


**Metodología Desarrollo y Calidad**  
**Normativa Técnica Java**

*05 de Octubre de 2012*

 <b>JUNTA DE ANDALUCÍA</b> <small>CONSEJERÍA DE HACIENDA Y ADMINISTRACIÓN PÚBLICA</small>	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

## Hoja de Control del Documento

Información del Documento			
<b>Título</b>	Metodología Desarrollo y Calidad Normativa Técnica Java		
<b>Nombre del fichero</b>	DGPD_CCI_Normativa Técnica Java v01r01.odt		
<b>Versión</b>	1.0		
<b>Elaborado por</b>	Centro de Calidad Integral	<b>Fecha Elaboración</b>	03/07/2012
<b>Aprobado por</b>		<b>Fecha Aprobación</b>	
<b>Confidencialidad</b>			

Control de Versiones			
Versión	Descripción de los cambios	Elaborado por	Fecha Elaboración
1.0	Elaboración inicial del documento	Centro de Calidad Integral	03/07/2012
1.1	Actualización de las métricas a utilizar	Centro de Calidad Integral	05/10/2012

Lista de Distribución	
Apellidos, Nombre	Cargo / Función




## ÍNDICE

<b>1. INTRODUCCIÓN.....</b>	<b>5</b>
1.1. Objetivo.....	5
1.2. Alcance.....	5
1.3. Ámbito de Responsabilidades.....	5
<b>2. NORMATIVA DE CODIFICACIÓN Y MEJORES PRÁCTICAS.....</b>	<b>6</b>
2.1. Convenio de codificación general para todos los lenguajes.....	6
2.2. Construcción por Capas.....	7
2.2.1. Funcionalidades capa de presentación.....	7
2.2.1.1. Catálogo de componentes de interfaz.....	9
2.2.2. Funcionalidad de la capa de negocio.....	16
2.2.3. Funcionalidades de la capa de persistencia.....	17
2.3. Codificación.....	19
2.3.1. SONAR.....	20
2.3.2. Documentación en java.....	23
2.3.2.1. Javadoc.....	23
2.3.3. Normas de codificación en Java.....	25
2.3.4. Pautas de Codificación validadas por PMD, CheckStyle y FindBugs.....	35
<b>3. INTERFAZ DE USUARIO.....</b>	<b>46</b>
3.1. Esquema general de la aplicación.....	46
3.1.1. Pantallas de primer nivel.....	46
3.1.2. Pantallas de segundo nivel.....	47
3.2. Funcionamiento general de la aplicación.....	48
3.2.1. Acceso a la aplicación.....	48
3.2.2. Desconexión.....	48
3.2.3. Atrás.....	48
3.2.4. Listados.....	48
3.2.5. Formularios de introducción de datos.....	50
3.2.6. Ayuda.....	51
3.2.7. Ampliar imagen o su descripción.....	51
3.2.8. Nombre de usuario y desconexión.....	51
3.2.9. Anclajes.....	51
3.3. Prototipos de pantallas.....	51
3.4. Prototipos de manipulación de datos.....	53
3.4.1. Usuario y Contraseña.....	53
3.4.2. NIF.....	53
3.4.3. Correo electrónico.....	54
3.4.4. Teléfono.....	54
3.4.5. Dirección Postal.....	55
3.4.6. Código Cuenta Corriente.....	55
3.4.7. Campos numéricos.....	56
3.4.8. Fechas.....	56
3.4.9. Horas.....	56



3.4.10. URL.....	56
3.4.11. Nombre y Apellidos.....	57
3.5. Manual de estilo genérico.....	57
3.5.1. Gama cromática.....	57
3.5.2. Tipografía.....	57
3.5.3. Elementos gráficos.....	58
3.5.4. Creación de página.....	58
3.5.5. Buscador genérico.....	58
3.5.6. Tablas.....	58
3.5.7. Menú horizontal.....	58
3.5.8. Formularios.....	58
3.5.9. Enlaces.....	58
3.5.10. Listados.....	58
<b>4. SEGURIDAD EN JAVA.....</b>	<b>59</b>
4.1. Análisis y Diseño.....	59
4.2. Desarrollo.....	59
<b>5. RENDIMIENTO EN JAVA.....</b>	<b>67</b>
5.1. Rendimiento en las Bases de Datos.....	67
5.1.1. Particionamiento de Tablas.....	67
5.1.2. Detallar los atributos en las consultas, inserciones o actualizaciones.....	67
5.1.3. Uso de índices para la consultas.....	68
5.2. Rendimiento de los servicios WEB.....	68



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

## 1. INTRODUCCIÓN

### 1.1. Objetivo

El objetivo del documento es recopilar la normativa de codificación y mejores prácticas para los proyectos de desarrollo de la DGPD, así como definir una interfaz de usuario común para las aplicaciones.


### 1.2. Alcance

Todos los proyectos de desarrollo de la DGPD.

### 1.3. Ámbito de Responsabilidades

De aplicación a los proveedores y responsables de proyectos de sistemas de Información de la DGPD, así como por el Centro de Calidad Integral que velará por su cumplimiento en cada sistema o aplicación.



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

## 2. NORMATIVA DE CODIFICACIÓN Y MEJORES PRÁCTICAS

Se propone recopilar y unificar las reglas de codificación de los proyectos Java en un único documento así como incluir las mejores prácticas en este ámbito tecnológico para cada sistema de información o aplicación de la DGPD.

En general se siguen las recomendaciones realizadas en estos aspectos en MADEJA.

Esta normativa irá acompañada de las herramientas de soporte necesarias para asegurar su cumplimiento con el menor impacto tanto en los equipos de desarrollo como en el centro de calidad integral.


### 2.1. Convenio de codificación general para todos los lenguajes

Es muy común que el estilo de programación dependa en gran medida del lenguaje de programación que se haya elegido para desarrollar la aplicación, pero hay una serie de convenciones válidas para cualquier tipo de lenguaje. A continuación se ofrece un conjunto de pautas para mejorar las buenas prácticas en la codificación.



Nomenclatura de variables	Las variables deben nombrarse de manera que facilite la comprensión del código fuente. Por ello deben de mantener una cierta lógica con el modelo de negocio que representan. De esta manera, el desarrollador obtiene un código más intuitivo y que se puede mantener.
Uso de la indentación	Debe utilizarse un estilo de sangría, en lenguajes de programación que usan llaves para tabular o delimitar bloques lógicos de código. El uso de un estilo lógico y consistente hace que el código realizado sea más legible.
Controlar el tamaño del código fuente	Se debe de controlar el tamaño del código fuente. No debe sobrepasarse un determinado número de líneas de código porque implicaría probablemente un mal diseño. En el caso de Java, no deben de superarse las dos mil líneas de código.
Uso de variables booleanas en las estructuras de decisión	En estructuras de decisión se recomienda el uso de variables booleanas, ya que la no utilización de estas variables provoca que el código sea más complejo y difícil de entender, provocando incluso algunos errores en su concepción.
Uso de estructuras de control lógicas	El uso de estructuras de control lógicas para bucles también es parte de un buen estilo de programación. Ayuda a quien esté leyendo el código a entender la secuencia de ejecución (en programación imperativa).
Manejo de los espacios blancos	Los lenguajes de formato libre ignoran frecuentemente los espacios en blanco. El buen uso del espaciado en la disposición del código es, por tanto, considerado un buen estilo de programación.
Eliminar el uso de número mágicos	En ocasiones, se tiende a introducir números para realizar acciones condicionales o comprobaciones en el código. Es mucho más recomendable el uso de constantes para estas situaciones, de manera que el coste del mantenimiento del código se reduce ante cualquier modificación que afecte a la estructura.
Tamaño de los métodos	La longitud de un método no debe de exceder de cien líneas de código sin una causa justificada. En ocasiones, se intenta agrupar excesiva funcionalidad dentro de un mismo método. Esto provoca que la detección de un error sea compleja, dificultando la legibilidad y mantenimiento del código.
Número máximo de parámetros	Se recomienda no exceder de 10 parámetros en las llamadas a métodos si no existe una causa justificada para ello. De esta manera se asegura que los



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

	métodos no se encuentran sobrecargados de funcionalidad, favoreciendo el mantenimiento eficiente del código.
Búsqueda por cadenas Texto	Cuando se elaboren búsquedas por cadenas de texto no se debe hacer discriminación por mayúsculas, minúsculas o acentuación a la hora de obtener los registros asociados a la búsqueda.

## 2.2. Construcción por Capas

La programación por capas es un estilo de programación en el que el objetivo primordial es la separación de la lógica de negocios de la lógica de diseño. La ventaja principal de este estilo es que el desarrollo se puede llevar a cabo en varios niveles y, en caso de que sobrevenga algún cambio, sólo se ataca al nivel requerido sin tener que revisar entre código mezclado.



### 2.2.1. Funcionalidades capa de presentación

La capa de presentación es la que ve el usuario (también se la denomina "capa de usuario"), presenta el sistema al usuario, le comunica la información y captura la información del usuario en un mínimo proceso (realiza un filtrado previo para comprobar que no hay errores de formato). Esta capa se comunica únicamente con la capa de negocio. También es conocida como interfaz gráfica y debe tener la característica de ser "amigable" (comprensible y fácil de usar) para el usuario.



Separación de la lógica de negocio y presentación

La vista se debe limitar únicamente a la solicitud o presentación de información al usuario, y nunca deberá procesar ni tomar decisiones en ella. Esto será competencia de las capas inferiores de la aplicación.






Desacoplamiento entre capas	<p>Las clases Controlador solo deben contener lógica de presentación. No deben incluir lógica de negocio ni de persistencia. Las clases de acción (Action o Controller) deben contener sólo lógica de presentación. Deberá evitarse el acoplamiento de las capas de la aplicación, es decir, cada capa debe tomar la responsabilidad que le corresponde.</p> <p>Las vistas sólo pueden contener lógica de visualización. Al no permitirse scriptlet en las Vistas, la posibilidad de inyectar lógica de negocio o persistencia en estas disminuye.</p> <p>Las clases de acción o controladores (Action en Struts, Controller en JSF) deben contener sólo lógica de presentación. Por ello, no deberían tener una gran carga de desarrollo. Invocarán a clases de la lógica de negocio que encapsulen buena parte del código de la aplicación. En ningún caso debería haber lógica de persistencia en los Action o Controller.</p>
Modularidad de la vista	<p>Cada página no debe hacer referencia a nada que no esté relacionado con su propia funcionalidad. Una vez en ejecución se podrá integrar la página con la cabecera, pies y menú que le corresponda, aumentando así la reusabilidad de las páginas y el encapsulamiento de funcionalidad.</p>
Internacionalización	<p>Las aplicaciones deben estar preparadas para que puedan adaptarse a diferentes idiomas y convenciones (formatos de fecha, moneda, etc.) sin necesidad de realizar cambios en el código. Esto se consigue con la internacionalización (i18n).</p>
Validaciones en la capa de presentación	<p>La vista será la responsable de la primera validación de los datos introducidos por los usuarios. Las validaciones deben ser sólo de obligatoriedad de campos y validaciones formales.</p> <p>Nunca deberá realizarse una validación de negocio desde esta capa. La validación de campo obligatorio no va asociada al tipo de dato, sino a la función que se está codificando, por ello este tipo de validación se reflejará directamente en la página.</p> <p>En cambio, las validaciones de formato sí son independientes del contexto de la función.</p>
Conversiones en la capa de presentación	<p>Los formularios JSF recogen entradas de usuarios en forma de cadenas de caracteres. Estas cadenas para su uso en el interior de la aplicación, se deben convertir a los objetos adecuados, bien tipos base de Java (integer, long, double, boolean) o bien tipos complejos propios de la aplicación. Aquí se pueden encontrar dos situaciones:</p> <ul style="list-style-type: none"> <li>Algunos objetos de negocio complejos u objetos intermedios (objetos que se usen en la capa de presentación como un previo al paso de datos a la capa de negocio) se podrán incorporar directamente en las páginas. Para esto se asignarán conversores en los campos cuyo valor se vaya a usar para crear o modificar estos objetos de negocio complejos.</li> <li>Clases conocidas, bases de java, con las que se trabajará directamente en la aplicación y que necesitan conversión desde y hacia cadenas de caracteres: números decimales, fechas...</li> </ul> <p>En el primer caso será necesario crear en el proyecto los conversores apropiados normalmente, aunque pueden reutilizarse los que nos ofrezca el framework con el que implementemos la capa, mientras que en el segundo caso es más normal que puedan usarse los conversores que provee el framework en su especificación.</p>





	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java



Elaborar un catálogo de controles	<p>Los controles deben permitir el tratamiento como componentes Java que junto con el tratamiento de eventos nos posibilita el extraer de la vista toda la lógica de interfaz.</p> <p>La consecuencia principal de este enfoque es que desde la vista nunca se manipularán los controles. El objetivo es que la construcción de la vista sea una tarea totalmente autónoma del resto de componentes de la función de negocio (básicamente de los beans de respaldo) y luego todas las piezas se integren perfectamente sin grietas.</p> <p>Es recomendable elaborar un catálogo con la lista de controles oficiales. En ella estará totalmente especificada la función del control, los posibles validadores/conversores que se le puedan asociar, los eventos que pueda lanzar y los atributos que puedan cambiarse para manipular su aspecto externo.</p>
-----------------------------------	---

### 2.2.1.1. Catálogo de componentes de interfaz

El objetivo de esta área es la definición de los elementos, pautas de uso y diseño de los componentes más utilizados dentro del desarrollo de aplicaciones, normalizando los componentes de interfaz que podrán ser utilizados en los nuevos desarrollos.

#### ➤ Componentes básicos de un formulario



Campo de texto (atributos obligatorios)	<p>El campo de texto debe incluir obligatoriamente los siguientes atributos del elemento INPUT: <b>id</b>, <b>name</b>, <b>size</b>, <b>maxlength</b> y <b>tabindex</b>. Los campos de texto constituyen la principal forma de entrada de datos en un formulario, permitiendo al usuario introducir información no normalizada mediante caracteres alfanuméricos.</p> <p>Los atributos obligatorios que deberán añadirse a todos los campos de texto de un formulario son los siguientes:</p> <ul style="list-style-type: none"> <li>• <b>id</b>: Identificación del elemento, que permitirá cumplir ciertas pautas de accesibilidad como son el uso no intrusivo del javascript.</li> <li>• <b>name</b>: Nombre del campo, necesario para que el elemento sea identificado en el envío (submit) del formulario.</li> <li>• <b>size</b>: Tamaño del campo en la pantalla ajustado al número de caracteres que contendrá habitualmente el mismo.</li> <li>• <b>maxlength</b>: Longitud máxima de caracteres que podrán introducirse en el campo, evitando errores en el proceso de persistencia de la información por intentar almacenar un texto con una longitud mayor que la que soporte el campo de la base de datos.</li> <li>• <b>tabindex</b>: El índice de tabulación para cumplir con las pautas de usabilidad.</li> </ul> <p>En HTML5 existen nuevos atributos para este tipo de campo. Entre ellos, cuando proceda, serán obligatorios los siguientes:</p> <ul style="list-style-type: none"> <li>• <b>Required</b>: indica que es obligatorio introducir un valor en el campo para enviar ese formulario, algo que en XHTML y HTML4 se comprueba con Javascript o en el servidor.</li> <li>• <b>Min y Max</b>: el valor que se introduzca en el campo debe estar comprendido en un rango dado por los valores de estos atributos.</li> <li>• <b>Autofocus</b>: se utiliza en el primer elemento de un formulario para indicar que debe obtener el foco al cargar la página.</li> </ul>
---	---





Campo de texto (atributos opcionales)

Se recomienda incluir el atributo "accesskey" en ciertos campos de formulario más utilizados, para permitir un acceso directo por teclado a los mismos.

En HTML5 se han incorporado los siguientes cambios:

- **Pattern:** obliga a que el valor introducido en el campo cumpla con el patrón introducido.
- **Formaction, formenctype, formmethod, formnovalidate y formtarget:** atributos que modifican la acción, el método de envío, la validación y el destino de un formulario respectivamente cuando se introduce algún valor en el campo.
- **Data-\*,** donde el asterisco puede ser cualquier nombre: permite la creación de atributos personalizados, que posteriormente se pueden obtener con Javascript
- **Spellcheck:** indicará que el valor introducido en el componente debe pasar el corrector ortográfico.
- **Form:** contendrá el nombre del formulario al que pertenezca el elemento. De esta forma se puede colocar un componente en cualquier parte de una página.

Campo de texto de "sólo lectura"

Dado que el campo de texto en estado disabled aporta una visualización menor (texto en gris oscuro sobre cuadro gris), se utilizará el atributo readonly para la visualización de campos de texto en los que no se permita al usuario editar su contenido.

Botón de formulario

Para implementar botones de acción en una pantalla, no se podrán utilizar imágenes con un enlace sino que se realizará mediante botones haciendo uso de la etiqueta **button**.

Los tres tipos de botones disponibles son:

- **submit:** Cuando se activa realiza la presentación del formulario. Solamente puede haber un botón submit por formulario
- **button:** No tienen un comportamiento por defecto y están asociados a la ejecución de scripts.
- **reset:** Restablece los valores de los componentes del formulario a sus valores originales. No recomendado atendiendo a las pautas de usabilidad.

En HTML5 se han incorporado los siguientes atributos:

- **Formaction, formenctype, formmethod, formnovalidate y formtarget:** atributos que modifican la acción, el método de envío, la validación y el destino de un formulario respectivamente cuando se introduce algún valor en el campo.
- **Data-\*,** donde el asterisco puede ser cualquier nombre: permite la creación de atributos personalizados, que posteriormente se pueden obtener con Javascript
- **Form:** contendrá el nombre del formulario al que pertenezca el elemento. De esta forma se puede colocar un componente en cualquier parte de una página.

Selección booleana


Para campos de selección de tipo booleano se deberá utilizar un componente "checkbox".

```
<input type="checkbox" name="seleccion">
```

En HTML5 se han incorporado los siguientes atributos:


- **Formaction, formenctype, formmethod, formnovalidate y formtarget:** atributos que modifican la acción, el método de envío, la validación y el destino de un formulario respectivamente




	<h2 style="text-align: center;">Metodología Desarrollo y Calidad</h2>
	<h3 style="text-align: center;">Normativa Técnica Java</h3>


	<p>cuando se introduce algún valor en el campo.</p> <ul style="list-style-type: none"> <li>• <b>Data-*</b>, donde el asterisco puede ser cualquier nombre: permite la creación de atributos personalizados, que posteriormente se pueden obtener con Javascript</li> <li>• <b>Form:</b> contendrá el nombre del formulario al que pertenezca el elemento. De esta forma se puede colocar un componente en cualquier parte de una página.</li> <li>• <b>Required:</b> indica que es obligatorio introducir un valor en el campo para enviar ese formulario, algo que en XHTML y HTML4 se comprueba con Javascript o en el servidor.</li> </ul>
<p>Selección de dos o tres posibles valores</p>	<p>Para implementar una selección simple sobre un conjunto reducido de elementos (menor o igual a 3), se deberá utilizar una lista de componentes "radio" siguiendo el siguiente formato:</p> <pre>&lt;input type="radio" name="sexo" value="hombre"&gt;Hombre&lt;br&gt; &lt;input type="radio" name="sexo" value="mujer"&gt;Mujer</pre> <p>Se debe asegurar que el atributo <b>name</b> de todas las etiquetas es el mismo para que cuando sea realizado el <b>submit</b> el valor seleccionado sea único.</p> <p>En HTML5 se han incorporado los siguientes atributos:</p> <ul style="list-style-type: none"> <li>• <b>Formaction, formenctype, formmethod, formnovalidate y formtarget:</b> atributos que modifican la acción, el método de envío, la validación y el destino de un formulario respectivamente cuando se introduce algún valor en el campo.</li> <li>• <b>Data-*</b>, donde el asterisco puede ser cualquier nombre: permite la creación de atributos personalizados, que posteriormente se pueden obtener con Javascript</li> <li>• <b>Form:</b> contendrá el nombre del formulario al que pertenezca el elemento. De esta forma se puede colocar un componente en cualquier parte de una página.</li> <li>• <b>Required:</b> indica que es obligatorio introducir un valor en el campo para enviar ese formulario, algo que en XHTML y HTML4 se comprueba con Javascript o en el servidor.</li> </ul>
<p>Campo de tipo fecha</p>	<p>Para la introducción normalizada de fechas en un formulario se utilizará un componente compuesto por dos elementos:</p> <ul style="list-style-type: none"> <li>• Campo de texto con una longitud adecuada para incluir una fecha y hora si fuera necesario. Permitirá introducir el dato manualmente, admitiendo tanto el carácter '-' como el '/' como separador de día, mes y año. El formato de fecha predeterminado será DD/MM/AAAA y tendrá que ser validado tanto en el cliente como en el servidor.</li> <li>• Un enlace mediante un icono que permita visualizar una ventana emergente que contenga un componente Calendario. Este componente visualizará de forma predeterminada el día actual y permitirá seleccionar cualquier fecha, cargándola automáticamente en el campo de texto asociado.</li> </ul> <p>En HTML5 se han incorporado los siguientes cambios: Para los campos de tipo fecha, se ha desarrollado un tipo específico:</p> <pre>&lt;input type="date" ...&gt;</pre> <p>Este tipo verifica automáticamente que la fecha introducida sea una fecha válida.</p>




