

SOLID

Principles

What is SOLID?

SOLID is an acronym for five design principles in sw development:

Single Responsibility: A class should do one thing and hence a single reason to change.

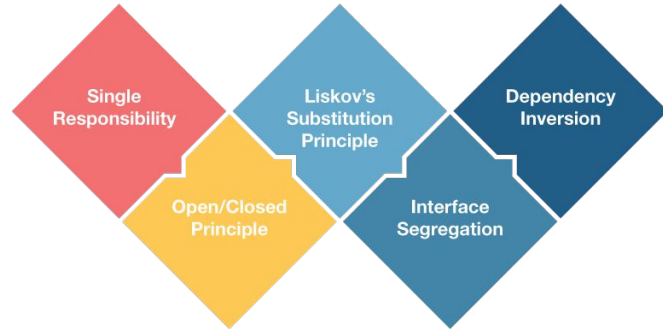
Open - Closed: Open for extension and closed for modification

Liskov Substitution: Subclasses should be substitutable for their base classes

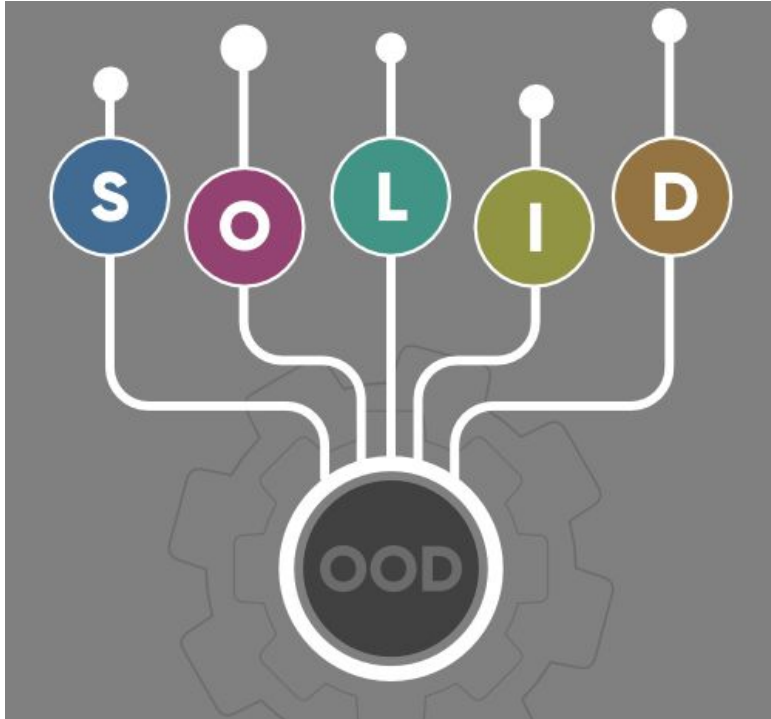
Interface Segregation: Keep interfaces separate. Clients should not be forced to implement a function they do not need.

Dependency Inversion: Classes should depend on interfaces or abstract classes instead of concrete classes and functions

S.O.L.I.D.



Background



- The five design principles was first published by Robert C. Martin (Uncle Bob) in 2000 in his paper *Design Principles and Design Patterns*.
- Intention is to make Object Oriented designs more understandable, flexible, and maintainable.
- The SOLID acronym was introduced later, around 2004, by Michael Feathers

Single Responsibility

A Class should only have one responsibility and only one reason to change.

Why?

1. **Testing** – A class with one responsibility will have far fewer test cases.
2. **Lower coupling** – Less functionality in a single class will have fewer dependencies.
3. **Organization** – Smaller, well-organized classes are easier to search than monolithic ones.

```
1 public class Book {
2
3     private String name;
4     private String author;
5     private String text;
6
7     //constructor, getters and setters
8
9     // methods that directly relate to the book properties
10    public String replaceWordInText(String word, String replacementWord){
11        |   return text.replaceAll(word, replacementWord);
12    }
13
14    public boolean isWordInText(String word){
15        |   return text.contains(word);
16    }
17 }
18
```

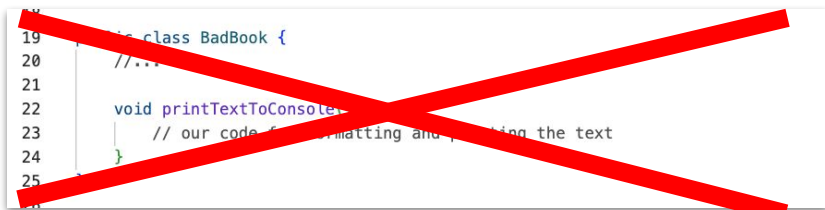
Consider this book class

Single Responsibility (2)

Need a way of printing the content

Might be tempted to add method to the class.

```
19 class BadBook {
20     //...
21
22     void printTextToConsole() {
23         // our code for formatting and printing the text
24     }
25
26 }
```



We don't do that!

