Elementos básicos requeridos para el desarrollo de software de calidad

Departamento Académico de Computación
División Académica de Ingeniería
ITAM

Objetivos

1) Crear conciencia en maestros y alumnos de la importancia de generar software de calidad, que pueda ser fácilmente probado, corregido y mantenido.
2) Crear conciencia en maestros y alumnos de que el desarrollo de software es una actividad ingenieril, con toda la rigurosidad que eso implica.
3) Desarrollar en los alumnos buenos hábitos de programación desde el primer semestre.
4) Proporcionar un marco de referencia que permita a todos los maestros que impartan materias de desarrollo de software, manejar los mismos criterios.

Contenido

1- Descripción

Se presentan algunos elementos básicos que deben incorporarse, a través de la enseñanza y la práctica, al proceso personal de desarrollo de software de los alumnos.

Estos elementos empezarán a proporcionarse a partir de la primera materia orientada a la elaboración de programas, por medio de ejemplos, de las especificaciones de las tareas, controles y exámenes. Los mismos se seguirán aplicando a lo largo de todos los semestres, logrando así crear los buenos hábitos de programación buscados.

Si bien el objetivo de las materias en las cuales se darán estos elementos, es enseñar a analizar un problema y a encontrarle y diseñarle algorítmicamente una solución, es muy importante no descuidar los aspectos presentados en este documento. Se debe intentar que el alumno los vaya incorporando desde el primer día.

> ***Es menor el esfuerzo requerido para formar un buen hábito que el necesario para corregir uno malo.***

Este documento cubre seis temas fundamentales. El primero que se expone, *layout*, proporciona los lineamientos para escribir código cuidando su presentación. Es decir, aspectos que ayudan a que el código fuente "luzca bien", y en consecuencia sea más legible, más fácil de entender y de mantener. En el capítulo de *variables* se presentan algunas ideas sobre el uso de las mismas: nombres, alcances, y otros aspectos interesantes. En el cuarto capítulo, *instrucciones*, se explican los elementos a tener en cuenta para la codificación legible y segura de las instrucciones, considerando secuencias, decisiones, ciclos y recursión. El quinto capítulo, *métodos*, presenta los aspectos más importante a tener en cuenta al diseñar e implementar métodos. El sexto está dedicado a presentar los aspectos más relevantes del diseño de tipos abstractos de datos y la representación de los mismos por medio de clases. Finalmente, el último capítulo ofrece los lineamientos para documentar correctamente un programa. En algunos de los temas se incluye al menos una lista de revisiones, que ayuda a verificar a maestros y a alumnos si se tuvieron en cuenta los puntos importantes. Cabe destacar que los temas fueron elaborados por varios profesores del Departamento Académico de Computación.

En el documento también se incluyó un apéndice con algunos diagramas de flujo correspondientes a operaciones básicas, con el objetivo de homogeneizar el enfoque con el cual éstas se enseñan. Además se presenta un esquema para seguimiento de algoritmos por medio de un mapa de memoria.

Los temas tratados aplican a los lenguajes de programación C, C++ y Java.

2- Layout

Algunos criterios que deben considerarse cuando se define un estilo de programación:

- ¿El esquema del formato resalta la estructura lógica del código?
- ¿Puede usarse consistentemente?
- ¿Resulta en código que es fácil de mantener?
- ¿Mejora la lectura del código?

> *"Many aspects of layout are religious issues. Try to separate objective preferences from subjective ones. Use explicit criteria to help ground your discussions about style preferences."*

2.1) Sangría: 4 espacios

2.2) Líneas que se dividen

a) Dividir después de una coma, antes o después de un operador.
b) Incluir una sangría en la línea que continúa.
c) En expresiones complicadas, poner condiciones separadas en líneas separadas.
d) Mantener juntos elementos que estén relacionados.

2.3) Declaraciones de variables: Ordenar las declaraciones por tipo y función

2.4) Emulación de bloques puros:

```
if  (condición) {
   instrucciones;
}

if  (condición) {
   instrucciones;
}
else {
   instrucciones;
}

if  (condición1) {                // "switch"
   instrucciones;
} else  if  (condición2) {
   instrucciones;
} else if  (condición3) {
   instrucciones;
} else {
   instrucciones;
}

switch (condición) {
   case ABC:
      instrucciones:
      /* sigue */
```

```
    case DEF:
       instrucciones;
       break;
    default:
       instrucciones;
       break;
}

try {
   instrucciones;
} catch (ExceptionClass e) {
   instrucciones;
} finally {
   instrucciones;
}

for ( ) {          // o while
   instrucciones;
}
```

2.5) Comentarios

a) En cada línea de comentario dejar la misma sangría que en la línea de código "que comenta".
b) Colocar por lo menos una línea en blanco antes de cada línea de comentario.
c) Si se usa Javadoc, se debe indicar antes de la línea de encabezado de la clase o de la interface.

2.6) Lista de revisiones

Se presenta la lista de revisiones que ayuda a verificar el cumplimiento de los elementos presentados en esta sección.

Lista de revisiones: layout

Propósito:	Verificar que se han considerado todos los elementos importantes relacionados con el estilo de programación.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
Estructuras de control	
¿Los bloques secuenciales están separados entre sí por una línea en blanco?	
¿Las expresiones complicadas están formateadas para facilitar su lectura?	
Individual	
¿Se usan espacios para facilitar la lectura de expresiones lógicas y argumentos de métodos?	
¿Las líneas que continúan tienen la sangría adecuada?	

¿Cada línea contiene a lo más una instrucción?	
¿Ninguna instrucción tiene efectos secundarios?	
Comentarios	
¿Los comentarios tienen la misma sangría que el código que comentan?	
¿El estilo de los comentarios es fácil de mantener?	
Métodos	
¿Los argumentos están formateados de manera que son fáciles de leer, modificar y comentar?	
¿Se usan líneas en blanco para separar partes de un método?	
Clases	
¿Los métodos están claramente separados por líneas en blanco?	

3- Variables

3.1) Inicialización

a) Lo ideal es que las variables sean declaradas en el inicio.
b) Usar constantes siempre que sea posible.
c) Inicializar los atributos en el constructor.
d) Reinicializar las variables con código ejecutable cerca de donde van a ser utilizadas.
e) Validar los valores de entrada que sean estructuralmente necesarios.

3.2) Alcance

a) El alcance de una variable hace referencia a la sección del programa donde la misma puede usarse.
b) No usar variables globales.
c) Los atributos de las clases deben ser preferentemente declarados como privados.

3.3) Uso

a) Utilizar una variable para sólo un propósito.
b) Asegurarse de que todas las variables que fueron declaradas son usadas.

3.4) Nombre

a) Debe ser legible y apropiado.
b) Debe describir completamente lo que representa.
c) No utilizar abreviaciones crípticas ni ambiguas.
d) Debe expresar qué hace en lugar de cómo lo hace.
e) Las variables de estado deben llevar un nombre descriptivo, evitar llamarlas *bandera*.
f) Los nombres de las variables temporales también deben ser descriptivos y no se deben usar nombres como: temp, x, o algo similar.
g) Los nombres *hecho* o *encontrado* son correctos para variables booleanas porque son acciones que se pueden calificar como verdaderas o falsas. Algo está hecho o no está hecho, fue encontrado o no fue encontrado. En cambio, *archivoFuente* o *estado* no son buenos nombres. Se podrían sustituir por *archivoFuenteDisponible o estadoOK.*

h) Escribir las constantes en mayúsculas y utilizando nombres descriptivos de para qué se usan y no cuánto valen. El nombre CINCO es incorrecto, mientras que CICLOS_NECESITADOS es correcto. Por convención de Java, si consta de más de una palabra, éstas se separan con guión bajo.
i) Los nombres de las variables, de los métodos y de los objetos deben comenzar con minúscula mientras que los nombres de las clases e interfaces deben comenzar con mayúscula y escribirse en singular. Posteriormente los caracteres van en minúsculas. Si el nombre está compuesto por varias palabras entonces las iniciales de las palabras van en mayúscula.
j) Si se usan abreviaciones se debe escribir una tabla como comentario, al principio del programa, indicando el significado de las mismas.
k) El nombre de una interface depende de su propósito. Si el principal propósito es proporcionarle al objeto cierta habilidad entonces el nombre debería ser un adjetivo que describa dicha habilidad (Por ejemplo, Comparable, Serializable). En caso contrario se debería utilizar un sustantivo o frase sustantivada.
l) Los nombres de las variables y de las constantes deberían ser sustantivos o frases sustantivadas. Nombres de un solo carácter se deberían evitar, excepto para variables temporales. Nombres comunes para variables temporales son: i, j, k, m, n.
m) Los nombres de los métodos deberían ser verbos imperativos. Por ejemplo, si un método muestra el valor del atributo peso, entonces se debería llamar getPeso(), regresaPeso(), devuelvePeso() o muestraPeso().
n) Se deben evitar nombres similares, por ejemplo: valorTotal y totalValor.

3.5) Tipos de datos

a) Evitar que aparezcan cantidades, caracteres o cadenas de caracteres a mitad del programa sin explicación. Utilizar constantes. Las únicas literales que deberían aparecer en un programa son el 0 y el 1.

b) Utilizar tipos de datos estructurados (struct o class) para simplificar las listas de parámetros.

