Buenas Practicas

Autor : Angel Castiglia

Email : castiglia_angel@yahoo.com.ar , angel@castiglia.net

Buenos Aires, Argentina

Buenas Prácticas

Sobre el documento

Este documento presenta una serie de temas relacionados con las buenas practicas, cual está dividido en dos partes.

La primera parte es una introducción y toca temas como la necesidad de tener un método claro de desarrollo, la construcción de reglas claras aplicables al método, las actividades que incumben al ciclo de vida del desarrollo, utilizar patrones de diseño, el pase del diseño a la implementación y allí se tratan temas como la persistencia y las convenciones que se deben seguir para escribir un código claro y que sea fácil de mantener. Al finalizar el tratamiento de cada uno de los temas antes mencionados hay una conclusión, esto significa que cada una de las conclusiones tiene su fundamento teórico (el porque).

La segunda parte trata ya mas en detalle las buenas prácticas como las convenciones recomendadas para escribir código Java sugeridas por Sun.

Buenas Prácticas – Primera Parte

Introducción

Hacia un método de desarrollo

La experiencia y un uso acertado de la práctica es importante para realizar un desarrollo pero no es suficiente, se deben seguir una serie de reglas (método). Las “buenas prácticas” no solo son aplicables a la programación sino a todo el proceso de desarrollo.

Un proceso de desarrollo de programas tiene como objetivo la formalización de las actividades relacionadas con la elaboración de sistemas informáticos. La formalización de un proceso de desarrollo tiende a dotar a las empresas de un conjunto de mecanismos que, cuando se aplican sistemáticamente, permiten obtener de manera repetitiva y fiable sistemas de programas de calidad constante. Por naturaleza, la descripción del proceso es general porque no es posible definir autoritariamente un estándar único, adoptando a todas las personas, a todos los tipos de aplicaciones y a todas las culturas. Conviene mas bien hablar de un marco configurable, eventualmente refinado, de manera consensuada por la práctica y la implementación de productos ampliamente adoptados por la comunidad de usuarios.

Un método de desarrollo comprende:

- elementos de modelado que son los módulos conceptuales básicos;
- una notación cuyo objetivo es asegurar la coherencia visual de los elementos de modelado;
- un proceso que describe las etapas a seguir en el desarrollo del sistema;
- experiencia, mas o menos formalizada.

La segmentación del modelo permite gestionar la complejidad reduciendo la amplitud del estudio a una parte, un subconjunto o un punto de vista. De esta manera, cuando el todo es demasiado complejo para ser comprendido de un solo golpe, la comprensión global puede extraerse por la percepción paralela de varias vistas disyuntivas pero concurrentes. La elección de los punto de vistas, es decir, de lo que se ha modelado, influencia en gran medida en la manera de aproximar el problema y , por lo tanto, la forma de las soluciones adoptadas. No existe un modelo universal y los niveles de detalle, de precisión o de fidelidad pueden variar. Los mejores modelos están en contacto con la realidad.

Conclusión : es una “buena práctica” tener un método claro de desarrollo.

Construcción de software - Método

Construcción de buenas reglas

Ciertamente no es fácil legislar sobre construcción de software y es grande el peligro de producir reglas inútiles, poco meditadas o incluso dañinas. Las principios que se enumeran a continuación, basadas en el análisis del papel de la metodología en el software, pueden ayudarnos a evitar tales peligros.

- Bases teóricas : las reglas de metodología del software deben basarse en una teoría sobre el tema subyacente.
- Bases prácticas : las reglas de metodología del software deben estar respaldadas por una amplia experiencia práctica.
- Experiencia en reutilización
- Tipología de reglas : una regla pude ser de recomendación (invita a seguir un cierto estilo) o absoluta (plantea que hay que trabajar de una cierta manera); y puede anunciarse de forma positiva (indicando lo que se debe hacer) o negativa (indicando lo que no se debe hacer).
- Excepciones : si una regla metodológica presenta una guía que es aplicable en general pero que tiene excepciones, las excepciones deben anunciarse como parte de la regla.

Conclusión : es una "buena práctica" basar el método sobre reglas claras.

Proceso de desarrollo - Actividades

Un proceso de desarrollo es un método de organizar las actividades relacionadas con la creación, presentación y mantenimiento de los sistemas de software.

Una de las características principales de un proceso de desarrollo es el ciclo de vida iterativo. Un ciclo de vida iterativo se basa en el agrandamiento y perfeccionamiento secuencial de un sistema a través de múltiples ciclos de desarrollo de análisis, diseño, implementación y pruebas. El ciclo de vida iterativo se basa en una idea muy simple, cuando un sistema es demasiado complejo para ser comprendido, diseñado o realizado de golpe, o incluso las tres cosas a la vez, es mejor realizarlo en varias etapas por evoluciones. En la naturaleza, los sistemas complejos que funcionan son siempre evoluciones de sistemas mas simples. Las iteraciones deberán estar controladas por casos de uso, es decir que los ciclos de vida iterativos de desarrollo se organizan a partir de los requerimientos del caso de uso.

El ciclo de vida iterativo se basa en la evolución de prototipos ejecutables, medibles y, por lo tanto, en la evaluación de elementos concretos. Se opone al ciclo de vida en cascada que se basa en la elaboración de documentos. Las entregas fuerzan al equipo a dar resultados concretos regularmente, lo que permite evitar el síndrome del 90% terminado, con aún el 90% por hacer. El desarrollo regular de las

iteraciones facilita el tener en cuenta los problemas; los cambios se incorporan en las iteraciones futuras en lugar de distraer e interrumpir los esfuerzos.

