# Implementacja logiki biznesowej

## Projektowanie i programowanie systemów internetowych I

mgr inż. Krzysztof Rewak

9 kwietnia 2018

Wydział Nauk Technicznych i Ekonomicznych
Państwowa Wyższa Szkoła Zawodowa im. Witelona w Legnicy

# Plan prezentacji

1. Logika biznesowa
2. Narzędzia
3. Serwisy
4. Repozytoria
5. Wstrzykiwanie zależności
6. Zdarzenia
7. Podsumowanie

# Logika biznesowa

# Logika

Według SJP **logika** to nauka zajmująca się znajdowaniem praw rządzących ludzkim rozumowaniem, a także wnioskowaniem.

# Proces biznesowy

**Procesem biznesowym** jest natomiast seria zadań, których wykonanie w odpowiedniej kolejności doprowadzi do zrealizowania celu.

# Logika biznesowa

Wynikiem syntezy tych dwóch pojęć może być **logika biznesowa**.

Upraszczając, jest to sposób przeniesienia procesów biznesowych do programu komputerowego, ale zachowując wszystkie reguły domeny.

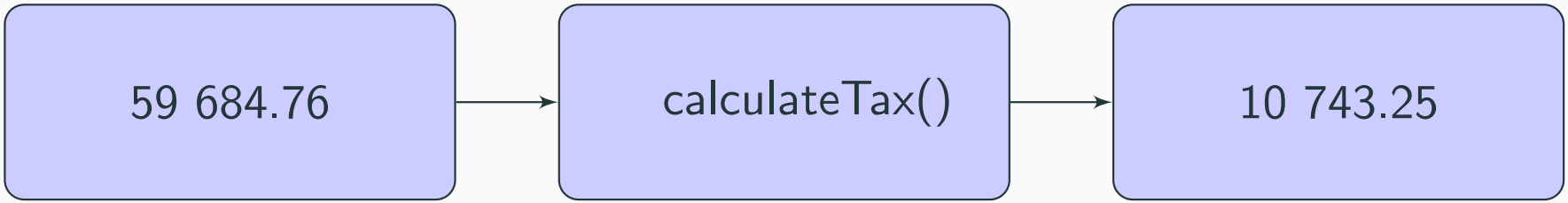
# Domena

Domeną nazywamy dziedzinę, w której przestrzeni będzie funkcjonowało nasze oprogramowanie.

System, który będzie wykorzystywany przez księgową małego przedsiębiorstwa, powinien implementować zasady rządzące księgowością małego przedsiębiorstwa. Trzeba mieć świadomość, że każda domena, choćby z wierzchu do siebie podobna, jest inna od pozostałych.

# Domena

Obliczanie podatku od dochodu mogłoby wydawać się proste, ale wszystko zależy od poznania domeny.




Jeżeli nasza wiedza kończy się na powyższym wykresie, szybko można założyć, że funkcja obliczająca wygląda następująco:

```
func calculateTax(income double) double {
  return income * 18 / 100;
}
```

# Domena

Wszystko okej, ale co w przypadku...

- przekroczenia produ podatkowego?
- rozliczania się z małżonkiem?
- rozliczania się w innym państwie?
- dziesiątek innych przypadków?

# Domena

Jednym z najczęstszych źródeł problemów przy programowaniu systemów internetowych (ale nie tylko!) jest brak zrozumienia domeny i procesów biznesowych, które mają zostać ujęte w tymże systemie.

# Domena

Dobrze zamodelowany i zaprojektowany proces to często ponad połowa wykonanej pracy.

# Modelowanie domeny

Jak poprawnie zamodelować proces biznesowy?

- na kartce i z ołówkiem lub na tablicy i z mazakiem,
- korzystając z bardziej wyrafinowanych metod takich jak *event storming*,
- lub w dowolny inny zrozumiały dla programistów sposób.

