Zasady programowania i wzorce projektowe

v. 1.0.0

Plan

1. Wstęp
2. KISS
3. YAGNI
4. DRY
5. SOLID
6. Module Pattern
7. Revealing Module Pattern
8. Factory Pattern
9. Model View Controler

Wstęp

Wstęp

O tworzeniu oprogramowania

Tworzenie oprogramowania jest sztuką. Byle jakie rozwiązanie stworzyć łatwo. Tylko po co? Chaotycznie napisane aplikacje i systemy są często udręką. Ciężko jest je rozwijać i utrzymywać.

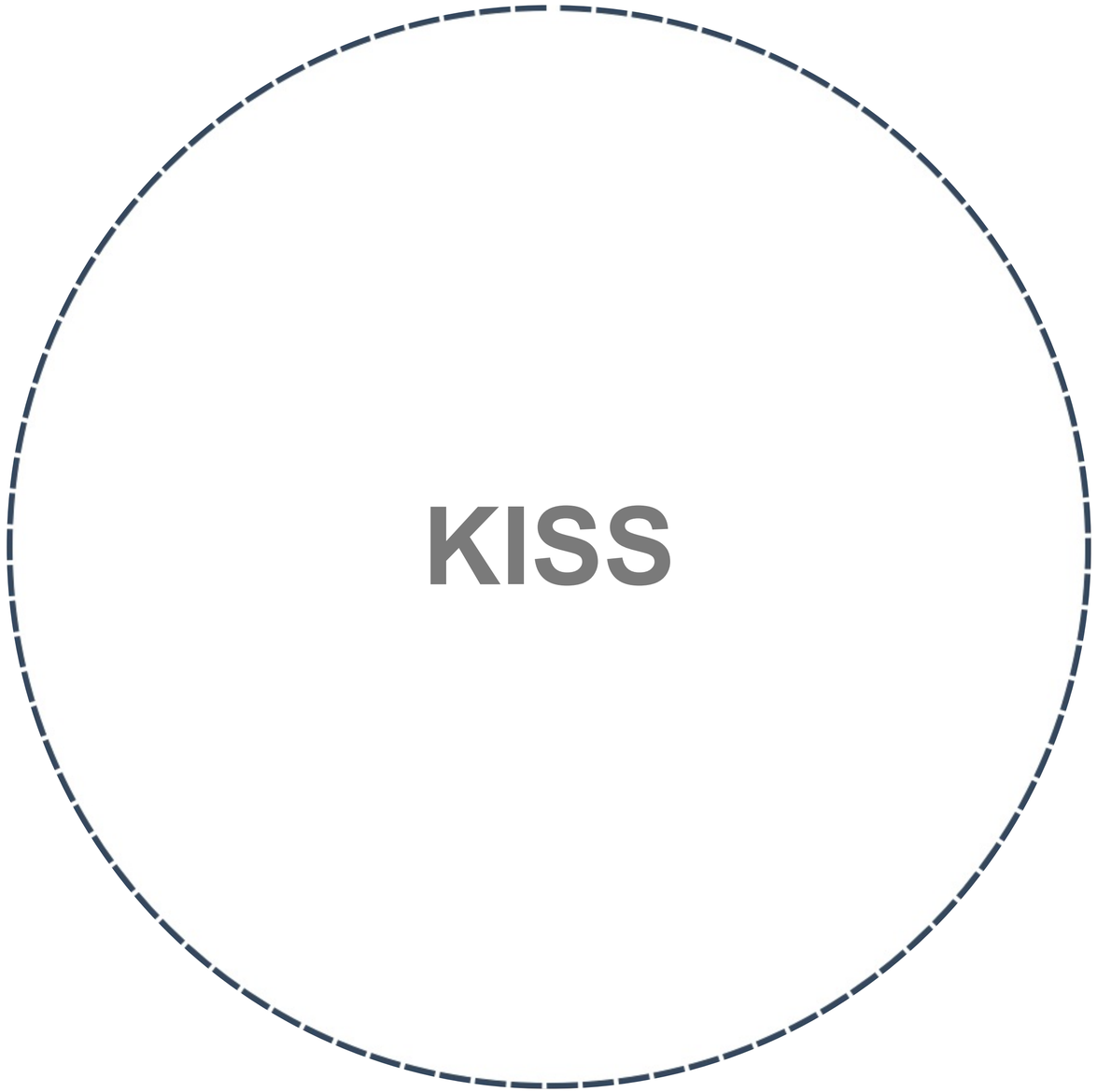
Warto zatem narzucić na projekt sztywne ramy. Zastosowanie odpowiednich ograniczeń i trzymanie się zasad pozwala uchronić się przed wieloma problemami często występującymi podczas pisania aplikacji.

Korzystaj z doświadczeń innych deweloperów i nie odkrywaj koła na nowo.

Treść tutorialu

- **zasady tworzenia oprogramowania pomagające uczynić kod czytelniejszym,**
- **wzorce projektowe pomagające rozwiązać potencjalne problemy.**

Zasady tworzenia oprogramowania


KISS

KISS

Keep It Simple, Stupid!

Jeśli piszemy prostą aplikację, której zadaniem jest np. dodawanie zadań do listy - to tworzenie kodu w oparciu o wszystkie lub najnowocześniejsze rozwiązania tworzy nadmierną abstrakcje i skomplikowanie.

