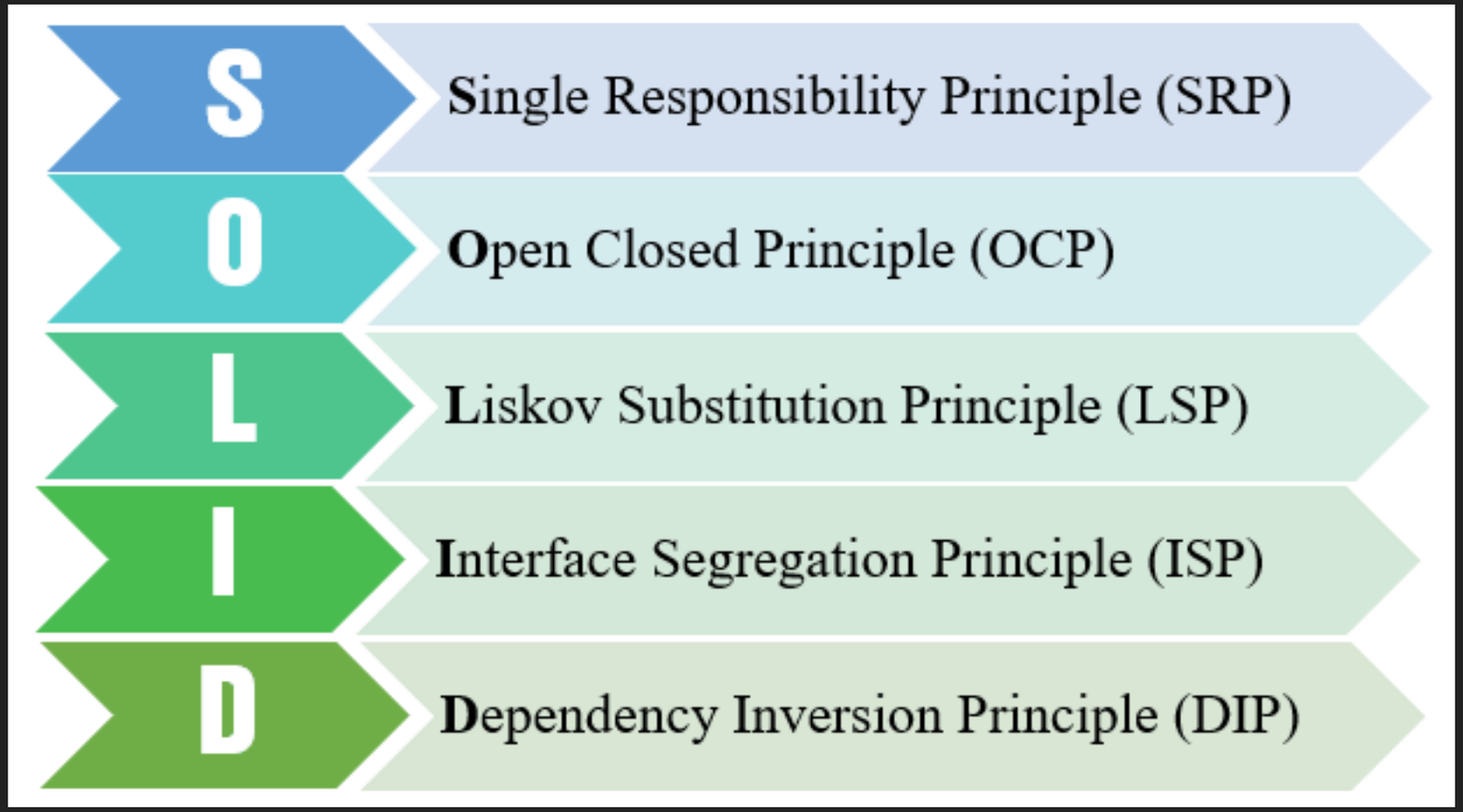
SOLID, DRY, KISS
PRAWO DEMETER, CODE SMELL

PRZYGOTOWAŁ: DAWID PLEŚNIARSKI

S.O.L.I.D

SOLID

- S.O.L.I.D to akronim przedstawiający zestaw wytycznych. Wytyczne te stosuje się podczas pisania programów w sposób obiektowy. Samo słówko *solid* można przetłumaczyć jako solidny, konkretny, mocny. Ta gra słów pewnie też miała spore znaczenie dla popularności samego akronimu.