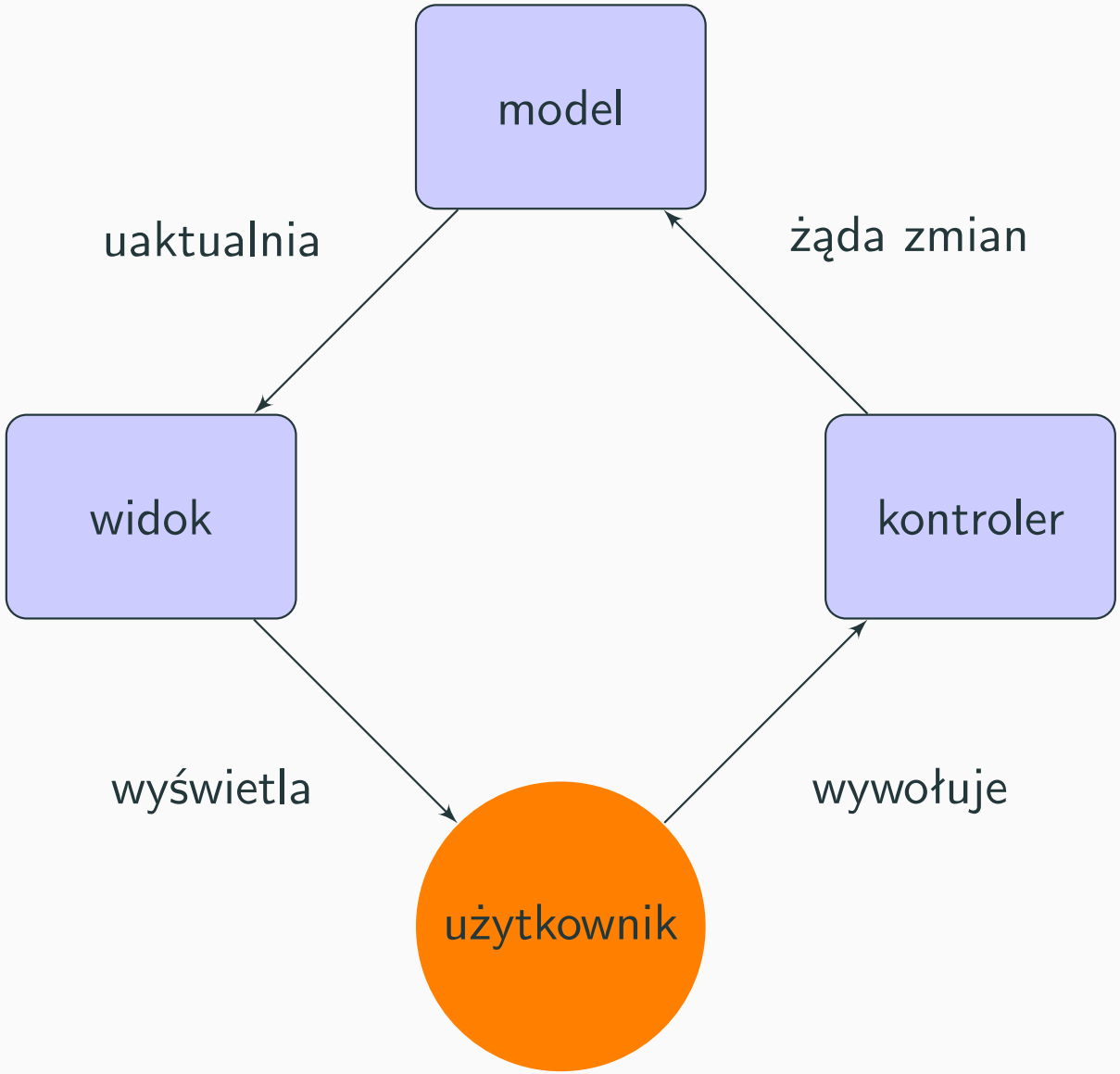
Często może się okazać, że dwóch profesjonalistów stworzy dwa różne modele tego samego procesu biznesowego. Czy to źle? Skądże!