	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java


 <p>Campo de subida de fichero</p>	<p>La etiqueta html <code>&lt;input&gt;</code> con el tipo "file" permite definir en un formulario un campo de subida de fichero que será enviado al servidor mediante el envío del mismo.</p> <p>Para que el fichero asociado al campo se envíe junto con el formulario, será necesario establecer el tipo de contenido (<b>content type</b>) al valor "<b>multipart/form-data</b>", tal y como se observa en el siguiente ejemplo:</p> <pre>&lt;form id="formulario" action="/procesarPetición" method="multipart/form-data"&gt;     &lt;input id="fichero" name="fichero" type="file"&gt;     ... &lt;/form&gt;</pre>
<p>Área de texto</p>	<p>El componente área de texto permite introducir en un formulario una información alfanumérica que supere la longitud de una línea. Este componente es similar al campo de texto pero permite varias líneas con la posibilidad de una barra de desplazamiento vertical si fuera necesario. Alternativamente, se podrá habilitar un componente de editor enriquecido para que permita al usuario formatear el texto introducido. Este componente deberá cumplir las pautas de accesibilidad establecidas.</p> <p>En HTML5 se han incorporado los siguientes cambios:</p> <ul style="list-style-type: none"> <li>• <b>Formaction, formenctype, formmethod, formnovalidate y formtarget:</b> atributos que modifican la acción, el método de envío, la validación y el destino de un formulario respectivamente cuando se introduce algún valor en el campo.</li> <li>• <b>Data-*,</b> donde el asterisco puede ser cualquier nombre: permite la creación de atributos personalizados, que posteriormente se pueden obtener con Javascript</li> <li>• <b>Spellcheck:</b> indicará que el valor introducido en el componente debe pasar el corrector ortográfico.</li> <li>• <b>Form:</b> contendrá el nombre del formulario al que pertenezca el elemento. De esta forma se puede colocar un componente en cualquier parte de una página.</li> <li>• <b>Required:</b> indica que es obligatorio introducir un valor en el campo para enviar ese formulario, algo que en XHTML y HTML4 se comprueba con Javascript o en el servidor.</li> </ul>

### ➤ Elementos de agrupación de componentes


 <p>Tabla de datos</p>	<p>La forma más eficiente de presentar un gran número de elementos que comparten atributos y de los que se requiere relativa poca información, es mediante la utilización de tablas de resultados.</p> <p>Si bien las tablas (etiqueta <b>&lt;table&gt;</b>) no están permitidas para la maquetación de los elementos de una pantalla, son necesarias para la visualización de listados.</p> <p>Las tablas deberán incluir el atributo <b>summary</b> para informar al usuario del propósito de la misma por motivos de accesibilidad. La direccionalidad de una tabla (por defecto de izquierda a derecha) puede producirse mediante el atributo <b>dir</b>, aunque no se recomienda: <code>&lt;table dir="RTL"&gt;</code></p> <p>Característica que debe cumplir todo componente tabla:</p>
---	---





	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

	<ul style="list-style-type: none"> <li>• Siempre debe incluir una ordenación por defecto que debe denotarse mediante algún mecanismo en la columna correspondiente.</li> <li>• Se permitirá ordenar por cualquier de sus columnas de forma ascendente o descendente pulsando en la cabecera de la columna.</li> <li>• Admitirá selección simple o múltiple (o ninguna) según las circunstancias.</li> <li>• Incluirá una paginación de elementos a partir de un cierto número de filas configurable.</li> <li>• Permitirá acceder a una página concreta de elementos.</li> </ul> <p>La combinación de diferentes etiquetas <b>html</b> y de eventos gestionados tanto en cliente como en servidor, permiten implementar estas funcionalidades para el componente tabla de datos.</p>
Panel	<p>El panel será el elemento de agrupación y contenedor básico para todo formulario. Para agrupar y contener componentes de un formulario web en una pantalla se utilizará el componente <b>Panel</b>. Este componente está disponible en múltiples librerías de componentes en distintas tecnologías. El renderizado en html del Panel deberá ser una capa basada en la etiqueta <b>&lt;div&gt;</b>.</p>
Pestaña	<p>Aunque en general, las pestañas no están recomendadas para la agrupación de componentes en un formulario, se permite su utilización cuando las pantallas de modificación se reutilizan para la consulta de las entidades. En este caso, los formularios con muchos componentes y elementos distintos, presentados de una sola vez al usuario, no permiten una interacción cómoda con la pantalla y dificultan identificar datos individuales. En tal caso, se deberá utilizar pestañas ya que permiten agrupar información relacionada y presentarla a medida que el usuario la va requiriendo bajo petición (pulsando en la pestaña correspondiente).</p> <p>Si bien la pestaña no se corresponde con una etiqueta <b>html</b> específica, existen librerías de componentes en diferentes tecnologías que realizan implementaciones de este componente. En todos los casos, se utilizará la renderización html del componente generado deberá utilizar etiqueta <b>div</b> para definir la capa que contenga los componentes incluidos en la pestaña.</p> <p>En la siguiente imagen puede observarse un conjunto de pestañas que agrupan datos de un formulario. Mediante la utilización de estilos e imágenes es posible simular el efecto de pestaña seleccionada y pestaña desactivada:</p> 
Separador	<p>Cuando se tenga la necesidad de diferenciar apartados de una pantalla se puede hacer uso de una barra horizontal denominada Separador. El componente de interfaz <b>Separador</b> permitirá separar apartados dentro de un mismo formulario mediante la visualización de una barra o línea separadora.</p> <p>En <b>html</b> existe la etiqueta <b>&lt;hr&gt;</b> que permite realizar esta labor, aunque mediante la utilización de CSS es posible implementar un Separador.</p>




 <b>JUNTA DE ANDALUCÍA</b> <small>CONSEJERÍA DE HACIENDA Y ADMINISTRACIÓN PÚBLICA</small>	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

## ➤ Componentes de Selección


<p>Pocas opciones posibles</p> 	<p>La elección de un único elemento de una lista de pocos valores se realizará mediante un componente de lista desplegable. El formato a utilizar seguirá el siguiente ejemplo:</p> <pre>&lt;select id="provincia" name="provincia"&gt;   &lt;option value="Almeria"&gt;Almeria&lt;/option&gt;   &lt;option value="Cádiz"&gt;Cádiz&lt;/option&gt;   &lt;option value="Córdoba"&gt;Córdoba&lt;/option&gt;   &lt;option value="Granada"&gt;Granada&lt;/option&gt;   &lt;option value="Huelva"&gt;Huelva&lt;/option&gt;   &lt;option value="Jaen"&gt;Jaen&lt;/option&gt;   &lt;option value="Málaga"&gt;Málaga&lt;/option&gt;   &lt;option value="Sevilla"&gt;Sevilla&lt;/option&gt; &lt;/select&gt;</pre> <p>En HTML5 se han incorporado los siguientes cambios:</p> <ul style="list-style-type: none"> <li>• <b>Formaction, formenctype, formmethod, formnovalidate y formtarget:</b> atributos que modifican la acción, el método de envío, la validación y el destino de un formulario respectivamente cuando se introduce algún valor en el campo.</li> <li>• <b>Data-*</b>, donde el asterisco puede ser cualquier nombre: permite la creación de atributos personalizados, que posteriormente se pueden obtener con Javascript</li> <li>• <b>Form:</b> contendrá el nombre del formulario al que pertenezca el elemento. De esta forma se puede colocar un componente en cualquier parte de una página.</li> <li>• <b>Required:</b> indica que es obligatorio introducir un valor en el campo para enviar ese formulario, algo que en XHTML y HTML4 se comprueba con Javascript o en el servidor.</li> </ul>
<p>Muchas opciones de selección posibles</p>	<p>Dado que no es operativo realizar una selección sobre una lista desplegable con muchos elementos, se considerará la utilización de un componente de campo de texto con "escritura predictiva". Así pues, se presentará al usuario un campo de texto normal que, a medida que se escribe, vaya comparando el texto introducido con los datos posibles, presentando dichos datos en una lista posicionada justo debajo del campo de texto. Esta lista permitirá seleccionar al usuario la opción que desea sin necesidad de introducirla completamente.</p>
<p>Opciones dependientes de otro campo</p> 	<p>Cuando las opciones de una lista desplegable dependan de un campo anterior, se informará claramente en el formulario de esta circunstancia para que el usuario sea consciente y pueda realizar la selección previa. Para cumplir con las pautas de usabilidad, en el componente de lista desplegable dependiente se deberá informar al usuario de que otro campo requiere ser informado previamente.</p> <p>El campo principal y la lista dependiente deberán disponerse de forma adyacente uno después del otro.</p>
<p>Selección múltiple</p>	<p>Cuando el usuario tiene a su disposición varias opciones para elegir pudiendo seleccionar más de una, se utilizará un componente de selección múltiple. Si el número de elementos de la lista es pequeño, se utilizará un componente de lista donde se permita seleccionar varios elementos de la misma.</p> <p>La utilización de la etiqueta &lt;select&gt; con el atributo "multiple" será una posible opción si se va a realizar una selección múltiple de elementos de una lista que contenga pocos elementos. El formato sería como en el siguiente ejemplo:</p> <pre>&lt;SELECT multiple="multiple" size="4" name="component-select"&gt;</pre>



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

	<pre>&lt;OPTION selected value="Component_1_a"&gt;Component_1&lt;/OPTION&gt; &lt;OPTION selected value="Component_1_b"&gt;Component_2&lt;/OPTION&gt; &lt;OPTION&gt;Component_3&lt;/OPTION&gt; &lt;OPTION&gt;Component_4&lt;/OPTION&gt; &lt;OPTION&gt;Component_5&lt;/OPTION&gt; &lt;OPTION&gt;Component_6&lt;/OPTION&gt; &lt;OPTION&gt;Component_7&lt;/OPTION&gt; &lt;/SELECT&gt;</pre> <p>En HTML5 se han incorporado los siguientes cambios:</p> <ul style="list-style-type: none"> <li>• <b>Formaction, formenctype, formmethod, formnovalidate y formtarget:</b> atributos que modifican la acción, el método de envío, la validación y el destino de un formulario respectivamente cuando se introduce algún valor en el campo.</li> <li>• <b>Data-*</b>, donde el asterisco puede ser cualquier nombre: permite la creación de atributos personalizados, que posteriormente se pueden obtener con Javascript</li> <li>• <b>Form:</b> contendrá el nombre del formulario al que pertenezca el elemento. De esta forma se puede colocar un componente en cualquier parte de una página.</li> <li>• <b>Required:</b> indica que es obligatorio introducir un valor en el campo para enviar ese formulario, algo que en XHTML y HTML4 se comprueba con Javascript o en el servidor.</li> </ul>
--	--

## ➤ Otros Componentes

<p>Barra de progreso</p> 	<p>Para informar al usuario del grado de avance de una cierta tarea que esté ejecutando la aplicación, se recomienda utilizar un componente de barra de progreso que transmita esta información.</p> <p>La barra de progreso permite al usuario conocer el grado de avance de una tarea que generalmente tiene duración elevada y le da la posibilidad de determinar el tiempo restante para su finalización. Evita la frustración del usuario al tener que esperar a la finalización de una acción sobre la aplicación sin ningún tipo de información.</p> <p>En HTML5 se han incorporado los siguientes cambios:</p> <p>Para generar una barra de progreso se ha desarrollado un componente específico "<b>progress</b>". En conjunción con JavaScript puede representar el progreso de una tarea o proceso. Sus atributos son:</p> <ul style="list-style-type: none"> <li>• <b>Value:</b> especifica la cantidad completada de la tarea</li> <li>• <b>Max:</b> especifica la cantidad de trabajo total de la tarea</li> </ul> <p>Ejemplo:</p> <pre>&lt;progress id="p" max="100" value="0"&gt;&lt;/progress&gt;</pre>
--	--





### 2.2.2. Funcionalidad de la capa de negocio

La definición de los límites de cada capa nos permitirá definir correctamente la colaboración que proveerá cada una de ellas y descubriremos que la capa intermedia es inevitablemente la lógica de negocios. Esto permitirá obtener una infraestructura robusta y lista para la extensión y el crecimiento como proveedora de servicios. Existen tres tipos de componentes de negocio fundamentales:

- Reglas de negocio, implementan la funcionalidad de negocio del sistema.
- Entidades de negocio, representan las entidades del sistema.
- Workflow, implementan procesos de negocio en los cuales participan entidades y lógica de negocio.



Elementos que conforman la capa de negocio	La capa de negocio consta de la lógica del sistema: reglas de negocio, workflow de negocio, operaciones no persistentes. Todas las operaciones persistentes son delegadas a la capa de acceso de datos.
Reglas de negocio	Toda regla de negocio debe ser implementada en esta capa. La capa de negocio debería ser vista como un conjunto de servicios expuestos a las capas de presentación de las aplicaciones. La idea es que todos los módulos que requieren una funcionalidad la encuentren en un solo lugar y con una sola versión, no hay réplica de funcionalidades en un formulario u otro lugar.
Validaciones de los datos	Esta capa debe de garantizar que los datos requeridos para procesarla fueron debidamente validados, y sólo si es exitosa la validación se puede iniciar el flujo del proceso de negocio. Para ello debe comprobarse la validez de los datos obtenidos desde la capa de presentación para preservar a la aplicación de ataques de código malintencionado.
Comunicaciones entre capas. La entidad de negocio.	Es muy importante definir la estrategia de comunicación entre las capas. Se crean entidades que carecen de métodos, solo tienen propiedades, campos y posiblemente atributos, que nos pueden servir para almacenar la información como la asociación de las propiedades a las columnas de la base de datos. La comunicación entre capas se establece mediante objetos del modelo.
Transacciones	El manejo de las transacciones debe de realizarse desde la capa de negocio, no desde la capa de acceso a datos. La capa de acceso a datos proporciona los datos pero las transacciones de los mismos se manejan desde la capa de negocio.
Tratamiento de excepciones en la capa	Se debe de establecer un modelo de excepciones que se propague a la capa de presentación. Las excepciones son generadas desde la capa de acceso a datos y deben ser encapsuladas en esta capa para trasladarlas de forma correcta a la capa superior.
Encapsular la capa en servicios	Los servicios de negocio son el "puente" entre un usuario y los servicios de datos. Responden a peticiones del usuario (u otros servicios de negocio) para ejecutar una tarea de este tipo. Cumplen con esto aplicando procedimientos formales y reglas de negocio a los datos relevantes. Cuando los datos necesarios residen en un servidor de bases de datos, garantizan los servicios de datos indispensables para cumplir con la tarea de negocio o aplicar su regla. Esto aísla al usuario de la interacción directa con la base de datos.
Control sobre el uso de	El control de acceso al servicio de negocio debe hacerse en la capa de negocio, puesto que podemos tener distintas capas de presentación. El





servicios

control se puede hacer a nivel de servicio vertical (cada fachada) o a nivel de método dentro de cada servicio.

### 2.2.3. Funcionalidades de la capa de persistencia

Si la aplicación está diseñada con orientación a objetos, la persistencia se logra por serialización del objeto o almacenamiento en una base de datos. Las bases de datos más populares hoy en día son relacionales.

El modelo de objetos difiere en muchos aspectos del modelo relacional. La interfaz que une esos dos modelos se llama asociación objeto-relacional (ORM en inglés). Una capa de persistencia encapsula el comportamiento necesario para mantener los objetos. O sea: leer, escribir y borrar objetos en el almacenamiento persistente (base de datos).

#### Asociación Objeto-Relacional

Se debe usar un mapeador Objeto Relacional para implementar la capa de persistencia.

La asociación objeto-relacional (más conocido por su nombre en inglés, Object-Relational Mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). Hay paquetes comerciales y de uso libre disponibles que desarrollan la asociación relacional de objetos, aunque algunos programadores prefieren crear sus propias herramientas ORM.

Un objeto está compuesto de propiedades y métodos. Como las propiedades representan a la parte estática de ese objeto, son las partes que se dotan de persistencia. Cada propiedad puede ser simple o compleja.

- Por simple, se entiende que tiene algún tipo de datos nativos como por ejemplo: entero, coma flotante o cadena de caracteres.
- Por complejo se entiende algún tipo definido por el usuario, ya sean objetos o estructuras.


Por relación se entiende asociación, herencia o agregación. Para dotar de persistencia las relaciones, se usan transacciones, ya que los cambios pueden incluir varias tablas.

Para vincular las relaciones, se usan los identificadores de objetos (OID). Estos OID se agregan como una columna más en la tabla donde se quiere establecer la relación. Dicha columna es una clave foránea a la tabla con la que se está relacionada. Así, queda asignada la relación. Recordar que las relaciones en el modelo relacional son siempre bidireccionales.


#### Uso del patrón DAO

Se recomienda crear una clase DAO por cada objeto de negocio del sistema. El patrón CRUD, reconocido como el patrón más importante del acceso a datos indica que cada objeto debe ser creado en base de datos para que sea persistente. Para esto es necesario asegurar que existen operaciones que permiten a la capa inferior (de acceso a datos) leerlo, actualizarlo o simplemente borrarlo.

Mediante la implementación de DAO's pueden proporcionarse las

	operaciones CRUD necesarias para cada aplicación. No es obligatorio que cada DAO implemente todas las operaciones CRUD (puede no ser necesario en la lógica funcional del sistema). En general (aunque esto es una decisión de diseño), por cada objeto de negocio en el sistema, crearemos un DAO distinto.
Manejo de la cache	<p>Se recomienda el uso de la cache en las aplicaciones, para reducir tiempos de lectura. Pero con precaución ante los posibles problemas de integridad de datos que se pueden dar.</p> <p>En la mayoría de las aplicaciones, se aplica la regla del 80-20 en cuanto al acceso a datos, el 80% de accesos de lectura accede al 20% de los datos de la aplicación. Esto significa que hay un conjunto de datos dinámicos que son relevantes a todos los usuarios del sistema, y por lo tanto accedido con más frecuencia. Las aplicaciones de sincronización de caché normalmente necesitan escalarse para manejar grandes cargas transaccionales. Así, múltiples instancias se pueden procesar simultáneamente.</p> <p>Es un problema serio para el acceso a datos desde la aplicación, especialmente cuando los datos involucrados necesitan actualizarse dinámicamente a través de esas instancias. Para asegurar la integridad de datos, la base de datos comúnmente juega el rol de árbitro para todos los datos de la aplicación. Es un rol muy importante dado que los datos representan la parte de valor más significativa de una organización. Desafortunadamente, este rol no está fácilmente distribuido sin introducir problemas importantes, especialmente en un entorno transaccional.</p> <p>Es común para la base de datos usar replicación para lograr datos sincronizados, pero comúnmente ofrece una copia offline del estado de los datos más que una instancia secundaria activa. Es posible usar bases de datos que puedan soportar múltiples instancias activas, pero se pueden volver caras en cuanto a mantenimiento y escalabilidad, debido a que introducen el bloqueo de objetos y la latencia de distribución. La mayoría de los sistemas usan una única base de datos activa, con múltiples servidores conectados directamente a ella, soportando un número variable de clientes.</p> <p>En esta arquitectura, la carga en la base de datos se incrementará linealmente con el número de instancias de la aplicación en uso, a menos que se emplee alguna caché. Pero implementar un mecanismo de caché en esta arquitectura puede traer muchos problemas, incluso corrupción en los datos, porque la caché en el servidor 1 no conocerá los cambios en el servidor 2.</p>
 <p>Permitir la concurrencia de usuarios</p>	<p>Para la mayoría de los casos se recomienda el uso del bloqueo optimista, que es soportado por la mayoría de los motores de persistencia como la opción por defecto. El bloqueo pesimista se empleará solo en casos concretos.</p> <p>La capa de persistencia debe permitir que múltiples usuarios trabajen en la misma base de datos y proteger los datos de ser escritos erróneamente. También es importante minimizar las restricciones en su capacidad concurrente para ver y acceder.</p> <p>La integridad de datos es un riesgo cuando dos sesiones trabajan sobre la misma tupla: la pérdida de alguna actualización está asegurada. También se puede dar el caso, cuando una sesión está leyendo los datos y la otra los está editando: una lectura inconsistente es muy probable.</p> <p>Hay dos técnicas principales para el problema: bloqueo pesimista y bloqueo optimista. Con el primero, se bloquea todo acceso desde que el usuario</p>



 <b>JUNTA DE ANDALUCÍA</b> <small>CONSEJERÍA DE HACIENDA Y ADMINISTRACIÓN PÚBLICA</small>	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

	<p>empieza a cambiar los datos hasta que se hace COMMIT en la transacción. Mientras que en el optimista, el bloqueo se aplica cuando los datos son aplicados y se van verificando mientras los datos son escritos.</p> <p>En la mayoría de los casos el bloqueo optimista es suficiente, controlando los lost-updates. Para casos concretos, por ejemplo en los que se necesite generar algo como un número de factura sin que queden huecos entre uno y otro se podría usar el bloqueo pesimista.</p>
Evitar las referencias circulares entre objetos	Evitar las referencias circulares entre objetos de la capa de persistencia. Se debe de intentar evitar las referencias circulares, facilitando la localización de los objetos. De esta manera, se devuelve el objeto solicitado sin necesidad de realizar un recorrido para localizarlo, lo que supone un acceso más efectivo.
Buen uso de la información oculta	No mostrar aquellos datos que se almacenan en un objeto, por motivos internos al sistema, pero que no pertenecen al modelo de datos en sí. En ocasiones hay columnas en la tabla de BBDD que no necesitan ser vinculadas a una propiedad del objeto. Columnas que contienen información necesaria pero que no forman parte del modelo de objetos. En esta categoría entran los mecanismos de concurrencia: fecha y versión de objeto. Al leer el objeto para mostrarlo, esta información que es mantenida por el framework no tiene por que mostrarse.
Actualización en cascada	Utilizar el borrado en cascada siempre que sea posible. Utilizar la posibilidad que ofrecen los frameworks de la actualización en cascada de objetos. Una actualización en cascada permite que las modificaciones hechas a un objeto se repliquen en los objetos relacionados. De esta manera se mejora el mantenimiento de los objetos, se asegura que los cambios introducidos se replican de manera eficiente manteniendo la integridad de los datos.

