

Principios de diseño

Algunas definiciones

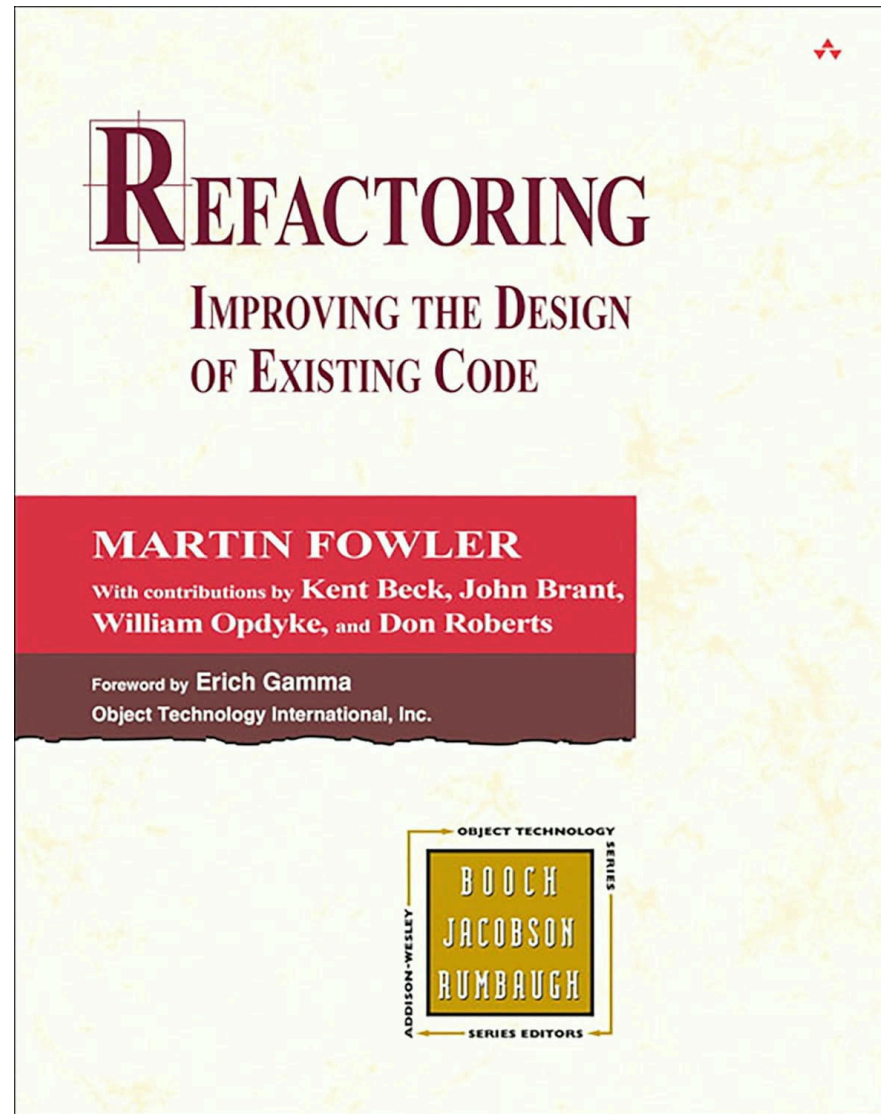
Code smell: Un indicio de que algo no está bien en el código. [🔗](#)

Deuda técnica: Es el costo implícito del trabajo adicional futuro resultante de elegir una solución fácil sobre una más robusta. [🔗](#)

Refactoring: Proceso de reestructuración del código para mejorar su legibilidad, mantenibilidad y extensibilidad, sin cambiar su comportamiento externo. [🔗](#)

Cualquier idiota puede escribir código que una computadora pueda entender. Los buenos programadores escriben código que los humanos pueden entender.

