Universidad de Las Américas

Facultad de Ingenierías y Ciencias Agropecuarias

Ingeniería de Software

Informe de laboratorio

1. **DATOS DEL ALUMNO:** Camila Cabrera

2. **TEMA DE LA LECTURA:**

La lectura como tema general se enfoca en la claridad del código. En el capitulo I se enfoco en definir que se entenderá en el libro como un código claro y porque en ocasiones se obtiene código difícil de mantener. Brinda perspectivas de varios autores y los pilares a destacar de cada uno. En el capitulo II se enfoca en 19 reglas para poder nombrar correctamente a las entidades en un sistema, bridando las mejores practicas y ejemplos de uso.

3. **FUENTE:**

Libro: Código Limpio: Manual de estilo para el desarrollo ágil de software (Clean Code: A Handbook of Agile Software Craftsmanship") de Robert C. Martin.

Bibliografía:
Cecil, R. (2018). Código Limpio Manual de estilo para el desarrollo ágil de softwre. epublibre.

4. **RESUMEN:**

CAPITULO I: "CÓDIGO LIMPIO"

El código no podrá desaparecer jamás porque es la representación de los requisitos del sistema. Es decir, el código es el ejecutable de la recopilación de requisitos. Inclusive a pesar del nivel de abstracción de los lenguajes de programación, siempre es importante que el código sea preciso y formal para que el equipo pueda comprenderlo.

Código incorrecto

El código correcto es importante porque durante mucho tiempo se ha sufrido la ausencia de este. Grandes empresas con productos estrella han fracasado debido a grandes fallos en el código que han superado sus capacidades. Dado esto, en ocasiones un programador debe sortear el código incorrecto mientras revisa un sistema. Alguna de las razones por las que se produce esto es: el apuro de entrega, el cansancio, la presión de un superior, etc. Estas razones orilla a los programadores a concluir que un programa que funciona es mejor que nada, y después podrán arreglarlo. Desconociendo la premisa de LeBlac: Después es igual a nunca.

El coste de un desastre

Un equipo que empieza muy rápido en el desarrollo probablemente acabe muy despacio. Cada vez que hay un cambio o modificación si no se conoce que otra parte del proyecto se afecta se crea un desastre. Esto baja la productividad y caen en la tentación de añadir más personas al desarrollo lo que solo genera más caos, porque los nuevos integrantes desconoces de la importancia de cada módulo y de cómo afectara un cambio.

El gran cambio de diseño

Cuando un código se vuelve casi insostenible, se busca la reingeniería de este. El equipo encargado de su rediseño asume el reto de crear nuevas funcionalidades y que sea mantenible, sin embargo, en ocasiones toma demasiado tiempo, y al final vuelve a resultar un código no muy mantenible. Repitiéndose la historia.

Actitud

Un programador profesional, debe ser sincero respecto a los tiempos y los objetivos del sistema. Debe defender el código con la misma intensidad que los demás involucrados en sus respectivas áreas. No se debe ceder siempre a la voluntad de los jefes que no entienden el riego de un posible desastre en el código.

El enigma

“No se cumple un plazo de entrega cometiendo errores” (Cecil, 2018), es necesario validar que el código sea limpio.

¿El arte del código limpio?

“La única forma de avanzar es mantener el código limpio” (Cecil, 2018), pero reconocer esto no significa que sepamos como crearlo. Para esto se requiere el uso disciplinado de miles de técnicas, aplicando esto se transforma código incorrecto en código correcto.

Concepto de código limpio

Hay definiciones como programadores. (6 autores explorados)

