Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

P.I. 155 - P.F. 175

REGLAS DE CALIDAD PARA LA CODIFICACIÓN ESTANDARIZADA EN LENGUAJE C: UNA PROPUESTA PARA LA ENSEÑANZA A NIVEL SUPERIOR

Quality rules for Coding Standard in Language C for under graduates students

Dr. Edgar Danilo Domínguez Vera. danilo.uanl71@gmail.com
M.C. Valentín Belisario Domínguez Vera. vdominguezv@hotmail.com
M.C. Arturo del Ángel Ramírez. arturo.delan@uanl.edu.mx
M.C. José Antonio Moreno Barrios. joamoreno@prodigy.net.mx

RESUMEN.

El presente artículo es una investigación documental que reúne las recomendaciones de Deitel y Humphrey para generar código estandarizado en Lenguaje C. Las notaciones Húngara, Camello, Pascal y de Guión Bajo fueron las primeras propuestas para estandarizar los nombres de los identificadores en los programas. Las recomendaciones y las notaciones se combinan para dar lugar a otras sugerencias que pueden ser aplicadas en la enseñanza de la programación de computadoras. El objetivo es que los estudiantes desarrollen sus habilidades de programación de una manera pedagógica, de fácil y rápida asimilación del conocimiento.

Los beneficios de la codificación estandarizada y su impacto en los costos de mantenimiento del software, la productividad de los Ingenieros de Software y la competitividad de las empresas desarrolladoras están ampliamente documentados. Pero también lo está la reticencia, por parte de los Ingenieros de Software, para apegarse a estos estándares. El cambio de cultura debe hacer desde que el recurso humano está en formación profesional, es decir, desde la docencia.

Uno de los requisitos para que las empresas que desarrollan software, obtengan ciertas certificaciones es que cuenten con documentos de codificación estandarizada. Es hora de que la Facultad de Ingeniería Mecánica y Eléctrica cuente con un documento consensado, entre los docentes, que permita estandarizar, dentro de lo posible, la enseñanza de la programación de computadoras y sirva de base para los organismos que acreditan a los programas de estudio.

ABSTRAC.

This documental research gathers Deitel and Humphreys' recommendations for coding standard in C. Hungarian, Camel, Pascal, and Underscore conventions are the first proposals to have standardized names for identifiers. Recommendations and conventions are combined to have other suggestions to be taught in programming courses. Our goal is that student can develop their programming skills in a pedagogical way, easy and quickly assimilation of knowledge.

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Coding standards benefits and its impact to maintaining cost, Software Engineer's productivity and application software enterprises competitiveness are widely documented. So it is, Software Engineer's reluctance to obey such documents. If we want to change this attitude, we must do it when the under graduate students are in school.

Documents for coding standard are a requirement to get some certifications for software developer's enterprises. It is time for Facultad de Ingeniería Mecánica y Eléctrica, to get an agreed document, between professors, to standard education, as possible, in programming courses. Also, such document can be used for organizations that grant accreditation to study plans in software technology.

PALABRAS CLAVE: Codificación Estandarizada.

KEY WORDS: Coding Standard.

INTRODUCCIÓN.

Los beneficios que tiene la codificación estandarizada en la industria del desarrollo de software son diversos (Wang, Wang, Li, Li, & Du, 2010). Algunas empresas tienen estos documentos porque es un requisito para obtener certificaciones (Rose, 2011), pero se ha observado que los ingenieros de software tienden a no tomarlos en cuenta al momento de hacer su trabajo (Li, 2006). Se puede inferir que este descuido se debe a la falta de información de los desarrolladores de los beneficios de la codificación estandarizada, situación que puede corregirse desde su formación profesional. No obstante lo anterior, pocos son los libros de texto de programación de computadoras que abordan estos temas para acostumbrar a los estudiantes, y futuros ingenieros de software, a adherirse a estos estándares.

Algunos beneficios de la codificación estandarizada es el mejoramiento de la comunicación en los equipos de desarrollo de desarrollo de software, reduce los errores de programación y mejora la calidad del software (Li, 2006). Lo anterior repercute en la competitividad de las empresas de software (Wang, Wang, Li, Li, & Du, 2010) y en la productividad de sus trabajadores porque se mejora su facilidad de mantenimiento teniendo esto impacto en la reducción de los costos de mantenimiento (Hegedüs, 2013) (Pressman, 2010, pp. 11, 49).

El presente artículo es una investigación documental de las recomendaciones que hacen Deitel & Deitel (2004) y Humphrey (2009) para el establecimiento de las reglas de calidad para la codificación estandarizada en lenguaje C. El primer objetivo es tomar todas esas recomendaciones y ajustarlas de manera pedagógica para que la asimilación del aprendizaje sea más fácil para los estudiantes a nivel licenciatura. Un segundo objetivo es concientizar a los cuerpos académicos sobre la importancia de contar con documentos que fomenten la codificación estandarizada desde que el recurso humano está en formación.

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Finalmente, este artículo es un llamado a los organismos que acreditan carreras en tecnologías de software, para que consideren la conveniencia de solicitar, los documentos de codificación estandarizada, como un requisito para obtener la acreditación.

El presente esquema estandarizado puede ser una guía quienes publiquen material académico tendiente a enseñar el uso de Lenguaje C y para la creación de una herramienta que permita medir el tamaño del software, principalmente cuando la medida de medición son las líneas de código (LOC) (Humphrey, 2009, pág. 50). Al igual que The Motor Industry Software Reliability Association (MISRA) (Takai, Kobayashi, & Agusa, 2001) emite lineamientos para el desarrollo de software para componente electrónicos usados en la industria automotriz; nada impide que la comunidad de académicos de la informática en México publique un documento semejante tendiente a fomentar las mejores prácticas para la enseñanza de la programación.

METODOLOGÍA.

La presente investigación documental se basa en la lectura de libro de Deitel & Deitel (2004) de la que se han extraído las recomendaciones para los errores comunes de programación, buenas prácticas de programación, tips para prevenir errores, tips de rendimiento, tips de portabilidad y observaciones de ingeniería de software. Hemos tomado lo mejor de ellas, las hemos combinado con la plantilla de codificación estandarizada de Humprey (2009) para ofrecer estas reglas de calidad para la codificación estandarizada aplicadas a la enseñanza del lenguaje C (p.50-52).

