estandars de nomenclatura

**Índice**

[**I Estándares de documentación interna** 2](#_Toc432259879)

[**1.2 Bloques de documentación interna de la plantilla.** 2](#_Toc432259880)

[**1.3 Módulo de Documentación Bloquear** 3](#_Toc432259881)

[**II Controles** 4](#_Toc432259882)

[**2.2 Apertura vertical entre conceptos** 6](#_Toc432259883)

[**2.4 Densidad vertical** 8](#_Toc432259884)

[**2.5Funciones dependientes** 9](#_Toc432259885)

[**2.6Orden Vertical** 11](#_Toc432259886)

[**2.7Sangrado** 15](#_Toc432259887)

[**2.8 Romper el sangrado** 18](#_Toc432259888)

[**2.9.1 Es preferible desplegar y sangrar los ámbitos** 18](#_Toc432259889)

[**2.10Buenas prácticas de programación.** 19](#_Toc432259890)

[Tamaño de sangrado 19](#_Toc432259891)

[**2.11 Añadir llaves** 19](#_Toc432259892)

[**III Estándares en la codificación** 19](#_Toc432259893)

[**IV Estándares de diseño de Bases de datos** 20](#_Toc432259894)

# **I Estándares de documentación interna**

Tenemos que realizar correctamente la documentación interna para mejorar la legibilidad de un módulo de software. Para buena práctica de documentación interna tenemos que utilizar lineamientos generales de desarrollo de software, El SISPEG ha acordado que un archivo que contiene uno o más módulos de software o un archivo script de Shell debe tener un bloque de comentario en su comienzo que contenga la siguiente información básica:

El nombre del autor que creó el archivo

• La fecha de creación del archivo

• Grupo de desarrollo del autor (por ejemplo HSEB, HSMB)

• Descripción (descripción general de los efectos de los módulos)

El SISEPG ha llegado con un pequeño conjunto de elementos que se siente debe seguirse independientemente del lenguaje de programación que se utiliza.

Para mantener la adecuada producción de fácil lectura y mantener los programas la sangría debe utilizarse para:

• Hacer hincapié en el cuerpo de una sentencia de control, como un lazo o una instrucción select

• Hacer hincapié en el cuerpo de una sentencia condicional

• Hacer hincapié en un nuevo bloque de alcance

Un mínimo de 3 espacios se utilizarán para sangrar. En general, sangría por tres o cuatro espacios se considera que es adecuado. Una vez que el programador elige el número de espacios para sangrar es importante que se aplique de forma coherente esta cantidad.

# **1.2 Bloques de documentación interna de la plantilla.**

Bloques de documentación interna de la plantilla. Los ejemplos utilizan un comentario estilo C, que es también es aplicable a C ++ y Java. Cuando la documentación de otros idiomas, utilice el comentario estilo aplicable a ese idioma.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Autor:

\* Fecha de Creación del archivo:

\* Grupo de Desarrollo:

\* Descripción:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# **1.3 Módulo de Documentación Bloquear**

Definición: Este bloque de documentación precede a cada módulo en el archivo de origen.

(Los elementos opcionales se denotan mediante el uso de paréntesis.)

/ \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Nombre del Módulo:

\* Autor:

\* Fecha de Creación del módulo:

\* Descripción:

\* Los argumentos de llamadas:

\* Nombre Entrada / Salida Tipo Descripción

\* Archivos necesarios / Bases de datos:

\* Las rutinas no sistema llamado:

\* Nombre Descripción

\* Valor de retorno:

\* Tipo Descripción

\* Códigos de error / Excepciones:

\* Supuestos específicos del sistema operativo en su caso:

\* Variables locales:

\* Tipo Nombre Descripción

\* Variables globales utilizados:

\* Tipo Nombre Origen Descripción

\* Constante y Macro Suplentes:

\* Descripción Nombre del encabezado del archivo

\* Modificación Historia:

\* Fecha desarrollador Acción

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* /

# **II Controles**

Los controles que se agreguen al código deberán ser precedidos de un prefijo el cual haga referencia al mismo, así como un nombre relacionado a la función que se realizara con dicho control. Se utilizara el método de escritura CamelCase para el nombre que se le asigne.

