Last name: Saltos Taco

Name: Paul Alexander **NRC:** 7490

Career: Telecommunications Engineering

Date: 01-02-2022

Subject: OOP

Software Engineering Principles

INTRODUCTION TO THE SOLID PRINCIPLES

There are five specific aspects of object-oriented programming that each **SOLID** principle addresses, with each letter representing a principle. Fortunately, this acronym makes the five principles relatively easy to memorize:

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

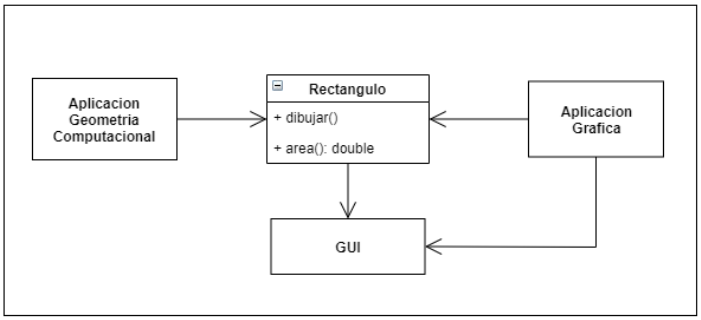
None of these principles is truly exclusive. On the contrary, it could be argued that they are mutually inclusive. Some of them represent multiple strategies and each of them plays a role in achieving a single goal. Or, in other cases, they are by-products: Proper adherence to one of these SOLID practices can naturally beget another.

SOFTWARE ENGINEERING PRINCIPLES PAPER

This document presents some important general principles, which are central to develop software successfully, and that deal with both the software engineering process and the Final product. The right process will help develop the desired product, but also the product desired will affect the choice of process to use. A traditional software engineering problem is put the emphasis on the process or the product to the exclusion of the other, however, both are important.

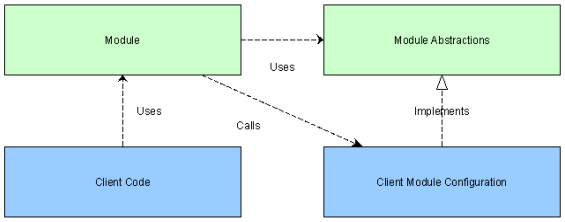
Single Responsibility Principle

It is true when our class only does one thing. It tells us that a module has only one reason to change. For software engineering, it states that each module or class should have responsibility for only one part of the functionality provided by the software, and this responsibility must be fully encapsulated by the class.



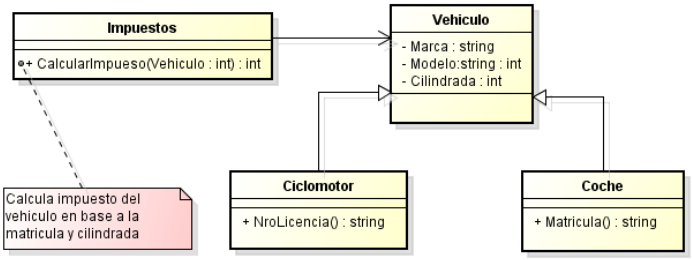
Open/Closed Principle

This principle tells us that a software entity should be open to extension but closed to modification.This helps us keep adding functionality with the assurance that it won't affect existing code. New functionality will mean adding new classes and methods, but in general it shouldn't mean modifying what has already been written. It is usually solved using polymorphism. Instead of forcing the main class to know how to perform an operation, it delegates the operation to the objects it uses, so it doesn't need to know explicitly how to perform it. These objects will have a common interface that they will implement specifically according to your requirements.



Liskov Substitution Principle

It is a principle of object-oriented programming. and can be defined as: Every class that inherits from another can be used as its parent without needing to know the differences between them.
If somewhere in our code we are using a class, and this class is extended, we have to be able to use any of the child classes and the program is still valid.
This forces us to make sure that when we extend a class we are not altering the behavior of the parent.



Interface Segregation Principle

It helps us not to force any class to implement methods it doesn't use. This will avoid problems that can lead to unexpected errors and unwanted dependencies. It also helps us reuse code more intelligently.

That no class should depend on methods it doesn't use. Therefore, when creating interfaces that define behaviors, it is important to make sure that all classes that implement those interfaces will need and be able to add behaviors to all methods.
Otherwise, it's better to have several smaller interfaces. The module that describes the interface doesn't have to know anything about our code, yet we can work with it just fine.

```
interface Product {
    val name: String
    val stock: Int
    val numberOfDisks: Int
    val releaseDate: Int
}

class CD : Product {
    ...
}
```

```
class CD : Product {
    ...
    override val recommendedAge: Int
        get() = throw UnsupportedOperationException()
}
```

Dependency Inversion Principle

It forces us to organize our code in a very different way than we are used to, and against what logic initially dictates, but in the long run it compensates for the flexibility it gives to the architecture of our application.
In object-oriented design, the dependency inversion principle is a specific way of decoupling software modules. By following this principle, the conventional dependency relationships established from high-level policy-setting modules to high-level policy-setting dependency modules. low-level modules are inverted, making high-level modules independent of the low-level module's implementation details.

```
class Shopping { ... }

class ShoppingBasket {
    fun buy(shopping: Shopping?) {
        val db = SqlDatabase()
        db.save(shopping)
        val creditCard = CreditCard()
        creditCard.pay(shopping)
    }
}

class SqlDatabase {
    fun save(shopping: Shopping?) {
        // Saves data in SQL database
    }
}

class CreditCard {
    fun pay(shopping: Shopping?) {
        // Performs payment using a credit card
    }
}
```