Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

XIV. Zasady SOLID-nego programowania

1. SOLID – zasady efektywnego programowania obiektowego

1.1. W roku 1994, Robert Martin sformułował pięć zasad skutecznego (ang. effective) programowania obiektowego. Nadał im mnemoniczny skrót SOLID, pochodzący od pierwszych liter nazw poszczególnych zasad. Zasady te nazywają się, w kolejności:

- SRP (ang. Single responsibility principle) – zasada jednej odpwiedzialności,
- OCP (ang. Open/closed principle) – zasada otwarte/zamknięte,
- LSP (ang. Liskov substitution principle) – zasada podstawienia Liskov,
- ISP (ang. Interface segregation principle) – zasada segregacji interfejsów,
- DIP (ang. Dependency inversion principle) – zasada odwrócenia zależności.

1.2. Zasada jednej odpowiedzialności dotyczy budowy klasy i stwierdza, że klasa powinna być odpowiedzialna za wykonywanie jednego typu działań. Innymi słowy, klasa nie powinna być nadmiernie rozbudowana, a jej funkcjonalność powinna mieć charakter jednorodny. Projektowanie rozbudowanych klas pogarsza czytelność kodu i stwarza problemy przy jego refaktoryzacji. Nadmiernie rozbudowana klasa jest jak szwajcarski scyzoryk.
Zasada SRP jest pochodną metody „dziel i zwyciężaj”, przyjmowanej przy projektowaniu rozlicznych algorytmów w informatyce.

Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

Przykład źle zaprojektowanej klasy (kod poniższego i dalszych, błędnie skonstruowanych programów został zaczerpnięty ze strony https://springframework.guru/solid-principles-object-oriented-programming/

```
class Text {
     String text;`
     String author;
     int length;
     String getText() { ... }
     void setText(String s) { ... }
     String getAuthor() { ... }
     void setAuthor(String s) { ... }
     int getLength() { ... }
     void setLength(int k) { ... }

     /*methods that change the text*/

     void allLettersToUpperCase() { ... }
     void findSubTextAndDelete(String s){ ... }

     /*method for formatting output*/
     void printText() { ... }
}
```

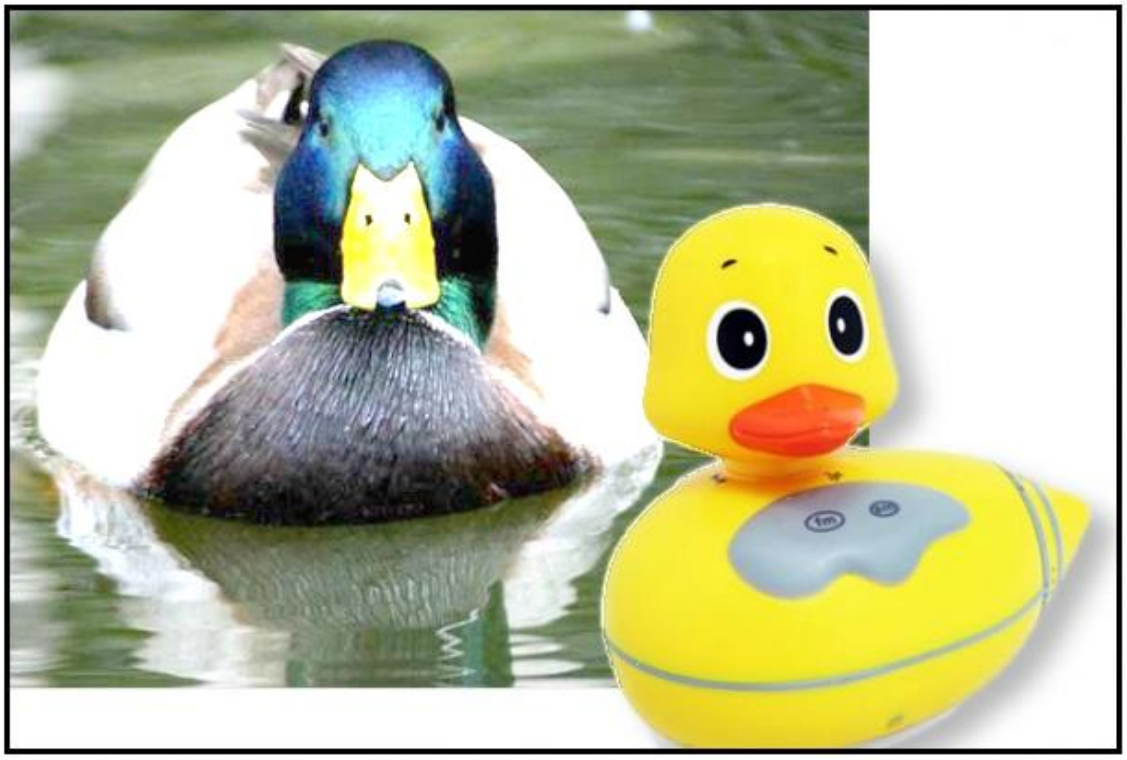
1.3. Zasada otwarte/zamknięte stwierdza, że wszelkie zmiany dokonywane w aplikacji w procesie jej utrzymywania i rozwoju powinny polegać nie na modyfikacji istniejącego kodu (zamknięte), lecz na rozszerzaniu funkcjonalności poprzez dodawanie nowego kodu (otwarte).
Jeden z zalecanych sposobów implementacji tej zasady polega na intensywnym używaniu polimorfizmu.

