GUIA DE ESTÁNDARES DE DESARROLLO

Guía de estándares de desarrollo
Secretaría Distrital de Salud
Bogotá, Enero 2015
V 1.3.8

REVISION Y CONTROL DE CAMBIOS

Revisión y Versiones

Nombres y Apellidos	Versión Aprobada	Cargo	Fecha

Control de Cambios

Fecha	Autor	Versión	Descripción del cambio
12/01/2015	Eduardo Hernández Gómez.	1.3.8	Verificación de los capítulos de Introducción y Generalidades del estándar.
21/01/2015	Angela Consuelo Romero Parra Eduardo Hernández Gómez.	1.3.8	Estructuración de los capítulos del documento
22/01/2015	Angela Consuelo Romero Parra	1.3.8	Se elaboran los capítulos 2.1 al 2.5
22/01/2015	Eduardo Hernández Gómez.	1.3.8	Se elaboran los capítulos 2.6 al 2.8 y el capítulo 3.0.

Contenido

INTRODUCCION

La Secretaria Distrital de Salud se encuentra adelantando la construcción de soluciones de software con el propósito de facilitar en cada área las labores que se realizan y que constituyen actividades misionales de la entidad.

Debido a esto, la Dirección TIC consiente del creciente aumento en el desarrollo de aplicaciones que actualmente se están llevando a cabo al interior de la Secretaria Distrital de Salud y comprometida con la optimización de cada uno de ellos, ha establecido el estándar para el proceso de desarrollo de software con el objetivo de aplicar las mejores prácticas de la ingeniería de software para garantizar la calidad en cada uno de los aplicativos y sistemas de información desarrollados.

Cada una de las políticas y lineamientos mencionados en éste documento deben ser implementados en su totalidad considerando los aspectos que se mencionan en este documento.

1. GENERALIDADES DEL ESTANDAR

1.2 Esencia de los estándares

Los estándares son lineamientos, directrices y protocolos que se establecen con el propósito de normalizar la escritura del código que conforma un desarrollo de software, procurando la consistencia y reusabilidad del mismo. Emplear un lenguaje claro al definir cada uno de los elementos de software a través de convenciones permite mantener el código de manera eficaz y eficiente, así mismo agiliza el proceso de detección de errores toda vez que los estándares sean claros y se apliquen cabalmente.

Al considerar la selección del estándar de desarrollo se deben tener en cuenta aspectos técnicos como son la definición de variables por medio de una nomenclatura adecuada, la documentación requerida, las herramientas de verificación entre otros conceptos que serán objeto de estudio en este documento. Además de la selección del estándar se requiere el compromiso por parte de los desarrolladores de aplicar los lineamientos mencionados y cumplir con todas las especificaciones que sean solicitadas por la Dirección de Planeación y Sistemas.

Es necesario mencionar que cada desarrollo de software que se ha llevado a cabo sin aplicar ningún estándar en su construcción, requiere una dedicación especial de recurso humano y de tiempo debido a que se cuenta solamente con los conceptos aplicados arbitrariamente por el desarrollador a cargo; este comportamiento constituye una mala práctica en cuanto a construcción de software debido a que se limita la escalabilidad del sistema y su reusabilidad que son conceptos ampliamente utilizados al considerar aspectos como calidad y desempeño del software.

1.3. Selección del sistema de notación

Uno de los aspectos más relevantes en la definición del estándar de programación es la selección del sistema de notación, debido a que este lineamiento rige directamente la escritura del código fuente del aplicativo. A través de sistema de notación se define la nomenclatura y las convenciones para cada uno de los elementos de software como son los tipos de datos, funciones, formularios y demás componentes que conforman un aplicativo.

Notaciones como Reddick, Pascal Casing y Camel Casing son las mas usadas actualmente al construir soluciones de software; cada una de ellas tiene aspectos particulares pero a través del análisis y estudio realizado a cada una de ellas se logró concluir que la mas óptima para ser implementada es la Reddick.

