.UBAfiuba
FACULTAD DE INGENIERÍA

Principios de diseño

TB025 - Paradigmas de Programación

Curso Essaya

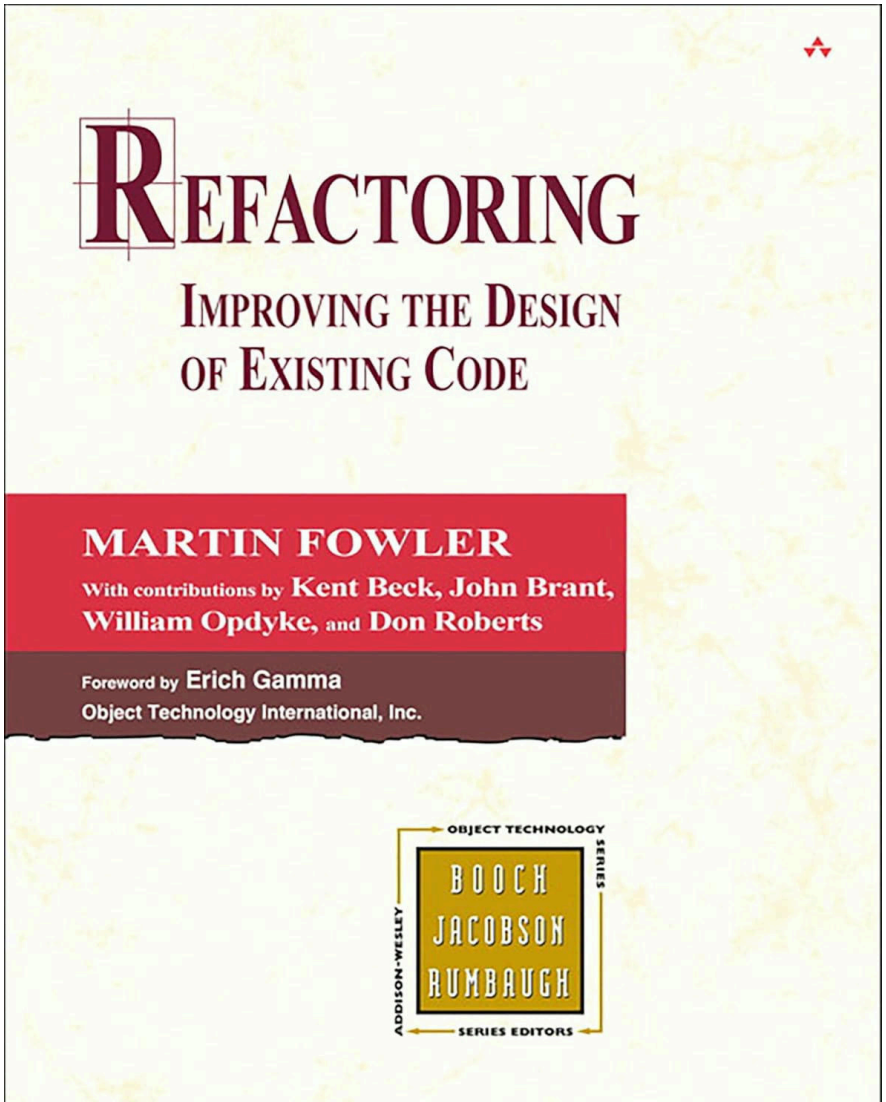
Algunas definiciones

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. #

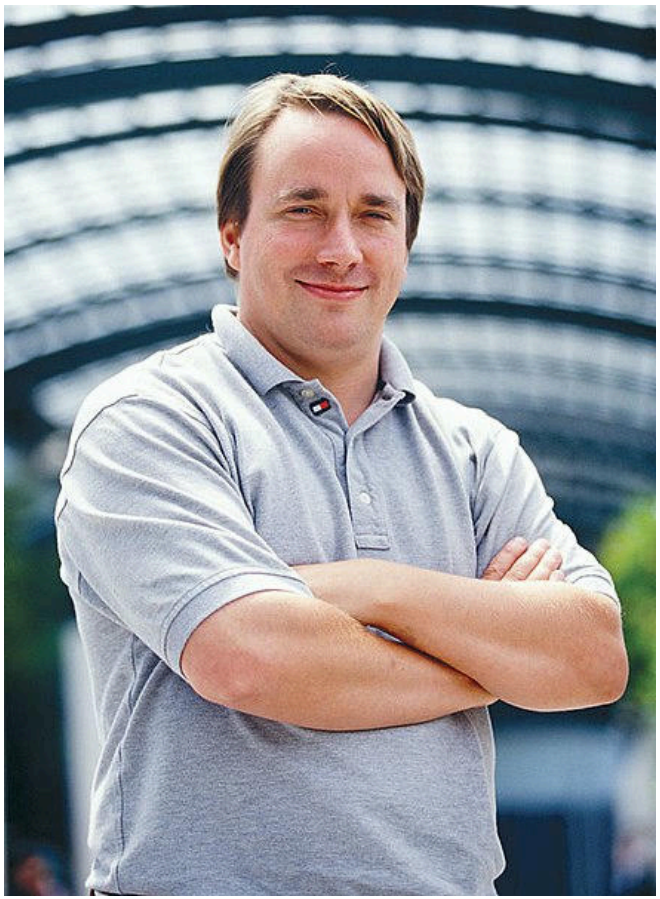
> *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*
>
> *— Martin Fowler*



No solo es acerca del código

...git actually has a simple design, with stable and reasonably well-documented data structures. In