'S' JAK SAMODZIELNY

- Zasada ta mówi, że każda z klas jest odpowiedzialna za jedną, konkretną rzecz.
- Starajmy się unikać pisania klas typu „szwajcarski scyzoryk"
  Klasa taka jest „do wszystkiego i do niczego"

Klasa AdultPerson zawiera 2 metody nie leżące w obowiązku tej klasy, sprawdzające poprawność adresu e-mail oraz czy osoba jest pełnoletnia. Prędzej czy później dojdzie do redundancji kodu jeśli będziemy chcieli użyć tych metod poza instancją klasy AdultPerson.

```java
public class AdultPerson {
    private String name;
    private String surname;
    private String emailAddress;
    private int age;

    public AdultPerson(String name, String surname, String emailAddress, int age) {
        this.name = name;
        this.surname = surname;
        if(isAdult(age)){
            this.age = age;
        }
        if(emailValidator(emailAddress)){
            this.emailAddress = emailAddress;
        }
    }
    private boolean emailValidator(String emailAddress){
        String regex = "^[\\w-_\\.+]*[\\w-_\\.]\\@([\\w]+\\.)+[\\w]+[\\w]$";
        return emailAddress.matches(regex);
    }
    private boolean isAdult(int age){
        return age >= 18;
    }
```

Rozdzielamy klasę AdultPerson

```java
public class AdultPerson {
    private String name;
    private String surname;
    private String emailAddress;
    private int age;

    public AdultPerson(String name, String surname, String emailAddress, int age) {
        this.name = name;
        this.surname = surname;
        this.age = age;
        this.emailAddress = emailAddress;
    }
```

W oddzielną klasę wrzucamy nasze metody

```java
public class PersonValidator {
    public boolean isEmailAddressValid(String emailAddress){
        String regex = "^[\\w-_\\.+]*[\\w-_\\.]\\@([\\w]+\\.)+[\\w]+[\\w]$";
        return emailAddress.matches(regex);
    }

    public boolean isPersonAdult(int age){
        return age >= 18;
    }

}
```

'O' JAK OPEN/CLOSED

- Open - otwarty na rozbudowę
- Closed - zamknięty na modyfikacje

Klasa Calculator jest zamknięta na rozbudowę.
W momencie dodania nowego kształtu np: Circle będziemy zmuszeni do modyfikacji metody area() w klasie Calculator.

```
public class Calculator {

    public int area(Object shape){
        if(shape instanceof Square){
            return ((Square) shape).a * ((Square) shape).a;
        } else if(shape instanceof Rectangle){
            return ((Rectangle) shape).a * ((Rectangle) shape).b;
        }
        return 0;
    }
}
class Square{
    int a;

    public Square(int a) {
        this.a = a;
    }
}
class Rectangle{
    int a;
    int b;

    public Rectangle(int a, int b) {
        this.a = a;
        this.b = b;
    }
}
```

Zła praktyka

- W takiej sytuacji warto użyć polimorfizmu.
Dzięki użyciu polimorfizmu i mechanizmu
dziedziczenia w dobry sposób dbamy o zasadę otwarty/ zamknięty, co pozwala dodawać coraz to nowe kształty. Każda nowa klasa rozszerzająca klasę abstrakcyjną Shape nadpisuje metodę area() z klasy Shape i implementuje ją na własny sposób.

- Nic nie stoi na przeszkodzie aby do naszych figur dodać Trójkąt i zaimplementować wzór na jego pole.

```
public class Calculator {

    public int area(Shape shape){
        return shape.area();
    }
}
class Square extends Shape{
    int a; //constructor
    @Override
    public int area() {
        return a * a;
    }
}
class Rectangle extends Shape{
    int a;
    int b; //constructor

    @Override
    public int area() {
        return a * b;
    }
}
abstract class Shape {
    public abstract int area();
}
```

```
public class Triangle extends Shape {
    int a;
    int h;
    public Triangle(int a, int h) {
        this.a = a;
        this.h = h;
    }
    @Override
    public int area() {
        return (a * h)/2;
    }
}
```

'L' JAK LISKOV SUBSTITUTION

- **Zasada podstawienia Liskov** mówi o tym, że w miejscu klasy bazowej można użyć dowolnej klasy pochodnej (zgodność wszystkich metod).
- Stosowanie się do tej zasady pozwala na dostarczenie
alternatywnych implementacji danej funkcjonalności
bez jakiejkolwiek zmiany kodu.

