Guía de desarrollo, Anexo 01

Normas de Codificación

Oficina de Calidad

GERENCIA INFORMÁTICA

JOSEFA VALCÁRCEL, 44
28027-MADRID

Índice General

Índice de Ilustraciones y Tablas

1 Introducción

1.1 Objetivo

El objetivo del presente documento es el de describir una serie de guías y buenas prácticas a la hora de escribir el código, que permitan la legibilidad y mantenibilidad del mismo.

1.2 Audiencia

Este documento está dirigido a todas las personas que colaboren en labores relacionadas con la gestión, desarrollo, auditoría, implantación y explotación de los sistemas de información de la gerencia de informática de la Dirección General de Tráfico, a partir de ahora DGT.

1.3 Glosario

Los términos y acrónimos que se utilizan en este documento y en el resto de documentos de la guía se encuentran recogidos por orden alfabético en el Anexo 30. Glosario con el objetivo de facilitar su lectura y comprensión.

2 Normas de estilo Java

A continuación se expone una serie de normas de estilo y codificación generales que deberán seguir los proyectos desarrollados para la DGT. El cumplimiento de las normas de codificación será verificado mediante el análisis de código realizado por la herramienta *Sonarqube*.

Se puede consultar el listado completo de reglas de codificación que deben cumplir los proyectos en la plataforma *Sonarqube* provista por la Oficina de Calidad a través de la siguiente url: http://192.168.76.234:8080/sonar/coding_rules

Se aconseja encarecidamente la instalación de plugin **SonarLint** dentro del IDE de desarrollo de cada miembro del equipo de desarrollo. En caso de duda consultar anexo 39.

2.1 Criterios de nomenclatura

En el desarrollo de software utilizar una política de nomenclatura correcta es de vital importancia para la comprensión y el mantenimiento del código. Cuando se asignen nombres a clases, métodos, variables, etc. se deberán seguir las siguientes pautas:

- Usar nombres identificativos, que expliquen su contenido o su funcionalidad, que revelen intenciones.
- Usar nombres que se puedan pronunciar y que se puedan buscar.
- Usar nombres de dominio de las soluciones o, en su defecto, nombres de dominio del problema.
- No se deben usar nombres con un solo carácter (excepto para índices) ya que no aportan información.
- No se debe utilizar una misma palabra para enumerar dos conceptos distintos.
- Se debe utilizar una única palabra para cada concepto y se debe mantener la consistencia a lo largo de todo el desarrollo.
- Usar palabras en castellano. Como norma general se debe cumplir con esta pauta excepto para todos aquellos términos técnicos o expresiones cuya traducción al castellano no clarifique el sentido del mismo.
- Se debe poder distinguir entre varias palabras que componen una misma declaración, para ello se debe usar la notación *lowerCamelCase*.

2.2 Normas de codificación

2.2.1 Declaración de clases e interfaces

Las normas de codificación más importantes a la hora de declarar clases e interfaces son las que se enumeran a continuación:

- Los nombres deben ser sustantivos, excepto cuando se tenga una razón de peso para no hacerlo. Deben ser simples y descriptivos.
- El nombre de la clase tiene que describir la responsabilidad que esta desempeña.
- Se deberán evitar los nombres genéricos del tipo *Manager*, *Helper*, *Data*, *Info*, etc.
- Las clases deben ser de tamaño reducido, deben tener una única responsabilidad.
- El número máximo permitido de métodos por clase será 35.

2.2.2 Declaración de métodos

Las normas más importantes a tener en cuenta a la hora de declarar métodos son las siguientes:

- Se deben nombrar usando el duplo verbo-objeto (`mostrarDialogo()` o `buscarPrimerPuntoFrontera()`).
- Los métodos deben ser de tamaño reducido, su longitud no debería superar las 100 líneas de código (LoC), siendo el tamaño recomendado de 10 líneas.
- Los métodos no podrán soportar más de 7 parámetros, siendo el número máximo recomendado de 3 argumentos. En el caso de que sean necesarios más deberá quedar justificada dicha necesidad.
- Los métodos solo deben hacer una cosa. No deben causar efectos secundarios, deben realizar exclusivamente aquello que indica su nombre.
- Si un método invoca a otro, deberán estar verticalmente próximos y el método invocado deberá estar debajo.
- Los métodos con elevada afinidad deberán encontrarse próximos, tanto si se llaman entre sí como si no.
- El nombre del objeto es implícito, y no debe aparecer en el nombre del método.
- Los métodos *get*/*set* deben ser usados para acceder directamente al atributo.
- Los métodos que retornan un valor deben tener un nombre que refleje lo que retorna (*getObjectState()*).