4- Instrucciones

4.1) Secuencia de instrucciones

a) Instrucciones que requieren un cierto orden: es importante dejar las dependencias claramente expresadas. Algunas ideas:

a.1) *Organizar el código de tal manera que las dependencias sean obvias*. Por ejemplo incluir un método que inicialice algunos elementos (variables, objetos, etc.), el cual será invocado antes que los métodos que usen dichos elementos.

a.2) *Usar nombres para los métodos de tal manera que ayuden a hacer obvias las dependencias*. Por ejemplo:
inicializaCostos();
calculaCostos();

a.3) *Usar parámetros en los métodos de tal manera que ayuden a hacer obvias las dependencias*. Por ejemplo:

```
gastosPersonales(datoGasto);
gastosVentas(datoGasto);
//Indicaría que usan el mismo dato y en ese orden. O bien hacer:
datoGasto = gastosPersonales(datoGasto);
datoGasto=  gastosVentas(datoGasto);
//Esto último deja aún más clara la dependencia entre las instrucciones.
```

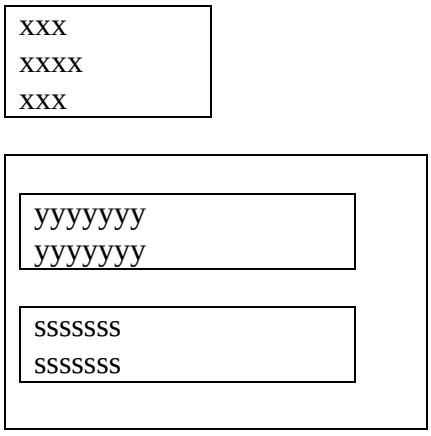
a.4) *Documentar aquellas dependencias que sean poco claras*.

a.5) *Verificar dependencias críticas por medio del uso de variables o código para el manejo de errores*. Por ejemplo usar una variable que tome cierto valor si se realizó alguna actividad, para posteriormente evaluar esa variable antes de realizar la actividad que depende de la primera.

b) Instrucciones que NO requieren un cierto orden: a pesar de que para la ejecución correcta no se requiera un orden, se puede establecer "orden" para mejorar la legibilidad, desempeño y mantenimiento. El principio recomendado para ordenar: *mantener las acciones relacionadas juntas*. Algunas ideas:

b.1) *El código es más fácil de leer de arriba hacia abajo que realizando "saltos" a diferentes partes del mismo*.

b.2) *Agrupar instrucciones relacionadas* (por los datos, por el tipo de actividad, etc.) Para determinar si están correctamente agrupadas, se sugiere dibujar recuadros alrededor de las instrucciones. Si los cuadros quedan independientes entre sí, están bien ordenadas. Si los cuadros quedan superpuestos, están mal ordenadas. Este agrupamiento puede dar origen a métodos que engloben a esos grupos de instrucciones.



- ✓ El principio más importante para ordenar las instrucciones son las *dependencias*.
- ✓ Las *dependencias deben hacerse obvias* por medio del uso de: parámetros,

comentarios, nombres adecuados para los métodos y por medio del manejo de código de errores.

✓ Si no existen dependencias de orden, *mantener juntas a las instrucciones relacionadas*.

4.2) Estructuras selectivas

a) Instrucciones if (simple y doble). Algunas ideas:

a.1) *Escribir primero el caso "normal" y luego los casos "inusuales o poco frecuentes"*. Asegurarse que el código resulta claro para los casos normales. Esto ayuda a la legibilidad y al desempeño.

a.2) *Asegurarse que se bifurca correctamente con las igualdades*. Analizar los casos de >= (en lugar de >) y de <= (en lugar de <), si el lenguaje lo permite.

a.3) *Escribir el caso normal luego del if en lugar de luego del else*. Es decir, escribir el código que resulta de una decisión cerca de esa decisión. Tener todos los casos positivos junto a los if y los negativos con los else ayuda a leer y mantener el código. Por ejemplo, en C:

```
ap = fopen("XXXXX", "r");
if (ap != NULL) {
   //Trabajamos con el archivo.
   fscanf(ap, "%i", &num);
   if (num > 0)
      //Hacemos algo.
   else
      //Caso de error. No usamos el número.
}
else
   //Caso de error. No trabajamos con el archivo.
```

Otro ejemplo en Java:

```
if (numCalificaciones > 0) {
   promedio = sumaCalificaciones / numCalificaciones;
   System.out.println("\nEl promedio es: " + promedio);
}
else
   System.out.println("\nNo hay calificaciones: no se pudo calcular el promedio");
```

a.4) *Escribir condiciones que expresen claramente lo que se está evaluando*.

a.5) *Siempre se deben incluir instrucciones para el if*. Es decir se debe evitar dejar vacía esa sección de la instrucción. Por ejemplo:

```
if (condición)
  ;
else
   //Hacemos algo.
```

Se puede reescribir como:

```
if (!condición)
    //Hacemos algo.
```

a.6) *Cuando se prueba el código, probar de igual manera el if y el else*. Ambos pueden ser causa de errores.

b) Instrucciones if (anidadas). Algunas ideas:

b.1) *Simplificar condiciones complejas por llamadas a funciones booleanas*. Por ejemplo:

```
if (caracter >= 'a' && caracter <= 'z')
   //Hacemos algo.
else if (caracter == '.' || caracter == ',' || caracter == ';' || caracter == '!' ||
          caracter == '?')
         // Hacemos algo.
….
```

Se puede reescribir como:

Lenguaje C (notación explícita)	Lenguaje C (notación simplificada)

`if (esLetraMinuscula(caracter) == 1)`
`  //Hacemos algo.`
`else if (esSignoPuntuacion(caracter) == 1)`
`  // Hacemos algo.`
`….`
	`if (esLetraMinuscula(caracter))`
	`  //Hacemos algo.`
	`else if (esSignoPuntuacion(caracter))`
	`  // Hacemos algo.`
	`….`

Lenguaje Java (notación explícita)	Lenguaje Java (notación simplificada)

`if (esLetraMinuscula(caracter) == true)`
`  //Hacemos algo.`
`else if (esSignoPuntuacion(caracter) == true)`
`  // Hacemos algo.`
`….`
	`if (esLetraMinuscula(caracter))`
	`  //Hacemos algo.`
	`else if (esSignoPuntuacion(caracter))`
	`  // Hacemos algo.`
	`….`

b.2) *Escribir primero los casos más comunes*. Retomando el ejemplo anterior, es más probable encontrar letras que signos en una frase. Ayuda a la legibilidad, mantenimiento y prueba del código.

b.3) *Asegurarse que todos los casos quedan cubiertos*. Incluir un else con alguna instrucción apropiada por si se detecta un caso no previsto.

b.4) *Reemplazar if/else anidados por switch() si el lenguaje lo soporta y si las características del problema lo permiten*.

c) Instrucción switch. Algunas ideas:

c.1) *Ordenar los casos por frecuencia*. Escribir primero aquellos que ocurran más frecuentemente. Tiene la ventaja de que los casos más comunes se encuentran primero (legibilidad y mantenimiento).

c.2) *Escribir el caso normal primero*. Si se tiene un caso normal y varias excepciones, entonces escribir primero el normal y luego los otros. Documentar apropiadamente.

c.3) *Ordenar los casos alfabética o numéricamente*. Si los casos son igualmente importantes, ordenarlos alfabética o numéricamente.

c.4) *Escribir acciones simples en cada caso*. Si se requieren muchas instrucciones o instrucciones muy complejas, reemplazarlas por un método.

c.5) *No utilizar variables auxiliares para forzar el uso del switch*. Si por el tipo de datos no se puede usar el switch, usar anidamiento de ifs. El uso de variables auxiliares puede resultar confuso y ser causa de errores. Por ejemplo en Java:

```
accion = commando.charAt(0);
switch (accion) {
   case ‘c’: copy();
             break;
   case ‘d’: delete();
             break;
    …
   default: algo();
}
```

La variable “accion” controla al switch, pero no tiene toda la información dada por el usuario. En este caso, si el usuario ingresa cualquier palabra que empiece con c se entenderá que dio la orden “copy”, aunque no sea así.

Se sugiere reescribir de la siguiente manera:

```
if (comando.equals(“copy”))
    //Hace algo.
```

```
else if (comando.equals("delete"))
        //Hace algo.
```

c.6) *Usar el default para representar el caso por omisión*. No usar el default para el último caso. Impide detectar correctamente errores y dificulta incluir posteriormente nuevos casos.

c.7) *Usar el default para detectar errores.* Si el default no se usa para el procesamiento de algún caso (y por lo tanto, "no va a ocurrir"), es útil para escribir algún mensaje de diagnóstico.

- ✓ En los if/else prestar atención al orden de las cláusulas if y else. El caso normal debe ser claramente identificable.
- ✓ En if/else anidados y switch ordenar los casos de tal manera que el código sea legible.
- ✓ Para detectar errores usar el default o los else, según la instrucción usada.

