Universidad de las Ciencias Informáticas

Facultad 3

Título: Propuesta de una Guía para estandarizar la codificación en la Universidad de las Ciencias Informáticas.

Trabajo de Diploma para optar por el título de
Ingeniero en ciencias Informáticas

Autor(es): *Yaquelin Y. Morales Rodríguez*
Adisneisy C. Román Remedio

Tutor(es): *Michael González Jorrín*
Yosvany Márquez Ruiz

Junio de 2007

DECLARACIÓN DE AUTORÍA

Declaramos que somos las únicas autoras de este trabajo y autorizamos a la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

Autoras:

Yaquelin Yaimee Morales Rodríguez

_____

Adisneisy Caridad Román Remedio

_____

Tutores:

Michael González Jorrín

_____

Yosvanys Márquez Ruiz.

_____

Adisneisy

Agradezco:

A Yaquelin por haber sido mi compañera de tesis y Amiga.

A Michael González por tomar la difícil tarea de ser nuestro tutor y por la paciencia que nos dedicó durante el desarrollo del trabajo.

A mi mamá por todo su amor, comprensión y esfuerzo dedicado en el transcurso de estos años.

A mi papá y a mi hermano por todo su cariño y apoyo.

A mis amigas más fieles Nimia, Eleinne, Ana y Yaklin.

Agradezco a Yanet García por su ayuda en los momentos difíciles de la carrera.

A Yosvany por toda la ayuda brindada.

A mis compañeros de grupo y profesores por todos los momentos compartidos.

A mis tías y tíos, primos y primas, y a mis abuelas.

A Nery, a Pepin y la abuela Momia.

A Lantecita por aprender a quererme.

A ti Maikel por ser la persona más especial que he conocido y por convertirte en el amor de mi vida.

Agradezco además la oportunidad de haber estudiado en la UCI y a todo el que de una forma u otra aportó su granito de arena en este trabajo.

Yakelin

Agradezco:

A Adis, por ser mi amiga y compañera de tesis.

A mis padres, y a mi hermano que siempre me han dado la confianza y el amor que he necesitado.

A mi gran amiga Gisellys por dejarme contar con su amistad, tan valiosa para mí.

A Jorge, mi novio, por ser comprensivo y hacerme vivir tan buenos momentos.

A mis amigas Yoanna, Yanara y Teresa por permitirme ser parte de sus vidas.

A mis amigos Alex y Anner por quererme como soy, con mis virtudes y defectos.

A mi tutor Michael, que más que mi tutor siempre fue un gran amigo para mí.

A mi tutor Yosvany por ser mi amigo, y ayudarme siempre que lo he necesitado.

A mi suegra Gloria, por quererme y hacerme sentir como una hija para ella.

A mi prima Yami, por quererme y ayudarme durante estos 5 cursos de la carrera.

A mi tía Jaque, por ser mi tía más querida, y preocuparse por mí bienestar.

A mi abuelita Cuca, que tanto quiero y me quiere.

A todas las personas que han colaborado a forjar en mí una mejor persona.

Adisneisy:

A la memoria de mi tío Papito.

A mi papá por todo su cariño.

A ti Migdalia por ser la mejor madre del mundo

A ti Maikel por ser tan especial y por convertirte en mi Tatico.

Yakys:

A mis padres que son la luz de mi vida.

A mi amiga del alma, Gisellys.

A mi vida, Jorge.

"No creas en los consejos". Vete a vivir la vida que te enseñará todo lo que necesitas saber.

Paulo Coelho

Resumen

La Industria Cubana del Software (ICSW) está llamada a convertirse en una significativa fuente de ingresos para la economía nacional. La Universidad de las Ciencias Informáticas y el sistema de empresas cubanas vinculadas a este trabajo jugarán un papel importante en este sentido, de ahí la importancia de desarrollar software y que este sea de alta calidad. El presente trabajo, titulado: Propuesta de una Guía que ayude a estandarizar la codificación en la Universidad de las Ciencias Informáticas (UCI), hace un estudio de las buenas prácticas que contribuyen a mejorar la codificación y la calidad del código fuente de los programas para asegurar la calidad del software. Los objetivos que se persiguen son: Incrementar el conocimiento sobre las buenas prácticas destinadas a este fin y conocer su estado en la universidad para poder alcanzar el objetivo principal, que consiste en proponer una guía que contribuya a estandarizar la codificación en la UCI. Como resultado se obtuvo una guía con los elementos comunes de los estándares de codificación que permite mediante dos listas de chequeo, evaluar los diferentes estándares por lenguaje para saber cual es más óptimo y la calidad del código fuente de los programas.

INDICE

Introducción..... 11

Capítulo 1 Fundamentación teórica 20

1.1 Introducción..... 20

1.2 Calidad del código. Calidad de Software..... 20

1.3 Buenas Prácticas para el mejoramiento de la codificación 21

1.4 Estado de la aplicación de las buenas prácticas en la UC. 22

1.5 Refactorización. 23

1.5.1 Clasificaciones de la refactorización..... *24*

1.5.2 ¿Qué es refactorizar? *24*

1.5.3 Bases Generales de un Proceso de Refactorización *25*

1.5.4 ¿Cuando se debe refactorizar? *26*

1.5.5 ¿Cuando no se debe refactorizar? *27*

1.5.6 ¿Cómo se relaciona la refactorización con otras prácticas de programación? *28*

1.5.6.1 Relación Refactorización- Pruebas Unitarias 28

1.5.6.2 Relación Refactorización- Patrones de diseño..... 29

1.5.7 Ventajas de la refactorización..... *29*

1.5.8 Desventajas de la refactorización..... *30*

1.5.8.1 Problemas de la Refactorización 31

1.5.9 Pruebas de Software 31

1.5.9.1 Pruebas Unitarias 32

1.5.9.2 Características de las pruebas unitarias..... 33

1.5.10 Pruebas de caja blanca *34*

1.5.11 Clasificación de las Pruebas de Caja blanca..... *35*

1.5.11.1 Métricas de cobertura 35

1.5.11.2 Métricas de complejidad 39

1.5.11.3 ¿Cuando se deben aplicar las pruebas de Caja Blanca?..... 40

1.5.12 Ventajas de las pruebas de caja blanca. *41*

1.6 Métricas 42

1.6.1 Definiciones de Métricas..... *43*

1.6.2 Métrica del software..... *43*

1.6.2.1 Características de las métricas del software 44

1.6.3 Métricas del Producto *44*

1.6.4 Métricas Técnicas. 45
1.6.5 Métricas de calidad. 45
1.6.6 Métricas de de Halstead. 46
1.6.7 Métricas de McCabe. 48
1.6.8 Otros tipos de métricas. 48
1.6.9 ¿Cuando se debe medir? 48
1.6.10 Ventajas de métricas 49
1.6.11 Cuáles son los costes de no medir. 50
1.6.12 Desventajas de las métricas. 50
1.7 Revisiones de código. 51
1.7.1 ¿Qué son las Revisiones? 52
1.7.2 Revisiones técnicas y de gestión. 53
1.7.3 Inspecciones de código 53
1.7.3.1 Necesidad de las revisiones 53
1.7.4 ¿Qué son las Inspecciones? 54
1.7.4.1 Objetivos de las inspecciones de código 54
1.7.4.2 ¿Para qué sirven las inspecciones? 55
1.7.5 El Proceso de Inspección 55
1.7.6 Relaciones con otras buenas prácticas de codificación 56
1.7.6.1 Revisiones de código y estándares de codificación 56
1.7.6.2 Métricas en inspecciones. 57
1.7.7 Recomendaciones con respecto al equipo de inspección. 58
1.7.8 Ventajas de las Inspecciones de código. 58
1.8 Patrones 59
1.8.1 Definición de Patrón 59
1.8.2 Características de los patrones. 59
1.8.3 Patrones de diseño. 60
1.8.4 Objetivos de los patrones de diseño. 61
1.8.5 Elementos que caracterizan a un patrón de diseño 62
1.8.6 Descripción de los patrones de diseño. 62
1.8.7 Cualidades de los patrones de diseño. 62
1.8.8 Clasificación de los patrones de diseño 63
1.8.8.1 Descripción de los patrones creacionales 64
1.8.8.2 Descripción patrones estructurales. 64

1.8.8.3 Descripción patrones de comportamiento 64
1.8.9 ¿Cuándo utilizar patrones de diseño? *66*
1.8.10 Ventajas de los patrones de diseño *66*
1.9 Estándares de Codificación *67*
1.9.1 ¿Qué es un estándar? *67*
1.9.2 ¿Qué es un estándar de codificación? *68*
1.9.3 Objetivos de los estándares de codificación *69*
1.9.4 ¿Qué comprende un estándar de codificación? *69*
1.9.4.1 Convenciones de legibilidad del código 70
1.9.4.2 Documentación del código 70
1.9.5 Principales estándares de programación *70*
1.9.6 ¿Porqué adoptar los estándares de codificación? *71*
1.9.7 Ventajas de los estándares de codificación *72*
1.10 Conclusiones 73
Capítulo 2 Propuesta y Validación 74
2.1 Introducción 74
2.2 ¿Por qué se eligieron los estándares de codificación? 74
2.3 Necesidad de los estándares de codificación 75
2.4 Necesidad de una guía 76
2.5 Propuesta de una Guía de Estándar Genérico (GEG) 76
2.5.1 Proceso para obtener la Guía de Estándar Genérico *76*
2.5.1.1 Elegir los lenguajes de programación a utilizar 76
2.5.1.2 Determinar los Estándares de codificación a utilizar 77
2.5.1.3 Determinar estándar guía 78
2.5.1.4 Agrupar elementos comunes de los estándares por lenguaje 78
2.5.1.5 Determinar los indicadores comunes 79
2.5.1.6 Comparación de indicadores comunes con el estándar guía 80
2.5.1.7 Plantear la Propuesta 80
2.5.2 Variables para evaluar y controlar los estándares *86*
2.5.2.1 Introducción 86
2.5.2.2 Formato 88
2.5.2.3 Ficheros 90
2.5.2.4 Identación 91
2.5.2.5 Comentarios 92

2.5.2.6 Declaraciones 94
2.5.2.7 Sentencias 94
2.5.2.8 Espaciado 97
2.5.2.9 Convenciones 98
2.5.2.10 Nomenclatura 99
2.5.2.11 Prácticas de programación 102
2.5.2.12 Ejemplos de código 104
2.5.2.13 Recursos 104
2.5.2.14 URLs de Ejemplos 104
2.5.2.15 Glosario de Términos 105
2.5.3 Nivel de acción de la Guía de Estándar Genérico *105*
2.5.4 Herramientas de evaluación *106*
2.5.4.1 Lista de chequeo para estándares 107
2.5.4.2 Lista de chequeo para código fuente 110
2.5.5 ¿Cómo utilizar la guía? *111*
2.5.6 Opiniones de los expertos. *111*
***Conclusiones* 116**
***Recomendaciones* 117**
***Bibliografía* ..... ¡Error! Marcador no definido.**
***Anexos* ..... ¡Error! Marcador no definido.**

Introducción

El desarrollo de software nunca ha sido tan complejo como lo es ahora. Los desarrolladores de software trabajan intensivamente con el conocimiento. No sólo deben comprender nuevas tendencias y tecnologías, sino que necesitan saber cómo aplicarlas de forma rápida y productiva. (PEREZ 2006)

Este planteamiento a pesar de haberse hecho en el 2002, se mantiene vigente, pues aunque han pasado unos cinco años, el desarrollo de software, hoy, es una actividad muy compleja, debido al gran crecimiento en la industria de productos de software, que provoca que en el mercado internacional, sea mayor la competencia entre los mismos, donde la calidad se ha convertido en un factor de mucho peso a la hora de comercializar estos productos de software.

Durante los últimos 40 años, se ha trabajado muy duro con el objetivo de desarrollar técnicas para mejorar la calidad del software, las cuales deben ser aplicadas por los desarrolladores durante todo el ciclo de vida del software y de esta forma garantizar calidad en sus productos desde el principio de su confección hasta la fase de entrega a los clientes. Sin embargo, a pesar de los esfuerzos realizados el proceso de desarrollo de software sigue teniendo problemas en cuanto a que los costos son cada vez más elevados y los productos no poseen la calidad requerida.

Estos problemas están dados porque muchos desarrolladores suelen medir la calidad de un software cuando éste ya está terminado o en fases próxima a su finalización y no hacen énfasis en las fases anteriores y en especial no analizan la calidad del código fuente, ya que muchos programadores tienen la falsa convicción de que no introducen errores o que simplemente no hace falta usar las buenas prácticas destinadas a mejorar la codificación. Lo que trae consigo que se generen diversos problemas, entre ellos, que el equipo de desarrollo generador del código fuente no sepa como entender las metas a lograr, no puedan inspeccionar y probar su trabajo de manera adecuada, no puedan predecir el tamaño y esfuerzo de siguientes proyectos, no exista unificación de criterios en el equipo de desarrollo provocando mala comunicación entre los programadores y

además no se puede reutilizar el código fuente y no se eliminen zonas oscuras del código.

Como se puede ver son muchos los problemas existentes en la fase de implementación por la no utilización de las buenas prácticas de programación, lo que trae como consecuencia que la calidad de los productos en muchas ocasiones sea mala y no resuelva las necesidades de los clientes.

A pesar de estos problemas antes mencionados, muchos países siguen invirtiendo en la industria del software y para muchos de ellos la exportación de software se ha convertido en una importante fuente de ingresos para su economía, tal es el caso de la India que en el 2005 alcanzó ingresos de 17.200 millones de dólares y se ha trazado como objetivo generar ingresos de 50.000 millones de dólares en el período 2008-2009. Otros como Korea en solo dos años lograron incrementar los suyos de 50 a 290 millones de dólares.

Estas cifras describen cuantitativamente lo que puede representar para la economía de cualquier país del mundo tener éxito en el desarrollo de la producción de software para la exportación, lo que ha impulsado a que países en desarrollo y del tercer mundo a pesar de estar en desventaja, tecnológicamente, respecto a los países del primer mundo, estén adentrándose en el mercado internacional de software.

Por ejemplo en Latinoamérica países como México, Brasil y Chile han definido una visión explícita del rumbo que le desean otorgar al desarrollo de su industria de software y están implementando estrategias y políticas específicas. El Salvador, Guatemala y Panamá han iniciado esfuerzos por estimular sus industrias de software.

Cuba es otro de los países de Latinoamérica que siendo del tercer mundo, ha decidido incursionar en el mercado internacional de software, razón por la que hace algunos años y en aras de crear un mayor desarrollo de la informática y las comunicaciones en todo el país, ha venido dedicando numerosos esfuerzos y recursos, para lograr este objetivo. Muestra de ello es que en el 2000 se creó el Ministerio de la Informática y las Comunicaciones (MIC). Hasta el 2003 se había introducido más de 50 mil computadoras

en las 12 mil escuelas del país cifra que entre el 2004 y 2006 se ha incrementado notablemente.

Una experiencia importante en el tema de formación la constituyen los Joven Club de Computación y Electrónica, que han permitido el acceso gratuito a las tecnologías de la información y las comunicaciones a personas de todas las edades, principalmente jóvenes y niños.

Los Joven Club, que ya suman 600, están presentes en todos los municipios del país. Cuentan con su propia red nacional TINORED y tienen más de 6 800 computadoras. Estas instituciones, tienen apreciables logros en la atención a grupos sociales en desventaja, la producción de software y en la generación de contenidos locales.(MINREX 2007)

Además de los programas a cargo de la informática en las universidades del país, a mediados del 2002 se creó la Universidad de las Ciencias Informáticas (UCI), como Universidad de nuevo tipo, nacida al calor de la batalla de ideas, convirtiéndose en un novedoso modelo de formación que combina el estudio con la producción y la investigación. La cual ha sido apoyada por la creación de los Institutos Politécnicos de Informática (IPI).

Además basado en el interés por perfeccionar el trabajo hacia las exportaciones surgió INCUSOFT, marca registrada que identificará a los productos de la denominada Industria Cubana del Software, y fue presentada en la feria internacional Informática 2004.

Otro paso efectivo para penetrar los mercados internacionales fue la creación, en enero de 2003, de AVANTE, Agencia de Negocios del Ministerio de la Informática y las Comunicaciones, que opera mediante convenios de colaboración y facilita estudios de mercados, servicios de inteligencia y gestión, tendencias e integración de mercados.(HERRERA 2004)

La Industria Cubana del Software (ICSW) está llamada a convertirse en una significativa fuente de ingresos nacional, como resultado del correcto aprovechamiento de las ventajas del considerable capital humano disponible. La Universidad de las Ciencias

Informáticas y el sistema de empresas cubanas vinculadas a este trabajo jugarán un papel importante en el desarrollo de la Industria Cubana del Software, y en la materialización de los proyectos asociados al programa cubano de informatización.(HERRERA 2004)

El planteamiento anterior evidencia la gran responsabilidad que tiene la Universidad de la Ciencias Informáticas en sus manos y el alto compromiso que tiene con el país de lograr esta meta.

En los proyectos productivos de la Universidad se han creado una serie de estrategias para la organización del trabajo y para lograr que los productos de software tengan la calidad requerida para el cliente y como meta de los propios desarrolladores, pero a pesar de ello todavía existen algunos problemas a lo largo del ciclo de desarrollo de software y específicamente en la implementación, como:

- Código ilegible.
- Código duplicado.
- Existencia de código muerto.
- Estructura del código muy complicada.
- Código mal estructurado.
- Métodos que necesitan muchos parámetros.
- El código no permite entender el diseño.
- Demasiados números de clases y de métodos.
- El código no es uniforme y difícil de leer.
- Es difícil de mantener el software.
- Código poco robusto y poco confiable.
- Exceso de niveles de anidamiento.

Problemas que influyen negativamente en el desempeño del trabajo en el equipo de desarrollo, en el perfeccionamiento individual del propio desarrollador y en el proceso de codificación, que se pudieran prevenir y/o corregir con la aplicación de Buenas Prácticas como:

- Las Pruebas de caja blanca.
- Los Estándares de codificación.
- Las Métricas.
- Las Revisiones de código.
- Los Patrones de diseño.
- La Refactorización.

Los problemas que pueden ser solucionados por la aplicación de las Buenas Prácticas, describen la **situación problémica** ante la cual se enfrenta la Universidad de las Ciencias Informáticas y que queda planteada de la siguiente forma:

En los proyectos productivos de la Universidad de las Ciencias Informáticas donde se desarrolla software para la exportación y para el consumo nacional, no se usan suficientemente las Buenas Prácticas de programación, las cuales ayudan a optimizar y mejorar el proceso de codificación de software. Específicamente no se utilizan los estándares de codificación de manera adecuada o simplemente no se usan lo que dificulta un buen diseño del código. Esta situación, aparejado a la no existencia de una guía que permita uniformar la escritura de código en la universidad, la cual se ve afectada negativamente en conjunto con el desarrollo del software, evidencian la necesidad de utilización de las buenas prácticas y en especial de los estándares de codificación, teniendo en cuenta el lenguaje de programación, para obtener mejoras en el proceso de codificación, en el de aprendizaje de programación, en la optimización del código fuente de los programa, en la obtención de código legible que brinde la posibilidad de reutilizarlo y mantenerlo luego y en el aseguramiento de la calidad del software.

A partir de la situación problémica descrita el **problema científico** queda planteado de la siguiente manera:

- ❖ No existe una guía en la Universidad de las Ciencias Informáticas, que permita a los programadores elegir el estándar más óptimo a utilizar en los proyectos productivos, esto provoca una inadecuada estandarización o uniformidad en la codificación.

Después de haberse elaborado el problema científico se puede plantear como **objeto de estudio**:

- ❖ Las Buenas Prácticas de Codificación.

Dentro del objeto de estudio se definirá como **campo de acción**:

- ❖ Estándares de Codificación aplicables al entorno UCI.

Al conocer la situación problémica y el problema científico, el **objetivo general** que se ha trazado es:

- ❖ Proponer una guía que contribuya a estandarizar la codificación en la Universidad de las Ciencias Informáticas.

Los **objetivos específicos** que a tener en cuenta para lograr el cumplimiento del objetivo general son los que se plantean a continuación con sus respectivas **tareas de la investigación**:

1. Incrementar el conocimiento sobre el tema teórico que se aborda en la investigación. (Estándares, buenas prácticas, calidad, calidad del código).

   Las **tareas** trazadas para alcanzar este objetivo son las siguientes:

   - ❖ Revisar bibliografía sobre buenas prácticas, calidad del código, ingeniería de software y calidad.
   - ❖ Revisar documentación referente a las buenas prácticas.
   - ❖ Revisar las bases de datos especializadas.
   - ❖ Conceptuar las buenas prácticas de programación teniendo en cuenta las definiciones, clasificaciones, objetivos, ventajas, desventajas y herramientas.
   - ❖ Establecer relaciones entre las diferentes prácticas de programación.

- ❖ Investigar sobre las buenas prácticas de programación que más se utilizan para el mejoramiento de la codificación.
- ❖ Realizar un estudio del los principales estándares de codificación que se aplican para lograr una buena recopilación de sus elementos comunes.

2. Conocer sobre la situación que presenta la Universidad con respecto a las buenas prácticas, estándares de codificación, calidad y calidad del código.

Las **tareas** para lograr este objetivo son:

- ❖ Obtener los elementos comunes de los estándares, que sean aplicables a todos los lenguajes de programación, que se utilizan en los proyectos de la Universidad.
- ❖ Realizar estudios en la Universidad sobre el estado de la aplicación de las buenas prácticas estudiadas.
- ❖ Realizar encuestas y entrevistas a expertos en el tema.

3. Obtener una guía general para la aplicación y comprobación de los elementos comunes.

Las **tareas** trazadas para alcanzar este objetivo son las siguientes:

- ❖ Escribir una propuesta de los niveles y lugares de aplicación la guía obtenida.
- ❖ Escribir dos listas de chequeo para verificar la aplicación correcta de la guía a los estándares de codificación y al código fuente.

Hipótesis

Si se elaborara una guía, con los elementos generales de los estándares de codificación, para el establecimiento de los mismos y se logra que se utilice de forma correcta en todos los proyectos productivos de la UCI, entonces, se podrá establecer una uniformidad en la codificación, a nivel de Universidad, lo que permitirá que se desarrolle software con mayor calidad.

Diseño metodológico

Para el desarrollo de la investigación del presente trabajo se propone utilizar dos métodos de la investigación: **El método teórico** y **el método empírico**.

Dentro del método teórico se utilizará *el Histórico- Lógico mediante* el estudio de la teoría que permitirá llegar a las conclusiones a partir de los conocimientos que se adquieran durante la investigación y para analizar la trayectoria por la que atraviesan los elementos que se tratan en el marco teórico, manejándose la evolución de dichos elementos y sus conexiones históricas fundamentales. En este método se utilizarán principalmente la búsqueda y las consultas bibliográficas.

El *método empírico* se utilizará mediante la realización de *encuestas* y *entrevistas*, que se harán con el fin de de obtener conocimientos a cerca de la situación de las buenas prácticas, y en especial de los estándares de codificación, utilizando para las encuestas como población la UCI y como muestra los proyectos productivos. Además para medir la validez de la solución del problema se empleará como método la entrevista de expertos.

