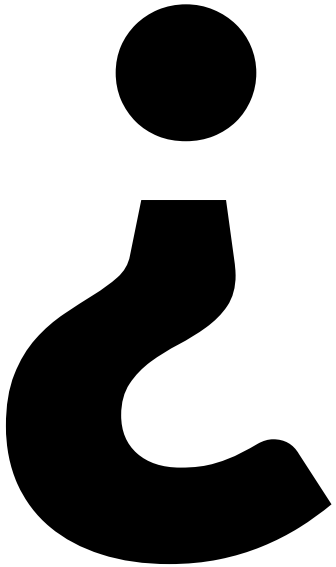
Principios SOLID

UNRN
Universidad Nacional
de Río Negro

IIXX
2024

r17



Dudas de los TP activos


¿Preguntas?

Abran hilo

que son los

¿Patrones de diseño?

1 Vocabulario

2

Soluciones comunes a problemas comunes


¿Preguntas?

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?

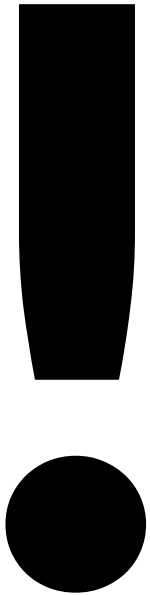
- Identificación y seguimiento
- Priorización
- Prevención*
- Refactorización*

Está más que claro que no hay deuda técnica en un programa que no se desarrolla continuamente

Aplica a

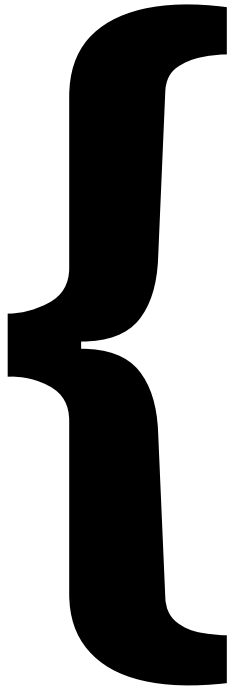
más que nada a

Software de gran escala

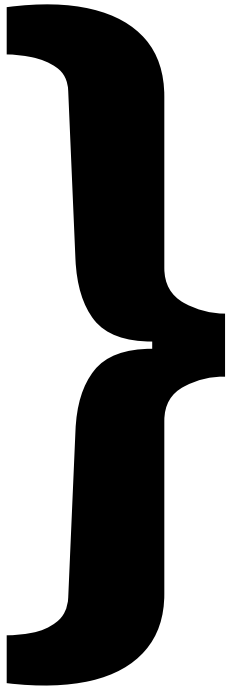


UNRN | Universidad Nacional de Río Negro

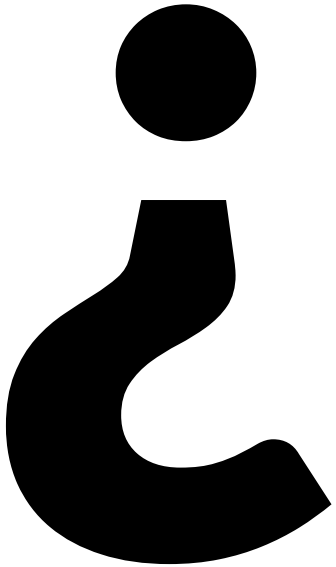

¿Preguntas?



Acoplamiento




¿Preguntas?



Cómo identificar deuda técnica

code/design

smells

Este diseño tiene un tufillo…

Rigidez

Rigidez

Un sistema donde una clase central maneja todas las funcionalidades (monolito). 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.

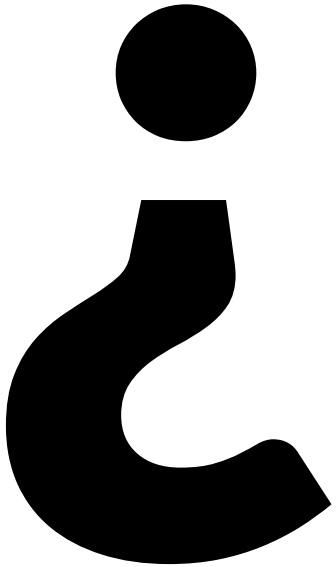
Viscosidad

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.

{ La "intensidad" del olor es lo que indica cuánto hay que SOLIDificar }


¿Preguntas?