Kod powinien być optymalny - a co za tym idzie - prosty. Dzięki temu mamy łatwy w zrozumieniu i utrzymaniu kod. Do tego służy zasada **KISS** - twórz kod jak najmniej skomplikowany, skrojony do konkretnych potrzeb.

Autorem zasady **KISS** jest inżynier lotnictwa – Kelly Johnson, który w latach 60. XX wieku zajmował się konstruowaniem samolotów. Podkreślał wtedy, że należy tworzyć samoloty, które da się naprawić jedynie dzięki najpotrzebniejszym narzędziom oraz stosunkowo niewielkiej wiedzy.

KISS

KISS a programowanie

Nowoczesne języki programowania umożliwiają pisanie rozbudowanych rozwiązań dla różnego rodzaju problemów. Czasami programiści mogą czuć potrzebę napisania mądrych abstrakcji, rozwiązujących każdy możliwy przypadek w aplikacji używając jak najnowszej składni.

Zasada KISS stwierdza, że rozwiązanie jest lepsze, gdy używa tylko tylu funkcji, abstrakcji, itp. ile jest faktycznie potrzebnych.

Rozwiązanie zgodne z zasadą KISS może wydawać się nudne, a nawet głupie, ale jest proste i zrozumiałe. Zasada **KISS stwierdza, że nie ma żadnej wartości w rozwiązaniu sprytnym, jest natomiast w rozwiązaniu prostym i czytelnym.**

Z drugiej strony osoba, która będzie kontynuować projekt napisany przez nas, będzie musiała poświęcić więcej czasu na zrozumienie działania tych mądrych rozwiązań.

KISS

Przykład w JavaScripcie

Funkcja **isFirstHigher** przyjmuje dwa parametry i sprawdza, czy pierwszy z nich jest większy od drugiego. Zasada KISS nie jest tu zastosowana.

Co możemy zmienić?

Kod

```
function isFirstHigher(a, b) {
  if (a > b) {
    return true;
  } else {
    return false;
  }
}
```

KISS

Dobre rozwiązanie

Funkcja jest niepotrzebnie skomplikowana – wystarczy zwrócić nasz warunek. JavaScript sprawdzi, czy pierwszy parametr jest większy, i zwróci wartość logiczną.

Kod

```
function isFirstHigher(a, b) {
  return a > b;
}
```

KISS

Inny przykład

Stosując KISS pamiętaj, że im mniej kodu tym lepiej. Używaj tylko tylu zmiennych i funkcji ilu potrzebujesz. Staraj się unikać tworzenia wielu funkcji, które nie dają potrzebnego efektu.

Kod

```
function multiply(a, b){
  var firstNumber = a;
  var secondNumber = b;
  return firstNumber * secondNumber;
}

function calc(first, second) {
  multiply(first, second)
}

calc(1, 2);
```

KISS

Inny przykład

Jak widzisz w w tym przypadku - uprościliśmy maksymalnie ilość zmiennych i abstrakcji. Kod jest dużo bardziej czytelny, a co za tym idzie - prostszy.

Kod

```
function multiply(a, b){
  return a * b;
}

multiply(1, 2);
```

Don't Repeat Yourself!

Don't Repeat Yourself!

Jedna z najważniejszych zasad

Nie powtarzaj kodu, gdyż zazwyczaj istnieje eleganckie rozwiązanie pozwalające ograniczyć liczbę duplikatów.

Warunki pisania czystego kodu

- Odnajdywanie powtórzeń w swoim kodzie.
- Eliminowanie powtórzeń dzięki odpowiedniej praktyce i właściwej abstrakcji.

Zasadę opisali Andy Hunt i Dave Thomas w książce Pragmatic Programmer. Jest to podstawa wielu innych dobrze znanych metod i wzorców projektowania oprogramowania.

Autorzy podkreślają, że zasada odnosi się nie tylko do kodu, lecz także do czynności wykonywanych przez programistów.

Don't Repeat Yourself!

Przykład w JavaScripcie

Tworzymy zmienne reprezentujące użytkowników. Do każdej zmiennej przypisujemy losową liczbę w zależności od numeru użytkownika.
Zasada DRY nie jest tu zastosowana.

Co możemy zrobić?

Kod

```
var user_1 =
  Math.round(Math.random() * 2 - 1);
var user_2 =
  Math.round(Math.random() * 3 - 2);
var user_3 =
  Math.round(Math.random() * 4 - 3);
//...
var user_100 =
  Math.round(Math.random() * 101 - 100);
```

Don't Repeat Yourself!

Napisz funkcję

Aby przestać powtarzać kod, możemy napisać funkcję zwracającą losową liczbę. Nadal jednak ręcznie tworzymy użytkowników.

Kod

```
function getRandomNumber(num) {
  return Math.round(
    Math.random() * (num + 1) - num
  );
}
var user_1 = getRandomNumber(1);
var user_2 = getRandomNumber(2);
var user_3 = getRandomNumber(3);
//...
var user_100 = getRandomNumber(100);
```

Don't Repeat Yourself!

Zwracaj do tablicy, napisz pętlę

Zmienne zastąpimy tablicą, w której przechowywać będziemy wartości związane z użytkownikami. Tworzenie nowych użytkowników możemy skrócić przy pomocy pętli.

