultacie autor Node’a wrócił do swojego rodzinnego kraju, aby komercyjnie zajmować się zainicjowanym przez siebie projektem

### Główna idea node.js

Przez większość czasu istnienia serwerów www operacje wejścia/wyjścia były na nich wykonywane w sposób blokujący. To podejście ilustruje Rys. 2.24. Wynik zapytania SQL jest zwracany przez funkcję query(), która odda sterowanie dopiero po otrzymaniu odpowiedzi z bazy danych. W tym czasie wątek czeka i kolejna linia, czyli wywołanie console.log() nie wykona się.

var results = query("SELECT \* FROM books WHERE title = 'The Hobbit'");

console.log(results);

//[ { id: 1, title: 'The Hobbit', price: 29.99 } ]

Rys. . - Blokujące odwołanie do bazy danych

Odwołania do pamięci cache procesora i pamięci RAM są bardzo szybkie, ale już odwołania do dysku czy zasobów dostępnych przez sieć są bardzo wolne. W czasie oczekiwania na odpowiedź serwer nie robi nic. Żeby w tym czasie nie blokować możliwości obsługi kolejnych żądań, serwery www tworzą większe ilości wątków.

Wadą takiego rozwiązania jest, zwiększone zużycie pamięci (konieczna jest alokacja stosu dla każdego wątku), ponadto przełączanie kontekstu kosztuje cykle procesora. Nie jest to uciążliwe, w większości aplikacji, niestety staje się problemem w wypadku serwerów obsługujących bardzo dużo prostych żądań. W takim przypadku konsumpcja pamięci i czas potrzebny na odpowiedź znacznie rosną.

Alternatywnym podejściem jest zastosowanie nieblokujących operacji wejścia/wyjścia. Dzięki temu można uzyskać dużą responsywność serwera mając do dyspozycji tylko jeden wątek. Istotę operacji asynchronicznych przedstawia Rys. 2.25. Tym razem funkcja query() zwraca sterowanie natychmiastowo, ale w zamian przyjmuje dodatkowy argument – funkcję, która zostanie wywołane po otrzymaniu rezultatów z bazy danych.

query("SELECT \* FROM books WHERE title = 'The Hobbit'", function (err, results) {

    console.log(results);

    //[ { id: 1, title: 'The Hobbit', price: 29.99 } ]

});

Rys. . - Nieblokujące odwołanie do bazy danych

Dzięki mechanizmowi wywołań zwrotnych proces serwera nigdy nie czeka bezczynnie na długą operację wejścia/wyjścia, zamiast tego może zająć się obsługą kolejnych żądań i wrócić do kodu działającego na otrzymanych danych dopiero wtedy gdy są one dostępne.

Interesującą rzeczą związaną Node.js jest to, czemu takie podejście nie zostało zaimplementowane wcześniej. Ryan Dahl prezentuje kilka potencjalnych powodów, które dzieli na dwie kategorie kulturowe i infrastrukturalne.

**Powód kulturowy:**

* **Programiści nie są przyzwyczajeni do pisania nieblokującego kodu.**

**Powody infrastrukturalne:**

* **W przypadku serwera działającego na tylko jednym wątku wszystkie długotrwałe operacje muszą być nieblokujące. Wystarczy jeden blokujący element wejścia/wyjścia i cała aplikacja zostanie zablokowana.**
* **Niektóre biblioteki nie mówią wprost czy ich funkcje stosują operacje wejścia/wyjścia, co sprawia problem z użyciem ich w środowisku jednowątkowym.**
* **W wielu popularnych językach np. C nie ma wsparcia dla wywołań zwrotnych – brak funkcji anonimowych i domknięć – przez co pisanie kodu nieblokującego jest trudne.**
* **Wiele ważnych bibliotek nie oferuje asynchronicznych interfejsów, przerzucając na użytkownika konieczność otoczenia ich wątkami.**
* **Niektóre technologie oferują asynchroniczne serwery np. EventMachine dla Rubiego czy Twisted dla Pythona, ale cały ekosystem jest synchroniczny co utrudnia pisanie oprogramowania.**

**JavaScript jest językiem, który doskonale sprawdza się w tworzeniu aplikacji asynchronicznych. Posiada on funkcje anonimowe i domknięcia, dzięki którym można sobie znacznie łatwiej radzić z wywołaniami zwrotnymi. Ponadto praca na jednym wątku jest tutaj czymś naturalnym - silniki JavaScript w przeglądarkach są jednowątkowe jeśli nie liczyć technologii WebWorkers (która jednak izoluje dodatkowe wątki od siebie i wątku głównego).**

**Te aspekty zaowocowały kulturą skierowaną w kierunku programowania zorientowanego na zdarzenia, które nie jest zbyt popularne innych językach, dzięki czemu JavaScript nadaje się idealnie do pisania wydajnych, jednowątkowych aplikacji.**

### **Podstawowe założenia node.js**

Ryan Dahl oparł Node.js o kilka podstawowych założeń:

* Żadna funkcja nie powinna wykonywać synchronicznie operacji wejścia/wyjścia. Zamiast tego powinna przyjmować funkcję, która zostanie zawołana gdy wyniki trafią do pamięci.
* Jeśli jakaś funkcja nie oferuje mechanizmu wywołania zwrotnego to znaczy, że odwołuje się tylko do pamięci RAM, a jej wykonanie jest szybkie.
* Przed programistami nie będą ukrywane przydatne niskopoziomowe funkcjonalności.
* Node.js powinien posiadać wbudowane wsparcie dla najważniejszych protokołów takich jak np. DNS i HTTP.
* Wparcie dla zaawansowanych funkcjonalności protokołu HTTP np. chunked encoding.
* API powinno być przyjazne dla programistów.
* Node.js powinien być wydawany na prostej licencji - MIT License.
* Powinien mieć mało zależności.
* Powinien być przyjemny w użyciu.

### Niskopoziomowa architektura node.js

Schemat niskopoziomowej architektury został przedstawiony na diagramie Rys. 2.26. Node.js składa się z dwóch części – pierwsza oznaczona na niebiesko jest napisana w języku C, druga oznaczona na zielono jest napisana w JavaScript.

Najważniejszym elementem Node’a jest maszyna wirtualna Google V8, która potrafi kompilować kod JavaScript, zapewniając wysoką wydajność wykonywanych na niej programów.

Większość bibliotek języka C jest synchroniczna, dlatego Node posiada pulę wątków, które wykonują blokujące operacje wejścia/wyjścia i przekazują informacje o ich wykonaniu do pętli zdarzeń.

Rys. . - Niskopoziommoa architektura Node.js

Wyżej na diagramie znajduje się warstwa wiążąca, której zadaniem jest połączenie kodu napisanego w C i JavaScript.Na samej górze jest Bibliioteka standardowa Node, czyli cały kod serwera, który został napisany w JavaScripcie. Ta część składa się z największej ilości kodu.

Ideą, która przyświecała autorowi przy projektowaniu architektury, była chęć całkowitej izolacji programisty od aspektów wielowątkowości. Ryhan Dahl chciał, żeby każdy kto będzie pragnął użyć wątków musiał napisać swój kod w C, pozostawiając tym samym część JavaScriptową prostą i czystą.

## Język JavaScript w aplikacjach desktopowych

Poza zastosowaniem do tworzenia aplikacji webowych, mobilnych i serwerowych, JavaScript znalazł również swoje miejsce na desktopach.

Jednym z najbardziej znaczących przykładów jest możliwość tworzenia aplikacji JavaScript i HTML na nowy system operacyjny Microsoftu – Windows 8. Poza możliwością korzystania ze standardowych funkcjonalności HTML5 i JavaScript, aplikacje przeznaczone na tą platformę mogą używać też systemowego API - WinRT, dzięki któremu ich zakres możliwości jest znacznie większy niż standardowych aplikacji webowych [28].

Innym zastosowaniem JavaScript jest wykorzystanie języka w celu pisania skryptów, rozszerzających działanie aplikacji. Znany producent oprogramowania graficznego – firma Adobe – oferuje możliwość automatyzacji większości swoich produktów za pomocą języka JavaScript [29].

Wiele współczesnych przeglądarek pozwala na tworzenie pluginów właśnie z użyciem omawianego języka.

# Jakość kodu JavaScript

## Definicja jakości kodu

Istnieją dwa aspekty badania jakości kodu: funkcjonalny i strukturalny. Aspekt funkcjonalny mówi o tym jak dobrze aplikacja realizuje założenia projektowe, czy działa według specyfikacji. Aspekt strukturalny mówi o tym w jakim stopniu kod spełnia założenia niefunkcjonalne, które umożliwiają dodawanie funkcjonalności, czyli np. jak czytelny jest kod, albo jak łatwo dodawać do niego nowe elementy [30].

Badanie aspektu funkcjonalnego odbywa się poprzez przeprowadzenie testów oprogramowania i sprawdzenie jak daleko aplikacja odbiega od założeń projektowych. Testowanie może przybierać wiele form od manualnego testowania całości systemu (manualne testowanie integracyjne) aż do automatycznego testowania poszczególnych komponentów systemu np. klas (testowanie jednostkowe).

Badanie aspektu strukturalnego jest dużo trudniejsze, o ile każdy doświadczony programista potrafi stwierdzić „na oko” czy kod jest wysokiej jakości czy też nie, to zdefiniowanie formalnych kryteriów, które by to oceniały nie jest już proste.

## Wartość kodu wysokiej jakości

Potrzeba wytwarzania kodu o wysokiej jakości funkcjonalnej jest oczywista. Oprogramowanie powinno rozwiązywać rzeczywiste problemy klienta poprzez realizowanie uzgodnionych funkcjonalności, jeśli tego nie robi, klient po prostu za nie nie zapłaci.

Zupełnie inaczej wygląda sytuacja z jakością strukturalną. Kadra zarządzająca projektami jest skupiona na dostarczaniu konkretnych, widzialnych dla klienta funkcjonalności. Z oczywistych przyczyn chce te cele osiągnąć w jak najkrótszym czasie i przy użyciu jak najmniejszej ilości zasobów. Z tych powodów managerowie często niechętnie patrzą na kosztujące dodatkowy czas sposoby zapewniania strukturalnej jakości kodu.

### Ewolucja oprogramowania i prawa Lehmana

Niemożliwym jest rozwijanie rzeczywistego systemu informatycznego bez wykonywania czynności konserwacyjnych. Zauważył to profesor Meir M. Lehman i stworzył zestaw praw opisujących zachowanie się systemów o zmieniających się wymaganiach [31]:

1. **Ciągła zmiana** - System musi być ciągle ulepszany albo będzie działał coraz gorzej.
2. **Rosnące skomplikowanie** - Kiedy system ewoluuje jego poziom skomplikowania narasta chyba, że dodatkowa praca jest wykonana żeby temu zapobiec.
3. **Autoregulacja** – proces ewolucji systemu podlega autoregulacji.
4. **Zachowanie stabilności organizacyjnej** – średni poziom globalnej aktywności ewolucji sytemu jest niezmienny w czasie życia produktu
5. **Zachowanie znajomości** – w czasie ewolucji systemu wszyscy powiązani z nim programiści, sprzedawcy, użytkownicy i inni ludzie muszą zachować panowanie nad jego treścią i zachowaniem w celu osiągnięcia satysfakcjonującej ewolucji. Zbyt szybki wzrost uniemożliwia utrzymanie panowania nad systemem.
6. **Ciągły wzrost** – Zawartość funkcjonalna systemu musi być wraz z upływem czasu życia systemu ciągle zwiększana w celu zachowania satysfakcji użytkowników.
7. **Malejąca jakość** – Jakość systemu będzie spadała wraz z upływem jego czasu życia, chyba że zostaną podjęte rygorystyczne prace konserwacyjne i adaptacyjne
8. **System informacji zwrotnej** – procesy ewolucji systemu są systemami wielopoziomowymi, wielopętlowymi i składającymi się z wielu agentów dostarczających informację zwrotną.

Konsekwencją praw Lehmana jest konieczność ciągłego poprawiania i zmieniania systemu. Cykl życia oprogramowania nie kończy się po jego wytworzeniu. Różne badania dowodzą, że koszty utrzymania systemów stanowią około 75% - 90% kosztów posiadania [32], [33]. Z tego 20% to naprawianie niewykrytych błędów, 25% dostosowywanie oprogramowania do zmieniającego się środowiska, 5% ulepszenia wydajnościowe, a 50% dodawanie nowych funkcji [32].

### Dług technologiczny

Czasem z powodu złego zarządzania, albo nagłych zmian wymagań w projekcie powstaje presja czasu. Programiści są zmuszani do porzucenia istotnych zadań utrzymaniowych jak np. refactoring kodu, czy pisanie testów jednostkowych. Takie zachowanie ma na celu nadrobienie strat czasowych. Poprzez zaniechanie ważnych czynności konserwacyjnych zaczyna powstawać dług technologiczny.

Dług nie zawsze ma swoje źródło w błędach zarządzania, czasem wynika z niewiedzy i brakuj umiejętności programistów, którzy nie potrafią zadbać o wysoką jakość kodu.

Niezależnie od przyczyn powstawania długu technologicznego próby jego spłacenia zawsze są bolesne. Najłatwiej jest to zrobić na samym początku, gdy czasowe zadłużenie jest niewielkie. Wraz z postępami w projekcie czynności konserwacyjne pominięte we wczesnych jego fazach powodują coraz większe spowolnienie pracy. Po pewnym czasie utrzymanie projektu jest tak drogie, że zwyczajnie nie opłaca się tego dalej robić. W rezultacie projekt upada [34].

### Wpływ jakości strukturalnej na proces utrzymania oprogramowania

W poprzednich sekcjach pokazano jak istotny udział w cyklu życia oprogramowania ma utrzymanie kodu.

Około 50% kosztów utrzymania stanowią koszty zapoznawania się z kodem [33]. Spowodowane jest to tym, ze programista najpierw musi zrozumieć jak działa pokaźna część systemu, aby dokonać w nim jakichkolwiek zmian.

Z tego powodu najlepszym sposobem na obniżenie kosztów utrzymania jest poprawa strukturalnej jakości kodu, poprzez usunięcie martwych fragmentów (czyli takich, które nie są nigdy wywoływane), redundancji i poprawienie czytelności [33].

Jakość kodu jest istotna nie tylko w procesie utrzymania, ale też wytwarzania oprogramowania. Współcześnie wiele projektów programistycznych odbywa się w procesie iteracyjnym, z wykorzystaniem metodyk zwinnych, których założeniem jest to iż wymagania ewoluują w czasie wytwarzania oprogramowania. W celu zapewnienia wysokiej podatności kodu na zmiany, ten musi być stale utrzymywany w dobrej kondycji.

## Praktyczne sposoby zapewniania jakości kodu JavaScript

We wcześniejszej części tego rozdziału pokazano dlaczego jakość kodu jest istotna. Ten podrozdział skupia się na praktycznym podejściu do zagadnienia jakości i sposobach jej zapewniania w codziennej pracy programisty.

### Formatowanie

Składnia języka programowania stawia przed programistami zestaw ograniczeń, wciąż w dużej mierze pozostawiając jednak wolną rękę. Możliwe przez to jest zapisanie tej samej logiki na zupełnie różne sposoby. Formatowanie kodu nie ma najczęściej wpływu na rozumienie go przez kompilator bądź interpreter, jednak jest niezwykle ważne z punktu widzenia programisty.

Doskonałą ilustracją tego zagadnienia jest przedstawienie obok siebie zwykłego kodu i kodu zminifikowanego. Minifikacja to proces zmniejszania objętości kodu JavaScript w celu zmniejszenia ilości danych wysyłanych z serwera www do przeglądarki.

Rys. 3.5 zawiera prostą funkcję liczącą n-ty element ciągu Fibonacciego. Dokładnie to samo zadanie realizuje Rys. 3.6, który powstał na bazie oryginalnego kodu po użyciu online’owego narzędzia minifikującego <http://jscompress.com/>.

var fibo = function (n) {

    if (n <= 2) {

        return 1;

    }

    var previous = 1;

    var current = 1;

    var tmp = 0;

    for (var i = 2; i < n; i++) {

        tmp = current;

        current = current + previous;

        previous = tmp;

    }

    return current;

}

Rys. 3.1 - Obliczanie n-tego elementu ciągu Fibonacciego - wersja normalna

var fibo = function (a) { if (a <= 2) { return 1 } var b = 1; var c = 1; var d = 0; for (var e = 2; e < a; e++) { d = c; c = c + b; b = d } return c }

Rys. . - Obliczanie n-tego elementu ciągu Fibonacciego - wersja zminifikowana

Różnicę w czytelności widać gołym okiem - zminifikowana wersja kodu jest zupełnie niezrozumiała dla człowieka. Jest to oczywiście skrajny przykład, ale zaprezentowana funkcja jest też bardzo prosta.

Wiele współczesnych języków obiektowych oferuje ogólnie przyjęte standardy formatowania kodu. W przypadku Javy i C#-a są to standardy wprowadzone przez producentów i nie podlegające zbytniej dyskusji przez środowisko programistów.

W przypadku języka JavaScript jest inaczej – nie ma jednego ogólnie przyjętego stylu stosowanego przez wszystkich piszących w nim aplikacje. Powstało kilka różnych standardów kodowania np. „Code Conventions for the JavaScript Programming Language” Douglasa Crockforda, czy „Google JavaScript Style Guide”.

#### Wcięcia

Jako, że JavaScript podobnie jak we wszystkich innych językach rodziny C dobrą praktyką jest wcinanie zawartości bloków kodu. Znacznie poprawia to czytelność i pozwala wyraźnie odróżnić fragmenty aplikacji od siebie. Jest to technika tak dobrze przyjęta, że nie wymaga dodatkowego uzasadnienia.

#### Nawiasy klamrowe

Powszechnie stosuje się praktykę umieszczania otwierającego nawiasu klamrowego w tej samej linii co instrukcja, której on dotyczy. Taka technika zwana jest w żargonie programistycznym „egipskimi nawiasami” [35]. Podobne podejście jest stosowane w języku Java. W JavaScript nie jest to kwestia jedynie subiektywnych preferencji, taki styl wynika przede wszystkim z faktu, że język automatycznie umieszcza średniki na końcu każdej linii. Ten mechanizm może powodować błędy.

Taki przypadek zilustrowano poniżej - Rys. 3.7. Funkcja foo nie zwróci obiektu, tak jakby się to mogło wydawać, tylko undefined. Dzieje się tak, gdyż interpreter JavaScript automatycznie doda średnik zaraz za instrukcją return.

function foo()

{

    return

    {

        text: "aaa"

    }

}

console.log("%j", foo());

//undefined

Rys. 3.3 - Problem automatycznie dodawanych nawiasów

#### Format nazewnictwa

Zgodnie z konwencją przyjętą w bibliotece standardowej nazwy wszystkich zmiennych, a także pól, metod i funkcji zaczynają się z małej litery. Z wielkiej rozpoczynają się funkcje będące konstruktorami, informuje się w ten sposób programistę, że powinien ich używać jedynie ze słowem kluczowym new.

### Dobre praktyki czystego kodu

#### Funkcje

Funkcje są najmniejszą jednostką organizacji kodu i mają duży wpływ na jego czytelność.

Wielu programistów uważa, że jedynym zadaniem funkcji jest eliminacja redundancji poprzez umieszczenie w nich wielokrotnie używanych fragmentów kodu. W praktyce nie mniej ważne jest to, że funkcje pozwalają na dekompozycję problemu na mniejsze znajdujące się na różnych poziomach abstrakcji.

Funkcje powinny robić tylko jedną rzecz. Na przykład funkcja logIn() nie powinna dodatkowo wykonywać inicjalizacji mechanizmu logowania. Takie ukryte efekty uboczne są bardzo mylące dla programisty czytającego kod i wymagają od niego zapoznania się nie tylko z nazwą, ale także ciałem funkcji.

Dobre funkcje powinny przyjmować mało parametrów. W większości przypadków, jeśli funkcja przyjmuje więcej niż trzy parametry, oznacza to, że należy ją poprawić. Można to zrobić poprzez przebudowę albo zgrupowanie parametrów w logiczne obiekty jeśli to możliwe [36].

W przypadkach, w których dostarczamy do funkcji informacje konfiguracyjne, dopuszczalne jest użycie opcjonalnych parametrów nazwanych. Składnia JavaScript nie oferuje wspomnianego mechanizmu, typowym obejściem tego problemu jest użycie jednego parametru będącego obiektem. Ta technika jest intensywnie wykorzystywana w znanej bibliotece JavaScript – jQuery. Użycie nazwanych parametrów w jQuery ilustruje Rys. 3.8.

