LINEAMIENTOS DE DESARROLLO

PROYECTO:

DISEÑO E IMPLEMENTACIÓN DEL SISTEMA DE PLANEACIÓN, SEGUIMIENTO Y EVALUACIÓN

DOCUMENTO

PREPARADO PARA:



DOCUMENTO

PREPARADO POR:

Un dibujo de una cara feliz

Descripción generada automáticamente con confianza baja

CONTROL DE VERSIONES

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ELABORÓ | REVISÓ | VERSIÓN | FECHA DE ENTREGA | FECHA DE ACTUALIZACIÓN Y ENTREGA |
| Adalberto Marin Coca Pacheco | Alejandro López León | 1.0 | 21/07/23 | 21/07/23 |
|  |  |  |  |  |
|  |  |  |  |  |

SECCIÓN DE FIRMAS

|  |  |  |
| --- | --- | --- |
| Responsable  Juan C. Velázquez Martínez  Scrum Máster  Tecnología con un enfoque humano Kohmi, S. de R.L. de C.V. | Validó  Gloria Natali Guillén Valencia  Subdirectora de Área  Comisión Nacional para la Mejora Continua de la Educación | Autorizó  Adrián Castelán Cedillo  Director General  Comisión Nacional para la Mejora Continua de la Educación |

**ÍNDICE DE CONTENIDO**

[1 INTRODUCCIÓN 5](#_Toc140911298)

[1.1 Objetivo 5](#_Toc140911299)

[1.2 Alcance 5](#_Toc140911300)

[1.3 Referencias 5](#_Toc140911301)

[2 ESTÁNDARES DE CODIFICACIÓN 5](#_Toc140911302)

[2.1 Idioma 6](#_Toc140911303)

[2.2 Sangrado 6](#_Toc140911304)

[2.3 Declaraciones. 6](#_Toc140911305)

[2.4 Paquetes. 6](#_Toc140911306)

[2.4.1 Nombrado. 8](#_Toc140911307)

[2.5 Clases e Interfaces. 9](#_Toc140911308)

[2.5.1 Nombrado. 9](#_Toc140911309)

[2.6 Variables. 10](#_Toc140911310)

[2.6.1 Nombrado. 10](#_Toc140911311)

[2.7 Constantes. 10](#_Toc140911312)

[2.8 Métodos. 11](#_Toc140911313)

[2.9 Tamaño de la línea. 12](#_Toc140911314)

[2.10 Formato general. 12](#_Toc140911315)

[2.10.1 Comentarios 12](#_Toc140911316)

[2.10.1.1 Archivo (encabezado). 12](#_Toc140911317)

[2.10.1.2 Clases. 13](#_Toc140911318)

[2.10.1.3 Métodos 13](#_Toc140911319)

[2.10.2 Sentencias import. 14](#_Toc140911320)

[2.10.3 Cuerpo de la clase o interface. 14](#_Toc140911321)

[2.10.3.1 Atributos. 14](#_Toc140911322)

[2.10.3.2 Métodos. 14](#_Toc140911323)

[2.11 Parámetros. 15](#_Toc140911324)

[2.12 Sentencias de control del flujo. 16](#_Toc140911325)

[2.13 Bloques. 16](#_Toc140911326)

[2.14 Comentarios dentro del código. 17](#_Toc140911327)

[2.15 Sentencia for. 18](#_Toc140911328)

[2.16 Sentencia while. 18](#_Toc140911329)

[2.17 Sentencia do-while. 18](#_Toc140911330)

[2.18 Sentencia switch. 19](#_Toc140911331)

[2.19 Condicionales. 20](#_Toc140911332)

[2.20 Espacios en blanco en el código. 20](#_Toc140911333)

[2.21 Mensajes de depuración. 21](#_Toc140911334)

[2.22 Más consideraciones de código. 21](#_Toc140911335)

[2.23 Pseudocódigo. 22](#_Toc140911336)

[2.24 Reglas generales para los archivos “.java”. 22](#_Toc140911337)

[3 PERFORMANCE 23](#_Toc140911338)

[4 MEJORES PRÁCTICAS DE HIBERNATE. 23](#_Toc140911339)

[4.1 Para cada clase que participa en una colección. 23](#_Toc140911340)

[4.2 Como cargar datos relacionados. 23](#_Toc140911341)

[4.3 Anotaciones y mapas. 23](#_Toc140911342)

[4.4 Uso de memoria cache. 24](#_Toc140911343)

[4.5 Asociaciones. 24](#_Toc140911344)

[4.6 Consultas. 24](#_Toc140911345)

[4.7 Soluciones estándar. 24](#_Toc140911346)

[5 LINEAMIENTOS DE DISEÑO. 24](#_Toc140911347)

[6 LINEAMIENTOS DE MANEJO DE REPOSITORIO DE VERSIONADO DE CÓDIGO. 24](#_Toc140911348)

[7 ESPECIFICACIÓN TÉCNICA DE LA INTERFAZ. 27](#_Toc140911349)

# INTRODUCCIÓN

De acuerdo con la IEEE-Computer Society durante la construcción de un software surge un número de cuestiones clave que son críticas para la integridad general y la funcionalidad de la solución de software y que deben solucionarse. Estas pueden comprender:

* Minimizar la complejidad.
* Anticiparse al cambio.
* Construir soluciones de software que se puedan verificar.
* Construir soluciones de software con apego a normas.
* Construir soluciones de software que sean seguras.

El presente documento ofrece una quía para solucionar estas cuestiones.

## Objetivo

Brindar una guía para que el programador:

* Escriba código que sea simple y fácil de leer (mediante el uso de normas, usando técnicas especiales de codificación, usando técnicas enfocadas a la calidad, empleando buenas prácticas de programación).
* Minimice la complejidad en el diseño (mediante el uso de abstracción, reducir el acoplamiento entre módulos, escribiendo módulos tan cohesivos e independientes como sea posible y mediante la aplicación de patrones de software).
* Reducir la complejidad del proceso de desarrollo.