4.3) Estructuras repetitivas

a) Selección del ciclo. Algunas ideas:

a.1) *while y do-while*. Cuando no se sabe el número de veces. Difieren en el lugar donde se evalúa la condición, y por lo tanto determinan si el ciclo se ejecuta al menos una vez o no.

a.2) *for*. Cuando sabemos el número de veces que se ejecutará. Se establece en la primera línea la condición Y NO se debe incluir en el ciclo ninguna instrucción que altere esa condición. **Insistir en esto, que el lenguaje lo permita no debe ser una razón para usar un for cuando por la naturaleza del problema se requiere un while o do-while.**

b) Control del ciclo: inicialización, alteración de la condición, forma de terminar, etc. Algunas ideas:

b.1) *Entrar a los ciclos por un único punto: arriba*.

b.2) *Escribir las instrucciones para inicializar el ciclo justo antes del mismo*. Ayuda cuando se hacen copias de parte del código, correcciones o generalizaciones. Aquellas instrucciones relacionadas al ciclo deben mantenerse junto a él.

b.3) *Uso del for preferentemente antes que otros ciclos*. Si el problema lo permite, elegir el for en lugar de los otros ciclos, ya que en una sola línea se expresa la inicialización, la alteración de la variable de control y la condición involucrada. Si hubiera cambios, hay más posibilidades de hacerlos sin OMITIR a algunas de las partes.

b.4) *No usar for cuando el while sea más apropiado*. Insistir en usar el ciclo que sea más apropiado a cada problema.

b.5) *No usar {} para encerrar la instrucción del ciclo. Sólo usar {} si se tienen dos o más instrucciones en el ciclo*.

b.6) *Evitar ciclos vacíos*. Aquellos en los cuales en la condición se incluye todo el código.

b.7) *Actualizar las variables de control siempre al final/principio del ciclo*. En los ciclos while con bandera de fin de datos, una lectura antes de entrar al ciclo y otra al final del mismo.

b.8) *Cada ciclo ejecuta una sola función*. Los ciclos deben verse como a los métodos, cada uno hace UNA sola cosa y la hace bien.

b.9) *Asegurarse que el ciclo termina*. Considerar todos los posibles casos.

b.10) *Expresar de manera obvia la condición de terminación del ciclo*. Poner el control del ciclo en un solo lugar.

b.11) *NO alterar la variable de control de un for para provocar que termine*.

b.12) *Evitar escribir código que dependa del valor final del índice de un ciclo for*. El valor final puede variar dependiendo del lenguaje, de la implementación, de si el ciclo termina normal o anormalmente. Sugerencia: asignar a otra variable el valor deseado en el lugar apropiado dentro del ciclo.

b.13) *Probar los ciclos con los casos que puedan presentarse*. Mentalmente y/o con la prueba de escritorio.

b.14) *Usar nombres significativos para las variables de control de ciclos anidados*. Aumenta legibilidad. Evita errores. Cuando la variable de control sea sólo eso (es decir, no está relacionada con la información que se procesa en el ciclo), se pueden usar i, j y k.

b.15) *Escribir ciclos que no excedan una página*.

b.16) *Limitar a máximo 3 ciclos anidados*. Si se requiere más, usar métodos.

b.17) *Escribir el código de ciclos grandes en métodos*. Si el ciclo está bien definido, se podrá pasar el código que encierra a un método.

b.18) *NO utilizar while (true) o do-while(true).*

- ✓ Mantener lo más simple posible a los ciclos: legibilidad.
- ✓ Minimizar el anidamiento de ciclos.

- ✓ Definir una entrada y salida del ciclo clara.
- ✓ La actualización de la variable de control debe hacerse en un solo lugar.
- ✓ Nombrar apropiadamente la variable de control de un ciclo.
- ✓ Probar (mentalmente o con prueba de escritorio) el ciclo, para verificar que opera correctamente en todos los posibles casos y que termina correctamente ante todas las posibles condiciones.

4.4) Estructuras especiales de control

a) Múltiples return desde un método. Algunas ideas:

a.1) *Minimizar el número de return*. En soluciones iterativas se escribirá un solo return, mientras que en las recursivas –si la solución lo justifica- se pueden incluir varios return.

b) Recursión. Algunas ideas:

b.1) *Asegurarse de incluir un estado base en la definición recursiva del problema*. Se debe garantizar que la recursión se detenga en algún punto del procesamiento.

b.2) *Escribir el estado base antes que la solución recursiva*.

b.3) *Incluir contadores de seguridad*. En aquellos casos donde no se pueda garantizar que se llega al estado base antes de que se sature la pila, se sugiere incluir un contador de seguridad (como parámetro o miembro de una clase). Por ejemplo:

```
XXXXXXXX(int contSegur) {
    if (contSegur > LIMITE_SEGUR)  //Uso de un contador de seguridad para
        yyy;                       //detener la recursion.
    else {
        contSegur = contSegur +1;
        …
        XXXXXXXX(contSegur);       //Llamada recursiva.
```

b.4) *Limitar la recursión a un método*. Evitar la recursión cíclica *(A llama a B, B llama a C, C llama a A)* y a más de un nivel. Resulta difícil de entender, de detectar errores y de mantener.

b.5) *Considerar la pila de memoria*. Con el objeto de controlar qué puede pasar con el espacio de memoria en tiempo de corrida se sugiere:

Primero, si se usa un contador de seguridad, establecer el límite de tal manera que se tenga en cuenta qué tanto espacio se puede ocupar ANTES de caer en un desbordamiento de la pila.
Segundo, ser cuidadoso con el uso de variables.

b.6) U*so de recursión en funciones como el cálculo del factorial y Fibonacci*. Es importante destacar a los alumnos que la recursión no es una herramienta para resolver este tipo de problemas. Relación con la práctica del siguiente punto.

b.7) *Analizar alternativas de solución*. Se deben considerar distintas alternativas de solución ante un problema, valorando recursos, legibilidad, etc.

- ✓ Usar múltiples return **sólo** en casos de funciones recursivas que así lo requieran.
- ✓ Usar **cuidadosamente** la recursión.

4.5) Listas de revisiones

Se presentan las listas de revisiones que ayudan a verificar el cumplimiento de los elementos presentados en esta sección.

Lista de revisiones: secuencia de instrucciones

Propósito:	Verificar que se han considerado todos los elementos importantes en la escritura de secuencia de instrucciones.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
¿Las dependencias entre las instrucciones resultan obvias?	
¿Los nombres de los métodos hacen obvias las dependencias?	
¿Los parámetros de los métodos hacen obvias las dependencias?	
¿Los comentarios aclaran aquellas dependencias poco obvias?	
¿En caso de dependencias críticas se incluyó manejo de variables especiales -código de errores-?	
¿El código es legible de arriba hacia abajo?	
¿Las instrucciones relacionadas están juntas?	
¿Los grupos de instrucciones relacionadas pueden escribirse como métodos?	

Lista de revisiones: estructuras selectivas

Propósito:	Verificar que se han considerado todos los elementos importantes en la escritura de instrucciones selectivas.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado

if/else (simples)	
¿Queda claro en el código el caso normal?	
¿El if/else bifurca correctamente en el caso de la igualdad?	
¿El else está correctamente definido?	
¿El if y el else son usados correctamente?	
¿El caso normal está contemplado por el if?	
¿La condición está expresada claramente?	
if/else (anidados)	
¿Las pruebas muy complejas fueron reemplazadas por llamadas a funciones booleanas?	
¿Los casos más comunes son probados primero?	
¿Están cubiertos todos los casos?	
¿Los if/else anidados son una mejor opción que un switch?	
Switch	
¿Los casos fueron ordenados por importancia?	
¿Si no existe orden por importancia, fueron ordenados alfabética o numéricamente?	
¿Las acciones en cada caso son simples? (Opción: llamadas a métodos.)	
¿Se evalúa una variable "real" –no una generada a partir de la real-?	
¿Se usa el default para referenciar al caso por omisión?	
¿Se usa el default para detectar y reportar casos inesperados?	
¿Se incluyó el break en cada caso?	

Lista de revisiones: estructuras repetitivas

Propósito:	Verificar que se han considerado todos los elementos importantes en la escritura de instrucciones repetitivas.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
Elección y creación de ciclos	
¿Es apropiado usar un for?	
¿Es apropiado usar un while o do-while?	
Controlar el ciclo	
¿Se ingresa al ciclo desde arriba?	
¿La inicialización está justamente antes del ciclo?	
¿Si es un for, el encabezado está dedicado al control del ciclo?	
¿Se usan {} para encerrar el cuerpo del ciclo, sólo si éste tiene al menos dos instrucciones?	
¿El cuerpo del ciclo tiene al menos una instrucción?	
¿La(s) variable(s) de control está(n) (agrupadas) al inicio/fin del ciclo?	
¿El ciclo realiza sólo una función –como lo haría un método bien definido-?	
¿El ciclo es lo suficientemente pequeño para verse completo?	

¿Existe un nivel de anidamiento menor o igual a 3 ciclos?	
¿Si el ciclo es largo, está claramente escrito?	
¿Si es un for, el código no altera la variable de control?	
¿Se usa una variable auxiliar para guardar el valor de la variable de control del ciclo for, para utilizarse posteriormente fuera del mismo?	
¿La variable de control tiene un nombre significativo, o si el caso lo permite se usa i, j o k?	
¿El ciclo termina ante todas las condiciones?	
¿La condición de terminación es obvia?	

Lista de revisiones: estructuras especiales de control

Propósito:	Verificar que se han considerado todos los elementos importantes en el uso del return y de la recursión.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
Return	
¿Cada método usa el return sólo cuando es necesario?	
Recursión	
¿El método recursivo incluye código para detener la recursión?	
¿El estado base se escribió antes que la solución recursiva?	
¿El método recursivo incluye un contador de seguridad para garantizar que termina?	
¿La recursión se limita a un método?	
¿El límite de profundidad de la recursión considera el límite de capacidad de la pila?	
¿Es la recursión la mejor manera de resolver el problema? ¿Es mejor que la iteración?	

5- Métodos

Un **método,** también llamado método o procedimiento, es un conjunto pequeño de líneas de código escrito con un propósito específico. Es más fácil mostrar lo que **no** se puede considerar un método de alta calidad, que mostrar aquel que sí lo es.

5.1) Métodos de baja calidad

a) Nombre del método inadecuado.
b) El método no está documentado.
c) El método tiene un mal *layout*.

d) El método lee y escribe variables globales, forma incorrecta para la comunicación entre otros métodos.
e) El método no tiene un único propósito.
f) El método no está protegido contra datos incorrectos –división por cero-.
g) Los parámetros se pasan en forma errónea.
h) El método tiene demasiados parámetros.
i) Los parámetros no están ordenados y están mal documentados.

