|  |  |  |
| --- | --- | --- |
| Documento de Estándares | 8 de septiembre  2009 | |
|  | | Construcción de software |

## Universidad de Talca, Facultad de ingeniería

**Indice**

**Diccionario…………………………………………………………………………………………………………………………2**

**Estándar de interfaz…………………………………………………………………………………………………………..3**

**Estructura de la interfaz………………………..………………………..…………………………………….4**

**Descripción de los componentes de la interfaz………………………..…………………………….5**

**Opciones de módulo………………………..………………………..………………………..………………..6**

**Datos………………………..………………………..………………………..………………………..………………7**

**Botones principales de acceso a los módulos………………………..………………………………..9**

**Estándar conexión………………………..………………………..………………………..………………………………..11**

**Estándar SQL………………………..………………………..………………………..………………………..………………20**

**Modelo relacional………………………..………………………..………………………..………………………………..23**

**Modelo base de datos………………………..………………………..………………………..………………………….25**

**Estándar de documentación de código actionScript………………………..……………………….…………26**

**Estándar de documentación de código java………….………………………..……………………….…………28**

**Estándar del usuario basado en IEEEStd 1063-2001…………………………………………………………...38**

**Testing………………………………………………………………………………………………………………………………..40**

**Referencias………………………………………………………………………………………………………………………..49**

Diccionario

**Veterinaria: Se** refiere al establecimiento del doctor Bonati, en el que se ofrecen los servicios de: Peluquería, Hotelería, Pet-Shop y Clínico (Pabellón, Policlínico).  
  
**Cliente:** Es una persona que usa un servicio de la clínica veterinaria y que no necesariamente tiene una o mas mascotas.  
  
**Mascota:** Es un animal, que dispone de datos tales, como nombre, fecha de nacimiento y esta asociado al registro de un cliente.  
  
**Ficha:** Es un documento digital que posee cada mascota, en el que aparecen todos los registros de atenciones realizadas y servicios realizados. Ordenados por fecha y por servicio. Esta ficha esta asociada a un cliente, de ahí que se distingue una ficha para cada mascota que tenga un cliente.  
  
**Usuario de Sistema:** Es un trabajador de la clínica, que usa el sistema y que cuenta con una cuenta de usuario.  
  
**Registrar:** Acción de  guardar un conjunto de datos en el sistema.  
 **Editar:** Acción que modifica uno o varios datos para posteriormente guardar los datos.  
  
**Borrar:** Registro que se borra temporalmente del sistema.  
  
**Purgar:** Registro que se borra completamente de la base de datos o sistema.  
  
**Visualizar:** Acción en la que se representan de alguna forma específica los datos en la pantalla o archivo.  
  
**Mantener:** Acción de borrar, registrar y editas uno o varios datos.  
  
**ID Mascota**: Numero identificador único, generado por el sistema, para identificar de manera única a una mascota en el sistema.  
  
**RUT Cliente**: Referencia única hacia un cliente.  
  
**Cruza**: Se refiere al apareamiento de animales.

**Listar**: Acción de mostrar los datos al usuario y que no están necesariamente ordenados.  
  
**Eliminar:** Datos que no se muestran en el programa pero que si están en la base de datos.

**Policlínico:** Lugar donde la mascota se hospeda después de haber sido intervenida quirúrgicamente.

**Pabellón**: Lugar donde se hacen las intervenciones quirúrgicas a las mascotas

Estándar de Interfaz

1.- Luchar por la consistencia o coherencia:Este punto está enfocado a las diferentes terminologías y estructuras de pantallas para situaciones o escenarios semejantes, estos deben utilizarse en anuncios, menús y pantallas de ayuda, esto enfocado a los nombres de botones, utilizar los mismos nombres para la misma acción, distribución de las pantallas en cuanto a contenidos.

2.- Utilizar frecuentemente accesos directos:Esto consiste en presentar botones o link de acceso rápido para acceder a las diferentes prestaciones que tiene el sistema, permitir al usuario utilizar estos accesos se fundamenta en que en la medida que el tiempo en que usa el software aumenta, también lo hace la necesidad de este por reducir los pasos o acciones para realizar acceder a cierto servicio, disminuyendo el tiempo de respuesta y aumentando el ritmo de la interacción. En este punto se pueden utilizar acrónimos y abreviaturas, teclas de función, comandos ocultos, entre otros.

3.- Ofrece comentarios informativos:Por cada operador de acción, debe haber algún sistema de retroalimentación. Para acciones frecuentes y de menor uso, la respuesta puede ser modesta, mientras que para los poco frecuentes y las principales acciones, tales como errores graves del sistema, que puedan producir perdidas de información, la respuesta debería ser más sustancial.

4.- Diseño de diálogo para producir la clausura:Este punto está enfocado a la retroalimentación informativa, que se produzca en la conclusión de un grupo de acciones, esto da a los usuarios la satisfacción de logro, una sensación de alivio, esto en parte indica al operador que debe continuar con las acciones, o que se debe preparar para realizarlas.

5.- Ofrece una manipulación de errores simples:En la medida de lo posible, diseñar el sistema para que el usuario no ocasione un grave error. Si aparece un error, el sistema debería ser capaz de detectar el error y ofrecer de manera sencilla y comprensible una manera para identificar el dicho error.

6.- Permitir un fácil retroceso de las acciones**:** Esta característica alivia la ansiedad, ya que el usuario sabe que los errores se pueden deshacer, esto lleva a que el usuario tenga menos miedo al explorar el software y las diferentes opciones desconocidas para el usuario. Las unidades de reversibilidad pueden ser una sola acción, una entrada de datos, o un grupo de acciones.

7.- Apoyo interno a un enfoque de control total**:** Los usuarios experimentados desean el sentido de que están a cargo del sistema y que el sistema responde a sus acciones. Para esto es necesario diseñar el sistema para que los usuarios inicien las acciones en lugar de que inicien las respuestas. Y que este pueda ingresar y salir de diversas opciones de manera fácil, sin pasar por procesos intermedios.

8.- Reducir la carga de la memoria a corto plazo:Este punto presenta vital importancia debido a las limitaciones de recursos humanos, en cuanto al procesamiento de la información en la memoria a corto plazo, esto exige que se muestren de manera sencilla, y que el software no ponga a prueba la memoria del usuario, si no que este le facilite al usuario el recordar la información, mediante un escenario de interacción.

## Estructura de la interfaz



**Imagen 1**

En la **imagen 1** se puede apreciar la interfaz que se desarrollo, la cual no necesariamente será la definitiva en cuanto a la forma de presentar los datos o las entradas de datos, más bien esto está enfocado a los espacios que ocupara la información que se presenta en este software, esto se puede apreciar claramente con los paneles, pudiendo visualizar claramente cómo estará dividido el programa o la interfaz que interactuara con el usuario.



**Imagen 2**

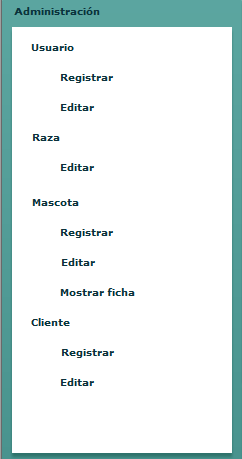
En **la imagen 2** se puede observar cómo se debería estructurar el software, y la forma en que deberían presentarse los datos y los menús correspondientes a cada sub-modulo, esto al momento de seleccionar dicho modulo. Es por esto que se ha definido la mayor parte de la forma de estructurar la visualización de datos, el ingreso de datos, y en qué lugares deberán aparecer partes importantes de una interfaz como imágenes, botones, etc.

La forma de utilizar el software, explicado rápidamente, sería hacer login con la cuenta de usuario correspondiente. Luego el deberá seleccionar un modulo del sistema, que son los que se encuentran en la barra horizontal, después de esto aparecerá en el panel izquierdo Opciones del modulo, los diferentes servicios que este modulo presentara, se selecciona un servicio y en el panel de datos, aparecen los datos a ingresar o las diferentes visualizaciones. Y a continuación se seleccionara el botón correspondiente a la acción que se desea realizar del panel de Opciones de selección.

## Descripción de los componentes de la interfaz

En el software se definieron sectores que no deberían modificarse al momento de ingresar a cada sub-modulo, para lograr una consistencia en el software y evitar que al abrir cada sub-modulo aparezca una estructura diferente de los datos. Es por esto que se definió que solo deberán cambiar sectores internos de los paneles: Opciones del módulo, Datos, Opciones de selección.