## Alcance

Esta información está dirigida a todos los ingenieros relacionados con la fase de construcción de una solución de software basada en la plataforma Java Enterprise Edition y los frameworks relacionados específicamente Querkus, Spring Web API, Hibernate.

## Referencias

<http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>

<http://oreilly.com/catalog/javapt/chapter/ch04.html#_Toc474748094>

# ESTÁNDARES DE CODIFICACIÓN

Estos estándares se encuentran alineados a lo descrito en la Guía al modelo de diseño y para su correcta implementación los usuarios involucrados se deberán apegar al proceso de desarrollo vigente en la DGTIC de la SecretarÍa de Educación Pública, además se debe apegar al framework del Anexo Técnico de MEJOREDU.

## Idioma

Todo el código será escrito en el idioma español. A continuación, se describen las partes del código que deberán ser escritas en el idioma requerido:

* Nombres de paquetes
* Nombres de clases
* Nombres de métodos
* Nombres de variables y atributos.
* Comentarios de todo tipo.

Existen algunas restricciones en el uso del idioma, mismas que serán listadas en la sección correspondiente más adelante.

## Sangrado

El sangrado del código debe ser de 4 espacios. No se deben utilizar tabuladores por la diversidad de trato que ofrecen los editores sobre este carácter.

## Declaraciones.

Se debe utilizar una sola línea por cada declaración de datos y por cada línea programada a menos que el utilizar más de una sentencia por línea facilite su lectura.

## Paquetes.

La estructura general del proyecto debe cumplir con los siguientes paquetes como mínimo:

|-- main

| `--java

| `--resources

| `-- bundle

| `-- conf

| `-- reportes

| `--webapp

| `-- css

| `-- images

| `-- js

| `-- pages

| `-- WEB-INF

|-- test

Donde:

| Paquete | Descripción |
| --- | --- |
| main | Contendrá el código fuente del proyecto |
| java | Contendrá todos los archivos java de la aplicación |
| resources | Contendrá todos los recursos de la aplicación como son archivos de propiedades, internacionalización, configuración y plantillas de reportes (jasper reports) |
| webapp | Contendrá todos los archivos correspondientes a la parte web útiles para publicar el portal de Descubridor de Servicios API (Swagger), como son página inicial Index, hojas de estilo, javascript, imágenes y las propias paginas jsf. |
| test | Contendrá las pruebas unitarias del proyecto |

Tabla 1 – Estructura general del proyecto

A partir del paquete java, los nombres de los subpaquetes deberán describir la funcionalidad que ese paquete encapsula, deben denotarse con minúsculas y en español.

Deben de comenzar con el nombre del dominio de la organización de manera inversa, y continuar incluyendo coordinación, siglas del sistema, subsistema o módulo.

Donde:

|  |  |
| --- | --- |
| Paquete | Descripción |
| coordinación | Indica la coordinación a la que pertenece MEJOREDU el sistema. |
| siglas del sistema | Se refiere al nombre corto con el que se identifica una aplicación. |
| módulos | Se refiere al módulo o subsistema del sistema. |

Tabla 2 - Funcionalidad que encapsulan los paquetes

Adicionalmente se pueden desglosar los paquetes a mayor detalle indicando paquetes de integración, servicios y presentación.

Donde:

| Paquete | Descripción |
| --- | --- |
| integración | El paquete de integración es responsable de proveer acceso a recursos de bases de datos |
| servicios | El paquete de servicios es responsable de realizar cualquier procesamiento del negocio. |
| presentacion | Este paquete encapsula todos los beans y validadores de la presentación. |

Tabla 3 - Desglose de paquetes

*Ejemplo*

*mx.edu.sep.dgtic.mejoredu.catalogos.integracion*

*mx.edu.sep.dgtic.mejoredu.seguridad.servicios*

*mx.edu.sep.dgtic.mejoredu.reportes.presentacion*

El paquete de integración puede estar conformado por los siguientes directorios:

|  |  |
| --- | --- |
| Paquete | Descripción |
| modelos | En este paquete contendrá las clases planas de transporte de información entre capas, es decir, los pojos del sistema. |
| Enum | Paquete que contendrá las clases tipo enums para categorizar información del sistema. |
| dao | Este paquete debe contener las interfaces que contiene la definición de los métodos que van a operar la persistencia. |
| impl | Este paquete debe contener las implementaciones de las interfaces dao. |
| hbm | Este paquete debe contener las clases persistentes con sus respectivos mapeos. |
| xml | Este paquete debe contener los archivos hbm.xml que contienen los queries nombrados a los que la aplicación haga referencia. |

Tabla 4 - Paquete de directorios

*Ejemplo*

*mx.edu.sep.dgtic.mejoredu.reportes.integracion.dao*

*mx.edu.sep.dgtic.mejoredu.reportes.integracion.dao.impl*

*mx.edu.sep.dgtic.mejoredu.seguridad.integracion.hbm*

*mx.edu.sep.dgtic.mejoredu.seguridad.integracion.hbm.xml*

## Nombrado.

* El nombre del paquete debe contener solo letras minúsculas, aun cuando éste se forme de más de una palabra, por ejemplo:

*mx.edu.sep.dgtic.mejoredu.atencedentesginecologicos.presentacion*

* No debe incluir ningún espacio, carácter especial, acento o letras inexistentes en inglés (Ej.: ñ).