5.2) Razones válidas para crear métodos

a) Aparte de la computación en sí misma, el método es el invento simple más grande en ciencia de la computación. El método hace el programa más fácil de leer y comprender que cualquier otra característica de los lenguajes de programación.
b) El método también es la técnica más importante en computación para ahorrar espacio y mejorar el desempeño de los programas.
c) Facilitan el manejo –administración- intelectual del programa.
d) Reducen la complejidad. Después de escritos, evitan conocer detalles de cualquier tipo sobre su funcionamiento interno.
e) Introducen una abstracción intermedia y comprensible.
f) Evitan duplicación de código.
g) Útiles para esconder instrucciones irrelevantes para la lógica.
h) Útiles para esconder operaciones de apuntadores. Estas tienden a ser difíciles de leer y propensas a errores.
i) Incrementa la portabilidad.
j) Mejoran el desempeño del programa. Se puede optimizar el código en un solo lugar, en lugar de hacerlo en varios.
k) Además, muchas de las razones para crear una clase pueden ser buenas razones para crear un método:
    k.1) Separar –aislar- la complejidad.
    k.2) Esconder detalles de implementación.
    k.3) Limitar efectos de cambios.
    k.5) Facilitar la generación de código reusable.
    k.6) Permitir puntos centrales de control.

5.3) Métodos de dos o tres líneas

Construir un método para incorporar dos o tres líneas de código puede parecer un exceso. Sin embargo, la experiencia muestra cuán útil puede ser un simple y pequeño método.

5.4) Diseño de métodos

Cohesión. La cohesión (o coherencia) refiere a la forma en que se relacionan las operaciones en un método. Una función como *cosine()* tiene más cohesión que la función *cosineAndTan()* –trata de hacer más de una cosa en el mismo método-. Un estudio sobre 450 métodos mostró que 50% de los métodos coherentes estaban libres de errores. Otro estudio mostró que aquellos que no eran coherentes tenían 7 veces más errores que los otros y fueron 20 veces más costosos de arreglar.

5.5) Tipos de cohesión

a) Cohesión funcional. Representa el mejor ejemplo de cohesión: *sin()*, *getCustomerName()*, *eraseFile()*, *calculateLoanPayment()*, etc.
d) Cohesión secuencial. Las operaciones se realizan en secuencia. Dada fecha de nacimiento, calcular edad y luego años para el retiro.
e) Cohesión comunicacional. Las operaciones se relacionan porque utilizan solamente los mismos datos.
f) Cohesión temporal. Las operaciones se relacionan porque se realizan al mismo tiempo. Algunos sostienen que la cohesión temporal es inaceptable.
g) Cohesión en el procedimiento. Las operaciones se realizan en un orden específico y no necesitan estar juntas por ninguna otra razón. Obtener nombre del empleado, luego dirección y luego número telefónico.
h) Cohesión lógica. La lógica es la razón por la que se combinan operaciones.
i) Cohesión coincidente. Ocurre cuando las operaciones en el método no tienen una relación discernible.

5.6) Buenos nombres para métodos

a) Describir todo lo que el método hace.
b) El nombre debe ser tan largo como sea necesario.
c) No diferencie los nombres de los métodos por números.
d) Evite nombres o verbos vagos para describir métodos –no dan a entender qué hace el método-.
e) Para nombrar una función, use una descripción del valor que regresa.
f) Para nombrar un procedimiento, utilice un verbo en infinitivo seguido de un objeto. *printDocument()*.
g) Utilice opuestos. Agregar/quitar, comenzar/finalizar, crear/destruir, incrementar/decrementar, etc.

5.7) Tamaño de un método

¿Cuán largo puede ser un método? Teóricamente el máximo es una o dos pantallas, una o dos páginas de código. IBM limita los métodos a 50 líneas. Si escribe métodos mayores a 200 líneas, sea cuidadoso.

5.8) Uso de parámetros en un método

a) El orden de los parámetros debe ser en función de su entrada/salida. Primero los de entrada, luego los de entrada/salida, últimos los de salida. No los ubique por orden alfabético o aleatoriamente.
b) Haga uso de todos los parámetros.
c) Si diferentes métodos utilizan parámetros similares, incorpore los parámetros en un orden consistente.
d) Documente valores o rangos aceptables para los parámetros en la llamada al método. No espere a hacerlo cuando escribe el método.
e) Limite el número de parámetros a 7.
f) Considere nombres de los parámetros que hagan referencia a entrada, entrada/salida, modificación, salida.

g) Asegúrese de que parámetros actuales coincidan con parámetros formales. Por ejemplo, usar un entero cuando se requiere un punto flotante.

5.9) Consideraciones sobre parámetros en un método

Una función es un método que regresa un valor. Un procedimiento es un método que no regresa valores. La distinción entre estos dos tipos de métodos es semántica.

a) Una función regresa **un** valor. El nombre de la función debe estar relacionado al valor que regresa.
b) Un procedimiento no regresa valores. Su nombre está ligado a la actividad que en él se desarrolla. Los parámetros pueden ser de entrada, modificación y salida.
c) Recomendación: Use la función sólo si su propósito es regresar un valor asociado con el nombre de la función. De otra forma, use un procedimiento.

5.10) Listas de revisiones

Se presentan las listas de revisiones que ayudan a verificar el cumplimiento de los elementos presentados en esta sección.

Lista de revisiones: métodos de alta calidad

Propósito:	Verificar que se han considerado todos los aspectos importantes para el diseño de métodos de alta calidad.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
¿La razón para crear el método es suficiente?	
¿El nombre del método es apropiado? Use verbos como nombre del procedimiento y descripción del valor que regresa para la función.	
¿El nombre del método –procedimiento- describe todo lo que el método hace?	
¿Se han establecido convenciones de nombres para las operaciones comunes?	
¿Tiene cohesión el método?	
¿Es apropiada la longitud del método?	

Lista de revisiones: parámetros de métodos

Propósito:	Verificar que se han considerado todos los aspectos importantes al establecer los parámetros del método.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado

¿Los parámetros del método son suficientemente consistentes?	
¿Los parámetros se encuentran en orden apropiado –considerando además el orden en métodos similares-?	
¿Están documentados los parámetros?	
¿Los métodos tienen 7 o menos parámetros?	
¿Se utiliza cada parámetro de entrada?	
¿Se utiliza cada parámetro de salida?	
¿Si el método es una función, regresa siempre un valor válido?	

6- Clases

6.1) Tipo abstracto de datos (ADTs: Abstract Data Type)

Conjunto de datos y operaciones que manipulan dichos datos agrupados explícitamente.

a) Ventajas de usar ADTs.

a.1) Permiten ocultar (a los demás ADTs o partes del programa) los detalles de implementación.

a.2) Las actualizaciones se hacen de manera local, sin afectar el resto del programa.

a.3) Facilitan la definición de interfaces más descriptivas de su función y significado.

a.4) Facilitan el mejoramiento del rendimiento (enfocándose a los 2-3 métodos clave, fáciles de aislar e identificar).

a.5) Hacen que el código erróneo sea más fácil de identificar.

a.6) Hacen que el código sea más fácil de entender (auto-documentado).

a.7) Evitan el envío de datos continuamente entre distintas partes de un programa (excepto las que los necesitan, más fáciles de identificar).

a.8) Permiten trabajar con conceptos "aterrizados", con semántica clara en "el mundo real", más bien que con detalles de implementación súper-especializados.

b) Lineamientos para el uso de ADTs.

b.1) Referirse a un ADT independientemente (porque no importa) del medio en el que está almacenado o la forma en que está implementado (disco o RAM, lista, pila o arreglo, etc.)

b.2) Definir tipos de datos u objetos comunes en los programas como ADTs (e.g., Archivo).

b.3) Definir objetos simples como ADTs (e.g., Luz, que sólo tiene dos estados posibles, prendida o apagada, y dos formas de alterar su comportamiento: apagar o prender).

6.2) Clase

Implementación de un ADT en un lenguaje de programación particular, más la herencia y el polimorfismo que puedan estar asociados con dicho ADT en dicha implementación.

6.3) Interface de clase

Abstracción (descrita "públicamente") de lo que representa la clase (sus datos y operaciones visibles externamente).

a) Características de una buena interface de clase.

a.1) Debe ser independiente de los detalles de implementación, debe ser una descripción genérica/abstracta del concepto que se está representando.
a.2) Los conjuntos de datos y de comportamientos de la interface de clase deben ser lógicos, deben estar relacionados entre sí, deben ser conjuntos coherentes.

b) Lineamientos para la creación de interfaces de clases.

b.1) La interface debe presentar un nivel de abstracción uniforme.
b.2) La semántica de la abstracción representada por la clase debe ser clara (¿exactamente qué concepto representa la clase y qué significado tienen los objetos que son instancias de la clase?).
b.3) A la hora de decidir qué comportamientos se necesitan, verificar si se requiere proporcionar métodos "opuestos" en la interface. (Si hay un *apagaLuz*, ¿también se necesita un *prendeLuz*? Si hay un *leeEstatus*, ¿también se necesita un *modificaEstatus*?)
b.4) Si hay dos subconjuntos de métodos mutuamente excluyentes en la interface (uno de los cuales opera sobre algunos de los atributos y el otro sobre los demás atributos) quiere decir que realmente se tienen dos clases (que se están confundiendo como si fuera una sola).
b.5) Hacer que la interface dependa lo menos posible de suposiciones acerca de la forma en la que se van a usar los métodos (aunque dichas suposiciones estén documentadas, e.g., "el método A sólo se debe usar después del método B").
b.6) Evitar la posible "erosión" de la abstracción representada por una clase al modificar la clase (e.g., si una clase antes claramente representaba el concepto de Empleado y se tiene que modificar, asegurarse de que después de las modificaciones siga siendo una abstracción al mismo nivel, coherente, etc.).
b.7) No agregar miembros públicos a una clase cuando no son coherentes con la abstracción representada por la clase.
b.8) Asegurarse de que la abstracción representada por una clase sea "buena": muchas veces implica verificar la cohesión entre los miembros públicos de la clase.

