ESTÁNDARES PARA EL DESARROLLO DE APLICACIONES

# ESTÁNDAR PARA EL DISEÑO DE LA BASE DE DATOS

## Generalidades

* Los nombres de las tablas y vistas deben ir en MAYÚSCULAS y en plural, para identificar rápidamente en una sentencia SQL o consulta SQL si el nombre que estamos observando pertenece a una tabla o vista.
* Si tiene un subconjunto grande de grupo de tablas, tiene sentido dar prefijo a este grupo de tablas. el prefijo debe estar separado por un subguión.

Ejemplo: SALES\_MOVEMENTS, SALES\_DETAILS, SALES\_LOCATION

* Los nombres de las tablas y campos deben escribirse en inglés, por que las palabras en inglés en su mayoría ocupan menor espacio que las palabras en español.

Comparación: USUARIOS – USERS, DEPARTAMENTOS – DEPARTMENTS, TITULOS – TITLES

* Los nombres de los campos en las tablas y vistas deben especificarse bajo el estilo de escritura LowerCamelCase, en el que se define que la inicial de cada palabra debe ir en mayúsculas, exceptuando la primera palabra, la cual debe ir en minúsculas; además, todas las palabras deben ir juntas.

Ejemplo: salary, idEmployee, birthDate, photo, firstName, lastName

* Para los nombres de tablas, vistas, procesamientos almacenados, Disparadores, etcétera, únicamente se utilizarán caracteres alfabéticos, salvo que la naturaleza del nombre requiera dígitos numéricos. Queda prohibido el uso de otro tipo de caracteres cómo los signos de puntuación o símbolos.
* Usar el siguiente patrón de nomenclatura para los procedimientos almacenados o stored procedures(sp):

sp<ApplicationName>\_[<groupName>\_]<ActionType><tableNameOrLogicalInstance>

dónde actionType es: Get, Delete, Update, Write, Archive, Insert, Log… es decir, verbos de acción

Ejemplo: spTiendaEnLinea\_GetProductDetails

* Usar el siguiente patrón de nomenclatura para los disparadores o triggers:

TR\_<TableName>\_<ActionType><Description>

Ejemplo: TR\_PRODUCTS\_LogProductChanges

* Para índices: IX\_<TableName>\_<ColumnsSeparatedBy>

Ejemplo: IX\_UserDetails\_UserId

* Para claves primarias: PK\_<TableName>

Ejemplo: PK\_ProductDetails

* Para claves foráneas: FK\_<TableName1>\_<TableName2>

Ejemplo: FK\_UserDetails\_Emails

* Para Default: DF\_<TableName>\_<ColumnName>

Ejemplo: DF\_UserDetails\_UserName

* La estructura de la base de datos debe estar normalizada hasta la tercera forma, para un almacenamiento eficiente de los datos en la base de datos.
* Evitar el uso de SELECT \* en las consultas SQL. En su lugar, escribir las columnas requeridas después de la sentencia SELECT.

Ejemplo: SELECT userName, password FROM USER\_DETAILS

* Evite utilizar tablas temporales y tablas derivadas, ya que utiliza más I/O de disco. en su lugar utiliza CTE (Common Table Expression); su alcance está limitado a la siguiente declaración en la consulta SQL
* Formato correcto de consultas SQL utilizando sangrías:

Example: *Correct Format*

**SELECT** userName, password

**FROM** UserDetails ud

**INNER JOIN** Employee e

**ON** e.idEmp = ud.idUser

* Utilice palabras en MAYÚSCULAS para todas las palabras clave SQL.

Ejemplo: **SELECT, UPDATE, INSERT, WHERE, INNER JOIN, AND, OR, LIKE**

* Debe haber una CLAVE PRIMARIA en todas las tablas de la base de datos con el ID de nombre de columna. Es una práctica común utilizar la clave principal como columna de IDENTIDAD.
* Si "Una tabla" hace referencia a "Otra tabla" que no sea el nombre de columna usado en la referencia, debe usar la siguiente regla: Columna de otra tabla: <UnaTabla> ID

Ejemplo: Si la tabla del usuario hace referencia a la tabla del empleado que el nombre de la columna utilizada en la referencia debe ser UserID donde User es el nombre de la tabla y la columna principal de ID de la tabla User y UserID es la columna de referencia de la tabla del empleado.

* Las columnas con restricción de valor predeterminado no deben permitir valores nulos.
* Practique el uso de la CLAVE PRIMARIA en las sentencias WHERE de las instrucciones UPDATE o DELETE, ya que esto evitará posibles errores.
* Siempre cree un procedimiento almacenado en la misma base de datos donde exista su tabla relevante, de lo contrario reducirá el rendimiento de la red.

**INSERT INTO** Table A (column1, column2)

**SELECT** column1, column2

**FROM** Table B

**WHERE** ….

* Evite usar espacios dentro del nombre de los objetos de la base de datos; Esto puede crear problemas con las aplicaciones y herramientas de acceso a datos de front-end.
* No use palabras reservadas para nombrar objetos de la base de datos, ya que esto puede llevar a algunas situaciones impredecibles.
* Practique la redacción de comentarios en procedimientos almacenados, activadores y lotes de SQL, siempre que algo no sea muy obvio, ya que no afectará el rendimiento.
* No utilice caracteres comodín al principio de la palabra mientras realiza la búsqueda con la palabra clave LIKE, ya que da como resultado un escaneo de índice.
* Utilice sangrías en su código para una mejor lectura.
* No utilice la opción RECOMPILE para el procedimiento almacenado, ya que reduce el rendimiento.
* Coloque siempre las declaraciones DECLARE al inicio del código en el procedimiento almacenado. Esto hará que el optimizador de consultas reutilice los planes de consulta.
* Ponga las sentencias SET al principio (después de DECLARE) antes de ejecutar el código en el procedimiento almacenado.

Ejemplo: Use los bloques BEGIN ... END solo cuando haya varias declaraciones dentro de un segmento de código condicional.

* Para expresar un apóstrofe dentro de una cadena, anide comillas simples (dos comillas simples). Ejemplo: SET @sExample = 'Autoridad de SQL'
* Cuando trabaje con condiciones de rama o expresiones complicadas, use paréntesis para aumentar la legibilidad.

Ejemplo:

**IF** ((**SELECT** 1

**FROM** TableName

**WHERE** 1 = 2) **ISNULL**)

* Para marcar una sola línea como comentario use (‐‐) antes de la instrucción. Para marcar la sección del código como comentario use (/\*...\*/).
* Evite el uso de CROSS JOIN si es posible.
* Si no necesita un conjunto de resultados, use una sintaxis que no devuelva un conjunto de resultados.

**IF** **EXIST** (**SELECT**  1

**FROM** UserDetails

**WHERE** UserID = 50)

* Más bien que,

**IF** **EXISTS** (**SELECT** **COUNT** (UserID)

**FROM** UserDetails

**WHERE** UserID = 50)

* Use el plan de ejecución gráfica en el Analizador de consultas o los comandos SHOWPLAN\_TEXT o SHOWPLAN\_ALL para analizar las consultas SQL. Sus consultas deben hacer una "Búsqueda de índice" en lugar de una "Exploración de índice" o una "Exploración de tabla".
* No prefija los nombres de procedimientos almacenados con “SP\_”, ya que **“SP\_”** está reservado para los procedimientos almacenados del sistema.

