Clean Code

Programación Orientada a Objetos

APU San Salvador de Jujuy

Ing. José Zapana

Introducción

Cualquier código tiene que cumplir con 2 características:

1. Que funcione
2. Que sea mantenible

¿Qué parte es más fácil?
¿Qué parte es más importante?

Deuda técnica

- Como desarrolladores, la mayor parte del tiempo estamos leyendo y modificando código escrito por nosotros mismos o por alguien más
- Generalmente nos enfrentamos a un código fuente que ***NO entendemos en forma intuitiva***
- Estos factores afectan a la productividad
- Por otro lado, crear deuda técnica es “normal” y a menudo inevitable

Deuda técnica

Deuda técnica - costos económicos

- Tiempo en realizar mantenimiento
- Tiempo en refactorizar código
- Tiempo en comprender código
- Tiempo adicional en la transferencia del código

Esquema de deuda técnica Martin Fowler

Imprudente

No hay tiempo, sólo copia y pega esto de nuevo (hay que terminar como sea)

Prudente

Tenemos que entregar rápido, luego refactorizaremos

Imprudente Inadvertida

¿Qué son los patrones de diseño?

Prudente advertida

Ahora sabemos cómo deberíamos haberlo hecho

La mala calidad de software siempre la acaba pagando o asumiendo alguien

Reflexiones

- A nivel de proyecto son los programadores los que controlan el código
- ¿Estamos creando monstruos incontrolables?
- No es bueno acostumbrarse a generar deuda técnica porque NO es normal
- Sólo hay una sóla forma de hacer las cosas, hacerlas bien

¿Cómo se paga la deuda técnica?

Refactorización

- Es un proceso que tiene como objetivo mejorar el código sin alterar su comportamiento para que sea más entendible y tolerante a cambios
- Usualmente para lograr el objetivo esperado es imprescindible contar con pruebas unitarias

Recetas para evitar la deuda técnica

- Estudiando y aprendiendo de experiencias propias y ajenas
- Clean Code
- Pruebas Unitarias
- Herramientas para análisis de calidad de código
- ...

Clean Code

Grady Booch, autor de *Object Oriented Analysis and Design with Applications*

El código limpio es simple y directo. El código limpio se lee como un texto bien escrito. El código limpio no oculta la intención del diseñador sino que muestra nítidas abstracciones y líneas directas de control.

"Big" Dave Thomas, fundador de OTI, el padrino de la estrategia Eclipse

El código limpio se puede leer y mejorar por parte de un programador que no sea su autor original. Tiene pruebas de unidad y de aceptación. Tiene nombres con sentido. Ofrece una y no varias formas de hacer algo. Sus dependencias son mínimas, se definen de forma explícita y ofrece una API clara y mínima. El código debe ser culto en función del lenguaje, ya que no toda la información necesaria se puede expresar de forma clara en el código.

Principales características

- Nombres con sentido: Los nombres deben reflejar la intencionalidad. Variables, funciones, parámetros, etc.
- Comentarios claros y oportunos: un comentario debe aclarar el porqué del código y no lo que el código hace.
- Simplicidad: “Simple al nivel de tontos” (Code Simplicity: The fundamentals of software, 2012. Max Kanat-Alexander).
- Eliminar todo el código que no se usa: genera confusión, consume recursos, etc.
    - Ej. “este código es de una funcionalidad que se usaba antes y ahora se cambió...”
- Principio de Responsabilidad Unica (principios SOLID)
- ...

Principio de Responsabilidad única

- Una pieza de código sólo debe tener una razón por la que cambiar.
- Esto sucederá si un módulo sólo tiene una sola responsabilidad
  - Ver ejemplo al más adelante