Przykład: posiadamy 3 kolekcje różnego typu, tworzymy metodę która ma na celu wyświetlić wszystkie elementy z danej kolekcji. Jak widać, metoda printCollection() jest w stanie wypisać elementy z 3 różnych typów kolekcji bez jej zmiany implementacji.

```java
public class DoSomethingWithCollection {

    public static void printCollection(Collection<String> someCollection){
        someCollection.stream()
                .forEach(System.out::println);
    }
}
```

```java
    public static void main(String[] args) {
        List<String> someList = new ArrayList<>();
        someList.add("asdf");
        someList.add("qwer");
        someList.add("zxcv");

        Set<String> someSet = new HashSet<>();
        someSet.add("asdf");
        someSet.add("qwer");
        someSet.add("zxcv");

        Queue<String> someQueue = new PriorityQueue<>();
        someQueue.add("asdf");
        someQueue.add("qwer");
        someQueue.add("zxcv");

        DoSomethingWithCollection.printCollection(someList);
        DoSomethingWithCollection.printCollection(someSet);
        DoSomethingWithCollection.printCollection(someQueue);

    }
```

PRZYPADKI ZŁAMANIA PODSTAWIENIA LISKOV:

- Źle rozplanowany mechanizm dziedziczenia
- Zastosowanie dziedziczenia bez mechanizmu polimorfizmu
- Klasy pochodne nadpisują metody klasy bazowej zastępując jej niepasującą logikę, np:
klasa abstrakcyjna Animal dostarcza metodę runAnimal(). O ile w przypadku psa ta metoda ma sens i zastosowanie, tak w przypadku np. ryby jest to nie logiczne.

```
abstract class Animal {
    String name; //get set
    public abstract void runAnimal();
}

class Dog extends Animal{

    String name; //get, set
    @Override
    public void runAnimal() {
        System.out.println("The dog is running 🐶");
    }
}

class Fish extends Animal{
    String name; //get set
    @Override
    public void runAnimal() {
        System.out.println("The fish cannot run :( 🐟");
    }
}
```

'I' JAK INTERFEJSY

- „I" pochodzi od Interface Segregation.
  Wytyczna ta mówi, abyś rozdzielał interfejs klasy na mniejsze fragmenty. Chodzi o uniknięcie sytuacji, w której klasa jest zmuszana do wprowadzenia zależności z interfejsu, których nie będzie używała.
- Lepiej zaimplementować kilka interfejsów niż implementować nie potrzebne metody.

```java
public interface ObjectFormatter {
    byte[] formatToPdf();
    String formatToXml();
    String formatToYaml();
}
```

```java

interface YamlFormatter {
    String formatToYaml();
}

interface PdfFormatter {
    byte[] formatToPdf();
}

interface XmlFormatter {
    String formatToXml();
}
```

```java
public class FormatToPdf implements ObjectFormatter {
    @Override
    public byte[] formatToPdf() {
        return new byte[0]; // potrzebuję
    }

    @Override
    public String formatToXml() {
        return null;    //nie potrzebuję
    }

    @Override
    public String formatToYaml() {
        return null;    //nie potrzebuję
    }
}
```