### 2.3. Codificación

Como ya se ha visto en el procedimiento de Gestión de la Configuración e Integración Continua en proyectos Java, para automatizar las verificaciones del código se recomienda la utilización de la herramienta **SONAR**. Existen para sonar una serie de plugins como **pmd**, **checkstyle**, **findbugs** que las facilitan.

Los objetivos que se persiguen con la estandarización de la codificación son:

- Promover la generación de código fuente de calidad.
- Unificar el uso de librerías y utilidades de apoyo.
- Proponer plugins para el desarrollo con IDEs.
- Promover el uso de patrones de diseño software

A través de las recomendaciones de MADEJA, se dispone de una matriz de verificación, en función del área al que pertenecen, que recogen las pautas de codificación vistas en los puntos anteriores:

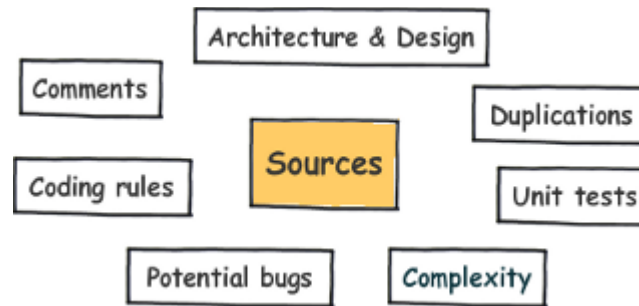
<http://www.juntadeandalucia.es/servicios/madeja/node/1405/download/MET001EVERVerificacionesMADEJA.ods.zip>

Para usar estas verificaciones en **SONAR**, puede descargarse el perfil de la herramienta desde MADEJA: <http://www.juntadeandalucia.es/servicios/madeja/node/1406/download/perfilsonarMADEJA.rar>



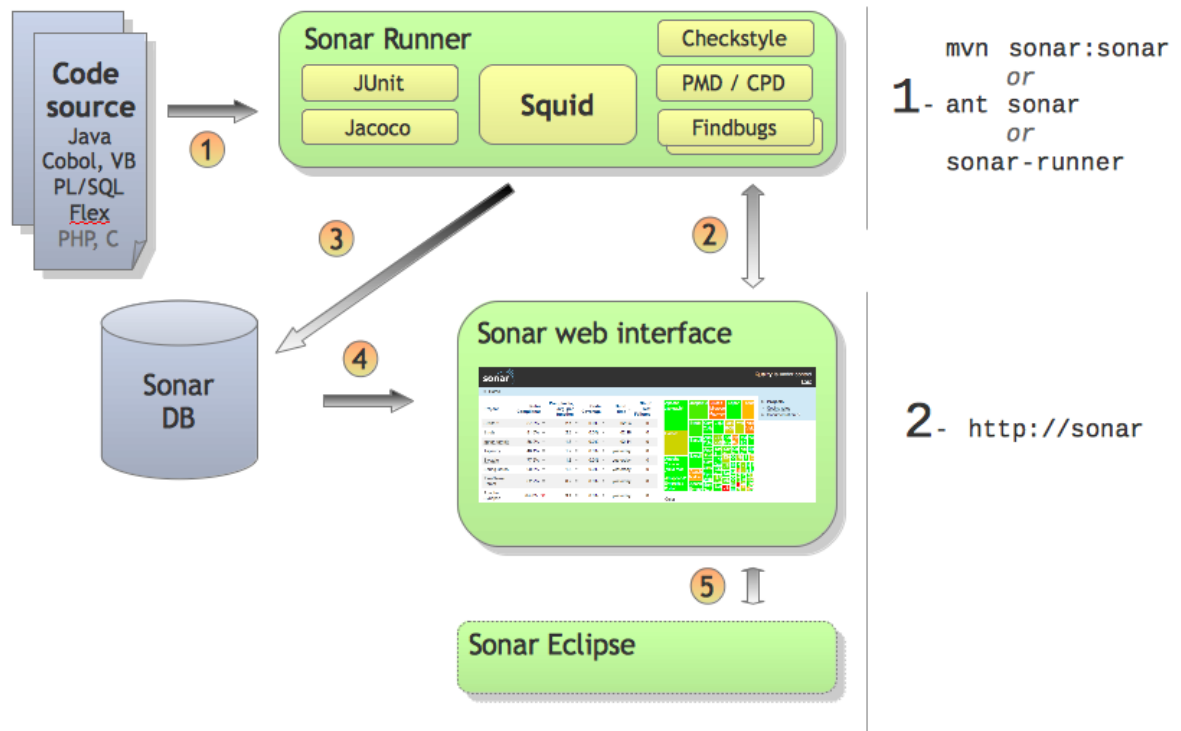
### 2.3.1. SONAR

Según su propia web, Sonar es una plataforma que permite gestionar la calidad del código controlando los 7 ejes principales de dicha calidad del código:

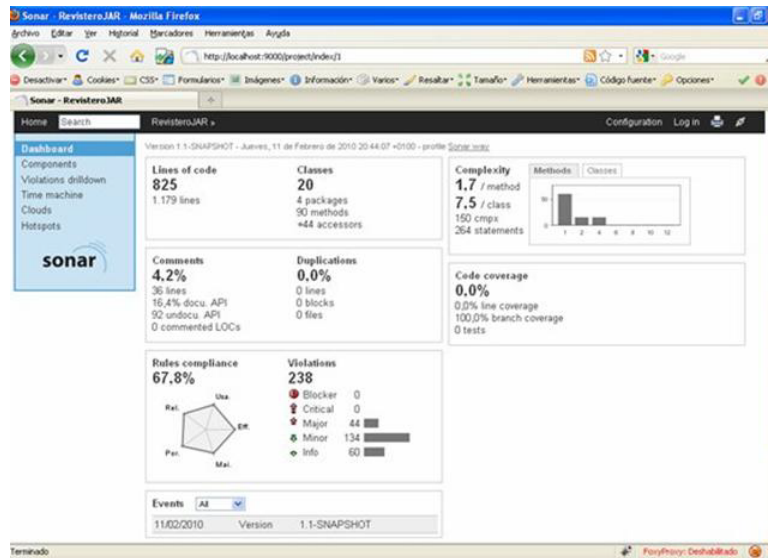


- Arquitectura y diseño
- Duplicaciones
- Pruebas unitarias
- Complejidad
- Errores potenciales
- Reglas de codificación
- Comentarios

**Sonar** realiza varios análisis del código a través de otras herramientas (Checkstyle, PMD, Cobertura...) y presenta de manera unificada a través de su interfaz la información generada por ellas en forma de métricas.



A través de la interfaz de Sonar podemos ver de forma detallada los puntos débiles de los proyectos como errores potenciales en el código, escasez de comentarios, clases demasiado complejas, escasez de cobertura de las pruebas unitarias, y más.

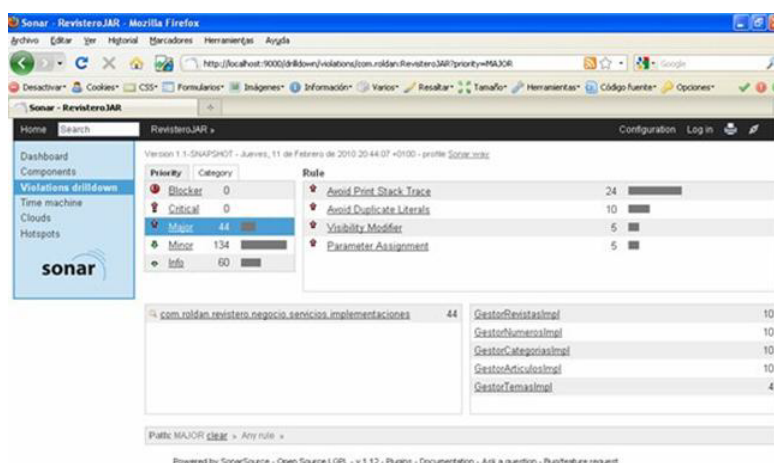


Aquí se empieza a ver la potencia de Sonar. En una interfaz compacta nos muestra una cantidad importante de información sobre el código del proyecto como:

- Medida en líneas de código
- Complejidad de los componentes
- Cantidad de comentarios introducidos
- Duplicidad del código
- Cobertura del código
- Conformidad respecto a ciertas reglas de codificación
- Violaciones del código

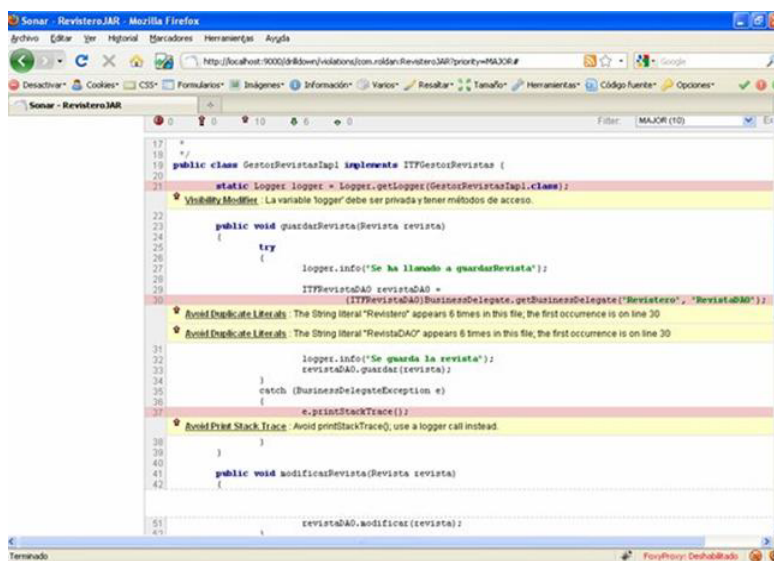
Para ampliar la información, podemos, por ejemplo, acceder a las violaciones tipo **GRAVE** de las reglas de codificación:





Terminado

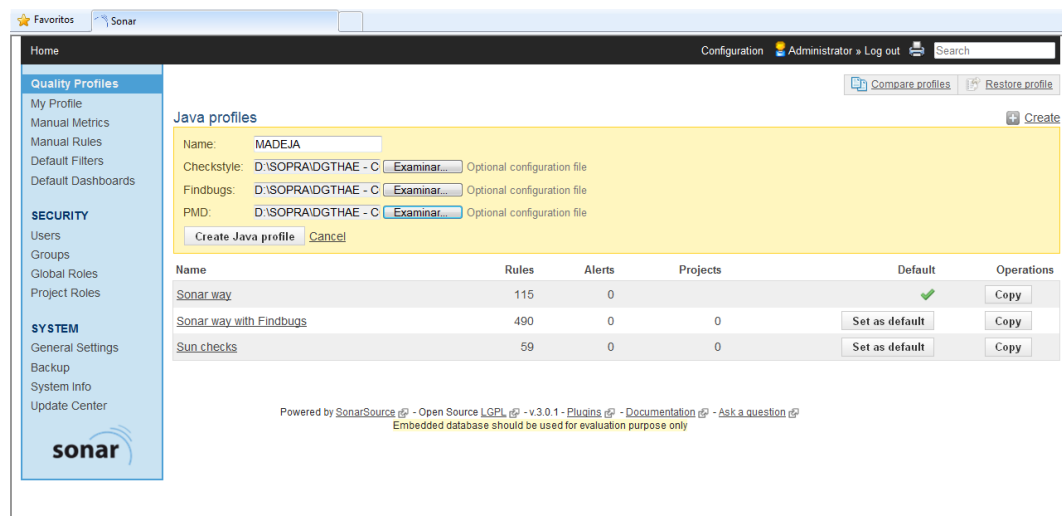
Sonar nos muestra información sobre las reglas que se están incumpliendo, y los paquetes y clases en que esto ocurre. Y, si pinchamos sobre una de las clases **Sonar** muestra cuál es el código que está incumpliendo estas reglas y da indicaciones para corregirlo:



Para incorporar las pautas de codificación definidas en MADEJA, nos conectaremos a la aplicación como administrador y, en la opción "Configuration" nos aparecerá una lista de los perfiles existentes en sonar.

Creamos un nuevo perfil "MADEJA" incorporando los tres ficheros XML que hemos bajado desde la url indicada anteriormente:





**Java profiles**

Name: MADEJA

Checkstyle: D:\SOPRAIDGTHAE - C [Examinar...] Optional configuration file

Findbugs: D:\SOPRAIDGTHAE - C [Examinar...] Optional configuration file

PMD: D:\SOPRAIDGTHAE - C [Examinar...] Optional configuration file

Create Java profile Cancel

Name	Rules	Alerts	Projects	Default	Operations
Sonar_way	115	0		✓	Copy
Sonar_way with Findbugs	490	0	0	Set as default	Copy
Sun_checks	59	0	0	Set as default	Copy

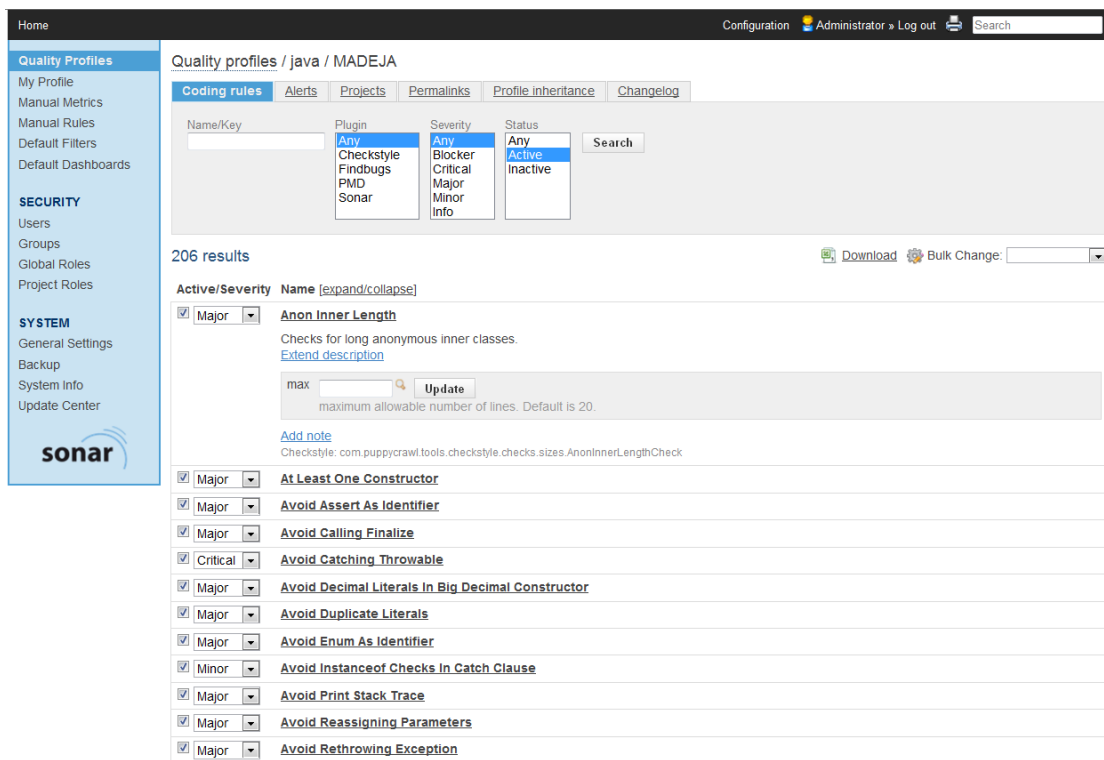
Powered by SonarSource - Open Source LGPL - v.3.0.1 - Plugins - Documentation - Ask a question

Embedded database should be used for evaluation purpose only

**Java profiles**

Name	Rules	Alerts	Projects	Default	Operations
MADEJA	206	0		✓	Backup Rename Copy
Sonar_way	115	0	0	Set as default	Copy
Sonar_way with Findbugs	490	0	0	Set as default	Copy
Sun_checks	59	0	0	Set as default	Copy

Pudiendo acceder a cada una de las pautas:



Quality profiles / java / MADEJA

Coding rules Alerts Projects Permalinks Profile inheritance Changelog

Name/Key Plugin Severity Status Search

Any Any Any

Checkstyle Blocker Active

Findbugs Critical Inactive

PMD Major

Sonar Minor Info

206 results

Download Bulk Change:

Active/Severity Name [expand/collapse]

Major Anon Inner Length

Checks for long anonymous inner classes.

Extend description

max Update

maximum allowable number of lines. Default is 20.

Add note

Checkstyle: com.puppycrawl.tools.checkstyle.checks.sizes.AnonInnerLengthCheck

Major At Least One Constructor

Major Avoid Assert As Identifier

Major Avoid Calling Finalize

Critical Avoid Catching Throwable

Major Avoid Decimal Literals In Big Decimal Constructor

Major Avoid Duplicate Literals

Major Avoid Enum As Identifier

Minor Avoid Instance of Checks In Catch Clause

Major Avoid Print Stack Trace

Major Avoid Reassigning Parameters

Major Avoid Rethrowing Exception

## 2.3.2. Documentación en java

### 2.3.2.1. Javadoc

JAVADOC, es una herramienta del SDK que permite documentar de una manera rápida y sencilla las clases y métodos que se proveen. Es de gran utilidad para la comprensión del desarrollo. Por





ello debemos de estandarizar su uso para facilitar la comprensión de forma global entre todos los desarrolladores.

A continuación definimos una serie de reglas elementales:



Los comentarios de JAVADOC se generan desde el código fuente y por lo tanto hay que incluir etiquetas especiales para poder interpretarlas en la generación. La etiqueta que determina un comentario JAVADOC es `/**..*/`

Es preferible el uso de la tercera persona en los comentarios, ya que la documentación suele estar destinada a un público amplio

Se ubican siempre antes de los clases, métodos, interfaces y atributos a describir

Un comentario JAVADOC está compuesto de una definición seguida de un bloque de Tags relacionadas

Las etiquetas principales que se usan en Javadoc son:



@author	Autor del elemento a documentar
@version	Versión del elemento de la clase
@return	Indica los parámetros de salida
@exception	Indica la excepción que puede generar
@param	Código para documentar cada uno de los parámetros
@see	Una referencia a otra clase o utilidad
@deprecated	El método ha sido reemplazado por otro

## ➔ Comentarios de Clases



Antes de la declaración de la clase se introducirá un comentario de documentación que contenga la descripción de la clase y el autor, la versión y la fecha

### ➤ Ejemplos:

```
/**
 * Frase corta descriptiva
 * Descripción de la clase
 * @author Nombre Apellido / Empresa
 * @version 0.1, 2004/05/30
 */
```

## ➔ Comentarios de Métodos







Especificar descripción, parámetros, tipo de retorno, excepciones que se lanzan. Si es importante dentro del método la llamada a otro método, también se referenciará. No es necesario documentar los métodos GET/SET.

➤ **Ejemplos:**

```
/**  
 * Frase corta descriptiva  
 * Descripción del método.  
 * Mención al uso{@link es.loquesea.$app.util.Otra#unMetodo unMetodo}.  
 * @param param1 descripción del parámetro.* @return qué devuelve el método.  
 * @exception tipo de excepción que lanza el método y en qué caso  
 * @see paquete.Clase#metodo Código al que se hace referencia  
 * @throws IllegalArgumentException el param1 no tiene el formato deseado  
 */
```

### 2.3.3. Normas de codificación en Java

- Utilizar verbos o sintagmas verbales al asignar nombres a los métodos.
- No utilizar nombres que requieran distinción entre mayúsculas y minúsculas. Los componentes se deben poder utilizar en los lenguajes que distinguen, y en los que no distinguen, entre mayúsculas y minúsculas. Los lenguajes que no hacen esta distinción no pueden diferenciar, dentro del mismo contexto, dos nombres que difieren sólo en el uso de mayúsculas y minúsculas.
- No crear una función con nombres de parámetros que difieran sólo en las mayúsculas y minúsculas. El siguiente ejemplo es incorrecto.



```
void myFunction(string a, string A)
```

- En ningún caso utilizar nombres de métodos que difieran sólo en las mayúsculas y minúsculas.



```
void calculate();  
void Calculate();
```

Esta norma será revisada por Sonar a través de la regla Error: no se encontró el origen de la referencia.

➤ **Sentencias Simples.**



Cada línea debe contener como máximo una sentencia:



```
argv++;    // Correcto
argc++;    // Correcto
```



```
argv++; argc--; // Incorrecto
```

Esta norma será revisada por Sonar a través de la regla **Error: no se encontró el origen de la referencia**

### ➤ Sentencias Compuestas.

Las sentencias compuestas contienen una lista de sentencias simples encerradas entre llaves:

- Las sentencias simples encerradas entre llaves, deben indentarse uno o más niveles que la sentencia compuesta.
- La llave de apertura debe estar al final de la línea que empieza la sentencia compuesta; La llave de cierre debe empezar una nueva línea y estar indentado con el principio de la sentencia compuesta.
- Las llaves se usan alrededor de todas las sentencias, incluso para sentencias simples, cuando éstas forman parte de una estructura de control como una sentencia if-else o for.