## Opciones de módulo



**Imagen 3**

En este panel, se encontraran los servicios o procesos principales que presenta el modulo que fue seleccionado, los cuales si tienen sub-categorías, deberían aparecer indentados, como se muestra en la **imagen 3**, esto servirá de gran forma para retroceder rápidamente a secciones anteriores o servicios anteriores, ya que mientras se esté en el mismo modulo, estarán visibles las opciones anteriores y las de otros servicios relacionados con el modulo, por lo que si el usuario desea ingresar a otro servicio de este modulo, lo podrá hacer de manera rápida. Esto cubre en gran parte la regla número 2 de los anteriormente mencionados, el de presentar accesos directos. De igual forma se pueden definir teclas o combinaciones de teclas que permitan acceder rápidamente a cada servicio.

Para este panel el porte será definido por cada realizador, y a la vez deberá ajustar el panel de datos para que calcen en la pantalla y no queden espacios vacios.

Es importante destacar que la forma en que se mostraran los datos es la siguiente: El titulo del panel debe ser el del modulo al cual pertenece, y los servicios que se prestan se deben dividir, agrupándolos en sustantivos si se puede decir, por ejemplo:

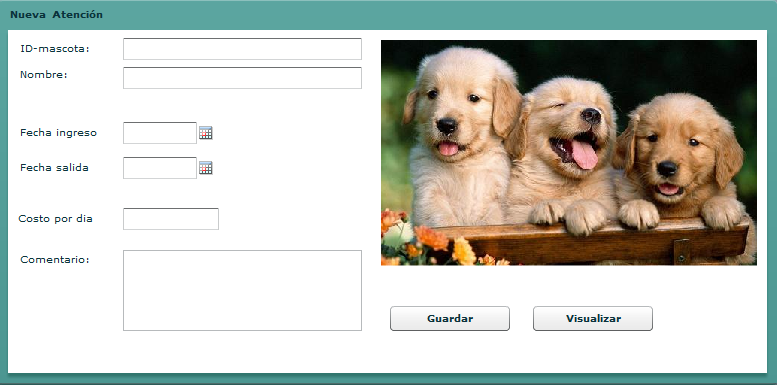


De esta forma las acciones deben ir indentadas mas a la derecha de una “cosa” o sustantivo a la cual se le está aplicando la acción, de esta forma se entenderá fácilmente que se editara la raza o que se registrara una mascota.

Ya que el profesor Narciso Cerpa menciono que debe ser coherente lo que va en este panel, en cuanto a que los paneles aparezcan en el mismo tipo de palabra, aquí en este ejemplo los paneles deberían aparecer al presionar los verbos que son editar o registrar, etc. Y a su vez los sub-servicios que aparecen indentados deberían ser todos acciones.

En este panel se cubre también el punto número 7 de las reglas de oro, ya que estas opciones aparezcan constantemente, permite al usuario ingresar y salir fácilmente de las opciones.

## Datos



**Imagen 4.1**



**Imagen 4.2**

En este panel aparecerán los diversos campos a visualizar o en los cuales se ingresaran datos, aquí deberían cambiar los datos que se entregan o ingresan en los diferentes módulos, pero se debe seguir un patrón especifico, en cuanto a la forma de presentar la forma de multi-selección, las imágenes, los botones, etc. El titulo de este panel debe ser el nombre de la acción o servicio que está prestando, en este caso se está realizando una Nueva Atención.

En el caso de presentar muchos datos a ingresar o mostrar, el panel se dividirá en dos partes iguales, en las presentando dos columnas de datos, para aprovechar de mejor forma el espacio.

En cuanto a los campos en los que se deba hacer una selección múltiple, esta deberá representarse con un combo box. A excepción de cuando se puedan seleccionar más de una opción, pues aquí se utilizaran otros componentes, que permitan esto, como CheckBox o RadioButton.

Si se presentan imágenes deberán aparecer en el lado derecho superior de éste panel, es decir deberá dividir el panel en dos, y utilizar la división derecha para visualizar la imagen.

Para los ingresos de datos comunes se utilizara el componente TexInput, que presenta Flex. Y para el ingreso de comentarios si es que se necesita, se utilizo TextArea. Mientras que para seleccionar una fecha se utilizo el DateField.

Como en este panel pueden existir diferentes botones, para visualizar o agregar datos, y que no son tan importantes para ir en el panel de selección, se creó un botón como componente, el cual será utilizado en el interior de este panel, para evitar que existan diferentes modelos y tamaños de botón, además se creó de un porte adecuado para una palabra bastante larga para evitar que después se pierda la palabra y aparezca “visuali…” por ejemplo. Y se definió que este botón no debe llevar imagen, solo llevaran imagen los del panel de opciones de selección. El botón creado se puede ver en la **imagen 4.2.** Y todas estas estructuras o patrones se pueden observar en la **imagen 4.** Al definir las diferentes estructuras que deben ir dentro de este panel, y la forma en que debe aparecer por pantalla los datos, se cubre claramente el punto 1 de las reglas de oro, el cual es el de luchar por la consistencia.

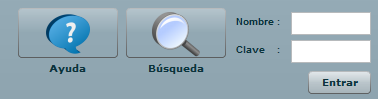
## Botones principales de acceso a los módulos



**Imagen 6**

Es donde se encuentra el acceso directo a los diferentes módulos, es desde aquí donde se accederá a los diversos módulos del sistema, además estos botones, o panel aparecen en todos los módulos en la misma posición, por lo que es una rápida forma de cambiar de un modulo a otro. Esto permite con una estructura más clara, permitiendo una consistencia en el sistema, al ingresar a cada modulo, punto 1 de las reglas de oro, y sirven a la vez de acceso directo, ya que al ingresar a un subsistema no deberá retroceder todo lo que ingreso, sino que bastara con presionar un botón, y podrá ingresar a otro subsistema, confirmando previamente la acción. Cubriendo el punto número 2 de las reglas de oro. También cumple el punto 7, permitiendo un apoyo interno al control total, ya que permite salir o ingresar a un sistema de forma fácil, sin procesos intermedios.

## Botones Rápidos



**Imagen 7**

En la **imagen 7** se muestran los diferentes botones que se encuentran en la parte superior derecha de las pantalla. Esto cubre los puntos 1, ya que siempre están en pantalla, permitiendo acceder fácilmente a ellos y siempre estarán en la misma posición aunque cambien de módulos. Se utiliza como acceso directo a algunos servicios, y permite salir fácilmente de otros, o acceder a esas opciones fácilmente cubriendo los puntos 2 y 7.

Son los que en cierta forma, presentan funcionalidades más generales del sistema, las cuales serán funciones de ayuda, búsqueda de datos importantes dentro del sistema, y el login en donde iniciara sesión cualquier miembro de la clínica veterinaria. Mensajes



**Imagen 8**

En este panel se muestra los resultados que arrojan los procesos de operaciones internas del sistema, resultados tales como: se ha guardado con éxito, modificado con éxito.Este sector del software cubre claramente las reglas numero 3 y 4 de las reglas de oro, ya que se presentan los resultados de operaciones al concluir, y además información relacionada con el proceso que se está llevando a cabo, como ayuda posiblemente.

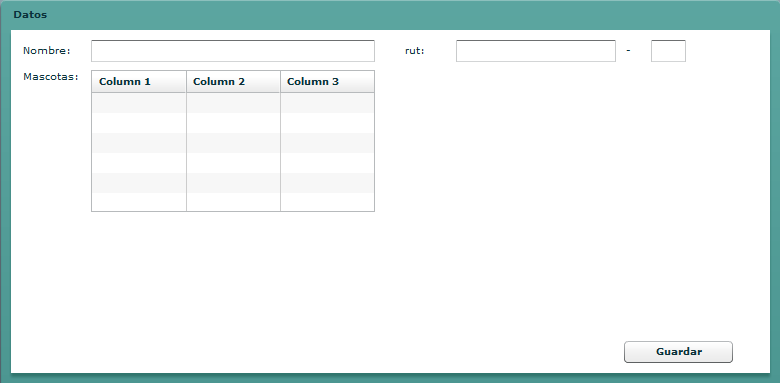


Imagen 9

En la imagen 9 se muestra como se representaran las relaciones 1 a 1, 1 a n y n a n. La forma que tendrán los botones y el lugar en donde preferentemente irán.

Se modifico el color de los paneles el cual ahora será:

Border color 🡪 #: 15ad8f

Las letras tendrán el color de default:

# : 0b333c

Estándar Conexión

En Flex, se crearan unas clases equivalentes, a las clases entidad de la capa 2, para enviarlos desde la capa de presentación los objetos, serializando con esto las entradas captadas de la interfaz.