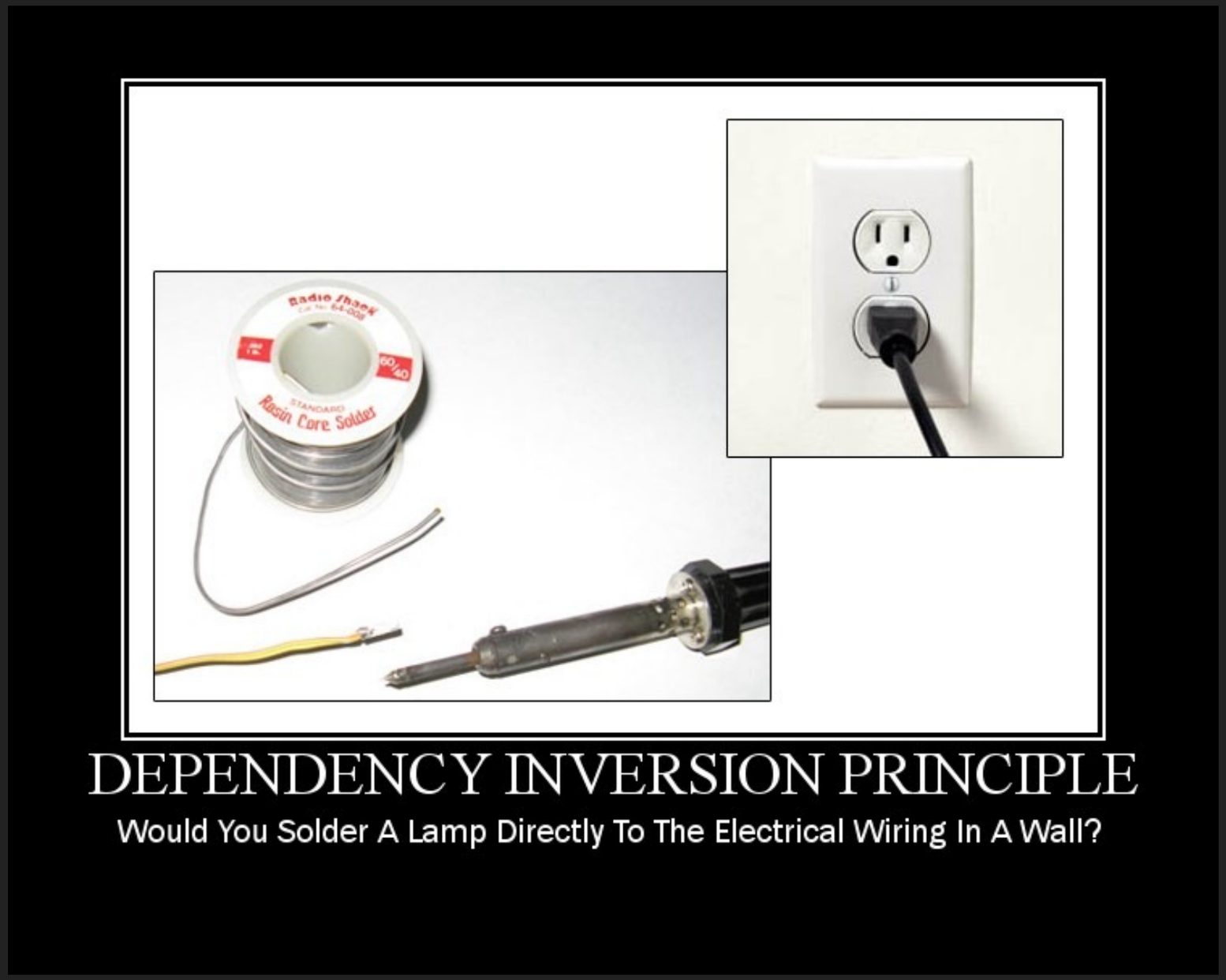
```java
public class FormatToPdf implements PdfFormatter {
    @Override
    public byte[] formatToPdf() {
        return new byte[0];
    }
}
```

'D' JAK DEPENDENCY INVERSION

- D pochodzi od Dependency Inversion Principle.
  Reguła ta mówi, że wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych.

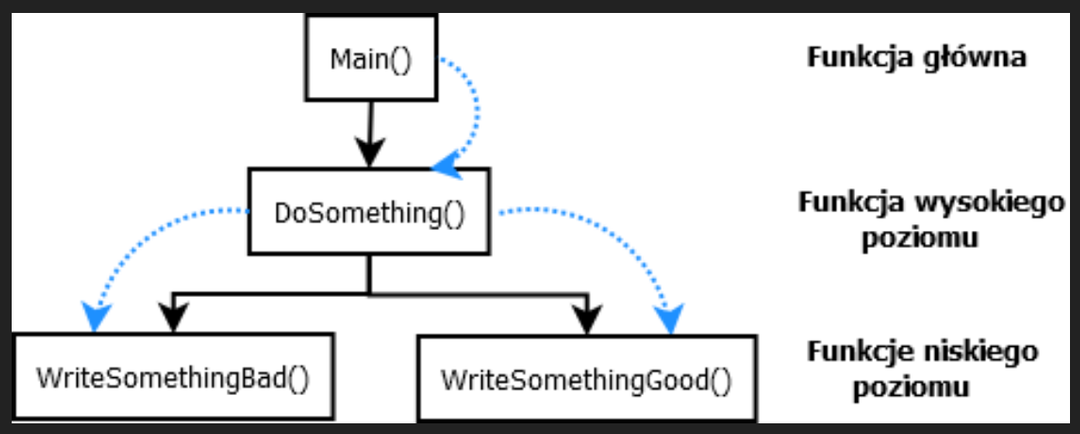
- Zasada ta polega na używaniu interfejsu polimorficznego wszędzie tam, gdzie jest to możliwe.