Esta investigación también se sustenta en la observación que durante más de 20 años se ha hecho en las aulas de la Facultad de Ingeniería Mecánica y Eléctrica de la Universidad Autónoma de Nuevo. La cual no deja de ser subjetiva pero es igualmente valiosa en términos cualitativos. Este artículo carece de mediciones estadísticas pero consideramos que es el camino para posteriores investigaciones cuantitativas.

La presente propuesta comienza con el establecimiento de un formato general de un programa en lenguaje C, donde sus elementos no cambien de ubicación y siempre se haga bajo esta misma plantilla. Esto evitará ambigüedades en los estudiantes quienes en sus primeros contactos con la programación se desorientan cuando estos elementos aparecen o desaparecen de un programa a otro o simplemente cambian de lugar. Después de esto, la propuesta continúa con recomendaciones para dar nombre a constantes, variables y funciones definidas por el usuario. Se concluye con otras sugerencias cuyo objetivo es evitar que el código luzca amontonado y de difícil lectura para el programador o quien vaya a darle mantenimiento al software.

ANTECEDENTES

La relación que existe entre los costos de mantenimiento y la mantenibilidad del software ha hecho que los investigadores dediquen esfuerzos para encontrar aquellos elementos permitan disminuir esos costos (Takai, Kobayashi, & Agusa, 2001).

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Dentro de esas líneas de investigación están las prácticas al momento de escribir código. Un lugar especial ocupa las convenciones para dar nombres a los identificadores: variables, constantes y funciones definidas por el usuario, clases, métodos, etc.

Las 4 notaciones más conocidas son la húngara, pascal, camello y de guión bajo (Wang, Wang, Li, Li, & Du, 2010). En todas se ellas se procura que los nombres de los identificadores sean significativos, esto es, que el significado denotativo del conjunto de palabras utilizadas para darle nombre al identificador, ayuden a inferir qué es lo que hace o cuál es la funcionalidad que tiene.

La notación húngara consiste en utilizar las primeras letras de los nombres de los identificadores como un prefijo, en letras minúsculas, que denota el alcance y tipo del identificador. Posterior al prefijo, se utilizan palabras significativas cuyas primeras letras deben estar con mayúscula y el resto en minúscula. Ejemplo*: iNumeroEmpleado*. La i minúscula indica que la variable es entera (Wang, Wang, Li, Li, & Du, 2010).

La notación camello consiste en utilizar varias palabras significativas, en la primera de ellas todas las letras son en minúscula. De la segunda palabra en adelante, la primer letra debe estar en mayúscula y el resto en minúscula, como se muestra en el ejemplo: *imprimirComprobantePago()* (Wang, Wang, Li, Li, & Du, 2010).

La notación pascal es similar a la camello, solo que desde la primer palabra significativa, la primer letra de cada palabra es mayúscula, como se muestra en el ejemplo: *NumeroEmpleado, ImprimirComprobantePago()* (Wang, Wang, Li, Li, & Du, 2010) (Deitel & Deitel, 2004, pág. 29).

La notación de guión bajo utiliza este símbolo para separar a las palabras significativas, con letras en minúscula, que son el nombre del identificador; (Wang, Wang, Li, Li, & Du, 2010) (Deitel & Deitel, 2004, pág. 29) como se muestra en el ejemplo:
imprimir_comprobante_pago, comisiones_totales.

Deitel & Deitel (2004) señalan que la aplicación consistente de convenciones mejoran de manera importante la claridad del programa (p. 53). Las características de un buen programa es que son fáciles de entender, fáciles de modificar y arrojan los resultados correctos. Por tanto, ellos sugieren que al escribir código, el ingeniero de software debe buscar la claridad de los programas. Argumentan que a veces vale la pena perder un poco de eficiencia en cuanto al uso de la memoria o del procesador, a favor de la creación de programas más claros. Que en ocasiones, las consideraciones relacionadas con el rendimiento —de los recursos computacionales— se alejan demasiado de las consideraciones para lograr la claridad (Deitel & Deitel, 2004, pág. 185).

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Deitel & Deitel (2004) se refieren a la claridad del programa como la acción que evita que el código luzca amontonado, esto mediante la debida identación o sangrado del texto del programa (p.53). Un código amontonado es aquel no tiene suficientes espacios en blanco (Humphrey, 2009, p. 51) por lo que se dificulta al ojo humano percibir fácilmente los elementos del programa.

Sin embargo, nosotros creemos que la claridad de los programas también puede referirse al hecho de un código que sea fácilmente entendible por alguien que no lo programó pero que tiene conocimiento avanzados del lenguaje de programación en cuestión.

Los efectos positivos de la codificación estandarizada se ven en el mantenimiento al que todo software se ve sometido durante su vida útil. Disminuir de los costos de mantenimiento del software, mejorar la precisión de las estimaciones y aumentar en la productividad de las personas que lo construyen, es otro los de beneficios de contar con un documento de esta naturaleza (Wang, Wang, Li, Li, & Du, 2010) (Hegedüs, 2013).

La opcionalidad y disposición de los elementos en un código de Lenguaje C hace que su aprendizaje sea desconcertante para quienes empiezan a utilizarlo, máxime cuando es el primer lenguaje de programación que aprenden en su vida. Este artículo propone reglas extras a las ya dispuestas en la sintaxis de este lenguaje de programación. En un principio puede parecer rígida y muy predeterminada, pero al final de cuentas permitirá que quienes empiecen a aprenderlo, lo hagan de manera más sencilla y con la posibilidad de sentirse más cómodos en la adquisición de nuevo conocimiento.

La creación de este tipo de recomendaciones no es ninguna novedad pues *The Motor Industry* Software *Reliability Association* (MISRA) ha buscado promover las mejores prácticas para el desarrollo de sistemas electrónicos de seguridad que están empotrados en los vehículos terrestres. Para esto publicaron en 1994, la primera versión del subconjunto de reglas para la programación en lenguaje C para sistemas empotrados. La tercera y última versión de este documento fue publicada en el año 2012 y está disponible en http://www.misra-c.com/.