## Clases e Interfaces.

## Nombrado.

* Cada palabra del nombre de una clase debe empezar con mayúscula
* El nombre de las clases y las interfaces debe ser un mnemónico que represente el uso o contenido de las mismas.
* Se deben utilizar sustantivos en singular.
* No debe incluir ningún espacio, carácter especial, acento o letras inexistentes en inglés (Ej.: ñ).
* Se debe acortar los nombres de las clases de modo que un par de palabras sea lo suficientemente descriptivo evitando la repetición de frases relacionadas al negocio, y que se infieren por el módulo al que pertenece la clase, por ejemplo, si existe un módulo llamado Climaterio y Menopausia, el nombre de sus clases deberá evitar mencionar “Climaterio y Menopausia” en cada una de ellas, así como en sus métodos.

Por ejemplo, si se tuviera una clase que será empaquetada dentro de *mx.edu.sep.dgtic.mejoredu.evaluacion.servicios* podría ser nombrada como sigue:

*PlanSubsecuenteService //Correcto*

*PlanSubsecuenteSeguridadDePlanificacionService //incorrecto*

* No deben utilizarse abreviaciones a menos que sean de uso común para el negocio (por ejemplo: Sep, Dgtic, Curp, Rfc)
* Adicionalmente se debe usar prefijos y sufijos con base en la naturaleza o representación de la clase o interface:

|  |  |
| --- | --- |
| Tipo | Nomenclatura |
| Clase abstracta | Usar la palabra “*Abstract*” como prefijo |
| Clase base | Usar la palabra “*Base*” como Sufijo |
| Clases de excepción | Usar la palabra “*Exception*” como sufijo |
| Clases de utilería | Usar la palabra “Util” como sufijo |
| Clases de servicio | Usar la palabra “Service” como sufijo |
| Clases de acceso a datos | Usar la palabra “*Dao*” como sufijo |
| Data transfer object | Usar la palabra “*Dto*” como sufijo |
| Factory | Usar la palabra “*Factory*” como sufijo |
| Definición de constantes | Usar la palabra “*Constantes*” como prefijo |
| Implementación de interfaces | Usar la palabra “Impl” como sufijo |
| Clases persistentes de hibernate (POJO) | Se usará únicamente el nombre correspondiente a la entidad que representan. |
| Clases tipo llave primaria de Hibernate | Se usará el nombre de la entidad de la cual es llave primaria más el sufijo “ID”, podrán nombrarse conforme al nombrado de la columna que corresponda a nivel base de datos, con el fin de aumentar la adaptación y comprensión del código. |
| VO | Se usará el nombre que describe la naturaleza de su información, conocido como Valor del Objeto (Value Object). |
| Controller | Clase encargada para publicar servicios REST |
| Clases de Enumeración | Se usará la palabra “Enum” como prefijo |
| Clases de prueba | Se usará la palabra “Test” como sufijo |

## Variables.

## Nombrado.

* Los identificadores de las variables deben iniciar con letras minúsculas; si existen palabras internas cada una de ellas deberá comenzar con una letra mayúscula y el resto de ellas escribirse con letras minúsculas (camelCase).
* Los identificadores de las variables no deben comenzar con \_ o $.
* Los identificadores de las variables deben ser cortos pero significativos y de tipo mnemónico, es decir, que sugiera al lector cuál es su uso.
* No deben utilizarse abreviaciones a menos que sean de uso común (por ejemplo: cve, id). Ejemplo:

*String cvePresupuestal = “0006060606060”;*

* Deben evitarse los identificadores de un solo carácter a menos que se trate de variables con poco uso y de tiempo de vida corto, tales como contadores. Ejemplo:

*for (int i = 0; i < list.size(); i++)*

*{*

*…*

* Los nombres que representan colecciones de objetos se escriben en plural. Esto mejora la legibilidad y devuelven al usuario una guía inmediata del tipo de variable y de las operaciones que se pueden realizar con el objeto. Ejemplo:

*Collection puntos;*

String[] args;

* En los nombres que representan colecciones de objetos no se debe anteponer algún prefijo lst o list
* No puede contener ningún carácter especial, espacio, etc.
* Se debe especificar una declaración por línea.

## Constantes.

* Los identificadores de las constantes deben cumplir con el criterio de nombrado de las variables salvo que en su totalidad deben ser escritas con letras mayúsculas y en caso que esté formada por más de una palabra cada una de ellas debe estar separada por el “guión bajo” (underscore) “\_”. Ejemplo:

*static final int VERDADERO = 0;*

*static final int FALSO = 1;*

*static final String SEPARADOR\_LINEA = Clase.getProperty("separador.linea");*

* Cualquier número que no tiene un significado por sí mismo, debe declararse como una constante. Ejemplo:

*private static final int NUM\_JUGADORES = 11;*

## Métodos.

* Los identificadores de los métodos deben escribirse bajo el estándar “acción-resultado” (verbo-sustantivo).
* Todas las letras deben ser minúsculas y en caso de contar con palabras internas la primera letra de cada una de ellas debe ser mayúscula y el resto minúsculas.
* Se debe acortar los nombres de los métodos de modo que un par de palabras sea lo suficientemente descriptivo evitando la repetición de frases relacionadas al negocio, y que se infieren por el módulo al que pertenece la clase, por ejemplo, si existe un módulo llamado Climaterio y Menopausia, el nombre de los métodos deberá evitar mencionar “Climaterio y Menopausia” en cada una de ellos.

Por ejemplo, si se tuviera una clase que será empaquetada dentro de *mx.edu.sep.dgtic.mejoredu.seguridad.servicios* la firma de sus métodos podrían nombrarse como sigue:

*registrarBitacoraSubsecuente //Correcto*

*registrarLaBitacoraUsuairioSeguridadMenopausia //incorrecto*

* Con la finalidad de aumentar la legibilidad y facilitar el mantenimiento futuro de la aplicación, se establece que el tamaño máximo de líneas por método no puede sobrepasar más de 80. Ejemplo:

*public void validarRFC(UsuarioDTO usuario);*

*public UsuarioDTO getUsuario(UsuarioId id);*

* Una clase no debe contener más de 20 métodos públicos.
* Los métodos de una clase no deben ser sólo métodos públicos.
* Cuando una expresión no entre en una línea, romperla de acuerdo con estos principios:

Romper después de una coma.