Przykład źle zaprojektowanej (i modyfikowanej) klasy:

```
public class HealthInsuranceSurveyor {
    public boolean isValidClaim() {
        System.out.println("HealthInsuranceSurveyor:
        Validating health insurance claim...");
        /*Logic to validate health insurance claims*/
        return true;
    }
}

public class ClaimApprovalManager {
    public void processHealthClaim
                    (HealthInsuranceSurveyor surveyor) {
        if(surveyor.isValidClaim()) {
            System.out.println("ClaimApprovalManager:
              Valid claim. Currently processing claim
              for approval....");
         }
    }
}

public class ClaimApprovalManager {
    public void processHealthClaim
                    (HealthInsuranceSurveyor surveyor) {
        if(surveyor.isValidClaim()) {
            System.out.println("ClaimApprovalManager:
              Valid claim. Currently processing claim
              for approval....");
         }
    }

    public void processVehicleClaim
                    (VehicleInsuranceSurveyor surveyor) {
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager:
              Valid claim. Currently processing claim
              for approval....");
        }
    }
}
```

Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

1.4. Zasada podstawienia Liskov stanowi, że: jeśli w programie typ S jest podtypem typu T, to wtedy obiekty typu T mogą być zastępowane przez obiekty typu S bez zmiany istotnych właściwości programu. Zasada została sformułowana przez Barbarę Liskov w roku 1988. Uważa się, że jej naruszanie prowadzi do powstawania kodu trudnego w utrzymaniu.
Implementacja powyższej zasady odbywa się przez intensywne stosowanie paradygmatów dziedziczenia i polimorfizmu.



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Przykład źle zaprojektowanego kodu:

```
class TrasportationDevice
{
   String name;
   String getName() { ... }
   void setName(String n) { ... }

   double speed;
   double getSpeed() { ... }
   void setSpeed(double d) { ... }

   Engine engine;
   Engine getEngine() { ... }
   void setEngine(Engine e) { ... }

   void startEngine() { ... }
}

class Car extends TransportationDevice
{
   @Override
   void startEngine() { ... }
}

class Bicycle extends TransportationDevice
{
   @Override
   void startEngine() /*problem!*/
}
```

1.5. Zasada segregacji interfejsów została po raz pierwszy użyta przez R. Martina w trakcie konsultacji udzielanych przez niego firmie XeroX. W pierwotnym brzmieniu, zacytowanym przez niego w podręczniku „Agile Software Development Principles, Patterns and Practices", brzmiała jak następuje:

"Clients should not be forced to depend on methods that they do not use".

W programowaniu obiektowym, w kontekście stosowania interfejsów, niniejsza zasada stwierdza, że interfejsy nie powinny by sztucznie „nadmuchiwane" metodami, które nie będą używane przez implementujące je klasy. Klasa implementująca interfejs musi przecież zagwarantować implementację każdej ujętej w nim metody (choćby symboliczną). Co więcej, dowolna modyfikacja interfejsu

pociąga za sobą konieczność odpowiedniej modyfikacji klas, które go implementują.
Zasada ISP zaleca odchudzanie obszernych interfejsów do postaci małych, spójnych interfejsów, deklarujących metody do pełnienia pojedynczej roli (tzw. interfejsy ról).

