OOP HW18 Software Engineering Principles

DATE: 01th february 2021

SHUGULI REINOSO ALAN JESITH

INTRODUCTION TO THE SOLID PRINCIPLES

These principles were called S.O.L.I.D. for its acronym in English:

- **S: Single responsibility principle**
- **O: Open/closed principle**
- **L: Liskov substitution principle**
- **I: Interface segregation principle**
- **D: Dependency inversion principle**

Applying these principles will make your work much easier, both your own and that of others (it is very likely that your code will end up being read by many other developers throughout its life cycle). Some of the advantages of applying it are:

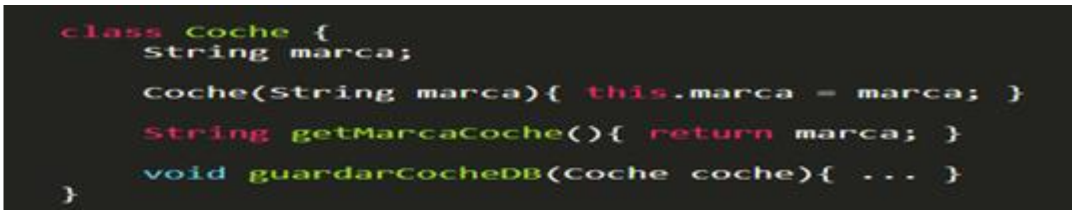
- **Easier and faster code maintenance**
- **It allows adding new functionalities more easily**
- **Promotes greater code reusability and quality, as well as encapsulation**

We are going to see each of these principles in detail, along with basic examples, which, despite not being applicable in the real world, I hope will provide enough clarity for you to be able to understand and apply these principles in your developments.

S: Single Responsibility Principle

As its name suggests, it establishes that a class, component or microservice must be responsible for only one thing (the much acclaimed term "decoupled" in English). If, on the contrary, a class has several responsibilities, this implies that the change in one responsibility will cause the modification in another responsibility.

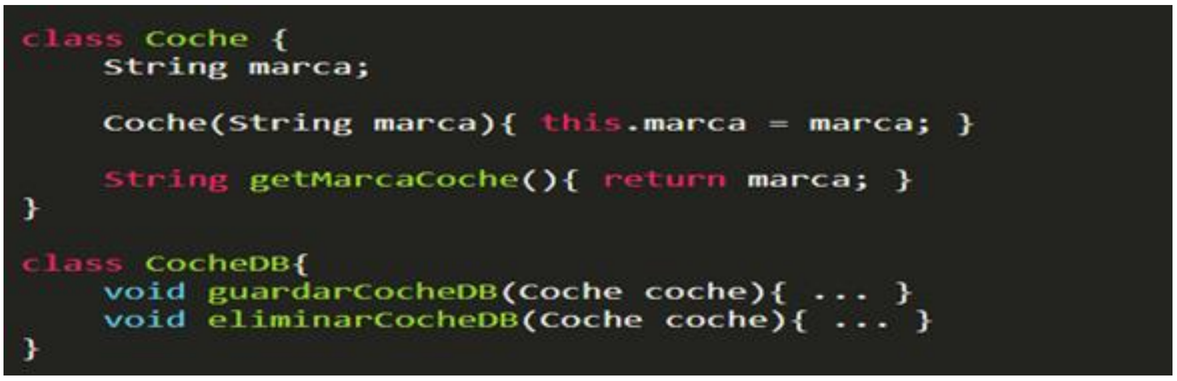
Consider this example:



```
class Coche {
     String marca;
     Coche(String marca){ this.marca = marca; }
     String getMarcaCoche(){ return marca; }
     void guardarCocheDB(Coche coche){ ... }
}
```

As we can see, the Car class allows both access to the properties of the class and operations, so the class already has more than one responsibility.

To avoid this, we need to separate the responsibilities of the class, so we can create another class that takes care of the operations.



