UNIVERSITY OF MISKOLC,
FACULTY OF MECHANICAL ENGINEERING
AND INFORMATION TECHNOLOGY

Software Technology Lab

GEIAL316-B2a

Software design patterns

Tamás Tompa, PhD
assistant professor
Department of Information Technology

Miskolc, 2026

OOP

- A software based **on the OO (object-oriented) paradigm** is built **from the totality of objects communicating with each other**

- **Object:**
  - **an independent entity** with properties and attributes
  - real world (a part of it) is a **model**
  - **has state**: current values of fields
  - **behavior**: actions can be performed on it, which changes its state
  - **has a relationship** with other objects

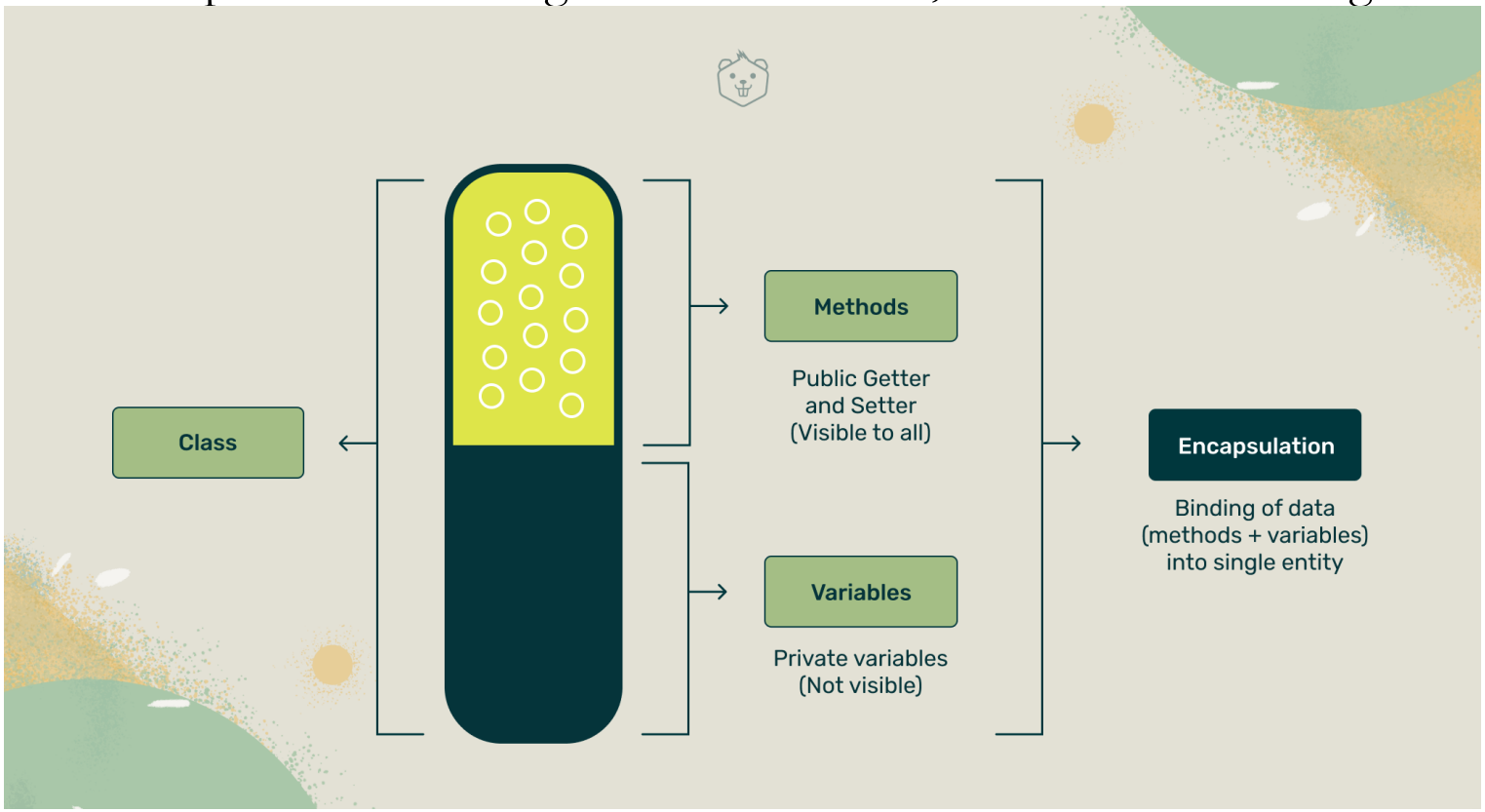
Forrás: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

OOP Principles

- **Encapsulation**
  - the class encloses the data of the given type of object and the operations working with it into a unit, and treats it as a single unit
- **Information hiding**
  - the implementation of the operations of the class is not visible outside the class
- **Inheritance**
  - a class can be created by being a descendant of another class, which thus inherits the properties of the parent class
- **Polymorphism**
  - polymorphism, the operation of certain behaviors depends on the environment, the operation of a method depends on the given class

OOP Principles

- **Encapsulation**
  - **ahe class encloses the data of the given type of object** and the operations working with it into a unit, and treats it as a single unit



Source: https://www.crio.do/blog/encapsulation-in-java/

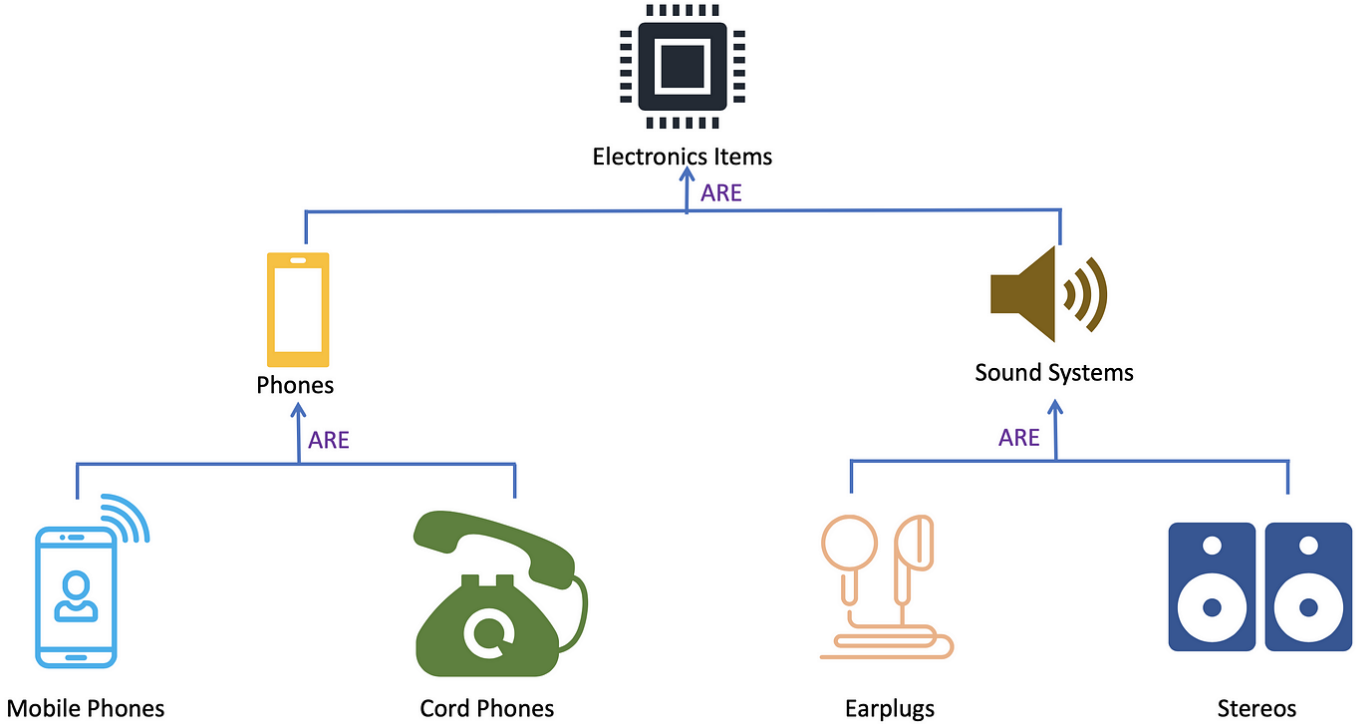
OOP Principles

- **Information hiding**
  - **the implementation of the operations of the class is not visible outside the class**