Tabla#1: Visión de autores sobre código limpio

Autor	Visión del autor	Análisis
Bajarne Stroustruo, inventor de C++ y autor de The c++ Proramming Language	• Elegante y eficaz • Lógica directa • Dependencias mínimas • Procesamiento de errores completo • Hace una cosa bien	• Código elegante = fácil de leerlo. • El código incorrecto tiende a aumentar el desastre. • El procesamiento de errores = prestar atención a los detalles • Código limpio= atención al detalle • El código limpio es concreto, cada cosa tiene un fin no se mezclan funciones.
Grady Booch, autor de bject Oriented Analysis and Desing with Aplications	• El Código limpio es simple y directo. • Se lee como un texto bien escrito. • No oculta la intención del diseñador. • Nítidas abstracciones y líneas directas de control	• Perspectiva de legibilidad. • Mostrar claramente el problema a resolver • Especifico y no especulativo • Solo incluir lo necesario
“Big” Dave Thomas, fundador de OTI, el padrino de Eclipse	• El código limpio se lee y mejora por un programador que no es su autor. • Pruebas de unidad y de aceptación • Nombres con sentido • Una y no varias formas de hacer algo • Dependencias mínimas • Forma explícita, API clara y mínima • Culto en función de lenguaje	• Facilita la labor de mejora de otros. • Diferencia entre fácil de leer y fácil de cambiar • Limpieza de código relacionado con pruebas. • Código sin pruebas no es limpio. • Código de tamaño reducido, cuanto más pequeño mejor. • Legible para los humanos
Michael Feathers, autor de Working effectively with Legacy Code	• El Código limpio parece escrito por alguien a quien le importa • Nada evidente que hacer para mejorarlo • Pensó en todos los aspectos posibles • Disfrutar el código	• Dar importancia • Dedicado su tiempo para que sea sencillo y prestó atención a los detalles

Ron Jeffreies, autor de Extreame Programming Installed y Extreme Programming Adventires in C#	Reglas del código simple de Beck, por prioridad • Ejecutar todas las pruebas • No contiene duplicados • Expresa todos los conceptos de diseño del sistema • Minimiza el número de entidades (clases, métodos, funciones y similares)	• Duplicación= idea que no se acabó de representar correctamente. • Incluir nombres con sentido, dispuesto a cambiarlos varias veces. • Si un objeto o método hace más de una cosa se debe segmentar. • Abstracción de métodos
Ward Cunningham, inventor de Wiki, FIT, inventor de programación eXtreme. Patrones de diseño, pogrmacion orientada a objetos.	• Cada rutina que leemos resulta sr lo que esperábamos. • Parece que el leguaje se creo para el problema en cuestión.	• Leer código limpio no lo hace llevarse sorpresas, no hay esfuerzo en leerlo. • Evidente, sencillo y atractivo • Simplificarlo todo • Responsabilidad que el lenguaje parezca sencillo

Escuela de pensamientos

Ninguna escuela de pensamiento tiene la razón absoluta, se actúa como si las enseñanzas y las técnicas fuera correctas. Una escuela no anula las enseñanzas de otra diferente se complementa y aprende.

Somos autores

Somos autores y escribimos para lectores que juzgan nuestro esfuerzo. No se puede escribir código sin leerlo. El código que se escribe hoy será fácil en medida que el código circundante sea fácil de leer. Si es fácil de escribir tiene que ser fácil de leer.

Regla del Boy Scout

Limpiar el código con el tiempo. “Dejar el campamento más limpio de lo que se ha encontrado”, entregar el código más limpio de lo que se recibió. Nombres correctos, simplificación de clases y métodos, simplificación de una instrucción if.

Precuela y principios

Principio SRP = principio de la responsabilidad única , OCP = principio abierto/cerrado, DIP= Principio de inversión de dependencias.

CAPITULO II: "NOMBRES CON SENTIDOS"

Reglas para crear nombres correctos. A continuación, se mencionas las reglas identificadas, cada una se explora por individual más adelante.

Tabla#2: Listado de reglas para elaboración de nombres

Número de Regla	Nombre
1	Usar nombres que revelen las intenciones
2	Evitar la desinformación
3	Realizar distinciones con sentido
4	Usar nombres que se puedan pronunciar
5	Usar nombres que se puedan buscar
6	Evitar codificaciones
7	Notación húngara
8	Prefijos de miembros
9	Interfaces e Implementaciones
10	Evitar asignaciones mentales
11	Nombre de clases
12	Nombre de métodos
13	No se exceda con el atractivo
14	Una palabra por concepto
15	No haga juegos de palabras
16	Usar nombres de dominios de soluciones
17	Usar nombres de dominios de problemas
18	Añadir contexto con sentido
19	No añadir contextos innecesarios

1. *Usar nombres que revelen las intenciones*

Es importante elegir un nombre correcto, no esta mal cambiar el nombre en un futuro siempre que se identifique una mejora. Para elegir el nombre, este debe responder a 3 cuestiones básicas:

- ¿Por qué existe?
- ¿Qué hace?