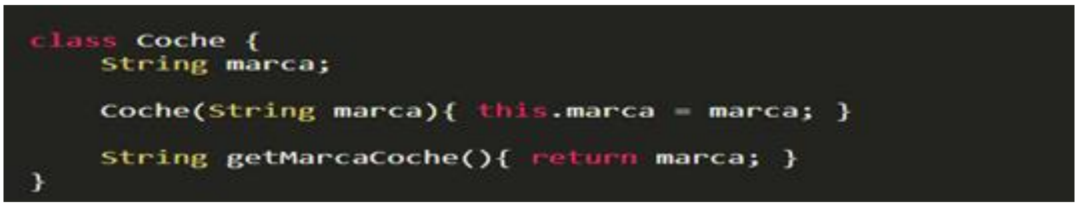
```java
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }
}

class CocheDB{
    void guardarCocheDB(Coche coche){ ... }
    void eliminarCocheDB(Coche coche){ ... }
}
```

Our program will be much more cohesive and encapsulated by applying this principle.

O: Open/closed principle

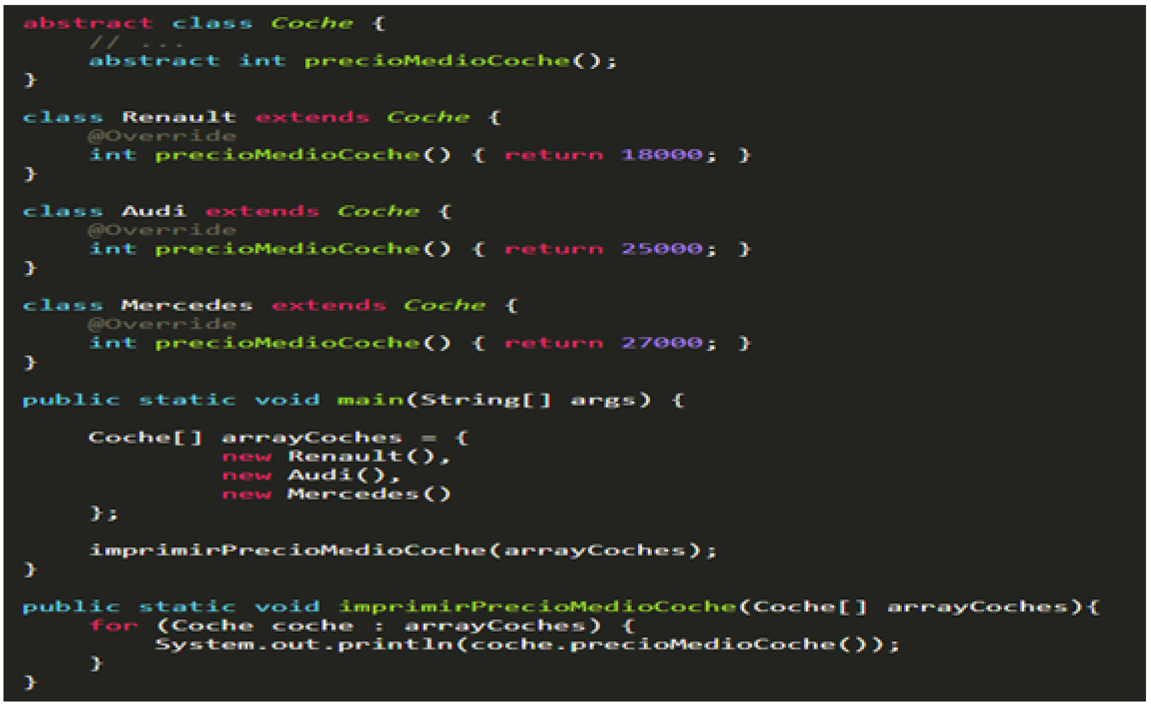
It states that software entities (classes, modules, and functions) should be open for extension, but closed for modification.