$.ajax({

    type: "POST",

    url: "/user",

    data: { name: "Kermit", surname: "the Forg", street: "Sesame Street" }

}).done(function (msg) {

    alert("Character created: " + msg);

});

Rys. . - Nazwane parametry

Każda funkcja powinna mieć nazwę, która w pełni opisuje jej zachowanie. Jeśli stworzenie odpowiedniej nazwy jest niemożliwe, oznacza to najprawdopodobniej, że funkcja jest zbyt skomplikowana. Nazwy powinny zaczynać się od czasowników, gdyż zadaniem funkcji jest wykonywanie czynności.

#### Komentarze

Przydatność komentarzy jest wysoce przeceniana. Bogato skomentowany kod nie musi być łatwy w zrozumieniu, podobnie kod pozbawiony komentarzy nie musi być trudny. Zamiast wyjaśniania działania fragmentu kodu komentarzem powinien on być po prostu napisany, tak żeby komentarza nie wymagał. Jeśli komentarz jest konieczny, oznacza to najczęściej, że nie jest wystarczająco czytelny [37].

Kod aplikacji nieustannie się zmienia w czasie prac nad nim. Największym problemem komentarzy jest to, że podczas zmian, często zostają one pominięte i błyskawicznie tracą swoją wartość informacyjną. Zamiast pomagać w zrozumieniu działania aplikacji jedynie to utrudniają.

Bardzo złym pomysłem jest wymuszanie na programistach komentowania kodu za pomocą narzędzi do analizy statycznej. Powoduje to powstawanie bezsensownych komentarzy, tworzonych tylko w celu spełnienia narzuconych wymagań. Rys. 3.9 pokazuje komentarz, który przekazuje dokładnie tą samą informację co nazwa metody. Jest to tylko szum, który programista z czasem uczy się ignorować, ignorując tym samym ważne informacje.

this.changeOwnerName = function (newName) {

    /// <param name="newName">Changes owner's name</param>

    this.ownerName = newName

};

Rys. . - Zbędny komentarz

Występowanie komentarzy nie zawsze jest rzeczą złą. Istnieje kilka przypadków, w których są one pożądane. Biblioteka, która będzie używana w zewnętrznych aplikacjach powinna zostać skomentowana, gdyż użytkownik zazwyczaj nie ma ochoty czytać kodu źródłowego przed użyciem. Innym przypadkiem, w którym są one wskazane jest wyjaśnianie dziwnych konstrukcji, które mogłyby wywołać konsternację innych programistów. W dobrym kodzie, o niskiej gęstości komentarzy taka informacja jest od razu widoczna i trudna do przeoczenia [37].

#### SOLID

SOLID to mnemonik wymyślony przez Roberta C. Martina, opisujący zestaw zasad dobrego programowania obiektowego. Nie są one sztywnymi prawami, lecz raczej wskazówkami dotyczącymi tworzenia dobrego kodu [38]. JavaScript nie jest klasycznym językiem obiektowym, ale jak zostanie pokazane niżej, również w nim znajdują zastosowanie zasady SOLID.

##### Zasada jednej odpowiedzialności (**S**ingle responsibility principle)

Zasada jednej odpowiedzialności (SRP) mówi, że klasa powinna być odpowiedzialna tylko za jedną rzecz. Wszystko co się w niej znajduje, wszystkie pola i metody powinny służyć jednemu, konkretnemu celowi. Nie oznacza to, że klasa może mieć tylko jedną publiczną metodę, raczej to że wszystkie powinny dotyczyć tego samego.

Robert C. Martin definiuje SRP następująco: „klasa nigdy nie powinna mieć więcej niż jednego powodu do zmiany” [39]. Ta definicja pomaga w sprawdzeniu czy kod narusza zasadę jednej odpowiedzialności.

Rys. 3.11 ilustruje naruszenie zasady SRP przez klasę TransactionRepository. Można to udowodnić wykazując, że ma ona więcej niż jeden powód do zmiany. Pierwszym takim powodem jest zmiana sposobu odwoływania się do tabeli w bazie danych np. zmiana nazwy tabeli. Drugim powodem konieczność zmienienia formatu, w jakim wyświetlany jest raport.

var TransactionRepository = function () {

    this.getAllTransactions = function () {

        return db.query("SLECT id, date, invoice, paid, price " +

                        "FROM transaction");

    }

    this.saveTransaction = function (transaction) {

        db.insert("INSERT INTO transaction VALUES ({0})", transaction);

    }

    this.printTransactionsReport = function () {

        var transactions = this.getAllTransactions();

        console.log("------------------------------------");

        for (var i = 0; i < transactions.length; i++) {

            var transaction = transactions[i];

            console.log("Invoice: %s", transaction.invoice);

            console.log("  Id: %s", transaction.id);

            console.log("  Price: %s", transaction.price);

            console.log("  Paid: %s", transaction.paid);

            console.log("------------------------------------");

        }

    }

}

Rys. . - Kod naruszający zasadę jednej odpowiedzialności

Klasa TransactionRepository powinna zajmować się jedynie operacjami związanymi z bazą danych, które są odpowiedzialnością różną od wyświetlania raportu. Rozwiązaniem tego problemu jest ekstrakcja dodatkowej klasy ReportPrinter, której zadaniem będzie jedynie wyświetlanie raportów. Zaprezentowane to zostało na Rys. 3.12.

var TransactionRepository = function () {

    this.getAllTransactions = function () {

        return db.query("SLECT id, date, invoice, paid, price " +

                        "FROM transaction");

    }

    this.saveTransaction = function (transaction) {

        db.insert("INSERT INTO transaction VALUES ({0})", transaction);

    }

}

var ReportPrinter = function (transactionRepository) {

    this.printTransactionsReport = function () {

        var transactions = transactionRepository.getAllTransactions();

        console.log("------------------------------------");

        for (var i = 0; i < transactions.length; i++) {

            var transaction = transactions[i];

            console.log("Invoice: %s", transaction.invoice);

            console.log("  Id: %s", transaction.id);

            console.log("  Price: %s", transaction.price);

            console.log("  Paid: %s", transaction.paid);

            console.log("------------------------------------");

        }

    }

}

Rys. 3.7 - SRP kod po refactoringu

##### Zasada otwarte/zamknięte (**O**pen/closed principle)

Zasada otwarte/zamknięte (OCP) mówi, że „elementy systemu powinny być otwarte na rozszerzanie ale zamknięte na modyfikacje” [40]. Oznacza to, że zmiana zachowania takiego elementu np. klasy powinna być możliwa bez modyfikacji jego kodu źródłowego.

Naruszenie tej zasady lustruje Rys. 3.13. Zadaniem klasy PizzaOrder jest obliczenie całkowitego kosztu zamówienia. Pizzeria oferuje promocję – w przypadku zamówienia jedynie pikantnych pizz klient dostaje zniżkę 10% na całość zamówienia.

var PizzaOrder = function () {

    var that = this;

    this.pizzas = [];

    this.calculatePrice = function () {

        var priceSum = this.pizzas.reduce(function (prev, current) {

            return prev + current.price \* current.count;

        }, 0);

        var totalPrice = priceSum;

        if (containsOnlySpicyPizzas()) {

            totalPrice \*= 0.90;

        }

        return totalPrice;

    }

    function containsOnlySpicyPizzas() {

        return that.pizzas.every(function (pizza) {

            return pizza.ingredients.some(function (ingredient) {

                return ingredient == "chili";

            });

        });

    }

}

Rys. 3.8 - PizzaOrder narusza OCP

Problem pojawia się w wypadku konieczności dodania kolejnej promocji. Żeby to uczynić trzeba zmienić klasę, dopisując kod sprawdzający czy warunki, w której klientowi przysługuje zniżka zostały spełnione. Dokładanie kodu obsługującego kolejne promocje do PizzaOrder będzie skutkowało znacznie rosnącym komplikowaniem się klasy i utrudnionym testowaniem jednostkowym.

W celu rozwiązania powstałego problemu można przeprowadzić refactoring i zmienić klasę na taką jaką prezentuje Rys. 3.14. Teraz żeby dodać kolejną zniżkę wystarczy napisać funkcję, która ją wylicza (dwie takie funkcje zawiera obiekt someDiscounts), a następnie zarejestrować zniżkę w instancji klasy PizzaOrder, poprzez dodanie jej do odpowiedniej tablicy.

var someDiscounts = {

    containsOnlySpicyPizzas: function (pizzas) {

        var discount = 0.9;

        return pizzas.every(function (pizza) {

            return pizza.ingredients.some(function (ingredient) {

                return ingredient == "chili";

            });

        }) ? discount : 1.0;

    },

    containsSomeLargePizzas: function(pizzas){

        var discount = 0.85;

        return pizzas.some(function (pizza) {

            return pizza.size == sizes.large;

        }) ? discount : 1.0;

    }

}

...

var PizzaOrder = function () {

    var that = this;

    this.pizzas = [];

    this.discounts = [];

    this.calculatePrice = function () {

        var priceSum = this.pizzas.reduce(function (prev, current) {

            return prev + current.price \* current.count;

        }, 0);

        var totalPrice = priceSum;

        this.discounts.forEach(function(discount){

            totalPrice \*= discount(that.pizzas);

        });

        return totalPrice;

    }

}

var pizzaOrder = new PizzaOrder();

...  
pizzaOrder.discounts = [someDiscounts.containsOnlySpicyPizzas,

    someDiscounts.containsSomeLargePizzas];

pizzaOrder.calculatePrice()

Rys. 3.9 - PizzaOrder po refactoringu

Pamiętając o tym, że zasady SOLID są bardziej wskazówkami niż prawami, należy rozważnie używać zasady OCP. Umieszczanie w klasie wszystkich możliwych punktów rozszerzeń jest błędem. Zamiast tego powinno się na początku stawiać na jak najprostszy projekt, dopiero później komplikując go, jeśli zajdzie potrzeba.

##### Zasada podstawienia Liskov (**L**istokv substitution principle)

Sformowana w 1987 roku przez Barbarę Liskov zasada podstawienia (LSP) brzmi:  
„Jeśli dla każdego obiektu o1 typu S istnieje obiekt o2 typu T taki, że dla wszystkich programów P zdefiniowanych dla T, zachowanie P zachowa swoje oczekiwane właściwości jeśli o1 zostanie zastąpione przez o2, wtedy S jest zamiennikiem T.” [41].

Z pozoru ta zasada wydaje się nie mieć sensu w językach cechujących się duck-typingiem, gdyż parametry funkcji i zmienne mogą przyjmować obiekty dowolnego typu. Jak jednak zostanie pokazane poniżej jest to nieprawda.

Rys. 3.15 pokazuje problem zasady LSP. MenuPrinter, korzysta z klasy RestPizzaRepository, która ma jedną metodę getAllPizzas() zwracającą listę wszystkich istniejących pizz. Pobieranie tej listy odbywa się za pośrednictwem protokołu HTTP, przy użyciu REST-owego API.

var RestPizzaRepository = function () {

    this.getAllPizzas = function () {

        return restClient.get("http://localhost:9090/pizza");

    }

};

var MenuPrinter = function (pizzaRepository) {

    this.printMenu = function () {

        var allPizzas = pizzaRepository.getAllPizzas();

        for (var i = 0; i < allPizzas.length; i++) {

            var pizza = allPizzas[i];

            var ingredientsText = pizza.ingredients.join(", ");

            console.log("%s .............. %d", pizza.name, pizza.price);

            console.log(ingredientsText);

            console.log();

        }

    }

}

...

var pizzaRepository = new RestPizzaRepository();

var menuPrinter = new MenuPrinter(pizzaRepository);

menuPrinter.printMenu();

Rys. 3.10 - Ilustracja problemu LSP – RestPizzaRepository

Wyobraźmy sobie teraz, że założenia system się zmieniły i dane mają być pobierane nie przy użyciu HTTP, ale bezpośrednio z bazy danych. W tym celu tworzona jest nowa klasa DatabasePizzaRepository (Rys. 3.16), która oferuje dokładnie ten sam interfejs czyli metodę getAllPizzas() zwracającą listę wszystkich pizz. Struktura zwracanych danych jest taka sama. Różnica występuje tylko w metodzie initializeConnection() – przed pobraniem danych trzeba zainicjalizować połączenie do bazy danych. W rezultacie zwykła podmiana RestPizzaRepository na DatabasePizzaRepository spowoduje błąd! Nie można pobrać danych bez otwarcia połączenia.

var DatabasePizzaRepository = function () {

    this.initializeConnection = function () {

        db.open("localhost:9991;DBNAME=pizza\_db");

    };

    this.getAllPizzas = function () {

        return db.query("SLECT \* FROM pizzas");

    }

};

...

var pizzaRepository = new DatabasePizzaRepository();

var menuPrinter = new MenuPrinter(pizzaRepository);

menuPrinter.printMenu();

Rys. 3.11 - Ilustracja problemu LSP – DatabasePizzaRepository

Rozwiązaniem przedstawionego problem mogłaby np. być leniwa inicjalizacja. Metoda getAllPizzas() mogłaby inicjalizować połączenie sama, jeśli nie zostałoby to zrobione wcześniej.

Zasada podstawienia Liskov definiuje silniejszą zależność między obiektami niż dziedziczenie czy implementacja interfejsu. Informacja o tym, że klasa A i klasa B implementują ten sam interfejs I, mówi jedynie o strukturze obiektów klasy A i B. Informacja o tym, że klasa A jest zamiennikiem klasy B, mówi o tym co znacznie ważniejsze czyli o ich zachowaniu.

Rozszerzmy teraz definicję LSP na obiekty, które w JavaScript mogą występować nawet bez obecności klas, nawet rozumianych tak jak zdefiniowano je w rozdziale 2.

„Jeśli dla każdego obiektu o1 istnieje obiekt o2 taki, że dla wszystkich programów P zdefiniowanych dla o1, zachowanie P zachowa swoje oczekiwane właściwości jeśli o1 zostanie zastąpione przez o2, wtedy o1 jest zamiennikiem o2.”

##### Zasada segregacji interfejsów (**I**nterface segregation principle)

Zasada segregacji interfejsów (ISP) mówi, że klienci nie powinni być zmuszeni do polegania na interfejsach, których nie używają [42]. Jest ona dość problematyczna w języku JavaScript, gdyż ten nie posiada mechanizmu interfejsów, pozwalających na zawężenie zestawu dostępnych dla klienta (użytkownika klasy) metod. Z tego powodu najpierw zostanie omówiona na przykładzie w języku C#.

Rys. 3.17 zawiera dwie klasy Manager i Kitchen, które operują na obiektach implementujących interfejs IWorker. W praktyce jednak używają zupełnie innych metod tego interfejsu. Może to powodować problem, w sytuacji gdy mamy do czynienia z klasą, która tak naprawdę nie powinna dostarczać obu funkcjonalności np. RoboWorker pracuje ale nie je.

interface IWorker

{

    void Work();

    void Eat();

}

class HumanWorker : IWorker

{

    public void Work()

    {

        Console.WriteLine("Working like a human");

    }

    public void Eat()

    {

        Console.WriteLine("Eating like a human");

    }

}

class RoboWorker : IWorker

{

    public void Work()

    {

        Console.WriteLine("Working like a robot");

    }

    public void Eat()

    {

        throw new NotSupportedException("I don't eat!");

    }

}

class Manager

{

    public void Manage(IWorker worker)

    {

        worker.Work();

    }

}

class Kitchen

{

    public void Feed(IWorker worker)

    {

        worker.Eat();

    }

}

Rys. . - Naruszenie zasady ISP w języku C#

interface IWorkable

{

    void Work();

}

interface IEatable

{

    void Eat();

}

...

class RoboWorker : IWorkable

{

    public void Work()

    {

        Console.WriteLine("Working like a robot");

    }

}

class Manager

{

    public void Manage(IWorkable worker)

    {

        worker.Work();

    }

}

class Kitchen

{

    public void Feed(IEatable worker)

    {

        worker.Eat();

    }

}

Rys. . - Mniejsza ziarnistość interfejsów pozwala zachować ISP

Rozwiązanie tego problemu prezentuje Rys. 3.18. IWorker został rozdzielony na dwa mniejsze interfejsy IWorkable i IEatable. Teraz klasa Manager zajmuje się jedynie obiektami, które pracują, a Kitchen tylko tymi, które jedzą.

W JavaScript podobny problem nie występuje, bo RoboWorker nigdy nie implementowałby szerokiego interfejsu IWorker, gdyż takowe nie istnieją w tym języku. Z drugiej strony zarówno klasa Kitchen jak i Manager miałaby dostęp do wszystkich metod obiektu klasy HumanWorker. W większości przypadków podejmowanie działań przeciwko temu nie ma sensu, jednak czasem, kiedy udostępniamy jakieś obiekty na zewnątrz biblioteki, dobrym pomysłem, może być fizyczne zawężenie ich interfejsu.

To zagadnienie ilustruje Rys. 3.19. Najpierw zdefiniowana jest klasa z dwoma metodami work i eat, później stworzona jest jej instancja. Następnie w celu ograniczenia interfejsu jedynie do metody eat, tworzona jest fasada eatableAdaptingFacade. Może być ona przekazana do metod feed obiektu kitchen, ale już nie zadziała w metodzie manage obiektu manager.

var HumanWorker = function () {

    this.work = function (task) {

        console.log("Working on" + task);

    };

    this.eat = function (food) {

        console.log("Eating " + food);

    };

}

var humanWorker = new HumanWorker();

var eatableAdaptingFacade = {

    eat: function (food) {

        humanWorker.eat(food);

    }

}

var manager = {

    manage : function(worker){

        worker.work("cleaning");

    }

}

var kitchen = {

    feed: function (worker) {

        worker.eat("pizza");

    }

}

kitchen.feed(eatableAdaptingFacade);

//próba wywołania zakomentowanej niżej linii

//spowoduje błąd!

//manager.manage(eatableAdaptingFacade);

Rys. . - Ograniczenie interfejsu za pomocą fasady adaptującej

W przypadku częstego tworzenia obiektów o ograniczonym interfejsie przydatne może się okazać stworzenie funkcji, która potrafi „okroić” dostarczony obiekt ze zbędnych metod. Przykład takiej funkcji widoczny jest na Rys. 3.20. Funkcja wrapInFacade() przyjmuje dwa parametry, pierwszym jest obiekt, którego interfejs będzie ograniczany, drugim lista metod, które zostaną upublicznione. Pętla for … in iteruje po wszystkich kluczach dostarczonego obiektu i udostępnia te, które spełniają dwa warunki: są funkcjami i znajdują się na podanej liście.

Udostępnianie polega na dodaniu do fasady metody o takiej samej nazwie jak oryginalna i wywołanie tej z takim samym zestawem parametrów.

function wrapInFacade(object, methodsToExpose) {

    var facade = {};

    function includeInFacade(fieldName) {

        return  typeof object[fieldName] == "function" &&

                methodsToExpose.indexOf(fieldName) != -1;

    }

    for (fieldName in object) {

        if (includeInFacade(fieldName)) {

                facade[fieldName] = function (includeInFacade) {

                    object[fieldName].apply(object, arguments);

            };

        }

    }

    return facade;

}

...

var humanWorker = new HumanWorker();

var eatableAdaptingFacade = wrapInFacade(humanWorker, ["eat"]);

Rys. . - Zawężanie interfejsu obiektu, za pomocą autogenerowanej fasady

##### Zasada odwrócenia zależności (**D**ependency inversion principle)

Zasada odwrócenia odpowiedzialności (DIP) mówi, że:

* Moduł wysokiego poziomu nie powinien polegać na niskopoziomowych modułach. Oba powinny polegać na abstrakcjach.
* Abstrakcje nie powinny polegać na szczegółach. Szczegóły powinny polegać na abstrakcjach.

var encryption = {

    decrypt : function(srcFileName, dstFileName, password){

        fileContens = file.read(srcFileName);

        uncryptedContents = performDecryption(fileContens, password);

        file.write(decryptedContents);

    }

}

Rys. 3.16 - Obiekt encryption łamie zasadę DIP

Naruszenie zasady ilustruje Rys. 3.21. Funkcja decrypt() przyjmuje nazwę pliku, z którego sama odczytuje zawartość do odszyfrowania. Następnie również sama zapisuje odszyfrowaną zawartość do innego pliku.