Ejemplos:

**Label: lbl**

lblnombre, lblapellidoPaterno, lblcurp, lbl

**Button: btn**

btnabrir, btncerrar, btncargarArchivo, btnrespaldarBaseDatos

**Radio Button: rbtn**

rbtnpedido, rbtnsexo, rbtnpaisResidencia

**Toggle Button: tbtn**

tbtnnuevoPedido, tbtnactivar

**Checkbox: chk**

chknuevo, chkespañol, chkingles

**Choice Box: chb**

chbnuevo, chbanadir

**Text Field: txt**

txtnombre, txtapellidoPaterno

**Password Field: psw**

pswusuario, pswnuevo

**Scroll Bar: scrb**

scrbizquierda, scrbderecha

**List View: lsv**

lsvnueva, lsvcompras

**Table View: tbv**

tbvcompras, tbvventasNuevas

**Tree View: trv**

trvventasNuevas, trvcompras

**Tree Table View: trtv**

trtvcompras, trtvempleados

**Combo Box: cb**

cbpedidosNuevos, cbempleados

**Slider: sld**

sldhorizontal, sldvertical

**TableColumn: tbc**

tbcempleados, tbcnombre

**Hyperlink: hyl**

hylpcMax, hylescuela

**HTMLEditor: html**

htmlnuevo, htmldiario

# **2.2 Apertura vertical entre conceptos**

La práctica totalidad del código se lee de izquierda a derecha y de arriba a abajo. Cada línea representa una expresión o una cláusula, y cada grupo de líneas representa un pensamiento completo. Estos pensamientos deben separarse mediante líneas en blanco.

Hay líneas en blanco que separan la declaración del paquete, las importaciones y las funciones. Es una regla muy sencilla con un profundo efecto en el diseño visual del código. Cada línea en blanco es una pista visual que identifica un nuevo concepto independiente. Al avanzar por el listado, la vista se fija en la primera línea que aparece tras una línea en blanco.

Ejemplo:

**package** vista;

**import** javax.swing.\*;

**public** **class** SplashWindow **extends** javax.swing.JWindow{

**public** SplashWindow(java.awt.Frame f, **int** waitTime){

**super**(f);

JLabel l = **new** JLabel(**new** ImageIcon("src/images/pcmax.jpg"));

getContentPane().add(l,java.awt.BorderLayout.***CENTER***);

pack();

java.awt.Dimension screenSize = java.awt.Toolkit.*getDefaultToolkit*().getScreenSize();

java.awt.Dimension labelSize = l.getPreferredSize();

setLocation(screenSize.width/2 -(labelSize.width/2),screenSize.height/2 -(labelSize.width/2));

setVisible(**true**);

**final** **int** pause = waitTime;

**final** Runnable closeRunner = **new** Runnable(){

**public** **void** run(){

setVisible(**false**);

dispose();

}

};

**final** Runnable waitRunner = **new** Runnable(){

**public** **void** run(){

**try**{

Thread.*sleep*(pause);

SwingUtilities.*invokeAndWait*(closeRunner);

}

**catch**(Exception e){

e.printStackTrace();

}

}

};

setVisible(**true**);

Thread splashThread = **new** Thread(waitRunner,"SplashThresad");

splashThread.start();

}

}

Si eliminamos las líneas en blanco, se oscurece la legibilidad del código.

**package** vista;

**import** javax.swing.\*;

**public** **class** SplashWindow **extends** javax.swing.JWindow{

**public** SplashWindow(java.awt.Frame f, **int** waitTime){

**super**(f);

JLabel l = **new** JLabel(**new** ImageIcon("src/images/pcmax.jpg"));

getContentPane().add(l,java.awt.BorderLayout.***CENTER***);

pack();

java.awt.Dimension screenSize = java.awt.Toolkit.*getDefaultToolkit*().getScreenSize();

java.awt.Dimension labelSize = l.getPreferredSize();

setLocation(screenSize.width/2 -(labelSize.width/2),screenSize.height/2 -(labelSize.width/2));

setVisible(**true**);

**final** **int** pause = waitTime;

**final** Runnable closeRunner = **new** Runnable(){

**public** **void** run(){

setVisible(**false**);

dispose();

}

};

**final** Runnable waitRunner = **new** Runnable(){

**public** **void** run(){

**try**{

Thread.*sleep*(pause);

SwingUtilities.*invokeAndWait*(closeRunner);

}

**catch**(Exception e){

e.printStackTrace();

}

}

};

setVisible(**true**);

Thread splashThread = **new** Thread(waitRunner,"SplashThresad");

splashThread.start();

}

}

# **2.4 Densidad vertical**

Si la apertura separa los conceptos, la densidad vertical implica asociaciones. Por tanto, las líneas de código con una relación directa deben aparecer verticalmente densas.