A lo largo del desarrollo, se muestran ciertos prototipos a los usuarios y a los clientes. La demostración de los prototipos presenta numerosas ventajas:

- El usuario se coloca delante de situaciones de uso concretas que le permiten estructurar mejor sus deseos y comunicarlos al equipo de desarrollo;
- El usuario se hace colaborador del proyecto; toma su parte de responsabilidad en el nuevo sistema, y de hecho, lo acepta más fácilmente;
- El equipo de desarrollo está más motivado debido a la proximidad del objetivo;
- La integración de los diferentes componentes del programa se realiza de manera progresiva, durante la construcción, sin el efecto bin bang al aproximarse al fecha de entrega;
- Los progresos se miden por programas demostrables en lugar de documentos o estimaciones como en el ciclo de vida en cascada. Se dispone así de elementos objetivos y pueden evaluar los progresos y el estado de avance con mayor fiabilidad.

En contrapartida, el ciclo de vida iterativo exige más atención e implicación de todos los actores del proyecto. Debe ser presentado y comprendido por todos: los clientes, los usuarios, los desarrolladores, el control de calidad, los probadores, los documentalistas. Todos deben organizar su trabajo en consecuencia.

Conclusión :es una “buena práctica” realizar iteraciones en el ciclo de vida de un desarrollo.

Patrones de diseño

Dividir un problema en partes siempre ha sido uno de los objetivos de una buena programación orientada a objetos. Para ello utilizaremos Patrones de Diseño (Design Patterns) que son modelos de trabajo enfocados a dividir un problema en partes de modo que nos sea posible abordar cada una de ellas por separado para simplificar una resolución. Un patrón de diseño es un conjunto de reglas que describen como afrontar tareas y solucionar problemas que surgen durante el desarrollo de software. En este documento se realizará una breve explicación de los que se utilizan en el modelo.

¿ De donde salen los patrones de diseño ?

Su origen es aplicable al arquitecto Christopher Alexander que los aplico a la arquitectura en los años 70 y se han vuelto tan "famosos" en todo el mundo menciona por una cosa llamada "la pandilla de los cuatro" (Gang of Four) o GoF.
El reciente inter s del mundo del software por los patrones tiene su origen, o mejor dicho, su explosion a partir de 1995, tras la aparicion y el exito del libro "Design Patters: Elements of Reusable Object-Oriented Software" de la pandilla de los cuatro. Ellos, Erich

Gamma, Richar Helm, Ralph Johnson y John Vlissides, se dedicaron a recopilar una serie de patrones (hasta 23) aplicados habitualmente por expertos dise adores de software orientado a objetos, y al hacerlos p blicos... la "fiebre" estallo, pero como os he dicho, ni son los inventores, ni son los nicos implicados, solo les menciono porque es realmente imposible leer algo sobre patrones y no hacerlo sobre la GoF.
Por ultimo mencionare a Craig Larman, que es uno de los autores mas interesantes que he encontrado sobre el tema, que ha definido de los patrones GRASP (patrones generales de software para asignar responsabilidades), los cuales mencionare tambi n aqu .

Dos conceptos clave

Existen dos términos que aparecerán en muchas ocasiones y que son un objetivo permanente del diseño orientado a objetos, como son la cohesión y el acoplamiento.
Podr amos definir la cohesi n de una clase (o de un paquete, o de lo que sea) como la relaci n entre los distintos elementos de la clase, normalmente sus m todos. Es decir, la cohesi n dice que todos los elementos de una clase tiene que trabajar en la misma direcci n, es decir, hacia un mismo fin. Por ejemplo, una clase "Coche" deber a ocuparse de cosas relacionadas con el coche en si, como acelerar y frenar, pero no de cosas ajenas a l como manipular informaci n referente a su seguro. La cohesi n es una medida relativa, en el sentido de que depende de lo que cada uno piense que es la funci n de la clase, pero lo importante es mantener una cohesi n lo mas alta posible. Existen diferentes tipos de cohesi n (funcional, secuencial, etc), pero creerme si os digo que eso ahora mismo no es importante.
Respecto al acoplamiento, se podr a decir que es la interdependencia existente entre dos clases, paquetes, etc. Esto ocurre normalmente cuando una clase (o paquete) necesita saber demasiados detalles internos de otra para su funcionamiento, es decir, rompe el encapsulamiento del que tanto se habla en la programaci n orientada a objetos. Tambi n existen diversos tipos de acoplamiento (funcional, de datos, etc.), pero al igual que ocurr a con la cohesi n, eso no nos importa ahora, no es el objetivo de estos art culos. Por supuesto, para tener un dise o correcto, f cil de mantener y modular, cuanto m s bajo acoplamiento haya entre las clases (o paquetes), pues mejor.

Conclusión : es una “buena práctica” utilizar patrones de diseño.