En Java, se reciben los objetos serializados y mediante las clases de control, se comunica con una clase de la capa 3 que gestiona la conexión con la base de datos.

## Archivos necesarios y organización

La organización de los archivos es la siguiente:

Flex:

* Paquete *transferObjects*: Se crea todas las que son equivalentes a las clases entidad. Dichas clases solo deben ser declarados los atributos. Su instanciación es decir los objetos, serán enviados a Java.

Java:

* Paquete *dao*: Posee una clase que establece la conexión de Jboss con la BD. Sera utilizada por todas las clases de la capa de base de datos.
* Paquete *dao.postgres*: tiene las clases “mantener” las cuales disponen de los métodos para inserción / búsqueda / edición de datos sobre la BD. Cada método contiene sentencias SQL.
* Paquete *services:* Posee las clases de control, que instancian las clases entidad y que instancian las clases de los paquetes *dao.postgres y dao.*
* Paquete *transferObject*: Clases entidad, correspondiente a la capa 2.
* Directorio *WebContent/WEB-INF/flex*: Existe un archivo llamado *remoting-config.xml*  el cual deberá tener registrado todas las clases del paquete *services.*

Servidor:

* Directorio *jboss-4.0.5.GA/server/default/deploy:* Existe un archivo llamado *DBConnectionServer-ds.xml* se deberán configurar los siguientes tags
  + <connection-url>
  + <user-name>
  + <password>

## Normas y procedimientos

Flex: transferObject:

**Creación**: Nuevo -> ActionScript Class

**Convención de nombres**: Primera letra mayúscula

Ejemplo: Cliente.as

**Contenido**: No posee constructor ni métodos, solo atributos.

**Ejemplo:**

**package** transferObjects

{

[**Bindable**]

[RemoteClass(alias=**"transferObjects.Cliente"**)]

**public** **class** Cliente

{

**public** **var** rut:String;

}

}

Java: dao:

No se modifica. Contiene el archivo: DBConnectionManager.java

Java: dao.postgres:

**Creación**: Nuevo -> Class

**Convención de nombres**: Primera letra mayúscula, finalizado con “DB”,

Ejemplo: ClienteDB.java

**Contenido**: Constructor con parámetro <conection> y métodos que realizan acción mantener.

**Ejemplo:**

package dao.postgres;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import transferObjects.Cliente;

public class ClienteDB

{

PreparedStatement insert;

PreparedStatement selectAll;

public PersonDB(Connection connection)

{

try

{

insert = connection.prepareStatement(query);

}

catch (SQLException e)

{

e.printStackTrace();

}

}

public int addCliente(Cliente c){

}

}

Java: dao.service

**Creación**: Nuevo -> Class

**Convención de nombres**: Primera letra mayúscula, finalizado con “Service”,

Ejemplo: ClienteService.java

**Contenido**: Sin constructor, con métodos que realizan acción mantener.

**Ejemplo:**

package services;

import java.sql.Connection;

import java.sql.SQLException;

import dao.DBConnectionManager;

import dao.postgres.ClienteDB;

import transferObjects.Cliente;

public class PersonService

{

public int insertCliente(Cliente cliente)

{

int result=0;

try

{

Connection connection=DBConnectionManager.getConnection();

ClienteDB cdb= new ClienteDB (connection);

result= cdb.insertCliente(cliente);

connection.close();

} catch (SQLException e)

{

e.printStackTrace();

}

return result;

}

Java:transferObjects:

**Creación**: Nuevo -> Class

**Convención de nombres**: Primera letra mayúscula.

Ejemplo: Cliente.java

**Contenido**: Constructor con parámetro para la instanciación de una entidad. Métodos get y set de los atributos.

**Ejemplo:**

**package** transferObjects;

**public** **class** Cliente

{

**private** String nombre;

**public** String getNombre () {

**return** nombre;

}

**public** **void** setNombre(String nombre) {

**this**.nombre = nombre;

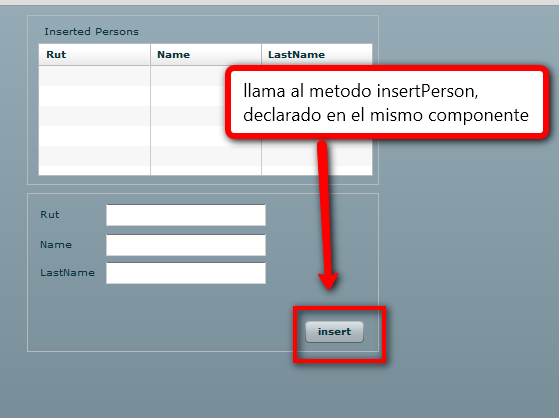
}

}

## Ejemplo

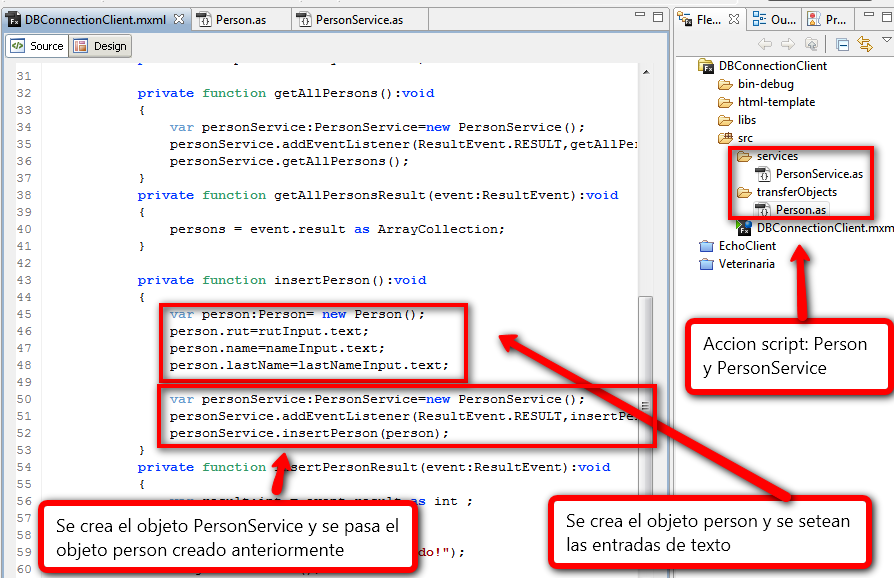
Considerando el ejemplo realizado en la ayudantía.

Clases que interactúan en Flex



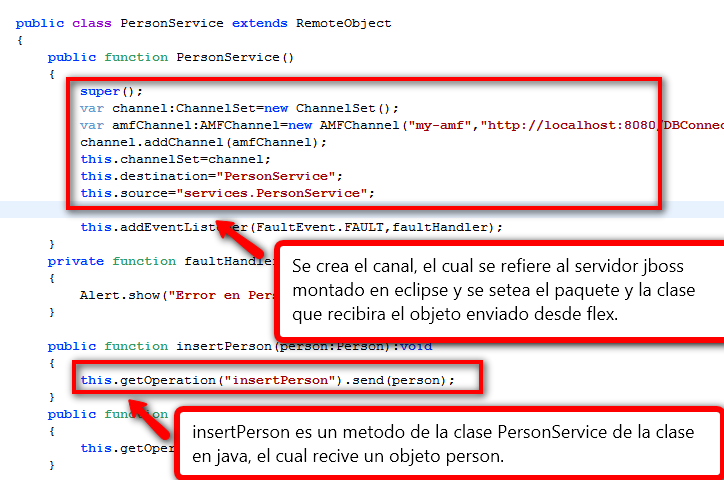
EL control insert llama a un método.

Observacion: En las propiedades de botton solo se hace el llamado al método inserPerson, en este caso no se pasa directamente los inputText de rut, name y lastName, debido a que pueden ser recogidos desde el mismo componente.



El botón insert, llama a la función insertPerson, en la cual se instancia una persona, y se asignan a los atributos, las entradas de los inputText.

Este objeto persona, es pasado al actionScript PersonService.