Source: https://www.infobrother.com/Tutorial/C++/C++_Information_Hiding

OOP Principles

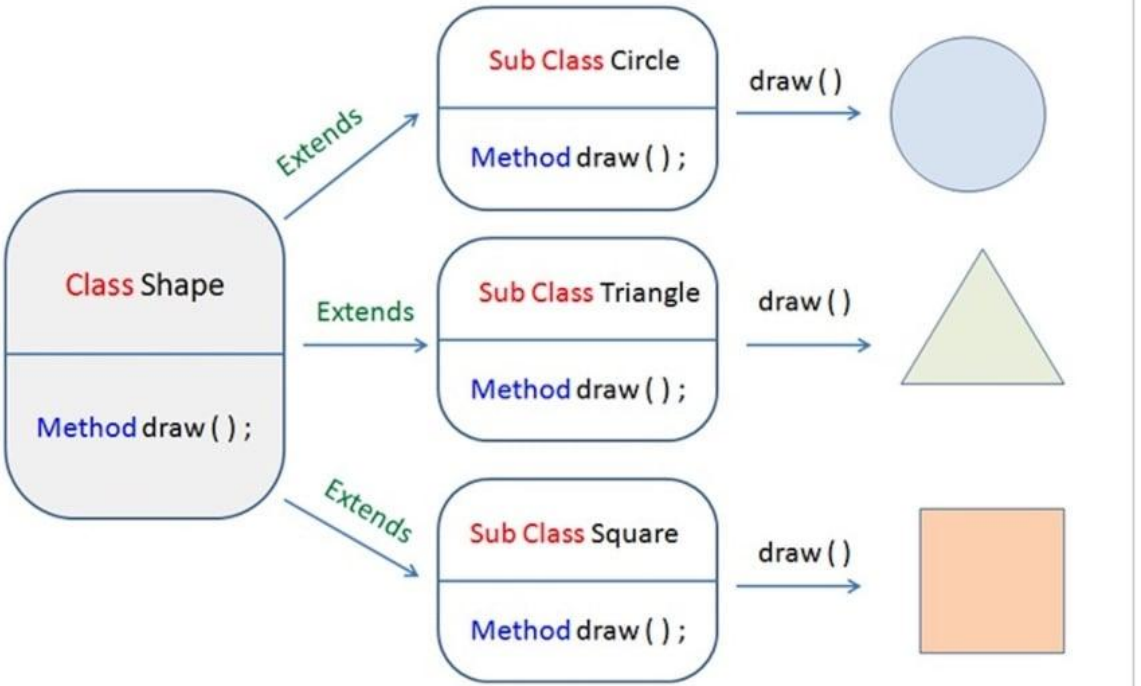
- **Inheritance**
  - **a class can be created by being a descendant of another class,** which thus inherits the properties of the parent class



Forrás: https://towardsdatascience.com/how-to-code-inheritance-in-java-beginners-tutorial-in-oop-d0fc0a71be98

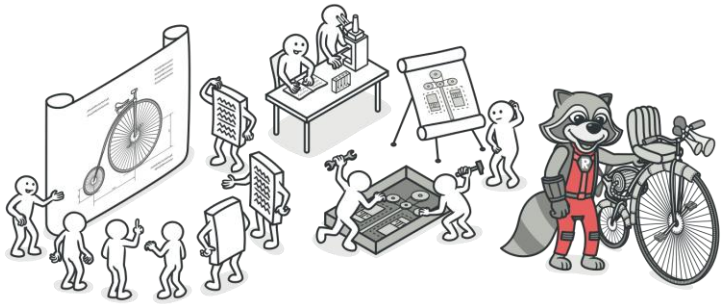
OOP Principles

- **Polymorphism**
  - polymorphism, **the operation of certain behaviors depends on the environment**, the operation of a method depends on the given class



Forrás: https://dev.to/divide_r_conquer/three-characteristics-of-oop-polymorphism-1lna

Design Patterns



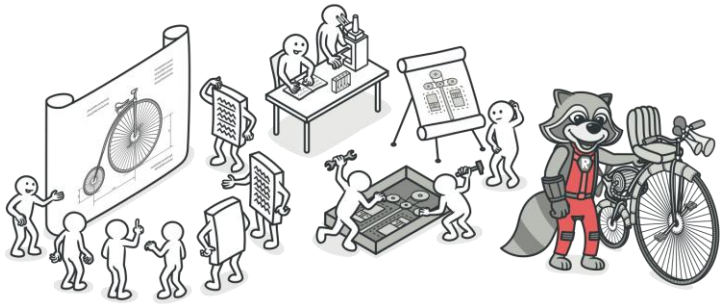
"Each pattern describes a problem that crops up again and again in our environment, and then describes the core of the solution in such a way that the solution can be used a million times without ever doing the same thing twice."

Christopher Alexander

- Christopher Alexander's **suggestion in the construction of buildings and cities that it should be implemented based on patterns**
  - but the same **is true for object-oriented software**
  - objects and surfaces instead of walls and doors
- In the case of complex systems, it is very important
- **Defined templates for structure**
  - lots of existing template (Abstract factory, Adapter, Composite, Decorator, Factory method etc.)

Source: https://refactoring.guru/design-patterns

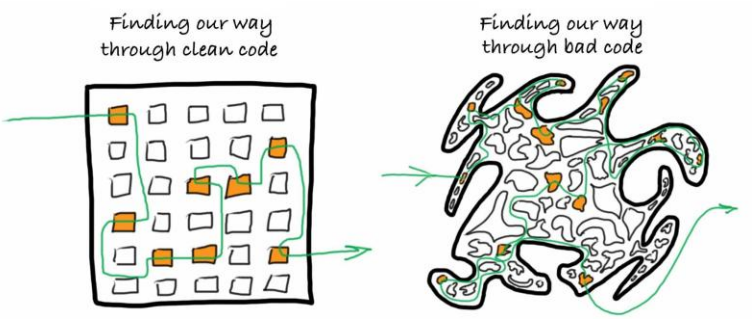
Design Patterns



- **Elements of a design pattern**
  - **Name: can be referred to**, refers to the problem, its solution and the consequence
  - **Problem: when to apply** the given pattern, describes the problem and its context (a set of conditions when the given pattern can be applied)
  - **Solution:** description of the elements with their powers, relationships and opportunities for cooperation **that build the plan**. Specifies a template, provides an abstract description of the design problem
  - **Consequence:** advantages **and disadvantages of using the given sample** (storage space, time use, recyclability)

Causes of their formation

- **Bad code**
  - Have we ever seen this? ☺
  - more time to understand than to rewrite...
  - difficult to explain how its work
  - unnecessary iterations
  - variables and functions names are meaningless
  - I modify it a little → everything goes wrong



- **Short deadlines**
  - → fast solutions
    - "It will be good that way"
    - "It works, doesn't it?"
    - "I'll rewrite it later"
    - "Temporarily (forever...) that's good"

Source: https://lvivity.com/how-to-write-good-code

Causes of their formation

- **Code Rot**
  - **The source code gradually becomes obsolete**, unmaintainable, and increasingly difficult to work with
    - technologies used are becoming obsolete
    - complexity increases, transparency decreases
    - the code becomes fragmented (spaghetti code, dea
    - no tests or updates
    - they add new features without any thought
    - "It's okay now, I'll rewrite it later"
    - a little hake here, a little hake there, many if many for nestled together
      - because of the quick fixes, the whole code has to be rewritten later because there are too many compromises
    - it happens over a long time, iterate by iterate
  - **as a result, after a while, the code starts to "rot"**
    - → result: an opaque, unreadable code is created

Causes of their formation

- **Code Rot**

  - at the level of dependencies, new things appear
    - unplanned
    - the consequence of „hecking”
  - use of old libraries and technologies that are no longer compatible with the new versions
  - documentation is deprecated because code changes, but descriptions aren't updated
  - prevention
    - refactoring
    - automated testing
    - update dependencies
    - write clean code
    - appropriate documentation

Causes of their formation

- **Code Rot Example**

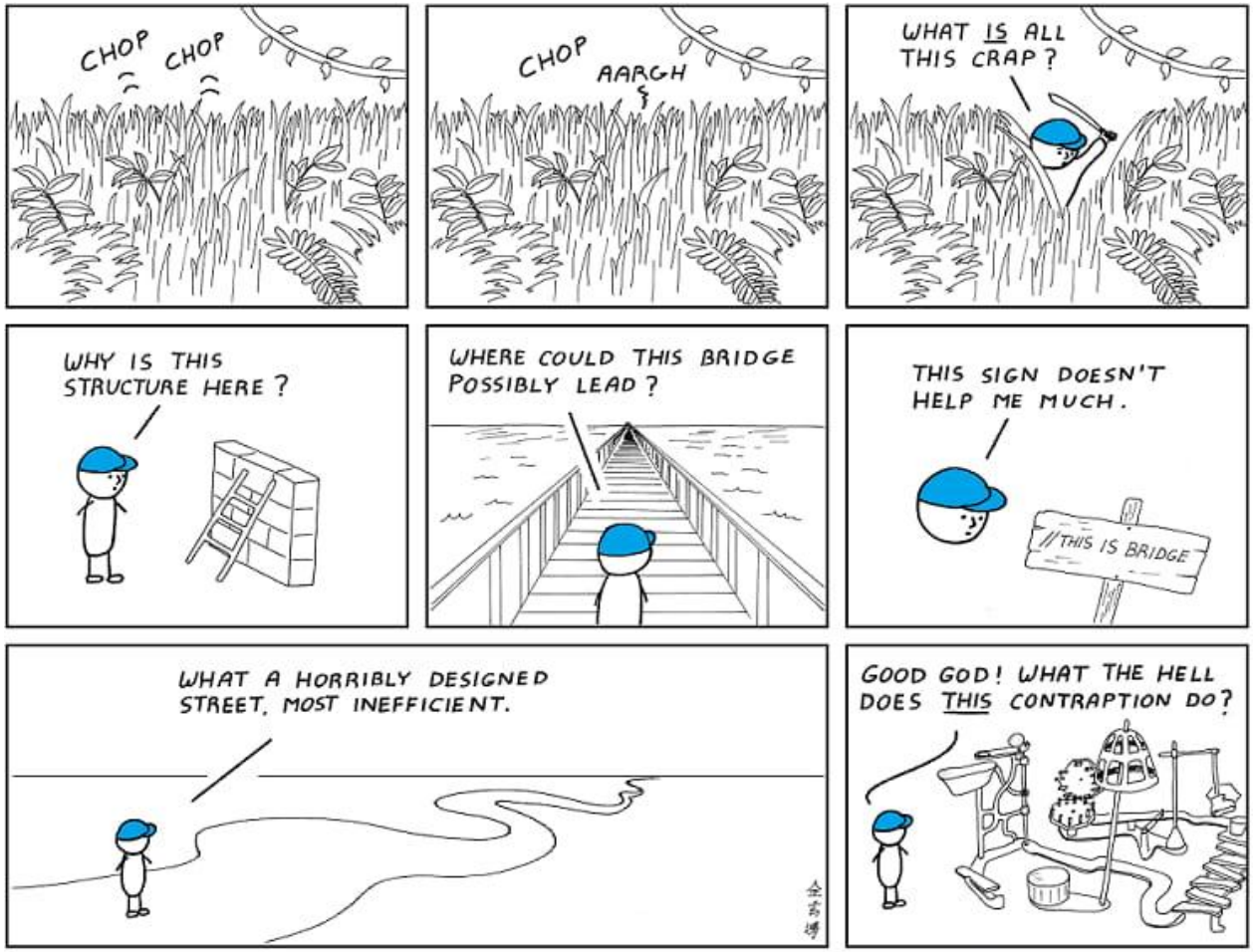
```
def calculate_price(price, tax):
    return price + (price * tax)
```

```
def calc_price(price, tax, discount=0, currency="$"):
    final_price = price + (price * tax)
    if discount > 0:
        final_price -= discount

    print(f"{currency}{final_price}")
    return final_price
```

```
def calculate_price(price, tax, discount=0):
    return price + (price * tax) - discount

def format_price(amount, currency="$"):
    return f"{currency}{amount:.2f}"
```