Del diseño a la implementación

La escritura de código es una extensión del proceso de diseño. Hay que tomar decisiones mientras se escribe el código, pero cada una de ellas debería de afectar solamente a una pequeña parte del programa, de tal manera que fuera fácil cambiarlas. El código del programa es la personificación última de la solución del problema, así que la forma en que se escriba será importante para la mantenibilidad y la extensibilidad.
La manera más fácil de implementar un diseño orientado a objetos colleva el uso de un lenguaje orientado a objetos, pero incluso los lenguajes orientado a objetos ofrecen distinto grados de apoyo para los conceptos orientado a objetos. Cada lenguaje supone un compromiso entre potencia conceptual, eficiencia y compatibilidad.

Conclusión : es una “buena práctica” realizar un exhaustivo examen del lenguaje de programación que se utilizará, para que se ajuste a los requerimientos del desarrollo.

Persistencia

Ejecutar una aplicación orientada a objetos implica crear y manipular un cierto número de objetos. ¿ Qué sucederá con estos objetos cuando la ejecución finalice ?. Los objetos transitorios desaparecerán con la sesión en curso; pero hay muchas aplicaciones que también necesitan objetos persistentes, que sigan existiendo entre sesiones consecutivas. Quizás sea necesario compartir objetos persistentes entre varias aplicaciones, lo cual hace surgir la necesidad de una base de datos

Esta visión general de los problemas de la persistencia, y de sus soluciones, se examinarán tres enfoques que los desarrolladores orientado a objetos tienen a su disposición para manipular objetos persistentes. Pueden basarse en mecanismos de persistencia del lenguaje de programación, y del entorno de desarrollo para sacar y meter estructuras de objetos en un almacenamiento permanente. Pueden combinar la tecnología de objetos con base de datos de las clases mas frecuentemente disponible (no orientada a objetos) : las bases de datos relacionales. O bien pueden utilizar uno de los novedosos sistemas de base de datos orientadas a objetos que pretenden trasladar a las bases de datos ideas básicas de la tecnología de objetos. Otros mecanismos de almacenamiento pueden ser archivos planos, base de datos jerárquicas, etc..

Cualquiera sea la base de datos que se utilice se debe cumplir con el principio de clausura de la persistencia. Dicho principio dice que siempre que un mecanismo de almacenamiento almacene un objeto, debe almacenar junto con él a todos los dependientes de dicho objeto. Siempre que un mecanismo de recuperación recupere un objeto almacenado con anterioridad, debe también recuperar cualquier dependiente de dicho objeto que aún no haya sido recuperado.

Un esquema o estructura de persistencia es un conjunto reutilizable, y generalmente expansible, de clases que prestan servicios a los objetos persistentes. Suele escribirse un esquema de persistencia para trabajar con bases de datos relacionales. Si se emplea una base de datos orientada a objetos no hace falta un esquema de persistencia. Pero si se utiliza otro mecanismo de almacenamiento se recomienda de servirse de un esquema (framework).

Los esquemas presentan una gran reutilización. En particular, el esquema debería ofrecer las siguientes funciones:

- Guardar y recuperar objetos en un mecanismo de almacenamiento persistente.
- Transacciones del tipo commit y rollback.

El diseño deberá dar soporte a las siguientes cualidades :

- Poder extenderse para soportar cualquier mecanismo de almacenamiento, como base de datos relacionales, archivos planos, etc..
- Requerir un mínimo de modificación del código actual.
- Facilidad de uso
- Ser muy transparente : existe en el trasfondo sin forzarla.

Conclusión : es una "buena práctica" realizar pruebas de performance, analizar el costo de implementación, utilización de datos persistidos por otros sistemas, existencia de frameworks que agilicen la implementación, etc. antes de seleccionar una base de datos orientada a objetos o una relacional.

Código

Escribir código que se compile y se ejecute es sólo una parte del desarrollo de software. También podemos necesitar depurar aplicaciones, ampliarlas, añadir cambios, y compartir partes de código. En otras palabras, nuestro código debe escribirse de forma legible para que sea fácil de entender y consistente. Analizaremos las convenciones en la escritura del código.

Las convenciones de código son importantes para los programadores por un gran número de
razones:

- El 80% del coste del código de un programa va a su mantenimiento.
- Casi ningún software lo mantiene toda su vida el autor original.
- Las convenciones de código mejoran la lectura del software, permitiendo entender código nuevo mucho más rápidamente y más a fondo.
- Si distribuyes tu código fuente como un producto, necesitas asegurarte de que esta bien hecho y presentado como cualquier otro producto.
- Para que funcionen las convenciones, cada persona que escribe software debe seguir la convención.

Conclusión : es una "buena práctica" utilizar y respetar las convenciones en la escritura del código.

Buenas Prácticas – Segunda Parte

Convenciones de código para el lenguaje de programación Java

1 – Introducción

1.1 Por qué convenciones de código

Las convenciones de código son importantes para los programadores. Vamos a presentar en este documento las convenciones recomendadas para escribir código Java.

2 - Nombres de ficheros

Esta sección lista las extensiones más comunes usadas y los nombres de ficheros.

2.1 Extensiones de los ficheros

El software Java usa las siguientes extensiones para los ficheros:

- Fuente Java .java
- Bytecode de Java .class

2.2 Nombres de ficheros comúnes

Los nombres de ficheros más utilizados incluyen:
GNUmakefile El nombre preferido para ficheros "make". Usamos gnumake para construir nuestro software.
README El nombre preferido para el fichero que resume los contenidos de un directorio particular.

