Apéndice 1

Buenas Prácticas de Programación

Cómo escribir un programa de buena calidad y no morir en el intento

Prácticas recomendadas

¿Cuál es el secreto para obtener programas que cumplan con las características de calidad más importantes? Cumplir las siguientes pautas son un medio seguro para ir en el camino correcto:

- **Planificar el diseño**: Hacer un esbozo del algoritmo y refinarlo progresivamente hasta convertirlo en ejecutable.
  Esta estrategia de esbozar para luego detallar lleva a programas más organizados y de más fácil depuración y mantenimiento.

- **Encapsular**: Emplear procedimientos tratando que el código asociado a cada operación importante, y cada tipo de datos, quede colocado en un solo lugar del programa (módulo o clase, según corresponda), lo cual permitirá su fácil localización y mantenimiento.
  Encapsular facilita la depuración y el mantenimiento.

- **Aprovechar módulos existentes**: En la mayoría de los casos la experiencia que se encuentra presente en un programa existente puede mejorar la creación de un nuevo algoritmo.
  Emprender cada programa como si fuera el primero que se realiza es una causa frecuente de ineficacia.

- **Escribir código claro y bien documentado**: Usar nombres de identificadores apropiados, indentar, incluir comentarios aclarando el objetivo de cada parte del programa y, en el caso de métodos, explicar su objetivo, parámetros, resultado, pre y/o postcondiciones, es la tarea más importante de un programador.
  Escribir código claro y bien documentado ayuda al mantenimiento del software.

- **Diseñar el programa basado en casos de prueba** (testing cases): Desde el momento que se comienza a diseñar un programa, pensar en los casos de prueba que debe resistir. Documentarlos. Usarlos para hacer trazas durante la etapa de diseño y volverlos a ejecutar a todos cada vez que se modifique el código correspondiente.
  El propósito del desarrollo guiado por pruebas es lograr un código limpio que sea correcto.

- **Diseñar interfaces amigables**: Pensar las interfaces, ya sean gráficas, por consola o por cualquier otro dispositivo, de manera que sean simples de usar, con mensajes claros y que eviten que los usuarios comentan equivocaciones.
  Las interfaces simples y claras mejoran la usabilidad del producto software.

- **Escribir código bien estructurado**: Nunca usar saltos arbitrarios (exit) especialmente dentro de bucles, y respetar que haya un único punto de salida (return) en todos los métodos.
  Respetar el flujo normal del programa favorece la depuración y mantenimiento.

Más recomendaciones, sugerencias y ejemplos a continuación.

Diseñar interfaces amigables

Cuando se escribe un programa se debe tener en cuenta que la manera de operarlo sea simple y que reduzca el esfuerzo del usuario. Para ello se recomienda seguir estas pautas:

- Los mensajes para indicar QUE debe ingresar el usuario, deben ser cortos y no ambiguos.
- Para ingresar OPCIONES DE MENÚ pueden utilizarse números o letras. Si se utilizan letras, tenga cuidado que el código sea consistente en el tratamiento de las mayúsculas y minúsculas.
- Si lo que se debe ingresar es una opción de un tipo ENUMERADO (ej: estado civil, cuyas opciones son casado / soltero / viudo / divorciado, etc), no se recomienda leer texto porque el usuario siempre puede cometer errores de tipeo. Por eficiencia se recomienda codificar el ingreso con un número entero o una letra y luego asignar una cadena predefinida como constante. Otra opción válida es guardar el valor entero o letra leído y mantener el significado en una segunda estructura, que sirve de diccionario. Importante: Sea claro en el mensaje para que el usuario sepa qué es lo que debe ingresar. Si se utilizan letras, tenga cuidado que el código sea consistente en el tratamiento de las mayúsculas y minúsculas.
  - Ejemplo: Indique su estado civil (<C>Casado - <S>Soltero - <V>Viudo - <D>Divorciado): _
- En opciones SI/NO, utilice siempre el tipo booleano para almacenar y para leer codifíquelo apropiadamente con una letra o número, indicando claramente al usuario lo que debe ingresar para indicar SI y qué para indicar NO. Si se utilizan letras, tenga cuidado que el código sea consistente en el tratamiento de las mayúsculas y minúsculas.
  - Ejemplo: ¿Quiere ingresar otro número? (<S>Si - <N>No): _
- Si la entrada DEBE ser de tipo String (por ejemplo apellidos, ciudades, etc), siempre convierta el valor ingresado a mayúsculas antes de guardarlo, ya que cadenas con distintos usos de mayúsculas y minúsculas (Ej: YO, yo, Yo, yO) son cuatro cadenas distintas y puede resultar en inconsistencias al buscar datos.
- Las variables de tipo real deben ser mostradas en formato acorde, por ejemplo delimitando la cantidad de dígitos decimales a 2 en caso de dinero, etc.
  - Ejemplo: Su saldo es $ 938.90

Elegir nombres de identificadores significativos

Los nombres de variables, constantes, clases y métodos se llaman IDENTIFICADORES. Un identificador siempre comienza con una letra y luego es una sucesión de letras, números o el caracter "_" (llamado guión bajo, subrayado o underscore). Se recomienda utilizar nombres que expresen claramente lo que se guarda o lo que hace el procedimiento, para ayudar a comprender mejor el programa.