Ejemplo:

/\*\*

\* nombre del paquete

\*\*/

**package** controller;

/\*\*

\* Import necesarios

\*\*/

**import** javafx.fxml.FXML;

**import** model.Reporte;

/\*\*

\* Nombre de la clase Reporte

\*\*/

**public** **class** ControladorReporte {

Reporte r;

/\*\*

\* Constructor de la clase Reporte

\*\*/

**public** ControladorReporte(){

r = **new** Reporte();

}

/\*\*

\* Metodo para cargar el reporte Actor

\*\*/

@FXML **public** **void** clickActor(){

r.cargarReporte("src/view/Reportes/actor.jrxml");

r.mostrarReporte();

}

/\*\*

\* Metodo para cargar el reporte Films

\*\*/

@FXML **public** **void** clickFilm(){

r.cargarReporte("src/view/Reportes/filmbueno.jrxml");

r.mostrarReporte();

}

}

El listado a continuación es mucho más fácil de leer. Lo apreciamos a simple vista, o al menos yo lo hago, sin tener que mover la cabeza ni la vista. El listado anterior nos obliga a forzar la vista y a mover la cabeza para alcanzar el mismo nivel de comprensión.

Ejemplo:

**package** controller;

**import** javafx.fxml.FXML;

**import** model.Reporte;

**public** **class** ControladorReporte {

Reporte r;

**public** ControladorReporte(){

r = **new** Reporte();

}

@FXML **public** **void** clickActor(){

r.cargarReporte("src/view/Reportes/actor.jrxml");

r.mostrarReporte();

}

@FXML **public** **void** clickFilm(){

r.cargarReporte("src/view/Reportes/filmbueno.jrxml");

r.mostrarReporte();

}

}

# **2.5Funciones dependientes**

Si una función invoca otra, deben estar verticalmente próximas, y la función de invocación debe estar por encima de la invocada siempre que sea posible. De este modo el programa fluye con normalidad. Si la convención se sigue de forma fiable, los lectores sabrán que las definiciones de función aparecen después de su uso. Fíjese en el fragmento de FitNesse del código siguiente.

La función superior invoca las situadas por debajo que, a su vez, invocan a las siguientes. Esto facilita la detección de las funciones invocadas y mejora considerablemente la legibilidad del módulo completo.

Ejemplo:

public class WikiPageResponder implements SecureResponder {

protected WikiPage page;

protected PageData pageData;

protected String pageTitle;

protected Request request;

protected PageCrawler crawler;

public Response makeResponse(FitNesseContext context, Request request)

throws Exception {

String pageName = getPageNameOrDefault(request, "FrontPage");

loadPage(pageName, context);

if (page == null)

return notFoundResponse(context, request);

else

return makePageResponse(context);

}

privare String getPageNameOrDefault (Request request, String defaultPagoName)

{

String pageName = request.getResource();

if (StringUtil.isBlank(pageName))

pageName = defaultPageName;

return pageName;

}

protected void loadPage(String resource, FitNesseContext context)

throws Exception {

WikiPagePath path = PathParser.parse(resource);

crawler = context.root.getPaqeCrawler();

crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());

page = crawler.getPage(context.root, path);

if (page != nulll )

pageData = page.getData();

}

private Response notFoundResponse(FitNesseContext context, Request request)

throws Exception {

return new NotFoundResponder() .makeResponse(context, request);

}

private SimpleResponse makePageResponse(FitNesseContext context)

throws Exception {

pageTitle = PathParser.render(crawler.getFullPath(page));

String html - makeHtml(context);

SimpleResponse response = new SimpleResponse();

response.setMaxAge(O);

response.setContent(html);

return response;

}

…

Además, este fragmento es un buen ejemplo de ubicación de constantes en un nivel correcto. La constante FrontPage se podría haber ocultado en la función geLPageNameOrDefault, pero eso habría ocultado una constante conocida y esperada en una función de nivel inferior de forma incorrecta. Es mejor pasar la constante desde un punto en el que tiene sentido a la posición en la que realmente se usa.

# **2.6Orden Vertical**

Por lo general, las dependencias de invocaciones de funciones deben apuntar hacia abajo. Es decir, la función invocada debe situarse por debajo de la que realice la invocación. Esto genera un agradable flujo en el código fuente, de los niveles superiores a los inferiores.