3 - Organización de los ficheros

Un fichero consiste de secciones que deben estar separadas por líneas en blanco y comentarios opcionales que identifican cada sección.
Los ficheros de más de 2000 líneas son incómodos y deben ser evitados.

3.1 Ficheros fuente Java

Cada fichero fuente Java contiene una única clase o interface pública. Cuando algunas clases o interfaces privadas están asociadas a una clase pública, pueden ponerse en el mismo fichero que la clase pública. La clase o interfaz pública debe ser la primera clase o interface del fichero.
Los ficheros fuentes Java tienen la siguiente ordenación:

- Comentarios de comienzo
- Sentencias package e import
- Declaraciones de clases e interfaces

3.1.1 Comentarios de comienzo

Todos los ficheros fuente deben comenzar con un comentario en el que se lista el nombre de la clase, información de la versión, fecha, y copyright:

```
/*
* Nombre de la clase
*
* Informacion de la version
*
* Fecha
*
* Copyright
*/
```

3.1.2 Sentencias package e import

La primera línea no-comentario de los ficheros fuente Java es la sentencia package. Despuésvde esta, pueden seguir varias sentencias import. Por ejemplo:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```

Nota: El primer componente de el nombre de un paquete único se escribe siempre en minúsculas con caracteres ASCII y debe ser uno de los nombres de dominio de último nivel, actualmente com, edu, gov, mil, net, org, o uno de los códigos ingleses de dos letras que especifican el país como se define en el ISO Standard 3166, 1981.

3.1.3 Declaraciones de clases e interfaces

La siguiente tabla describe las partes de la declaración de una clase o interface, en el orden en que deberían aparecer.

	Partes de la declaración de una clase o interface	Notas
1	Comentario de documentación de la clase o interface (/**...*/)	Ver "Comentarios de documentación"
2	Sentencia class o interface	
3	Comentario de implementación de la clase o interface si fuera necesario (/*...*/)	Este comentario debe contener cualquier información aplicable a toda la clase o interface que no era apropiada para estar en los comentarios de documentación de la clase o interface.
4	Variables de clase (static)	Primero las variables de clase public, después las protected, después las de nivel de paquete (sin modificador de acceso) , y despúes las private.
5	Variables de instancia	Primero las public, después las protected, después las de nivel de paquete (sin modificador de acceso), y después las private.
6	Constructores	
7	Métodos	Estos métodos se deben agrupar por funcionalidad más que por visión o accesibilidad. Por ejemplo, un método de clase privado puede estar entre dos métodos públicos de instancia. El objetivo es hacer el código mas legible y comprensible.

4 - Indentación

Se deben emplear cuatro espacios como unidad de indentación. La construcción exacta de la indentación (espacios en blanco contra tabuladores) no se especifica. Los tabuladores deben ser exactamente cada 8 espacios (no 4).

4.1 Longitud de la línea

Evitar las líneas de más de 80 caracteres, ya que no son manejadas bien por muchas terminales y herramientas.
Nota: Ejemplos para uso en la documentación deben tener una longitud inferior, generalmente no más de 70 caracteres.

4.2 Rompiendo líneas

Cuando una expresión no entre en una línea, romperla de acuerdo con estos principios:

- Romper después de una coma.
- Romper antes de un operador.
- Preferir roturas de alto nivel (más a la derecha que el "padre") que de bajo nivel (más a la izquierda que el "padre").
- Alinear la nueva línea con el comienzo de la expresión al mismo nivel de la línea anterior.
- Si las reglas anteriores llevan a código confuso o a código que se aglutina en el margen derecho, indentar justo 8 espacios en su lugar.

Ejemplos de como romper la llamada a un método:

```
unMetodo(expresionLarga1, expresionLarga2, expresionLarga3,
         expresionLarga4, expresionLarga5);

var = unMetodo1(expresionLarga1,
                unMetodo2(expresionLarga2,
                          expresionLarga3));
```

Ahora dos ejemplos de ruptura de líneas en expresiones aritméticas. Se prefiere el primero, ya que el salto de línea ocurre fuera de la expresión que encierra los paréntesis.

```
nombreLargo1 = nombreLargo2 * (nombreLargo3 + nombreLargo4
               - nombreLargo5) + 4 * nombreLargo6; // PREFERIDA

nombreLargo1 = nombreLargo2 * (nombreLargo3 + nombreLargo4
                               - nombreLargo) + 4 * nombreLargo6;
          // EVITAR
```

Ahora dos ejemplos de indentación de declaraciones de métodos. El primero es el caso convencional. El segundo conduciría la segunda y la tercera línea demasiado hacia la izquierda con la indentación convencional, así que en su lugar se usan 8 espacios de indentación.

```
//INDENTACION CONVENCIONAL

unMetodo(int anArg, Object anotherArg, String yetAnotherArg,
         Object andStillAnother) {
    ...
}


//INDENTACION DE 8 ESPACIOS PARA EVITAR GRANDES INDENTACIONES

private static synchronized metodoDeNombreMuyLargo(int unArg,
        Object otroArg, String todaviaOtroArg,
        Object unOtroMas) {
    ...
}
```

