SOLID

Single responsibility. Principio de única responsabilidad.

Una clase debe tener una única razón para cambiar. Una refactorización o un arregle de un bug no es una razón de cambio. Las razones de cambio son cambios de los requerimientos.

Ejemplo: Método de la clase registroDeUsuario. Crea un usuario y encripta una contraseña. Los métodos están correctamente implementados.

```
public void crearUsuario(String mail, String password){
        final String claveSecreta= “sos_inimputable_hermano”;
        String paswordEncriptada = encriptar(password, claveSecreta);

        Usuario nuevoUsuario = new Usuario(mail, passwordEncriptada);
        repositorioDeUsuarios.guardar(nuevoUsuario);
}
```

Esta porción de código viola el principio de única responsabilidad (Es decir, en verdad es **la clase** quien viola el principio entonces) porque en las lineas amarillas se está encargando de la encriptación de la contraseña y en las celestes hace lo que se espera que haga, crear un usuario. En este caso, la encriptación es lo que esta “de más”.

Esta clase tendría dos motivos para cambiar:

1. Si se cambia la forma en la que se registran lo usuarios.
2. Si se cambia la forma que se encripta.

Para evitar esto se debe delegar. Ejemplo de código:

```
public class registradorDeUsuario{
	public void crearUsuario(String mail, String password){

		String passwordEncriptada = EncriptadorContrasenias.encriptar(password);
		Usuario nuevoUsuario = new Usuario(mail, passwordEncriptada);
		RepositorioDeUsuarios.guardar(nuevoUsuario);
	}
}
```

Registro de Usuario delega la encriptación a Encriptador Contrasenias.

Open/Close Abierto/Cerrado.

Se debe poder cambiar el comportamiento sin modificar el código ya existente. Se puede lograr utilizando herencia o delegación.

Ejemplo de código.

```
public class Rectangulo {
	public int base;
	public int altura;

	public Rectangulo(int base, int altura){
		this.base = base;
		this.altura = altura;
	}
}

public class CalculadorArea {
	public int calcularAreaTotal(Collection<rectangulo> rectangulos){
		int area = 0;

		for(Rectangulo rectangulo : rectangulos){
			area = area +(rectangulo.base + rectangulo.altura);
		}
	return area;
```

```
}
```

Este código rompe el principio de abierto/cerrado porque rompe el encapsulamiento (Tell don’t ask). Si yo cambio la figura, este código no me sirve.

Para evitar esto, podemos aplicar este código:

```
public class CalculadoArea{
        public int calcularAreaTotal(Collection<figura> figuras){
                int area = 0;
                for(IFigura figura : figuras){
                        area = area + figura.area();
                }

                return area;
}
```

Delega el cálculo del área a la figura, todas las figuras que use deben cumplir el contrato de forma y sepan calcular su área. Siendo IFigura una interfaz que puedo implementarla en triángulos, cuadrados, etc.

Liskov. Principio de sustitución de Liskov.

Las clases heredadas deben poder ser utilizadas a través de su clase madre sin la necesidad de que el usuario sepa la diferencia. Se debe cumplir la condición “es un” al aplicar la herencia.

Ejemplo de código.

```
Public class Pato{
        public void nadar(){ System,out.printIn(“NADO”);}
        public void hacerCuack { System.out.printInt(“CUAK CUAK”)}
        public void volar(){ System.out.printInt(“VUELO”);
}
public class PatoDeGoma extends Pato {
        public void nadar(){ System,out.printIn(“Nadandooooo”);}
        public void hacerCuack { System.out.printInt(“Cuakeandoooo”)}
        public void volar(){ throw new Error(“Te la debo...”);}
}
```

Alguien que tenga una instancia de patoDeGoma y vea la interfaz, va a pensar que el patoDeGoma puede volar (y no es así). Puede tener otra implementación en donde “volar” no haga nada, pero sigue estando raro. En el caso que me lance un error debería implementar un “try – catch” para contemplarlo en el código que, por ejemplo, me haga volar una colección de patos, entre otros ejemplos poco felices. Si no cumplimos este principio, tampoco cumplimos el Open/Close porque pedimos más información de la que tendríamos que saber.