# Narzędzia

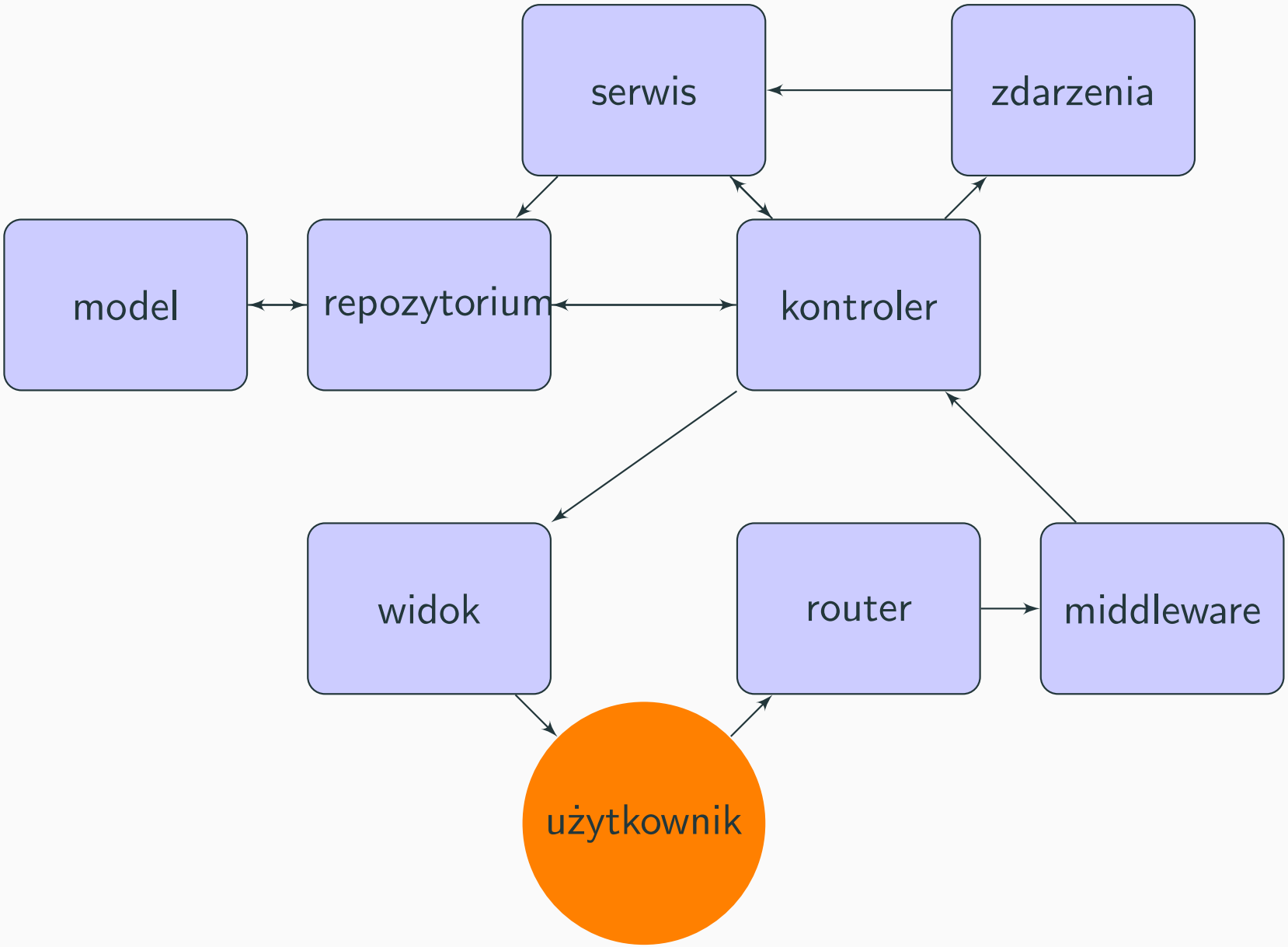
# Jak zaimplementować logikę biznesową?

Każdy problem wymaga indywidualnego podejścia. Czy to oznacza, że mamy wynajdować koło na nowo?