Saltar de líneas por sentencias if deberá seguir generalmente la regla de los 8 espacios, ya que la indentacion convencional (4 espacios) hace difícil ver el cuerpo. Por ejemplo:

```
//NO USAR ESTA INDENTACION

if ((condicion1 && condicion2)
    || (condicion3 && condicion4)
    ||!(condicion5 && condicion6)) { //MALOS SALTOS
    hacerAlgo(); //HACEN ESTA LINEA FACIL DE OLVIDAR
}

// USAR ESTA INDENTACION

if ((condicion1 && condicion2)
        || (condicion3 && condicion4)
        ||!(condicion5 && condicion6)) {
    hacerAlgo();
}

//O USAR ESTA

if ((condicion1 && condicion2) || (condicion3 && condicion4)
        ||!(condicion5 && condicion6)) {
```

```
    hacerAlgo();
}
```

Hay tres formas aceptables de formatear expresiones ternarias:

```
alpha = (unaLargaExpresionBooleana) ? beta : gamma;

alpha = (unaLargaExpresionBooleana) ? beta
                                    : gamma;

alpha = (unaLargaExpresionBooleana)
        ? beta
        : gamma;
```

5 - Comentarios

Los programas Java pueden tener dos tipos de comentarios: comentarios de implementación y comentarios de documentación. Los comentarios de implementación son aquellos que también se encuentran en C++, delimitados por /*...*/, y //. Los comentarios de documentación (conocidos como "doc comments") existen sólo en Java, y se limitan por /**...*/. Los comentarios de documentación se pueden exportar a ficheros HTML con la herramienta javadoc.
Los comentarios de implementación son para comentar nuestro código o para comentarios acerca de una implementación particular. Los comentarios de documentación son para describir la especificación del código, libre de una perspectiva de implementación, y para ser leídos por desarrolladores que pueden no tener el código fuente a mano.
Se deben usar los comentarios para dar descripciones de código y facilitar información adicional que no es legible en el código mismo. Los comentarios deben contener sólo información que es relevante para la lectura y entendimiento del programa. Por ejemplo, información sobre como se construye el paquete correspondiente o en que directorio reside no
debe ser incluida como comentario.
Son apropiadas las discusiones sobre decisiones de diseño no triviales o no obvias, pero evitar duplicar información que esta presente (de forma clara) en el código ya que es fácil que los comentarios redundantes se queden desfasados. En general, evitar cualquier comentario que pueda quedar desfasado a medida que el código evoluciona.
Nota: La frecuencia de comentarios a veces refleja una pobre calidad del código. Cuando se sienta obligado a escribir un comentario considere re escribir el código para hacerlo más claro.
Los comentarios no deben encerrarse en grandes cuadrados dibujados con asteriscos u otros caracteres.
Los comentarios nunca deben incluir caracteres especiales como backspace.

5.1 Formatos de los comentarios de implementación

Los programas pueden tener cuatro estilos de comentarios de implementación: de bloque, de una línea, de remolque, y de fin de línea

5.1.1 Comentarios de bloque

Los comentarios de bloque se usan para dar descripciones de ficheros, métodos, estructuras de datos y algoritmos. Los comentarios de bloque se podrán usar al comienzo de cada ficehro o antes de cada método. También se pueden usar en otro lugares, tales como el interior de los métodos. Los comentarios de bloque en el interior de una función o método deben ser indentados al mismo nivel que el código que describen.
Un cometario de bloque debe ir precedido por una línea en blanco que lo separe del resto del código.

```
/*
* Aqui hay un comentario de bloque.
*/
```

Los comentarios de bloque pueden comenzar con /*-, que es reconocido por **indent**(1) como el comienzo de un comentario de bloque que no debe ser reformateado. Ejemplo:

```
/*-
* Aqui tenemos un comentario de bloque con cierto
* formato especial que quiero que ignore indent(1).
*
* uno
* dos
```

```
* tres
*/
```

Nota: Si no se usa **indent**(1), no se tiene que usar /*- en el código o hacer cualquier otra concesión a la posibilidad de que alguien ejecute **indent**(1) sobre él.
Ver también "Comentarios de documentación"

5.1.2 Comentarios de una línea

Pueden aparecer comentarios cortos de una única línea al nivel del código que siguen. Si un comentario no se puede escribir en una línea, debe seguir el formato de los comentarios de bloque. (ver sección 5.1.1). Un comentario de una sóla línea debe ir precedido de una línea en blanco. Aquí un ejemplo de comentario de una sola línea en código Java (ver también "Comentarios de documentación"):

```
if (condicion) {
    /* Código de la condicion. */
    ...
}
```

5.1.3 Comentarios de remolque

Pueden aparecer comentarios muy pequeños en la misma línea que describen, pero deben ser movidos lo suficientemente lejos para separarlos de las sentencias. Si más de un comentario corto aparecen en el mismo trozo de código, deben ser indentados con la misma profundidad.
Aquí un ejemplo de comentario de remolque:

```
if (a == 2) {
    return TRUE;       /* caso especial */
} else {
    return isPrime(a); /* caso gerenal */
}
```

5.1.4 Comentarios de fin de línea

El delimitador de comentario // puede convertir en comentario una línea completa o una parte de una línea. No debe ser usado para hacer comentarios de varias líneas consecutivas; sin embargo, puede usarse en líneas consecutivas para comentar secciones de código. Ejemplos de los tres estilos:

```
if (foo > 1) {
    // Hacer algo.
    ...
}
else {
    return false; // Explicar aqui por que.
}

//if (bar > 1) {
//
//   // Hacer algo.
//   ...
//}
//else {
//   return false;
//}
```