2.2.3 Formato del código

Se recomienda utilizar la configuración de formato de código que ofrecen los IDE's autorizados para adecuarse a las reglas establecidas por *"Java Code Conventions"*. Ver Anexo para configurar el formato de código, al final de este documento.

Se enumeran a continuación las reglas más importantes:

- El ancho de una línea de código no debe exceder los 150 caracteres.
- Las llaves de principio irán en la misma línea de código y las de final en su propia línea.
- En ningún caso se deberá escribir más de una instrucción por línea.
- La distancia vertical entre conceptos relacionados deberá ser mínima.
- Las variables se deben declarar lo más próximo posible a su uso.
- Las variables de control de bucles deben declararse en la instrucción del bucle.
- Las variables de instancia deben declararse en la parte superior de la clase.
- Se debe evitar romper el sangrado del código. Debe primar la legibilidad.

2.2.4 Uso de sentencias switch en la creación de instancias

Las sentencias *switch* deberán evitarse siempre que sea posible en la creación de instancias de clases. Siempre que sea necesario se deberá contemplar como alternativa la creación de una fábrica abstracta que utilice la instrucción *switch* para la creación de instancias polimórficas

2.2.5 Declaración de variables

A la hora de declarar variables, se deben tener en cuenta las siguientes indicaciones:

- Deben dejar claro qué contienen.
- Deben ser fáciles de leer, y tan cortas como sea posible sin que pierda su significado.
- Usar plural para *arrays* y colecciones de objetos.
- Deben ser auto-descriptivas (indice y no idx, contadorNodos y no cnt)
- Usar nombre de variables estándares:
    - Index: i, j, k

  - Excepción: e, y si hay múltiples usar nombre completo sqlException
  - Tipos genéricos: T (*type*), S (*struct*), E (*enum*), K (*key*), V (*value*)
- Las constantes (*static final*) deben escribirse en mayúsculas y separadas por guion bajo (PRIORITY_NORMAL, PRIORITY_HIGH, DEFAULT_PRICE_PER_POUND).
- Sólo se podrán añadir nombres de contexto en aquellos casos donde pueda existir ambigüedad.
- No se podrán usar nombres con variaciones mínimas.
- No se podrá indicar el tipo de la variable en su nombre (notación húngara).
- No se deberán añadir prefijos ni sufijos.
- Las variables de instancia se deberán declarar siempre en la parte superior de la clase.

2.2.6 Organización de miembros

Se recomienda separar los miembros de una clase según su visibilidad (*private, protected, public*). El orden no es relevante en cuestión de rendimiento, pero se propone que sea el siguiente:

1. *public*
2. *protected*
3. *private*

La palabra reservada *final* puede ser usada siempre que se desee, para:

- Comunicar la intención de esa variable claramente.
- Ayudar al compilador a realizar pequeñas optimizaciones.
- Según donde se use, puede tener más de un significado.
- Una Clase final no puede ser extendida
- Un método final no puede ser sobre-escrito
- Campos, parámetros y variables locales final no pueden variar su valor una vez han sido inicializados

2.2.7 Variables locales

Hay dos casos posibles en los cuales estas variables pueden ser inicializadas con algún valor por defecto (*null, 0, false* ó *empty* en el caso de un *String*):

- Variables que deben ser visibles fuera de un bloque *try*, en cuyo caso deben ser declaradas e inicializadas por defecto antes del *try*
- Variables de ciclo, que deben ser inicializadas con algún valor justo antes del ciclo

2.2.8 Validación de argumentos de métodos

En la primera línea del método se deben validar sus argumentos, para capturar y tratar algún evento de error cuanto antes. Algunas validaciones comunes:

- Comprobar que el objeto es no nulo
- Comprobar que un texto no está vacío
- Comprobar que un número está en un rango determinado

2.2.9 Sentencias

2.2.9.1 Sentencias simples

Cada línea de código debe contener como mucho una sentencia. Ejemplo:

```
argv++; argc--; //Incorrecto al haber dos sentencias en una misma línea
```

2.2.9.2 Sentencias de retorno

Una sentencia *return* con un valor no debe usar paréntesis.