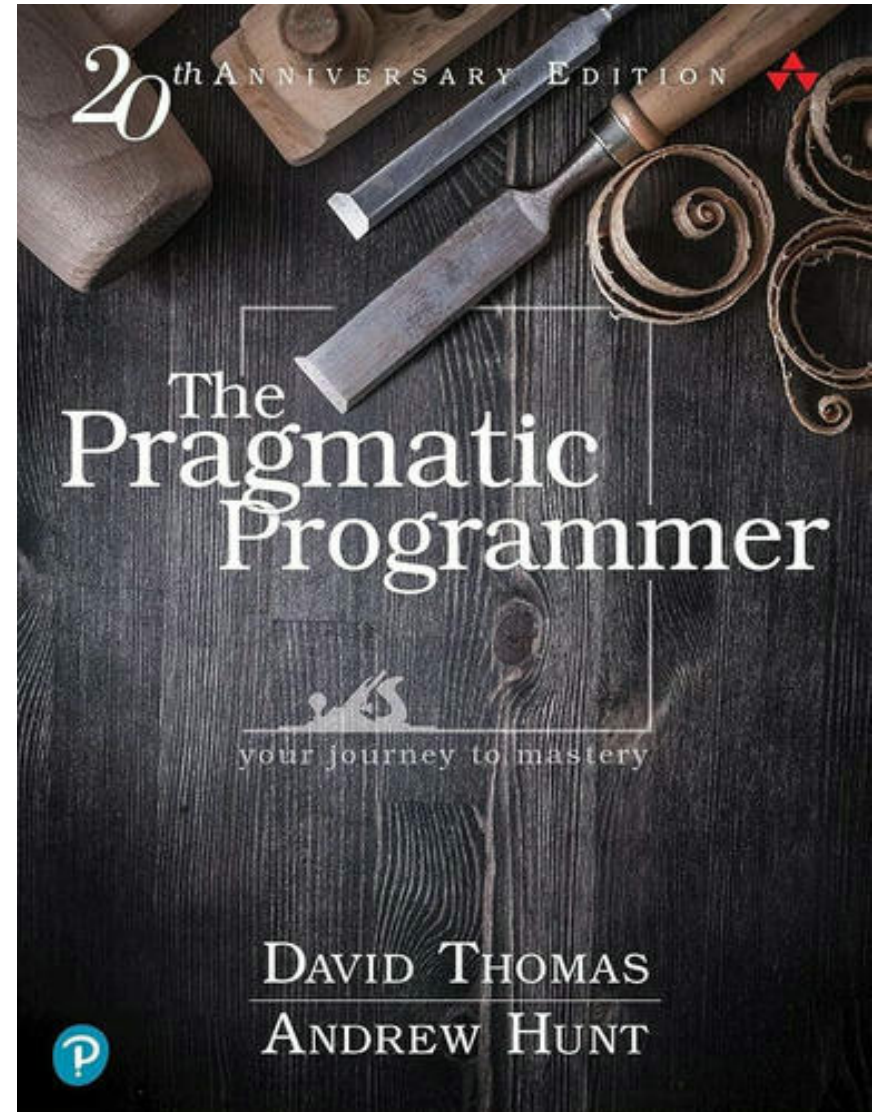
— Martin Fowler



La teoría de la ventana rota

Se rompe una ventana en un edificio de departamentos, pero nadie la arregla. Queda rota. Luego se rompe otra cosa. Tal vez sea un accidente, tal vez no, pero tampoco se arregla. Aparece un graffiti. Se acumula más y más daño. Muy rápidamente se obtiene un crecimiento exponencial: todo el edificio se deteriora. Los inquilinos se mudan. Entra el crimen. Y has perdido el juego.