Estructura de la tesis

La tesis está estructurada por tres capítulos y estos a su vez por epígrafes. A continuación se hace una breve reseña sobre el contenido que se trata en cada uno de ellos.

El capítulo 1 expone el panorama de cómo se comporta el estado de las buenas prácticas en el mundo y en la Universidad, así como el objetivo, las características, las clasificaciones, las ventajas, las desventajas y cuando se debe usar cada una, y la relación que existe entre ellas. Se definen conceptos como el de buenas prácticas, calidad, refactorización, pruebas de caja blanca, estándares de codificación, revisiones de código, patrones de diseño y además se exponen sus antecedentes.

El capítulo 2, muestra por qué en el trabajo se le presta mayor atención a los estándares de codificación. Además se hace una selección de estándares de codificación según algunos lenguajes de programación que se utilizan en la Universidad; se eligen los elementos comunes de los estándares de codificación y se hace la presentación de la propuesta que da solución al problema científico, o sea, una guía que establece los elementos generales de los estándares de codificación. También se brindan dos listas de chequeo, una con el objetivo de medir los aspectos que debe tener un estándar de codificación, para ser seleccionado y aplicado según el proyecto y otra, para el código de

los programas. Finalmente se muestra la validación de la propuesta presentada para dar solución al problema.

1. Capítulo 1 Fundamentación teórica

1.1 Introducción

En este capítulo se hace un estudio sobre las buenas prácticas para mejorar la codificación del software, se presenta la situación en la que se encuentra la Universidad con respecto a este tema, así como la importancia que tiene cada una para el mejoramiento de la codificación. Se exponen conceptos importantes como el de Calidad de Software, Refactorización, Pruebas unitarias, Métricas, Patrones de diseño, Revisiones de Código y Estándares de codificación.

1.2 *Calidad del código. Calidad de Software*

La calidad de software es definida por Roger Pressman como:

"Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente." (PRESSMAN 1998)

Es de gran importancia tener en cuenta la calidad desde que se comienza a desarrollar el software y hasta que finalice el mismo, es decir durante todo su ciclo de vida, pues siempre se debe tratar de que las fallas sean mínimas y poco costosas, ya que cuanto más tarde se encuentre una falla, más caro resulta eliminarla. Por ello es necesario que se le preste mucha atención principalmente al proceso de codificación, pues es la fase donde se construye el programa y cada error que se introduzca, influiría negativamente en el desarrollo del software y traería como consecuencia que posteriormente se dedicara más tiempo a la corrección de estos errores, de ahí que la calidad del código sea definitoria en la calidad de Software.

Alcanzar la perfección, es imposible, por lo tanto hay que buscar formas no solo para encontrar errores, sino también para prevenirlos y para mejorar la codificación y asegurar, la obtención de un código de alta calidad y por ende la calidad del software.

Los programadores, que son los encargados de escribir el código y de hacerlo muy rápidamente y además hacer que sea altamente flexible, reutilizable y fácil de mantener, necesitan de nuevas formas y mejores

prácticas que mejoren la codificación, para cualquier lenguaje de programación. Por esta razón muchas han sido las prácticas que se han introducido y que ayudan a obtener un software de calidad.

1.3 Buenas Prácticas para el mejoramiento de la codificación

Las buenas prácticas, desde que comenzaron a aparecer, han sido una parte integral de cualquier práctica en la ingeniería del software. Pero para hablar de algunas de ellas es necesario conocer algunos elementos por los que se caracterizan.

Dodani cita como características de una buena práctica:
• Práctica probada en un contexto. Y entre las prácticas probadas es la que da mejores resultados.
• Está bien documentada y se usa ampliamente.
• Es un elemento bastante ambiguo, del que tampoco se encuentra una definición concreta sobre el término. (DODANI 2003)

Definir un concepto general de buenas prácticas es un poco difícil, por lo que la mejor manera de tratarlas es agrupando los diferentes tipos de prácticas, en este caso enfocados a las que están destinadas, en el ámbito del desarrollo de software a mejorar el proceso de codificación y el código de los programas.

Entre las buenas prácticas que se utilizan para mejorar la codificación se encuentran:

- Las Pruebas de caja blanca.
- Los Estándares de codificación.
- Las Métricas.
- Las Revisiones de código.
- Los Patrones de diseño.
- La Refactorización.
- Listas de chequeo.
- Adiestramiento de los programadores.
- La Programación en pares.

- Aprender English.
- Diseño de un entorno que guíe al desarrollador para corregir y prevenir errores.

Estas y muchas otras se pueden mencionar, pero es preciso tener en cuenta, que no se debe utilizar un grupo numeroso de prácticas, pues pueden atrasar y entorpecer el proceso, de ahí la necesidad de saber cuáles son las más apropiadas y las que más pueden ayudar en el mejoramiento del código y en el arte de codificar.

Para el desarrollo de este trabajo se hizo una investigación de cuales son las más usadas y las que más se recomienda utilizar, a nivel mundial y se seleccionaron: *las Pruebas de caja blanca, la Refactorización, las Revisiones de código, los Patrones de diseño, las Métricas y los Estándares de codificación.*

1.4 Estado de la aplicación de las Buenas Prácticas en la UCI.

Para obtener conocimiento de la situación que tienen las buenas prácticas que se seleccionaron para desarrollar en este trabajo en la Universidad, se hicieron varias entrevistas a programadores de proyectos como GPI de la facultad 7, Simpro de la facultad 5, Prisiones de la facultad 4, Telebanca, y Aduana de la facultad 4, Registro y Notarías de la facultad 3, Control de Acceso de la 1 y entre otros. También se incluyeron trabajadores adjuntos pertenecientes a SOFTEL.

Como resultado de las encuestas se pudo constatar que el conocimiento sobre buenas prácticas no está muy fomentado en la Universidad. Muy pocos de los entrevistados tienen conocimiento de todas las buenas prácticas, y los que más adiestrados están trabajan con estándares de codificación propios, que ellos mismos se han creado para organizar su trabajo.

Algunos proyectos, en el esfuerzo de lograr una uniformidad en la codificación establecen guías a seguir en cuanto a las directivas de declaración. Todo esto demuestra la necesidad de trabajar más sobre los estudiantes acerca de estos temas, que son vitales para formar un programador confiable, que trabaje con calidad. La Universidad necesita divulgar más y establecer el trabajo con Estándares de Codificación que

organicen y aseguren la codificación durante la etapa de implementación. Por lo tanto, se hace necesario un estudio específico sobre las buenas prácticas, para brindar un mayor conocimiento de las mismas y es lo que se pretende hacer en los restantes epígrafes de este capítulo.

1.5 Refactorización.

El término refactorización (refactoring) se atribuye a Opdyke, quien lo introdujo por primera vez en 1992.(OPDYKE 1992)

Sobre esta práctica, se han dado muchas definiciones por diferentes estudiosos del tema en disímiles bibliografías, de las cuales para este trabajo se ha hecho una selección y se ha elaborado una más general a partir de las anteriores, para lograr un mayor entendimiento de qué es en realidad *La refactorización*.

Una *refactorización* es una transformación parametrizada de un programa preservando su comportamiento, que automáticamente modifica el diseño de la aplicación y el código fuente subyacente.

Para Fowler las R*efactorizaciones* son sólo los cambios realizados en el software para hacerlo más fácil de modificar y comprender, por lo que no es una optimización del código, ya que esto en ocasiones lo hace menos comprensible, no soluciona errores, ni mejora algoritmos. Típicamente, una refactorización es una transformación muy simple, que tiene un fácil (pero no necesariamente trivial) impacto en el código fuente de la aplicación.(FOWLER 2000)

Por su parte Crespo y Marquéz destacan cómo la refactorización es una forma especial de reestructuración. Y denominan reestructuración a la modificación directa de los elementos de software. Una reestructuración que involucra un conjunto de elementos relacionados y transforma la forma en que estos se relacionan, se denomina reorganización. (MARQUEZ 2001)

Las Refactorizaciones pueden verse como una forma de mantenimiento preventivo, cuyo objetivo es disminuir la complejidad del software en anticipación a los incrementos de complejidad que los cambios pudieran traer.

1.5.1 Clasificaciones de la Refactorización

Las refactorizaciones se pueden clasificar de acuerdo a la incidencia teniendo en cuenta dos criterios.

1-Si las refactorizaciones actúan sobre código entonces las entidades serían: sistema, clases, atributos, métodos, parámetros, instrucciones, variables locales.

2-Si se aplican sobre refactorizaciones de diagramas de estado, las entidades serían: acción, estado y transición.

1.5.2 ¿Qué es Refactorizar?

De acuerdo con Martin Fowler, en su excelente libro, Refactoring (Refactorización):

Refactorizar es el proceso de realizar en un sistema de software modificaciones en el código con el objetivo de mejorar su estructura interna, sin alterar su comportamiento externo. Es una manera disciplinada de limpiar código que minimiza las oportunidades de introducir los bugs. En esencia, cuando se refactoriza se está mejorando el diseño del código después que haber sido escrito.

"El proceso de cambiar un sistema software para mejorar su estructura interna, preservando el comportamiento externo de código. El proceso de refactorización busca mejorar el diseño de código una vez que este ha sido escrito, de esta forma se puede hacer frente al mantenimiento de un código de calidad en las continuas reestructuraciones al que se expone en la etapa de mantenimiento".

Un concepto más completo, resultado de los anteriores y vinculado con términos tan importantes como la calidad de software es:

Refactorizar significa cambiar el código internamente sin alterar su funcionalidad externa con motivos de mejorar el diseño y obtener un código más simple. Utilizando técnicas que permitan descubrir y cambiar el código de mala calidad y aplicándose donde se estime conveniente ya sea, a nivel de objeto, entre dos objetos, entre grupos de objetos y en escala grande.

No obstante, obtener un código sencillo no es nada fácil.La Refactorización debe ser una práctica que ayude al programador a crear un código más sencillo y estructurado, mejorando el diseño del software y haciéndolo factible a nuevos cambios cuando no se cuenta con un período de tiempo suficiente.

1.5.3 Bases Generales de un Proceso de Refactorización

La refactorización es una técnica aplicable a casi cualquier metodología de desarrollo de software. Si bien es cierto que cada metodología en particular puede tratarla de una manera concreta (generalmente marcando “cuándo” y “cuánto” refactorizar), existe una secuencia general que se recomienda utilizar a la hora de refactorizar, similar a la siguiente:

- Revisar código y diseño para identificar refactorizaciones.
- Aplicar una refactorización cada vez, sin cambiar la funcionalidad.
- Aplicar pruebas unitarias, después de cada refactorización sin excepción.
- Repetir los pasos anteriores para encontrar más refactorizaciones que aplicar.

Además, existen algunas importantes reglas heurísticas a tener en cuenta durante el proceso de refactorización.

- **No añadir funcionalidad a la vez que se refactoriza**.

La regla básica de la refactorización es no cambiar la funcionalidad del código o su comportamiento observable externamente. El programa debería comportarse exactamente de la misma forma antes y después de la refactorización. Si el comportamiento cambia, entonces no se puede decir que la refactorización no ha estropeado lo que antes ya funcionaba.

- **Uso estricto de las pruebas.**

Al realizar pruebas en cada paso se reduce el riesgo del cambio, siendo este un requisito obligatorio tras la aplicación de cada refactorización individual.

- **Aplicar muchas refactorizaciones simples.**

Cada refactorización individual puede realizar un pequeño progreso, pero el efecto acumulativo de aplicar muchas refactorizaciones resulta en una gran mejora de la calidad, legibilidad y diseño del código.

1.5.4 ¿Cuándo se debe refactorizar?

Respecto a cuándo se debería refactorizar hay que decir que la refactorización es algo que debe hacerse de forma constante y en pequeños pasos, sin tener por qué dedicar tiempo adicional o exclusivo.

La refactorización es una práctica, que al igual que ninguna otra práctica o técnica de programación, se puede aplicar sin antes tener un conocimiento sólido de en qué situaciones y bajo qué condiciones se pueden aplicar. Para su aplicación es necesario tener en cuenta que la principal finalidad de la misma es ayudar al programador a mejorar el diseño de su código y por transitividad, el software correspondiente, no a complicarlo más de lo que pueda estar o traerle complicaciones que no tenía.

Por tanto para que se tenga una noción de cuándo es que se debe refactorizar a continuación se citan una serie de situaciones ante las cuales es necesario refactorizar:

- Se quiere agregar una nueva funcionalidad al software y el código existente es difícil de entender.
- Cambiar algo en el diseño podrá facilitar cambios futuros.
- Se está haciendo mantenimiento a un software.
- Al depurar. Muchas veces el hecho que haya un defecto a corregir, debe ser tratado como una señal de que el código es confuso y necesita refactorizarse.
- Se está haciendo revisión de código.
- Se detecte funcionalidad repetida.
- Se tiene una idea para simplificar la estructura del código.
- Si la estructura del código es muy complicada.
- Se identifica código mal estructurado.
- Los diseños supongan un riesgo para la futura evolución del sistema.

- Se está solucionando un “bug”.
- Se desea reducir y simplificar el código.
- Existe algún indicador o “bad smell”(malos olores) como también se le conoce que deba solucionarse.
- Se encuentran puntos en común de una historia con otras implementadas y se quiera obtener una solución común que cubra ambos casos, partiendo de la historia ya implementada.

Toda esta serie de causas que inducen el uso de la Refactorización demuestran la necesidad de alguna técnica o método preventivo que las reduzca al menor número posible. Y es que se deben considerar los medios existentes en la actualidad, que con su aplicación, difícilmente las refactorizaciones se enfrentarían a situaciones tan catastróficas, lo cual facilitaría en gran medida su trabajo y la posibilidad de obtener un mejor resultado.

1.5.5 ¿Cuándo no se debe refactorizar?

Si bien es importante conocer las diferentes situaciones en las cuales se recomienda refactorizar también es necesario tener conocimiento de en cuáles no se debe llevar a cabo esta acción, pues este es un proceso que si se aplica bien y cuando es necesario tiene buenos resultados pero si por el contrario se hace cuando no se debe, estos resultados pueden ser nefastos.

Las siguientes situaciones son las principales en la que no se debe refactorizar, pues no es recomendable cambiar el código:

- Es más fácil hacer el código de nuevo.
- El código no funciona.
- La fecha de entrega comprometida es demasiado cerca.
- La complejidad de la solución refactorizada es mayor que la actual.

Estas situaciones son actividades que requieren de un método organizado, que afecte lo menos posible la confección del producto, y que establezca cómo se debe llevar a cabo el proceso de forma controlada, para ello se deben tener en cuenta una serie de actividades como:

- La identificación de zonas del sistema a refactorizar.
- Determinación de un conjunto de refactorizaciones a aplicar.
- Aplicación del conjunto elegido.
- Computar los efectos de la refactorización sobre atributos de calidad del software.
- Mantener la consistencia entre los distintos artefactos software.

1.5.6 ¿Cómo se relaciona la Refactorización con otras Buenas Prácticas de programación?

Cuando se quiere implantar la refactorización de manera progresiva a nivel de equipo, como una buena práctica para mejorar el código es necesario tener en cuenta los pasos que permiten que sea un proceso organizado, en los cuales se incluye, una serie de vínculos de las refactorizaciones con otras técnicas como las pruebas unitarias, los patrones de diseño e incluso con el uso de herramientas, ya sea para mejorar su aplicación, como para implementarlas a partir de ellas.

1.5.6.1 Relación Refactorización- Pruebas Unitarias

La refactorización depende en gran medida, de la existencia de pruebas unitarias para que el proceso no sea arriesgado ni demasiado costoso, además las utilizan para verificar que la funcionalidad implementada no ha sufrido problemas.

Las pruebas unitarias, facilitan enormemente a las refactorizaciones disminuir sus riesgos, dado que es posible comprobar cuando esta concluye que todo sigue funcionando satisfactoriamente.

Permiten conocer si el desarrollo sigue cumpliendo los requisitos que implementaba.
Disminuyen el riesgo, dado que es el desarrollador que realiza la refactorización quien puede comprobar que no ha estropeado la funcionalidad implementada por otros. Evitan efectos colaterales, además de

disminuir el tiempo necesario para comprobar que todo sigue funcionando, lo que permite avanzar en pasos más pequeños si se desea.

De ahí que la ausencia de pruebas unitarias durante la refactorización, conduce a un proceso inseguro, que nunca sabrá si se introdujeron nuevos errores. No se debe refactorizar sin tests (pruebas), salvo para las refactorizaciones más sencillas y realizadas con herramientas especializadas.

1.5.6.2 Relación Refactorización- Patrones de diseño

Una buena base teórica sobre las refactorizaciones más comunes permite al programador detectar “Bad Smells” de los que no es consciente y que harán que su código se degrade progresivamente. La mejor forma de enfocar correctamente una refactorización y evitar caer en espirales de refactorizaciones, es conocer muy bien los patrones de diseño. Conocerlos perfectamente permite identificar y ver con claridad el objetivo de la refactorización.

Utilizarlos mejora la comunicación entre los miembros del grupo al compartir una base de conocimiento muy importante, facilitando las refactorizaciones y hace más eficientes la resolución no sólo de problemas conocidos, sino también de problemas desconocidos al haber adquirido una forma con mejor arquitectura para enfocar los problemas.

1.5.7 Ventajas de la refactorización

Refactorizar es una forma de mejorar la calidad y por tanto es una forma de hacer que el desarrollador se sienta satisfecho de su trabajo, si logra los objetivos de la refactorización.
Esta técnica posibilita un mejoramiento continuo del código, ya sea durante la implementación o posterior a esta. Por estas razones para ilustrar las ventajas que puede tener esta práctica sería más fácil hacerlo teniendo en cuenta una serie de parámetros en las que la refactorización influye positivamente y sería conveniente o importante comenzar destacando su influencia sobre la calidad del software.

- Las Refactorizaciones aportan un código de calidad, sencillo y bien estructurado, que cualquiera pueda leer y entender sin necesidad de haber estado integrado en el equipo de desarrollo durante varios meses**.** Permiten que el desarrollador aprenda a trabajar en un entorno

en el que no se cuenta con mucho tiempo. Ayudan a encontrar errores, al posibilitar la comprensión del código y a visualizar errores.

- Agiliza el proceso de construcción del software, evitando la duplicación de código y simplificando el diseño, muy beneficioso cuando se tiene que realizar modificaciones, tanto para corregir errores como para añadir nuevas funcionalidades.

- Posibilita el diseño Evolutivo en lugar de Gran Diseño Inicial pues refactorizar permitirá ir evolucionando el diseño según se incluyan nuevas funcionalidades, lo que implica muchas veces cambios importantes en la arquitectura, añadir cosas y borrar otras.

- Evita la Reescritura de código pues en la mayoría de los casos refactorizar es mejor que rescribir. No es fácil enfrentarse a un código que no se conoce y que no sigue los estándares que se debe utilizar, pero nunca la mejor opción es empezar de cero. Sobre todo en un entorno donde el ahorro de costes y la existencia de sistemas lo hacen imposible.

- Independencia de lenguaje: En este sentido se presenta un lenguaje modelo que contiene el suficiente poder expresivo para representar las construcciones abstractas necesarias en la definición y análisis de refactorizaciones y avanzar hacia la posibilidad de establecer una cierta independencia del lenguaje en este campo.

1.5.8 Desventajas de la refactorización

Las refactorizaciones también tienen una serie de desventajas, que no la descartan como buena práctica a utilizar, pero que demuestra que es una técnica con limitaciones por lo que necesita auxiliarse de otras prácticas.

- Muchas de las refactorizaciones se pierden en "micro-eficiencias", que son tan pequeñas que un buen compilador optimizador las puede deshacer al compilar.

- Dependen de la existencia de pruebas unitarias para que el proceso no sea arriesgado ni demasiado costoso.

- Se convierten en un estorbo cuando no existe acoplamiento, ya que las refactorizaciones grandes, afectan el diseño del código significativamente, y obligan a realizar cambios en las mismas, lo que provoca una situación incómoda.

- En ocasiones incluso algunas refactorizaciones muy pequeñas pueden provocar la aparición de bugs muy difíciles de identificar posteriormente.

1.5.8.1 Problemas de la Refactorización

Entre los problemas detectados en el ámbito de la refactorización que la hacen desventajosa se pueden destacar:

Cambio de las Interfaces. Muchas refactorizaciones cambian la interfaz. Lo que no es problema si se tiene acceso a todo el código, ya que se cambiaría de nombre al servicio y se renombrarían todas las llamadas a ese método. El problema aparece cuando las interfaces son usadas por código que no se puede encontrar y/o cambiar.

Problemas para Bases de Datos. La mayoría de las aplicaciones están fuertemente acopladas al esquema de la base de datos. Esta es una de las razones por la que la base de datos es difícil de cambiar. Otra razón es la migración de los datos de una base de datos a otra, algo bastante costoso.

1.5.9 Pruebas de Software

La calidad desempeña un rol determinante para la competitividad de las empresas y la satisfacción del cliente y para que esta sea asegurada en el desarrollo del software debe ir acompañado de actividades que la garanticen.

La prueba es un elemento crítico para la garantía de la calidad del software. La importancia de los costes asociados a los fallos motiva la creación de un proceso de pruebas minuciosas y bien planificadas.

Hoy en día se calcula que la fase o proceso de pruebas representa más de la mitad del coste de un programa, ya que requiere un tiempo similar al de la programación lo que obviamente acarrea un alto costo económico cuando estos no involucran vidas humanas, puesto que en este último caso el costo suele superar el 80% siendo esta etapa más cara que el propio desarrollo y diseño de los distintos programas que conforman un sistema.

El proceso de prueba está dividido en varios tipos de pruebas dentro de las cuales las más importantes son: pruebas unitarias, pruebas de integración, pruebas de sistema y pruebas de aceptación. En este epígrafe se abordará un caso específico de las pruebas unitarias: las pruebas de Caja Blanca.

1.5.9.1 Pruebas Unitarias

Una prueba de unidad es la que prueba un hardware individual o unidades del software o grupos de unidades relacionadas. Una unidad es un componente de software que no puede ser dividido en otros componentes. (IEEE 1990a)

A las pruebas que tienen como objetivo probar el comportamiento de cada uno de los componentes de forma independiente y comprueban qué módulos concretos cumplen las funcionalidades esperadas se les llama prueba de unidad.

Estas están divididas en dos criterios el de pruebas de caja negra o funcional y el de pruebas de caja blanca o estructural.

El primer criterio o pruebas de caja negra, se enfoca en la interfaz y en probar la entrada y salida de datos, verificando la funcionalidad del software, prescinde de los detalles del código y se limita a lo que se ve desde el exterior e intenta descubrir casos y circunstancias en los que el módulo no hace lo que se espera que haga.

Mientras que el otro criterio, pruebas de Caja Blanca, son aquellas que por el apoyo que ofrecen al proceso de codificación, son un caso contrario de las de caja negra y son a las que se le dedicará este epígrafe, pues se puede decir que miran con lupa el código escrito e intentan que falle, con el objetivo de probar la ejecución de todos los caminos independientes, así como encontrar cualquier error de código que afecte al producto.

Las características generales de las pruebas unitarias se hacen extensivas para sus dos casos particulares razón por la que tanto las pruebas funcionales como las estructurales tienen que cumplir fundamentalmente con tres normas fundamentales y con las características que aparecen a continuación.