If we continue with the Car class:



```java
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }
}
```

To comply with this principle, we could do the following:



```java
abstract class Coche {
    // ...
    abstract int precioMedioCoche();
}

class Renault extends Coche {
    @Override
    int precioMedioCoche() { return 18000; }
}

class Audi extends Coche {
    @Override
    int precioMedioCoche() { return 25000; }
}

class Mercedes extends Coche {
    @Override
    int precioMedioCoche() { return 27000; }
}

public static void main(String[] args) {

    Coche[] arrayCoches = {
            new Renault(),
            new Audi(),
            new Mercedes()
    };

    imprimirPrecioMedioCoche(arrayCoches);
}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        System.out.println(coche.precioMedioCoche());
    }
}
```

Each car extends the abstract class Car and implements the abstract method priceAverageCar().

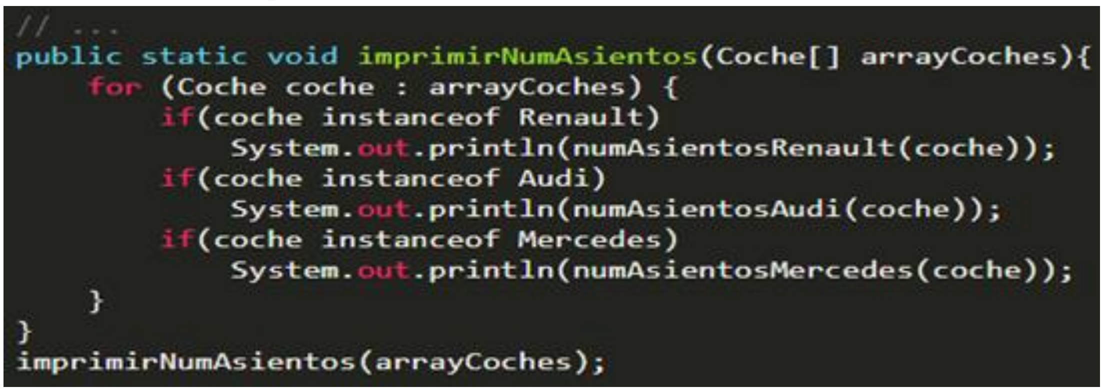
Thus, each car has its own implementation of the method priceAverageCar(), so the method printPriceAverageCar() iterates through the array of cars and only calls the method priceAverageCar().

Now, if we add a new car, AverageCarPrice() will not have to be modified. We will only have to add the new car to the array, thus fulfilling the open/closed principle.

L: Liskov substitution principle

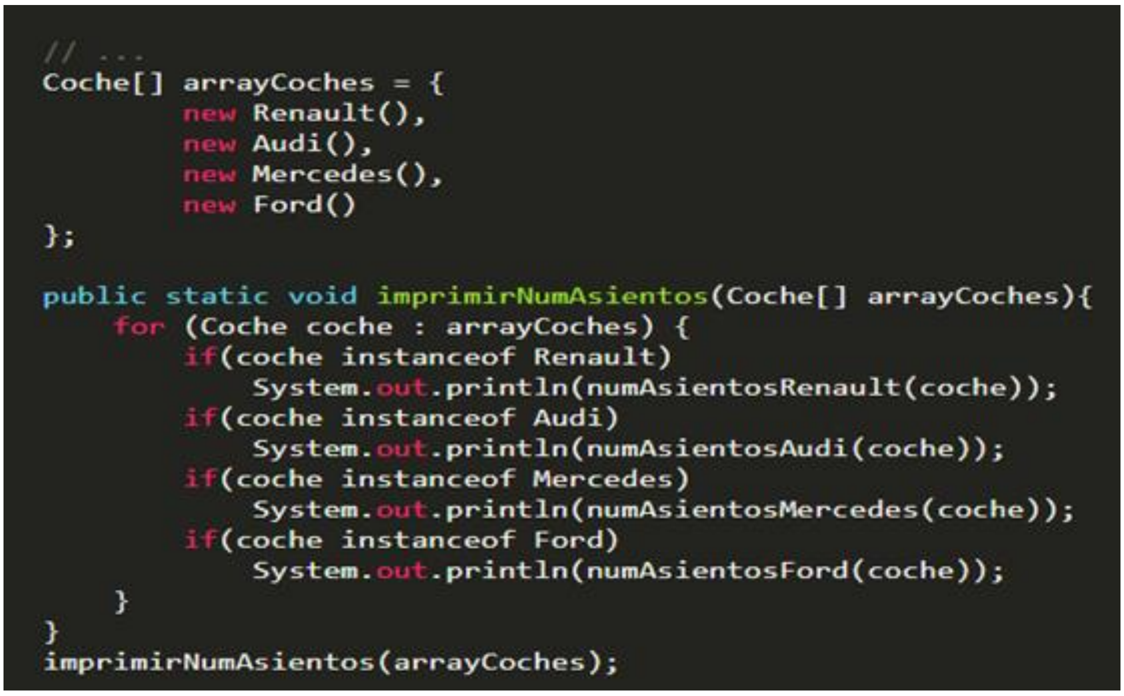
Complying with this principle will confirm that our program has an easy-to-understand class hierarchy and reusable code.

