Programaci#n Web 2 - UNLaM - 2020 2do Cuatrimestre

Conociendo Principios y Autores del Desarrollo de Software

Introducci#n a los Principios del Desarrollo de Software

Este documento tiene como prop#sito introducir conceptos y autores fundamentales en la industria del software, buscando **despertar la curiosidad** del estudiante y motivar una investigaci#n m#s profunda. Los principios presentados est#n **alineados con el Manifiesto #gil**, y muchos de ellos provienen de sus firmantes o figuras clave del desarrollo #gil.

Autores Referentes:

- **Kent Beck** (TDD)
- **Robert C. Martin** (SOLID, Clean Code)
- **Jeff Sutherland** (SCRUM)
- **Martin Fowler**

Estos autores, aunque diversos, se alinean en el mismo objetivo, haciendo referencias constantes a los valores del Manifiesto #gil:

- Individuos e interacciones sobre procesos y herramientas
- Software funcionando sobre documentaci#n extensiva
- Colaboraci#n con el cliente sobre negociaci#n contractual
- Respuesta ante el cambio sobre seguir un plan

CLEAN CODE (Robert C. Martin)

El libro "Clean Code" enfatiza la importancia de escribir c#digo **limpio y funcional**. Algunos principios destacados son:

- **Legibilidad ante todo:** El c#digo debe leerse de manera similar a un diario o libro.
- **Detectar "Code Smells":** Reconocer y abordar el c#digo de mala calidad (ilegible o poco escalable).
- **Software f#cil de cambiar:** La flexibilidad es clave; el software debe ser modificable.
- **Regla del Boy Scout:** Dejar el c#digo m#s limpio de lo que se encontr#.
- **Nombres representativos:** Variables y funciones deben tener nombres claros y significativos.
- **Funciones de una sola responsabilidad:** Cada funci#n debe hacer *una sola cosa y hacerla bien*.
- **M#nimo de par#metros:** Las funciones no deber#an recibir m#s de 2 par#metros (0 es ideal, 3 no es deseable).
- **Comentarios innecesarios:** El c#digo que no necesita comentarios es, por definici#n, buen c#digo.
- **Convenci#n de nombres:** Objetos como sustantivos y m#todos como verbos.
- **No temer al cambio:** La capacidad de refactorizar r#pidamente se logra con una buena bater#a de tests, idealmente escritos con **TDD**.

DRY: Don't Repeat Yourself (No te repitas)

Este principio busca **reducir la duplicaci#n** de cualquier "pieza de informaci#n", incluyendo:

- Datos almacenados en una base de datos.
- C#digo fuente de un programa de software.

- Informaci#n textual o documentaci#n.

La duplicaci#n incrementa la dificultad de cambios, perjudica la claridad y puede generar inconsistencias. La aplicaci#n eficiente de DRY asegura que los cambios en un proceso solo requieran modificaci#n en un #nico lugar.

KISS: Keep It Simple, Stupid! (Mantenlo Sencillo, Est#pido)

Establece que los sistemas funcionan mejor si se mantienen **simples**. La simplicidad debe ser un objetivo clave del dise#o, evitando cualquier complejidad innecesaria.

YAGNI: You Aren?t Gonna Need It (No vas a necesitarlo)

Consiste en **no a#adir funcionalidades innecesarias o no solicitadas**. La tentaci#n de escribir c#digo "por si acaso" en el futuro sacrifica tiempo que podr#a usarse en funcionalidades b#sicas y a#ade complejidad que debe ser depurada, documentada y soportada.

SLAP: Single Level of Abstraction Principle (Principio de Nivel #nico de Abstracci#n)

Propone dividir el programa en funciones o m#todos con:

- Una #nica responsabilidad.
- Pocas l#neas de c#digo.
- Un **#nico nivel de abstracci#n**.

Las funciones deben hacer una sola cosa y hacerla bien, utilizando nombres claros para identificar su responsabilidad, incluso si invocan otras funciones privadas.

CQS: Command and Query Separation (Separaci#n de Comandos y Consultas)

Ideado por Bertrand Meyer, este principio establece que **cada m#todo debe ser un comando o una consulta, pero no ambos**.

- Un **comando** realiza una acci#n y puede modificar el estado.
- Una **consulta** devuelve datos y no debe tener efectos colaterales (es decir, no debe modificar el estado). En otras palabras, hacer una pregunta no debe cambiar la respuesta.

TDD: Test-Driven Development (Desarrollo Guiado por Pruebas) (Kent Beck)

Es una pr#ctica de ingenier#a de software que combina tres pasos clave:

1. **Escribir la prueba primero:** Se escribe una prueba unitaria y se verifica que falle.
2. **Implementar el c#digo:** Se escribe el c#digo m#nimo necesario para que la prueba pase.
3. **Refactorizaci#n:** Se mejora el c#digo escrito, asegurando que todas las pruebas sigan pasando.

El objetivo de TDD es lograr un **c#digo limpio que funcione**, garantizando que el software cumple con los requisitos al traducirlos directamente en pruebas.

SOLID (Robert C. Martin)

SOLID es un acr#nimo que agrupa cinco principios fundamentales de la programaci#n orientada a objetos y el dise#o, introducidos por Robert C. Martin. Su aplicaci#n conjunta facilita la creaci#n de sistemas **f#ciles de mantener y extender** con el tiempo, eliminando malos dise#os y apoyando la refactorizaci#n continua en el desarrollo #gil y TDD.

1. **S - Single-responsibility Principle (Principio de Responsabilidad #nica):**

    - Cada m#dulo o clase debe tener una y solo una raz#n para cambiar; es decir, debe ser responsable de una #nica cosa.

2. **O - Open-closed Principle (Principio de Abierto/Cerrado):**

    - Las entidades de software (clases, m#dulos, funciones) deben estar **abiertas para su extensi#n**, pero **cerradas para su modificaci#n**. Esto significa que se debe poder extender su comportamiento sin alterar su c#digo fuente existente.

3. **L - Liskov Substitution Principle (Principio de Sustituci#n de Liskov):**

    - Los objetos de un programa deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del programa. Una clase heredada debe complementar, no reemplazar, el comportamiento de la clase base.

4. **I - Interface Segregation Principle (Principio de Segregaci#n de la Interfaz):**

    - Ning#n cliente debe verse obligado a depender de m#todos que no utiliza. Es preferible tener muchas interfaces peque#as y espec#ficas a una interfaz grande y monol#tica.

5. **D - Dependency Inversion Principle (Principio de Inversi#n de la Dependencia):**

    - Las dependencias deben recaer sobre **abstracciones (interfaces)**, no sobre clases concretas (implementaciones). Tanto los m#dulos de alto nivel como los de bajo nivel deben depender de abstracciones.