Causes of their formation

- **Why do we need good code?**

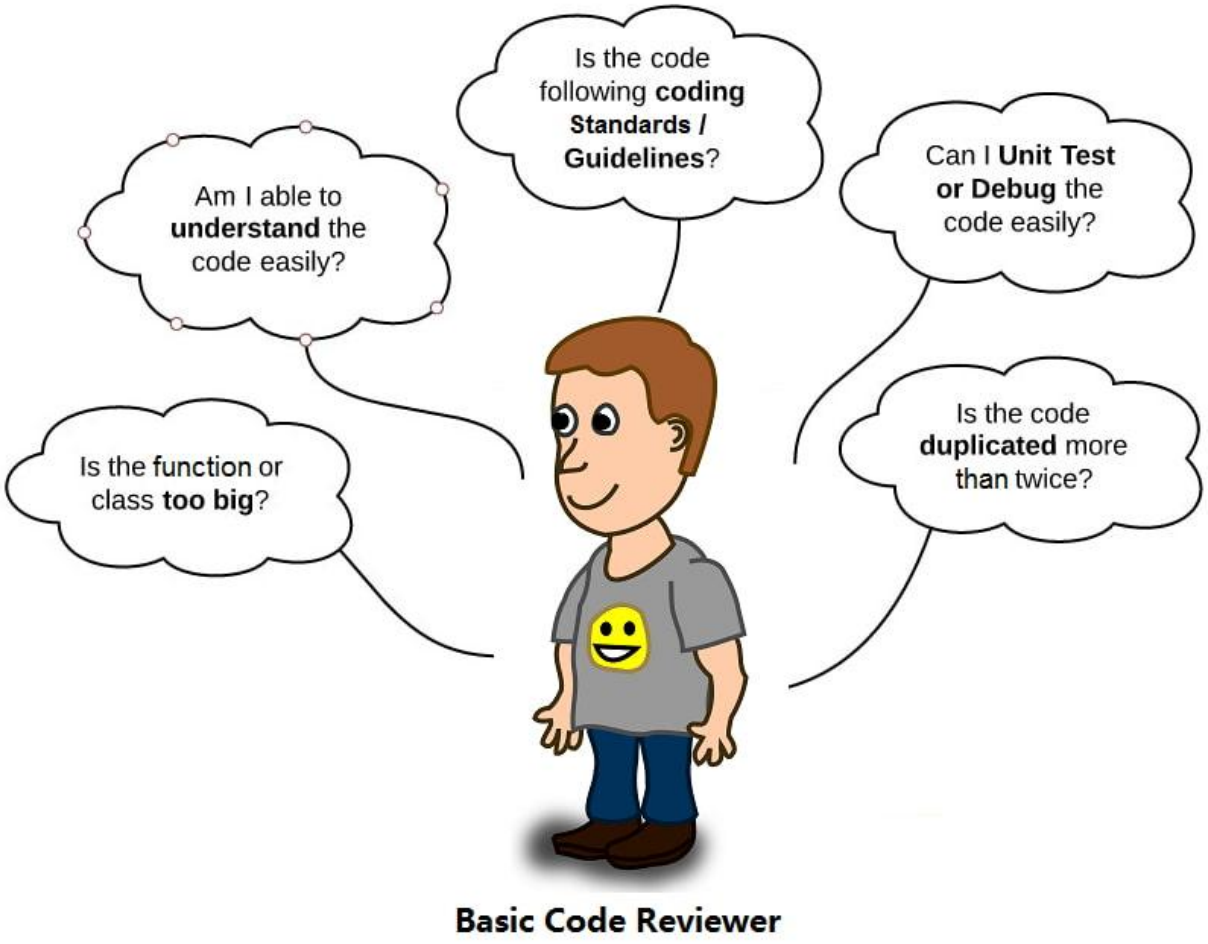


I hate reading other people's code.

Source: https://lvivity.com/how-to-write-good-code

Causes of their formation

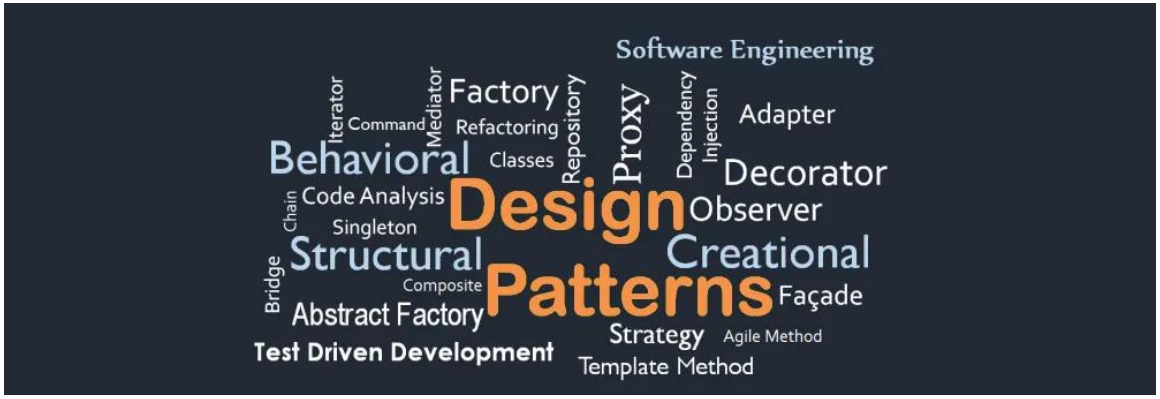
- **Is the code good?**



Source: https://lvivity.com/how-to-write-good-code

Design, patterns, rules

- **Some design patterns, rules**
  - **KISS (Keep it simple, stupid)**
  - **Demeter's Law**
  - **Separation of Concerns (SoC)**
  - **DRY (Don’t Repeat Yourself)**
  - **SOLID principles**

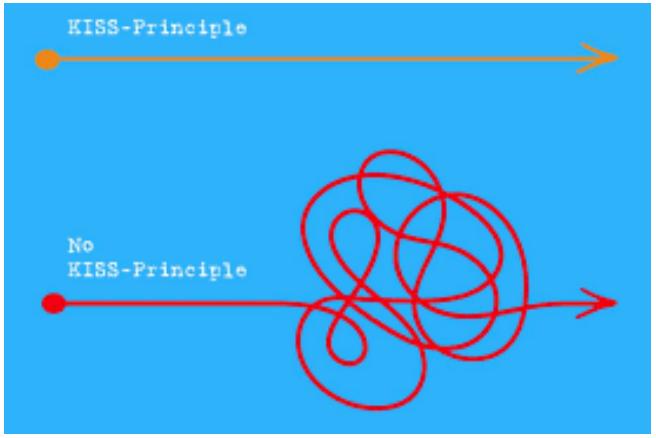


Source: https://www.nilebits.com/blog/2022/04/software-design-patterns/

KISS

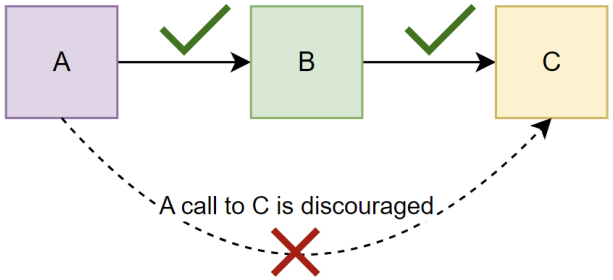
K.I.S.S
KEEP.IT.SIMPLE, STUPID!