Kod

```
function getRandomNumber(num) {
  return Math.round(
    Math.random() * (num + 1) - num
  );
}
var users = [];
for (var i = 0; i <= 100; i++) {
  users.push(getRandomNumber(i));
}
```

You Aren't Gonna Need It!

You Aren't Gonna Need It!

YAGNI

Wdrażaj to, co jest nam potrzebne – brzmi kolejna istotna zasada programowania.

Jest wiele powodów, dla których ta zasada istnieje. Dzięki niej skupiamy się na tym, co należy zrobić teraz. Maksymalizujemy wydajność programistów i zmniejszamy czas potrzebny do powstania systemu czy też aplikacji.

Nowe funkcjonalności są drogie, zarówno w rozwijaniu, jak i utrzymaniu, a użytkownicy muszą się ich uczyć. Nie należy zatem wprowadzać znacznych zmian, gdy nie są one potrzebne.

You Aren't Gonna Need It!

Przykład w jQuery

Mamy nałożone na przycisk nasłuchiwanie zdarzeń. Po kliknięciu dodajemy lub zabieramy klasę w elemencie div. Jest to jedyny kod, który ma mieć strona.

Co możemy zmienić?

Kod

```
$(function() {

  var button = $("button");
  var div = $("div");
  button.on("click", function(event) {
    div.toggleClass("myChange");
  })

});
```

You Aren't Gonna Need It!

Czysty JavaScript

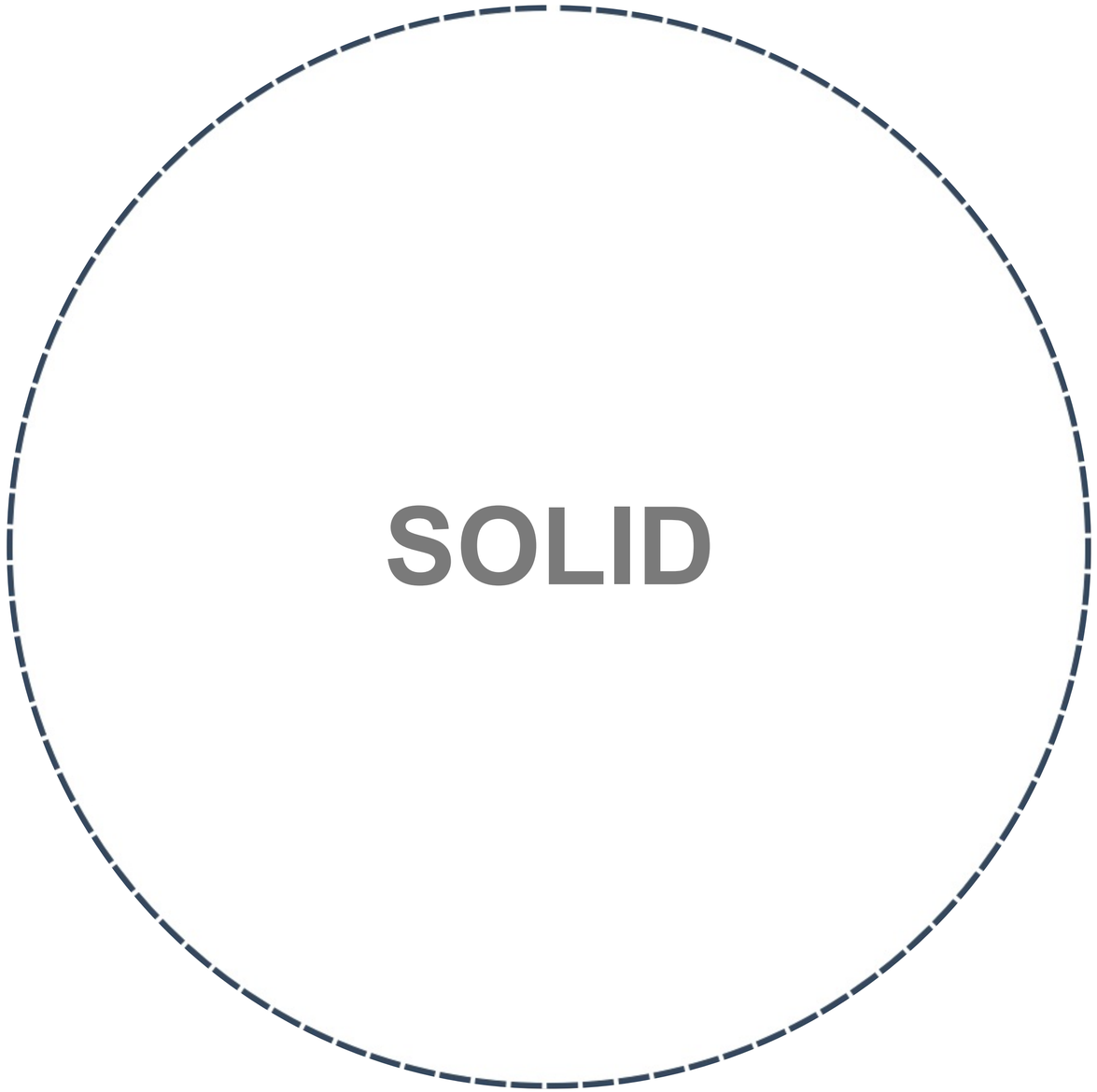
Nasz kod jest bardzo mały, nie potrzebujemy zatem jQuery. Po co ładować bibliotekę, którą wykorzystujemy tylko w jednym miejscu?