1.5.9.2 Características de las pruebas unitarias

- Todo el código debe tener sus correspondientes pruebas de unidad.
- El código debe pasar todas sus pruebas de unidad antes de ser liberado o publicado.
- Cuando se encuentra un bug, se hacen pruebas de unidad adicionales para él.
- Puede realizarse paralelamente en varios módulos.
- La depuración es el proceso que elimina el error, por eso es importante que las pruebas se pasen siempre al 100%.
- Deben ser automatizables, por ejemplo, formando parte del proceso de compilación.
- La etapa de pruebas no debe ser posterior a la confección de un programa, tiene que ser paralela a la programación. En casos que sea anterior, primero probar y después programar.

Las tres normas que hay que tener en cuenta para que una prueba sea exitosa y que se consideran adecuadas son:

- La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.

- Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.

- Una prueba tiene éxito si descubre un error no detectado hasta entonces.

1.5.10 Pruebas de caja blanca

La prueba de Caja Blanca es:
La prueba que toma en cuenta el mecanismo interno de un sistema o componente. (IEEE 1990b)

Se definen a partir del conocimiento que se tiene de la estructura interna del sistema.

Los objetivos principales son:

- Verificar que se ejecuten todas las instrucciones del programa.
- Hacer que el programa ejecute todas sus sentencias al menos una vez.
- Intentan garantizar que todos los caminos de ejecución del programa quedan probados.

Además pruebas estructurales analizan un programa para determinar las trayectorias y utilizan este análisis para escoger los datos de prueba que sean más útiles (La finalidad de los datos de prueba es encontrar errores de funcionamiento), usando la estructura de control (camino básico, condición, bucles) para obtener dichos casos de prueba.

La prueba de caja blanca es conocida también como prueba estructural, prueba de caja limpia y prueba de caja transparente. (PRESSMAN 1998)

En otras bibliografías se tratan como pruebas de código, debido a que este tipo de prueba se centra en la estructura interna (implementación) del programa para elegir los casos de prueba.

Un concepto más específico de caja blanca es el que se da en la quinta edición del Pressman:

La prueba de caja blanca, denominada a veces prueba de caja de cristal, es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de prueba que:

- Garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo.
- Ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa.
- Ejecuten todos los bucles en sus límites y con sus límites operacionales.
- Ejerciten las estructuras internas de datos para asegurar su validez.(PRESSMAN 1998)

1.5.11 Clasificación de las Pruebas de Caja blanca.

Los métodos de caja blanca se pueden clasificar en dos grupos:

- Los basados en métricas de cobertura.
- Los basados en métricas de complejidad.

1.5.11.1 Métricas de cobertura

La cobertura es una medida del porcentaje de código que ha sido probado o “cubierto” con las pruebas. A medida que se ejecuten pruebas sobre el sistema la cobertura irá aumentando hasta alcanzar dicho 100%. Evidentemente, el límite del 100% nos indica que debemos de parar de ejecutar pruebas. Los criterios para definirla se obtienen de los elementos que conforman los programas a cubrir.

Las métricas de cobertura sirven de base a una serie de técnicas de caja blanca y se debe tener en cuenta que todo programa se puede representar mediante un grafo de flujo de control, donde cada nodo (círculo) es una sentencia o una secuencia de sentencias que representan una o más acciones del módulo. Los arcos (aristas o flechas) dirigidos en el grafo representan el flujo de control entre los distintos nodos.

Para cada conjunto de datos de entrada el programa se ejecutará a través de un camino concreto dentro de este grafo. Cuando el programa incluye estructuras iterativas, el número de posibles caminos en el grafo puede ser infinito.

Una prueba de caja blanca exhaustiva requeriría la generación de un caso de prueba por cada posible camino. Como esto no es posible, por lo general, se utilizan métricas que dan una indicación de la calidad de un determinado conjunto de casos de prueba en función del grado de cobertura del grafo que consiguen.
Las métricas que más se utilizan son:

- La cobertura de sentencias o condiciones.
- Cobertura de condición/decisión o cobertura de segmentos entre decisiones.
- Cobertura de decisiones de ramificación.
- Cobertura de caminos.
- Cobertura de bucles.

1.5.11.1.1 Cobertura de sentencias o condiciones

Después de pasar las pruebas de caja negra se obtiene a mano un listado del programa y se van marcando las líneas de código que se ejecutan, de esta forma se puede determinar que porcentaje de líneas ha sido ejecutado alguna vez, porcentaje que se conoce como cobertura de sentencias. En este tipo de cobertura se trata de generar los casos de prueba necesarios para que cada sentencia o instrucción del programa se ejecute al menos una vez.

Como un segmento es una secuencia de sentencias sin puntos de decisión a la cobertura de sentencias, también se le conoce la cobertura de segmentos la cual debe tener en cuenta que el número de sentencias es finito, por lo tanto hay que ser precavido a la hora de elegir cuándo parar además no se suele pasar por todas las sentencias sino por una mayoría elegida adecuadamente. Es muy recomendable alcanzar una elevada cobertura de sentencias, aunque no siempre es posible por premura de tiempo o medios.

En la práctica, el proceso de pruebas termina antes de llegar al 100%, pues puede ser excesivamente laborioso y costoso provocar el paso por todas y cada una de las sentencias. Pues la existencia de código muerto puede implicar la imposibilidad de llegar al 100% de cobertura. Un valor del 100% en cobertura de segmentos implica un valor del 100% en cobertura de trazas, puesto que toda traza está incluida en algún segmento.

1.5.11.1.2 Cobertura de condición/decisión o cobertura de segmentos entre decisiones

La Cobertura de condición/decisión permite conocer que ocurre cuando la expresión booleana es más compleja como es el caso de (if (condición1 || condición2) {Haz Esto}) donde hay solo dos ramas pero cuatro posibles combinaciones, por lo tanto, hay que definir un criterio de cobertura sobre las condiciones. Además consiste en exigir el criterio de cobertura de condiciones obligando a que se cumpla también el criterio de decisiones. En este tipo de cobertura se le da tratamiento a la mezcla de dos coberturas, las de condición y las de decisión.

La cobertura de decisiones consiste en escribir casos suficientes para que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso. En general, una ejecución de pruebas que cumple la cobertura de decisiones cumple también la cobertura de sentencias.

En cambio, en la cobertura de condiciones se trata de diseñar tantos casos como sea necesario para que cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez. No se puede asegurar que si se cumple la cobertura de condiciones se cumple necesariamente la de decisiones.

1.5.11.1.3 Cobertura de decisiones de ramificación

A este tipo de cobertura se le debe dar tratamiento a los segmentos opcionales (if (condición) {EjecutaEsto}) probando cuándo la condición se cumple y cuando falla. Además contribuye al refinamiento recorriendo todas las posibles salidas de los puntos de decisión (y excepciones).

Se habla de una cobertura de ramas al 100% cuando se ha ejercitado todas y cada una de las posibles vías de ejecución controladas por condiciones. La cobertura de ramas es indiscutiblemente deseable; pero habitualmente es un objetivo excesivamente costoso de alcanzar en su plenitud. Si se logra una cobertura de ramas del 100%, esto llevaría implícita una cobertura del 100% de los segmentos, pues todo segmento está en alguna rama.

1.5.11.1.4 Cobertura de caminos

La cobertura de caminos (secuencias de sentencias) es el criterio más elevado: cada uno de los posibles caminos del programa se debe ejecutar al menos una vez. Se define camino como la secuencia de sentencias encadenadas desde la sentencia inicial del programa hasta su sentencia final. Para reducir el número de caminos a probar, se habla del concepto de camino de prueba (test path): un camino del programa que atraviesa, como máximo, una vez el interior de cada bucle que encuentra.

Especialistas recomiendan que se pruebe cada bucle tres veces: una sin entrar en su interior, otra ejecutándolo una vez y otra más ejecutándolo dos veces. Esto último es interesante para comprobar cómo se comporta a partir de los valores de datos procedentes, no del exterior del bucle (como en la primera iteración), sino de las operaciones de su interior.

1.5.11.1.5 Cobertura de bucles

Los bucles no son más que segmentos controlados por decisiones. Así, la cobertura de ramas cubre plenamente la esencia de los bucles. Pero eso es simplemente la teoría, pues en la práctica los bucles son una fuente inagotable de errores, todos triviales, algunos mortales por lo tanto se debe hacer pruebas.

- Para bucles WHILE con cero, una y más de una ejecución.
- Para bucles DO/WHILE con una, y más de una ejecución.

Para los FOR basta con ejecutarlos una vez por la seguridad que brindan pues en su cabecera está definido el número de veces que se va a ejecutar y de ello se encarga el compilador. Si dentro del bucle

se altera la variable de control, o el valor de alguna variable que se utilice en el cálculo del incremento o del límite de iteración, entonces eso es un bucle FOR con trampa.

Si el programa contiene bucles LOOP, o simplemente bucles con trampa, la única cobertura aplicable es la de ramas. El riesgo de error es muy alto; pero no se conocen técnicas sistemáticas de abordarlo, salvo rescribir el código.

Hay que tener presente que ello es aplicable para todos los bucles a los que alguien se pueda enfrentar, sean bucles simples, anidados (bucles internos y externos), concatenados (uno a continuación del otro) o no estructurados.

Los bucles constituyen el elemento de los programas que genera un mayor número de problemas para la cobertura de caminos. Su tratamiento no es sencillo ni siquiera adoptando el concepto de camino de prueba.

1.5.11.2 Métricas de complejidad

La complejidad ciclomática es una métrica de software extremadamente útil pues proporciona una medición cuantitativa de la complejidad lógica de un programa. El valor calculado como complejidad ciclomática define el número de caminos independientes del conjunto básico de un programa y nos da un límite superior para el número de pruebas que se deben realizar para asegurar que se ejecute cada sentencia al menos una vez.

Las métricas de complejidad más utilizadas en la generación de casos de prueba son las de McCabe:

- Complejidad ciclomática (arcos - nodos + 2 * número de componentes conexos)
- Complejidad esencial (complejidad ciclomática - número de sub-grafos propios de entrada y salida única)
- Complejidad real (número de caminos ejecutados)

1.5.11.2.1 Utilización de la complejidad ciclomática de McCabe

La utilización de la métrica de McCabe es una métrica del software extremadamente útil. Esta métrica es un indicador del número de caminos independientes que existen en un grafo. El propio McCabe definió como un buen criterio de prueba la consecución de la ejecución de un conjunto de caminos independientes, lo que implica probar un número de caminos igual al de la métrica.(MCCABE 1994)

La complejidad de McCabe V (G) se puede calcular de las tres maneras siguientes a partir de un grafo de flujo G:

V (G) = a n + 2, siendo a el número de arcos o aristas del grafo y n el número de nodos.
V (G) = r, siendo r el número de regiones cerradas del grafo.
V (G) = c + 1, siendo c el número de nodos de condición.

El valor V (G) brinda un límite superior para el número de caminos independientes que componen el conjunto básico y, consecuentemente, un valor límite superior para el número de pruebas que se deben diseñar y ejecutar para garantizar que se cubren todas las sentencias del programa.

1.5.11.3 ¿Cuándo se deben aplicar las pruebas de Caja Blanca?

Para realizar las pruebas se requiere gente que disfrute encontrando errores, por eso no es bueno que sea el mismo equipo de desarrollo el que lleve a cabo este trabajo.

En las pruebas de caja blanca no es mala idea probar un 85% de las ramas y dar por terminado luego de esto.

En dependencia del proyecto en cuestión y del modelo de proceso elegido, las pruebas de caja blanca se pueden aplicar:

- Después del análisis, el diseño y la programación.
- En forma paralela a las fases citadas.

- Repetirse varias veces durante la duración del proyecto.
- Cuando se quiere encontrar la mayor cantidad de errores posible.
- Cuando se quieran probar todos los caminos independientes.
- Cuando se quieran probar todas las condiciones.
- Cuando se quieran probar los bucles.
- Cuando se quiera verificar la implementación interna.
- Cuando se quiere controlar el funcionamiento de pequeñas porciones de código como subprogramas (en la programación estructurada) o métodos (en POO).
- Cuando se considera que un módulo está terminado se realizan las pruebas sistemáticas, el objetivo de buscar fallos a través de un criterio específico, estos criterios se denominan "pruebas de caja negra y de caja blanca", que forman parte de las pruebas unitarias.
- Cuando se quieren probar módulos sueltos usando una fase informal y una fase sistemática. La fase informal, la lleva a cabo el propio codificador en su despacho, y consiste en ir ejecutando el código para convencerse de que “básicamente, funciona”. Consiste en pequeños ejemplos que se intentan ejecutar. Si el módulo falla, se suele utilizar un depurador para observar la evolución dinámica del sistema, localizar el fallo, y repararlo.

1.5.12 Ventajas de las pruebas de caja blanca.

Las pruebas de caja blanca son muy beneficiosas para el desarrollo del software, es la mejor dentro de los métodos de prueba para verificar que se recorran todos los caminos y detectar un mayor número de errores. Entre las principales ventajas que brinda se encuentran:

- Unificación de Estándares de codificación. Las pruebas obligan a todo el mundo a programar de la misma manera. Y eso, facilita el mantenimiento.
- Mejor Estructura de Código. Porque para probar es necesario que el código esté bien estructurado. Y eso, también es bueno.
- Se puede refactorizar sin miedo.
- Ayudan también a comprobar que una instalación funciona correctamente.
- Ayudan a encontrar errores de codificación.

- El programador se vuelve más consciente de sus decisiones de implementación.
- Consiguen encontrar defectos de complicada reproducibilidad por otros tipos de test.
- Consiguen encontrar operaciones que podrían proporcionar problemas futuros.
- Al igual que las revisiones de código permiten que la acción de encontrar errores sea considerablemente más sencilla, ya que se hace con el código a mano.
- También sirve de comprobación a la hora de enumerar los caminos.
- Reducen de forma drástica la necesidad de pruebas al finalizar un módulo.
- Cuando se utilizan en la programación orientada a las pruebas ayudan a priorizar y comprobar la evolución del desarrollo y ofrecen realimentación inmediata.
- Permiten conseguir un código simple y funcional de manera rápida cuando se implementan y codifican la solución.
- Cuando se desarrollan enfocadas a la programación extrema permiten la realización de pruebas completas y la pronta detección de problemas de incompatibilidad.
- Reducen el riesgo de introducir errores cuando se modifica un código ajeno al programador que revisa dado que avisarán al instante si algo deja de funcionar.

1.6 Métricas

Para efectuar una estimación correcta de los costes de un proyecto informático de construcción de software, se debe disponer de unidades de medida que permitan relacionar las diferentes actividades de la construcción del software con el esfuerzo en horas de trabajo o con el coste monetario que representa tener su construcción.

Medir el software no es en absoluto una actividad sencilla, pero es una necesidad hacerlo y en ocasiones se dificulta mucho, lo que ha provocado el nacimiento de varias métricas o unidades de medida, destinadas a cuantificar diferentes aspectos, que pueden ser relevantes en el desarrollo de software y se pueden clasificar en tres grupos, orientadas a evaluar tres grandes magnitudes: la calidad, el tamaño (a menudo del producto) y la productividad (frecuentemente del proceso de construcción del producto), aunque lo hacen desde muchos puntos de vista diferentes.

En este caso el principal interés se enfoca a las métricas orientadas a evaluar la calidad del código fuente midiendo procesos, productos y proyectos. Entonces es importante definir en realidad qué es una Métrica.

1.6.1 Definiciones de Métricas

Una métrica es una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado. Incluye el método de medición.

Las métricas son un buen medio para entender, monitorizar, controlar, predecir y probar el desarrollo software y los proyectos de mantenimiento.(BRIAND 1996)

Según [Fenton] la métrica es la correspondencia de un dominio empírico (mundo real) a un mundo formal, matemático. La medida incluye al valor numérico o nominal asignado al atributo de un ente por medio de dicha correspondencia.

Las métricas son observaciones cuantitativas (mediciones). Una medición es una actividad que usa la definición de la métrica para producir el valor de una medida. Una medida, de acuerdo a las [ISO 14598-1:1999] es un número o categoría asignada a un atributo de una entidad mediante una medición.
En general,

Una métrica está definida para uno o más atributos. Dos métricas pueden relacionarse mediante una función de transformación. El tipo de dicha función de transformación va a depender del tipo de escala de ambas métricas. Una métrica puede expresarse en una unidad.

1.6.2 Métrica del software

Las métricas del software son definidas por un amplio rango de actividades diversas, que miden aspectos del software.

- Medidas y modelos de estimación de coste y esfuerzo
- Medidas modelos de productividad.

- Aseguramiento y control de calidad.
- Recogida de datos.
- Medidas y modelos de calidad.
- Modelos de fiabilidad.
- Modelos y evaluación de ejecución.
- Complejidad computacional o algorítmica.
- Métricas estructurales o de complejidad.

1.6.2.1 Características de las métricas del software

- Simple y fácil de calcular.
- Empírica e intuitiva.
- Sin ambigüedades y objetiva.
- Consistente en el empleo de unidades y tamaños.
- Independiente del lenguaje de programación.
- Eficaz para aumentar la calidad del software.
- Medidas deben estar identificadas para soportar los objetivos de la organización.
- Medidas deben estar bien definidas y unidas al proceso software global.
- El proceso debe ser consistente, repetible y evolucionando continuamente.
- Por otra parte no existen estándares para las métricas y, por lo tanto existe ayuda limitada para la recolección y análisis de datos.

1.6.3 Métricas del Producto

Están las métricas del producto que son las que miden diferentes aspectos del software obtenido, a menudo a partir de código fuente expresado en un lenguaje informático determinado (Lenguajes de programación, de diseño, de especificación)

Estas se dividen en dos clases:

Métricas dinámicas: recolectadas por las mediciones hechas en un programa en ejecución. Ayudan a valorar la eficiencia y la fiabilidad de un programa.

Las métricas estáticas: recolectadas por las mediciones hechas en las representaciones del sistema como el diseño, el programa o la documentación. Ej. Métricas del código, métricas de calidad y de complejidad.

1.6.4 Métricas Técnicas.

Estas se presentan en el libro de Ingeniería del software de Pressman. Estas métricas se derivan de una relación empírica según las medidas contables del dominio de información del software y de evaluaciones de complejidad.

Se aplica las métricas para valorar la calidad de los productos de ingeniería o los sistemas que se construyen.

Proporcionan una manera sistemática de valorar la calidad basándose en un conjunto de reglas claramente definidas.
Se aplican a todo el ciclo de vida permitiendo descubrir y corregir problemas potenciales.

1.6.5 Métricas de calidad

Una métrica es una medida estadística y no cuantitativa que se aplica a todos los aspectos de calidad de software, los cuales deben ser medidos desde diferentes puntos de vista como el análisis, construcción, funcional, documentación, métodos y procesos, usuario, entre otros.

Este tipo de métricas te permite predecir la calidad midiendo atributos internos del código, como el tamaño y suponiendo que existe una relación entre lo que se puede medir y lo que se quiere saber.

A menudo es imposible medir los atributos de calidad del software en forma directa.
Los atributos como la complejidad, la mantenibilidad y la comprensión se ven afectados por diversos factores y no existen métricas directas para ellos.

Más bien es necesario medir un atributo interno del software (como el tamaño) y de forma ideal, existe una relación clara válida entre los atributos de software internos y externos.

Una métrica de calidad que se utiliza es:

DENSIDAD DE DEFECTOS = número de defectos / tamaño del producto

Esta medida está muy relacionada con la forma y el proceso de búsqueda y detección de defectos. Nos puede decir más de la calidad de este proceso que del producto. Un producto en el que se han detectado pocos defectos puede indicar que el proceso de pruebas ha sido poco exhaustivo y la mayor parte de los defectos permanecen ocultos.

1.6.6 Métricas de de Halstead.

Además de abordar la medición de la longitud de código, se ocupan de la complejidad de este. Halstead introdujo el concepto de token como medida más interesante que las LOC. Su objetivo era lograr una medida independiente del lenguaje de programación. Un tokens es la unidad sintáctica más elemental distinguible por un compilador. Estos tokens pueden clasificarse como operadores u operandos.

Las métricas de Halstead consideran que un programa está formado por una serie de partículas, las cuales pueden ser consideradas como operadores u operandos. Mientras que los operadores son los símbolos que afectan el valor u orden del operando.

Para esto se utiliza un conjunto de medidas que pueden obtenerse una vez que se ha generado o estimado el código después de completar el diseño.

Estas medidas son:

n1: número de operadores distintos que aparecen en el programa.

n2: número de operandos distintos que aparecen en el programa.
N1: número total de operadores.
N2: número total de operandos.

A partir de estas métricas básicas se definen un conjunto de métricas para las características de un programa:

n=vocabulario
N= longitud del programa
V= volumen
V*= volumen potencial
L= nivel del programa ()
D= dificultad del programa
E=esfuerzo de programación (esfuerzo de desarrollo)
T= tiempo de programación (tiempo de desarrollo)
I= contenido de inteligencia
λ = nivel del lenguaje (una constante para un lenguaje dado)

Ejemplos:

Longitud N se puede estimar como: $N = n1\log_2 n1 + n2\log_2 n2$
El volumen se define como: $V = N\ n1\log_2 (n1 + n2)$ y variará con el lenguaje de programación y representa el volumen de información (en bits) necesarios para especificar un programa.

En general además de aspectos como los mencionados en las medidas anteriores como es el caso del nivel de un programa (una medida de la complejidad del software), la complejidad del software, el volumen mínimo potencial para un algoritmo, el volumen real (número de bits requeridos para especificar un programa), nivel del lenguaje (una constante para un lenguaje dado) y otras características las métricas de Halstead ayudan a medir la calidad del código fuente.

1.6.7 Métricas de McCabe

Otras métricas orientadas a mejorar la calida del código son las métricas de la utilización de la métrica de McCabe ha sido muy popular en el diseño de pruebas desde su creación. Esta métrica es un indicador del número de caminos independientes que existen en un grafo. El propio McCabe definió como un buen criterio de prueba la consecución de la ejecución de un conjunto de caminos independientes, lo que implica probar un número de caminos igual al de la métrica.

Un número de estudios de la industria han indicado que un rango V (G) más alto, hay mayor probabilidad de errores.

Estas métricas son las que se utiliza como el método de la complejidad ciclomática dentro de las pruebas de caja blanca.

1.6.8 Otros tipos de métricas

Eficiencia: La cantidad de recursos informáticos y de código necesarios para que un programa realice su función.

Facilidad de mantenimiento: El esfuerzo necesario para localizar y arreglar un error en un programa.

Reusabilidad (capacidad de reutilización): Hasta donde se puede volver a emplear un programa (o partes de un programa) en otras aplicaciones.

Interoperatividad: El esfuerzo necesario para acoplar un sistema con otro.

1.6.9 ¿Cuándo se debe medir?

La medida es una parte esencial para comprender qué afecta a la calidad, oportunidad, utilidad y funcionalidad y en la mejora de los procesos y productos software.