6.4) Encapsulación

La abstracción permite "administrar" la complejidad de lo que se está modelando, haciendo que los detalles de implementación sean irrelevantes afuera de su alcance. La encapsulación es un concepto aun más restringido: hace que los detalles sean totalmente invisibles—aun si se quisieran conocer, no hay acceso a ellos desde afuera.

Lineamientos para la buena encapsulación:

a) Minimizar la accesibilidad de las clases y sus miembros.
b) No hacer que los atributos estén "expuestos" públicamente—hacerlos privados y permitir acceso a ellos (tanto para leer como para modificar sus valores) sólo a

través de métodos públicos (que pueden controlar lo que sucede al leer o modificar sus valores y "tomar nota" de las consultas o los cambios en caso necesario).

c) No incluir los detalles de implementación privados (i.e., que no deben ser relevantes afuera de la clase) en la interface de la clase.
d) No suponer nada acerca de los usuarios de las clases, sus conocimientos, sus habilidades, etc. Lo que describa la interface de la clase debe ser suficiente para que la clase se pueda usar correctamente, y la clase debe estar protegida en contra de posibles "usos incorrectos" (que puedan hacer que "truene" el programa o la clase).
e) No hacer que un método sea público sólo porque utiliza otros métodos públicos—la decisión de si se debe hacer público o no debe basarse en si su funcionalidad forma parte de la abstracción presentada por la interface de la clase o no.
f) En caso de duda, favorecer la legibilidad del código por encima de la modificabilidad del mismo (mucha gente va a tener que leer un método muchas veces más de las que se va a tener que modificar la forma en que está escrito el método).
g) Hacer que la encapsulación no sólo sea sintáctica, sino también semántica.
h) Evitar que haya conexiones demasiado fuertes entre dos clases (pues entonces no se habrán logrado aislar los miembros de una clase de la otra clase y se habrá violado la encapsulación).

6.5) Aspectos de implementación

a) Implementar relaciones *has-a* (tiene-un) mediante membresía como primera opción, y a través de la herencia privada sólo en caso de que no haya otra opción. Ejemplo 1: los empleados tienen un nombre, una dirección, un sueldo, etc. Por eso *nombre*, *dirección* y *sueldo* son miembros (atributos) de la clase *Empleado*. Ejemplo 2: las listas simplemente ligadas tienen un nodo que es el primer nodo de la lista. Por eso *primero* es un atributo (de tipo *Nodo*) de la clase *ListaSimplementeLigada*.
b) Una clase que tenga más de 7+/-2 atributos quizá se pueda subdividir y volverse varias clases, pues en caso contrario representa un concepto demasiado complejo, difícil de entender y usar de manera correcta. Si los atributos son de tipos primitivos (*int*, *double*, etc.) se pueden permitir hasta 9, si son de tipos compuestos no debería haber más de 5.
c) Implementar relaciones *is-a* (es-un) mediante herencia. Si las instancias de una clase derivada no son casos más especializados de las instancias de la superclase, quizá la herencia no es el mecanismo correcto para definir la relación entre las dos clases. Liskov Substitution Principle (LSP): debe poderse usar una subclase a través de la interface de su clase base sin que el programador que esté usando dicha interface necesite saber que existe una diferencia entre las dos clases.
d) Si una clase no está diseñada para que se deriven otras clases de ella, documentarlo y además prohibir la herencia explícitamente (definiendo a la clase como *final* en Java, haciendo que sus miembros no sean *virtual* en C++, haciendo que sus miembros no sean *overridable* en Visual Basic, etc.).

e) Asegurarse de que se hereden únicamente los miembros que se desea que se hereden.
f) No reusar nombres de miembros privados de superclases en clases derivadas (se presta a confusión, aunque al ser privados no se heredan y los miembros "equivalentes" en las clases derivada sean diferentes).
g) Colocar las interfaces, los atributos y los métodos más comunes lo más alto posible en la jerarquía de clases (pues se necesitan tener disponibles comúnmente, i.e., en muchas de las clases y subclases derivadas) con tal de que no se viole la semántica de la abstracción representada por la clase en la que se coloquen.
h) Las clases de las que sólo existe una instancia son sospechosas, pues podría ser que la "clase" realmente es una instancia (¡y que se tiene una instancia de dicha instancia!)
i) Las clases base de las que sólo existe una clase derivada son sospechosas, pues la clase derivada podría ser la única que realmente se necesita definir como tal.
j) Si una clase derivada redefine un método "heredado" y no hace nada en esa redefinición, quiere decir que la definición original (heredada) no era correcta, no contemplaba todos los casos, no era realmente genérica. Mejor corregir o completar la definición del método en la clase base.
k) No crear jerarquías de clases demasiado complejas. El tamaño máximo de una jerarquía debe ser tal que contenga un total de 7+/-2 clases interrelacionadas (i.e., quizá tendrá 2-3 niveles).
l) Aprovechar el polimorfismo (i.e., versiones diferentes, en las clases derivadas, de métodos que se llaman de la misma forma) en lugar de usar estructuras de control de tipo *case* para identificar de qué tipo es el objeto que se esté manipulando en ese momento para saber a qué método invocar.
m) Evitar la herencia múltiple lo más posible debido a la complejidad que introduce y al hecho de que muchas veces se desea usar para crear una nueva "clase" que combina las características de otras pero que no representa claramente una abstracción nueva, una unidad coherente.

6.6) Lineamientos para la implementación de atributos y métodos

a) Minimizar lo más posible la cantidad de métodos de una clase.
b) Deshabilitar explícitamente la generación automática, por parte del compilador, de métodos y operadores indeseados (e.g., en C++ se genera automáticamente un operador de asignación para cada clase que se defina, en Java se genera automáticamente un constructor vacío si hay alguna otra versión del constructor definida para una clase, etc.). Una forma de lograr esto es definiéndolos (aunque estén vacíos) y haciéndolos privados.
c) Minimizar la cantidad de llamadas hechas desde un método a otros métodos (*fan-out*).
d) Minimizar la cantidad de llamadas indirectas (a través de concatenaciones del operador punto) a otras clases. La ley de Démeter: el objeto A puede llamar a cualquiera de sus métodos, y si el objeto A crea una instancia del objeto B, el objeto A puede invocar a cualquiera de los métodos de B, pero si B regresa algún otro objeto C, el objeto A no debe invocar (directamente) a los métodos de C.

e) Minimizar la cantidad de interconectividad ("colaboración") que hay entre las clases, incluyendo la cantidad de objetos instanciados, la cantidad de métodos invocados sobre objetos instanciados, la cantidad de métodos invocados sobre objetos regresados por objetos instanciados, etc.
f) Todos los atributos deben ser inicializados en todos los constructores, si es posible.
g) Si se necesita tener una copia de un objeto (i.e., dos instancias diferentes pero idénticas en los valores de sus atributos), preferiblemente que sea una copia "profunda" (el valor de cada atributo de la copia es una copia del valor original) más bien que "no profunda" (el valor de cada atributo de la copia es una referencia o apuntador al valor original), con el fin de minimizar complejidad e interdependencia entre los objetos (a cambio usando un poco más de memoria).
h) Minimizar (en Java) los métodos get's y set's. Ayuda a mantener el encapsulamiento

6.7) Lineamientos para la definición de clases

a) Una clase puede modelar objetos del "mundo real" u objetos abstractos.
b) Una clase debe definirse para reducir la complejidad de un programa, para evitar tener que pensar en sus detalles internos de implementación (una vez que ya está definida), para minimizar el número de líneas de código de los programas, para facilitar el mantenimiento de los programas, para hacer que sea más fácil identificar los lugares en los que ocurren errores en un programa, para limitar el impacto de los cambios en los programas, para facilitar la reusabilidad.
c) Las clases facilitan el trabajo en equipo, la creación de "familias" de programas relacionados, el manejo de datos globales o compartidos.
d) Evitar clases todo-poderosas (que continuamente acceden a y modifican los valores de los atributos de otras clases), clases irrelevantes (e.g., clases con muchos atributos pero sin métodos) y clases que representan verbos/acciones (clases con muchos métodos pero sin atributos).
e) El nombre de la clase debe ser un sustantivo singular, que indique claramente el concepto que está representando. Evitar nombres como: Ejercicio20.

6.8) Listas de revisiones

Se presentan las listas de revisiones que ayudan a verificar el cumplimiento de los elementos presentados en esta sección.

Lista de revisiones: tipos abstractos de datos y abstracción

Propósito:	Verificar que se han considerado todos los aspectos importantes relacionados a los ADTs y a la abstracción.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado

ADTs	
¿Se evaluaron y diseñaron las clases en el programa desde el punto de vista de tipos de datos abstractos?	
Abstracción	
¿Cada clase tiene un objetivo principal/central?	
¿Cada clase tiene un nombre descriptivo de su objetivo principal?	
¿La interface de cada clase presenta una abstracción coherente?	
¿La interface de cada clase permite que sea obvia la manera de usar la clase?	
¿La interface de cada clase es lo suficientemente abstracta para que no se tenga que saber (ni pensar en) cómo están implementados los servicios que proporciona la clase? En otras palabras, ¿se puede tratar la clase como si fuera una caja negra?	
¿Los servicios que proporciona cada clase son lo suficientemente completos para que las demás clases no tengan que interferir con sus datos internos?	
¿Se ha removido de cada clase la información irrelevante (no relacionada con la demás información contenida en esa clase)?	
¿Se ha considerado subdividir cada clase en clases parciales, y se ha realizado esta subdivisión lo más posible?	
¿Conforme se ha modificado cada clase se ha mantenido la integridad de la interface de la clase?	