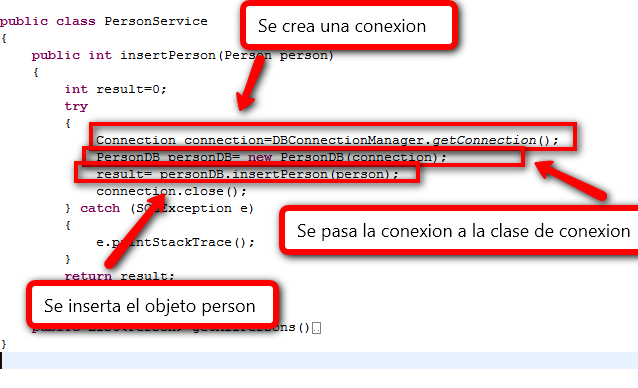


Este action script posee la configuración del servidor y la clase a la cual será enviado el objeto person.

En el método insertPerson, se pasa un objeto person.

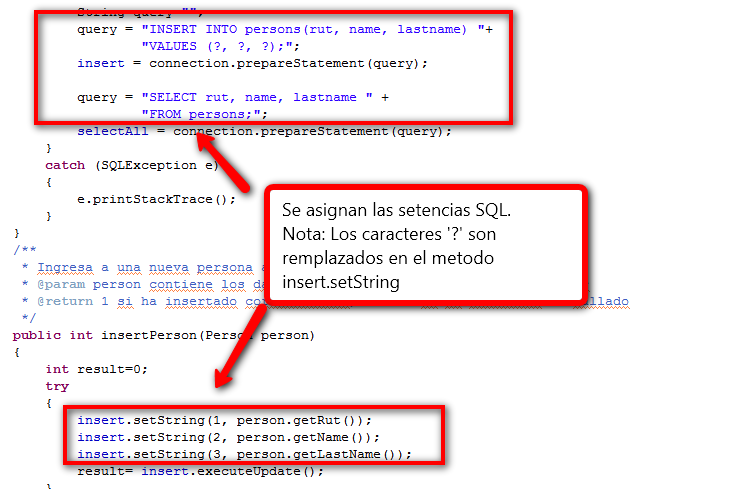
Clases que interactúan en Java

Clase que recibe el objeto que fue enviado desde el action script de flex.



Se declara una conexión, y un objeto que utilizara la conexión e insertara los datos.

Clase PersonDB, se recibe la conexión y el objeto person, y se crea la sentencia SQL



El objeto insert mediante el método setString, pasa en orden los atributos, para esto se debe indicar numéricamente la posición y el valor.

En el caso de insert.setString(1, person.getRut()) corresponde al primer símbolo ‘?’ del atributo “query”, en el cual se inserta el rut.

Estándar SQL

## Introducción

Dentro de nuestra clase tendremos nuestra “variable de conexión”, la cual utilizaremos para poder manipular los datos.

conn = (Connection) Mysql.connect("jdbc:mysql://"+host+":"+port+"/"+bd,user,pass,conn);

A su vez tendremos una variable del tipo String, en la que almacenaremos la modificación solicitada a realizar en nuestra base de datos.

private String SQL="";

Para finalmente hacer la inserción de la modificación, con los parámetros anteriores.

Mysql.insertSQL(conn, SQL);

O la selección de algún dato desde una lista

public ArrayList<String> getServicios(int numColumnas)throws SQLException{

SQL = "SELECT \* FROM Servicio";

return Mysql.selectSQL(conn, SQL, numColumnas);

}

A continuación se especifican las distintas acciones en SQL para la manipulación de datos.

## 1. Select

La recuperación de los datos en el lenguaje SQL se realiza mediante la sentencia SELECT, seleccionar. Esta sentencia permite indicar al SGBD la información que se quiere recuperar. Esta es la sentencia SQL, con diferencia, más habitual. La sentencia SELECT consta de cuatro partes básicas:

* La cláusula SELECT seguida de la descripción de lo que se desea ver, los nombres de las columnas a seleccionar. Esta parte es obligatoria.
  + Las columnas a seleccionar se enumeran sin más en la cláusula SELECT. Si se desea seleccionar todas las columnas de una tabla se puede hacer enumerando a todas las columnas o colocando un asterisco, \*, en su lugar.
* La cláusula FROM seguida de la especificación de las tablas de las que se han de obtener los datos. Esta parte es obligatoria.
  + La cláusula FROM define las tablas de las que se van a seleccionar las columnas.
* La cláusula WHERE seguida por un criterio de selección, una condición. Esta parte es opcional.
  + Es aquí donde debemos proponer la condición que han de cumplir todas las filas para salir en el resultado de la consulta
* La cláusula ORDER BY seguida por el criterio de ordenación. Esta parte es opcional.
  + Se utiliza para especificar el criterio de ordenación de la respuesta a la consulta. Por defecto la ordenación es ascendente, aunque se puede especificar un orden descendente

Una primera aproximación a la sintaxis de la sentencia SELECT puede mostrarnos la siguiente expresión:

SELECT {\* | {columna,}+}

FROM {tabla,}+

[WHERE condición]

[ORDER BY {expresiónColumna [ASC | DESC],}+];

**Ejemplo**

public ArrayList<String> getCita(int id,int numColumnas ) throws SQLException {

SQL = "SELECT \* FROM Cita WHERE Cita.idCita = '"+id+"';";

return Mysql.selectSQL(conn, SQL, numColumnas);

}

## 2. Actualizar

Otra de las operaciones más comunes es la modificación de la información almacenada en las tablas. Para ello se utiliza el comando UPDATE cuya sintaxis se muestra a continuación.

UPDATE tabla SET {columna = expresión,}+ [WHERE condición];

Se especificará en la cláusula SET las columnas que se actualizarán y con qué valores. La cláusula WHERE indica las filas con las que se va a trabajar. Si se omite la actualización afectará a todas las filas de la tabla.

**Ejemplo**

public void setServicio(Servicio s) throws SQLException {

SQL = "UPDATE Servicio SET Servicio.descripcion = '"+s.getDescripcion()+"', Servicio.Admistrador\_idAdmistrador='"+s.getAdministrador()+"';";

Mysql.insertSQL(conn, SQL);

}

## 3. Borrar

Con insertar y modificar, la otra operación que completa el trio es la de borrado de filas. La sintaxis es la que sigue:

DELETE FROM tabla [WHERE condición];

Borrará todas las filas que cumplan la condición especificada en la cláusula WHERE. Si esta cláusula no se fija, se borrarán todas las filas de la tabla. Aquí cabe decir que aunque con DELETE borremos todas las filas de una tabla, no borramos la definición de la tabla del diccionario y podemos insertar datos posteriormente en la tabla. Esta es una diferencia con la sentencia DROP TABLE, que produce la eliminación tanto del contenido de la tabla como de la definición de la misma.

**Ejemplo**

public void delServicio(int id) throws SQLException {

SQL = "DELETE FROM Servicio WHERE Servicio.idServicio = '"+id+"';";

Mysql.insertSQL(conn, SQL);

}

## 4. Insertar

El comando que permite insertar filas en las tablas es el siguiente.

INSERT INTO tabla [({columna,}\*)] VALUES ({expresión,}+);

Sólo especificaremos las columnas donde insertar y su orden cuando no insertemos datos en todas ellas o no lo hagamos en el mismo orden en que definimos la tabla. La asociación columna-valor es posicional. Los valores deben cumplir con los tipos de datos definidos. Los valores de tipo caracter y fecha deben ir encerrados entre comillas simples, ('').

**Ejemplo**

public void addServicio(Servicio s) throws SQLException {

SQL = "INSERT INTO Servicio VALUES ('"+s.getIdServicio()+"','"+ s.getNombre()+"','"+s.getCosto() +"','"+ s.getIdAdministrador()+"','"+s.getDescripcion()+");";

Mysql.insertSQL(conn, SQL);

}

Modelo Relacional

**post\_operatorio**(hora, fecha, costo, responsables, rut\_cliente, nombre\_mascota, medicamentos, alimentos, fechaNacimiento\_mascota, indicaciones, idPostOperatorio)

**pre\_operatorio**(hora, fecha, fechaNacimiento\_mascota, costo, responsables, rut\_cliente, nombre\_mascota, observaciones, síntomas, diagnostico, idPreOperatorio)

**cirugía**(hora, fecha, costo, fechaNacimiento\_mascota, responsables, rut\_cliente, nombre\_mascota, estado, diagnostico, tipo, idCirugia)

**consulta**(hora, fecha, costo, fechaNacimiento\_mascota, responsables, rut\_cliente, nombre\_mascota, anamnesis, idConsulta)

**control**(hora, fecha, costo, fechaNacimiento\_mascota, responsables, rut\_cliente, nombre\_mascota, descripcion, recomendaicon, próximaFecha, idControl)