Estas normas serán revisadas por Sonar a través de las reglas **LeftCurly, RightCurly y NeedBraces**

### ➤ Sentencias de Retorno.

Una sentencia de retorno no debe usar paréntesis a menos que el valor de retorno sea más obvio de esa forma.



```
return;
return myDisk.size();
```

### ➤ Sentencias IF / IF-ELSE.

En las sentencias del tipo IF/IF-ELSE tanto el cuerpo para el IF como el del ELSE siempre deben incorporar la apertura y cierre de llaves aunque solamente tengan una instrucción. Deben tener la siguiente estructura:



```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}
```



```
if (condition) {  
    statements;  
} else if (condition) {  
    statements;  
} else {  
    statements;  
}
```

Esta norma será revisada por Sonar a través de las reglas **LeftCurly**, **RightCurly** y **NeedBraces**

#### ➤ Sentencias FOR.

Las sentencias del tipo FOR siempre deben incorporar la apertura y cierre de llaves aunque solamente tengan una instrucción. Debe tener la siguiente estructura:



```
for (initialization; condition; update) {  
    statements;  
}
```

Un caso excepcional es la sentencia FOR vacía, es decir, aquella en la que todo el trabajo se hace en las cláusulas de inicialización, condición y actualización. Ésta debe tener la siguiente estructura:



```
for (initialization; condition; update);
```

Esta norma será revisada por Sonar a través de las reglas **LeftCurly**, **RightCurly** y **NeedBraces**

#### ➤ Sentencias WHILE.

Las sentencias del tipo WHILE siempre deben incorporar la apertura y cierre de llaves aunque solamente tengan una instrucción. Debe tener la siguiente estructura:



```
while (condition) {  
    statements;  
}
```

Un caso excepcional es la sentencia while vacía, que debería tener la siguiente forma:



```
while (condition);
```

Esta norma será revisada por Sonar a través de las reglas **LeftCurly**, **RightCurly** y **NeedBraces**.

#### ➤ Sentencias DO-WHILE.

Las sentencias del tipo DO-WHILE siempre deben incorporar la apertura y cierre de llaves aunque solamente tengan una instrucción. Debe tener siempre la siguiente estructura:



```
do {  
    statements;  
} while (condition);
```

No se permitirán estructuras DO-WHILE vacías.

Esta norma será revisada por Sonar a través de las reglas **LeftCurly**, **RightCurly**, **NeedBraces** y **EmptyBlocks**.

➤ **Sentencias SWITCH.**

Las sentencias del tipo SWITCH debe tener la siguiente estructura.



```
switch (condition) {  
    case ABC:  
        statements;  
        /* falls through */  
    case DEF:  
        statements;  
        break;  
    case XYZ:  
        statements;  
        break;  
    default:  
        statements;  
        break;  
}
```

Toda sentencia switch debe incluir un valor en la cláusula **“default”**.

Esta norma será revisada por Sonar a través de las reglas **LeftCurly**, **RightCurly**, **NeedBraces** y **DefaultComesLast**.

➤ **Sentencias TRY-CATCH.**

Las sentencias del tipo TRY-CATCH debe tener la siguiente estructura:



```
try {  
    statements;  
} catch (ExceptionClass e) {  
    statements;  
}
```

Una sentencia try-catch también podría ir seguida de un bloque **finally**, que se ejecuta sin importar si se ha completado con éxito o no el bloque **try**.



```
try {  
    statements;  
} catch (ExceptionClass e) {  
    statements;  
} finally {  
    statements;  
}
```

Esta norma será revisada por Sonar a través de las reglas **LeftCurly** y **RightCurly**.

➤ Deben evitarse comparaciones propensas a generar excepciones:



```
Object a = null;  
//...  
if ( a.equals("S") ) {  
    //...  
}
```

El código escrito en el ejemplo anterior arrojaría una excepción del tipo **“NullPointerException”** si el objeto **“a”** fuese **“null”**. Por tanto, es más seguro hacer:



```
Object a = null;
//...
if ( "S".equals(a) ) {
    //...
}
```

En el siguiente ejemplo se comparan dos objetos que podrían ser “null”. Si el objeto “a” fuese “null” lanzaría, como en el caso anterior, la excepción correspondiente. El problema seguiría existiendo si se invertiera el orden de los objetos “a” y “b”.



```
Object a = null;
Object b = null;
//...
if ( a.equals(b) ) {
    //...
}
```

El código siguiente muestra la forma de evitar la excepción (la expresión es “true” únicamente si ambos objetos son null o el método “equals” devuelve “true”):



```
Object a = null;
Object b = null;
//...
if ( (a != null) ? a.equals(b) : b == null ) {
    //...
}
```

Esta norma será revisada por Sonar a través de las reglas Error: no se encontró el origen de la referencia y Error: no se encontró el origen de la referencia **y Error: no se encontró el origen de la referencia.**

- Las constantes numéricas (literales) no deben ser codificadas directamente, excepto para los valores -1, 0 y 1, que pueden aparecer en bucles for como valores contadores.  
Esta norma será revisada por Sonar a través de la regla Error: no se encontró el origen de la referencia.
- Deben evitarse asignaciones múltiples del mismo valor a varias variables en una única sentencia. Dificulta la lectura de código. Por ejemplo:



```
fooBar.fChar = barFoo.lchar = 'c'; // EVITAR!
```

Esta norma será revisada por Sonar a través de la regla Error: no se encontró el origen de la referencia.

- No se deben utilizar las sentencias embebidas en otras sentencias tratando de mejorar el rendimiento en tiempo de ejecución, como muestra el siguiente ejemplo:





```
d = (a = b + c) + r; // EVITAR!
```

Debe ser escrito como:



```
a = b + c;  
d = a + r;
```

Esta norma será revisada por Sonar a través de la regla Error: no se encontró el origen de la referencia.

- Se debe utilizar los paréntesis para asegurar problemas de precedencia de operadores, además de facilitar la comprensión del código.



```
if (a == b && c == d) // EVITAR!
```



```
if ((a == b) && (c == d)) // CORRECTO
```

- No se debe disparar de forma explícita el Recolector de Basura Java (**Garbage Collector**), a través de las llamadas **java.lang.System.gc()** o **java.lang.Runtime.gc()**, ya que la recolección de basura es un proceso muy costoso que bloquea todos los **threads** de la MVJ (Máquina Virtual Java) en ejecución.

Es la propia MVJ la que debe determinar el mejor momento para lanzar este proceso.

Para casos en que la aplicación se bloquee a menudo, debido a la carga de objetos en memoria, es recomendable buscar un punto de la misma, desde donde pueda hacerse la llamada al **Garbage Collector**, sin penalizar al usuario que la está ejecutando.

- La clase **“String”** proporciona métodos eficientes aunque la creación de sus instancias supone un proceso costoso desde el punto de vista del rendimiento. Por otra parte, el compilador de Java recurre a **“StringBuffers”** para realizar las operaciones con **“Strings”**, tal y como muestra el siguiente ejemplo:



```
x = "a" + 4 + "c";  
//Se compila al equivalente:  
x = new  
StringBuffer().append("a").append(4).append("c").  
toString();
```

Por tanto, debe tenerse en cuenta que:

Es muy recomendable evitar la creación innecesaria de objetos “String”. En el siguiente ejemplo puede verse como la sentencia interna del bucle crea un nuevo “String” en cada iteración del ciclo:



```
String s = new String();  
while ( notDone() ) {  
    s = s + msg.next();  
}
```

Por ello, es más eficiente, recurrir a implementaciones basadas en “StringBuffer”, tal y como muestra el siguiente ejemplo:



```
StringBuffer sb = new StringBuffer(msg.size());  
//inicializa el StringBuffer al tamaño correcto,  
si se conoce  
while ( notDone() ) {  
    sb.append(msg.next());  
    //El crecimiento del StringBuffer implica la  
    creación de objetos  
}
```

Por otra parte, debe considerarse que los “StringBuffer” nunca reducen su tamaño; únicamente lo aumentan por lo que no suele ser recomendable su reutilización.

Estas normas serán revisadas por Sonar a través de las reglas Error: no se encontró el origen de la referencia y Error: no se encontró el origen de la referencia.

- **Sincronización de sentencias en el interior de bucles:** No se deben establecer bloques sincronizados dentro de un bucle, ya que, por cada iteración se debe liberar el bloqueo y volver a retomarlo.



```
for (...){  
    synchronized(lock){  
        // Código sincronizado  
    }  
}
```

Es más eficiente el código que muestra el siguiente ejemplo:



```
synchronized(lock){  
    for (...){  
        // Código sincronizado  
    }  
}
```

- Se debe evitar la utilización de la interfaz SingleThreadModel desde Servlets, salvo que sea estrictamente necesario el desarrollo de servlets sincronizados, a través de la interfaz SingleThreadModel, ya que dicha interfaz, permite la ejecución del método service() del servlet por un único thread en cada instante de tiempo. Un servlet hace uso de la interfaz



SingleThreadModel al declararse de la siguiente forma:



```
Public class MiServlet extends HttpServlet  
implements SingleThreadModel {  
    ...  
}
```

Estas normas serán revisadas por Sonar a través de la regla Error: no se encontró el origen de la referencia.





### ➤ Acceso a ficheros:

Es conveniente reducir los accesos a disco, utilizando las clases `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader` y `BufferedWriter` como “recubrimiento” al resto de clases de acceso a fichero del paquete `java.io`. (subclases de `java.io.InputStream`, subclases de `java.io.OutputStream`, subclases de `java.io.Reader` y subclases de `java.io.Writer`).

Ejemplo sin utilizar buffer; la siguiente sentencia obliga a un acceso a disco por cada byte a leer:



```
// cuenta el número de saltos de línea en un
fichero.
FileInputStream fis = new
FileInputStream("miFichero.in");
int cnt = 0;
int b;
while ((b = fis.read()) != -1) {
    if (b == '\n') {
        cnt++;
    }
}
```

Ejemplo utilizando buffer;



```
// cuenta el número de saltos de línea en un
fichero.
FileInputStream fis = new
FileInputStream("miFichero.in");
BufferedInputStream bis = new
BufferedInputStream(fis);
int cnt = 0;
int b;
while ((b = bis.read()) != -1) {
    if (b == '\n') {
        cnt++;
    }
}
```

A continuación se detallan las clases del paquete `java.io` de uso más frecuente, y su utilización correcta con buffers:

Clases de `java.io` que deben estar encapsuladas en `BufferedReader/BufferedInputStream`:

- `FileReader`
- `FileInputStream`
- `InputStreamReader`

Clases de `java.io` que deben estar encapsuladas en `BufferedWriter/BufferedOutputStream`:

- `FileWriter`
- `FileOutputStream`
- `OutputStreamWriter`

Clases de `java.io` que deben recibir en su constructor un `Bufferedxxx` como parámetro:

- DataInputStream
- DataOutputStream
- ObjectInputStream
- ObjectOutputStream
- PrintStream
- PrintWriter

La manera más adecuada de leer líneas completas de un fichero de texto es a través del método `BufferedReader.readLine()`. No se debe utilizar `DataInputStream.readLine()`, ya que dicho método está obsoleto y no convierte adecuadamente bytes a caracteres. Para leer líneas de texto, cualquier código de la forma:



```
DataInputStream d = new DataInputStream(in);
```

debería reemplazarse por:



```
BufferedReader d = new BufferedReader(new  
InputStreamReader(in));
```


Debe limitarse el uso de la clase `File` del paquete `java.io`, ya que debe solicitar la información referente a las propiedades del fichero (como longitud, fecha de última modificación, si se trata de un directorio o un fichero) al sistema operativo subyacente, lo cual resulta muy costoso en términos de rendimiento.

- Los bloques `try/catch` son poco costosos en términos de rendimiento, cuando la excepción no se produce. El esfuerzo de proceso para establecer un bloque `try/catch` es muy pequeño; sin embargo, el coste del tratamiento de la excepción cuando ésta se produce no es trivial, pudiéndose llegar a penalizar el rendimiento si la excepción salta con relativa frecuencia (en cuyo caso no sería una excepción y no debería ser tratada como tal). Por ello, se recomienda la utilización de excepciones para tratar condiciones de error en un programa Java cuando se espera que éstas se produzcan de forma esporádica.

La principal alternativa al uso de excepciones es la utilización de retorno de códigos de estado; sin embargo este último no ofrece mayor rendimiento, ya que el uso de excepciones, reduce el número de parámetros en las llamadas a método y por consiguiente que éstos se ejecuten más rápidamente que utilizando códigos de estado. Muchas veces, resulta más eficaz dejar que salte la excepción y capturarla, que ir controlando que dicha excepción no se produzca.

Esta norma será revisada por Sonar a través de la regla **ExceptionAsFlowControl**.



	<p style="text-align: center;"><b>Metodología Desarrollo y Calidad</b></p> <p style="text-align: center;">Normativa Técnica Java</p>
--	--

#### 2.3.4. Pautas de Codificación validadas por PMD, CheckStyle y FindBugs

En este apartado se definen las reglas de revisión estática de código para sistemas de información J2EE. Con esta revisión se pretende comprobar la conformidad de un sistema de información sobre el código en base a la normativa de codificación descrita anteriormente y a estándares de desarrollo y codificación publicados en MADEJA.

La verificación estática permite detectar defectos y vulnerabilidades existentes en las aplicaciones de manera temprana.


Desde el punto de vista de las verificaciones analizadas, éstas son priorizadas en función de su nivel de severidad, es decir, según el impacto que pueda implicar en el comportamiento del sistema de información su incorrecta aplicación:

- **Bloqueante**, puede producir un bloqueo en la ejecución del sistema
- **Crítica**, una parte significativa del sistema es ineficaz y puede afectar seriamente a la ejecución de la aplicación
- **Grave**, puede afectar a la pérdida de funcionalidad del sistema, tal como se especifica en los requisitos
- **Leve**, conlleva a un problema tolerable durante el uso del sistema
- **informativa**, pequeña incidencia sin implicación en la ejecución del sistema

Las reglas de revisión son categorizadas, según los parámetros de calidad sobre los que centran, de la siguiente manera:

- **Reglas de Mantenibilidad:** garantizar la facilidad con la que un sistema de información puede ser modificado para corregir fallos, mejorar su funcionamiento o adaptarse a cambios en el entorno
- **Reglas de Usabilidad:** grado de eficacia y eficiencia con la que un sistema satisface requisitos específicos de usuarios en un contexto concreto, es decir, tratan aspectos referidos a la capacidad de un software de ser comprendido, aprendido, usado y ser atractivo para el usuario, en condiciones específicas de uso
- **Reglas de Fiabilidad:** verificaciones enfocadas a medir la probabilidad que un programa funcione correctamente en un entorno concreto y durante un periodo de tiempo concreto.
- **Reglas de Portabilidad:** orientadas a garantizar la portabilidad del sistema entre entornos hardware/software
- **Reglas de Eficiencia:** búsqueda de redundancia de operaciones que pueden implicar una aplicación menos eficiente.




 <b>JUNTA DE ANDALUCÍA</b> <small>CONSEJERÍA DE HACIENDA Y ADMINISTRACIÓN PÚBLICA</small>	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

Por defecto se validarán todas las reglas propuestas por MADEJA, determinando, para cada proyecto y según la prioridad / severidad de la misma, si se cambia su estado.

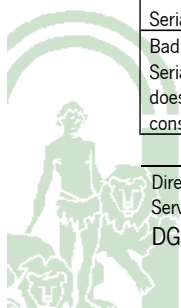
Título	Descripción	Plugin	Prioridad	Estado
Correctness - equals method always returns false	Indica que la clase está definiendo un método de comparación "equals" que siempre devuelve FALSE	findbugs	BLOQUEANTE	Activo
Correctness - equals method always returns true	Indica que la clase está definiendo un método de comparación "equals" que siempre devuelve TRUE	findbugs	BLOQUEANTE	Activo
Equals Hash Code	Revisa que las clases que sobrescriban el método equals() también sobrescriban el método hashCode()	checkstyle	BLOQUEANTE	Activo
Security - Hardcoded constant database password	Revisa que no exista código que cree una conexión con base de datos utilizando código "a fuego" con la clave de acceso en una constante. Cualquiera que acceda al código fuente o al código compilado puede ver fácilmente la clave de acceso	findbugs	BLOQUEANTE	Activo
Avoid Catching Throwable	No se deben capturar los errores a través de Throwable ya que su alcance es muy amplio. Incluye temas de tiempo de ejecución tales como OutOfMemoryError que deben ser expuestos y tratados de forma separada	pmd	CRITICA	Activo
Bad practice - Class defines equals() and uses Object.hashCode()	La clase define el método equals y utiliza Object.hashCode()	findbugs	CRITICA	Activo
Bad practice - Class defines hashCode() and uses Object.equals()	La clase define hashCode() y utiliza Object.equals()	findbugs	CRITICA	Activo
Bad practice - Class defines hashCode() but not equals()	Revisa que en caso que sea sobrescrito el método hashCode(), necesariamente es también sobrescrito el método equals(), y viceversa	findbugs	CRITICA	Activo
Bad practice - Class inherits equals() and uses Object.hashCode()	La clase hereda equals() y utiliza Object.hashCode()	findbugs	CRITICA	Activo
Bad practice - Clone method may return null	El método Clone devuelve null en algunas circunstancias	findbugs	CRITICA	Activo
Bad practice - equals() method does not check for null argument	El método equals() no comprueba la recepción de argumentos nulos	findbugs	CRITICA	Activo
Bad practice - Method may fail to close database resource	Revisa que los recursos creados en un método se cierren en todas las rutas de escape del mismo	findbugs	CRITICA	Activo
Bad practice - Method may fail to close database resource on exception	Revisa que los recursos creados en un método se cierren en todas las excepciones que se pueden dar en dicho método	findbugs	CRITICA	Activo
Bad practice - Random object created and used only once	Se ha creado un objeto Random pero se ha utilizado sólo una vez. Es preferible crear el objeto y guardarlo para que cada vez que se necesite se genere un nuevo número, ya que, en otro caso, la calidad del número aleatorio es mediocre e ineficiente	findbugs	CRITICA	Activo
Bad practice - serialVersionUID isn't final	La clase define un campo serialVersionUID que no es definitivo. El campo debe ser definitivo si está destinado a especificar la versión del UID	findbugs	CRITICA	Activo
Bad practice - Static initializer creates instance before all static final fields assigned	El inicializador estático de la clase crea una instancia de la clase antes de que todos los campos estáticos finales sean asignados	findbugs	CRITICA	Activo
Correctness - An apparent infinite loop	Existencia de bucle infinito que no parece que tenga condición de finalización	findbugs	CRITICA	Activo
Correctness - An apparent infinite recursive loop	Existencia de método que se llama a sí mismo incondicionalmente	findbugs	CRITICA	Activo
Correctness - Call to equals() comparing different types	Se están realizando llamadas al método equals() comparando tipos de objeto diferentes	findbugs	CRITICA	Activo
Correctness - Call to equals() with null argument	Se realiza llamada al método equals() con un argumento nulo	findbugs	CRITICA	Activo
Correctness - Double assignment of field	Existencia de método en el que se está realizando una doble asignación de campo. Por ejemplo: <pre>int x,y; public void foo() {     x = x = 17; }</pre>	findbugs	CRITICA	Activo
Correctness - Exception created and dropped rather than thrown	Se está creando un objeto de tipo Excepción pero no se hace nada al tratarla	findbugs	CRITICA	Activo

Título	Descripción	Plugin	Prioridad	Estado
Correctness - hasNext method invokes next	El método hasNext() está invocando al método next(). Esta acción normalmente es errónea ya que el método hasNext() no efectúa ningún cambio de estado en el iterador y el método next() sí.	findbugs	CRITICA	Activo
Correctness - Method assigns boolean literal in boolean expression	Revisa que en una sentencia que conlleve una condición (if, while...) no aparezcan asignaciones de variables booleanas (evita errores de programación del tipo if(a=b)... que debería ser if(a==b))...	findbugs	CRITICA	Activo
Correctness - Null pointer dereference	Se hace referencia a un puntero Nulo, lo cual elevará la excepción NullPointerException	findbugs	CRITICA	Activo
Correctness - Self assignment of field	Existencia de método en el que se está asignando una variable a sí misma. Por ejemplo: x= x;	findbugs	CRITICA	Activo
Correctness - Signature declares use of unhashable class in hashed construct	Existencia de método, variable o clase que declara un método equals() pero el método hashCode() es unhasable, incumpliendo la condición que objetos iguales tengan Códigos Hash iguales	findbugs	CRITICA	Activo
Correctness - Use of class without a hashCode() method in a hashed data structure	Una clase define un método equals() pero no un hashCode()	findbugs	CRITICA	Activo
Dodgy - Complicated	¿Está seguro que el bucle <b>for</b> está incrementando la variable correcta? Parece que otra variable es inicializada y chequeada por el bucle <b>for</b> .	findbugs	CRITICA	Activo
Dodgy - Double assignment of local variable	Se está realizando una doble asignación de variable en el código.	findbugs	CRITICA	Activo
Dodgy - Initialization circularity	Existencia de inicialización circular	findbugs	CRITICA	Activo
Dodgy - Method uses the same code for two branches	Existencia de método que realice las mismas acciones en dos ramas diferentes.	findbugs	CRITICA	Activo
Dodgy - Self assignment of local variable	Existencia de método que asigna una variable local a sí misma. Por ejemplo: <pre>public void foo() {     int x = 3;     x = x; }</pre>	findbugs	CRITICA	
Dodgy - Vacuous comparison of integer value	Se está realizando una comparación de Integers que siempre devuelve el mismo valor	findbugs	CRITICA	Activo
Empty Catch Block	Se debe evitar usar bloques catch vacíos. Comprobar que cuando un método de una API lanza una excepción verificable, está tratando de indicar al código cliente que debe realizar alguna acción que revierta esta situación. Si la excepción verificable no tiene sentido en el contexto donde se ha capturado, se debe convertir en una excepción no verificable y lanzarla nuevamente, pero nunca debe ignorarse con un bloque catch vacío y luego continuar con el flujo normal de ejecución	pmd	CRITICA	Activo
Empty Finally Block	Se debe evitar usar bloques finally vacíos	pmd	CRITICA	Activo
Empty If Stmt	Existencia de instancias donde una condición es chequeada pero no se realiza nada en ella	pmd	CRITICA	Activo
Empty While Stmt	Revisa que no existan estamentos <i>while</i> vacíos	pmd	CRITICA	Activo
Equals Null	Revisa que no se utilice la función equals() para comprobar si un objeto es nulo	pmd	CRITICA	Activo
Multithreaded correctness - Condition.await() not in loop	Existencia de método que contiene una llamada a java.util.concurrent.await() (o variantes) el cual no está en un bucle.	findbugs	CRITICA	Activo
Multithreaded correctness - Constructor invokes Thread.start()	El constructor de la clase está invocando al método Thread.start()	findbugs	CRITICA	Activo
Multithreaded correctness - Incorrect lazy initialization and update of static field	Después de fijar un valor a un atributo, el objeto almacenado en esa localización es actualizado o consultado. La asignación de valor a un atributo es visible al resto de hilos de forma inmediata. Esto puede provocar una inconsistencia en los valores de los atributos del objeto si otro hilo accede antes de iniciar todos los atributos, a no ser que se establezca otro mecanismo que evite el acceso al objeto de otros hilos hasta que se finalice completamente la inicialización del mismo.	findbugs	CRITICA	Activo
Multithreaded correctness - Incorrect lazy initialization of static field	Revisa si se están realizando inicializaciones de atributos de manera no sincronizada, de tal manera que los hilos no tienen garantizada la inicialización completa del objeto cuando el método es llamado por varios hilos.	findbugs	CRITICA	Activo
Multithreaded correctness - Method calls Thread.sleep() with a lock held	Un método está realizando la llamada a Thread.sleep() teniendo objetos bloqueados, de tal manera que puede provocar situación de DeadLock	findbugs	CRITICA	Activo
Multithreaded correctness - Mismatched notify()	Revisa si existen métodos que llamen a Object.notify() o Object.notifyAll() sin bloquear el objeto. Esto provoca que se eleve la excepción <i>IllegalMonitorStateException</i> .	findbugs	CRITICA	Activo
Multithreaded correctness - Mismatched wait()	Revisa si existen métodos que llamen a Object.wait() sin bloquear el objeto. Esto provoca que se eleve la excepción <i>IllegalMonitorStateException</i> .	findbugs	CRITICA	Activo