Como sucede en los artículos del periódico, esperamos que los conceptos más importantes aparezcan antes y que se expresen con la menor cantidad de detalles sobrantes. Esperamos que los detalles de nivel inferior sean los últimos. De este modo, podemos ojear los archivos de código y captar el mensaje en las primeras funciones sin necesidad de sumergirnos en los detalles.

Ejemplo:

package modelo;

import java.sql.Connection;

import java.sql.DriverManager;

import controlador.ControladorErrores;

public class Conexion {

private ControladorErrores ce;

private String bd;

private String usuario;

private String contrasenia;

private String puerto;

private String direccionip;

private String servidor;

private static Conexion instancia;

private Connection con;

/\*

\* Constructor sin parametros que permitira definir un dato por default de la clase

\*/

private Conexion(){

bd = "pcmax2.1";

usuario = "administrador";

contrasenia = "maki3001";

puerto = "5432";

direccionip = "127.0.0.1";

servidor = "jdbc:postgresql://";

ce= new ControladorErrores();

con= null;

}

/\*

\* Constructor con parametros que permita inicializar con valores personalizados

\*/

private Conexion(String bd, String usuario, String contrasenia, String puerto, String direccionip){

this.bd = bd;

this.usuario = usuario;

this.contrasenia = contrasenia;

this.puerto = puerto;

this.direccionip = direccionip;

this.servidor = "jdbc:postgresql://";

con= null;

}

/\*

\* Metodo para recuperar la instancia de la clase de conexion

\*/

public static Conexion getInstance(){

if(instancia == null)

{

instancia = new Conexion();

}

return instancia;

}

/\*

\* Método para conectar al servidor Postgresql

\*/

public String conectar(){

try {

//verifica que este el driver en el proyecto

Class.forName("org.postgresql.Driver");

//Establecemos conexion

servidor = "jdbc:postgresql://";

con = DriverManager.getConnection(servidor+direccionip+":"+puerto+"/"+bd, usuario, contrasenia);

System.out.println("Se hizo la conexión");

return "Conexion éxitosa";

} catch (Exception ex) {

// TODO: handle exception

ce.printlog(ex.getMessage(), this.getClass().toString());

return "No se establecio la conexión, Consulta a su administrador.";

}

}

/\*

\* Método para desconectar del servidor de Postgresql

\*/

public String desconectar(){

try {

//Cerrar la conexión

con.close();

System.out.println("Se ha desconectado del servidor");

return "Se ha desconectado del servidor";

} catch (Exception ex) {

// TODO: handle exception

ce.printlog(ex.getMessage(), this.getClass().toString());

return "La conexión está siendo ocupada. No se puede desconectar.";

}

}

/\*

\* metodo para recuperar la conexion

\*/

public Connection getConexion(){

return con;

}

# 

# **2.7Sangrado**

Un archivo de código es una jerarquía más que un contorno. Incluye información que pertenece a la totalidad del archivo, a sus clases individuales, a los métodos de las clases, a los bloques de los métodos y a los bloques de los bloques. Cada nivel de esta jerarquía es un ámbito en el que se pueden declarar nombres y en el que se interpretan declaraciones e instrucciones ejecutables.

Para que esta jerarquía de ámbitos sea visible, sangramos las líneas de código fuente de acuerdo a su posición en la jerarquía. Las instrucciones al nivel del archivo, como las declaraciones de clases, no se sangran. Los métodos de una clase se sangran un nivel a la derecha de la clase. Las implementaciones de dichos métodos se implementan un nivel a la derecha de la declaración de los métodos. Las implementaciones de bloques se implementan un nivel a la derecha de su bloque contenedor y así sucesivamente.

Los programadores dependen de este sistema de sangrado. Alinean visualmente las líneas a la izquierda para ver el ámbito al que pertenece. De este modo pueden acceder rápidamente a los ámbitos, como por ejemplo a implementaciones de instrucciones if o while, que no son relevantes para la situación actual. Buscan en la izquierda nuevas declaraciones de métodos, variables e incluso clases. Sin el sangrado, los programas serian prácticamente ilegibles.