La notación Reddick presenta características sobresalientes como son: altamente descriptiva, eficiente, fácil de mantener y muy consistente que se adaptan a las necesidades de la entidad, considerando que muchos de los desarrollos de software pueden ser empleados de manera transversal para varias dependencias.

La notación Reddick proporciona claridad a través de prefijos del tipo de dato y del contexto del elemento a definir, esta característica contribuye a entender el código fuente de una manera más intuitiva sin entrar en el detalle de hacer un seguimiento exhaustivo al código fuente.

1.4. Tipo de dato

Todo lenguaje de programación posee una definición para cada tipo de dato que permite identificar el atributo que soporta y el tamaño de la información que puede contener. Los datos mas comunes empleados por la mayoría de lenguajes de programación se denominan “Primitivos” estos tipos de datos son: Integer, Long, Boolean, String, Double, Object.

Es necesario tener cuidado al hacer uso del tipo de datos para definir un elemento como por ejemplo una variable, debido a que se puede hacer un uso indebido de la memoria cuando no se requiere; Por ejemplo para una variable que almacena un valor máximo de 99 no se requiere un tipo de dato ”Long” en su lugar es correcto definirla como “Integer”.

Asignar erróneamente un tipo de dato constituye una mala práctica y debe ser corregida para optimizar los bloques de código que se ejecuten. Esta directriz hace parte de la optimización del código que busca mejorar la aplicación del estándar.

2. IMPLEMENTACION DEL ESTANDAR

2.1. Organización de archivos

2.1.1. Archivo fuente

Cada archivo fuente debe contener una única clase o interfaz pública. El nombre del archivo tiene que coincidir con el nombre de la clase. Cuando existan varias clases privadas asociadas funcionalmente a una clase pública, podrán colocarse en el mismo archivo fuente que la clase pública. La clase pública debe estar situada en primer lugar dentro del archivo fuente.

En todo archivo fuente distinguimos las siguientes secciones:

- Comentarios de inicio.

- Sentencia de paquete.
- Sentencias de importación.
- Declaraciones de clases e interfaces.

- Comentarios de inicio

Todo archivo fuente debe comenzar con un comentario que incluya el nombre de la clase, información sobre la versión del código, la fecha y el copyright. El copyright indica la propiedad legal del código, el ámbito de distribución, el uso para el que fue desarrollado y su modificación, se debe especificar que la propiedad del código es de la Secretaría Distrital de Salud.

Dentro de estos comentarios iniciales podrían incluirse adicionalmente comentarios sobre los cambios efectuados sobre dicho archivo (mejora, incidencia, error, etc.). Estos comentarios son opcionales si los archivos están bajo un sistema de control de versiones bien documentado del cual se deberá presentar las evidencias necesarias, en caso contrario se recomienda su uso. Estos comentarios constituyen el historial de cambios del archivo. Este historial es único para cada archivo y permitirá conocer rápidamente el estado y la evolución que ha tenido el archivo desde su origen.

A continuación se muestra un EJEMPLO de comentario de inicio para la clase "JceSecurity.java".

```
/*
 * @(#)JceSecurity.java 1.50 04/14/2015
 *
 * Copyright 2015 SDS, Inc. Todos los derechos reservados.
 * SUN PROPRIETARY/CONFIDENTIAL.
 */

/**
 *  Esta clase crea una instancia de las clases de motores de…
 * Registrado en la clase java.security.Security object.

 * @autor Juan Perez
* @version 1.50, 04/14/2015
 * @desde 1.4
 */
```

Para .NET se muestra un ejemplo de comentario de inicio para la función “InsertarUbicacionFisicaEspediente”

```
''' <summary>
   ''' Insertar los datos de la ubicación física para un expediente dado.
```

```
''' </summary>
''' <returns></returns>
''' <remarks>
''' '11/01/2011 Modificado Juan Perez
''' </remarks>
```

- Sentencias de paquete

La primera línea no comentada de un archivo fuente debe ser la sentencia de paquete, que indica el paquete (`Namespace`) al que pertenece(n) la(s) clase(s) incluida (s) en el archivo fuente. Por ejemplo para JAVA

Para JAVA

```
package javax.crypto;
```

Para .NET

```
namespace System.Numerics
```

- Sentencias de importación

Tras la declaración del paquete se incluirán las sentencias de importación de los paquetes necesarios. Esta importación de paquetes obligatorios seguirá el siguiente orden:

- Paquetes del JDK de java o MS Framework .Net.
- Paquetes de utilidades no pertenecientes al JDK de Java o al MS Framework .Net, de frameworks de desarrollo o de proyectos opensource tales como apache, hibernate, springframework o terceros previa autorización por la Dirección TIC de la SDS.
- Paquetes de la aplicación.