5.2 Comentarios de documentación

Nota: Ver "Ejemplo de fichero fuente Java" para ejemplos de los formatos de comentarios descritos aquí.
Para más detalles, ver "How to Write Doc Comments for Javadoc" que incluye información de las etiquetas de los comentarios de documentación (@return, @param, @see):
http://java.sun.com/products/jdk/javadoc/writingdoccomments.shtml
Para más detalles acerca de los comentarios de documentación y javadoc, visitar el sitio web de javadoc:
http://java.sun.com/products/jdk/javadoc/

Los comentarios de documentación describen clases Java, interfaces, constructores, métodos y atributos. Cada comentario de documentación se encierra con los delimitadores de comentarios /**...*/, con un comentario por clase, interface o miembro (método o atributo). Este comentario debe aparecer justo antes de la declaración:

```
/**
* La clase Ejemplo ofrece ...
*/
public class Ejemplo { ...
```

Darse cuenta de que las clases e interfaces de alto nivel son están indentadas, mientras que sus miembros los están. La primera línea de un comentario de documentación (/**) para clases e interfaces no esta indentada, subsecuentes líneas tienen cada una un espacio de indentación (para alinear verticalmente los asteriscos). Los miembros, incluidos los constructores, tienen cuatro espacios para la primera línea y 5 para las siguientes.
Si se necesita dar información sobre una clase, interface, variable o método que no es apropiada para la documentación, usar un comentario de implementación de bloque (ver sección 5.1.1) o de una línea (ver sección 5.1.2) para comentarlo inmediatamente *después* de la declaración. Por ejemplo, detalles de implementación de una clase deben ir en un comentario de implementación de bloque *siguiendo* a la sentencia class, no en el comentario de documentación de la clase.
Los comentarios de documentación no deben colocarse en el interior de la definición de un método o constructor, ya que Java asocia los comentarios de documentación con la *primera declaración después* del comentario.

6 - Declaraciones

6.1 Cantidad por línea

Se recomienda una declaración por línea, ya que facilita los comentarios. En otras palabras, se prefiere

```
int nivel; // nivel de indentación
int tam;   // tamaño de la tabla
```

antes que

```
int level, size;
```

No poner diferentes tipos en la misma línea. Ejemplo:

```
int foo, fooarray[]; //ERROR!
```

Nota: Los ejemplos anteriores usan un espacio entre el tipo y el identificador. Una alternativa aceptable es usar tabuladores, por ejemplo:

```
int    level;         // nivel de indentacion
int    size;         // tamaño de la tabla
Object currentEntry; // entrada de la tabla seleccionada actualmente
```

6.2 Inicialización

Intentar inicializar las variables locales donde se declaran. La única razón para no inicializar una variable donde se declara es si el valor inicial depende de algunos cálculos que deben ocurrir.

6.3 Colocación

Poner las declaraciones solo al principio de los bloques (un bloque es cualquier código encerrado por llaves "{" y "}".) No esperar al primer uso para declararlas; puede confundir a programadores no preavisados y limitar la portabilidad del código dentro de su ámbito de visibilidad.

```
void myMethod() {
    int int1 = 0;      // comienzo del bloque del método
    if (condition) {
        int int2 = 0; // comienzo del bloque del "if"
        ...
    }
```

```
}
```

La excepción de la regla son los índices de bucles for, que en Java se pueden declarar en la sentencia for:

```
for (int i = 0; i < maximoVueltas; i++) { ... }
```

Evitar las declaraciones locales que ocultan declaraciones de niveles superiores. por ejemplo, no declarar la misma variable en un bloque interno:

```
int cuenta;
...
miMetodo() {
    if (condicion) {
        int cuenta = 0; // EVITAR!
        ...
    }
    ...
}
```

6.4 Declaraciones de class e interfaces

Al codificar clases e interfaces de Java, se siguen las siguientes reglas de formato:

- Ningún espacio en blanco entre el nombre de un método y el paréntesis "(" que abre su lista de parámetros
- La llave de apertura "{" aparece al final de la misma línea de la sentencia declaración
- La llave de cierre "}" empieza una nueva línea indentada para ajustarse a su sentencia de apertura correspondiente, excepto cuando no existen sentencias entre ambas, que debe aparecer inmediatamente después de la de apertura "{"

```
class Ejemplo extends Object {
    int ivar1;
    int ivar2;

    Ejemplo(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int metodoVacio() {}
    ...
}
```

- Los métodos se separan con una línea en blanco

7 - Sentencias

7.1 Sentencias simples

Cada línea debe contener como mucho una sentencia. Ejemplo:

```
argv++;           // Correcto
argc--;          // Correcto
argv++; argc--; // EVITAR!
```

7.2 Sentencias compuestas

Las sentencias compuestas son sentencias que contienen listas de sentencias encerradas entre llaves "{ sentencias }". Ver la siguientes secciones para ejemplos.

- Las sentencias encerradas deben indentarse un nivel más que la sentencia compuesta.
- La llave de apertura se debe poner al final de la línea que comienza la sentencia compuesta; la llave de cierre debe empezar una nueva línea y ser indentada al mismo nivel que el principio de la sentencia compuesta.

- Las llaves se usan en todas las sentencias, incluso las simples, cuando forman parte de una estructura de control, como en las sentencias if-else o for. Esto hace más sencillo añadir sentencias sin incluir bugs accidentalmente por olvidar las llaves.