Ejemplo: SP <Nombre de la aplicación> \_ [<Nombre del grupo> \_] <Acción> <tabla / instancia lógica>

* Incorpore sus combinaciones y cálculos complicados que se requieren con frecuencia en una vista para no tener que repetir esas uniones o cálculos en todas sus consultas. En su lugar, simplemente seleccione de la vista.
* No consulte ni manipule los datos directamente en su aplicación de front-end, en su lugar, cree procedimientos almacenados y deje que sus aplicaciones accedan a procedimientos almacenados.
* Evite utilizar los tipos de datos ntext, text e image en nuevos trabajos de desarrollo. Use nvarchar (max), varchar (max) y varbinary (max) en su lugar.
* No almacene archivos binarios o de imagen (objetos binarios grandes o BLOB) dentro de la base de datos. En su lugar, almacene la ruta al archivo binario o de imagen en la base de datos y úselo como un puntero al archivo real almacenado en un servidor.
* Utilice el tipo de datos CHAR para una columna no anulable, ya que será la columna de longitud fija, el valor NULL también bloqueará los bytes definidos.
* Evite usar sentencias de SQL dinámico. El SQL dinámico tiende a ser más lento que el SQL estático, ya que SQL Server genera un plan de ejecución cada vez que se ejecuta el tiempo de ejecución.
* Minimizar el uso de Nulls. Porque incurren en mayor complejidad en consultas y actualizaciones. Las funciones ISNULL y COALESCE son útiles para tratar con valores NULL
* Use tipos de datos Unicode, como NCHAR, NVARCHAR o NTEXT si es necesario, ya que usan el doble de espacio que los tipos de datos que no son Unicode.
* Siempre use la lista de columnas en las instrucciones INSERT de las consultas SQL. Esto evitará problemas cuando cambie la estructura de la tabla.
* Realice todas las comprobaciones de **integridad referencial** y validaciones de datos utilizando **restricciones** en lugar de **disparadores**, ya que son más rápidos. Limite el uso de disparadores solo para auditorías, tareas personalizadas y validaciones que no se pueden realizar mediante restricciones.
* Siempre acceda a las tablas en el mismo orden en todos los procedimientos almacenados y los disparadores de forma consistente. Esto evitará puntos muertos.
* No llame a funciones repetidamente en procedimientos almacenados, disparadores, funciones y lotes, en vez de eso, llame a la función una vez y almacene el resultado en una variable, para su uso posterior.
* Con la transacción inicial y final, use siempre la variable global **@@ERROR**, inmediatamente después de las instrucciones de manipulación de datos (INSERTAR / ACTUALIZAR / BORRAR), de modo que si hay un error, la transacción puede revertirse.
* El uso excesivo de GOTO puede llevar a un código difícil de leer y entender.
* No utilice números de columna en la cláusula ORDER BY; se reducirá la legibilidad de la consulta SQL.

Ejemplo: Declaración errónea

**SELECT** id, name, password

**FROM** USER\_DETAILS

**ORDER** **BY** 2

Ejemplo: Declaración correcta

**SELECT** id, name, password

**FROM** USER\_DETAILS

**ORDER** **BY** name

* Para evitar viajes de la aplicación hacia SQL Server, deberíamos recuperar varios conjuntos de resultados de un único procedimiento almacenado en lugar de usar parámetros de salida.
* La instrucción RETURN está diseñada para devolver solo el estado de ejecución, pero no los datos. Si necesita devolver datos, use los parámetros de OUTPUT.
* Si el procedimiento almacenado siempre devuelve el conjunto de resultados de una sola fila, entonces considere devolver el conjunto de resultados utilizando los parámetros de OUTPUT en lugar de la instrucción SELECT, ya que ADO controla los parámetros de OUTPUT más rápido que los conjuntos de resultados devueltos por las declaraciones SELECT.
* Los índices efectivos son una de las mejores maneras de mejorar el rendimiento en una aplicación de base de datos.
* El comando BULK INSERT ayuda a importar un archivo de datos en una tabla o vista de base de datos en un formato especificado por el usuario.
* Utilice **columnas dispersas** para reducir los requisitos de espacio para valores nulos.
* Use la instrucción MERGE para implementar múltiples operaciones DML en lugar de escribir instrucciones separadas INSERT, UPDATE, DELETE.
* Cuando se recuperan con frecuencia algunos registros en particular, aplique el Índice filtrado para mejorar el rendimiento de las consultas, una recuperación más rápida y reducir los costos de mantenimiento del índice.
* Usar la sugerencia del optimizador de consultas NOLOCK se considera una buena práctica para mejorar la concurrencia en un sistema ocupado.
* La cláusula EXCEPT o NOT EXIST se puede usar en lugar de LEFT JOIN o NOT IN para mejorar el rendimiento.

Ejemplo:

**SELECT** EmpNo, EmpName

**FROM** EmployeeRecord

**WHERE** salary> 1000 **AND** salary

**NOT IN** (**SELECT** salary

**FROM** EmployeeRecord

**WHERE** salary> 2000);

(Recomendado)

**SELECT** EmpNo, EmpName

**FROM** EmployeeRecord

**WHERE** salary> 1000

**EXCEPT**

**SELECT** EmpNo, EmpName

**FROM** EmployeeRecord

**WHERE** salary> 2000

**ORDER BY** EmpName

Fuente: <https://blog.sqlauthority.com/i/dl/SQLServerGuideLines.pdf>

# ESTÁNDAR PARA EL DISEÑO DE INTERFAZ DE USUARIO

## Principios de usabilidad

* Recuerde que el propósito de cualquier aplicación de software es permitir a un grupo de personas llevar a cabo un conjunto de tareas específicas. De tal forma que lo primero que establecer al diseñar una aplicación es quiénes son sus usuarios y lo que quiere permitirles hacer.
* No limite su base de usuarios, permitir accesibilidad e incluir cierto nivel de internacionalización. La internacionalización significa diseñar software de manera que pueda funcionar en diferentes entornos lingüísticos. La localización es el proceso de traducir realmente mensajes, etiquetas y otros elementos de la interfaz de una aplicación a otro idioma.
* Crear una coincidencia entre su aplicación y el mundo real. Use siempre palabras, frases y conceptos que sean familiares para el usuario en lugar de términos del sistema subyacente. Use términos relacionados con el conocimiento del usuario sobre las tareas que soporta su aplicación.
* Haga que su aplicación sea consistente. La consistencia permite a los usuarios aplicar su conocimiento existente sobre su entorno informático y otras aplicaciones para entender una nueva aplicación. Esto no solo permite que los usuarios se familiaricen con las nuevas aplicaciones más rápidamente, sino que también ayuda a crear una sensación de comodidad y confianza en el entorno general.
* Mantener informado al usuario. Permita siempre al usuario saber qué está pasando en su aplicación, proporcionando contexto adecuado en el momento adecuado. El usuario no debería tener que adivinar el estado del sistema o de su aplicación. Cuando el usuario realiza una acción, proporcione información para indicar que el sistema ha recibido la entrada y está trabajando en ella. La información puede ser visual, sonora o ambas. Si el sistema tarda mucho en procesar la solicitud, proporcione toda la información que sea posible sobre la duración que tendrá la operación. Los tipos de retroalimentación adecuados incluyen (pero no se limitan a): cambios en el cursor, imágenes animadas, indicadores de progreso, información sonora como un pitido y mensajes de error. Los mensajes de error deberían usar un lenguaje simple, aclarando el estado del problema y proporcionando soluciones para indicar al usuario cómo resolver la situación actual, si es posible. Es crítico que la respuesta sea clara y precisa. Si muestra algún indicador de progreso para mostrar el estado de finalización de una tarea y es impreciso, el usuario perderá confianza en los indicadores de progreso y verá el entorno menos útil. Si muestra algún mensaje de error genérico que indica que hay un problema pero falla al proporcionar suficiente información para diagnosticar o resolverlo, los usuarios no podrán continuar con su tarea.
* Mantenerlo simple y bonito. Su aplicación debe permitir al usuario concentrarse en su tarea actual. Por lo tanto, diseñe su aplicación para que solo muestre información y elementos de interfaz útiles y relevantes.
* Dé el control al usuario. Un usuario debería sentir siempre que tiene el control y que es capaz de hacer lo que quiere cuando quiere.
* Perdonar al usuario. Todo el mundo comete errores. Aunque esté explorando y aprendiendo cómo utilizar el sistema, o sea un experto que simplemente pulsa tecla equivocada, errar es humano. Su aplicación debe, por lo tanto, permitir al usuario deshacer rápidamente los resultados de sus acciones.
* Proporcionar manipulación directa. Siempre que sea posible, permita al usuario actuar sobre datos y objetos directamente en vez de mediante diálogos o comandos explícitos. Por ejemplo, es más intuitivo arrastrar un círculo por un diagrama que seleccionar el comando «Mover» de un menú mientras el círculo está marcado.