Escribir programas en C no garantiza la portabilidad en todas las plataformas tecnológicas, existen muchos problemas entre los diferentes compiladores de C. Con frecuencia los programadores se enfrentarán directamente con las variaciones entre compiladores y computadoras. Por lo que se recomienda que cuando haya alguna duda de cómo funciona una característica de lenguaje C, realice un programa sencillo para que vea lo que sucede (Deitel & Deitel, 2004, pág. 16).

Las propuestas aquí presentadas no pretenden apegarse a alguna notación específica para dar nombres a identificadores, sino más bien se toman ideas de ellas para aceptar aquéllas pueden influir en la facilidad de transmisión del conocimiento hacia personas que inician en la programación de computadoras

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

ESTRUCTURA GENERAL DE UN PROGRAMA EN LENGUAJE C.

La codificación en lenguaje C permite que la disposición y la opcionalidad de los elementos que lo componen puedan variar de manera considerable. Esto hace que los principiantes puedan desconcertarse y por tanto considerar como difícil el aprendizaje del mismo. Por elementos vamos a considerar a las librerías de la biblioteca estándar, las directivas de preprocesador, la declaración de constantes, la declaración de variables, la declaración de funciones, sus llamadas y sus definiciones, etc.

En esta sección proponemos un orden predeterminado de dichos elementos y además se le agrega un comentario previo al elemento. Se recomienda que cuando un elemento no se escriba, se conserve el comentario para dejar en claro que ahí debe o puede ir dicho elemento, en este caso, agregue un comentario señalando que en ese programa no existe ese elemento.

La disposición de los elementos que componen a un programa en Lenguaje C debe hacerse de acuerdo al siguiente formato. Se recomienda separar las declaraciones y las instrucciones ejecutables de una función mediante una línea en blanco, para resaltar donde terminan las declaraciones y donde comienzan las instrucciones ejecutables (Deitel & Deitel, 2004, pág. 29).

Muchos programas pueden dividirse de manera lógica en tres fases: una fase de inicialización que especifica el valor inicial de las variables del programa; una fase de procesamiento que introduce los valores de los datos y ajusta las variables del programa de acuerdo con ello; y una fase de terminación que calcula e imprime los resultados finales (Deitel & Deitel, 2004, pág. 63).

Utilice una línea de comentario con al menos 60 asteriscos para separar la definición de la función main de las demás definiciones de funciones creadas por el usuario, y entre las mismas funciones. Esto ayudará a visualizar donde empieza y dónde termina cada función.

```
/* Nombre del programa.c  */
/* Descripción breve del programa  */
/* Librerías de la biblioteca estándar de C  */
#include <archivo.h>

/* Directivas de preprocesador  */
#define CONSTANTES valor

/*  Variables globales, estructuradas o ennumeradas */
TipoDeDato NombreVariable;

/*  declaracion de funciones o prototipos */
TipoDeDatoSalida NOMBRE_FUNCION( TipoDeDato ArgumentoDeEntrada, TipoDeDato
ArgumentoDeEntrada );
```

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

```
/* La función principal del programa */
main()
{
    /*  Variables locales a main() */
    TipoDeDato NombreVariable

    Cuerpo de la función main() /* escriba las instrucciones ejecutables del programa */
    NOMBRE_FUNCION( ArgumentoDeEntrada, ArgumentoDeEntrada ); /* llamar a la función */
} /* Fin función main */

/* ****************************************************************** */

/* Definición de las funciones */
/* Comentario que describa el propósito de la función */
TipoDeDatoSalida NOMBRE_FUNCION( TipoDeDato ArgumentoDeEntrada, TipoDeDato
ArgumentoDeEntrada )
{
    /* Variables locales de la función NOMBRE_FUNCION  */
    Tipo NombreVariable
    Cuerpo de la función NOMBRE_FUNCION /* escriba las instrucciones de la función */
    return( parámetro o argumento de salida ); /* escriba el argumento que regresa la función */
} /* Fin funcion NOMBRE_FUNCION  */
```

IDENTIFICADORES PARA VALORES CONSTANTES.

De acuerdo al orden propuesto de los elementos en un programa de lenguaje C, veremos los identificadores para 1) valores constantes, 2) valores variables y 3) funciones definidas por el usuario. En los tres casos elija identificadores de 31 caracteres o menos. Esto ayudará a garantizar las portabilidad y evitará algunos problemas sutiles de programación (Deitel & Deitel, 2004, pág. 29). Otorgue a dichos identificadores nombres significativos para que sean programas autodocumentados (Deitel & Deitel, 2004, pág. 29).

Los valores constantes pueden almacenarse en un identificador mediante la cláusula **define**. Estos valores no pueden cambiar de valor durante la ejecución del programa. Ejemplos como el valor de PI, E o la aceleración que experimenta un cuerpo en caída libre, etc.

Declare estos identificadores otorgándoles un nombre significativo, es decir, que con el puro nombre se pueda o ayude a deducir a cuál es su función. Si el nombre de la constante requiere un número, escríbalo contiguo a la letras. Dele preferencia a usar el número al final del nombre.

Para dar nombre a las constantes utilice únicamente letras mayúsculas; en constantes con más de dos palabras utilice guión bajo (Deitel & Deitel, 2004, pág. 184). Vea la tabla 1.

Para constantes de enumeración utilice solo letras mayúsculas (Deitel & Deitel, 2004, págs. 146, 378). Tanto para la constante enumerada como para sus identificadores.

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Asignar un valor a una constante de enumeración después de que se define es un error de sintaxis (Deitel & Deitel, 2004, págs. 146, 378).

Tabla 1. Ejemplos aceptados de nombres de constantes

PI	GRAVEDAD_METRICA
GRAVEDAD	GRAVEDAD_METRICA1
VALOR_MAXIMO	GRAVEDAD_METRICA2
VALOR_MINIMO	GRAVEDAD_ANGLOSAJON
IMPUESTO_VALOR_AGREGADO	GRAVEDAD1_METRICA
GRAVEDAD_METROS_SEGUNDOS2	GRAVEDAD2_METRICA
GRAVEDAD_PIES_SEGUNDOS2	ERROR_ESTADISTICO

Ejemplos aceptados para identificadores enumerados con valores constantes

enum DIAS { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO }

IDENTIFICADORES PARA VALORES VARIABLES.

