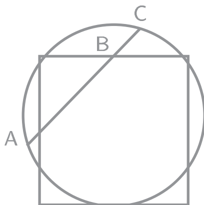
Inżynieria oprogramowania

Radosław Klimek

2015-23

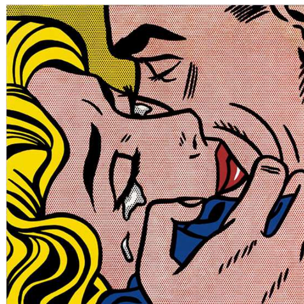


http://home.agh.edu.pl/rklimek

1 Zasady projektowania obiektowego SOLID/KISS/DRY

1 Zasady projektowania obiektowego SOLID/KISS/DRY

Zasady projektowania obiektowego SOLID/KISS/DRY (na przykładzie języka Java)



Roy LICHTENSTEIN: *Pocałunek V*

Zasady SOLID

SOLID zbiór pięciu podstawowych zasad, którymi należy się kierować programując obiektowo:

- wypracowane zostały one przez programistów na przestrzeni lat,
- finalnie sformułował i opisał je znany amerykański programista Robert Martin,
- zasady są poruszane w publikacjach dotyczących programowania obiektowego i dobrych praktyk programowania.

Co to jest SOLID?

SOLID to skrót od pierwszych liter zasad:

Litera	Skrót	Nazwa angielska Nazwa polska
S	SRP	Single responsibility principle Zasada jednej odpowiedzialności
O	OCP	Open closed principle Zasada otwarte zamknięte
L	LSP	Liskov substitution principle Zasada podstawiania Liskov
I	ISP	Interface segregation principle Zasada segregacji interfejsów
D	DIP	Dependency inversion principle Zasada odwracania zależności

Single Responsibility Principle

SRP – pierwsza zasada SOLID była już znana wcześniej jako zasada jedności i brzmi ona:

> Klasa lub metoda powinna mieć jeden i tylko jeden powód do zmiany.

- Każda klasa musi być odpowiedzialna za jedną konkretną rzecz.
- Powinien istnieć jeden konkretny powód do modyfikacji danej klasy.
- Stosowanie tej zasady znacząco zwiększa ilość klas w programie, a jednocześnie zmniejsza ilość klas typu scyzoryk szwajcarski. Takim mianem określa się wielkie kilkuset linijkowe klasy, skupiające za dużo funkcjonalności.

SRP – przykład naruszenia zasady

```
class Person {
  public string Name { get; set; }
  public string Lastname { get; set; }
  public string City { get; set; }
  public string Street { get; set; }
  public int HouseNumber { get; set; }
  public string Email { get; set; }
  public Person(string name, string lastname, string email) {
    Name = name;
    Lastname = lastname;
    Email = ValidateEmail (
  }
  private string ValidateEmail(string email) {
    if email.Contains ("@") || email.Contains (".")) {
      throw new FormatException("Email address has a wrong format!");
    }
    return email;
  }
}
```

SRP – przykład naruszenia zasady (cd)

- Klasa `Person` narusza zasadę SRP, bo ma więcej niż jeden powód do zmiany:
  - chcemy dodać np. nr telefonu pracownika (zmiana danych osobowych),
  - chcemy zwalidować adres e-mail bez wyrzucania wyjątku.
- Klasa zawiera w sobie metodę sprawdzającą poprawność adresu e-mail, a nie powinno to należeć obowiązku klasy `Person`.
- Klasa `Person` nie powinna zawierać atrybutów, które nie są z nią powiązane (adres zameldowania).

SRP – przykład klas nienaruszających zasady

```
class Address {
    public string City { get; set; }
    public string Street { get; set; }
    public int HouseNumber { get; set; }
}
class Person {
    public string Name { get; set; }
    public string Lastname { get; set; }
    public string Email { get; set; }
    public Address PersonAddress { get; set; }
    public Person(string name, string lastname, string email) {
        Name = name;
        Lastname = lastname;
        Email = email;
    }
}
class EmailValidator {
    public void ValidateEmail(string email) {
        if (!email.Contains("@") || !email.Contains(".")) {
            throw new FormatException("Email address has a wrong for
        }
    }
}
```

Open/closed principle

OCP – drugą z zasad SOLID jest zasada otwarty/zamknięty. Zasada ta została opisana przez Bertranda Meyera w 1988 roku w książce Object-Oriented Software Construction, i brzmi ona:

> Elementy systemu powinny być otwarte na rozbudowę, ale zamknięte na modyfikacje.