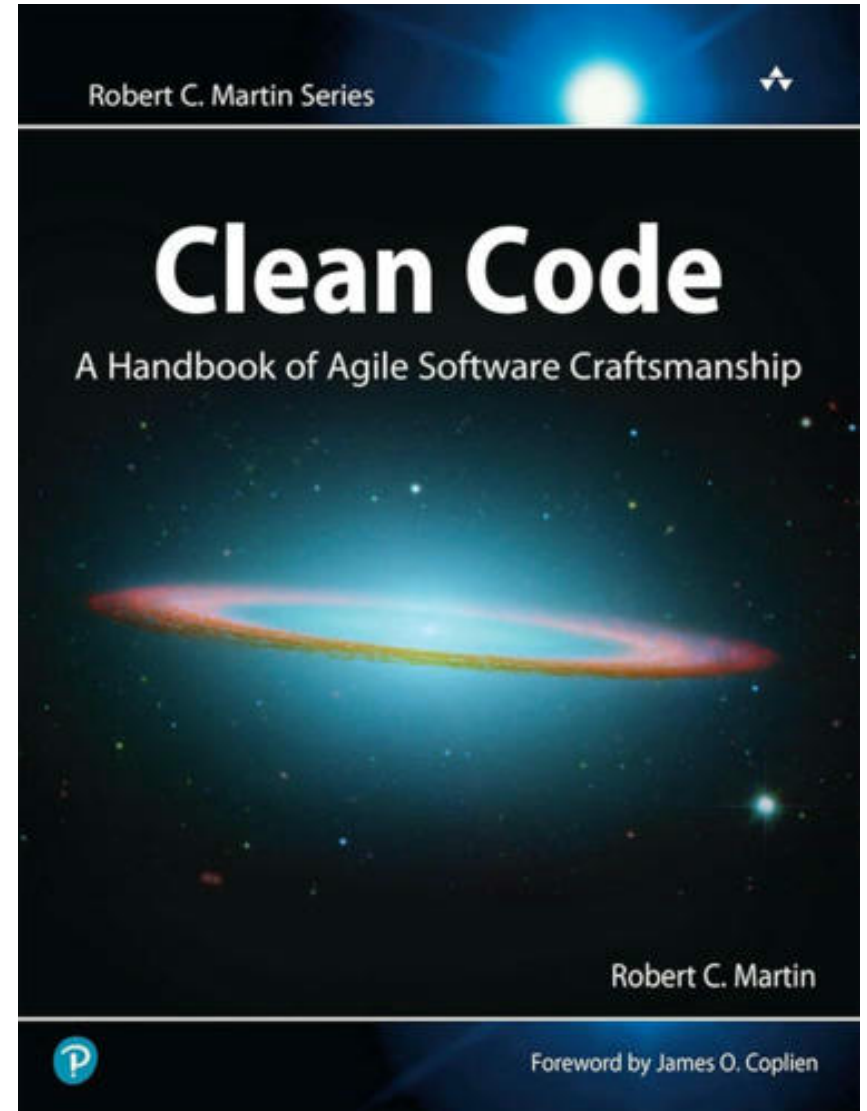
— Andrew Hunt



La regla de los Boy Scouts

*Siempre deja el campamento un
poco mejor de lo que lo
encontraste.*

— Robert “Uncle Bob” C. Martin



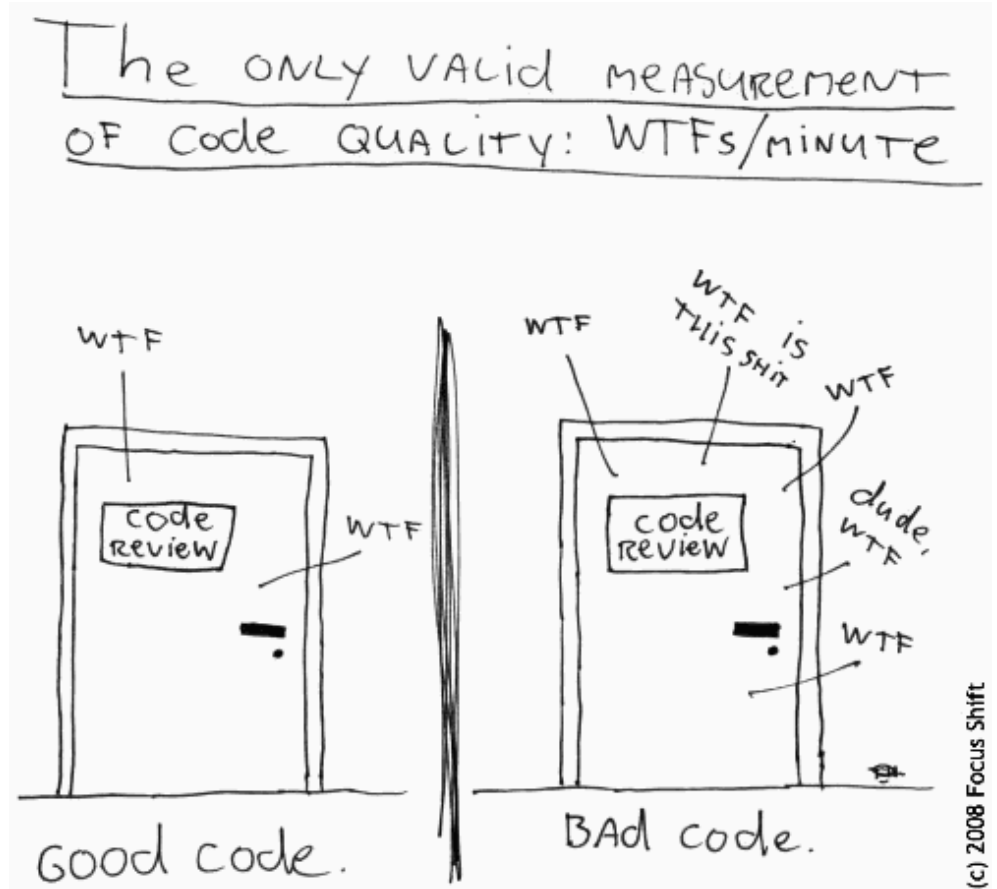


Principios de diseño

- YAGNI (You Ain't Gonna Need It)
- KISS (Keep It Simple, Stupid!)
- DRY (Don't Repeat Yourself)
- PoLA (Principle of Least Astonishment)
- KOP (Knuth's Optimization Principle)
- SoC (Separation of Concerns Principle)
- Alta cohesión, bajo acoplamiento

OOP:

- TDA (Tell, Don't Ask!)
- PoLK (Principle of Least Knowledge)
- EDP (Explicit Dependencies Principle)
- SOLID (SRP, OCP, LSP, ISP, DIP)
- Composition over inheritance



Una advertencia...

Ojo: los principios de diseño **no son dogmas**.

No se deben aplicar ciegamente, sino que deben ser considerados como guías para mejorar la calidad del código y la arquitectura del software.

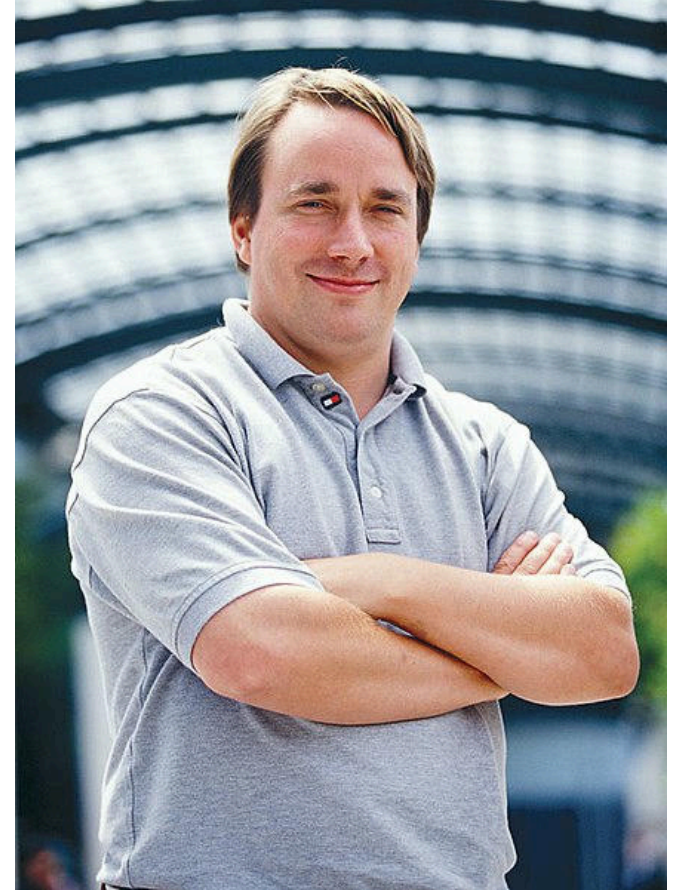
Muchos de estos principios pueden entrar en conflicto entre sí, y es importante evaluar cada situación y decidir cuál es el más adecuado para el contexto.



No solo es acerca del código

Los malos programadores se preocupan por el código. Los buenos programadores se preocupan por las estructuras de datos.