Let's look at an example:



```java
// ...
public static void imprimirNumAsientos(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche instanceof Renault)
            System.out.println(numAsientosRenault(coche));
        if(coche instanceof Audi)
            System.out.println(numAsientosAudi(coche));
        if(coche instanceof Mercedes)
            System.out.println(numAsientosMercedes(coche));
    }
}
imprimirNumAsientos(arrayCoches);
```

The program must know each type of Car and call its associated numSeats() method. Thus, if we add a new car, the method must be modified to accept it.

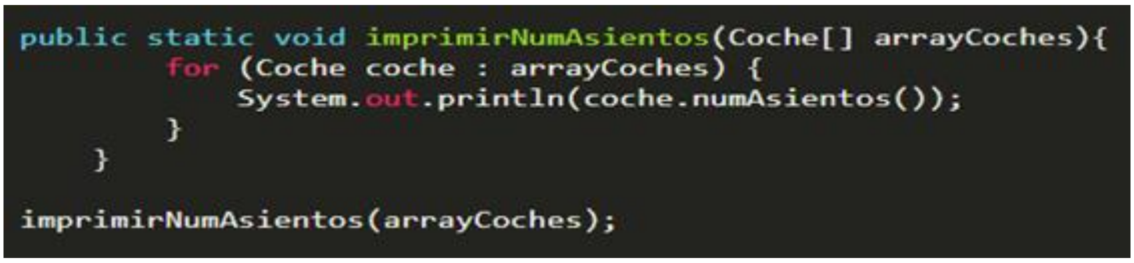


```java
// ...
Coche[] arrayCoches = {
        new Renault(),
        new Audi(),
        new Mercedes(),
        new Ford()
};

public static void imprimirNumAsientos(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche instanceof Renault)
            System.out.println(numAsientosRenault(coche));
        if(coche instanceof Audi)
            System.out.println(numAsientosAudi(coche));
        if(coche instanceof Mercedes)
            System.out.println(numAsientosMercedes(coche));
        if(coche instanceof Ford)
            System.out.println(numAsientosFord(coche));
    }
}
imprimirNumAsientos(arrayCoches);
```

For this method to comply with the principle, we will follow these principles:

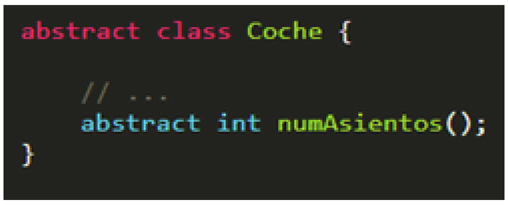
- **If the superclass (Car) has a method that accepts a parameter of the type of the superclass (Car), then its subclass (Renault) should accept either a type of the superclass (Car) or a type of the subclass (Renault) as an argument.**
- **If the superclass returns a type of itself (Car), then its subclass (Renault) should return either a type of the superclass (Car) or a type of the subclass (Renault).**