Se debe medir el software:

- Cuando se quiere indicar la calidad del producto.
- Cuando se quiere evaluar la productividad de las personas.
- Cuando se quiere evaluar los beneficios derivados del uso de nuevos métodos y herramientas.
- Cuando se quiere establecer una línea de base para la estimación.
- Cuando se quiere justificar el uso de nuevas herramientas y la necesidad de formación.
- Cuando se quiere caracterizar, evaluar, predecir y mejorar el producto y el proceso.
- Cuando se quiere medir el rendimiento de los proyectos de una empresa.
- Cuando se quieren evaluar las inspecciones del código.
- Cuando se quiere evaluar las actividades de mejora del proceso de software.
- Cuando se quiere administrar el proceso de pruebas para poder tener evidencia al momento de expresar una opinión, y también para comparar diferentes sistemas o procesos.
- Cuando se quieren establecer comparaciones entre productos, procesos.

1.6.10 Ventajas de las métricas

Las métricas permiten especificar en el mundo formal, la correspondencia de un atributo del mundo empírico. Sirven de base a Métodos Cuantitativos de Evaluación o Predicción. Posibilitan gestionar proyectos planificándolos y dándole seguimiento. Constituyen la línea base del proceso de medición, permitiendo una comunicación clara y el uso del proceso consistentemente.

Permiten describir, clasificar y calificar productos. Ayudan a mejorar procesos facilitando comprensión y el control de los mismos. Permiten obtener medidas muy influyentes a la hora de tomar decisiones respecto al producto y al proceso desarrollado. Mejoran la calidad del producto. Incrementan la productividad del equipo de desarrollo. Mejoran la estimación y planificación del proyecto. Mejoran la gestión del proyecto. Mejoran la cultura de calidad de la compañía. Mejoran la satisfacción del cliente. Incrementan la visibilidad del proceso software. Ayudan a enfocar sobre las áreas de problema mediante el análisis de tendencias. La supervisión de las actividades de mejora del proceso ayuda a identificar lo que funciona y lo que no funciona. La medición por sí misma no mejora el proceso, pero la visibilidad permite profundizar en la

planificación, control, gestión y mejora. Sus datos históricos ayudan a predecir y planear mientras que actuales frente a los planificados ayudan a comunicar el progreso y soportan la toma de decisiones

Sus resultados sirven de base para tomar importantes decisiones en el negocio y para establecer comparaciones entre productos, y métodos.

1.6.11 Cuáles son los costes de no medir

- Incapacidad para estimar/planificar de forma realista determinar el progreso evaluar la calidad reconocer las oportunidades de mejora reconocer mejoras.
- Pérdida de posición competitiva.
- No permiten controlar qué es lo que ocurre en los proyectos.
- No permiten mejorar los procesos y los productos.
- No se facilita la explicación de decisiones de rechazo para aceptar contribuciones, cuando el código no cumple con unos mínimos requisitos.
- Se pierde la visibilidad sobre el estado y profundidad del problema de manera más rápida y fiable.

A través de las métricas, mediante el análisis de los resultados de las distintas pruebas, se pueden ver las tendencias en seguridad del desarrollo y saber si es necesaria más formación para los integrantes del mismo o si algún proceso no se está realizando de forma correcta y determinar cuales son los puntos fuertes y débiles en la implantación del desarrollo seguro.

Las métricas dinámicas ayudan a valorar la eficiencia y la fiabilidad de un programa mientras que las métricas estáticas ayudan a valorar la complejidad.

1.6.12 Desventajas de las métricas

Muy pocas métricas han sobrevivido a la fase de definición y se usan en la industria. Esto se debe a múltiples problemas, entre ellos:

- Las métricas no se definen siempre en un contexto en el que el objetivo de interés industrial que se pretende alcanzar. Incluso si el objetivo es explícito, las hipótesis experimentales a menudo no se hacen explícitas.
- Las definiciones de métricas no siempre tienen en cuenta el entorno o contexto en el que serán aplicadas.
- No siempre es posible realizar una validación teórica adecuada de la métrica porque el atributo que se quiere medir no siempre está bien definido.
- Un gran número de métricas nunca se ha validado empíricamente.
- Las medidas no son absolutas, simplemente proporcionan comprensión (conocimiento profundo) del proceso software.
- La Medición no puede identificar, explicar, o predecir todo.
- La mayoría de los resultados requieren más de una medida para caracterizar y comprender.
- La medición no tiene valor a menos que se comprenda con la globalidad del proceso software.
- Medir implica varios usuarios en diversos niveles por toda la organización.
- La métrica no puede interpretar por sí sola un concepto medible (Necesidad de Indicadores).
- La comparación directa de datos de programas deberá evitarse porque dos programas no son iguales.
- Las evaluaciones basadas en medición son solo tan buenas como la oportunidad, consistencia y precisión de los datos de entrada.

Esta situación desventajosa que poseen las métricas, ha conducido frecuentemente a cierto grado de ambigüedad en las definiciones, propiedades y asunciones de las métricas, haciendo que el uso de las mismas sea difícil, la interpretación peligrosa y los resultados contradictorios.

1.7 Revisiones de código

Las técnicas de Evaluación estática de artefactos del desarrollo se les conoce de modo genérico por **Revisiones**. Las revisiones pretenden detectar manualmente defectos en cualquier producto del desarrollo. Por manualmente entiéndase, que el producto en cuestión (sea requisito, diseño, código,) está

impreso en papel y los revisores están analizando ese producto mediante la lectura del mismo, sin ejecutarlo.

1.7.1 ¿Qué son las Revisiones?

Reunión formal donde se analizan de forma estructurada los resultados (parciales o finales) de un proyecto software para evaluar el proceso y producto software.

Existen varios tipos de revisiones, dependiendo de qué se busca y cómo se analiza ese producto.

Se pueden distinguir:

Revisiones informales: también llamadas inadecuadamente sólo Revisiones (lo cual genera confusión con el nombre genérico de todas estas técnicas). Las Revisiones Informales no dejan de ser un intercambio de opiniones entre los participantes.

Revisiones formales o Inspecciones. En las Revisiones Formales, los participantes son responsables de la fiabilidad de la evaluación, y generan un informe que refleja el acto de la revisión. Por tanto, sólo se considera como técnica de evaluación las revisiones formales, puesto que las informales se pueden considerar un antepasado poco evolucionado de esta misma técnica.

Walkthrough. Es una revisión que consiste en simular la ejecución de casos de prueba para el programa que se está evaluando. No existe traducción exacta en español y a menudo se usa el término en inglés. Quizás la mejor traducción porque ilustra muy bien la idea es *Recorrido*. De hecho, con los walkthrough se recorre el programa imitando lo que haría la computadora.

Auditorías. Las auditorías contrastan los artefactos generados durante el desarrollo con estándares, generales o de la organización. Típicamente pretenden comprobar formatos de documentos, inclusión de toda la información necesaria. Es decir, no se tratan de comprobaciones técnicas, sino de gestión o administración del proyecto.

1.7.2 Revisiones técnicas y de gestión

Por otro lado, según el objeto que se revise, se suele diferenciar entre las revisiones con orientación técnica y las revisiones orientadas a la gestión (también conocidas como revisiones de proyecto).

Las revisiones técnicas más comunes son:

- Revisión de la especificación de requisitos.
- Revisión del diseño.
- Revisión del código.
- Revisión de las pruebas.
- Revisión del manual de usuario.

1.7.3 Inspecciones de código

Para hablar de inspecciones de código es necesario saber a que se le llama control de la calidad que es una serie de revisiones, y pruebas utilizados a los largo del ciclo de desarrollo para asegurar que cada producto cumple con los requisitos que le han sido asignados.

En este marco se pueden ver a las inspecciones como una implementación de las revisiones formales del software las cuales representan un filtro para el proceso de ingeniería de software, éstas se aplican en varios momentos del desarrollo y sirven para detectar defectos que pueden así ser eliminados.

1.7.3.1 Necesidad de las Revisiones

El trabajo técnico necesita ser revisado porque lo realizan personas y errar es humano. La segunda razón por la que se necesitan las revisiones técnicas es porque, aunque la gente es buena descubriendo algunos de sus propios errores, algunas clases de errores se le pasan más fácilmente al que los origina que a otras personas.

1.7.4 ¿Qué son las Inspecciones?

Las inspecciones de software son un método de análisis estático para verificar y validar un producto software manualmente. Los términos Inspecciones y Revisiones se emplean a menudo como sinónimos. Sin embargo, como ya se ha visto, este uso intercambiable no es correcto.

Las Inspecciones son un proceso bien definido y disciplinado donde un equipo de personas calificadas, Analizan un producto software, usando una técnica de lectura con el propósito de detectar defectos. El objetivo principal de una inspección es detectar faltas antes de que la fase de prueba comience. Cualquier desviación de una propiedad de calidad predefinida es considerada un defecto.

1.7.4.1 Objetivos de las inspecciones de código

- Incrementar la productividad.
- Detectar antes los errores.
- Mejorar la calidad del Software.
- Mantener informados a otros miembros del equipo sobre desarrollos de otros grupos.
- Marcar la finalización de una etapa del desarrollo.
- Producir software mantenible.

Las inspecciones tienen como salida:

- Documento con la descripción del producto revisado y los resultados de la revisión.
- Defectos encontrados en el producto revisado.
- Áreas problemáticas.
- Recomendaciones sobre posibles mejoras.

1.7.4.2 ¿Para qué sirven las inspecciones?

Las inspecciones sirven para detectar desviaciones respecto a las especificaciones de calidad.

Las inspecciones de código se realizan con el objetivo de detectar e identificar anomalías de software, incluyendo errores y desviaciones de estándares y especificaciones. Normalmente se realizan a partir de la solicitud del emprendedor o como plan de respuesta a un cierre de pruebas de un producto.

Además de ayudar a descubrir no conformidades en el código fuente, las inspecciones se usan para enseñar a evitar defectos sistemáticos y los beneficios de la utilización de estándares.

Como parte del alcance de la actividad de inspección de código, se identifica un conjunto de 10 criterios, que se evalúan y sobre los cuales se reportan los hallazgos y se generan las recomendaciones a los grupos de desarrollo:

- Lógica de programación
- Estándar de codificación
- Documentación de código (headers (encabezados), comments (comentarios))
- Importar
- Inicialización
- Parámetros de llamada a métodos
- Uso de estructuras de anidamiento
- Modularidad
- Mecanismo reutilización de código
- Manejo de errores

1.7.5 El Proceso de Inspección

Para aprender a realizar inspecciones es necesario conocer el proceso que debe seguirse y luego las técnicas de lectura.

Las Inspecciones constan de dos partes: Primero, la comprensión del artefacto que se inspecciona; Y en segundo lugar, la búsqueda de faltas en dicho artefacto. Más concretamente, una inspección tiene cuatro fases principales:

1. *Inicio:* El objetivo es preparar la inspección y proporcionar la información que se necesita sobre el artefacto para realizar la inspección.

2. *Detección de defectos:* Cada miembro del equipo realiza individualmente la lectura del material, compresión del artefacto a revisar y la detección de faltas. Las Técnicas de Lectura ayudan en esta etapa al inspector tanto a comprender el artefacto como a detectar faltas. Basándose en las faltas detectadas cada miembro debe realizar una estimación subjetiva del número de faltas remanentes en el artefacto.

3. *Colección de defectos:* El registro de las faltas encontradas por cada miembro del equipo es compilado en un solo documento que servirá de base a la discusión sobre faltas que se realizará en grupo. Utilizando como base las faltas comunes encontradas por los distintos inspectores se puede realizar una estimación objetiva del número de faltas remanentes. En la reunión se discutirá si las faltas detectadas son falsos positivos (faltas que algún inspector cree que son defectos pero que en realidad no lo son) y se intentará encontrar más faltas ayudados por la sinergia del grupo.

4. *Corrección y seguimiento:* El autor del artefacto inspeccionado debe corregir las faltas encontradas e informar de las correcciones realizadas a modo de seguimiento.

1.7.6 Relaciones con otras Buenas Prácticas de codificación

1.7.6.1 Revisiones de código y estándares de codificación

Aunque la legibilidad y la mantenibilidad son el resultado de muchos factores, una faceta del desarrollo de software en la que todos los programadores influyen especialmente es en la técnica de codificación. El mejor método para asegurarse de que un equipo de programadores mantenga un código de calidad es establecer un estándar de codificación sobre el que se efectuarán luego revisiones del código de rutinas.

1.7.6.2 Métricas en Inspecciones

El proceso de inspección puede ser medido para analizar distintos aspectos del proceso (planificación, monitoreo, control, mejora, etc.) y poder maximizar su eficacia así como corregir posibles desvíos que puedan producirse durante la inspección.

En nuestra opinión las mediciones deben llevarse a cabo para poder formar una base de datos con los distintos proyectos con el fin de utilizarla a la hora de planificar nuevas inspecciones. Tomando como base las métricas propuestas por Jack Barnard (se utilizan los siguientes indicadores:

- Total de líneas no comentadas el código fuente inspeccionado.
- Promedio de líneas de código inspeccionado.
- Tasa de preparación promedio.
- Tasa de inspección promedio.
- Promedio de esfuerzo por KLOC.
- Promedio de esfuerzo por falla detectada.
- Promedio de fallas detectadas por KLOC.
- Este es un indicador claro de la calidad del código.
- Porcentaje de reinspecciones.
- Eficiencia en la remoción de defecto.

Además sería útil medir, la cantidad promedio de productos de trabajo inspeccionados por cada inspector, así como catalogar la complejidad de los productos de trabajo a inspeccionar, ya que estos dos valores darían una visión mas clara sobre la productividad de la inspección y un parámetro importante a tener en cuenta para la planificación de futuras inspecciones.

1.7.7 Recomendaciones con respecto al equipo de inspección

No se debe descuidar la comunicación que debe existir entre los inspectores y el equipo de desarrollo. Se debe tener en cuenta aspectos como la forma en que se comunican los defectos que existan en el software, ya que por una reacción normal el autor del "producto de trabajo" este intentará justificarlo y muchas veces esa justificación se desvía de su objetivo principal si el autor se siente "atacado" por el inspector.

Se deberá seleccionar cuidadosamente al grupo de inspección, éste deberá ser "respetado" por el equipo de desarrollo en cuanto a sus conocimientos profesionales y del proyecto ya que de no ser así esto será sin dudas una fuente de conflicto permanente.

1.7.8 Ventajas de las Inspecciones de código

Para señalar las ventajas que pueden brindar las inspecciones recordar su objetivo que dicho de otra forma es la implementación de un proceso formal de revisión detallada del producto por parte de pares y un moderador, con el propósito de encontrar defectos en una etapa muy temprana del desarrollo del producto.

Como beneficios básicos es importante destacar que las inspecciones permiten:

- Encontrar errores lo más temprano posible en el ciclo de desarrollo.
- Asegurar que todos los participantes están de acuerdo en la parte técnica del trabajo.
- Verificar que el trabajo cumple con los criterios preestablecidos.
- Completar formalmente una tarea técnica.
- Suministrar información sobre el producto y el proceso de inspección.

Además se pueden destacar otras ventajas que serían secundarias y es que ellas también permiten:

- Asegurar que las personas asociadas estén familiarizadas técnicamente con el producto.
- Ayudar a crear un equipo técnico efectivo.
- Ayudar a utilizar los mejores talentos de la organización.
- Ayudar a los participantes a desarrollar sus habilidades como revisores.

1.8 Patrones

En los últimos años los patrones crecen en aceptación, extendiéndose a otras muchas áreas dentro de la construcción y el mantenimiento de sistemas software (entornos Web, tiempo real, etc). Su utilización, si bien aún le queda mucho camino por recorrer, comienza a tener suficiente madurez, así como la incorporación de esta técnica al soporte automatizado.

1.8.1 Definición de Patrón

Cada patrón es una regla de tres partes, que expresa la relación entre cierto contexto, cierto sistema de fuerzas que ocurren repetidamente en ese contexto y cierta configuración software que permite a estas fuerzas resolverse a sí mismas"(ALEXANDER 1993)

1.8.2 Características de los patrones.

Solucionar un problema: Los patrones capturan soluciones, no sólo principios o estrategias abstractas.

Ser un concepto probado: capturan soluciones demostradas, no teorías o especulaciones.

La solución no es obvia: los mejores patrones generan una solución a un problema de forma indirecta.

Describe participantes y relaciones entre ellos: describen módulos, estructuras del sistema y mecanismos complejos.

El patrón tiene un componente humano significativo: todo software proporciona a los seres humanos confort y calidad de vida (estética y utilidad).

- *Patrones de arquitectura:* se expresa una organización o esquema estructural fundamental para sistemas software. Proporciona un conjunto de subsistemas predefinidos y sus responsabilidades.
- *Patrones de diseño:* proporciona esquemas para refinar subsistemas o componentes de un sistema.
- *Patrones de programación:* describe la implementación de aspectos de componentes.
- *Patrones de análisis:* prácticas que aseguran la consecución de un buen modelo de un problema y su solución.
- *Patrones organizacionales:* describen la estructura y prácticas de las organizaciones humanas.

1.8.3 Patrones de diseño

Un patrón de diseño es:

- Una solución estándar para un problema común de programación.
- Una técnica para flexibilizar el código haciéndolo satisfacer ciertos criterios.
- Un proyecto o estructura de implementación que logra una finalidad determinada.
- Un lenguaje de programación de alto nivel
- Una manera más práctica de describir ciertos aspectos de la organización de un programa.
- Conexiones entre componentes de programas.
- La forma de un diagrama de objeto o de un modelo de objeto.

(Patrones de Diseño 2001)

“Un patrón es una descripción de objetos que se comunican y clases a la medida para resolver un problema de diseño general en un contexto particular“. (GAMMA 1995)

Cada patrón es una regla de tres partes, la cual expresa una relación entre un cierto contexto, un problema y una solución. El patrón es una cosa que tiene su lugar en el mundo y al mismo tiempo, la regla que dice cómo crear esa cosa y cuándo se debe crear. Es al mismo tiempo una cosa y un proceso; al mismo tiempo una descripción de una cosa que tiene vida y una descripción del proceso que la generó” (ALEXANDER 1993)

Si bien, no existe una definición única, todas expresan una conceptualización parecida, relacionando tres ideas fundamentales: problema – contexto – solución. No obstante, es fácil confundir el concepto de patrón con otros similares. Y podrían definirse muchos más sobre este tema, sin olvidar antes, las características que no deben tener, entre las que se pueden mencionar:

- Invenciones, teoría o ideas no probadas.
- Soluciones que sólo han funcionado una vez.
- Principios abstractos o heurísticos.
- Aplicaciones universales para cualquier contexto.

1.8.4 Objetivos de los Patrones de diseño

El principal objetivo de los patrones de diseño es capturar buenas prácticas que permitan mejorar la calidad del diseño de sistemas, determinando objetos que soporten roles útiles en un contexto específico, encapsulando complejidad, y haciéndolo más flexible.

Los patrones de diseño describen una estructura a la cual muchas veces se recurre para resolver problemas de diseño. Estos no dependen del lenguaje de implementación y permiten resolver problemas complejos, direccionando la cooperación efectiva entre componentes.

1.8.5 Elementos que caracterizan a un Patrón de diseño

Un patrón de diseño tiene cuatro elementos característicos:

- El nombre del patrón, describe el problema de diseño, su solución, y consecuencias en una o dos palabras. Tener un vocabulario de patrones nos permite hablar sobre ellos.
- El problema describe cuándo aplicar el patrón. Se explica el problema y su contexto. Puede describir estructuras de clases u objetos que son sintomáticas de un diseño inflexible. Se incluye una lista de condiciones.
- La solución describe los elementos que forma el diseño, sus relaciones, responsabilidades y colaboraciones. No se describe un diseño particular. Un patrón es una plantilla.
- Las consecuencias son los resultados de aplicar el patrón

1.8.6 Descripción de los Patrones de diseño

- Nombre
- Objetivo (Problema)
- Contexto
- Aplicabilidad (Fuerzas)
- Solución
- Consecuencias
- Implementación
- Usos en el API de Java
- Código del ejemplo
- Patrones relacionados

1.8.7 Cualidades de los Patrones de diseño

Encapsulamiento y abstracción. Cada patrón encapsula un problema bien definido y su solución en un dominio particular.

Extensión y variabilidad: Cada patrón debería ser abierto por extensión o parametrización por otros patrones, de tal forma que pueden aplicarse juntos para solucionar un gran problema.

Generatividad y composición: Cada patrón una vez aplicado genera un contexto resultante, el que concuerda con el contexto inicial de uno o más de uno de los patrones del catálogo.

Equilibrio: Cada patrón debe realizar algún tipo de balance entre sus efectos y restricciones.

1.8.8 Clasificación de los Patrones de diseño

Los patrones de diseño se pueden clasificar teniendo en cuenta
Respecto a su propósito:

- *Creacionales:* Resuelven problemas relativos a la creación de objetos.
- *Estructurales:* Resuelven problemas relativos a la composición de objetos.
- *De Comportamiento:* Resuelven problemas relativos a la interacción entre objetos.

Respecto a su ámbito:

- *Clases:* Relaciones estáticas entre clases.
- *Objetos:* Relaciones dinámicas entre objetos.

Como ejemplo de patrones de diseño se pueden mencionar los patrones de GOF atendiendo a su propósito y su ámbito.

- *Creacionales y clases:* Método Fábrica.
- *Creacionales y objeto:* Fábrica Abstracta, Constructor, Prototipo, Singleton.
- *Estructural y clases:* Adaptador.
- *Estructural y objeto:* Adaptador, Puente, Compuesto, Decorador, Fachada, Peso Mosca, Apoderado.
- *Comportamiento y Clases:* Intérprete, Método, Plantilla.
- *Comportamiento y objetos:* Cadena de Responsabilidad, Comando, Iterador, Mediador, Memento, Observador, Estado, Estrategia y Visitante.

1.8.8.1 Descripción de los Patrones creacionales

- *Fábrica Abstracta:* Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar su clase concreta.
- *Constructor:* Permite a un objeto construir un objeto complejo especificando sólo su tipo y contenido.
- *Método Fábrica*: Define una interfaz para crear un objeto dejando a las subclases decidir el tipo específico al que pertenecen.
- *Prototipo:* Permite a un objeto crear objetos personalizados sin conocer su clase exacta a los detalles de cómo crearlos.
- *Singleton:* Garantiza que solamente se crea una instancia de la clase y provee un punto de acceso global a él.

1.8.8.2 Descripción Patrones estructurales

- *Adaptador:* convierte la interfaz que ofrece una clase en otra esperada por los clientes.
- *Puente:* desacopla una abstracción de su implementación y les permite variar independientemente.
- *Compuesto:* permite gestionar objetos complejos e individuales de forma uniforme.
- *Decorador:* extiende la funcionalidad de un objeto dinámicamente de tal modo que es transparente a sus clientes.
- *Fachada:* simplifica los accesos a un conjunto de objetos relacionados proporcionando un objeto de comunicación.
- *Peso Mosca:* usa la compartición para dar soporte a un gran número de objetos de grano fino de forma eficiente.
- *Proxy:* proporciona un objeto con el que controlamos el acceso a otro objeto.

1.8.8.3 Descripción Patrones de comportamiento

- *Cadena de responsabilidad:* evita el acoplamiento entre quien envía una petición y el receptor de la misma.