**vacuna**(hora, fecha, costo, fechaNacimiento\_mascota, responsables, rut\_cliente, nombre\_mascota, precio, nombre, descripcion, fechaCaducidad, laboratorio)

**alojamiento**(hora, fecha, fechaNacimiento\_mascota, costo, responsables, rut\_cliente, nombre\_mascota, canil, fecha-salida, comentario, tiempoEstadia, solicitudes, idEstadia)

**baño**(hora, fecha, costo, fechaNacimiento\_mascota, responsables, rut\_cliente, nombre\_mascota, tipo, comentario, idBaño)

**corte**(hora, fecha, costo, fechaNacimiento\_mascota, responsables, rut\_cliente, nombre\_mascota, nombre, descripcion, raza, foto, idCorte)

**servicio**(nombre, nombresAdministradores, descripcion, rut\_cliente)

**hora\_bloqueada**(fecha, hora, nombre\_usuario, nombre\_servicio)

**usuario**(nombre, contraseña, cargo, privilegio)

**pertenece**(nombre\_usuario, nombre\_servicio)

**cliente**(rut, estado, teléfonos, nombre, correo)

**domicilio**(rut, numero, calle, comuna)

**mascota**(raza, estado, fotos, nombre, fechaNacimiento, sexo, edad, rut\_cliente)

**aviso**(hora, fecha, descripcion, correo\_cliente\_web, nombre\_mascota, fechaNacimiento\_mascota, rut\_cliente)

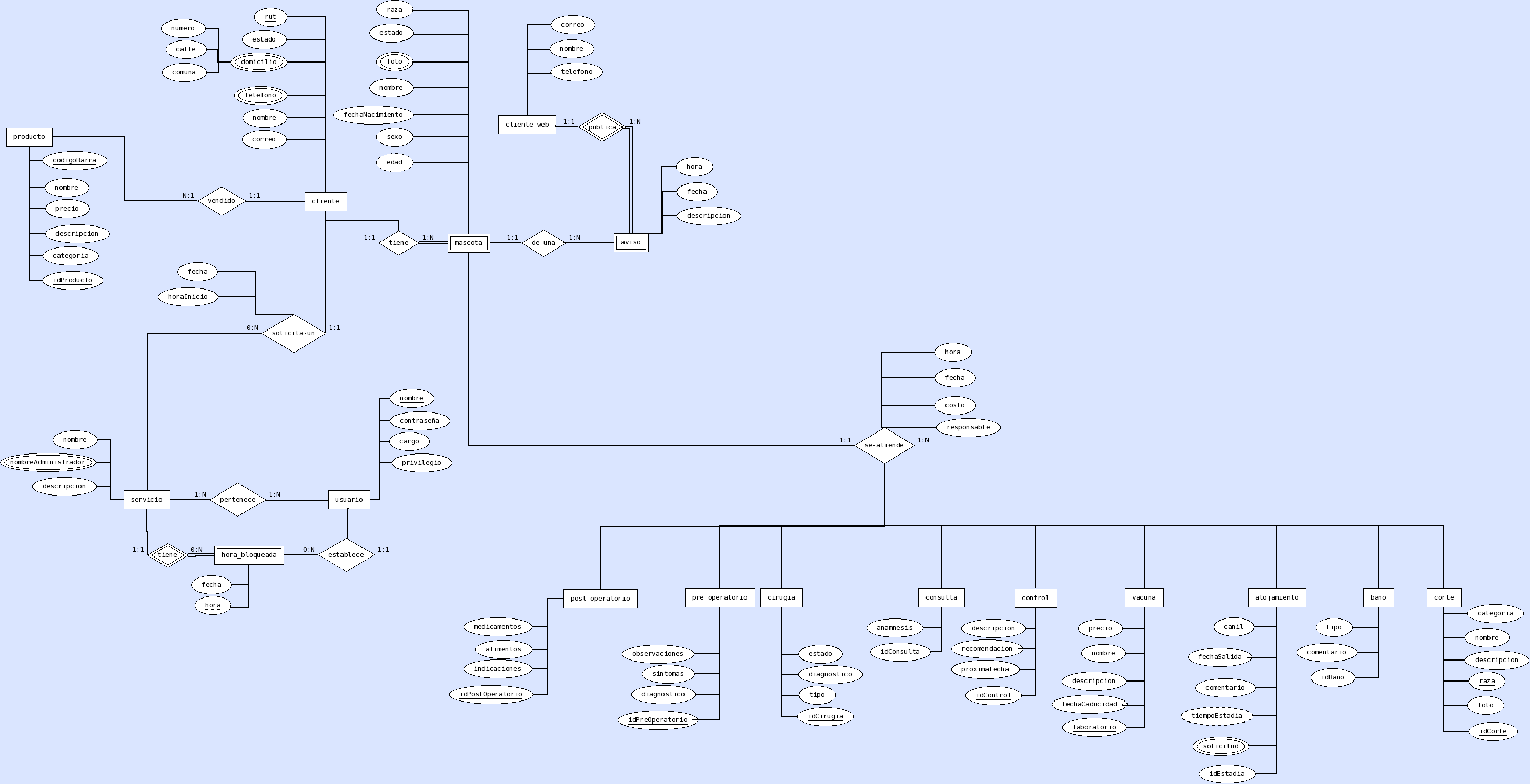
**cliente\_web**(correo, nombre, telefono)

**producto**(codigoBarra, nombre, precio, descripcion, categoría, idProducto)

**solicita-cita**(rut\_cliente, nombre\_servicio, fecha, horaInicio)

**producto-vendido**(rut\_cliente, codigoBarra, idProducto)

Modelo Base de datos



Estándar de documentación de código de actionScript

**Estructura general de un archivo fuente**

//=======================================================================

// FECHA CREACIÓN:

// AUTOR:

// …. *Comentarios generales*

//=======================================================================

package *xxxx*;

import *xxxxxx* ;

....

## Tags usados en ASDoc

|  |  |
| --- | --- |
| **Tag** | **Descripción** |
| **@**private | Impide que una propiedad o método aparezca en la explicación html porque si no se especifica ningún comentario, ni este tag, aparecerá de todas formas. |
| **@**default | Explica el valor predeterminado de una propiedad. |
| @param | Definición de un parámetro de un método |
| @return | Describe el valor devuelto de un método, aunque la clase del valor devuelto aparecerá de todas formas. |
| @see | Referencia hacia otro elemento de la clase, otra clase (del mismo paquete o de otro distinto) o hacia un sitio externo. |
| @example | Enuncia un ejemplo de uso del elemento que comentemos. El código que vayamos a poner en el ejemplo va entre etiquetas **<listing version="3.0" ></listing>** |
| @langversion | Versión de actionScript |
| @author | Autor de la clase o archivo |

## Comentario de clases e interfaces

/\*\*

\* La primera frase de un multi-parrafo de una clase.

\* <p>segundo parrafo.</p>

\*

\* @langversion ActionScript 3.0

\*

\* @author "Ray Koch"

\*/

class SomeThing() {

## Comentario de métodos

/\*\*

\* Aquí van los comentarios del método

\*

\* @langversion ActionScript 3.0

\*

\* @param a Describe param a aquí.

\* @param b Describe param b aquí.

\*

\* @see otraFuncionAsociada

\*

\* @example myFunction("daddy-o", 3234);

\*/

public function myFunction(a:String, b:Number):void {}

## Comentario de atributos

/\*\* String de propiedad privada \*/

private var privateProperty:String = "Keep Off: Private Property";

ESTANDAR DE DOCUMENTACION DE CODIGO JAVA

## Estructura general de un archivo fuente

//=======================================================================

// FECHA CREACIÓN:

// AUTOR:

// …. *Comentarios generales*

//=======================================================================

package *xxxx*;

import *xxxxxx* ;

....

## Tags usados en javadoc

|  |  |
| --- | --- |
| **Tag** | **Descripción** |
| @author | Nombre del desarrollador. |
| @deprecated | Indica que el método o clase es antigua y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores. |
| @param | Definición de un parámetro de un método, es requerido para todos los parámetros del método. |
| @return | Informa de lo que devuelve el método, no se puede usar en constructores o métodos "void". |
| @see | Asocia con otro método o clase. |
| @throws | Excepción lanzada por el método |
| @version | Versión del método o clase. |
| @inv | Invariante de una clase |
| @since | Utilizada para indicar la versión del JDK |
| @pre | Precondición de un método o constructor. |
| @pos | Poscondición de un método o constructor. |
| @exception | Excepción lanzada por el método |

## Documentación del código

El código debe comentarse utilizando la sintaxis apropiada para uso de *javadoc*, teniendo en cuenta que para la producción de la documentación deben incorporarse los tags particulares que no hagan parte del estándar.