- Wszystkie zależności powinny w jak największym stopniu zależeć od abstrakcji a nie od konkretnego typu



- Na przykładzie obok została przedstawiona zła praktyka, ponieważ obsługuje tylko 2 nastroje: happy i sad. A co jeśli mamy też inne nastroje np. scared? Wtedy nasze 2 funkcje nie wystarczą. W tym momencie moduł wysokiego poziomu (doSomething) zależy od modułu niskiego poziomu (writeSomethingGood/Bad). Czyli zasada dependency inversion została złamana.

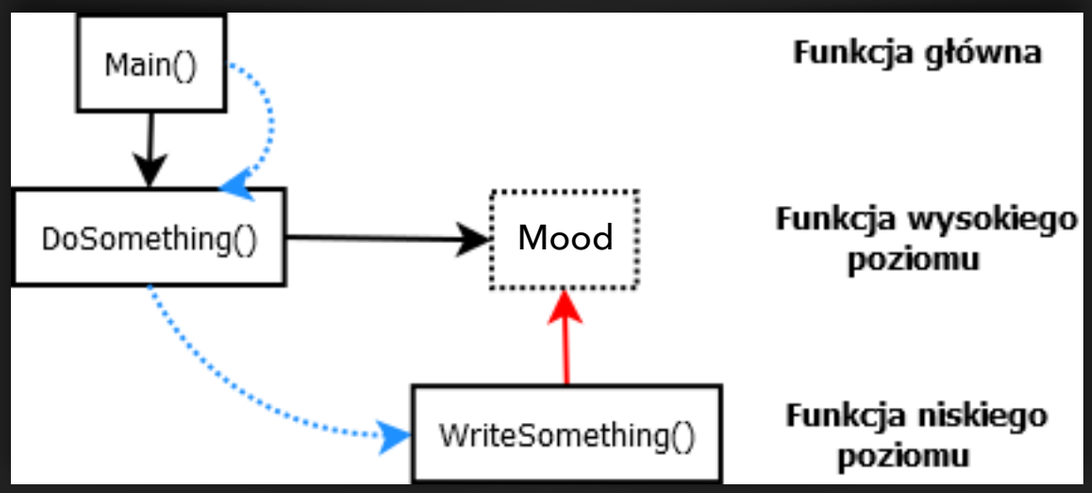
- Na diagramie poniżej czarne strzałki ukazują zależności między klasami, a niebieskie strzałki przepływ sterowania.




```
    public static void main(String[] args) {
        MoodLogic.doSomething( mood: "happy");
    }
}
class Logic{
    static void writeSomethingGood(){
        System.out.println(":)");
    }

    static void writeSomethingBad(){
        System.out.println(":(");
    }
}
class MoodLogic{
    static void doSomething(String mood){
        if(mood.equals("happy")){
            Logic.writeSomethingGood();
        }else if(mood.equals("sad")){
            Logic.writeSomethingBad();
        }
    }
}
```