# MVC?


model
uaktualnia
żąda zmian
widok
kontroler
wyświetla
wywołuje
użytkownik

# A może MRVRMCES?



# Serwisy

# Serwisy

Spójrzmy na przykładowy kontroler `ArticleController`:

# Serwisy

```
class ArticleController (Controller)

  method updateArticle(Id: Int, Request: Request): Void
    @Article := Article::Get(Id)
    foreach @Tag in @Article.Tags
      if Array.in(@Tag.Id, Request.Get("tags"))
        if not @Tag.Selected
          @Article.Delete(@Tag)
        endif
      else
        if @Tag.Selected
          @Article.Add(@Tag)
        endif
      endif
    endforeach
    if @Article.Update(Request.Except("tags"))
      Mail(@Article.User.Email, "Article has been changed.")
    endif
  endmethod

endclass
```

# Serwisy

Może lepiej go nieco skrócić i przerzucić część logiki do innych klas?

```
class ArticleController (Controller)

  method updateArticle(Id: Int, Request: Request): void
    @Article := Article::Get(Id)

    @ArticleTagManager = new ArticleTagManager
    @ArticleTagManager.ManageTags(@Article, Request.Get("tags"))

    if @Article.Update(Request.Except("tags"))
      @MailService = new MailService
      @MailService.SetReceiver(@Article.User)
      @MailService.SetMessasge("Article has been changed.")
      @MailService.Send()
    endif
  endmethod

endclass
```

# Serwisy

Serwis służy przede wszystkim do odseparowania logiki.

# Serwisy

Dzięki wykorzystaniu serwisów można:

- zmniejszyć niepotrzebną redundancję w kodzie,
- wykorzystać tę samą funkcjonalność w wielu miejscach,
- realizować zasadę pojedynczej odpowiedzialności,
- wygodnie testować aplikację.

# Testowanie serwisów

O ileż łatwiej jest wywołać w teście klasę pojedynczego serwisu z konkretnymi danymi niż przebijać się przez cały kontroler i budować chociażby skomplikowane parametry z zapytania serwera?

# Testowanie serwisów

```
class PromoteUserToAdminService implements UserServiceInterface {

  protected $user;
  protected $grantor;

  public function setUser(User $user): self {
    $this->user = $user;
    return $this;
  }

  public function setGrantor(User $grantor): self {
    $this->grantor = $grantor;
    return $this;
  }

  public function run(): bool {
    $this->user->role = User::ADMIN;
    Log::info(/* message */);
    return $this->user->save();
  }

}
```

# Testowanie serwisów

```
class ExampleTest extends TestCase {

  public function userPromotionTest(): void {
    $user = User::create(["login" => "jsmith"]);
    $grantor = User::create([
      "login" => "admin",
      "role" => User::ADMIN,
     ]);

    $service = new PromoteUserToAdminService();
    $service->setUser($user);
    $service->setGrantor($grantor);

    $this->assertTrue($service->run());
  }

}
```

# Repozytoria

# Repozytoria

**Repozytorium** służy do oddzielenia logiki biznesowej od warstwy modelowania i mapowania danych.

# Repozytoria

Ideą stojącą za repozytoriami jest potrzeba ujednolicenia przetwarzania danych.

# Repozytoria

Wyobraźmy sobie kontroler wypisujący wszystkie produkty w sklepie internetowym. Czy z poziomu kontrolera powinniśmy wywoływać połączenie do bazy danych, pobierać zmapowane na modele informacje i przekazać dalej do widoku? A co w przypadku, gdy dane czasami będą pobierane z cache'a? Albo z pliku? Albo z jeszcze innego miejsca?

# Repozytoria

```
public interface Repository<T> {
    void create(T item);
    void update(T item);
    void delete(T item);
    List<T> get();
    T get(int id)
}
```

# Repozytoria

Repozytorium można oczywiście dostosować do własnych potrzeb, ale należy uważać, aby nie przekombinować!

Tzw. *overengineering* może doprowadzić do wielu problemów z zaciemnieniem kodu na czele.

# Wstrzykiwanie zależności

# Wstrzykiwanie zależności

Repozytoria i serwisy są wygodnymi narzędziami, jednak dodanie każdego nowego komponentu wymaga utworzenia bezpośrednich zależności między klasami. Okazuje się, że nie zawsze jest to dobrym rozwiązaniem.

# Wstrzykiwanie zależności

Z pomocą może przyjść wzorzec projektowy wstrzykiwania zależności (ang. *dependency injection*.