## Ventanas

* Use una ventana modal sólo si el permitir la interacción con otras partes de la aplicación mientras la ventana está abierta puede hacer que se pierdan datos o que haya cualquier otro problema serio. Proporcione una manera clara de salir de la ventana modal, como un botón Cancel en una alerta.
* Una alerta proporciona información sobre el estado de una aplicación del sistema, o pide información esencial sobre cómo proceder con una tarea en particular. Se distingue de otras ventanas en que el usuario no la solicita directamente, y generalmente contiene un mensaje o una pregunta en lugar de controles editables. Ya que las alertas son una intrusión no bienvenida en el trabajo del usuario, no las use salvo que sea necesario para prevenir pérdidas potenciales de datos u otros problemas graves.
* Ventanas de progreso. Se puede usar una ventana de progreso para proporcionar información durante una operación que dura más de un segundo.
* Diálogos. Un diálogo proporciona un intercambio de información, o diálogo, entre el usuario y la aplicación. Use un cuadro de diálogo para obtener información adicional del usuario que se necesita para llevar a cabo un comando o tarea en particular.
* Asistentes. Un asistente es una ventana secundaria que guía al usuario a través de una operación dividiéndolo en pasos secuenciales. Los asistentes son útiles para hacer operaciones complejas menos intimidantes, ya que restringen la información que el usuario puede ver en un momento dado.

## Menús

* Cuando diseñe una nueva aplicación, coloque los elementos de menú comunes en las mismas ubicaciones en las que aparecen en otras aplicaciones, ya que esto hace que sea mucho más fácil para el usuario aprender.
* Normalmente, la barra de menú está visible todo el tiempo y es accesible desde el teclado, por lo que debe hacer que todos los comandos disponibles en su aplicación están disponibles desde la barra de menú.
* La mayoría de las aplicaciones tienen muchas funciones en común, como Cortar , Copiar , Pegar y Salir . Para facilitar el aprendizaje y la memorización, estos elementos del menú y los menús en los que aparecen deben aparecer con las mismas etiquetas y en el mismo orden en cada aplicación. Los mismos comandos también deben comportarse de la misma manera en diferentes aplicaciones, para evitar sorprender al usuario.

## Barras de herramientas

* El diseño cuidadoso y consistente de la barra de herramientas acelera la tarea del usuario al dar acceso directo a las funciones que de otra manera estarían ocultas en un menú. Sin embargo, úsalos solo para las funciones más importantes. Tener demasiados controles en la barra de herramientas reduce su eficiencia al hacer que sean más difíciles de encontrar, y demasiadas filas de barras de herramientas reducen la cantidad de espacio de pantalla disponible para el resto de la aplicación.
* Divida las cajas de herramientas con más de dieciséis elementos en categorías. El tamaño óptimo para una categoría está entre cuatro y diez elementos.
* La efectividad de las barras de herramientas se incrementa al mantener un nivel de consistencia entre diferentes aplicaciones. La barra de herramientas es una de las primeras partes de su aplicación que un usuario verá la primera vez que la ejecute, por lo que al proporcionar una barra de herramientas que le resulte familiar, puede hacer que se sienta cómodo al usar su aplicación de inmediato.

## Controles

* A veces no tiene sentido permitir que el usuario interactúe con un control en el contexto actual, por ejemplo, al presionar un botón Pegar cuando el portapapeles está vacío. En estos momentos, haga que el control sea insensible para minimizar el riesgo de error del usuario. Si bien un control no es sensible, aparecerá atenuado y no podrá recibir el foco, aunque las tecnologías de asistencia como los lectores de pantalla aún podrán detectarlo e informar.
* Un buen aviso o mensaje de error contiene dos elementos: Una breve descripción del problema y una lista de las formas en las que el usuario puede solucionar el problema. Ambos elementos se deberían presentar en un lenguaje no técnico, libre de jergas, a menos que el público al que está dirigido sea particularmente técnico.

## Recomendaciones

* Proporciona comentarios inmediatos a los usuarios, incluso cuando no puedes cumplir con sus solicitudes de inmediato.
* Permite que los usuarios realicen otros trabajos mientras las operaciones prolongadas se completan, especialmente las operaciones no solicitadas por los usuarios, como reclamar la memoria no utilizada u otras operaciones de "mantenimiento".
* Proporciona suficiente información para que los usuarios entiendan lo que están haciendo y organizan la información de acuerdo con las habilidades de los usuarios para comprenderla y reaccionar ante ella.
* Haz saber a los usuarios cuando el procesamiento está en progreso.
* Permite a los usuarios saber o estimar cuánto tiempo durarán las operaciones prolongadas.
* Permite a los usuarios establecer el ritmo de trabajo, cuando es posible, y les permiten detener las tareas solicitadas que se iniciaron pero no se terminaron.

## Tiempos de respuesta aceptables

* Haga que cada retraso de respuesta en su aplicación sea lo más breve posible, a menos que los usuarios necesitan tiempo para ver la información mostrada antes de que se borre. El retardo de respuesta aceptable para cada evento se basa en la sensación típica de un usuario de que el evento es un punto lógico en el que se debe detener o pausar. Cuanto mayor sea ese sentido, más dispuesto estará el usuario a esperar una respuesta.
* Si su aplicación tarda demasiado en responder, los usuarios se sentirán frustrados. Si no puede mostrar toda la información que un usuario ha solicitado, primero muestre la información más importante, no olvide mostrar comentarios tan pronto como sea posible.
* Ahorre tiempo mostrando resultados aproximados mientras calcula los resultados finales.
* Si es probable que los usuarios repitan un comando que consume mucho tiempo en una sucesión rápida, ahorra tiempo simulando los efectos del comando en lugar de procesarlo repetidamente. Por ejemplo, si un usuario agrega varias filas a una tabla almacenada en una base de datos, puede mostrar cada fila nueva inmediatamente, pero demorar la creación de cada fila nueva en la base de datos hasta que el usuario termine de agregar todas las filas.
* Trabajar por delante. Prepárese para ejecutar el comando que es más probable que siga el comando actual. Es decir, use el tiempo de inactividad para anticipar las próximas solicitudes probables de los usuarios. Por ejemplo, cuando el usuario de una aplicación de correo electrónico lee el mensaje nuevo que se muestra en ese momento, la aplicación podría prepararse para mostrar el siguiente mensaje nuevo.
* Utilice el procesamiento de fondo. Realice tareas menos importantes, como el servicio de limpieza, en segundo plano, permitiendo a los usuarios continuar trabajando.
* Retrasar el trabajo que no sea urgente. Hazlo más tarde, cuando haya más tiempo disponible.
* Deseche las operaciones innecesarias. Por ejemplo, para retroceder varias páginas en un navegador web, un usuario puede hacer clic varias veces en el botón Atrás del navegador en una rápida sucesión. Para mostrar la página final solicitada más rápidamente, es posible que el navegador no muestre las páginas visitadas entre la página actual y la página final.
* Utilice la gestión dinámica del tiempo. En el tiempo de ejecución, cambie la forma en que su aplicación prioriza la entrada del usuario y otro procesamiento, según el estado actual de la aplicación. Por ejemplo, si un usuario escribe texto en un documento de procesamiento de texto mientras imprime otro, la aplicación de procesamiento de texto puede retrasar la tarea de impresión si el usuario pasa a una tarea de edición (como cortar y pegar texto) que requiere mayores recursos.
* La retroalimentación del puntero cambia la forma del puntero. Por ejemplo, un puntero de ocupado indica que una operación está en curso y que el usuario no puede realizar otras tareas. Un puntero de ocupado indica que una operación está en curso pero la ventana aún es interactiva.
* Las animaciones de progreso muestran cuánto de una operación se ha completado, o solo que una operación está en curso. Normalmente, estos toman la forma de una barra de progreso o una lista de verificación de progreso.