7.3 Sentencias de retorno

Una sentencia return con un valor no debe usar paréntesis a menos que hagan el valor de retorno más obvio de alguna manera. Ejemplo:

```
return;
return miDiscoDuro.size();
return (tamanyo ? tamanyo : tamanyoPorDefecto);
```

7.4 Sentencias if, if-else, if else-if else

La clase de sentencias if-else debe tener la siguiente forma:

```
if (condicion) {
    sentencias;
}

if (condicion) {
    sentencias;
} else {
    sentencias;
}

if (condicion) {
    sentencia;
} else if (condicion) {
    sentencia;
} else{
    sentencia;
}
```

Nota: Las sentencias if usan siempre llaves {}. Evitar la siguiente forma, propensa a errores:

```
if (condicion) //EVITAR! ESTO OMITE LAS LLAVES {}!
    sentencia;
```

7.5 Sentencias for

Una sentencia for debe tener la siguiente forma:

```
for (inicializacion; condicion; actualizacion) {
    sentencias;
}
```

Una sentencia for vacía (una en la que todo el trabajo se hace en las clausulas de inicialización, condicion, y actualizacion) debe tener la siguiente forma:

```
for (inicializacion; condicion; actualizacion);
```

Al usar el operador coma en la cláusula de inicialización o actualización de una sentencia for, evitar la complejidad de usar más de tres variables. Si se necesita, usar sentencias separadas antes de bucle for (para la cláusula de inicialización) o al final del bucle (para la cláusula de actualización).

7.6 Sentencias while

Una sentencia while debe tener la siguiente forma:

```
while (condicion) {
    sentencias;
}
```

Una sentencia while vacía debe tener la siguiente forma:

```
while (condicion);
```

7.7 Sentencias do-while

Una sentencia do-while debe tener la siguiente forma:

```
do {
    sentencias;
} while (condicion);
```

7.8 Sentencias switch

Una sentencia switch debe tener la siguiente forma:

```
switch (condicion) {
case ABC:
    sentencias;
    /* este caso se propaga */
case DEF:
    sentencias;
    break;
case XYZ:
    sentencias;
    break;
default:
    sentencias;
    break;
}
```

Cada vez que un caso se propaga (no incluye la sentencia break), añadir un comentario donde la sentencia break se encontraría normalmente. Esto se muestra en el ejemplo anterior con el comentario /* este caso se propaga */.
Cada sentencia switch debe incluir un caso por defecto. El break en el caso por defecto es redundante, pero prevee que se propague por error si luego se añade otro caso.

7.9 Sentencias try-catch

Una sentencia try-catch debe tener la siguiente forma:

```
try {
    sentencias;
} catch (ExceptionClass e) {
    sentencias;
}
```

Una sentencia try-catch puede ir seguida de un finally, cuya ejecución se ejecutará independientemente de que el bloque try se halla completado con éxito o no.

```
try {
    sentencias;
} catch (ExceptionClass e) {
    sentencias;
} finally {
    sentencias;
}
```

8 - Espacios en blanco

8.1 Líneas en blanco

Las líneas en blanco mejoran la facilidad de lectura separando secciones de código que están lógicamente relacionadas.
Se deben usar siempre dos líneas en blanco en las siguientes circunstancias:

- Entre las secciones de un fichero fuente
- Entre las definiciones de clases e interfaces.

Se debe usar siempre una línea en blanco en las siguientes circunstancias:

- Entre métodos
- Entre las variables locales de un método y su primera sentencia
- Antes de un comentario de bloque (ver sección 5.1.1) o de un comentario de una línea (ver sección 5.1.2)
- Entre las distintas secciones lógicas de un método para facilitar la lectura.

8.2 Espacios en blanco

Se deben usar espacios en blanco en las siguientes circunstancias:

- Una palabra clave del lenguaje seguida por un paréntesis debe separarse por un espacio. Ejemplo:

```
while (true) {
...
}
```

  Notar que no se debe usar un espacio en blanco entre el nombre de un método y su paréntesis de apertura. Esto ayuda a distinguir palabras claves de llamadas a métodos.

- Debe aparecer un espacio en blanco después de cada coma en las listas de argumentos.

- Todos los operadores binarios excepto . se deben separar de sus operandos con espacios en blanco. Los espacios en blanco no deben separadar los operadores unarios, incremento ("++") y decremento ("--") de sus operandos. Ejemplo:

```
a += c + d;
a = (a + b) / (c * d);
while (d++ == s++) {
    n++;
}
printSize("el tamaño es " + foo + "\n");
```

- Las expresiones en una sentencia for se deben separar con espacios en blanco. Ejemplo:

```
for (expr1; expr2; expr3)
```

- Los "Cast"s deben ir seguidos de un espacio en blanco. Ejemplos:

```
miMetodo((byte) unNumero, (Object) x);
miMetodo((int) (cp + 5), ((int) (i + 3)) + 1);
```

9 - Convenciones de nombres

Las convenciones de nombres hacen los programas más entendibles haciéndolos más fácil de leer. También pueden dar información sobre la función de un identificador, por ejemplo, cuando es una constante, un paquete, o una clase, que puede ser útil para entender el código.

