UNIVERSIDAD DE GUADALAJARA
Centro Universitario de Ciencias Exactas e Ingenierías
División de Tecnologías para la Integración Ciber-Humana

Profesor M. en C. Ing. Luis Alberto Muñoz Gómez
Requerimientos de Valor Agregado en Código Fuente

@ Excelente ortografía en las impresiones de pantalla y comentarios en código fuente.

A. Todos los identificadores de variables y constantes tengan nombres claros y significativos, de modo que sea fácil saber qué dato almacenarán. Usar al menos 3 caracteres por cada palabra incluida en el identificador (por ejemplo “max” en “maxCalif”). Así mismo, declarar las variables de entrada de datos, del tipo de dato acorde al conjunto de valores asignables a este (p. ejem: “totalComputadoras” debe ser de tipo entero y no real). Evitar dislexia en los identificadores.

B. Todo identificador debe ser necesario para la operación del programa.

C. Cumplir con las convenciones para nomenclatura de variables y constantes (variables en minúsculas camelCase, constantes en mayúsculas con separador de palabras “_” y solo los arreglos tengan nombre en plural y, si y solo si se trata de modelar una colección, tal como “prácticas”).

D. Todo valor asignado a un identificador debe ser necesario para la operación de alguna parte del programa.

E. Utilizar constantes simbólicas en lugar de numéricas siempre que sea posible con ellas mejorar el mantenimiento y/o la legibilidad del programa. Los números mágicos 0 y 1 y cualquier otro se reemplacen por constantes simbólicas si y solo si se les puede otorgar un nombre claro y significativo, de lo contrario pueden quedar como números mágicos. En el caso de las constantes cadena, declarar constantes simbólicas para ellas si se trata de nombres de archivos, y también para cualquier otra cadena reutilizable.

F. A excepción de que un requerimiento lo solicite, no incluir la redacción de los requerimientos en el código fuente.

G. Las líneas de código fuente sean máximo de 100 caracteres.

H. Código fuente indicando como comentario en la parte superior de cada archivo entregado: NickName del(la) autor(a), número y título de práctica/actividad/tarea/ejercicio (según corresponda); luego, anotar en el archivo con el código fuente principal (main.c, main.cpp, etc. el que fuese) y solo en el principal, el tiempo invertido en resolver el entregable (contado desde leer el problema hasta resolverlo, a tiempo corrido medido de la forma más precisa posible); escribir el tiempo en el formato "Tiempo=HH:MM", p. ejem "Tiempo=01:25"; en el caso de ser un programa traducido del originalmente implementado escribir de la forma “TiempoTraduccion=HH:MM”, p. ejem: “TiempoTraduccion=00:20”; en el caso de ser la traducción al inglés escribir de la forma “Time=HH:MM”, p. ejem “Time=00:25”; el programa originalmente implementado siempre sea en idioma español, y sin importar cuál lenguaje de programación se aplique como original. Existen programas como la práctica 1, práctica 2, etc. que son evolución del entregable anterior; para estos el tiempo a anotar sea solo el tiempo que haya tomado el agregar las nuevas funcionalidades al programa (incluido el tiempo invertido en leer el documento correspondiente).

I. Cumplir con las convenciones ANSI o Java sobre indentación de código fuente.

J. Código fuente libre de instrucciones anuladas mediante comentarios.
K. El código fuente libre de errores de compilación y advertencias.
L. Código fuente libre de comentarios describiendo el funcionamiento a detalle.
M. Evitar declarar variables que se usen para lectura una sola vez después de definirlas, a excepción de que en la descripción del problema a resolver existan conceptos con el mismo nombre que el usado para la variable.
N. Programa libre de desbordamiento de arreglos en lectura o escritura.
O. Declarar las banderas con un nombre valuable como verdadero/falso o si/no (p. ejem: "mayorDeEdad" y "mesValido"). Evitar comparar banderas contra constante numérica o simbólica.
P. Para iterar arreglos usar nombres como i,j,k, etc. (de un solo caracter) a excepción de que se requiera la variable para su lectura posterior a la estructura de control iterativa.
Q. Toda instrucción sea indispensable para las instrucciones subsecuentes; por ejemplo, en un programa que validara una fecha en base a día, mes y año, si se trata de un día del mes de Enero, sería inapropiado comprobar si se trata de un año bisiesto, pues esto solo sería necesario para caso del mes de Febrero.
R. El código fuente sea portable, es decir, que compile sin errores y sea ejecutable de igual forma, desde una consola Unix o MS-DOS y mostrándose igual en ejecución. Así mismo, evitar de ser posible en C++ usar librerias de C de la forma <stdlib.h> (usar en su lugar su versión C++ <cstdlib>).
S. Dada una serie de estructuras de control if, las cuales verifiquen expresiones repetidas en el programa, *factorizarlas* usando la estructura if anidada.
T. Evitar repetir instrucciones siempre que sea posible *factorizarlas*, colocando el código *factorizado* antes y/o después de la estructura de control selectiva.
U. Evitar el uso de la selectiva if como disparador (if sin else) siempre que sea posible en su lugar una selección canónica (if, else if, else if, …,else). En caso de ser indispensable un disparador sin "else" escribir un comentario en codigo fuente explicando el porqué.
V. Evitar anidar la selectiva if, en el else de otro if, si la primera es todo lo que hay escrito en el else de ese otro if.
W. Evitar el uso del operador relacional == de una misma variable contra 2 ó más enteros siempre que en su lugar se pueda utilizar la estructura de control "switch". Así mismo todo switch cuente con default conteniendo al menos la proposición nula.
X. Evitar repetir la presencia en diversas partes del programa, de los mismos cómputos (como sumas, comparaciones, o generación de cadenas de caracteres), cuando sea posible y apropiado memorizar en una variable el valor izquierdo producido.
Y. Evitar en código fuente repetir la presencia de la misma estructura de control secuencial, siempre que sea posible aplicar la estructura de control iterativa. Así mismo usar la estructura de control iterativa, donde sea apropiado según el problema a resolver.
Z. Usar la estructura de control do-while, donde al menos una línea de código se itere al menos una vez; usar while donde se itere cero o más veces; usar for cuando sea mejor (en uso de líneas de código fuente y/o legibilidad) que usar while o do-while.
AA. Todas las subrutinas tengan nombres claros y significativos, en notación camelCase y de modo que sea fácil saber cuál es su función en el programa. Siempre que sea posible usar la forma verboComplemento (p. ejem: "mostrarMenu()" y "calcularIngresos()"). En el caso de los prototipos, citar los nombres de los parámetros. Evitar dislexia en los identificadores. En el caso de métodos booleanos relativos a banderas en un objeto, sean de la forma ser/estar (p. ejem: Persona.esMayorDeEdad() para la bandera "Persona.mayorDeEdad").
BB. Ejercer la programación modular siempre que sea posible reutilizar código.