En Java, por convención, se utilizan los nombres con el siguiente formato:

- **Variables**: Comienzan y siguen con minúscula. Si están formados por más de una palabra, cada palabra empieza con mayúscula (excepto la primera). Ejemplos: *cantidad, suma, esMayor, pos1.*
- **Clases**: Comienzan con mayúscula y siguen con minúscula. Si están formados por más de una palabra, cada palabra empieza con mayúscula. Se recomienda que indiquen un sustantivo. Ejemplos: *Lista, ListaOrdenada, MiContador.*
- **Métodos**: Comienzan y siguen con minúscula. Si están formados por más de una palabra, cada palabra empieza con mayúscula. Se recomienda que comiencen con un verbo. Ejemplos: *sumar, obtenerPeso, esIgual, insertarElemento.*
- **Constantes**: Comienzan y siguen con mayúsculas, con las palabras internas separadas por el carácter underscore "_". Ejemplos: *MAXIMO_ELEMENTO, PI, ALTURA_MINIMA.*

Otras recomendaciones importantes:

- Elegir identificadores que expresen el contenido de la variable, el tipo que representa la clase, o la acción, de manera clara. NO UTILICE nombres ambiguos como *band, bandera.*
- Evitar nombres del tipo *aux* o *temp*, salvo que se utilicen expresamente con esa idea (por ejemplo al hacer un intercambio de valores entre variables es posible utilizar una variable llamada *temp* para el valor que se guarda sólo por un momento)

- Evitar nombres formados por una o dos letras, como *a* o *p1*, salvo que se utilicen puntualmente. Por ejemplo, algunos usos válidos son:
  - para indicar subíndices de estructuras lineales o matrices: i, j, k
  - para guardar temporalmente elementos de una estructura: x

Recordar:

- Java distingue entre mayúsculas y minúsculas, por lo tanto los identificadores *pos, POS, Pos* son todos distintos.
- Las palabras reservadas del lenguaje como *class, this, for, long* NO pueden usarse como identificadores. Ver en Figura 1.1 la lista completa de palabras reservadas del lenguaje Java.

abstract	do	implements	protected	throw
boolean	double	import	public	throws
break	else	instanceof	rest	transient
byte	extends	int	return	true
case	false	interface	short	try
catch	final	long	static	void
char	finally	native	strictfp	volatile
class	float	new	super	while
const*	for	null	switch	
continue	goto*	package	synchronized	
default	if	private	this	

Figura 1.1: Palabras reservadas del lenguaje Java

Indentar el código apropiadamente

La "indentación" o "identación" es una manera de hacer más entendible el código tanto para uno mismo como para otros programadores.

Un código bien indentado es mucho más legible, se ve mucho más profesional y da muchas más ganas que leerlo que un código desordenado.

El único secreto es poner todas las sentencias que se encuentran al mismo nivel de ejecución con la misma cantidad de espacios por delante, para que queden perfectamente encolumnadas y se detecten fácilmente donde comienza y termina cada bloque. También se pueden agregar líneas en blanco entre conjuntos de sentencias que están relacionadas para facilitar la lectura. En la Figura 1.2 se pueden ver ejemplos de código mal y bien indentado.

```
public static int fact(int n) {
int resultado;
    if (n > 1) {
resultado = fact(n - 1) * n;}
else {
resultado = 1;}
   return resultado;
}
```

(a) mal indentado

```
public static int fact(int n) {
    int resultado;

    if (n > 1) {
        resultado = fact(n - 1) * n;
    } else {
        resultado = 1;
    }

    return resultado;
}
```

(b) bien indentado

Figura 1.2: Ejemplo de mala vs. buena indentación

Para indentar es preferible utilizar una convención y respetarla. Se suele agregar un tabulado por nivel o también una cantidad fija de espacios. La opción de usar 3 ó 4 espacios de diferencia entre cada nivel suele ser suficiente para que el código quede claro y la línea no sea demasiado larga. No mezclar espacios y tabulados al indentar. Es mejor elegir uno de los dos!