- **Keep it simple, stupid**
  - 1960s, Navy
  - it cames from aeronautical engineer Kelly Johnson (1910–1990)
  - usually the simplest solutions are the best
  - everything should be done as simply as possible
  - avoiding unnecessary complexity
  - variable names describe exactly what value they store
  - method names reflect the purpose of the method
  - comment only where necessary
  - avoiding global variables
  - etc...



Forrás: https://medium.com/illumination/the-secret-to-success-keeping-it-simple-with-the-kiss-principle-5442b0b99b3f

https://dev.to/kwereutosu/the-k-i-s-s-principle-in-programming-1jfg

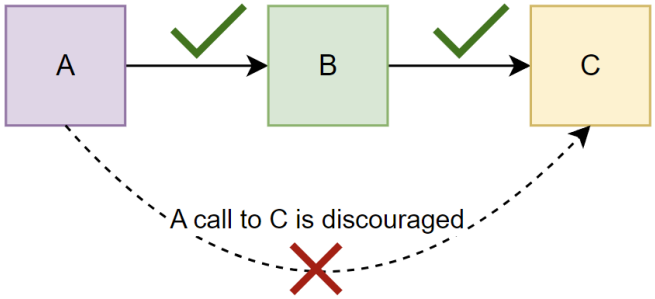


Demeter's Law

- **Don't talk to strangers**
  - **objects should avoid accessing the internal data and methods of other objects, an object can only interact with its direct dependencies**
  - restrict object messaging
  - narrows down the range of methods that can be called
  - minimizes communication between departments
  - the code will be less dependent on internal implementation details
  - 5 rules
  - e.g.: `Gamer`, `Weapon` classes
    - `Gamer`: `Weapon` adattag, `attack()` method, `Weapon`: `use()` method
    - Gamer class calls weapon.use() directly in its attack() method
    - Gamer class only communicates with the directly related Weapon object

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Demeter's Law



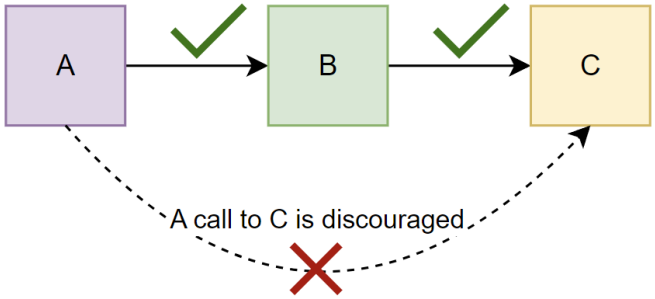
- **Rule 1**
  - **a Class C X method can only invoke C methods**

```
class Greetings {
    String generalGreeting() {
        return "Welcome" + world();
    }

String world() {
    return "Hello World";
    }
}
```

  - `generalGreeting()` method calls the `world()` method in the same class, this is correct because they belong to the same class

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Demeter's Law



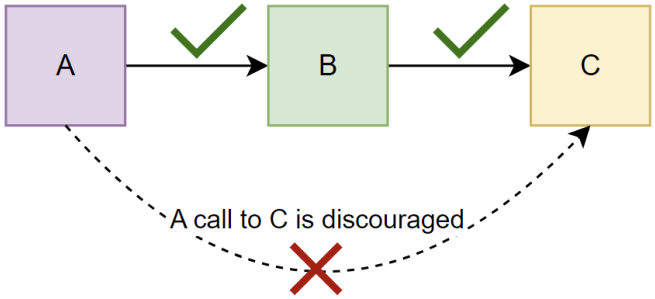
- **Rule 2**
  - **a method X of a class C invokes only the methods of the object created by X**

```
String getHelloBrazil() {
    HelloCountries helloCountries = new HelloCountries();

    return helloCountries.helloBrazil();
}
```

  - the `getHelloBrazil()` method created the object, so it can invoke its methods

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Demeter's Law



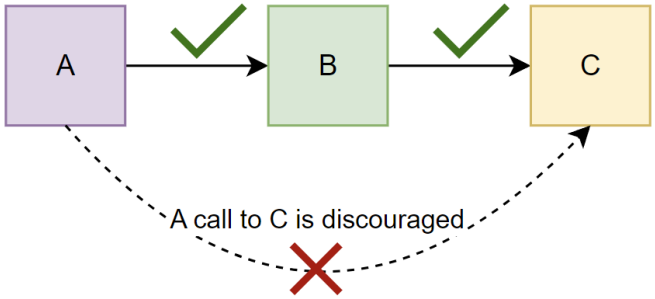
- **Rule 3**
  - **X method can only invoke an object passed to X as an argument**

```
String getHelloIndia(HelloCountries helloCountries) {
    return helloCountries.helloIndia();
}
```

- argument of `getHelloIndia() method` is `HelloCountries` reference
- invoke `helloCountries` methods without violating this rule

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Demeter's Law



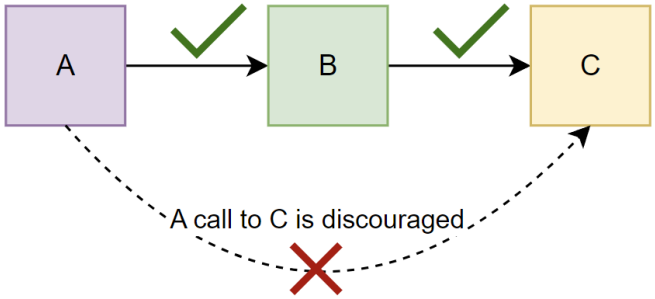
- **Rule 4**
  - **the X method of a class C invokes only the method of the object in the instance variable of C**

```
HelloCountries helloCountries = new HelloCountries();

String getHelloJapan() {
    return helloCountries.helloJapan();
}
```

  - `helloCountries` instance variable in the class. Invoke `helloJapan()` method on the instance variable within the `getHelloJapan()` method

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Demeter's Law



- **Rule 5**
  - **a method X of a class C can only invoke the method of the static field created in C**

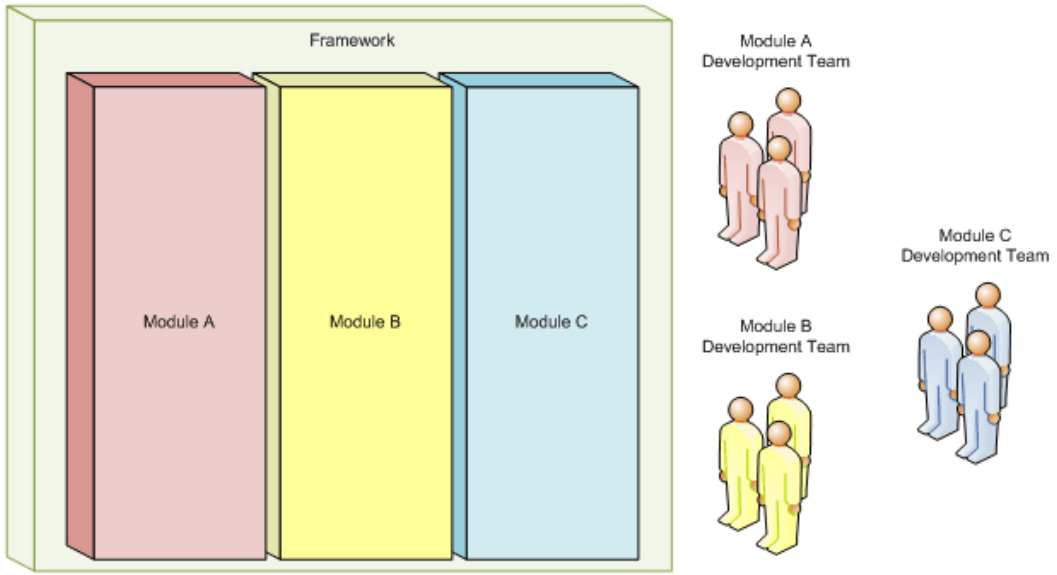
```
static HelloCountries helloCountriesStatic = new HelloCountries();

String getHellStaticWorld() {
    return helloCountriesStatic.helloStaticWorld();
}
```

  - `getHellStaticWorld()` invokes the `helloStaticWorld()` method on the static object created in the class

Forrás: https://blog.bitsrc.io/the-law-of-demeter-in-microservices-build-loosely-coupled-systems-32775d3d577d; https://www.baeldung.com/java-demeter-law

Separation of Concerns (SoC)

