

Principios SOLID

UNRN

Universidad Nacional
de Río Negro



**Antes de
empezar
unos conceptos**

Deuda

técnica

Causas

- Presión de tiempo
- Falta de conocimiento
- Requisitos cambiantes
- Falta de pruebas

Tipos

- Deliberada
- Accidental

Consecuencias

- Aumento del costo y tiempo de desarrollo
- Disminución de la calidad del software
- Frustración del equipo
- Pérdida del sueño y cabello

¿Cómo gestionarla?

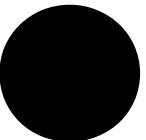
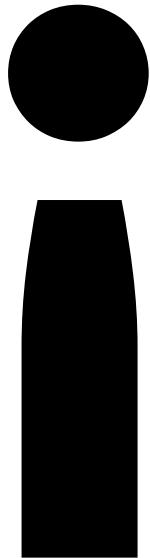
- Identificación y seguimiento
- Priorización
- Prevención*
- Refactorización*

El plan de
pagos

Aplica a

más que nada a

**Software de
gran escala**



**Está más que claro que la
deuda técnica aplica a un
programa que se
desarrolla continuamente**

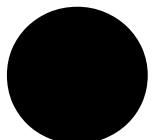
—

**Y es un nombre
general para los
defectos y atajos**



¿Preguntas?





Cómo identificar deuda técnica

{code/design} smells

**Cuanto
dependemos de
la estructura
interna de otro
objeto**

Acoplamiento



Rigidez

Rigidez

Si quieres cambiar el comportamiento de una funcionalidad, debes modificar la clase central, lo que puede afectar a otras partes del sistema que dependen de ella.

Fragilidad

Fragilidad

Un sistema con un alto acoplamiento entre clases. Si modificas una clase, es probable que afecte a otras clases que dependen de ella, lo que puede provocar errores en cascada.

Inseparabilidad

Inseparabilidad

Un sistema donde una clase tiene múltiples responsabilidades. Si quieres reutilizar una parte de la funcionalidad de esa clase, debes llevarla completa, incluso las partes que no necesitas.

Viscosidad

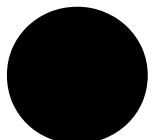
Viscosidad

Un sistema donde es más fácil añadir un método a una clase existente que crear una nueva clase para una nueva funcionalidad. Esto puede llevar a clases sobrecargadas con responsabilidades y dificultar la comprensión y el mantenimiento del código.

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?





Cómo podemos prevenir la deuda técnica

SOLID

Principios base para software orientado a objetos sólido

SOLID... ¿snake?



Qúe son Principios de diseño

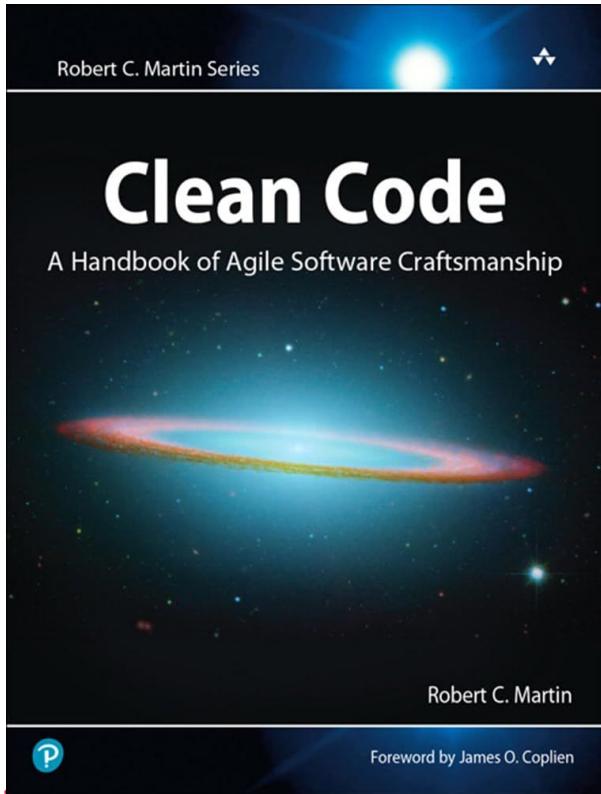
SOLID

SOLID

Es un acrónimo de cinco principios de diseño que facilitan el desarrollo del software orientado a objetos.