## Comentario de clases e interfaces

/\*\*

\* *Descripción de la clase*

\* @author *Nombre del desarrollador 1*

\* @author *Nombre del desarrollador 2*

\* @version *número versión y fecha*

\* @inv Invariante de clase

\*/

Ejemplo de documentación de una clase

/\*\*

\* Se utiliza para intentar convertir una cadena en un tipo

\* numérico, pero la cadena no tiene el formato adecuado

\*

\* @author unascribed

\* @version 1.16, 02/02/00

\* @see java.lang.Integer#toString()

\* @since JDK1.0

\*/

public

class NumberFormatException extends IllegalArgumentException {

/\*\*

\* Construye una <code>NumberFormatException</code> con ningún mensaje de detalle.

\*/

public NumberFormatException () {

super();

}

/\*\*

\* Construlle una <code>NumberFormatException</code> con la

\* especificación detallada del mensaje.

\*

\* @param s el detalle del mensaje va aquí

\*/

public NumberFormatException (String s) {

super (s);

}

}

## Comentario de atributos

/\*\* Descripción del atributo \*/

Ejemplo de documentación de atributos

/\*\*

\* Número de alumnos

\*/

private int alumnos;

## Comentario de métodos constructores

/\*\*

\* *Descripción de lo que hace el método*

\* @param p *Descripción del parámetro*.

\* @pre *precondición uno*

\* @pre *precondición dos*

\* @pos *poscondición uno*

\* @pos *poscondición dos*

\* @exception *TipoExcepción*, *Condiciones que causan la excepción*

\*/

Ejemplo de documentación de un constructor.

public class Medio {

protected String clave;

protected String titulo;

protected String genero;

protected int duracion;

protected Fecha fecha;

/\*\*

\* Constructor predeterminado

\*/

public Medio() {

}

/\*\*

\* Constructor que inicializa los atributos de la clase

\* @param clave Clave de la canción o película

\* @param titulo Título de la canción o película

\* @param genero Género de la canción o película

\* @param duracion Duración de la canción o película

\* @param fecha Fecha de liberación de la canción o película

\*/

public Medio(String clave, String titulo, String genero, int duracion,

Fecha fecha) {

this.clave = clave;

this.titulo = titulo;

this.genero = genero;

this.duracion = duracion;

this.fecha = fecha;

}

}

## Convenciones de nombres

**Clases:**

Los nombres de las clases deben ser **sustantivos**, cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas. Intentar mantener los nombres de las clases simples y descriptivas. Usar palabras completas, evitar acrónimos y abreviaturas (a no ser que la abreviatura sea mucho más conocida que el nombre completo).

Ejemplos:

class **C**liente;

class **I**magen**A**nimal;

## Métodos:

Los métodos deben ser **verbos**, cuando son compuestos tendrán la primera letra en minúscula, y la primera letra de las siguientes palabras que lo forma en mayúscula.

Ejemplos:

**e**jecutar();

**e**jecutar**R**apido();

**e**jecutar**M**uy**R**apido();

1. **Nombres de métodos para Administración:**
2. Registrar Objeto: **add**Objeto.
3. Editar Objeto: **edit**Objeto.
4. Eliminar Objeto: **del**Objeto.
5. Retornar Objeto: **get**Objeto.
6. Guardar Objeto: **set**Objeto.

Donde “Objeto” es el nombre de la clase como: Usuario, Cliente, Mascota, etc.

Para el caso de los objetos como Usuario, Mascota, Cliente, etc. sus métodos para interacción de atributos son:

1. Retornar Atributo: **get**Atributo
2. Guardar Atributo: **set**Atributo

Donde “Atributo” es el nombre completo del atributo pero con mayúscula por ejemplo, para retornar atributo “nombre” de la Clase Cliente el método se llamaría getNombre.

1. **Parámetros de Métodos**

El parámetro de entrada del método es el nombre completo en minúscula del atributo al que se refiere, Por ejemplo:

getUsuario(Usuario **usuario**)

setNombre(String **nombre**)

## Variables:

Excepto las constantes, todas las instancias y variables de clase o método empezarán con minúscula. Las palabras internas que lo forman (si son compuestas) empiezan con su primera letra en mayúsculas (como el caso de los métodos). Los nombres de las variables deben ser cortos pero con significado. Los nombres de variables de un solo carácter se deben evitar, excepto para variables índices temporales. Nombres comunes para variables temporales son i, j, k, m, y n para enteros (muy utilizados como contadores en ciclos for y while); c, d, y e para caracteres.

## Prefijos en Variables Globales:

En el caso de que se utilicen variables que son únicamente de un objeto, y que son utilizadas en alguna clase de interfaz u otra en que intervienen otras clases u objetos, se recomienda utilizar un prefijo, que debe ser el nombre de la clase u objeto, seguido por un ‘\_’ y luego el nombre de la variable.

Ejemplos:

String mascota\_nombre;

String cliente\_nombre;

String mascota\_edad;

String cliente\_edad;

## Constantes:

Los nombres de las variables declaradas como constantes deben ir totalmente en mayúsculas separando las palabras con un subguión ("\_").

Ejemplos:

static final int ANCHURA\_MINIMA = 4;

static final int ANCHURA\_MAXIMA = 99;

static final int NUMERO\_DE\_RAZAS = 100;

## Hábitos

## Proporcionando acceso a variables de instancia y de clase:

Para que la parte lógica sea netamente orientada a objetos y el sistema sea seguro solo se permitirán atributos privados en una clase, por ende se restringe absolutamente la utilización de atributos públicos. Para poder acceder a estos se utilizaran métodos. Ya que a menudo las variables de instancia no necesitan ser asignadas/consultadas explícitamente, a menudo esto sucede como efecto lateral de llamadas a métodos.

## Referencias a variables y métodos de clase:

Evitar usar un objeto para acceder a una variable o método de clase (static). Usar el nombre de la clase en su lugar.

Ejemplos:

CORRECTO

metodoDeClase();

UnaClase.metodoDeClase();

EVITAR

unObjeto.metodoDeClase();

## Asignaciones de variables:

Evitar asignar el mismo valor a varias variables en la misma sentencia. Es difícil de leer.

Ejemplo:

EVITAR

fooBar.fChar = barFoo.lchar = 'c';

No usar asignaciones caladas como un intento de mejorar el rendimiento en tiempo de ejecución. Ese es el trabajo del compilador.

Ejemplos:

EVITAR

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

CORRECTO

a = b + c;

d = a + r;

## Paréntesis:

En general es una buena idea usar paréntesis en expresiones que implican distintos operadores para evitar problemas con el orden de precedencia de los operadores. Incluso si parece claro el orden de precedencia de los operadores, podría no ser así para otros, no se debe asumir que otros programadores conozcan el orden de precedencia.

Ejemplos:

EVITAR

if (a == b && c == d)

CORRECTO

if ((a == b) && (c == d))

## Variables temporales:

Evitar variables temporales que no son posteriormente utilizadas o que son utilizadas una sola vez, esto para simplificar código, obviamente esto es aplicable si no complica su interpretación.

Ejemplos:

EVITAR

double basePrice = anOrder.basePrice();

return(basePrice > 100);

CORRECTO

return(anOrder.basePrice() > 1000);

En el ejemplo se puede ver que basePice es únicamente utilizada para retornar un valor en un función, pero esto se puede realizar de manera más directa y retornando anOrder.basePrice();

En caso de que una expresión de condición sea muy larga y compleja es mejor utilizar la forma anteriormente descrita como errónea.

Ejemplos:

EVITAR

if( ( platform.toUpperCase().indexOf(“MAC”) > -1 ) &&

( browser.toUpperCase().indexOf(“IE”) > -1) &&

wasInitialized() && resize > 0 ){

// do something

}

CORRECTO

final boolean isMacOs = plataform.toUpperCase().indexOf(“MAC”) > -1;

final boolean isIEBrowser = plataform.toUpperCase().indexOf(“IE”) > -1;

final boolean wasResized = resize > 0;

if (isMacOs && isIEBowser && wasInitialized() && wasResized){

// do something

}

## Comentarios de métodos analizadores

Un método analizador es el que retorna un valor, el nombre de la mayoría de estos métodos comienzan con “get”

/\*\*

\* *Descripción de lo que hace el método*

\* @param p *Descripción del parámetro*.

