

# Zasady SOLID

## Zaawansowane metody programowania

---

mgr inż. Krzysztof Rewak

25 lutego 2019

Wydział Nauk Technicznych i Ekonomicznych

Państwowa Wyższa Szkoła Zawodowa im. Witelona w Legnicy

# Plan prezentacji

1. S

2. O

3. L

4. I

5. D

6. Podsumowanie

# SOLID

**SOLID** to mnemoniczny akronim zaproponowany przez amerykańskiego programistę Roberta C. Martina.

# SOLID

Dotyczy on pięciu podstawowych zasad tworzenia oprogramowania. Stosowanie się do nich powinno zwiększyć jakość tworzonego kodu pod względem jego czytelności, elastyczności i zdolności do szeroko rozumianego utrzymywania.

W programistycznym świecie zasady SOLID to podstawa, zarówno przy projektowaniu i implementacji systemów informatycznych, jak i przy rekrutacji nowych członków zespołu.

Część z tych zasad bywa czasami naturalnie odkrywana przez mniej doświadczonych programistów. Warto jednak wykorzystać mnemonikę Martina, aby w przyszłości móc bezpośrednio odpowiedzieć na pytania innych programistów.

S

---

# Zasada pojedynczej odpowiedzialności

**S** (lub **SRP**) to *single responsibility principle*, czyli zasada pojedynczej odpowiedzialności.



# Zasada pojedynczej odpowiedzialności

Najprawdopodobniej najprostsza do samodzielnego wywnioskowania, zasada jednej odpowiedzialności mówi, że jedna klasa powinna realizować tylko jeden cel.

# Zasada pojedynczej odpowiedzialności

W *Czystym kodzie* Martin sugeruje, że:

- 
-

# Zasada pojedynczej odpowiedzialności

W *Czystym kodzie* Martin sugeruje, że:

- klasy powinny być małe,
-

# Zasada pojedynczej odpowiedzialności

W *Czystym kodzie* Martin sugeruje, że:

- klasy powinny być małe,
- klasy powinny być mniejsze niż są.

# Zasada pojedynczej odpowiedzialności

Funkcje można mierzyć licząc liczbę linii kodu, z których się składają. Im więcej, tym zazwyczaj gorzej.

Klasy natomiast można zmierzyć za pomocą metryki tzw. odpowiedzialności.

# Zasada pojedynczej odpowiedzialności

Wyobraźmy sobie klasę implementującą dany interfejs `UserService`:

# Zasada pojedynczej odpowiedzialności

```
interface UserService

    public UserService()
    public UserService(User user)

    public UserService createNewUser(array data)
    public UserService editUser(array data)
    public UserService deleteUser()

    public UserService activateUser()
    public UserService deactivateUser()

    public Permissions getPermissions()
    public UserService promoteToAdmin()
    public UserService demoteFromAdmin()

    public UserService resetPassword()
    public UserService resetPassword(string password)
```

# Zasada pojedynczej odpowiedzialności

Jaki jest zakres odpowiedzialności tej klasy?

Co by się stało, gdybyśmy się umówili na odrzucenie nic nie mówiących nazw takich jak `UserService` i nazywali klasy zgodnie z ich odpowiedzialnością?



# Zasada pojedynczej odpowiedzialności (ałąć!)

```
interface CreatingEditingAndDeletingUserPlusSomeMumboJumboWithPermissionsAndRolesOhWaitAndPasswordsToo

    public UserService()
    public UserService(User user)

    public UserService createUser(array data)
    public UserService editUser(array data)
    public UserService deleteUser()

    public UserService activateUser()
    public UserService deactivateUser()

    public Permissions getPermissions()
    public UserService promoteToAdmin()
    public UserService demoteFromAdmin()

    public UserService resetPassword()
    public UserService resetPassword(string password)
```

# Zasada pojedynczej odpowiedzialności

Idea stojąca za SRP zakłada, że klasa powinna mieć tylko jeden *powód do zmiany*. Czyż poniższa klasa nie wygląda lepiej?



```
class UserPromoter  
  
    public UserPromoter(User user)  
    public UserPromoter promoteToAdmin()
```

# Zasada pojedynczej odpowiedzialności

Wygląda na to, że SRP jest faktycznie proste... jednakże jest również najczęściej łamane.

Podstawowym powodem takiego (przykrego) stanu jest strach przed zbytnim *napompowaniem* projektu wieloma klasami. Ale czy jednak klasa z czterdziestoma metodami na prawdę jest lepsza od czterdziestu klas z jedną metodą?

0



# Zasada otwarte-zamknięte

**O** (lub **OCP**) to *open/closed principle*, czyli zasada otwarte-zamknięte.

# Zasada otwarte-zamknięte

Zgodnie z zasadami klasy powinny być **otwarte** na rozszerzenia i **zamknięte** na modyfikacje.

# Zasada otwarte-zamknięte

Co to znaczy?

Chodzi o to, aby w razie potrzeby zmian, istniejący już kod nie był modyfikowany.