Se recomienda minimizar en la medida de lo posible el uso de importaciones del tipo "package.*", pues dificultan la comprensión de las dependencias existentes entre las clases utilizadas por la aplicación. En caso contrario, se recomienda utilizar comentarios de línea tras la importación.

```
import java.io.*; // BufferedReader, PrintWriter, FileInputStream, File
import java.util.ArrayList;

import org.apache.log4j.Logger;
import org.apache.lucene.analysis.Analyzer;
import es.provincia.organismo.corporativas.atlas.vo.AgendaVO;
import es.provincia.organismo.atlas.vo.AnuncioVO;
import es.provincia.organismo.atlas.vo.OrganigramaVO;
```

2.1.2. Declaraciones de clases e interfaces

La siguiente tabla describe los elementos que componen la declaración de una clase o interfaz, así como el orden en el que deben estar situados.

Elementos de declaración de una clase / interfaz	Descripción
Comentario de documentación de la clase/interfaz / ... */**	Permite describir la clase/interfaz desarrollada. Necesario para generar la documentación de la api mediante javadoc.
Comentario de implementación de la clase/interfaz, si es necesario /* ... */	Este comentario incluye cualquier información que no pueda incluirse en el comentario de documentación de la clase/interfaz.
Variables de clase (estáticas)	En primer lugar las variables de clase públicas (public), después las protegidas (protected), posteriormente las de nivel de paquete (sin modificador), y por último las privadas (private).
Variables de instancia	Primero las públicas (public), después las protegidas (protected), luego las de nivel de paquete (sin modificador), y finalmente las privadas (private).
Métodos	Deben agruparse por funcionalidad en lugar de agruparse por ámbito o accesibilidad. Por ejemplo, un método privado puede estar situado entre dos métodos públicos. El objetivo es desarrollar código fácil de leer y comprender.

2.2. Sangría

Como norma general se establecen 4 caracteres como unidad de sangría. Los entornos de desarrollo integrado (IDE) más populares, tales como Eclipse, NetBeans o Visual Studio .Net, incluyen facilidades para formatear código Java.

Como norma general se establecen 4 caracteres como unidad de sangría. Los entornos de desarrollo integrado (IDE) más populares, tales como Eclipse, NetBeans o Visual Studio .Net, incluyen facilidades para formatear código Java o .Net (C#).

2.2.1. Longitud de línea

La longitud de línea no debe superar los 80 caracteres por motivos de visualización e impresión

2.2.2. División de líneas

Cuando una expresión ocupe más de una línea, esta se podrá romper o dividir en función de los siguientes criterios,

- Tras una coma.
- Antes de un operador.
- Se recomienda las rupturas de nivel superior a las de nivel inferior.
- Alinear la nueva línea con el inicio de la expresión al mismo nivel que la línea anterior.
- Si las reglas anteriores generan código poco comprensible, entonces estableceremos tabulaciones de 8 espacios.