## Diseño visual

* El diseño visual no se trata solo de hacer que tu aplicación se vea bonita. El buen diseño visual tiene que ver con la comunicación. Una aplicación bien diseñada facilitará al usuario comprender la información que se presenta y les mostrará claramente cómo pueden interactuar con esa información. Si puedes lograr todo eso, tu aplicación se verá bien para el usuario, ¡incluso si no tiene gráficos o logotipos de fantasía!
* El color es una buena herramienta para comunicar información en una interfaz de usuario. Sin embargo, el color siempre debe considerarse como una adición útil a su diseño, no como una necesidad. Nunca dependa solo de los colores para mostrar información importante, y tenga en cuenta que si los colores no se pueden percibir correctamente (por ejemplo, el usuario tiene un sistema de 8 bits o es ciego al color), su aplicación aún debería poder utilizarse.

## Iconos

* Los íconos son una metáfora gráfica que presenta una imagen visual que el usuario asocia con un objeto, estado u operación en particular. Cuando un usuario ve un buen icono, se le recuerda inmediatamente el elemento que representa, ya sea una aplicación en el menú del panel o el estado "alineado a la derecha" en una barra de herramientas del procesador de textos.
* Los iconos de alto contraste son versiones enormemente simplificadas de los iconos regulares existentes de una aplicación. Se dibujan con dos colores, blanco y negro, y bordes más gruesos. Este estilo permite distinguir los iconos de alto contraste cuando los ve un usuario con una discapacidad visual.