\* @pre *precondición uno*

\* @return *valor a retornar*

\* @exception *TipoExcepción*, *Condiciones que causan la excepción*

\*/

Ejemplo de documentación de un método analizador

/\*\*

\* Retorna las unidades de café del inventario

\*

\* @return int

\*/

public int getUnidCafe() {

return coffee;

}

## Comentario de métodos modificadores

Un método analizador es el que retorna un valor, el nombre de la mayoría de estos métodos comienzan con “set”

/\*\*

\* *Descripción de lo que hace el método*

\* @param p *Descripción del parámetro*.

\* @pre precondición uno

\* @pos poscondición uno

\* @exception *TipoExcepción*, *Condiciones que causan la excepción*

\*/

Ejemplo de documentación de un método modificador

/\*\*

\* Inserta un título en la clase descripción.

\* Al ser el título obligatorio, si es nulo o vacío se lanzará

\* una excepción.

\*

\* @param titulo El nuevo título de la descripción.

\* @throws IllegalArgumentException Si titulo es null, está vacío o contiene solo espacios.

\*/

public void setTitulo (String titulo) throws IllegalArgumentException

{

if (titulo == null || titulo.trim().equals(""))

{

throw new Exception ("El título no puede ser nulo o vacío");

}

else

{

this.titulo = titulo;

}

}

## Otros comentarios internos

// ................…

public class Inventario {

//código de Invetario aquí

}

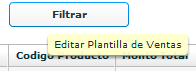
Estándar del usuario, basado en IEEEStd 1063-2001

## Información sobre los comandos de software

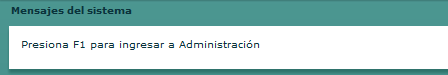
Es importante que el usuario se adapte lo más pronto posible al uso de la aplicación, para ello es importante que se le digan cuales son las formas más rápidas de realzar las acciones.

Esto puede realizarse de diversas formas pero principalmente mediante toolTips(1) o una área reservada para estos mensajes(2).

(1):



(2):



## Formatos de las advertencias, precauciones y notas

Todas las advertencias de este tipo deben ser claramente distinguibles, para esto se deben hacer popUp de la siguiente forma, se trata principalmente de los errores en que puede incurrir el usuario y no de errores del sistema:

## Advertencias y precauciones:

Nombre del mensaje

Imagen relacionada

Descripción de los riesgos

Opciones disponibles

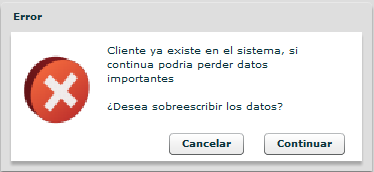
Ejemplo:

*Nombre del mensaje:* Error

*Imagen relacionada:* Error.png

*Descripción de los riesgos:* Cliente ya existe en el sistema, si continúa podría perder datos importantes

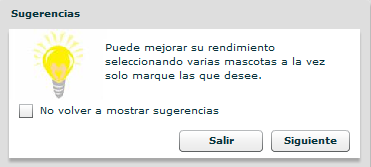
*Opciones disponibles:* Sobrescribir los datos, (cancelar, continuar)



## Consejos:

Al igual que los errores y advertencias los consejos deben ser mostrados a través de mensajes popUp, pero estos deber tener una apariencia mucho menos severa ya que no representan un peligro para los datos del sistema

**Ej:**



## Errores del sistema

El sistema tendrá un numero de errores que son conocidos y que en su memento no pudieron ser solucionados ya sea por el costo de este, o la falta de tiempo, por ello estos errores en algún momento pueden suceder, no es importante decirle al usuario en qué momento sucede el error sino que cual es la forma de que no le afecte en demasía. En este caso se debe mostrar el mensaje de error correspondiente, “Error grave”, y los caminos disponibles entre los cuales puede existir una vía de comunicación con los desarrolladores para reportar el error.

Testing

## Estándar de Testing unitario

En lo referente a los distintos test que se realizan en la ingeniería de software, explicaremos a continuación los test unitarios, los cuales pequeños programas cuya finalidad es testear el correcto funcionamiento de la clases. Con esta forma verificamos el funcionamiento sea como se estipulaba en los requerimientos funcionales del sistema.

Generalmente los test unitarios envían mensajes a las clases que se están testeando y verifican si los resultados de dichos mensajes satisface los casos de prueba para cada funcionalidad (pueden ser los métodos).

En términos prácticos el objetivo de las pruebas unitarias es separar cada parte del programa y demostrar que cada parte por si sola es correcta. En el mismo sentido se verifica que cierta funcionalidad se ejecuta con parámetros y/o especificaciones que se dejaron claro en los casos de uso.

Estas pruebas además de lo descrito anteriormente se pueden obtener ciertas ventajas como:

**Ayuda al cambio:** Se facilita que el programador cambie código para conseguir mejoras en la estructura, dado que permite hacer cambios y por ende verificar si funcionan o no.

**Contribuye a la integración:** Dado que con estas pruebas previas se llega a la integración con cierta seguridad de que el código está funcionando.

**Documentación del código:** Estas mismas pruebas corresponderán a documentación, ya que observará como utilizarlo.

**Separación de la interfaz y la implementación:** Debido a que la mayor interacción entre los casos de prueba y la funcionalidad bajo prueba son las interfaces, se podría cambiar una de las dos sin afectar a la otra.

**Los errores se encontrarán más acotados y por ende más fáciles de encontrar:** Esto es por el hecho que cada testeo unitario, hará pruebas justamente de varias funcionalidades y habrá menor margen de funcionalidades que no se hallan revisado

**Tipo de pruebas que realizaremos**

**Pruebas de Caja Blanca:** En ellas se revisa el código paso a paso, con datos que se asignarán en un método. Con el fin de automatizar este testeo usaremos el Framework JUnit, en el cual se instancia una clase Test en donde se pruebas los métodos de cada clase y luego de su previa ejecución se verifican con los obtenidos, viendo así la real funcionalidad y ver si queda aprobado.

**Pruebas de Caja Negra:** Estas se limitan solamente a que ciertos datos de entrada observar la como responde la salida haciendo caso omiso de cómo ocurrió al interior. Estas se realizarán la interfaz Flex con la cual interactuará el usuario.

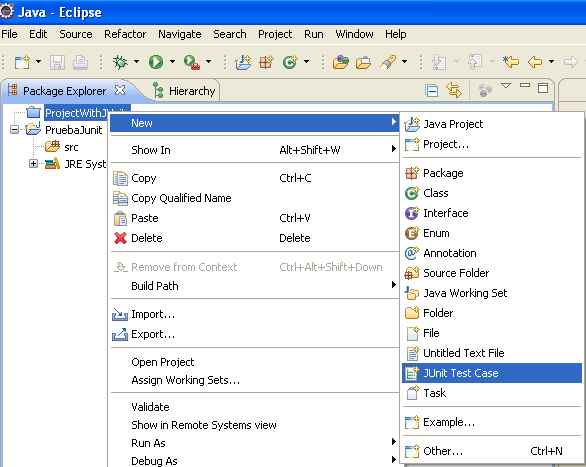
## Test unitario con JUnit

A continuación mostraremos los pasos de cómo aplicar esta prueba de testing a las funcionalidades del sistema.

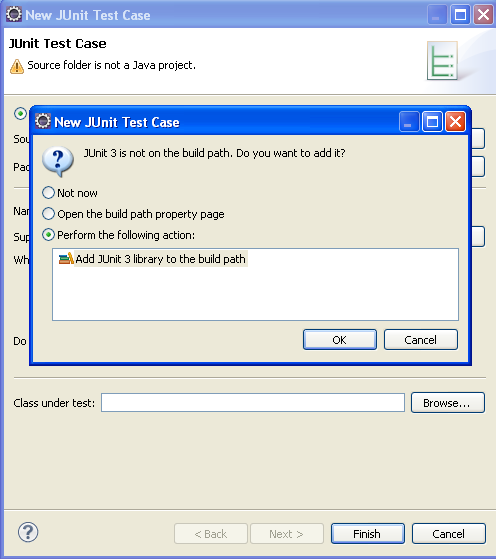
* Ejecutar Eclipse
* Hacer clic en **File🡪New 🡪Project**, luego elegir **Java** y hacer clic en **Next**.
* En la ventana que aparece escriba **PruebaJUnit** como nombre del proyecto y luego finish.