Taka konstrukcja kodu ogranicza możliwości ponownego użycia. Obiekt encryption może być wykorzystany tylko w przypadku gdy źródłem danych i miejscem składowania rezultatu działania jest plik. Metoda decrypt() polega na module niskiego poziomu (file) zamiast na abstrakcji.

Można to poprawić poprzez refactoring kodu, tak aby polegał on na abstrakcyjnej idei strumienia. Ulepszoną wersję obiektu encryption prezentuje Rys. 3.22. Dzięki takiej konstrukcji metody decrypt() możliwe jest jej zastosowanie zarówno w wypadku plików, jak i np. komunikacji przez sieć.

var encryption = {

    decrypt: function (srcStream, dstStream, password) {

        fileContens = srcStream.read();

        uncryptedContents = performDecryption(fileContens, password);

        dstStream.write(decryptedContents);

    }

}

Rys. 3.17 - Obiekt encryption po refactoringu

### Testy jednostkowe

Testowanie jest bardzo ważnym elementem procesu wytwarzania oprogramowania. Wprawdzie to, że system poprawnie przeszedł testy nie jest gwarancją, iż jest on wolny od błędów, ale przynajmniej pozwala wykryć wiele błędów.

Najwyższym z poziomów testów, którym może być poddany system są systemowe testy integracyjne. Sprawdzają one czy system działa poprawnie w integracji z innymi systemami [43]. Ten rodzaj testów jest wykonywany zazwyczaj przez specjalnie wydzielony zespół testerów. Przeciwieństwem systemowych testów integracyjnych są testy jednostkowe.

Zadaniem testów jednostkowych upewnienie się, że małe elementy kodu działają poprawnie. Takimi elementami mogą być np. klasy, obiekty albo poszczególne funkcje [44].

#### Testy jednostkowe w JavaScript

Przy obecnych rozmiarach projektów, testy jednostkowe powinny być traktowane jako konieczność. Są one szczególnie istotne w językach dynamicznych, gdzie nawet proste błędy, takie jak literówki wychwytywane są dopiero w czasie działania programu.

W świecie JavaScript istnieje wiele bibliotek testowania jednostkowego. Można je podzielić na dwie grupy:

* Wzorowane na Javowym JUnit
  + QUnit
  + YUI Test
  + JSUnit
* Wzorowane na RSpecu z Rubiego
  + Jasmine
  + Mocha

Ostatnio coraz większą popularność zdobywa druga grupa. Testy opisuje się w nich za pomocą języka domenowego, dzięki czemu powstała w ten sposób specyfikacja jest możliwa do zrozumienia nawet dla osoby nie będącej programistą.

#### Test-driven development

Istnieją dwie przeciwstawne szkoły tworzenia testów jednostkowych. Pierwsza z nich zakłada pisanie testów po napisaniu funkcjonującego kodu. Zadaniem takich testów jest przede wszystkim zabezpieczyć już istniejące funkcjonalności przed przypadkowym zepsuciem.

Druga, zwana test-driven development (TDD) zakłada podejście odwrotne – testy powinny być pisane najpierw, dopiero później kod który ma je spełnić.

Pierwsze podejście jest obciążone poważnym problemem – jeśli test zostanie napisany niepoprawnie, czyli tak, że jego warunki są zawsze spełniane, programista nigdy nie dowie się o potencjalnym błędzie w kodzie, który stworzył. Dlatego właściwe jest podejście drugie. Czasem, po dodaniu dodatkowego kroku zwane jest ono red-green-refactor i w całości prezentuje się następująco [45]:

1. **Red** – programista tworzy test po czym go uruchamia. Test nie powinien się powieść.
2. **Green** – programista implementuje funkcjonalność niezbędną do spełnienia testu, po czym uruchamia test ponownie. Test powinien się powieść.
3. **Refactor** – programista przeprowadza refactoring napisanego kodu w celu poprawy jego jakości. Dzięki temu, że kod jest pokryty testem, programista będzie wiedział czy podczas refactoringu nie zepsuł działającej funkcjonalności.

#### Główne zalety TDD

Podejście test-driven development charakteryzuje się szeregiem zalet pozytywnie wpływających na jakość kodu. Są nimi:

* **Automatyczne testy regresyjne** – zestaw dobrze napisanych testów jednostkowych pozwala natychmiast wykryć wprowadzenie błędu do istniejącego kodu aplikacji.
* **Wspieranie dobrych praktyk SOLID** – testowanie źle napisanego kodu jest trudne. Jeśli programista najpierw tworzy testy, a później implementację to musi on stworzyć kod łatwo testowalny. Mając na uwadze łatwość testowania, programista stworzy kod wysokiej jakości podążający za zestawem dobrych praktyk SOLID. Na przykład będzie automatycznie podążał za zasadą SRP, gdyż klasy mające wiele odpowiedzialności są skomplikowane, co rodzi potrzebę tworzenia trudnych testów.
* **Mniejsza potrzeba użycia debuggera** – dzięki drobnoziarnistym testom jednostkowym potrzeba używania debuggera zostaje ograniczona. Programista często może znaleźć błąd jedynie dzięki informacji, że dany test się nie powiódł.
* **Szybsze debugowanie** – debugowanie testów jednostkowych jest szybsze niż debugowanie całego systemu. Często używając debuggera na całym systemie programista musi najpierw zaczekać na jego inicjalizację, a następnie doprowadzić do odpowiedniego stanu. Może to trwać dużo czasu. Testy jednostkowe oferują możliwość debugowania komponentów systemu w izolacji i w ustalonych z góry sytuacjach.
* **Dokumentacja kodu** – testy dokumentują kod informując o tym jak zachowują się dane komponenty i demonstrując przykłady ich użycia.
* **Ukierunkowanie pracy** – testy pomagają ukierunkować pracę. Programista cały czas wie co ma zrobić, dzięki czemu skupia się tylko na małym elemencie kodu [46].

#### Przykład podejścia TDD do tworzenia kodu w JavaScript

Poniżej przedstawiono kilka pierwszych kroków procesu implementacji prostej klasy za pomocą podejścia test-driven development.

Pierwszym krokiem będzie implementacja testu, który wymaga aby samochód zaraz po utworzeniu miał wyłączony silnik.

**Kod testów:**

describe("Car", function () {

    describe("when car is created", function () {

        var car;

        beforeEach(function () {

            car = new Car();

        });

        it("should have engine turned off", function () {

            expect(car.isEngineStarted).to.be.false;

        });

    });

});

**Kod aplikacji:**

var Car = function () {

}

Rys. . - Krok pierwszy - klasa Car nie spełnia testów

Test ten nie powiedzie się:

1) Car when car is created should have engine turned off:

expected undefined to be false

Daje to gwarancję, iż nie jest on zawsze spełniany, a więc najprawdopodobniej jest przydatny. Kolejnym krokiem będzie napisanie odpowiedniego kodu, który doprowadzi do spełnienia testu:

var Car = function () {

    this.isEngineStarted = false;

}

Rys. . - Zaimplementowanie brakującej funkcjonalności w klasie Car

✔ 1 test complete (1ms)

Po serii podobnych kroków powstanie zestaw testów i pełna implementacja klasy Car.

**Kod testów:**

describe("Car", function () {

    describe("when car is created", function () {

        var car;

        beforeEach(function () {

            car = new Car();

        });

        it("should have engine turned off", function () {

            expect(car.isEngineStarted).to.be.false;

        });

        it("should have 0 speed", function(){

            expect(car.speed).to.be.equal(0);

        });

        it("should have 50 acceleration", function(){

            expect(car.acceleration).to.be.equal(50);

        });

    });

    describe("when engine is not started", function(){

        var car;

        beforeEach(function () {

            car = new Car();

            car.isEngineStarted = false;

        });

        it("calling start engine turns engine on" ,function(){

           car.startEngine();

           expect(car.isEngineStarted).to.be.true;

        });

        it("cannot accelerate" ,function(){

            car.accelerate();

            expect(car.speed).to.be.equal(0);

        });

    });

    describe("when engine is started", function(){

        var car;

        beforeEach(function () {

            car = new Car();

            car.isEngineStarted = true;

        });

        it("can accelerate", function(){

            car.accelerate();

            expect(car.speed).to.be.equal(50);

        });

    });

});

**Kod aplikacji:**

var Car = function () {

    this.isEngineStarted = false;

    this.speed = 0;

    this.acceleration = 50;

    this.startEngine = function () {

        this.isEngineStarted = true;

    }

    this.accelerate = function () {

        if (this.isEngineStarted) {

            this.speed += this.acceleration;

        }

    }

}

Rys. . - Ostatni krok implementacji klasy Car

Zbiór testów tworzy specyfikację klasy, która całkowicie opisuje jej zachowanie. Specyfikacja jest zapisana językiem możliwym do zrozumienia także dla ludzi nie będących programistami. Na Rys. 3.26 widać czytelną dla nie-programisty specyfikację wygenerowaną na podstawie testów.

Car

when car is created

✓ should have engine turned off

✓ should have 0 speed

✓ should have 50 acceleration

when engine is not started

✓ calling start engine turns engine on

✓ cannot accelerate

when engine is started

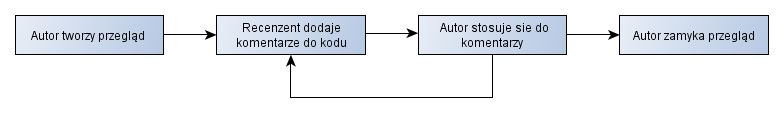
✓ can accelerate

Rys. . - Specyfikacja klasy Car

### Przeglądy kodu

Przeglądy kodu to praktyka, która pozwala zachować wysoką jakość strukturalną tworzonego oprogramowania. Polegają one na analizie fragmentów kodu źródłowego przez człowieka, który nie jest ich autorem w celu wychwycenia potencjalnych błędów [47].

Proces przeglądu kodu z udziałem jednego recenzenta może wyglądać następująco [48]:



Rys. . – Proces przeglądu kodu z udziałem jednego recenzenta

## Metryki jakości kodu

W celu mierzenia strukturalnej jakości kodu obiektowego powstały różne metryki. W tym podrozdziale zostaną przytoczone najważniejsze z nich.

### Złożoność cyklomatyczna

Złożoność cyklomatyczna (CC – cyclomatic complexity) to miara ilości niezależnych ścieżek wykonania funkcji [49]. Wylicza się ją na podstawie grafu przepływu sterowania. Dla przykładu funkcja printAge() pokazana na Rys. 3.1 ma złożoność równą dwa, gdyż są tylko dwie możliwe drogi przebiegu sterowania przez funkcję: jedna w przypadku wieku większego lub równego osiemnaście, druga w przypadku wieku mniejszego niż 18.

function printAge(age) {

    if (age >= 18) {

        console.log("Adult: " + age);

    } else {

        console.log("Underage: " + age);

    }

}

Rys. 3.23 - Złożoność cyklomatyczna

Wartość złożoności cyklomatycznej ma istotny związek z ilością błędów w kodzie. Badania przeprowadzone przez grupę Enerjy potwierdzają występowanie tej zależności. Pliki, które miały wartość CC równą 11 miały prawdopodobieństwo zawierania błędów równe 28%. Te o wartości CC 38 miały już prawdopodobieństwo występowania błędów 50%, natomiast najbardziej złożone o wartości ZC 74 prawie zawsze zawierały błędy (prawdopodobieństwo 98%) [50].

Uzyskane wyniki są zgodne z intuicją – program o wielu możliwych ścieżkach wykonania trudniej analizować i łatwiej popełnić w nim błędy.

### Kohezja

W programowaniu obiektowym kohezja określa wewnętrzną spójność elementów klasy. Cecha ta określa to jak metody działają na danych. W przypadku klasy o idealnie wysokiej kohezji każda metoda obiektu operuje na każdej danej. Nie występuje sytuacja, w której metody odwołują się jedynie do podzbioru pól [51].

Ilustracją zagadnienia kohezji w JavaScript są dwa poniższe kody źródłowe. W przypadku Rys. 3.2, metody withdraw() i deposit() operuję jedynie na polu ballance, natomiast metoda changeOwnerName() jedynie na polu ownerName. W związku z tym klasa BankAccount cechuje się relatywnie niską kohezją.

var BankAccount = function () {

    this.ballance = 1000;

    this.ownerName = "Kermit the Frog";

    this.withdraw = function (ammount) {

        this.ballance -= ammount;

    };

    this.deposit = function (ammount) {

        this.ballance += ammount;

    };

    this.changeOwnerName = function (newName) {

        this.ownerName = newName

    };

}

Rys. . - Obiekt o relatywnie niskiej kohezji

Obiekty o niskiej kohezji można zazwyczaj rozdzielić na kilka mniejszych. Przykładem jest Rys. 3.3, na którym z klasy BankAccount wyizolowano AccountOwner.

var BankAccount = function () {

    this.ballance = 1000;

    this.withdraw = function (ammount) {

        this.ballance -= ammount;

    };

    this.deposit = function (ammount) {

        this.ballance += ammount;

    };

}

var AccountOwner = function () {

    this.ownerName = "Kermit the Frog";

    this.changeOwnerName = function (newName) {

        this.ownerName = newName

    };

}

Rys. . - Obiekty o wysokiej kohezji

Podzielenie obiektu o idealnie wysokiej kohezji na mniejsze powinno być niemożliwe.

Konkretnymi próbującymi mierzyć kohezję są na przykład:

* **Lack of Cohesion of Methods 1** (LCOM1), która “mierzy brak odwołań do wspólnych atrybutów klasy i przyjmuje wartość równą różnicy liczby metod odwołujących się do wspólnego atrybutu klasy oraz liczby metod o rozłącznych zbiorach wykorzystywanych atrybutów” [52].
* **Lack of Cohesion of Methods 4** (LCOM4) - mierzy ona ilość połączonych komponentów klasy. Połączony komponent to zestaw powiązanych metod, czyli takich które odwołują się do tego samego pola, bądź wywołują siebie [53].

### Zależność

Dwie klasy są zależne gdy metody zadeklarowane w jednej korzystają pól lub metod drugiej. Przykład zależności w JavaScript został przedstawiony niżej na Rys. 3.4.

var Logger = function () {

    this.write = function (message) {

        console.log(new Date() + " " + message);

    }

}

var AccountOwner = function () {

    var \_logger = new Logger();

    this.ownerName = "Kermit the Frog";

    this.changeOwnerName = function (newName) {

        this.ownerName = newName;

        \_logger.write("Changed name to " + newName);

    };

}

Rys. . - Powiązanie między "klasami" Logger i Account Owner

Klasy AccountOwner i Logger są powiązane, gdyż AccountOwner korzysta z metody write instancji klasy Logger.

Metryką, która mierzy zależności jest Coupling between Object Classes (CBO). Wysoka wartość CBO jest niepożądana, gdyż klasy mające wiele zależności są trudne w ponownym użyciu, ponadto są one silnie podatne na zmiany mające w miejsce w innych częściach aplikacji. Za zbyt wysoką uznawana jest wartość metryki powyżej 14 [54].

### Poziom pokrycia

Poziom pokrycia to metryka związana z testowaniem oprogramowania. Określa ona w jakim stopniu kod źródłowy programu został przetestowany [55].

Istnieje kilka odmian omawianej metryki:

* **Pokrycie instrukcji** – „Odsetek instrukcji wykonywalnych, które zostały przetestowane przez zestaw testowy.” [56]
* **Pokrycie rozgałęzień** – „Odsetek rozgałęzień sprawdzony przez zestaw przypadków testowych.” [56]
* **Pokrycie ścieżek** - „Odsetek ścieżek w module wykonanych przez zestaw testowy.” [56]
* **Pokrycie warunków** – „Odsetek pokrycia warunków, jaki został sprawdzony przez zestaw testów. 100% pokrycia warunków wymaga, aby każdy pojedynczy warunek każdego wyrażenia decyzyjnego został przetestowany zarówno dla wartości Prawda jak i Fałsz.” [56]

Wysoki poziom pokrycia sam w sobie nie gwarantuje kodu wysokiej jakości, natomiast jego niski poziom jest sygnałem, iż prawdopodobnie testowana aplikacja zawiera ukryte błędy [57]. Uzyskanie wysokiej wartości omawianej metryki nie powinno być celem samym w sobie, a raczej efektem ubocznym kodu napisanego z zachowaniem TDD [58].

100% pokrycie kodu nie jest zazwyczaj efektywne kosztowo w dużych projektach. W niektórych zastosowaniach jest jednak ono wymagane. Przykładem jest branża lotnicza. Według standardu DO-178B kod, w którym błędy mogą prowadzić do uszczerbku na zdrowiu pasażerów musi spełniać 100% pokrycie instrukcji. Rozsądnym poziomem pokrycia dla większości projektów jest wartość około 70-80% [59].

## Dziedzina problemu

Dziedziną problemu, do którego należy implementowana aplikacja jest zarządzanie zadaniami. Przez lata powstało wiele metodyk wspierających zarządzanie zadaniami zarówno na poziomie indywidualnych osób jak i całych projektów.

### Metodyka GTD

Jedną z nich jest opracowana przez Davida Allena – Getting things done (GTD), która została opisana w książce pod tym samym tytułem. Za chwytliwą nazwą kryje się kilka istotnych spostrzeżeń z dziedziny psychologii oraz prostych algorytmów, pozwalających na zwiększenie produktywności indywidualnych osób.

David Allen zauważył, że przyczyną słabej produktywności są problemy ze skupieniem uwagi. Wynikają one z tego, że ludziom w czasie pracy nad ważnym zadaniem przypominają się inne rzeczy, które mają do zrobienia. Problemem nie jest przeładowanie informacją i ilość spraw, które człowiek ma do załatwienia, lecz raczej to, że pamiętaniem o nich zajmuje się umysł.

Według twórcy metodyki pracą do zrobienia jest dowolna akcja, którą należy wykonać. Dla umysłu nie ma rozróżnienia między czynnościami czysto zawodowymi, a tymi związanymi z życiem osobistym.

Zdolność do bycia efektywnym jest wprost proporcjonalna do zdolności koncentracji. Zdolność koncentracji jest natomiast wprost proporcjonalna do zdolności eliminacji rozpraszających bodźców. Takimi bodźcami są właśnie różnie sprawy do załatwienia zmagazynowane w umyśle.

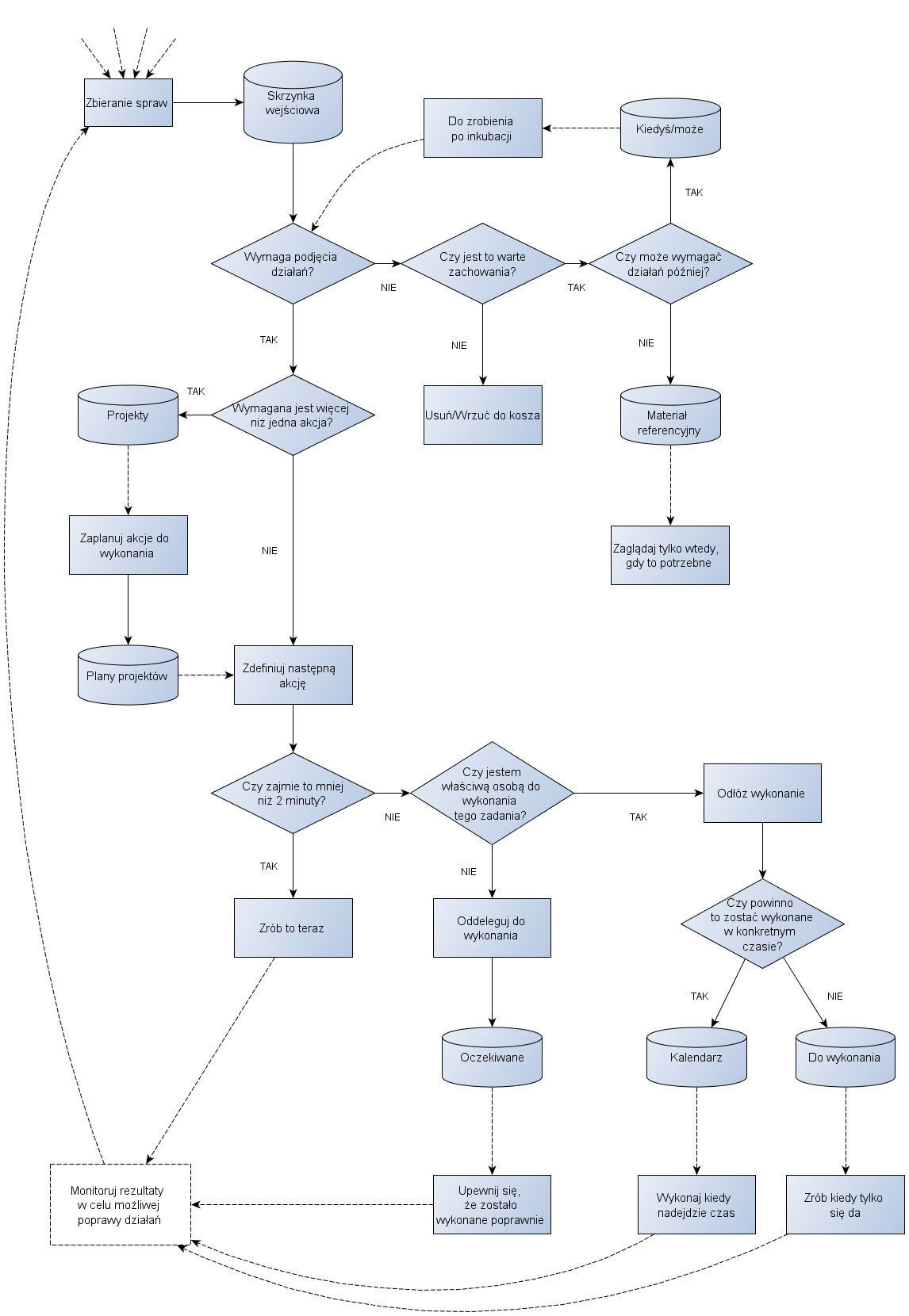
David Allen wskazuje na to, że umysł jest wyjątkowo kiepskim miejsce do przechowywania list spraw do zrobienia. Za przykład podaje to, że o kupieniu baterii do latarki najczęściej przypominamy sobie wtedy, gdy jest awaria prądu, a nie wtedy gdy jesteśmy w sklepie. Jako alternatywę proponuje przeniesienie wszystkich bieżących spraw z głowy do zewnętrznego systemu.