Wszystko możemy zrobić w czystym JavaScripcie.

Kod

```
document.addEventListener(
"DOMContentLoaded", function(){

  var button =
    document.querySelector("button");
  var div =
    document.querySelector("div");
  button
    .addEventListener("click",
      function(event) {
        div.classList
          .toggle("myChange");
    })
});
```


SOLID

SOLID

Zrozumiale i znacznie łatwiej

Czas na zbiór zasad odnoszący się do projektowania oprogramowania. Zostały one stworzone, aby uczynić oprogramowanie bardziej zrozumiałe, łatwiejsze w utrzymaniu i zarządzaniu.

Zasady te zostały rozpowszechnione przez Roberta C. Martina, który jest bardzo doświadczonym programistą, autorem wielu publikacji branżowych. Napisał m.in. książkę *Mistrz czystego kodu. Kodeks postępowania profesjonalnych programistów* (Warszawa 2013).

Zbiór zasad

- **Single Responsibility Principle**
- **Open/Closed Principle**
- **Liskov Substitution Principle**
- **Interface Segregation Principle**
- **Dependency Inversion Principle**

SOLID 1/5

Single Responsibility Principle – zasada pojedynczej odpowiedzialności

Wyobraźmy sobie sytuację - chcemy napisać kalkulator. Stworzenie jednej funkcji, którą nazwalibyśmy np. calc() miałaby wykonywać wszystkie obliczenia np. dodawanie i odejmowanie. Edycja takiego kodu byłaby bardzo czasochłonna - funkcja byłaby długa, zagmatwana i ciężka do zrozumienia.

Rozwiązanie stanowi **SOLID.** Funkcja powinna rozwiązywać tylko jedno zadanie - np. funkcja do dodawania powinna tylko dodawać itp.

Nie stosowanie tej zasady może to doprowadzić do sytuacji, w której tworzymy bardzo dużo funkcji mocno zależnych od innych funkcji. Zastosowanie zasady SOLID sprawia, że taki kod łatwiej jest utrzymywać oraz rozszerzać.

SOLID 1/5

Single Responsibility Principle – zasada pojedynczej odpowiedzialności

Kod po prawej stronie spełnia tę zasadę. Każda funkcja jest odpowiedzialna tylko za jedną rzecz - czyni to kod dużo czytelniejszy i łatwiejszy w zrozumieniu.

```
function add(a, b) {
        return a + b;
}

function divide(a, b) {
  if (b === 0) {
    return 'Nie mozna dzielić przez 0';
  }
  return a / b;
}
```

SOLID 2/5

Open/Closed Principle – zasada otwarte/zamknięte

Obiekty, funkcje i inne konstrukcje powinny być łatwe do rozszerzenia, ale uniemożliwiać modyfikacje.

Zasada ta sprawia że dużo prościej będzie utrzymać nasze klasy czy funkcje. Mniej pracy wymaga dodawanie nowych funkcjonalności, ponieważ nie musimy się martwić o to, że nasza funkcja będzie miała wpływ na inny kawałek kodu.

SOLID 2/5

Open/Closed Principle – zasada otwarte/zamknięte

Sytuacje demonstruje kod po prawej stronie. Funkcja zwraca tylko rzeczy, które sama przetworzyła. W związku z tym zmienne nie są dostępne poza funkcją.

```
sum() {
    var area = [];
    for (shape in this.shapes) {
      area.push(shape.area());
    }
    return area.reduce(
      function(pv, cv) {
        cv = cv + pv;
      }, 0
    )
  }
```

SOLID 3/5

Liskov Substitution Principle – zasada podstawiania Liskov

Obiekty w programie powinny być wymienne z instancjami ich podtypów bez zmiany poprawności tego programu.

Mówiąc prosto - funkcja powinna działać przewidywalnie bez względu na to czy jako parametr przekażemy klasę bazową czy też klasę, która po niej dziedziczy.

Jest to najtrudniejsza zasada do zrozumienia. Chodzi w niej o projektowanie klas w taki sposób, aby rozszerzały tylko klasę bazową bez niepotrzebnego nadpisywania właściwości, a nie nowo utworzone klasy na podstawie klasy abstrakcyjnej. Dzięki tej zasadzie ograniczamy liczbę niepotrzebnego kodu a ten, który już mamy, jest łatwiejszy w rozszerzaniu i utrzymaniu.

SOLID 3/5

Liskov Substitution Principle – zasada podstawiania Liskov

Wytłumaczenie na przykładzie

W matematyce kwadrat jest prostokątem - jest to zrozumiałe. Wyobraźmy sobie, że tworzymy klasę **Rectangle**, która posiada metody **setWidth()** i **setHeight()** ustawiające szerkość i wysokość danego prostokąta.

Na podstawie klasy **Rectangle** tworzymy klasę **Square**, czyli kwadrat. Ale kwadrat zawsze ma taką samą szerokość jak i wysokość, więc musimy nadpisać metody, w taki sposób, że **setWidth()** ustawia i szerokość i wysokość. Analogicznie działa **setHeight()**.