fact, I'm a huge proponent of designing your code around the data, rather than the other way around, and I think it's one of the reasons git has been fairly successful [...] I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. ***Bad programmers worry about the code. Good programmers worry about data structures and their relationships.***

— Linus Torvalds

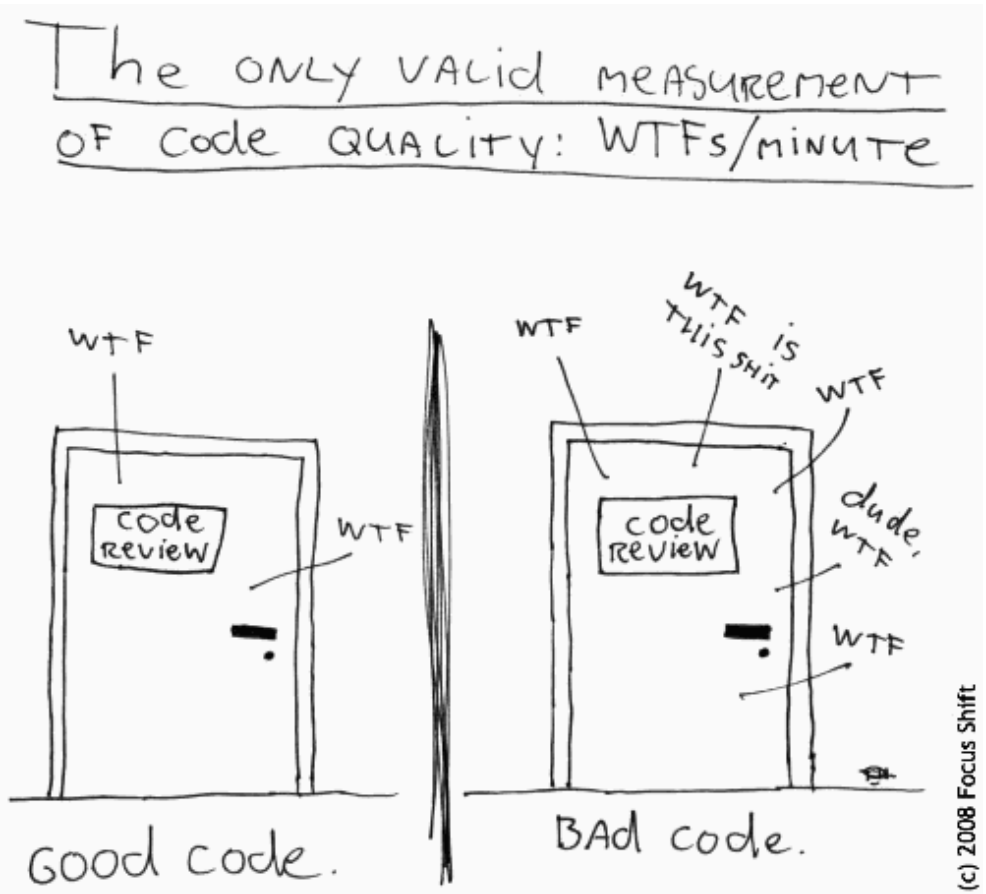


Principios de diseño

- 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:

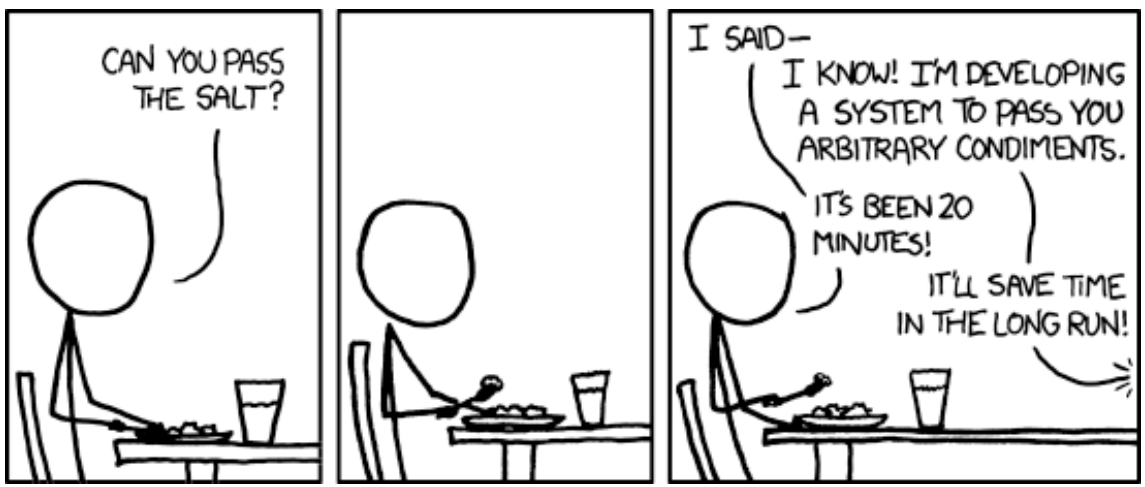
- TDA (Tell, Don’t Ask!)
- PoLK (Principle of Least Knowledge)
- EDP (Explicit Dependencies Principle)
- SOLID (SRP, OCP, LSP, ISP, DIP)
- Composition over inheritance



Fuente

YAGNI: You Ain’t Gonna Need It

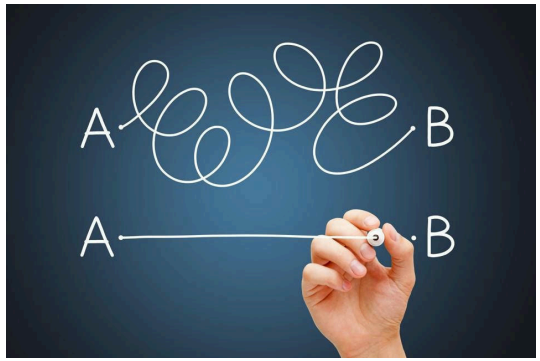
¡No lo vas a necesitar!: Solo agregar una funcionalidad si es requerida.



Fuente

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;
}
```

Fuente

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

✘ No

```
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;
    }
}
```

✔ Sí

```
public class Punto {
    public final int x;
    public final int y;

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

DRY: Don’t Repeat Yourself

¡No te repitas!

✘ No

```
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!");
}
```

✔ Sí

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

DRY: Don’t Repeat Yourself (cont.)

✘ No

```
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;
}
```

✔ Sí

```
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;
}
```

DRY: Don’t Repeat Yourself (cont.)

✘ No

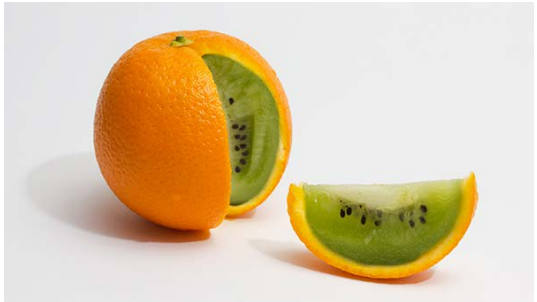
```
public class CursoParadigmas {
  public void inicio() {
    System.out.println("Bienvenido a TB025!");
  }

  public void fin() {
    System.out.println("Gracias por cursar TB025!");
  }
}
```

✔ Sí

```
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 + "!");
  }
}
```

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.