- Celem zasady jest możliwość dodawania nowych funkcjonalności, bez konieczność modyfikowania poszczególnych elementów systemu.
- Jest to bardzo ważna zasada zwłaszcza gdy w systemie często pojawiają zmiany.
- Z jednej strony przestrzeganie tej zasady pozwala na tworzenie elastycznego systemu, który jest łatwy do rozbudowy.
- Z drugiej strony ograniczamy ryzyko wprowadzenie do systemu nowych błędów (zmiana deklaracji jakiejkolwiek metody może spowodować awarię systemu).

OCP – przykład naruszenia zasady

```
class Square {
        public int A { get; set; }
}
class Rectangle {
        public int A { get; set; }
        public int B { get; set; }
}
class Calculator {
        public int Area(object shape) {
                if (shape is Square) {
                        Square square = (Square)shape;
                        return square.A * square.A;
            }
                else if (shape is Rectangle) {
                    Rectangle rectangle = (Rectangle)shape;
                            return rectangle.A * rectangle.B;
             }
                    return 0;
        }
}
```

OCP – przykład naruszenia zasady (cd)

- W powyższym przykładzie dodanie jakiejkolwiek nowej figury, wiąże się z koniecznością modyfikacji istniejącej klasy. Jest to ewidentne złamanie zasady otwarty/zamknięty, ponieważ klasa nie jest otwarta na rozbudowę.
- Dzięki użyciu polimorfizmu można obarczyć implementacją metody liczącej pole figury każdą klasę reprezentującą figurę. Kod będzie dodatkowo o wiele prostszy.

OCP – przykład nienaruszenia zasady

```
abstract class Shape {
        public abstract int Area();
}
class Square : Shape {
        public int A { get; set; }

        public override int Area() {
                return A * A;
        }
}
class Rectangle : Shape {
        public int A { get; set; }
        public int B { get; set; }

        public override int Area() {
                return A * B;
        }
}
class Calculator {
        public int Area(Shape shape) {
                return shape.Area();
        }
}
```

Liskov substitution principle

LSP – trzecia z zasad SOLID to zasada podstawień Liskov. Została ona opisana przez Barbarę Liskov i brzmi ona:

> 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.

- Czyli nie powinieneś zauważyć różnicy między używaniem podtypu i typu bazowego.
- Oznacza to, że w całości musi być zachowana zgodność interfejsu i wszystkich metod.

Liskov substitution principle (cd)

Implementując w poprawny sposób zasadę podstawienia Liskov, nie wolno posługiwać się konstrukcjami warunkowymi aby wymusić poprawne działanie oprogramowania np.:

```
public void funkcja(){
    if(obiekt_klasy_bazowej)
            //do sth..
    else if(obiekt_klasy_pochodnej)
            //do sth..
    }
```

Aby uniknąć tego typu sytuacji stosujemy polimorfizm.

LSP – przykład naruszenia zasady

```
class frezarka {
     public void opercja( ){
          console.writeline("frezarka działa");
     }
}
class obrabiarka : frezarka {
      public void operacja ( ) {
           console,writeline("obrabiarka działa");
      {
}
```

Kolejny przykład:

```
obrabiarka obr = new obrabiarka( );
obr.operacja ( );
                // wynik:  ''orabiarka działa''

frezarka frea = obr;
frez.operacja ( );
                 // wynik: '' frezarka działa''
```

LSP – przykład nienaruszenia zasady

```
class frezarka {
     public  virtual void opercja( ){
          console.writeline("frezarka działa");
     }
}
class obrabiarka : frezarka {
      public  override void operacja ( ) {
           console,writeline("obrabiarka działa");
      {
}
```