Ejemplos:

```
unMetodo(expresionLarga1, expresionLarga 2, expresionLarga 3,
    expresionLarga 4, expresionLarga 5);


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

Igual aplica para desarrollo en .NET

2.3. Comentarios

Distinguimos dos tipos de comentarios: los comentarios de implementación y los de documentación.

2.3.1. Comentarios de implementación

Estos comentarios se utilizan para describir el código ("el cómo"), y en ellos se incluye información relacionada con la implementación, tales como descripción de la función de variables locales, fases lógicas de ejecución de un método, captura de excepciones, etc.

Distinguimos tres tipos de comentarios de implementación:

- Comentarios de bloque:

Permiten la descripción de ficheros, clases, bloques, estructuras de datos y algoritmos.

En JAVA

```
/*
 * Esto es un comentario
 * de bloque
 */
```

Para .NET

```
''' <summary>
''' Selecciona todos los datos de un expediente
''' con base en su codigo de base de datos.
''' </summary>
''' <param name="IdExpediente"></param>
''' <returns></returns>
''' <remarks>
''' Modificado 03/01/2012 Juan Perez
''' Modificado 08/08/2014 Juan Perez
''' </remarks>
```

- Comentarios de línea:

Son comentarios cortos localizados en una sola línea y tabulados al mismo nivel que el código que describen. Si ocupa más de una línea se utilizará un comentario de bloque. Deben estar precedidos por una línea en blanco.

En JAVA

```
/* Mostrar mensaje del expediente si existe la variable de sesión*/
```

En .NET

```
'Mostrar mensaje del expediente si existe la variable de sesión
```

- Comentario a final de línea

Comentario situado al final de una sentencia de código y en la misma línea.

En JAVA

```
int contador = 4 + 10;  // Inicialización del contador
contador++;   /* Incrementamos el contador */
```

En .NET

```
int contador = 4 + 10;  ' Inicialización del contador
contador++;  'Incrementamos el contador
```

2.3.2. Comentarios de documentación

Los comentarios de documentación, se utilizan para describir la especificación del código, desde un punto de vista independiente de la implementación, de forma que pueda ser consultada por desarrolladores que probablemente no tengan acceso al código fuente.

El apartado 2 de este documento describe el uso de comentarios de documentación.

2.4. Declaraciones

2.4.1. Una declaración por línea

Se recomienda el uso de una declaración por línea, promoviendo así el uso de comentarios. Ejemplo,

Para JAVA

```
int idUnidad;   // Identificador de la unidad organizativa
String() funciones; // Funciones de la unidad
```

Para .NET

```
Dim li_peso(15) As Integer
```

2.4.2. Inicialización

Toda variable local tendrá que ser inicializada en el momento de su declaración, salvo que su valor inicial dependa de algún valor que tenga que ser calculado previamente.

Para JAVA

```
int idUnidad = 1;
String[] funciones = { "Administración", "Intervención", "Gestión" };
```

Para .NET

```
Dim digito_chequeo As Integer = -1
```

2.4.3. Localización

Las declaraciones deben situarse al principio de cada bloque principal en el que se utilicen, y nunca en el momento de su uso.

La única excepción a esta regla son los índices de los bucles "for", ya que, en Java, pueden incluirse dentro de la propia sentencia "for".

Para JAVA

```
for (int i=0; contador<10; i++) {
 ...
}
```

Para .NET

```
For i As Integer = 0 To 14
             li_suma += CInt(strNit.Substring(i, 1)) * li_peso(i)
Next
```

Se debe evitar el uso de declaraciones que oculten a otras declaraciones de ámbito superior.

```
int contador = 0;  // Inicio del método
public void unMetodo() {
 if (condicion) {
  int contador = 2;  // ¡¡ EVITAR !!
  ...
 }
 ...
}
```

2.4.4. Declaración de clases / interfaces

Durante el desarrollo de clases / interfaces se deben seguir las siguientes reglas de formateo:

- No incluir ningún espacio entre el nombre del método y el paréntesis inicial del listado de parámetros.
- El carácter inicio de bloque ("{") debe aparecer al final de la línea que contiene la sentencia de declaración.
- El carácter fin de bloque ("}") se sitúa en una nueva línea tabulada al mismo nivel que su correspondiente sentencia de inicio de bloque, excepto cuando la sentencia sea nula, en tal caso se situará detrás de "{".
- Los métodos se separarán entre sí mediante una línea en blanco.

Para JAVA

```
public classe ClaseEjemplo extends Object {

 int variable1;
 int variable2;

 public ClaseEjemplo() {
```

```
 variable1 = 0;
 variable2 = 1;
}
 ...
}
```

Para .NET
Durante el desarrollo de clases / interfaces se deben seguir las siguientes reglas de formateo:

- No incluir ningún espacio entre el nombre del método y el paréntesis inicial del listado de parámetros.
- La frase de cierre (End Function) debe quedar al mismo nivel de la frase de inicio (Public Function) por ejemplo.