# Zasada otwarte-zamknięte (ałąć!)

```
import Rewak/ZMP/W02/Grades/Grade

class FinalGradeCalulcator

    Grade grades = []
    Grade finalGrade = new Grade()

    public GradesCalulcator calculate()
        sum = 0
        weight = 0

        for(Grade grade in grades)
            sum += grade.getValue()
            weight++

        finalGrade.setValue(sum / weight)

    return this
```



# Zasada otwarte-zamknięte

Kod z poprzedniego slajdu liczy średnią ocen studenta z podanych ocen cząstkowych. Nic trudnego.

Czy została zachowana zasada otwarte/zamknięte? Co się stanie jeżeli będziemy chcieli dodać do tego wszystkiego wagę ocen?

# Zasada otwarte-zamknięte (ałąć!)

```
import Rewak/ZMP/W02/Grades/Grade
import Rewak/ZMP/W02/Grades/ProjectGrade

class FinalGradeCalulcator

    Grade grades = []
    ProjectGrade projectGrade
    Grade finalGrade = new Grade()

    public GradesCalulcator calculate()
        sum = 0
        weight = 0

        for(Grade grade in grades)
            sum += grade.getValue()
            weight++

        if(projectGrade)
            sum += projectGrade.getValue()
            weight += 2

        finalGrade.setValue(sum / weight)

    return this
```

# Zasada otwarte-zamknięte

Lipa.

Żeby dodać możliwość ważenia średniej musieliśmy zmienić sposób przeprowadzania obliczeń. A co jeżeli, będziemy mieli więcej rodzajów ocen?

# Zasada otwarte-zamknięte

```
import Rewak/ZMP/W02/Interfaces/WeightedGrade
import Rewak/ZMP/W02/Grades/Grade

class FinalGradeCalulcator

    WeightedGrade grades = []
    Grade finalGrade = new Grade()

    // (...)

    public GradesCalulcator calculate()
        sum = 0
        weight = 0

        for(WeightedGrade grade in grades)
            sum += grade.getValue()
            weight += grade.getWeight()

        finalGrade.setValue(sum / weight)

    return this
```

# Zasada otwarte-zamknięte

Korzystając z interfejsu możemy uelastyczyć nasz serwis.

Czy dana implementacja rozwiąże wszystkie nasze problemy? Oczywiście, że nie.

A czy poprawi jakoś kodu?

L



# Zasada podstawienia Liskov

**L** (lub **LSP**) to *Liskov substitution principle*, czyli zasada podstawienia Liskov.

# Zasada podstawienia Liskov

*Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów.*



# Zasada podstawienia Liskov

Barbara Liskov sformułowała tę zasadę w *Data Abstraction and Hiererchy* w 1987. Z całej piątki SOLID to właśnie LSP brzmi najbardziej skomplikowanie, ale czy na pewno jest trudna do zrozumienia?

# Zasada podstawienia Liskov

Korzystając z dziedziczenia powinniśmy tak tworzyć nowe klasy, żeby jedynie rozszerzały możliwości rodzica.

Roszerzały, ale nie modyfikowały!

# Zasada podstawienia Liskov (ałąć!)

```
class Rectangle

    public void setHeight(int h)
        height = h

    public void setWidth(int w)
        width = w

    public int calculateArea()
        return height * width

class Square extends Rectangle

    public void setHeight(int x)
        height = x
        width = x

    public void setWidth(int x)
        height = x
        width = x
```

# Zasada podstawienia Liskov

Każdy kwadrat jest prostokątem, jasna rzecz.

Problem pojawi się jednak w miejscu, gdy *podstawimy* obiekt klasy Square pod miejsce Rectangle. Pierwszy dziedziczy po drugim, więc powinny zachowywać się tak samo. Ale czy naprawdę się tak zachowują?

# Zasada podstawienia Liskov (ałąć!)



```
Rectangle rectangle = new Rectangle()  
rectangle.setHeight(5)  
rectangle.setWidth(10)  
rectangle.calculateArea() // 50
```

```
Rectangle figure = new Square()  
figure.setHeight(5)  
figure.setWidth(10)  
figure.calculateArea() // 100
```

```
Square square = new Square()  
square.setHeight(5)  
square.setWidth(10)  
square.calculateArea() // 100
```

# Zasada podstawienia Liskov

Oczywiście w językach takich jak C++ można wykorzystać modyfikatory `virtual`, aby panować nad sytuacjami tego rodzaju, aczkolwiek sam kod wciąż będzie pogwałceniem zasady podstawienia Liskov.

I



# Zasada segregacji interfejsów

**I** (lub **ISP**) to *interface segregation principle*, czyli zasada segregacji interfejsów.



# Zasada segregacji interfejsów

Programiści często sami dochodzą do stwierdzenia jakoby wiele dedykowanych interfejsów jest lepszych niż jeden ogólny.