Lista de revisiones: encapsulación

Propósito:	Verificar que se han considerado todos los aspectos importantes de la encapsulación.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
¿Cada clase minimiza lo más posible el acceso a sus miembros?	
¿Cada clase evita lo más posible exponer (al exterior de la clase) sus atributos?	
¿Cada clase esconde lo más posible los detalles de su implementación?	
¿Cada clase está diseñada de tal forma que evite hacer suposiciones sobre sus usuarios (incluyendo sus clases derivadas)?	
¿Cada clase es independiente de las demás?	

Lista de revisiones: herencia

Propósito:	Verificar que se han considerado todos los aspectos importantes de la herencia.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado

¿La herencia se está usando únicamente para modelar relaciones (entre clases) de tipo es-un ("is-a")? Es decir, las clases derivadas cumplen con el Principio de Sustitución de Liskov?	
¿La documentación de la clase describe claramente la estrategia de herencia utilizada en un programa?	
¿Las clases derivadas evitan re-definir métodos que no debieran ser re-definibles (por motivos de herencia)?	
¿Las interfaces, los atributos y los métodos comunes entre varias clases están lo más alto posible dentro de la jerarquía de clases?	
¿Las jerarquías de clases son relativamente "chaparras"?	
¿Todos los atributos de la clase base de una jerarquía son privados más bien que protegidos?	

Lista de revisiones: implementación de clases

Propósito:	Verificar que se han considerado todos los aspectos importantes relacionados a la implementación.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
Lenguaje utilizados para la implementación	
¿Se han investigado los aspectos específicos del lenguaje de implementación con respecto a la forma de definir clases?	
Implementación	
¿Cada clase contiene siete o menos atributos?	
¿Cada clase minimiza las llamadas directas e indirectas a los métodos de otras clases?	
¿La clase tiene un nombre adecuado al concepto que representa?	
¿Cada clase "colabora" (interactúa) con las demás sólo cuando realmente es necesario?	
¿Los atributos de cada clase se inicializan en el método constructor de la clase?	
¿Cada clase está diseñada para que sus instancias sean copias "profundas" más bien que poco "profundas" (a menos de que haya una muy buena razón para que sean poco "profundas")?	

7- Comentarios

"*The main contributor to code-level documentation isn't comments, but GOOD PROGRAMMING STYLE*" (McConnell, 2004).

7.1) Líneas generales para escribir comentarios

a) Los buenos comentarios no repiten el código o lo describen. Deben explicar, a un nivel más alto de abstracción que el código, qué es lo que se intenta hacer.
b) Los comentarios deben ser como los subtítulos de un libro o la tabla de contenido del programa.
c) Si los comentarios no son "buenos", son poco útiles y peor que no haber puesto nada.

7.2) Categorización de comentarios

Existen seis tipos de comentarios, los que:

a) Repiten el código: no aportan nada nuevo.
b) Explican el código: son útiles cuando el código es confuso…la solución no es comentarlo sino ¡MEJORARLO!
c) Marcan el código: empleados durante el desarrollo para hacer hincapié en trozos de código por mejorar o terminar. Si están estandarizados en estilo permiten al programador realizar búsquedas mecánicas para corregir/terminar el código.
d) Resumen el código: idealmente en una o dos líneas.
e) Describen la intención del código: sirven para explicar el propósito de una sección de código. Deben operar al nivel del problema más que de la solución.
f) Ofrecen información que no puede ser proporcionada por el código en sí mismo: copyright, números de versión, referencias, notas acerca del diseño, etc.

En un programa terminado sólo debe haber comentarios del tipo d, e y f.

7.3) Comentar de manera eficiente

a) Usar estilos que no cueste trabajo mantener y permitan modificar fácilmente lo ya escrito. (Por ejemplo: no enmarcar líneas con * alineados al principio y fin de la misma, no rellenar con … para agregar un comentario en una línea, etc.)
b) Usar el proceso de programación con pseudo código para reducir el tiempo de escribir comentarios. Se refiere a describir en lenguaje natural qué se hará, posteriormente se programa y los comentarios ya están escritos.
c) Integrar la escritura de comentarios al estilo de desarrollo. No dejar los comentarios para cuando ya se haya terminado el proyecto.
d) Emplear como máximo una densidad de un comentario por cada diez líneas de código.

7.4) Técnicas para escribir comentarios

a) Comentar líneas individuales. Si el código está bien escrito, sólo es necesario comentar líneas cuando:
   a.1) La línea es suficientemente complicada como para ameritar una explicación, o
   a.2) La línea tuvo alguna vez un error y se quiere dejar registro de éste.

b) Comentarios al final de una línea. Son difíciles de formatear y mantener, además al tener que ser cortos suelen sacrificar claridad. Debido a lo anterior es mejor evitarlos, salvo en los siguientes casos:
b.1) Cuando se describen variables al declararlas.
b.2) Para marcar fin de bloques (por ejemplo //end while)

c) Comentar párrafos de código. Idealmente 1 o 2 líneas que anteceden al trozo de código que se desea explicar/describir. Algunos lineamientos son:
c.1) Los comentarios deben ser al nivel de "intención", por ejemplo para el siguiente trozo de código:

```
done = false;
maxLen = inputString.length();
i = 0;
while ( !done && ( i < maxLen)) {
        if ( inputString.charAt(i) == '$' ) {
                done = true;
        }
        else
                i++;
}
```

Un comentario de párrafo adecuado antes del while será

```
// find the command-word terminator ($)
```

y uno poco adecuado sería:

```
//find "$" in inputString
```

c.2) Enfocar los comentarios de párrafo en el *por qué* más que en el *cómo*.
c.3) Usar comentarios para preparar al lector a lo que sigue. Un buen comentario siempre precede al código que describe.
c.4) Concentrar los esfuerzos en hacer el código legible y luego sólo si es necesario agregar comentarios.
c.5) Evitar abreviaturas.
c.6) Establecer diferencias entre comentarios mayores y comentarios menores, por ejemplo agregando una línea que marque el fin del comentario mayor, o poniendo comentarios mayores que inicien con mayúscula mientras que los menores inician con minúscula, o iniciar los comentarios menores con … en vez de al margen. Los comentarios menores son detalles al comentario mayor.
c.7) Comentar cualquier parte de código que puede provocar error de ejecución en ciertas circunstancias, o alguna característica del lenguaje o ambiente que no esté documentada.
c.8) Justificar violaciones al estilo de programación.
c.9) No comentar código "tricky": ¡rescribirlo!

d) Comentar declaraciones de datos. Sirve para describir aspectos de la variable que su nombre no puede describir por sí mismo. Es conveniente escribir comentarios que describan:
d.1) Las unidades de los datos numéricos.
d.2) El rango permitido de valores numéricos.
d.3) El significado de un cierto código (si el lenguaje soporta tipos enumerados usarlos, en caso contrario describir cada uno).

d.4) Limitantes en los datos de entrada (por ejemplo el máximo valor de n cuando representa el número de celdas de un arreglo).
d.5) Las banderas a nivel de bits de variables tipo bit field.
d.6) Si las variables son globales.

e) Comentar estructuras de control. Idealmente se realiza justo antes de la estructura que se desea describir y al final con un comentario de línea. Este último comentario es muy útil cuando se encuentran ciclos anidados o muy largos. Si el código final tiene muchos comentarios de fin de estructura de control, valdría la pena revisarlo para simplificarlo.

f) Comentar métodos. Usualmente se realiza como un prólogo a la declaración de cada método. Típicamente es necesario que en el prólogo haya que:
f.1) Incluir una descripción del método en una o dos líneas. Si no se puede describir de manera breve quizá haya que re-pensar si está bien hecho y posiblemente re-programarlo.
f.2) Diferenciar parámetros de entrada y parámetros de salida. La documentación de parámetros es en el momento de declararlos (uno por línea, con comentarios de una línea), sólo si el nombre de los mismos no es suficientemente descriptivo.
f.3) Documentar el estado en el que se espera recibir ciertos parámetros (por ejemplo arreglos ordenados, valores validados, etc.).
f.4) Comentar las limitaciones del método, por ejemplo precisión del valor de salida si es numérico.
f.5) Documentar los efectos globales del método, a qué variables globales afectó y cómo.
f.6) Documentar la fuente (libro, revista) de los algoritmos que se están usando.

No todas las partes TIENEN que venir en el prólogo, sólo las que se consideren necesarias dependiendo de la complejidad del método.

"Good routine names are key to routine documentation" (McConnell, 2004).

g) Comentar clases, archivos y programas. Líneas generales para comentar clases:
g.1) Describir la filosofía de diseño, alternativas de diseño que se consideraron y descartaron, etc.
g.2) Describir limitaciones, suposiciones sobre datos de entrada y salida, responsabilidades en el manejo de errores, efectos globales, referencias de fuentes de algoritmos, etc.
g.3) Comentar la interface de clase, al menos cada parámetro y cada valor regresado.
g.4) No documentar los detalles de implementación en la interface de clase.

h) Líneas generales para documentar archivos.
h.1) Describir el propósito y contenido de cada archivo.
h.2) Escribir nombre, dirección de correo electrónico y teléfono en un comentario de bloque.
h.3) Incluir una etiqueta de control de versión. En CVS
// id
automáticamente se expande en

// $id: ClassName.java.v 1.1 2007/02/15 6:02:43 ismene Exp $

h.4) Incluir avisos (notices) legales en un comentario de bloque.

h.5) Poner un nombre al archivo que describa su contenido.