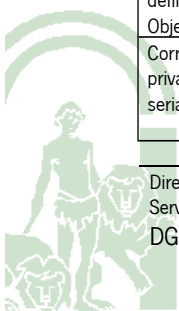


	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java


Título	Descripción	Plugin	Prioridad	Estado
Security - A prepared statement is generated from a nonconstant String	Revisa si el código crea una sentencia SQL preparada desde una cadena no constante. De esta forma, puede utilizarse inyección de SQL que haga que la sentencia haga algo inesperado o no deseado.	findbugs	CRITICA	Activo
Security - Array is stored directly	Revisa que constructores y métodos que reciban arrays como parámetros no los usen directamente, se debe usar una copia de ellos	pmd	CRITICA	Activo
Security - Empty database password	Revisa de la existencia de código que crea la conexión a una base de datos sin indicar la password o indicándola vacía	findbugs	CRITICA	Activo
Security - JSP reflected cross site scripting vulnerability	Comprueba la existencia de código que escriba un parámetro HTTP directamente en la salida JSP, esto permite traspasar la medida de seguridad XSS (Cross-Site Scripting) existente en la mayoría de los navegadores	findbugs	CRITICA	Activo
Security - Nonconstant string passed to execute method on an SQL statement	Comprueba la existencia de métodos que invocan la ejecución de sentencias SQL con cadenas que son generadas dinámicamente. Es preferible utilizar sentencias preparadas al ser más eficiente y menos vulnerable a ataques de inyección de SQL	findbugs	CRITICA	Activo
Security - Servlet reflected cross site scripting vulnerability	Comprueba la existencia de código que escribe un parámetro HTTP directamente en la salida del servlet, vulnerando la seguridad XSS del navegador.	findbugs	CRITICA	Activo
Unconditional If Statement	Revisa que no se usen sentencias condicionales cuya condición siempre sea cierta o falsa.	pmd	CRITICA	Activo
Useless Operation On Immutable	Comprueba la existencia de una operación sobre un objeto inmutable (String, BigDecimal o BigInteger) no cambiará el objeto, sino que el resultado de la operación será un nuevo objeto.	pmd	CRITICA	Activo
Anon Inner Length	Comprobar la existencia de clases largas anónimas interiores. En caso de ser mayor, debe ser nominada (named inner class). max=20 el valor máximo de líneas que se considera que debe tener una clase interna anónima	checkstyle	GRAVE	Activo
At Least One Constructor	Revisa que existe un constructor, al menos, definido en una clase. Con esto facilitamos la carga dinámica de clases de tipo desconocido en tiempo de compilación mejorando el rendimiento de nuestra aplicaciones	pmd	GRAVE	Activo
Avoid Assert As Identifier	Evitar usar 'assert' como identificador	pmd	GRAVE	Activo
Avoid Calling Finalize	Evitar llamar al método finalize() de un objeto. Éste método será utilizado por Garbage Collector cuando determina que no existen más referencias de un objeto	pmd	GRAVE	Activo
Avoid Decimal Literals In Big Decimal Constructor	Evitar utilizar literales decimales en la construcción de un objeto tipo BigDecimal; es preferible usar literales tipo String. Por ejemplo: // construcción errónea BigDecimal bd=new BigDecimal(1.123); // construcción válida BigDecimal bd=new BigDecimal("1.123"); // construcción válida BigDecimal bd=new BigDecimal(12);	pmd	GRAVE	
Avoid Duplicate Literals	Revisa la existencia de código con variables de tipo cadena con los mismos literales, pudiendo mejorar el rendimiento declarando el campo como constante	pmd	GRAVE	Activo
Avoid Enum As Identifier	Evitar usar 'enum' como identificador	pmd	GRAVE	Activo
Avoid Print Stack Trace	Comprueba el uso del método printStackTrace() en vez de utilizar la llamada a un logger.	pmd	GRAVE	Activo
Avoid Reassigning Parameters	Revisa que no se reasignen los parámetros de entrada de un método	pmd	GRAVE	Activo
Avoid Rethrowing Exception	Revisa que no se registra más de una vez una excepción en un método. Sólo aumenta la complejidad de ejecución y hace menos comprensible el código	pmd	GRAVE	Activo
Avoid Throwing Null Pointer Exception	Evitar lanzar una excepción de tipo 'NullPointerException' (esta excepción es lanzada habitualmente por la máquina virtual)	pmd	GRAVE	Activo
Bad practice - Class defines equals() but not hashCode()	Existencia de clases que sobrescriben el método equals() del objeto pero no el hashCode(), de tal manera que la clase viola el premisa que mismos objetos deben tener el mismo código Hash	findbugs	GRAVE	Activo
Bad practice - Class implements Cloneable but does not define or use clone method	La clase implementa Cloneable pero no define o utiliza el método Clone()	findbugs	GRAVE	Activo
Bad practice - Class is Serializable	Comprueba la existencia de una clase que implementa la interface Serializable pero no define el campo serialVersionUID	findbugs	GRAVE	
Bad practice - Class is Serializable but its superclass doesn't define a void constructor	Existe una clase que implementa la interface Serializable y su superclase no lo hace	findbugs	GRAVE	Activo



Título	Descripción	Plugin	Prioridad	Estado
Bad practice - Comparator doesn't implement Serializable	Existe una clase que implementa la interface Comparator pero no implementa Serializable. Según el caso será necesario la implementación de Serializable o no. Si un comparador es utilizado para construir una colección ordenada como TreeMap, entonces TreeMap deberá ser Serializable sólo si el comparador lo es.	findbugs	GRAVE	Activo
Bad practice - Comparison of String objects using == or !=	Detecta el uso de los operadores == o != para comparar objetos de tipo String en vez de utilizar el método equals() del objeto.	findbugs	GRAVE	
Bad practice - Comparison of String parameter using == or !=	Detecta el uso de los operadores == o != para comparar parámetros de tipo String en vez de utilizar el método equals() del objeto.	findbugs	GRAVE	
Bad practice - Empty finalizer should be deleted	Detecta el uso del método finalize(), el cual está en desuso.	findbugs	GRAVE	
Bad practice - Finalizer does not call superclass finalizer	Comprueba la llamada al método finalize() de una clase que no llama al método finalize() de la superclase.	findbugs	GRAVE	
Bad practice - Finalizer only nulls fields	Revisa la existencia de métodos finalize() que sólo asignan valor null a los atributos.	findbugs	GRAVE	
Bad practice - Method ignores exceptional return value	Revisa el uso de funciones sin comprobar el valor devuelto tras la ejecución de la misma. En muchas ocasiones, las funciones devuelven valores que informan sobre el resultado de su ejecución y debe verificarse.	findbugs	GRAVE	Activo
Bad practice - Method invoked that should be only be invoked inside a doPrivileged block	Comprueba si existen métodos que para ser invocados necesitan de la comprobación de permisos de seguridad de tal manera que el método debe ser invocado en un bloque doPrivileged	findbugs	GRAVE	Activo
Bad practice - Method invokes dangerous method runFinalizersOnExit	Comprueba si se está realizando llamada al método System.runFinalizersOnExit o Runtime.runFinalizersOnExit. Debe eliminarse la invocación a estos métodos	findbugs	GRAVE	Activo
Bad practice - Method might drop exception	Comprueba si existen métodos en los que pueden eliminarse excepciones ya que no son tratadas o debe crearse el tratamiento de dicha excepción	findbugs	GRAVE	
Bad practice - Method might ignore exception	Existencia de excepciones en los métodos que pueden ser ignoradas	findbugs	GRAVE	
Bad practice - Needless instantiation of class that only supplies static methods	Comprueba la existencia en el código de instancias de clases que sólo tienen métodos estáticos. Estas clases no necesitan ser instanciadas, tan sólo se accede a sus métodos	findbugs	GRAVE	Activo
Bad practice - Superclass uses subclass during initialization	Comprueba el uso de una subclase en la inicialización de una clase cuando la subclase no ha sido aún inicializada. Por ejemplo: <pre>public class CircularClassInitialization {     static class InnerClassSingleton extends CircularClassInitialization {         static InnerClassSingleton singleton = new InnerClassSingleton();     }      static CircularClassInitialization foo = InnerClassSingleton.singleton; }</pre>	findbugs	GRAVE	Activo
Boolean Expression Complexity	Restringir el uso de operadores booleanos anidados (&&,  , &,   y ^) a una profundidad indicada (por defecto 3)	checkstyle	GRAVE	Activo
Clone method must implement Cloneable	El método clone() solo debe ser definido en clases que sean implementaciones de la interfaz Cloneable, excepto si se trata de un método final que sólo lanza la excepción CloneNotSupportedException	pmd	GRAVE	Activo
Clone Throws Clone Not Supported Exception	El método clone() debe lanzar siempre la excepción CloneNotSupportedException	pmd	GRAVE	Activo
Close Resource	Revisa que los recursos (conexiones, resultsets, statements...) son siempre cerrados después de su uso	pmd	GRAVE	Activo
Compare Objects With Equals	Evitar comparar objetos mediante el operador ==. Utilizar en su lugar el método equals().	pmd	GRAVE	Activo
Constructor Calls Overridable Method	Comprueba la existencia de constructores de clase que invocan a métodos sobrescritos	pmd	GRAVE	
Correctness - equals() method defined that doesn't override equals(Object)	Comprueba la existencia de un método equals() que no sobrescribe el método general equals(Object) de java.lang.Object class. De hecho, hereda el método equals(Object) de la superclase. La clase debería definir boolean equals(Object)	findbugs	GRAVE	Activo
Correctness - equals() method defined that doesn't override Object.equals(Object)	La clase define un método equals() que no sobrescribe el método normal equals(Object) definido en la clase base java.lang.Object. La clase, probablemente, debería definir un método boolean equals(Object)	findbugs	GRAVE	Activo
Correctness - Method must be private in order for serialization to work	Comprueba si existen clases que implementan la interface Serializable y definen un método a medida para serializa/deserializar. Pero el método no es declarado como privado, lo cual puede hacer que sea ignorado por la API de serialización / deserialización	findbugs	GRAVE	Activo






 <b>JUNTA DE ANDALUCÍA</b> <small>CONSEJERÍA DE HACIENDA Y ADMINISTRACIÓN PÚBLICA</small>	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

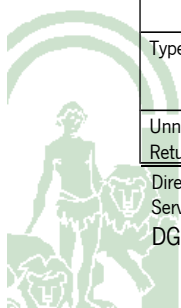
Título	Descripción	Plugin	Prioridad	Estado
Cyclomatic Complexity	Revisa que métodos, constructores, inicializadores estáticos... no posean una complejidad ciclomática superior al valor indicado en el parámetro max. Parámetros: max=20 especifica el valor máximo establecido para la complejidad ciclomática de bloques	checkstyle	GRAVE	Activo
Default Comes Last	Revisa que en todo bloque SWITCH aparezca la cláusula default	checkstyle	GRAVE	Activo
Dodgy - Class implements same interface as superclass	Revisa si existen clases que implementan una interface que ya ha sido implementada por su superclase	findbugs	GRAVE	Activo
Dodgy - Transient field of class that isn't Serializable.	Comprueba si existen campos marcados como transitorios en clases no serializables por lo que no tienen ningún efecto el marcado del campo	findbugs	GRAVE	Activo
Double Checked Locking	com.puppycrawl.tools.checkstyle.checks.coding.DoubleCheckedLockingCheck	checkstyle	GRAVE	
Empty Block	Revisa que no existan bloques vacíos (CATCH, DO, ELSE, FINALLY, FOR, TRY, WHILE, IF) Parámetros: option=stmt establece que en los bloques debe existir, al menos, una sentencia (no basta con la existencia de un comentario).	checkstyle	GRAVE	Activo
Empty Finalizer	Comprueba si existen métodos finalize() vacíos	pmd	GRAVE	
Empty Static Initializer	Revisa que no existan inicializadores estáticos vacíos	pmd	GRAVE	Activo
Empty Switch Statements	Revisa que no existan sentencias Switch vacías	pmd	GRAVE	Activo
Empty Try Block	Revisa que no existan bloques try vacíos	pmd	GRAVE	Activo
Exception As Flow Control	Revisa que no se utiliza excepciones para controlar el flujo de una aplicación	pmd	GRAVE	Activo
Excessive Class Length	Se debe evitar la construcción de clases con un número alto de líneas de código. En caso de superar el límite convendría refactorizar la clase. minimum=1000 establece el número máximo de líneas que puede contener una clase (excluidas las líneas de comentarios)	pmd	GRAVE	
Excessive Method Length	Se debe evitar la construcción de métodos con un número alto de líneas de código minimum=100 establece el número máximo de líneas que puede contener un método (excluidas las líneas de comentarios)	pmd	GRAVE	
Excessive Parameter List	Se debe evitar la construcción de métodos con un número alto de argumentos. minimum=10 establece el número máximo de parámetros de un método	pmd	GRAVE	
File Length	Se debe evitar la construcción de clases con un número alto de líneas de código. En caso de superar el límite convendría refactorizar la clase minimum=100 establece el número máximo de líneas que puede contener una clase (excluidas las líneas de comentarios)	checkstyle	GRAVE	Activo
Final Class	Comprueba que una clase que sólo tiene constructores privados se declara como final	checkstyle	GRAVE	Activo
Finalize Only Calls Super Finalize	Si el método finalize() es implementado, éste sólo debe invocar al método finalize() de la superclase.	pmd	GRAVE	
Finalize Overloaded	El método Finalize() no debe tener parámetros	pmd	GRAVE	
Finalize Should Be Protected	Cuando se sobrescribe el método finalize(), el nuevo método debe declararse como protected.	pmd	GRAVE	
For Loops Must Use Braces	Comprueba la Existencia de bloques For que no están entre llaves { }	pmd	GRAVE	
If Else Stmts Must Use Braces	Los estamentos If Else deben utilizar '{' y '}'	pmd	GRAVE	
If Stmts Must Use Braces	Los estamentos If deben utilizar '{' y '}'	pmd	GRAVE	
Illegal Throws	Comprueba si existen invocaciones a java.lang.Error o java.lang.RuntimeException	checkstyle	GRAVE	Activo
Immutable Field	Identifica la existencia de campos privados cuyos valores nunca cambian salvo cuando son inicializados	pmd	GRAVE	Activo
Inefficient String Buffering	Revisa que no se utilice la concatenación no literal durante la construcción de un objeto StringBuffer o durante la ejecución del método String.append(). Por ejemplo:  <pre>public class Foo {     void bar() {         // Evitar concatenaciones del tipo         StringBuffer sb=new StringBuffer("tmp="+System.getProperty("java.io"));          // Usar mejor este tipo concatenación         StringBuffer sb = new StringBuffer("tmp = ");         sb.append(System.getProperty("java.io"));     } }</pre>	pmd	GRAVE	Activo




Título	Descripción	Plugin	Prioridad	Estado
Instantiation To Get Class	Comprueba si se instancia un objeto a través de la llamada al método getClass() sobre él. Es preferible utilizar el componente público .class para ello	pmd	GRAVE	Activo
Insufficient String Buffer Declaration	El tamaño en caracteres del argumento para StringBuffer.append() debe ser acorde con el tamaño de construcción del StringBuffer	pmd	GRAVE	Activo
Javadoc Method	Revisa que los métodos y constructores de una clase estén documentados. Además, revisa que se utilicen las etiquetas de Javadoc en la declaración de constructores y métodos de una clase (@param, @return, @throws). No será necesario si posee anotaciones. - scope=private establece el ámbito donde la regla será revisada - allowMissingParamTags=false indica si se revisa la documentación de parámetros de un método (@param) - allowMissingThrowsTags=false indica si se revisa la documentación de excepciones tratadas en un método (@throws) - allowMissingReturnTag=false indica si se revisa la documentación de valores de retorno de un método (@return)	checkstyle	GRAVE	Activo
Javadoc Type	Revisa que las clases e interfaces estén documentadas. - scope=private establece el ámbito donde la regla será revisada - authorFormat=false indica si se revisa la documentación de autor (@author) - versionFormat=false indica si se revisa la documentación de versión (@version) - allowUnknownTags=false indica si se permite el uso de etiquetas no definidas	checkstyle	GRAVE	Activo
Javadoc Variable	Revisa que los atributos de las clases estén documentados. scope=private establece el ámbito donde la regla será revisada	checkstyle	GRAVE	
Line Length	Revisa que la longitud de una línea no supere el máximo permitido max=80 indica el máximo número de caracteres para una línea	checkstyle	GRAVE	Activo
Local Final Variable Name	Revisa la política de nombrado de atributos finales. format=[a-z][a-zA-Z0-9]*\$ indica la política de nombrado de atributos finales	checkstyle	GRAVE	Activo
Local Variable Name	Revisa la política de nombrado de los atributos de una clase o interfaz (no finales) format=[a-z][a-zA-Z0-9]*\$ indica la política de nombrado de atributos	checkstyle	GRAVE	Activo
Malicious code vulnerability - Field isn't final but should be	Existencia de un campo estático de tipo public que no es final y podría ser modificado por código malicioso o por accidente por otro paquete. El campo debería hacerse final para evitar esta vulnerabilidad	findbugs	GRAVE	Activo
Malicious code vulnerability - Field should be both final and package protected	Revisa la existencia de campos estáticos cambiantes que pueden ser modificados por código malicioso o por otros paquetes. El campo debería hacer al paquete de tipo protected y/o hacer final al campo para evitar esta vulnerabilidad	findbugs	GRAVE	Activo
Malicious code vulnerability - Finalizer should be protected	Comprueba la existencia de métodos finalize() en alguna clase no siendo declarado de tipo protected	findbugs	GRAVE	
Member name	Chequea que el nombre de las variables se ajusta a un determinado formato	checkstyle	GRAVE	
Method Length	Se debe evitar la construcción de métodos con un número alto de líneas de código. En caso de superar el límite convendría dividir el método. minimum=100 establece el número máximo de líneas que puede contener un método (excluidas las líneas de comentarios)	checkstyle	GRAVE	Activo
Method Name	Revisa la política de nombrado de métodos. format=[a-z][a-zA-Z0-9]*\$ indica la política de nombrado de métodos	checkstyle	GRAVE	Activo
Missing Constructor	Comprueba que las clases definen un constructor en vez de utilizar el constructor por defecto	checkstyle	GRAVE	Activo
Missing Serial Version UID	Comprueba la existencia de clases serializables que no proveen el campo serialVersionUID	pmd	GRAVE	Activo
Missing Static Method In Non Instantiatable Class	Revisa si existen clases con constructores privados que no tienen ningún método o campo estático	pmd	GRAVE	Activo
Modified Control Variable	Comprueba si la variable de control de un bucle es modificada en el bloque asociado	checkstyle	GRAVE	Activo
More Than One Logger	Comprueba si existe más de un logger para una clase	pmd	GRAVE	Activo
Multithreaded correctness - Unconditional wait	Comprueba si el método contiene una llamada a java.lang. Object.wait () que no es coordinado por un flujo de control condicional. El código debería verificar que la intención de esperar no está satisfecha antes de llamar a wait()	findbugs	GRAVE	Activo
Mutable Exception	Comprueba que no existan excepciones inmutables	checkstyle	GRAVE	Activo
Null Assignment	Revisa que no se produzcan asignaciones de variables a null excepto en su declaración	pmd	GRAVE	Activo


 <b>JUNTA DE ANDALUCÍA</b> CONSEJERÍA DE HACIENDA Y ADMINISTRACIÓN PÚBLICA	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