Ejemplo sin aplicar sangrado:

package view;

import javafx.application.Application;

import javafx.fxml.FXML;

import javafx.fxml.FXMLLoader;

import javafx.stage.Stage;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.scene.layout.BorderPane;

public class Main extends Application {

@Override

public void start(Stage primaryStage) {

try {

Parent root = FXMLLoader.load(getClass().getResource("fxml/reporte.fxml"));

Scene scene = new Scene(root);

//scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());

primaryStage.setScene(scene);

primaryStage.show();

} catch(Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

launch(args);

}

}

Ejemplo aplicando sangrado:

package view;

import javafx.application.Application;

import javafx.fxml.FXML;

import javafx.fxml.FXMLLoader;

import javafx.stage.Stage;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.scene.layout.BorderPane;

public class Main extends Application {

@Override

public void start(Stage primaryStage) {

try {

Parent root = FXMLLoader.load(getClass().getResource("fxml/reporte.fxml"));

Scene scene = new Scene(root);

//scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());

primaryStage.setScene(scene);

primaryStage.show();

} catch(Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

launch(args);

}

}

# **2.8 Romper el sangrado**

En ocasiones tenemos la tentación de romper la regla de sangrado con instrucciones if breves, bucles while breves o funciones breves. Siempre que se sucumbe a esta tentación, se acaba por volver a aplicar el sangrado. Por ello, es recomendable evitar replegar ámbitos a una línea, como en este ejemplo:

Ejemplo:

public class Main extends Application {

@Override

public void start(Stage primaryStage) {

try {

Parent root = FXMLLoader.load(getClass().getResource("fxml/reporte.fxml"));

Scene scene = new Scene(root);

//scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());

primaryStage.setScene(scene);

primaryStage.show();

} catch(Exception e) {e.printStackTrace();}

}

public static void main(String[] args) {launch(args);}

}

# **2.9.1 Es preferible desplegar y sangrar los ámbitos**

Ejemplo:

public class Main extends Application {

@Override

public void start(Stage primaryStage) {

try {

Parent root = FXMLLoader.load(getClass().getResource("fxml/reporte.fxml"));

Scene scene = new Scene(root);

//scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());

primaryStage.setScene(scene);

primaryStage.show();

} catch(Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

launch(args);

}

}

# **2.10Buenas prácticas de programación.**

# Tamaño de sangrado

El tamaño de sangrado que se utilizara y se recomienda es el dado por el tabulador el cual es práctico y ofrece el espacio necesario para distinguir la diferencia entre sección.

# **2.11 Añadir llaves**

Cada vez que introduzca una llave izquierda de apertura, {, en su código, introduzca inmediatamente la llave derecha de cierre, }, y vuelva a colocar el indicador entre las llaves para comenzar a introducir el cuerpo del programa. Esto ayuda a evitar que falten llaves

# **III Estándares en la codificación**

Al comenzar un proyecto de software, es muy importante establecer un estándar de codificación para asegurarse de que todos los programadores del proyecto trabajen de forma coordinada. Cuando el proyecto de software incorpore código fuente previo, o bien cuando realice el mantenimiento de un sistema de software creado anteriormente, el estándar de codificación debería establecer cómo operar con la base de código existente

**Clases**.- El nombre de las clases es recomendable que empiece con la primera letra mayúscula y si se compone de más palabras, se deben de seguir utilizando la primera letra de cada palabra en mayúscula a este método se le conoce como Pascal Case, ejemplo:

Public class HolaMundo{

}

**Métodos**.- El nombre de los métodos es recomendable iniciar la primera letra en minúscula y a partir de la segunda palabra la primera letra en mayúscula, este método es conocido como camel case, ejemplo:

Public void guardarCliente(){

}

**Constantes**.- Las constantes deben ser escritos la primera letra en minuscula y las demas al inicio de cada palabra en mayúsculas separadas por guiones bajos. Los nombres de constantes también pueden contener dígitos en su caso, pero no como el primer carácter, ejemplo:

static final nombre\_Constante = valor;

**Variables**.- Los nombres de variable no deben comenzar con guión bajo ( \_ ) o el signo de dólar ( $ ) caracteres, aunque ambos se admiten. La elección de un nombre de variable debe estardiseñada para indicar al observador casual la intención de su uso, ejemplo:

Public String nombreVariable;

**Funciones**.- Las funciones en base de datos también son parte del código del sistema, para esto es recomendable que los nombres de las funciones inicien con un prefijo de función + el nombre de la función, ejemplo:

# **IV Estándares de diseño de Bases de datos**

Uno de los instrumentos que facilitan la tarea a lo largo del ciclo de vida del desarrollo de un software para asegurar la calidad del mismo, es la adopción de estándares de diseño de bases de datos.