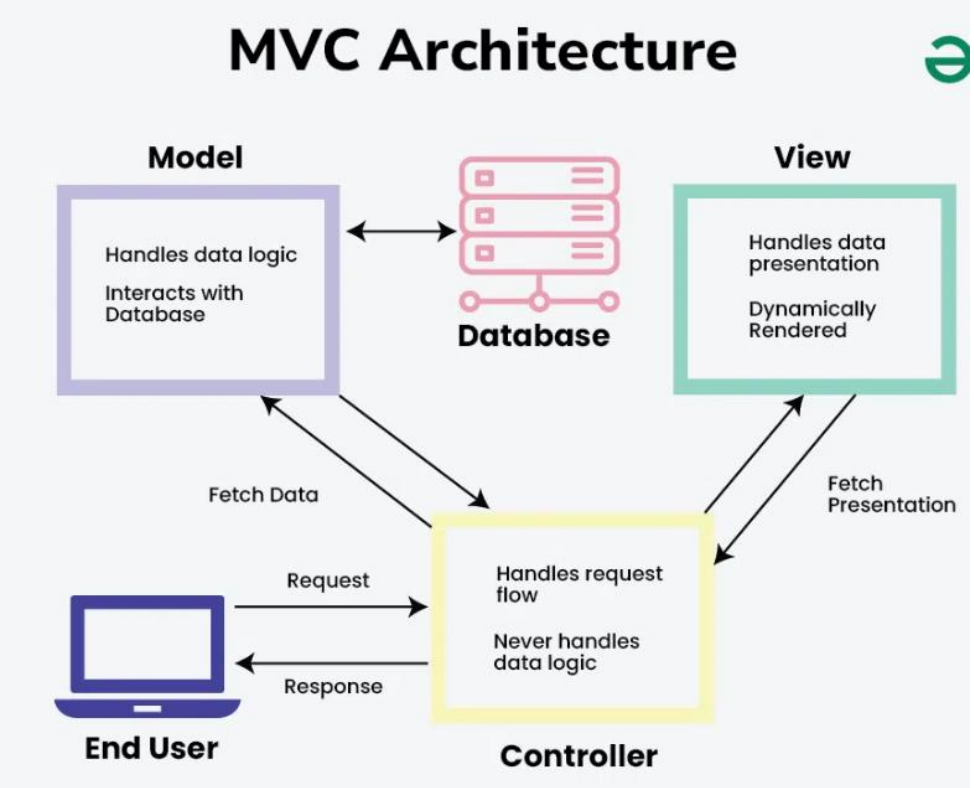
- **Separation of Concerns**
  - **each software component has a well-defined role and they do not overlap**
  - by separating different functions and areas of responsibility, the complexity of the system is reduced
  - parts **should be designed to do only one specific thing** and not take into account the details of the other modules

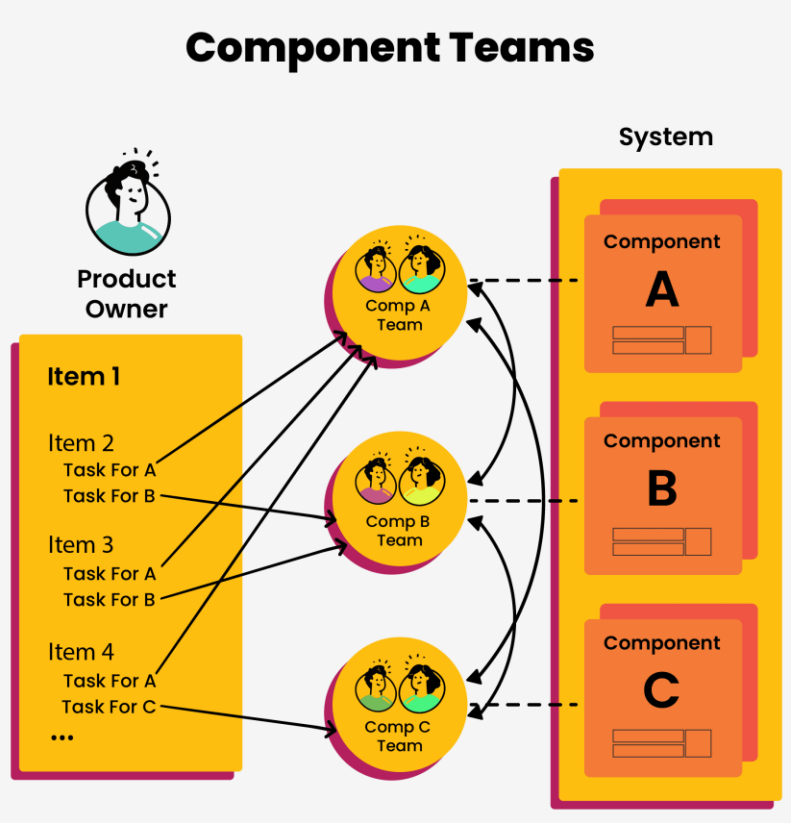


Forrás: https://dev.to/suspir0n/soc-separation-of-concerns-5ak7

Separation of Concerns (SoC)

- **Separation of Concerns**
  - e.g.: MVC, component-based development





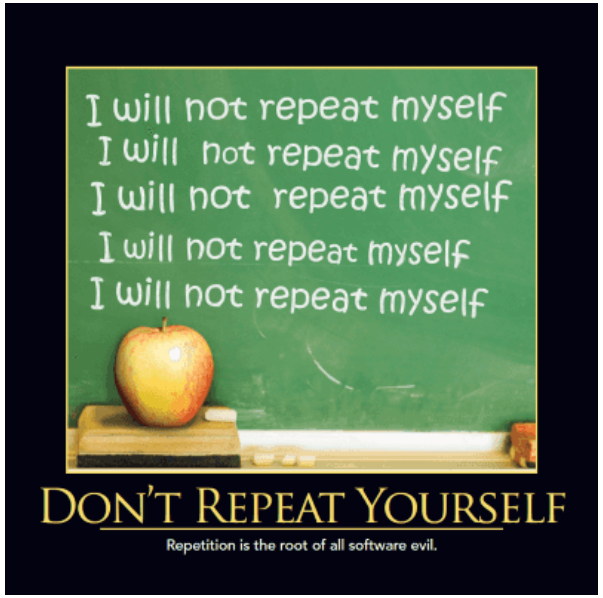
Source: https://www.geeksforgeeks.org/mvc-architecture-system-design/
https://marutitech.com/guide-to-component-based-architecture/

DRY (Don't Repeat Yourself)

- **Avoid code repetition**
  - **nothing should be repeated in the code**
    - → if it needs to be modified, it must be modified at every place where it occurs
  - each business logic or function should be included only once
  - whether the repetition is due to "Copy Paste" programming or a poor understanding of how to apply abstraction, it will decrease the quality of the code

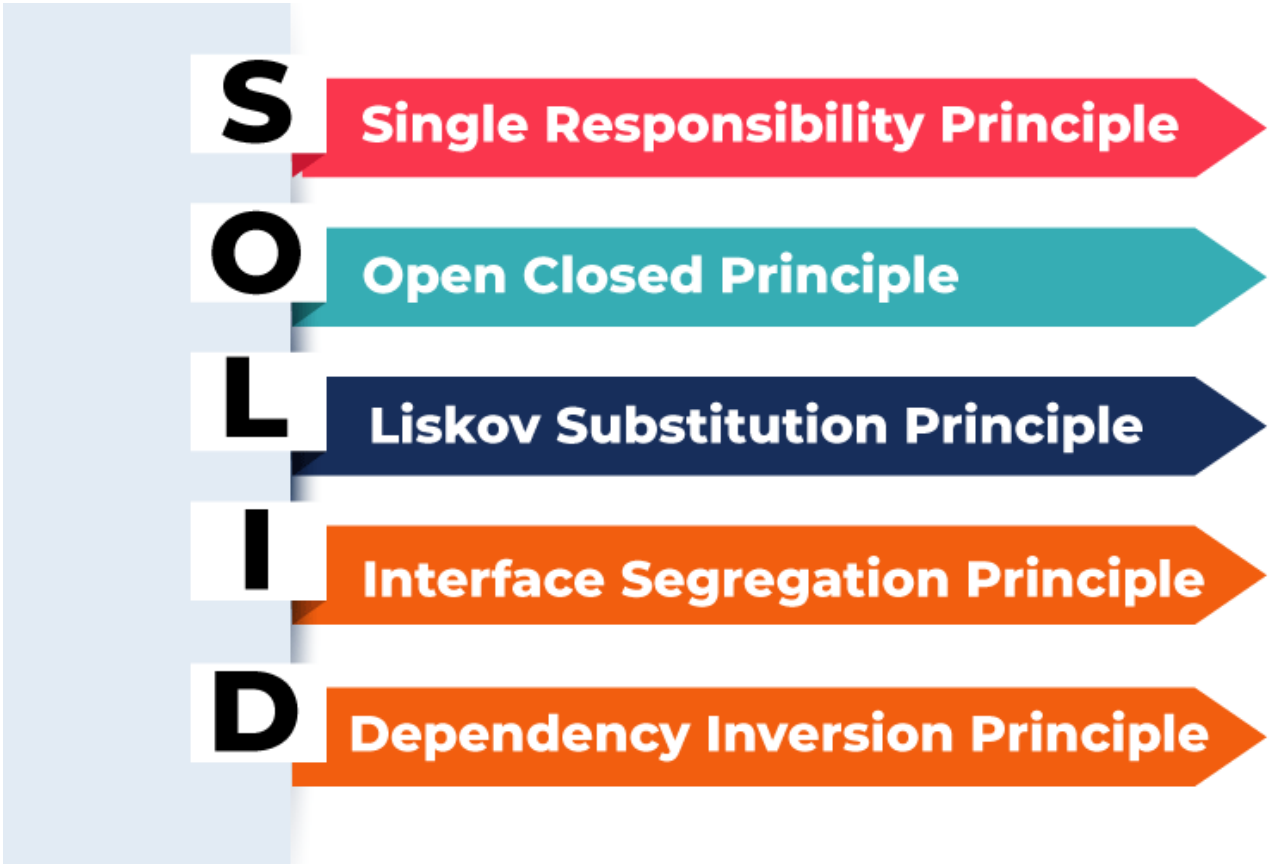
```python
def calculate_discounted_price(price):
    return price * 0.9

def calculate_vip_discounted_price(price):
    return price * 0.8
```

```python
def apply_discount(price, discount_rate):
    return price * (1 - discount_rate)

print(apply_discount(100, 0.1))
print(apply_discount(100, 0.2))
```