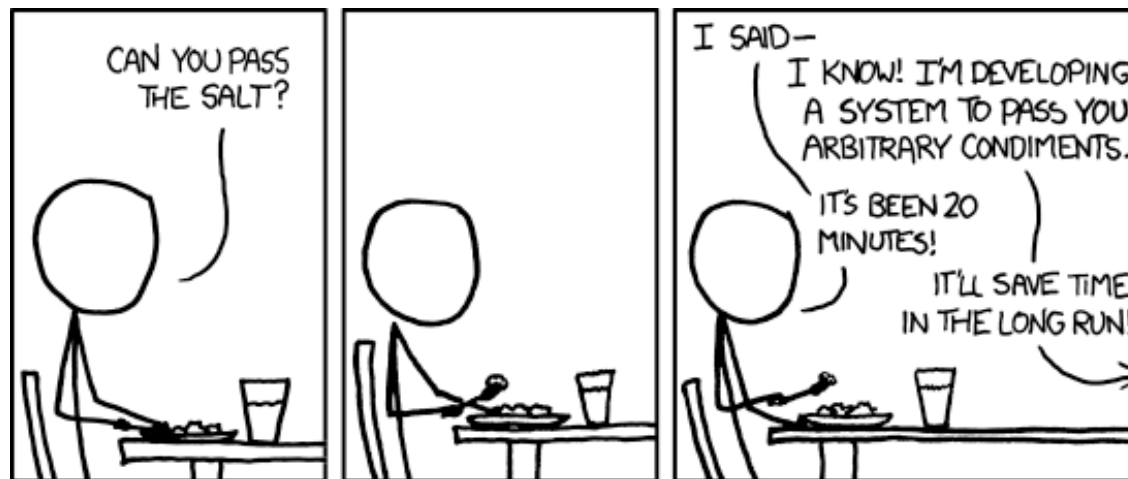
— Linus Torvalds





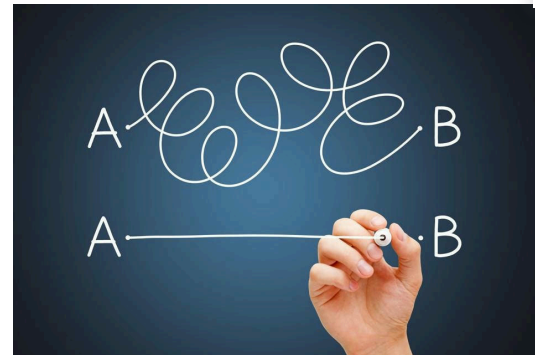
YAGNI: You Ain't Gonna Need It

¡No lo vas a necesitar!: Solo agregar una funcionalidad si es requerida.



KISS: Keep It Simple, Stupid!

¡Mantenlo simple, estúpido!



✗ No

```
public int esPar(int n) {  
    if (n < 0) {  
        n = -n;  
    }  
    if (n == 0) {  
        return true;  
    }  
    if (n == 1) {  
        return false;  
    }  
    return esPar(n - 2);  
}
```

✓ Sí

```
public int esPar(int n) {  
    return n % 2 == 0;  
}
```

KISS: Keep It Simple, Stupid! (cont.)

```
public class Punto {  
    private final int x;  
    private final int y;  
  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
}
```

✗ No

```
public class Punto {  
    public final int x;  
    public final int y;  
  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

✓ Sí

DRY: Don't Repeat Yourself

¡No te repitas!

```
public static void main(String[] args) {  
    System.out.println("¡No voy a repetir código!");  
    System.out.println("¡No voy a repetir código!");  
    System.out.println("¡No voy a repetir código!");  
    System.out.println("¡No voy a repetir código!");  
    System.out.println("¡No voy a repetir código!");  
}
```

✗ No

```
public static void main(String[] args) {  
    for (int i = 0; i < 5; i++) {  
        System.out.println("¡No voy a repetir código!");  
    }  
}
```

✓ Sí

DRY: Don't Repeat Yourself (cont.)

```
public void depositar(int monto) {  
    if (monto < 0) {  
        throw new IllegalArgumentException("...");  
    }  
    saldo += monto;  
}  
  
public void retirar(int monto) {  
    if (monto < 0) {  
        throw new IllegalArgumentException("...");  
    }  
    saldo -= monto;  
}
```

✗ No

```
void chequearMonto(int monto) {  
    if (monto < 0) {  
        throw new IllegalArgumentException("...");  
    }  
}  
  
public void depositar(int monto) {  
    chequearMonto(monto);  
    saldo += monto;  
}  
  
public void retirar(int monto) {  
    chequearMonto(monto);  
    saldo -= monto;  
}
```

✓ Sí

DRY: Don't Repeat Yourself (cont.)

```
public class CursoParadigmas {  
    public void inicio() {  
        System.out.println("Bienvenido a TB025!");  
    }  
  
    public void fin() {  
        System.out.println("Gracias por cursar TB025!");  
    }  
}
```

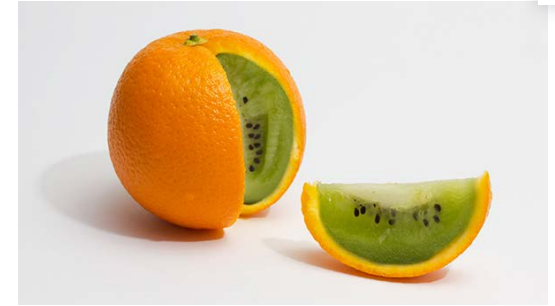
✗ No

```
public class CursoParadigmas {  
    public static final String CODIGO = "TB025";  
  
    public void mostrarMensaje(String mensaje) {  
        System.out.println(mensaje);  
    }  
  
    public void inicio() {  
        mostrarMensaje("Bienvenido a " + CODIGO + "!");  
    }  
  
    public void fin() {  
        mostrarMensaje("Gracias por cursar " + CODIGO + "!");  
    }  
}
```

✓ Sí

POLA: Principle of Least Astonishment

Principio del menor asombro: Un componente de un sistema debe comportarse como la mayoría de sus usuarios esperan que se comporte, y no sorprender con comportamientos inesperados.



```
public class CuentaBancaria {  
    private int saldo;  
  
    public void depositar(int monto) {  
        if (monto < 0) {  
            return;  
        }  
        saldo += monto;  
    }  
  
    public int getSaldo() {  
        return saldo;  
    }  
}
```

✗ No

```
public class CuentaBancaria {  
    private int saldo;  
  
    public void depositar(int monto) {  
        if (monto < 0) {  
            throw new IllegalArgumentException("...");  
        }  
        saldo += monto;  
    }  
  