Samo umieszczenie listy spraw w zewnętrznym system jest niewystarczające. Do pełnego zaimplementowania metodyki użytkownik musi wyrobić w sobie nawyk regularnego przeglądania zewnętrznego systemu i porządkowania trafiających do niego zadań. Dopiero wtedy umysł będzie mógł przestać zajmować się składowaniem tej informacji [60].

Cały proces pracy z zadaniami wyglądać powinien następująco:

1. zbieranie – zbieranie wszystkiego, co odciąga uwagę od pracy. W skład rzeczy do zebrania wchodzą np. e-maile, telefony do wykonania, artykuły do przeczytania czy pomysły do przemyślenia
2. przetwarzanie i organizacja – te dwa kroki należą do najbardziej skomplikowanych. Dokładny algorytm przetwarzania i organizacji zostanie przedstawiony niżej
3. przegląd list zadań – żeby system był efektywny użytkownik musi wyrobić w sobie nawyk przeglądania zadań. Przeglądanie zaczyna się od **Kalendarza** i listy **Do wykonania**. Raz w tygodniu należy dokonać przeglądu pozostałych list.
4. wykonywanie zadań – zadania do wykonania powinny być wybierane na podstawie:
   1. kontekstu - inne zadania do wykonania w pracy, inne w sklepie inne w domu
   2. dostępnego czasu – nie ma sensu zabierać się za zadanie, którego nie można skończyć
   3. dostępnej energii – czasem brakuje sił na wykonanie trudnych zadań, wtedy nie ma sensu rozpoczynać pracy nad nimi
   4. priorytetów – sprawy ważne mają pierwszeństwo nad mało istotnymi

Allen proponuje konkretny algorytm wykonywania procesu przetwarzania i organizacji zadań. Został on zapisany przez naukowców z Free University of Brussels w postaci schematu blokowego [61], który prezentuje Rys. 4.1.



Rys. .- Proces przetwarzania i organizacji zadań

### Analiza metodyki GTD

W poprzednim podrozdziale została przedstawiona metodyka zarządzania zadaniami Getting things done wraz z algorytmem przetwarzania i organizacji zadań. Na podstawie tych informacji zostanie teraz przed analiza przedstawionego zagadnienia pod kątem wytworzenia aplikacji wpierającej zarządzanie zadaniami przy użyciu GTD.

Pierwszym punktem procesu GTD jest zbieranie wszystkich spraw odciągających uwagę. Użytkownik powinien móc w dowolnej chwili wyrzucić taką sprawę z umysłu i zapisać ją w swoim bezpiecznym systemie. Możliwie ciągła dostępność systemu sugeruje użycie smartphone’a w celu składowania pojawiających się spraw.

Następnie powinien następować proces przetwarzania i organizacji. Przyglądając się diagramowi Rys. 4.1, można zauważyć, że zadania składowane są na oddzielnych listach, które nazwiemy folderami:

* Skrzynka wejściowa
* Kiedyś/może
* Projekty
* Materiał referencyjny
* Plany projektów
* Oczekiwane
* Kalendarz
* Do wykonania

Warto zwrócić uwagę na jeden specjalny element, który nie jest zwyczajnym folderem – Kalendarz. Zadaniem kalendarza jest przechowywanie zadań powiązanych z konkretną datą – np. spotkanie o godzinie 13:00. Doskonale do tego nadają się aplikacje wbudowane w większość współczesnych telefonów.

# Projekt aplikacji do zarządzania zadaniami

Drugą częścią niniejszej pracy jest stworzenie przykładowej aplikacji przy użyciu technik opisanych w części pierwszej. Pozwoli to na wykazanie, iż zastosowanie dobrych praktyk stosowanych w językach C# i Java ma również zastosowanie w języku JavaScript i prowadzi do tworzenia kodu wysokiej jakości. Zetknięcie się teorii z praktyką umożliwi również wyciągnięcie istotnych wniosków dotyczących badanego zagadnienia.

W tym rozdziale zostaną wyspecyfikowane konkretne wymagania funkcjonalne przykładowej aplikacji, a także sposób badania zagadnień związanych z jakością kodu.

## Cel projektu

Celem implementacyjnej części pracy jest utworzenie przenośnej aplikacji, która posłuży za poligon doświadczalny do rozważań na temat jakości kodu. Wprawdzie sam problem rozwiązywany przez stworzoną aplikację nie będzie szczególnie istotny z punktu widzenia pracy magisterskiej, jednak nie może być ona oderwana od rzeczywistości i musi oferować rozwiązanie rzeczywistego problemu.

Mimo tego, niektóre niezbyt istotne z punktu widzenia omawianej tematyki zagadnienia zostaną potraktowane pobieżnie, a inne zupełnie pominięte. Zostanie to jednak uczynione bez szkody dla badanego problemu.

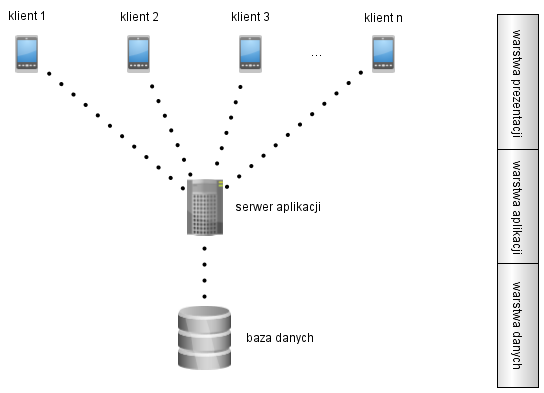
## Wymagania

Na podstawie zebranych informacji jawi się projekt aplikacji: powinien być to zbiór folderów, w których można umieszczać zadania. Ponadto powinna istnieć możliwość przemieszczania zadań pomiędzy folderami. Aplikacja powinna być dostępna na urządzeniach mobilnych. W celu zapewnienia bezpieczeństwa systemu GTD, wszystkie zadania powinny być również umieszczone na zewnętrznym serwerze, dzięki czemu po utracie telefonu będzie można bez problemu wszystko odtworzyć. Aplikacja nie może, jednak polegać na ciągłym dostępie do Internetu musi działać również w trybie offline.

## Projekt architektury wysokiego poziomu

Naturalną fizyczną architekturą dla wymagań przedstawionych w poprzednim rozdziale jest architektura trójwarstwowa. Zakłada ona podział systemu na trzy warstwy (ang. tier), z których każda może być umieszczona na innej fizycznej maszynie [62]. Jej schemat został przedstawiony na Rys. 5.1.

Najwyżej znajduje się warstwa prezentacji, której zadaniem jest wyświetlanie danych i bezpośrednia interakcja z użytkownikiem. Niżej warstwa aplikacji, której zadaniem jest operowanie na danych. Najczęściej składa się ona z jednego lubi wielu serwerów aplikacji. Najniższą warstwą jest warstwa danych. Jej celem jest przechowywanie stanu aplikacji. Najczęściej składa się ona z jednego lub wielu serwerów bazodanowych.



Rys. . - Architektura trójwarstwowa

### Warstwa prezentacji

Wybór technologii wykonania warstwy prezentacji został zdeterminowany przez tematykę wytwarzania kodu wysokiej jakości w języku JavaScript. Framework wykorzystany do budowy aplikacji klienckiej musiał wykorzystywać wspomniany język oraz oferować przenośność kodu.

#### PhoneGap

Najbardziej popularnym rozwiązaniem tworzenia przenośnych aplikacji hybrydowych, na którym bazuje wiele innych, bardziej skomplikowanych platform jest PhoneGap. Powstał on jako produkt kanadyjskiej firmy Nitobi, która została zakupiona przez Adobe. Następnie został otwarty i udostępniony pod nazwą Cordova. Projekt w praktyce rozwija się jako Cordova, i wykorzystujący Cordovę PhoneGap – do tej nazwy prawa ma Adobe [63].

Praca z PhoneGapem zazwyczaj wygląda następująco:

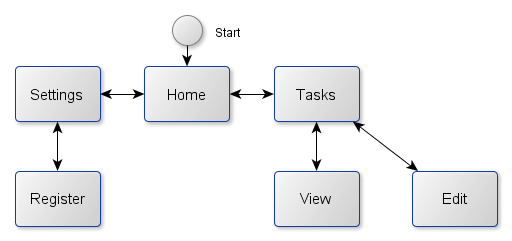
1. Programista tworzy szablon aplikacji. Zawiera on kod aplikacji natywnej, która będzie kontenerem dla HTML-a i JavaScriptu.
2. Programista pisze aplikację na wybraną platformę i testuje ją na emulatorze bądź urządzeniu.
3. W celu przeniesienia aplikacji na inną platformę, tworzony jest nowy szablon we właściwym dla danego systemu środowisku.
4. Do szablonu zostaje przeniesiony kod HTML i JavaScript. Jeśli aplikacja nie korzystała z własnych pluginów, to powinno wystarczyć do uruchomienia jej na zupełnie innej platformie.

Framework PhoneGap został wybrany jako rozwiązanie do tworzenia kodu warstwy widoku w omawianej aplikacji. Zdecydowało o tym kilka czynników:

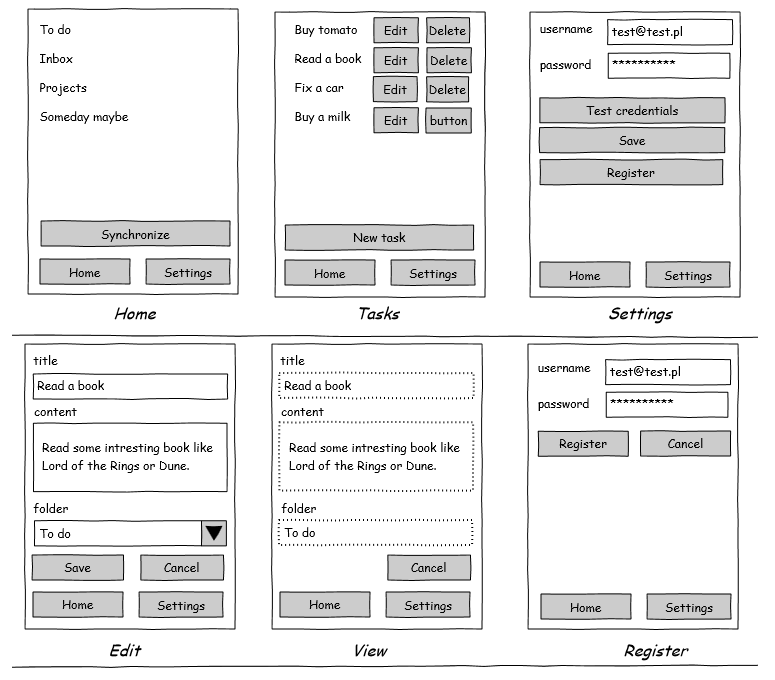
* Jest prosty, skupia się tylko na współpracy kodu JavaScript z urządzeniem, nie dostarcza serwera, ani skomplikowanej abstrakcji na HTML.
* Jest popularny, dzięki czemu łatwo znaleźć rozwiązania częstych problemów czy przykładowy kod.
* Jest darmowy – wszystkie elementy platformy są darmowe, zarówno natywne biblioteki jak i kod JavaScript.

#### Projekt interfejsu użytkownika

Projekt interfejsu został przedstawiony na dwóch diagramach. Pierwszy - Rys. 5.3 – przedstawia projekt widoków aplikacji. Drugi - Rys. 5.2 – przedstawia diagram przejść między widokami.



Rys. . - Diagram przejść między widokami



Rys. . - Projekt interfejsu

### Warstwa aplikacji

W warstwie aplikacji oczywistym wyborem było użycie Node.js-a. Jest on obecnie jedynym dostępnym serwerem aplikacji programowanym w JavaScript. Technologia ta została dokładnie opisana w rozdziale 2 i opis ten nie będzie powielany.

### Komunikacja między serwerem i klientem

Istnieje wiele rozwiązań pozwalających na komunikację maszyn połączonych przez sieć. Jednym z nich są usługi internetowe, a platformy technologiczne takie jak .NET i Java oferują bogate wsparcie dla nich.

JavaScript nie posiada jeszcze odpowiedniego zestawu narzędzi, dlatego preferowanym podejściem jest użycie lżejszego rozwiązania, jakim jest stworzenie REST-owych serwisów, przesyłających dane w formacie JSON.

#### REST

REST to termin stworzony w 2000 roku przez Roya Fieldinga. Sam w sobie nie jest on bezpośrednio związany z tworzeniem API pozwalającego na komunikację klienta z serwerem. Opisuje on styl architektury opartej na zasobach, która spełnia zestaw wymagań [64]:

* **Klient-serwer** – w architekturze REST następuje oddział klienta od serwera. Klient nie powinien znać szczegółów implementacji, czy sposobu składowania danych. Serwer nie powinien wiedzieć jak wygląda np. interfejs użytkownika.
* **Bezstanowość** – każde żądanie klienta powinno zawierać wszystkie informacje potrzebne do jego wykonania. Nie może występować np. taka sytuacja w której klient najpierw wybiera zasób, do którego będzie dodawał elementy, a później, w kolejnych żądaniach wspomniane elementy dodaje. Innymi słowy serwer nie może polegać na informacja uzyskanych z poprzednich żądań.
* **Cache’owalność** – architektura powinna umożliwiać cache’owalność, a co za tym idzie odpowiedzi pochodzące z serwera powinny informować jeśli dany zasób można przechowywać w cache’u.
* **System warstw** – klient nie powinien móc stwierdzić czy jest połączony bezpośrednio z serwerem, czy to połączenie odbywa się z użyciem pośredników.
* **Kod na żądanie (opcjonalna)** – możliwość przesyłania kodu do wykonania po stronie klienta.
* **Ujednolicony interfejs** – to wymaganie składa się z czterech zasad:
  + **Identyfikacja zasobów** – zasoby powinny być rozróżnialne przy użyciu np. URI
  + **Manipulacja zasobami, poprzez manipulację reprezentacją** – klient powinien mieć dość informacji, w celu manipulacji zasobami serwera
  + **Samoopisujące się wiadomości** –każda wiadomość powinna zawierać informacje wystarczające do jej przetworzenia
  + **Hipermedia jako silnik stanu aplikacji** –akcje dostępne dla klienta powinny być opisane wewnątrz zasobów dostarczanych przez serwer

REST choć od niego niezależny jest silnie powiązany z protokołem HTTP. W rzeczywistości World Wide Web jest największą implementacją podążającą za zasadami opisanymi przez styl architektoniczny.

#### ROA

Oparcie się jedynie na ograniczeniach narzucanych przez REST przy budowaniu serwisów internetowych może okazać się niewystarczające. Autorzy książki RESTful Web Services – Leonard Richardson i Sam Ruby zauważają, że REST nie definiuje ściśle architektury, jest jedynie zestawem wskazówek. Nie odnosi się on w żaden sposób do protokołu HTTP, ani nawet do www.

W celu ograniczenia niejednoznaczności i szerokiej dowolności autorzy wspomnianej książki zdefiniowali termin Resource-oriented architecture (ROA) – Architektura zorientowana na zasoby. Niniejszy opis powstał w oparciu o książkę RESTful Web Services [64].

Ruby i Richardson wprowadzają szeroką definicję terminu zasób. Jest nim coś co jest reprezentowane jako strumień bitów np. dokument, wiersz w bazie danych czy nawet rezultat działania algorytmu. Przykładami zasobów są:

* Wersja 1.0.3 jakiegoś oprogramowania
* Najnowsza wersja jakiegoś oprogramowania
* Informacje na temat meduzy
* Kolejna liczba pierwsza występująca po 1024
* Związek pomiędzy Alice i Bob-em

Każdy zasób powinien być adresowalny, czyli być dostępnym pod co najmniej jednym URI. Jeśli coś nie ma adresu, oznacza to że nie jest dostępne w Sieci i w związku z tym nie jest zasobem w myśl ROA, gdyż klient manipuluje zasobami za pośrednictwem adresów.

Adresy powinny być opisowe i ustrukturalizowane. Dobrym przykładem adresu jest: */szukaj/Meduza*, złym */chce-wiedziec-o/Meduza*. Pierwszy prezentuje logiczną strukturę, dzięki czemu łatwo jest go użyć klientowi w sposób nie przewidziany przez twórcę API.

Nie mogą istnieć dwa takie same zasoby, ale jeden zasób może być wskazywany przez kilka adresów. Dobrym przykładem jest dostęp do paczek z oprogramowaniem. Wersja 1.0.3 jakiegoś programu może być dostępna pod dwoma różnymi adresami *http://jakisprogram.pl/wersje/1.0.3.zip* oraz *http://jakisprogram.pl/wersje/najnowsza.zip*. Oczywiście w momencie gdy powstanie wersja 1.0.4, drugi adres przestanie wskazywać to samo co pierwszy.

Poza adresowalnością, bardzo ważną cechą ROA jest bezstanowość, która jest narzucona bezpośrednio przez zasady REST. W ROA wynika ona z faktu, iż wszystkie interesujące klienta informacje są dostępne jako zasoby, a zasoby muszą być zgodnie z zasadą adresowalności dostępne za pomocą adresów.

Przy omawianiu aspektu bezstanowości pojawia się problem – można stwierdzić iż oznacza ona, że serwer nie może zapisywać żadnych informacji. Czyli np. użytkownik nie może dodać nowego wpisu na blogu, gdyż jego przechowywanie wymaga pamiętania stanu. Takie stwierdzenie wynika z nierozróżnienia miedzy stanem aplikacji i stanem zasobów. Zakaz utrzymywania stanu odnosi się tylko do tego pierwszego, który powinien znajdować się jedynie po stronie klienta. Zasoby mogą posiadać stan, który zapisywany najczęściej jest najczęściej w bazie danych.

Dostęp do zasobów w architekturze ROA odbywa się za pośrednictwem protokołu HTTP. Oferuje on cztery podstawowe metody służące do wykonywania operacji na nich:

* **GET** – zadaniem tej metody jest pobieranie reprezentacji zasobu
* **PUT** – modyfikacja istniejącego zasobu, albo utworzenie nowego
* **POST** – utworzenie nowego zasobu
* **DELETE** - usunięcie zasobu

Z początku mylącym może być fakt, iż zarówno PUT jak i POST pozwalają na utworzenie nowego zasobu. Różnica między zastosowaniem tych dwóch metod jest taka, że PUT używany jest na nowym URI, który nie identyfikuje żadnego zasobu, natomiast POST na URI, pod którym już istnieje zasób. Metoda POST jest używana do tworzenia zasobów podrzędnych.