Według zasady podstawienia Liskov, w dużym uproszczeniu, klasy nadrzędne (z których dziedziczymy), możemy podmienić klasami stworzonymi na podstawie dziedziczenia bez zmiany działania programu. Wróćmy do naszego przykładu. Wyobraźmy sobie program, który tworzy wiele prostokątów i kwadratów. Według zasady podstawienia Liskov, jeśli wymienimy wszystkie wywołania klasy **Rectangle** na klasę **Square** - program powinien dalej działać. W naszym przypadku tak się nie stanie.

SOLID 3/5

Liskov Substitution Principle – zasada podstawiania Liskov

Przykład demonstrujący znajduje się po prawej stronie. Obiekt jest tworzony za pomocą funkcji Object.assign, a jego prototyp jest ustalany w zależności od innej funkcji. Daje to elastyczność tworzenia obiektów.

W przykładzie użyto metod getPrototypeOf oraz setPrototypeOf. Służą one do tworzenia dziedziczenia w javascripcie. Pierwsza pobiera prototyp z danego obiektu, a druga go nadaje. Więcej informacji znajdziesz na stronie MDN link

```
var prototypRodzica =
    Object.getPrototypeOf(
      areaCalculator()
  );
  // Objekt zależny od kontekstu
  var nowyObiekt = {}
  //ustanowienie dziedziczenia
  Object.setPrototypeOf(
    nowyObiekt, prototypRodzica
  );
  newObject.szerokość = '100px';
  return newObject;
```

SOLID 4/5

Interface Segregation Principle – zasada segregacji interfejsów

Klasa udostępnia tylko te interfejsy, które są niezbędne do zrealizowania konkretnej operacji.

Interfejs jest to kod, który jest udostępniany publicznie, dostępny dla innych funkcji. Funkcje prywatne nie znajdują się w środku return - a co za tym idzie - nie są widoczne poza funkcją.

Funkcje z zewnątrz nie powinny być zmuszane do polegania na metodach, których nie używają.

Chodzi w zasadzie o to, aby nie tworzyć obiektów mających metody nieprzydatne w danym projekcie. Javascript nie ma interfejsów. Możemy jednak tę zasadę zastosować do tego języka.

SOLID 4/5

Interface Segregation Principle – zasada segregacji interfejsów

```
class A {
  constructor(text) {
    this.text = text;
  }
  toUpper() {
    return this.text.toUpperCase();
  }
  toLower() {
    return this.text.toLowerCase();
  }
}
class B extends A {
  constructor(text) {
    super(text);
  }
}
```

SOLID 4/5

Interface Segregation Principle – zasada segregacji interfejsów

Popatrz na przykład z poprzedniego slajdu. Mamy dwie klasy **A** oraz **B**, gdzie **B** dziedziczy po **A**. Załóżmy, że w naszym kodzie korzystamy tylko z metody **toUpper()** na obiektach utworzonych przy pomocy klasy **B**. W takim przypadku klasa **B** niepotrzebnie dziedziczy metodę, z której nie korzysta.

SOLID 5/5

Dependency Inversion Principle – zasada odwracania zależności

Podmioty muszą polegać na abstrakcjach, a nie na konkretach.

Moduł wysokiego poziomu nie może polegać na module niskiego poziomu, ale powinnien on zależeć od abstrakcji.

W skrócie - nasz kod powinien być pisany w taki sposób, żeby inne części aplikacji mogły używać stworzonych przez nas abstrakcji.

Javascript nie ma również abstrakcji. Zasadę tę możemy jednak zastosować w przypadku funkcji wyższego rzędu. Z funkcjonalnego punktu możemy używać naszych abstrakcji, które będą przekazane przez funkcje wyższego rzędu do obiektów potomnych.

SOLID 5/5

Dependency Inversion Principle – zasada odwracania zależności

Kod po prawej stronie ilustruje tę zasadę. Konkretna abstrakcja (zmienna abst) jest tworzona wewnątrz funkcji, w związku z tym jest ona tworzona w zależności od kontekstu. Dzięki temu uniezależniamy abstrakcję od miejsca jego położenia.

```
function(radius) {
  var abst = manageShapeInterface(
    () => basics.area()
  );
};
var cubo = function(length) {
  var abst = manageShapeInterface(
    () => basics.area() +
          complex.volume();
  );
};
```

Wszystkie zasady razem

Nie przesadzaj z zasadami

Wszystkie wymienione zasady do tej pory mają na celu pomóc programiście w tworzeniu lepszego kodu.

Nie zawsze jesteśmy w stanie zastosować wszystkich zasad lub mogą one się ze sobą gryźć. Nie ma w tym jednak nic złego. Całokształt programu zawsze będzie zależeć od wiedzy i doświadczenia danego architekta.

Wzorce projektowe

Wzorce projektowe

Czym one są?

Wzorce projektowe to sprawdzone, uniwersalne rozwiązania często występujących problemów w programowaniu. Pokazują one zależności między klasami i obiektami. Ułatwiają tworzenie, rozwijanie i utrzymywanie kodu źródłowego. Nie są to jednak bezpośrednie rozwiązania tylko ich opisy. Znajomość wzorców projektowych jest przydatną bronią w arsenale każdego programisty.

Wzorce projektowe w informatyce zostały oparte na wzorcach projektowych w architekturze. Pomysł ten został stworzony przez amerykańskiego architekta Christophera Alexandra. Z założenia miały ułatwiać konstruowanie przestrzeni, w których często występują podobne problemy. Pomysł nie przyjął się w architekturze, ale został zaakceptowany przez programistów.