Los valores variables pueden almacenarse en un identificador utilizando los tipo de datos predefinidos por lenguaje C como *int, float, double o char.* Los identificadores pueden incluir letras, números y guiones bajos, pero en estas reglas para la codificación estandarizada se disponen los siguientes lineamientos.

Otorgue a los identificadores para valores variables un nombre significativo (Deitel & Deitel, 2004, pág. 29), es decir, que con el puro nombre se pueda o ayude a deducir a qué se refiere o cuál es su funcionalidad. Evite abreviaciones o variables de una sola letra (Humphrey, 2009, p. 51).

La primer letra de la palabra debe ser una letra en mayúscula, las demás en minúscula. Cuando dos palabras describan mejor a una variable, la primer letra de la segunda palabra deberá ser mayúscula y el resto en minúscula (Deitel & Deitel, 2004, pág. 29). Si el nombre de la variable requiere un número, escríbalo contiguo a las letras. Dele preferencia a usar el número al final del nombre.

A los identificadores que sean variables apuntadores coloque el prefijo ptr (pointer) en minúscula para hacer más claro que estas variables son apuntadores y, por lo tanto, que deben manipularse de manera apropiada. (Deitel & Deitel, 2004, pág. 235). La primer letra de la variable apuntador que sea mayúscula. Ejemplo **ptrNotaRemision.**

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Las variables que apunten a un archivo de datos comience con ptf (pointer to file), en minúscula, el siguiente caracter deberá ser mayúscula. Ejemplo: **ptfProveedores**

Cuando genere un tipo de estructura utilizando struct, unión o typedef struct siempre proporcione un nombre del registro, el cual es conveniente para que posteriormente se declaren nuevas variables correspondientes a la estructura. Otorgue un nombre de registro significativo para ayudar a autodocumentar al programa (Deitel & Deitel, 2004, pág. 357). Agregue el prefijo en minúscula reg. Si otorga una etiqueta al registro agregue el prefijo etq en minúscula. La primer letra del nombre del registro o la etiqueta deberá ser mayúscula y las siguientes minúsculas. Ejemplo **regCedulaProfesional, etqPagoBruto.**

Cuando utilice el nombre del registro para declarar nuevas variables correspondientes a la estructura. Agregue el prefijo vtr en minúscula, que significa variable tipo registro. La primer letra de la variable tipo registro será mayúscula y las demás en minúscula. Ejemplo **vtrNotaVenta.**

Cuando utilice arreglos de memoria utilice los prefijos au para arreglos unidimensionales, ab para arreglos bidimensionales y am para arreglos multidimensionales. Vea la tabla 2.

Tabla 2. Ejemplos aceptados de nombres de variables

Matricula	Salario1Parcial
Nombre	*ptrValorX
Total1	auVector[100]
ContadorPares	abMatriz[10] [10]
PagoNeto1	auNombre[100] [40]

```
/* variables en una estructura o campos */
typedef struct {
                  int NumeroDeEstado;
                  char Descripcion[41];
                  long int Poblacion;
                 } regCenso2010;    /* nombre del registro */

FILE *ptfCenso2010; /* puntero al archivo */

regCenso2010 vtrCenso2010; /* variable tipo registro */
```

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

IDENTIFICADORES PARA FUNCIONES DEFINIDAS POR EL USUARIO

C es un lenguaje de programación orientado a funciones. Las funciones como printf(), scanf(), if(), while(), do{…}while(), for(), pow(), sqrt(), strcmp(), se les llama predefinidas porque la acción que realizan está especificada en las librerías de la biblioteca estándar de C, que se incluyen en la codificación de un programa mediante la directiva de preprocesador **#include.**

Lenguaje C permite que cada programador invente sus propias funciones a las que se les denomina funciones definidas por el usuario. A los programas que no utilizan funciones definidas por el usuario, Deitel & Deitel (2004) les llama monolíticos (pag.159), por el contrario, a los programas que utilizan dichas funciones les llama programas funcionalizados o modularizados (por módulos) (pag.128 y 159).

Lo anterior permite aceptar la siguiente recomendación: Funcionalizar los programas de manera sencilla y jerárquica, promueve la buena ingeniería de software. Sin embargo, tiene un precio. Un programa altamente funcionalizado, comparado con un programa monolítico (es decir, de una sola pieza) sin funciones, hace un gran número de llamadas a funciones y esto consume tiempo de ejecución en el procesador de la computadora. Sin embargo, aunque los programas monolíticos se ejecutan mejor, son más difíciles de programar, probar, corregir, mantener y evolucionar (Deitel & Deitel, 2004, pág. 159).

Cuando se manejan funciones definidas por el usuario se debe considerar tres cosas 1) la declaración de la función o prototipo. 2) la llamada a la función y 3) la definición de la función. Estas tres elementos deben concordar en cantidad, tipo, orden de argumentos y parámetros, y en el tipo del valor de retorno (Deitel & Deitel, 2004, pág. 134).

Por lo general la declaración de la función o prototipo se escribe al inicio del programa, antes de comenzar la definición de la función main. El prototipo consiste de tres elementos 1) El tipo de dato del argumento que retorna (también llamado argumento de salida), solo puede devolver un argumento. 2) El nombre de la función. 3) Los argumentos de entrada y su tipo de datos; uno, varios o ningún argumento. Olvidar el punto y coma al final del prototipo de la función es un error de sintaxis (Deitel & Deitel, 2004, pág. 135). Elegir nombres significativos de funciones y de argumentos hace que los programas sea más legibles, y ayuda a evitar el uso excesivo de comentarios (Deitel & Deitel, 2004, pág. 133).

La llamada de la función se escribe dentro de *main* o dentro de la definición de otra función y consiste en escribir el nombre de la función y los nombres de los argumentos de entrada.