Además es **importante**, en el diálogo de comentario a métodos y fuera de él, agregar comentarios que marquen al programa, de esta manera se puede buscar/encontrar fácilmente ciertas partes del código. En Java y C++ es común usar:

/**	inicio de método
@keyword	donde keyword es un código que describe el tipo de comentario: @param para parámetros @version @throws

7.5) Listas de revisiones

Se presentan las listas de revisiones que ayudan a verificar el cumplimiento de los elementos presentados en esta sección.

Lista de revisiones: documentación general

Propósito:	Verificar que se han considerado todos los aspectos importantes relacionados a la documentación de código en general.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
¿Es posible que alguien lea el código e inmediatamente lo entienda?	
¿Los comentarios explican la intención del código o resumen lo que hace, en vez de sólo repetirlo?	
¿Se usa el proceso de programación con pseudo código para reducir el tiempo de escritura de comentarios?	
¿El código confuso ha sido re-escrito en vez de comentado?	
¿Los comentarios están actualizados?	
¿Los comentarios son claros y correctos?	
¿El estilo de comentarios permite que sean fácilmente modificados?	

Lista de revisiones: documentación de sentencias y párrafos

Propósito:	Verificar que se han considerado todos los aspectos importantes relacionados a la documentación de sentencias y párrafos.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
¿El código evita comentarios de fin de línea?	
¿Los comentarios se enfocan en el *por qué* más que en el *cómo*?	
¿Los comentarios preparan al lector para el código que sigue?	
¿Cada comentario es importante?, ¿Los comentarios redundantes han sido borrados o mejorados?	
¿Las sorpresas han sido documentadas?	
¿Se han evitado las abreviaturas?	
¿Es clara la distinción entre comentarios mayores y comentarios menores?	
¿Está documentado el código alrededor de un error o característica del lenguaje de programación no documentada?	

Lista de revisiones: documentación de declaración de datos

Propósito:	Verificar que se han considerado todos los aspectos importantes relacionados a la documentación de la declaración de datos.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
¿Han sido comentadas cada una de las variables del programa?	
¿Han sido comentados los rangos de valores numéricos de las variables?	
¿Han sido comentadas las restricciones en datos de entrada?	
¿Se han comentado las banderas a nivel de bits?	
¿Cada una de las variables globales han sido comentadas cuando se declararon?	
¿Cada una de las variables globales han sido identificadas por convención de nombre o comentario o ambas?	
¿Los números mágicos han sido remplazados por constantes o variables en vez de sólo haber sido comentados?	

Lista de revisiones: documentación de estructuras de control

Propósito:	Verificar que se han considerado todos los aspectos importantes relacionados a la documentación de las estructuras de control.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
¿Cada sentencia de control ha sido comentada?	
¿El final de cada estructura de control compleja ha sido comentada o, de ser posible, simplificada para no necesitar comentario?	

Lista de revisiones: documentación de métodos

Propósito:	Verificar que se han considerado todos los aspectos importantes relacionados a la documentación de métodos.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

Elemento	Revisado
¿El propósito de cada método ha sido comentado?	
¿Han sido comentadas las características de los métodos (cuando sean relevantes) como: datos de entrada/salida, suposiciones sobre interfaces, limitaciones, correcciones de errores, efectos globales y fuentes de los algoritmos empleados?	

Lista de revisiones: documentación de archivos, clases y programas

Propósito:	Verificar que se han considerado todos los aspectos importantes relacionados a la documentación de la declaración de datos.
Uso:	Conforme verifiques que se ha cubierto cada elemento, márcalo en el recuadro de la derecha según corresponda. Si lo consideras necesario, puedes incluir nuevos elementos a la lista.

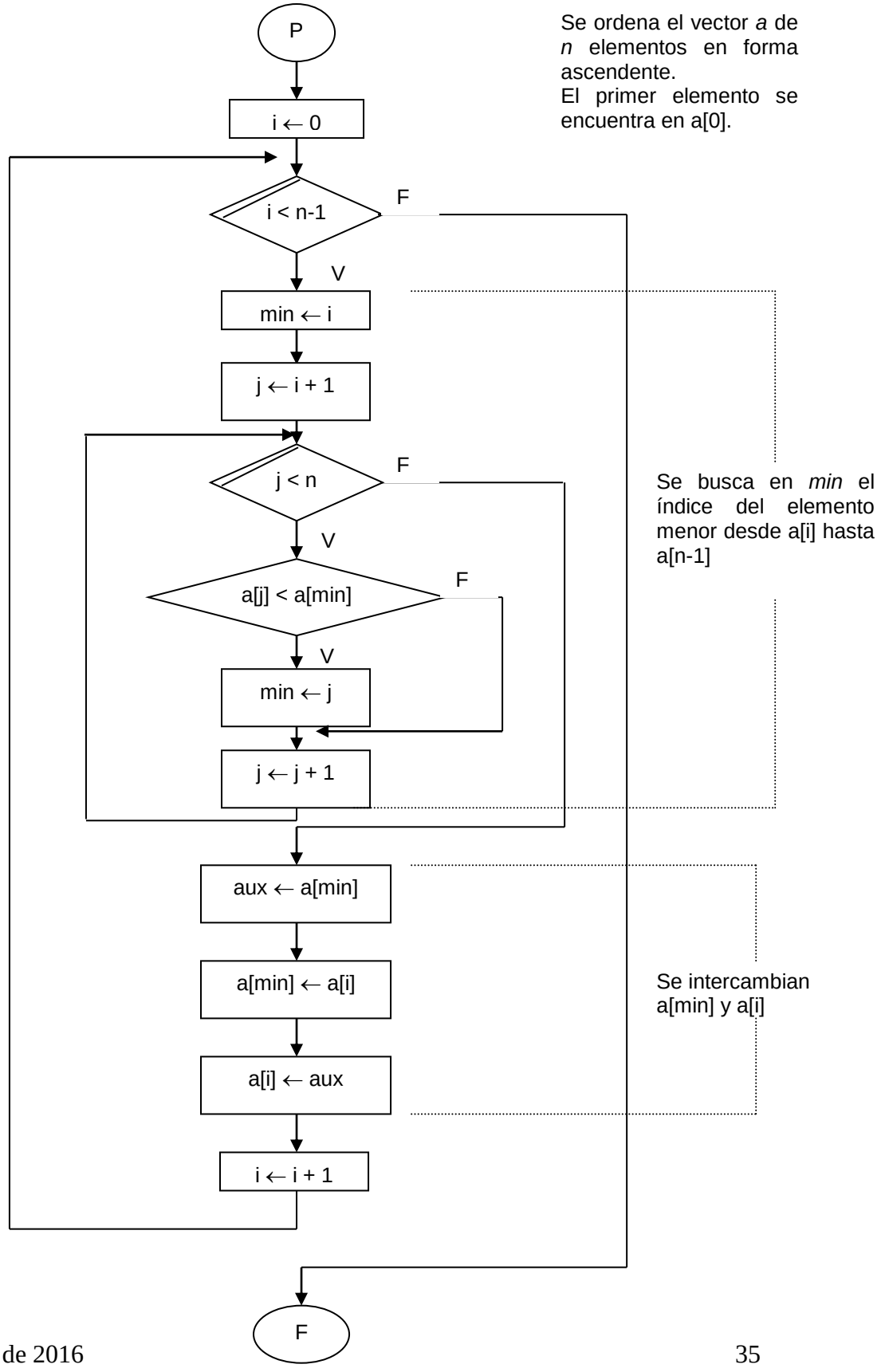
Elemento	Revisado
¿El propósito de cada archivo ha sido descrito?	
¿En el listado se encuentra el nombre del autor (programador), dirección de correo electrónico y teléfono?	

Referencia

(McConnell, 2004). McConnell, S. *Code Complete 2*, Microsoft, segunda edición, 2004.

Apéndice

ALGORITMO DE ORDENAMIENTO: SELECCIÓN DIRECTA

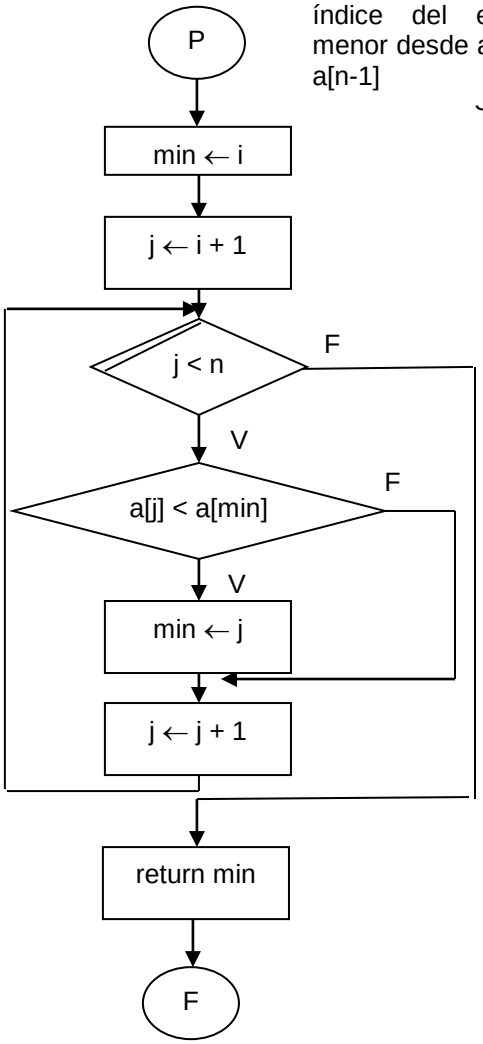


ALGORITMO DE ORDENAMIENTO: SELECCIÓN DIRECTA

Utilizando funciones para encontrar el mínimo y realizar el intercambio

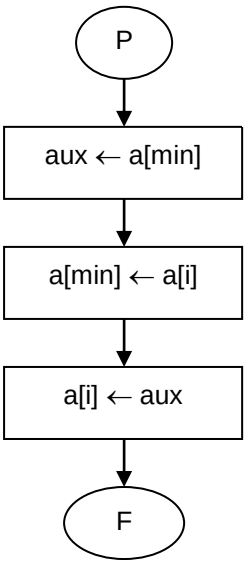
Submétodo: minimo

Se busca en *min* el índice del elemento menor desde a[i] hasta a[n-1]