Innym dopuszczalnym sposobem wykorzystania metody POST jest wykonywanie na zasobie operacji dopisywania (ang. append). Dla przykładu mając zasób reprezentujący log, można do niego jedynie pisać i wyświetlić go w całości dlatego poprawnym będzie zastosowanie na nim metod POST i GET.

#### Struktura zasobów

W celu umożliwienia części klienckiej interakcji z częścią serwerową, zgodnie ze standardami ROA została zdefiniowana następująca struktura zasobów:

|  |  |  |
| --- | --- | --- |
| metoda | uri | opis |
| POST | /user/{username}/change | Dodaje zmianę do loga zmian. Jest to zasób wyliczany, na co pozwalają zasady ROA. Dodanie zmian powoduje aktualizację zasobu task. |
| GET | /user/{username}/task | Pobiera wszystkie zadania dla danego użytkownika. Niemożliwe jest bezpośrednie edytowanie zadań. Odbywa się ono pośrednio poprzez dodawanie nowych zmian. |
| GET | /user/{username} | Pobiera detale danego użytkownika. |
| PUT | /user/{username} | Tworzy nowego użytkownika na podstawie przekazanych detali. |

Tabela .- Struktura zasobów ROA

### Warstwa danych

Tematyka niniejszej pracy każe przyjrzeć się dostępności baz danych programowanych w języku JavaScript. Jak się okazuje istnieją co najmniej dwa rozwiązania spełniające założone wymaganie, są nimi:

* CouchDb
* MongoDb

Oba DBMS-y zaliczają się do zbioru NoSQL, który zawiera wszystkie bazy danych z wyłączeniem relacyjnych.

#### CouchDb

CouchDb to otwarta dokumentowa baza danych, która jest napisana w funkcyjnym języku Erlang. Pierwsza wersja bazy została udostępniona w 2005 roku.

W przeciwieństwie do relacyjnych baz danych, dane w CouchDb składowane są nie w tabelach, składających się z rekordów lecz w kolekcjach zawierających dokumenty, w formacie JSON, które w przeciwieństwie do tabel z baz relacyjnych nie mają narzuconego z góry schematu.

Dostęp do danych obywa się za pomocą widoków, których kod pisze się w JavaScript. Dostępne są dwa rodzaje widoków [65]:

* Oparte o funkcję map –wyświetlanie zbioru elementów w zmienionej formie
* Oparte o funkcję reduce – wyświetlanie danych po zgrupowaniu (np. suma wartości jakiegoś pola)

Głównymi zaletami bazy są [66]:

* Zdolność do bezawaryjnej obsługi nagłych skoków ruchu
* Skalowalność
* Prostota użycia
* Bezproblemowa obsługa brakujących danych
* Wsparcie dla pracy offline na urządzeniach mobilnych

Ostatnia zaleta wygląda bardzo interesująco z punktu widzenia wytwarzania urządzeń mobilnych. CouchDb oferuje Syncpoint API – API dla Node.js-a pozwalające na automatyczną synchronizację lokalnej i zdalnej bazy danych. Dzięki temu w przypadku utraty połączenia z siecią, urządzenie może bez problemów pracować z lokalną wersją bazy danych, która zostanie zsynchronizowana ze zdalną po odzyskaniu dostępu do Internetu.

Po stronie klienta rozwiązanie wymaga instalacji lekkiej wersji bazy - TouchDB. Aplikacja jest obecnie dostępna na platformy Mac OS X, iOS , Linux, Windows i Android. Brak wsparcia dla np. Windows Phone stanowi niestety dużą wadę w przypadku tworzenia programu, który ma być dostępny na wielu platformach. Z tego powodu lokalna baza wraz z replikacją nie mogła być użyta do tworzenia opisywanej aplikacji.

#### MongoDb

Bardzo podobnym rozwiązaniem do CouchDb jest baza MongodDb. Została ona napisana w języku C++, a pierwsza wersja ukazała się w 2009 roku.

Jest to podobnie jak CouchDb dokumentowa baza danych, ale przechowuje dokumenty nie w formacie JSON tylko BSON (Binary JSON), który jest rozszerzeniem tego pierwszego. Pola mogą przyjmować wartości następujących typów [67]:

* **string**
* **integer** (32 lub 64 bity) – BSON poza liczbami zmiennoprzecinkowymi wspiera też możliwość składowania liczb całkowitych
* **double** (64 bity)
* **date** – w przeciwieństwie do JSON-a występuje tu oddzielny typ dla daty
* **byte array** – BSON umożliwia składowanie danych binarnych
* **boolean**
* **null**
* **BSON object** – dokumenty BSON można w sobie zagnieżdżać podobnie jak ma to miejsce w JSON-ie
* **BSON array -** w dokumentach można składować tablice zawierające zbiory dokumentów
* **regular expression**
* **JavaScript code** – w dokumentach można składować kod JavaScript

MongodDb nie obsługuje widoków - dostęp do danych odbywa się za pomocą zapytań, które są dokumentami BSON, mogącymi zawierać specjalne elementy. Takie rozwiązanie nie jest powszechnie spotykane dlatego zostanie szerzej omówione na przykładzie.

Tabela 5.2 przedstawia zawartość kolekcji *items*. Każdy wiersz jest oddzielnym dokumentem. Pole \_id ma specjalne znaczenie – jest ono unikatowe w obrębie całej kolekcji.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| \_id | name | unitPrice | | inStock |
| ObjectID(504da27fcabcb22c214d7411) | "tomato" | 2.99 | 20.0 | |
| ObjectID(504da27fcabcb22c214d7412) | "potato" | 1.99 | 17.0 | |
| ObjectID(504da27fcabcb22c214d7413) | "orange" | 4.99 | 3.0 | |
| ObjectID(504da27fcabcb22c214d7414) | "banana" | 4.59 | 7.0 | |
| ObjectID(504da27fcabcb22c214d7415) | "apple" | 1.59 | 25.0 | |
| ObjectID(504da27fcabcb22c214d7416) | "pineapple" | 10.99 | 5.0 | |
| ObjectID(504da280cabcb22c214d7417) | "mango" | 4.99 | 4.0 | |

Tabela . - Dokumenty w kolekcji items

W celu pobrania np. dokumentu oznaczającego jabłka należy użyć metody find na kolekcji items:

> db.items.find({name: "apple"})

Jako parametr find przekazywany jest document BSON, który ma pole name ustawione na “apple”. MongoDb wyszuka dokumenty, które mają pasujące właściwości i w rezultacie zwróci następującą kolekcję:

[

{

"\_id" : ObjectId("504da27fcabcb22c214d7415"),

"name" : "apple",

"unitPrice" : 1.59,

"inStock" : 25

}

]

Dopasowania dokumentów nie są ograniczone tylko do prostego przyrównywania pól. Zapytanie:

> db.items.find({unitPrice: {$gt: 3, $lt : 5}})

zwróci wszystkie dokumenty, które mają wartość unitPrice należącą do zbioru (3, 5):

[

{

"\_id" : ObjectId("504da27fcabcb22c214d7413"),

"name" : "orange",

"unitPrice" : 4.99,

"inStock" : 3

},

{

"\_id" : ObjectId("504da27fcabcb22c214d7414"),

"name" : "banana",

"unitPrice" : 4.59,

"inStock" : 7

},

{

"\_id" : ObjectId("504da280cabcb22c214d7417"),

"name" : "mango",

"unitPrice" : 4.99,

"inStock" : 4

}

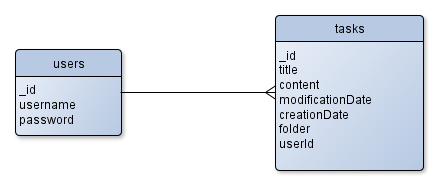
]

Dokument zagnieżdżony, który jest przypisany do pola unitPrice dokumentu wzorcowego pozwala wyspecyfikować skomplikowane kryteria, na podstawie których baza stwierdzi czy nastąpiło dopasowanie.

MongoDb została wybrana z powodu większej popularności.

### Schemat danych

Pomimo tego, że MongoDb jest bazą, która nie wymaga ścisłego zdefiniowania schematu dokumentów, to jednak w praktyce dane muszą mieć ustrukturalizowaną formę. Schemat bazy danych przedstawia Rys. 5.6.



Rys. . - Schemat bazy danych

Na wyjaśnienie zasługuje to, czemu lista zadań nie została zagnieżdżona wewnątrz danych użytkownika pod postacią listy dokumentów. Zadania będą często dodawane, co wiązałoby się z koniecznością przenoszenia elementów kolekcji users w nowe miejsca na dysku, a jest to niewydajne [68].

## Narzędzia

Ważnym elementem każdego projektu programistycznego jest zestaw narzędzi, które będą użyte w czasie wytwarzania aplikacji. Dobrze dobrane potrafią znacznie przyspieszyć i ułatwić pracę.

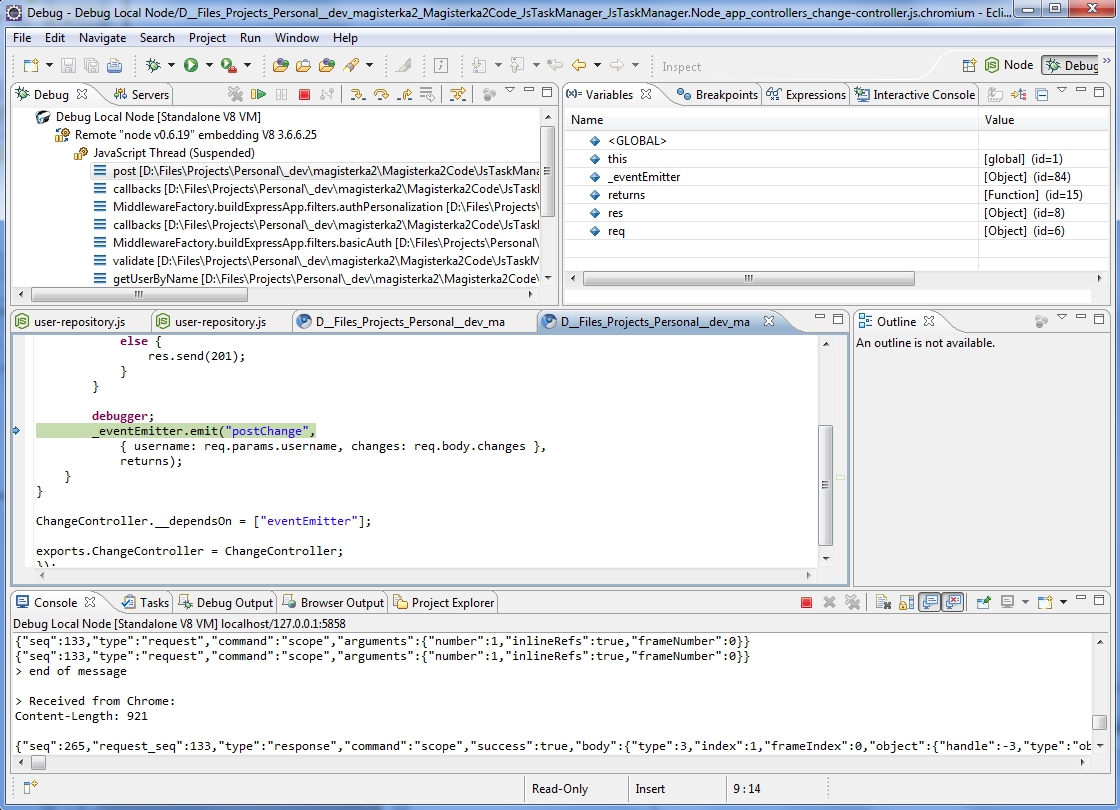
#### Środowisko programistyczne

Głównym środowiskiem programistycznym wykorzystanym w projekcie było Visual Studio 2010. Niestety wsparcie dla JavaScriptu po stronie serwera jest dosyć ubogie we wszystkich dostępnych IDE. Nie radzą sobie one przede wszystkim z obsługą funkcji require(), przez co nie ma wsparcia dla podpowiadania składni opartej na kodzie pochodzącym z innych modułów.

Wybór Visual Studio nad innymi narzędziami wynikał głównie z mojego przyzwyczajenia do technologii firmy Microsoft.

#### Debugger

Na początku jedyną możliwością debuggowania aplikacji było wykorzystanie klienta działającego w trybie tekstowym. Obecnie możliwe jest użycie pluginu do Eclipse, który potrafi połączyć się za pomocą socketów z silnikiem V8 i tym samym przeprowadzać debugging w środowisku graficznym, widocznym na Rys. 5.7.



Rys. . - Debugger V8

Debugger i edytor w środowisku Eclipse nie integrują się ze sobą. Debuggowanie odbywa się na serwerowych kopiach kodu JavaScript, dlatego nie ma np. możliwości postawienia breakpointów w czasie programowania, które spowodują zatrzymanie uruchomionej aplikacji. Zamiast tego trzeba polegać na dopisywanych bezpośrednio w kodzie instrukcjach *debugger*.

#### System kontroli wersji

Istotnym problemem, którego nie można pominąć w czasie tworzenia aplikacji jest zarządzanie zmieniającym się kodem. W niniejszym projekcie w tym celu wykorzystany został Mercurial, który należy do rozproszonych systemów kontroli wersji.

Oferuje on proste i wygodne tworzenie gałęzi, możliwość pracy offline i odtworzenia repozytorium centralnego na podstawie kopii znajdującej się na maszynie programisty. Ta ostatnia cecha przydała się w praktyce, gdyż maszyna, na której przechowywany był kod uległa awarii.

## Testowanie jednostkowe aplikacji

Poniższy podrozdział opisuje temat testowania jednostkowego aplikacji JavaScript, związane z nim problemy i ich praktyczne rozwiązania.

#### Biblioteki do testów jednostkowych

W celu zapewnienia jakości kodu zarówno w aspekcie strukturalnym jak i funkcjonalnym, aplikacja była w całości tworzona przy użyciu Test Driven Development. Z pewnymi drobnymi właściwy kod był pisany tylko i wyłącznie po stworzeniu odpowiednich testów.

W celu efektywnego testowania, niezbędne jest wybranie odpowiednich narzędzi. W rozdziale 3 zostało przedstawionych kilka framewowrków przeznaczonych do testów jednostkowych w JavaScript. Zostały one podzielone na dwie grupy

* Wzorowane na JUnit
* Wzorowane na RSpec

Narzędzia z drugiej grupy zostały uznane za bardziej pożądane ze względu na opisowość testów w nich tworzonych. Do drugiej grupy należą dwie biblioteki: Jasmine i Mocha.

Jasmine została stworzona z myślą o testowaniu kodu w przeglądarce, jednak jest też dostępna wersja działająca pod Node.js. Biblioteka posiada własny zestaw asercji napisany w stylu expect. Przykład Rys. 5.9 pokazuje prosty test napisany w Jasmine.

describe("A suite", function () {

    it("contains spec with an expectation", function () {

        var num = 5 + 2;

        expect(num).toEqual(7);

    });

});

Rys. . - Przykład testu napisanego w Jasmine

Testowana wartość jest przekazywana do funkcji expect(), która zwraca obiekt, na którym można dokonać testu za pomocą fluentowego interfejsu. Dzięki temu kod czyta się jak zdanie w języku naturalnym.

Alternatywą dla Jasmine jest Mocha. W przeciwieństwie do Jasmine lepiej sprawuje się lepiej w środowisku Node i domyślnie nie wspiera uruchamiania w przeglądarce (choć jest to możliwe). Dużą zaletą Mocha jest runner, czyli aplikacja, która pozwala rekurencyjnie przeszukać dany katalog i wykonać wszystkie znajdujące się w nim testy.

Mocha nie posiada własnych asercji, zamiast tego polega na zewnętrznych bibliotekach np. Chai. Rys. 5.10 prezentuje ten sam test napisany przy użyciu dwóch różnych stylów asercji expect i should.

describe("A expect suite", function () {

    it("contains spec with an expectation", function () {

        var num = 5 + 2;

        expect(num).to.be.equal(7);

    });

});

describe("A should suite", function () {

    it("contains spec with an expectation", function () {

        var num = 5 + 2;

        num.should.be.equal(7);

    });

});

Rys. . - Przykłado testu napisanego w Mocha z użyciem Chai expect i should

Inną ważną zaletą Mocha jest dobre wsparcie dla testowania asynchronicznego kodu, który często pojawia się zwłaszcza w aplikacjach Node.js. Rys. 5.11 ilustruje test, który z oczywistych powodów (5+2 = 7, nie 15) nie powinien się powieść. Jednak po uruchomieniu go Mocha zwróci informację, iż wszystkie testy zostały wykonane poprawnie.

describe("async tests", function () {

    it("should fail", function () {

        setTimeout(function () {

            expect(5 + 2).to.equal(15);

        }, 100);

    });

});

Rys. . - Ten test nie powinien się powieść

Dzieje się tak dlatego, że asercja zostanie wykonana asynchronicznie po odczekaniu 100 milisekund. Mocha, jednak nie wie o tym i dlatego od razu zakończy wykonywanie testu, a skoro nie napotkano błędów oznacza to, że test wykonał się poprawnie.

Framework oferuje bardzo wygodne rozwiązanie tego problemu, które ilusturje Rys. 5.12. Od poprzedniego różni się on tylko dwoma rzeczami. Jedną z nich jest to, że funkcja przekazana do it() ma parametr done.

describe("async tests", function () {

    it("should fail", function (done) {

        setTimeout(function () {

            expect(5 + 2).to.equal(15);

            done();

        }, 100);

    });

});

Rys. . - Ten test nie powiedzie się

Mocha wykrywa ten fakt za pomocą właściwości length obiektu typu function i zakłada, że test zawiera kod asynchroniczny. Wobec tego będzie czekać, aż nastąpi wywołanie funkcji podstawionej pod parametr done. Dzięki temu asercja wykona się i test nie uda się.

Do testowania części serwerowej aplikacji wybrany został framework Mocha ze względu na dobre wsparcie dla Node.js-a i asynchronicznych operacji. Wybór stylu asercji jest sprawą bardzo subiektywną. Do użycia w projekcie został wybrany styl expect, ponieważ używa go również Jasmine.

#### Testowanie widoków w aplikacji klienckiej

W aplikacji klienckiej do celów testowania jednostkowego został zastosowany, opisany już wcześniej framework Jasmine. Czynnikiem decydującym było wsparcie dla przeglądarki Internet Explorer – pod tą przeglądarką Mocha nie działa (dotyczy to nawet wersji 9). Jest to spowodowane tym, że niektóre funkcje JavaScript, które są obecne w V8 i używane przez framework nie zostały jeszcze zaimplementowane w produkcie Microsoftu.

Problemem, który pojawia się przy próbach testowania kodu aplikacji klienckiej jest testowanie widoków. Kod HTML i manipulujący nim JavaScript są zazwyczaj silnie powiązane, co powoduje trudności w testowaniu jednostkowym. Problem ten ilustruje Rys. 5.18.

//Kod html

Name:

<span id="nameSpan"></span>

<br />

Surname:

<span id="surnameSpan"></span>

//kod JavaScript

function setName() {

    $("#nameSpan").text("Tony");

    $("#surnameSpan").text("Stark");

}

Rys. . - Silne powiązanie między kodem JavaScript i HTML

Funkcja setName() jest zależna od kodu HTML poprzez bezpośrednie odwołania do jego elementów o id nameSpan i surnameSpan. Przetestowanie takowego kodu jest możliwe jedynie w całości – trzeba utworzyć fragment sztucznego, przeznaczonego jedynie dla testów drzewa DOM, na którym operować będzie JavaScript, a później sprawdzić czy stan struktury odpowiada oczekiwaniom. To podejście choć zawiera oczywistą zaletę – nie pomija żadnej części aplikacji – ma też bardzo istotną wadę – jest uciążliwe.

Alternatywą jest jedynie testowanie kodu manipulującego widokiem. Wzorcem projektowym, który to umożliwia jest Model View View-Model (MVVM). Został on wymyślony przez firmę Microsoft na potrzeby technologii opartych na języku XAML: WPF i Silverlight.

Wzorzec zakłada istnienie trzech warstw:

* **Model** – jest to dokładnie taka sama warstwa co we wzorcu MVC. W tym miejscu znajduje się logika biznesowa aplikacji, a także odwołania do np. bazy danych czy Web Service’ów.
* **View (Widok)** – jest to sam interfejs, nie zawierających żadnych zdarzeń obsługujących go. W przypadku aplikacji webowych widok będzie zazwyczaj zawierał sam kod HTML.
* **View-Model** – warstwa pośrednicząca między widokiem i modelem. Kod JavaScript nie manipuluje bezpośrednio widokiem, a jedynie stanem View-Modelu. Zmiany w nim są automatycznie odzwierciedlane w stanie widoku za pomocą mechanizmu data-bindingu.