Cómo podemos prevenir la deuda técnica

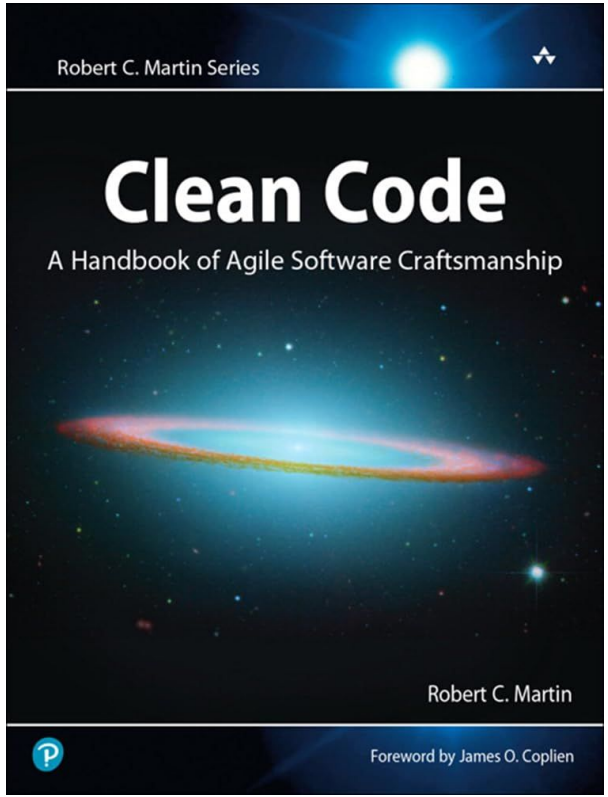
SOLID

Principios base para software orientado a objetos sólido

SOLID...
¿snake?

Qúe son Principios de diseño

Robert Martin ~2000

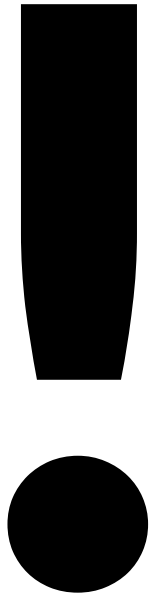


Martin, Robert C., editor. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2009.

SOLID

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

Estos lineamientos no son dogmas



Siempre podemos dejar deuda técnica también

Pero saber que hemos tomado un atajo es importante

Principle

SRP	Single Responsibility
OCP	Open/Closed
LSP	Liskov Substitution
ISP	Interface Segregation
DIP	Dependency Inversion

Principle

SRP	Responsabilidad única
OCP	Abierto/Cerrado
LSP	Substitución de Liskov
ISP	Segregación de interfaces
DIP	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
    }
}
```

¯_(ツ)_/¯

Solución posible

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


¿Preguntas?

Principio Abierto/Cerrado

O

C

P

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

Calculadora TP8

Si queremos agregar una nueva `Operacion`;

¿que tenemos que cambiar?

Dada esta clase para filtrar `Contacto's`

```
interface Filtro {
    boolean aplicar(Contacto producto);
}
```

Y diversas implementaciones

```
class FiltroColor implements Filtro {
    // ...
}

class FiltroTamaño implements Filtro {
    // ...
}
```

Podemos implementar un filtro con el criterio 'dinámico'

```
class ProductFilter {
    public List<Producto> filtrar(List<Producto> productos,
Filtro filtro) {
        // ...
    }
}
```

y si queremos agregar nuevas operaciones

```
public class OperacionBinaria{

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

¡Igual funciona!

Como contraejemplo

Solución posible

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


¿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 la corrección del programa.

*Barbara Liskov

Calcular en Operación

Podemos llamar
`calcular()`
a nivel de
`Operacion`

Si tenemos que

Hacer pattern matching en todas las llamadas, entonces no es reemplazable

O si el comportamiento no es homogéneo.

Otros contra-ejemplos

Solución

Es muy importante establecer y luego respetar el contrato de la clase base.


¿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

Contra-ejemplo

`ArregloOrdenado` como sub-clase de `ArregloDinámico`

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

Solo si hace falta


¿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.

Dada esta interfaz

```
interface Motor {
    void encender();
    void apagar();
    void acelerar();
}
// y un par de clases que la implementan
class MotorGasolina implements Motor { ... }
class MotorElectrico implements Motor { ... }
```

Es importante depender de la abstracción

```
class Coche {
    private Motor motor;

    public Coche(Motor motor) {
        this.motor = motor;
    }

    // ...
}
```

Es importante depender de la abstracción

```
class Coche {
    private MotorElectrico motor;

    public Coche(MotorElectrico motor) {
        this.motor = motor;
    }

    // ...
}
```

Posible solución

Buscar una abstracción que sea correcta para los casos en los que debemos especializar.


¿Preguntas?

