ITS SOLUTIONS

MEJORES PRACTICAS DE CÓDIGO LIMPIO

GERENCIA DE PROYECTOS & SERVICIOS

1 ALCANCE

Documentar las buenas prácticas que ITS Solutions SAS por medio de la Gerencia de Proyectos & Servicios, establece deben ser conocidas y puestas en práctica por los colaborades de la empresa cuyo rol este identificado como DESARROLLADOR DE SOFTWARE en cualquiera de los niveles Practicante, Junior, Semi Senior, Senior.

Se entiende por Buenas o Mejores Prácticas, como la recopilación de prácticas que ha sido demostrado funcionan de forma correcta y producen buenos resultados, por lo tanto, se recomienda seguirla como modelo.

2 DIRIGIDO A

Empleados directos, contratistas y practicantes cuyo rol y/o cargo dentro de la empresa esté relacionado con DESARROLLO DE SOFTWARE, independiente del tipo de proyecto al que sea asignado (Fabrica de Software o Staffing Pro), en cualquier caso, el colaborador debe asegurar y validar la lectura de este documento dentro de su proceso de Onboarding y Capacitaciones periódicas programadas por la empresa.

3 QUE SE ENTIENDE POR CÓDIGO LIMPIO

Código limpio, es un término bastante utilizado en el nicho de la ingeniería de software y particularmente en la rama de Desarrollo o Programación, y se trata básicamente de escribir el código de una manera sencilla que permita así su fácil lectura, entendimiento y mantenimiento.

Para conseguir estos tres propósitos (leer, entender y mantener fácilmente) dentro de ITS Solutions, se recomienda seguir estas 5 prácticas establecidas y conocidas en la industria de desarrollo de software, y adoptarlas como “Mejores Practica De Código Limpio”.

1. Legibilidad del código y código limpio
2. Comente con prudencia
3. Aplicar refactorización con frecuencia
4. Reducción de la deuda técnica
5. Seguir los principios de KISS, YAGNI, DRY y SOLID

3.1. LEGIBILIDAD DEL CÓDIGO Y CÓDIGO LIMPIO

La legibilidad del código a menudo se considera una de las características de mayor calidad en el software, La mala legibilidad del código genera problemas como errores y software menos estable, y es posible que se deba volver a escribir parte del software.

3.2. COMENTE CON PRUDENCIA

Aunque el código limpio debería explicarse por sí mismo, hay casos en los que los comentarios son necesarios para proporcionar contexto adicional o explicar algoritmos complejos. Sin embargo, se debe evitar los comentarios excesivos, ya que pueden saturar el código y dificultar su lectura.

El desarrollador debe centrarse en escribir código expresivo y utilizar los comentarios con moderación, sobre todo cuando explique el porqué, no el qué, no se trata de explicar todo el alcance por medio de comentarios.

3.3. APLICAR REFACTORIZACIÓN CON FRECUENCIA

La refactorización de software es el proceso de modificar o reestructurar el código existente para hacerlo más fácil de entender, más fácil de mantener y más fácil de cambiar.

La refactorización permitirá agregar flexibilidad al desarrollo para cumplir con requisitos cambiantes o adaptarse a futuros cambios de código, disminuyendo codificaciones repetitivas.

Esto implica:

- ✓ Comprender el código y los requisitos existentes.
- ✓ Reestructurar el código para un mantenimiento más simple.
- ✓ Realizar cambios en el existente para lograr la flexibilidad requerida.
- ✓ Validar el cambio contra los requisitos y, Finalmente, poner el cambio en producción.

3.4. REDUCCIÓN DE LA DEUDA TÉCNICA

La deuda técnica se utiliza en la industria del software para cubrir la faltantes (deuda) en un desarrollo derivados de:

- Errores
- Código heredado
- Documentación faltante.

Por lo general, se usa indistintamente con los faltantes (deuda) de diseño.

Cuando los equipos de desarrollo toman “medidas alternas” (atajos), para acelerar la entrega de una funcionalidad o proyecto, asumen una deuda técnica, que luego debe refactorizarse.

La deuda técnica es el resultado de priorizar la entrega rápida sobre el código perfecto, lo cual puede generarse por malas prácticas en la ejecución de la planeación.

Construir un mejor producto es considerar su código existente como una deuda. La refactorización y la reducción de la deuda permiten a los equipos de trabajo priorizar las necesidades actuales y asegurarse de que su deuda sea lo más pequeña posible.

Además, el código debe ser fácil de entender, mantenible, libre de errores y confiable, ya sea por tu grupo de desarrollo, por el propio del cliente o por un tercero desconocido para la empresa (se da en mayor cantidad en los casos de mantenimiento y soporte donde terceros han desarrollado el producto y no se conocen detalles al respecto).

3.5. SEGUIR LOS PRINCIPIOS DE KISS, YAGNI, DRY Y SOLID

Una característica muy común en la mayoría de los desarrolladores de software en la codificación es proponer soluciones complicadas a problemas sencillos.

- La mejor práctica es evaluar, preferiblemente en grupo, soluciones simples a problemas complejos.

Una solución avanzada no significa "sofisticación", sino aprovechar plenamente el poder del código, realizado mediante técnicas de uso de bases de datos, uso de recursos de red y uso de recursos de hardware una optimización del software en ejecución, evitando el “overhead administrativo” del código con métodos prácticos de ingeniería, sin trivializar la solución, ni tampoco diseñar un laberinto para llegar al objetivo.

Si estas buenas prácticas no son consideradas en los workshops técnicos, es muy altamente probable que las consecuencias salgan a la luz en las fases de pruebas de calidad, aseguramiento y/o aceptación, presentado ahí los “overhead administrativos”.

3.5.1. KISS ¡Mantenlo simple!:

A menudo, los desarrolladores hacen las cosas más complejas de lo necesario.

3.5.2. YAGNI No lo vas a necesitar:

Los desarrolladores agregan funcionalidades para satisfacer los requisitos futuros y, por lo general, esto no es una buena idea.

3.5.3. DRY No te repitas a ti mismo:

Nos recuerda que el código duplicado genera más código, y más código es más difícil y costoso de mantener.

3.5.4. SOLID:

es un acrónimo acuñado por Michael Feathers basado en los principios de la programación orientada a objetos.

S: Single Responsibility Principle **(SRP)**.

Principio de responsabilidad única

O: Open/Closed Principle **(OCP)**.

Principio Abierto/Cerrado

L: Liskov Substitution Principle **(LSP)**.

Principio de Sustitución de Liskov

I: Interface Segregation Principle **(ISP)**.

Principio de Segregación de Interfaz

D: Dependency Inversion Principle **(DIP)**

Principio de Inversión de Dependencia

Principio de sustitución de Liskov o LSP es un principio de la programación orientada a objetos. y puede definirse como: Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ella

4 GESTIÓN DE CAMBIOS Y DISTRIBUCIÓN

4.1. CONTROL DE COPIAS

Copia No Controlada: Esta versión del documento ha sido descargada o impresa y no está sujeta al proceso de control de documentos del S.I.G. Se recomienda revisar regularmente el repositorio de documentos del S.I.G para asegurarse de utilizar la versión más actualizada.

4.2. REGISTRO DE CAMBIOS

Versión	Causa del Cambio	Responsable del Cambio	Aprobó	Fecha del Cambio
1	Creación del documento	Gustavo Sanabria Director de Proyectos y Servicios	Freddy Rojas Gerente de Proyectos y Servicios	16/04/2017

4.3. CONTROL DE DISTRIBUCIÓN

Nombre y Apellidos/Sitios
Jonathan Prieto- Analista Calidad y procesos
Intranet de la empresa