Título	Descripción	Plugin	Prioridad	Estado
Package name	Revista la política de nombrado de paquetes. format=^[a-z]+(\.[a-zA-Z_][a-zA-Z0-9_]*)*\$ indica la política de nombrado de paquetes	checkstyle	GRAVE	Activo
Parameter Assignment	Revista que no se realicen operaciones de asignación en parámetros de llamadas a métodos	checkstyle	GRAVE	Activo
Parameter Name	Revista la política de nombrado de parámetros. format=^[a-z][a-zA-Z0-9]*\$ indica la política de nombrado de parámetros	checkstyle	GRAVE	Activo
Parameter Number	Chequea el número de parámetros de los métodos y constructores	checkstyle	GRAVE	Activo
Performance - Method invokes inefficient new String() constructor	Comprueba si se está realizando la llamada al constructor String sin argumentos.	findbugs	GRAVE	Activo
Performance - Method invokes inefficient new String(String) constructor	Utilizar el constructor java.lang.String(String) desperdicia memoria porque el objeto así construido es funcionalmente indistinguible de la cadena pasada como parámetro. En este caso es mejor utilizar la cadena utilizada como argumento directamente.	findbugs	GRAVE	Activo
Performance - Unread field	Comprueba la existencia de campos en las clases que no son accedidos. En este caso, es preferible eliminar dichos campos.	findbugs	GRAVE	Activo
Performance - Unread field: should this field be static?	Comprueba la Existencia de campos que no son accedidos. La clase contiene una instancia de campo final que es inicializado con un valor estático en tiempo de ejecución. Hay que considerar hacer que el campo sea estático.	findbugs	GRAVE	Activo
Performance - Unused field	Comprueba si existen campos que nunca son utilizados.	findbugs	GRAVE	Activo
Preserve Stack Trace	Elevar una nueva excepción desde un bloques catch sin pasarle la excepción original a la nueva excepción puede causar que la pila de traza original se pierda, complicando una depuración eficiente	pmd	GRAVE	
Replace Enumeration With Iterator	Revista que no se implementen datos de tipo Enumeration. Sustituirlos por java.util.Iterator.	pmd	GRAVE	Activo
Replace Hashtable With Map	Revista que no se implementen datos de tipo Hashtable. Sustituirlos por java.util.Map.	pmd	GRAVE	Activo
Replace Vector With List	Comprueba el uso de Vector. Hay que considerar sustituirlo con el objeto más reciente java.util.ArrayList si las operaciones con hilos seguros no son requeridas	pmd	GRAVE	Activo
Return From Finally Block	Comprueba si existe la sentencia return en un bloques finally, lo cual puede descartar excepciones	pmd	GRAVE	Activo
Security - HTTP cookie formed from untrusted input	Comprueba si existe código que construye una cookie HTTP usando un parámetro HTTP no verificado.	findbugs	GRAVE	Activo
Security - HTTP Response splitting vulnerability	Comprueba si se están escribiendo parámetros HTTP en las cabeceras	findbugs	GRAVE	Activo
Simplify Boolean Expression	Comprueba la existencia de sentencias booleanas complejas que deban ser simplificadas	checkstyle	GRAVE	Activo
Simplify Boolean Return	Comprueba la existencia de sentencias booleanas complejas que deban ser simplificadas en return	checkstyle	GRAVE	Activo
Simplify Conditional	No es necesario comprobar los nulos antes de un "instance of", ya que esta instrucción devuelve false cuando se le pasa un argumento nulo	pmd	GRAVE	
Static Variable Name	Revista la política de nombrado de atributos estáticos (no finales). format=^[a-z][a-zA-Z0-9]*\$ indica la política de nombrado de atributos estáticos	checkstyle	GRAVE	Activo
String Buffer Instantiation With Char	Comprueba si se están instancia String Buffer pasando como argumentos, en vez de un número para el tamaño, una cadena, ya que es convertida a Integer y provoca que el tamaño interno del buffer sea mayor de lo esperado	pmd	GRAVE	Activo
String Instantiation	Comprueba si se están instanciando objetos de tipo String	pmd	GRAVE	
String To String	Comprueba si se están realizando llamadas al método toString() sobre objetos de tipo String	pmd	GRAVE	Activo
System Println	Comprueba si se están realizando llamadas al método SystemPrintln que normalmente es utilizado para tareas de depuración del código	pmd	GRAVE	Activo
Too many methods	Revista que en una clase se definan más de un número determinado de métodos. No se tienen en cuenta métodos que comiencen con get o set. Con ello se evita darle mucha funcionalidad a una clase (alta cohesión) debiéndose refactorizar la clase en caso de superar dicho límite. maxMethods=15 indica el máximo número de métodos definidos en una clase	pmd	GRAVE	Activo
Type Name	Revista la política de nombrado de clases e interfaces. format=^[a-z][a-zA-Z0-9]*\$ indica la política de nombrado de clases e interfaces	checkstyle	GRAVE	Activo
Unnecessary Local Before Return	Comprueba la Existencia de variables locales que no son necesarias	pmd	GRAVE	Activo

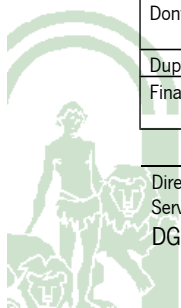


 <b>JUNTA DE ANDALUCÍA</b> <small>CONSEJERÍA DE HACIENDA Y ADMINISTRACIÓN PÚBLICA</small>	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

Título	Descripción	Plugin	Prioridad	Estado
Unused formal parameter	Revisa que no existan parámetros formales declarados en una clase que no sean utilizados posteriormente	pmd	GRAVE	Activo
Unused local variable	Revisa que no existan variables locales declaradas en una clase que no sean utilizadas posteriormente	pmd	GRAVE	Activo
Unused Private Field	Revisa que no existan campos privados declarados en una clase que no sean utilizados posteriormente	pmd	GRAVE	Activo
Use Correct Exception Logging	En el uso del log de excepciones, para asegurarse la impresión completa de stacktrace, utilizar la sentencia logging con dos argumentos, un String y un Throwable	pmd	GRAVE	Activo
Use Equals To Compare Strings	Revisa el uso del método equals() para comparar cadenas en lugar de utilizar los operadores de comparación "==" y "!=".	pmd	GRAVE	Activo
Use Index Of Char	Para acceder a un determinado carácter de un objeto de tipo String, utilizar la llamada String.indexOf(char) para obtener el índice del carácter de forma más rápida	pmd	GRAVE	Activo
Useless Overriding Method	Comprueba si existen métodos sobrescritos que llaman al mismo método definido en la superclase	pmd	GRAVE	Activo
While Loops Must Use Braces	Comprueba que los bucles while utilicen los delimitadores { y }	pmd	GRAVE	
Avoid Star Import	Revisa que no aparezcan imports que utilicen la notación '*'	checkstyle	INFORMATIVA	Activo
Design For Extension	Revisa que las clases permitan su extensión por herencia	checkstyle	INFORMATIVA	Activo
File Tab Character	Revisa que no existan caracteres de tabulación en los ficheros	checkstyle	INFORMATIVA	
Generic Whitespace	Revisa que el uso de espacios en blanco delante y detrás de los caracteres '<' y '>' sean correctos (atendiendo a los tipos genéricos o parametrizados). Por ejemplo: Uso de espacios correcto: List<Integer> x = new ArrayList<Integer>(); List<List<Integer>> y = new ArrayList<List<Integer>>(); Uso de espacios incorrecto: List < Integer > x = new ArrayList < Integer > (); List < List < Integer > > y = new ArrayList < List < Integer > > ();	checkstyle	INFORMATIVA	Activo
Indentation	Revisa la indentación de las líneas de código. - BasicOffset=4 establece el número de espacios que se utilizan para un nuevo nivel de indentación - braceAdjustment=0 establece el número de espacios que se utilizan para la siguiente línea - caseIndent=4 establece el número de espacios que se utilizan para la etiqueta case	checkstyle	INFORMATIVA	Activo
Left Curly	Revisa que se cumpla la política de apertura de llaves en clases, métodos y otros bloques. option=eol la llave de apertura '{' debe aparecer al final de la línea de la declaración del bloque. Por ejemplo:  if (condición) { ... maxLineLength=80 indica el máximo número de caracteres para la línea de definición del bloque	checkstyle	INFORMATIVA	Activo
Magic Number	Revisa que no existan "números mágicos" en el código, entendiendo por número mágico todo literal numérico no definido como constante ignoreNumbers = -1, 0, 1, 2 indica los números que no son considerados como "números mágicos"	checkstyle	INFORMATIVA	Activo
Modifier Order	Comprueba que el orden de los modificadores esté conforme a las sugerencias de las especificaciones del lenguaje Java	checkstyle	INFORMATIVA	
Need Braces	Revisa que todos los bloques de instrucciones aparezcan entre llaves, incluidas los bloques simples (con una sola sentencia). Por ejemplo:  if(condición) { instrucción; }	checkstyle	INFORMATIVA	Activo
Performance - Method invokes toString() method on a String	Comprueba que no se esté realizando la llamada al método String en un objeto de tipo String ya que es redundante	findbugs	INFORMATIVA	Activo
Redundant import	Comprueba si existen sentencias import redundantes en el código	checkstyle	INFORMATIVA	Activo
Redundant Modifier	Comprueba si existen modificadores redundantes en la definición de interfaces y anotaciones.	checkstyle	INFORMATIVA	Activo
Redundant Throws	Comprueba si existen excepciones declaradas en cláusulas throws duplicadas, excepciones no chequeadas o subclases de otras excepciones declaradas	checkstyle	INFORMATIVA	Activo


 <p><b>JUNTA DE ANDALUCÍA</b> CONSEJERÍA DE HACIENDA Y ADMINISTRACIÓN PÚBLICA</p>	<p align="center"><b>Metodología Desarrollo y Calidad</b></p> <hr/> <p align="center">Normativa Técnica Java</p>
---	--

Título	Descripción	Plugin	Prioridad	Estado
Right Curly	<p>Revisa que se cumpla la política de cierre de llaves en bloques ELSE, TRY y CATCH.</p> <p><i>option=same</i> la llave de cierre '}' debe aparecer en la misma línea de la declaración del bloque.</p> <p>Por ejemplo:</p> <pre>if (condición) {     ... } else {     ... }</pre> <p><i>shouldStarLine=true</i> indica que el cierre se produzca en una nueva línea</p>	checkstyle	INFORMATIVA	Activo
Unnecessary Final Modifier	Revisa el uso de la cláusula final en métodos que contengan una clase tipo final, ya que éstos la llevan implícita.	pmd	INFORMATIVA	Activo
Unused Imports	Revisa que no existan import declarados en una clase que no sean utilizados posteriormente	checkstyle	INFORMATIVA	Activo
Unused imports	Revisa que no existan import declarados en una clase que no sean utilizados posteriormente	pmd	INFORMATIVA	Deshabilitada
Unused Modifier	Revisa que no existan modificadores declarados en una clase que no sean utilizados posteriormente	pmd	INFORMATIVA	Activo
Avoid Instanceof Checks In Catch Clause	<p>En bloques 'catch' sólo se debe tratar la excepción que se captura. Evitar utilizar sentencias del siguiente tipo:</p> <pre>try {     //instrucciones } catch (Exception e) {     if (e instanceof IOException) {         cleanup();     } }</pre> <p>Utilizar en su lugar:</p> <pre>try {     //instrucciones } catch (IOException e) {     //cleanup(); }</pre>	pmd	LEVE	Activo
Bad practice - Finalizer does nothing but call superclass finalizer	Comprueba la Existencia del método finalize() que solamente invoca al método finalize() de la superclase, haciendo que sea redundante.	findbugs	LEVE	Activo
Bad practice - Iterator next() method can't throw NoSuchElementException	El método next() debe ser modificado para que eleve la excepción NoSuchElementException si es invocado cuando no hay más elementos que devolver	findbugs	LEVE	Activo
Collapsible If Statements	<p>Detecta sentencias condicionales anidadas que pueden ser agrupadas en una sola, siempre que la condición final no sea muy compleja.</p> <p>Por ejemplo:</p> <pre>//evitar este anidamiento condicional if(x){     if(y){         // do stuff     } } //usar en su lugar if(x &amp;&amp; y){     // do stuff }</pre>	pmd	LEVE	
Correctness - Creation of ScheduledThreadPoolExecutor with zero core threads	Comprueba si se está creando un ScheduledThreadPoolExecutor con el número de hilos a 0, lo cual nunca ejecutará nada. En estos casos, es ignorado el intento de cambiar el tamaño máximo del pool	findbugs	LEVE	Activo
Correctness - Field not initialized in constructor	Revisa que los atributos de una clase son inicializados en algún constructor de ella. Se evita la posibilidad de aparición de excepciones de tipo NullPointerException	findbugs	LEVE	Activo
Correctness - Method ignores return value	Comprueba si no se está chequeando el valor devuelto por una función después de ser invocada	findbugs	LEVE	Activo
Dodgy - Class is final but declares protected field	Comprueba si existen clases declaradas como finales con campos de tipo protected	findbugs	LEVE	Activo
Dont Import Sun	Comprueba si existe una sentencia de importación del paquete sun.* ya que no es portable, siendo necesario modificarla	pmd	LEVE	Activo
Duplicate Imports	Existencia de sentencias import de clases duplicadas	pmd	LEVE	Activo
Final Field Could Be Static	Comprueba si existen campos final a los que se les asigna un valor constante, pudiendo ser de tipo static	pmd	LEVE	



Título	Descripción	Plugin	Prioridad	Estado
Local variable could be final	Revista que las variables que sólo son asignadas una vez deben ser declaradas como finales	pmd	LEVE	Activo
Method Argument Could Be Final	Comprueban si existen argumentos en los métodos que no son reasignados, pudiendo ser declarados final	pmd	LEVE	
Use String Buffer Length	Comprueba si se están utilizando las siguientes llamadas para conocer la longitud de una cadena, using <code>StringBuffer.toString().equals("")</code> or <code>StringBuffer.toString().length() == ...</code> , en vez de utilizar <code>StringBuffer.length()</code>	pmd	LEVE	Activo
Useless String Value Of	No es necesario invocar a <code>String.valueOf()</code> para concatenar a un String, se puede utilizar el argumento <code>valueOf()</code> directamente	pmd	LEVE	
OverrideBothEqualsAndHashCode		pmd	GRAVE	
MethodReturnsInternalArray	Comprueba si se devuelve en los métodos de forma directa arrays internos, en vez de devolver una copia de ellos	pmd	GRAVE	Activo
MissingBreakInSwitch	Debe haber al menos una sentencia break en cada case de switch	pmd	GRAVE	Activo
BrokenNullCheck	Comprueba si un método es invocado sobre un objeto que tiene valor Null, pudiendo elevar una excepción	pmd	GRAVE	Activo
ProperCloneImplementation	Comprueba si el objeto Clone() es implementado a partir de super.clone(). Ejemplo:  <code>class Foo{ public Object clone(){ return new Foo(); // This is bad } }</code>	pmd	GRAVE	Activo
UncommentedEmptyConstructor	Existen instancias donde el constructor no contiene ninguna sentencia	pmd	CRITICA	Activo
UnnecessaryConstructor	Detecta constructores que no son necesarios: cuando hay un constructor, público, tiene el cuerpo vacío y no tiene argumentos	pmd	CRITICA	Activo
UnusedPrivateMethod	Detecta métodos privados que han sido declarado pero que no son utilizados	pmd	CRITICA	Activo
EmptySynchronizedBlock	Detecta la existencia de bloques de sincronización vacíos	pmd	GRAVE	Activo



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

### 3. INTERFAZ DE USUARIO

El objetivo de este apartado es la estandarización de la apariencia de las aplicaciones de la DGPD de tal forma que se defina un estilo y éste se aplique a todas las aplicaciones que se desarrollen.

En MADEJA se ha publicado una serie de directrices para la normalización de las **Interfaces**, proporcionando prototipos de pantallas con la distribución de los elementos visuales y componentes dinámicos, persiguiendo la mejora en la usabilidad y accesibilidad de las mismas.

#### 3.1. Esquema general de la aplicación

##### 3.1.1. Pantallas de primer nivel

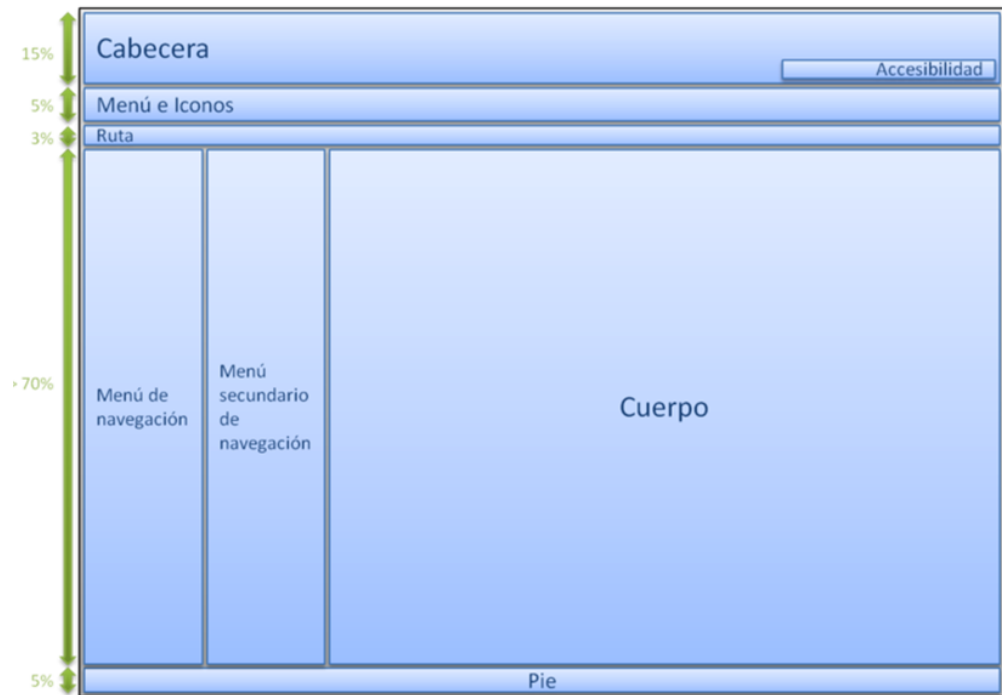
Se corresponden con las pantallas principales que ocupan todo el espacio de la ventana. No se incluyen aquí los cuadros de diálogo y otras pantallas auxiliares.

##### ○ **Elementos de la pantalla:**

Las pantallas de la aplicación deben disponer de los siguientes elementos organizativos:

- **Cabecera:** Cabecera de la aplicación que contempla como estándar la inclusión del logotipo y denominación del Organismo, así como el logotipo y denominación de la aplicación. Todo ello bien sobre un fondo corporativo o sobre el fondo representativo del diseño de la aplicación.  
  
Puede contener elementos referentes a una mejora en la accesibilidad, incluyendo iconos que faciliten cambios de tamaño en el texto, resalte de colores, etc.
- **Ruta:** Establece la ruta (migas de pan) desde el inicio de la aplicación a la situación actual donde se encuentre el usuario. Permite la navegación a los diferentes niveles por los que se ha navegado previamente a la llegada a la pantalla actual dentro de la aplicación.
- **Menú Principal:** Estandarización del menú principal, ubicado inicialmente en la barra superior, conteniendo los iconos para el cierre de la aplicación, acceso a la ayuda, vuelta al inicio de la aplicación, un método de contacto, etc.
- **Menú de Navegación:** Menú que permita la navegación a otros elementos de la aplicación, en el que se contempla hasta dos niveles de menú de navegación y se propone su ubicación como barra lateral izquierda para cumplir con las pautas de usabilidad.
- **Pie:** Establecer la barra inferior de la aplicación, permitiendo contemplar el aviso legal, la propiedad de contenidos y la propiedad de la aplicación, o cualquier información importante que deba ser mostrada en el uso del Sistema de información.

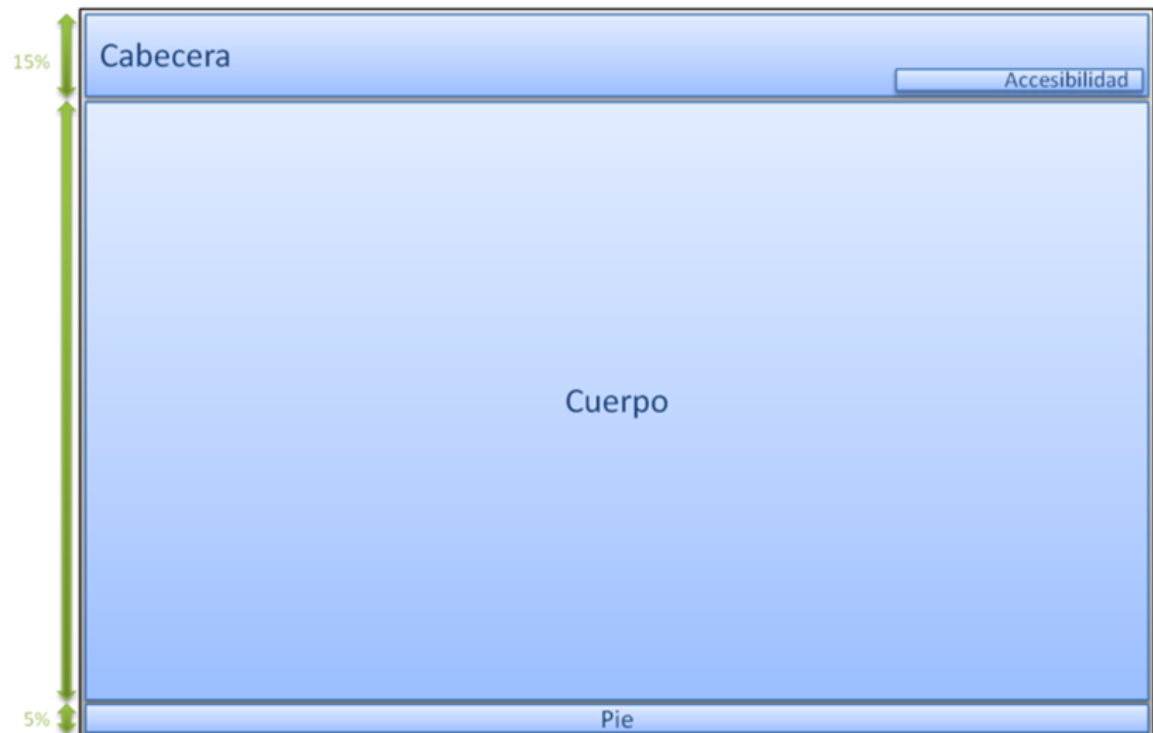




### 3.1.2. Pantallas de segundo nivel


Son las que se utilizan para apoyar elementos de las pantallas de primer nivel. Por ejemplo: pantallas de información, de ayuda, de muestra de imágenes a mayor tamaño, etc.