Robert Martin

~2000



Estos lineamientos

no

son dogmas

**Siempre podemos dejar
deuda técnica para
después *también***



**Pero saber que hemos
tomado un atajo es
importante**

Principle

S_RP
O_CP
L_SP
I_SP
D_IP

Single Responsibility
Open/Closed
Liskov Substitution
Interface Segregation
Dependency Inversion

Principio

S**RP**
O**CP**
L**SP**
I**SP**
D**IP**

Responsabilidad única
Abierto/Cerrado
Substitución de Liskov
Segregación de interfaces
Inversión de dependencias

Principio de responsabilidad única

S

R

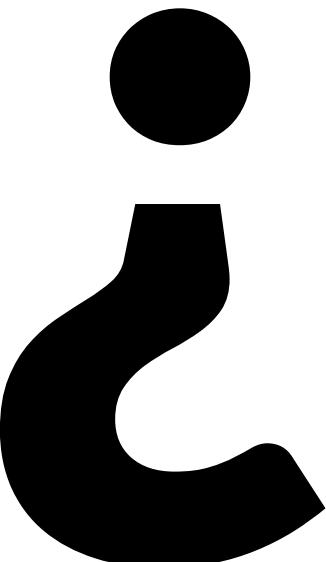
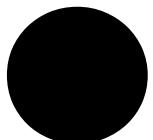
P

Una clase debe tener una, y solo una,
razón para cambiar.

Ejemplo

```
class Suma extends Operacion{
    public int calcular(int a, int b) {
        return a + b;
    }
}
```

Implementa la Operación de cálculo, Suma



Cómo identificar múltiples responsabilidades

¿Cuáles son las responsabilidades de esta clase?

```
class Factura {  
    // Atributos de la factura  
    public double calcularTotal() {  
        // Lógica para calcular el total  
    }  
    public void generarPDF() {  
        // Lógica para generar el PDF  
    }  
    public void enviarPorCorreo(String destinatario) {  
        // Lógica para enviar por correo  
    }  
}
```

En líneas generales

Separar las responsabilidades en diferentes clases y que el comportamiento se pueda asignar de manera dinámica.

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?



Principio Abierto/Cerrado

O

C

P

Establece que las clases deben estar abiertas a la extensión, pero cerradas a la modificación.



Calculadora

Sí queremos agregar una nueva
Operacion;
¿qué tenemos que cambiar?

y si queremos agregar nuevas operaciones



```
public class OperacionBinaria{  
  
    public String toString(){  
        if (this.getClassName().equals("Suma")) {  
            return izquierdo + "+" + derecho;  
        // resto de los operadores.  
    }  
}
```

¡Igual funciona!

Dada esta clase para filtrar

```
interface Filtrador<T> {  
    boolean aplicar(T instancia);  
}
```

Y diversas implementaciones

```
class FiltroColor implements Filtrador<Producto> {  
    // ...  
}
```

```
class FiltroTamaño implements Filtrador<Producto> {  
    // ...  
}
```

Podemos implementar un filtro con el criterio 'dinámico'

```
class Productos {  
    List<Producto> bodega;  
  
    public List<Producto> filtrar(Filtrador<Producto> filtro) {  
        List<Producto> filtrado = new LinkedList<>();  
        for(Producto p: bodega){  
            if(filtro.aplicar(p)){  
                filtrado.add(p);  
            }  
        }  
    }  
}
```

La lista de Producto quizas sea un atributo

En líneas generales

Usar interfaces o clases abstractas para definir el comportamiento común y crear clases con el comportamiento específico.

Sin depender de cambios en el código existente

A grandes rasgos,
no usar
getters / setters



¿Preguntas?

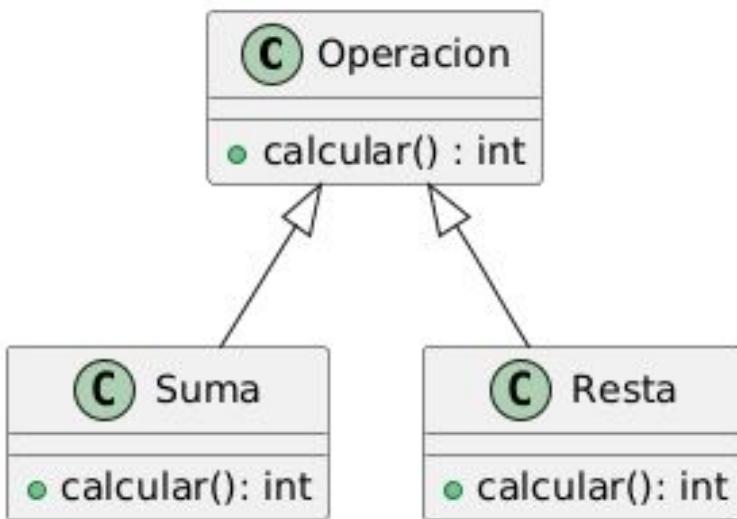


Principio de substitución de Liskov*

L
S
P

Los objetos de una clase base deben ser reemplazables por instancias de sus clases derivadas sin alterar el comportamiento del programa.