✘ No

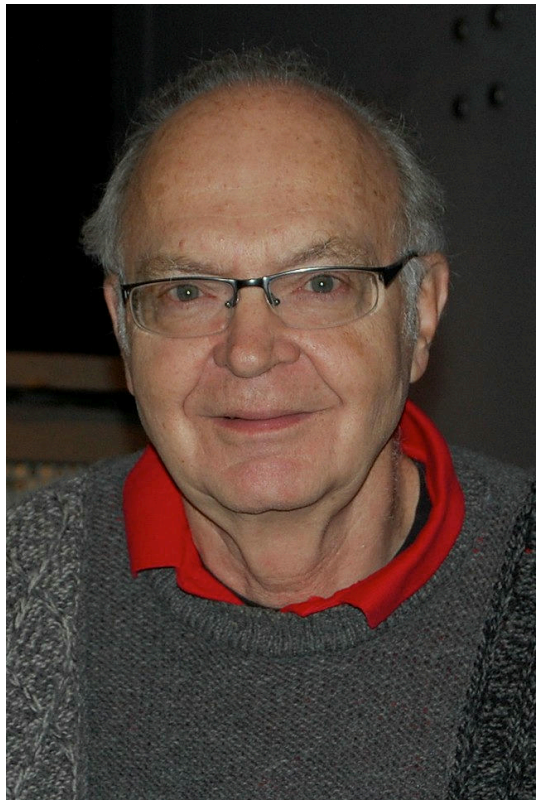
```
public class CuentaBancaria {
    private int saldo;

    public void depositar(int monto) {
        if (monto < 0) {
            return;
        }
        saldo += monto;
    }

    public int getSaldo() {
        return saldo;
    }
}
```

✔ Sí

```
public class CuentaBancaria {
    private int saldo;

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

    public int getSaldo() {
        return saldo;
    }
}
```

Fuente

KOP: Knuth’s Optimization Principle



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

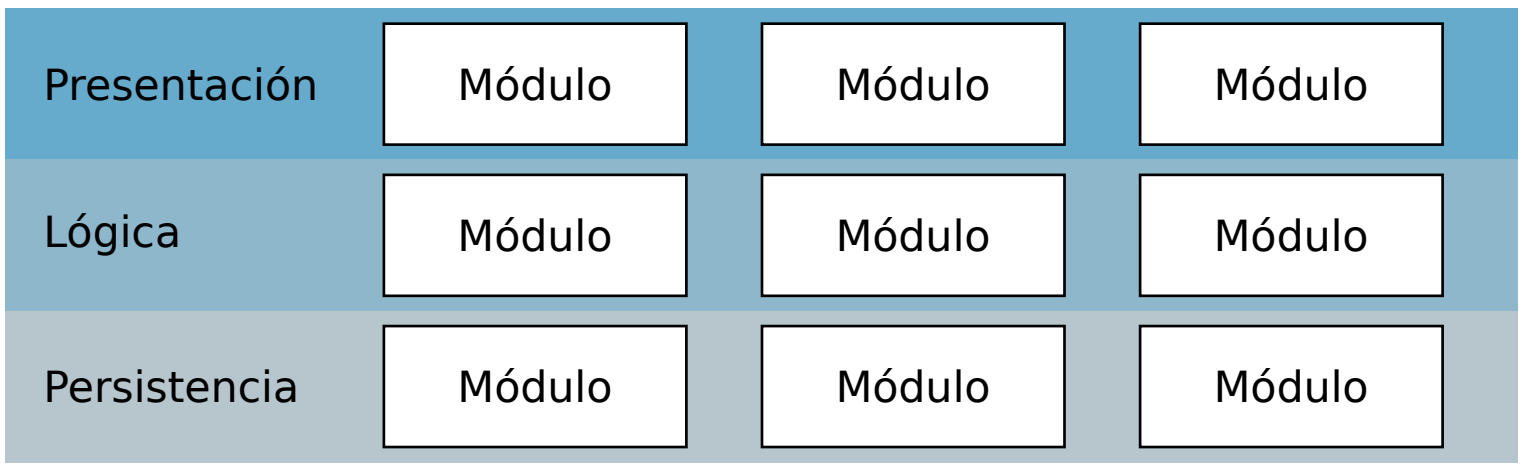
> *The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times;* ***premature optimization is the root of all evil*** *(or at least most of it) in programming.*
>
> *— Donald Knuth*

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

Fuente

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.