- *Comando:* encapsula una petición de un comando como un objeto.
- *Interprete:* dado un lenguaje define una representación para su gramática y permite interpretar sus sentencias.
- *Iterador:* acceso secuencial a los elementos de una colección.
- *Mediador:* define una comunicación simplificada entre clases.
- *Memento:* captura y restaura un estado interno de un objeto.
- *Observador:* una forma de notificar cambios a diferentes clases dependientes.
- *Estado:* modifica el comportamiento de un objeto cuando su estado interno cambia.
- *Estrategia:* define una familia de algoritmos, encapsula cada uno y los hace intercambiables.
- *Método de Template:* define un esqueleto de algoritmo y delega partes concretas de un algoritmo a las subclases.
- *Visitante:* representa una operación que será realizada sobre los elementos de una estructura de objetos, permitiendo definir nuevas operaciones sin cambiar las clases de los elementos sobre los que opera.

Si es importante la descripción de un patrón no lo es menos la forma de agruparlos. Existen varias alternativas, siendo este punto uno de los que más problemas derivan a la hora de explotar el uso de patrones. Así, se distinguen principalmente:

Catálogos de Patrones: colecciones de patrones relacionados (quizá de manera pobre o informal). Se suelen subdividir en alguna categoría, cubrir un problema pequeño y pueden incluir alguna referencia entre ellos. Estos añaden una cantidad muy pequeña de estructura y organización a la colección de patrones.(BUSCHMANM 1996)

Sistemas de Patrones: conjuntos cohesivos de patrones relacionados que trabajan juntos para soportar la construcción y evolución de una arquitectura en su totalidad, añadiendo una profunda estructura, una rica interacción entre patrones y uniformidad a la colección de patrones.(BUSCHMANM 1996)

Lenguajes de Patrones: Colección estructurada de patrones que se construyen uno sobre otro para transformar necesidades y restricciones dentro de una arquitectura. Proporcionando un conjunto de patrones que resuelven problemas en un dominio

específico. Los lenguajes documentan las relaciones entre patrones. Son una colección de patrones que forman un vocabulario para comprender y comunicar ideas. Generalmente se muestran como grafos cuyos nodos son patrones y cuyas aristas representan las relaciones semánticas entre los mismos.

1.8.9 ¿Cuándo utilizar Patrones de diseño?

La primera regla de los patrones de diseño coincide con la primera regla de la optimización: retrasar. Del mismo modo que no es aconsejable optimizar prematuramente, no se deben utilizar patrones de diseño antes de tiempo. Seguramente será mejor implementar algo primero y asegurarse de que funciona, para luego utilizar el patrón de diseño para mejorar las flaquezas; esto es cierto, sobre todo, cuando aún no ha identificado todos los detalles del proyecto (si comprende totalmente el dominio y el problema, tal vez sea razonable utilizar patrones desde el principio, de igual modo que tiene sentido utilizar los algoritmos más eficientes desde el comienzo en algunas aplicaciones).

Los patrones de diseño pueden incrementar o disminuir la capacidad de comprensión de un diseño o de una implementación, disminuirla al añadir accesos indirectos o aumentar la cantidad de código, disminuirla al regular la modularidad, separar mejor los conceptos y simplificar la descripción. Una vez que se aprenda el vocabulario de los patrones de diseño le será más fácil y más rápido comunicarse con otros programadores que lo conozca también.

1.8.10 Ventajas de los Patrones de diseño

Los patrones de diseño han dado la posibilidad de organizar las clases en estructuras comunes y bien probadas modificando el sistema para mejorar su flexibilidad y extensibilidad, incrementando la facilidad para adaptar el software a los cambios de especificación e incrementando su posible reutilización y además:

- Facilitan la localización de los objetos que formarán el sistema.
- Facilitan la determinación de la granularidad adecuada.
- Especifican interfaces para las clases.

- Especifican implementaciones (al menos parciales).
- Facilitan el aprendizaje y la comunicación entre programadores.

1.9 Estándares de Codificación

Cada programador sigue cierta metodología al desarrollar una aplicación, mantienen ciertos lineamientos de ingeniería de software, pero tienen diferentes requerimientos los cuales siguen un tronco común, que es el de identificar lo que el sistema debe hacer y las restricciones que este debe presentar. Muchas son las vías utilizadas y entre ellas una de las más conocidas son los estándares de codificación.

1.9.1 ¿Qué es un Estándar?

Un estándar, de acuerdo a la organización ISO, se define como que "contribuye para hacer la vida más fácil, y para incrementar la confiabilidad y efectividad de los bienes y servicios que utilizamos". Adicionalmente, la ISO agrega, “acuerdos documentados que contienen especificaciones técnicas u otros criterios, para ser utilizados constantemente como reglas, lineamientos o definiciones de características, para asegurar que materiales, productos, procesos y servicios son adecuados para sus propósitos”.

Por su parte el BSI (British Standard Institute), describe a un estándar como “una especificación publicada que establece un lenguaje común, y contiene una especificación técnica u otro criterio, que está diseñado para ser usado constantemente, como una regla, un lineamiento o una definición”.

Los estándares no son leyes, son documentos que definen características (dimensiones, colores, aspectos de seguridad) de productos, servicios o procesos de acuerdo a criterios técnicos o tecnológicos que describen el estado del arte.

En resumen los estándares son:

- Reglas que se siguen.
- Una base de comparación.
- Un principio para juzgar cuan bueno es algo.

- Una medida de la calidad, cantidad o nivel.
- Un consenso de opiniones entre individuos, grupos u organizaciones.
- Son obligatorios o voluntarios.

1.9.2 ¿Qué es un Estándar de codificación?

Los estándares de codificación pueden definirse como reglas específicas a una lengua que reducen perceptiblemente el riesgo de que los desarrolladores introduzcan errores, comprendiendo los aspectos de la generación de código y repercutiendo directamente en la legibilidad y la extensibilidad de cualquier proyecto de Software y permitiendo su mantenibilidad.

A los estándares de codificación para programación también se le conoce como estándares de programación o como convenciones de código, que no son más que ciertas reglas de notación y nomenclatura que se usan y se siguen durante la fase de implementación (codificación) de una aplicación.

Los estándares de codificación son reglas específicas de cada lenguaje de programación, cuyo cumplimiento reduce de forma significativa, la posibilidad de cometer errores no detectados por los compiladores. Al implementar y verificar el cumplimiento de estos estándares de codificación, se evitan los errores en la introducción del código, reduciendo el tiempo y coste de las actividades de depuración y pruebas necesarias para la detección y corrección de los mismos.

Una definición más especifica, pero que a la vez abarca los aspectos fundamentales, que caracterizan un estándar de codificación para la programación, es la que resulta, de la extracción de algunos de los elementos mencionados en las definiciones anteriores.

Estándar de codificación:

Un conjunto de reglas de notación y nomenclatura, específicas de cada lenguaje de programación, que se usan y se siguen durante la fase de implementación (codificación) de una aplicación y reducen perceptiblemente el riesgo de que los desarrolladores introduzcan errores que no son detectados por lo compiladores, reduciendo el tiempo y

coste de las actividades de depuración y pruebas necesarias para la detección y corrección de los mismos.

1.9.3 Objetivos de los estándares de codificación

El objetivo del estándar es ofrecer un conjunto de guías que permitan lograr una unificación de conceptos, de esquema de trabajo y de presentación y organización del código para proyectos de software que se vayan a desarrollar, ya sea por parte de un solo programador o por un equipo de ellos; específicamente, se busca:

- Ayudar a escribir programas modularizados de tal manera que siempre se trate de mantener un grado bajo de acoplamiento y un grado alto de cohesión en ellos, de manera que se facilite la reutilización de los módulos, al tiempo que estos sean auto-contenidos funcionalmente.

- Que los proyectos realizados tengan una misma filosofía de estructuración, organización y presentación y esta permita que su desarrollo pueda llevarse a cabo por varias personas trabajando independientemente.

- Agilizar el desarrollo de los proyectos automatizando tanto el proceso de obtención del ejecutable con una herramienta, que permita reducir el tiempo empleado en las fases iterativas de compilación y pruebas, como algunas labores administrativas.

- Que los programadores no pierdan tiempo en la búsqueda (personal o grupal) de cómo presentar, organizar y documentar el código.

1.9.4 ¿Qué comprende un Estándar de codificación?

Un estándar de codificación completo comprende todos los aspectos de la generación de código. Los programadores deben implementar un estándar de forma prudente, éste debe tender a ser práctico. El código fuente debe reflejar un estilo armonioso, como si un mismo programador hubiera escrito todo el código de una sola vez.

Con un estándar de codificación no solo se busca definir la nomenclatura de las variables, objetos, métodos y funciones, sino que también tiene que ver con el orden y legibilidad del código escrito. Siguiendo esta idea, se pueden definir 3 partes principales dentro de un estándar de programación:

- *Convención de nomenclatura*: Cómo nombrar variables, funciones, métodos.
- *Convenciones de legibilidad de código:* Cómo indentar el código.
- *Convenciones de documentación:* Cómo establecer comentarios, archivos de ayuda.

1.9.4.1 Convenciones de legibilidad del código

La legibilidad del código es un aspecto al que es necesario prestar una atención especial. Cualquiera que sea el proyecto, los miembros del mismo pasarían mucho tiempo escribiendo, leyendo y revisando el código fuente. Se calcula que un programador pasa la mitad del tiempo tratando de entender qué hace el código. Este trabajo se logra hacer más fácil y sencillo siguiendo las convenciones de nomenclatura de los estándares y a la vez haciendo el código legible y bien documentado.

1.9.4.2 Documentación del código

Esto tiene que ver con los comentarios explicatorios y aclaratorios que se establecen en el código para futuras referencias. Muchas veces se puede olvidar la finalidad de un proceso con complicada algoritmia. Los comentarios ayudan al programador a recordar los puntos claves de cada parte del código (sobre todo cuando ha pasado un tiempo largo de haberlo codificado). Además sirve para que los demás miembros del equipo de desarrollo puedan entender también dichos bloques de código.

1.9.5 Principales estilos de programación

- *Notación húngara:*

Esta convención se basa en definir prefijos para cada tipo de datos y según el ámbito de las variables. También es conocida como notación: REDDICK.

La idea de este tipo de notación ¡ es dar mayor información al nombre de la variable, método o función ¡ definiendo en ella un prefijo que identifique su tipo de dato y ámbito Un ejemplo de ello es el que a continuación se ilustra:

iIntEdad: según la definición puede verse que esta variable es de tipo INTEGER (entero) y que representa la edad de alguna persona.

prStrNombre: En este caso la variable tiene el prefijo: "prInt", lo que significa que es un parámetro por referencia (pr) de tipo STRING que representa un nombre

gStrConexion: En este caso se trata de una variable global (g) de tipo STRING (cadena) que representa cierta información de conexión.

Notación PascalCasing:

Pascal-Casing es como la notación húngara pero sin prefijos. En este caso, los identificadores y nombres de variables, métodos y funciones están compuestos por múltiples palabras juntas, iniciando cada palabra con letra mayúscula. Por ejemplo:

DoSomething: Este nombre de método está compuesto por dos palabras, ambas iniciando con letra mayúscula.

- *Notación camelCasing (Notación de Camello)*

Camel-Casing es común en Java. Es parecido al Pascal-Casing con la excepción que la letra inicial del identificador no debe estar en mayúscula sino en minúscula. *doSomething*: Este nombre de método está compuesto por dos palabras, la primera todo en minúsculas y la segunda iniciando con letra mayúscula

1.9.6 ¿Porqué adoptar los Estándares de codificación?

- Reducen la probabilidad de introducir errores.

- Evitan algunos errores ocultados o inesperados.
- Evitan los bugs que se despliegan en el software.
- Hacen el código más uniforme y más fácil de leer.
- Hacen más fácil la mantención del software.
- Aumentan la robustez y la confiabilidad.
- Ayudan en la revisar el código entregado.

1.9.7 Ventajas de los estándares de codificación

Establecer un estándar de codificación y nomenclatura puede tomar mucho tiempo. De ahí la necesidad de encontrar o elaborar aquel que se ajuste más al proyecto en el que se trabaje o de crear algún mecanismo que proporcione elegir el estándar más óptimo. Pero una vez establecido un estándar, los beneficios son muchos:

- Los nombres de variables serían mnemotécnicos con lo que se podría saber el tipo de dato de cada variable con sólo ver el nombre de la variable.
- Los nombres de variables serían sugestivos, de tal forma que se podría saber el uso y finalidad de dicha variable o función fácilmente con solo ver el nombre de la variable.
- La decisión de poner un nombre a una variable o función sería mecánica y automática, puesto que seguiría las reglas definidas por el estándar establecido.
- Permiten el uso de herramientas automáticas de verificación de nomenclaturas.
- Asegurar la legibilidad del código, facilitando el debugging del mismo.
- Proveer una guía para el encargado de mantenimiento/actualización del sistema, con código claro y bien documentado.
- Facilitan la portabilidad entre plataformas y aplicaciones

1.10 Conclusiones

En el desarrollo de este capítulo se estudiaron seis casos particulares de las buenas prácticas para mejorar la codificación, evidenciando la importancia de cada una para obtener código de calidad. Con la elaboración del capítulo se lograron los objetivos de incrementar el conocimiento sobre las buenas prácticas de codificación y conocer sobre su situación en la Universidad, así como la de los estándares de codificación, calidad y calidad del código. Lo que posibilitó conocer cuándo se debe aplicar cada una de las prácticas y cuáles son las que se utilizan para prevenir y/o eliminar errores en el software. Por lo tanto se puede arribar a la conclusión de que es necesario hacer uso de todas las Buenas Prácticas en uno u otro momento del desarrollo de software, con el objetivo de mejorar el código, así como la importancia que tiene fomentar el estudio y seguimiento de las que principalmente, prevengan la introducción de errores y faciliten el trabajo a las demás.

2 Capítulo 2 Propuesta y Validación

2.1 Introducción

En este capítulo, se muestra por qué en el trabajo se le presta mayor atención a los estándares de codificación. Además se hace una selección de estándares de codificación según algunos lenguajes de programación que se utilizan en la Universidad. Se seleccionan los elementos comunes de los estándares y se hace la presentación de la propuesta que da solución al problema científico, la cual consiste en una guía que establece los elementos generales de los estándares de codificación. También se brindan dos listas de chequeo que tienen el objetivo de medir aspectos importantes en la implementación, como es el caso de la selección de un Estándar de codificación y la revisión del código fuente. Finalmente se muestra la validación de la propuesta presentada.

2.2 ¿Por qué se eligieron los estándares de codificación?

Después de haberse hecho un estudio sobre las buenas prácticas de programación, se escogió como práctica a desarrollar en este capítulo, los estándares de codificación.

La selección de esta buena práctica no está basada en que esta sea superior a las demás estudiadas, sino por el hecho de que la aplicación de esta práctica, le facilita el trabajo a las otras, permitiendo una mayor eficiencia a la hora de refactorizar, de hacer las revisiones de código, de aplicar las métricas, los patrones y las pruebas de caja blanca además de ser los que proporcionan uniformidad al código.

Los estándares benefician en una u otra medida la aplicación de las prácticas, debido a que su utilización permite que parezca que el código fue escrito por un mismo programador proporcionando legibilidad al código, claridad de la lógica subyacente, la facilidad para la realización de las pruebas y su mantenimiento y reutilización.

Además facilitan la lectura y la determinación de la estructura del programa, permitiendo saber por simple observación, qué partes del programa se relacionan entre sí, dónde se producen las rupturas principales y cuáles proposiciones están contenidas dentro de cada ciclo o las alternativas de una proposición condicional.

2.3 Necesidad de los estándares de codificación

Independientemente de los algoritmos usados, hay muchas formas y estilos de programar. La legibilidad de un programa es muy importante a veces el programador no le presta la atención que merece.

Los programas, a lo largo de su vida, se van quedando obsoletos debido a cambios en su entorno. Un programa pensado para una determinada actividad, es muy normal tener que modificarlo porque cambie dicha actividad o porque se decida incluirle nuevas posibilidades que antes no estaban previstas. De ahí las múltiples versiones que sacan al mercado las empresas de programación. Además, debido a la dificultad de algunos programas para probarlos exhaustivamente, a veces, se descubren errores cuando el programa lleva funcionando cierto tiempo.

En ocasiones hay que modificar un programa escrito hace cierto tiempo o escrito por otro programador, y es en ese momento cuando salen los problemas de legibilidad de un programa. Para poder modificarlo, primero hay que comprender su funcionamiento, y para facilitar esta tarea el programa debe estar escrito siguiendo unas normas básicas. La tarea de mantenimiento del software (corregir y ampliar los programas) es una de las tareas más arduas del ciclo de vida del software. Por eso, al programar se debe intentar que los programas sean lo más expresivos posibles, para ahorrar tiempo, dinero y trabajo la hora de modificarlos.

Las normas de estilo en programación, son tan importantes que todas las empresas dedicadas a programación en todo el mundo, imponen a sus trabajadores una mínima uniformidad, para facilitar el intercambio de programas y la modificación por cualquier trabajador, sea o no el programador inicial. Por supuesto, cada programa debe ir acompañado de una documentación adicional, que aclare detalladamente cada módulo del programa, objetivos, algoritmos usados, ficheros.

La realidad es que estos problemas se pueden resolver en gran medida tan solo con la utilización de Estándares de codificación por parte del equipo de desarrollo de software.

2.4 Necesidad de una guía

Cada lenguaje de programación define su propio estándar de codificación y existen muchos por lenguajes por lo que siempre que se va afligir un estándar se pierde tiempo en la elección de cuál será el mejor para que todo el equipo de desarrollo trabaje de forma uniforme.

De ahí la necesidad de una guía que permita hacer la selección de Estándares de codificación y permitiendo que se ahorre tiempo y que elija el más óptimo.

2.5 Propuesta de una Guía de Estándar Genérico (GEG)

En este epígrafe se recogen los siete pasos fundamentales que se deben realizar para elaborar una guía de estándar genérico que contenga elementos generales de los Estándares de codificación a los que se les llamará variables de las cuales algunas tendrán valores.

2.5.1 Proceso para obtener la Guía de Estándar Genérico

El proceso de obtención de los elementos comunes que formarán parte de la propuesta consta de siete pasos fundamentales, los cuales se abordarán a continuación y finalizarán con la confección de dicha Guía de Estándar Génerico.

2.5.1.1 Elegir los lenguajes de programación a utilizar

El primer paso es la elección de los lenguajes de programación que aportarán sus Estándares de codificación.
Aquí se eligen los lenguajes a tener en cuenta a la hora de seleccionar los estándares que se estudiarán para la elaboración de la propuesta, pues cada lenguaje de programación define sus propios estándares de codificación. Los lenguajes deben ser definidos en correspondencia del grado en que estos se utilicen en la Institución en la cual se establecerá la Guía de Estándar Genérico.

En este caso, para el estudio de los estándares hubo que seleccionar primero los lenguajes de programación que más se utilizan en la UCI. Para determinar cuales son los más utilizados se consultó el *"Resumen de Arquitecturas, Plataformas y Tecnologías utilizadas en la Producción y las impartidas en la Docencia", documento resultado de un estudio hecho por la dirección de tecnologías y que fue* presentado en el Congreso de

Producción en el mes de abril del 2007. En dicho documento se determinó que en la Universidad se utilizan 18 lenguajes de programación, siendo HTML, PHP, XML y Java los más utilizados y c++ el que más se imparte. A partir de dicho estudio se hizo una selección de los lenguajes que se utilizarían como punto de partida para la búsqueda de los estándares involucrados en el proceso, en este caso se destacan: JAVA, PHP, C#, C++.

2.5.1.2 Determinar los Estándares de codificación a utilizar

Una vez seleccionados los lenguajes, en este paso se elegirán los estándares de codificación con los que se trabajará para la obtención de sus elementos comunes. Estos estándares deberán ser confiables y pertenecer a cualquiera de los lenguajes de programación escogidos.

Para el caso de la propuesta que se presenta en este capítulo, se eligieron varios estándares de codificación específicos lenguajes como PHP, C#, C++, Java y otros generales independientes del lenguaje de programación.

Para PHP:

- Estándar de codificación de Mambo escrito por el equipo de Mambo (2007-03-03).
- Estándar de codificación PEAR (2007-06-03).

Para C#:

- Msdn Técnicas de Codificación del Visual Studio.
- Convenciones de código para el lenguaje de programación C #.
- Estándar de codificación de C# de Vic Hartog and Dennis Domen (2005-05-19).
- Guía de estilo de Codificación en C# por Mike Kruger (versión 3).
- Guía de estilo de codificación por Y10K (2006-10-30).
- Estándar de codificación de C# por Juval Lowy (2007).

Para C++:

- Estándar de codificación de C++ (2000-01-05).
- Estándar de codificación de C++ (2001-07).
- Estándar de codificación de C++ (2007-01-09).
- Guía de estilo de codificación en C++ (2007-04).

Para Java:

- Especificación del Lenguaje Java, de Sun Microsystems, Inc. Sus principales contribuciones por Scott Hommel (1999-04-20).
- Estándar de Codificación para Java de WisDOT por Barry Rowe, Phil Staley y Cindy Williamson (1999-11-08)
- Escribiendo código robusto en Java por Scott W. Ambler (2000-01-15)
- Estándar de Codificación para Java por NEJUG (2002-03)
- Convenciones de Código para el lenguaje de programación JAVA™ traducido por Jesús Pérez Alcalde (2007-01-25)

Otros Generales:

- Estándar codificación DOTNET por Giovanny Fernández (2005-04-30).
- Estándares de codificación de sistemas por la Dirección General del Gobierno Digital de Shubut (2006-10-18).
- Estilo de codificación del núcleo Linux traducido por David Marín Carreño.
- Reglas de Estilo para la Codificación por J. f Díaz

2.5.1.3 Determinar estándar guía

En este paso, después de revisar los estándares elegidos anteriormente, se hace una selección de uno de ellos, el que se considere más útil, con ayuda de algún punto de comparación, que puede ser establecido por la institución o por el propio desarrollador de la propuesta. Dicho estándar se utiliza como guía para la elaboración de la propuesta de estándar genérico utilizándolo como plantilla.

La guía que se determinó fue el Estándar de Convenciones de código para el lenguaje JAVA TM de Microsystem, escogido por su gran prestigio dentro del software libre, y el más utilizado dentro de ese lenguaje, y además por si la Universidad decide emigrar a software libre.

2.5.1.4 Agrupar elementos comunes de los estándares por lenguaje

Una vez revisados todos los estándares por lenguaje, se extraen de ellos los elementos comunes, y se crea un solo estándar que reúna las características coincidentes entre ellos.

Para el caso que se soluciona, se hace una selección de los elementos que componen los estándares anteriormente seleccionados, que pertenecen a un mismo lenguaje, y a partir de ahí se genera un nuevo estándar, resultado de la agrupación de los elementos de dichos elementos. Es decir, se obtendrá un solo estándar para cada lenguaje que reunirá las características coincidentes y más importantes de cada uno de ellos. Para lograr esto se realiza una serie de pasos que se describen a continuación.

Pasos para obtener un estándar por lenguaje: // hasta aki revise ortografia y espacios

1. Traducir los elementos de los estándares a un mismo lenguaje.

Es necesario en este paso trabajar con elementos que sean entendibles, por ello es necesario traducirlos y llevarlos a un mismo idioma. En este caso se tradujeron al idioma español.