Si mi diseño cumple el principio, debería pasar que en cualquier momento se puede intercambiar una instancia de patoDeGoma con una instancia de Pato.

Forma de resolver este problema podemos crear una interfaz.

```
public interface nadador{
        void nadar();
}
public interface Cuackeador {
        void hacerCuak();
}
piblic interface Volador{
        void volar();
}
```

```
public class Pato implements Cuackeador, Nadador, Volador{
        @Override
        public void hacerCuak(){ System.out.println(“CUAK CUAK”);}
        @Override
        public void nadar(){ System.out.println(“NADO”);}
        @Override
        public void volar(){ System.out.println(“VUELO”);}
}
public cass PatoDeGoma implements Cuakeador, Nadador{
        @Override
        public void hacerCuak(){ System.out.println(“CUAK CUAK”);}
        @Override
        public void nadar(){ System.out.println(“NADO”);}
}
```

-------- Una clase x que implementa la siguiente porción de código:

```
public void hacerVolar(Collection<Volador> voladores){
        for(Volador volador : voladores){volador.volar();}
}
```

En la colección solamente voy a tener voladores, NUNCA un patoDeGoma.

Interface Segregation. Principio de segregación de la interfaz.

Los clientes no deben ser forzados a depender de métodos que no utilizan. Muchas interfaces específicas son mejores que una interfaz de propósito general. Es necesario aplicarlo cuando se tiene una clase con varios métodos, de los cuales solamente me interesan algunos.

Ejemplo de código:

```
public interface ByteUtils{
        int Read(byte[] b); // Read into buffer
        int Write(byte[] b); // Write into buffer
        byte[] Trim(byte[] b, String exclusions); //Trim buffer by removing bytes from the
exclusions.
}
```

No poner interfaces muy grandes, sino cortarlas en pedazos para que no se tienten en usar más de lo que deberían usar.

```
Public interface Writer { int Write(byte[] b); }
public interface Reader {int Read)byte[] b);}
public Interface Trimmer {int Trim(byte[] b, String exclusions);}

public class ReadWriter mplements Write, Reader { /** }
ó puedo hacer
public class TrimReader implements Trimmer, Reader {…}
```

Dependency Inversion. Principio de inversión de dependencia.

Se debe depender de las abstracciones y no de las implementaciones. Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones. Las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones.

Ejemplo de código.

```
Public class SaludadorBienvenida {
	private EnviadorMail enviadorMail;

	public saludadorBienvenida(EnviadorMail enviadorMail) {this.enviadorMail =
enviadorMail);}
	public boolean saludar(String nombre){
		String saludo = “Hola “ + nombre +” !!! Gracias”;
		return this.enviadorMail.enviar(saludo);
	}
}
```

La clase SaludadorBienvenida depende de EnviadorMail porque delega parte de su comportamiento. En este codigo se viola el principio al tener la clase concreta EnviadorMail ya que, si quiero cambiar el comportamiento de como se envian estos mails, es medio estático. ¿Como arreglarlo? Dependiendo de una abstracción (una clase abstracta o una interfaz)

Codigo mejor:

```
public interface Enviable {
	public boolean enviar(String saludo);
}
public class EnviadorSlack implents Enviable{
	@Override
	public boolean enviar(String saludo){
		return true; //Envia Slack
	}
}
public class EnviadorSMTP implements Enviable{
	@Override
	public boolean enviar(String saludo){
		return true; //Envia por mail
	}
}
```

```
public class SaludadorBienvenida{
	private Enviable enviador; // Con esto invertí la dependencia.

	public SaludadorBienvenida(Enviable enviable) {this.enviador = enviador)}
	public boolean saludar(String nombre){
		String saludo = “Hola “ + nombre;
		return this.enviador.enviar(saludo);
	}
}
```

Ahora, SaludoBievenida no especifica que enviaMail, no depende de una clase en concreta que es difícil de refactorizar a futuro, sino de una interfaz Enviable que recibe por parámetro a la hora de inicializar (puede ser mail, slack, instagram, etc que implemente la interfaz).