Ahora crearemos el test, en donde no es requisito tener ya escrito el codigo de la clase(s) que utilizaremos:

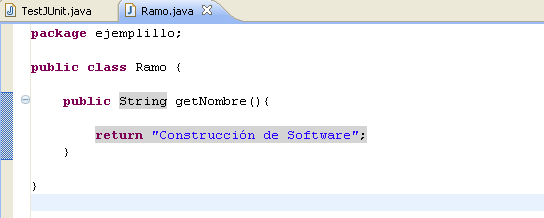
Hacemos clic derecho sobte el proyecto **PruebaJUnit** y seleccionamos JUnit Test Case como en la imagen



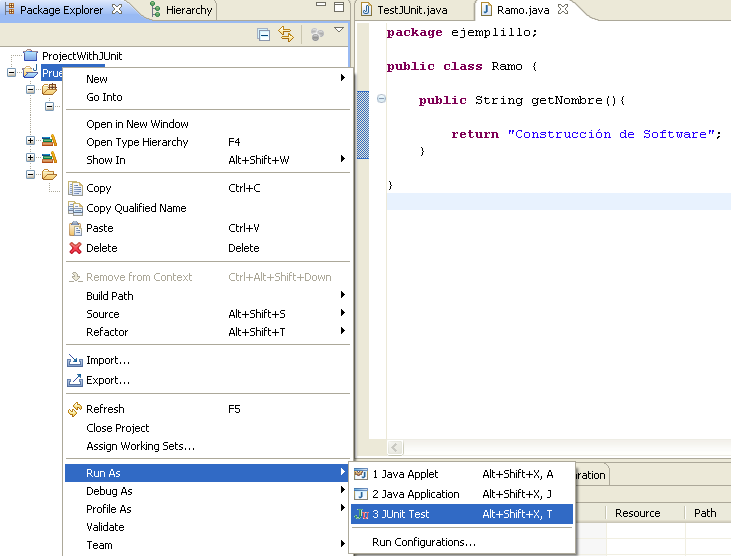
Luego ponen ok agregando la librería.



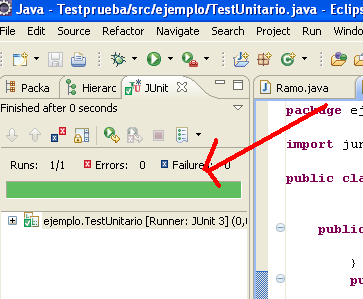
* Luego agregamos algún código referente a los métodos que tendrá nuestra clase a probar, en este caso implementaremos un método que nos retorne un como respuesta el ramo que estamos tomando.
* Ahora hacer clic derecho sobre el proyecto y crear la clase que probaremos, en este ejemplo se llamará Ramo(debe tener el mismo package)
* Le asignamos los atributos necesarios y sus métodos, en este ejemplo solo un método:



Finalmente para probar la clase hacemos clic derecho en el proyecto Run as🡪TestJUnit test:



Deberá aparecerles en la esquina superior izquierda una franja color verde, indicando que esta Ok la clase, de caso contario será color rojo en donde indicará el porqué de la falla.



## Plantilla a utilizar para llevar testear en modo caja negra

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID Caso de uso:** | | |  | | |
| **ID/Nombre requerimiento** | | | **Nivel de prueba:** | | |
| **Nombre Escenario:** | | | **Ejecutor de la prueba:** | | |
| **ID Nombre Casos de prueba:** | | | **Fecha Ejecución de prueba:** | | |
| **Condiciones para que se ejecute el caso de prueba:** | | | | | |
|  | | | | | |
|  | | | | | |
| **Pasos a seguir** | **Condición** | **Valor** | | **Resultado esperado** | **Resultado Obtenido** |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
| **Criterios de Aprobación del Caso de Prueba: Cumplir con 90% los resultados** | | | | | |
| **Decisión de aprobación del Caso de prueba: Aprobó: \_\_ Falló: \_ \_** | | | | | |
| **Nota:** | | | | | |

## Testing de integración.

**Objetivos:**

Las pruebas de integración que se realizaran tendrán como objetivo principal detectar las fallas de coherencia semántica a nivel funcional y estructural, tanto de semántica estática (se llama correctamente a otra función), como de semántica dinámica (en donde una función recibe de otra lo que esperaba). En cada una de las pruebas de integración se tratara de identificar todas las posibles llamadas y los retornos de la función, con tal de lograr una cobertura de todos los posibles escenarios de la función.

Si los resultados que obtengamos por la funcionalidad son diferentes de los esperados aunque sea en un solo caso, esto inequívocamente muestra que la funcionalidad esta incorrecto, por contraste, un comportamiento correcto del sistema en un numero finito de pruebas, nos garantizara la correctitud de la función. Por lo tanto el testing de integración será considerado como el medio de de garantizar que el comportamiento de la función es correcto.

El testing de integración debiera:

* Ayudar a localizar errores, no solo detectar su presencia si no que también a plantear la posible solución. Las pruebas deberían estar organizados de forma tal que ayuden a aislar errores.
* Ser certero o seguro, lo cual nos permita aumentar la confianza en el testing. La precisión dependerá de que tan especifica y concreta sea la prueba del testing.

**Criterio del testing:**

Para cada funcionalidad se agrupara cada una de las diferentes operaciones que realizara una función, de tal manera que realicemos como mínimo una prueba por operación.

**Estrategia a usar:**

La estrategia que se usara será la integración por dependencia de las funciones. Estrategia que consiste en testear las funciones que son independientes, las cuales estarán testeadas por el testing de unidad, para que una vez realizado se proceda a testear las funciones dependientes.

Para la creación de las pruebas de integración se deberán considerar los siguientes aspectos:

* Identificar la interacción de la funcionalidad, a nivel de sincronización de datos y integración funcional.
* La cantidad de errores existentes en el testing de unidad de la función.
* La importancia de la funcionalidad.
* Identificar la forma de acceso e invocación a la funcionalidad.

## Creación de pruebas de testing de integración:

Las pruebas deben ser creadas pretendiendo verificar la correcta funcionalidad de la función, lograr otorgar los datos requeridos y manipularlos y lograr una buena conexión con la base de datos.

## Plantilla para casos de prueba de testing de integración:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Datos Iníciales** | | | | | | | |
| **Caso de Uso:** | | | | **Nivel de Prueba:** | | | |
| **Autor de Funcionalidad:** | | | | **Encargado de Funcionalidad :** | | | |
| **Nombre de Escenario:** | | | | **Fecha de ejecución:** | | | |
| **Procedimiento Casos de Prueba** | | | | | | | |
| **Id Prueba** | **Historia** | **Descripción Prueba** | **Parámetros Entrada** | | **Resultado Esperado** | **Resultado Obtenido** | **Estado** |
| **1** |  |  |  | |  |  |  |
| **2** |  |  |  | |  |  |  |

## Explicación de plantilla:

**Caso de uso:** Se debe mencionar la funcionalidad que se probara.

* Ej: Registrar Mascota.

**Nivel de Prueba:** Se refiere al nivel de de prueba que es sometida, en nuestro caso será prueba de integración.

**Autor de funcionalidad:** Es la persona que realizo la funcionalidad.

**Encargado de la funcionalidad:** ES la persona que debe realizar la prueba de testing.

**Nombre de Escenario:** Se refiere al contexto de la función que se quiere

* Ej: Integrar la funcionalidad de mantener mascota.

**Fecha de Ejecución:** Se requiere a la fecha en la que se realizan las pruebas.

**Historia:** Se refiere a la prueba que se realizara.

* Ej: Consulta por la dirección de un cliente.

**Descripción de prueba:** Se describe la prueba a realizar.

**Parámetros de entrada:** Corresponde parámetros que se probaran.

**Resultado Esperado:** Corresponde al resultado que se espera de la prueba.

**Resultado Obtenido:** Corresponde al resultado obtenido de la prueba.

**Estado:** Corresponde al estado de la prueba. Los estados pueden ser los siguientes:

* **Aprobado:** La prueba ha sido aprobada.
* **Sin ejecutar:** La prueba no ha sido ejecutada.
* **Corregido:** La prueba ha sido corregida.
* **No Aprobado:** La prueba no ha sido aprobada.
* **Critico:** La prueba no ha sido aprobada y requiere ser corregida lo antes posible.

Referencias

**[1]** [**http://www.programacion.com/java/articulo/jap\_junit/**](http://www.programacion.com/java/articulo/jap_junit/)

**[2]** [**http://www.lab.dit.upm.es/~lprg/material/apuntes/pruebas/testing.htm**](http://www.lab.dit.upm.es/~lprg/material/apuntes/pruebas/testing.htm)