Przykład użycia MVVM przedstawia Rys. 5.19. Kolorami zaznaczono, które elementy Widoku są powiązane z polami View-Modelu. Data-binding dba o to, żeby zawsze były one zsynchronizowane. W przypadku edytowalnych elementów działa to w obie strony.

//Kod html

<div id="containerDiv">

    Name: <span data-bind="text: name"></span>

    <br />

    Surname: <span data-bind="text: surname"></span>

    <br />

    <a href="#" data-bind="click: setName">Set name</a>

</div>

//Kod JavaScript

var PersonKnockout = function () {

    this.name = ko.observable("");

    this.surname = ko.observable("");

    this.setName = function () {

        this.name("Tony");

        this.surname("Stark");

    }

}

Rys. . - Odwrócenie zależności kodu HTML i JavaScript

Poprzez zastosowanie MVVM zależność między kodem JavaScript i HTML nie została przerwana lecz odwrócona. Poprzednio kod JavaScript był zależny od HTML, a teraz jest na odwrót, gdyż atrybut data-bind specyfikuje z jakim polem View-Modelu jest powiązany dany element. Pozwala to na testowanie samej logiki manipulującej widokiem, pomijając całkowicie jego strukturę.

Przykładowy zestaw testów specyfikujących zachowanie widoku przedstawia Rys. 5.20. Istotne jest, to że testy nigdzie nie odwołują się do elementów drzewa DOM – View-Model jest testowany w zupełnej izolacji.

describe("PersonViewModel", function () {

    describe("right after intialization", function () {

        var view = new PersonKnockout();

        beforeEach(function () {

            view = new PersonKnockout();

        });

        it("should have name set to empty string", function () {

            expect(view.name()).toEqual("");

        });

        it("should have name set to empty string", function () {

            expect(view.surname()).toEqual("");

        });

    });

    describe("when setName was called", function () {

        var view = new PersonKnockout();

        beforeEach(function () {

            view = new PersonKnockout();

            view.setName();

        });

        it("should have name set to Tony", function () {

            expect(view.name()).toEqual("Tony");

        });

        it("should have surname set to Stark", function () {

            expect(view.surname()).toEqual("Stark");

        });

    });

});

Rys. . - Testy View-Modelu

W projekcie użyta została biblioteka biblioteka MVVM stworzona przez pracownika Microsoftu Steve’a Sandersona - Knockout.js. Czerpie ona bezpośrednio z rozwiązań zastosowanych w technologiach WPF i Silverlight próbując przenieść je na grunt aplikacji webowych. Knockout oferuje wsparcie dla mechanizmu data-bindingu, posiada także dobrą dokumentację.

## Plan badania jakości kodu stworzone aplikacji

W celu wykazania, iż możliwe jest wykonanie aplikacji JavaScript z kodem wysokiej jakości, zostanie zmierzonych kilka parametrów stworzonych klas i metod.

### Mierzenieilości naruszeń SOLID

Jak zostało to opisane w rozdziale trzecim, zastosowanie zasad SOLID opisanych w rozdziale trzecim pozwala poprawić jakość kodu obiektowego.

W fazie badania kodu wykonanego w części implementacyjnej niniejszej pracy, zostanie zbadane ile razy każda obecna w systemie klasa narusza zasady jednej odpowiedzialności, otwarte/zamknięte i odwrócenia zależności. Pozostałe – segregacji interfejsów i podstawienia Liskov zostaną pominięte. Pierwsza z nich ma małe znaczenie w języku JavaScript, za wyjątkiem udostępniania API dla biblioteki, druga jest sprawdzana poprzez testy jednostkowe.

Jako, że SOLID są raczej wskazówkami niż ścisłymi zasadami nie istnieją żadne wytyczne mówiące na temat tego jaka ilość naruszeń jest w kodzie dopuszczalna. Niewątpliwie jednak zdecydowana większość klas powinna omawiane zasady respektować..

### Mierzenie wartości metryk jakości kodu

W rozdziale trzecim został opisany szereg metryk związanych z jakością kodu. Wybrane z nich – złożoność cyklomatyczna metod, CBO i poziom pokrycia – zostaną zmierzone dla zaimplementowanej aplikacji. Żadna z metryk nie powinna przekroczyć wartości granicznych opisanych wcześniej.

### Subiektywna ocena jakości kodu

Jako, że postrzeganie jakości kodu jest w dużej mierze subiektywne, kod utworzonej aplikacji zostanie poddany ocenie przez innych programistów znających język JavaScript. Zostaną oni poproszeni o przejrzenie źródeł i ocenienie jakości strukturalnej całego projektu w skali od 0 do 5.

# Implementacja aplikacji do zarządzania zadaniami

Pierwsza część niniejszego rozdziału opisuje implementację aplikacji wspierającej zarządzanie zadaniami według opisanej wyżej metodyki GTD. Druga prezentuje wyniki badań jakości kodu.

## Implementacja

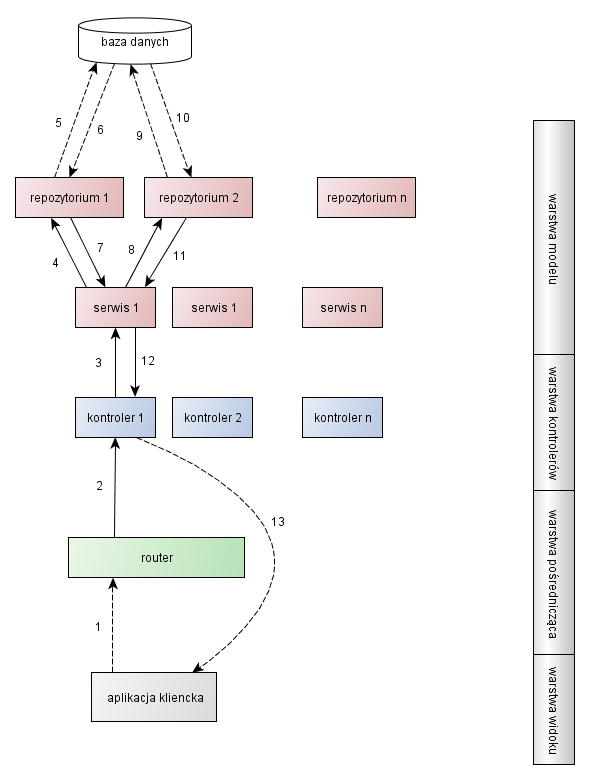
Poprzedni rozdział zawierał informacje na temat architektury aplikacji, podziału jej na warstwy fizyczne i wyboru głównych technologii, o które będzie ona oparta. W tym podrozdziale każda z warstw zostanie opisana na niższym poziomie wraz z jej architekturą logiczną, stosem technologicznym i napotkanymi w czasie implementacji problemami.

### Aplikacja serwerowa

#### Architektura niskiego poziomu

Poza podziałem na warstwy fizyczne, który został omówiony wcześniej, każda z części (za wyjątkiem bazy danych) została podzielona na warstwy logiczne.

W przypadku aplikacji serwerowej zastosowano architektoniczny wzorzec projektowy MVC, z tym, że widoki nie są udostępniane przez serwer, tak jak ma to miejsce w wypadku aplikacji webowych.



Rys. . - Przebieg obsługi żądania przez wszystkie warstwy aplikacji serwerowej

Diagram przedstawiony na Rys. 5.13 prezentuje podział aplikacji na warstwy. Zostanie on omówiony na przykładzie obsługi żądania.

1. Aplikacja kliencka wysyła żądanie HTTP w celu otrzymania zasobu w sensie ROA. Jest ono odbierane przez warstwę pośredniczącą, zawierającą router, który decyduje jaki kontroler i jaka jego metoda ma je obsłużyć.
2. Wywołana zostaje odpowiednia metoda kontrolera 1, gdzie z niskopoziomowego reprezentującego żądanie HTTP wyciągane są istotne informacje.
3. Kontroler 1 odwołuje się do serwisu 1, które otrzymuje już czyste dane. Należy on do warstwy modelu i działa na innym poziomie abstrakcji niż kontroler, dlatego „nie ma świadomości”, że obsługuje żądanie HTTP.
4. Serwis 1 odwołuje się do repozytorium 1, w celu uzyskania danych potrzebnych mu do wykonania zadania. Serwis nie wie, że odwołuje się do dokumentowej bazy danych. W przypadku zmiany decyzji projektowych mógłby odwoływać się np. do pliku XML, albo bazy SQL.
5. Repozytorium 1 odwołuje się do bazy danych. Jest ono już świadome tego z jaką bazą pracuje. Wymiana sposobu składowania danych wymaga wymiany repozytorium.
6. Baza zwraca dane, o które prosiło repozytorium 1.
7. Repozytorium 1, jeśli jest taka konieczność, przetwarza otrzymane z bazy dane i przekazuje je do serwisu.
8. Serwis na podstawie właśnie otrzymanych danych dokonuje odwołania się do repozytorium 2, w celu pobrania innego zestawu danych.
9. Repozytorium 2 odwołuje się do bazy
10. Baza zwraca dane
11. Repozytorium 2 zwraca dane do serwisu 1.
12. Serwis 1ma już wszystkie dane jakich potrzebuje. Dokonuje teraz na nich operacji i rezultat zwraca do kontrolera 1.
13. Kontroler 1 przygotowuje i wysyła odpowiedź dla klienta - np. dodaje odpowiednie nagłówki http, albo serializuje dane do odpowiedniej postaci.

#### Warstwa pośrednicząca

Zadaniem warstwy pośredniczącej w wypadku omawianej aplikacji jest głównie przeprowadzanie routingu czyli decydowanie o tym, która metoda, którego kontrolera ma obsłużyć żądanie o danym adresie URL i metodzie HTTP.

Do projektu została wybrana biblioteka Express, które jest najpopularniejszym rozwiązaniem tego typu, a ponadto posiada dobrą dokumentację.

#### Kontener IoC

Pisanie aplikacji zgodnie z założeniami TDD wymaga tworzenia słabo powiązanego kodu. Przykładowo klasa MyService, nie może polegać bezpośrednio na implementacji MyRepository, gdyż w takiej sytuacji test dotyczący MyService, będzie musiał operować na bazie danych. Testowanie jednostkowe zakłada izolację komponentów, dlatego też w celu przetestowania klasy MyService należy referencję do instancji MyRepository zastąpić mockiem, czyli obiektem, który ma właściwość zastępowalności względem instancji MyRepository. Dzięki użyciu mocka, kod MyService nie będzie operował na bazie.

Wprawdzie w JavaScript możliwe jest podmienienie dowolnych komponentów w czasie działania programu, więc teoretycznie możliwe jest zastąpienie konstruktora MyService przez konstruktor tworzący mocka, ale takie rozwiązanie jest nieeleganckie, gdyż ukrywa zależności klasy.

Zamiast uciekania się do sztuczek, kod omawianej aplikacji tworzony był zgodnie z zasadą odwrócenia zależności, opisaną w rozdziale 3.

Niestety dość szybko pojawił się problem - budowanie grafu obiektów aplikacji stawało się coraz bardziej skomplikowane. Ten problem w technologiach takich jak .NET czy Java rozwiązuje się za pomocą kontenerów IoC.

IoC, czyli Odwrócenie Sterowania (Inversion of Control) to wzorzec projektowy zakładający przeniesienie na zewnątrz komponentu odpowiedzialności za wykonywanie określonych zadań. Przykładem może być odpowiedzialność polegająca na tworzeniu instancji obiektów.

Martin Fowler w swoim artykule „Inversion of Control Containers and the Dependency Injection pattern” zwraca uwagę na fakt, iż termin Inversion of Control jest bardzo ogólny i wprowadza inny - Dependency Injection – Wstrzykiwanie Zależności, który bardziej pasuje do opisu działania kontenerów IoC [69].

Wstrzykiwanie zależności to dostarczenie do klasy A, konkretnych zależności D1, D2 … DN, których ona wymaga. Ten proces odbywa się zazwyczaj na jeden z dwóch sposobów:

* Przez konstruktor
* Przez właściwości

Przyjęło się, że pierwsza opcja jest stosowana w przypadku zależności wymaganych, a druga opcjonalnych. Przykład ręcznego wstrzykiwania zależności przez konstruktor w JavaScript prezentuje Rys. 5.14.

var logger = new Logger();

var connectionProvider = new ConnectionProvider({

    logger: logger,

    config: app.config

});

var service1 = new Service1({

    repository1: new Repository1({ connectionProvider: connectionProvider }),

    repository2: new Repository2({ connectionProvider: connectionProvider }),

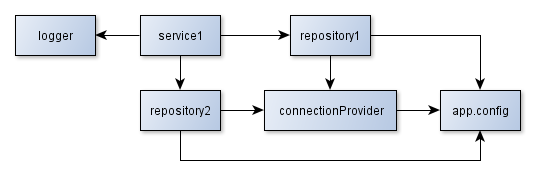
    logger: logger

});

Rys. . - Ręczne budowanie grafu obiektów

Klasa Service1, nie wie z jakich konkretnie implementacji repository1, repository2 i loggera korzysta. Istotne jest jedynie to, aby były one zamiennikami w myśl Zasady podstawienia Liskov opisanej w rozdziale 3. Podobna sytuacja ma miejsce w przypadku tworzenia repository1 i repository2.

Zależności obiektów, widocznych na powyższym przykładzie ilustruje graf skierowany widoczny na Rys. 5.15. Nawet przy tak prostych zależnościach utworzenie całego grafu jest problematyczne.



Rys. . - Prosty graf zależności

Alternatywą jest powierzenie czynności instancjonowania obiektów wraz zależnościami specjalnemu obiektowi, który zrobi to automatycznie, czyli kontenerowi IoC.

Utworzenie pokazanego wyżej grafu obiektów z użyciem kontenera prezentuje Rys. 5.16.

Repository1.\_\_dependsOn = ["connectionProvider"];

...

Repository2.\_\_dependsOn = ["connectionProvider"];

...

ConnectionProvider.\_\_dependsOn = ["config", "logger"];

...

Service1.\_\_dependsOn = ["repository1", "repository2", "logger"];  
...

var kernel = new Kernel();

kernel.bind("service1", Service1);

kernel.bind("connectionProvider", ConnectionProvider);

kernel.bind("repository1", Repository1);

kernel.bind("repository2", Repository2);

kernel.bind("logger", Logger);

kernel.bindFunction("config", function () { return app.config; });

var service1 = kernel.get("service1");

Rys. . - Automatyczne budowanie grafu obiektów

W językach takich jak C# i Java implementację w kontenerze wiąże się zazwyczaj z odpowiednim interfejsem. W JavaScript nie ma interfejsów, więc zależności są rozpoznawane po nazwach. Konstruktor może posiadać pole \_\_dependsOn, będące tablicą z listą nazw zależności.

Następnie każda nazwa wiązana jest z konkretną implementacją za pomocą metody bind() obiektu kernel. W celu uzyskania instancji obiektu powiązanego z daną nazwą, należy użyć metody get().

Wybór kontenerów IoC w języku JavaScript jest bardzo skromny, gdyż nie jest to wzorzec popularny w tym środowisku. Z konieczności, na potrzeby aplikacji został napisany własny kod realizujący tą funkcjonalność.

Użyty w projekcie algorytm budowania instancji prezentuje się następująco:

function zbuduj\_instancję(nazwa){

var nazwyZależności = pobierzNazwyWszystkichZależnościDla(nazwa)

var zależności = {}

dla każdej nazwy zależności:

zbuduj jej instancję rekurencyjnie za pomocą zbuduj\_instancję()

uzyskaną instancję przypisz do słownika zależności

var konstruktor = pobierzKonstruktorDla(nazwa)

return new konstruktor(zależności)

}

#### Dostęp do bazy danych

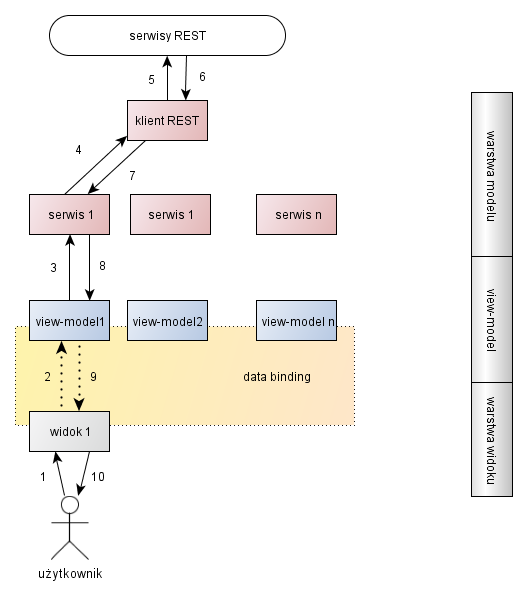
W celu zapewnienia dostępu do bazy danych, użyty został sterownik MongoDB Node.JS Driver 1.1.6. Oferuje on składnię bardzo zbliżoną do tej, której używa się bezpośrednio do zarządzania bazą danych za pośrednictwem konsoli administracyjnej. Dzięki temu w aplikacji było możliwe pominięcie bardziej skomplikowanych metod dostępu do bazy jak np. maper obiektowo dokumentowy.

### Aplikacja kliencka

Drugą częścią omawianego systemu jest klient, będący aplikacją mobilną.

#### Architektura niskiego poziomu

Podobnie jak część serwerowa, aplikacja kliencka nie może obyć się bez przemyślanej architektury logicznej. Rys. 5.5 przedstawia przepływ obsługi zdarzenia przez wszystkie warstwy aplikacji klienckiej.



Rys. . - Przebieg obsługi zdarzenia przez wszystkie warstwy aplikacji klienckiej

1. Użytkownik dokonuje akcji, np. klika na przycisk znajdujący się na widoku1.
2. Za pomocą mechanizmu data-bindingu view-model1 jest powiadamiany o tej akcji i wywołane zostaje odpowiednie zdarzenie.
3. View-model1 odwołuje się do serwisu1, który jest częścią modelu, w celu uzyskania danych, dzięki którym będzie mógł zaktualizować swój stan.
4. Serwis1 używa klienta REST żeby uzyskać potrzebne mu do działania dane.
5. Klient REST woła odpowiednie serwis.
6. Serwis zwraca żądane dane.
7. Klient REST przetwarza dane i zwraca je do serwisu1.
8. Serwis1 dokonuje operacji na danych i rezultat zwraca do view-modelu1.
9. View-model1 aktualizuje swój stan np. zmienia wartość jakiejś właściwości. Dzięki mechanizmowi data-bindingu widok1 jest aktualizowany automatycznie.
10. Zmiany w widoku1 trafiają do użytkownika.

#### Proces wytwarzania aplikacji klienckiej

Największym problemem, z którym trzeba się zmierzyć wytwarzając aplikacje w technologii PhoneGap jest brak możliwości podłączenia debuggera do aplikacji działającej w przeglądarce urządzenia mobilnego bądź emulatora.