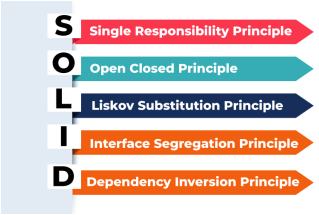
Source: https://deviq.com/principles/dont-repeat-yourself

SOLID elvek



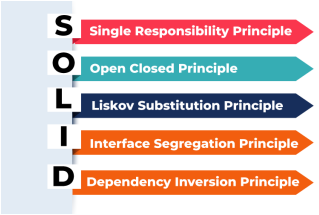
Forrás: https://www.geeksforgeeks.org/single-responsibility-in-solid-design-principle/

Single Responsibility Principle



- **A class/object should be responsible for only one thing**
- **Have one reason to change**
- Separate class for different tasks
- Benefits
  - testing: fewer test cases per class
  - concentration: less function, less dependency
  - organization: smaller, well-organized classes provide better searchability

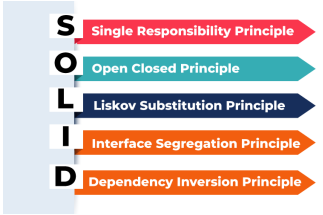
Single Responsibility Principle



- Example
  - there is a `Book` class in which we model some attributes of a book
  - We don't write a method that prints them to the console
    - Why? That's not the job of that department
    - Solution: A `BookPrinter` , or more generally an `ObjectPrinter` class, which can contain the dump methods

- Another example
  - `Student` class: `addGrade`, `setName` methods
    - Registration of grades by the instructor, name change by the secretary
    - The 2 responsibilities are blurred
    - Solution: data storage in the Student class, different classes for different responsibilities

Single Responsibility Principle

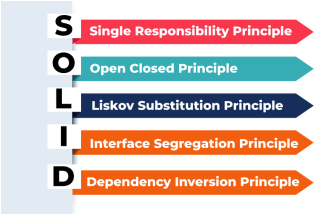



```java
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters

    // methods that directly relate to the book properties
    public String replaceWordInText(String word, String replacementWord){
        return text.replaceAll(word, replacementWord);
    }

    public boolean isWordInText(String word){
        return text.contains(word);
    }
}
```

```java
public class BookPrinter {

    // methods for outputting text
    void printTextToConsole(String text){
        //our code for formatting and printing the text
    }

    void printTextToAnotherMedium(String text){
        // code for writing to any other location..
    }
}
```

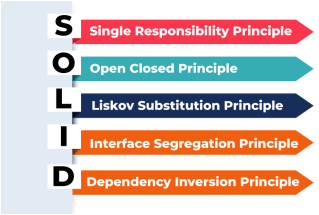
Source: https://www.baeldung.com/solid-principles

Open/Closed Principle

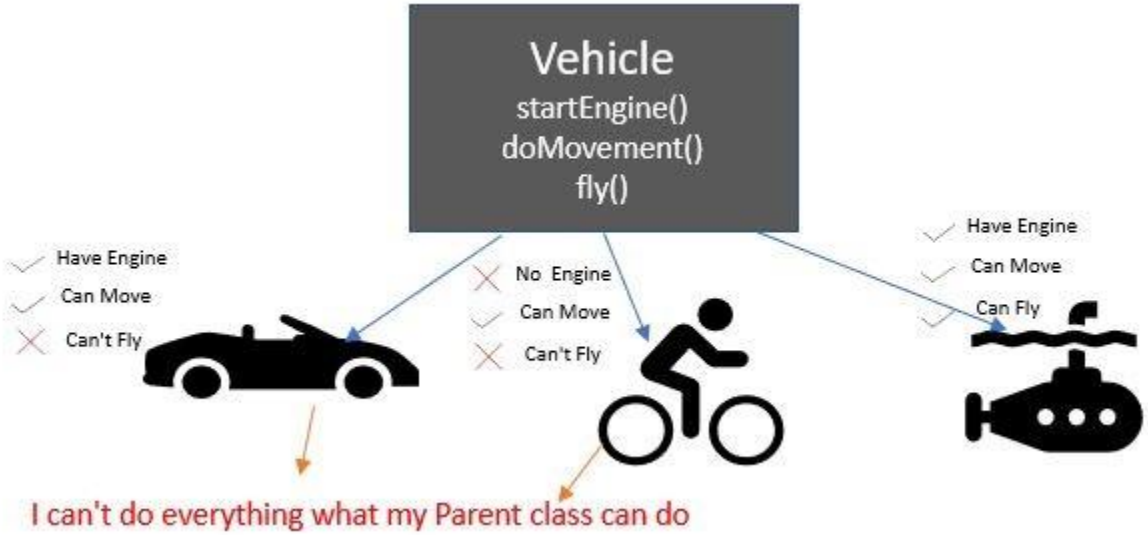


- **A class or module should be open to extending, but closed to modification**
- What we have already finalized (method, object, etc.), we do not modify it, but only expand it
- Advantage: avoidance and prevention of bugs
- Example
  - there is a method that prints out the items of a list row by row
  - but from now on, we want you to put this in a single line
  - we don't modify the previous method, but create a new method for the task

Liskov Substitution Principle

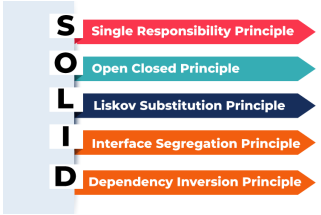


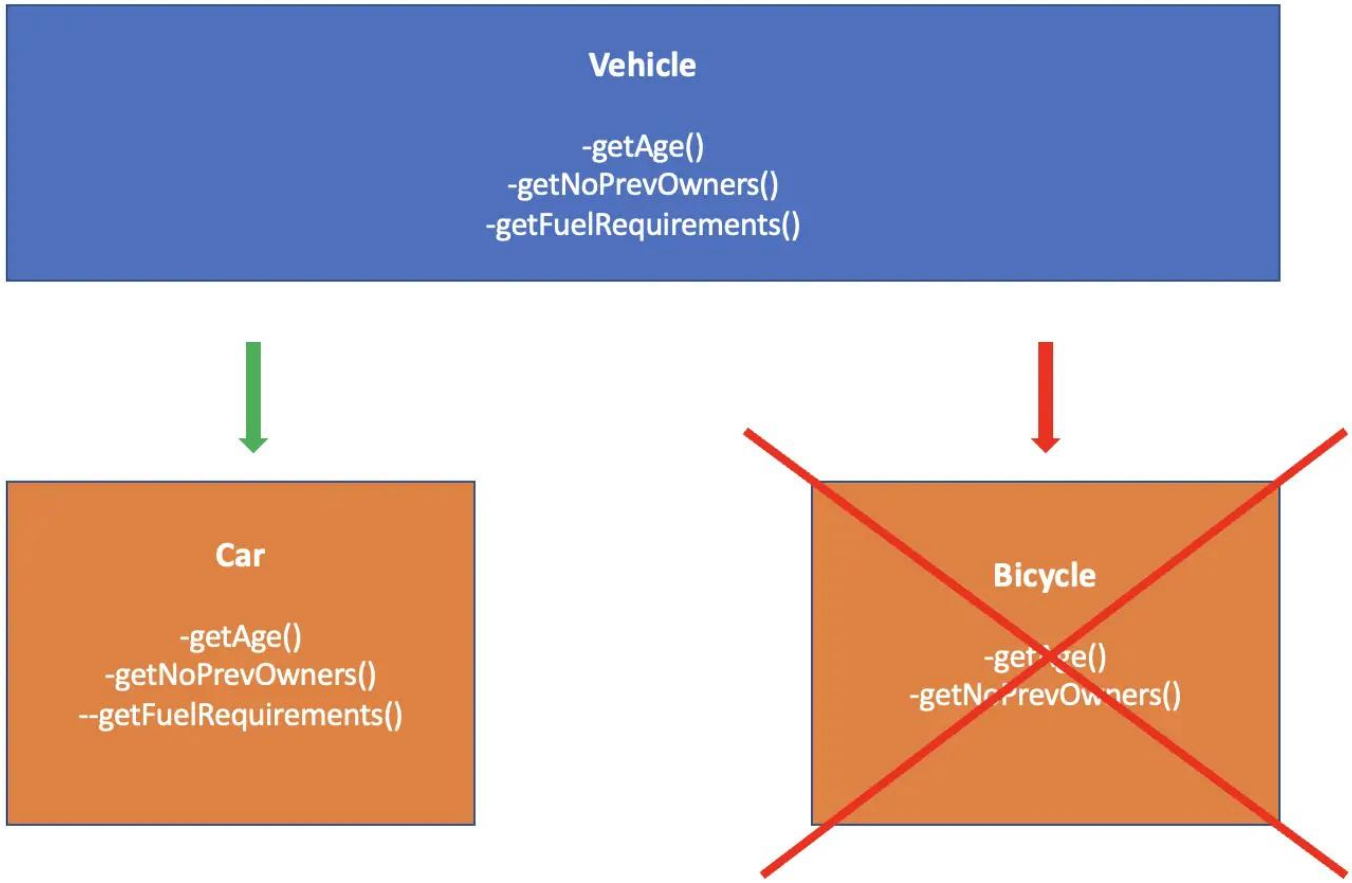
- **Each parent class can be replaced by a descendant, but without any problems**
- If class A is a subtype of class B, then we should be able to replace B with A without any problem
- All descendants know the same thing as the ancestor