Fuente: <https://developer.gnome.org/hig-book/3.4/index.html.es>

# ESTÁNDAR PARA EL DISEÑO Y DESARROLLO DE FRONTEND

* Optimizar el peso de las imágenes: se debe bajar al máximo posible el peso de las imágenes; cuando esto no sea posible hacerlo por su tamaño, se debe reducir el número de colores disponibles y la resolución.
* Probar el sitio con las versiones para diferentes sistemas operativos de diversos visualizadores de páginas (browsers); especialmente hacerlo con versiones de Microsoft Internet Explorer, Netscape Communicator, Mozilla, Opera y Safari.
* Automatizar lo más que se pueda los procesos de desarrollo.
* Comenta el HTML, esto es importante cuando se trabaja en equipo o cuando hay que mantener el mismo proyecto durante un tiempo para entender que es cada elemento. Se vuelve más importante cuando tenés muchos <div> que existen sólo por razones de diseño.
* Minificar el HTML, esto es muy importante ya que la minificación ayuda a que el HTML pese menos (permitiendo que el usuario lo descargue más rápido) y además elimina los comentarios que hayamos dejado. Por último debido a que la minificación deja todo el código en una línea ayuda a evitar el espacio en blanco entre elemento al usar display: inline-block.
* Carga el CSS lo antes posible. Mientras más rápido el navegador descargue y lea el CSS más rápido va a terminar de renderizar la página. Hay que tratar de cargar los archivos CSS lo antes posible, lo ideal es que sea justo después de la etiqueta <title>.
* Inserta en tu HTML el CSS crítico para tu aplicación. La carga de los archivos de CSS es una operación bloqueante, esto quiere decir que el navegador va a dejar de leer el HTML hasta que termine de descargar el archivo CSS. Esto trae el problema de que si el archivo es muy pesado el navegador va a estar mucho tiempo sin hacer nada mientras espera que descargue. Para evitar este problema se puede detectar cuál es el CSS crítico para tu aplicación y agregar ese CSS en una etiqueta <style> directo en el HTML y el resto de estilos cargarlos de forma diferida usando JS al final del documento. El CSS crítico básicamente es el CSS que se utiliza “above the fold”, osea que está en la parte visible del viewport al momento de entrar al sitio y sin haber hecho scroll down. Para poder detectar cual parte de tu CSS es crítico hay herramientas como Penthouse que tiene un plugin para Grunt o se puede implementar con Gulp fácilmente.
* Carga el JS al final. La carga de archivos JS es una operación bloqueante por lo que detiene el renderizado de la página, si además de esto el JS se ejecuta inmediatamente al ser descargado puede dar errores si se trata de manipular el DOM antes de que se renderice el HTML. Para evitar estos problemas es mejor siempre colocar las etiquetas <script> al final del HTML, justo antes de la etiqueta </body>. Esto permite que el usuario pueda interactuar con el resto del sitio mientras el JS se descarga y ejecuta.
* Cargá cualquier JS no indispensable de forma asíncrona. La etiqueta <script> tiene un atributo llamado async que permite realizar la carga de una archivo de forma asíncrona. Para archivos como los de Google Analytics es mejor siempre cargarlos de esta forma para evitar que esta carga se convierta en una operación bloqueante. Esto solo hay que hacerlo en archivos como Google Analytics que no son indispensables para el uso de nuestra aplicación web. Para los archivos de la aplicación web es mejor no usar async.
* Usá la menor cantidad de etiquetas posibles. Cada etiqueta HTML que coloquemos ralentiza el renderizado de nuestra aplicación web. Siempre hay que tratar de usar la menor cantidad posible y evitar la “divtitis”, osea poner un <div> dentro de otro, dentro de otro y así. Cosa muy común si se usa Bootstrap o similares.
* Comprime todas tus imágenes. Algo que muchos no hacen es comprimir lo más posible las imágenes de un sitio, esto lleva a que haya imágenes muy grandes y pesadas, sin que sea necesario tanto tamaño y calidad. Lo mejor es cargar imágenes del tamaño necesario exacto (el doble para retina display) y lo más comprimidas posibles. La compresión puede hacerse de dos formas, sin pérdida y con pérdida, en el caso de con pérdida se le baja la calidad hasta lo mínimo aceptable para que se siga viendo bien. En el caso de la sin pérdida se eliminan los meta datos que tienen las imágenes (vídeos y audios también) y se reduce un poco de espacio gracias a eso. Esto se puede hacer fácilmente con una herramienta llamada imagemin que tiene plugins para Gulp y Grunt.
* Evitá los @import en el CSS. Cuando el navegador está leyendo un archivo CSS y se encuentra con un @import deja de leer el archivo CSS original y empieza a descargar y leer el archivo importado y recién cuando termina sigue con el original. Debido a que la carga de CSS es una operación bloqueante el estar haciendo esto ralentiza el renderizado de nuestra aplicación por lo que es mejor evitar el uso de @import en los archivos CSS, en lugar de eso es mejor combinar los archivos o al menos usar dos etiquetas <link>.
* Minificar el CSS. Esto a pesar de ser una gran mejora de rendimiento y velocidad es algo que todavía no se realiza en muchos sitios o aplicaciones web. El minificar el CSS, al igual que con el HTML, reduce el peso del archivo y elimina los comentarios, permitiendo una carga más rápido del archivo y terminar el renderizado antes.
* Agrupa los media query. Aunque durante el desarrollo es mejor dejar los media query al final de cada componente de nuestro CSS, para saber fácilmente qué elementos se están modificando, al momento de generar el CSS de producción es mejor agrupar todos los media query al final del archivo. Esto además reduce la cantidad de media queries ya que es muy común usar los mismos break points en múltiples lugares, lo cual se combinaría en una única media query. Esto lo pueden hacer con group-css-media-queries y sus plugins de Gulp y Grunt.
* Elimina el CSS innecesario. Al usar Bootstrap o similares es muy común no usar todos los estilos que este trae. También es bastante común que durante el desarrollo queden clases que ya no se utilizan y se dejan “por las dudas” y que el usuario está descargando al bajar el CSS aunque no los necesite. Al momento de generar el CSS de producción lo mejor es borrar estos estilos y dejar solo los que sí son necesarios para nuestra aplicación web. Esto se puede realizar con herramientas como UnCSS y sus plugins de Gulp y Grunt.
* Usa preprocesadores. Una vez dominamos CSS el incorporar pre-procesadores a nuestro proceso de desarrollo nos da varias ventajas como puede ser la posibilidad de separar nuestros estilos y luego importarlos para generar un único archivo final, usar variables para guardar colores, tamaños de fuente, etc. o anidar nuestros estilos para evitar repetir parte del nombre de una clase.
* Separa tus hojas de estilos en varios archivos por cada 4095 selectores. Hay algo que muchos desarrolladores FrontEnd no saben y es que IE hasta la versión 9 tiene un límite de 4095 en la cantidad de selectores permitidos por cada archivo CSS. Una vez que se alcanza ese límite IE simplemente ignora el resto de estilos dejando parte de tu sitio sin estilos. Para evitar este problema dividí tu CSS en múltiples archivo por cada 4095 selectores de forma que IE pueda leer correctamente todos los estilos de tu aplicación. Se puede realizar esto usando una herramienta llamada Bless y sus plugins de Gulp y Grunt.
* Arma una guía de estilos al escribir código JS. No todos los desarrolladores escribimos código de la misma forma y cuando trabajamos en equipo es importante que todos estemos de acuerdo en cómo nombrar variables, funciones, donde dejar espacios, como indentar, etc. Para lograr eso lo mejor es usar una guía de estilos que nos diga cómo escribir nuestro código. Podemos crear una propia para nuestro proyecto con nuestros gustos o podemos usar algunas ya hechas que hay en internet como la guía de AirBnB. Si combinamos esto con herramientas como ESLint, JSLint o JSHint podemos definir nuestra guía en nuestro linter para que nos marque como errores cuando no sigamos nuestra guía. La guía de AirBnB tiene versiones para usar con estas tres herramientas.
* Modularizar el código JS. Separar el código en archivos nos permite hacer que nuestro código sea más fácil de reutilizar y hacer unit-testing para comprobar que funcione correctamente. Además de esto modularizar hace más fácil mantener el código a largo plazo, otra ventaja es que es más fácil escalar una aplicación modular. Para poder modularizar nuestro código hay varias formas, la mejor es usar los módulos de ECMAScript 2015, sistema oficial de JS, o al menos el de CommonJS. El sistema AMD (Asynchronus Module Definition) no tiene sentido ya que al final hasta que no carguen las dependencias de tu código no vas a ejecutarlo, igual que harías con CommonJS o ES2015.
* Comenta el código JS. Similar al primer punto, es importante comentar nuestro código para que cualquier desarrollador que trabaje con nosotros sepa que hace. Alcanza con dejar un pequeño comentario antes de crear cada función explicando que hace y qué parámetros recibe, aunque para bloques de código complicados es buena idea agregar un comentario explicando qué hace.
* Documenta tus funciones y APIs. Esto viene de la mano con documentar el código, es importante dejar documentado en algún sitio que hacen nuestras funciones o si tenemos alguna API que hace y cómo funciona. Es decir, que valores recibe, que hace con esos valores y que valores devuelve.
* Evita ensuciar el scope global. Cuando no se trabaja con módulos es muy común dejar un montón de variables en el scope global de nuestra aplicación (global en Node.js y window en navegadores) lo que puede causar problemas al intentar usar una librería de terceros o al agregar más código. Lo mejor es tratar de encapsular el código en NameSpaces, esto lo pueden lograr usando una función autoejecutable (IIFE) que encapsule el código y lo ejecute. Esto se soluciona si se usan sistemas de módulos.
* Evitá estilizar el DOM con JS. Si necesita modificar los estilos de un elemento del DOM de forma dinámica lo mejor es agregar o quitar clases con JS y tratar de evitar modificarlo directamente con JS usando .style. Esto mismo aplica para las animaciones de elementos del DOM, es mucho mejor realizarlas con CSS y no con JS. Esto se debe a que las animaciones hechas por CSS usan la GPU para ser procesadas, mientras que las hechas por JS usan la CPU, esto hace que las hechas por CSS sean mucho fluidas tanto en desktop como en mobile.
* Reduce el acceso al DOM. La parte más lenta de cualquier aplicación web es el DOM, cada acceso al DOM nos cuesta mucho tiempo por lo que es mejor evitar todo lo posible acceder al DOM. Si necesitan editar múltiples veces el mismo elemento es mejor acceder una vez y guardar el elemento en una constante que luego puedan usar para manipular el elemento sin volver a acceder al mismo. Si usan React.js o alguna otra librería que implemente el concepto de Virtual DOM ya no van a tener este problema pues no accederán directamente al DOM.
* Reduce los Event Listeners. Tener muchos event listeners al mismo tiempo ralentiza nuestras aplicaciones. Si necesitan escuchar múltiples elementos (por ejemplo cada elemento de un listado) es mejor escuchar al elemento padre y usar event.target para verificar si el evento es disparado desde el elemento deseado.
* Reduce las peticiones al servidor. Cada petición ralentiza un poco más la carga de la aplicación, es mejor tratar de reducir la cantidad de peticiones al mínimo para que nuestra aplicación cargue más rápido.
* Usá compresión GZIP. La compresión GZIP ayuda a reducir el peso de los archivos que nuestro servidor HTTP envía al navegador haciendo que nuestra aplicación cargue más rápido. Siempre deberíamos servir todos nuestros archivos estáticos usando compresión GZIP. Implementarla depende del servidor, pero normalmente es cambiar una pequeña parte de la configuración.
* Cargá de forma diferida imágenes, vídeos, etc. Es mejor solo cargar las imágenes, vídeos, fuentes, audios, etc. que el usuario necesita inmediatamente ya que están dentro del viewport inicial (están above the fold) y cargar de forma diferida (lazy load) el resto de forma que solo se descarguen si el usuario lo necesita o al menos hasta que el contenido sí necesario se termine de cargar.
* Evitá los src vacíos. Algunos navegadores si encuentran una etiqueta <img> o <source> con el src vacío o sin un atributo src van a hacer una petición a la misma página, lo que implica una petición extra e inútil. Si vas a cargar de forma diferida imágenes y vídeos es mejor dejar una imagen por defecto en el atributo src que se pueda descargar una vez y sirva para todas las imágenes y luego cuando el usuario haga scroll cambiar el src por la imagen correspondiente.
* Precarga el contenido. Trata de adelantarte a lo que el usuario va a hacer y carga el contenido que pueda necesitar antes de que lo necesite. Esto se puede lograr empezando a cargar el contenido cuando el usuario haga hover en algún link o botón permitiéndonos cargar todo el contenido que el usuario vaya a necesitar para luego mostrárselo inmediatamente cuando haga click. Google hace esto mismo para precargar el contenido del primer resultado de búsqueda, así este carga más rápido al entrar.
* Evitá los 404. Asegúrate de evitar cualquier 404 en todas las peticiones. Cada petición toma tiempo y si va a terminar dando un error 404 es mejor evitarla, verifica siempre que tus scripts, hojas de estilos, imágenes, etc. se cargan correctamente.
* Renderiza la primera carga en el servidor. Con el aumento de popularidad de las Single-Page Applications (SPA) se empezó a renderizar todo en el cliente. Aunque esto es bueno para los cambios de una vista a otra de nuestra aplicación, para la carga inicial no lo es ya que esta primer carga ahora necesita que terminemos de iniciar nuestro JS para poder recién renderizar el contenido. Siempre deberías enviar el HTML con todo el contenido en la primera carga, de esta forma mientras se inicializa la SPA el usuario ya puede ver el contenido y no una pantalla en blanco. Una vez iniciada la SPA el cambio de vistas sí debería ser realizado mediante JS, incluyendo el obtener el contenido vía una API.
* No le quites funcionalidad al usuario. Si el usuario tiene una funcionalidad por defecto no deberías quitarle esa funcionalidad. Un ejemplo es que si el usuario puede hacer zoom en móviles no deberíamos quitarle esa posibilidad. Otro ejemplo muy común es en los formularios, el usuario por defecto puede enviarlos apretando Enter/Return mientras hace foco en un <input> o enviarlo al hacer click en un botón de Submit. Al usar AJAX es común que el envío de los datos por AJAX se agregue al evento click del botón, en lugar de eso es mejor agregarlo al evento submit del formulario lo cual obtendría tanto el click del botón como el Enter/Return de los inputs.
* Recordá las acciones del usuario y el estado de la aplicación. Si el usuario empieza hacer algo, se va a otra página y vuelve para atrás deberías recordar que hizo y en donde se quedó. Un ejemplo muy común es que si el usuario empezó a llenar un formulario de varios pasos y en algún punto quiere volver al paso anterior deberías recordar lo que el usuario ingresó.