calcular en Operacion



Donde se espera una Operacion, puede ir cualquier subclase

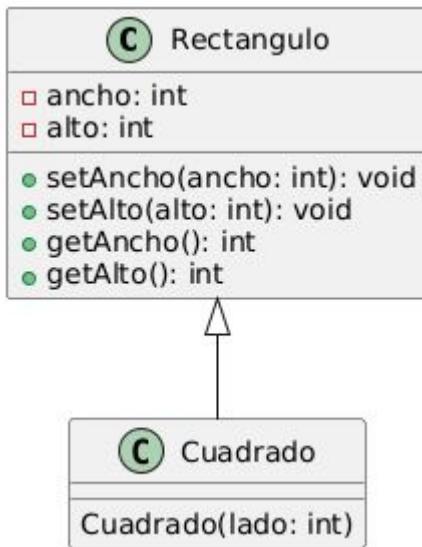
Sí tenemos que

Hacer *pattern matching / instanceof* en todas las llamadas, entonces no es reemplazable o flexible

El comportamiento no es homogéneo.

Figuras geométricas

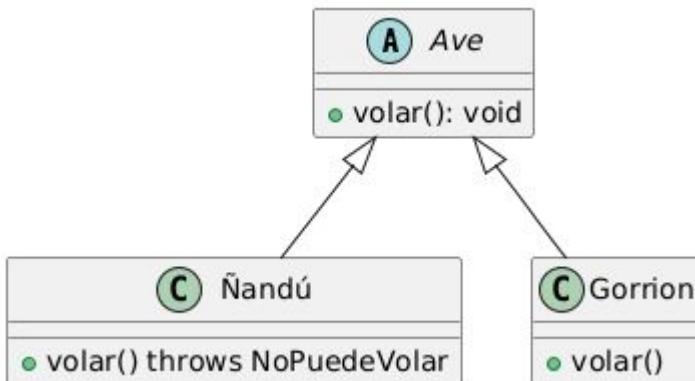
CONTRA-
EJEMPLO



En este ejemplo Cuadrado no redefine `setAlto` y `setAncho`, lo que rompería la definición de la forma geométrica.

Ave / vuelo

CONTRA-
EJEMPLO



La abstracción a Ave para incluir al Ñandú no es la correcta porque no es posible implementar el método.

En líneas generales

Es muy importante establecer y luego respetar la *idea general*¹ de la clase base.

¹{contratos, efectos secundarios e invariantes}

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?

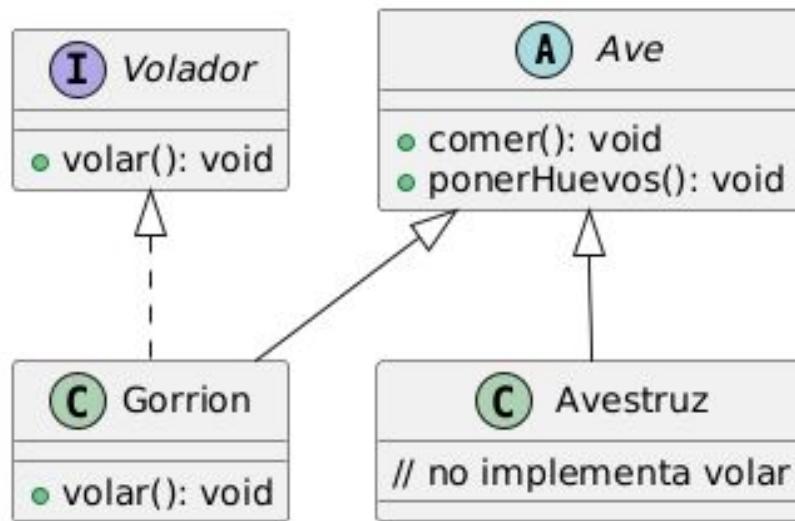


Principio de Segregación de Interfaces

Establece que los clientes no deben ser forzados a depender de interfaces que no usan.