Source: https://www.linkedin.com/pulse/liskov-substitution-principle-mamata-raote-she-her-

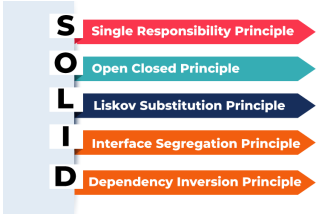
Liskov Substitution Principle



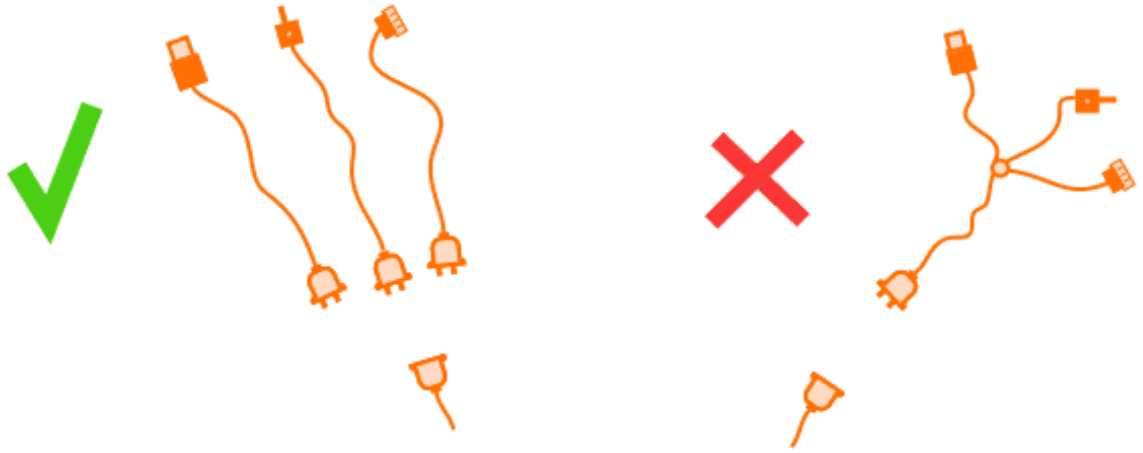


Source: https://www.baeldung.com/solid-principles

Interface Segregation Principle

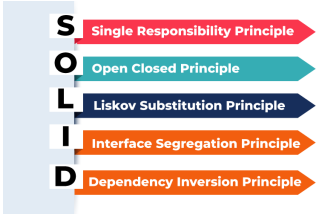


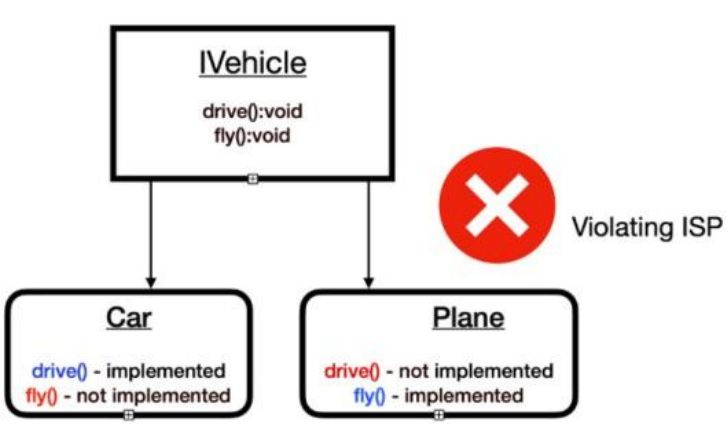
- **Multiple but well-specified interfaces are better than a generic one**
- Don't create too large interfaces, if we want to describe a lot of things in them, should rather separate them into several smaller ones
- Advantage: less dependency

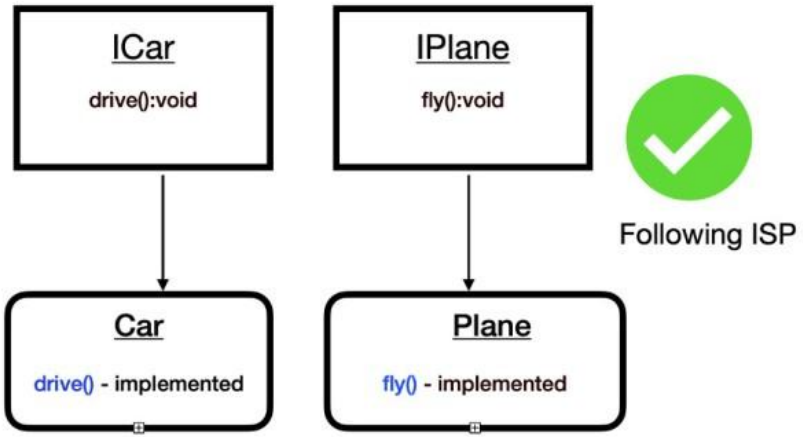


Forrás: https://h-benkachoud.medium.com/interface-segregation-principle-ec256f3828fe

Interface Segregation Principle

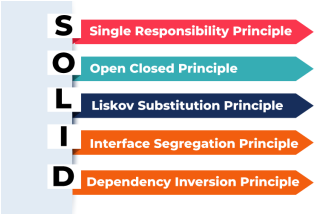




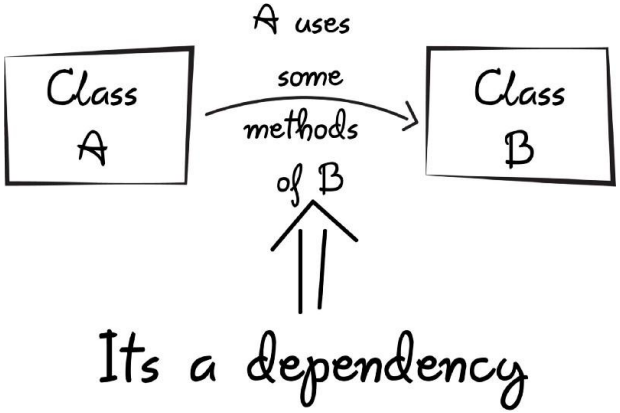


Source: https://www.baeldung.com/solid-principles

Dependency Inversion Principle

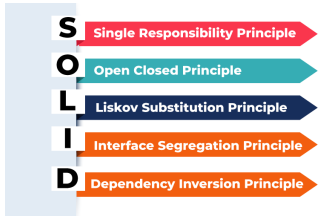


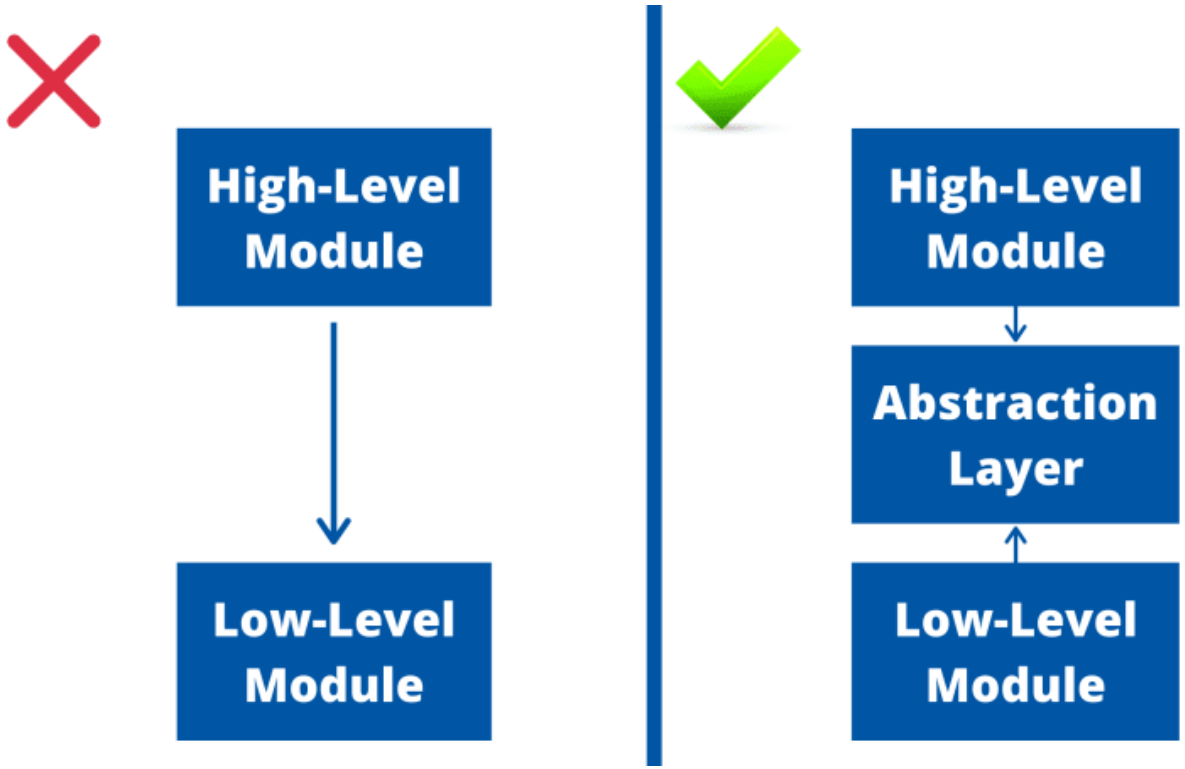
- **It should not be dependent on concrete implementation, but on abstraction**
- Dependency
  - relationship between layers, modules, dependencies
  - one depends on the other: one needs the other
- Loosely coupled
  - loosely related objects
- Dependency injection
  - inject another into one
  - implementation: field, constructor, set
  - the environment decides which implementation is used (IoC - Inversion of Control)