    public int getSaldo() {  
        return saldo;  
    }  
}
```

✓ Sí

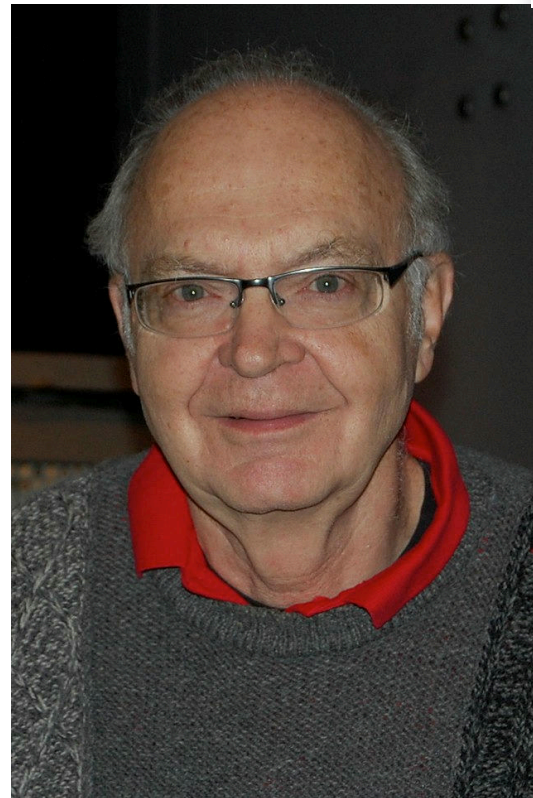
KOP: Knuth's Optimization Principle

Principio de optimización de Knuth: No optimizar el código prematuramente.

*The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; **premature optimization is the root of all evil** (or at least most of it) in programming.*

— Donald Knuth

En caso de decidir optimizar, **medir el rendimiento** (*profiling*) del código antes y después, para asegurarse de que la optimización realmente mejora el rendimiento.





SoC: Separation of Concerns

Separación de incumbencias: Dividir un sistema en partes independientes que abordan diferentes aspectos o dominios relacionados con el problema a resolver.

Presentación	Módulo	Módulo	Módulo
Lógica	Módulo	Módulo	Módulo
Persistencia	Módulo	Módulo	Módulo

Alta cohesión

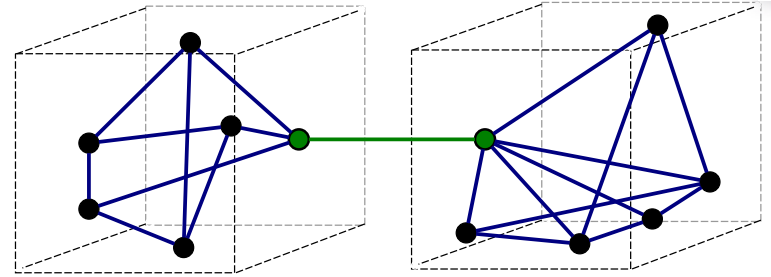
Cohesión: es la medida en que dos elementos de un módulo están relacionados entre sí.

Es preferible que los módulos tengan **alta cohesión**; es decir, que sus elementos (variables, funciones, clases, métodos) estén estrechamente relacionados y trabajen juntos para cumplir un propósito específico.

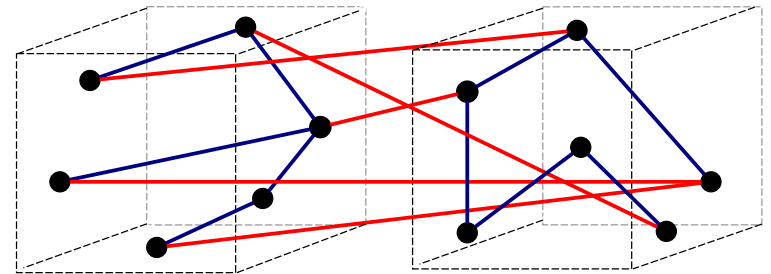
Bajo acoplamiento

Acoplamiento: es la medida en que un módulo depende de otros módulos.

Es preferible que los módulos tengan **bajo acoplamiento**; es decir, que dependan lo menos posible de otros módulos, lo que facilita su reutilización, mantenimiento y prueba.



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)



Alta cohesión

✗ Baja cohesión

```
void f() {  
    int x = 42;  
    // ...  
    // ...  
    // ...  
    // ...  
    // ...  
    System.out.println(x);  
    // ...  
    // ...  
    // ...  
    // ...  
    // ...  
}
```

✓ Alta cohesión

```
void f() {  
    // ...  
    // ...  
    // ...  
    // ...  
    // ...  
    int x = 42;  
    System.out.println(x);  
    // ...  
    // ...  
    // ...  
    // ...  
    // ...  
}
```

Bajo acoplamiento

✗ Alto acoplamiento

```
public class Cliente {  
    private String nombre;  
    private String apellido;  
  
    public String nombreCompleto() {  
        return nombre + " " + apellido;  
    }  
}
```

```
public class Presentacion {  
    private PrintStream out;  
  
    public void bienvenida(Cliente c) {  
        String s = "Bienvenido, " + c.nombreCompleto() + "!";  
        out.println(s);  
    }  
}
```

```
Presentacion presentacion = new Presentacion(System.out);  
Cliente cliente = new Cliente("Juan", "Pérez");  
presentacion.bienvenida(cliente);
```

✓ Bajo acoplamiento

```
public class Mensajes {  
    public String bienvenida(String nombreCompleto) {  
        return "Bienvenido, " + nombreCompleto + "!";  
    }  
}  
  
public class Presentacion {  
    private PrintStream out;  
  