La definición de la función o funciones se escriben después de la llave que cierra a main; consiste en el encabezado de la función y el cuerpo de la función. El encabezado contiene los mismos tres elementos de la declaración de la función pero no debe terminar en punto y coma, sino que se sustituye por las llaves que abren y cierran (Deitel & Deitel, 2004, págs. 132, 133). Las instrucciones que se escriben entre las llaves es a lo que se llama cuerpo de la función

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

En los programas que contienen muchas funciones, a menudo *main* se implementa como un grupo de llamadas a funciones que realizan el grueso del trabajo del programa. Cada función debe limitarse a realizar una sola tarea bien definida; el nombre de la función debe expresar de manera clara dicha tarea. Esto facilita la abstracción y promueve la reutilización de software. Si usted no puede elegir un nombre conciso que exprese lo que hace la función, es posible que su función intente realizar demasiadas tareas. Por lo general, es mejor dividir dicha función en varias funciones pequeñas. (Deitel & Deitel, 2004, pág. 131).

Una función no debe ser más grande que una página. Mejor aún, una función no debe ser más grande que la mitad de una página. Las funciones pequeñas promueven la reutilización del software. Los programas deben escribirse como colecciones de funciones pequeñas. Esto hace que los programas sean fáciles de escribir, depurar, mantener y modificar. Una función que tiene un gran número de parámetros podría realizar demasiadas cosas. Considere dividirla en funciones más pequeñas para realizar tareas separadas. El encabezado de la función debe caber, si es posible es una sola línea de código (Deitel & Deitel, 2004, pág. 133).

En la codificación de los programas, incluya la declaración de la función o prototipos de todas las funciones para aprovechar las capacidades de verificación de tipos de C. Utilice la directiva de preprocesador **#include** para obtener los prototipos de función correspondientes a las funciones de la biblioteca estándar, a partir de los encabezados en las bibliotecas apropiadas, o para obtener encabezados que contengan prototipos de funciones desarrolladas por usted y/o sus compañeros de grupo (Deitel & Deitel, 2004, pág. 135). El compilador ignora los nombres de los parámetros en los prototipos de la función, sin embargo, para efectos de documentación, inclúyalos (Deitel & Deitel, 2004, pág. 135).

Es un error finalizar una directiva de preprocesador **#define** o **#include** con un punto y coma, ya que no son instrucciones en C (Deitel & Deitel, 2004, pág. 183).

El formato para la declaración de la función o prototipo es
*TipoDeDatoSalida***NOMBRE_FUNCION** *(TipoDeDato ArgumentoDeEntrada, TipoDeDato ArgumentoDeEntrada);*

Ejemplos de declaración de función, la cual debe terminar con punto y coma.

/* Declaración de funciones o prototipos */

```
void MENU();
void INICIAR();
regAlumno ALTAS( regAlumno vtrAlumno );
regAlumno BAJAS( regAlumno vtrAlumno );
regAlumno CAMBIOS( regAlumno vtrAlumno );
int BUSCAR( char Matricula[8] );
int BUSCAR_PANTALLA( char Matricula [8] );
void BORRAR_REGISTRO( char Matricula [8] );
void CAMBIAR_REGISTRO( regAlumno vtrAlumno, char Mat[8] );
int VALIDAR_MATRICULA( char Matricula [8] );
char CAMBIAR_LETRAS( regAlumno vtrAlumno, char Variable[30] );
void IMPRIME( regAlumno vtrAlumno );
```

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

El formato para llamada a la función es
NOMBRE_FUNCION(*ArgumentoDeEntrada, ArgumentoDeEntrada*);

En la llamada a la función, no escriba el TipoDeDatos que devuelve la función, ni el TipoDeDatos de los parámetros o argumentos.

Ejemplos de llamadas a la función, la cual debe terminar con punto y coma.

```
/* Llamada de funciones */

MENU();
INICIAR();
ALTAS( vtrAlumno );
BAJAS( vtrAlumno );
CAMBIOS( vtrAlumno );
BUSCAR( Matricula );
BUSCAR_PANTALLA(Matricula );
BORRAR_REGISTRO(Matricula );
CAMBIAR_REGISTRO(vtrAlumno, Matricula );
VALIDAR_MATRICULA(Matricula );
CAMBIAR_LETRAS( vtrAlumno, Variable );
IMPRIME( vtrAlumno );
Encontrado = BUSCAR(Matricula );
```

El formato para la definición de la función

*TipoDeDatoSalida***NOMBRE_FUNCION**(*TipoDeDato ArgumentoDeEntrada, TipoDeDato ArgumentoDeEntrada*)
/* encabezado de la función */

```
{
   /* Variables locales de la función NOMBRE_FUNCION  */
   Tipo NombreVariable;
   Cuerpo de la función NOMBRE_FUNCION /* escriba las instrucciones de la función */
   return( parámetro o argumento de salida ); /* escriba el argumento que regresa la función */
} /* Fin función NOMBRE_FUNCION */
```

Ejemplos de definición de la función de la función, la cual NO debe terminar con punto y coma (Deitel & Deitel, 2004, pág. 133).

```
void MENU()
{
  /* Escriba las instrucciones que forman parte del cuerpo de la función */
  return;
} /* Fin funcion MENU */

void INICIAR()
{
  /* Escriba las instrucciones que forman parte del cuerpo de la función */
  return;
} /* Fin funcion INICIAR */

regAlumno ALTAS( regAlumno vtrAlumno )
{
  /* Escriba las instrucciones que forman parte del cuerpo de la función */
  return( nombre del parámetro o argumento de salida );
} /* Fin funcion ALTAS */

int BUSCAR(char Matri[8])
{
  /* Escriba las instrucciones que forman parte del cuerpo de la función */
  return( nombre del parámetro o argumento de salida );
} /* Fin funcion BUSCAR */

regAlumno BAJAS( regAlumno vtrAlumno )
{
  /* Escriba las instrucciones que forman parte del cuerpo de la función */
  return(nombre del parámetro o argumento de salida );
```