Przykład źle zaprojektowanego kodu:

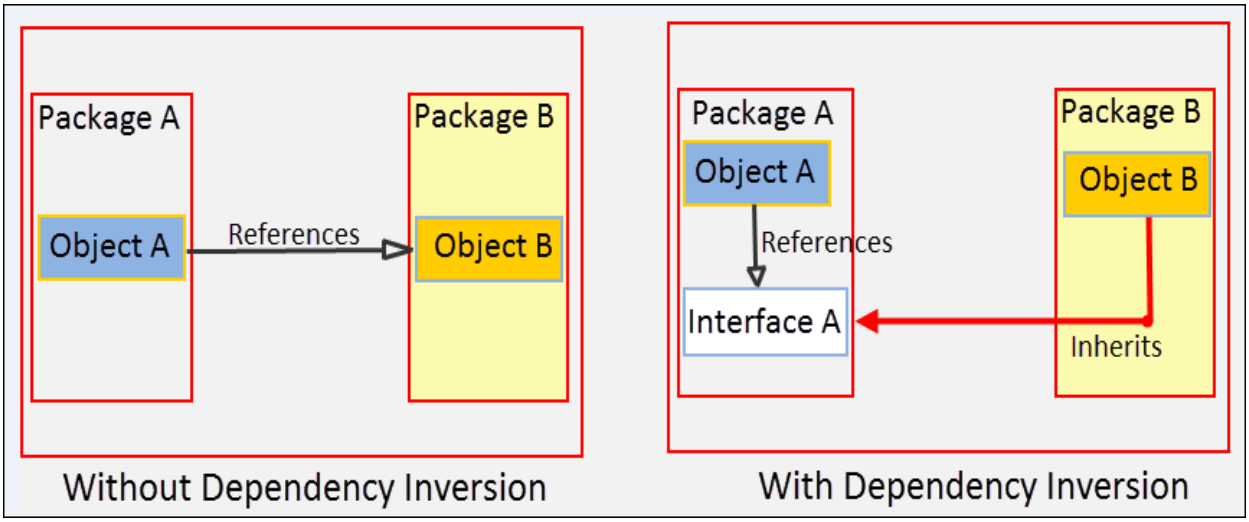
```
public interface Toy {
    void setPrice(double price);
    void setColor(String color);
    void move();
    void fly();
}

public class ToyHouse implements Toy {
    double price;
    String color;
    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){}
    @Override
    public void fly(){}
}
```

1.6. Stosowanie zasady odwrócenia zależności ma na celu unikanie ścisłego wiązania kodu i jego konsekwencji. Ścisłe powiązanie wystąpi np. między klasami A i B takimi, że obiekt klasy B powstaje wewnątrz klasy A. Dowolna zmiana dokonana wewnątrz klasy B może w takiej sytuacji wymusić konieczność zmiany także klasy A. Aby tego uniknąć, należy raczej wiązać kod w sposób luźny.
Postulat odwrócenia zależności stanowi, że:

– moduł znajdujący się w hierarchii na wysokim poziomie nie powinien zależeć od modułów niskopoziomowych. Należy zmierzać do tego, by oba te moduły uzależnić od pewnego modułu abstrakcyjnego;

– moduły abstrakcyjne nie powinny zależeć od modułów szczegółowych; zależność powinna zachodzić w kierunku odwrotnym.

Oto przykład rozwiązania pewnego problemu przy użyciu zasady odwrócenia zależności:



Przykład źle zaprojektowanego kodu:

```
public class LightBulb {
  public void turnOn() {
    System.out.println("LightBulb: Bulb turned on...");
  }
  public void turnOff() {
    System.out.println("LightBulb: Bulb turned off..");
  }
}
```

```
public class ElectricPowerSwitch {
  public LightBulb lightBulb;
  public boolean on;
  public ElectricPowerSwitch(LightBulb lightBulb) {
    this.lightBulb = lightBulb;
    this.on = false;
  }
  public boolean isOn() {
    return this.on;
  }
  public void press() {
    boolean checkOn = isOn();
    if (checkOn) {
        lightBulb.turnOff();
        this.on = false;
    } else {
            lightBulb.turnOn();
            this.on = true;
    }
  }
}
```