    public void mostrarMensaje(String s) {  
        out.println(s);  
    }  
}
```

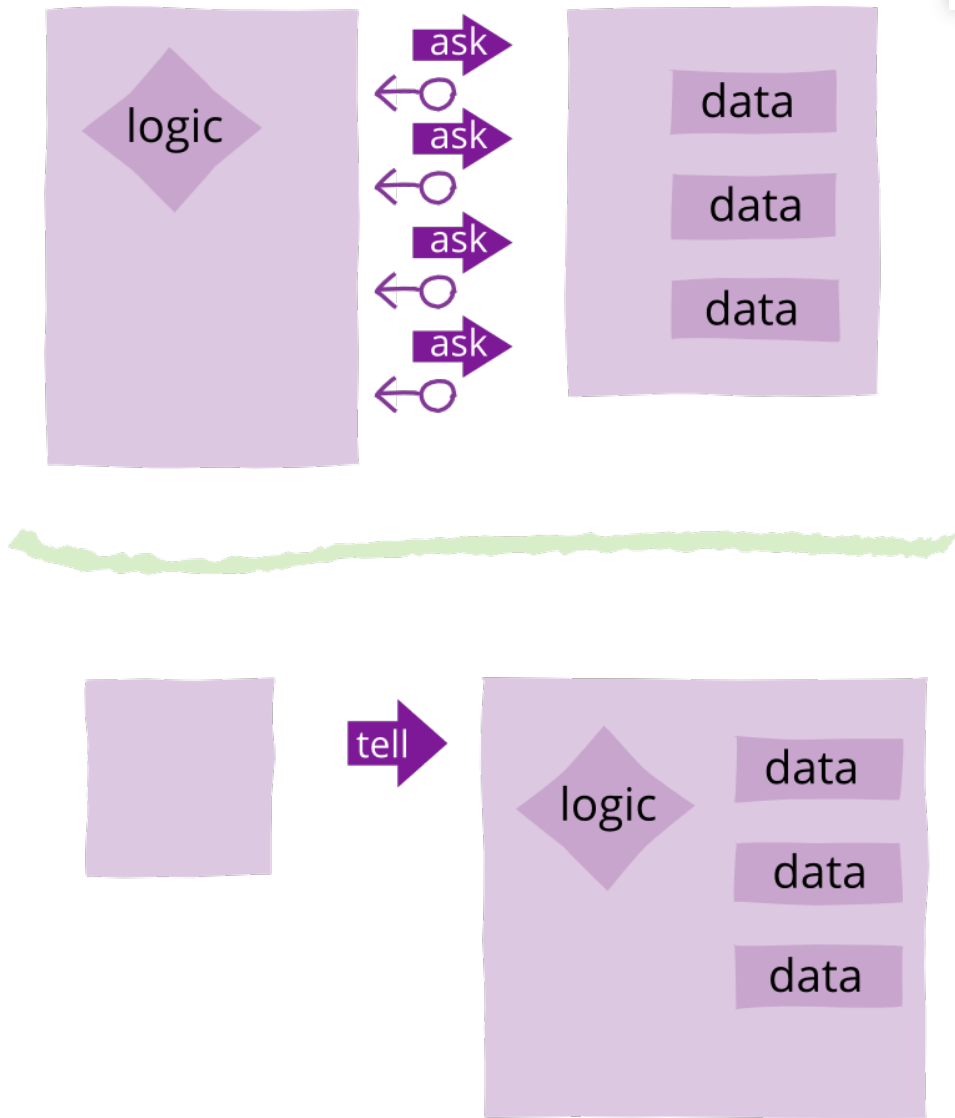
```
Mensajes mensajes = new Mensajes();  
Presentacion presentacion = new Presentacion(System.out);  
Cliente cliente = new Cliente("Juan", "Pérez");  
String mensaje = mensajes.bienvenida(cliente.nombreCompleto());  
presentacion.mostrarMensaje(mensaje);
```



TDA: Tell, don't ask!

¡Di qué hacer, no preguntes!

- ✗ No: Solicitarle a un objeto que indique su estado y luego realizar una acción en base a su respuesta.
- ✓ Sí: Solicitarle al objeto que lleve a cabo la acción él mismo.



TDA: Tell, don't ask! (cont.)

```
class Logger {  
    public boolean habilitado;  
  
    public void log(String message) {  
        System.out.println(message);  
    }  
}  
  
public class CuentaBancaria {  
    private Logger logger;  
    private int saldo;  
  
    public CuentaBancaria(Logger logger) {  
        this.logger = logger;  
    }  
  
    public void depositar(int monto) {  
        saldo += monto;  
        if (logger.habilitado) {  
            logger.log("Depositando " + monto);  
        }  
    }  
}
```

✗ No

```
class Logger {  
    public boolean habilitado;  
  
    public void log(String message) {  
        if (habilitado) {  
            System.out.println(message);  
        }  
    }  
}  
  
public class CuentaBancaria {  
    private Logger logger;  
    private int saldo;  
  
    public CuentaBancaria(Logger logger) {  
        this.logger = logger;  
    }  
  
    public void depositar(int monto) {  
        logger.log("Depositando " + monto);  
        saldo += monto;  
    }  
}
```

✓ Sí



PoLK: Principle of Least Knowledge

Principio del menor conocimiento o Ley de Demeter: Para promover el bajo acoplamiento, cada módulo debería conocer lo menos posible sobre otros módulos.

Aplicado a objetos, un método `f` de una clase `C` solo debería invocar métodos de:

- la propia clase `C`;
- los objetos que son atributos de `C`;
- los objetos recibidos por `f` como argumentos;
- los objetos instanciados en `f`.

PoLK: Principle of Least Knowledge (cont.)

```
class Universidad {  
    private List<Carrera> carreras;  
  
    public List<Estudiante> getInscriptos(String codCarrera,  
                                           String codCurso) {  
        Carrera car = buscarCarrera(codCarrera);  
        Curso cur = car.buscarCurso(codCurso);  
        return cur.getInscriptos();  
    }  
}  
  
class Carrera {  
    private List<Curso> cursos;  
  
    public Curso buscarCurso(String codCurso) {  
        // ...  
    }  
}
```

✗ No

```
class Universidad {  
    private List<Carrera> carreras;  
  
    public List<Estudiante> getInscriptos(String codCarrera,  
                                           String codCurso) {  
        Carrera car = buscarCarrera(codCarrera);  
        return car.getInscriptos(codCurso);  
    }  
}  
  
class Carrera {  
    private List<Curso> cursos;  
  
    public List<Estudiante> getInscriptos(String codCurso) {  
        // ...  
    }  
  