Kolejny przykład:

```
obrabiarka   obr = new obrabiarka( );
obr.operacja ( );
              // wynik:  ''obrabiarka działa''

frezarka     frez = obr;
frez.operacja ( );
               // wynik: '' obrabiarka działa''
```

LSP – przykład nienaruszenia zasady (cd)

Albo też można wykorzystać struktury abstrakcyjne tj.: interfejs oraz klasa abstrakcyjna:

```
interface IMaszyna {
      void operacja();
}
class frezarka : IMaszyna {
        public virtual void operacja() {
              console.WriteLine("Frezarka␣działa");
        }
}
class Obrabiarka : Frezarka {
        public override void Operacja() {
               Console.WriteLine("Obrabiarka␣działa");
        }
 }
```

Interface segregation principle

ISP – czwarta zasada segregacji interfejsów mówi, że:

> Interfejsy powinny być na tyle małe i konkretne, aby nie zmuszały klas do implementacji metod, których nie potrzebują.

- Inaczej klasa nie powinna zależeć od metod, których nie używa.
- Interfejsy powinny odpowiadać za niewielką funkcjonalność – lepiej jest stworzyć wiele małych interfejsów niż kilka rozbudowanych. Łatwo tutaj zauważyć, że zasada segregacji interfejsów jest powiązana z zasadą pojedynczej odpowiedzialności.

ISP – przykład naruszenia zasady

```
public interface IEmployee {
        void Work();
        void Eat();
}
public class Employee : IEmployee {
        public void Work() {
                //Code to work
        }
      public void Eat() {
      //Code to eat
    }
}
public class Robot : IEmployee { public void Work() {
//Code to work
}
 public void Eat() {
throw new NotImplementedException(); }
}
```

ISP – przykład naruszenia zasady (cd)

Kolejny przykład:

```
public class Program {
    static void Main() {
        var employess = new List {
                new Employee(),
                new Robot()
            };
        foreach (var employee in employess)
            employee.Eat();
            //Unhandled exception
    }
}
```

ISP – przykład naruszenia zasady (cd)

- W tym przykładzie mamy interfejs `IEmployee`, który został zaimplementowany przez klasy `Employee` oraz `Robot`. Łamiemy zasadę ISP. Klasa `Robot` nie potrzebuje metody `Eat`, mimo to musi w tym przypadku taką metodę zaimplementować.
- Jeżeli w kodzie jakiejś klasy są metody, które rzucają wyjątek `NotImplementedException`, lub po prostu są puste, to sygnał, że prawdopodobnie nasz interfejs jest zbyt obszerny.
- Należy podzielić zbyt ogólny interfejs, na kilka (w tym przypadku dwa) bardziej szczegółowych interfejsów.

ISP – przykład nienaruszenia zasady

```
public interface IWork {
        void Work();
}
public interface IEat {
            void Eat();
}
public class Employee : IWork, IEat {
        public void Work() {
                //Code to work
        }
        public void Eat() {
                //Code to eat
        }
}
public class Robot : IWork {
        public void Work() {
                //Code to work
        }
}
```

Dependency inversion principle

DIP – ostatnia to zasada odwracania zależności, składająca się z dwóch punktów:

1. Moduły wysokiego poziomu nie powinny zależeć od modułów niższego poziomu. Oba rodzaje modułów powinny zależeć od abstrakcji.
2. Abstrakcje nie powinny zależeć od szczegółów. Szczegóły powinny zależeć od abstrakcji.

Głównym celem zasady DIP jest zmniejszenie zależności od konkretnych implementacji. Możemy to uzyskać za pomocą abstrakcji (interfejsów). Jeśli kod zależy od interfejsu to mamy małą zależność. Dzięki temu nasz kod nie zmienia się lub zmienia się bardzo rzadko.

