ESTANDAR DE PROGRAMACION

PROYECTO:

APLICACIÓN MÓVIL PARA EL SEGUIMIENTO MUNDIAL DE CASOS DE COVID-19 CON DATOS EN TIEMPO REAL

INTEGRANTES:

- PILCO QUISPE, Mireya Flavia
- SALAMANCA CONTRERAS, Fiorella Rosmery
- ZAVALA VENEGAS, Luis Ángel

TACNA - PERÚ
2020

HISTORIAL DE VERSIONES

Fecha	***Versión***	***Descripción***	***Autor***
26/05/2020	1.0	Creación del documento	MP, FS, LZ

ESTANDAR DE PROGRAMACION

1. OBJETIVO

El objeto del presente documento es el establecimiento de los estándares o convenciones de programación empleados en el desarrollo de software sobre la plataforma Java. Este modelo de programación está basado en los estándares recomendados por Sun Microsystems, que han sido difundidos y aceptados ampliamente por toda la comunidad Java, y que han terminado por consolidarse como un modelo estándar de programación de facto.

Estas normas son muy útiles por muchas razones, entre las que destacan:

- Facilitan el mantenimiento de una aplicación. Dicho mantenimiento constituye el 80% del coste del ciclo de vida de la aplicación.

- Permite que cualquier programador entienda y pueda mantener la aplicación. En muy raras ocasiones una misma aplicación es mantenida por su autor original.

- Los estándares de programación mejoran la legibilidad del código, al mismo tiempo que permiten su compresión rápida.

2. ALCANCES

El estándar de programación se utilizará en todos los desarrollos de software realizados para la Aplicación Móvil para el Seguimiento Mundial de Casos De Covid-19 con Datos en Tiempo Real.

3. DEFINICIONES

- ✔ Metodología de Programación Permite realizar mantenimientos en menor tiempo y costo, migrar código consistentemente, habilita a los programadores moverse entre proyectos diferentes y asegura la comunicación técnica entre desarrolladores.

- ✔ Sistema Conjunto de elementos que interactúan entre sí para lograr un objetivo común.

4. RESPONSABILIDAD

- ✔ De los analistas programadores Cada analista programador deberá utilizar el estándar de programación en los desarrollos de software posteriores a la fecha de aprobación del mismo.

- ✔ Es responsabilidad del Analista Supervisor controlar la aplicación del estándar de programación en los desarrollos de software posteriores a la fecha de aprobación del mismo.

5. ESTANDAR DE PROGRAMACION DE JAVA

5.1. Organización de ficheros

Las clases en Java se agrupan en paquetes. Estos paquetes se deben organizar de manera jerárquica, de forma que todo código desarrollado para el Ayuntamiento de Málaga tendrá que estar incluido dentro del paquete "eu.malaga".
Dentro del paquete principal las clases se organizarán en subpaquetes en función del área, organismo o sección del Ayuntamiento al que pertenezca el código desarrollado. Por ejemplo, si estamos desarrollando un servicio web de inscripción a un curso de programación Java del IMFE las clases de dicho servicio se incluirían en el paquete "eu.malaga.imfe.webservices.cursojava" o similar.

Un fichero consta de secciones que deben estar separadas por líneas en blanco y comentarios opcionales que identifiquen cada sección.

Deben evitarse los ficheros de gran tamaño que contengan más de 1000 líneas. En ocasiones, este tamaño excesivo provoca que la clase no encapsule un comportamiento claramente definido, albergando una gran cantidad de métodos que realizan tareas funcional o conceptualmente heterogéneas.

5.2. Fichero fuente Java (.java)

Cada fichero fuente Java debe contener una única clase o interfaz pública. El nombre del fichero 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 fichero fuente que la clase pública. La clase pública debe estar situada en primer lugar dentro del fichero fuente.

En todo fichero fuente Java distinguimos las siguientes secciones:

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

5.3. Comentarios de inicio

Todo fichero 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.

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

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

```
/*
 * @(#)NombreFichero.java version(1.0) fecha (dd/MM/yy)
 *
```

```
* UPT
* Construction de Software II
*/

/**
*
 * @author Nombre Apellido
* @version version(1.50), fecha (dd/MM/yy)
 * @since versionanterior(1.4)
 */
```

5.4. Sentencias de paquete

La primera línea no comentada de un fichero fuente debe ser la sentencia de paquete, que indica el paquete al que pertenece(n) la(s) clase(s) incluída(s) en el fichero fuente. Por ejemplo,

```
package javax.crypto;
```

5.5. 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.
Sentencia class / interface	
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).
Constructores	
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.

5.6. Sangría

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

5.7. Longitud de línea

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

5.8. 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();
}
```

5.9. Comentarios

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

5.10. 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.

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

- 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.

```
/* Esto es un comentario de línea */
// Esto es otro comentario de línea
```

- Comentario a final de línea

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

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

5.11. Comentarios de documentación

Los comentarios de documentación, también denominados "comentarios javadoc", 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.

5.12. Declaraciones

- Una declaración por línea

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

```
int idUnidad;   // Identificador de la unidad organizativa
String[] funciones; // Funciones de la unidad
```

5.13. 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.

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

5.14. 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.

```
public classe ClaseEjemplo extends Object {
 int variable1;
 int variable2;

 public ClaseEjemplo() {
  variable1 = 0;
  variable2 = 1;
 }
 ...
}
```

5.15. Sentencias

Cada línea debe contener como máximo una sentencia. Ejemplo,

```
int contador++;
int variable--;
```

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,

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

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

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

En el bloque "catch" siempre se imprimirá una traza de error indicando el tipo de excepción generada y posteriormente se elevará dicha excepción al código invocante, salvo que la lógica de ejecución de la aplicación no lo requiera.

Siempre se utilizará el bloque "finally" para liberar recursos y para imprimir trazas de monitorización de fin de ejecución.

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

5.16. 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;

5.17. 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 maýusculas.
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

5.18. 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.

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

5.19. 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. Debe evitarse el uso de nombres de variables con un sólo carácter, excepto para variables temporales.

Unidad unidad;
Agenda agenda;
Tramite tramite;

5.20. 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;

5.21. Documentación: javadoc

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".

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 cuando 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)");

        }
    }
}
```