Beneficios

Código más mantenible

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

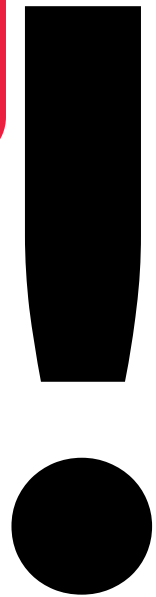
Las clases con una sola responsabilidad son más fáciles de probar unitariamente.

Código más flexible

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

En definitiva

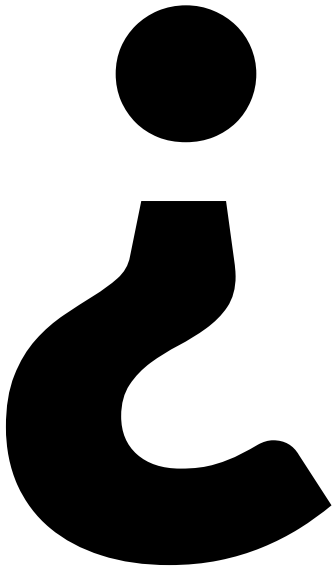
Menos deuda técnica




¿Preguntas?

Complete el proyecto y necesito incorporar cambios

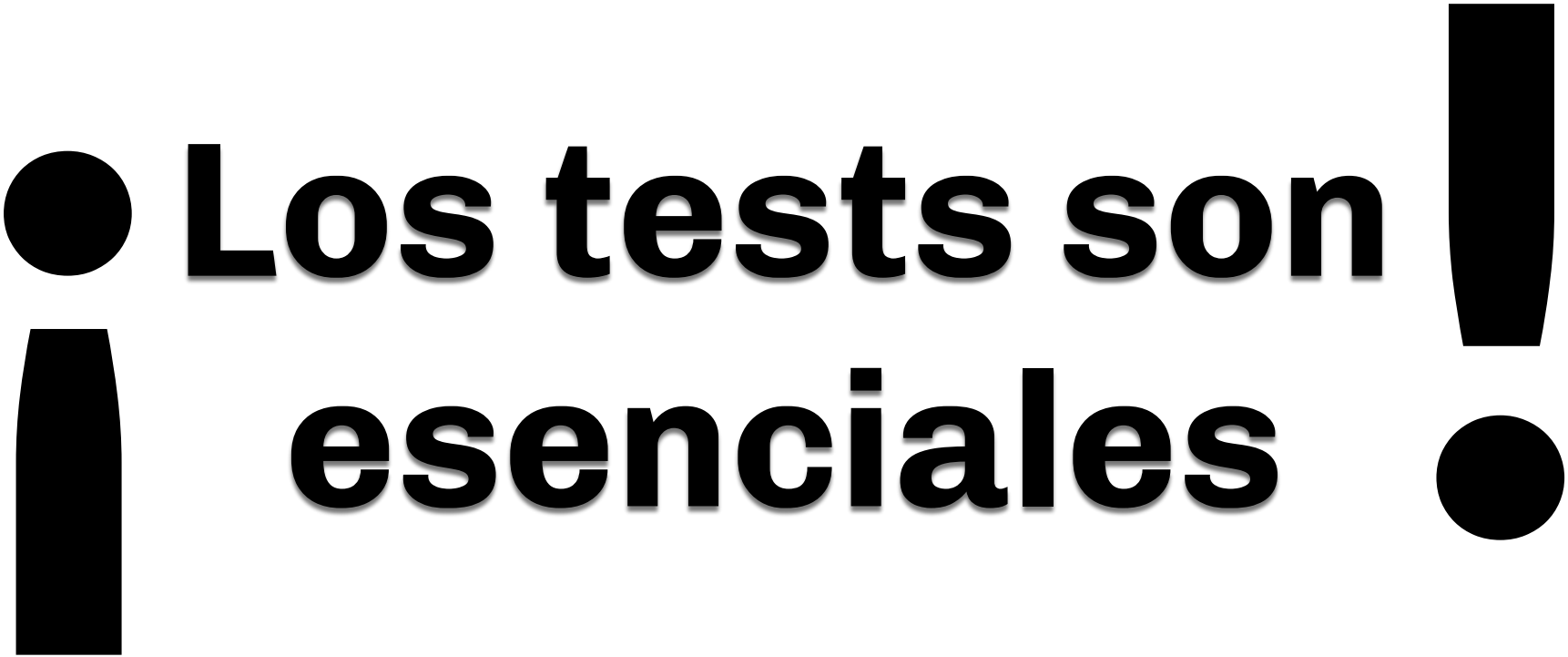
Pero mi código está en el



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

unrn.edu.ar

UNRN Universidad Nacional de Río Negro

| unrionegro