Fuente

Alta cohesión

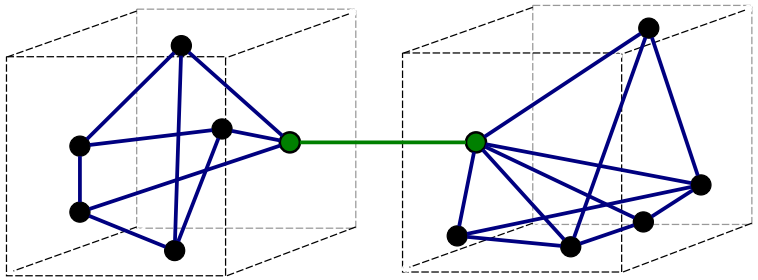
Cohesión: es la medida en que dos elementos de un módulo están relacionados entre sí.

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.

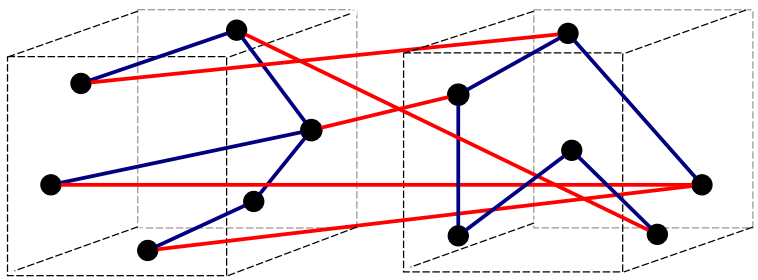
Bajo acoplamiento

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)