Romper antes de un operador.

Alinear la nueva línea con el comienzo de la expresión al mismo nivel de la línea anterior.

Ejemplos de cómo romper la llamada a un método:

*unMetodo(expresionLarga1, expresionLarga2, expresionLarga3,*

*expresionLarga4, expresionLarga5);*

*var = unMetodo1(expresionLarga1,*

*unMetodo2(expresionLarga2,*

*expresionLarga3));*

* Los métodos que realicen alguna operación de inserción, actualización o eliminación en la base de datos no deben indicar valor de retorno.
* Los métodos que realicen alguna consulta en la base de datos deben retornar siempre algún resultado.
* Los métodos de consulta en los DAOs deben recibir datos primitivos como parámetros en lugar de objetos de dominio completos, a menos que para la consulta se requieran la mayoría de los atributos del objeto de dominio.
* Los métodos de inserción deben comenzar con la palabra insertar, los de actualización o modificación con la palabra actualizar y los de búsqueda o consulta con la palabra consultar.

## Tamaño de la línea.

Cada línea de codificación no debe exceder de 80 caracteres. De no ser suficientes para albergar la sentencia completa, la alineación depende del tipo de información que se muestra en dicha línea:

|  |  |
| --- | --- |
| Tipo de información | Alineación |
| Texto | * Se continúa en el siguiente reglón alineado a la línea anterior. |
| Sentencias | * Sentencias donde se ven involucrados operadores y que exceden el número de caracteres permitido en una línea, se tratan con la siguiente política: El operador debe estar al final de la línea y el siguiente renglón comienza después del símbolo de asignación. Ejemplo:   *variable = nombreDeVariableGrandeQueHaceAlgo + "algún stringgggg" +*  *algunaOtraVariableOString;* |

Tabla 5 - Tamaño de la línea

## Formato general.

Cada archivo de código fuente debe dividirse en las siguientes secciones:

* Comentarios del archivo (encabezado)
* Declaración del paquete
* Sentencias import
* Comentarios de la clase
* Cuerpo de la clase o interface

## Comentarios

## Archivo (encabezado).

Se deben escribir al principio del archivo y contiene la siguiente información:

* Proyecto: SEP - (Nombre del proyecto)
* Archivo: Nombre del archivo con extensión
* Creado: (Fecha de creación. Formato: dd/mm/aaa)
* Derechos Reservados de Copia (c) – [Consultoria] / [Nomnbre de la Dependencia] – aaaa

Dejando una línea en blanco entre cada elemento.

Ejemplo

*/\*\**

*\* Proyecto: SEP - MEJOREDU*

*\**

*\* Archivo: NombreDeArchivo.java*

*\**

*\* Creado: 12/06/223*

*\**

*\* Derechos Reservados de Copia (c) - [Consultoria] / MEJOREDU - 2023*

*\*/*

## Clases.

Se deben escribir después de las sentencias import y antes del cuerpo de la clase o interface, y contiene la siguiente información:

* Descripción de la clase, responsabilidades, propósito, caso de uso, requerimiento funcional que lo originó, etc.
* Autor de la clase (usando el tag de JavaDoc @author)

Dejando una línea en blanco entre cada elemento. Ejemplo:

*/\**

*\* Implementación del patron J2EE “Service Locator”<br>*

*\**

*\* Este usa un mecanismo de cache interno para almacenar*

*\* y obtener objetos.<br>*

*\**

*\* @author <Nombre> <Apellido>*

*\*/*

## Métodos

Se debe escribir al inicio del método y contiene la siguiente información:

* Descripción del método
* Parámetros que recibe (en caso de que aplique)
* Parámetros de retorno (en caso de que aplique)
* Excepciones que puedan ocurrir y la causa
* Clave del caso de uso o casos de uso que se involucran

/\*\*

\* Adds two Complex objects and produces a third object that represents

\* their sum.

\* @param c1 A Complex object

\* @param c2 Another Complex object

\* @return A new Complex object that represents the sum of

\* <code>c1</code> and <code>c2</code>.

\* @exception java.lang.NullPointerException

\* If either argument is <code>null</code>.

\* @UserHistory (names = { "CUNENMENM72", "CUNEADRAU03" })

\*/

public static Complex add(Complex c1, Complex c2) {

return new Complex(c1.x + c2.x, c1.y + c2.y);

}

Los comentarios del método deben incluir un control de sus versiones y sus modificaciones, es decir se pueda ver una bitácora con los cambios que trae un nuevo redeploy. Lo que permitirá realizar verificaciones más rápidas.

## Sentencias import.

Se deben escribir después de la declaración del paquete de la clase y antes de los comentarios de la clase o interface. Existen dos consideraciones importantes en la declaración de las sentencias import:

1. Evitar el “ \* “ en el import. Importar todas las clases de un paquete conduce a un fuerte acoplamiento entre paquetes y puede conducirnos a problemas cuando una nueva versión de una librería introduce conflictos de nombres.
2. Eliminar imports no utilizados. Una declaración de import se considera no utilizada si:
   * No es referenciada en el archivo.
   * Es un duplicado de otro import.
   * La clase importada es del paquete java.lang.
   * La clase importada es del mismo paquete.