**Iterable, Iterator y
Comparable son muy
buenos ejemplos**

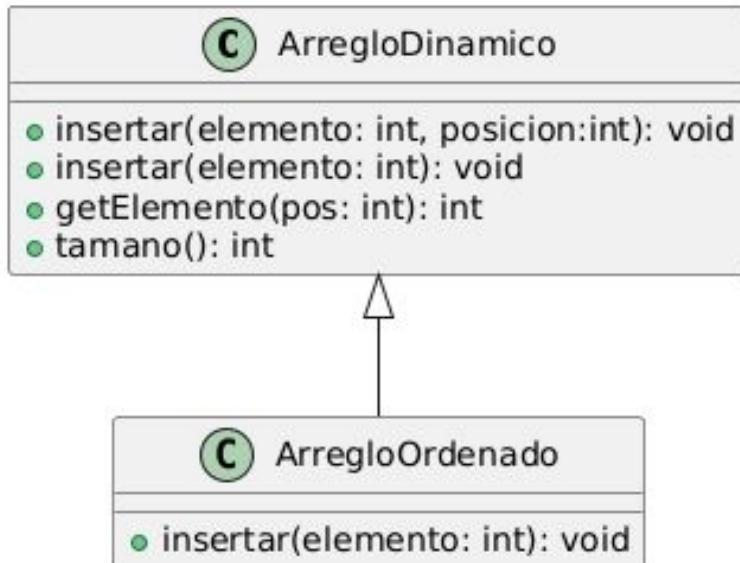
Ave / vuelo II



Si es necesaria una agrupación de Ave / Vuelo, una interfaz nos da la flexibilidad que podemos combinar con herencia.

Inserción Ordenada en ArregloDinámico

CONTRA-
EJEMPLO



Insertar sin posición en **ArregloDinamico** implica asumir una posición, este es un método que tiene más sentido en **ArregloOrdenado**

**Es muy fácil
caer en que todo
va a una interfaz
reducida**

En líneas generales

Dividir el comportamiento en interfaces específicas a una acción tanto como sea razonable.

Siguiendo la definición que vimos, con los -able , -ible y -tor

**Solo si hace
falta**

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?



Principio de Inversión de Dependencias

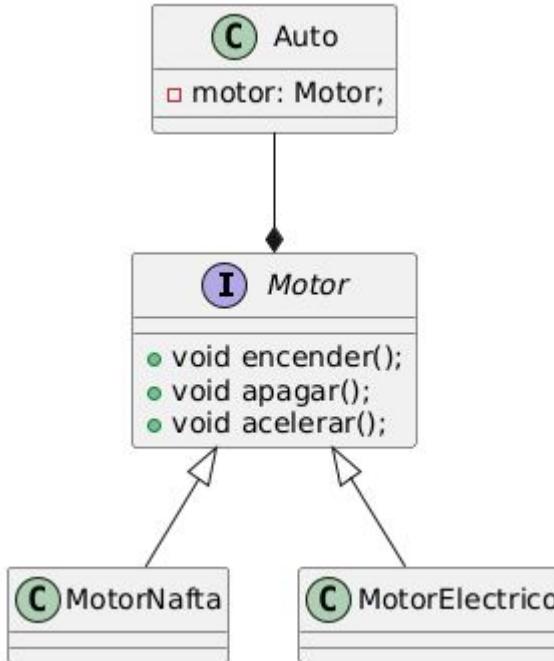
D

I

P

establece que las clases de alto nivel no deben depender de clases de bajo nivel, sino que ambos deben depender de abstracciones.

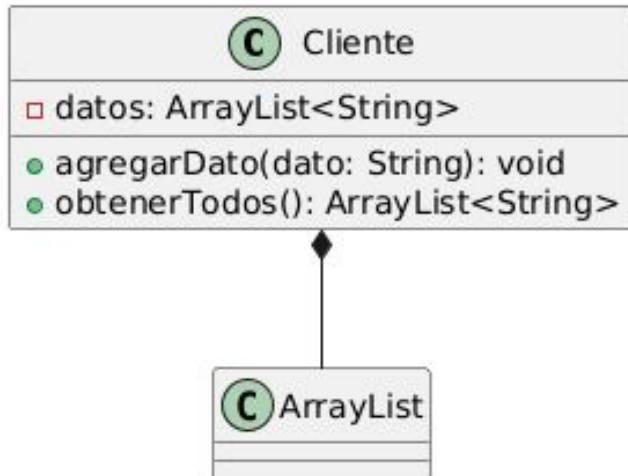
Auto / Motor



Esto nos permite cambiar rápidamente el tipo de **Motor** sin cambio alguno en el **Auto**