Source: https://www.baeldung.com/solid-principles
https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/

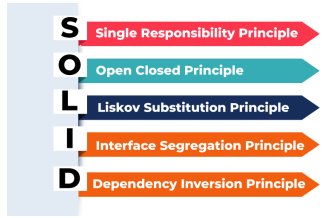
Dependency Inversion Principle

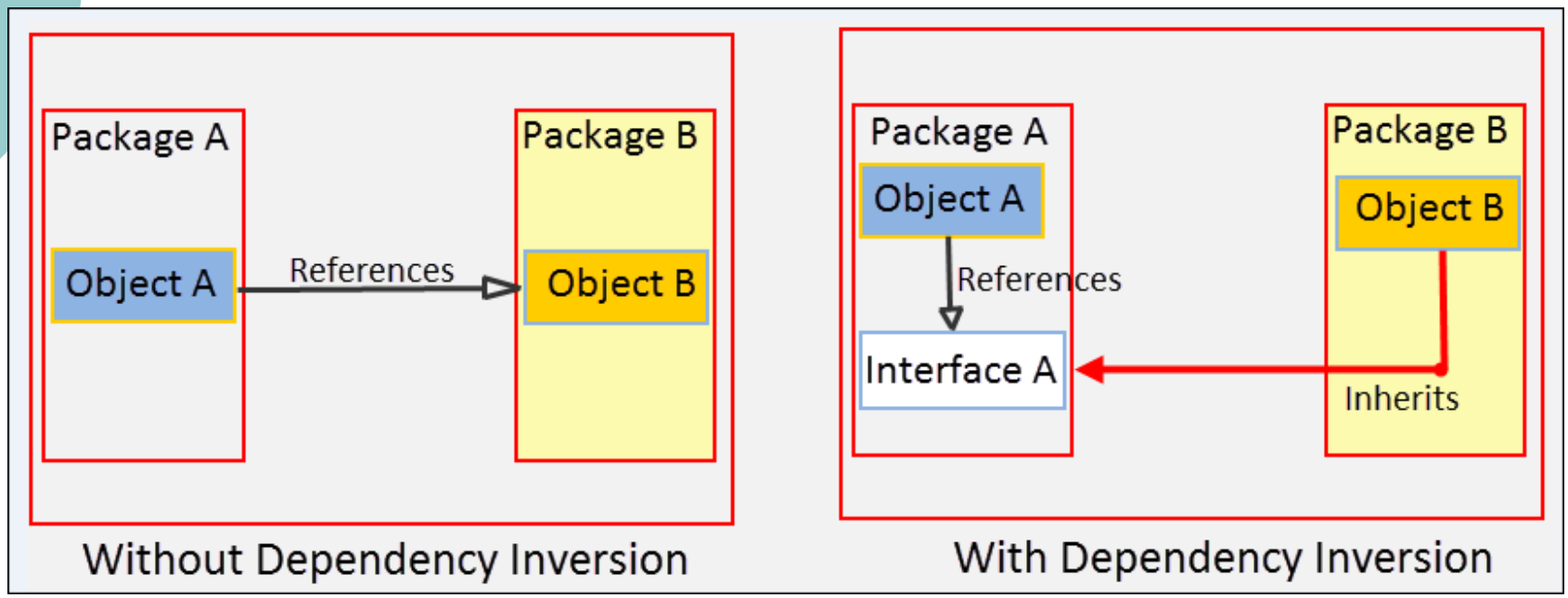




Forrás: https://www.linkedin.com/pulse/dependency-inversion-principledip-mamata-raote-she-her-

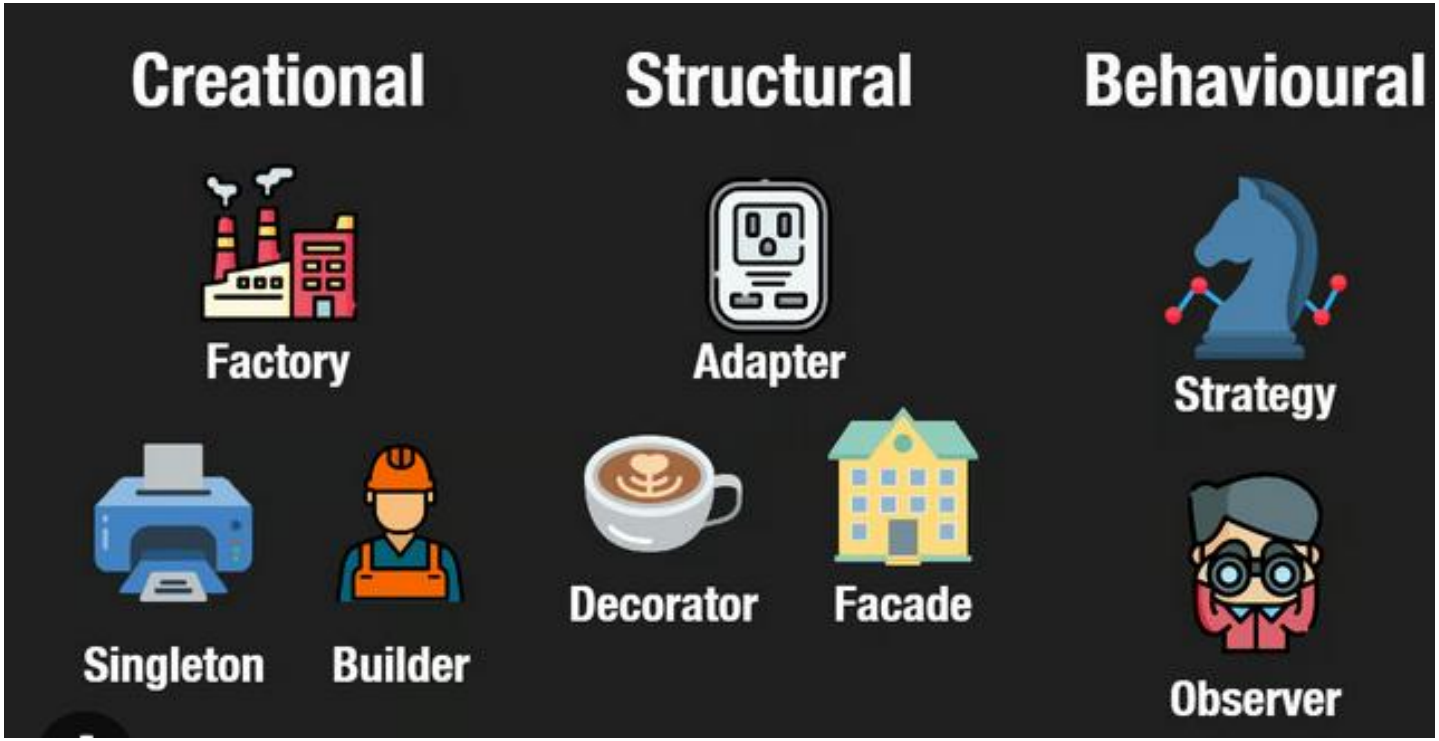
Dependency Inversion Principle





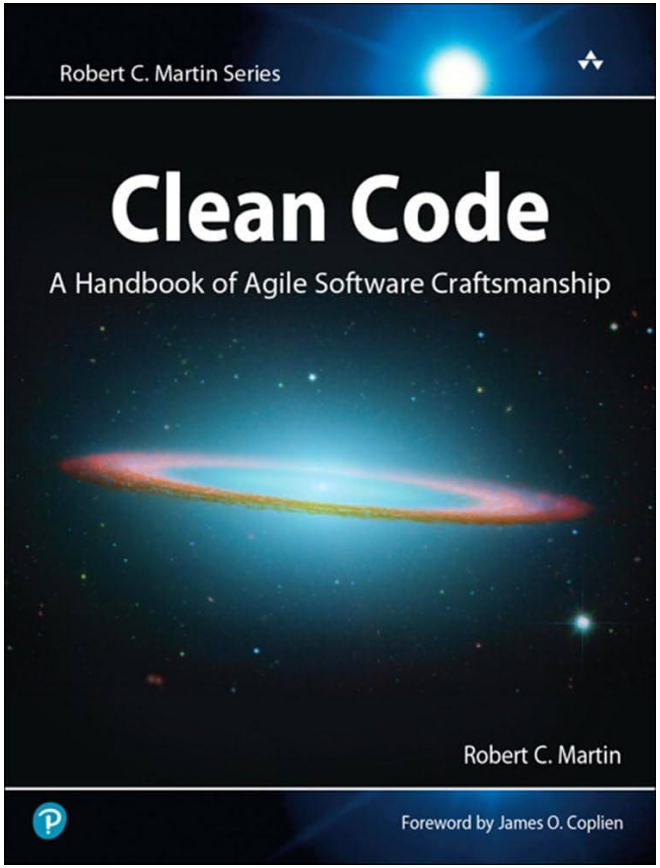
Source: https://www.baeldung.com/solid-principles

Additional program design patterns



Source: https://neetcode.io/courses/design-patterns/0

Recommended books



Thank you for your attention!