Rather, we create a new class for this purpose:

```
26
27 public class BookPrinter {
28
29     // methods for outputting text
30     void printTextToConsole(String text){
31         //our code for formatting and printing the text
32     }
33
34     void printTextToAnotherMedium(String text){
35         // code for writing to any other location..
36     }
37 }
38
```

Open - Closed

Classes should be open for extension but closed for modification..

Why?

In doing so, we stop ourselves from modifying existing code and causing potential new bugs in an otherwise happy application. (*)

(*) Not when fixing bugs in existing code

```
1
2 public class Guitar {
3
4     private String make;
5     private String model;
6     private int volume;
7
8     //Constructors, getters & setters
9 }
10
```

```
10
11 ∨ public class SuperCoolGuitarWithFlames extends Guitar {
12
13     private String flameColor;
14
15     //constructor, getters + setters
16 }
17
```

Liskov Substitution

If class A is a subtype of class B, we should be able to replace B with A without disrupting the behavior of our program.

Consider a Car interface that all cars should implement:

```
1
2 public interface Car {
3     void changeGear();
4     void accelerate();
5 }
6
```

```
7  public class PetrolCar implements Car {
8      private GearExchange gearExchange;
9      private Engine engine;
10
11     // Constructors, getters, setters
12
13     public void changeGear() {
14         gearExchange.disconnect();
15         gearExchange.setTo(1);
16         gearExchange.connect();
17     }
18
19     public void accelerate() {
20         engine.increasePower(1000);
21     }
22 }
23
```

Liskov Substitution (2)

Implementing an Electric car does not have a gearbox. ElectricCar can therefore not implement changeGear()

```
23
24 public class ElectricCar implements Car {
25
26     public void changeGear() {
27         throw new AssertionError("No gear exchange available");
28     }
29
30     public void accelerate() {
31         engine.increasePower(1000);
32     }
33 }
34
```


Interface Segregation

Larger interfaces should be split into smaller ones.

Why?

we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

Let's consider an interface for a BearKeeper:

```
3
4 public interface BearKeeper {
5     void washTheBear();
6     void feedTheBear();
7     void petTheBear();
8 }
9
```

While petting bears might seem fun, we might not want to do that. Better to segregate the interfaces:

```
10 public interface BearCleaner {
11     void washTheBear();
12 }
13
14 public interface BearFeeder {
15     void feedTheBear();
16 }
17
18 public interface BearPetter {
19     void petTheBear();
20 }
21
22 public class BearCarer implements BearCleaner, BearFeeder {
23
24     public void washTheBear() {
25         //I think we missed a spot...
26     }
27
28     public void feedTheBear() {
29         //Tuna Tuesdays...
30     }
31 }
```

Dependency Inversion

Refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.

Consider this class:

```
2  ∨ public class Windows98Machine {
3
4      private final StandardKeyboard keyboard;
5      private final Monitor monitor;
6
7  ∨  public Windows98Machine() {
8      |      monitor = new Monitor();
9      |      keyboard = new StandardKeyboard();
10     |
11     |
12     | }
13     }
```

- tightly coupled with *StandardKeyboard* class

```
14
15     public interface Keyboard { }
16
17
```

```
17
18     public class Windows98Machine{
19
20         private final Keyboard keyboard;
21         private final Monitor monitor;
22
23         public Windows98Machine(Keyboard keyboard, Monitor monitor) {
24             this.keyboard = keyboard;
25             this.monitor = monitor;
26         }
27     }
```

Dependency injection to enable Keyboard dependency. Modify also the *StandardKeyboard* for this.

```
28
29     public class StandardKeyboard implements Keyboard { }
30
```

Why not?

criticism

So What?

What is my conclusion?

Questions?

References

- Design Principles and Design Patterns (Robert C Martin):
http://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf
- Baeldung: <https://www.baeldung.com/solid-principles>
- SOLID design principles explained:
<https://medium.com/bgl-tech/what-are-the-solid-design-principles-c61feff33685>
- SOLID Principle in Programming: Understand With Real Life Examples:
<https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/>
- Deconstructing SOLID (Ted Kaminski): <https://www.tedinski.com/2019/04/02/solid-critique.html>

Thank you

How to contact me:

sten.johnsen@bouvet.no

@stenjo on Twitter