Module Pattern

Module Pattern – wzorzec modułu

Problem

Pisząc aplikację bardzo często tworzymy jakieś uniwersalne komponenty lub części aplikacji np. slider. Żeby stworzyć potrzebny kod, musimy stworzyć kilka funkcji i zmiennych, których następnie będziemy używać. Stworzone przez nas abstrakcje będą potrzebne tylko do działania tego konkretnego kodu - a co za tym idzie, nie są one potrzebne w zasięgu globalnym naszego kodu.

Rozwiązanie

Tworząc obiekty w Javascripcie, możemy użyć zmiennych i metod w środku funkcji - przez co nie będą one widoczne na zewnątrz funkcji. Dzięki temu jesteśmy w stanie ochronić część naszej funkcjonalności przed użyciem z zewnątrz.

Wzorzec modułu zamyka prywatność, stan i organizację przy użyciu domknięć. Zapewnia w ten sposób łączenie publicznych i prywatnych metod i zmiennych. Dzięki temu wzorcowi widoczne są tylko wzkazane przez nas elementy, a cała reszta zostaje schowana w zamknięciu prywatnym.

Module Pattern – wzorzec modułu

Rozwiązanie c.d.

Daje to nam czyste rozwiązanie do ograniczanie dostępu do logiki naszego kodu, udostępniając tylko te funkcje, które chcemy wykorzystać w innych częściach naszej aplikacji.

Wzorzec modułu ma wielu autorów, w tym Richarda Cornforda, który zajał się tym tematem na początku XXIw. Później został on popularyzowany przez Douglasa Crockforda w jego wykładach na temat inżynierii oprogramowania.

Module Pattern

Przykład

```
var myModule = (function () {
  var mojaPrywatnaZmienna = 0; // Prywatna zmienna
  var mojaPrywatnaMetoda = function(foo) {
      console.log(foo);
  }; // Prywatna metoda
  return { // Zwracamy obiekt
    mojaPublicznaZmienna: "foo", // Publiczna zmienna
    mojaPublicznaMetoda: function(bar) { // Publiczna metoda
      mojaPrywatnaZmienna++; // Inkrementacja prywatnej zmiennej
      mojaPrywtnaMetoda(bar); // Wywołanie prywatnej metody
    }
  };
})();
```

Module Pattern – wzorzec modułu

Omówienie

W przykładzie na poprzednim slajdzie mamy zaimplementowany Wzorzec Modułu. Nasz kod jest zamknięty w **IIFE**. Na zewnątrz przy użyciu słowa kluczowego `return` zwracamy obiekt. W owym obiekcie znajdują się zmienne i metody publiczne czyli takie, do których mamy dostęp w innych miejscach w kodzie.

Nasze zmienne i metody publiczne mają jako jedyne mają dostęp do zmiennych i metody prywatnych poprzez **domknięcia**. Zmienne i metody prywatne są ukryte przed całym kodem - nie jesteśmy się do nich dostać, chyba, że przez metody i zmienne publiczne.

Module Pattern

Plusy

Czysta składnia, zrozumiała dla programistów specjalizujących się w innych językach programowania.

Wsparcie prywatnych danych. Części publiczne modułu mogą korzystać z części prywatnych, które są osłonięte przed resztą kodu.

Minusy

Dostęp do danych publicznych i prywatnych następuje w inny sposób.

Nie możemy dostać się do prywatnych danych, gdy dodamy nowe metody w dalszej części aplikacji.

Trudność pisania testów sprawdzające cześci prywatne modułu.

Revealing Module Pattern

Revealing Module Pattern

Omówienie

Jest to to lekko zmodyfikowana wersja wzorca modułu opisanego wyżej. Programista Christian Heilmann (https://christianheilmann.com/) był sfrustrowany faktem, że musiał powtarzać nazwę głównego obiektu, gdy chciał wywołać jedną publiczną metodę z innej lub uzyskać dostęp do zmiennych publicznych.

Różnica tkwi w tym, że w zwracanym obiekcie są tylko te metody, które chcemy udostępnić (wszystkie metody są prywatne, udostępniamy tylko te, które chcemy żeby były publiczne).

Revealing Module Pattern

```
var mojOdkrytyModul = (function  {
        var prywatnaZmienna = "Programmer",
            publicznaZmienna = "Hello!";
        function prywatnaFunkcja() {
            console.log("Name:" + prywatnaZmienna);
        }
        function publiczneUstawienie(strName) {
            publicznaZmienna = strName;
        }
        function publicznePobranie() {
            prywatnaFunkcja();
        }
        return { // Zwracamy wskaźniki na prywatne metody i właściwości
            pobranie: publicznePobranie,
            powitanie: publicznaZmienna,
            ustawienie: publiczneUstawienie
        };
})();
```

Revealing Module Pattern

Plusy

Zgodna składnia.

Większa czytelność kodu niż w przypadku wzorca modułu.

Minusy

Jeśli prywatna funkcja związana jest z publiczną, to publiczna funkcja nie może być nadpisana.

Nie możemy również nadpisać publicznych funkcji związanych z prywatnymi zmiennymi.