Fuentes: <https://medium.com/@sergiodxa/buenas-pr%C3%A1cticas-del-desarrollo-frontend-40f44621841>

<https://youtu.be/QVFoI9HNJ34>

# ESTÁNDAR PARA EL DISEÑO Y DESARROLLO DE BACKEND

* Se debe documentar debidamente el código que se produce.

* Testear es obligatorio, incluso después de un pequeño cambio en el código, incluso si vas mal de tiempo, incluso si solo has cambiado el nombre de una variable, incluso si solo has insertado un comentario. Además, encontrar un error en tu código debe hacerte feliz, porque eso significa que tu software tiene un bug menos.

* Evitar el uso de variables globales. Una variable global es una variable colocada fuera del ámbito de varias funciones y que es globalmente accesible por todas ellas. No se debe confundir con un atributo de una clase, que es un campo dentro de un registro que varias funciones manipulan. Un consejo útil para distinguir atributos de variables globales es pensar si la variable corresponde a una característica identificable del conjunto de individuos representado por una clase. El uso de variables globales pueda resultar tentador por lo cómodo de evitar tener que estar pasando la variable global como parámetro. Sin embargo, el uso de variables globales puede tener efectos laterales bastante peligrosos (cf [4]). El problema es que al ser la variable globalmente accesible, ésta puede, por malicia o simple infortunio, ser modificada fácilmente por terceras funciones, asignando a la variable un valor no deseado. Este valor no deseado puede desencadenar una serie de fallos en cadena. Al ser la variable accesible por un número potencialmente largo de funciones, una vez producido el fallo, depurar el programa hasta averiguar cuál es la causa del fallo puede ser un proceso largo y tedioso. En muchas Escuelas de Ingeniería Informática el uso no justificado de variables globales es causa directa de suspenso en un examen o práctica de programación.

NOTA: Linus Torvalds realiza la misma recomendación para la creación de código para el núcleo de Linux: sólo usar variables globales cuando es realmente necesario.

NOTA: El uso de variables globales no será causa directa de suspenso en las prácticas y pruebas correspondientes a las asignaturas que el autor de esta página imparta en la Universidad de Cantabria o centros similares, pero si rebajará sensiblemente la calificación de las mismas. Un abuso en el uso de variables globales si será causa automática de suspenso.

* Evitar el uso de sentencias goto, break y continue. Evitar el uso de sentencias que rompan el flujo secuencial de ejecución de un programa es el principio básico de la programación estructurada. Las sentencias tipo goto (cf. [1]) dificultan la legibilidad, depuración y verificación de programas, al permitir a la ejecución de un programa realizar saltos arbitrarios. Aquellas funciones que hacen uso de sentencias goto son mucho más difíciles de reutilizar y depurar.

El uso de sentencias break y continue suele estar asociado a técnicas de construcción de bucles artesanales, donde primero se codifica un bucle y luego se comprueba su corrección mediante prueba y error usando depuración. El no usar estas sentencias obliga normalmente a pensar primero qué condición o invariante debe cumplirse durante la ejecución del bucle, y a continuación codificar el bucle. Esta segunda técnica resulta mucho más ingenieril, es la base para la posterior verificación formal del bucle y suele contribuir a mejorar la legibilidad de los programas.

Por ejemplo. en un bucle tipo while libre de sentencias break, sabemos que el bucle terminará cuando la condición deje de cumplirse. Por tanto, para saber si un bucle terminará, bastará comprobar que en el cuerpo del bucle se realiza algún cómputo que converge hacia tal condición. Además, en ausencia de sentencias break, a la salida del bucle sabemos que la negación de la condición del bucle siempre será un predicado verdadero, lo que constituye la base para seguir diseñando el programa.

Comentar finalmente, que las sentencias break y continue son versiones degeneradas de la sentencia goto. La misión principal de estas instrucciones es forzar la salida de un bucle o su continuación. Si el programa fuese modificado y las sentencias break o continue quedan anidadas dentro de nuevos bucles, éstas podrían dejar de cumplir su función. Al modificar el programa las sentencias break y continue podrían salir de, o continuar con, un bucle que no es el deseado, respectivamente. En ambos casos, la aplicación presentaría un comportamiento no deseado.

* Usar un único return por función, que se colocará como última sentencia de la función. Este consejo una consecuencia de uno de los principios de la programación estructurada. Dicho principio establece que los programas deberían tener un único punto de entrada y un sólo punto de salida. Además, enlazado con el consejo anterior, un return puede considerarse como un GOTO hacia el final de la función. El uso de un único return por función, colocado consecuentemente al final de la misma, facilita tanto la depuración como la adaptabilidad de los programas.

Por ejemplo, si estamos depurando un programa y queremos saber qué valor devuelve una función, en caso de existir un único return, bastará con o imprimir el valor devuelto antes de invocar dicho return; o poner un punto de ruptura en tal sentencia. Si por el contrario tuviésemos, pongamos por ejemplo, 5 sentencias return en la función, tendríamos que colocar y gestionar 5 puntos de ruptura durante la depuración del programa.

Como ejemplo de mejora de la adaptabilidad, supongamos ahora que tenemos una función que devuelve una longitud en centímetros. Supongamos también que tenemos que adaptar nuestro software para que trabaje en el sistema anglosajón, devolviendo ahora en lugar de una cantidad en centímetros, la misma cantidad pero expresada pulgadas. La adaptación requerida sería tan fácil como invocar una función cmApulgadas(..) sobre el valor devuelto por la función. Si hay un sólo return, sólo tendremos que modificar una línea de código. Si tenemos, pongamos por ejemplo, 5 sentencias return, tendremos que modificar 5 líneas de código, lo cual es más propenso a errores.

* Evitar escribir funciones y procedimientos demasiado largas. Según estudios psicológicos, la mente humana no es capaz de procesar más de 6 ó 7 detalles diferentes a la vez (y a mí, particularmente, me parece éste un número bastante generoso). Las funciones demasiado largas suelen contener un número de detalles superior a este límite. Ello dificulta su legibilidad y comprensión y por tanto, su mantenimiento.

En general, en funciones demasiado largas hay trozos claramente diferenciados de código, débilmente acoplados. Cada uno de estos bloques suele realizar una tarea distinta. Por ejemplo, es habitual en muchas funciones que al principio se preparen los datos para realizar un cómputo, o se inicialice alguna estructura. A continuación se realizan una serie de cálculos y por último se presenten por la salida los resultados.