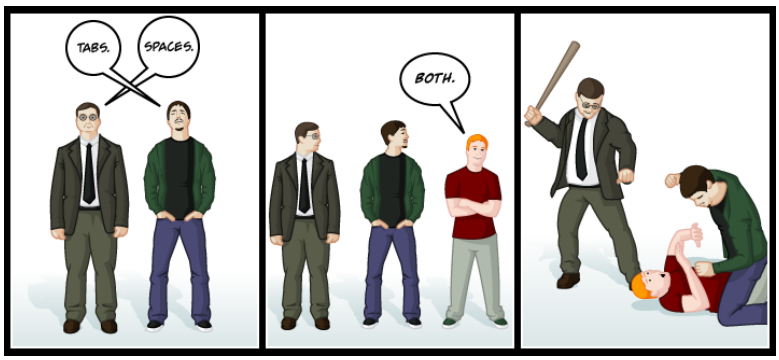

Figura 1.3: Por qué no es recomendable mezclar tabs y espacios al indentar

Además, es importante saber que los entornos de programación permiten indentar automáticamente:

- Netbeans: se indenta todo el código de la clase automáticamente, presionando las teclas ALT-SHIFT-F de manera conjunta
- Eclipse: se deben pintar las líneas deseadas y luego presionar CTRL-I

> *No hay excusa para que el código no esté bien indentado.*

Agregar comentarios adecuados en el código

Al programar, una de las cosas que siempre se debe recordar es agregar comentarios. Importante: Al medir el rendimiento por líneas de código escritas, los comentarios no entran en la estadística.

En el lenguaje Java existen tres formas de poner comentarios.

- Cuando la línea de comentario solo ocupa una linea de código: En este caso deberemos de anteponer dos barras inclinadas (//) antes del texto. Ejemplo: *// Comentario de una línea*
- Para comentarios de más de una línea se empieza por una barra inclinada y un asterisco (/*) y finaliza a la inversa, asterisco y barra inclinada (*/). Ejemplo: */* Comentario de varias líneas */*
- El último caso son los comentarios para la herramienta de documentación JavaDoc. En este caso, antes del comentario pondremos una barra inclinada y dos asteriscos (/**) y finaliza con un asterisco y una barra inclinada (*/). Puede ocupar una o más líneas. Ejemplo */** Comentario para JavaDoc */*

¿Qué hay que documentar?

Hay que añadir explicaciones a todo lo que no es evidente. No hay que repetir lo que se hace sino explicar POR QUé se hace, por ejemplo:

- ¿De qué se encarga la clase?
- ¿Qué está haciendo un grupo de sentencias en particular?
- ¿Qué hace este método? ¿Cuál es el uso esperado? ¿? Qué precondiciones y/o postcondiciones tiene?
- ¿Para qué se usa una variable? ¿Cuál es el uso esperado? Si es una medida ¿en qué está expresada? Por ejemplo: milímetros, centímetros, metros, pesos, dolares, etc
- ¿Qué algoritmo estamos usando? ¿Cuál es la fuente: autor, libro, página web? ¿Qué limitaciones tiene el algoritmo? ¿y su implementación? ¿Cómo se podría mejorar si hubiera tiempo?
- Separar las sentencias en bloques visualmente detectables y agregarles un comentario que indique su objetivo ayuda a organizar incluso al momento de programar (además de que luego será más fácil hacer modificaciones).

¿Cuándo hay que poner un comentario?

- Por obligación (sobre todo al usar Javadoc):
  - al principio de cada clase,
  - al principio de cada método,
  - para cada variable de clase.
- Por conveniencia (una línea):
  - al principio de un fragmento de código no evidente,
  - adentro de un bucle.
- Y por si acaso (una línea):
  - siempre que hagamos algo raro, que pueda ser necesario verificar o modificar más adelante,
  - siempre que el objetivo del código no sea evidente.

Figura 1.4: Un comentario debe ser información útil para el futuro

> *Documentar el código tiene el objetivo de facilitar la tarea del programador y de quienes tengan que realizar luego el mantenimiento del sistema.*

Diseñar en base a un buen lote de casos de prueba

Que un programa funcione con un par de casos de prueba no quiere decir que el programa sea CORRECTO.

Es necesario diseñar el programa, desde sus inicios, considerando todos los casos que pueden diferenciarse en el tratamiento, documentando los casos generales y especiales y, después de hacer cualquier modificación, volver a ejecutarlos a todos, ya que un cambio en un lugar puede afectar a varios casos de prueba y puede introducir nuevos errores.