Nombres con sentido

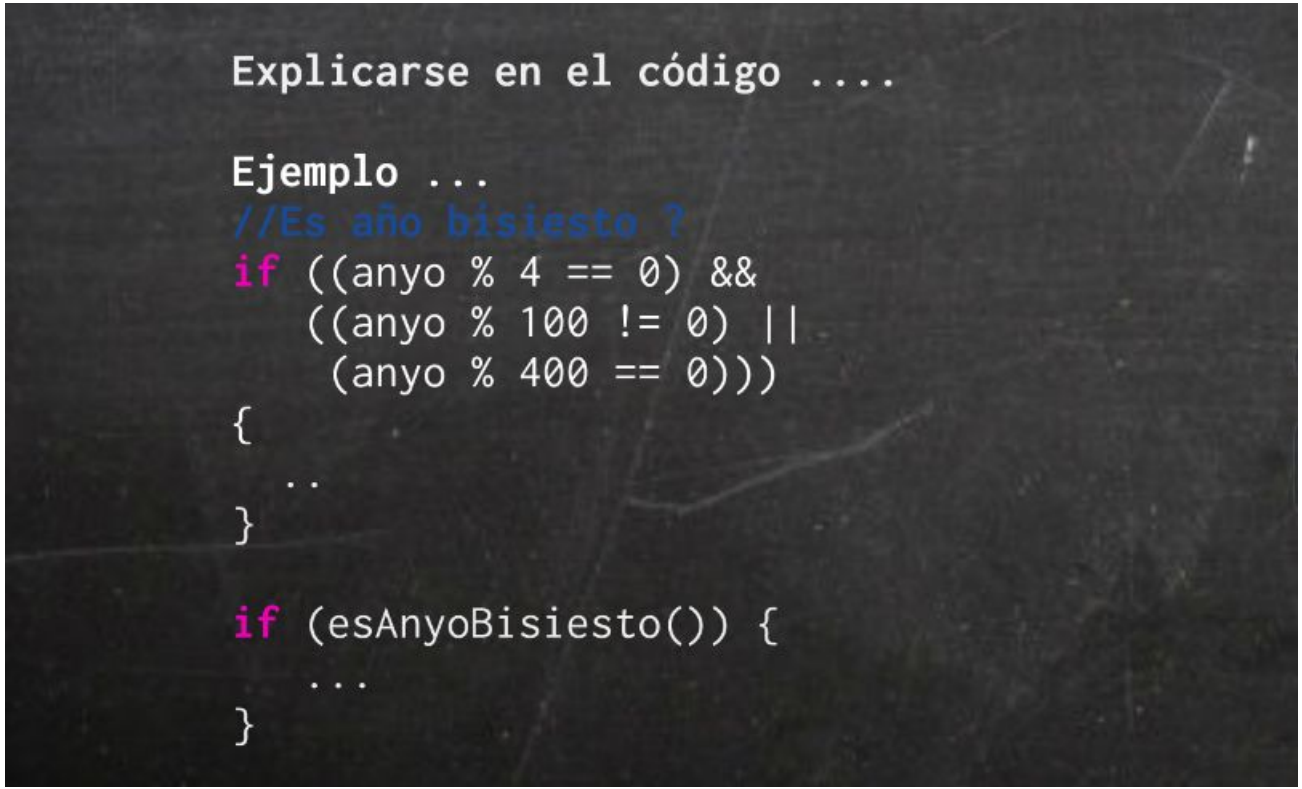
Utilizar nombres que tengan sentido y que revelen intenciones...

Por ejemplo:

```
Mal  : int d; //cantidad de días transcurridos
Bien : int cantidadDeDiasTranscurridos;
```

```
Mal  : public List getLista() // obtiene lista..
Bien : public List<Curso> obtenerListaDeAlumnos()
```

Claridad de código



Utilizar nombres pronunciables

Ejemplo :

```
class Televisor  {
   int chipXTCODTT; // Sensor de apagado
   int chipXTCODTR; // Sensor de encendido
}

class Televisor  {
   int sensorDeApagado;
   int sensorDeEncendido;
}
```

Ejemplo

- Calcular el salario de los empleados de una fábrica considerando que el mismo se calcula con la fórmula: salario = sueldo básico + adicionales - descuentos. El sueldo básico depende de la categoría del empleado, si la categoría es igual a 1 su valor es de 100000, si la categoría es igual a 2 su valor es 120000 y si la categoría es igual a 3 su valor es 150000. Además los adicionales son iguales para todos con un valor de 30000 y los descuentos son el 15 por ciento del sueldo básico

Solución 1

```
def calcularSalario(categoria):
    sueldoBasico = 0
    if categoria == 1:
        sueldoBasico = 100000
    elif categoria == 2:
        sueldoBasico = 120000
    elif categoria == 3:
        sueldoBasico = 150000

    adicionales = 30000
    descuentos = 0.15 * sueldoBasico

    salario = sueldoBasico + adicionales - descuentos
    return salario


salario_empleado = calcularSalario(2)
print(salario_empleado)
```

Solución Mejorada

```python
def calcularSueldoBasico(categoria):
    sueldoBasico = 0
    if categoria == 1:
        sueldoBasico = 100000
    elif categoria == 2:
        sueldoBasico = 120000
    elif categoria == 3:
        sueldoBasico = 150000
    return sueldoBasico
def calcularAdicionales():
    return 30000
def calcularDescuentos(sueldoBasico):
    return 0.15 * sueldoBasico
def calcularSalario(categoria):
    sueldoBasico = calcularSueldoBasico(categoria)
    return sueldoBasico + calcularAdicionales() - calcularDescuentos(sueldoBasico)

salario_empleado = calcularSalario(2)
print(salario_empleado)
```

El código limpio es simple y directo. El código limpio se lee como un texto bien escrito. El código limpio no oculta la intención del diseñador sino que muestra nítidas abstracciones y líneas directas de control.

Ejemplo de documentación con Java

```
/**
 * Returns the trigonometric sine of an angle.  Special cases:
 * <ul><li>If the argument is NaN or an infinity, then the
 * result is NaN.
 * <li>If the argument is zero, then the result is a zero with the
 * same sign as the argument.</ul>
 *
 * <p>The computed result must be within 1 ulp of the exact result.
 * Results must be semi-monotonic.
 *
 * @param   a   an angle, in radians.
 * @return  the sine of the argument.
 */
@IntrinsicCandidate
public static double sin(double a) {
    return StrictMath.sin(a); // default impl. delegates to StrictMath
}
```

Lecturas recomendadas

- Martin, Robert C. (2008). Clean Code
- Martin, Robert C. (2002). Agile Software Development: Principles, Patterns, and Practices
- Martin, Robert C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design.
-