- ¿Cómo se usa?

Si requiere un comentario, no es un buen nombre.

2. *Evitar la desinformación*

Evitar poner nombres que su significado estén lejos de lo que se busca representar. Por ejemplo si se coloca UsuarioLista, lo mas probable es que se asuma que es un tipo de dato <List> , en caso de no serlo generaría malos entendidos, sería mejor colocar grupoUsuario o simplemente Usuarios.

Evitar colocar nombres muy similares como controladorfortalezaspersonaje y controladorfortalecimientopersonaje, toma mucho tiempo ver la diferencia y podrían ser confundidos.
Evitar colocar nombres como la vocal O que se puede confundir con cero 0 o la letra l que puede ser confundida con 1.

3. *Realizar distinciones con sentido*

Los programadores en ocasiones se encuentran frente a escenarios donde es tentado a usar el mismo nombre para dos objetos. Esta claro que esto no podría ser en el mismo ámbito entonces decide cambiar arbritaria mente uno de los nombres aumentando números o palabras adicionales. Pero si existe esta necesidad se debería detener a pensar cual es el significado que se le va a dar., y nombrarlo apropiadamente.

Poner a, b, c ni siquiera desinforma porque no está informando en un principio. Por ende se debe diferenciar los nombres de forma que se aprecien las diferencias y el lector comprenda su uso.

4. *Nombres que se puedan pronunciar*

Poner nombres que se puedan pronunciar para que sea más fácil explicarlo y comprenderlo es más fácil leer el código y entender las variables que verse en la necesidad de explicar a un nuevo integrante del equipo a lo que se esta haciendo referencia.

5. *Usar nombres que se puedan buscar*

Los nombres de una sola letra y las constantes son difíciles de encontrar en el código. Es más fácil encontrar num_max_alum_clas que el número 7. Es más difícil encontrar la variable e a una variable con nombre.

6. *Evitar codificaciones*

Evitar codificaciones es una carga adicional para un nuevo miembro del equipo tener que aprenderse las codificaciones a parte de la lógica del código.

7. *Evitar codificaciones*

En los lenguajes actuales y los compiladores es más fácil saber el tipo de dato y que se nos recuerde durante la codificación, por lo cual no es necesario mantener notificaciones como a notación húngara.

8. *Prefijos de miembros*

No es necesario añadir prefijos, los usuarios aprenden a ignorar el prefijo y fijarse en la parte descriptiva del nombre.

9. *Interfaces e Implementaciones*

No es necesario colocar en el caso de interfaces la letra I antes de su nombre. Puede ser un exceso de información.

10. *Evitar asignaciones mentales*

No es bueno poner nombres que los lectores deban traducir. Es común usar nombres de una sola letra en bucles, pero en otro contexto es una opción limitada y pobre.

11. *Nombres de clases*

Los nombres de clase no deben ser un verbo

12. *Nombres de métodos*

Los nombres de métodos beben ser un verbo. En caso de ser métodos de acceso o modificación deberían servir el estañar **javabean** get, set, is. En el caso de sobrecarga de constructores se debe colocar un nombre que explique sus parámetros.

13. *No se exceda con el atractivo*

Los nombres deben ser claros, no usar coloquialismos.

14. *Una palabra por concepto*

Elegir una palabra por cada concepto. Poner nombres sugerentes que puedan ser fácil de identificar en otros módulos.

15. *No hacer juegos de palabras*

No usar la misma palabra para finalidades distintas.

16. *Usar nombres de dominios o soluciones*

Usar nombres técnicos, no conviene explicar a un colega algo que podría haber entendido si se usaba el nombre común.

17. *Usar nombres de dominios de problemas*

Cuando no exista un nombre técnico hay que usar un nombre del dominio del problema.

18. *Añadir contexto con sentido*

Usar prefijos como ultimo recurso, es mejor usar una clase.

19. *Añadir contexto con sentido*

Los nombres cortos son mejores que los extensos siempre que sean claros.