A continuación se utiliza el código de la función *factorial* que se presenta en la Figura 1.5 como ejemplo. En este caso se deberá considerar como mínimo un lote de 3 casos de prueba para asegurar que se han cubierto la condición y ambas ramas de la alternativa:

- un valor n mayor a 1 para ejecutar el caso recursivo (por ejemplo 5, que asegura que va a llamar recursivamente más de una vez),
- un valor n menor a 1 para probar el caso base, sin que sea invocado recursivamente (por ejemplo 0 o cualquier número negativo),
- el valor n igual a 1, porque es el límite de la condición.

```java
public static int fact(int n) {
    int resultado;

    if (n > 1) {
        resultado = fact(n - 1) * n;
    } else {
        resultado = 1;
    }

    return resultado;
}
```

Figura 1.5: Código de la función recursiva factorial(n)

Regla de ORO: "Desconfiar de la correctitud del código hasta que se haya demostrado lo contrario"

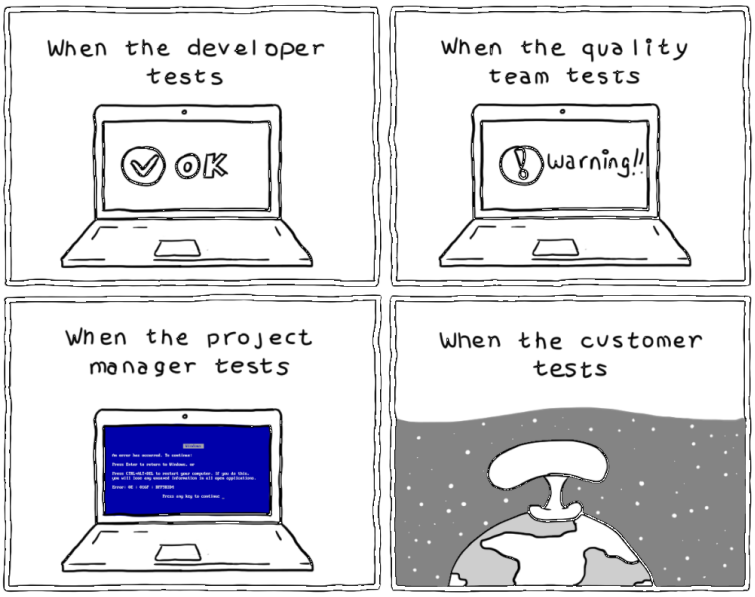

Figura 1.6: Probable ciclo de vida de código mal testeado

Evitar el uso de sentencias de salto arbitrario

Las sentencias *break*, *exit* y *continue* son sentencias que permiten saltar a puntos del programa por fuera del flujo normal.

Evitar el uso de estas sentencias es un principio básico de la programación estructurada. Este tipo de sentencias dificultan la legibilidad, depuración y verificación de los programas.

El uso de sentencias *break* y *continue* suele estar asociado a técnicas de construcción de bucles artesanales que tienen sentido sólo en lenguajes de bajo nivel. Al no usarlas se obliga a pensar primero que condición o invariante debe cumplirse durante la ejecución del bucle, y luego codificar el bucle correspondiente usando la sentencia *while* o *for* más apropiada. Esta es la manera correcta: es la base para la posterior verificación formal del bucle y contribuye a mejorar la legibilidad de los programas.

Por ejemplo, en un bucle *while* libre de sentencias *break*, sabemos que el bucle terminará cuando la condición deje de cumplirse. Por tanto, para saber si un bucle terminará, bastará comprobar que en el cuerpo del bucle se realiza algún cómputo que converge hacia tal condición. Además, en ausencia de sentencias *break*, a la salida del bucle sabemos que la negación de la condición del bucle siempre será verdadero, lo que constituye la base para seguir diseñando el programa. Esto no es así cuando se sale del bucle usando la sentencia *break*.

Usar un único return por función

Es recomendable escribir un único punto de retorno (return) por método (función) que se colocará como última sentencia del mismo.

Este consejo es consecuencia de uno de los principios de la programación estructurada que dice que los programas deben tener un único punto de entrada y un único punto de salida. El uso de un único return por función, colocado consecuentemente al final de la misma, facilita tanto la depuración como la adaptabilidad de los programas.

Por ejemplo, si estamos depurando un programa y queremos saber que valor devuelve una función, en caso de existir un único return, bastará con imprimir el valor devuelto antes de invocar dicho return; o poner un punto de ruptura del debugger en dicha sentencia. Si hubiera 4 sentencias return, tendríamos que colocar y gestionar 4 puntos de ruptura durante la depuración del programa. Luego, cualquier modificación del método que se refiera al resultado del mismo, deberá ser chequeada en 4 lugares en lugar de sólo uno.