Tipos de Identificadores	Reglas para nombres	Ejemplos

Paquetes	El prefijo del nombre de un paquete se escribe siempre con letras ASCII en minúsculas, y debe ser uno de los nombres de dominio de alto nivel, actualmente com, edu, gov, mil, net, org, o uno de los códigos ingleses de dos letras que identifícan cada país como seespecifica en el ISO Standard 3166, 1981. Los subsecuentes componentes del nombre del paquete variarán de acuerdo a las convenciones de nombres Internas de cada organización. Dichas convenciones pueden especificar que algunos nombres de los directorios correspondan a divisiones, departamentos, proyectos o máquinas.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese
Clases	Los nombres de las clases deben ser sustantivos, cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas. Intentar mantener los nombres de las clases simples y descriptivos. Usar palabras completas, evitar acrónimos y abreviaturas (a no ser que la abreviatura sea mucho más conocida que el nombre completo, como URL or HTML).	class Cliente; class ImagenAnimada;
Interfaces	Los nombres de las interfaces siguen la misma regla que las clases.	interface ObjetoPersistente; interface Almacen;
Métodos	Los métodos deben ser verbos, cuando son compuestos tendrán la primera letra en minúscula, y la primera letra de las siguientes palabras que lo forma en mayúscula.	ejecutar(); ejecutarRapido();
Variables	Excepto las constantes, todas las instancias y variables de clase o método empezarán con minúscula. Las palabras internas que lo forman (si son compuestas) empiezan con su primera letra en mayúsculas. Los nombres de variables no deben empezar con los caracteres subguión "_" o signo del dólar "$", aunque ambos estan permitidos por el lenguaje. Los nombres de las variables deben ser cortos pero con significado. La elección del nombre de una variable debe ser un mnemónico, designado para indicar a un observador casual su función. Los nombres de variables de un solo carácter se deben evitar, excepto para variables índices temporales. Nombres comunes para variables temporales son i, j, k, m, y n para enteros; c, d, y e para caracteres.	int i; char c; float miAnchura;

Constantes	Los nombres de las variables declaradas como constantes deben ir totalmente en mayúsculas separando las palabras con un subguión ("_"). (Las constantes ANSI se deben evitar, para facilitar su depuración.)	static final int ANCHURA_MINIMA = 4;
		
		static final int ANCHURA_MAXIMA =999;

10 - Hábitos de programación

10.1 Proporcionando acceso a variables de instancia y de clase

No hacer ninguna variable de instancia o clase pública sin una buena razón. A menudo las variables de instancia no necesitan ser asignadas / consultadas explícitamente, a menudo esto sucede como efecto lateral de llamadas a métodos.

Un ejemplo apropiado de una variable de instancia pública es el caso en que la clase es esencialmente una estructura de datos, sin comportamiento. En otras palabras, si usarías la palabra struct en lugar de una clase (si Java soportara struct), entonces es adecuado hacer las variables de instancia públicas.

10.2 Referencias a variables y métodos de clase

Evitar usar un objeto para acceder a una variable o método de clase (static). Usar el nombre de la clase en su lugar. Por ejemplo:

```
metodoDeClase(); //OK
UnaClase.metodoDeClase(); //OK
unObjeto.metodoDeClase(); //EVITAR!
```

10.3 Constantes

Las constantes numéricas (literales) no se deben codificar directamente, excepto -1, 0, y 1, que pueden aparecer en un bucle for como contadores.

10.4 Asignaciones de variables

Evitar asignar el mismo valor a varias varibles en la misma sentencia. Es difícil de leer. Ejemplo:

```
fooBar.fChar = barFoo.lchar = 'c'; // EVITAR!
```

No usar el operador de asignación en un lugar donde se pueda confundir con el de igualdad. Ejemplo:

```
if (c++ = d++) { // EVITAR! (Java lo rechaza)
...
}
```

se debe escribir:

```
if ((c++ = d++) != 0) {
...
}
```

No usar asignación embebidas como un intento de mejorar el rendimient en tiempo de ejecución. Ese es el trabajo del compilador. Ejemplo:

```
d = (a = b + c) + r; // EVITAR!
```

se debe escribir:

```
a = b + c;
d = a + r;
```

10.5 Hábitos varios

10.5.1 Paréntesis

En general es una buena idea usar paréntesis en expresiones que implican distintos operadores para evitar problemas con el orden de precedencia de los operadores. Incluso si parece claro el orden de precedencia de los operadores, podría no ser así para otros, no se debe asumir que otros programadores conozcan el orden de precedencia.

```
if (a == b && c == d) // EVITAR!
if ((a == b) && (c == d)) // CORRECTO
```

10.5.2 Valores de retorno

Intentar hacer que la estructura del programa se ajuste a su intención. Ejemplo:

```
if (expresionBooleana) {
    return true;
} else {
    return false;
}
```

en su lugar se debe escribir

```
return expressionBooleana;
```

Similarmente,

```
if (condicion) {
    return x;
}
    return y;
```

se debe escribir:

```
return (condicion ? x : y);
```

10.5.3 Expresiones antes de `?' en el operador condicional

Si una expresión contiene un operador binario antes de ? en el operador ternario ?: , se debe colocar entre paréntesis. Ejemplo:

```
(x >= 0) ? x : -x;
```

10.5.4 Comentarios especiales

Usar XXX en un comentario para indicar que algo tiene algún error pero funciona. Usar FIXME para indicar que algo tiene algún error y no funciona.