Sólo debe existir un único punto de retorno de un método y éste debe ser siempre la última sentencia en el método.

2.2.9.3 Sentencias switch

Deben seguirse las siguientes pautas para su implementación:

- No se pueden dejar sentencias *switch* vacías.
- Siempre debe tener implementado un caso por defecto.
- El caso por defecto siempre debe ir al final del resto de casos.
- En cada caso debe existir una sentencia break.

2.3 Documentación del código

El código fuente generado deberá estar debidamente documentado mediante *Javadoc* y comentarios dentro del código. *Javadoc* es una herramienta que permite:

- Saber qué hace un método sin necesidad de leer su implementación, lo cual lleva normalmente más esfuerzo que leer un texto
- Por otro lado, la implementación puede ser verificada mirando su *Javadoc*, para corregir bugs sólo leyendo qué se supone que hace el método, sin necesidad de ejecutarlo.

Se debe usar *Javadoc* para la documentación de todos los artefactos (Clases, métodos,…) definidos en un proyecto java.

Para la correcta generación del *Javadoc* de un proyecto se deberán tener en cuenta los puntos que se describen en los siguientes apartados.

2.3.1 Idioma

Se utilizará única y exclusivamente el castellano para generar la documentación.

Los comentarios deben ser claros y concisos. Deben expresar la intención, el propósito y no qué se hace internamente. Debe ser escrito asumiendo que la persona que los leerá tiene conocimientos del lenguaje y no necesariamente de la aplicación.

2.3.2 Clases

Las clases Java contenidas en los proyectos deberán documentarse debidamente con los siguientes elementos:

- Frase concisa y descriptiva de la clase.
- Frases (o párrafos) informando de las funcionalidades que la clase ofrece, su finalidad, etc. Este elemento es opcional siempre y cuando con la primera frase de la descripción no quede ninguna duda de la funcionalidad de la clase comentada.
- Ejemplos de uso o referencias a código donde exista un ejemplo de uso (a través del tag @link).

2.3.3 Métodos

Los métodos deberán documentarse incluyendo la descripción y su firma (parámetros, valor de retorno y excepciones que lanza si procede)

2.3.4 Guía de estilo

- Utilización de la tercera persona del singular.
- La primera frase de la descripción debe ser un breve, pero completo, resumen (aproximadamente de 10 palabras) describiendo la finalidad del método.
- Comienzo de la descripción con un verbo dado que el método implementa una operación.
- Evitar redundancia, es decir, no describir con palabras lo que se entiende con un simple vistazo al nombre del método a documentar.
- En ningún caso se deberán utilizar comentarios para aclarar defectos del código.
- En caso utilizar código HTML, se recomienda no abusar de su uso y éste debe estar bien formado.
- En ningún caso se deberán utilizar comentarios como registro de los cambios realizados en el código.
- En ningún caso se deberá dejar código comentado.

2.4 Duplicidad del código

Se debe evitar duplicar código. El código duplicado es un problema grave que aumenta el tamaño del código y empeora drásticamente la mantenibilidad del software. Por ejemplo, si tenemos un algoritmo duplicado en tres funciones o clases distintas, requerirá una modificación triple si alguna vez cambia el algoritmo.

Se auditará el código de forma automática para asegurar que no haya código duplicado. La existencia de bloques de código duplicado supondrá un error que deberá ser solucionado para poder realizar la aceptación del código entregado.

3 Normas de estilo SQL

El presente capítulo describe las normas aplicables, tanto para la escritura de sentencias del lenguaje SQL como recomendaciones importantes de uso de sus diferentes posibilidades y cláusulas.

3.1 Normas de codificación

En caso de ser necesario, se seguirán las siguientes recomendaciones a la hora de escribir sentencias SQL. Las presentes normas de estilo serán auditadas por el departamento de calidad.

3.1.1 Referencias a columnas

No usar la notación SELECT *. En su lugar se debe listar cada columna explícitamente, a fin de independizar la sentencia, de los posibles cambios en la estructura de la tabla.