    public Curso buscarCurso(String codCurso) {  
        // ...  
    }  
}
```

✓ Sí



EDP: Explicit Dependencies Principle

Las clases y métodos deben requerir explícitamente los objetos necesarios para funcionar correctamente, en lugar de asumir que están disponibles en el contexto.

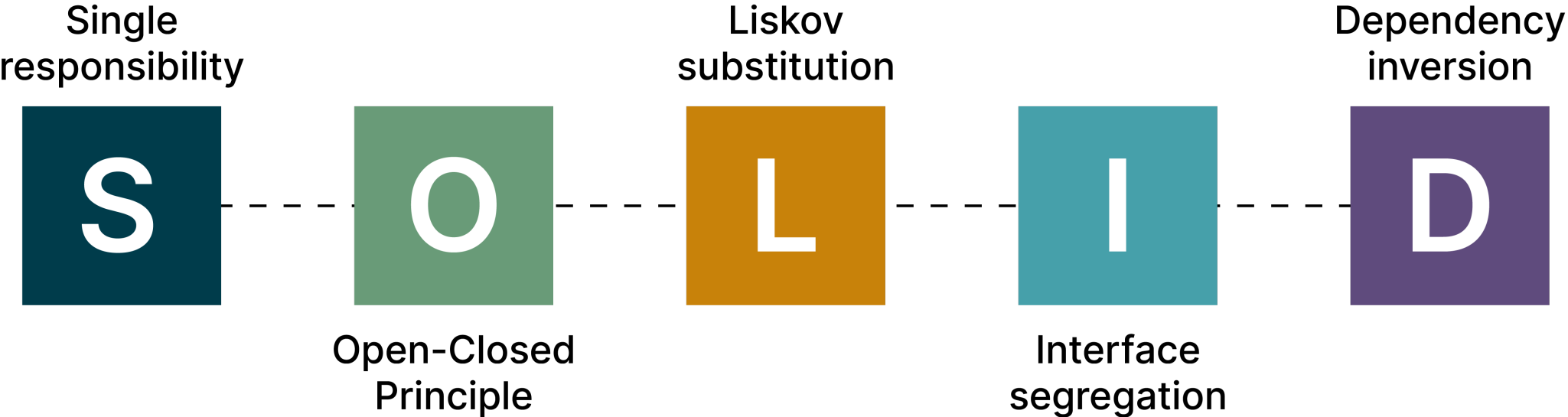
```
public class Logger {  
    public void log(String s) {  
        System.out.println(s);  
    }  
}
```

✗ No

```
public class Logger {  
    private final PrintStream out;  
  
    public Logger(PrintStream out) {  
        this.out = out;  
    }  
  
    public void log(String s) {  
        out.println(s);  
    }  
}
```

✓ Sí

Principio SOLID



SRP: Single Responsibility Principle

Principio de responsabilidad única: Cada clase debe tener una única responsabilidad o propósito.

```
public class Producto {  
    public String nombre;  
    public double valor;  
  
    public String toString() {  
        return "Producto: " + nombre + ", Valor: " + valor;  
    }  
  
    public void guardar() {  
        FileWriter writer = new FileWriter("productos.txt");  
        writer.write(toString() + "\n");  
        writer.close();  
    }  
  
    public void mostrar() {  
        System.out.println(toString());  
    }  
}
```

✗ No

```
public class Producto {  
    public String nombre;  
    public double valor;  
  
    public String toString() {  
        return "Producto: " + nombre + ", Valor: " + valor;  
    }  
}  
  
public class BaseDeDatos {  
    public void guardarProducto(Producto producto) {  
        FileWriter writer = new FileWriter("productos.txt");  
        writer.write(producto.toString() + "\n");  
        writer.close();  
    }  
}  
  
public class Consola {  
    public void mostrarProducto(Producto producto) {  
        System.out.println(producto.toString());  
    }  
}
```

✓ Sí

OCP: Open/Closed Principle

Principio abierto/cerrado: Las clases deben estar abiertas para su extensión, pero cerradas para su modificación.

Es decir, debe ser posible agregar nuevas funcionalidades a una clase sin modificar su código existente.

```
public class Enemigo {  
    String tipo;  
  
    public int getAtaque() {  
        switch (tipo) {  
            case "goblin": return 10;  
            case "orco": return 15;  
            case "dragon": return 35;  
        }  
    }  
  
    public int getDefensa() {  
        switch (tipo) {  
            case "goblin": return 5;  
            case "orco": return 10;  
            case "dragon": return 20;  
        }  
    }  
}
```

✗ No

```
public interface Enemigo {  
    int getAtaque();  
    int getDefensa();  
}  
  
public class Goblin implements Enemigo {  
    public int getAtaque() { return 10; }  
    public int getDefensa() { return 5; }  
}  
  
public class Orco implements Enemigo {  
    public int getAtaque() { return 15; }  
    public int getDefensa() { return 10; }  
}  
  
public class Dragon implements Enemigo {  
    public int getAtaque() { return 35; }  
    public int getDefensa() { return 20; }  
}
```

✓ Sí



OCP: Tipos enumerativos

```
public enum TipoEnemigo {
    GOBLIN(10, 5),
    ORCO(15, 10),
    DRAGON(35, 20);

    private final int ataque;
    private final int defensa;

    Enemigo(int ataque, int defensa) {
        this.ataque = ataque;
        this.defensa = defensa;
    }

    public int getAtaque() {
        return ataque;
    }

    public int getDefensa() {
        return defensa;
    }
}
```

```
public class Enemigo {
    public final TipoEnemigo tipo;

    public Enemigo(TipoEnemigo tipo) {
        this.tipo = tipo;
    }
}
```

LSP: Liskov Substitution Principle

Principio de sustitución de Liskov: Una instancia de una clase debe poder ser sustituida por una instancia de una clase derivada sin “romper” el comportamiento del programa.



```
public class Naipe {  
    private String palo;  
    private int numero;  
  
    public Naipe(String palo, int numero) {  
        this.palo = palo;  
        this.numero = numero;  
    }  
  
    public String getPalo() { return palo; }  
  
    public int getNumero() { return numero; }  
  
    public String mostrar() {  
        return "%s de %d".formatted(numero, palo);  
    }  
}
```

✗ No

```
public class Comodin extends Naipe {  
    public Comodin() {  
        super("Comodín", 0);  
    }  
  
    @Override public String getPalo() {  
        throw new UnsupportedOperationException("Un comodín no tiene palo");  
    }  
  
    @Override public int getNumero() {  
        throw new UnsupportedOperationException("Un comodín no tiene número");  
    }  
  
    public String mostrar() {  
        return "Comodín";  
    }  
}
```


LSP: Liskov Substitution Principle (cont.)

✓ Sí:

```
public interface Naipe {  
    public String mostrar();  
}
```

```
public class NaipeComun implements Naipe {  
    private String palo;  
    private int numero;  
  
    public NaipeComun(String palo, int numero) {  
        this.palo = palo;  
        this.numero = numero;  
    }  
  
    public String getPalo() { return palo; }  
  
    public int getNumero() { return numero; }  
  
    public String mostrar() {  
        return "%s de %d".formatted(numero, palo);  
    }  
}  
  
public class Comodin extends Naipe {  
    public String mostrar() {  
        return "Comodín";  
    }  
}
```

DIP: Dependency Inversion Principle

Principio de inversión de dependencias:

- Las clases de alto nivel no deben depender de clases de bajo nivel; ambas deben depender de abstracciones.
- Las abstracciones no deben depender de detalles; los detalles deben depender de las abstracciones.

