ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

FACULTAD DE INGENIERIA EN ELECTRICIDAD Y COMPUTACION

**Taller de Principios SOLID – Diseño de Software**

**MSc. David Jurado**

# Objetivos:

* Identificar violaciones a los principios SOLID.
* Corregir código que viole principios SOLID.
* Utilizar Git como herramienta de apoyo al trabajo colaborativo.

# Instrucciones

* Trabajar en grupos de máximo 5 integrantes.
* Debe utilizarse un repositorio de Github para este nuevo grupo.
* En el paquete desacargado hay un proyecto compuesto por 10 paquetes (2 por cada principio, mal y bien).
  + Los paquetes que terminan con **‘violation’** en su nombre, contiene clases e interfaces que viola un principio SOLID. Las tres primeras letras del nombre del paquete le indican el principio que se está violando.
* Dentro de este documento, deben indicar como cada subpaquete está violando el principio SOLID correspondiente. Todos deben estar de acuerdo en el problema y como solucionarlo.
* Cada integrante debe corregir por lo menos un subpaquete, colocando la solución en el subpaquete ‘**solution’** respectivo y subirlo al repositorio remoto de Github con su propio usuario.

# Conteste

Explique brevemente como cada principio es violado en el código analizado y cuál sería la posible solución.

**-SRP:** Se defina una clase MaquinaHaceTodo que contiene métodos de todo tipo, desde imprimir informe, servir café, calentar y enfriar habitación, fabricar carro, bicicleta, etc. Este tipo de clases no cumple el principio de Responsabilidad Única ya que abarcar muchas funcionalidades. La posible solución es crear clases específicas para cada método que cumpla con el objetivo del método. Por ejemplo, los métodos de enfriar y calentar habitación pueden ir a una clase Termostato y así con los demás casos.

**-OCP:** Este principio se viola en la clase testeadorDeGiro ya que, en el caso de requerir la existencia de nuevos vehículos, se deberá ir agregando nuevos métodos. Lo ideal sería agregar métodos de giro a cada vehículo por medio de una interfaz, de esa manera la clase testeador requerirá un Vehiculo y llamara a su método correspondiente.

**-LSP:** En la clase Carro, LSP es violado debido a que no se valida que el Radio este encendido cada que se quiera usar el método escuchar. Por lo tanto, no cumple LSP al momento de que no se puede reemplazar la clase Padre por el hijo (Carro) ya que el método no serviría.

**-ISP:** Aquí podemos decir que “Muchas interfaces específicas son mejores que una única más general”, ya que las clases que implementen la interfaz maquinaria, se ven forzados a implementar métodos innecesarios (clases que usan la interfaz maquinaria se ven forzadas a depender de interfaces que no usan). Lo que podemos realizar es implementar interfaces específicas, evitando un acoplamiento innecesario.

**-DIP:** El problema que existe aquí es que cuando deseáramos agregar otro tipo de medio de transporte deberíamos de crear más clases específicas para manejar los asientos y las llantas de ellos si posee alguna de ellas. Por lo que la solución radica que se debe de crear una interface que implemente los transportes y ellos se encarguen de implementar los métodos si los necesitan.

# Entregable

* El enlace al repositorio de GitHub con las soluciones y este archivo Word con el análisis grupal.
* Subir el enlace a Sidweb.