KISS, DRY, YAGNI i inne

Zaawansowane metody programowania

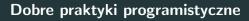
mgr inż. Krzysztof Rewak

6 marca 2019

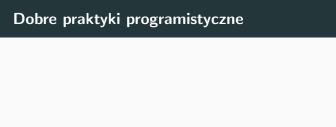
Wydział Nauk Technicznych i Ekonomicznych Państwowa Wyższa Szkoła Zawodowa im. Witelona w Legnicy

Plan prezentacji

- 1. KISS
- 2. DRY
- 3. YAGNI
- 4. I inne?
- 5. Podsumowanie



 ${\sf Czym}\ {\sf sa}\ {\sf dobre}\ {\sf praktyki}\ {\sf programistyczne?}$



Najprawdopodobniej każdy programista może przedstawić własny zestaw praktyk, które uważa za dobre.

Dobre praktyki programistyczne

(ciekawą rzeczą oczywiście będzie porównanie dwóch takich zestawów i przekonanie się, że część *zasad* brzmi podobnie, część dotyczy całkowicie różnych rzeczy, a część w zasadzie sobie przeczy)

Dobre praktyki programistyczne

Podobnie jak z zasadami SOLID, pewne rzeczy bardzo często są odkrywane przez programistów *samodzielnie*. Dobrze jednak znać najpopularniejsze reguły, aby umieć odnaleźć się w prawdziwym świecie.

KISS

KISS

KISS.

Keep It Simple, Stupid.

Nie komplikuj, głupcze.



Apoteoza minimalizmu, implementacja brzytwy Ockhama, leonardowska *Prostota jest szczytem wyrafinowania*.

KISS

KISS jest ciekawym zagadnieniem, ponieważ stosowane jest w wielu dziedzinach: od projektowania silników odrzutowców począwszy, a na tworzeniu filmów animowanych skończywszy.



Czym KISS będzie w programowaniu?



Warto wspomnieć przynajmniej dwa podejścia: upraszczanie nazewnictwa raz upraszczanie logiki biznesowej.



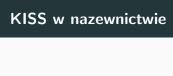
• • •

GradesCalculator implements Calculations

public GradesCalculator addGradeToCalculator(Grade grade)

public GradesCalculator addGradesToCalculator(Collection grades)

public GradesCalculator performCalculations() public string getCalculationsResult()



Czy naprawdę sens ma podkreślanie, że dodajemy oceny do kalkulatora w nazwie metody?

• • •

GradesCalculator implements Calculations

public GradesCalculator calculate() public string getResults()

public GradesCalculator addGrade(Grade grade)

public GradesCalculator addGrades(Collection grades)

KISS w logice biznesowej

Lepiej? Pewnie, że lepiej.

Ale spójrzmy na to jak działa taki kalkulator.

• • •

GradesController extends Controller

this.calculator = calculator

public Response calculate(Request request) User user = User.findById(request.get("id"))

this.calculator.addGrades(user.getGrades()) return response(this.calculator.getResults())

public GradesController(GradesCalculator calculator)

KISS w logice biznesowej

Kod powyżej się nie uruchomi, ponieważ nakomplikowaliśmy z interfejsem kalkulatora.

• • •

GradesCalculator implements Calculations

public GradesCalculator calculate() public string getResults()

public GradesCalculator addGrade(Grade grade)

public GradesCalculator addGrades(Collection grades)

KISS w logice biznesowej

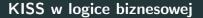
Jako autorzy tego kalkulatora doskonale wiemy, że nie ma co brać wyniku jeżeli uprzednio nie uruchomimy obliczania tegoż wyniku. To logiczne!

Problem polega na tym, że dla innego programisty nie będzie to wcale takie proste. Pamiętać, że musimy wywołać jakąś metodę zanim wywołamy inną? *Keep it simple, stupid!*

KISS w logice biznesowej

Jakie rozwiązanie powinno nam przyjść do głowy, aby uprościć taki kalkulator?

Dodać rzucanie wyjątku w momencie, gdy ktoś uruchomi pobranie wyniku? Czy to będzie faktycznie prostsze?



A może zahermetyzować obliczenia i wywołać je przy pobieraniu wyniku? Ale co z SRP?

KISS w logice biznesowej

Im większa klasa, tym więcej rzeczy będzie można prawdopodobnie uprościć. Spójrzmy na kontroler w serwisie randkowym, który umożliwia nadpisanie danych o użytkowniku:

ProfileController extends Controller

• • •

```
public Response update(Request request, string id)
    this.validateRequest(request)
    User user = this.users.findById(request.get("id"))
    user.visibleName = request.get("visibleName")
    user.firstName = request.get("firstName")
    user.lastName = request.get("lastName")
    user.birthDate = request.get("birthDate")
    user.about = request.get("about")
    user.avatarUrl = request.get("avatarUrl")
    if(request.get("password") === request.get("password2"))
        user.password = this.hasher.make(request.get("password"))
    user.height = request.get("height")
    user.weight = request.get("weight")
    user.hairColor = request.get("hairColor")
    user.skinColor = request.get("skinColor")
    user.eveColor = request.get("eveColor")
    user.education = request.get("education")
    user.job = request.get("job")
    user.hideRealName = request.get("hideRealName")
    user.hideBirthdate = request.get("hideBirthdate")
    user.profileVisibility = request.get("profileVisibility")
    user.allowsAdultsContent = request.get("allowsAdultsContent")
    user.allowsCommercials = request.get("allowsCommercials")
    user.allowsBefriending = request.get("allowsBefriending")
    user.allowsPrivateMessagesFromStrangers = request.get("allowsPrivateMessagesFromStrangers")
    this.mailer.send("profile-updated", user)
    user.save()
```

KISS

Im większa klasa, tym więcej rzeczy faktycznie można uprościć.