El esquema de este tipo de pantallas es:



Hay que tener en cuenta:



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

- Las pantallas de segundo nivel no incluirán en ningún caso menú principal.
- Cualquier pantalla de segundo nivel dispondrá de algún mecanismo para volver a la pantalla origen.
- Las pantallas de segundo nivel no deben navegar a nuevas pantallas de segundo nivel.

### **3.2. Funcionamiento general de la aplicación**

En este punto se pretende normalizar una serie de comportamientos de las aplicaciones para que faciliten su uso.

De esta manera se pretende conseguir que se pueda garantizar similitud en el comportamiento de aplicaciones que se desarrollen de forma independiente.

#### **3.2.1. Acceso a la aplicación**

Se distinguen dos situaciones:

- Si la aplicación requiere autenticación: Se accederá a la pantalla principal tras la autenticación positiva
- Si la aplicación no requiere autenticación para navegar, pero si existe la posibilidad de hacerlo, debe permitirlo en cualquiera de sus pantallas de primer nivel y, al realizar la autenticación, debe volver a la navegación a la pantalla en la que se encontraba.

#### **3.2.2. Desconexión**

Al desconectar de una aplicación redirigir al usuario a la pantalla de inicio o, en su defecto, a la de acceso.

#### **3.2.3. Atrás**

Toda aplicación debe poseer siempre un botón que permita volver a la pantalla anterior. Este botón puede ser el botón atrás del navegador o puede ser parte de la propia aplicación, en cuyo caso debe anularse el del navegador si no realiza esta acción de forma correcta.

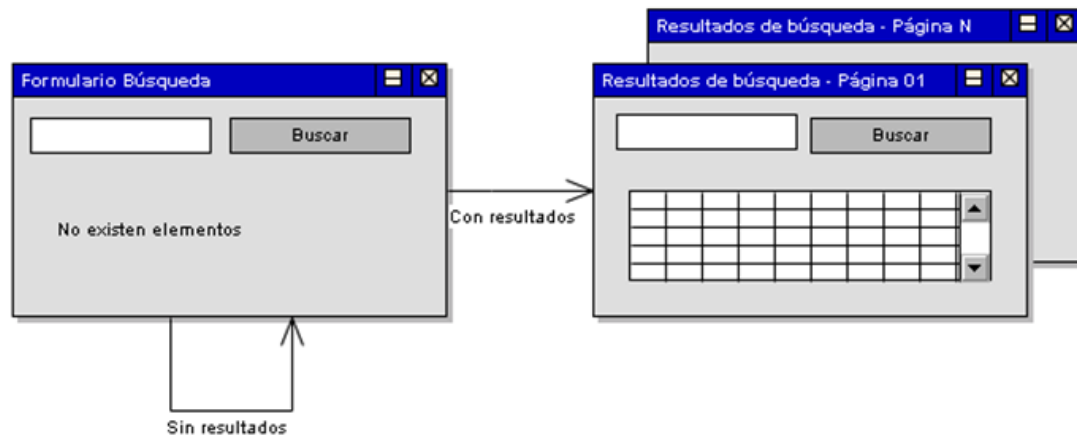
#### **3.2.4. Listados**

Una aplicación puede contener una pantalla específica para realizar búsquedas. En ese caso, si el número de campos de búsqueda es pequeño, los resultados pueden mostrarse en la misma pantalla. Cuando se accede a ella por primera vez, debe aparecer un mensaje que indique que no existen resultados de búsqueda o que aún no se ha realizado ninguna. La zona donde ubicar este mensaje es donde se mostrarían los resultados de la búsqueda.

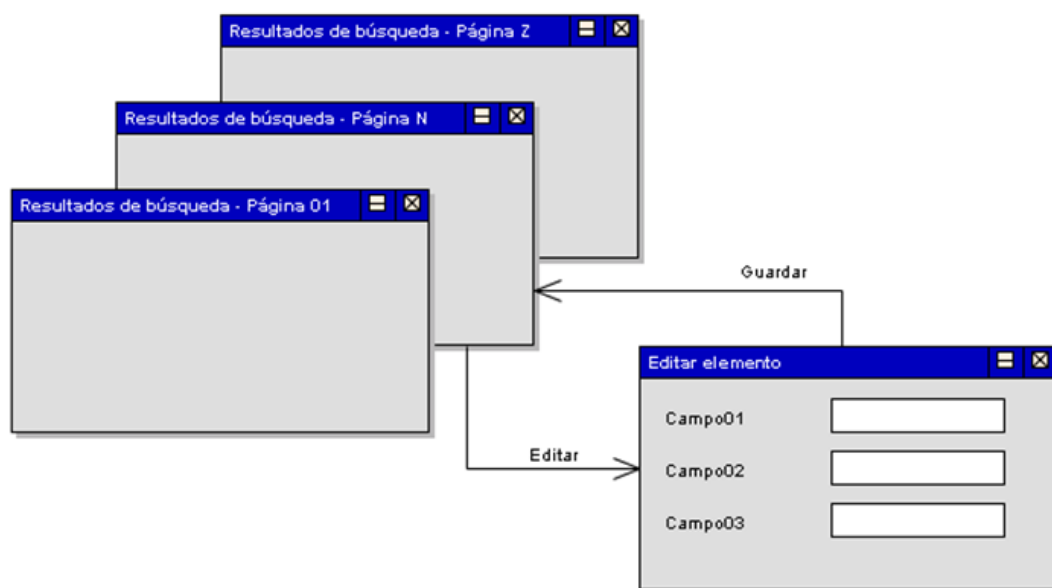
Mostrar un mensaje que indique que no existen resultados cuando la búsqueda no devuelva ningún elemento.







Muchos de los listados que se ofrecen en las aplicaciones permiten operar con elementos que aparecen en ellos. Cuando se realiza alguna operación, por ejemplo de modificación o borrado de un elemento, la aplicación debe redirigir al usuario a la página del listado donde se encontraba antes de realizar esta acción.



Cuando los usuarios se mueven entre las diferentes páginas de un listado y/o entre los elementos que lo forman, debe mantenerse el orden seleccionado, la paginación y el texto de búsqueda, en caso de que la lista sea el resultado de esa operación.

Los elementos de un listado se deben mostrar en una tabla de datos y ocupando la anchura máxima de la pantalla. De esta forma se facilita la comparación entre los elementos y la lectura de los diferentes campos.

Los tamaños de las columnas de un listado deben ajustar su tamaño en función de los datos que contienen, en detrimento del tamaño de las cabeceras.



### 3.2.5. Formularios de introducción de datos

Se pueden plantear dos situaciones: validación manual y validación automática:

#### - **Validación Manual:**

Cuando los formularios poseen varias pantallas y/o datos sensibles que no pueden ser validados de forma automática, como ocurre por ejemplo con transferencias bancarias, debe existir una pantalla en la que el usuario pueda validar los datos que ha introducido. Las pantallas y su flujo de navegación debe ser el siguiente:

1. Formulario de introducción de datos: el usuario introduce los datos a enviar.  
Si se envía el formulario, pasar al punto 2.
2. Pantalla de validación de datos introducidos: el usuario puede revisar los datos que ha introducido.  
Si se envía el formulario, pasar al punto 3.  
Si se desean modificar los datos, volver al punto 1.
3. Envío definitivo: los datos se envían definitivamente y se informa al usuario del resultado

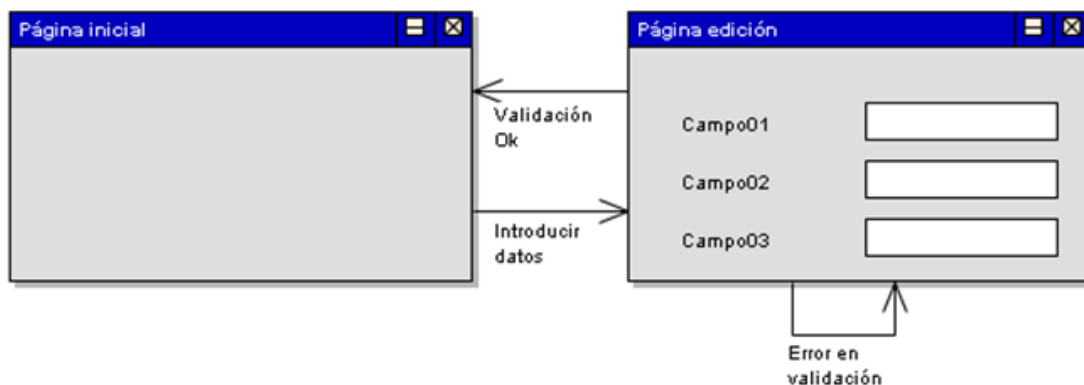


#### - **Validación automática:**

Si un formulario es validado de forma automática, debe mantener las siguientes pantallas y flujos de navegación:

1. Formulario de introducción de datos: el usuario introduce los datos a enviar.  
Si se envía el formulario y pasa la validación, pasar al punto 2.  
Si se envía el formulario y no pasa la validación, volver al punto 1 e informar al usuario de los errores.
2. Envío de datos: los datos se envían y se informa al usuario del resultado.





### 3.2.6. Ayuda

Si en cualquier momento se accede a una página de ayuda se debe ofrecer algún mecanismo para volver a la página en la que se encontraba el usuario. Además, deben mantenerse elementos como la paginación y el orden, en el caso de que se accediese a la ayuda desde un listado, y el texto de búsqueda, en el caso de que se accediese desde el resultado de una búsqueda.

### 3.2.7. Ampliar imagen o su descripción

Los contenidos que se presentan en las aplicaciones pueden contener imágenes. A veces, estas imágenes pueden ser mostradas con un tamaño mayor al que aparece en el contenido utilizando una página distinta; otras veces lo que se desea es mostrar una descripción ampliada de la misma. En ambos casos, debe ofrecerse algún mecanismo para volver a la página en la que se encontraba el usuario manteniendo elementos que existiesen como paginación y orden de elementos.

### 3.2.8. Nombre de usuario y desconexión

Cuando un usuario se ha autenticado en una aplicación, deben mostrarse el nombre de usuario y un enlace para desconectarse en todas las pantallas. Una excepción serían las pantallas de ayuda y las que muestran las imágenes ampliadas o las descripciones extensas de las mismas, en las que no es necesario mostrar estos elementos.

### 3.2.9. Anclajes

Cuando se muestra en pantalla un contenido que provoca la aparición de la barra de desplazamiento vertical, debe incluirse un hipervínculo a la parte superior de la pantalla en la parte inferior de la página. De esta forma se facilitará el acceso a los elementos constantes de navegación.

## 3.3. Prototipos de pantallas

En MADEJA (Subsistemas, Interfaz de usuario, Normalización de interfaces, prototipos de pantallas) están disponibles distintos esquemas de pantallas según la funcionalidad a implementar: búsquedas avanzadas, búsqueda simple, login, consulta de detalle, introducción de datos, ayuda, error, listado simple, listado avanzado, confirmación de eliminación, inicio y de contenidos que pueden utilizarse para los diseños de las pantallas de las aplicaciones que se desarrollen en la DGPD.

Algunos ejemplos son:



Búsqueda simple de elementos

Logotipo Junta de Andalucía

Logotipo Organismo

Logotipo Aplicación

Menú horizontal

Estás en: Inicio >> Búsqueda simple de elementos

Búsqueda simple de elementos

Campo01

Campo02

Buscar

Resultados de la búsqueda

Número resultados a mostrar: 25 | 50 | 100 | Todos

Imprimir

Exportar a: Atom | CSV | PDF

	Columna01	Columna02	Columna03	
<input type="checkbox"/>	texto	dígitos	texto	☒☒☒☒
<input type="checkbox"/>	texto	dígitos	texto	☒☒☒☒
<input type="checkbox"/>	texto	dígitos	texto	☒☒☒☒
<input type="checkbox"/>	texto	dígitos	texto	☒☒☒☒
<input type="checkbox"/>	texto	dígitos	texto	☒☒☒☒
<input type="checkbox"/>	texto	dígitos	texto	☒☒☒☒
<input type="checkbox"/>	texto	dígitos	texto	☒☒☒☒
<input type="checkbox"/>	texto	dígitos	texto	☒☒☒☒
<input type="checkbox"/>	texto	dígitos	texto	☒☒☒☒
<input type="checkbox"/>	texto	dígitos	texto	☒☒☒☒

27 Resultados

Página 1 de 3

Pie de página

Introducción de datos

Logotipo Junta de Andalucía

Logotipo Organismo

Logotipo Aplicación

Menú horizontal

Estás en: Inicio >> Introducción de datos

Introducción de datos

Campo01

Campo02

Campo03

● Opción01

○ Opción02

Campo04

Opción A

Campo05

□ Opción01


□ Opción02

Campo06

Enviar

Pie de página



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

### 3.4. Prototipos de manipulación de datos

Los componentes de manipulación de datos se utilizan normalmente en formularios. Éstos suelen incluir una secuencia de comandos en cliente que valida los datos proporcionados por el usuario antes de enviar la información al servidor.

En general, lo mejor es realizar en el cliente tantas comprobaciones como sea posible pero todas deben repetirse en el servidor.

La doble validación formada por cliente y servidor garantiza que aunque el cliente tenga desactivado Javascript se validen los contenidos.

Las pautas a seguir en este apartado según MADEJA son:

#### 3.4.1. Usuario y Contraseña

El nombre de usuario y la contraseña deben tener un tamaño mínimo de 6 caracteres y un máximo de 12. Para solicitar el nombre de usuario debe utilizarse un campo de tipo "text".

Para insertar una contraseña se debe utilizar un campo de tipo "password". De esta forma se consigue que todos los caracteres escritos en él sean visualizados como asteriscos. Al no poder visualizar los caracteres, es más difícil detectar errores en la introducción de datos por lo que la contraseña debe solicitarse por duplicado.

#### 3.4.2. NIF

Dependiendo de si se trata de una persona física o jurídica el NIF poseerá un formato u otro pero siempre tendrá un tamaño de 9 caracteres en total. Utilizar un campo de tipo "input type='text'".


En el caso de que el NIF sea de tipo DNI, se puede comprobar que la letra indicada se corresponde con la numeración introducida. Para calcular la letra y poder comprarla, se utiliza el algoritmo de módulo 23:

```
function calculaLetraDni(digitosDelDNI) {
    var tablaLetras = 'TRWAGMYFPDXBNJZSQVHLCKE';
    return tablaLetras.charAt(digitosDelDNI % 23);
}
```

Para validar el NIF que posee un tipo distinto a DNI, se debe seguir el siguiente algoritmo:

- Obtener los 7 dígitos centrales
- Sumar los dígitos de las posiciones pares y almacenar el resultado en A
- Para cada uno de los dígitos de las posiciones impares, multiplicarlo por 2 y sumar los dígitos del resultado almacenándolo en B
- Sumar A + B y almacenar el resultado en C
- Tomar el dígito de las unidades de C, que se llamará E
- Si el dígito E es distinto de 0 lo restaremos a 10 y almacenamos el resultado en D
- Si el dígito E es 0 y el dígito de control es numérico, tomar D=0



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

- Obtener el dígito de control a partir de D:
  - Si ha de ser numérico es directamente D
  - Si es una letra, se corresponde con la relación: J = 0, A = 1, B = 2, C = 3, D = 4, E = 5, F = 6, G = 7, H = 8, I = 9

### 3.4.3. Correo electrónico

Para solicitar el correo electrónico debe utilizarse un campo de tipo "text" que permita introducir hasta 256 caracteres.

Una dirección de correo electrónico válida debe cumplir las siguientes condiciones:

- Contener "@"
- La longitud de la parte local (antes del símbolo "@") debe estar comprendida entre 1 y 64 caracteres.
- La longitud de la parte de dominio (después del símbolo "@") debe estar comprendida entre 4 y 255 caracteres.
- La longitud total debe ser menor o igual a 256 caracteres.
- La parte local y la parte de dominio deben comenzar por una letra o dígito y no deben contener dos símbolos "." consecutivos
- La parte local y la parte de dominio pueden contener letras, números y los caracteres ".", "\_" y "-".
- La parte del dominio debe terminar con un símbolo "." y entre dos y cuatro caracteres alfabéticos.
- El chequeo de registros DNS con la parte del dominio debe ser de tipo "A" o "MX".

### 3.4.4. Teléfono

Para introducir un número de teléfono de ámbito nacional, debe utilizarse un componente de tipo "text" que permita introducir 9 caracteres.

Para introducir un número de teléfono de ámbito internacional, debe utilizarse un componente de tipo "text" que permita introducir 15 caracteres.


El número de teléfono internacional podrá estar construido de las siguientes formas:

- Signo "+" + prefijo internacional del país + número de abonado. Ejemplo: +34912345678
- Caracteres "00" + prefijo internacional del país + número de abonado. Ejemplo: 0034912345678

Para validar un número de teléfono de ámbito nacional, se deben tener en cuenta las siguientes condiciones:

- Longitud igual a 9 caracteres.
- Los caracteres deben ser numéricos.



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

- Si el número es de línea fija, debe comenzar por 9.
- Si el número es de línea móvil, debe comenzar por 6 o 7.

### 3.4.5. Dirección Postal

Para introducir la dirección sólo se deben utilizar tres campos:

- **Dirección.** Debe permitir introducir todos los elementos necesarios para localizar un domicilio dentro de una localidad. Estos elementos pueden ser: tipo de vía, nombre, número, bloque, planta, letra, etc.
- **Localidad.** Almacena los caracteres correspondientes al nombre de la localidad. Debe estar limitado a 48 caracteres.
- **Código postal.** Contiene los dígitos correspondientes al código postal, por lo que se debe limitar a 5 caracteres

Para validar la dirección postal, se deben realizar las siguientes comprobaciones:

- Verificar la existencia de la localidad
- Verificar la existencia del código postal
- Verificar la correspondencia del código postal con la localidad

### 3.4.6. Código Cuenta Corriente


El Código Cuenta Cliente (CCC) es un código utilizado en España por las entidades financieras (bancos y cajas) para la identificación de las cuentas de sus clientes. Consta de veinte dígitos que están divididos en cuatro bloques distintos de acuerdo con la siguiente estructura:

- Los primeros cuatro dígitos son el Código de la Entidad, que coincide con el Número de Registro de Entidades, NRBE del Banco de España.
- Los siguientes cuatro dígitos identifican la oficina.
- Los siguientes dos dígitos son los llamados dígitos de control, que sirven para validar el CCC.
- Los últimos diez dígitos identifican unívocamente la cuenta.

Para obtener cada uno de los dígitos de control se realiza el procedimiento siguiente:

- Para el primer dígito: puesto que el código conjunto de Entidad y de Oficina tiene tan solo ocho cifras, se completa con dos ceros (00) por la izquierda para hacer la comprobación.
- Cada uno de los dígitos que componen el código se multiplica por un factor asociado a su posición en el código. Los factores para cada posición, de izquierda a derecha, son: 1, 2, 4, 8, 5, 10, 9, 7, 3, 6.
- A continuación, se suman los diez productos obtenidos.
- El resultado de esta suma se divide por 11 y se anota el resto que produce la división.



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

- Este resto se resta de 11 para obtener el dígito de control correspondiente a cada uno de los códigos.
- Puesto que estamos interesados en obtener solo una cifra, si la cantidad resultante fuese 10, se tomará en su lugar el dígito 1; y si fuese 11, el 0.

### 3.4.7. Campos numéricos

Cuando se introduce un número decimal, se debe permitir el uso del punto (".") y la coma (",") como separadores decimales.

Los campos numéricos deben alinearse a la derecha y respetar la posición del separador decimal para que puedan compararse de una forma más sencilla.

### 3.4.8. Fechas

Para introducir una fecha debe utilizarse un campo de tipo "text". Según el formato de fecha que se utilice, se pueden necesitar hasta 10 caracteres (dd-mm-aaaa), por lo que el componente debe permitir esa longitud máxima.

Cuando se solicita una fecha al usuario, deben aceptarse los siguientes formatos:

- d-m-aa
- d-mm-aa
- dd-mm-aa
- d-m-aaaa
- d-mm-aaaa
- dd-mm-aaaa

Además, deben aceptarse indistintamente como separador los símbolos guión ("-") y barra inclinada ("/").

### 3.4.9. Horas

Para introducir una hora debe utilizarse un campo de tipo "text" que debe tener un tamaño máximo de 8 caracteres para que también se puedan incluir los segundos.

Al solicitar una hora, se debe utilizar siempre el formato de 24 horas y se deben aceptar las siguientes entradas:

- H:MM
- H:MM:SS
- HH:MM
- HH:MM:SS

### 3.4.10. URL

El formato general de una URL es "esquema://máquina/directorio/archivo" donde:





- **Esquema:** generalmente indica el protocolo de red que se usa para recuperar la información del recurso identificado. Un URL comienza con el nombre de su esquema, seguido por dos puntos, seguido por una parte específica del esquema. Algunos ejemplos de esquemas son: http, https, ftp, mailto, ldap
- **Máquina:** es la forma de identificar al servidor. Normalmente es el dominio, aunque a veces suele contener el puerto.
- **Directorio:** ruta hasta la carpeta que contiene el archivo buscado
- **Archivo:** archivo que se desea localizar. También puede contener variables que son enviadas a través de la URL

Se deben validar los caracteres de una URL, siendo los siguientes los caracteres válidos:

- Alfabéticos: a b c d e f g h i j k l m n o p q r s t u v w x y z  
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- Numéricos: 0 1 2 3 4 5 6 7 8 9
- Especiales: - \_ . ! \* ' ( ) ~
- Reservados: \$ & + , / : ; = ? @ # %

#### 3.4.11. Nombre y Apellidos

Para introducir el nombre y apellidos debe utilizarse un campo distinto para cada uno de tipo "text". Este campo debe permitir insertar hasta 120 caracteres.