```
Public Function digitoVerificacion(ByVal strNit As String) As Integer

    ....
        .....
End Function
```

2.5. Sentencias

Las sentencias pertenecientes a un bloque de código estarán tabuladas un nivel más a la derecha con respecto a la sentencia que las contiene.

El carácter inicio de bloque "{" debe situarse al final de la línea que inicia el bloque. El carácter final de bloque "}" debe situarse en una nueva línea tras la última línea del bloque y alineada con respecto al primer carácter de dicho bloque.

Todas la sentencias de un bloque deben encerrarse entre llaves "{ ... }", aunque el bloque conste de una única sentencia. Esta práctica permite añadir código sin cometer errores accidentalmente al olvidar añadir las llaves. Ejemplo,

Para JAVA

```
if (condicion) {
 variable++;
}
```

Para .NET

```
If Not String.IsNullOrEmpty(strNit) AndAlso strNit.Trim.Length > 0 Then
....
End If
```

La sentencia "try/catch" siempre debe tener el formato siguiente,

Para JAVA

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

Para .NET

```
Try
        For i As Integer = 0 To 14
           ....
        Next
        If digito_chequeo >= 2 Then
           …
        End If
 Catch ex As Exception
```

2.6. Espacios en blanco

Las líneas y espacios en blanco mejoran la legibilidad del código permitiendo identificar las secciones de código relacionadas lógicamente.

Se utilizarán espacios en blanco en los siguientes casos:

Entre una palabra clave y un paréntesis. Esto permite que se distingan las llamadas a métodos de las palabras clave. Por ejemplo:

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

Tras cada coma en un listado de argumentos. Por ejemplo:

```
objeto.unMetodo(a, b, c);
```

Para separar un operador binario de sus operandos, excepto en el caso del operador ("."). Nunca se utilizarán espacios entre los operadores unarios (p.e., "++" o "--") y sus operandos. Por ejemplo:

```
a += b + c;

a = (a + b) / (c + d);

contador++;
```

Para separar las expresiones incluidas en la sentencia "for". Por ejemplo:

```
for (expresion1; expresion2; expresion3)
```

Al realizar el moldeo o "casting" de clases. Ejemplo:

```
Unidad unidad = (Unidad) objeto;
```

2.7. Nomenclatura de identificadores

Las convenciones de nombres de identificadores permiten que los programas sean más fáciles de leer y por tanto más comprensibles. También proporcionan información sobre la función que desempeña el identificador dentro del código, es decir, si es una constante, una variable, una clase o un paquete, entre otros.

Un buen nombre da mucha más información que cualquier otra cosa. Para conseguir buenos nombres hay que usar nombres descriptivos y claros. Deben ser legibles y evitar codificaciones complejas.

2.7.1. Paquetes

Se escribirán siempre en letras minúsculas para evitar que entren en conflicto con los nombres de clases o interfaces. El prefijo del paquete siempre corresponderá a un nombre de dominio de primer nivel, tal como: es, eu, org, com, net, etc.

El resto de componentes del paquete se nombrarán de acuerdo a la siguiente jerarquía: sds.identificadorsubdireccion.identificadordireccion.componente

Ejemplos:

co.sds.subsecretariacorporativa.direcciontic.conexionsql

co.sds.subsecretariasaludpublica.subdireccionvigilancia.primerainfancia

co.sds.subsecretariaserviciosaseguramiento.direccionaseguramiento.afiliacion

java.util.ArrayList

java.util.Date

java.util.Properties

javax.servlet.http.HttpServletRequest

javax.servlet.http.HttpServletResponse

2.7.2. Clases e interfaces

.Los nombres de clases deben ser sustantivos y deben tener la primera letra en mayúsculas. Si el nombre es compuesto, cada palabra componente deberá comenzar con mayúsculas.

Los nombres serán simples y descriptivos. Debe evitarse el uso de acrónimos o abreviaturas, salvo en aquellos casos en los que dicha abreviatura sea más utilizada que la palabra que representa (URL, HTTP, etc.).

Las interfaces se nombrarán siguiendo los mismos criterios que los indicados para las clases. Como norma general toda interfaz se nombrará con el prefijo "I" para diferenciarla de la clase que la implementa (que tendrá el mismo nombre sin el prefijo "I").

class Ciudadano
class OrganigramaDAO
class AgendaService
class IAgendaService

2.7.3. Métodos

Los métodos deben ser verbos escritos en minúsculas. Cuando el método esté compuesto por varias palabras cada una de ellas tendrá la primera letra en mayúsculas.

Los métodos deben ser cortos, hacer una única cosa y mantenerse dentro del mismo nivel de abstracción. Se deberá reducir al mínimo el número de argumentos y se deberá eliminar toda la duplicidad. Ejemplos:

```
public void insertaUnidad(Unidad unidad);
public void eliminaAgenda(Agenda agenda);
public void actualizaTramite(Tramite tramite)
```

2.7.4. Variables

Las variables se escribirán siempre en minúsculas. Las variables compuestas tendrán la primera letra de cada palabra componente en mayúsculas.

Las variables nunca podrán comenzar con el carácter "_" o "$". Los nombres de variables deben ser cortos y sus significados tienen que expresar con suficiente claridad la función que desempeñan en el código (deben ser descriptivos). Debe evitarse el uso de nombres de variables con un sólo carácter, excepto para variables temporales, deben ser legibles y evitar codificaciones complejas

```
Unidad unidad;
Agenda agenda;
Tramite tramite;
```

2.7.5. Constantes

Todos los nombres de constantes tendrán que escribirse en mayúsculas. Cuando los nombres de constantes sean compuestos las palabras se separarán entre sí mediante el carácter de subrayado "_".

```
int LONGITUD_MAXIMA;
int LONGITUD_MINIMA;
```

2.8. Prácticas de programación

2.8.1. Visibilidad de atributos de instancia y de clase

Los atributos de instancia y de clase serán siempre privados, excepto cuando tengan que ser visibles en subclases herederas, en tales casos serán declarados como protegidos.

El acceso a los atributos de una clase se realizará por medio de los métodos "get" y "set" correspondientes (propiedades), incluso cuando el acceso a dichos atributos se realice en los métodos miembros de la clase.

```
public class Unidad {
```

```
 private int id;
 private String nombre;
 ...

 public void actualizaUnidad(Unidad unidad) {
  this.setId(unidad.getId());
  this.setNombre(unidad.getNombre());
 }

 ...
}
```

2.8.2. Referencias a miembros de una clase

Evitar el uso de objetos para acceder a los miembros de una clase (atributos y métodos estáticos). Utilizaremos en su lugar el nombre de la clase. Por ejemplo:

```
metodoUtilidad();    // Acceso desde la propia clase estática
ClaseUtilidad.metodoUtilidad(); // Acceso común desde cualquier clase
```

2.8.3. Constantes

Los valores constantes (literales) nunca aparecerán directamente en el código. Para designar dichos valores se utilizarán constantes escritas en mayúsculas y se declararán, según su ámbito de uso, o bien en una Clase de constantes creada para tal efecto, o bien en la clase donde sean utilizadas.

```
// Uso incorrecto
codigoErrorUsuarioNoEncontrado = 1;
...
switch (error) {
  case codigoErrorUsuarioNoEncontrado:
    ...
}

// Uso correcto
public final int CODIGOERROR_USUARIONOENCONTRADO = 1;
...
switch (error) {
  case CODIDOGERROR_USUARIONOENCONTRADO:
    ...
}
```

2.8.4. Asignación sobre variables

Se deben evitar las asignaciones de un mismo valor sobre múltiples variables en una misma sentencia, ya que dichas sentencias suelen ser difíciles de leer.