Im prostszy kod, tym łatwiej się go czyta. Może warto zastanowić się czy programista więcej kodu w swoim życiu przeczyta, czy wyprodukuje?

DRY.

Don't repeat yourself.

Nie powtarzaj się.

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Każdy fragment domeny musi mieć jedną, jednoznaczną i wyraźną reprezentację w systemie.

- The Pragmatic Programmer, Andy Hunt i Dave Thomas



DRY, według jego autorów, dotyczy przede wszystkim kodu, ale również baz danych, testów, wdrażania, a nawet dokumentacji.

Do podstawowego zastosowania młodzi programiści dochodzą bardzo szybko. Bo czymże innym jest zamykanie pewnych obszarów kodu do funkcji, jeżeli nie wykrozystaniem metodologii DRY?

```
Recipes implements Repository

public Collection all()

Collection recipes = Recipe.all()

return response(recipes.map(recipereturn {
```

```
return response(recipes.map(recipe => (
            id: recipe.id.
            name: recipe.name,
            author: recipe.author.name
public Collection latest()
    Collection recipes = Recipe.descending("created_at").limit(10).get()
    return response(recipes.map(recipe => (
            id: recipe.id.
            name: recipe.name,
            author: recipe.author.name
public Collection get(string id)
    Recipe recipe = Recipe.getById(id)
    return response({
        id: recipe.id,
       name: recipe.name,
       author: recipe.author.name
```

```
• • •
```

public Collection get(string id)
Recipe recipe = Recipe.getById(id)
return response(this.map(recipe))
protected Object map(Recipe recipe)
returd {
 id: recipe.id,
 name: recipe.name,
 author: recipe.author.name

```
Recipes implements Repository

public Collection all()
    Collection recipes = Recipe.all()
    return response(recipes.map(recipe => return this.map(recipe)))

public Collection latest()
    Collection recipes = Recipe.descending("created_at").limit(10).get()
    return response(recipes.map(recipe => return this.map(recipe)))
```



Nie tylko wydzielanie funkcji i metod pomaga w stosowaniu DRY.



LegalDrinkingAgeGuard implements BeforeOrderCompletionMiddleware

public Cart handle(Cart cart)

return cart

throw new IllegalDrinkingAgePurchase()

if(cart.items.containsAlcohol() and cart.user.profile.age < 18)</pre>



Bardzo dobrą praktyką jest korzystanie z nazwanych stałych, aby nie powtarzać wielokrotnie tych samych *magicznych liczb*.

• • •

 $Legal Drinking Age Guard\ implements\ Before Order Completion Middle ware$

public const MINIMUM_LEGAL_AGE = 18

throw new IllegalDrinkingAgePurchase()

public Cart handle(Cart cart) if(cart.items.containsAlcohol() and cart.user.profile.age < MINIMUM_LEGAL_AGE)</pre>

return cart



DRY należy rozumieć nie tylko na poziomie metod w klasach, ale również jako regułę dla całych projektów.

DRY

Czy pisanie za każdym razem od nowa tych samych funkcjonalności ma sens? W kończy powtarzamy kolejny i kolejny raz tą samą rejestrację użytkownika, ten sam mechanizm wysyłania mejla z potwierdzeniem, to samo wylogowanie, to samo uwierzytelnienie JWT, ten sam system praw...



 ${\sf Zatem}\ {\sf czy}\ {\sf używanie}\ {\sf frameworka}\ {\sf wpisuje}\ {\sf się}\ {\sf w}\ {\sf regulę}\ {\sf DRY?}$

DRY

A może wewnątrz frameworka warto stworzyć własny zestaw do postawienia świeżego projektu? Albo własny szkielet pod kolejnego CRUD-a? Albo własną implementację systemu kolejkowego?

DRY

Grunt to to, aby *nie powtarzać* swojej pracy.

l nie dlatego, że ktoś napisał o tym w swojej książce, ale przede wszystkim dlatego, że każda rzeczy powtórzona n razy, za którymś razem zostanie powtórzona błędnie.

YAGNI.

You aren't gonna need it.

Nie będziesz tego potrzebować.

Dwie przedstawione wcześniej reguły (podobnie jak zeszłowykładowy SOLID) mówią, że warto zastanowić się nad upraszczaniem kodu lub budową pewnej abstrakcji, aby łatwiej nam się żyło w przyszłości.

YAGNI na pierwszy rzut zaleca coś całkowicie innego.