If we re-implement the above method:



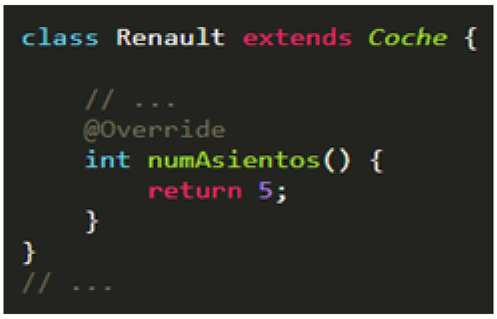
```
public static void imprimirNumAsientos(Coche[] arrayCoches){
        for (Coche coche : arrayCoches) {
            System.out.println(coche.numAsientos());
        }
    }

imprimirNumAsientos(arrayCoches);
```

Now the method doesn't care about the type of the class, it just calls the numSeats() method of the superclass. It only knows that the parameter is of type car, either Car or one of its subclasses.
For this, now the Car class must define the new method:



```
abstract class Coche {

    // ...
    abstract int numAsientos();
}
```

And the subclasses must implement said method:

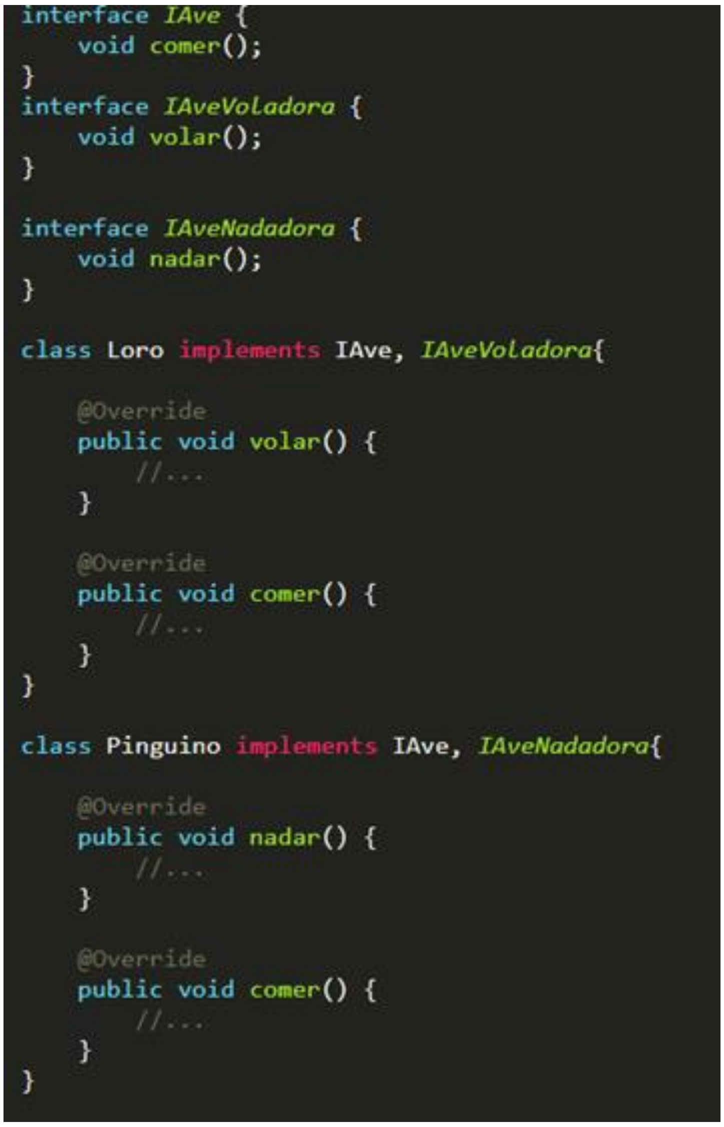


```
class Renault extends Coche {

    // ...
    @Override
    int numAsientos() {
        return 5;
    }
}
// ...
```