## Cuerpo de la clase o interface.

Debe estructurarse de la siguiente manera:

* Declaración de la clase o interface
* Atributos
* Métodos

## Atributos.

Se sugiere ordenar la declaración de las variables dependiendo de su modificador de acceso de la siguiente manera:

* *Public*
* *Ninguno (package)*
* *Protected*
* *Private*

## Métodos.

Los métodos se agruparán por funcionalidad y posteriormente por acceso. Los primeros métodos a definir deben ser los Constructores. El orden por modificador de acceso es el siguiente:

* *Public*
* *Protected*
* *Private*

Los nombres de los métodos se escribirán seguidos del paréntesis de la lista de parámetros, sin dejar espacio. Hay que tomar en cuenta los siguientes criterios para nombrar los métodos:

1. Para métodos que retornan algún objeto, se deben nombrar de acuerdo a lo que retornan. Ejemplo:

*public Collection getJugadores() {*

*…*

*}*

1. Los métodos que no devuelven valores (void) deben nombrarse de acuerdo a la función que realizan. Ejemplo:

*public void generaDigitoVerificador(String nss) {*

*…*

*}*

Los prefijos get/set deben usarse cuando se accede o modifica directamente un atributo. Este es un estándar establecido por Sun Microsystems. Ejemplo:

*empleado.getNombre();*

*empleado.setTelefono(nuevoTelefono);*

Nota: Para los atributos tipo boolean en lugar de utilizar get se utiliza el prefijo is. Ejemplo

*paciente.isTrabajadorSEP();*

Los bloques de código de los métodos se separarán por una línea.

El cuerpo de la función y/o procedimiento debe ir comprendido entre los caracteres “{ }” (llaves de apertura y cierre. La llave de apertura estará delimitada por un espacio en blanco después de la definición del inicio del bloque se código, excepto en aquellos casos en que se exceda el número de caracteres en una misma línea. La llave de cierre deberá permanecer sin ninguna otra definición sobre la misma línea.

Si un método llega a ser muy largo se vuelve difícil de entender. Es recomendable que los métodos no excedan las 80 líneas de código, sin contar comentarios y líneas en blanco. Si se da el caso de que un método exceda este número de líneas usualmente debería ser refactorizado en varios métodos individuales que se enfoquen sobre una tarea específica.

La complejidad ciclomática[[1]](#footnote-1) y [[2]](#footnote-2) aceptable para un método debe ser de entre 10 puntos, si esta métrica sale de la norma, se deberán aplicar medidas para mitigar el riesgo, tales como desarrollar pruebas unitarias de mayor cobertura o como se indica en el párrafo anterior, refactorizar el método.

Es conveniente ayudarse de un plugin que calcule las métricas del software tal como Metrics Plugin for Eclipse[[3]](#footnote-3).

## Parámetros.

Se deben usar nombres de variables como se describe en el punto 1.6 de este documento. Ejemplo:

*public Usuario(int Ide, String Nombre, String Direccion, String NumeroTelefonico) {*

*ide = Ide;*

*nombre = Nnombre;*

*direccion = Direccion;*

*numeroTelefonico = NumeroTelefonico;*

*}*

El máximo número de parámetros de un método o constructor no debe ser mayor a 10.

No deben existir espacios en blanco entre el identificador de una definición de un método, entre la definición de un constructor, entre una llamada a un método o entre una invocación de un constructor; y el paréntesis izquierdo de la lista de parámetros. Ejemplo:

*public void getNombre() {*

*…*

*}*

*public Usuario(int id, String name, String address, String phoneNumber) {*

*…*

*}*

*…*

*setDescripcion(“el string de la descripción”);*

*…*

## Sentencias de control del flujo.

1. Al declarar una sentencia, en la cual como parte de la sentencia y delante del identificador se exige un paréntesis de apertura “(“, se debe incluir un espacio en blanco antes del paréntesis de apertura. Ejemplo:

*if (condición) {*

*…*

1. El código incluido en las sentencias de control debe cumplir con el estándar de sangrado.

## Bloques.

Cuando nos referimos a bloques estamos haciendo referencia a:

* Cuerpo de clases o interfaces
* Cuerpo de métodos
* Bloques estáticos
* Sentencias while, do, for, if-else, try-catch, switch, finally.

1. En todo bloque debe de existir la llave que abre el bloque “{“ y la llave que cierra el bloque “}“. Aún y cuando la sentencia de control procese una sola instrucción.
2. La llave que abre el bloque “{“ debe estar separada por un espacio en blanco a partir de la definición del bloque y con la misma sangría que tiene la sentencia correspondiente. Ejemplo:

*if (condición) {*

*…*

1. La llave que cierra el bloque “}” debe estar ubicada en una línea nueva y con la misma sangría que tiene la sentencia correspondiente. Ejemplo:

*try {*

*...*

*}*

*catch (…*

1. No deben de existir bloques vacíos. Siempre debe de existir al menos una instrucción en un bloque. Ejemplo:

*…*

*catch (NumberFormatException nfe) {*

*log.info(“Info: ”, nfe);*

*}*

## Comentarios dentro del código.

Los comentarios que expliquen el flujo del programa se escribirán en una sola línea:

*// Usado en validación de perfil*

*resultado = getReferencias();*

1. Si el comentario no cabe en una sola línea, se creará un comentario en bloque (/\* \*/) exceptuando los siguientes casos:
   1. Las variables que por su definición no permitan la inferencia acerca de su uso se deberá escribir un comentario de uso y/o relación sobre dicha línea:

*// Máximo numero de caracteres aceptados*

*byte maxNumCarac = 6;*

* 1. Si el comentario se refiere sólo a una línea del programa este debe colocarse sobre dicha línea:

*// Asegurandose que siempre comience con 0*

*int indice = 0;*

* 1. Si el comentario se refiere sólo a un conjunto de líneas del programa este se debe colocar sobre dichas líneas y deberán cumplir con el estándar de sangrado:

*// Validado aquí debido a validación de GUI no existente*

*// Usamos Calendar en vez de Date para manipular fechas*

*Calendar calendar = calendar.getInstance();*

*if (mes < 12) {*

*calendar.set(Calendar.MONTH, mes);*

*}*

*return EXITO;*

## Sentencia for.

Existen diversas formas de declarar una sentencia for, a continuación, se listarán algunas de ellas donde se debe tener especial cuidado al escribirlas:

1. Inicializador vacío. Si el inicializador de la sentencia es vacío no se deben incluir espacios entre el paréntesis de apertura “(” y el carácter “;” que separa el inicializador de la condición. Ejemplo:

*for (; i < j; i++, j--) {*

*…*

*}*

## Sentencia while.

La sentencia while debe tener el siguiente formato:

*while (condicion) {*

*Instrucciones;*

*}*

## Sentencia do-while.

La sentencia while debe tener el siguiente formato:

*do {*

*Instrucciones;*

*} while (condición);*

Se debe evitar el uso de “break” y “continue” en el cuerpo de cualquier ciclo, al menos que la declaración mejore la legibilidad del código.