En ese caso, por ejemplo, se recomienda dividir la función en tres subfunciones inicializa(..), calcular(..) e imprimir(..). No importa que las expectativas iniciales de reutilización de estas funciones sean prácticamente nulas. En general, si dos trozos de código pueden aparecer juntos en una sola función o separados en dos subfunciones, la opción recomendada siempre es separarlos, salvo justificación irrefutable en contra.

Según Linus Torvalds, creador del núcleo de Linux, "las funciones deberían ser cortas y dulces, y servir para un único propósito. Deberían ocupar una o dos pantallas estándares de texto, es decir, una pantalla de 24 líneas por 80 caracteres de ancho." (Ver aquí)

* Evitar copiar y pegar trozos cuasi idénticos de código a lo largo de una aplicación software.

Si un trozo de código, por ejemplo, una serie de sentencias destinadas a imprimir una matriz por pantalla, se copia y pega cada vez que se necesite dicha funcionalidad, el resultado será una aplicación con bloques idénticos de código dispersos a través de múltiples lugares de una aplicación software. El principal problema viene cuando hay que modificar este bloque de código. Por ejemplo, si simplemente queremos añadir un recuadro a la matriz, deberemos modificar de la misma forma múltiples puntos de nuestra aplicación, lo que es propenso a errores y puede provocar problemas de consistencia. En lugar de copiar y pegar trozos de código a lo largo de una aplicación software, la práctica recomendada es encapsular dicho trozo de código en una función e invocar cuando se necesite.

NOTA: Muchos entornos de desarrollo trae funcionalidades para encapsular trozos de códigos en funciones tan automáticamente como sea posible. Suele aparecer en utilidades de refactorización bajo el nombre de "ExtractMethod".

NOTA: El abuso de la técnica de "cortar y pegar" será causa directa de un detrimento notorio en la calificación de las prácticas y pruebas correspondientes a las asignaturas que el autor de esta página imparta en la Universidad de Cantabria o centros similares.

* Colocar cada clase o módulo en un fichero separado.

Usar un fichero diferente para cada módulo o clase permite su reutilización a nivel individual entre aplicaciones. La única excepción a esta normal es que una clase o módulo esté muy fuertemente acoplada a otra, siendo por tanto imposible que funcione de forma separada a esta última. Sólo en este caso particular estaría justificado colocar ambos módulos o clases en un mismo fichero.

NOTA: Las colocación de todos los módulos o clases de una aplicación software en un único fichero supondrá automáticamente un cero, con total independencia de la calidad de su contenido, en la calificación de las prácticas y pruebas correspondientes a las asignaturas que el autor de esta página imparta en la Universidad de Cantabria o centros similares.

* Colocar la función main en una clase o módulo separado e independiente.

La función main suele representar múltiples lenguajes de programación (e.g., C, C++, C# o Java) el punto de entrada a la aplicación. Su único objetivo es iniciar la aplicación. Por tanto su funcionalidad no pertenece a ninguno de los módulos o clases en los que se descompone una aplicación. Por tanto, la práctica recomendada es crear un módulo o clase Runner o Launcher que contenga únicamente la función main y las funciones auxiliares que ésta pudiese necesitar. Eliminando la función main de los módulos o clases en los que se descompone una aplicación incrementamos la reutilización de estos elementos a nivel individual.

* Evitar el uso de elementos no habituales de un lenguaje.

Muchos lenguajes, como C, contienen elementos que constituyen excepciones a la regla general. Normalmente, estos elementos son solamente conocidos por aquellos programadores que han leído el manual de referencia del lenguaje en profundidad, y que en contra de la opinión del autor de esta página, se conocen como programadores avanzados.

Por ejemplo, en C los enteros pueden usarse como valores booleanos en las condiciones de las estructuras de control. En C, el cero se considera como falso y cualquier valor distinto de cero como verdadero. Por tanto, si queremos escribir una función que divida dos números en caso de que el denominador no sea cero, podríamos escribir el código que aparece a la izquierda de la Tabla 1. No obstante, esta forma de escribir código es críptica y por tanto desaconsejable. Las únicas justificaciones para escribir código de esta forma son por vanidad o vagancia. En el último caso, el programador busca ahorrar escribir unos cuantos caracteres. En el primer caso, el programador "avanzado" busca incrementar su ego d, mediante su reconocimiento como un conocedor de los secretos más ocultos del lenguaje de programación. En ambos casos, se trata de aptitudes en absoluto recomendables.

El código debe ser escrito para que sea entendido por cuantos más programadores mejor. Por tanto, debe escribirse de una forma sencilla y que pueda ser bien entendida.

* Usar siempre llaves ({}) en las estructuras de control.

Muchos lenguajes, inspirados en C, permiten la posibilidad de suprimir las llaves que delimitan un bloque de código ({}) cuando dicho bloque contiene una sola sentencia. Esto suponía una ventaja en los 70, los 80 e incluso los 90. Las ventajas son: (1) reducir el tamaño del código fuente, cuando cada kilobyte era importante; (2) reducir el espacio que ocupaba un función en pantalla (Ver aquí).

Hoy en día una humilde memoria USB es capaz de almacenar varios gigabytes, siendo el coste de un kilobyte despreciable. De igual modo, los monitores ya han superado con creces el límite de las 25 líneas, por lo que intentar comprimir el espacio que un bloque de código ocupa en pantalla ha dejado de ser relevante. Lenguajes modernos como Java mantienen esta característica, básicamente por compatibilidad con C y C++. Se concluye por tanto que el evitar el uso de llaves como no tiene ningún beneficio práctico hoy en día y debe ser atribuido a la simple pereza del programador. La pereza, aparte de ser un pecado capital, es una cualidad en absoluto admirable.

Por otro lado, el omitir llaves puede generar varios problemas (Ver aquí). En primer lugar, podemos encontrarnos con el caso en el que tengamos que extender dicho bloque de código con más sentencias. Por ejemplo, podríamos necesitar añadir una sentencia para imprimir algún valor con objeto de depurar un programa. Si hemos evitado el uso de llaves, puede que no caigamos en la cuenta de que el bloque de código no está delimitado por llaves (error muy común cuando se llevan varias horas programando). La nueva sentencia quedará, en contra de nuestras intenciones, fuera de la estructura de control.

En segundo lugar, algunos lenguajes admiten sentencias vacías. Puede darse el caso de que, por error, si no usamos llaves escribamos un ; tras la estructura de control, que ejecutará por tanto una sentencia vacía (e.g. for(int i=1;i<MAX\_VECTOR;i++);). Encontrar y solucionar estos errores suele ser un proceso bastante largo y tedioso. Por tanto, se aconseja usar siempre llaves para delimitar los bloques de código pertenecientes a una estructura de control, incluso cuando no sean necesarias.

* Colocar al lado de una llave que cierre un bloque de código un indicativo de que tipo de estructura cierra.

Es común cuando se anidan diversos bloques de código encontrarse con líneas de código que simplemente sean una secuencia de cierre de llaves tipo }}}}. En estos casos, es muy fácil perder la pista de qué tipo de bloque de código o estructura cierra cada llave. Por tanto, se aconseja poner al lado de cada llave algún comentario que indique que tipo de estructura o bloque de código se cierra.

La práctica habitual es usar: (1) el nombre del tipo de estructura de control que cierra; (2) el nombre de la función, en el caso del cierre defunciones; (3) el nombre de la clase, registro o módulo, en sus respectivos casos. Esto ayuda a asociar cada llave con la estructura que cierra.

* Inicializar siempre las variables cuando se declaran.

Existen lenguajes como Java que inicializan las variables a un valor constante por defecto cuando éstas son declaradas. Por ejemplo, las variables de tipo entero son inicializadas a cero. Otros lenguajes, como C ó C++, no inicializan las variables por defecto ningún valor. Por tanto, en estos lenguajes si no se asigna ningún valor inicial a las variables, éstas contendrán un valor indeterminado, presentando la aplicación software probablemente un comportamiento no determinista.