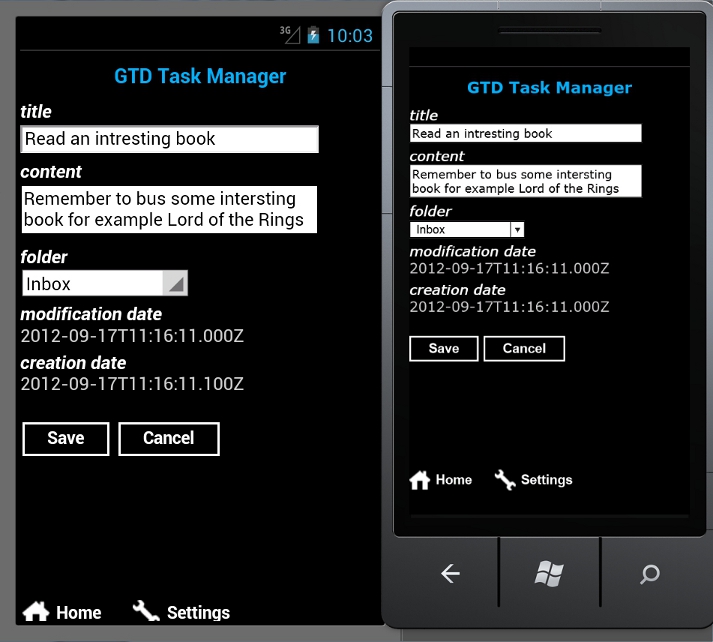
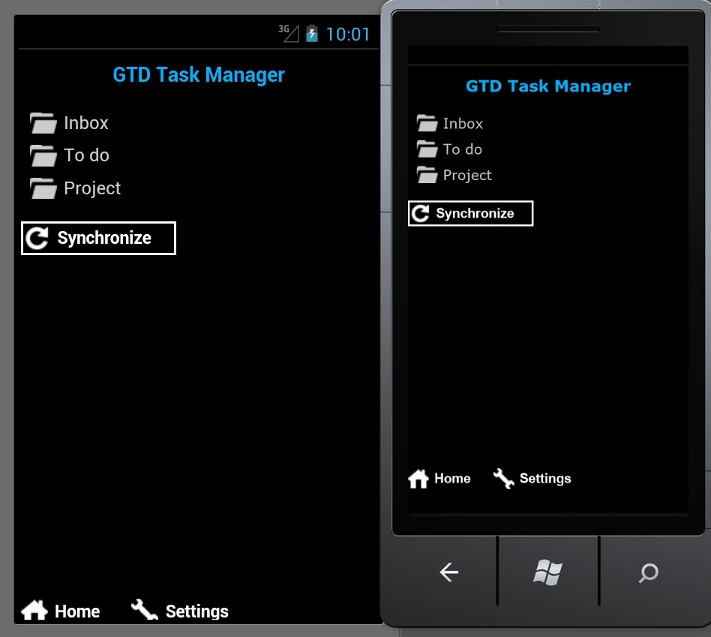
Częściowym rozwiązaniem tego problemu okazało się użycie biblioteki phonegap-desktop.js. Pozwala ona na uruchomienie aplikacji PhoneGap bezpośrednio w przeglądarce. Dzięki jej zastosowaniu bieżące testowanie aplikacji odbywało się w Google Chrome (biblioteka miała problemy z działaniem pod innymi przeglądarkami), a tylko raz na jakiś czas program był uruchamiany na emulatorze. Testy jednostkowe uruchamiane były w Internet Explorerze 9, czyli środowisku najbardziej zbliżonym do tego, na którym miała działać aplikacja.

## Rezultat prac

Jako rezultat prac powstał nieskomplikowany system do zarządzania zadaniami w metodyce GTD. Składa się on z dwóch części aplikacji mobilnej, wykonanej w technologii PhoneGap i serwerowej, wykonanej w technologii Node.js. Aplikacja mobilna może działać w trybie offline i potrafi zsynchronizować listę zadań, nawet w przypadku edytowania jej z dwóch różnych klientów.

Dzięki wykonaniu przy użyciu technologii PhoneGap, aplikacja może zostać uruchomiona na większości dostępnych platform mobilnych takich jak Android, iOS, Blackberry, WebOS czy Symbian.

Dla testu została ona uruchomiona poza WP7 również na platformie Android, co odbyło się bez problemów. Jedyną potrzebną do tego czynnością, było przeniesienie plików JavaScript i HTML do projektu Javowego, stworzonego w środowisku Eclipse. Rys. 5.22 przedstawia zrzuty ekranu aplikacji mobilnej prezentujące jej interfejs na emulatorze WP7 in Androida.



Rys. . - Zrzut ekranu z aplikacji mobilnej działającej na emulatorze Androida (po lewej) i WP7 (po prawej)

## Badanie jakości kodu

Zgodnie z założonym w poprzednim rozdziale planem, kod zaimplementowanej aplikacji został poddany analizie w celu zbadania jego jakości. Poniżej przedstawiono wyniki badań oraz opis procesu ich uzyskiwania.

### Wyniki pomiarów ilości naruszeń zasad SOLID

Mierzenie ilości naruszeń zasad SOLID zostało wykonane manualnie. Jako, że wspomniane zasady nie są formalnymi metrykami, to nie istnieją żadne narzędzia pozwalające wykonać takie pomiary automatycznie dla kodu w języku JavaScript. Wyniki pomiarów przedstawiono w dwóch tabelach widocznych poniżej.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Plik | Klasa | Naruszenia SRP | Naruszenia OCP | Naruszenia DIP |
| \application.js |  |  |  |  |
|  | Application | 0 | 0 | 0 |
| \common\app-error.js |  |  |  |  |
|  | AppError | 0 | 0 | 0 |
| \common\ioc-container.js |  |  |  |  |
|  | Kernel | 0 | 0 | 0 |
| \controllers\change-controller.js |  |  |  |  |
|  | ChangeController | 0 | 0 | 0 |
| \controllers\home-controller.js |  |  |  |  |
|  | HomeController | 0 | 0 | 0 |
| \controllers\task-controller.js |  |  |  |  |
|  | TaskController | 0 | 0 | 0 |
| \controllers\user-controller.js |  |  |  |  |
|  | UserController | 0 | 0 |  |
| \repositories\connection-provider.js |  |  |  |  |
|  | ConnectionProvider | 0 | 0 | 0 |
| \repositories\task-repository.js |  |  |  |  |
|  | TaskRepository | 0 | 0 | 0 |
| \repositories\user-repository.js |  |  |  |  |
|  | UserRepository | 0 | 0 | 0 |
| \routing\authentication-provider.js |  |  |  |  |
|  | AuthenticationProvider | 0 | 0 | 0 |
| \routing\middleware-factory.js |  |  |  |  |
|  | MiddlewareFactory | 0 | 0 | 0 |
| \routing\router.js |  |  |  |  |
|  | Router | 0 | 0 | 0 |
| \routing\routes-configurator.js |  |  |  |  |
|  | RoutesConfigurator | 0 | 1 | 0 |
| \services\events-dispatcher.js |  |  |  |  |
|  | EventsDispatcher | 0 | 1 | 0 |
| \services\task-service.js |  |  |  |  |
|  | TaskService | 0 | 0 | 0 |
| \services\user-service.js |  |  |  |  |
|  | UserService | 0 | 0 | 0 |

Tabela . - Naruszenia SOLID w części serwerowej

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Plik | Klasa | Naruszenia SRP | Naruszenia OCP | Naruszenia DIP |
| \infrastructure\application.js |  |  |  |  |
|  | application | 1 | 0 | 3 |
| \infrastructure\router.js |  |  |  |  |
|  | router | 1 | 0 | 1 |
| \infrastructure\view-management.js |  |  |  |  |
|  | view\_management | 0 | 0 | 0 |
| \services\task-service.js |  |  |  |  |
|  | TaskService | 0 | 0 | 0 |
| \services\user-service.js |  |  |  |  |
|  | UserService | 0 | 0 | 0 |
| \services\user-settings-store.js |  |  |  |  |
|  | UserSettingsStore |  | 1 | 0 |
| \utils\Object.js |  |  |  |  |
|  | Object | 0 | 0 | 0 |
| \utils\rest-api-client.js |  |  |  |  |
|  | RestApiClient | 0 | 0 | 0 |
| \utils\workbook.js |  |  |  |  |
|  | Workbook | 1 | 0 | 0 |
|  | WorkbookBuilder | 0 | 0 | 0 |
| \view-model\folders.js |  |  |  |  |
|  | FoldersViewModel | 0 | 0 | 0 |
| \view-model\home.js |  |  |  |  |
|  | HomeViewModel | 0 | 0 | 0 |
| \view-model\main.js |  |  |  |  |
|  | MainViewModel | 0 | 0 | 0 |
| \view-model\register.js |  |  |  |  |
|  | RegisterViewModel | 0 | 0 | 0 |
| \view-model\settings.js |  |  |  |  |
|  | SettingsViewModel | 0 | 0 | 0 |
|  | CredentialsViewModel | 0 | 0 | 0 |
| \view-model\tasks.js |  |  |  |  |
|  | TasksViewModel | 0 | 0 | 0 |

Tabela . - Naruszenia SOLID w części klienckiej

Sumaryczna ilość naruszeń SOLID została zaprezentowana w Tabela 5.3. Naruszenia SRP dotyczą głównie kodu związanego z rozruchem aplikacji. Naruszenia OCP w części serwerowej występują głównie w kodzie wiążącym ścieżki bądź nazwy z konkretnymi metodami. W praktyce nie są to zbyt duże problemy, gdyż dotyczą klas, które mają być modyfikowane. Naruszenia DIP występują głównie klasie inicjalizującej aplikację, która polega na szczegółach niskiego poziomu zamiast na abstrakcjach. Ze względu na specyfikę kodu inicjalizującego nie wydaje się to być istotnym problemem.

|  |  |  |  |
| --- | --- | --- | --- |
|  | SRP | OCP | DIP |
| Część serwerowa | 0 | 2 | 0 |
| Część kliencka | 3 | 1 | 4 |

Tabela . - Sumaryczna ilość naruszeń zasad SOLID

### Wyniki pomiarów wartości metryk jakości kodu

Pomiary wybranych metryk jakości kodu zostały dokonane po części automatycznie, a po części manualnie.

Metryka CC została policzona przy użyciu programu yardstick napisanego w JavaScript i uruchamianego pod node.js. Radzi sobie on najlepiej spośród wszystkich narzędzi z liczeniem złożoności cyklomatycznej, gdyż potrafi uwzględniać również złożoność zagnieżdżonych anonimowych funkcji, które są przekazywane jako wywołania zwrotne [70].

Metryka CBO została policzona ręcznie. Jej wartość pomija odwołania do obiektów wbudowanych w język JavaScript.

Poziom pokrycia został zmierzony za pomocą narzędzia JSCover. Działa ono na zasadzie instrumentacji kodu źródłowego i dokonania pomiarów w czasie wykonywania testów. Mierzy pokrycie instrukcji [71].

Wyniki przeprowadzonych pomiarów widoczne są w tabelach poniżej.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Plik | Klasa | Metoda | CC | CBO | Poziom pokrycia |
| \application.js |  |  |  |  | 0% |
|  | Application |  |  | 2 |  |
|  |  | start | 3 |  |  |
|  |  | wireDependencies | 3 |  |  |
|  |  | bindDependencies | 2 |  |  |
| \common\app-error.js |  |  |  |  | 93% |
|  | AppError |  |  | 0 |  |
|  |  | notFoundError | 1 |  |  |
|  |  | isNotFoundError | 1 |  |  |
| \common\ioc-container.js |  |  |  |  | 95% |
|  | Kernel |  |  | 1 |  |
|  |  | bind | 1 |  |  |
|  |  | bindFunction | 1 |  |  |
|  |  | get | 1 |  |  |
|  |  | regiterService | 1 |  |  |
|  |  | build | 5 |  |  |
|  |  | contructorActivationFactory | 1 |  |  |
|  |  | functionActivationFactory | 1 |  |  |
| \controllers\change-controller.js |  |  |  |  | 100% |
|  | ChangeController |  |  | 1 |  |
|  |  | post | 3 |  |  |
| \controllers\home-controller.js |  |  |  |  | 76% |
|  | HomeController |  |  | 1 |  |
|  |  | index | 1 |  |  |
|  |  | test | 1 |  |  |
|  |  | restricted | 1 |  |  |
| \controllers\task-controller.js |  |  |  |  | 100% |
|  | TaskController |  |  | 1 |  |
|  |  | get | 3 |  |  |
| \controllers\user-controller.js |  |  |  |  | 86% |
|  | UserController |  |  | 1 |  |
|  |  | get | 3 |  |  |
|  |  | post | 3 |  |  |
| \repositories\connection-provider.js |  |  |  |  | 89% |
|  | ConnectionProvider |  |  | 2 |  |
|  |  | buildDbObject | 2 |  |  |
|  |  | open | 5 |  |  |
|  |  | db | 1 |  |  |
|  |  | close | 1 |  |  |
| \repositories\task-repository.js |  |  |  |  | 100% |
|  | TaskRepository |  |  | 2 |  |
|  |  | getUserTasks | 3 |  |  |
|  |  | createTask | 2 |  |  |
|  |  | saveTask | 2 |  |  |
|  |  | deleteTask | 2 |  |  |
| \repositories\user-repository.js |  |  |  |  | 100% |
|  | UserRepository |  |  | 1 |  |
|  |  | getUserByName | 3 |  |  |
|  |  | createUser | 3 |  |  |
| \routing\authentication-provider.js |  |  |  |  | 100% |
|  | AuthenticationProvider |  |  | 1 |  |
|  |  | validate | 3 |  |  |
| \routing\filters-builder.js |  |  |  |  | 100% |
|  |  | buildFilters | 1 |  |  |
| \routing\middleware-factory.js |  |  |  |  |  |
|  | MiddlewareFactory |  |  | 6 |  |
|  |  | buildExpressApp | 1 | 3 |  |
| \routing\router.js |  |  |  |  | 0% |
|  | Router |  |  | 3 |  |
|  |  | start | 1 |  |  |
| \routing\routes-configurator.js |  |  |  |  | 89% |
|  | RoutesConfigurator |  |  | 4 |  |
|  |  | configureRoutes | 6 |  |  |
| \services\events-dispatcher.js |  |  |  |  | 100% |
|  | EventsDispatcher |  |  | 2 |  |
|  |  | subscribe | 1 |  |  |
| \services\task-service.js |  |  |  |  | 91% |
|  | TaskService |  |  | 2 |  |
|  |  | getUserTasks | 5 |  |  |
|  |  | applyTaskChanges | 11 |  |  |
| \services\user-service.js |  |  |  |  | 93% |
|  | UserService |  |  | 1 |  |
|  |  | getUserByName | 4 |  |  |
|  |  | createUser | 2 |  |  |

Tabela . - Metryki kodu w części serwerowej

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Plik | Klasa | Metoda | CC | CBO | Poziom pokrycia |
| \infrastructure\application.js |  |  |  |  | 0% |
|  | application |  |  | 2 |  |
|  |  | init | 3 |  |  |
|  |  | userSettingsStore | 1 |  |  |
|  |  | saveWorkbook | 1 |  |  |
|  |  | backButton | 1 |  |  |
|  |  | pause | 1 |  |  |
|  |  | close | 1 |  |  |
|  |  | setupKernelAndBindings | 3 |  |  |
| \infrastructure\router.js |  |  |  |  | 43% |
|  | router |  |  | 2 |  |
|  |  | start | 1 |  |  |
|  |  | navigate | 1 |  |  |
|  |  | setupRoutes | 2 |  |  |
|  |  | setupViewResolving | 4 |  |  |
|  |  | getPhoneGapPath | 1 |  |  |
|  |  | goToPartialView | 2 |  |  |
|  |  | startSammy | 1 |  |  |
| \infrastructure\view-management.js |  |  |  |  | 0% |
|  | view\_management |  |  | 0 |  |
|  |  | initialize | 1 |  |  |
| \services\task-service.js |  |  |  |  | 100% |
|  | TaskService |  |  | 3 |  |
|  |  | getUserTasks | 2 |  |  |
|  |  | sendChanges | 3 |  |  |
|  |  | synchronize | 3 |  |  |
|  |  | sendChanges | 2 |  |  |
|  |  | getUserTasks | 1 |  |  |
| \services\user-service.js |  |  |  |  | 100% |
|  | UserService |  |  | 1 |  |
|  |  | areCredentialsCorrect | 3 |  |  |
|  |  | register | 3 |  |  |
| \services\user-settings-store.js |  |  |  |  | 92% |
|  | UserSettingsStore |  |  | 1 |  |
|  |  | setCredentials | 1 |  |  |
|  |  | getCredentials | 1 |  |  |
|  |  | saveToStorage | 2 |  |  |
|  |  | loadFromStorage | 3 |  |  |
|  |  | getFolders | 1 |  |  |
| \utils\Object.js |  |  |  |  | 100% |
|  | Object |  |  | 0 |  |
|  |  | shallowClone | 1 |  |  |
| \utils\rest-api-client.js |  |  |  |  | 100% |
|  | RestApiClient |  |  | 1 |  |
|  |  | call | 2 |  |  |
|  |  | addCredentialsToUrlIfNeeded | 2 |  |  |
|  |  | addCredentialsToDataIfNeeded | 3 |  |  |
|  |  | setCredentials | 1 |  |  |
| \utils\workbook.js |  |  |  |  | 98% |
|  | Workbook |  |  | 1 |  |
|  |  | buildFoldersFromTasks | 4 |  |  |
|  |  | saveToStorage | 2 |  |  |
|  |  | saveNewTask | 1 |  |  |
|  |  | saveExistingTask | 2 |  |  |
|  |  | saveExistingTransientTask | 3 |  |  |
|  |  | saveExistingPersitentTask | 4 |  |  |
|  |  | replaceTask | 1 |  |  |
|  |  | deleteTask | 2 |  |  |
|  |  | deleteTransientTask | 1 |  |  |
|  |  | deleteFromTasksCollection | 1 |  |  |
|  |  | deleteAssociatedChangeItem | 3 |  |  |
|  |  | deletePersistentTask | 5 |  |  |
|  |  | findAssociatedChangeAndSetItToDelete | 1 |  |  |
|  |  | getTasksFromFolder | 2 |  |  |
|  | WorkbookBuilder |  |  | 2 |  |
|  |  | newFromRestTasks | 3 |  |  |
|  |  | rebuildTasks | 2 |  |  |
|  |  | loadFromStorage | 7 |  |  |
|  |  | mapTaskToDataItem | 3 |  |  |
| \view-model\folders.js |  |  |  |  | 95% |
|  | FoldersViewModel |  |  | 2 |  |
|  |  | unload | 1 |  |  |
|  |  | synchronize | 3 |  |  |
|  |  | goToFolder | 1 |  |  |
| \view-model\home.js |  |  |  |  | 100% |
|  | HomeViewModel |  |  | 1 |  |
|  |  | goToDebug | 1 |  |  |
|  |  | goToSettings | 1 |  |  |
|  |  | goToFolders | 1 |  |  |
| \view-model\main.js |  |  |  |  | 100% |
|  | MainViewModel |  |  | 1 |  |
|  |  | goToHome | 1 |  |  |
|  |  | goToSettings | 1 |  |  |
| \view-model\register.js |  |  |  |  | 100% |
|  | RegisterViewModel |  |  | 2 |  |
|  |  | goToSettings | 1 |  |  |
|  |  | register | 3 |  |  |
| \view-model\settings.js |  |  |  |  | 100% |
|  | SettingsViewModel |  |  | 4 |  |
|  | CredentialsViewModel |  |  | 3 |  |
|  |  | testCredentials | 3 |  |  |
|  |  | goToRegister | 1 |  |  |
|  |  | saveCredentials | 1 |  |  |
| \view-model\tasks.js |  |  |  |  | 99% |
|  | TasksViewModel |  |  | 3 |  |
|  |  | loadTasksForCurrentFolder | 1 |  |  |
|  |  | viewDetails | 1 |  |  |
|  |  | setCurrent | 3 |  |  |
|  |  | editDetails | 1 |  |  |
|  |  | cancel | 1 |  |  |
|  |  | createNew | 1 |  |  |
|  |  | saveDetails | 3 |  |  |
|  |  | deleteTask | 1 |  |  |
|  |  | goToFolders | 1 |  |  |
|  |  | editingEnabled | 1 |  |  |

Tabela . - Metryki kodu w części klienckiej

Widoczna niżej Tabela 5.6 pokazuje średnie i maksymalne wartości badanych metryk. Najbardziej skomplikowana metoda występuje w części serwerowej i ma złożoność cyklomatyczną o wartości 11. Zgodnie z badaniami przedstawionymi w rozdziale trzecim, taka złożoność nie stanowi dużego zagrożenia ze względu na możliwość występowania błędów.

Maksymalna wartość CBO to 6. Jest ona daleka od przekroczenia bezpiecznej wartości – 14.

