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. 0
- 3. L
- 4. I
- 5. D
- 6. Podsumowanie



 ${\bf SOLID}$ 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 zdatności do szeroko rozumianego utrzymywania.

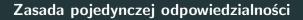
SOLID

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

SOLID

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

S



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

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

W Czystym kodzie Martin sugeruje, że:

- •
- •

W Czystym kodzie Martin sugeruje, że:

- klasy powinny być małe,
- •

W Czystym kodzie Martin sugeruje, że:

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

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

Klasy natomiast można zmierzyć za pomoca metryki tzw. 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)
```

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?

```
interface CreatingEditingAndDeletingUserPlusSomeMumboJumboWithPermissionsAndRolesOhWaitAndPasswordsToo

public UserService(User user)

public UserService createNewUser(array data)

public UserService editUser(array data)

public UserService activateUser()

public UserService activateUser()

public UserService activateUser()

public UserService deleteUser()

public UserService denermissions()

public UserService denoreFromAdmin()

public UserService denoteFromAdmin()

public UserService resetPassword()

public UserService resetPassword(string password)
```

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()
```

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?



 \mathbf{O} (lub $\mathbf{OCP})$ to open/closed principle, czyli zasada otwarte-zamknięte.

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

Co to znaczy?

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

Zasada otwarte-zamknięte (ałć!)

```
• • •
class FinalGradeCalulcator
   Grade grades = []
   Grade finalGrade = new Grade()
    public GradesCalulcator calculate()
        for(Grade grade in grades)
        finalGrade.setValue(sum / weight)
        return this
```

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 ważone oceny?

Zasada otwarte-zamknięte (ałć!)

```
• • •
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()
        weight = 0
        for(Grade grade in grades)
            sum += grade.getValue()
            weight += 2
        finalGrade.setValue(sum / weight)
        return this
```

Lipa.

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

```
• • •
import Rewak/ZMP/W02/Interfaces/WeightedGrade
import Rewak/ZMP/W02/Grades/Grade
class FinalGradeCalulcator
    WeightedGrade grades = []
    Grade finalGrade = new Grade()
    public GradesCalulcator calculate()
        for(WeightedGrade grade in grades)
            sum += grade.getValue()
        finalGrade.setValue(sum / weight)
        return this
```

Korzystając z interfejsu możemy uelastycznić nasz serwis.

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

A czy poprawi jakoś kodu?

L

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

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?

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

Roszerzały, ale nie modyfikowały!

Zasada podstawienia Liskov (ałć!)

```
• • •
class Rectangle
    public void setHeight(int h)
    public void setWidth(int w)
    public int calculateArea()
        return height * width
class Square extends Rectangle
    public void setHeight(int x)
    public void setWidth(int x)
```

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 (ałć!)

```
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.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.



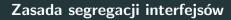
I (lub ISP) to *interface segregation principle*, czyli zasada segregacji interfejsów.

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

Zasada segregacji interfejsów (ałć!)

```
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()
```



A może lepiej byłoby to rozbić?

```
class UserTable implements JSONConvertable, CSVConvertable, PDFConvertable, XLSConvertable, XMLConvertable

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)
public Response getXML(Request request)
```



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

Po drugie, interfejsy mogą być puste.

Szybko się okaże, że ISP łączy się z SRP.



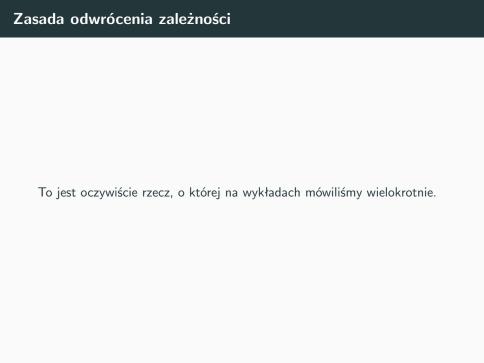
Zgodnie z zasadą lepiej mieć wiele krótkich interfejsów niż jeden wielki. Ale czy klasa z kilkoma interfejsami nie przeczy SRP?

D

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

Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych.

Zależności między nimi powinny wynikać z abstrakcji.



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.

```
class UserRegistrar

public void create(ValidatedRequest request)

User user = new User()

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

user.enall = request.get("enal")

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

Łatwizna.

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

Zasada odwrócenia zależności (ałć!)

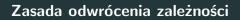
```
class UserRegistrar

public void create(ValidatedRequest request)

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



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

Zasada odwrócenia zależności (ałć!)

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

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 (ałć!)

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

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



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