Factory Pattern

Factory Pattern – wzorzec fabryki

Problem

Jedną z zasad SOLID jest zasada odwracania zależności. Dla przypomnienia - kod powinien działać abstrakcyjnie i nie być zależny od konkretnej implementacji. Tworząc obiekty za pomocą słowa kluczowego new - tworzymy obiekt w konkretnej implementacji. Aby uniezależnić tworzenie obiektów od stworzenia jego w konkretnym miejscu stosujemy wzorzec fabryki.

Rozwiązanie

Żeby uniezależnić tworzenie obiektów w zależności od kontekstu - potrzebujemy użyć wzorca Fabryki. Przykładowo - chcemy napisać funkcję, która w zależności od parametru stworzy nam konkretny obiekt.

Podstawowym celem Wzorca Fabryki jest łatwość rozszerzenia. Wzorzec Fabryki często jest stosowany w aplikacjach, które zarządzają, utrzymują lub manipulują dużą ilością różnych obiektów mających jednocześnie wiele cech charakterystycznych wspólnych (czyli metody i właściwości).

Factory Pattern – wzorzec fabryki

Przykład

```
function Klient(imie) {
  this.imie = imie;
  this.typ = "klient";
}
function Admin(imie) {
  this.imie = imie;
  this.typ = "admin";
}
function Fabryka(imie, typ) {
    if (typ === "klient") {
        return new Klient(imie);
    } else (typ === "admin") {
        return new Admin(imie);
    }
}
```

Factory Pattern – wzorzec fabryki

Omówienie

W przykładzie na poprzednim slajdzie mamy zaimplementowany prosty Wzorzec Fabryki. Mając dużo konstruktorów tworzących różne lecz podobne obiekty, agregujemy tworzenie obiektów do jednego miejsca - funkcji, która w zależności od parametrów tworzy interesujący nas obiekt.

Factory Pattern – wzorzec fabryki

Plusy

Łatwość zarządzania kodem gdy musimy generować dużo obiektów podobnych do siebie.

Łatwość zarządania obiektami gdy są bardzo mocno złożone.

Minusy

Wzorzec użyty w złym momencie niepotrzebnie zwiększy złożoność naszego kodu.

Problemy z napisaniem testów, w szczególności gdy wszystkie konstruktory zostaną zamknięte w Fabryce.

Model View Controller

Model View Controller

Problem

Tworząc aplikacje warto trzymać się jakieś struktury dotyczącej tworzenia funkcji, plików czy folderów. Dzięki temu nasz kod zyska na czytelności i prostocie. Z drugiej strony osoby, które dołączą do pisanego przez nas kodu będą mogły w łatwy sposób dodawać nowe funkcjonalności. Wzorzec MVC pomaga własnie w organizacji naszych aplikacji.

Definicja

MVC to architektoniczny wzorzec oprogramowania zachęcający do organizacji aplikacji przez separację problemów.

Kontrolery są warstwą między modelami a widokami. Ich zadaniem jest zmiana stanu modeli, gdy użytkownik wejdzie w interakcję z widokiem. Tylko tyle i aż tyle.

Wzorzec wymusza na nas izolację danych biznesowych (Model) od interfejsu użytkownika (View) przy pomocy trzeciego komponentu (Controller), którego zadaniem jest zarządzanie logiką i danymi wprowadzonymi przez użytkownika.

Model View Controller

Model

Modele zarządzają danymi aplikacji. Nie są połączone w żaden sposób z interfejsem użytkownika czy też warstwą prezentacji. Modele reprezentują unikatowe formy danych, których potrzebuje nasza aplikacja.

Jak działa Model?

Kiedy zmienia się Model, wysyła o tym informację do jakiegoś obserwatora – w naszym przypadku kontrolera (Controller), a ten, jeśli trzeba, zmienia odpowiednio widok (View).

Model View Controller

Widok

Widoki są wizualną reprezentacją modeli, gdzie pokazujemy aktualny stan aplikacji. W JavaScripcie poprzez widok będziemy rozumieli tworzenie oraz i zarządzanie elementami drzewa DOM.

Jak działa Widok?

Widok obserwuje Model przy pomocy Kontrolera. Gdy model się zmienia, informacja przechodzi aż do Widoku, aby ten mógł odpowiednio zareagować. W widokach nie powinno być żadnej logiki aplikacji – służą one wyłącznie do wyświetlania informacji.

Użytkownicy powinni mieć możliwość interakcji z widokami (na przykład wpisywania tekstu). Gdy nastąpi taka sytuacja, nie komunikujemy się bezpośrednio z modelem. Wysyłamy informację do kontrolera, on zajmie się zmianą stanu odpowiedniego modelu.