As we can see, now the printNumSeats() method does not need to know what type of car it is going to perform its logic with, it simply calls the numSeats() method of the Car type, since by contract, a subclass of Car must implement said method.

I: Interface Segregation Principle

This principle states that clients should not be forced to rely on interfaces they do not use.



```
interface IAve {
    void comer();
}
interface IAveVoladora {
    void volar();
}

interface IAveNadadora {
    void nadar();
}

class Loro implements IAve, IAveVoladora{

    @Override
    public void volar() {
        //...
    }

    @Override
    public void comer() {
        //...
    }
}

class Pinguino implements IAve, IAveNadadora{

    @Override
    public void nadar() {
        //...
    }

    @Override
    public void comer() {
        //...
    }
}
```

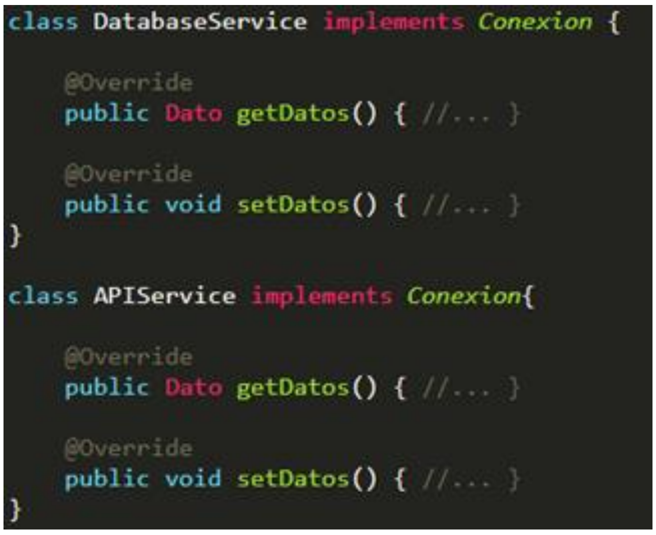
Thus, each class implements the interfaces that it really needs to implement its methods. When it comes to adding new features, this will save us a lot of time, and in addition, we comply with the first principle (Single Responsibility).

D: Dependency Inversion Principle

It states that the dependencies must be on the abstractions, not on the concretions. That is to say:

- **High level modules should not depend on low level modules. Both should depend on abstractions.**
- **Abstractions should not depend on details. Details should depend on abstractions.**

At some point our program or application will be made up of many modules. When this happens, it is when we must use dependency injection, which will allow us to control the functionalities from a specific place instead of having them scattered throughout the program. Also, this isolation will allow us to test much more easily.



```
class DatabaseService implements Conexion {

    @Override
    public Dato getDatos() { //... }

    @Override
    public void setDatos() { //... }
}

class APIService implements Conexion{

    @Override
    public Dato getDatos() { //... }

    @Override
    public void setDatos() { //... }
}
```

Thus, both the high-level module and the low-level module depend on abstractions, thus fulfilling the principle of dependency inversion. In addition, this will force us to comply with the Liskov principle, since the types derived from Connection (DatabaseService and APIService) are substitutable for their abstraction (Interface Connection).

SOFTWARE ENGINEERING GOALS

The goals of software engineering are straightforward and easy to understand – but they aren't always easy to meet. This is because there are so many different ways to approach software engineering and so many outcomes that are possible. While we do have best practices and there are standards in place, every software engineer has a different approach and sometimes they don't always mesh well with other members of an IT team.