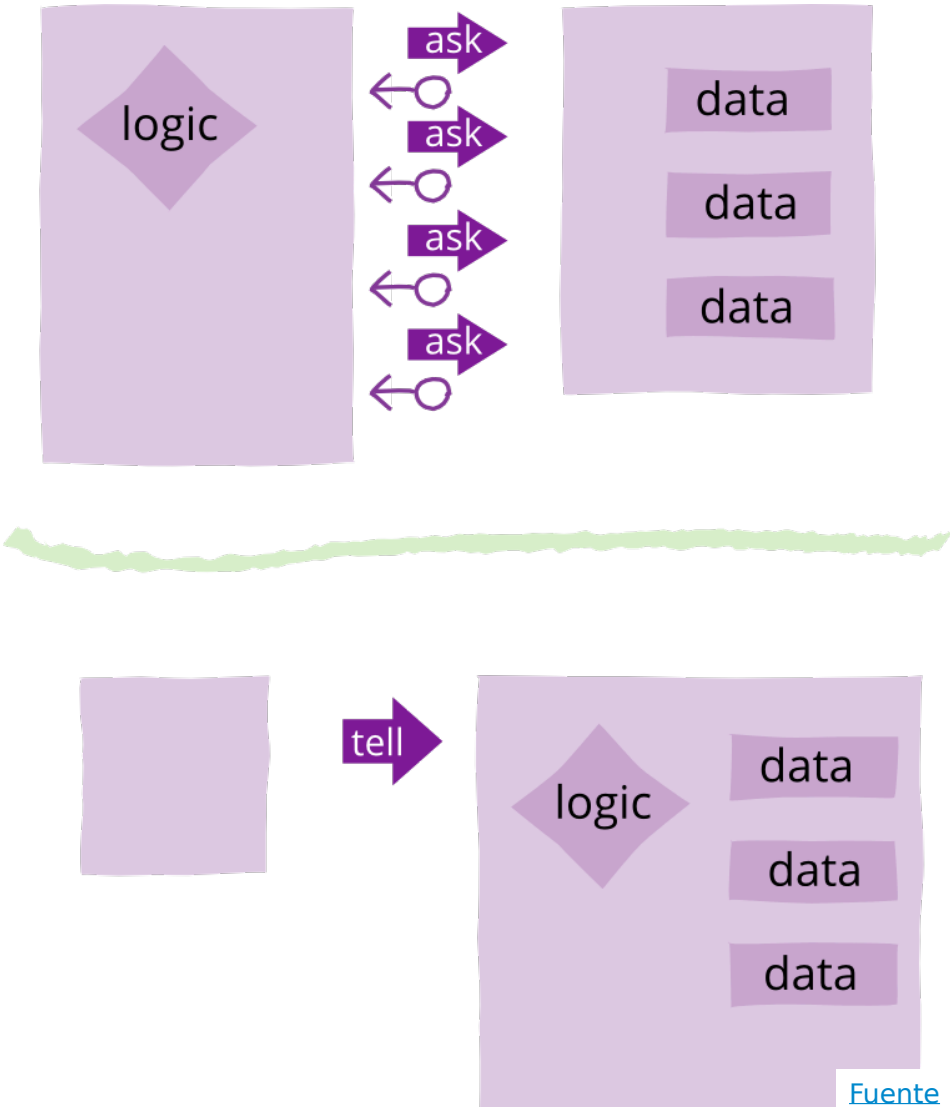
Fuente

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.



Fuente

TDA: Tell, don’t ask! (cont.)

✘ No

```
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);
        }
    }
}
```

✔ Sí

```
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;
    }
}
```

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`.

Fuente

PoLK: Principle of Least Knowledge (cont.)

✘ No

```
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) {
    // ...
  }
}
```

✔ Sí

```
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) {
    // ...
  }
}
```

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.

✘ No

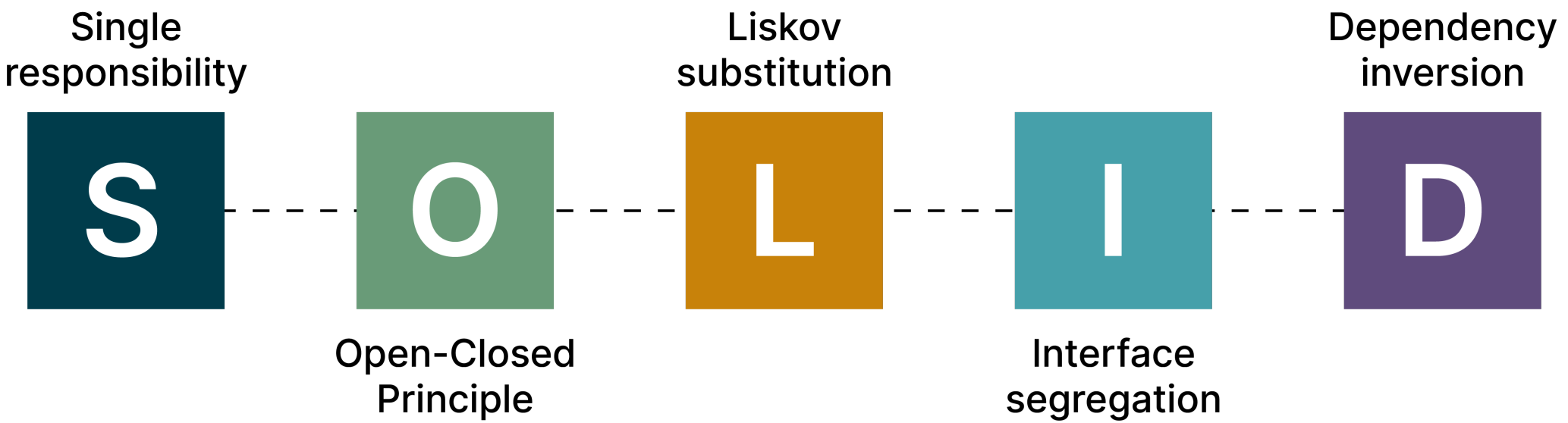
```
public class Logger {
    public void log(String s) {
        System.out.println(s);
    }
}
```

✔ Sí

```
public class Logger {
    private final PrintStream out;

    public Logger(PrintStream out) {
        this.out = out;
    }

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

Fuente

Principio SOLID



Fuente

SRP: Single Responsibility Principle

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

✘ No

```
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());
    }
}
```

✔ Sí

```
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());
```

```
    }
}
```

www.**ingenieria**.uba.ar

/ingenieriauba

/FIUBAoficial