## Sentencia switch.

En esta sección se listarán los estándares de cómo se debe declarar una sentencia switch.

* Cláusula “default”. Es una buena práctica que todas las declaraciones switch incluyan un caso default, aún si el desarrollador está seguro que todos los casos posibles están cubiertos. Ejemplo:

*switch (elemento) {*

*case 0: …*

*break;*

*case 1: …*

*break;*

*…*

*default: …*

*}*

* Cláusula “default” después de todos los casos. Java permite poner el default en cualquier lugar dentro de la declaración del switch. Es más entendible el código si está viene después del último caso. Ejemplo:

*switch (elemento) {*

*case 0: …*

*break;*

*case 1: …*

*break;*

*…*

*case N: …*

*break;*

*default: …*

*}*

* Cláusula “case”. Cada cláusula case debe finalizar con una declaración “break”, “return”, “throw” o “continue”. Ejemplo:

*switch (elemento) {*

*case 0: …*

*break;*

*case 1: …*

*break;*

*…*

*case N: …*

*break;*

*default: …*

*}*

## Condicionales.

Al utilizar sentencias condicionales se deben tomar en cuenta los siguientes estándares de manera que sea más fácil comprender la lógica que representan.

* Evitar expresiones lógicas complejas (más de 2 expresiones a comparar) asignando valores lógicos a expresiones, de esa forma las sentencias serán más fáciles de leer. Ejemplo:

*boolean esTerminado = (elementoNum < 0) || (elementoNum > maxElemento);*

*boolean esRepetido = (elementoNum == ultimoElemento);*

*if (esTerminado || esRepetido) {*

*…*

*}*

* En la parte if de la sentencia se maneja el resultado normal de la condición y la excepción se maneja en la parte else. Esto asegura que las excepciones no alteran el curso normal de ejecución.
* Se deben utilizar operadores lógicos de evaluación de corto-circuito.
* Cuando la operación a evaluar sea “y lógico” (&&) poner la expresión que potencialmente evalúe a false en el extremo izquierdo.
* Cuando la operación a evaluar sea “o lógico” (||) poner la expresión que potencialmente evalúe a true en el extremo izquierdo.
* Anidaciones de bloques “if-else”. La profundidad máxima de declaraciones anidadas de bloques if-else, está restringida a 3.

## Espacios en blanco en el código.

En este apartado se mostrará la forma correcta de incluir espacios en blanco en algunas partes del código.

* No se deben incluir espacios en blanco después de un token, entre estos tokens están los símbolos ‘~’, ‘--‘, ‘.’, ‘++’, ‘!’, ‘+’, ‘-‘ e inicialización de arrays. Ejemplo:

*if (condición) {*

*…*

*i++;*

*…*

*}*

* No deben de existir caracteres tabs (‘\t’) en el código fuente. Si se usa un ambiente de desarrollo distribuido, cuando en CVS se realiza un commit y los archivos llevan este carácter, se hace casi imposible de leer.
* Debe existir un espacio en blanco después de los siguientes tokens: ‘,’, ‘;’ y un TYPECAST (cast de objetos). Ejemplo:

*(String) it.next();*

* Deben estar rodeados de espacios en blanco los siguientes tokens: ‘=’, ‘&’, ‘&=’, ‘|’, ‘|=’, ‘^’, ‘:’ (parte del operador condicional “? :”), ‘/’, ‘/=’, ‘==’, ‘>=’, ‘<=’, ‘&&’, ‘finally keyword’, ‘!=’, ‘%=’, ‘+=’, ‘<<’, ‘\*’, ‘\*=’, etc. Ejemplo:

*if (i <= VALOR\_ESTATICO) {*

*i++;*

*…*

*}*

* No deben de existir espacios después del paréntesis izquierdo y antes del paréntesis derecho en las llamadas a constructores, llamadas a métodos, llamadas al constructor ‘super’ de una clase padre, en TYPECAST, y en general en las declaraciones que incluyan paréntesis izquierdo ‘(‘ y paréntesis derecho ‘)’. Ejemplo:

*if (condición) {*

*…*

*}*

## Mensajes de depuración.

* En caso de ser MEJOREDUsario incluir mensajes de depuración dentro del código, debe hacerse uso de un framework que estandarice el volcado de mensajes, como log4j, Apache-commons.

## Más consideraciones de código.

* Evitar asignaciones internas. Con excepción del ciclo ‘for’, todas las asignaciones deberían ocurrir en su propio nivel declarativo para facilitar la lectura. Ejemplo:

*String s = Integer.toString(i = 2);*

Este tipo de asignaciones dificultan ver todos los lugares donde la variable es modificada.