2. Seleccionar el estándar más general de cada lenguaje.

Para cada lenguaje se debe seleccionar una guía interna que permita organizar el proceso. El estándar seleccionado en cada lenguaje debe ser el que más características agrupe de los restantes.

3. Elaborar un estándar por cada lenguaje.

Una vez seleccionadas las guías internas por lenguajes se empezarán a comparar dichas guías con los restantes estándares correspondientes a sus lenguajes para obtener de ellos los elementos que pudieran faltarle y que se cumplan para dichos lenguajes. De esta forma se obtendrá un solo estándar por lenguaje a pesar de haberse trabajado con varios, lo cual facilitará el trabajo y lo hará más abarcador.

2.5.1.5 Determinar los indicadores comunes

En este paso se determinan los elementos comunes, a partir de los estándares obtenidos en el paso anterior. Dichos elementos servirán de indicadores para evaluar a los estándares y los elementos que se consideren relevantes (entiéndase por relevante algún aspecto que no necesariamente coincida con todos pero que sí se considere importante y sea parte de ellos) también serán incluidos, pero como atributos. Los indicadores que se determinan a partir de la coincidencia de los elementos comunes entre todos los estándares que ya fueron agrupados.

Como resultado puntual se obtuvieron para los lenguajes JAVA, C++, C#, PHP y otros generales, un total de 14 indicadores, como son Introducción, formato, fichero, Identación y otros más que se verán mas adelante en el paso 6.

2.5.1.6 Comparación de indicadores comunes con el estándar guía

En este paso, que es el último antes de plantear la propuesta, se debe establecer una comparación entre los elementos comunes determinados en los pasos anteriores, y los elementos del estándar que se escogió como guía. Esto permitirá darle un orden lógico a la propuesta y verificar que no quede ningún elemento importante fuera del marco de acción de la nueva propuesta.

2.5.1.7 Plantear la Propuesta

Una vez que se ha transitado por todos los pasos anteriores de forma correcta, ya se está en condiciones de plantear una propuesta de estándar genérico, que incluya los elementos comunes de los estándares anteriormente estudiados y agrupados.

La propuesta incluye un total de 14 indicadores y 61 elementos relevantes, como se muestra a continuación.

Guía de Estándar Genérico (GEG)

1. Introducción

Breve introducción al documento y a los atributos que incluye la misma.

- ***Por qué usar convenciones de código***

Necesidad de utilizar convenciones e importancia de la aplicación de estándares de codificación.

- ***Objetivos***

Objetivos de la Guía de Estándar Genérico (GEG).

- ***Alcance***

Beneficios de la aplicación de la GEG, y condiciones mínimas necesarias para su aplicación.

2. Formato

Definición del indicador Formato y los atributos que incluye.

- ***Sangría***

Estándar de sangría a utilizar para mantener el equilibrio en el código.

- ***Llaves***

Estilo a utilizar para trabajar con llaves.

- ***Diseño entrada y salida (E/S)***

Formatos que se deberán utilizar en la entrada y salida de datos en la interfaz de la aplicación.

- ***Tablas***

Trabajo con tablas.

- ***Paréntesis***

Cuándo y cómo utilizar los paréntesis.

- ***Fuente***

Se establece el tipo de fuente a utilizar en el código

- ***Operadores***

Trabajo con los operadores

- ***Estructura del programa***

Estructura del programa de acuerdo a su propósito.

3. Ficheros

Definición de un fichero y los atributos que incluye.

- ***Extensiones de los Ficheros***

Tabla que contiene el tipo de fichero y la extensión correspondiente al mismo de acuerdo al lenguaje de programación utilizado.

- ***Nombres de ficheros comunes***

Se plantean los nombres de los ficheros más utilizados con los que el software trabaja.

- ***Contenido de los archivos (interfaz e implement)***

Trabajo con archivos

4. Identación

Definición del indicador Identación y sus atributos.

❖ ***Reglas de delimitación***

Trabajo con delimitadores.

❖ ***Longitud de la línea***

Tamaño de línea para trabajar.

❖ ***Ruptura de la línea***

División de líneas grandes en más pequeñas.

5. Comentarios

Definición del indicador comentario y sus atributos.

❖ ***Comentarios de implementación***

Comentarios acerca del código o sobre una implementación en particular.

- ***Comentarios de bloque***

Comentarios para proporcionar descripciones de ficheros, métodos, estructuras de datos y algoritmos.

- ***Comentarios de una línea***

Son comentarios cortos que aparecen en una única línea al nivel del código que siguen.

- ***Comentarios de por detrás***

Son comentarios muy pequeños que aparecen en la misma línea que describen, pero deben ser movidos lo suficientemente lejos para separarlos de las sentencias.

- ***Comentarios de fin de línea***

Puede convertir en comentario una línea completa o una parte de una línea. Difíciles de leer.

❖ ***Comentarios de documentación***

Comentarios para especificar el código, libre de una perspectiva de implementación. Deben ser leídos por desarrolladores que pueden no tener el código fuente a mano.

❖ ***Comentarios especiales***

Comentarios que pueden aparecer en cualquier parte y señalan situaciones especiales dentro del código. Se utilizan para indicar que algo tiene algún error pero funciona.

6. Declaraciones

Definición del indicador Declaración y sus atributos.

- ***Número de declaraciones por línea***

Declaración en una sola línea.

- ***Inicialización***

Inicialización de variables

- ***Declaración de clases e interfaces***

Codificación ordenada de clases e interfaces

7. Sentencias

Definición del indicador Sentencia y sus atributos.

- ***Sentencias simples***

Una sentencia como máximo por línea

- ***Sentencias compuestas***

Lista de sentencias encerradas entre llaves.

- ***Sentencias de retorno***

Sentencias que devuelven valores.

- ***Sentencias if-else, if else-if else***

Evaluar condiciones. Ejecución selectivamente de sentencias basándose en algún criterio.

- ***Sentencias for***

Repetición de un código (una o más sentencias de programación) dependiendo de un contador.

- ***Sentencias While***

Repetición de la ejecución de unas sentencias un número indefinido de veces, siempre que se cumpla una condición.

- ***Sentencias do-while***

Se desconoce el número de veces que se ejecutará el bucle, pero se sabe que se ejecutará por lo menos una vez.

- ***Sentencias switch***

Sentencias condicionalmente basadas en alguna expresión. Utiliza la sentencia **break** para interrumpir el flujo y la sentencia **case** para continuarlo.

- ***Sentencias Try-Catch***

Cuando se quiere capturar y resolver un problema que ha generado una excepción.

8. Espaciado

Definición del indicador Espaciado y sus atributos.

- ***Líneas en blanco***

Separación de secciones de código que están relacionadas lógicamente para mejorar su lectura

- ***Espacios en blanco***

Espacios entre operadores, secciones de código u otros elementos que lo requieran.

9. Convenciones

Definición del indicador Convenciones y sus atributos.

- ***Estilo de Pascal***

Para identificadores de tres o más caracteres capitalizar la primera letra en el identificador y la primera letra de cada palabra concatenada subsecuentemente.

- ***Estilo de camello***

La primera letra de un identificador es minúscula y la primera letra de cada palabra concatenada subsecuentemente se capitaliza.

- ***Mayúsculas***

Para identificadores que consisten en abreviaturas de uno o dos caracteres de largo todas las letras se capitalizan.

- ***Directivas de declaración***

Definición de las abreviaturas más utilizadas.

10. Nomenclatura

Definición del indicador Nomenclatura y sus atributos.

- ***Variables***

Trabajo con variables

- ***Métodos***

Trabajo con métodos

- ***Parámetros***

Trabajo con parámetros

- ***Constantes***

Trabajo con constantes

- ***Clases***

Trabajo con clases

- ***Atributos***

Trabajo con atributos

- ***Propiedades***

Trabajo con propiedades

- ***Interfaces***

Trabajo con interfaces

- ***Funciones***

Trabajo con funciones

- ***Idioma***

Trabajo con el idioma

11. Prácticas de Programación

Definición del indicador Prácticas de Programación y sus atributos.

- ***Tratamiento de Excepciones***

Trabajo con excepciones

- ***Modularización***

Trabajo con módulos

- ***Anidamiento de instrucciones***

Trabajo con niveles de anidamiento de estructuras de control y funciones

- ***Visibilidad***

Definición de elementos públicos, privados, protegidos y package

- ***Inclusión de código***

Incluir código

- ***No números mágicos***

No utilización de números que puedan ser reemplazados por constantes.

12. Ejemplos de Código

Ejemplos de trabajos con el código

13. Recursos

Definición del indicador Recursos y sus atributos.

- ***URLs de Ejemplos***

Para denotar URLs de ejemplo y direcciones de e-mail en la documentación y comentarios,

- ***Libros recomendados***

Libros útiles

- ***Herramientas***

Herramientas útiles

14. Glosario de términos

Listado de términos que puedan dificultar la comprensión del contenido de la propuesta.

2.5.2 Variables para evaluar y controlar los estándares

En este paso se describen y ejemplifican los elementos o indicadores que forman parte de la propuesta. Se conceptualizan y se presentan reglas en cada uno de ellos a tener en cuenta para lograr una buena codificación.

A continuación se explican los 14 indicadores que se exponen en la propuesta y se plantean algunas reglas y valores a tener en cuenta durante su utilización.

2.5.2.1

2.5.2.2 Introducción

El desarrollo de la Guía de Estándar Genérico estuvo encaminado a lograr una herramienta que permitiera uniformar el proceso de selección de Estándares de Codificación en la Universidad de las Ciencias Informáticas., y por ende que estabilizara la codificación y disminuyera la intromisión de errores durante la etapa de implementación.

La Guía tiene 14 indicadores o variables de calidad que incluyen dentro alrededor de 61 atributos y dos herramientas, que consisten en dos listas de chequeo, todos orientados a evaluar los estándares de codificación y el código fuente.

2.5.2.2.1 Por qué usar convenciones de código

Uno de los instrumentos que facilitan la calidad a lo largo del ciclo de vida de un software son los estándares de estilo y codificación.

El uso de los estándares de codificación está llamado a:

- Asegurar la legibilidad del código entre distintos programadores, facilitando la depuración del mismo
- Proveer una guía para el encargado del mantenimiento y la actualización del sistema, con código claro y bien documentado.
- Facilitar la portabilidad entre plataformas y aplicaciones.
- Preveer la introducción de errores y propiciar la aplicación de buenas prácticas.

2.5.2.2.2 Objetivo

El objetivo de esta Guía de Estándar Genérico es ofrecer un conjunto de variables que permitan evaluar y/o controlar los estándares de codificación, logrando una unificación de conceptos, de esquema de trabajo y de presentación y organización del código para proyectos de software que se vayan a desarrollar en cualquier lenguaje, ya sea por parte de un solo programador o por un equipo de ellos.

Además la utilización de la guía, y en específico, de una de sus herramientas de evaluación, posibilita la revisión directa del código fuente, lo cual permite darle seguimiento tanto a los programadores como al trabajo realizado por ellos, esto contribuye a lograr una planificación real del producto y a la determinación de los posibles riesgos que se pueden presentar durante su desarrollo.

2.5.2.2.3 Alcance

Para aplicar la GEG es necesario primeramente tener en cuenta una serie de precondiciones. La guía solo debe ser utilizada para evaluar Estándares de Codificación pertenecientes a los lenguajes PHP, C#, C++ y Java, fuera de ellos podría tener alguna

funcionalidad pero no sería la más óptima. En caso de utilizarla para revisar el código fuente también se requiere que el mismo esté implementado en alguno de los lenguajes anteriormente mencionados.

Si el proyecto no es muy extenso, y no se requiere la utilización de todos los indicadores, la guía puede adaptarse a conveniencia del equipo de programación, disminuyendo su nivel de acción mediante la eliminación de las variables y atributos correspondientes que no sean necesarios.

La guía también puede ser utilizada como estándar a nivel de proyecto. Esto es posible con la asignación de valores a cada uno de los indicadores y atributos que ella incluye. En este caso se establecerían reglas a seguir durante la codificación que guiarían al programador en su trabajo, desde sus inicios.

Con ayuda de la guía es posible evaluar tanto el rendimiento de los Estándares como el del propio programador al evaluar su código. Esto daría una visibilidad del estado de la etapa de implementación en un momento determinado.

2.5.2.3 Formato

El formato hace que la organización lógica del código sea más clara. Si toma el tiempo de comprobar que el código fuente posee un formato coherente y lógico, les resultará de gran utilidad a usted y a otros programadores que tengan que descifrarlo.

2.5.2.3.1 Sangría

- Establecer tamaño estándar de sangría y mantener el equilibrio en las medidas (por ejemplo, ocho espacios corresponde a ocho tabulaciones) y usarlo siempre en todas las líneas de construcción lógica.
- Alinear las secciones de código mediante la sangría predeterminada, tenga en cuenta que ellas definen claramente dónde empieza y acaba un bloque de control.
- Definir sangrías grandes, que permitan ver mejor su funcionalidad y señales cuando se están anidando demasiado las cosas.

2.5.2.3.2 Llaves

- Incluir llaves en los bloques condicionales así sea de una sola sentencia (excepto VB).
- Usar un estilo que no provoque ninguna pérdida de la legibilidad.
- Las llaves siempre indican que la siguiente línea y hasta la llave de cierre se debe indentar al bloque incluído un nivel más arriba.
- Luego del encabezado de una función, sentencia if, else, etc, colocar la llave de apertura al mismo nivel de indentación que la palabra anterior, en la línea inmediatamente abajo de ella. La llave de cierre se coloca a ese mismo nivel de indentación

2.5.2.3.3 Diseño entrada y salida (E/S)

- Hacer la entrada fácil de proveer y la salida autoexplicativa.
- Usar formatos de entrada uniformes.
- Usar formatos de entrada libre (amplios) cuando sea posible.
- Usar entrada autoidentificativa. Permite valores por defecto. Muestra ambos en la salida.
- Ubicar la entrada y la salida en puntos específicos en las subrutinas.
- Hacer la entrada fácil de corregir.

2.5.2.3.4 Tablas

- Poner nombres a tablas en singular. Por ejemplo, usar Trabajador en lugar de Trabajadores.
- No repetir en las columnas de las tablas, el nombre de la tabla. Por ejemplo, evitar un campo llamado EstudianteApellido de una tabla llamada Estudiante
- No incorporar el tipo de datos en el nombre de una columna.

2.5.2.3.5 Paréntesis

- No poner paréntesis pegados a sentencias.
- Poner paréntesis pegados a las funciones.
- No poner paréntesis en valores de retorno cuando no sea necesario.
- Emplear paréntesis para evitar ambigüedades.
- Usar paréntesis generosamente en expresiones que tienen operadores mezclados para evitar problemas de precedencia de operadores.

2.5.2.3.6 Fuente

- Usar un único tipo de letra cuando se publiquen versiones impresas del código fuente.
- Usar siempre el mismo estilo en todo el código fuente.
- Controlar el Código Fuente con ayuda de herramientas.
- Dividir el código fuente de manera lógica entre diferentes archivos.
- Escribir claramente.
- Escribir claramente -no sacrificar claridad por "eficiencia".

2.5.2.3.7 Operadores

- No usar el operador de asignación (=) en un lugar donde se pueda confundir fácilmente con el operador de igualdad (==).
- Si una expresión contiene un operador binario antes de ? el operador ternario?: se debe colocar entre paréntesis.
- Cuando se use el operador coma en la cláusula de inicialización o actualización, evitar la complejidad de utilizar más de tres variables.

2.5.2.3.8 Estructura del programa

- Elegir una representación de datos que haga el programa sencillo.
- Dejar que los datos estructuren el programa.
- Tratar de hacer que la estructura del programa coincida con su propósito.
- Evite albergar múltiples clases en un solo archivo.

2.5.2.4 Ficheros

Conjunto de información (programas o datos) que el ordenador almacena en un disco o cinta de manera diferenciada variando en dependencia del lenguaje de programación. Los ficheros se identifican, y se diferencian por un NOMBRE y, opcionalmente, una EXTENSIÓN.

2.5.2.4.1 Extensiones de los Ficheros

- Especificar en una tabla, el tipo de fichero y la extensión correspondiente al mismo.
- Tener en cuenta para cada lenguaje, en que formato se deben ser guardados los archivos.

2.5.2.4.2 Nombres de ficheros comunes

- Nombrar los ficheros más utilizados por todo el softwares.
- Incluir los ficheros más nombrados independientemente del lenguaje.

2.5.2.4.3 Contenido de los archivos (interfaz e implement)

- Tratar las condiciones de fin de archivo de una manera uniforme.
- Separar las secciones de un fichero con líneas en blanco y comentarios opcionales que identifican cada sección.
- Albergar la interfase/implementación de UNA clase para cada archivo.
- Nombrar al archivo del mismo modo que a la clase (incluída capitalización).

2.5.2.5 Identación

Adentramiento de instrucciones de código para denotar su subordinación a otras instrucciones. Normalmente los niveles de adentramiento están basados en un espaciado fijo y corresponden a las tabulaciones. Por lo general cada nivel de adentramiento es de cuatro espacios aunque se sugieren ocho tabulaciones como mejor opción.

2.5.2.5.1 Reglas de delimitación

- Los delimitadores de funciones (p. ej. las llaves en C) deben estar en líneas independientes para denotar con claridad los límites de las mismas.
- Los delimitadores de estructuras de control tales como los ciclos y las estructuras de selecciones simples y múltiples, deben aparecer en líneas independientes para denotar claramente los límites del bloque de código que estas instrucciones abarcan.
- Limitar a una sola línea los encabezados de estructuras de control, simplificando las expresiones lógicas de la condición o descomponiendo las estructuras de control en otras más simples.

2.5.2.5.2 Longitud de la línea

- Establecer una longitud de línea máxima para el código.
- Siempre que sea posible, no colocar más de una instrucción por línea, a excepción de los bucles.
- Las líneas en un archivo de fuente no deben tener más de 80 caracteres. Se hace esto puesto que algunas impresoras pueden cortar su línea si se imprime su código fuente. Es duro calificar el código que es incompleto de este modo.

2.5.2.5.3 Ruptura de la línea

- Evitar las instrucciones largas. Dividirlas en varias líneas escogiendo puntos de corte que tengan sentido y aplicar una tabulación a todas las líneas después de la primera.
- Cuando se divida una línea en varias, aclarar que el código sigue en la línea de más abajo mediante un operador de concatenación colocado al final de cada línea, y no al principio.

2.5.2.6 Comentarios

Descripciones de código para facilitar información adicional que no es legible en el código mismo. Los comentarios deben contener sólo información que es relevante para la lectura y entendimiento del programa.

2.5.2.6.1 Comentarios de implementación

- Pueden tener cuatro estilos: de bloque, de una sola línea, por detrás y de final de línea.
- Se deben utilizar para comentar el código o para comentar una implementación particular.

2.5.2.6.1.1 Comentarios de bloques

- Pueden ser usados al principio de cada fichero, antes de cada método o en el interior de estos.
- Dentro de una función o método deben estar tabulados al mismo nivel que el código que describen.
- Deben estar precedidos por una línea en blanco para apartarlo del resto del código.

2.5.2.6.1.2 Comentarios de una línea

- Establezca una longitud de línea máxima para los comentarios y el código. Así no tendrá que desplazarse en el editor del código fuente, y conseguirá presentaciones impresas claras.
- Deben ir precedidos de una línea en blanco.
- Facilitan la comprensión del código, sobre todo en procedimientos complejos.

- Cada declaración de variable importante debe incluir un comentario en la misma línea que describa el uso de la variable que se declara.

2.5.2.6.1.3 Comentarios por detrás

- Son muy pequeños.
- Aparecen en la misma línea que describen, pero deben ser movidos lo suficientemente lejos para separarlos de las sentencias.

2.5.2.6.1.4 Comentarios de fin de línea

- Pueden convertir en comentario una línea completa o una parte de una línea.
- No deben ser usados para hacer comentarios de varias líneas consecutivas.
- Pueden usarse en líneas consecutivas para comentar secciones de código.
- Utilizarlos lo menos posible porque son más difíciles de leer.
- Son apropiados al anotar declaraciones de variables.

2.5.2.6.2 Comentarios de Documentación

- No se pueden aplicar a un espacio de nombres.
- Se pueden incorporar a nivel de clase, a nivel de variable y a nivel de método.
- En un comentario solo puede haber un único comando.

2.5.2.6.3 Comentarios especiales

- Se identifican con una etiqueta especial para ser fácilmente ubicados dentro de los archivos con las herramientas de búsqueda de los editores y además como llamada.
- Las etiquetas se definen con una sola palabra en mayúscula.
- Se utilizan cuando el implementador previó la necesidad de agregar algo pero no lo hizo, está pendiente y hay que hacerlo.
- Cuando hay un bug no resuelto.
- Cuando el implementador quiere emitir un criterio sobre algún trabajo mal hecho y que pudo hacer mejor.
- Cuando se quiere indicar que se esta en presencia de un código engañoso, y el mismo se quiere modificar y/o entender.
- Cuando se quiere alertar una zona peligrosa.

2.5.2.7 Declaraciones

Las declaraciones ayudan a establecer reglas que permitirán darle una mejor claridad al código fuente. Mejoran el proceso de inicialización de variables, y establecen puntos importantes a tener en cuenta cuando se trabaja con clases e interfaces., desde la perspectiva de dicho indicador.

2.5.2.7.1 Número de declaraciones por línea

- Declare una sola variable por línea, para una mayor claridad del código (excepto VB).
- Escriba una instrucción de código por línea (excepto un bucle en C, C++, C#, o JScript, como en for (i = 0; i < 100; i++)).
- No poner tipos diferentes en la misma línea.
- No debe usarse el separador coma para declarar múltiples variables y campos/componentes en estructuras de datos tipo registro en una misma línea por la poca claridad que ello acarrea para su lectura.

2.5.2.7.2 Inicialización

- Inicializar las variables locales donde se declaren (excepto si el valor inicial depende de algún cálculo que debe ocurrir primero
- Inicializar las variables locales a medida que son declaradas.

2.5.2.7.3 Declaración de clases e interfaces

- No dejar espacios en blanco entre el nombre del método y el paréntesis.
- La llave que comienza el bloque de código del método no debe aparecer a continuación de la declaración.
- La llave que cierra el bloque de código del método debe estar sola en una línea indentada en correspondencia a la llave que abre el bloque.
- Empezar los nombres de interfaz con el prefijo "I", seguido de un nombre o una frase nominal o con un adjetivo que describa el comportamiento de la interfaz.

2.5.2.8 Sentencias

Son oraciones que tienen determinadas funcionalidades en dependencia del lugar donde actúan, y la composición de elementos que tengan. Son la base de las estructuras de control y brindan una mayor organización durante la programación.

2.5.2.8.1 Sentencias simples

- Cada línea debería contener una sentencia como máximo.

2.5.2.8.2 Sentencias compuestas

- Son sentencias que contienen una lista de sentencias encerradas entre llaves.
- Las sentencias internas deberían estar tabuladas un nivel más que la sentencia compuesta.
- La llave de apertura debería estar al final de la línea que comienza la sentencia compuesta; la llave de cierre debería estar en una nueva línea y estar tabulada al nivel del principio de la sentencia compuesta.
- Las llaves se usan en todas las sentencias compuestas, incluídas las sentencias únicas, cuando forman parte de una estructura de control, esto hace más fácil introducir nuevas sentencias sin provocar errores accidentales al olvidarse añadir las llaves.