3.1.2 Expresiones complejas

Se agruparán las expresiones complejas mediante el uso de paréntesis, incluso cuando la sintaxis del lenguaje no lo requiera a fin de facilitar su entendimiento y evitar errores. En especial, especificar mediante el uso de paréntesis la precedencia de operaciones las aritméticas (suma, resta, multiplicación y división) y de los operadores lógicos (AND, OR y NOT)

3.1.3 Cálculos de grupos

Copiar exactamente las expresiones de grupo de la cláusula SELECT en la cláusula GROUP BY para evitar errores sintácticos. No se deben incluir expresiones de agrupación en la cláusula GROUP BY que no estén en la cláusula SELECT, para evitar resultados confusos.

3.1.4 Ordenaciones

No fiarse de las ordenaciones implícitas realizadas como resultado del empleo de índices o de la cláusula GROUP BY porque las reglas de ordenación implícita pueden cambiar en las nuevas

versiones de ORACLE. Utilizar los nombres de las columnas en lugar de los números de orden de la cláusula SELECT.

Evitar que las subconsultas hagan ordenaciones.

Los nombres de las columnas deben ser usados en el ORDER BY, esto hace que el código sea más sencillo de leer y mantener.

3.1.5 Consultas de múltiples tablas (combinaciones)

Se habrá de listar las tablas de la cláusula FROM en un orden determinado. En general, se listarán las tablas según su tamaño, primero las tablas que devuelven mayor número de filas o primero las de detalle y después las maestras.

Escribir las condiciones de la combinación (*join*) antes de cualquier otra condición de la cláusula WHERE. Listar las condiciones de la combinación (*join*) en el orden en que aparecen las tablas en la cláusula FROM.

Es recomendable que los campos de un join se definan con el mismo tipo de datos y misma longitud. En el caso de varchar2 la diferencia de longitud hace que se rellene a blancos el de menor longitud y se haga la comparación carácter a carácter. En cualquier caso esto no supone una penalización en el rendimiento.

Las sentencias SQL no deben contener combinaciones (join) de demasiadas tablas. Hay que evitar el uso de combinaciones de más de tres tablas.

3.1.6 Escritura de cálculos

Realizar los cálculos de la cláusula WHERE con constantes en lugar de utilizar columnas, siempre que sea posible, para mejorar el rendimiento de la consulta. La conveniencia de hacerlo así radica en que asociando el cálculo a la columna de la WHERE se desactivaría un posible índice asociado a la columna.

3.1.7 Cláusula WHERE

Las sentencias DELETE y UPDATE deben contener siempre una clausula WHERE para mantener la modificación de filas bajo control. En caso contrario podría resultar en perdida de datos.

Evitar funciones sobre columnas en la cláusula WHERE. Por ejemplo, evitar poner el operador de concatenación en un WHERE.

No mezclar tipos de datos, si una columna en un WHERE es numérico no usar comillas. Si es de tipo carácter usar siempre comillas.

3.1.8 Operador UNION

El operador UNION combina el resultado de dos o más consultas, esto ayuda a dividir un problema complicado en múltiples sentencias SQL más sencillas.

UNION es significativamente más lento que UNION ALL debito a que UNION elimina entradas duplicadas ejecutando un DISTINCT de forma interna para conseguirlo.

UNION ALL no elimina duplicados y devuelve todos los resultados de la consulta. Se ejecuta más rápido comparado con UNION. Sin embargo, la cantidad de datos devuelta por UNION ALL puede ser significativamente mayor que la devuelta por UNION. En una red lenta, el rendimiento obtenido por UNION ALL puede ser penalizado por el tiempo invertido en la transferencia un volumen muy grande de datos.

Por estos motivos se recomienda evitar el uso del operados UNION. Una posibilidad es substituir UNION por operadores OR.

3.1.9 Comentarios

Cuando se consultan varias tablas, usar alias, y emplear esos alias en la sentencia SELECT, de esta forma el gestor de bases de datos no necesita interpretar a que tabla pertenece cada columna.

3.1.10 Índices

No crear índices si no se van a usar.

Para columnas que normalmente van juntas en las consultas se puede considerar un la creación de un índice compuesto.

Evitar hacer consultas con condiciones por campos no indexados.

Evitar el recorrido completo de tablas muy grandes.

3.1.11 Inserciones

Se deben indicar explícitamente la lista de valores en un INSERT. Una sentencia INSERT donde no se indican explícitamente la lista de columnas que se van a insertar su funcionamiento depende de que la estructura de la tabla no cambie. Además, no indicar explícitamente la lista de columnas en un INSERT penaliza la comprensión del código.