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

```
} /* Fin funcion BAJAS */

void BORRAR_REGISTRO( char Matri[8] )
{
  /* Escriba las instrucciones que forman parte del
cuerpo de la función */
  return;
} /* Fin funcion BORRAR_REGISTRO */

regAlumno CAMBIOS( regAlumno vtrALumno )
{
  /* Escriba las instrucciones que forman parte del

cuerpo de la función */
  return( nombre del parámetro o argumento de
salida );
} /* fin funcion CAMBIOS */

void CAMBIAR_REGISTRO( regAlumno
vtrAlumno, char Matri[8] )
{
  /* Escriba las instrucciones que forman parte del
cuerpo de la función */
  return;
} /* Fin funcion CAMBIAR_REGISTRO */

int BUSCAR_PANTALLA( char Matri[8] )
{
  /* Escriba las instrucciones que forman parte del
cuerpo de la función */


 return(nombre del parámetro o argumento de
salida );
} /* Fin funcion BUSCAR_PANTALLA */

int VALIDAR_MATRICULA( char Matri[8] )
{
  /* Escriba las instrucciones que forman parte del
cuerpo de la función */
  return( nombre del parámetro o argumento de
salida );
} /* fin funcion VALIDAR_MATRICULA */

void IMPRIME( regAlumno vtrAlumno )
{
  /* Escriba las instrucciones que forman parte del
cuerpo de la función */
  return;
} /* Fin funcion IMPRIME */
```

Para identificar a las funciones definidas por el usuario otórgueles un nombre significativo (Deitel & Deitel, 2004, pág. 133), es decir, que con el puro nombre se deduzca o ayude a deducir qué es lo que hace la función. Utilice letras mayúsculas. En los nombres de las funciones con más de una palabra, utilice guión bajo para ayudar a su visualización. Evite nombres de funciones con una sola letra o abreviaciones. Si el nombre de la función requiere un número, escríbalo contiguo a las letras. Dele preferencia a usar el número al final del nombre. Vea tabla 3. En la declaración de la función y en la definición de la función, aun cuando un tipo de retorno omitido devuelve de manera predeterminada a un int, siempre establezca el tipo de retorno de manera explícita (Deitel & Deitel, 2004, p. 133).

Si la función no devuelve valores, termine la función con **return**.

Toda definición de función debe ser precedida por un comentario que describa el propósito de la función (Deitel & Deitel, 2004, pág. 25).

Agregue un comentario a la línea que contiene la llave derecha, } , que cierra toda función, incluyendo a main(). (Deitel & Deitel, 2004, pág. 26), para indicar que se ha finalizado la definición de la función.

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Tabla 3. Ejemplo de identificadores aceptados para funciones definidas por el usuario	
AREA()	SALARIO_NETO()
ALTAS()	PRECIO_VENTA()
TOTAL()	OPCION1()

Ejemplo de identificadores NO aceptados para funciones definidas por el usuario
Una sola palabra: Area(), altas(), Total(), factorial(), SalarioNeto, kf()

Dos palabras: SalarioNeto(), PRECIOventa(), SALARIONETO()

Cuando se utilizan funciones definidas por el usuario, se debe considerar que lenguaje C permite definir variables locales y globales. Se debe tener en cuenta que definir una variable como global, en lugar de hacerlo como local, permite que ocurran efectos colaterales, por ejemplo, cuando una función que no necesita acceso a la variable la modifica de manera accidental o maliciosa. En general, debe evitarse el uso de variables globales, excepto en ciertas situaciones con requerimientos especiales de rendimiento (Deitel & Deitel, 2004, pág. 148).

Las variables que se utilizan sólo en una función en particular, deben definirse como variables locales en esa función y no como variables externas (Deitel & Deitel, 2004, pág. 148).

Un error común de programación es utilizar de manera accidental el mismo nombre para un identificador en un bloque interno y en un bloque externo, cuando de hecho, el programador quiere que el identificador del bloque externo se encuentre activo durante la ejecución del bloque interno (Deitel & Deitel, 2004, pág. 148).

Evite nombres de variables que oculten nombres con alcances externos. Esto se puede llevar a cabo simplemente evitando el uso de identificadores duplicados en un programa (Deitel & Deitel, 2004, pág. 148).

ESPACIOS EN BLANCO

Escriba programas con suficiente espaciado, de manera que no aparezca amontonado (Humphrey, 2009, pág. 51).
Coloque un espacio en blanco después de cada coma (,), para hacer que los programas sean más legibles (Deitel & Deitel, 2004, pág. 30).
Coloque espacios en blanco en las librerías y el archivo incluido en el programa.
Deje un espacio en blanco después de los paréntesis abiertos y antes del paréntesis que cierra.

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Ejemplos aceptados de espacios en blanco

```
  #include <stdio.h>
float Fuerza, Masa, Aceleracion;
for( i = 1; i < N; i++ )
Suma = Numero1 + Numero2
printf( “Introduce un valor numérico \n” );
Sumatoria = Entero1 + Entero2;
scanf( “%d”, &Numero1 );
```

Ejemplos NO aceptados de espacios en blanco

```
float Fuerza,Masa,Aceleracion;
for(i=1;i<N;i++)
printf(“Introduce un valor numérico \n”);
Suma=Numero1+Numero2;
 scanf(“%d”,&Numero1);
```

MENSAJES DE SALIDA E INTRODUCCIÓN DE DATOS

Los mensajes que sean de salida al usuario pueden estar en mayúsculas y minúsculas para hacerlo más claro al usuario.
El último caracter que imprima cualquier función de impresión debe ser una línea nueva (\n). (Deitel & Deitel, 2004, p. 26).
Imprima variables, no las sustituya por una operación matemática.
Deje espacios en blanco después del paréntesis izquierdo y antes del derecho de la instrucción printf y scanf.
Use un scanf() para cada variable.

Ejemplos aceptados de mensajes de salida

```
printf(  “Dame un numero entero %d \n ” );
printf(  “La suma es de %d \n ”, Sumatoria );
```

Ejemplos NO aceptados de mensajes de salida

```
printf(“Introduce un valor numérico”);
printf(  “La suma es de %d \n ”, Entero1 + Entero2 );
```

Ejemplo aceptado de introducción de datos

```
scanf( “%d”, &Numero1 );
scanf( “%d”, &Numero2 );
```

Ejemplo No aceptado de introducción de datos

```
  scanf( “%d%d”, &Numero1, &Numero2 );
```

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

EXPRESIONES MATEMÁTICAS.

Cuando realice divisiones con expresiones cuyo de denominador pueda ser CERO, o que el radicando de una raíz cuadrada pueda ser negativo; haga una prueba explícita de este caso y manéjela de manera apropiada de su programa (tal como la impresión de un mensaje de error), en lugar de permitir que ocurra un error fatal. (Deitel & Deitel, 2004, págs. 13, 33, 62).