```
class Auto {  
    public void conducir() { ... }  
}  
  
class Valet {  
    public void estacionar(Auto auto) {  
        auto.conducir();  
    }  
}
```

✗ No

```
interface Conducible {  
    void conducir();  
}  
  
class Auto implements Conducible { ... }  
  
class Valet {  
    public void estacionar(Conducible c) { ... }  
}
```

✓ Sí

ISP: Interface Segregation Principle

Principio de segregación de interfaces: Una clase no debe depender de métodos de otras clases que no utiliza.

```
class Auto {  
    public void llenarTanque() { ... }  
    public void inflarRuedas() { ... }  
    public void verMotor() { ... }  
    public void usarRadio() { ... }  
    public void conducir() { ... }  
}  
  
class Mecanico {  
    public void reparar(Auto auto) {  
        auto.verMotor();  
        auto.llenarTanque();  
        auto.inflarRuedas();  
    }  
}  
  
class Valet {  
    public void estacionar(Auto auto) {  
        auto.conducir();  
    }  
}
```

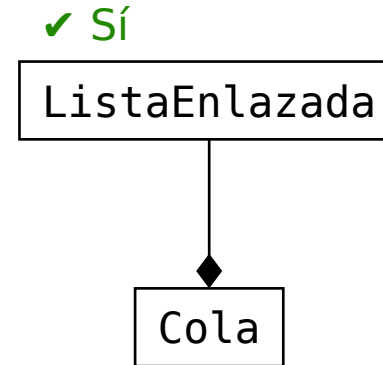
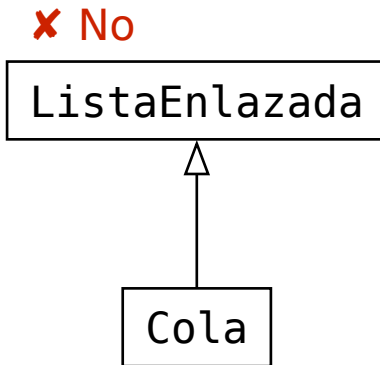
✗ No

```
interface Reparable {  
    void verMotor();  
    void llenarTanque();  
    void inflarRuedas();  
}  
  
interface Conducible {  
    void conducir();  
}  
  
class Auto implements Reparable, Conducible { ... }  
  
class Mecanico {  
    public void reparar(Reparable r) { ... }  
}  
  
class Valet {  
    public void estacionar(Conducible c) { ... }  
}
```

✓ Sí

Preferir composición sobre herencia

Preferir la composición de objetos en lugar de la herencia para reutilizar código y extender funcionalidades.



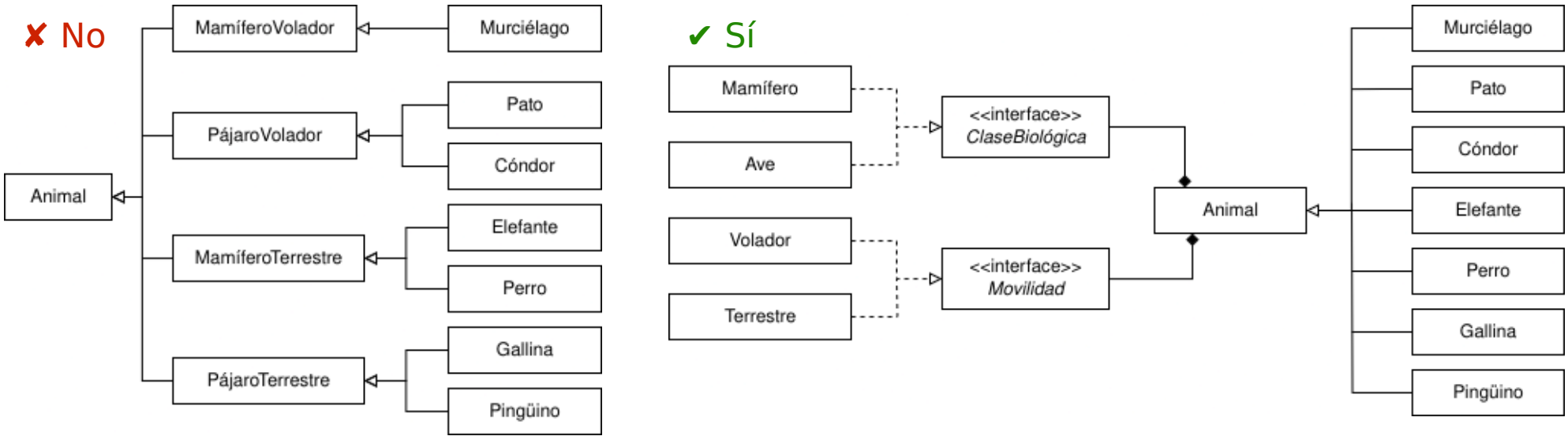
Preferir composición sobre herencia (cont.)

	Composición	Interfaces	Herencia
Reutilización de código	✓ Sí	✗ No	✓ Sí
Polimorfismo	✗ No	✓ Sí	✓ Sí
Acoplamiento	✓ Bajo	✓ Bajo	✗ Alto



Preferir composición sobre herencia (cont.)

Los **Patrones de diseño** como **Strategy** y **Bridge** suelen combinar la composición con interfaces para lograr un diseño flexible y extensible.



www.ingenieria.uba.ar

f    /ingenieriauba

 /FIUBAoficial