Dopóty programista nie powinien dodawać żadnych funkcjonalności, dopóki nie będą one naprawde potrzebne.

Always implement things when you actually need them, never when you just foresee that you need them.

Zawsze programuj rzeczy, kiedy faktycznie ich potrzebujesz. Nigdy, kiedy po prostu przewidujesz, że będziesz ich potrzebować.

- Ron Jeffries

Prostym przykładem może byc otrzymanie od klienta zadania utworzenia endpointa do uwierzytelniania użytkowników.

• • •

Feature: INT-0023 Guest should be able to log into the system

Scenario: Happy path authentication Given I am unauthenticated guest

When I try to log in with "login" login and "password" password
Then I should receive authentication token

Scenario: Incorrect credentials

enario: incorrect credentials Given I am unauthenticated guest When I try to log in with "login" login and "incorrent-password" password

When I try to log in with "login" login and "incorrent-password" password
Then I should receive error message
And I should receive "401" error code

Co powinniśmy zrobić?

Ustawić routing, przykładowo POST /api/login na kontroler AuthenticationController?

Utworzyć reprezentacje użytkownika dla ORM-a i kontroler?

• • •

AuthenticationController extends Controller

```
public Response authenticate(Request request)
```

string password = request.get("password")

let user = Users.where("login = \$login").first()

if no user

return response("Invalid credentials.", 401) if no hash.check(user.password, password) return response("Invalid credentials.", 401)

string token = jwt.generateToken() return response(token)

string login = request.get("login")





Wykonaliśmy zadanie, ale chciałoby się coś jeszcze zrobić, prawda?

Może dodać middleware, które nałożymy na każdy request i sprawdzimy czy mamy przesyłany token? Może wyabstrahować walidację poza kontroler? A może utworzyć dodatkowy serwis do logowania?

No i przecież trzeba zabezpieczyć podanego routa żeby zalogowany użytkownik nie wygenerował sobie nowego tokena. Zróbmy też zwracającą na razie true metodę sprawdzającą czy użytkownik w ogóle może zostać zalogowany; przecież kiedyś wprowadzimy banowanie użytkowników, prawda?

A może w przyszłości będzie więcej niż jeden rodzaj użytkownika, więc wypadałoby nałożyć interfejs na repozytorium użytkowników i wstrzyknąć je do kontrolera? A może przydałoby się opakować token w jakiś uniwersalny view model?

A może w przyszłości będzie więcej niż jeden rodzaj użytkownika, więc wypadałoby nałożyć interfejs na repozytorium użytkowników i wstrzyknąć je do kontrolera? A może przydałoby się opakować token w jakiś uniwersalny view model?

• • •

AuthenticationController extends Controller

public AuthenticationController(Authenticatable repository, Authenticator service)

set repository set service

public Response authenticate(AuthenticationValidatedRequest request) string login = request.get("login") string password = request.get("password")

service.authenticate(login, password) return response(new SystemToken(jwt.generateToken()))



Lepiej? Pewnie, że tak! Jest jakby SOLID-niej, jakby czyściej. Kod też się skomplikował, co może być nie do przeskoczenia dla mniej doświadczonych programistów.

 ${\sf Gdzie}\ {\sf w}\ {\sf tym}\ {\sf wszystkim}\ {\sf leży}\ {\sf YAGNI?}$

Stety-niestety trzeba wykorzystać własne i cudze doświadczenie oraz zdrowy rozsądek, żeby nakreślić granicę między tym, co warto przewidywać na przyszłość, a tym co powinno pozostać proste.

Na pewno przy takim zadaniu nie powinniśmy jeszcze kombinować z żadnymi middlewarami, żadnymi skomplikowanymi walidacjami, z żadnymi przyszłymi metodami sprawdzania czy ktoś nie został zbanowany.

Dlaczego? Przede wszystkim może to przecież zostać zdefiniowanym w nowych zadaniach, a nie jest w żaden sposób krytyczną funkcjonalnością na obecny moment. Skoro wydzieliliśmy uwierzytelnienie do osobnego serwisu, nie powinno być problemem dodanie później reguł sprawdzających dodatkowe warunki.

Kod powinien być elastyczny, ale na pewno nie rociągnięty. Programistom często się wydaje, że myślą podobnie jak klient. Nigdy nie powinniśmy bez konsultacji dodawać nowych funkcjonalności, bez względu jak logiczne mogłoyby się wydawać.

YAGNI nigdy nie powinno być wymówką na tworzenie złego kodu. Powinno byc przestrogą przed tworzeniem niepotrzebnego kodu, który zaciemnia projekt i zwiększa pole do pojawienia się błedów.

I inne?

I inne

Akronimowych reguł jest wiele, wiele więcej. Czy warto znać wszystkie? Być może.

Ale lepiej po prostu programować, najlepiej w zespole. Wówczas sami dojdziemy do własnych wniosków na temat sensownego wytwarzania oprogramowania.

Podsumowanie



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



Wszystkie informacje dot. kursu dostępne są pod adresem http://pwsz.rewak.pl/kursy/10