- Do poprzedniego programu został dodany interfejs (abstrakcja), zamiast Stringa przekazujemy obiekt implementujący ten interfejs
- Do interfejsu Mood istnieje zależność z metody writeSomething().
  Czerwona strzałka oznacza *odwrócenie zależności*. Wskazuje ona w kierunku przeciwnym do przepływu sterowania.
- Aby rozszerzyć program o kolejne nastroje wystarczy zaimplementować interfejs Mood oraz nadpisać metodę writeSomething().




```java
public class Main {
    public static void main(String[] args) {
        ScaredMood scaredMood = new ScaredMood();
        MoodLogic.doSomething(scaredMood);
    }
}
interface Mood{
    void writeSomething();
}
class GoodMood implements Mood{
    @Override
    public void writeSomething() {
        System.out.println("I have good mood! 😄");
    }
}
class BadMood implements Mood{
    @Override
    public void writeSomething() {
        System.out.println("I have bad mood! 😢");
    }
}
class ScaredMood implements Mood{
    @Override
    public void writeSomething() {
        System.out.println("I'm scared! 🙀");
    }
}
class MoodLogic{
    static void doSomething(Mood mood){
        mood.writeSomething();
    }
}
```

DON'T REPEAT YOURSELF!

Dry Code

NIE POWTARZAJ SIĘ!

- DRY CODE to podstawowa reguła tworzenia oprogramowania.
  Wielokrotnie powielony kod nie tylko pogarsza czytelność naszego programu, lecz także skutecznie utrudnia jego późniejsze modyfikacje i tym samym utrudniamy życie sobie, oraz współpracownikom.
  Przykład:
  Jeśli wpadł Ci do głowy pomysł, żeby skopiować sekwencje if-ów, pętli etc. pomyśl nad stworzeniem abstrakcji (wspólny interfejs, funkcja, klasa, wzorzec projektowy np. „strategia"), który będziesz mógł wielokrotnie użyć.

- Zalety:
  Lepsza czytelność kodu oraz prostota pielęgnacji. W razie potrzeby, zmieniasz implementacje tylko w jednym miejscu.

- Wady: Brak

- Jak możemy uniknąć duplikacji?
  Prosty przykład:
  Tworzymy klasę Vehicle
  oraz klasę Motorcycle.
  Motocykl też jest pojazdem więc
  po co dwa razy pisać markę,
  model itp?
  Wystarczy rozszerzyć klasę o
  klasę Vehicle i przejąć konstruktor
  z klasy po której dziedziczymy.
  Mając klasę Vehicle możemy na
  jej podstawie utworzyć też klasę
  Car, Truck itp.

- Unikniemy duplikacji
  przestrzegając zasad S.O.L.I.D.
  W szczególności zasady Open/
  Closed i podstawienie Liskov.

```java
public class Vehicle {
    String mark;
    String model;
    Integer productionYear;
    String vinNumber;
    EngineType engineType;

    public Vehicle(String mark, String model, Integer productionYear,
                   String vinNumber, EngineType engineType) {...}
}
enum EngineType {
    Diesel,
    Petrol,
    Hybrid,
    Electric,
    Lpg
}
class Motorcycle extends Vehicle{
    String typeOfDrive;
    public Motorcycle(String mark, String model, Integer productionYear,
                      String vinNumber, EngineType engineType, String typeOfDrive) {
        super(mark, model, productionYear, vinNumber, engineType);
        this.typeOfDrive = typeOfDrive;
    }
}
```

KEEP IT SIMPLE, STUPID!

Reguła KISS

KEEP IT SIMPLE!

- Kod który piszemy, ma być zrozumiały dla maszyny oraz prosty w czytaniu dla programisty
- Program działa najlepiej gdy jego kod polega na prostocie, a nie złożoności
- Staraj się, aby Twój kod wymagał jak najmniej myślenia podczas analizy
- Jeśli za miesiąc wracasz do swojego kodu i nie wiesz co tam się dzieje, to pomyśl co czuje osoba która widzi ten kod pierwszy raz.

KISS

Keep. It. Simple. Stupid.

- Ponadto regułę KISS warto rozszerzyć o dwie zasady:
- **Code for the Maintainer**
  Pisz kod tak, jakbyś robił to dla kogoś, kto będzie ten kod później utrzymywał
  Zadbaj aby Twój kawałek kodu nie wymagał dużo czasu do rozszyfrowania
- **Reguła Skauta**
  Zawsze pozostaw po sobie kod czystszy niż go zastałeś!
- Zalety stosowania KISS:
  Szybciej i prościej zrozumieć kod
  Zmniejsza prawdopodobieństwo wystąpienia bugów
  Łatwość pielęgnacji kodu

TALK ONLY WITH FRIENDS 🤝

Prawo Demeter

PRAWO DEMETER

- Prawo Demeter nakazuje odwoływania się tylko do “bliskich przyjaciół” czyli obiektów, które są bardzo dobrze znane danemu obiektowi.