2.5.2.8.3 Sentencias de retorno

- Cuando tiene un valor no debe usar paréntesis a menos que hagan el valor de retorno más obvio de alguna manera.

2.5.2.8.4 Sentencias if, if-else, if else-if else

- Las Sentencias if siempre llevan llaves.
- El tipo de sentencias if-else debería tener el siguiente formato:

```
if (<em>condición</em>) {
    <em>sentencias</em>;
}

if (<em>condición</em>) {
    <em>sentencias</em>;
} else {
    <em>sentencias</em>;
}

if (<em>condición</em>) {
    <em>sentencias</em>;
} else if (<em>condición</em>) {
    <em>sentencias</em>;
} else {
    <em>sentencias</em>;
}
```

2.5.2.8.5 Sentencias for

- Una sentencia *for* debería tener el siguiente formato:

```
for (<em>inicialización</em>; <em>condición</em>;
<em>actualización</em>) {
    <em>sentencias</em>;
}
```

- Una sentencia for vacía (aquella en la que todo el trabajo se hace en las cláusulas de inicialización, condición y actualización) debería tener el siguiente formato:

```
for (<em>inicialización</em>; <em>condición</em>;
<em>actualización</em>);
```

2.5.2.8.6 Sentencias While

- Una sentencia while debería tener el siguiente formato:

```
while (<em>condición</em>) {
    <em>sentencias</em>;
}
```

- Una sentencia while vacía debería tener el siguiente formato:

```
while (<em>condición</em>);
```

2.5.2.8.7 Sentencias do-while

- Una sentencia do-while debería tener el siguiente formato:

```
do {
    <em>sentencias</em>;
} while (<em>condición</em>);
```

2.5.2.8.8 Sentencias switch

- Si se definen variables dentro del case, encerrar todas las sentencias en un par de llaves.
- Cada vez que un caso continúa con el siguiente (no incluye una sentencia break), se añade un comentario donde iría la sentencia break.
- Todas las sentencias switch deberían incluir un caso por defecto.
- Una sentencia switch debería tener el siguiente formato:

```
switch (<em>condición</em>) {
    case ABC:
        <em>sentencias</em>;
        /* continua con el siguiente */

    case DEF:
        <em>sentencias</em>;
        break;

    case XYZ:
        <em>sentencias</em>;
        break;

    default:
        <em>sentencias</em>;
        break;
}
```

2.5.2.8.9 Sentencias Try-Catch

- Evite anidar bloques try/catch.
- Una sentencia try-catch debería tener el siguiente formato:

```
try {
    <em>sentencias</em>;
} catch (ExceptionClass e) {
    <em>sentencias</em>;
}
```

2.5.2.9 Espaciado

Ayudan a establecer separaciones en el código fuente para mejorar su estructura, y su legibilidad. Mejoran el entendimiento del mismo.

2.5.2.9.1 Líneas en blanco

- Separar por una línea en blanco la declaración de variables y estructuras de datos y las instrucciones ejecutables del cuerpo mismo de la función, para enfatizar donde terminan las primeras y dónde inician las segundas.

- Insertar una línea en blanco entre el encabezado documentativo de una función y el código de la función misma.

- Insertar una línea en blanco alrededor de bloques de código importantes, tales como ciclos, sentencias if y conjuntos de instrucciones claves.
- Insertar una línea en blanco entre la última línea de una función y su delimitador de cierre.

2.5.2.9.2 Espacios en blanco

- Utilice espacios antes y después de los operadores siempre que eso no altere la sangría aplicada al código.
- Use espacios en blanco para organizar a su antojo secciones de código. De tal manera que se comprenda la segmentación del código.
- Use espacios en blanco para proporcionar indicaciones organizativas del código fuente. Así, creará "párrafos" de código, que ayudarán al lector a comprender la segmentación lógica del software.

2.5.2.10 Convenciones

Se utilizan para definir tipos de notaciones que le garanticen legibilidad al código, haciéndolo fácil de leer y de escribir.

2.5.2.10.1 Estilo de Pascal

- Capitaliza la primera letra de cada palabra.
- Utilizar el estilo de Pascal para los nombres de los formularios. Prefijo ‘frm’.
- Utilizar el estilo de Pascal para los nombres de las librerías. Prefijo ‘lib’.
- Utilizar enumerados para los nomencladores.
- Puede utilizar el caso del PASCAL para los identificadores de tres o más caracteres.

Tipo	**Estilo**	**Notas**
Clase / Estructura	Pascal	
Interfase	Pascal	Comenzando con I
Clases de excepciones	Pascal	End with Exception
Métodos	Pascal	
Espacios de nombres	Pascal	
Propiedades	Pascal	

2.5.2.10.2 Estilo de camello

- La primera letra de un identificador es minúscula y la primera letra de cada palabra concatenada subsecuentemente se capitaliza. Por ejemplo: backColor.

Tipo	Estilo
Campos protegidos / privados	Camello
Parámetros	Camello
Variables locales	Camello

2.5.2.10.3 Mayúsculas

- Utilice a esta convención solamente para los identificadores que consisten en dos o pocas letras. Por ejemplo: System.IO System.Web.UI

2.5.2.10.4 Directivas de declaración

- Determinar cuáles son las abreviaturas más utilizadas y a partir de ahí definirlas.
- Para términos largos o utilizados con frecuencia, utilice abreviaturas para mantener las longitudes de los nombres dentro un límite razonable.
- Asegúrese de que sus abreviaturas sean coherentes a lo largo de toda la aplicación.
- Minimice el uso de abreviaturas; pero si las emplea, use coherentemente las que haya creado.
- Una abreviatura sólo debe tener un significado y, del mismo modo, a cada palabra abreviada sólo debe corresponder una abreviatura.

2.5.2.11 Nomenclatura

El esquema de nombres es una de las ayudas más importantes para entender el flujo lógico de una aplicación. Un nombre debe más bien expresar el "qué" que el "cómo". Si se utiliza un nombre que evite referirse a la implementación se estará conservando la abstracción de la estructura ya que la implementación está sujeta a cambios, de esta manera se describe qué hace la estructura y no como lo hace.

Las convenciones de nombrado hacen los programas más comprensibles haciéndolos más fáciles de leer. También pueden dar información acerca de la función del identificador (por ejemplo, si se trata de una constante, un paquete o una clase), que puede ayudar a entender el código.

2.5.2.11.1 Variables

- La elección del nombre de una variable debe ser un mnemónico, designado para indicar a un observador casual su función.
- Evitar asignar a varias variables el mismo valor en una sola sentencia.
- Los nombres de variables LOCALES deben ser cortos y concretos pero con significado.
- Las variables globales deben usarse solo si realmente se necesitan y siempre en mayúsculas, separando las palabras con guiones bajos (_). Excepto *true (verdadero), false (falso) y null (nulo).*
- Las variables miembros de clase se escriben con minúsculas.

2.5.2.11.2 Métodos

- Los nombres de métodos deben ser verbos o frases verbales escritos en notación de camello con la primera letra en minúscula.
- Evite llamadas a métodos dentro de sentencias condicionales.
- Establecer un orden para agrupar los métodos.
- Seleccionar prefijos para métodos estándar que a partir del nombre la semántica quede completamente definida.
- Los métodos miembros de clases pueden coexistir con otras clases de variables.

2.5.2.11.3 Parámetros

- Usar nombres que sean suficientemente descriptivos como para determinar el significado de la variable y su tipo.
- Usar el estilo camello
- Los parámetros siguen el mismo estándar de las variables.

2.5.2.11.4 Constantes

- Los nombres de las variables declaradas como constantes deben ir totalmente en mayúsculas separando las palabras con un subguión (“_”).

- Las constantes numéricas no deberían codificarse directamente, excepto -1, 0 y 1, que pueden aparecer en un bucle *for* como contadores.
- No utilizar constantes numéricas directamente en el código, se declara una variable constante y se utiliza en su lugar

2.5.2.11.5 Clases

- Una clase debe estar definida en orden descendente de la siguiente manera: Variables Miembro, Constructores, Enumeraciones, Estructuras o Clases anidadas, Propiedades y por ultimo los Métodos.
- La secuencia de declaración de las clases de acuerdo a los modificadores de acceso debe ser: *public (publica), protected (protegida), internal (interna) o private (privada)* .
- Los nombres de clases comienzan con mayúsculas, con las palabras que la forman en minúsculas y separadas por mayúsculas, de la misma forma que las variables miembro.
- El nombre debe ser descriptivo, evitando abreviaturas.

2.5.2.11.6 Atributos

- Cuando aplique más de un atributo a una estructura colóquelos individualmente en una línea y no todos juntos y separados por comas.
- El valor del atributo sólo puede contener letras (a-z y A-Z), dígitos (0-9), guiones (ASCII decimal 45), puntos (ASCII decimal 46), subguiones (ASCII decimal 95) y dos puntos (ASCII decimal 58). Recomendamos usar comillas incluso cuando sea posible eliminarlas.
- El valor de un atributo puede ser: una cadena de caracteres entre comillas (simples o dobles) que no contenga el símbolo de fin de marca ">" o un nombre como los definidos en el apartado anterior.

2.5.2.11.7 Propiedades

- Nombrar las propiedades utilizando sustantivos o frases en sustantivo.
- Utilizar el estilo pascal.
- Considerar nombrar las propiedades con el mismo nombre de su tipo.

2.5.2.11.8 Interfaces

- Los nombres de las interfaces siguen la misma regla que las clases.
- Para los nombres de interfaces se deben utilizar sustantivos, frases en sustantivo o adjetivos que describan comportamiento.
- Usar I como prefijo, seguido por una letra mayúscula (primera letra del nombre de la interfaz). Se debe usar el estilo de pascal.
- Con la excepción del código relacionado con la interfaz gráfica de la aplicación. Todos los nombres de variables y campos que contengan elementos de interfaz como botones, se les debe agregar al principio la abreviatura del tipo.
- Utilice interfaces antes que clases abstractas.

2.5.2.11.9 Funciones

- Las funciones deben ser cortas, fáciles, y deben evitar el uso de abreviaturas.
- La longitud máxima de una función es inversamente proporcional a la complejidad y grado de sangría de esa función.
- Se deben usar funciones de biblioteca.
- Los nombres de las funciones deben ser todos en minúsculas y la separación de palabras hechas con el caracteres.
- El nombre de toda función debe comenzar con el nombre del módulo al que pertenece y seguido por un nombre que describa el comportamiento de la función.

2.5.2.11.10 Idioma

- Todas las declaraciones deben ser en un idioma específico.
- En la [ISO639] se reservan códigos principales de dos letras para las abreviaturas de los idiomas. Entre estos códigos de dos letras están fr (francés), de (alemán), it (italiano), nl (neerlandés), el (griego), es (español), pt (portugués), ar (árabe), he (hebreo), ru (ruso), zh (chino), ja (japonés), hi (hindi), ur (urdu), y sa (sánscrito).
- Se sobreentiende que cualquier código de dos letras es un código de país de [ISO3166]

2.5.2.12 Prácticas de programación

2.5.2.12.1 Tratamiento de Excepciones

- Se deben capturar siempre en el nivel más alto del sistema, es decir en la capa de más abstracción, por lo general es la capa de interfaz de usuario.

- Ordene la captura de excepciones (catch) siempre en orden descendente desde la más particular hasta la más genérica.
- Siempre que sea posible prefiera la validación al manejo de excepciones.
- Es una directiva general evitar en todo caso crear excepciones personalizadas, pero en el caso de que sea necesario se deben seguir los pasos adecuados.

2.5.2.12.2 Modularización

- Modulariza usando subrutinas.
- Reemplaza expresiones repetitivas por llamadas a una función común.
- Cada módulo debería hacer una única cosa bien.
- Asegúrate que cada módulo oculte algo.
- Haz el acoplamiento entre módulos visible.

2.5.2.12.3 Anidamiento de instrucciones

- Evita niveles profundos de anidamiento de estructuras de control y de funciones.
- Más de tres niveles de anidamiento dificulta la comprensión de un programa.

2.5.2.12.4 Visibilidad

- No hacer ninguna instancia o campo de una clase público, deben ser privadas.
- Una función miembro pública se puede invocar por cualquier otra función miembro en cualquier otro objeto o clase.
- Una función miembro protegida se puede invocar por cualquier función miembro en la clase en la cual se define o cualesquiera subclases de esa clase.
- Una función miembro privada se puede invocar solamente por otras funciones miembro en la clase en la cual se define, pero no en las subclases.

2.5.2.12.5 Inclusión de código

- No utilización de números que puedan ser reemplazados por constantes.
- Cada archivo de inclusión debe contener un mecanismo que prevenga múltiples inclusiones del archivo.
- Salvo casos específicos y puntuales, utilizar require_once para incluir código incondicionalmente, e include_once para los casos condicionales (o sus equivalentes en otros lenguajes).

2.5.2.12.6 No números Mágicos

- No utilización de números que puedan ser reemplazados por constantes.

2.5.2.13 Ejemplos de código

- Se deben incluir ejemplos que puedan servirle de ayuda al programador.
- Deben especificarse los ejemplos según el lenguaje.

2.5.2.14 Recursos

Te permiten tener acceso a un conocimiento sin barreras pues los programadores pueden incrementar su experiencia y aclarar dudas en caso de tenerlas.

2.5.2.15 URLs de Ejemplos

- Para denotar URLs de ejemplo y direcciones de e-mail en la documentación y comentarios.

2.5.2.15.1 Libros recomendados

- Posibilitan ampliar el alcance del nivel de acción de la guía, en caso de que queden elementos por abordar.
- El programador puede apoyarse en ellos para trabajar adecuadamente.
- Tener en cuenta el lenguaje de programación para seleccionar los libros a recomendar.

Ej. *The Elements of Java Style*, editado por Al Vermeulen es un libro que trata de algunos de los puntos más importantes de la buena programación, tales como el buen uso de las convenciones de nombramiento, comentarios, e, incluso, los espacios en blanco y las sangrías etc.

2.5.2.15.2 Herramientas

- Deben seleccionarse de acuerdo al lenguaje de programación que se utilice.
- Deben incluir una breve explicación de su funcionalidad e nivel de acción.
- De ser posible incluir una recomendación de ante que situaciones se deben emplear.

Ej.

- for C++:
    - PC-Lint (Gimpel Software)

- CodeWizard/C++Test (Parasoft)

- for C#:
  - FxCop (Microsoft)
  - ClockSharp (TIOBE)
- for Java:
  - CheckStyle (SourceForge project)
  - JCSC (SourceForge project)
  - JTest/CodeWizard (Parasoft)

2.5.2.16 Glosario de Términos

Debe contener las palabras que puedan generar alguna duda en cuanto a su significado.

2.5.3 Nivel de acción de la Guía de Estándar Genérico

Después de descritos los elementos de la Guía de Estándar Génerico con una serie de reglas o valores generales es posible percatarse del alcance de la misma. Dicha propuesta, usada correctamente, permitirá lograr una uniformidad en el código a nivel de Universidad garantizándole más legibilidad, y un menor índice de errores. Esto será posible porque comprende 14 indicadores, que permiten evaluar y/o controlar los estándares de codificación utilizando herramientas de evaluación, de las que se abordarán más adelante.

Dicha propuesta incluye dos vertientes fundamentales de aplicación. La primera está orientada a trabajar con los estándares de codificación seleccionados por los proyectos, en dependencia del lenguaje de programación que utilicen, y si el mismo se incluye entre los que se utilizaron para elaborar la guía. En este caso, la propuesta de estándar genérico se aplicaría como punto de comparación o elemento mínimo común a tener en cuenta, para saber si el estándar seleccionado por el proyecto al que se hace referencia cumple con las mismas variables o indicadores que el guía, esto crearía un intervalo de selección, independiente del lenguaje, que garantizaría una similitud en la codificación, a nivel de Universidad y por lo tanto un mayor entendimiento, acoplamiento y calidad a la hora de producir software. Para lograr establecer la comparación se debe hacer uso de una lista de chequeo definida a partir de la guía, que permite conocer el estado del

estándar y si es el más indicado a aplicar, evaluándolo con ayuda de una escala previamente definida y la cual se explicará en el epígrafe 2.5.4

En el segundo caso, la propuesta permitiría definir una lista de chequeo, un poco más concreta que la establecida para el caso anterior, la cual se utilizaría para evaluar el código fuente directamente, y de esta forma, con ayuda de la escala especificada para dicha lista, evaluar al código y determinar el estado en que se encuentra, pudiendo inferir si en el proyecto se esta trabajando óptimamente o si se están presentado problemas que imposibilitan la creación del producto esperado, con la calidad que requiere, suficiente para cumplir las expectativas del cliente y las del equipo de desarrollo. Hasta es posible, utilizar la guía como indicador para evaluar a los proyectos de la Universidad y conocer cuales son los que más problemas presentan, y en base a ello brindar soluciones que permitan corregir dichos problemas.

2.5.4 Herramientas de evaluación

Las herramientas de evaluación que se utilizan en la propuesta son dos listas de chequeo definidas a partir de la misma, que en dependencia de las variables, trabajan sobre los estándares de codificación o sobre el código fuente.

Estas listas de chequeo fueron confeccionadas con el fin de tener una herramienta que ayude elegir estándares de codificación para cualquier lenguaje de programación, verificando que el Estándares de codificación que se elija cumpla con la mayor cantidad de elementos que brinda la lista de chequeo, que está basada en la guía propuesta que comprende los elementos comunes de los estándares de codificación.

Para hacer uso de las listas se tiene una escala con cuatro valores, máximo, adecuada, medio y mínimo, cada uno con un valor asignado descendentemente del 4 al 1.
Cuando se va a verificar si el estándar en cuestión cumple con los elementos establecidos en la guía propuesta se utiliza la lista de chequeo de la siguiente manera.

- Se verifica que el estándar posea las características de la lista de chequeo, asignándole un valor del uno al cuatro para saber en que medida el elemento tiene relación con el de la lista.

En caso que el elemento de la lista tenga otros elementos Ej. : 1) Formato: a)…b)… c)…a cada uno de los elementos anidados se le hace la misma operación y se calcula el promedio entre los valores obtenidos y dicho promedio dirá en que medida el estándar cumple con la guía y si es recomendable utilizarlo o no. En caso de que el valor final sea un número decimal se recomienda hacer un análisis pesimista, ya que el objetivo es abogar por la optimización y no por la conformidad.

2.5.4.1 Lista de chequeo para estándares

La lista de chequeo para evaluar estándares incluye todos los indicadores junto a sus atributos establecidos en la propuesta. Y se deberá aplicar a cualquier estándar para conocer si el mismo cumple con los requerimientos mínimos que se necesitan para utilizarse en cualquier proyecto productivo de la Universidad.

LISTA DE CHEQUEO PARA ESTANDARES

Máximo	4
Adecuado	3
Medio	2
Mínimo	1

Promedio el resultado y da un número decimal para evaluar el punto más general Ej. 2.5 está más cerca de2
Hacer un análisis pesimista.

Elemento	**Totalización**	**Adecuación**	
1. Introducción	Medio		
❖ Por qué usar convenciones de código		Mínimo	
❖ Objetivos		Máximo	
❖ Alcance			
2. Formato			

❖ Sangría	Adecuado		
❖ LLaves			
❖ Diseño entrada y salida (E/S)			
❖ Tablas			
❖ Paréntesis			
❖ Estilo y código fuente			
❖ Operadores			
❖ Estructura del programa			
3. Ficheros			
❖ Extensiones de los Ficheros			
❖ Nombres de ficheros comunes			
❖ Contenido de los archivos (interfaz e implement)			
4. Identación			
❖ Reglas de delimitación			
❖ Longitud de la línea			
❖ Ruptura de la línea			
❖			
5. Comentarios			
❖ Comentarios de implementación			
• Comentarios de bloque			
• Comentarios de una línea			
• Comentarios de por detrás			
• Comentarios de fin de línea			
❖ Comentarios de documentación			
❖ Comentarios especiales			
6. Declaraciones			
❖ Número de declaraciones por línea			
❖ Inicialización			

❖ Declaración de clases e interfaces			
7. Sentencias			
❖ Sentencias simples			
❖ Sentencias compuestas			
❖ Sentencias de retorno			
❖ Sentencias if			
❖ Sentencias if-else, if else-if else			
❖ Sentencias for			
❖ Sentencias While			
❖ Sentencias do-while			
❖ Sentencias switch			
❖ Sentencias Try-Catch			
8. Espaciado			
❖ Líneas en blanco			
❖ Espacios en blanco			
❖ Espacio entre términos			
9. Convenciones			
❖ Estilo de Pascal			
❖ Estilo de camello			
❖ Mayúsculas			
❖ Directivas de declaración			
10. Nomenclatura			
❖ Variables			
❖ Métodos			
❖ Parámetros			
❖ Constantes			
❖ Clases			
❖ Atributos			
❖ Propiedades			

❖ Interfaces			
❖ Funciones			
❖ Idioma			
11. Prácticas de Programación			
❖ Referenciar variables y métodos de clases			
❖ Excepciones			
❖ Modularización			
❖ Anidamiento de instrucciones			
❖ Visibilidad			
❖ Inclusión de código			
❖ No números mágicos			
12. Ejemplos de Código			
13. Recursos			
❖ URLs de Ejemplos			
❖ Libros recomendados			
❖ Herramientas			
14. Glosario de términos			

2.5.4.2 Lista de chequeo para código fuente

La lista de chequeo para código fuente incluye la mayoría de los elementos de la propuesta, excepto los que no estén relacionados directamente con el código como es el caso de la Introducción, los recursos y los indicadores de los mismos. Esto ayuda a conocer como se está ejerciendo la etapa de implementación de código en la UCI y cuales son los principales riesgos en la misma, que impiden la obtención del producto en tiempo y con calidad. Para ver la lista de chequeo para código remitirse al anexo 1.

2.5.5 ¿Cómo utilizar la guía?

- La Guía de Estándar Genérico se aplicará cuando se desee establecer un estándar de codificación para un proyecto que programe en los lenguajes C#, C++, Java y PHP.
- Una vez seleccionado el estándar por un proyecto, necesita antes de aprobarlo, evaluarlo con ayuda de la GEG para saber en que grado es bueno y esto se hace verificando que los elementos del estándar coincidan con los elementos de la propuesta.
- Para evaluar el estándar se utiliza la lista de chequeo para Estándares de codificación que la GEG establece, en la que se le asignarán valores a los elementos de la lista de que están contenidos en el estándar chequeo (dichos elementos son los mismos de GEG).
- Con la lista de chequeo se comparan los elementos del estándar seleccionado con los de la GEG y se evalúan los que coincidan.
- Los valores que se le asigna a los elementos tienen un equivalente numérico, del 1 al 4 de acuerdo al nivel de coincidencia.
- Una vez que se le han asignado los valores a los elementos que coinciden, estos se promedian para obtener un valor final que determine si el estándar es bueno o no.
- En caso de que el valor que se obtenga al promediar sea decimal se debe escoger el entero antecesor, pues la idea es buscar optimización, no conformidad. (Por ejemplo si el valor final es 2,5 el valor que se toma es 2).
- Una vez realizado este proceso se puede pasar al establecimiento del estándar según el resultado obtenido.
- Si el resultado obtenido no es bueno se recomienda buscar otro estándar del lenguaje de programación con el que se va a trabajar y hacerle el mismo proceso, luego se debe escoger el que mejor resultado obtenga.