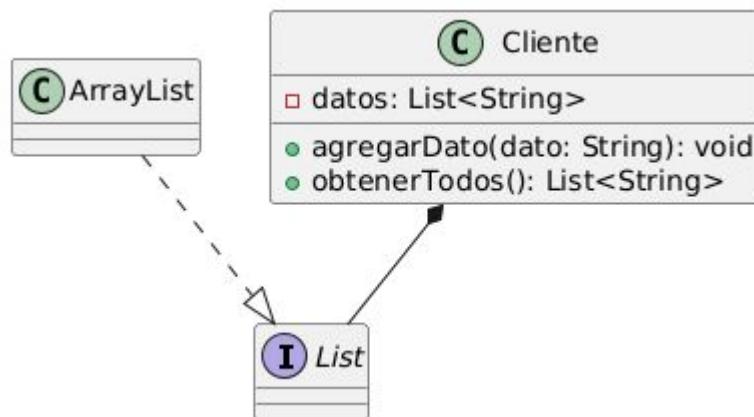
En Java collections

CONTRA-
EJEMPLO



Si necesitamos cambiar la implementación del conjunto, es necesario revisar que no estemos usando nada específico a la implementación elegida

Se indica el uso de las interfaces



Pueden existir excepciones, pero entendiendo que efectos secundarios tiene

En líneas generales

Usar las clases abstractas o interfaces más generales que sean posibles.



¿Preguntas?



Beneficios



Código más mantenable

Los cambios en una parte del sistema tienen menos probabilidades de afectar a otras partes.

Código más reutilizable

Las clases son más independientes y, por lo tanto, más fáciles de reutilizar en diferentes contextos.

Código más fácil de probar

Las clases con una sola responsabilidad son más fáciles de probar unitariamente.

Código más flexible

El sistema puede adaptarse más
fácilmente a los cambios en los requisitos.

**La «intensidad» del
olor es lo que
indica cuánto hay
que **SOLID**ificar**

En definitiva

Menos deuda técnica



¿Preguntas?

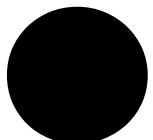


**Complete el
proyecto y
necesito
incorporar
cambios**



**Pero mi código
está en el**

VERAZ



Cómo podemos pagar la deuda técnica

Refactorizar es

el proceso de reestructurar el código existente sin cambiar su comportamiento externo.

● Los tests son
esenciales ●

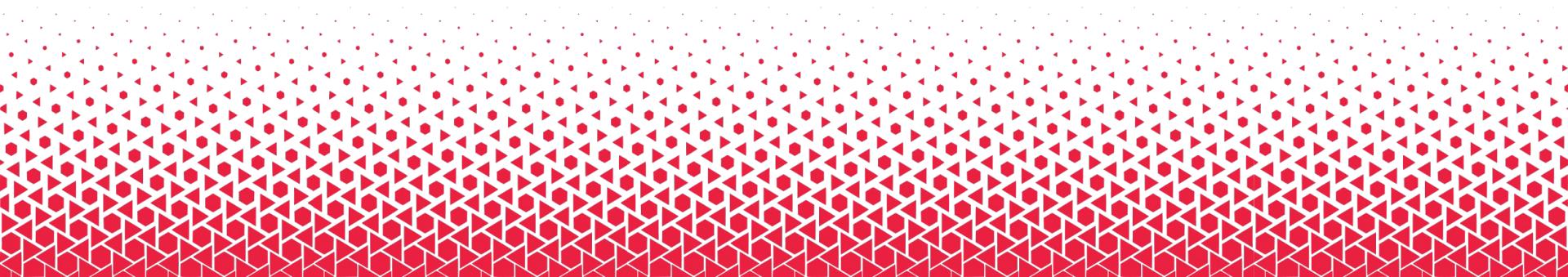
Ayudan a
garantizar que los
cambios no alteran
el funcionamiento

Continuará

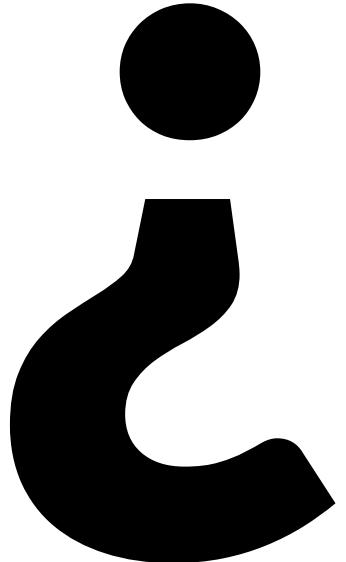
To Be Continued

más code
smells

God Class



Long Method



Dudas de los TP activos



A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?





**Abran
hilo**

Hasta la próxima



unrn.edu.ar

UNRN

Universidad Nacional
de Río Negro



| unrnionegro