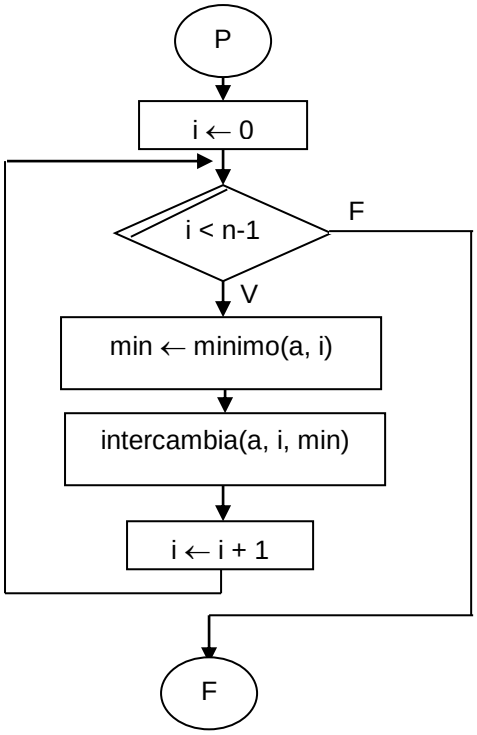


Submétodo: intercambia

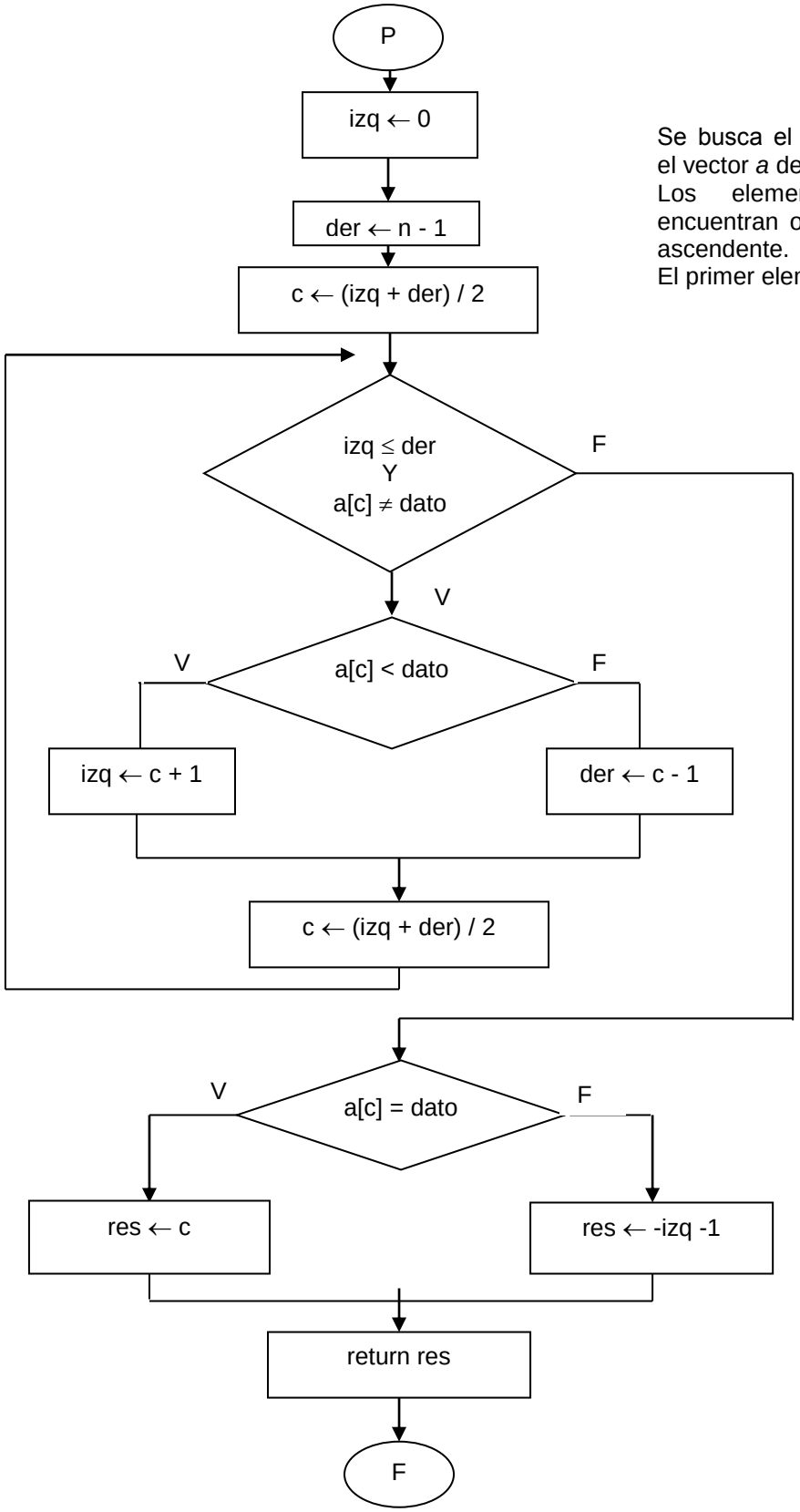
Se intercambian a[min] y a[i]



Se ordena el vector *a* de *n* elementos en forma ascendente.
El primer elemento se encuentra en a[0].



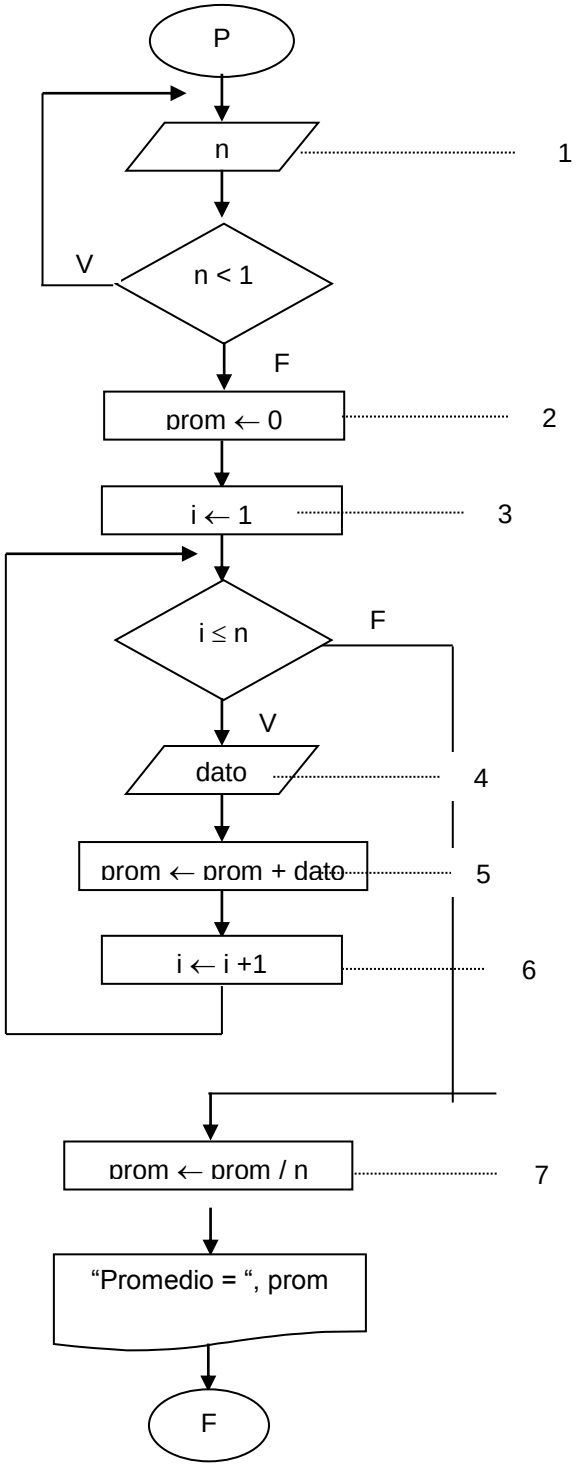
ALGORITMO DE BÚSQUEDA: BINARIA



Se busca el elemento "*dato*" en el vector *a* de *n* elementos.
Los elementos de *a* se encuentran ordenados en forma ascendente.
El primer elemento está en a[0].

REPRESENTACIÓN DE LA MEMORIA

Leer n enteros y calcular su promedio. Datos de prueba: 0, 3, 2, 4, 3



1. Establecer los datos de prueba.
2. Crear la tabla poniendo como encabezados de las columnas todas las variables que intervienen en el algoritmo.
3. Seguir el algoritmo anotando en cada fila el nuevo valor de la variable que se modificó en la misma.

instrucción	n	prom	i	dato
1	0			
1	3			
2		0		
3			1	
4				2
5		2		
6			2	
4				4
5		6		
6			3	
4				3
5		9		
6			4	
7		3		

IMPRIME

Promedio = 3