Saber qué lenguajes inicializan automáticamente las variables y cuáles no, y en caso de que las inicializan, a qué valor lo hacen, puede ser una tarea bastante compleja, dado el gran número de lenguajes de programación existentes. Por tanto, la práctica recomendada es, por seguridad, inicializar siempre explícitamente las variables con independencia de que el lenguaje lo haga o no. Algunos programadores podrían esgrimir el argumento de que esta solución es ineficiente porque las inicializaciones son redundantes. Sin embargo si las inicializaciones son redundantes y el compilador está bien diseñado, éste debería simplemente obviarlas. Además, un lenguaje que inicializa siempre las variables a un valor por defecto no es un lenguaje pensado para ser eficiente. Consecuentemente, no deberíamos preocuparnos por la eficiencia, más allá de lo razonable, cuando usamos este tipo de lenguajes.

* No declarar atributos de clases y registros como públicos, ni acceder a ellos directamente.

Este consejo se basa en el principio de ocultación de la información. Según este principio (cf. [5]), una aplicación software debe descomponerse en un conjunto de módulos software que se comunican entre sí a través de interfaces bien definidas. Estas interfaces deben exponer la funcionalidad ofrecida por cada módulo a la vez que esconden los detalles de su implementación. Siguiendo estos principios, el acceso a atributos de una clase o registro debe hacerse a través de una serie de métodos o funciones, denominadas getters y setters. Las responsabilidades de estos métodos son garantizar que estos atributos contengan valores correctos, de acuerdo con la semántica de cada campo y del registro o clase como unidad. El propósito principal es impedir que se puedan escribir en los atributos valores carentes de sentido (e.g., edades con valores negativos, o fechas tales como 30/02/2010).

Además, obligando a cambiar el valor de un atributo de una clase a través de un método setter podemos fácilmente hacer, por ejemplo, que una serie de objetos sean notificados cada vez que el valor de este atributo cambié (Ver aquí).

* No ser excesivamente estrictos con la ocultación de información.

A pesar de ser la ocultación de información algo positivo, su exceso puede tener efectos negativos. Uno de estos efectos negativos es que los módulos o clases no puedan ser adaptados por medio de alguna relación de especialización, tipo herencia. Por tanto, se aconseja no ser excesivamente protectores con la visibilidad de funciones y atributos de una clase, permitiendo que éstos puedan ser modificados por las subclases que hereden de dicha clase. Esto debería incrementar la adaptabilidad y facilidad de mantenimiento.

* No usar caracteres propios del castellano en la codificación de programas.

Las vocales acentuadas y la ñ, al no ser caracteres básicos ASCII, pueden sufrir alteraciones cuando un fichero de código se abre en dos computadoras diferentes, incluso si ambas contienen el mismo sistema operativo. Para evitar estos problemas se recomienda codificar directamente en inglés, lo que además facilita la legibilidad del código por parte de la comunidad no hispano hablante. En caso de no dominar el inglés, o simplemente preferir no usarlo, se recomienda renunciar a las vocales acentuadas y sustituir la ñ por su forma catalana (ny) o por su forma gallega (nh).

* Controlar desbordamientos en vectores cuando se accede a ellos.

Para evitar acceder a posiciones situadas fuera de los límites de un vector (array) cuando se accede a una posición del mismo, se recomienda dedicar al menos unos segundos a analizar si la expresión numérica usada para acceder a una posición determinada del vector podría, en algún caso, tomar algún valor que se situase fuera de sus límites. Si así fuese, según el lenguaje de programación que estemos usando, podríamos obtener: (1) una excepción en tiempo de ejecución (e.g Java); o (2) obtener un comportamiento del programa no determinista (e.g. C), puesto que en este segundo caso no podemos predecir que hay más allá de los límites de un vector. Por tanto no podríamos saber que valor estamos escribiendo o leyendo.

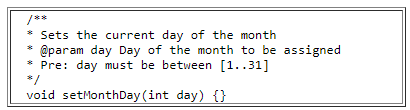
* Controlar que no se diferencian punteros nulos.

Se recomienda antes de desreferenciar un puntero, dedicar al menos unos segundos a analizar si éste pudiese ser nulo, con objeto de evitar acceder a posiciones de memoria indeseadas. Si se referencia un puntero nulo en tiempo de ejecución se obtiene, dependiendo del lenguaje de programación utilizado: (1) una excepción en tiempo de ejecución (e.g Java); o (2) un comportamiento no determinista de la aplicación software (e.g. C). Este segundo caso suele dar como resultado un acceso a una zona de memoria no permitida, con la consiguiente interrupción inmediata de aplicación software por parte del sistema operativo (en el mejor de los casos).

* Comprobar consistencia semántica de los argumentos de una función.

Una función podría verse obligada a usar para sus argumentos tipos de datos cuyo rango de valores abarque más valores de los deseados para dichos argumentos. Por ejemplo, sea la función setDia(int dia) que asigna a un atributo de una clase Fecha un valor para el día del mes. Aunque el tipo del argumento está declarado como int, dicho argumento sólo debería aceptar valores comprendidos entre el 1 y el 31. Además, se debería comprobar que el día es consistente con el mes de la fecha, ya que no todos los meses tienen 31 días.

Existen dos soluciones diferentes para este problema: (1) comprobar la coherencia de los argumentos antes de ejecutar el cuerpo de la función y lanzar una excepción si estos argumentos fuesen inconsistentes; o (2) declarar precondiciones para la invocación de esta función. La precondición advierte que el correcto funcionamiento de la función no está garantizado si no se respetan las precondiciones. El ejemplo de abajo muestra una función Java con precondiciones. Una precondición, usando lenguaje coloquial, es una forma de decir "el que avisa no es traidor".



* Acostumbrarse a depurar aplicaciones software sin usar depuradores.

Los depuradores de código son unas excelentes herramientas. Pero, por desgracia, a medida que se programan sistemas más y más complejos, dejan de ser útiles, dado que en muchas ocasiones, simplemente no pueden usarse. Por ejemplo, suelen fallar cuando se trata de integrar una aplicación con un sistema mantenido por terceros, cuyo código fuente no está disponible. No funcionan normalmente cuando cuando se trabaja con simuladores de tecnologías novedosas, como por ejemplo, de dispositivos móviles. Sin embargo, casi siempre dispondremos algún lugar donde poder imprimir mensajes y los valores de aquellas variables que sean de nuestro interés. Por eso se aconseja acostumbrarse a depurar aplicaciones software a base de imprimir mensajes por consola en lugar de usar depuradores.

* Expresar valores literales como constantes.

Cuando se codifica una aplicación software, en muchas ocasiones hay que usar valores literales (e.g. el valor del número e, el tamaño máximo de un vector, la dimensión de una matriz o el CIF de una empresa). Se aconseja expresar estos valores como constantes. Esto debería incrementar la adaptabilidad de la aplicación, dado que si estos valores mutacen, simplemente deberíamos modificar el valor asociado a una constante.

* La herencia nunca debe usarse para heredar atributos de una clase.

La herencia se ha considerado en muchas ocasiones un mecanismo de reutilización o de extensión. De hecho en Java la herencia se declara con la palabra clave extends. Sin embargo, la utilización de la herencia como simple mecanismo de reutilización suele resultar en una violación del principio de sustitución de Liskov [7]. Dicho principio establece que para que nuestro sistema de tipos sea seguro y los objetos que pertenecen a él tengan el comportamiento esperado, cada objeto de una clase hija debe poder ser utilizado de forma segura allá donde se requiera un objeto de la clase padre, en sustitución de éste. Esto lo que establece es que la herencia es una forma de reutilización del tipo, pero no necesariamente de su implementación. Por tanto, si lo que queremos es reutilizar implementación, es mucho más recomendable y seguro reutilizar por composición má delegación.

Fuente: <https://personales.unican.es/sanchezbp/teaching/faqs/programming.html>