El uso de estos estándares tiene innumerables ventajas, entre ellas:

• Asegurar la legibilidad del modelo de datos, inclusive para personas que no están   
relacionadas con el ambiente informático, en etapas de análisis y diseño;

• Facilitar la portabilidad entre motores de bases de datos, plataformas y aplicaciones;

• Facilitar la tarea de los programadores en el desarrollo de los sistemas.

Es por esto que la codificación de las tablas de las bases de datos a desarrollar debe cumplir   
ciertos requisitos, detallados en el presente documento. Estos requisitos pueden aplicarse a cualquier motor de bases de datos.

Brevemente se resume en los siguientes puntos:

Reglas generales:

• El nombre de la base de datos, debe comenzar con el prefijo BD seguido del nombre de la base de datos comenzando con la primera letra mayúscula.

Ejemplo: BDSudesegProduccion

• Los nombres de las tablas deben comenzar con el prefijo T seguido del nombre de la tabla con la primera letra en mayúscula, y en caso de ser nombre compuesto cada palabra debe empezar con mayúscula.

Ejemplo: TInformacionLaboral.

• Los nombres de los campos deben especificarse bajo el estándar lowerCamelCase. Este estándar especifica escribir las palabras compuestas eliminando los espacios y poniendo en mayúscula la primera letra de cada palabra. En caso de ser nombres compuestos se utilizara la variante lowerCamelCase, es decir, la primera letra del nombre en minúscula.

Ejemplo: camelCase: idPersona

Ejemplo: lowerCamelCase: idSolicitudesPendientes

• Únicamente se utilizarán caracteres alfabéticos, salvo que por la naturaleza del nombre se necesiten dígitos numéricos. Se prohíbe el uso de caracteres de puntuación o símbolos.

Ejemplo: localidadesCenso2003.

• Las letras acentuadas se reemplazarán con las equivalentes no acentuadas, y en lugar de   
la letra eñe (ñ) se utilizará (ni).

Ejemplos: anioExpediente, montoSenia.

• El nombre elegido debe ser lo más descriptivo posible, evitando términos ambiguos o que   
se presten a distintas interpretaciones.

Ejemplo: tiposMunicipios => categoriasMunicipios.

• El nombre no debe abreviarse, salvo que por necesidad específica deban especificarse más de una palabra en el mismo.

Ejemplo: ido => idOrganismo, freg => fechaRegistro

• Agregar comentarios a las bases de datos y los campos, sobre todo a los booleanos.

• Los nombres deben especificarse en plural, y de acuerdo a las reglas generales.

Ejemplos: departamentos, facturas, monedas.

• En el caso de tablas que se relacionan específicamente con otra tabla (ej. tablas tipo,   
nomencladores, entidades débiles), esta relación debe quedar expresada en el nombre.

Ejemplos: domiciliosPersonas, categoriasMunicipios.

• Las tablas de relación (objetos asociativos, representan relaciones de N a M) deben   
nombrarse utilizando los nombres de las tablas intervinientes, siguiendo un orden lógico   
de frase.

Ejemplos: localidadesMunicipios, facturasNotas

**4.1 Tablas**

Campos clave (Identificadores de tabla)

• Toda tabla debe poseer uno o más campos clave.

• Toda relación entre tablas debe implementarse mediante constraints (claves foráneas) con integridad referencial, de acuerdo al motor de base de datos utilizado.

• La integridad referencial deberá actualizar en cascada en todos los casos, y restringir el   
borrado salvo para las entidades débiles.

Ejemplos: no se podrá eliminar un registro de la tabla localidades que tenga ocurrencias en otras tablas; para este caso deberá implementarse el borrado lógico. Por el contrario, sí podrá habilitarse el borrado en cascada si la relación fuera entre las tablas facturas y renglonesFactura.

• Los campos clave deben ubicarse al inicio de la definición de la tabla (deben ser los   
primeros).

• El nombre del campo clave debe estar compuesto por “id” + nombre de la tabla en   
singular (para claves no compuestas). Dependiendo de la naturaleza de la entidad, el   
nombre de la tabla a usar es el de la misma tabla, o el de la relacionada.

Ejemplos: tabla localidades => idLocalidad.

• Las claves compuestas sólo deben utilizarse en casos específicos, por ejemplo, tablas de   
relación o entidades débiles. Si una tabla X con clave compuesta necesita ser referenciada   
desde otra tabla Y, deberá generarse un campo clave en X al inicio de la misma como   
“idX”, y generar un índice único en los campos que la identificaban.