```
int a = b = c = 2;  // Evitar
```

No utilizar el operador de asignación en aquellos lugares donde sea susceptible de confusión con el operador de igualdad. Por ejemplo:

```
// INCORRECTO
if ((c = d++) == 0) { }

// CORRECTO
c = d++;
if (c == 0) { }
```

No utilizar asignaciones embebidas o anidadas. Ejemplo:

```
c = (c = 3) + 4 + d;  // Evitar
```

Debería escribirse

```
c = 3;
c = c + 4 + d;
```

2.8.5. Otras prácticas

3. Paréntesis

Es una buena práctica el uso de paréntesis en expresiones que incluyan distintos tipos de operadores para evitar problemas de precedencia de operadores. Aunque la precedencia de operadores nos pueda parecer clara, debemos asumir que otros programadores no tengan un conocimiento exhaustivo sobre las reglas de precedencia.

```
if (w == x && y == z)     // INCORRECTO
```

```
if ((w == x) && (y == z)) // CORRECTO
```

4. Valores de retorno

Los valores de retorno tendrán que ser simples y comprensibles, de acuerdo al propósito y comportamiento del objeto en el que se utilicen.

```
// INCORRECTO
public boolean esProgramador(Empleado emp) {
   if (emp.getRol().equals(ROL_PROGRAMADOR)) {
  return true;   }
 else {
return false;   }    }

// CORRECTO
public boolean esProgramador(Empleado emp) {
   boolean esUnProgramador = false;
     if (emp.getRol().equals(ROL_PROGRAMADOR)) {
    esUnProgramador = true;
}
return esUnProgramador;
}
```

Expresiones en el operador condicional ternario

Toda expresión compuesta, por uno o más operadores binarios, situada en la parte condicional del operador ternario deberá ir entre paréntesis. Ejemplo:

```
(x >= y) ? x : y;
```

Comentarios especiales (TODO, FIXME, XXX)

XXX se utilizará para comentar aquella porción de código que, aunque no tenga mal funcionamiento, requiera modificaciones.

FIXME para señalar un bloque de código erróneo que no funciona.

TODO para comentar posibles mejoras de código, como puedan ser las debidas a optimizaciones, actualizaciones o refactorizaciones.

5. DOCUMENTACIÓN

Se aconseja, como buena práctica de programación, incluir en la entrega de la aplicación la documentación de los ficheros fuente de todas las clases. Dicha documentación será generada por la herramienta "javadoc" ó “NDoc”.

La herramienta "javadoc" construirá la documentación a partir de los comentarios (incluidos en las clases) encerrados entre los caracteres "/**" y "*/". Distinguimos tres tipos de comentarios javadoc, en función del elemento al que preceden: de clase, de variable y de método.

Dentro de los comentarios "javadoc" podremos incluir código html y etiquetas especiales de documentación. Estas etiquetas de documentación comienzan con el símbolo "@", se sitúan al inicio de línea del comentario y nos permiten incluir información específica de nuestra aplicación de una forma estándar.

Como norma general utilizaremos las siguientes etiquetas:

@author Nombre

Añade información sobre el autor o autores del código.

@version InformacionVersion

Permite incluir información sobre la versión y fecha del código.

@param NombreParametro Descripción

Inserta el parámetro especificado y su descripción en la sección "Parameters:" de la documentación del método en el que se incluya. Estas etiquetas deben aparecer en el mismo orden en el que aparezcan los parámetros especificados del método. Este tag no puede utilizarse en comentarios de clase, interfaz o campo. Las descripciones deben ser breves.

@return Descripción

Inserta la descripción indicada en la sección "Returns:" de la documentación del método. Este tag debe aparecer en los comentarios de documentación de todos los métodos, salvo en los constructores y en aquellos que no devuelvan ningún valor (void).

@throws NombreClase Descripción

Añade el bloque de comentario "Throws:" incluyendo el nombre y la descripción de la excepción especificada. Todo comentario de documentación de un método debe contener un tag "@throws" por cada una de las excepciones que pueda elevar. La descripción de la excepción puede ser tan corta o larga como sea necesario y debe explicar el motivo o motivos que la originan.

@see Referencia

Permite incluir en la documentación la sección de comentario "See also:", conteniendo la referencia indicada. Puede aparecer en cualquier tipo de comentario "javadoc". Nos permite hacer referencias a la documentación de otras clases o métodos.

@deprecated Explicación

Esta etiqueta indica que la clase, interfaz, método o campo está obsoleto y que no debe utilizarse, y que dicho elemento posiblemente desaparecerá en futuras versiones. "javadoc" añade el comentario "Deprecated" en la documentación e incluye el texto explicativo indicado tras la etiqueta. Dicho texto debería incluir una sugerencia o referencia sobre la clase o método sustituto del elemento "deprecado".

@since Version

Se utiliza para especificar cuándo se ha añadido a la API la clase, interfaz, método o campo. Debería incluirse el número de versión u otro tipo de información.

El siguiente ejemplo muestra los tres tipos de comentarios "javadoc",