3.1.12 Tablas

Las tablas deben ser referenciadas a través de un alias. El uso de alias ayuda a hacer más comprensible la consulta.

3.2 Recomendaciones de optimización

Las recomendaciones expuestas a continuación, aunque son beneficiosas de cara al rendimiento de la consulta, tienen generalmente el efecto de dificultar la comprensión de la lógica de las sentencias SQL. Por ello, deben aplicarse fundamentalmente en sentencias donde la optimización sea crítica o represente una ganancia apreciable.

3.2.1 Cláusula BETWEEN

Convertir este operador en dos comparaciones que utilicen operadores en los límites de la comparación.

3.2.2 Cláusula LIKE

Evitar el uso del símbolo comodín "%" en la primera posición de la cadena de comparación a fin de evitar la desactivación del índice. Evitar igualmente el uso de este operador con números y fechas a fin de no forzar una conversión implícita de datos a una cadena de caracteres.

Además, la cláusula LIKE no debe ser usada sin carácter comodín. En algunos casos el uso de LIKE sin carácter comodín puede devolver resultados diferentes que si se usa =.

Evitar el uso del predicado LIKE, se debe substituir por igual.

3.2.3 Cláusula OR

Escribir las expresiones OR entre paréntesis; si es posible, reducirla a una condición simple mediante el operador IN**.**

El optimizador de Oracle convierte la función IN en tantas expresiones OR como valores haya en la lista de la función IN. Por ello, según cómo se haya configurado el gestor Oracle, el optimizador puede elegir la realización de un recorrido completo en la tabla (full table scan) en vez de acceso a través de índices cuando en número de valores excede de un máximo variable (más de cinco o siete valores).

3.2.4 Cláusula IN

Evitar el uso de IN, usar en su lugar EXISTS. El operador EXISTS termina tan pronto encuentra un valor que satisface la condición, sin embargo IN recorrerá la tabla completa.

Evitar el uso de grandes listas de valores en sentencias SQL con la cláusula IN.

3.2.5 Cláusula UNION ALL

Es preferible usar UNION ALL a UNION. UNION ALL no elimina duplicados por lo que la consulta es mucho más eficiente.

3.2.6 Cláusula EXISTS

Las consultas SQL que usan subconsultas con EXITS son muy ineficientes debido a que la subconsulta es ejecutada para cada fila de la tabla externa. Hay formas más eficientes de escribir la mayoría de las consultas. Por ejemplo, usando INNER JOIN.

A continuación se muestra un cuadro resumen con las principales recomendaciones vistas en este documento y que se recomienda para el desarrollo de aplicaciones en la DGT.

Recomendación	No Conforme	Conforme
Las sentencias DELETE y UPDATE deben contener una clausula WHERE	UPDATE table SET status='0';	UPDATE table SET status='0' WHERE id>1000';
La cláusula LIKE no debe ser usada sin carácter comodín	SELECT nombre FROM table WHERE nombre LIKE 'aaa';	SELECT nombre FROM table WHERE nombre LIKE 'aaa%';
El valor de la cláusula LIKE no debe empezar con el carácter comodín	SELECT nombre FROM table WHERE nombre LIKE '%aaa';	
ROWNUM no debe usarse al mismo nivel que ORDER BY	SELECT nombre FROM table WHERE rownum < 10 ORDER BY nombre;	SELECT * FROM (SELECT nombre FROM table ORDER BY NOMBRE) WHERE rownum<10;
Las condiciones de la cláusula WHERE no deben ser contradictorias	SELECT nombre FROM table WHERE nombre = 'aaa' and nombre = 'bbb';	
Los nombre de las columnas deben ser usados en el ORDER BY	SELECT col2, col3 FROM table ORDER BY 1 ASC;	SELECT col2, col3 FROM table ORDER BY col2 ASC;
Las sentencias SQL no deben contener join de demasiadas tablas	Más de 3 join entre tablas	
Columnas claramente definidas en la sentencia SELECT	SELECT * FROM table;	SELECT col1, col2 FROM table;
No usar EXISTS con subconsultas	SELECT a.nombre FROM tablea a WHERE EXISTS (SELECT * FROM tableb WHERE a.id = b.id);	SELECT a.nombre FROM tablea a INNER JOIN tableb b ON a.id = b.id;
El operados de join (+) no debe ser usado	SELECT * FROM tablea a , tableb b WHERE a.id = b.id (+);	SELECT * FROM tablea a LEFT OUTER JOIN tableb b ON a.id = b.id;