Model View Controller

Kontroler

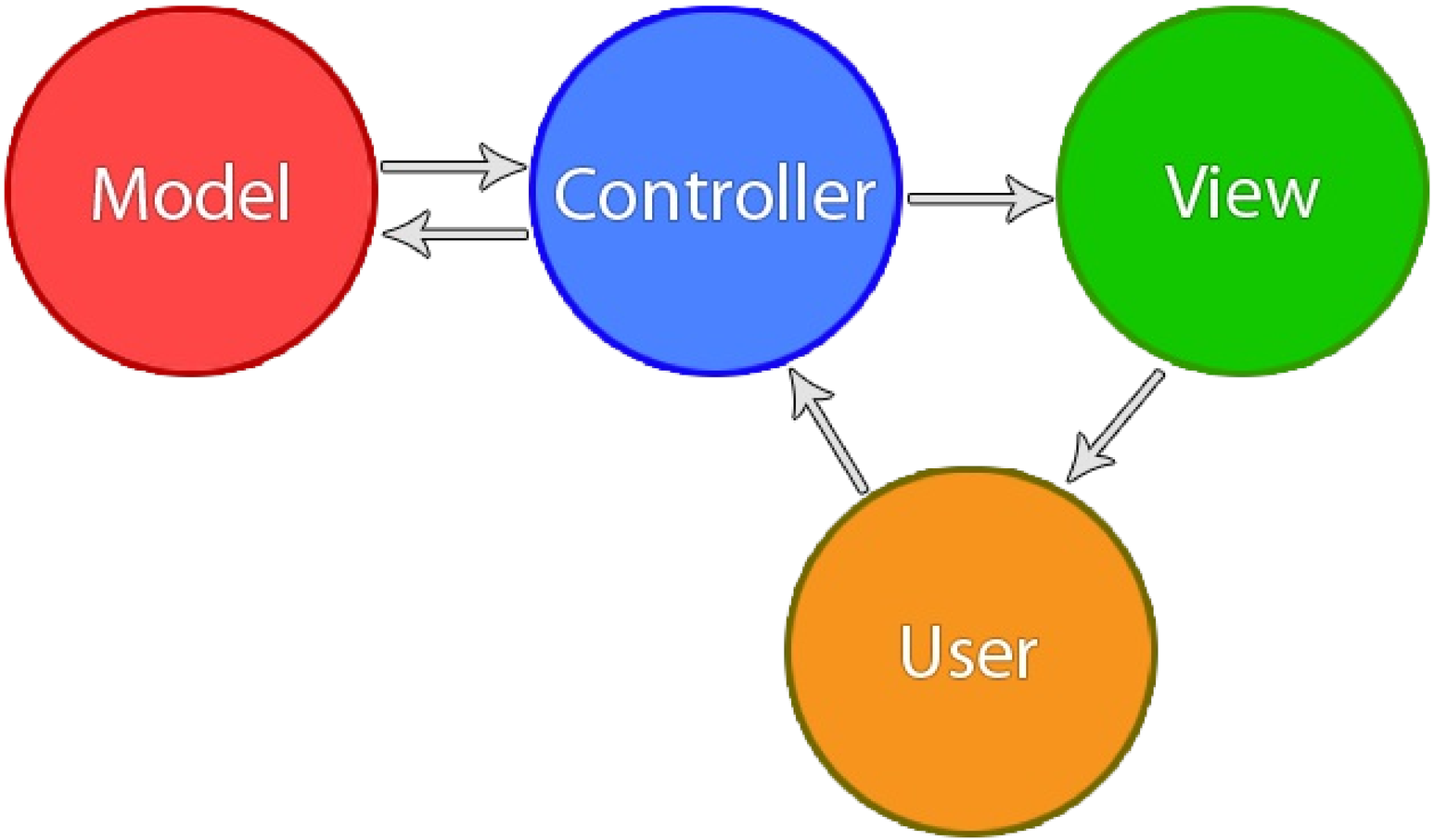
Kontrolery są sercem całego wzorca - ich zadaniem jest zarządzanie całą aplikacją na podstawie tego co znajduje się w Modelu i tego, co dzieje się postronie użytkownika.

Jak działa Kontroler?

Kontroler na podstawie danych w Modelu zarządza Widokiem. Gdy zmienia się Model, zmienia się automatycznie Widok.

Identycznie jest w drugą stronę - jeśli nastąpi jakaś interakcja z użytkownikiem, Kontroler wyciągnie z widoku odpowiednie dane, wykona swoją pracę i jeśli trzeba, to zmieni dane w Modelu. Jeśli tak się stanie - koło zatacza krąg - Kontroler zmienia Widok gdyż zmienił się Model.

Model View Controller



Źródło: http://lotsofprojects.com

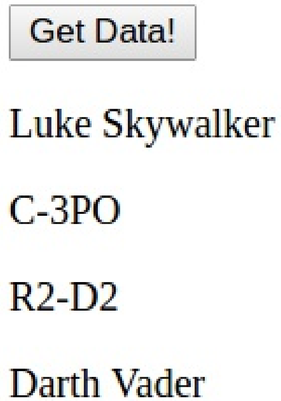
Model View Controller

MVC

Wyobraźmy sobie prostą stronę, na której znajduje się przycisk.

Przykład

Po naciśnięciu przycisku, łączymy się z zewnętrznym API (tutaj o Star Wars), pobieramy dane i je wyświetlamy.



Model View Controller

Omówienie

1. Kontroler sprawdza Model - nie ma danych, renderujemy przycisk.
2. Na przycisk jest nałożone nasłuchiwanie zdarzeń.
3. Użytkownik klika w przycisk.
4. Na kliknięcie zostanie wykonana odpowiednia metoda z Kontrolera.
5. Owa metoda z Kontrolera wywołuje metodę z Modelu, która ma za zadanie pobrać dane (np. Ajax).
6. Gdy Ajax zwróci dane, zmienia się Model.
7. Zmienia się Model, więc zostaje poinformowany o tym Kontroler.
8. Kontroler na zmianę Modelu renderuje nowy Widok.

Koniec