Oto kontroler, którego metoda wyświetla wszystkich zarejestrowanych użytkowników:

# Wstrzykiwanie zależności

```
public class UsersController : Controller
{

  public ActionResult All()
  {
    UserRepository repository = new UserRepository();
    var users = repository.ListAll();

    return View(users);
  }

}
```

# Wstrzykiwanie zależności

```
public class UsersController : Controller
{

  private readonly IUserRepository _repository;

  public UsersController(IUserRepository repository)
  {
    _repository = repository;
  }

  public ActionResult All()
  {
    var users = _repository.ListAll();
    return View(users);
  }

}
```

# Wstrzykiwanie zależności

Obiekty są w takim wypadku przekazywane jako już utworzone instancje, przez co architektura całej aplikacji jest bardziej elastyczna i modułowa.

Wstrzyknąć można - w zależności od języka - obiekt według klasy, klasy po dziedziczeniu lub po implementowanym interfejsie. Ta ostatnia metoda jest szczególnie warta wspomnienia, gdyż umożliwia pełne wykorzystanie wstrzykiwania zależności.

# Wstrzykiwanie zależności

Skąd aplikacja wie, co ma wstrzyknąć do której klasy? Można zarejestrować konkretne połączenia w odpowiednim miejscu:

```
public void ConfigureServices(IServiceCollection services)
{
  // (...)
  services.AddScoped<IUserRepository, UserRepository>();
}
```

Ale część frameworków (Laravel, Symfony) pozwalają na tzw. *auto-wiring* i automatyczne wstrzykiwanie zależności na podstawie przestrzeni nazw i nazw klas.

# Wstrzykiwanie zależności

DI pozwala na wygodne testowanie aplikacji.

Wyobraźmy sobie, że mamy repozytorium, które musi się podłączyć do bazy danych, pobrać rekordy, przetworzyć je i przekazać do testowanej przez nas klasy. Jeżeli testujemy inną funkcjonalność niż samo pobranie, możemy utworzyć tzw. *mock*, czyli zaślepkę z odpowiednio spreparowanymi danymi na czas testów.

# Zdarzenia

# Zdarzenia

Wiele nowoczesnych frameworków webowych pozwala na wykorzystywanie zdarzeń (*eventów*) lub sygnałów (*signals*).

Zdarzenie to proste powiadomienie o tym, że coś się ma wydarzyć. Odpowiedni nasłuchujący *listener* powinien takie zdarzenie odebrać, przetworzyć i uruchomić odpowiednie procedury.

# Zdarzenia

Wysłanie sygnału może wyglądać tak:

```
class CartController:

  # (...)

  def finish_order(self):
    # (...)
    place_order.send(sender=self.__class__, cart=self.cart)
```

... a odebranie następująco:

```
def place_order(sender, **kwargs):
  order = Order.objects.create()
  order.timestamp = datetime.datetime.now()
  # (...)
```

# Zdarzenia

Oczywiście nic nie stoi na przeszkodzie, aby przy realizacji wydarzenia wykorzystać na przykład uprzednio zaprojektowany serwis lub pobrać dane z repozytorium.

# Zdarzenia

Ponadto część frameworków oferuje wbudowane zdarzenia dla operacji na swoich modelach.

Przykładowo Laravel operuje na zdarzeniach: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored`.

Django: `pre_save`, `post_save`, `pre_delete`, `post_delete`, `m2m_changed`.

Wszystko powinno być jasno opisane w dokumentacji wybranego frameworka.

# Zdarzenia

```
class CourseGroupClass extends \Phalcon\Mvc\Model {

  // (...)

  public function afterCreate() {
    $students = $this->group->groupStudents;
    foreach($students as $student) {
      $grade = new Grade();
      $grade->save([
        "course_group_student_id" => $student->id,
        "course_group_class_id" => $this->id,
      ]);
    }
  }

}
```

# Podsumowanie

# Bibliografia i ciekawe źródła


- https://javastart.pl/static/programowanie-android/services/
- http://designpatternsphp.readthedocs.io/pl/latest/More/Repository/README.html
- http://designpatternsphp.readthedocs.io/pl/latest/Structural/DependencyInjection/README.html

# Pytania?

Kod prezentacji dostępny jest w repozytorium git pod adresem
`https://bitbucket.org/krewak/pwsz-ppsi`

Wszystkie informacje dot. kursu dostępne są pod adresem
`http://pwsz.rewak.pl/kursy/4`