- Trzymając się prawa Demeter oddzielamy od siebie dalekie części kodu, które nigdy nie powinny się ze sobą komunikować

```
    private Additions addition;

    public void newPizza(Pizza pizza){
        double pizzaPrice;
        double additionPrice;
        double totalPrice;
        // Możemy odwoływać się do własnych metod
        sayHelloToCustomer();
        //Możemy wywołać metody obiektów które stworzymy w klasie
        addition = Additions.Salami;
        additionPrice = addition.getAdditionPrice();
        // Możemy wywoływać metodę z obiektu podanego w metodzie
        pizzaPrice = pizza.getPizzaPrice();

        totalPrice = pizzaPrice + additionPrice;
        System.out.println("Total price" + totalPrice);
    }
    void sayHelloToCustomer(){ System.out.println("Welcome to our restaurant!"); }
}
```

- Klasyczny przykład złamania zasady Demeter.

```java
public class Pizzeria {

    private Additions addition;

    public void newPizza(Pizza pizza){
        double pizzaPrice;
        double additionPrice;
        double totalPrice;
        // Możemy odwoływać się do własnych metod
        sayHelloToCustomer();
        //Możemy wywołać metody obiektów które stworzymy w klasie
        addition = Additions.Salami;
        additionPrice = addition.getAdditionPrice();
        // Możemy wywoływać metodę z obiektu podanego w metodzie
        pizzaPrice = pizza.getPizzaPrice();

        // Złamanie prawa demeter
        pizza.getAnotherPizza().getAnotherAnotherPizza(); // obiekt A rozmawia z obiektem B oraz C

        totalPrice = pizzaPrice + additionPrice;
        System.out.println("Total price" + totalPrice);
    }
    void sayHelloToCustomer(){ System.out.println("Welcome to our restaurant!"); }
}
```

CODE SMELL 💩

Coś tu śmierdzi...

CODE SMELL

- Nazwa Code Smell odnosi się do pewnych cech programu, mówiących o złym sposobie implementacji oraz będące sygnałem do zrobienia refaktoryzacji.

- **Przykłady Code Smell:**

- Bardzo długie metody

- Duże klasy, posiadające zbyt wiele odpowiedzialności (złamanie zasady Single Responsibility z S.O.L.I.D

- Zazdrość o kod (feature envy) gdy dana metoda w bardzo dużym stopniu korzysta z danych z innej klasy. Taka metoda powinna zostać przeniesiona do klasy z której korzysta

- Lazy Class (leniwa klasa) gdy dana klasa posiada bardzo niewielki zakres odpowiedzialności

- Powielanie kodu (niestosowanie zasady D.R.Y)

- Contrived Complexity czyli zostały użyte skomplikowane wzorce projektowe, gdzie użycie prostszych byłoby wystarczające, oraz zgodne z regułą KISS

- Nazwy zmiennych i metod, które nic nam nie mówią.

- Klonowanie klas - pisanie niemalże identycznych klas zamiast użycia dziedziczenia

- Przykłady Code Smell (Nie róbcie tego w domu)

1. Sklonowane klasy
2. Publiczne pola
3. Niezrozumiałe nazwy zmiennych
4. „Szminkowanie” niezrozumiałego kodu komentarzami
5. Nazwy metod nie mówiące nic
6. Zagnieżdżone ify
7. Brak klamer w if

```java
public class Human {
    public String name;
    public String lastName;
    public int age;
    public String prop;

    public Human(String name, String lastName, int age, String prop) {...}

    // check if the data contains special characters
    boolean check(String name, String surName){
        if(name.matches( regex: "[a-z]")){
            if(surName.matches( regex: "[a-z]")){
                return true;
            }
        }
        return false;
    }
    // check if people adult
    boolean adult(int age){
        if(age >= 18) return false;
        return false;
    }
}

class Person {
    private String name; //get set
    private String lastName; //get set
    private int age; //get set
    private String prop; //get set
}
```

PYTANIA?

DZIĘKUJĘ ZA UWAGĘ

Repozytorium z projektem:

https://github.com/dawidplesniarski/programming-principles-lecture