# Zasada segregacji interfejsów (ałąć!)

```
class UserTable implements TableConverter

    public Response getJSON(Request request)
    public Response getCSV(Request request)
    public Response getPDF(Request request)
    public Response getXLS(Request request)
    public Response getXML(Request request)

    protected Collection get()
```

# Zasada segregacji interfejsów

A może lepiej byłoby to rozbić?

# Zasada segregacji interfejsów

```
class UserTable implements JSONConvertible, CSVConvertible, PDFConvertible, XLSConvertible, XMLConvertible

    public Response getJSON(Request request)
    public Response getCSV(Request request)
    public Response getPDF(Request request)
    public Response getXLS(Request request)
    public Response getXML(Request request)

    protected Collection get()
```

# Zasada segregacji interfejsów

Warto pamiętać o dwóch rzeczach przy korzystaniu z interfejsów.

# Zasada segregacji interfejsów

Po pierwsze, interfejsy można po sobie dziedziczyć wielokrotnie (również w językach w których nie występuje wielodziedziczenia).

# Zasada segregacji interfejsów

Po drugie, interfejsy mogą być puste.

# Zasada segregacji interfejsów

Szybko się okaże, że ISP łączy się z SRP.



# Zasada segregacji interfejsów

Zgodnie z zasadą lepiej mieć wiele krótkich interfejsów niż jeden wielki.  
Ale czy klasa z kilkoma interfejsami nie przeczy SRP?

**D**



# Zasada odwrócenia zależności

**D** (lub **DIP**) to *dependency inversion principle*, czyli zasada odwrócenia zależności.

# Zasada odwrócenia zależności

*Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych.*

*Zależności między nimi powinny wynikać z abstrakcji.*

# Zasada odwrócenia zależności


To jest oczywiście rzecz, o której na wykładach mówiliśmy wielokrotnie.

# Zasada odwrócenia zależności

Wyobraźmy sobie serwis wielokrotnego użytku rejestrujący użytkowników. Wykorzystajmy klasę `User` dziedzicząca po modelu z ORM-a typu *active record*.

Taki serwis może być wykorzystany w wystawionym publicznie kontrolerze rejestracji, w panelu administracyjnym, z poziomu konsoli, a i pewnie też w wielu innych miejscach.

# Zasada odwrócenia zależności



```
class UserRegistrar

    public void create(ValidatedRequest request)

        User user = new User()
        user.name = request.get("name")
        user.email = request.get("email")
        user.password = hash.make(request.get("password"))
```

# Zasada odwrócenia zależności

Łatwizna.

A co jeżeli będziemy mieli osobny model administratora? Zaczyna się robic problem?



# Zasada odwrócenia zależności (aść!)

```
class UserRegistrar

    public void create(ValidatedRequest request)

        User user = new User()
        user.name = request.get("name")
        user.email = request.get("email")
        user.password = hash.make(request.get("password"))

    public void createAdmin(ValidatedRequest request)

        Admin admin = new Admin()
        admin.name = request.get("name")
        admin.email = request.get("email")
        admin.password = hash.make(request.get("password"))
```

# Zasada odwrócenia zależności

Za dużo kodu się potwarza, więc może wypadałoby przesunąć część linijek do osobnej metody?

# Zasada odwrócenia zależności (ałąć!)

```
class UserRegistrar

    public void create(ValidatedRequest request)

        User user = new User()
        register(request, user)

    public void createAdmin(ValidatedRequest request)

        Admin admin = new Admin()
        register(request, admin)

    protected void register(ValidatedRequest request, user)

        user.name = request.get("name")
        user.email = request.get("email")
        user.password = hash.make(request.get("password"))
```

# Zasada odwrócenia zależności

W PHP czy Pythonie nie było problemu (w zależności oczywiście od ludzi robiących *code review*). Ale i tak wypadałoby określić jakiś typ łączący użytkownika i administratora.

Lepiej byłoby założyć, że jedno nie dziedziczy po drugim... na wszelki wypadek, gdyby zaraz miało pojawić się coś nowego.

# Zasada odwrócenia zależności (ałąć!)

```
class UserRegistrar

    public void create(ValidatedRequest request)

        User user = new User()
        register(request, user)

    public void createAdmin(ValidatedRequest request)

        Admin admin = new Admin()
        register(request, admin)

    protected void register(ValidatedRequest request, Registerable user)

        user.name = request.get("name")
        user.email = request.get("email")
        user.password = hash.make(request.get("password"))
```

# Zasada odwrócenia zależności

I co? Okazuje się, że zbudowaliśmy podręcznikowy przykład implementacji wzorca odwrócenia sterowania w postaci wstrzykiwania zależności.

# Zasada odwrócenia zależności



```
class UserRegistrar
```

```
    protected void create(ValidatedRequest request, Registerable user)
```

```
        user.name = request.get("name")
```

```
        user.email = request.get("email")
```

```
        user.password = hash.make(request.get("password"))
```

# Podsumowanie

---



**Pytania?**

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>