- **Software engineers aim to build quality products on time and within budget**
- **Some Desirable Qualities:**
    - understandability
    - testability
    - maintainability
    - efficiency
    - reliability
    - security
    - extensibility
    - openness
    - interoperability
    - reusability

Software Engineering Principles and Best Practices Examples

Software engineering best practices ensure that software developed by a delivery team, a contractor, or an individual developer is high quality and efficient. If applications do not meet these best practices, it puts you at risk for outages, security hazards, and failing systems adherence to software engineering best practices help to prevent that.

Iterative Development

An important development methodology best practice in software engineering is iterative development. Iterative development ensures that software flaws or risks are resolved before there has been a lot of time and effort put into the software. This development approach enables continuous testing and continuous integration which creates the opportunity for early feedback so that changes can be made swiftly.

Service-Based Architecture and Microservices

Service-Based Architecture and Microservices are some of the most critical best practices in software engineering today. Service-based architecture is a software design best practice where services are provided to other components by application components, through communication protocols. A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently. It has four properties:

- **Logically represents a business activity with a specified outcome.**
- **Self-contained.**
- **Black box for its consumers.**
- **May consist of additional underlying services.**

Software Modelling

Using visual modeling tools helps to improve the ability to manage software, rationalize, and maintain that software. It will also help to keep information among the team easy to understand and constant – which is especially important in agile development when multiple teams are working laterally to develop the same software or updating the same application portfolio.

Software Testing

Software Testing is another critical element of software engineering best practices and principles. A team wants to verify that all software developed is high quality and meets the requirements set forward in the planning stage.
Remember that changing software later in development is much, much costlier. Continuous testing from the start of development will help to avoid costly repairs later on or even after deployment. Software engineering practices that do not include testing will eventually fail – but not after costing a lot of time and money from your budget.

Software Engineering Practices and Theory

Software engineering theory and practice meld together computer science with artistry and design. It is a fine line to walk – software that is too “pretty” but doesn’t function isn’t effective, but software that isn’t written well can be difficult as well.

Abstraction is a theory in both art and software engineering. Abstraction is the simplification of a description into the bare-bones essentials. In software engineering theory, this means making code easy enough to read and edit. Elements have to be well named so that they are descriptive – typically only using a combination of verbs and nouns. It needs to be easy enough to understand that someone who has never looked at the code before will understand.

It also has to be accurate – you want to reflect the work that is complete or the action carried out by the string of code – nothing more, nothing less. Brevity should also be clear.

Software engineering theory also calls for separation of the code. One needs to treat what the software does and how the software does it independently. This does make it longer but helps with the clarity that is so essential in software engineering that will be maintained for a longer period of time.

THREE CORE PRINCIPLES

Modularity

Modularity exists in a software system when it is composed of loosely coupled and cohesive components that isolate each significant or modifiable component. design decision on a component and make sure related ideas are as similar as possible. Modularity can improve understandability, testability, maintainability, security and reusability.

Abstraction

For each component, there is an explicit and clear declaration of the component's functions. Exposed features and functionality should be no more or less than what other components may need. The abstraction principle can improve understandability, testability, maintainability, and reusability. It can also allow developers to follow modularity more effectively.

Encapsulation

As can be seen from the diagrams, the object variables are located in the center or core of the object. The methods surround and hide the core object from other objects in the program. 'To the packaging of the variables of an object with the protection of its methods is called encapsulation. Typically, the encapsulation is used to hide unimportant implementation details from other objects. So the implementation details can change at any time without affecting other parts of the project program.

Ways to encapsulate

1. **Standard (Default)**
2. **Open: Makes the class member accessible from outside the Class and anywhere in the Class program.**
3. **Protected: It is only accessible from the Class and the classes that inherit it (at any level).**
4. **Semi closed: It is only accessible from the inherited class.**
5. **Closed: It is only accessible from the Class.**

In encapsulation there are analyzers that can be semantic and syntactic.