2.5.6 Opiniones de los expertos.

Después de tener elaborada la propuesta de estándar genérico fue necesaria su validación, por lo que se le presentó a diferentes expertos en el tema y a algunos de los compañeros especialistas y dirigentes de los sectores beneficiados con su aplicación y capacitados para evaluar la GEG. Sus opiniones se recogieron mediante entrevistas.

Las preguntas hechas realizadas en las entrevistas fueron las mismas para todos y son las que a continuación aparecen.

Pregunta 1. Considera importante la aplicación de una guía de estándar genérico de codificación en los proyectos de la Universidad, para contribuir a normalizar la escritura de código.

Pregunta 2. Cree útil la aplicación de una Guía para seleccionar que estándar es más factible aplicar en un proyecto o en otro ámbito. Aporte tres elementos que apoyen su respuesta.

Pregunta 3. Cree que el procedimiento utilizado para confeccionar la guía es adecuado.

Pregunta 4. Considera indicado el método de utilizar las listas de chequeo como herramientas para evaluar los Estándares y el código fuente a partir de la Guía.

Pregunta 5. Cree adecuadas las variables de calidad usadas de la Guía de Estándar Genérico propuesto.

Opiniones de los expertos entrevistados.

Experto1. Ingeniero Renier Pérez García Director de tecnología.

El compañero considera totalmente importante la aplicación de la guía de estándar Génerico de codificación en los proyectos de la universidad.

- Cree totalmente útil la aplicación de una Guía para seleccionar qué estándar es más factible aplicar en un proyecto o en otro ámbito.
- Cree que el procedimiento utilizado para confeccionar la guía es totalmente adecuado.
- Considera totalmente indicado el método de utilizar las listas de chequeo como herramientas para evaluar los Estándares y el código fuente a partir de la Guía.
- Cree adecuadas las variables de calidad usadas de la Guía de Estándar Genérico propuesto.

Experto2. MSc. Eugenia G. Muñiz Lodos

Prof. Titular Adjunta

Investigador Auxiliar ICID

Más de 30 años de experiencia desarrollando software.

La compañera considera totalmente importante la aplicación de la guía de estándar Génerico de codificación en los proyectos de la universidad.

- Es totalmente útil la aplicación de una Guía para seleccionar qué estándar es más factible aplicar en un proyecto o en otro ámbito, pues ayuda a que no se olvide estandarizar elementos importantes y brinda recomendaciones muy prácticas.
- El procedimiento utilizado para confeccionar la guía es totalmente adecuado.
- Es totalmente indicado el método de utilizar las listas de chequeo como herramientas para evaluar los Estándares y el código fuente a partir de la Guía.
- Son totalmente adecuadas las variables de calidad usadas de la Guía de Estándar Genérico propuesto.

Su opinión general es que la GEG es muy importante sobre todo porque cuando se trabaja en grupo la utilización de Estándares de codificación es imprescindible, por lo que sería muy bueno utilizar un guía que permita seleccionar qué estándar utilizar.

Experto3. Ing. Ramses Delgado Martínez.
Director de calidad de Software de La Universidad de las Ciencias Informáticas.

El compañero considera totalmente importante la aplicación de la guía de estándar Génerico de codificación en los proyectos de la universidad.

- Cree totalmente útil la aplicación de una Guía para seleccionar qué estándar es más factible aplicar en un proyecto o en otro ámbito.
- Cree que el procedimiento utilizado para confeccionar la guía es totalmente adecuado.
- Considera totalmente indicado el método de utilizar las listas de chequeo como herramientas para evaluar los Estándares y el código fuente a partir de la Guía.
- Cree adecuadas las variables de calidad usadas de la Guía de Estándar Genérico propuesto.

Experto4. Ing. Ernesto Medina Delgado

El compañero considera totalmente importante la aplicación de la guía de estándar Génerico de codificación en los proyectos de la universidad.

- Cree totalmente útil la aplicación de una Guía para seleccionar qué estándar es más factible aplicar en un proyecto o en otro ámbito pues existen muchas guías de

codificación para cada lenguaje por lo cual en ocasiones resulta difícil encontrar la adecuada y conocer en que aspectos apoyarse para evaluar cada punto de la misma, esta guía puede reducir los tiempos de búsqueda y decisión y ayudar a obtener un resultado más sólido.

- Cree que el procedimiento utilizado para confeccionar la guía es en alguna medida adecuado.
- Considera que el método de utilizar las listas de chequeo como herramientas para evaluar los Estándares y el código fuente a partir de la Guía es indicado en alguna medida.
- Cree totalmente adecuadas las variables de calidad usadas de la Guía de Estándar Genérico propuesto.

De manera general opina que la guía es bastante completa, recoge los aspectos fundamentales que se deben considerar para la selección de un estándar de codificación para un lenguaje determinado, sin embargo adolece de un punto importante que no se evalúa ni se toma en cuenta en la mayoría de las guías actuales, y es lo relacionado con los aspectos concernientes a la seguridad de la codificación. Un buen estándar también debe incluir aquellos puntos que aseguran que se programe tanto eficientemente como de manera segura. Por ejemplo, se pueden mencionar desde que librerías no son convenientes utilizar, hasta que comparaciones son más eficientes o convenientes. Es por eso que la guía también debería valorar estos puntos en las variables de calidad para evaluar la solidez de un estándar a elegir.

2.5.6.1 Análisis de las Entrevistas

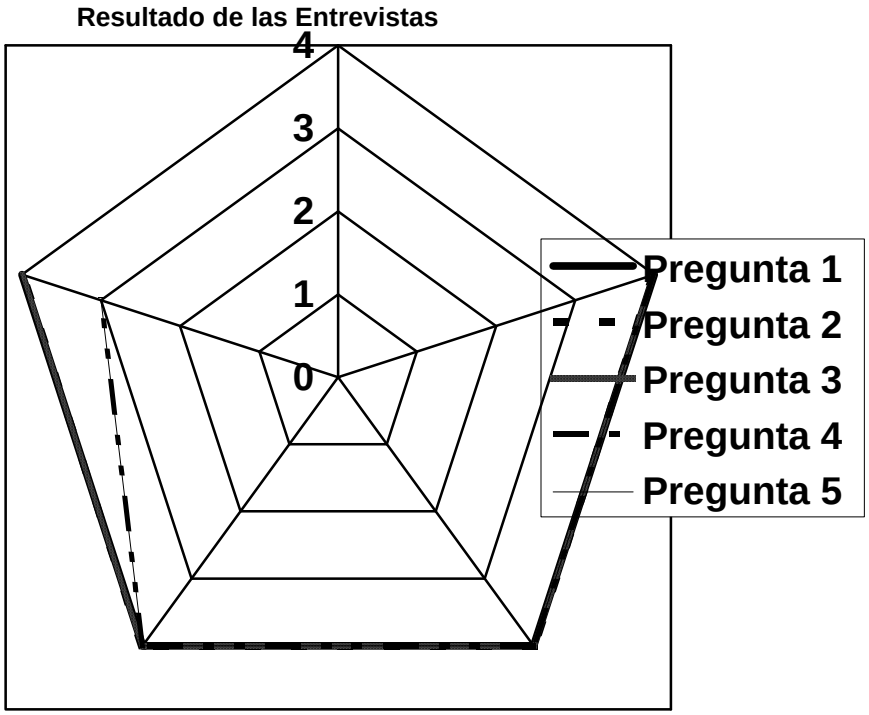
El análisis de las de la validación se hizo a partir de las respuestas de cada pregunta asignándole valores a las opciones de respuestas que brindaban las entrevistas para que se hiciera su selección.

A cada una de las 4 posibles respuestas se le asignaron valores desde 4 hasta 1.

Respuestas	Valores
Adecuadamente	**4**
En alguna medida.	**3**
Poco	**2**
Muy poco	**1**

Estos valores se ubicaron en una tabla para analizar gráficamente las respuestas de los expertos y saber en que medida coinciden sus criterios.

Actividades propuestas en la Encuesta	Exp.1	Exp.2	Exp.3	Exp.4	Total
Pregunta 1	4	4	4	4	16
Pregunta 2	4	4	4	4	16
Pregunta 3	4	4	4	4	16
Pregunta 4	4	4	4	3	15
Pregunta 5	4	4	4	3	15



En el gráfico se puede apreciar como tres de los expertos entrevistados coinciden totalmente otorgándole el máximo de puntos a los temas evaluados con las preguntas y solo uno no coincide.

Conclusiones

El desarrollo de este trabajo a través de las revisiones de documentos y diferentes bibliografías que posibilitó conceptuar los elementos investigados, logró que:

- Se incrementara el conocimiento a cerca de las buenas prácticas para el mejoramiento de la codificación, calidad, estándares y calidad del código.

- Se conociera sobre la situación que presenta la universidad con respecto a las buenas prácticas y estándares de codificación

- Se obtuviera una guía con los elementos comunes de los Estándares de codificación, que contribuye a estandarizar la codificación en la Universidad de las Ciencias Informáticas.

Además permitió llegar a la conclusión de que:

- La correcta aplicación de un buen estándar de codificación, mejora considerablemente el código fuente de los programas, debido a que su principal función es prevenir los errores.

- Para que se obtenga un código de calidad no solo se debe utilizar dichos estándares, sino que es necesario la aplicación de las demás prácticas estudiadas, ya que estas ayudarán a corregir los errores que no sean evitados por los Estándares, posibilitando mayor calidad al código y por ende al software.

Por último después de realizar las entrevistas a los expertos en el tema se pudo concluir que la GEG se puede aplicar en la Universidad de las Ciencias Informáticas y que posee gran utilidad debido a todos los entrevistados consideran totalmente importante la aplicación de la guía de estándar Génerico de codificación en los proyectos de la universidad.

Recomendaciones

El estudio de los estándares permitió conocer que ellos no eliminan los errores sino que ayudan a evitarlos y facilitan el trabajo a las otras buenas prácticas estudiadas, por lo que es importante, y se recomienda:

- Siempre que se programe se utilicen, por lo menos algunas de las otras prácticas estudiadas en el presente trabajo, destinadas a mejorar la codificación.

- Una vez aplicada la guía de estándar genérico, los Estándares que resulten tener las condiciones necesarias para ser aplicados, se publiquen en la Intranet con el objetivo de que todos los programadores, los conozcan y en un momento determinado puedan saber cual elegir según el lenguaje de programación.

Para utilizar la guía es necesario seguir las recomendaciones que se hacen a continuación:

- Leer antes el epígrafe ¿Cómo utilizar la guía de estándar genérico?

- Utilizar las dos listas de chequeo que se brindan como herramientas de evaluación.

- En el caso de que ninguno de los resultados obtenido sea bueno, se debe escoger cualquiera pues por malo que sea el estándar, es mejor opción utilizarlo que trabajar sin estándar**.**

Además de manera general, se recomienda que desde que comience la formación de los estudiantes en el primer año de la carrera, se les exija la utilización de los Estándares de codificación para que trabajen de forma uniforme y logren mayor calidad en el código de sus programas, así como el uso de las buenas prácticas que intervienen en el mejoramiento de la codificación.

Bibliografía

- Best Practices: PHP Coding Style.
- C# Coding Standard. 2005.
- C++ Programming Style Guidelines, 2007.
- Coding Solutions and Requirements. 2003.
- Coding Techniques and Programming Practices. 2007.
- Convenciones de código para el lenguaje de programación Java. 1999.
- Estándares de Codificación de sistemas. 2006. 1.
- IV workshop de investigadores en ciencia de la computación., 2002.
- Java code Conventions. 1997.
- Patrones de Diseño, 2001.
- Patrones de diseño., 1995.
- Propiedad colectiva del código, tests, y coding Standards. 2005.
- Recomendaciones para programación C++. 2007.
- Revisiones de código y estándares de codificación. 2006.
- FIGUEROA., P. Introducción a Patrones, 1997.
- ABDERRAHMAN, H. B. Why Coding Standard?, 2005.
- AGUILAR, M. T. Un millón de graduados en los Joven Club de Computación Gramma, 2006.
- ALEXANDER, C. An Introduction for Object-Oriented Designers, 1993.
- ARECHAVALA, Y. G. and F. D. C. GARCÍA. Calidad de software, 1990.
- AVILA, E. A. M. Aproximación a los Estándares y Accesibilidad Web, 2005.
- BRIAND, L. Property-based Software Engineering Measurement 1996. p.
- CARREÑO, D. M. Estilo de codificación del núcleo Linux.
- CATALDI, Z., LAGE, F.1, PESSACQ, R.2 Y GARCÍA MARTÍNEZ, R. Ingeniería de Software educativo, 1993.
- CHANDRASEKHAR, C. P. La informática países del tercer mundo, 2002.
- COCKBURN, L. W. A. A. The Costs and Benefits of Pair Programming, 2005.
- COLLADO, M. Introducción a las tecnologías y estándares XML, 2006.
- CRUGER, M. C# Coding estyle guide, 2002.
- CUEVAS, A. B. M. Y. J. M. Estándares y Guías, 2001.
- D'ADDAMIO, J. Revisión de código administrado, 2007.

- DEPARTAMENTO DE ELECTRÓNICA, S. E. I. C. D. D. P. Estándar para la codificación del lenguaje C++, 2005.
- DEVNET. Analista estima que responsables de exportar software cubano adoptan nueva estrategia
- 2004.
- DÍAZ, J. F. Reglas de Estilo para la Codificación.
- DODANI. Mejores Prácticas de Programación. 2003. p.
- EMILIO A. SANCHEZ, P. L., AND JOSE H. CANOS. Código de calidad: Integrando Patrones de
- diseño y Refactoring.
- FEDERRATAS. Un catalogo de pruebas y errores., 2006.
- FELIX PRIETO, Y. C., JOSE MANUEL MARQUES, Y MIGUEL ANGEL LAGUNA. Análisis de Conceptos Formales como soporte para la evolución de Frameworks de dominio.
- FERNÁNDEZ, G. Estándar de Codificación DOTNET, 2005.
- FERRER, J. P. Y. J. Refactorización en la práctica: peligros y soluciones.
- FERRÓN, M. J. and JUAN PABLO GONZÁLEZ Sistemas de Programas, 2001.
- FOWLER, M. DesignStaminaHypothesis, 2007.
- ---. Fail Fast, 2006.
- ---. Refactoring Home Page. 2000. p.
- --- Refactoring with Martin Fowler, 2002.
- GARCÍA, C. ¿Por qué maquetar con estándares?, 2005.
- GIRO Framework para la Reutilización de la Definición de Refactorizaciones 2004.
- GÓMEZ, D. La Industria del Software costarriquense y los mercados internacionales., 2004.
- GRANADA, R. G. El software libre y los beneficios del conocimiento 2004.

URL

http://cs.uns.edu.ar/WICC2002/Papers/WICC2002-ISBD.pdf
http://www.literateprogramming.com/javaconv.pdf
http://mit.ocw.universia.net/6.170/6.170/f01/pdf/lecture-12.pdf
agamenon.uniandes.edu.co/~pfiguero/soo/Magister_Patrones/contenido.html
www.design-nation.net/es/archivos/001902.php
www.informatica.unah.edu.hn/article.php?story=20070228085722457
http://www.dosmilmastres.com/blog.php?anotacionid=4
http://agamenon.uniandes.edu.co/~pfiguero/soo/Magister_Patrones/intropatrones.html
www.qasystems.de/downloads/deutsch/downloads/downloads_qacm/Why_Coding_Standard.pdf
http://g.oswego.edu/dl/ca/ca/ca.html
dialnet.unirioja.es/servlet/articulo?codigo=653101&orden=85927&info=link
www.proyectoweb.org/boletin/aproximacion-a-los-estandares-y-accesibilidad-web.html
www.cs.umd.edu/~basili/publications/technical/T90.pdf
http://es.tldp.org/NuLies/web/2.2/Documentation/CodingStyle
www.itba.edu.ar/capis/webcapis/RGMITBA/comunicacionesrgm/c-icie99-ingenieriasoftwareeducativo.pdf
www.redtercermundo.org.uy/revista_del_sur/texto_completo.php?id=2360
collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF
alcazaba.unex.es/old/departamentos/34/40/3802917/XML_OSI_final.pdf
www.csharpfriends.com/Articles/getArticle.aspx?articleID=336
griho.udl.es/ipo/doc/09Estand.doc
www.microsoft.com/spanish/msdn/articulos/archivo/190207/voices/ReviewingManagedCode.mspx
http://216.239.51.104/search?q=cache:x7YzoD9wa6wJ:progra.iteso.mx/estandares/estandar%2520codificacion%2520c%2B%2B/estandarcodificacion.pdf+Est%C3%A1ndar+para+la+codificaci%C3%B3n+del+lenguaje+C%2B%2B&hl=es&ct=clnk&cd=1&gl=cu&client=firefox-a
http://tips.org.uy/SPA/portal/NOTTexto.asp?Nro=16473&Pais=CUB
http://www.galeon.com/neoprogramadores/ruls4cod.htm
www.gast.it.uc3m.es/theses/phd_liliana.pdf
issi.dsic.upv.es/publications/archives/f-1068919065005/DOLMENSanchez.pdf
federratas.codigolibre.net/

si.ugr.es/~gedes/actividades/tallerEvolucion2001/docs/8DEF.pdf
www.elguille.info/colabora/NET2005/giovannyfernandez_EstandarCodificacionNET.htm
http://www.willydev.net/descargas/Refactor.pdf
http://www.ldc.usb.ve/~teruel/ci3711/test1/index.html
http://www.martinfowler.com/bliki/
www.martinfowler.com/ieeeSoftware/failFast.pdf
http://www.refactoring.com/
http://www.artima.com/intv/refactor.html
www.alzado.org/articulo.php?id_art=496
http://216.239.51.104/search?q=cache:MNgywyY__x4J:www.giro.infor.uva.es/Publications/2004/CLM04/+un+Framework+para+la+Reutilizaci%C3%B3n+de+la+Definici%C3%B3n+de+Refactorizaciones&hl=es&ct=clnk&cd=1&gl=cu&client=firefox-a
cegesti.org/exitoempresarial/publications/resumido%20Software.pdf
http://bulma.net/body.phtml?nIdNoticia=2048
http://www2.ing.puc.cl/~jnavon/IIC2142/guerrero.html
http://www.bitacoras.sidar.org/g4/index.php?2005/09/22/10-patrones-de-diseno-de-interaccion-iii-colecciones-y-lenguajes-de-patrones
www.papermountain.org/refactoring2
www.possibility.com/Cpp/CppCodingStandard.html
http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel1/2837/6079/00237006.pdf?arnumber=237006
www.governingwithcode.org/journal_articles/pdf/Gov_Char.pdf
http://www.dagbladet.no/development/phpcodingstandard/
www.ce.berkeley.edu/~fenves/ce224-1998/doc/rules.US.ps
www.monografias.com/trabajos6/isof/isof.shtml
www.quarkblog.org/2007/01/07/buenas-practicas-utilizando-control-de-versiones/
www.iro.umontreal.ca/~sahraouh/qaoose2005/paper2.pdf
www.mcs.vuw.ac.nz/courses/COMP301/2007T1/lectures/COMP301CodingStandards_2007.pdf
satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html
www.gruposeti.com/Recursos/PDFs/COMPLEJIDAD.pdf
http://www.cubaminrex.cu/Archivo/Presidente/2005/FC_130905.htm
www.cubaminrex.cu/Sociedad_Informacion/2007/DiscursoRamiro.htm
http://www.cubaminrex.cu/Sociedad_Informacion/Cifras.htm

www.jornada.unam.mx/2006/03/07/029a1pol.php
http://www.mcc.unam.mx/~cursos/Algoritmos/javaDC99-2/patrones.html
www.ciw.cl/recursos/Charla_Métricas_Indicadores.pdf
www.cs.uiuc.edu/users/opdyke/wfo.990201.refac.html
www.navegapolis.net/content/view/156/59/
www.kybeleconsulting.com/pages/articulos/ConocimientoDOO.html
http://216.239.51.104/search?q=cache:jMG0xxffm98J:www.di.uniovi.es/~cueva/investigacion/tesis/JuanRamon.ppt+El+desarrollo+de+software+nunca+ha+sido+tan+complejo+como+lo+es+ahora.+Los+desarrolladores+de+software+trabajan+intensivamente+con+el+conocimiento&hl=es&ct=clnk&cd=1&gl=cu&client=firefox-a
www.infor.uva.es/~felix/datos/priii/tr_patrones.pdf
www.zdnet.co.uk/tsearch/Software+development.htm
es.tldp.org/Presentaciones/200211hispalinux/gregorio2/progm-ext-soft-libre-html/
www.ati.es/gt/LATIGOO/OOp96/Ponen7/atio6p07.html
http://www.ati.es/gt/LATIGOO/OOp96/Ponen7/atio6p07.html
http://www.sc.ehu.es/jiwdocoj/remis/docs/aseguracal.htm
http://www.cert.org/books/secure-coding/moreinfo.html
www.cert.org/books/secure-coding/
http://www.internetnews.com/dev-news/article.php/3415121
www.gnu.org/gnu/thegnuproject.es.html
http://atlas.puj.edu.co/ftp/cursos/estandares/stdcod.ps.gz.
www.gnu-pascal.de/h-gpcs-es.html
http://www.ldc.usb.ve/~teruel/ci4712/clases2000/refactorizacion.html
forum.mambo-foundation.org/showthread.php?t=4234
www.fi.uba.ar/laboratorios/lie/Publicaciones/Plantesismodelo.pdf
http://www.agile-spain.com/agilev2/entrevista_a_martin_fowler
agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf

Referencias Bibliográficas.

- *ALEXANDER, C. An Introduction for Object-Oriented Designers, 1993.*
- *BRIAND, L. Property-based Software Engineering Measurement 1996. p.*
- *BUSCHMANM. Catálogos de Patrones: , 1996.*
- *DODANI. Mejores Prácticas de Programación. 2003. p. FOWLER, M. Refactoring Home Page. 2000. p.*
- *GAMMA. Patrones de diseño., 1995. p.*
- *HERRERA, L. Industria cubana del software busca mayor efectividad en el mercado Gramma Internacional, 2004.*
- *IEEE. IEEE standard for a software quality metrics methodology, 1990a.*
- *---. Pruebas, 1990b.*
- *MARQUEZ, Y. C. Y. J. M. Language Independent Metric Support towards Refactoring Inference, 2001.*
- *MCCABE. McCabe Object-Oriented Metrics Tool 1994.*
- *MINREX, C. Discurso pronunciado por el Comandante de la Revolución, Ramiro Valdés Menéndez, Ministro de la Informática y las Comunicaciones en el Acto Inaugural de la XII Convención y Expo. Internacional, Informática 2007, 2007.*
- *OPDYKE. Refactoring, 1992.*
- *Patrones de Diseño, 2001.*
- *PEREZ, J. R. P. Clasificación de usuarios basada en la detección de errores usando técnicas de procesadores de lenguaje., 2006. p.*
- *PRESSMAN, R. Ingeniería de software. Un enfoque práctico. 1998. p.*