**SOLID**

* **SOLID** - serie de principii ce pot fi aplicate asupra codului ca el sa fie cat mai bun
* **S - Single-resposnability** - se poate aplica la metode,clase,module etc. si inseamna ca fiecare dintre ele trebuie sa fac un singur lucru concret, adica sa aiba o singura responsabilitate. De ex, fie o clasa Book, ce are fielduri, getters, setters, toString. Asa e bine, caci clasa e doar pentru a stoca date. Insa, daca mai adaugam metode, ca printTextUpperCase(), replaceText() etc. incalcam principiul de single responsability. Aceste metode trebuie separate intr-o noua clasa.
* **O - open/closed - adica open for extension, closed for modification**. De exemplu, cream o clasa Guitar:

public class Guitar {

private String make;

private String model;

private int volume;

//Constructors, getters & setters

}

Dupa un timp, decidem ca vrem sa adaugam niste chestii in plus la guitara, ca flame, adica daca are culoare in flacari sau nu. Nu e o idee buna sa modificam clasa Guitar, caci am putea sa influentam modul in care ea functioneaza acum, de aceea, cel ami bine extindem clasa in una noua:

public class SuperCoolGuitarWithFlames extends Guitar {

private String flameColor;

//constructor, getters + setters

}

* **L - Liskov substitution** - Daca clasa A este un subtip a clasei B, atunci trebuie sa fim capabili sa inclouim B cu A fara a influenta programul. De exemplu: fie ca avem clasa Bird si metoda fly():

public class Bird{

public void fly(){}

}

Acum, clasa Duck extinde Bird si duck poate zbura, nu e nimic anormla ca acceseaza fly()

public class Duck extends Bird{}

Dar, si strutul(Ostrich) e pasare, dar el nu poate zbura

public class Ostrich extends Bird{}

clasa Ostrich incalca principiul Liskov Substitution, caci putem inlocui Bird cu Ostrich, dar stricam logica

Acum, sa zicem ca vrem sa cream o aplicatie ce se contecteaza la o baza de date

Cream:

interface Connection{

void connect();

}

Si cream clasa

class DataBaseConnection implements Connection{

@Override

void connect() { }

}

Si, cand vom folosi @Autowired la vreo Connection, si daca se va injecta un DataBaseConnection object, totul e bine.

Dar, daca cream sa zicem apoi un alt tip de conexiune, gen

class ServerConnection implements Connection{

@Override

void connect() { }

}

Si daca vom da @Autowired si asta va fi unica clasa, programul se va strica, caci nu se face conexiune la BD, ci la Server. De asta, ca solutie, trebuie interfete inca separate pentru ServerConnection si DataBaseConnection

* **I - Interface segregation** - interfetele mari trebuie descompuse in interfete mici. Asa ne asiguram ca clasele care implementeaza anumite metode sunt bazate strict anume pe acele metode. Daca o clasa implementeaza o inerfata ce are o metoda de care ea nu are nevoie, problema e ca clasa va trebui oricum sa suprascrie metoda.
* **D - dependecy inversion** - se bazeaza pe decuplarea la software modules. Asa, in loc de a folosi propriu zis obiecte, folosim interfete si deci high-level modules nu depind de low-leves modules.

De ex:

Fie clasa Windows98Machine:

public class Windows98Machine {

private final StandardKeyboard keyboard;

private final Monitor monitor;

public Windows98Machine() {

monitor = new Monitor();

keyboard = new StandardKeyboard();

}

}

Vedem ca noi oferim un obiect de tip Monitor si StandardKeyboard, care sunt bune pentru Windows 98

Totusi, facand asta, am cuplat clasele Windows98Machine, Monitor si StandardKeyboard impreuna. Acum, apare un windows nou, si Monitor si StandardKeyboard vor fi imbunatatite, dar problema e ca ele nu sunt cele care au fost pentru Windows98. Solutia este de a folosi interfete pentru Monitor si Keyboard in loc de clase concrete:

public class Windows98Machine{

private final Keyboard keyboard;

private final Monitor monitor;

public Windows98Machine(Keyboard keyboard, Monitor monitor) {

this.keyboard = keyboard;

this.monitor = monitor;

}

}

interfate Keyboard{ }

class StandardKeyboard implements Kwyboard

interface Monitor { }

class StandardMonitor implements Monitor {

Acum clasele sunt decuplate si comunica prin abstractizare. Putem usor schimba tipul de Keyboars sau de Monitor

Backed

Admin Dashboard Page:

1:00:00