Coloque espacios a cada lado de un operador binario. Esto hace que el operador resalte, y hace más claro el programa (Deitel & Deitel, 2004, pág. 30).
Los operadores unarios deben colocarse inmediatamente después de sus operandos, sin espacios intermedios (Deitel & Deitel, 2004, pág. 71).

Inicialice las variables al momento de declararlas (Deitel & Deitel, 2004, pág. 69), principalmente aquellas que lo requieran como los contadores o acumuladores (Deitel & Deitel, 2004, pág. 60) o que aparecen en ambas lados del operador de asignación.

No utilice valores flotantes de manera que se asuma una representación precisa, puede provocar resultados incorrectos. No compare la igualdad de valores flotantes (Deitel & Deitel, 2004, pág. 65).

No utilice variables de tipo float o double para realizar cálculos monetarios. La imprecisión de los números de punto flotante puede ocasionar errores que provoquen valores monetarios incorrectos (Deitel & Deitel, 2004, pág. 98).
Cuando utilice paréntesis deje un espacio en blanco antes y después de los operandos.

COMENTARIOS.

Use la barra inclinada hacia la derecha y asterisco para hacer comentarios; deje un espacio en blanco después de /* y antes de */.

Documente el código para que el lector pueda entender su operación (Humphrey, 2009, p. 51).
Los comentarios deben explicar tanto el propósito como el comportamiento del código (Humphrey, 2009, p. 51).

Comente la declaración de variables para indicar su propósito (Humphrey, 2009, p. 51).
Poner comentario a la llave que cierre una función o estructura. (Deitel & Deitel, 2004, pág. 41)
Toda función debe ser precedida por un comentario que describa su propósito (Deitel & Deitel, 2004, pág. 41)

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Ejemplo de un Comentario aceptado
/* Permite dar de alta un registro con campo llave matricula */

/* variable que calcula el pago antes de descontar los impuestos */
int PagoBruto;

Ejemplo de un comentario no aceptado
// no utilizar doble barra para comentarios
/*Permite dar de de alta un registro con campo llave matricula*/

INDENTACIÓN O SANGRIA

La aplicación consistente de convenciones para el sangrado, mejoran de manera importante la claridad del programa. Establezca sangría en el cuerpo de cada función o estructura. Aplique una sangría en cada nivel de llaves del previo. Deje 3 espacios en blanco en la sangría. (Deitel & Deitel, 2004, págs. 26,27, 53) (Humphrey, 2009, p. 51).

Coloque sangrías en el cuerpo de una instrucción if (Deitel & Deitel, 2004, págs. 38, 54).
Si existen muchos niveles de sangrado, cada nivel debe estar sangrado con el mismo número de espacios (Deitel & Deitel, 2004, pág. 54). Apertura y cierre de llaves deben de estar en una sola misma columna y alienados con su correspondiente símbolo.

Una instrucción larga puede distribuirse en varias líneas. Si una instrucción debe separarse a lo largo de varias líneas, elija límites que tengan sentido (como después de una coma, en una lista separada por comas). Si una instrucción se divide en dos o más líneas, coloque sangrías en todas las líneas subsiguientes (Deitel & Deitel, 2004, pág. 38).

Sangría Aceptada de una función o estructura

```
  if( X == 1 )
  {
    Z = Z + 1;
  }
  else
  {
    K = K + 2;
  }  /* fin del if */
```

Sangría No Aceptada de una función o estructura
Ejemplo No aceptado #1:

```
  if(x==1)
  { Z = Z + 1;}
  else
  {  K = K + 2; }
```

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Ejemplo No aceptado #2:

```
if(X==1)
{
Z = Z + 1;
}
else
{
K = K + 2;
}
```

Ejemplo No aceptado #3:

```
if(X==1)
{
     Z = Z + 1;
   }
  else
     {
          K = K +2;
}
```

LÍNEAS DE CÓDIGO.

Aunque está permitido, en un programa no debe haber más de una instrucción por línea. (Deitel & Deitel, 2004, pág. 38).

Cuando las instrucciones que son estructuras if, for, do…while, while, switch, etc. que usan las llaves se les llama bloque (Deitel & Deitel, 2004, pág. 56). Utilice llaves en las estructuras aun y cuando sea una instrucción sencilla.

Evite el uso de instrucciones vacías. Evite colocar punto y coma inmediatamente a la derecha del paréntesis que encierra a la condición de un if() (Deitel & Deitel, 2004, pág. 57).
Procure utilizar la instrucción break en cada caso de la instrucción switch() (Deitel & Deitel, 2004, pág. 102).

Procure utilizar el caso default en las instrucciones swicht() y que esté al final de todos los casos previos. Utilice break aunque no sea necesario (Deitel & Deitel, 2004, pág. 103).
Las instrucciones break y continue violan las normas de la programación estructurada. Minimice su utilización (Deitel & Deitel, 2004, pág. 107).

En expresiones que utiliza el operador lógico: &&, haga que la condición más propensa a ser falsa se encuentre hasta la izquierda. En donde se use el operador ||, haga que la condición más propensa a ser verdadera se encuentre hasta la izquierda, para ayudar a reducir el tiempo de ejecución de un programa (Deitel & Deitel, 2004, pág. 108).

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Convertir un tipo de datos de mayor nivel en la jerarquía a uno de menor nivel, puede modificar el valor del dato (Deitel & Deitel, 2004, pág. 136).

Líneas de código aceptadas
```
printf( “Introduce un valor numérico \n” );
 scanf( “%d”, &Numero1 );
```

Líneas de No código aceptadas
```
  printf( “Introduce un valor numérico \n” ); scanf( “%d”, &Numero1 )
```

CICLOS.

Controle los ciclos con valores enteros. (Deitel & Deitel, 2004, p. 92).
Sangre las instrucciones correspondientes al cuerpo de toda la instrucción de control. (Deitel & Deitel, 2004, pág. 92)

En un ciclo controlado por centinela, la indicación de entrada de datos debe recordar de manera explícita cuál es el valor del centinela. (Deitel & Deitel, 2004, pág. 64).