• Todo campo que represente un nombre o descripción, se colocará inmediatamente   
después de los campos clave, y se nombrará como a la tabla a la que pertenece, en   
singular.

Ejemplos:

tabla localidades => idLocalidad, localidad.

tabla   
sucursalesEmpresas   
=>   
idEmpresa,   
idSucursal,   
sucursal   
  
• Algunos campos que representan datos, de acuerdo a su representación conceptual en el   
ámbito del negocio, deberán prefijarse de la siguiente manera:

– – Fechas: fecha (ejemplo: Fecha de inscripción => fechaInscripcion)

–

• Números: num (ejemplo: Número de factura => numeroFactura)

Códigos: codigo (ejemplo: Código de producto: codigoProducto)

Los campos booleanos deberán nombrarse de acuerdo al estado correspondiente al valor   
1/Verdadero/True de los mismos.

Ejemplos: autorizado, oculto, vigente.

• Los campos de relación (foreign keys, claves foráneas) deben nombrarse de la misma   
manera que los campos clave (usando el nombre de la tabla a la que hacen referencia).   
Ejemplos: tabla personas => idTipoDocumento, idEstadoCivil

**4.2 Procedimientos almacenados (store procedure)**

Un procedimiento es un programa dentro de la base de datos que ejecuta una acción o conjunto de acciones específicas.

Un procedimiento tiene un nombre, un conjunto de parámetros (opcional) y un bloque de código.

Ventajas de usar SP

Compilación: La primera vez que se invoca un SP, el motor lo compila y a partir de ahí, se sigue usando la versión compilada del mismo, hasta que se modifique.

Esto significa que se tendrá un mejor rendimiento que las consultas directas que usan cadenas con las instrucciones, que se compilan cada vez que se invocan.

Automatización: si tenemos un conjunto de instrucciones SQL, las cuales queremos ejecutar de manera ordenada, un SP es la mejor manera de hacerlo.

Administración: cuando realizamos aplicaciones con un gran número de líneas de código, y queremos hacer cambios, solo implica modificar un SP y no toda la aplicación, lo que significa solo cambiamos los SP en el servidor y no tenemos que actualizar la aplicación en todos los equipos cliente.

Seguridad: una parte importante es que, a los usuarios de nuestra aplicación, solo les proporcionamos los permisos para ejecutar los procedimientos almacenados y no el acceso a todos los objetos de la base.

Programabilidad: Los SP admiten el uso de variables y estructuras de control como IF, Bucles, Case, etc. Además del manejo de transacción y permite controlar excepciones.

Para crear un procedimiento almacenado debemos emplear la sentencia

CREATE OR REPLACE FUNCTION.

CREATE OR REPLACE FUNCTION <nombre\_procedure> (param1

<tipo>, ...)

-- Sentencias del procedure

Ejemplo:

El siguiente ejemplo muestra un procedimiento almacenado, denominado spu\_addCliente que inserta un registro en la tabla "CLIENTES"

CREATE OR REPLACE FUNCTION spu\_addCliente(nombre varchar(100),

apellido1 varchar(100),apellido2 varchar(100),

nifCif varchar(20),fxNaciento timestamp)

RETURNS VOID AS $$

BEGIN

INSERT INTO CLIENTES

(nombre, apellido1, apellido2, nifcif, fxNaciento) VALUES

(nombre, apellido1, apellido2, nifCif, fxNaciento);

END;

$$

Language 'plpgsql';

El siguiente ejemplo muestra la ejecución del procedimiento almacenado anterior.

SELECT spu\_addCliente('Pedro', 'Herrarte', 'Sanchez',

'00000002323', ‘2011-02-02’)

Ejemplo:

El siguiente ejemplo muestra un procedimiento almacenado que devuelve valores.

CREATE or replace FUNCTION spu\_EstaEnNumerosRojos(numCuenta

varchar(20))

RETURNS numeric AS $SALDO$

BEGIN

IF (SELECT state FROM expedientes

WHERE code = numCuenta) = '4' then

RETURN 1;

ELSE

RETURN 0;

END IF;

END;

$SALDO$

Language 'plpgsql';

Ejemplo:

El siguiente ejemplo muestra como ejecutar el procedure y obtener el valor devuelto.

 select spu\_EstaEnNumerosRojos('1');