CC. El programa principal (main) describa de forma general en sus instrucciones de qué se trata todo el programa, de modo que no se requiera leer el interior de las subrutinas llamadas desde él.

DD. Todas las subrutinas sean máximo de 50 líneas, desde su encabezado y hasta su fin de ámbito.

EE. Evitar crear subrutinas cuando estas no sean reutilizables o no necesarias según el requerimiento anterior. Así mismo, evitar crear subrutinas por utilizar en alguna subrutina, una o más líneas sin código fuente.

FF. Para cada subrutina, declarar todas sus variables locales al principio de ella; lo anterior a excepción de que por eficiencia se justifique declararlas en un sub-ámbito.

GG. Declarar variables globales (o atributos de una clase), siempre que estas representen el estado del programa (o del objeto), es decir, que sería de interés almacenar sus valores en un archivo para luego recuperar el estado del programa desde el archivo.

HH. Evitar declarar variables como campo de un registro, o atributo de una clase, o globales de programa, cuyos datos no formen parte de las propiedades que describan la clase o registro en que se declaren, o bien no representen el estado del registro, o del objeto, o del programa, y que por tanto sea posible una implementación utilizando variables locales y paso de parámetros.

II. Evitar llamadas repetidas hacia una subrutina Y desde una misma subrutina X, cuando se usen los mismos valores como argumentos desde X, y se obtengan de Y los mismos resultados.

JJ. Toda instrucción del programa sea ejecutable, dependiendo del estado del programa, y sin necesidad de recompilarlo.

KK. Todas las clases/registros (y sus atributos/campos) cuenten con un nombre (identificador) claro y significativo, a modo que al leer su declaración sea intuitivo saber para qué se utiliza.

LL. Todo atributo pasivo se encuentre en la parte superior de su clase y declarar uno por cada línea. Ordenar los atributos (por un lado los pasivos y por otro los activos), primero los de mayor ocultamiento y a continuación los de menor ocultamiento.

MM. Todo atributo pasivo sea privado, a menos que exista una implementación que por eficiencia justifique (con comentarios) un menor ocultamiento de información, pero que igualmente el diseño cumpla con la propiedad de encapsulamiento. El modificador de acceso “private”, “public”, “protected” ó “package friendly” aplicado en atributos o métodos, sea el apropiado y del mayor ocultamiento posible.

NN. Evitar realizar entradas o salidas, desde o hacia consola, al interior de una “clase administradora” o al interior de sus objetos administrados.

OO. Cumplir con las convenciones Java (para clases, métodos, etc.).

PP. El Código fuente sea libre de instrucciones para suprimir advertencias.

QQ. Toda *factorización* no afecte el mantenimiento del programa, en cuanto a agregar más opciones a una estructura de control selectiva.

RR. Evitar usar la selección canónica if-else, cuando en este se efectúe la asignación a una sola variable, y que para el mismo efecto sea posible usar el operador de condición.

SS. Evitar el uso del modificador static, a menos que sea indispensable el uso de atributos de clase y métodos de clase, justificando su uso con comentarios en código. Así mismo utilizar dicho modificador cuando sea más eficiente el usar atributos de clase y métodos de clase que instanciar objetos sin estado.

TT. Apoyándose en variables locales, evitar dentro de un mismo método, la lectura repetitiva de celdas de un arreglo o atributos de un objeto distinto al objeto del método en ejecución, cuando tales atributos o celdas no se modifiquen entre una lectura y la siguiente. Análogamente evitar la escritura repetitiva, si es posible hacer una sola escritura con el resultado de la variable local.

UU. Si incluye comentarios, estos sean solo para ser procesables por Javadoc; en los métodos, dichos comentarios sean breves y referentes solo a los parámetros y resultados a obtener, y de las clases solo su propósito; los anteriores en términos de caja negra.

VV. Todo try-catch imprima en consola la traza de la pila, a menos que se implemente alguna acción correctiva para la excepción atrapada o bien se justifique con comentarios la falta de acción correctiva.

WW. Toda instrucción X que dependa de la correcta ejecución de una instrucción Y anterior, se encuentre dentro del mismo bloque try-catch que Y si Y lanza excepción.

XX. Las instrucciones para procesar eventos de usuario se limiten a sólo pasar parámetros hacia el/los objetos que represente(n) la lógica de la aplicación.

YY. El diseño del programa cuente con el mayor grado de cohesión y el menor grado de acoplamiento posibles.

ZZ. Codificar sus algoritmos de la mejor forma posible, considerando el mejor uso de los recursos (procesador, memoria, red) que requiera el programa en ejecución.