### 3.5. Manual de estilo genérico

El objetivo del manual es normalizar la estructura de los contenidos y el diseño de las aplicaciones software homogeneizando estilos y estructuras para facilitar el desarrollo de nuevas páginas y actualizaciones posteriores.

#### 3.5.1. Gama cromática

Utilizar el color blanco (#ffffff) para el fondo de página.

Se debe utilizar el gris oscuro para los textos en general, debido al alto contraste que ofrece sobre el fondo blanco (#383D44).

Utilizar el color corporativo básico "#087021" y sus complementarios "#CCC47C" y "#000000" como colores principales.


#### 3.5.2. Tipografía

La tipografía usada para todos los textos de la aplicación debe ser **Verdana, Arial o Georgia**, en este orden.

Especificar el tamaño de la fuente en unidades relativas, siendo el tamaño general del 70% o 0.7em.

Utilizar peso normal en los contenidos y negrita en títulos. Se debe eliminar el uso excesivo de la negrita porque destruye la uniformidad del gris tipográfico y produce ruido visual. Por ello, su uso se restringe a destacar títulos.



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

### 3.5.3. Elementos gráficos

Se debe utilizar un número reducido de imágenes para que la aplicación sea lo más rápida y ágil posible. Sin embargo, se recomiendan imágenes que contribuyan a hacer entender contenidos basados en texto y a hacer más atractivos textos extensos.

Los iconos que se utilicen en diferentes elementos deben tener formato **png** y un tamaño adecuado de entre los siguientes: **8x8** píxeles, **12x12** píxeles, **16x16** píxeles, **24x24** píxeles.

### 3.5.4. Creación de página

Se debe utilizar el tipo de documento XHTML 1.1.

Se deben utilizar hojas de estilo almacenadas en ficheros independientes.

### 3.5.5. Buscador genérico

El texto que debe mostrarse en el botón que acompañe al buscador es "Buscar".

El buscador genérico debe estar compuesto por un cuadro de texto y un botón que ejecute la búsqueda.

### 3.5.6. Tablas

Las tablas no deben utilizarse para diseñar, sólo para mostrar datos. Cada columna de una tabla debe tener una cabecera que indique claramente qué datos se presentan en ella.

Las filas que contengan datos deben mostrarse alternando colores de fondo diferentes.

### 3.5.7. Menú horizontal

El menú horizontal debe poseer un único nivel que dé acceso a las principales secciones de la aplicación.

### 3.5.8. Formularios

Todos los formularios de una aplicación deben poseer un título.

Los formularios deben estar divididos en grupos de campos relacionados a través de las etiquetas "<fieldset>" y "</fieldset>".

### 3.5.9. Enlaces

Los enlaces que se encuentren dentro de los contenidos, deben mostrarse subrayados para que sean fácilmente distinguibles.


Los enlaces que hayan sido visitados deben mostrarse en un color diferente a los no visitados.

### 3.5.10. Listados

Una aplicación puede mostrar diferentes listados para enumerar un conjunto de elementos o como resultado de una búsqueda. Todos los listados deben permitir ser exportados a diferentes formatos entre los que se deben incluir: **pdf** y **csv**.

Las aplicaciones suelen presentar los listados divididos en varias páginas para facilitar su lectura. Sin embargo, cuando desea realizar una exportación a uno de los formatos posibles, ésta debe hacerse para el conjunto completo de los datos que forman el listado.



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

## 4. SEGURIDAD EN JAVA

Una aplicación es segura si el comportamiento en su ejecución es el definido dentro de sus especificaciones previstas, sin mostrar alteraciones provenientes de modificaciones de terceros.

### 4.1. Análisis y Diseño

- Los análisis y diseños que se realicen tendrán en cuenta la aplicación de la Ley Orgánica de protección de Datos (LOPD).
- Toda aplicación que cuente con gestión documental (incorporación de documentos y archivos) deberá incorporar comprobación antivirus en la subida de documentos.
- Las aplicaciones deben incorporar un sistema de auditoría en el que queden reflejadas las acciones realizadas por cada usuario. Se pondrá especial énfasis en el registro de los intentos de autenticación, tanto los correctos como los fallidos.
  - Autenticación, intentos de identificación de usuarios y logout de la aplicación
  - Toda acción que modifique información funcional del sistema
  - Formato por defecto:
   
`<YYYY-MM-DD HH24:MM:SS> [<Clase>] <LEVEL> <mensaje>`
- Utilización de SSL, al menos, en la autenticación y en los accesos a páginas con información privada.
- Para la autenticación de los usuarios en cada aplicación se delegará en el sistema de servicio de directorio implantado en la Consejería (**LDAP**) utilizando los API definidos para ello.
- Las aplicaciones con autenticación tienen que disponer de la funcionalidad de logout visible en todas las páginas que requieran estar autenticado.
- El envío de la información de la página de autenticación debe ser realizada bajo SSL para evitar la transmisión sin encriptar del usuario y clave.
- Las aplicaciones bloquearán las cuentas de aquellos usuarios que no hayan accedido a ella durante más de un mes. Estas cuentas no serán eliminadas, pueden volver a ser activadas por los administradores de la aplicación

### 4.2. Desarrollo

- Los valores de los campos de los formularios se deben proteger para que no se tomen como tal los caracteres especiales, con el objetivo de evitar la inyección de código.
- No utilizar el método GET en el envío de información especialmente sensible, como la contraseña, en formularios web. El método GET muestra los valores de las variables en la propia cadena de la URL.
- No se utilizarán consultas SQL construidas mediante la concatenación de entradas del usuario. Utilizar siempre *preparedStatements* convenientemente parametrizados.





```
String param = getParameter("userName");

PreparedStatement ps = null;
RecordSet rs = null;
try {
    ps = conn.prepareStatement("SELECT * FROM user_table
WHERE username =      '" + param + "'");
    rs = ps.execute();
    if ( rs.next() ) {

        // trabajar con el registro actual
        ...
    }
}
catch (...) {
    ...
}
```



```
String param = getParameter("userName");

PreparedStatement ps = null;
RecordSet rs = null;
try {
    ps = conn.prepareStatement("SELECT * FROM user_table
WHERE username = '?'");
    ps.setString(1, param);
    rs = ps.execute();
    if ( rs.next() ) {

        // trabajar con el registro actual
        ...
    }
}
catch (...) {
    ...
}
```

- Evitar el cacheado de páginas con información privada mediante el uso de las cabeceras HTTP correspondientes.

HTTP/1.0:




```
<META HTTP-EQUIV="PRAGMA" CONTENT="NO-CACHE">
```

HTTP/1.1:



```
<META HTTP-EQUIV="CACHE-CONTROL" CONTENT="NO-CACHE">
```



	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

- Los ficheros generados dinámicamente para entregar al usuario no deben ser escritos a disco (ni siquiera en directorios temporales) salvo que sea necesaria su utilización posterior o se requiera su almacenaje. En tal caso, deberán ser escritos en los directorios y los permisos que se indiquen).
- Los datos de entrada del usuario serán validados en cada capa del sistema para garantizar la consistencia de cada capa de forma independiente.
- Se debe estipular un tiempo máximo de inactividad para la sesión, una vez transcurrido este tiempo, el usuario debe volver a identificarse antes de interactuar con la aplicación. En Tomcat puede realizarse de tres modos diferentes:
  - En el **web.xml** de la configuración del Tomcat, se establece el valor por defecto para todas las aplicaciones.
  - En el **WEB-INF/web.xml** de cada aplicación, se puede especificar un tiempo máximo de inactividad específico.
  - Estos valores se pueden sobrescribir desde el código de las aplicaciones mediante el método ***session.setMaxInactiveInterval***

Ejemplo de configuración en el WEB-INF/web.xml de una aplicación en Tomcat:



```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
  2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

<display-name>Nombre aplicacion</display-name>
<description>
  Descripcion...
</description>

<session-config>
  <session-timeout>
    20
  </session-timeout>
</session-config>

</web-app>
```

- Las aplicaciones que, excepcionalmente no hagan uso del **LDAP**, deben poseer la funcionalidad de cambiar la clave. Esta debe solicitar la clave antigua, la nueva clave y confirmación de la nueva clave, no permitiéndose claves nulas o en blanco.

El tamaño mínimo de la clave será de 6 caracteres permitiéndose por lo menos hasta 20 caracteres.

Las claves se almacenarán encriptadas. Nunca se desencriptarán, realizándose las comparaciones contra la clave encriptada. Para ello se utilizarán algoritmos de encriptación de



un sólo sentido.

Se ha demostrado que **MD5** es potencialmente rompible. Se recomienda el uso de **Use AES-128 SHA1 con 256 bit**.

- En la consulta para comprobar el usuario/clave sólo debe utilizarse el campo 'usuario' en el **where**:



```
public static bool isUsernameValid(string username) {
    Regex r = new Regex("^ [A-Za-z0-9]{16}$");
    return r.IsMatch(username);
}

PreparedStatement ps = null;
RecordSet rs = null;
try {
    isUsernameValid(pUsername);
    ps = conn.prepareStatement("SELECT * FROM user_table
WHERE username =
    '?'");
    ps.setString(1, pUsername);
    rs = ps.execute();
    if ( rs.next() ) {

        // trabajar con el registro actual
        ...
    }
}
catch (...) {
    ...
}
```

- Comprobar que la consulta devuelve siempre cero o un registro, para evitar accesos fraudulentos.
- En todo momento se debe seguir una política de no acceso por defecto, es decir, ante cualquier excepción que pudiera encontrarse en el sistema de autenticación, el usuario no obtendrá acceso al sistema. El siguiente psedo-código muestra como ante una excepción en el tratamiento del usuario, este siempre obtendría acceso.



```
bAuthenticated := true

try {
    userrecord := fetch_record(username)
    if userrecord[username].password != sPassword then
        bAuthenticated := false
    end if
    if userrecord[username].locked == true then
        bAuthenticated := false
    end if
    ...
}
```



```
catch {
    // perform exception handling, but continue
}
```



```
bAuthenticated := false
securityRole := null

try {
    userrecord := fetch_record(username)
    if userrecord[username].password != sPassword then
        throw noAuthentication
    end if
    if userrecord[username].locked == true then
        throw noAuthentication
    end if
    if userrecord[username].securityRole == null or banned
then
        throw noAuthentication
    end if

    ... other checks ...

    bAuthenticated := true
    securityRole := userrecord[username].securityRole
}
catch {
    bAuthenticated := false
    securityRole := null

    // perform error handling, and stop
}
```

### 1.1.1. Validación de campos de formularios

Validación de los datos de entrada en la capa de presentación de todos los campos de cada formulario:

- **Obligatoriedad**, todos los campos obligatorios deben ser completados.
- **Formato**, se debe comprobar que cada campo se ajusta al formato esperado.
- **Longitud mínima y máxima**, se debe asegurar que la longitud de la información suministrada por el usuario está entre el mínimo y el máximo esperado para cada campo.

### 1.1.2. Autocompletado

Solicitar la desactivación del autocompletado de los formularios en los que la información que se introduzca sea privada. Algunos navegadores permiten desactivar la función de autocompletar evitando su almacenamiento.

Para desactivar todo el formulario:





<form ... AUTOCOMPLETE="off">





Para desactivar un elemento del formulario:



```
<input ... AUTOCOMPLETE="off">
```

### 1.1.3. Tratamiento de los 'values' en la generación de elementos de formulario

En la generación de los elementos de formularios que permiten la incorporación de valores, como radio buttons, checkboxes y selects/options, se deben generar de modo que no se muestren los valores reales, ya que favorecen la aparición de un tipo de inyección de SQL.

La generación de esta información suele construirse de forma similar a:



```
<html:radio value="<%=acct.getCardNumber(1).toString( )%>"
property="acctNo">
<bean:message key="msg.card.name" arg0="<
%=acct.getCardName(1).toString( )%>"
/>

<html:radio value="<%=acct.getCardNumber(1).toString( )%>"
property="acctNo">
<bean:message key="msg.card.name" arg0="<
%=acct.getCardName(2).toString( )%>"
/>
```

Generando algo como:



```
<input type="radio" name="acctNo" value="455712341234">Gold
Card

<input type="radio" name="acctNo"
value="455712341235">Platinum Card
```

Si el valor que se obtiene de estos campos es utilizado directamente en una consulta SQL, puede ocurrir una forma de inyección de SQL permitiendo el acceso a información de otros usuarios. Esto es posible si la construcción del SQL es algo como:



```
String acctNo = getParameter('acctNo');
String sql = "SELECT acctBal FROM accounts WHERE acctNo =
'?';
PreparedStatement st = conn.prepareStatement(sql);
st.setString(1, acctNo);
ResultSet rs = st.executeQuery();
```

Para evitarlo se debe construir de modo que el formulario envíe el índice del elemento en lugar del valor concreto. Además se debe incluir en la consulta las condiciones necesarias para asegurar que la información devuelta pertenezca sólo al usuario actual, como su identificador.



```
String acctNo =
acct.getCardNumber(getParameter('acctIndex'));
```



```
String sql = "SELECT acctBal FROM accounts WHERE acct_id =  
'?' AND acctNo =  
'?'";  
PreparedStatement st = conn.prepareStatement(sql);  
st.setString(1, acct.getID());  
st.setString(2, acctNo);  
ResultSet rs = st.executeQuery();
```

Para ello se requiere también la generación de valores desde 1 a ... x, suponiendo que los distintos elementos se almacenan en una colección que puede ser iterada de forma similar a:



```
<logic:iterate id="loopVar" name="MyForm" property="values">  
<html:radio property="acctIndex" idName="loopVar"  
value="value"/>&nbsp;  <br />  
<bean:write name="loopVar" property="name"/><br />  
</logic:iterate>
```

La generación de HTML quedaría, por tanto:



```
<input type="radio" name="acctIndex" value="1" />Gold Credit  
Card  
  
<input type="radio" name="acctIndex" value="2" />Platinum  
Credit Card
```

- En toda nueva aplicación que se incorpore a la Consejería se le eliminarán tanto los usuarios y claves por defecto como los utilizados durante su desarrollo y pruebas.
- A cada persona que necesite interactuar con una aplicación se le asignará un usuario único para su utilización. No se compartirán usuarios de aplicaciones entre dos o más personas.



## 5. RENDIMIENTO EN JAVA

Al igual que la Seguridad de las Aplicaciones, la temática de Rendimiento debe ser contemplada desde el inicio del Diseño de los Sistemas de Información.

### 5.1. Rendimiento en las Bases de Datos

#### 5.1.1. Particionamiento de Tablas

Gracias a la técnica de particionado el tamaño de los índices se reduce considerablemente, con lo que aquellas partes más utilizadas pueden caber en memoria. Pero no sólo va a permitir mejorar el rendimiento de las consultas. Los procesos de carga y actualización masivos también se pueden acelerar en gran medida trabajando sobre una tabla y añadiéndola posteriormente al "conjunto de particiones".

La partición es completamente transparente a las aplicaciones.

Las principales ventajas del particionado son:

- Permite operaciones de gestión de cargas de datos, la creación de índices y la reconstrucción, y el backup en el nivel de partición, en lugar de en toda la tabla. Esto da lugar, a veces, a reducir considerablemente estas operaciones.
- Mejora el rendimiento de la consulta. En muchos casos, los resultados de una consulta se pueden lograr mediante el acceso a un subconjunto de particiones, en lugar de toda la tabla.
- Puede reducir significativamente el impacto del tiempo de inactividad programado para las operaciones de mantenimiento. La independencia de la partición sobre las operaciones de mantenimiento de la partición le permite realizar las operaciones de mantenimiento simultáneas en diferentes particiones de la misma tabla. También puede ejecutar simultáneamente las operaciones SELECT y DML en las particiones que no son afectadas por las operaciones de mantenimiento.
- El particionado aumenta la disponibilidad de bases de datos de misión crítica si las tablas e índices críticos se dividen en particiones para reducir las ventanas de mantenimiento, tiempos de recuperación, y el impacto de los fallos
- El particionado pueden aplicarse sin necesidad de modificar sus aplicaciones.

#### 5.1.2. Detallar los atributos en las consultas, inserciones o actualizaciones

En la medida de lo posible debemos de detallar los campos que se requieren y evitar el uso de la sentencia "Select \*", mejorando ostensiblemente el tiempo de respuesta en la acción a realizar.



### 5.1.3. Uso de índices para la consultas

La identificación del índice a usar está relacionado con las columnas que participan en las condiciones de la orden WHERE.

Existen casos en que la presencia de la columna no garantiza el uso de su índice, ya que éstos son ignorados, como en las siguientes situaciones cuando la columna indexada es:

- Evaluada con el uso de los operadores IS NULL o IS NOT NULL
- Modificada por alguna función, excepto por las funciones MAX(columna) o MIN(columna).
- Usada en una comparación con el operador LIKE a un patrón de consulta que comienza con alguno de los signos especiales (% \_).

#### Ejemplos:

```
SELECT nombre,articulo,valor
FROM clientes,ventas
WHERE nombre IS NOT NULL;
```

```
SELECT nombre,articulo,valor
FROM clientes,ventas
WHERE UPPER(nombre)>' '
;
```


```
SELECT nombre,articulo,valor
FROM clientes,ventas
WHERE nombre
LIKE '%DEPORTE%'
;
```

Los registros con valor NULL no forman parte del índice.

## 5.2. Rendimiento de los servicios WEB

Regla	Descripción
Minimizar el tráfico de la red agrupando funcionalidades en los servicios	Se debe diseñar la interfaz del servicio web intentando minimizar el tráfico de red. Para ello es necesario agrupar la funcionalidad del servicio de manera que el número de comunicaciones necesarias decaiga. Se deben construir servicios generales, es decir, que "hagan muchas cosas y regresen mucha información". Con ello se reduce la sobrecarga de la red y se mejora el tiempo de respuesta del servicio



 <b>JUNTA DE ANDALUCÍA</b> <small>CONSEJERÍA DE HACIENDA Y ADMINISTRACIÓN PÚBLICA</small>	<b>Metodología Desarrollo y Calidad</b>
	Normativa Técnica Java

Regla	Descripción
Controlar el tamaño y complejidad de los mensajes SOAP	<p>Si se generan mensajes SOAP de gran tamaño, facilitamos la creación de cuellos de botella ya que se necesita mucho tiempo de ejecución para interpretarlos. Se debe mantener el contenido del mensaje tan pequeño como sea posible.</p> <p>En el caso de encontrarnos con mensajes muy complejos, el proceso de serialización y el inverso de deserialización consumen mucho tiempo efectivo. A ser posible hay que diseñar los mensajes minimizando la complejidad de los mismos. Sin embargo, suele ser difícil mantener un equilibrio entre complejidad y tamaño en el diseño de los mensajes</p>
Usar los tipos de datos simples	En la medida de lo posible utilizar los tipos simples de SOAP (String, Int, Float, NegativeInteger) ya que cada nuevo tipo de dato utilizado introduce un tratamiento específico con un serializador para procesar el valor y convertirlo de XML a JAVA y viceversa. Para este proceso, se necesita más tiempo en el procesado y por ende, existe una bajada de rendimiento en el servicio
Recomendaciones de diseño en SOAP	<ul style="list-style-type: none"> <li>- Si existen intermediarios en la comunicación entre el cliente y el servicio (proxy, puerta de enlace), se debe minimizar la interpretación de mensajes que realizan.</li> <li>- Si se tiene control del intérprete de XML utilizado, considerar el uso de SAX o StAX ya que suelen proveer mayor rendimiento que DOM.</li> <li>- El formato Document/Literal de los mensajes SOAP es más pequeño y menos complejo que los mensajes de tipo RPC/SOAP.</li> </ul>
Aplicación de niveles de seguridad	Hay que evaluar a qué tipo de clientes está orientado el servicio. La seguridad tiene un impacto en el desempeño. No todo el tráfico de SOAP necesita estar protegido ya que el costo en rendimiento de una seguridad de punta a punta (por ejemplo, con WS-Security) es en muchos casos mucho mayor a implementar un mecanismo de seguridad a nivel de transporte como SSL. Definir políticas de seguridad en función del tipo de servicio y del entorno del mismo
Uso de la cache en los servicios web	La utilización de caché es una forma de mejorar el rendimiento de los sistemas que hacen uso intensivo del procesador. Sin embargo esto sólo aplica para los servicios de sólo lectura.



Regla	Descripción
Conexiones en función del tipo de contenido del mensaje	<ul style="list-style-type: none"> <li>- Las conexiones persistentes son buenas para el rendimiento en casos donde existan gran cantidad de mensajes con poco contenido. Para mensajes más grandes, la diferencia no es tan considerable.</li> <li>- Las conexiones de flujo continuo (streaming) son buenas para el rendimiento en casos donde existen mensajes con gran cantidad de contenido. La codificación HTTP "chunked" es un tipo de flujo continuo soportado por HTTP/1.1.</li> </ul>
Codificación binaria de algunos elementos del mensaje	En ocasiones, efectuar una codificación binaria de elemento contenidos en el mensaje puede resultar beneficioso. En el caso de tener una capa de transporte que sea lenta, o el procesamiento pudiera ser complejo, es interesante disponer un modelo de mensajería asíncrona