Se deben indicar explícitamente las lista de valores en un INSERT	INSERT INTO table VALUES (1, ‘a’, ‘b’);	INSERT INTO table (col1, col2, col3) VALUES (1, ‘a’, ‘b’);
Las tablas deben ser referenciadas a través de un alias	SELECT col1, col2, col3 FROM table a INNER JOIN tableb ON tablea.id = tableb.id;	SELECT a.col1, a.col2, b.col3 FROM table a INNER JOIN tableb b ON a.id = b.id;
Evitar el uso del operador UNION	SELECT nombre FROM table WHERE id=1 UNION SELECT nombre FROM table WHERE id=2;	SELECT nombre FROM table WHERE id=1 OR id=2;

3.3 Uso de PL/SQL

Su uso, como norma, no está permitido, ya que la lógica de la aplicación ha de estar contenida en las clases java de negocio. En caso de existir alguna restricción que requiera de su uso, se enviará, de forma previa a su desarrollo, una solicitud al departamento de calidad exponiendo las causas y la necesidad de su aplicación para que se estudie su necesidad y viabilidad por dicho departamento.

Sólo podrán utilizarse sentencias de este tipo si está debidamente aceptada la causa justificada y se ha hecho una excepción que se ha recogido formalmente en las condiciones de aplicación al proyecto.

4 Anexo: Configuración del formato de código

Una cosa importante en lo referente al "código limpio" es que el código deber estar en un formato correcto (tabulaciones, saltos de línea, etc.). El equipo de proyecto debe seguir el mismo estilo de código basado en "*java code conventions"*. Los IDE's facilitan esta tarea y permiten definir el formateo de código que deberá ser compartido por el equipo de proyecto.

4.1 Eclipse

Opción de menú: "Window > Preferences"

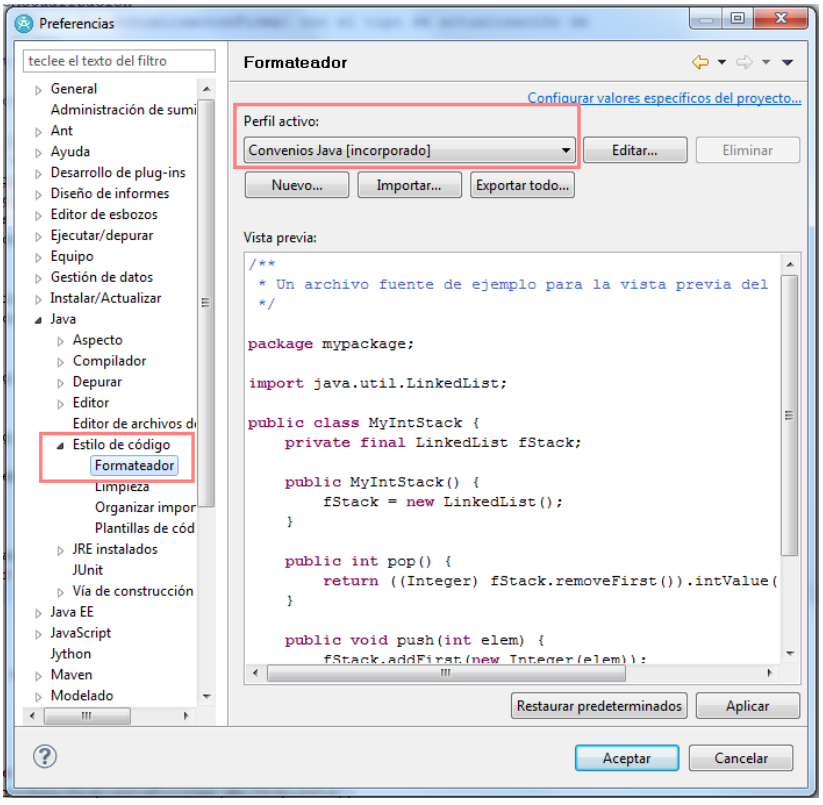

Ilustración 1: Formateador de código Eclipse

4.2 Android Studio

Opción de menú: “File >Settings”