DIP – przykład naruszenia zasady

```
public class Employee {
        public string Name { get; set; }
}
public class EmployeeRepository {
       public void Add(Employee employee){
                                  //Add employee to database
       }
}
public class EmployeeService {
        private EmployeeRepository _employeeRepository = new Employee
        public void Add(Employee employee) {
                        _employeeRepository.Add(employee);
        }
}
```

DIP – przykład naruszenia zasady (cd)

- W powyższym kodzie modułem wysokopoziomowym jest klasa `EmployeeService`, a modułem niskopoziomowym klasa `EmployeeRepository`.
- Moduł wysokopoziomowy w tym przykładzie zależy od modułu niskopoziomowego, ponieważ używa konkretnej implementacji, czyli klasy `EmployeeRepository`, a to właśnie jest łamaniem zasady DIP.
- Aby ten kod był zgodny z DIP, musimy właśnie odwrócić te zależności (jak mówi sama nazwa).

DIP – przykład nienaruszenia zasady

```
public class Employee {
     public string Name { get; set; }
}
public interface IEmployeeRepository {
    void Add(Employee employee);
}
public class EmployeeRepository : IEmployeeRepository {
    public void Add(Employee employee) {
            //Add employee to database
    }
}
public class EmployeeService {
    private IEmployeeRepository _employeeRepository;
    public EmployeeService(IEmployeeRepository employeeRepository)
                _employeeRepository = employeeRepository;
    }
    public void Add(Employee employee) {
            _employeeRepository.Add(employee);
    }
}
```

DIP – przykład nienaruszenia zasady (cd)

W powyższym przykładzie nasz serwis `EmployeeService` nie zależy już od konkretnej implementacji `EmployeeRepository`, a jedynie zależy od abstrakcji. W tym przypadku od interfejsu `IEmployeeRepository`. Zmiany w module niskopoziomowym nie mają wpływu na moduł wysokopoziomowy.

Inne zasady

Istnieją inne jeszcze zasady, z których popularniejsze to:

- KISS,
- DRY.

Zasada KISS

KISS – Keep It Simple, Stupid – dosłownie: zrób to prosto (idioto).

Zasada powstała w latach 60. XX wieku w środowisku amerykańskich inżynierów wojskowych i przypisywana inżynierowi lotnictwa Kelly'emu Johnsonsowi.

- Jej istotą miało być tworzenie projektowanych silników odrzutowych w tak prosty sposób, aby przeciętny mechanik mógł je naprawić w każdych warunkach, przy użyciu prostych narzędzi.
- Zasada ta znajduje zastosowanie w wielu działaniach których nadrzędnym celem jest wysoka skuteczność – począwszy od biznesu, poprzez projektowanie i design, a na życiu codziennym skończywszy.

Przykład naruszający zasadę KISS

```
class notKISSClass1 {
        int num1;
        int num2;
        public int Calculate1(int n1, int n2) {
                return n1 / n2;
        }
        int res;
        public notKISSClass1() {
                res = Calculate1(num1,num2);
        }
}
```

Przykład nienaruszający zasadę KISS

```
public class NumberCalculator {
        public int Number1 { get; set; }
        public int Number2 { get; set; }
        public int Result { get; set; }

        public NumberCalculator() {
              Result = DivideNumbers(Number1, Number2);
        }
        public int DivideNumbers(int Number1, int Number2) {
              return (Number1 / Number2);
        }
}
```

Zasada DRY

DRY – Don't Repeat Yourself – dosłownie: nie powtarzaj się.

- Zasada ta, który mówi, że należy unikać powtarzania tych samych części kodu w różnych miejscach. Pozwala to na uniknięcie kopiowania lub kopiowania z niewielką zmianą części kodu w inne miejsca.
- Główną zaletą DRY jest uniknięcie błędów popełnianych w trakcie kopiowania powtarzających się fragmentów kodu.
- Pozwala to zaoszczędzić czas, który tracimy, aby wprowadzić mało znaczące zmiany do kopiowanego kodu a następnie czas, który tracimy na szukanie zmian, które przeoczyliśmy w trakcie kopiowania.

Zasada DRY (cd)

Sposoby realizacji założeń DRY w językach programowania:

- funkcje (najbardziej powszechny),
- szablony lub makra,
- struktury,
- klasy,
- stałe,
- moduły,
- biblioteki,
- polimorfizm (dla programowania obiektowego).