20. OPINIÓN PERSONAL:

- En base a la lectura y específicamente a la sección “Actitud” y “Regla del Boy Scout” del capítulo I, puedo decir que concuerdo con la perspectiva del autor acerca de la importancia de la actitud en un programador profesional. Este menciona que el programador debe defender su postura ante las demás partes (jefes, comerciales, usuarios) en un proyecto, porque es su deber defender el código y no dejarse poner tiempos de entrega muy justos que afecten la claridad del código. Es decir, un programador es crítico y objetivo con su entorno social. Además, indica que como los Boy Scouts deberíamos dejar el código más limpio de cómo nos lo entregan. Por ende, un programador también es crítico y proactivo en cuanto a su entorno de desarrollo. Yo podría acotar dentro de esta línea una noción que, aunque suene un poco grotesca me parece pertinente, un programador además no podría ser ocioso o conformista. Al momento de escribir código y de diseñar un sistema es pertinente detenerse un segundo y replantearse si es la manera mas clara de realizarlo y en caso de que no lo sea deber se ágil y cambiar lo que se considere. Una pronta refactorización es mejor que arrastrar un problema o código ineficaz a todo el desarrollo del sistema. Es decir, un código limpio es resultado de un programador proactivo, critico, objetivo y ágil que vence la angustia de modificar su código y no es conformista. Todo esto con el objetivo de facilitarse la codificación a futuro y la eficiencia del proyecto.

- En el capítulo I también me parece muy interesante como todos los autores hablan sobre la importancia de la legibilidad del código y lo sencillo que debería ser el poder entender que se pretende hacer en cada módulo del sistema. En mi corto tiempo programando, puedo decir que este a mi parecer es la característica más fuerte de la claridad del código. Mis experiencias con un equipo de desarrolladores se remiten a los trabajos finales, sin embargo, me he dado cuenta de que es fundamental para su éxito que todos los miembros del equipo podamos entender el código realizado por cualquiera de nosotros. En ocasiones dada la poca experiencia no aplicamos principios de claridad del código, esto retrasa su desarrollo y aumenta la tensión del equipo. Por ende, puedo decir que programar no solo es codificar como se pueda si

no, es codificar pensando en quienes leerán y darán mantenimiento a mi código. No programamos para nosotros si no para los que vendrán.

- Respecto al capítulo II me pareció muy pertinente los consejos brindados en la nomenclatura, generalmente recibo criticas de mis compañeros por lo poco comprensibles que son mis nombres. Ahora entiendo que mi falla radica en el consejo número 4 “Usar nombres que se puedan pronunciar” en mi caso suelo caer en el error de acortar el nombre de las variables. Por ejemplo, si mi variable hace referencia a ingresos anuales del empleado, suelo colocar ingempanu, lo que obviamente no resulta claro ni pronunciable. Sin embargo, creo también que es necesario estandarizar y definir como se van a colocar los nombres compuestos en el quipo, por ejemplo, se podría acordar que se usaran las 5 primeras letras divididas por guiones bajo, el nombre de la variable quedaría así ingre_anual_emple. Es importante estandarizar y coordinar como se manejará la nomenclatura en el equipo.

- Dentro de todos los consejos brindados me pareció muy limitante el número 8 acerca de prefijos de miembros. Según el texto sería poco eficiente colocar prefijos que hagan alusión a un grupo de datos, porque los programadores los ignoran. Sin embargo, en ocasiones y dependiendo de como se haya decidido manejar el estándar en el equipo me parecen realmente útiles. Por ejemplo, si tenemos una biblioteca de clases llamada “útil”, donde existe una clase que valida los campos de ingreso (números, letras, caracteres, espaciales, correos) y esta es llamada desde el proyecto principal me parecería pertinente que el nombre de los métodos tenga el prefijo utl, solo para poder indicar en donde se encuentra este código y que el panorama del programador sea más claro.

21. TRES PREGUNTAS Y RESPUESTA SOBRE LA LECTURA:

a. **PREGUNTA 1:** ¿Cuáles son los prefijos estándar para los métodos de acceso o modificación de un atributo en una clase?
Respuesta: Se usan como prefijos get (acceso) y set (modificación)

b. **PREGUNTA 2:** ¿Por qué es importante usar nombres que se puedan pronunciar?
Respuesta: Es más fácil explicarlo y comprenderlo, facilita la lectura del código y entender las variables.

c. **PREGUNTA 3:** ¿Cuál es el problema de poner nombres codificados?

Respuesta: Es una carga adicional para un nuevo miembro del equipo tener que aprenderse las codificaciones a parte de la lógica del código o es complicado tener en mente siempre la codificación de los nombres.