* Números mágicos. Un número mágico es una literal numérica que no está definida como una constante. Por default, -1, 0, 1 y 2 no son considerados números mágicos. Ejemplo:

*String cad2 = cad1.substring(5, 8);*

* Cláusula “throws”. Evitar declarar excepciones redundantes, como duplicadas, no checadas o subclases de otra excepción declarada.
* Simplificar los returns boléanos. Evitar programar declaraciones de return boléanos complicados. Ejemplo:

*if (valida()) {*

*return false;*

*}*

*else {*

*return true;*

*}*

Debería ser escrito:

*return !valida();*

* Anidaciones de bloques “try”. La profundidad máxima de declaraciones anidadas de bloques try, está restringida a 2.
* “Catch” ilegales. Cachar java.lang.Exception, java.lang.Error o java.lang.RuntimeException no es aceptable. A menudo se simplifica el cachado de Exception en un intento de manejar múltiples clases de excepción, esto desafortunadamente conduce al código que inadvertidamente cache NPE, OutOfMemoryErrors, etc.
* Cantidad de sentencias “return”. El número máximo de sentencias return está restringido a 3. El tener muchos puntos de retorno puede indicar que el código está intentando hacer demasiado y puede ser difícil de entender.
  + Al utilizar la sentencia return no hay que utilizar paréntesis a menos que el valor de retorno no sea obvio.
* Literales iguales. Evitar definir múltiples ocurrencias de la misma cadena literal dentro de un archivo. La duplicación de código hace el mantenimiento más difícil, una opción para corregir esto puede ser reemplazar las múltiples ocurrencias con una constante.
* Sentencias por línea. Se debe incluir una sola sentencia por línea de código.
* Uso del token “this”. El uso del token “this” está permitido para aquellos casos en que el código se vea confuso. Ejemplo:

*public Constructor(String nombre, String apePaterno, String apeMaterno) {*

*this.nombre = nombre;*

*this.apePaterno = apePaterno;*

*this.apeMaterno = apeMaterno;*

*}*

* Estilo de definición de “arreglos”. El estilo para definir arreglos en el código fuente, se debe realizar al estilo Java. Ejemplo:

*public static void main(String[] args) {*

*…*

*}*

* Uso de clase System. Evitar el uso de la clase System. Sentencias como las siguientes deben evitarse:

*System.out.println(“cadena de salida”);*

*System.exit();*

* Un Hashmap es muy parecido a Hashtable con la diferencia de que sus métodos no estan sincronizados (por lo tanto es más rápido) y además Hashmap acepta null como dato llave o valor. Si se trabaja en un ambiente multihilos o si se busca compatibilidad con versiones anteriores de Java se debe usar Hashtable, de lo contrario usar Hashmap.

## Pseudocódigo.

Se describe como un comentario de líneas múltiples y debe cumplir con el estándar de sangrado.

La información anterior se incluirá a discreción del programador siempre y cuando esta proporcione un mayor significado del por qué de la clase o el método referido.

## Reglas generales para los archivos “.java”.

1. Cada archivo debe contener solo una clase o interface pública.
2. Cuando haya clases asociadas a la clase pública, se pueden colocar en el mismo archivo de la clase pública.
   1. La clase pública debe ser la primera clase del archivo.
3. Nueva línea al final del archivo. Cualquier archivo fuente y archivo de texto en general deben terminar con un carácter de nueva línea, especialmente si se está usando un sistema SCM como CVS. CVS imprimirá un warning cuando encuentre un archivo que no termine con una nueva línea.
4. Longitud del archivo. Si un archivo fuente llega a ser muy largo se vuelve difícil de entender. Es recomendable que los archivos no excedan las 1500 líneas de código. Si se da el caso de que una clase exceda este número de líneas usualmente debería ser refactorizada en varias clases individuales que se enfoquen sobre una tarea específica.

# PERFORMANCE

* Creación de objetos. Evitar el uso de objetos complejos (con referencias a otros objetos) ya que ocupan mayor memoria y por consiguiente se requerirá mayor tiempo para su generación, esto provocaría que la memoria cache no sea utilizada eficiente y posiblemente sería necesario utilizar memoria virtual (disco en vez de RAM), por lo que se afectaría directamente el performance de la aplicación.
* Evitar crear objetos que serán usados solo para invocación de métodos.
* Se debe centralizar la creación de objetos en un método Factory, esto es, implementando el patrón de diseño Factory.

# MEJORES PRÁCTICAS DE HIBERNATE.

## Para cada clase que participa en una colección.

Para cada clase que participa en una colección que forme parte del framework de colecciones del lenguaje Java deberán sobrecargarse los métodos generales de la clase Object (métodos equals y hashCode) e implementar la interfaz Comparable (método compareTo).

Es preferible usar un framework para calcular estos valores que programar el código por uno mismo, considere el uso de apache-hubernate-panache.

## Como cargar datos relacionados.

Se recomienda fuertemente el uso de anotaciones de Entidades, sin embargo, cuando se hagan mapas de Hibernate preferir Lazy Loading / Lazy Fetching cuando se hagan asociaciones entre las partes de una relación.

## Anotaciones y mapas.

Usar anotaciones para definir el mapeo de objetos Hibernate, si por alguna razón no es posible usar estas anotaciones, usar mapas en xml, (.hbm.xml), aunque considere que el uso de estos mapas ya no se recomienda.

## Uso de memoria cache.

Considere el uso de una memoria cache para almacenar los resultados de Hibernate, esto mejorará el desempeño de la aplicación.

## Asociaciones.

Al crear asociaciones entre elementos, preferir el uso de asociaciones bidireccionales, esto facilitará los queries de la aplicación.

## Consultas.

Evite escribir querys en código duro, en su lugar escribir los querys en los archivos de mapas (map.xml, cuando aplique) o externalizar los Strings.

## Soluciones estándar.

Considere el uso de Hibernate Search en el caso en que necesite hacer consultas de texto completo.

# LINEAMIENTOS DE DISEÑO.

Considere el uso de diseño para conocer las definiciones, normativas y estándares de diseño de base de datos. Con la finalidad de apegarse al nombrado de los atributos en las clases tipo entidad, objetos de valor (VO) y pojos del sistema.