Proporcione una acción dentro del ciclo que permita que la condición se haga falsa para evitar ciclos infinitos (Deitel & Deitel, 2004, pág. 57). Si el ciclo es controlado por centinela, elija un valor legítimo (Deitel & Deitel, 2004, pág. 61). Si es controlado por contador, permita que la variable controladora con cada vuelta se acerque a la condición de salida del ciclo, es decir, que se haga falsa la condición.

Para un ciclo controlado por contador, utilice el operador relacional <=. Por ejemplo para un ciclo que deba dar diez vueltas use la condición **contador <= 10,** en lugar de **contador < 11** (Deitel & Deitel, 2004, pág. 93).

Dentro de las secciones de inicialización y movimiento de una instrucción for(), sólo coloque expresiones relacionadas con las variables de control (Deitel & Deitel, 2004, pág. 94).
Evite colocar punto y coma inmediatamente a la derecha del paréntesis que cierra una instrucción **for()** o **while()** porque se convierte en una instrucción vacía (Deitel & Deitel, 2004, págs. 94, 104).

Aunque el valor de la variable de control puede modificarse en el cuerpo del ciclo for(), esto puede provocar errores sutiles. Es mejor no cambiarlo (Deitel & Deitel, 2004, pág. 95).
Aunque las instrucciones que preceden a **for()** y las instrucciones del cuerpo de un for(), a menudo se pueden fusionar dentro de un encabezado **for(),** evite hacerlo, ya que esto ocasiona que el programa sea más difícil de leer (Deitel & Deitel, 2004, pág. 96).

Limite el tamaño de los encabezados de las instrucciones de control a una sola línea (Deitel & Deitel, 2004, pág. 96).

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

Incluya las llaves en la instrucción **do…while**, incluso si no son necesarias (Deitel & Deitel, 2004, pág. 104).

Tener demasiados niveles de anidamiento, puede provocar que un programa sea difícil de entender. Como regla general, intente evitar el uso de más de tres niveles de anidamiento. (Deitel & Deitel, 2004, pág. 92).

Use llaves en los ciclos aunque no sean necesarias. (Deitel & Deitel, 2004, pág. 104)

ARREGLOS DE MEMORIA.

Los corchetes que se utilizan para encerrar el subíndice de un arreglo, en realidad se consideran como un operador en C. Los corchetes tienen el mismo nivel de precedencia que el operador de llamadas a función, es decir, el par de paréntesis que se colocan después del nombre de una función para llamar a esa función (Deitel & Deitel, 2004, pág. 179).

No Olvide inicializar los elementos de un arreglo que debieran inicializarse (Deitel & Deitel, 2004, pág. 182).
Proporcionar más inicializadores en un lista de inicialización que elementos en el arreglo, es un error de sintaxis (Deitel & Deitel, 2004, pág. 182).

Definir el tamaño de un arreglo como una constante simbólica hace que los programas sea más escalables (Deitel & Deitel, 2004, pág. 184).
No haga referencias a elementos que se encuentren fuera de los límites del arreglo (Deitel & Deitel, 2004, pág. 186).

Cuando se hace un ciclo en torno a un arreglo, el subíndice del arreglo nunca debe ser menor que cero y siempre debe ser menor que el número total de elementos del arreglo (tamaño - 1). Asegúrese que la condición de terminación del ciclo prevenga el acceso de elementos fuera de ese rango (Deitel & Deitel, 2004, pág. 186).

Proporcione a scanf un arreglo de caracteres lo suficientemente grande para almacenar una cadena escrita mediante el teclado, de lo contrario puede ocasionar la destrucción de los datos del programa y otros errores en tiempo de ejecución (Deitel & Deitel, 2004, pág. 189).

Cuando pase un arreglo a una función, también pase el tamaño del arreglo. Esto ayuda a hacer la función reutilizable en muchos programas. (Deitel & Deitel, 2004, pág. 249)

Año III, No. 06 Julio-Diciembre 2015
ISSN: 2395-9029

CONCLUSIONES.

La presente lista de recomendaciones aquí plasmadas no pretende ser exhaustiva, pero si el principio para concientizar, a los futuros ingenieros de software, sobre una cultura de calidad al momento de crear software.

La industria del software, como cualquier industria, está en constante búsqueda para reducir de los costos de producción mediante procesos que minimicen el consumo de recursos y con alta productividad. Una línea en ese sentido, es la generación código reutilizable y estandarizado.

Uno de los primeros retos es el establecimiento de reglas de calidad y convenciones que se usen consistentemente en la generación de código computacional. También lo será, la socialización de los beneficios entre los ingenieros de software para puedan considerarlos y aceptarlos. La etapa de formación académica es el momento propicio para comenzar a formar la cultura de calidad que requiere el desarrollo de software.

Este artículo abre la posibilidad de que se inicien varias investigaciones académicas más orientadas a confirmar o rectificar la relación entre código estandarizado y la reducción de costos al momento de generar o dar mantenimiento al software.

BIBLIOGRAFIA.

Deitel, D. &. (2004). Cómo Programar en C/C++ y Java . México: Pearson Prentice Hall.

Deitel, H. M., & Deitel, P. J. (2004). Cómo Programar en C/C++ y Java. México: Pearson Prentice Hall.

Hegedüs, P. (2013). Revealing the Efecct of Coding Practices on Software Maintainability. IEEE Conference Publication, 1-8.

Humphrey, W. S. (2009). PSP: A Self-Improvement for Software Engineers. Westford, Massachusetts: Pearson Education, Inc.

Li, X. (2006). Effectively Teaching Coding Standards in Programming. IEEE Conference Publications, 9-14.

Pressman, R. S. (2010). Ingeniería del Software: Un enfoque práctico. México, D.F.: Mc Graw Hill.

Rose, J.-P. (2011). Designing and Cheking Coding Standards for Ada. ACM SIGAda Ada Letter, 13-14.

Takai, Y., Kobayashi, T., & Agusa, K. (2001). Software Metrics Based on Coding Standards Violations. IEEE Conference Publications, 273-278.

Wang, Y., Wang, S., Li, X., Li, H., & Du, J. (2010). Identifier Naming Conventions and Software Coding Standards: A Case Study in One School of Software. IEEE Conference Publication, 1-4.