```
/**
 * UnidadOrganizativa.java:
 *
 *  Clase que muestra ejemplos de comentarios de documentación de código.
 *
 * @author jlflorido
 * @version 1.0, 05/08/2008
 * @see documento "Normas de programación v1.0"
 * @since jdk 5.0
 */
public class UnidadOrganizativa extends PoolDAO {

   /** Trazas de la aplicación */
   private Logger log = Logger.getLogger(UnidadOrganizativa.class);

   /** Identificador de la unidad organizativa */
   private int id;

   /** Nombre de la unidad organizativa */
   private String nombre;
```

```
/** Obtiene el identificador de esta unidad organizativa */
public int getId() {
   return id;
}

/** Establece el identificador de esta unidad organizativa */
public void setId(int id) {
   this.id = id;
}

/** Obtiene el nombre de esta unidad organizativa */
public String getNombre() {
   return nombre;
}

/** Establece el nombre de esta unidad organizativa */
public void setNombre(String nombre) {
   this.nombre = nombre;
}

/**
 * Inserta la unidad organizativa en el sistema.
 *
 * @param unidad Unidad organizativa a insertar
 * @throws Exception Excepción elevada durante el proceso de inserción
 */
public void insertarUnidad(UnidadOrganizativa unidad) throws Exception{

   log.debug("-> insertarUnidad(UnidadOrganizativa unidad)");

   Connection conn = null;
   PreparedStatement pstmt = null;
   StringBuffer sqlSb = null;

   try {
      conn = this.dameConexion();

      sqlSb = new StringBuffer("")
      .append("INSERT INTO ORG.UNIDAD_ORGANIZATIVA ")
      .append("(ID, NOMBRE) VALUES (?, ?)");

      pstmt = conn.prepareStatement(sqlSb.toString());
      pstmt.setInt(1, unidad.getId());
```

```
                pstmt.setString(2, unidad.getNombre());
                pstmt.executeUpdate();

            } catch (Exception e) {

                log.error("Error: error al insertar la unidad. " +
                "Descripción:" + e.getMessage(), e);

                throw e;

            } finally {

                log.debug("<- insertarUnidad(UnidadOrganizativa unidad)");

            }
        }
    }
```

La documentación generada por "javadoc" será la siguiente:

a) Página índice de toda la documentación generada:

b) Documentación de la clase "UnidadOrganizativa.java":

Method Detail

getId

```
public int getId()
```

Obtiene el identificador de esta unidad organizativa

---

setId

```
public void setId(int id)
```

Establece el identificador de esta unidad organizativa

---

getNombre

```
public java.lang.String getNombre()
```

Obtiene el nombre de esta unidad organizativa

---

setNombre

```
public void setNombre(java.lang.String nombre)
```

Establece el nombre de esta unidad organizativa

---

insertarUnidad

```
public void insertarUnidad(UnidadOrganizativa unidad)
                    throws java.lang.Exception
```

Inserta la unidad organizativa en el sistema

Parameters:
`unidad` - Unidad organizativa a insertar
Throws:
`java.lang.Exception` - Excepción elevada durante el proceso de inserción