Średni poziom pokrycia instrukcji w części serwerowej wynosi 83%, a w aplikacji klienckiej 82%. Jest on zaniżony przez kilka klas, które celowo nie były testowane, jak np. te inicjalizujące aplikację. Większość klas ma pokrycie zbliżone do 100%, co jest wynikiem oczekiwanym, gdyż kod był pisany zgodnie z metodyką TDD.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | CC | CBO | Poziom pokrycia |
| Część serwerowa | Maksimum | 11 | 6 | - |
| Średnia | - | 1,89 | 83% |
| Część kliencka | Maksimum | 7 | 4 | - |
| Średnia | - | 1,71 | 82% |

Tabela . - Średnie i maksymalne wartości metryk jakości kodu

### Wyniki badania subiektywnej oceny jakości kodu

O przejrzenie kodu źródłowego poproszono czterech programistów. Każdy z nich miał co najmniej dwa lata doświadczenia zawodowego w wytwarzaniu oprogramowania. Tabela 4.1 zawiera wyniki badań. Jak widać ogólna subiektywna ocena utworzonego kodu jest wysoka.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lp. | 1 | 2 | 3 | 4 | Średnia |
| Ocena | 4 | 4 | 4,5 | 4,5 | 4,25 |

Tabela . – Wyniki badania subiektywnej oceny jakości kodu

# Podsumowanie

W rozdziale 5 pokazano, że możliwe jest wykonanie aplikacji JavaScript cechującej się kodem wysokiej jakości. Jako, że omawiany język jest obiektowy to dobre praktyki takie jak SOLID również mają tutaj zastosowanie.

W praktyce największy wpływ na wytwarzany kod miała metodyka TDD. Klasy posiadające zbyt dużo zależności, bądź naruszające zasadę SRP stawały się uciążliwe w testowaniu, dlatego były sukcesywnie eliminowane w procesie refactoringu zgodnie z zasadą red-green-refactor. Ponadto testowanie pozwoliło eliminować literówki i ułatwiało proces ulepszania kodu, oferując automatyczne testy regresyjne, dzięki którym można było dokonywać modyfikacji bez obawy wprowadzenia błędów do aplikacji.

Niestety jak okazało się podczas wytwarzania aplikacji testy jednostkowe nie zastępują całkowicie statycznego typowania sprawdzanego przez kompilator. Jeśli klasa A, zależy od klasy B, ale w testach klasy A używane są mocki klasy B, to po zmianie interfejsu (metod i pól) klasy B i poprawieniu testów tejże klasy, w testach klasy A nie wyjdzie na jaw iż korzysta ona ze starego interfejsu, gdyż zamiast oryginalnej klasy B używany jest jej mock. W językach takich jak C# i Java ten problem nie występuje, gdyż mamy tam do czynienia z sprecyzowanymi w czasie kompilacji interfejsami, dzięki czemu mock ma zawsze taką samą strukturę jak klasa którą zastępuje. W JavaScript omawiane zjawisko utrudniało refactoring i propagację zmian po całej aplikacji.

Doświadczenia nabyte implementacji projektu, pozwalają również wyciągnąć kilka wniosków na temat aplikacji hybrydowych PhoneGap. Przede wszystkim wytwarzanie ich jest trudniejsze niż natywnych głównie z powodu braku możliwości debuggowania na urządzeniu bądź emulatorze. Ponadto języki takie jak C# czy Java zawierają mniej błędów projektowych niż JavaScript dzięki czemu programista nie musi ich omijać.

Uruchomienie aplikacji PhoneGap pod różnymi systemami operacyjnymi nie stanowi problemu. Ma to miejsce głównie dzięki temu, że większość kodu jest wykonywana w przeglądarce internetowej, a te dostępne na współczesnych urządzeniach mobilnych zazwyczaj bardzo dobrze radzą sobie z wykonywaniem kodu omawianego języka.

Przez kilka najbliższych lat JavaScript, jako język umożliwiający tworzenie dwóch obecnie najważniejszych rodzajów aplikacji czyli aplikacji mobilnych i webowych, będzie zyskiwał coraz większe znaczenie. Poziom skomplikowania kodu JavaScript będzie stale rósł, dlatego dbanie o jakość i stosowanie właściwych praktyk programowania będzie coraz bardziej istotne.

W przyszłości JavaScript może zostać zastąpiony przez inne rozwiązania jak na przykład języki transkompilowane [72] takie jak Dart czy CoffeScript . W takim przypadku zacząłby on pełnić rolę „assemblera Internetu” [73] i stał się zdecydowanie mniej istotny dla większości programistów.

# Bibliografia

|  |  |
| --- | --- |
| [1] | D. Crockford, „youtube,” 2007. http://youtu.be/v2ifWcnQs6M?t=3m55s. [Data uzyskania dostępu: 2012 sierpień 29]. |
| [2] | J. R. Mandle, „Wayback Machine,” http://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html. [Data uzyskania dostępu: 29 sierpień 2012]. |
| [3] | wikipedia, „Duck typing,” http://en.wikipedia.org/wiki/Duck\_typing. [Data uzyskania dostępu: 1 Wrzesień 2012]. |
| [4] | wikipedia, „Purely functional,” http://en.wikipedia.org/wiki/Purely\_functional. [Data uzyskania dostępu: 2 Wrzesień 2012]. |
| [5] | Ecama international, „ECMAScript Language Specification,” Ecma International, Czerwiec 2012. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. [Data uzyskania dostępu: 2012 Wrzesień 2012]. |
| [6] | Oracle, „What is Class?,” Oracle, http://docs.oracle.com/javase/tutorial/java/concepts/class.html. [Data uzyskania dostępu: 5 Wrzesień 2012]. |
| [7] | BenB, „Mozilla Developer Network,” 5 Marzec 2012. https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\_Objects/Object/proto. [Data uzyskania dostępu: 6 Wrzesień 2012]. |
| [8] | „TIOBE Programming Community Index,” TIOBE, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. [Data uzyskania dostępu: 2012 Wrzesień 1]. |
| [9] | „The Transparent Language Popularity Index,” http://lang-index.sourceforge.net/. [Data uzyskania dostępu: 1 Wrzesień 2012]. |
| [10] | K. Finley, „5 Ways to Tell Which Programming Languages are Most Popular,” 5 Czerwiec 2012. http://www.readwriteweb.com/hack/2012/06/5-ways-to-tell-which-programming-lanugages-are-most-popular.php. [Data uzyskania dostępu: 1 Wrzesień 2012]. |
| [11] | J. Brockmeier, „C#, Objective-C and JavaScript Move Up in TIOBE Index,” 2011 Grudzień 9. http://www.readwriteweb.com/cloud/2011/12/c-objective-c-and-javascript-m.php. [Data uzyskania dostępu: 2012 Wrzesień 1]. |
| [12] | K. Finley, „Github Has Surpassed Sourceforge and Google Code in Popularity,” 2011 Czerwiec 2. http://www.readwriteweb.com/hack/2011/06/github-has-passed-sourceforge.php. [Data uzyskania dostępu: 1 Wrzesień 2012]. |
| [13] | GitHub, „GitHub,” https://github.com/languages. [Data uzyskania dostępu: 1 Wrzesień 2012]. |
| [14] | M. Hendrickson, „State of the Computer Book Market, part 4: The Languages,” O'Reilly, 6 Kwiecień 2012. http://radar.oreilly.com/2012/04/computer-book-market-2011-part4.html. [Data uzyskania dostępu: 2012 Wrzesień 1]. |
| [15] | Enterprise Irregulars, „IT Jobs Recovery Continues, Picks Up Steam,” 10 Maj 2012. http://www.enterpriseirregulars.com/35176/it-jobs-recovery-continues-picks-up-steam/. [Data uzyskania dostępu: 2012 Wrzesień 1]. |
| [16] | Gartner, Inc., „Gartner Says Worldwide Sales of Mobile Phones Declined 2.3 Percent in Second Quarter of 2012,” Gartner, Inc., Sierpień 2012. http://www.gartner.com/it/page.jsp?id=2120015. [Data uzyskania dostępu: 2 Wrzesień 2012]. |
| [17] | „iOS,” wikipedia, http://en.wikipedia.org/wiki/Ios. [Data uzyskania dostępu: 2012 Wrzesień 5]. |
| [18] | „android.com,” http://source.android.com/source/licenses.html. [Data uzyskania dostępu: 2 Wrzesień 2012]. |
| [19] | „Windows Phone 7,” wikipedia, http://en.wikipedia.org/wiki/Windows\_Phone\_7. [Data uzyskania dostępu: 10 Wrzesień 2012]. |
| [20] | J. Angel, „Windows Phone 7 supports VB.Net and F#,” 25 Sierpień 2010. http://justinangel.net/WindowsPhone7VBnetAndFSharp. [Data uzyskania dostępu: 2 Wrzesień 2012]. |
| [21] | „Symbian,” wikipedia, http://en.wikipedia.org/wiki/Symbian. [Data uzyskania dostępu: 10 Wrzesień 2012]. |
| [22] | „Bada OS,” wikipedia, http://en.wikipedia.org/wiki/Bada\_OS. [Data uzyskania dostępu: 10 Wrzesień 2012]. |
| [23] | „BlackBerry OS,” wikipedia, http://en.wikipedia.org/wiki/BlackBerry\_OS. [Data uzyskania dostępu: 10 Wrzesień 2012]. |
| [24] | LetsGoMo, „Vikas Banga,” 19 Wrzesień 2010. http://letsgomo.com/mobile-web/estimates-of-mobile-phones-and-smartphones-supporting-html5/. [Data uzyskania dostępu: 2 Wrzesień 2012]. |
| [25] | „Touch Events version 1,” The World Wide Web Consortium, 15 Grudzień 2011. http://www.w3.org/TR/touch-events/. [Data uzyskania dostępu: 2012 Wrzesień 2]. |
| [26] | R. Dahl, „Ryan Dahl - History of Node.js,” 5 Październik 2011. http://youtu.be/SAc0vQCC6UQ. [Data uzyskania dostępu: 2 Wrzesień 2012]. |
| [27] | R. Dahl, „ Ryan Dahl: Introduction to Node.js,” 22 Wrzesień 2011. http://youtu.be/M-sc73Y-zQA. [Data uzyskania dostępu: 2 Wrzesień 2012]. |
| [28] | „WinRT,” wikipedia, http://en.wikipedia.org/wiki/WinRT. [Data uzyskania dostępu: 10 Wrzesień 2012]. |
| [29] | „JavaScript,” wikipedia, http://en.wikipedia.org/wiki/JavaScript. [Data uzyskania dostępu: 10 Wrzesień 2012]. |
| [30] | wikipedia, „Software quality,” http://en.wikipedia.org/wiki/Software\_quality. [Data uzyskania dostępu: 3 Wrzesień 2012]. |
| [31] | wikipedia, „Lehman's laws of software evolution,” http://en.wikipedia.org/wiki/Lehman%27s\_laws\_of\_software\_evolution. [Data uzyskania dostępu: 3 Wrzesień 2012]. |
| [32] | Galorath, „Accurately Estimate Your Software Maintenance Costs,” Galorath, http://www.galorath.com/index.php/software\_maintenance\_cost. [Data uzyskania dostępu: 3 Wrzesień 2012]. |
| [33] | Omnext, „How to save on software maintenance costs,” Omnext, Marzec 2010. http://www.sec.ee/docs/howtosave.pdf. [Data uzyskania dostępu: 3 Wrzesień 2012]. |
| [34] | wikipedia, „Technical debt,” http://en.wikipedia.org/wiki/Technical\_debt. [Data uzyskania dostępu: 2 Wrzesień 2012]. |
| [35] | J. Atwood, „New Programming Jargon,” 20 Lipiec 2012. http://www.codinghorror.com/blog/2012/07/new-programming-jargon.html. [Data uzyskania dostępu: 2012 Wrzesień 4]. |
| [36] | R. C. Martin, w *Clean Code*, Upper Saddle River, Prentice Hall, 2009, pp. 31-52. |
| [37] | R. C. Martin, „Clean Code,” Upper Saddle River, Prentice Hall, 2009, pp. 53-55. |
| [38] | R. C. Martin, „Getting a SOLID start,” 2009 Luty 13. http://blog.objectmentor.com/articles/2009/02/12/getting-a-solid-start. [Data uzyskania dostępu: 2012 Wrzesień 2012]. |
| [39] | R. C. Martin, „The Principles of OOD,” http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. [Data uzyskania dostępu: 5 Wrzesień 2012]. |
| [40] | wikipedia, „Open/closed principle,” http://en.wikipedia.org/wiki/Open/closed\_principle. [Data uzyskania dostępu: 5 Wrzesień 2012]. |
| [41] | D. Wampler, „ The Liskov Substitution Principle for "Duck-Typed" Languages,” 7 Wrzesień 2008. http://blog.objectmentor.com/articles/2008/09/06/the-liskov-substitution-principle-for-duck-typed-languages. [Data uzyskania dostępu: Wrzesień 5 2012]. |
| [42] | R. C. Martin, „The Interface Segregation Principle,” http://www.objectmentor.com/resources/articles/isp.pdf. [Data uzyskania dostępu: 5 Wrzesień 2012]. |
| [43] | „System integration testing,” http://en.wikipedia.org/wiki/System\_integration\_testing. [Data uzyskania dostępu: 6 Wrzesień 2012]. |
| [44] | wikipedia, „http://en.wikipedia.org/wiki/Unit\_testing,” http://en.wikipedia.org/wiki/Unit\_testing. [Data uzyskania dostępu: 6 Wrzesień 2012]. |
| [45] | J. Palermo, „Guidelines for Test-Driven Development,” Maj 2006. http://msdn.microsoft.com/en-us/library/aa730844%28v=vs.80%29.aspx. [Data uzyskania dostępu: 6 Wrzesień 2012]. |
| [46] | M. Fowler i B. Venners, „ Test-Driven Development,” 2 Grudzień 2002. http://www.artima.com/intv/testdriven2.html. [Data uzyskania dostępu: 21 Wrzesień 2012]. |
| [47] | „Software peer review,” wikipedia, http://en.wikipedia.org/wiki/Software\_peer\_review. [Data uzyskania dostępu: 20 Wrzesień 2012]. |
| [48] | Atlassian, „Defining your Workflow,” Atlassian, 12 Sierpień 2012. https://confluence.atlassian.com/display/CRUCIBLE/Defining+your+Workflow. [Data uzyskania dostępu: 20 Wrzesień 2012]. |
| [49] | A. Glover, „In pursuit of code quality: Monitoring cyclomatic complexity,” 28 Marzec 2006. http://www.ibm.com/developerworks/java/library/j-cq03316/. [Data uzyskania dostępu: 3 Wrzesień 2012]. |
| [50] | Enerjy, „McCabe Cyclomatic Complexity: the proof in the pudding,” Enerjy, http://www.enerjy.com/blog/?p=198. [Data uzyskania dostępu: 3 Wrzesień 2012]. |
| [51] | M. R. C., Clean Code, Upper Saddle River: Prentice Hall, 2009, pp. 140-141. |
| [52] | B. Walter, „Metryki obiektowe jako wskaźniki jakości kodu i projektu”. |
| [53] | Aivosto, „Cohesion metrics,” Aivosto, http://www.aivosto.com/project/help/pm-oo-ck.html. [Data uzyskania dostępu: 3 Wrzesień 2012]. |
| [54] | Aivosto, „Chidamber & Kemerer object-oriented metrics suite,” Aivosto, http://www.aivosto.com/project/help/pm-oo-ck.html. [Data uzyskania dostępu: 3 Wrzesień 2012]. |
| [55] | wikipedia, „Code coverage,” http://en.wikipedia.org/wiki/Code\_coverage. [Data uzyskania dostępu: 2012 Wrzesień 25]. |
| [56] | P. zbiorowa, „Słownik wyrażeń związanych z testowaniem,” http://www.testerzy.pl/images/baza\_wiedzy/artykuly/slownik-terminow-testowych-2-0-ver-0-99.pdf. [Data uzyskania dostępu: 2012 Wrzesień 28]. |
| [57] | B. Marick, „How to Misuse Code Coverage,” http://www.exampler.com/testing-com/writings/coverage.pdf. [Data uzyskania dostępu: 2012 Wrzesień 28]. |
| [58] | J. F. Smart, „Code coverage metrics and Functional Test Coverage,” 4 Maj 2011. http://weblogs.java.net/blog/johnsmart/archive/2011/05/04/code-coverage-metrics-and-functional-test-coverage. [Data uzyskania dostępu: 2012 Wrzesień 28]. |
| [59] | Bullseye Testing Technology, „Minimum Acceptable Code Coverage,” Bullseye Testing Technology, 2006. http://www.bullseye.com/minimum.html. [Data uzyskania dostępu: 2012 Wrzesień 28]. |
| [60] | D. Allen, „ David Allen: Getting Things Done,” 2 Styczeń 2008. http://youtu.be/Qo7vUdKTlhk. [Data uzyskania dostępu: 7 Wrzesień 2012]. |
| [61] | F. Heylighen i C. Vidal, „Getting Things Done: The Science behind Stress-Free Productivity,” Free University of Brussels, http://cogprints.org/6289/1/Heylighen-Vidal-GTD-Science.pdf. [Data uzyskania dostępu: 8 Wrzesień 2012]. |
| [62] | wikipedia, „Multitier architecture,” en.wikipedia.org/wiki/Multitier\_architecture. [Data uzyskania dostępu: 8 Wrzesień 2012]. |
| [63] | Brian, „PhoneGap, Cordova, and what’s in a name?,” 19 Marzec 2012. http://phonegap.com/2012/03/19/phonegap-cordova-and-what%E2%80%99s-in-a-name/. [Data uzyskania dostępu: 8 Wrzesień 2012]. |
| [64] | wikipedia, „Representational state transfer,” http://en.wikipedia.org/wiki/Representational\_state\_transfer. [Data uzyskania dostępu: 9 Wrzesień 2012]. |
| [65] | P. Stawicki, „CouchDb - bo dane to nie zawsze tabele,” 3 Wrzesień 2010. http://www.javaexpress.pl/article/show/CouchDB\_\_bo\_dane\_to\_nie\_zawsze\_tabele. [Data uzyskania dostępu: 10 Wrzesień 2912]. |
| [66] | „Why CouchDB?,” http://guide.couchdb.org/draft/why.html. [Data uzyskania dostępu: 10 Wrzesień 2012]. |
| [67] | wikipedia, „BSON,” http://en.wikipedia.org/wiki/BSON. [Data uzyskania dostępu: 10 Wrzesień 2012]. |
| [68] | 10gen, „Schema Design,” 10gen, http://www.mongodb.org/display/DOCS/Schema+Design. [Data uzyskania dostępu: 20 Wrzesień 2012]. |
| [69] | M. Fowler, „martinfowler.com,” 23 Styczeń 2004. http://martinfowler.com/articles/injection.html. [Data uzyskania dostępu: 18 Wrzesień 2012]. |
| [70] | J. Borg, „yardstick,” https://github.com/calmh/yardstick. [Data uzyskania dostępu: 2012 Wrzesień 15]. |
| [71] | siliconforks, „JSCoverage,” http://siliconforks.com/jscoverage/. [Data uzyskania dostępu: 2012 Wrzesień 25]. |
| [72] | wikipedia, „Source-to-source compiler,” http://en.wikipedia.org/wiki/Source-to-source\_compiler. [Data uzyskania dostępu: 20 Wrzesień 2012]. |
| [73] | S. Hanselman, „JavaScript is Assembly Language for the Web: Sematic Markup is Dead! Clean vs. Machine-coded HTML,” 6 Lipiec 2011. http://www.hanselman.com/blog/JavaScriptIsAssemblyLanguageForTheWebSematicMarkupIsDeadCleanVsMachinecodedHTML.aspx. [Data uzyskania dostępu: 20 Wrzesień 2012]. |
| [74] | Joyent, „Modules,” https://github.com/joyent/node/wiki/modules#wiki-web-frameworks-routers. [Data uzyskania dostępu: 18 Wrzesień 2012]. |
| [75] | K. Huggins, „barista,” http://kieran.github.com/barista/. [Data uzyskania dostępu: 18 Wrzesień 2012]. |
| [76] | A. Sellier, „journey,” https://github.com/cloudhead/journey. [Data uzyskania dostępu: 18 Wrzesień 2012]. |
| [77] | C. Condon, „ Connect router with syntactic sugar,” https://github.com/crcn/connect-router. [Data uzyskania dostępu: 18 Wrzesień 2012]. |
| [78] | „What is Ember.js?,” http://emberjs.com/documentation/. [Data uzyskania dostępu: 9 Wrzesień 2012]. |
| [79] | L. Richardson i S. Ruby, RESTful Web Services, wydanie pierwsze red., Sebastopol, California: O’Reilly Media, Inc., 2007, pp. 79-105. |
| [80] | wikipedia, „Encapsulation (object-oriented programming),” http://en.wikipedia.org/wiki/Encapsulation\_%28object-oriented\_programming%29. [Data uzyskania dostępu: 1 Wrzesień 2012]. |
| [81] | expressjs.com, „expressjs.com,” 18 Wrzesień 2012. http://expressjs.com/. |