# LINEAMIENTOS DE MANEJO DE REPOSITORIO DE VERSIONADO DE CÓDIGO.

El mecanismo de versionado el proyecto MEJOREDU, practicará la estrategia de versionado basado en GIT Git es un sistema de control de versiones distribuido gratuito y de código abierto diseñado para manejar todo, desde pequeños hasta Proyectos muy grandes con rapidez y eficiencia.

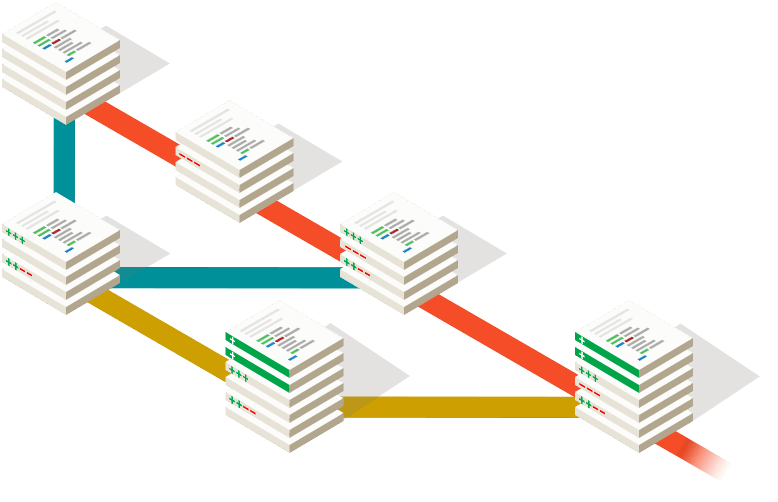


Ilustración 1 Mecanismo de versionado Git.

MEJOREDU, adopta el manejo de código por medio del repositorio github, en el repositorio con ruta web:

<https://github.com/ernestoMartinezEspinosa/sep-mejoredu>

Organizando la ramificación por ambiente y funcionalidad a desarrollar, como sigue:

|  |  |  |
| --- | --- | --- |
| Nombre de rama | Ambiente | Objetivo |
| master | Producción | Versión de código que será útil para desarrollar la solución tecnológica de ambiente productivo.  IMPORTANTE:  El merge será simpre de rama QA por el administrador del repositorio. |
| QA | QA | Versión de código que será útil para desarrollar la solución tecnológica de ambiente de control de calidad.  Rama creada a partir de rama padre master.  IMPORTANTE:  El merge será simpre de rama develop por el administrador del repositorio. |
| develop | Desarrollo | Versión de código que será útil para construir la solución tecnológica de ambiente de desarrollo. |
| FN-[Siglas]-[NombreDeFuncionalidad] | Desarrollo por cada programador | Versión de código que será útil para construir la solución tecnológica de ambiente desarrollo por cada programador  Donde:  [Siglas] Siglas del programador que está como responsable del cambio.  [NombreDeFuncionalidad] Es donde se especifica la funcionalidad a desarrollar.  Por ejemplo:  FN-EME-AccesoLoginAPI  Rama que es creada por Ernesto Martínez Espinosa y busca desarrollar el microservicio de Acceso al sistema (login). |

Tabla 6 - Organizando la ramificación por ambiente y funcionalidad

Todo cambio de contenido de rama obedece a sumar funcionalidad a la rama padre, en toda ocasión deberá realizarse por medio de la estrategia de combinación de ramas llamada PULL REQUEST.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Ilustración 3 - Cambio de contenido de rama

Las carpetas definidas para el proyecto podrán ser ajustadas a las necesidades de cada equipo de desarrollo, sin embargo, se sugiere fuertemente contener las siguientes carpetas de organización del repositorio que contiene código fuente, documentos de especificación ténica de las interfaces y lineamientos de diseño y desarrollo.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Ilustración 4 - Carpetas definidas del proyecto

| Nombre de carpeta | Objetivo |
| --- | --- |
| 01-PLANEACION | Documentos que siguen el plan de trabajo, de planeación y organización del proyecto. |
| 02-ANALISIS | Documentos requerimientos e insumos entregados por la organización cliente del proyecto. |
| 03-DISEÑO-ARQUITECTURA | Documentos que siguen el plan de trabajo, de plan de arquitectura y diseño de base de datos, así como todos los diagramas requeridos por la solución tecnológica para la organización del proyecto. |
| 04-IMPL | Contendrá la versión código de fuente de la solución tecnológica del proyecto, con todos los componentes tecnológicos como son:   * Proyecto fuente del portal WEB * Proyecto de cada uno de los microservicios   IMPORTANTE:  El desarrollador encontrará todo lo necesario para ambientar su entorno de trabajo. Referente a código fuente.  Interfaz de usuario gráfica, Aplicación  Descripción generada automáticamente |
| 05-PRUEBAS | Carpeta que contendrá los sets de datos y casos de prueba, así como los criterios de aceptación de las pruebas. |
| 06-CIERRE | Los documentos entregables que tienen el equipo de desarrollo por entregar serán contenidos en esta carpeta. |

Tabla 7 - Definición de versionamiento

# ESPECIFICACIÓN TÉCNICA DE LA INTERFAZ.

El documento que permite conocer técnicamente lo necesario para consumir los servicios Rest publicados por cada microservicio, será definido en el Documento Técnico de la Interfaz (ETI) un documento por cada microservicio de la solución tecnológica del proyecto.

A continuación, se anexa la plantilla del documento de especificación técnica de la interfaz.



1. http://www.literateprogramming.com/mccabe.pdf [↑](#footnote-ref-1)
2. Project Metrics Help, http://www.aivosto.com/project/help/pm-complexity.html [↑](#footnote-ref-2)
3. http://metrics.sourceforge.net/ [↑](#footnote-ref-3)