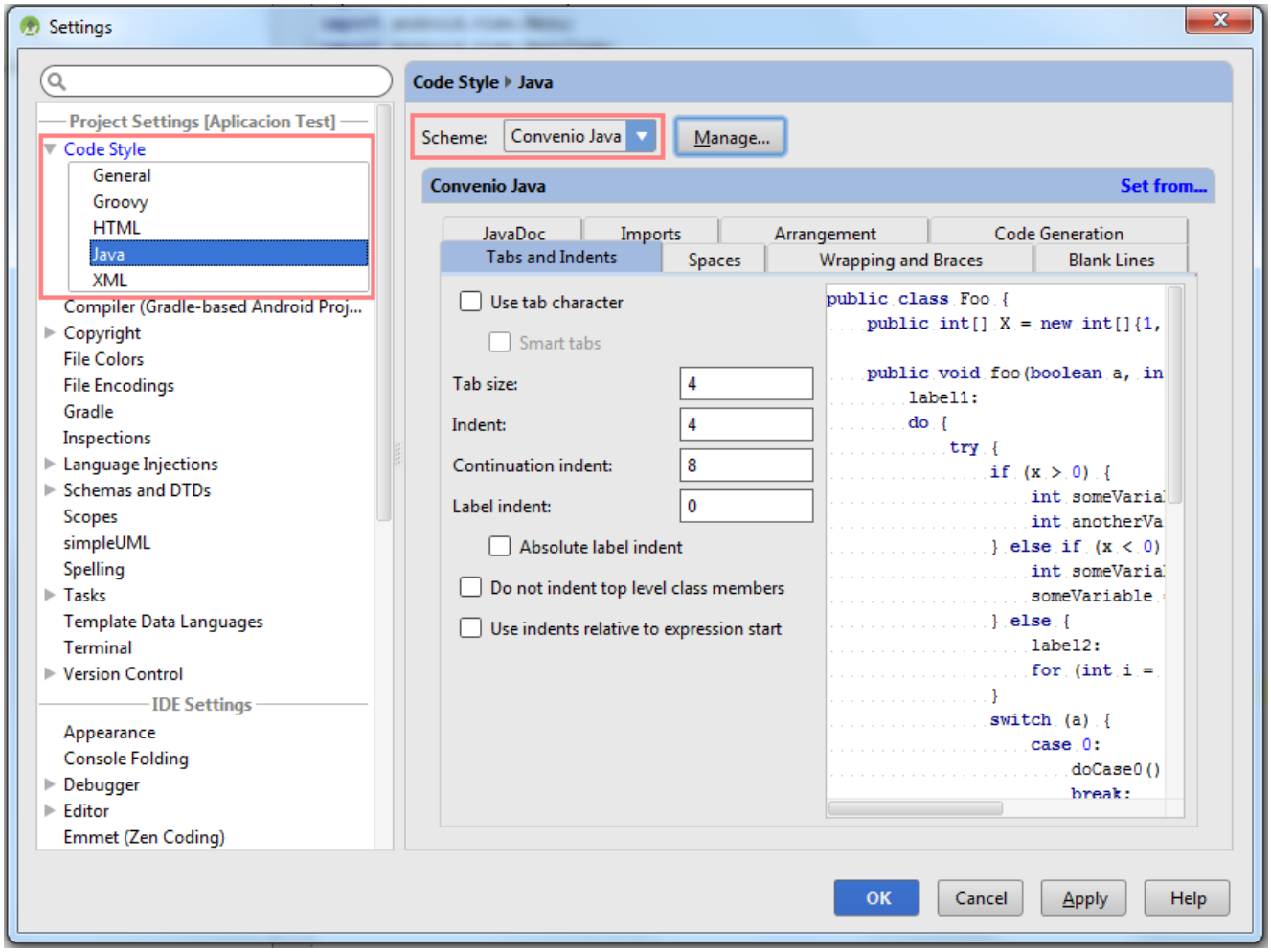

Ilustración 2: Formateador de código Android Studio

5 Bibliografía

1. **Martin, Robert C.** (2009). Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River, NJ: Prentice Hall
2. **Fowler, Martin. Beck, Kent. Brant, John. Opdyke, John. Roberts, Don.** (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley