

**Actividad 1: Siguiendo las buenas prácticas de programación vistas recientemente, (clean code, code convention & comment code) refactorizar el código proporcionado. Descarga: oop-code-C# o oop-code-JAVA, puedes elegir cualquiera de los dos lenguajes. Descripción de la actividad: El siguiente proyecto contiene la lógica para calcular el área y perímetro de varias figuras geométricas. Sin embargo el diseño del código está altamente acoplado, de forma que cualquier modificación en las funciones de cálculo del área.**

**Para esta actividad aplicamos el patron SOLID, se hacen dos versiones:**

### **1) Separado por archivos**

```

v actCalculatorShape
  > Circle.java
  > Cube.java
  > Main.java
  > Rectangle.java
  > Shape.java
  > Sphere.java
  > Square.java
  > Triangle.java
```

### **2) todo en un solo documento**

```

v activity1
  > Main.java
```

Se aplican los principios abordados en clase evitando aplicar mala practicas e ir en contra de los principios de clean code.

## **Se aplica los principios SOLID**

### **1. Single Responsibility Principle (SRP)**

Se crearon clases separadas para cada figura geométrica y se realiza la implementación, es decir, se realiza la lógica de cálculo de área y perímetro a estas clases.

### **2. Open/Closed Principle (OCP)**

Se creo una interfaz para definir un contrato o estructura común para todas las figuras geométricas, permitiendo que nuevas figuras se agreguen sin modificar el código existente.

### **3. Liskov Substitution Principle (LSP)**

Aseguraremos que las subclases puedan reemplazar a sus superclases sin alterar el comportamiento esperado.

### **4. Interface Segregation Principle (ISP)**

Creamos interfaces específicas para diferentes funcionalidades.

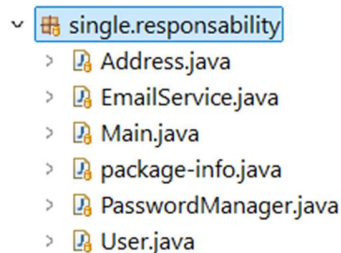
### **5. Dependency Inversion Principle (DIP)**

Dependeremos de abstracciones en lugar de clases concretas.

## Actividad 2

### Single Responsibility

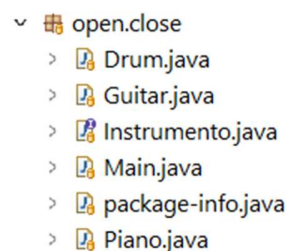
Para aplicar el Principio de Responsabilidad Única (SRP) del patrón SOLID, se debe dividir la clase User en varias clases, cada una con una única responsabilidad. User tiene responsabilidades de gestión de dirección, contraseñas y envío de correos electrónicos, que deben separarse en diferentes clases.



### Explicación

1. **User:** Solo maneja la información básica del usuario y la autenticación. Se le ha delegado la responsabilidad de actualizar la dirección a la clase Address.
2. **Address:** Se encarga exclusivamente de la lógica relacionada con la dirección del usuario.
3. **PasswordManager:** Encapsula la lógica para actualizar contraseñas, separándola de la clase User.
4. **EmailService:** Maneja el envío de correos electrónicos, evitando que la clase User tenga múltiples responsabilidades.

**OCP - Principio Abierto/Cerrado: El código debe estar abierto para la extensión, pero cerrado para la modificación.**



### Explicación:

- La interfaz Instrumento define un contrato para cualquier clase que represente un instrumento.
- Las clases **GUITAR**, Piano, y Drum implementan esta interfaz, asegurando que el método tocar esté disponible.
- El método **tocarNotas** a la clase **Main**, acepta un objeto que implementa **Instrumento** y toca el sonido n veces.

- En el método **tocarSinfonia**, se utilizan instancias de Instrumento en lugar de instancias específicas de clases para evitar la modificación directa de la lógica de impresión de sonidos.

**LSP - Principio de Sustitución de Liskov: Las instancias de las clases derivadas deben poder sustituir a las instancias de las clases base sin alterar el comportamiento del sistema.**

```

v liskov.substitution
  > Employee.java
  > Main.java
  > package-info.java
  > Programmer.java
  > Tester.java

```

#### Explicación:

- Al definir calcularPago como un método abstracto en Empleado, obligas a cada subclase a proporcionar su propia implementación.
- Cuando invocas calcularPago en la lista de Empleado, la implementación adecuada para cada tipo de empleado se utiliza automáticamente gracias al mecanismo de polimorfismo.
- Esto garantiza que el comportamiento del sistema sea correcto y consistente con el LSP, permitiendo que cada subclase de Empleado sea sustituida por su base sin alterar el comportamiento del sistema.

**ISP - Principio de Segregación de Interfaces: Las interfaces deben ser específicas y adaptadas a los requisitos particulares.**

```


v interfaces.segregation
  > CamaraAdvance.java
  > CameraAvanced.java
  > CameraBasic.java
  > CameraEconomic.java
  > CameraIntermediate.java
  > CameraSuper.java
  > CameraZoom.java






```

#### Explicación:

- Interfaces Especializadas: Las interfaces se dividen en CameraBasic, CameraAvanced, y CameraZoom, agrupando funciones relacionadas. Esto permite que cada clase de cámara implemente solo las interfaces relevantes para sus capacidades. Se crea la cámara SUPER que
- Flexibilidad y Claridad: Al implementar interfaces específicas, se asegura que las clases de cámara solo se vean obligadas a implementar métodos que realmente necesitan, evitando problemas derivados de implementar métodos innecesarios.

**DIP - Principio de Inversión de Dependencias: Las clases de alto nivel no deben depender de implementaciones de bajo nivel, sino de abstracciones.**

▼  dependency.inversion

- >  Database.java
- >  DatabaseInterface.java
- >  Main.java
- >  MongoDB.java
- >  MySQL.java

- Interfaz DatabaseInterface: Define los métodos que cualquier base de datos debe implementar.
- Clases MySQL y MongoDB: Implementan la interfaz y proporcionan implementaciones específicas para sus métodos.
- Inyección de Dependencia: La clase Database acepta una instancia de DatabaseInterface a través de su constructor, permitiendo que funcione con cualquier clase que implemente esta interfaz.

**Integrantes:**

**Repo:Github**

**Carlos Emilio Hoyos Medina**

**Enlace:**

**Jose Gregorio Castro**

**<https://github.com/jgcastro2783/ActividadRefactory>**