Guía de estilo de programación Velneo

Una primera versión con mucho futuro

Esta es la primera versión de este documento que nos gustaría que siga creciendo y evolucionando.

Deseamos contar con tu participación ya que este documento ha nacido con la idea de que pueda convertirse en tu herramienta de trabajo, tanto si eres un profesional autónomo como si formas parte de un equipo de desarrollo.

Estaremos encantados de saber que has usado este documento directamente o que los has utilizado como base para crear tu propia guía de estilo de desarrollo de aplicaciones con Velneo.

Sabemos que aún nos queda mucho para que esta documento pueda llegar a publicarse como un libro, por eso pedimos tu colaboración a la vez que tu comprensión para que sepas perdonarnos todas las erratas que encuentres.

Envíanos tus comentarios, correcciones y sugerencias a velneo@velneo.com

Guía de estilo de programación Velneo

¿Qué características tienen los buenos programadores?

- Un buen programador sabe trabajar en equipo.
- Desarrolla código fácil de mantener y entender.
- Consigue ser productivo tanto él como su equipo.
- Comparte su conocimiento y su código.
- Ayuda a formarse a otros compañeros.
- Colabora en la creación y mantenimiento de una guía de estilo.

¿A quién está orientado este documento?

A desarrolladores que desean utilizar un sistema diseñado para aprovechar todas las bondades de la plataforma Velneo, facilitando una forma de programación probada y fiable que acelera tu productividad al evitar tener que pensar en muchos aspectos del día a día de un programador.

¿Qué me aporta utilizar una guía de estilo?

Aporta múltiples ventajas como:

- No pensar.
    - Cuando tenga que programar objetos y el código.
    - Cuando tenga que encontrar un objeto, subobjeto o código.
    - Cuando tenga que organizar los nuevos objetos.
    - Cuando tenga que poner un identificador.
    - En general, en cualquier acción de desarrollo que deba ser mecánica.

- Todos los programadores de un equipo desarrollamos igual.
    - Poder entender el código de cualquier programador me aportará un importante ahorro de tiempo.
    - Que el resto del equipo entienda mi código sin tener que explicarlo.
    - Que al editar cualquier objeto me sienta cómodo, como lo estaría con cualquier objeto que hubiese desarrollado yo.
    - En definitiva, conseguimos que nuestras aplicaciones sean más fáciles de mantener, y por lo tanto hagan más rentable mis horas de trabajo.

¿Y si hay aspectos de esta guía que no me gustan?

El motivo principal por el que entregamos esta guía en formato editable Word es que la adaptes a tus criterios o los de tu equipo de desarrollo. En el peor de los casos esta guía supondrá un estupendo guión sobre el que podrás construir tu propia guía de estilo de programación.

¿Qué pasa con mi creatividad? ¿Se pierde al usar una guía de estilo?

Al contrario, usar una guía de estilo te va a permitir ser más productivo en la parte que menos valor aporta a tu programación. No tener que estar pensando en los criterios a aplicar te permite concentrarte en crear objetos con el mejor diseño, usabilidad, el código más optimizado posible. Es en esos aspectos donde la

creatividad de los desarrolladores debe brillar y no en aspectos que perjudiquen el trabajo en equipo o la mantenibilidad de la aplicación.

¿Por qué es tan importante el diseño en el desarrollo de aplicaciones?

Comenzaré por dar una definición de diseño con una frase de Charles Eames en respuesta a una entrevista en 1970 en la que decía lo siguiente:

"El diseño es un plan para disponer elementos
de la mejor forma posible para alcanzar un fin específico"

El diseño no tiene nada que ver con estas frases:

"Pon un logotipo bonito"
"le gusta a mi mujer, y ya sabes que las mujeres siempre tienen buen gusto"
"El diseño es una cuestión de gustos y estética"
"El diseño no es lo mío, dibujo muy mal"

Si analizamos bien una aplicación nos daremos cuenta que está compuesta por muchas piezas, unas son visibles para el usuario y otras no:

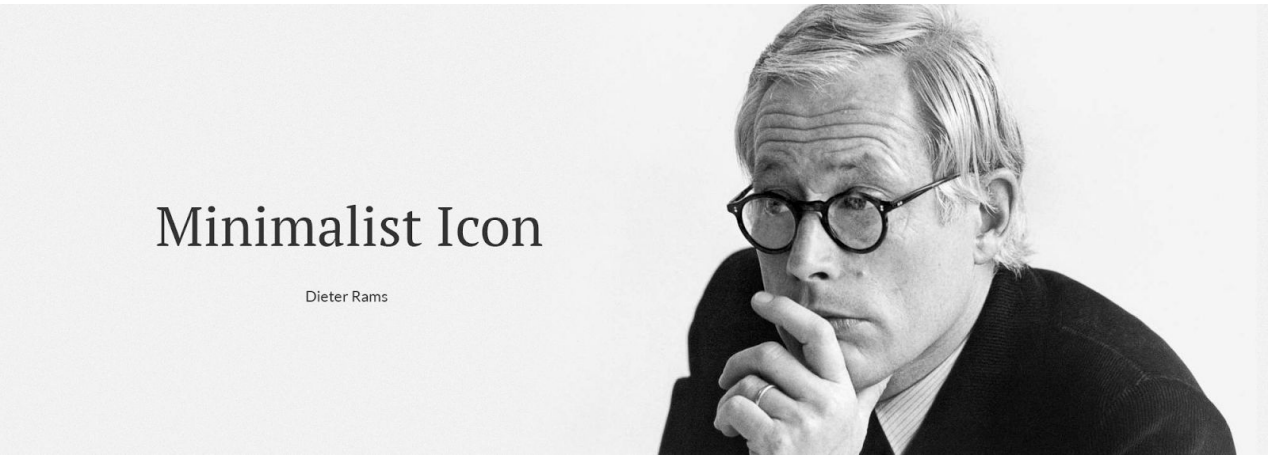
- Base de datos: Tablas, índices, actualizaciones, etc.
- Código: Procesos, funciones, manejadores, etc.
- Interfaz: Formularios, rejillas, informes, etc.

Lo interesante de todo esto es que el diseño se debe aplicar a todas las piezas que forman una aplicación, no exclusivamente a las que tengan que ver con la interfaz. Es en este punto donde trataremos de que este documento sirva para ayudarnos a analizar a fondo todas y cada una de las decisiones que tenemos que tomar en el desarrollo de una aplicación, ya que todas afectan al resultado final de la misma. Aspectos como la optimización afecta directamente a conseguir una buena experiencia de usuario, un buen diseño de formularios ayuda a mejorar la usabilidad y permitirá al usuario ser capaz de moverse por la aplicación sin necesidad de consultar manuales o vídeos de ayuda para entender cómo funciona.

El buen diseño

La palabra diseño es clave en este documento y para ir entrando en materia, vamos a enumerar los 10 principios del diseño que declaró Dieter Rams.

¿Quién es Dieter Rams?



Dieter Rams es un famoso diseñador alemán de la década de los 50 / 60, muy conocido por sus diseños para Braum y Vitsœ. Su manera de ver el diseño, con la máxima “Menos, pero con mejor ejecución” y muy centrado en la funcionalidad, marcó a otros muchos diseñadores, como es el caso de Jonathan Ive el actual Jefe de diseño de Apple. Dieter enunció lo que para él son los principios de diseño.

10 principios del diseño según Dieter Rams

Principio	Descripción
El buen diseño es innovador.	El diseño tiene una innovación ilimitada, porque cada nuevo avance tecnológico, permite crear nuevos productos mejor diseñados.
El buen diseño hace útil al producto.	El objetivo primordial de un producto es su utilidad. El diseño (la forma) debe ser primordialmente práctico y de manera secundaria tiene que satisfacer ciertos criterios de carácter psicológico y estético, evitando de estos criterios las características que no potencian su utilidad.
El buen diseño es estético.	El diseño bien ejecutado no carece de belleza. La calidad estética de un producto forma parte integral de su utilidad ya que los productos utilizados cotidianamente tienden a tener un efecto indirecto en las personas y su bienestar.

El buen diseño hace al producto comprensible.	Un buen diseño simplifica la estructura del producto y lo predispone a expresar claramente su función mediante la intuición del usuario. Idealmente su propósito será intuitivo para todo usuario.
El buen diseño es discreto.	Para que un producto sea discreto, tanto él como su diseño deben ser sobrios y neutros (a la vez). Un producto no debe ser una obra de arte o un objeto de decoración, que confunda y distorsione su uso, debe ser estéticamente atractivo, sí, pero debe carecer de evocaciones.
El buen diseño es honesto.	Un diseño honesto nunca intenta falsificar el auténtico valor e innovación del producto dado. Asimismo, un diseño verdaderamente honesto nunca trata de manipular al consumidor mediante promesas de una utilidad apócrifa, inexistente o más allá de la realidad física del producto.
El buen diseño perdura en el tiempo.	Toda moda es inherentemente pasajera y subjetiva. La correcta ejecución del buen diseño da como resultado productos inherentemente objetivos y anacrónicamente útiles. Estas cualidades se ven reflejadas cuando los usuarios tienen la tendencia de atesorar y favorecer aquellos productos bien diseñados incluso en aquellas sociedades cuyas tendencias de consumo claramente favorecen productos desechables.
El buen diseño es amigo del medioambiente.	Un buen diseño debe contribuir a la preservación del medio ambiente mediante la conservación de los recursos y la minimización de la contaminación física y visual durante el ciclo de vida del producto.
El buen diseño es consecuente con el mínimo detalle.	Menos, pero con mejor ejecución. Este enfoque fomenta los aspectos fundamentales de cada producto y por lo tanto evita arrastrarlos torpemente con todo aquello que no es esencial. El resultado ideal es un producto de mayor pureza y simplicidad.
El buen diseño es el menor diseño posible.	Menos, pero con mejor ejecución, este enfoque fomenta los aspectos fundamentales de cada producto y por lo tanto evita lastrarlos torpemente con todo aquello que no es esencial. El resultado ideal es un producto de mayor pureza y simplicidad.

Para terminar hacemos mención a una frase de Antoine de Saint Exupéry.

“La perfección no se alcanza cuando no hay nada más que añadir,

sino cuando no hay nada que quitar”

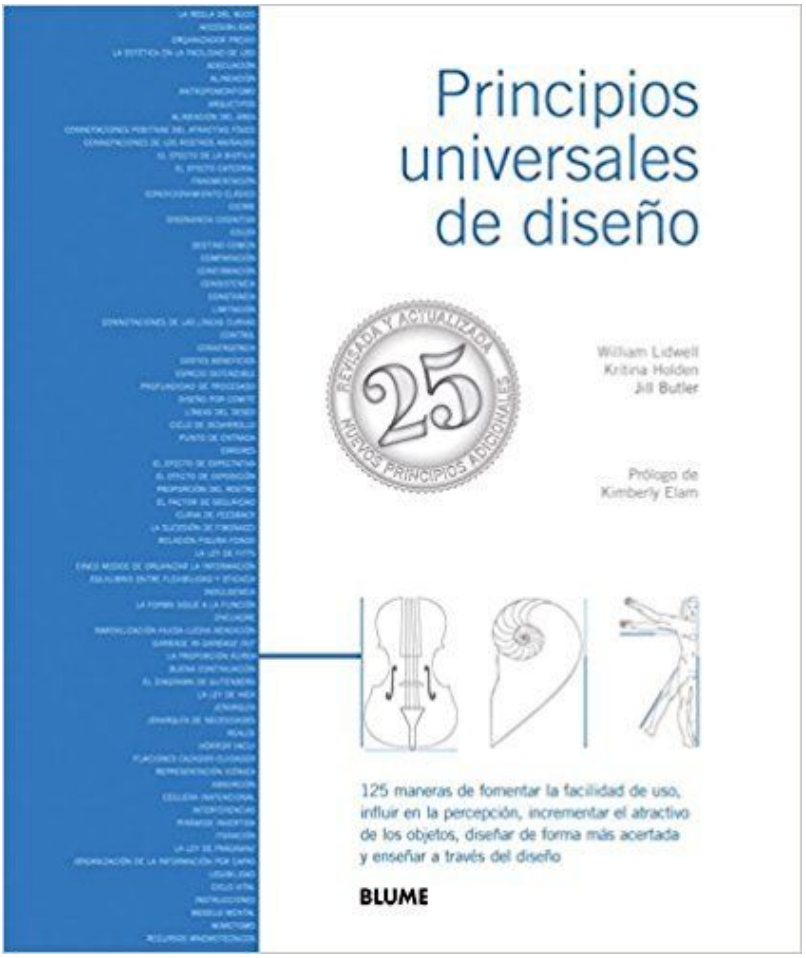
Principios universales de diseño & Experiencia

Para la creación de esta guía de estilo de programación Velneo no hemos inventado nada, al contrario, nos hemos apoyado en dos fuentes clave: los principios básicos de diseño y en la experiencia acumulada durante 2 décadas en el desarrollo de aplicaciones empresariales con plataformas de desarrollo Velneo.

Utilizaremos los principios universales de diseño para argumentar las decisiones que se han tomado en el desarrollo de aplicaciones Velneo documentadas en este guía.

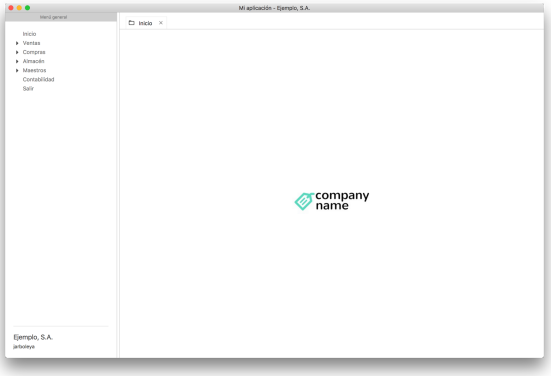
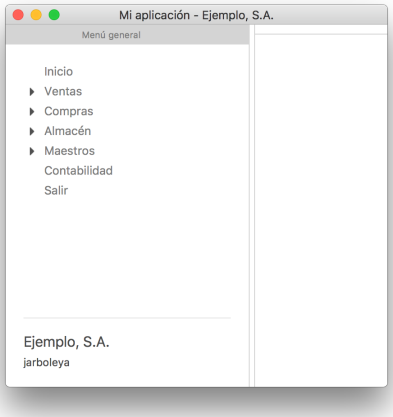
Los principios universales de diseño no son conjeturas, son reales y están basados en investigaciones sólidas, por ese motivo funcionan.

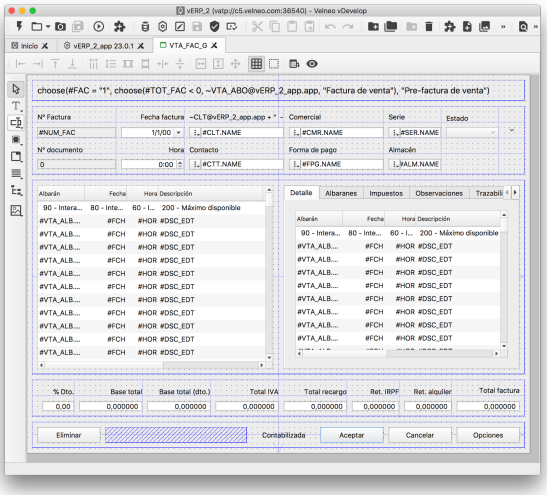
Hemos utilizado como base de los principios universales de diseño el libro “Principios Universales de Diseño” de William Lidwell, Kritina Holden y Jill Butler publicado por BLUME.



Principios universales de diseño con ejemplos de aplicación

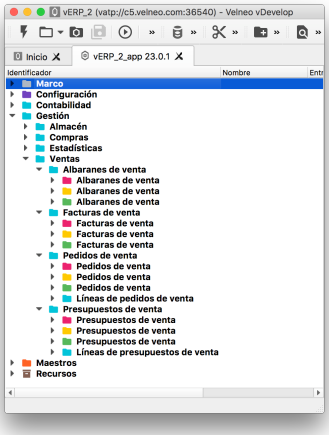
A lo largo del documento haremos hincapié en los diferentes principios de diseño que veremos aplicados en esta guía con su descripción y correspondiente ejemplo práctico.

Principios universales de diseño	Ejemplo de aplicación
El efecto de exposición La exposición repetida a estímulos hacia los cuales se tienen sentimientos neutros aumenta el atractivo de dichos estímulos.	No te dejes guiar por la primera impresión al ver el diseño de una aplicación. Espera a usarla durante varios días hasta que tus estímulos estén basados en la repetición. Aplicar diseño atemporal y duradero normalmente se percibe como simple y poco interesantes, pero con el uso se percibe la belleza de la funcionalidad. 
Regla del 80/20 El 80% del empleo de un producto implica el 20% de sus características.	Dedícale el 80% del tiempo de desarrollo a ese 20% de tu aplicación. Si una funcionalidad no es útil para el 80% de las empresas no debería estar incluida en el núcleo estándar de tus aplicaciones. 
Alineación Los elementos de un diseño deben estar alineados entre sí. De este modo se logra transmitir unidad y cohesión.	Utiliza la cuadrícula en el diseño de tus formularios. Aplica la misma alineación al mismo tipo de dato.



Fragmentación
Técnica que consiste en combinar unidades de información en un número limitado de unidades y fragmentos, de modo que la información resulte más fácil de procesar y recordar.

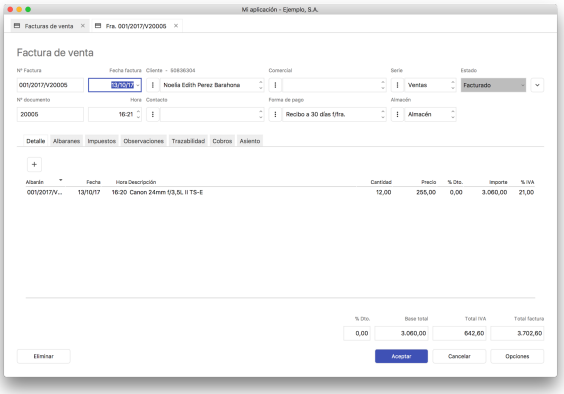
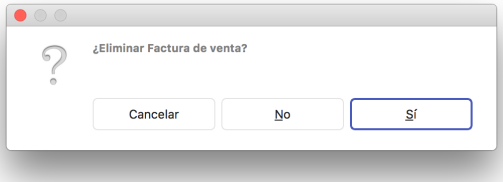
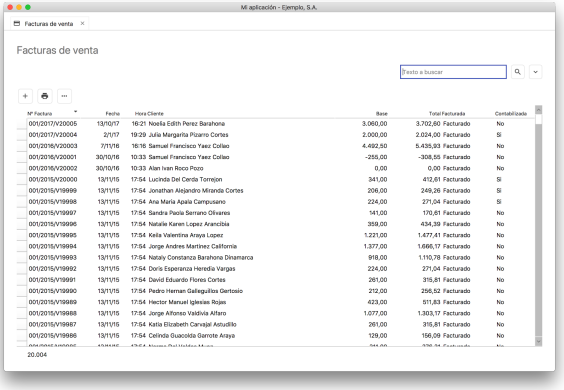
Debemos ser rigurosos en la aplicación del encarpetado de objetos de los proyectos para facilitar su organización y localización.

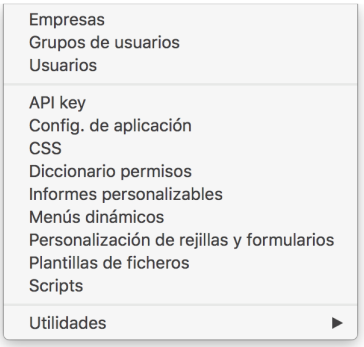


Color
El color se emplea en diseño para atraer la atención, agrupar elementos, indicar significados y realzar la estética.

El color ayuda a que el usuario localice sin esfuerzo cognitivo el botón de llamada a la acción, como “Aceptar” en los formularios.

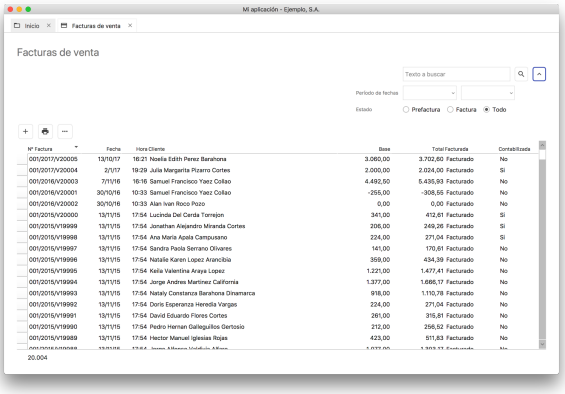
El borde del control de edición con el foco ayuda a su localización de forma rápida y precisa.

	
Confirmación Técnica para evitar acciones no intencionadas, que consiste en exigir la verificación de las acciones antes de llevarlas a cabo.	El cuadro de diálogo de confirmación que se muestra antes de la eliminación de un registro al pulsar el botón eliminar en un formulario. 
Consistencia La facilidad de uso de un sistema mejora cuando las partes similares del mismo se expresan de modo semejantes.	Todos los formularios deben mantener los títulos, cabecear, detalle y botones en posiciones similares. 
Limitación Método para reducir las acciones que se pueden llevar a cabo en un sistema.	La aplicación de permisos para tener acceso a opciones de menús o a la edición de registros. Opciones o datos solo disponibles para supervisores.



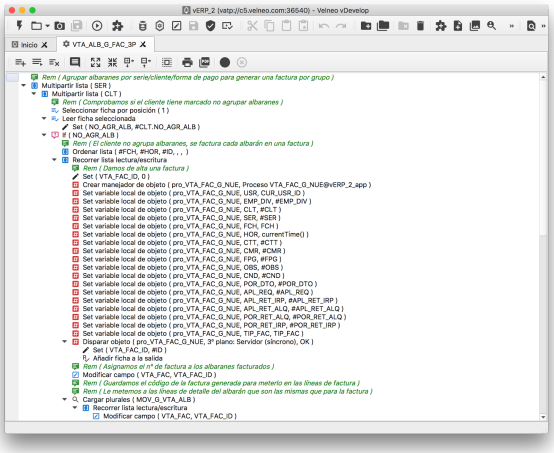
Control
El nivel de control proporcionado por un sistema debe guardar relación a la eficacia y los niveles de experiencia de las personas que utilicen dicho sistema.

Ocultar opciones avanzadas o datos de un formulario a los que se accede con un botón que evita que los usuarios con menos experiencia vean demasiados en pantalla.



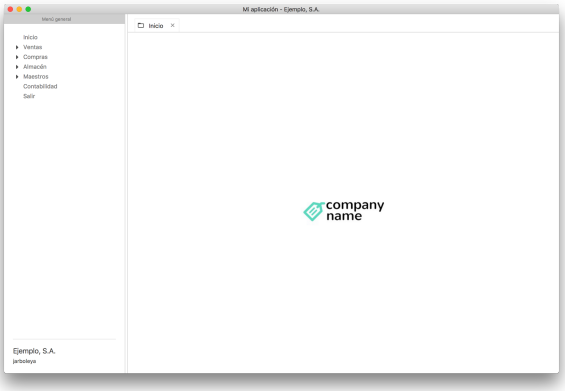
Diseño por comité
Proceso de diseño basado en la creación de consenso, toma de decisiones en grupo e iteración exhaustiva.

Maquetación de formularios, tamaños mínimos de controles, anchos de columnas de rejillas o el sistema de comentarios del código en procesos.



Punto de entrada
Tenemos tendencia a juzgar los libros por sus cubiertas, los edificios por sus vestíbulos y las webs por su portada.

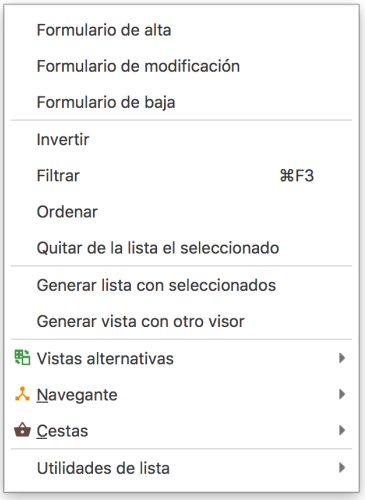
Diseña la inferfaz para que resulte sencilla y elegante. Mejor diseñar con tendencia al minimalismo que a la sobrecarga de contenido.



Ley de Fitts
El tiempo para desplazarse hasta un objeto es una función del tamaño de dicho objeto y de la distancia hasta el mismo.

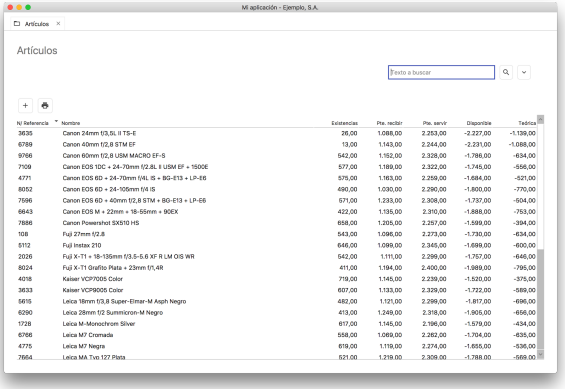
Aplica un tamaño generoso a los botones y aplicar un orden a los controles que reduzca el uso del ratón o la distancia al objetivo.

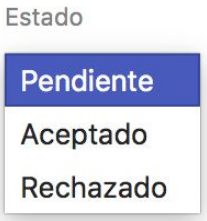
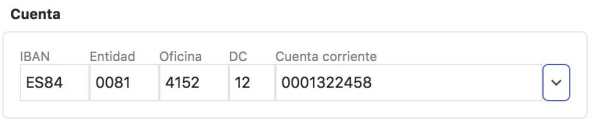
Utiliza menús contextuales o menús de botón para conseguir que el usuario no tenga que desplazarse para ejecutar acciones.



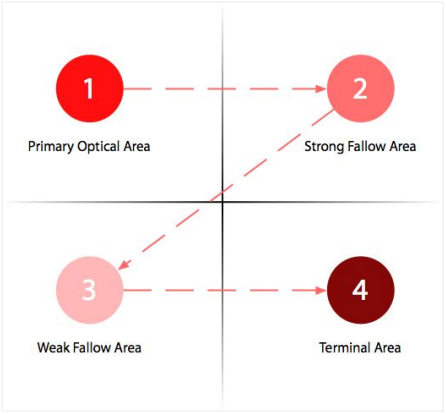
Cinco modos de organizar la información
Categoría, tiempo, ubicación, orden alfabético y continuo.

Usa el orden alfabético por el nombre en maestros y con claves alfabéticas, aplica orden temporal para documentos o registros con fecha y hora.

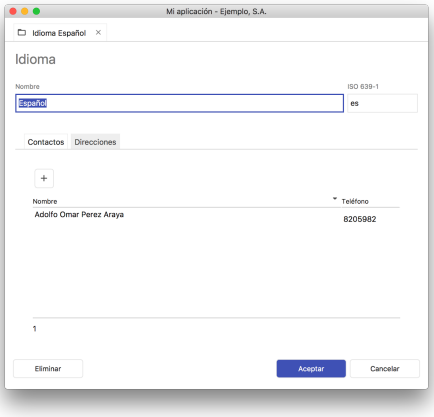


Equilibrio entre flexibilidad y eficacia A medida que aumenta la flexibilidad de un sistema, disminuye su eficacia.	Utiliza el menor número posible de opciones de menú. Busca el equilibrio entre las funcionalidades configurables y la complejidad de un exceso de configuración. 
La forma sigue a la función La belleza de un diseño constituye el resultado de la pureza de su función (en la naturaleza pasa lo contrario la función sigue a la forma)	Elimina los elementos redundantes como icono y texto o que no aportan funcionalidad de forma sencilla. Es mejor que el usuario tenga una única forma de hacer las cosas y que sea lo más sencilla posible. 
Garbage in-garbage out La calidad del rendimiento de un sistema depende de la calidad de la entrada de la información de dicho sistema.	Limitar el tamaño de los campos de entrada a su contenido o permitir solo la entrada de determinados valores correctos ayuda al usuario a no meter información basura. 

El diagrama de Gutenberg
Diagrama que describe el patrón general seguido por la vista cuando observamos una información homogénea distribuida de manera regular.



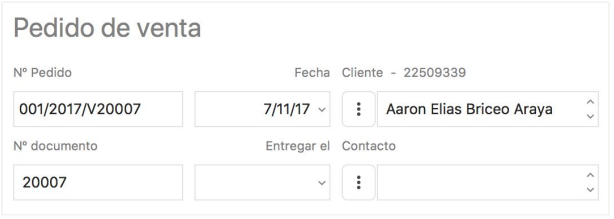
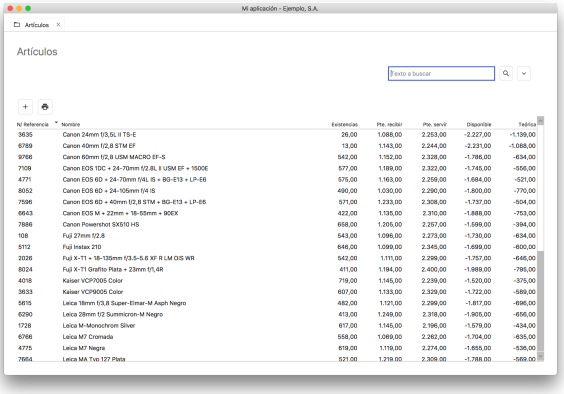
Ubicar los botones en la esquina inferior derecha de un formulario ayuda al usuario a comprender que es el paso final a realizar para cerrar la edición.

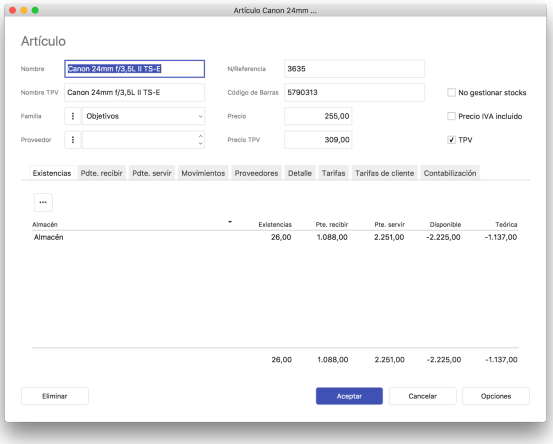
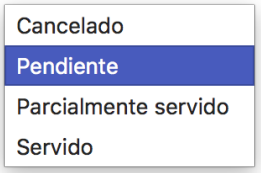


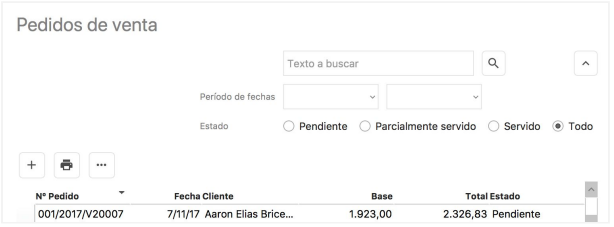
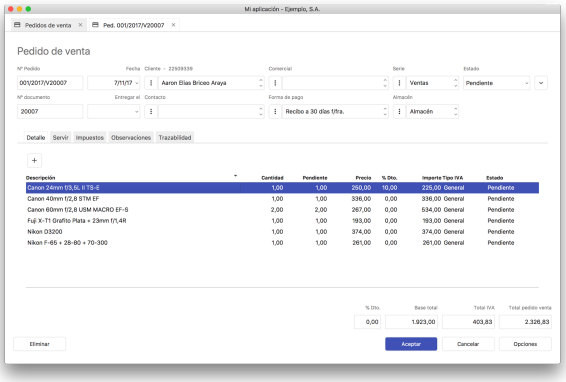
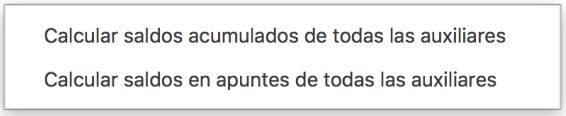
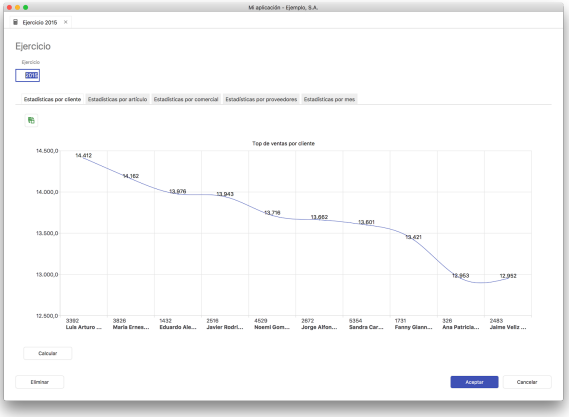
La ley de Hick
El tiempo que se tarda en tomar una decisión aumenta a medida que se incrementa el número de alternativas.

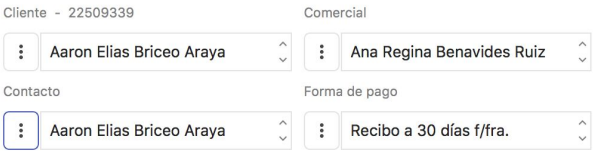
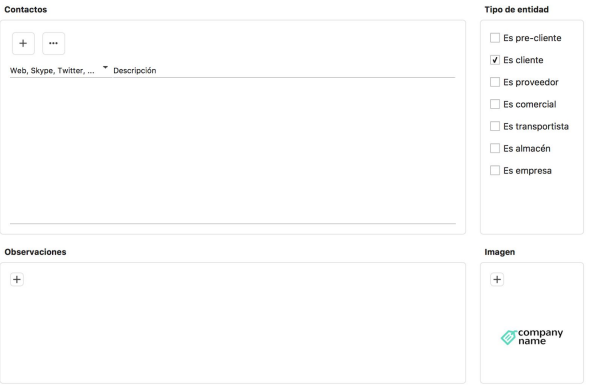
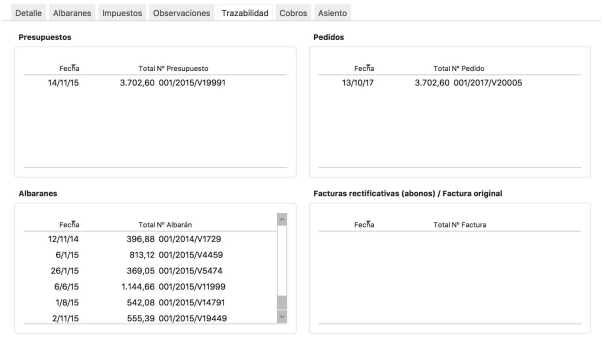
Reducir el número de opciones de los menús ayuda a acelerar la toma de decisiones.

Jerarquía de necesidades
Para que un diseño tenga éxito, debe satisfacer las necesidades básicas de las personas antes de intentar satisfacer otras necesidades más elevadas.

Realce Técnica eficaz para llamar la atención sobre los elementos de un diseño	El tamaño de letra del título nos ayuda a destacarlo sobre el resto de información.  Conviene no realzar más del 10% del diseño visible, los efectos se diluyen a medida que aumenta el porcentaje.
Hórror vacui Es una expresión latina que significa “temor al vacío” y hace referencia al deseo de llenar los espacios vacíos con información u objetos. A medida que el hórror vacui aumenta el valor percibido desciende.	El espacio en blanco de la cabecera de los menús aumenta el valor percibido por el usuario de nuestro producto. 
Representación icónica Uso de imágenes para facilitar la identificación y el recuerdo de señales y controles.	El uso de iconografía de Material Design de Google favorece que los usuarios identifiquen la funcionalidad del icono al ser ampliamente conocido por lo usuarios de dispositivos móviles o las aplicaciones de escritorio de Google. 
Legibilidad Claridad visual de un texto, por lo general basada en el tamaño, el tipo de letra, el contraste, los bloques de texto y el espaciado de los caracteres utilizados.	El nuevo estilo visual aplicado en las CSS de Velneo ayuda a la legibilidad de la información.

	
Modularidad Método para controlar la complejidad de un sistema, que consiste en dividir los grandes sistemas en múltiples sistemas de menor tamaño.	El menú de gestión es una muestra de modularidad. ▼ Ventas Presupuestos Pedidos Albaranes Facturas Cobros Remesas de cobros ▼ Compras Pedidos Albaranes Facturas Pagos Remesas de pagos ▼ Almacén Inventario valorado Movimientos Traspasos
La navaja de Ockham Ante la posibilidad de elegir entre dos diseños equivalentes desde el punto de vista funcional, conviene decantarse por el más sencillo.	Los nuevos combobox sin iconos son visualmente más sencillos de leer e interpretar. 
Revelación progresiva Estrategia para controlar la complejidad de la información que consiste en mostrar únicamente la información necesaria o requerida en un momento dado.	Los menús sencillos con botón de búsqueda avanzada facilitan el acceso a búsquedas más complejas solo cuando es necesario para el usuario.

	
Proximidad Los elementos cercanos entre sí se perciben como más relacionados que los que están muy separados.	En los formularios de documentos de compras y ventas los elementos de cabecera se ven claramente relacionados entre sí, al igual que ocurre con los totales del pie, sin embargo en la cabecera y los totales la distancia hace que no se perciba relación. 
Redundancia Uso de más elementos de los necesarios a fin de mantener el rendimiento de un sistema en caso de fallo de uno o más elementos del mismo.	Aunque las existencias o los saldos se calculan automáticamente, siempre es conveniente disponer de opciones de recálculo manual ante un posible fallo. 
Proporción señal-ruido La degradación de la señal tiene lugar cuando la información se presenta de manera ineficaz: letra poco clara, grafías inadecuadas o iconos y etiquetas ambiguos. La claridad de la señal mejora a través de la presentación sencilla y concisa de la información.	La representación mediante una gráfica clara y sencilla ayuda a interpretar la información de forma rápida y concisa. 

Similitud Los elementos similares se perciben como más relacionados entre sí que los que no lo son.	El uso del mismo icono en los botones de menú ayuda al usuario a entender que su funcionamiento es similar aunque sean campos diferentes. 
Conexión de lo uniforme Los elementos que comparten propiedades visuales uniformes, como el color, se perciben más relacionados entre sí que los que no guardan ninguna conexión.	El uso de cajas de grupo con controles del mismo tipo ayuda a entender que están conectados. 
Visibilidad El uso de un sistema mejora cuando su estado y los métodos de empleo son claramente visibles.	El uso de información como la trazabilidad ayuda a entender el funcionamiento del ciclo de compras o ventas. 

Soluciones

El nombre que damos a las soluciones es utilizado para crear la carpeta en disco que contendrá los proyectos. En principio los sistemas operativos actuales no deberían presentar ningún problema en el uso de caracteres acentuados, sin embargo es recomendable no usar caracteres especiales que no puedan ser utilizados en el nombre asignado a la carpeta.

Recomendaciones sobre el nombre de los proyectos

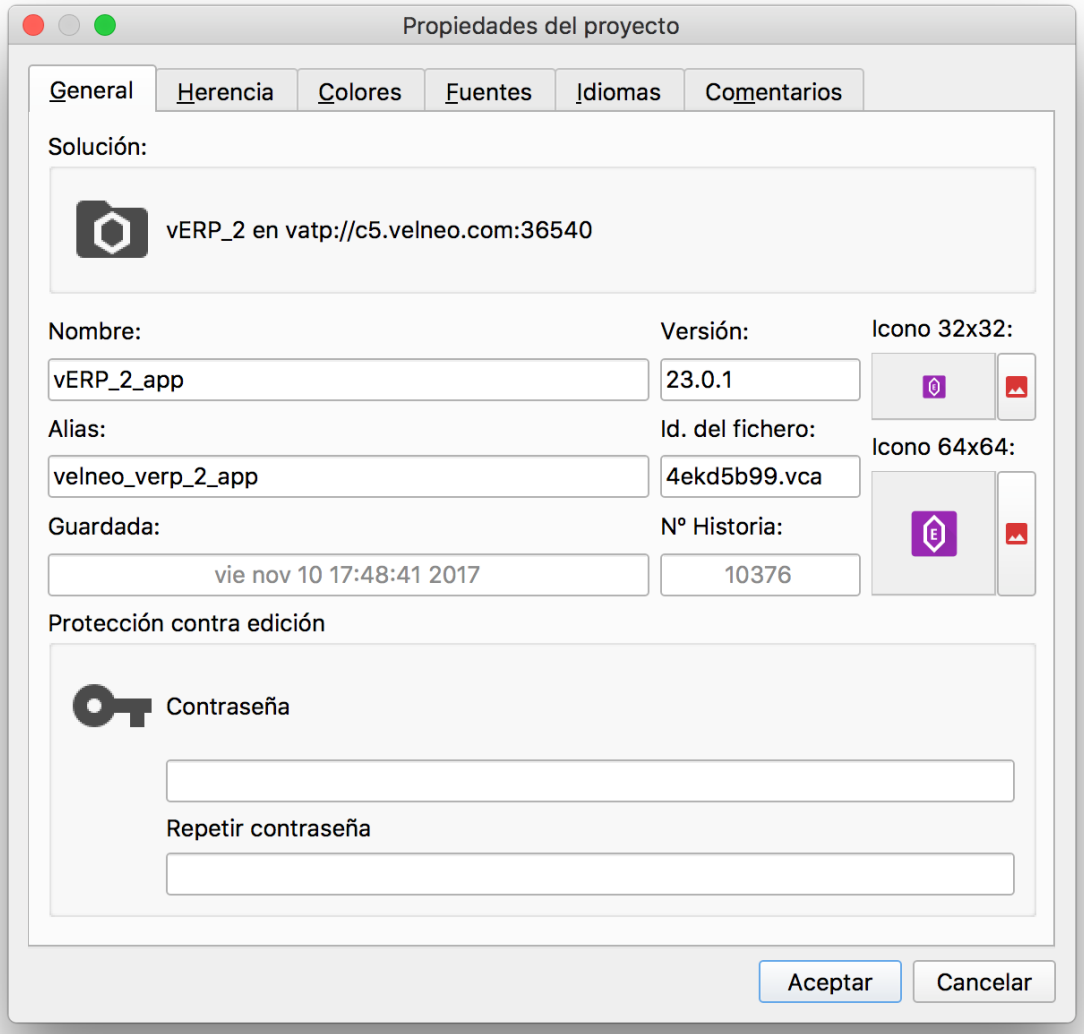
El nombre de ser único, descriptivo y lo más corto posible. Veamos algunos ejemplos.

No recomendable	Motivo
Gestión Integrada	Demasiado genérico
Gestión Integrada #1	Usa caracteres especiales
Gestión Integrada para Industrias Derivadas del Proceso Lácteo	Demasiado largo
GIIDPL	Difícil de recordar. No recomendable salvo que se el nombre de un producto estándar cuyas siglas se usan de forma constante.

Recomendable	Motivo
Gestión Integrada Ejemplosa	Corto y personalizado para mi empresa, lo que lo convierte en algo único
Ejemplosa GESINT	Corto y aplicando un nombre de producto o módulo
eGESINT	Nombre comercial de un producto

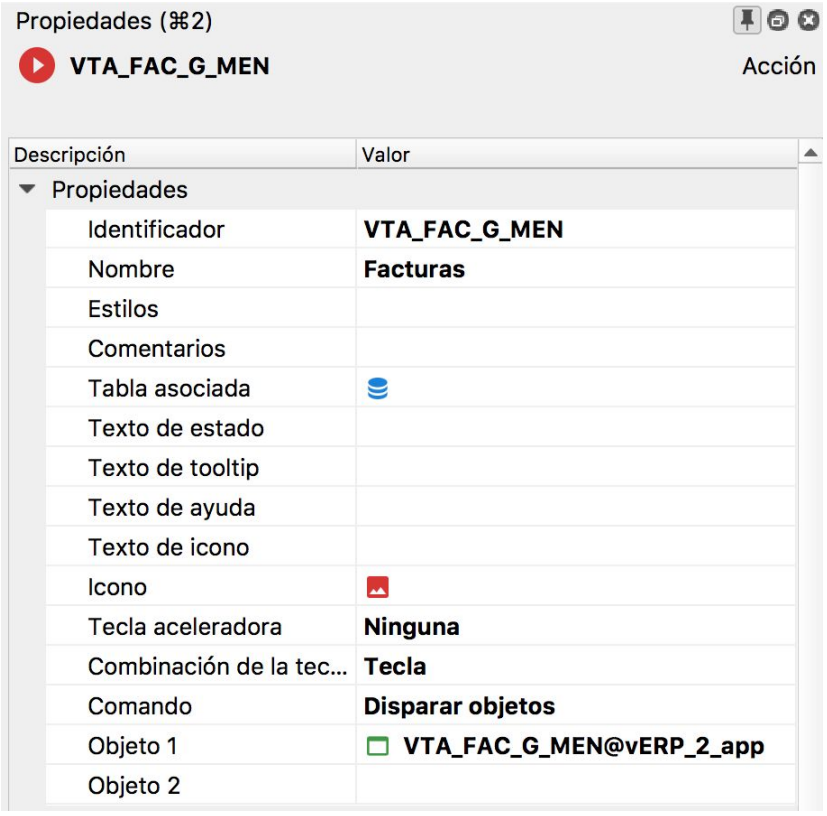
Proyectos

Los contenedores de objetos son la pieza clave en el diseño de la arquitectura de nuestras aplicaciones. Por este motivo es bueno tener presentes algunas recomendaciones a la hora de crear aplicaciones con mayor o menor complejidad.



Recomendaciones generales para proyectos de aplicación y datos

Recomendaciones generales para proyectos de aplicación y datos
La **longitud del nombre** o descripción de un proyecto no es un problema en sí, sin embargo la longitud del nombre nos afectará a la hora de poder ver los identificadores “completos” de los objetos. Por este motivo, debemos usar el criterio menos es más. En la siguiente imagen podemos observar que el identificador del objeto se puede leer entero.



Sin embargo, si el nombre del proyecto fuese “Gestión Integrada de Automoción” ya no entraría en este espacio. El resultado es que tendríamos que hacer muy ancho el dock donde se muestran las propiedades de un objeto. Por este motivo, es recomendable usar o nombres cortos, siglas o abreviaturas que permitan reducir el tamaño del nombre del proyecto.

El que existan varios proyectos con el mismo nombre, no supone un problema funcional debido a que a nivel interno se utiliza el “id del fichero” y no su nombre. Sin embargo, no es conveniente tener nombres duplicados ya que cuando los veamos juntos en un mismo esquema no podremos diferenciarlos de forma directa.

En el nombre de los proyectos se pueden dejar espacios en blanco entre las diferentes palabras, es conveniente que el equipo establezca el criterio de utilizar o no espacios en blanco para conseguir que todos los proyectos se creen con el mismo criterio.

Es conveniente añadir un **sufijo** al nombre del proyecto indicando si se trata de aplicación o datos, en esta guía utilizamos los sufijos “app” y “dat” respectivamente. Sin embargo, se puede utilizar prefijos más cortos como “a” y “d”.

El motivo por el que conviene utilizar estos sufijos está relacionado con la posibilidad de crear el mismo objeto en cualquier tipo de proyecto, por ejemplo podemos crear un proceso en el proyecto de aplicación o datos, si llamamos igual a ambos proyectos no podríamos saber cuando vemos el identificador del objeto donde podremos encontrarlo.

El **alias** es un datos "obligatorio" que no debemos olvidarnos de cubrir, ya que se utilizará en diferentes ámbitos de la aplicación, sobre todo al crear scripts de JavaScript en el que los identificadores de los objetos se componen utilizando el alias del proyecto que lo contiene y el identificador del propio objeto. Por este motivo la recomendación es añadir el alias al proyecto en el mismo momento de su creación.

Recomendaciones sobre el nombre de los proyectos

No recomendable	Motivo
Gestión Integrada	No identifica si es de datos o aplicación
Gestión Integrada #1 app	Usa caracteres especiales
Gestión Integrada para Industrias Derivadas del Proceso Lácteo app	Demasiado largo

Recomendable	Motivo

Ejemplosa_GESINT_app	Corto, único, con sufijo de tipo y sin espacios
eGESINT dat	Corto, único, con sufijo de tipo y con espacios
vERP_2_app	Corto, único, con sufijo de tipo y sin espacios

Diseño de la arquitectura de las aplicaciones

¿Es mejor tener un proyecto de datos o dividir las tablas en múltiples proyectos?
Aquí encaja perfectamente el principio de menos es más. Si podemos tener un único proyecto de datos será más fácil de programar, mantener y evolucionar.

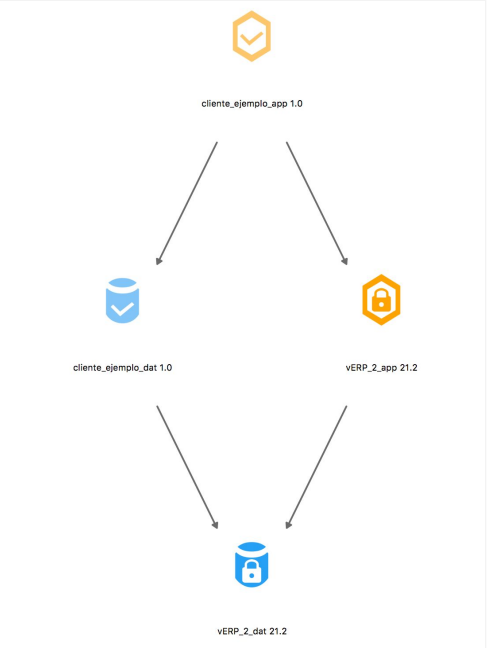
¿Cómo organizo mis tablas de diferentes módulos en un único proyecto de datos?
Aplicando una organización de encarpetado por módulo.

Dentro de cada módulo podremos crear subcarpetas con las tablas del mismo. Con esta organización si mañana queremos mover todas las tablas de un módulo a otro proyecto podremos hacerlo con un cortar y pegar.

¿Cuándo tiene sentido crear más de un proyecto de datos?
Hay varios motivos por los que es necesario crear más de un proyecto de datos:

1. Cuando tenemos un núcleo estándar para todas nuestras aplicaciones que no queremos tocar ni engordar con funcionalidades específicas de cada cliente o sector, y sobre ese núcleo desarrollamos una solución personalizada para un cliente o sector. En este caso se suele crear un proyecto de datos con las tablas específicas para ese cliente o sector que hereda del proyecto de datos del núcleo. Esto nos obligará a tener una instancia de datos para cada proyecto.

2. Cuando un proyecto va a contener tablas comunes a múltiples empresas, en este caso se crea una única instancia de datos para esas tablas comunes y para los datos específicos de cada empresa se crea un proyecto que heredado del de tablas comunes y que se instanciará una vez por cada empresa. Para poder crear esta estructura de instancias necesitaremos dos o más proyectos de datos.


cliente_ejemplo_app 1.0
cliente_ejemplo_dat 1.0
vERP_2_app 21.2
vERP_2_dat 21.2

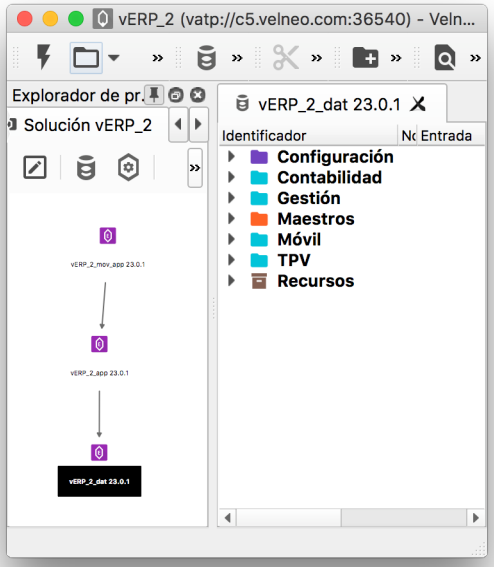
Organización de carpetas

No repitas la organización del inspector por tipo de objeto

Para eso ya tenemos el inspector de objetos por tipo, en su lugar debemos buscar una organización basada en la funcionalidad, por ejemplo por módulos. De esta forma facilitamos que si queremos copiar un módulo completo a otro proyecto podamos hacerlo de forma rápida y sencilla con un solo copiar/pegar.

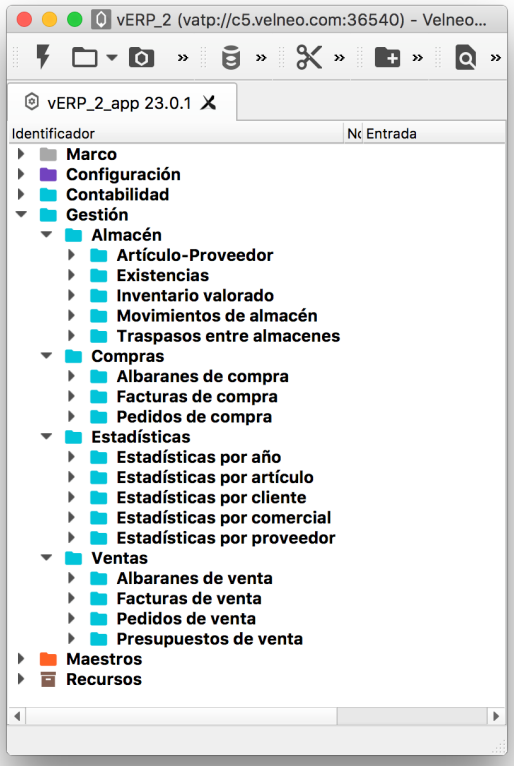
Mantén la misma estructura en los proyectos de datos y aplicación

Cuanto más homogénea sea la organización de los objetos más fácil nos resultará encontrar objetos. Si aplicamos el mismo criterio organizativo en los proyectos de aplicación y datos conseguiremos facilitar aún más la localización de objetos y la posibilidad de moverlos o copiarlos a otros proyectos.



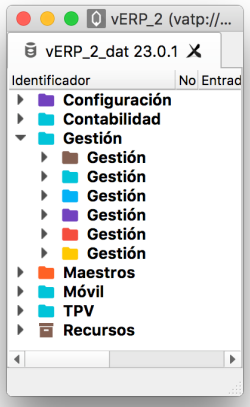
Crea una carpeta para módulo o grupo funcional de objetos

Las carpetas son contenedores de objetos, pero también de subcarpetas, por este motivo es conveniente una buena organización basada en módulos o grupos funcionales con la que podemos navegar a través de sus subcarpetas de forma rápida e intuitiva.



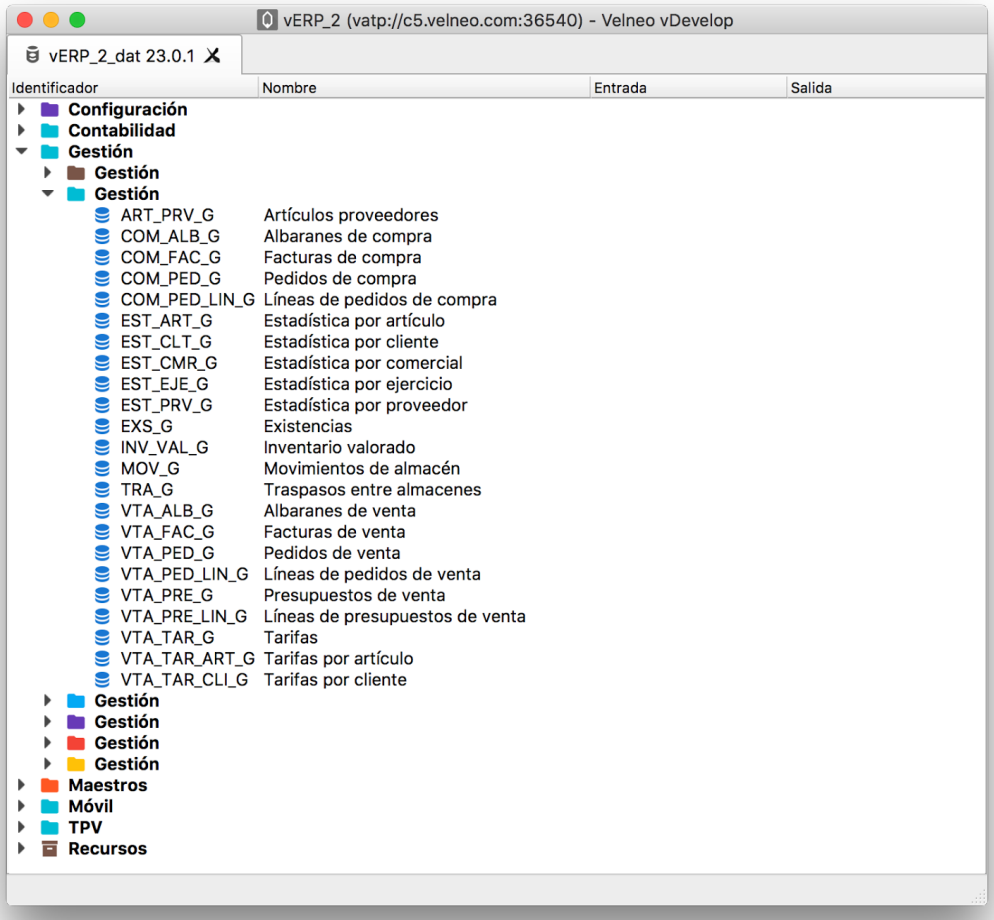
¿Cómo organizar los objetos del proyecto de datos dentro del módulo?

En el proyecto datos, dentro de cada módulo en el caso de que hubiese un gran números de objetos de un determinado podríamos hacer subcarpetas por submódulo. Si el número no es demasiado elevado podemos directamente crear subcarpetas por tipo de objeto.

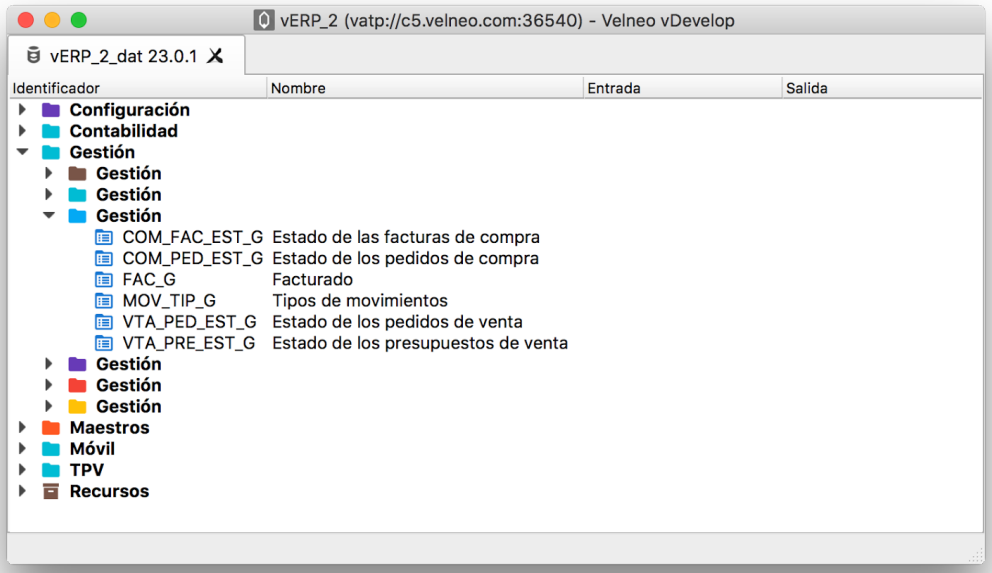


Si hemos creado esquemas, muy recomendable, crearemos una subcarpeta (icono Objetos 4) conteniendo todos los esquemas ordenados por orden alfabético del identificador.

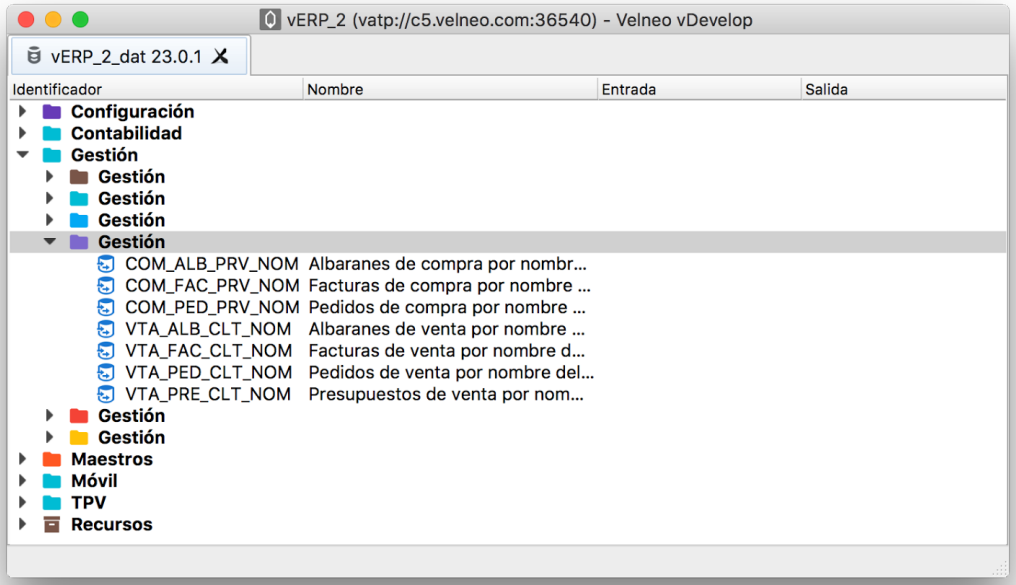
Las tablas se organizan en una subcarpeta (icono Objetos 1) por orden alfabético. Si hay muchas tablas se pueden crear subcarpetas por submódulo y dentro de cada una de ellas las tablas también por orden alfabético.



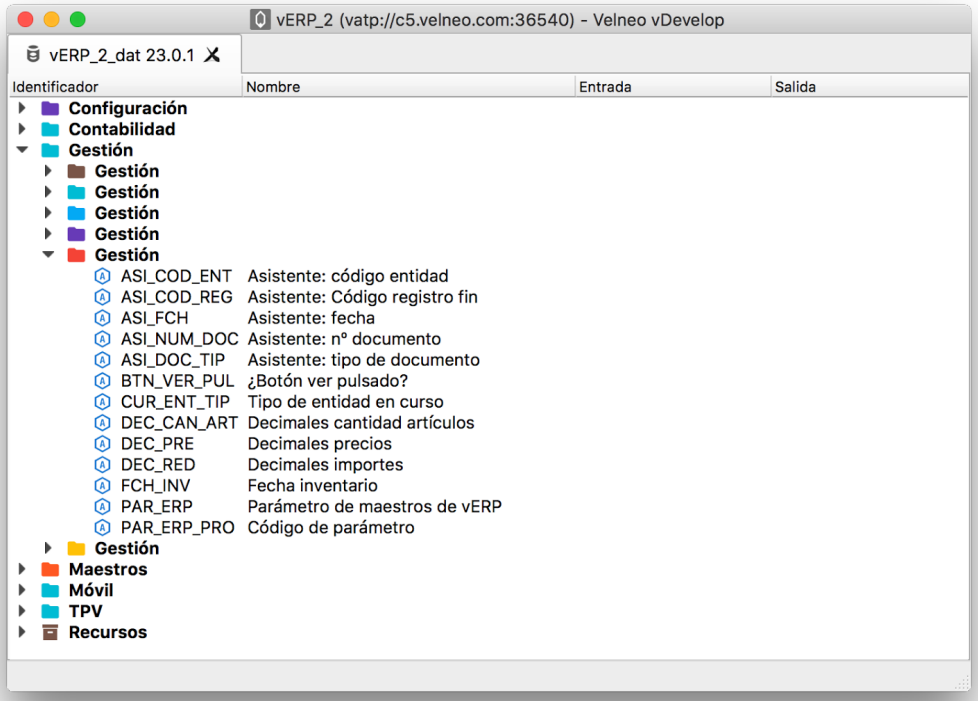
Las tablas estáticas se organizan en una subcarpeta (icono Objetos 9). Si hay muchas se aplica el criterio de subcarpeta por submódulo.



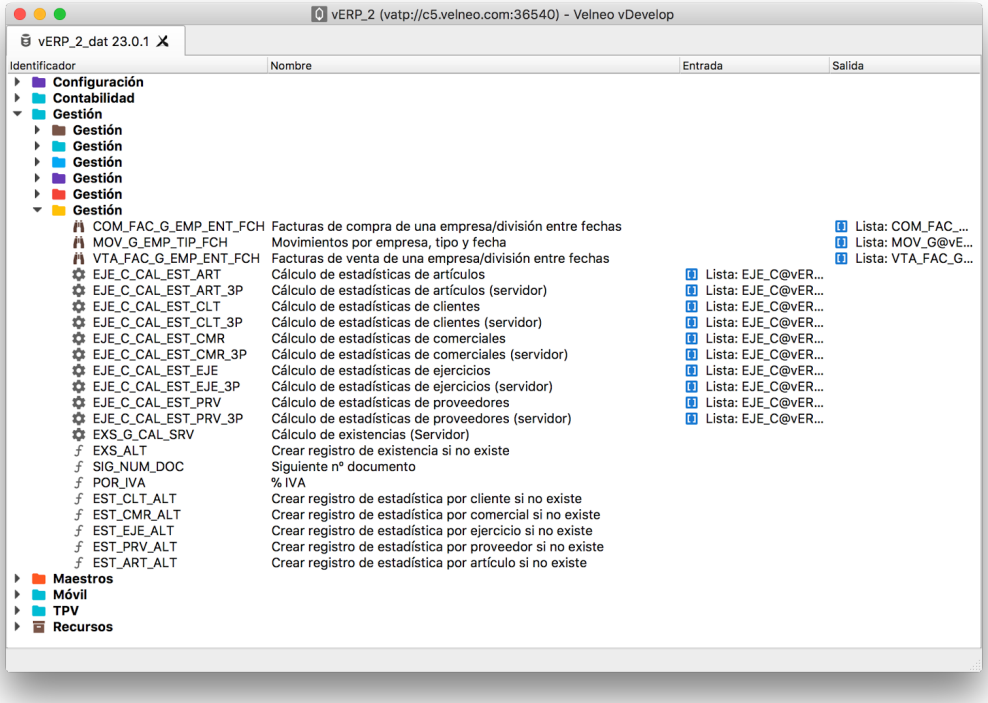
Los índices complejos se organizan en una subcarpeta (icono Objetos 5). Si hay muchas se aplica el criterio de subcarpeta por submódulo.



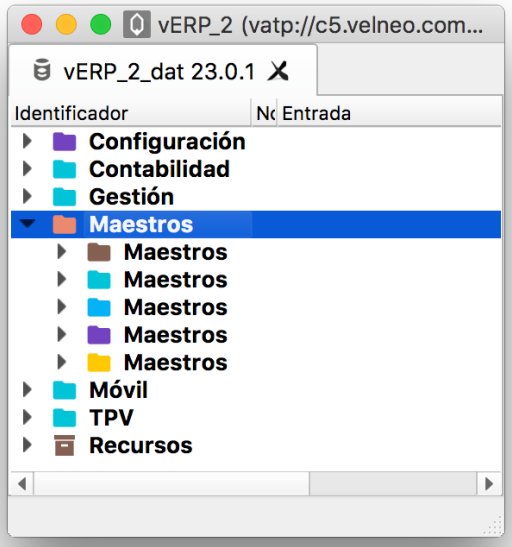
Las variables globales se organizan en una subcarpeta (icono Objetos 3). Si hay muchas se aplica el criterio de subcarpeta por submódulo.



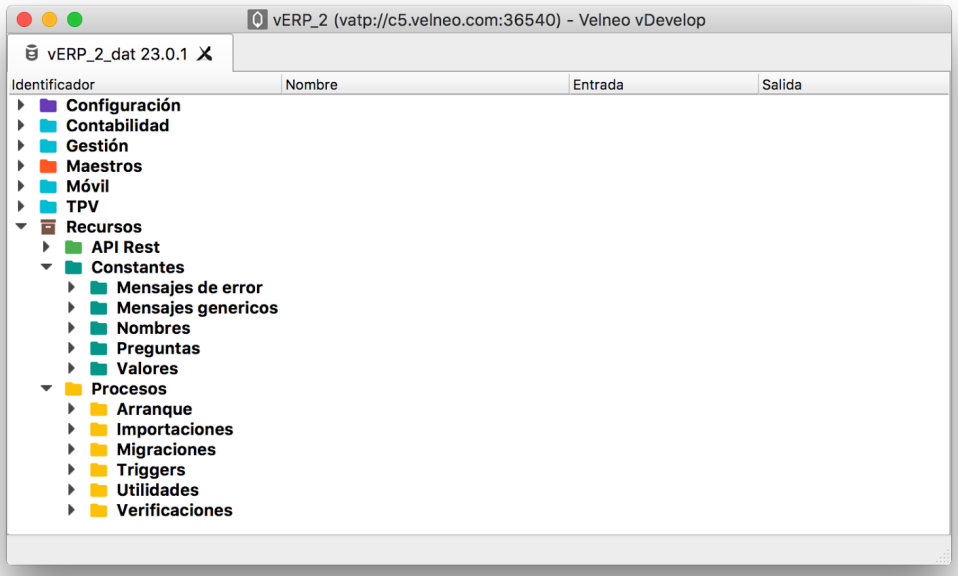
Los objetos de ejecución como procesos, funciones, búsquedas, tubos, etc. se organizan en una subcarpeta (icono General). Si hay muchos se aplica el criterio de subcarpeta por submódulo.



Los maestros generales que se usan en varios módulos se organizan en una carpeta llamada Maestros (icono Objetos 3). La organización interna de subcarpetas idéntica a la comentada para los módulos.



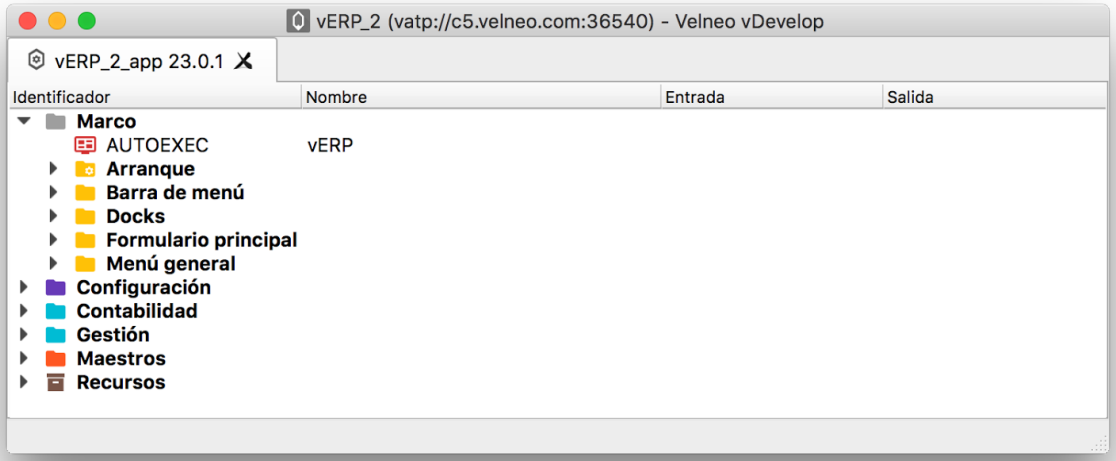
Para el resto de objetos que se usan de forma genérica utilizaremos la carpeta Recursos (icono Recursos) que contendrá subcarpeta para los diferentes recursos, por ejemplo constantes (icono Objetos 8) que a su vez contiene subcarpetas por el uso de las constantes (mensajes de error, mensajes genéricos, nombre, preguntas, valores, etc.). También es normal crear subcarpetas para procesos o funciones de uso general en la aplicación.



¿Cómo organizar los objetos del proyecto de datos dentro del módulo?

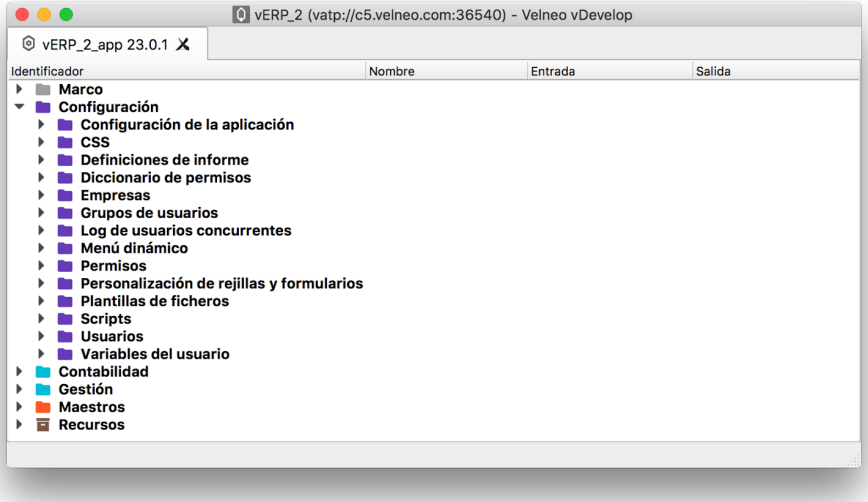
La organización de carpetas del módulo de aplicación es similar en la parte de módulos, a partir de ahí las subcarpetas siguen un criterio orientado a organizar los objetos teniendo presente que se usan para la interfaz de la aplicación.

En caso de que el proyecto contenga el objeto AUTOEXEC es recomendable poner la carpeta de Marco (icono Marco) la primera.

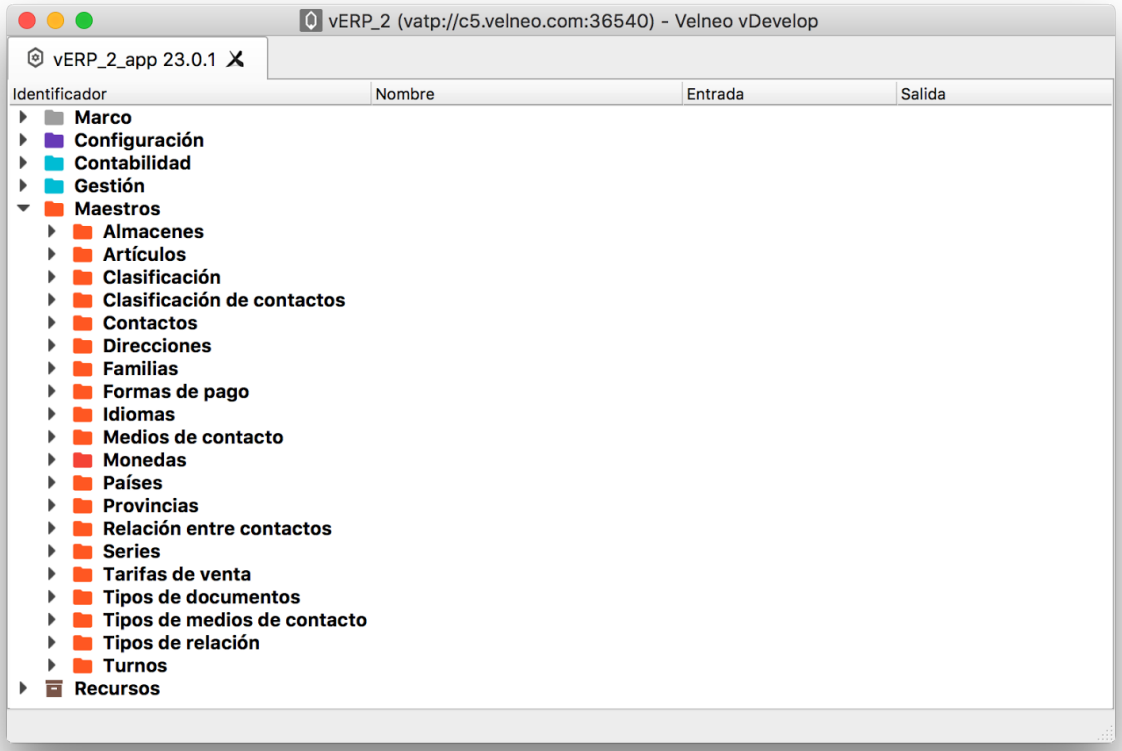


Esta carpeta contendrá diferentes subcarpetas organizadas por orden alfabético para contener objetos relaciones con los proceso arranque, barra de menú, docks, formulario principal y menú general. En definitiva objetos relaciones con el marco general de la aplicación.

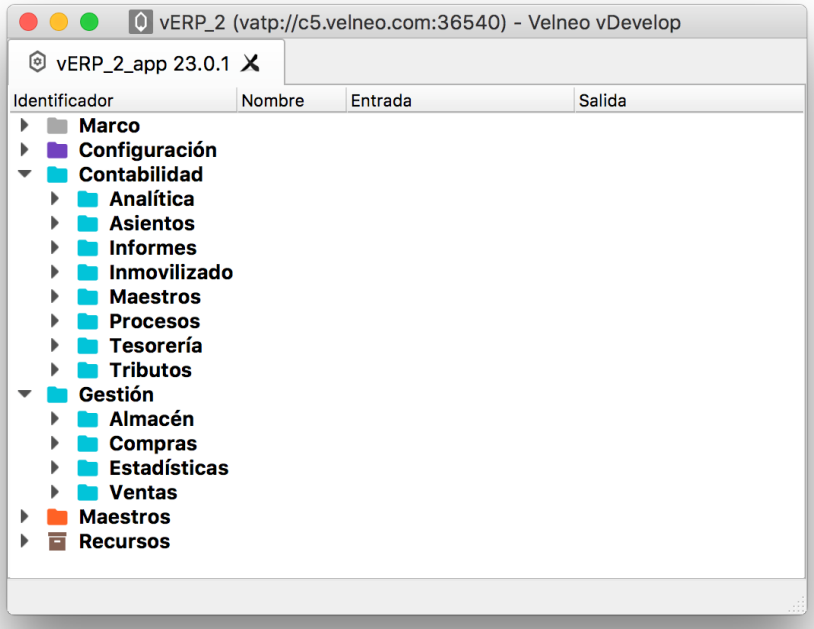
Cada de uno de los módulos dispondrá de una carpeta general y en su interior pueden darse 2 casos: Crear subcarpetas con submódulos o crear subcarpetas por tabla. Este segundo caso se da con las tablas de configuración (icono Objetos 5)



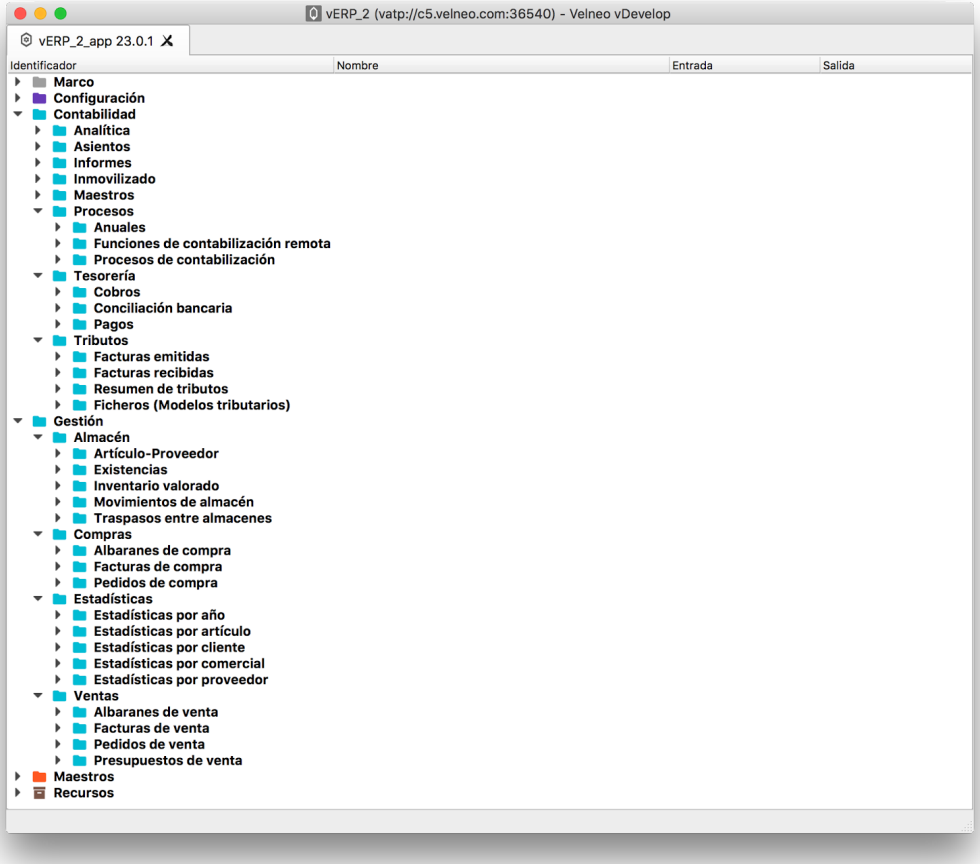
y con las tablas maestras (icono Objetos 3) comunes para todos los módulos.



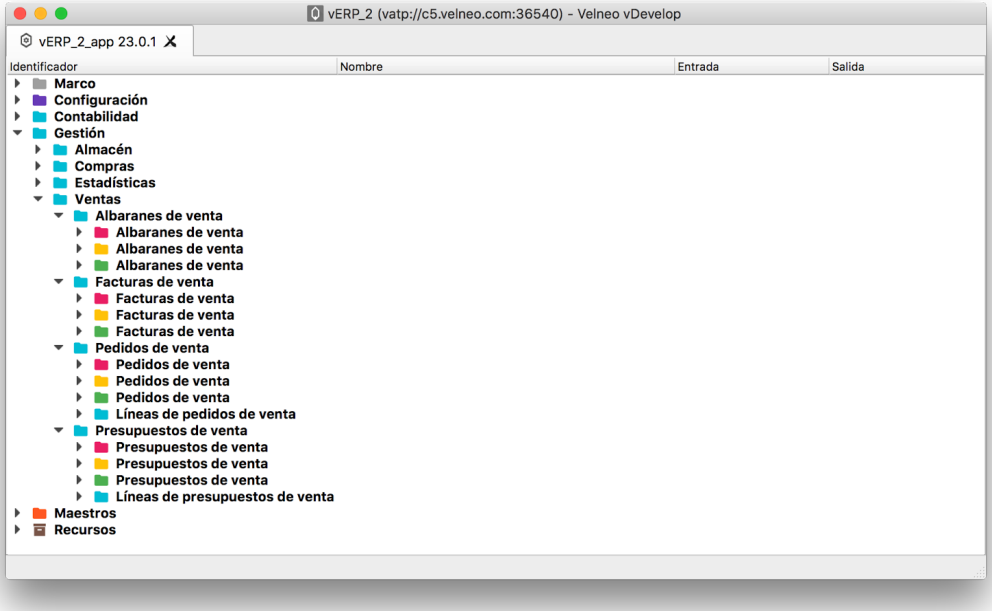
En las carpetas de módulos (icono Objetos 1) suelen crearse unas subcarpetas para organizar mejor funcionalmente las tablas.



Dentro de cada subcarpeta suelen estar las subcarpetas (icono Objetos 1) relativas a las diferentes tablas organizadas por orden alfabético.



Dentro de cada carpeta de tabla se aplica el criterio de organización que denominados “semáforo” por la coincidencia en los colores y orden con el objeto físico.



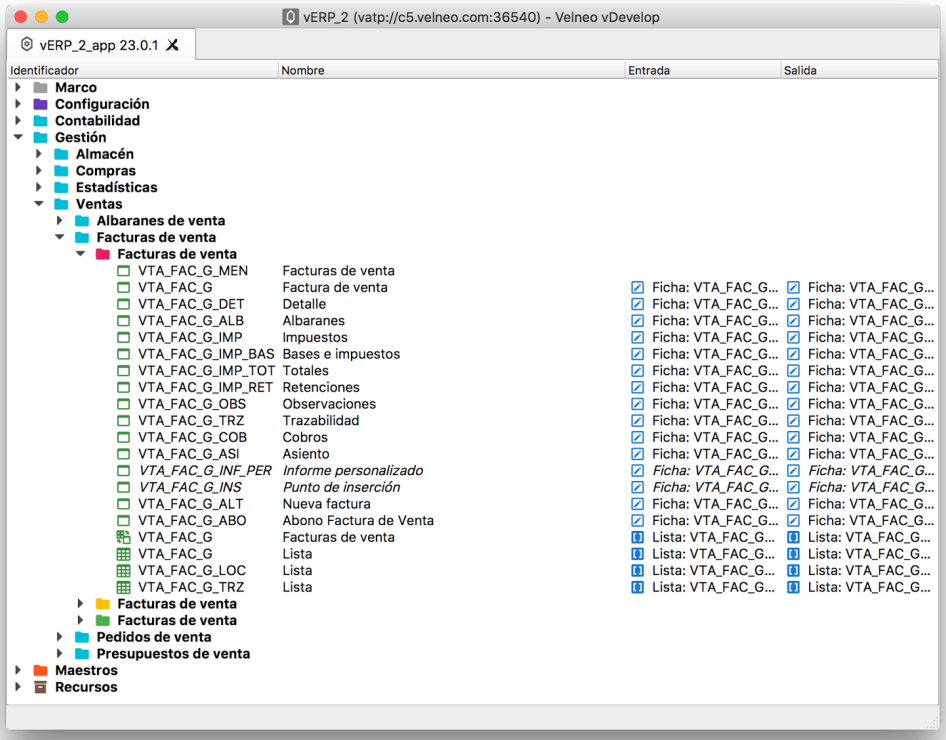
También podemos apreciar como dentro de una tabla se pueden organizar subcarpetas de tablas relacionadas (icono Objetos 1) como en el caso de las líneas de detalle de pedidos y presupuestos que están organizadas dentro de la carpeta de sus cabecera de documento correspondientes.

Usa la técnica del semáforo para organizar los objetos de interfaz de una tabla

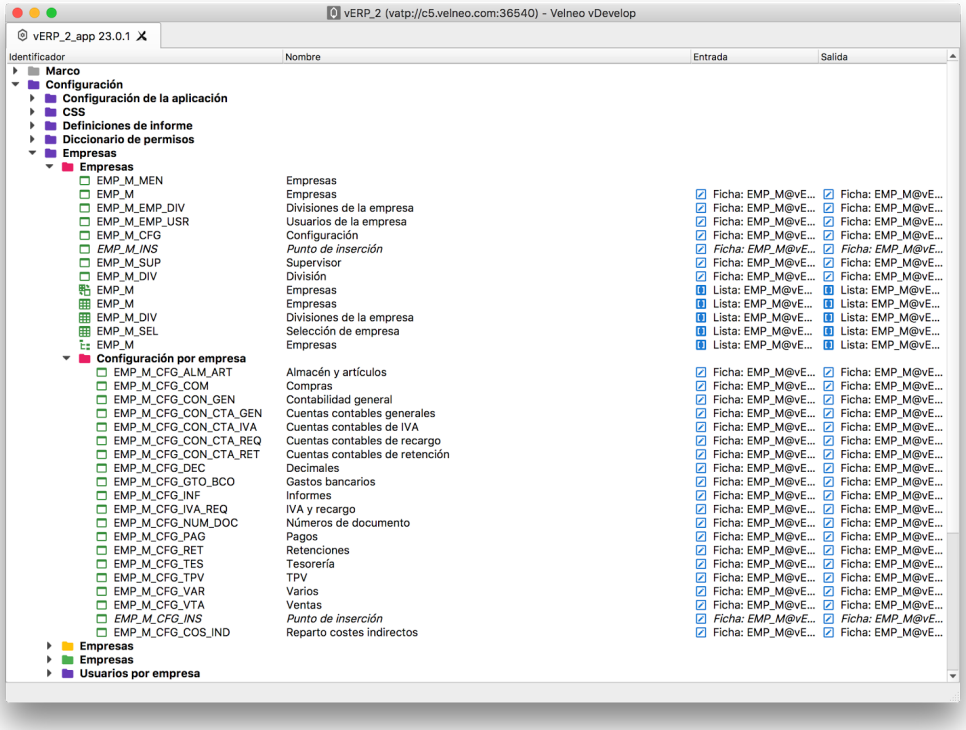
Por cada tabla es habitual tener que crear las 3 carpetas que describiremos a continuación, aunque en algunos casos puede ocurrir que solo tengamos que crear una o dos de ellas. Dentro los objetos se agrupan por tipo y dentro de cada tipo por orden alfabético del identificador salvo en algunas excepciones que se especifican.

En la carpeta roja o de interfaz (icono Interfaz) incluiremos todos los objetos que tiene que ver con la interfaz organizados de la siguiente forma:

- Menú.
- Formularios principales de edición. El formulario de edición irá en segundo lugar. En muchos casos se utiliza un único formulario para alta baja y modificación. En caso de tener formulario independientes podremos ubicarlos juntos por orden alfabético.
- Subformularios. Detrás de cada formulario se ubicarán los subformularios en el mismo orden en que están incluidos en el objeto separador de formularios, facilitando así su localización y edición.
- Formularios específicos.
- Formularios QML.
- Alternadores de lista.
- Multivistas.
- Rejillas.
- Rejillas avanzadas.
- Árboles visor de tablas.
- Casilleros.
- ComboViews.
- ListViews.
- ViewFlows.
- Listas QML.
- Gráficos.
- Informes.
- Esquemas con objetos visuales asociados.

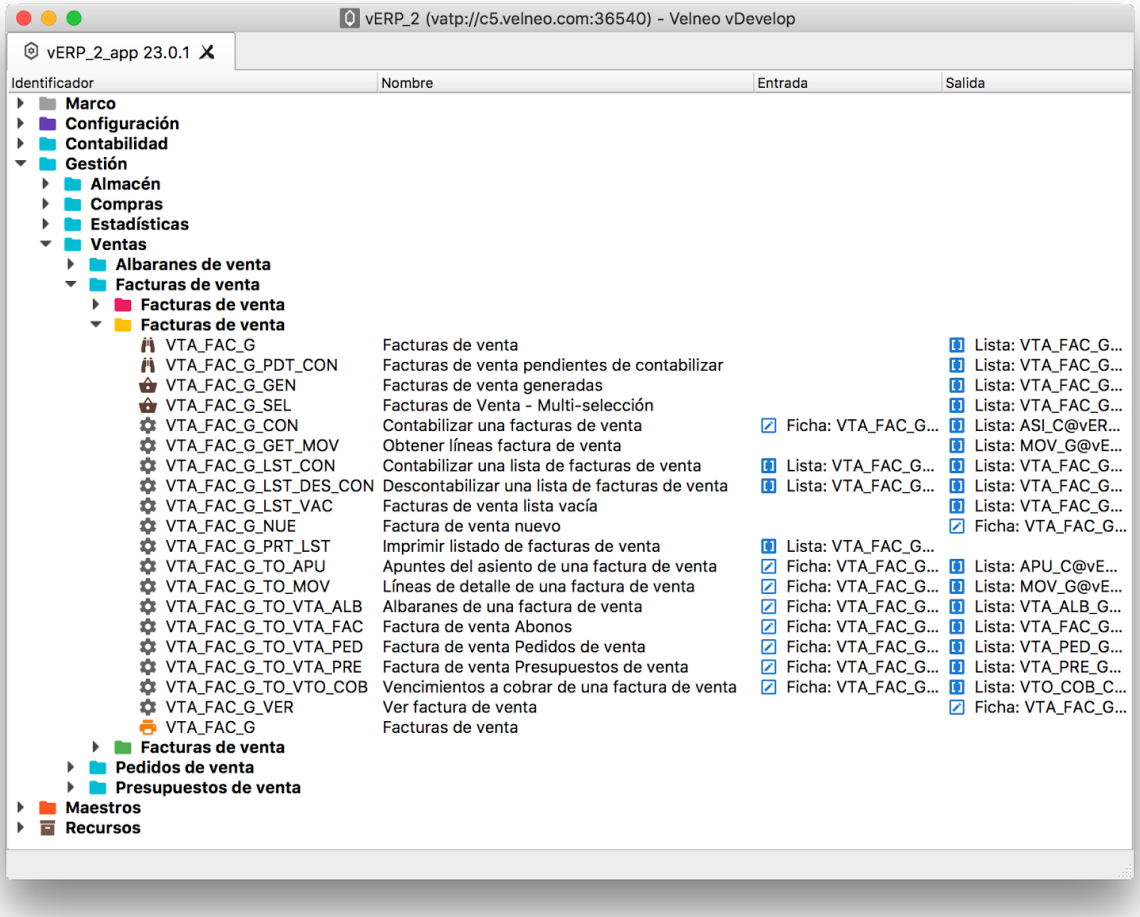


En caso de que existiesen muchos objetos de un determinado tipo podemos crear subcarpetas para organizarlas mejor, normalmente estas subcarpetas tendrán un nombre específico ya que los objetos estarán relacionados con alguna funcionalidad. En la imagen vemos un ejemplo de como se han agrupado en una subcarpeta todos los subformularios de configuración.



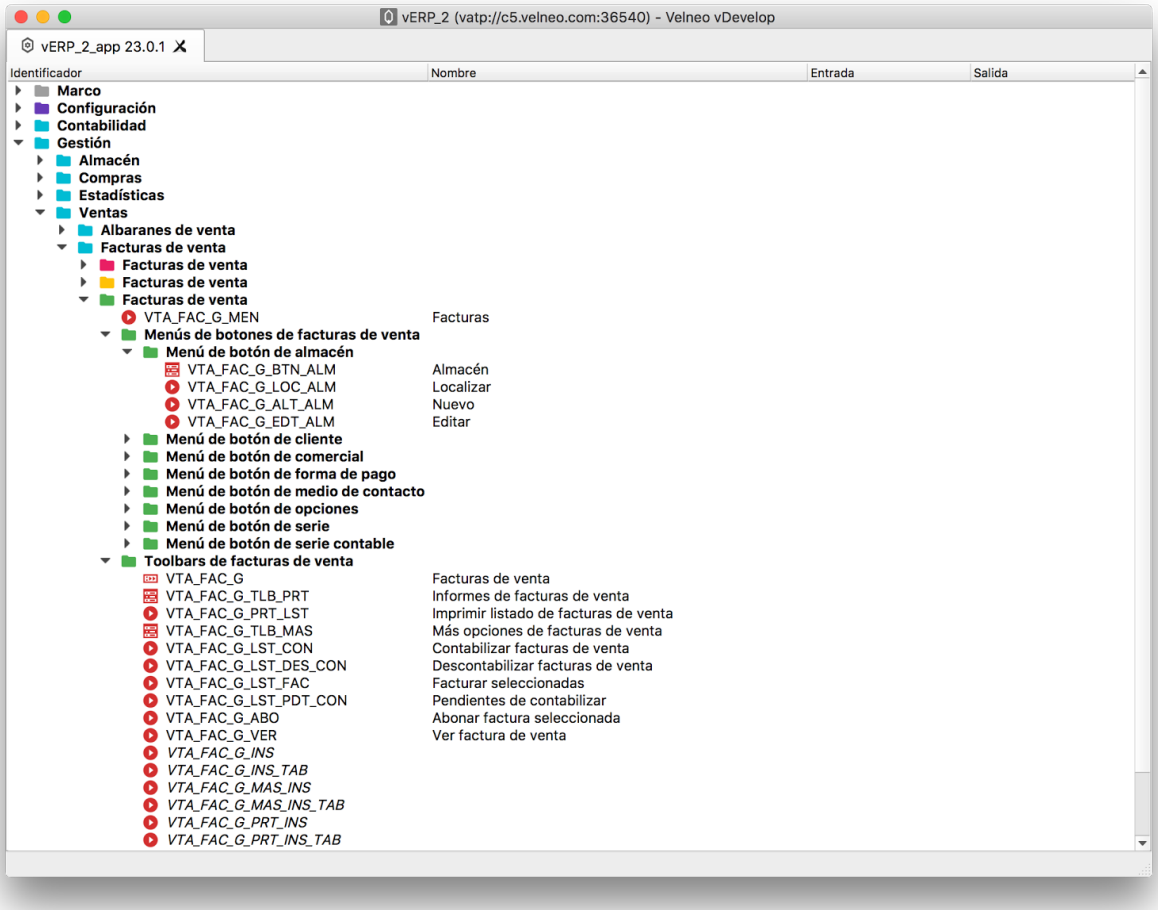
En la carpeta amarilla (icono General) incluiremos los objetos de ejecución que no tienen interfaz, se ubicarán por orden alfabético dentro de cada tipo. El orden de organización será el siguiente:

- Búsquedas.
- Localizadores.
- Lupas.
- Cestas.
- Procesos.
- Funciones.
- Tubos de ficha.
- Tubos de lista.
- Colas.
- Impresoras lógicas.
- Protocolos TCP/IP.
- Dispositivos serie.
- Librerías externas.
- Ficheros adjuntos.



En la carpeta verde (icono Acciones y Menús) incluiremos las acciones, menús y barras de herramientas asociadas a la tabla. El orden de objetos es el siguiente:

- Acción para ejecutar el menú.
- Subcarpeta general para todos los menús de botón.
    - Subcarpeta por cada menú de botón.
        - Menú.
        - Acciones incluidas en el menú.
- Subcarpeta con las toolbars
    - Toolbar.
    - Menú.
    - Acciones usadas en la toolbar o los menús.
    - Acciones para puntos de inserción.



Puntos de inserción en todas las toolbars y menús

Si estamos desarrollando un módulo que sirva de núcleo para nuestros desarrollos o un aplicación estándar que puede ser heredada por otras personalizaciones para sectores o clientes es importante añadir en todas las toolbars y menús un punto de inserción sin origen y en caso de que sea para una tabla añadir un segundo punto de inserción con el origen de la tabla.

Identificadores

Los identificadores son una pieza clave en el desarrollo de aplicaciones Velneo. Es el elemento que más vamos a utilizar en nuestros desarrollos para hacer referencia a cada objeto, subobjeto o control.

Identificadores cortos y descriptivos

El tamaño del identificador es muy importante no solo por su legibilidad sino porque es el código o referencia que se utilizará en el resto de puntos de la aplicación para ejecutar el objeto, subobjeto o hacer referencia a un control. Por lo tanto aquí el tamaño sí importa ya que si utilizamos identificadores muy largos el tamaño de nuestros proyectos pueden incrementarse notablemente.

¿Por qué usar abreviaturas?

Hay que tener en cuenta que un simple campo puede ser usado cientos de veces. Si su identificador tiene un tamaño de 50 caracteres estamos hablando de una ocupación de varios miles de bytes. Para reducir el tamaño de los identificadores podemos usar abreviaturas que nos aportarán 2 grandes beneficios:

1. Reducción del tamaño que nos beneficiará en el tamaño de los proyectos.
2. En el árbol de propiedades poder ver completo el identificador del objeto usado.

¿Por qué conviene usar un diccionario de abreviaturas?

Evidentemente, las abreviaturas también tienen una desventaja y es que debemos de interpretar la abreviatura para conocer la palabra a la que sustituye. Para facilitar esta labor es conveniente utilizar un diccionario de abreviaturas.

Además, el diccionario nos aporta otra gran ventaja, como es el conseguir que todos los programadores escribamos igual los identificadores. Precisamente, cuando no se usa un diccionario de términos o abreviaturas es habitual que cada programador escriba la misma palabra de una forma distinta, por ejemplo: *FACTURAS*, *FACTURA*, *FACT*, *FRA*, etc.

En definitiva, un equipo de desarrollo debe contar con un diccionario de abreviaturas que puede estar almacenado en una aplicación, una hoja de cálculo o un documento de texto, es muy importante que cuente con un sistema de búsqueda ágil. Lo fundamental para el equipo es que exista, que se utilice y que se mantenga actualizado. Es recomendable que al lado de la abreviatura se indique todos los términos que la utilizarán.

En el diccionario se puede aplicar por convenio el uso de palabras en un solo idioma, todo en Español, todo en Inglés o también ser menos estricto y utilizar palabras en su mayoría en tu idioma nativo permitiendo alguna excepción cuando aporte legibilidad y reducción de tamaño.

¿Por qué abreviaturas de 3 caracteres?

3 es un número que nos permite una gran combinación de caracteres alfanuméricos con un tamaño muy reducido.

Un problema que nos podemos encontrar con el uso de abreviaturas de 3 caracteres es la coincidencia de varios términos, por ejemplo IMP se puede usar para los términos “importe”, “importar” y también

“imprimir”. En muchos casos el contexto facilita la interpretación del significado, es decir si la abreviatura se escribiese sola, por ejemplo un campo IMP si está en una tabla de líneas de detalle es fácil interpretarlo como importe antes que importación o impresión. Sin embargo, en muchos casos los identificadores son compuestos de varias abreviaturas, de esta forma *IMP_TOT* es fácil interpretarlo como importe total, *SND_IMP* como senda de importación. Incluso para evitar estas coincidencias se pueden usar alteraciones o abreviaturas estándar del mercado, por ejemplo PRT (print) para imprimir. De esta forma si vemos *VTA_FAC_PRT* lo interpretaremos como impresión de la factura de venta.

Es cierto que con 4 caracteres sería más fácil evitar algunas coincidencias como las comentadas anteriormente, sin embargo la longitud de los campos se dispararía, incluso algunas abreviaturas serían más complejas de elaborar ya que cuantos más caracteres más decisiones hay que utilizar para la combinación de consonantes y vocales. En el ejemplo anterior *VNTA_FACT_PRNT* sería el mismo identificador de impresión de facturas de venta con abreviaturas de 4 caracteres, como podemos apreciar hay palabras difíciles de abreviar como venta que se podría haber abreviado como *VENT*, *VETA*, *VNTA* o *VTAS* no siendo ninguna de ellas demasiado satisfactoria.

Recordar 3 abreviaturas de 3 caracteres es más sencillo que de 4 ó más. Además, a medida que vamos desarrollando las aplicaciones nos daremos cuenta de que se van construyendo objetos cuyo identificador es cada vez más y más largo para poder expresar de forma concreta y única la funcionalidad del mismo. Por ejemplo para el formulario de detalle de una línea de pedido de venta podríamos encontrarnos con estas posibilidades:

Tipo	Identificador
Sin abreviar	*VENTA_PEDIDO_LINEA_DETALLE*
Abreviatura de 4	*VNTA_PEDI_LINE_DETA*
Abreviatura de 3	*VTA_PED_LIN_DET*

Ahora imagínate como se llamaría un tubo de ficha que genera una línea de factura de venta a partir de una línea de pedido.

Tipo	Identificador
Sin abreviar	*VENTA_PEDIDO_LINEA_TO_VENTA_FACTURA_LINEA*
Abreviatura de 4	*VNTA_PEDI_LINE_DETA_TO_VNTA_FACT_LINE*
Abreviatura de 3	*VTA_PED_LIN_DET_TO_VTA_FAC_LIN*

Aplica cada uno de los tipos a todos los identificadores de tu aplicación y podrás comprobar como te encontraras con objetos cuyos identificadores son extra largos. Sin duda las abreviaturas son un magnífico recurso para reducir el tamaño y facilitar la legibilidad.
Por último indicar que cuando se establece un máximo de 3 caracteres en las abreviaturas no implica que todas deban tener ese tamaño, también se admiten abreviaturas de menor tamaño como por ejemplo “A”, “OK”.

Evita el uso de preposiciones y conjunciones

Estas palabras no deben ser usadas cuando no aportan valor semántico significativo, algo que ocurre en la mayoría de los casos.

Utiliza el guión bajo como separador de abreviaturas

Velneo no permite el uso de espacios en blanco ni caracteres especiales en los identificadores, por este motivo esos caracteres son sustituidos de forma automática por el guión bajo "_". Para facilitar la legibilidad de los identificadores usaremos el separador entre cada abreviatura.

Tipo	Identificador
Sin separador, difícil de leer	*VTAPEDLINDETTOVTAFACLIN*
Con separador, más fácil de leer	*VTA_PED_LIN_DET_TO_VTA_FAC_LIN*

No uses como sufijo de los identificadores el tipo de objeto

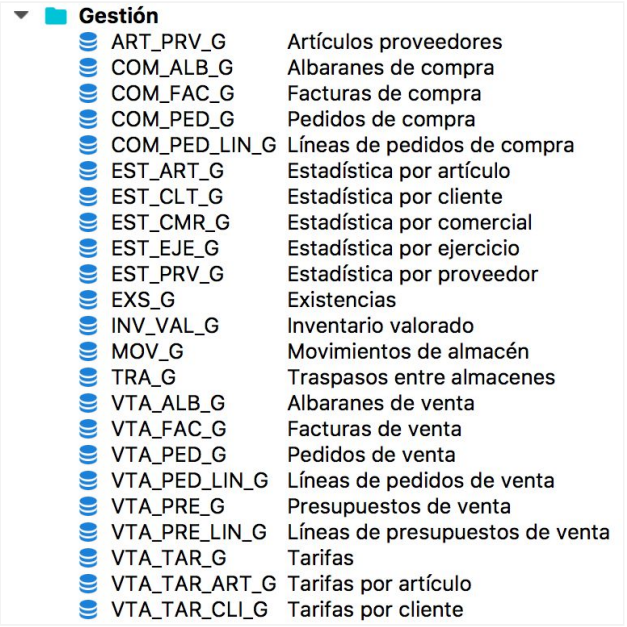
Hacerlo tiene 2 desventajas. La primera es aumentar el tamaño del identificador y la segunda es que estarás aplicando una información redundante ya que el tipo de objeto está representado por su icono y en la ventana de propiedad se indica el nombre del tipo de objeto.

Por lo tanto debemos evitar usar identificadores como *ACC_VTA_FAC_G_MEN* ya que como vemos además de ser más largo la información que aporta es redundante, e incluso lo más probable es que allí donde se use tan solo podríamos utilizar objetos de este tipo.

Usa el identificador de la tabla como prefijo de los objetos con ese origen

Una de las características de los objetos en Velneo es que disponen de origen y destino (ninguno, ficha o lista), por este motivo es muy importante poder identificar el origen de cada objeto sin necesidad de revisar en su propiedad el origen.

Por este motivo es importante que los identificadores de las tablas sean a la vez descriptivos y lo más cortos posible.

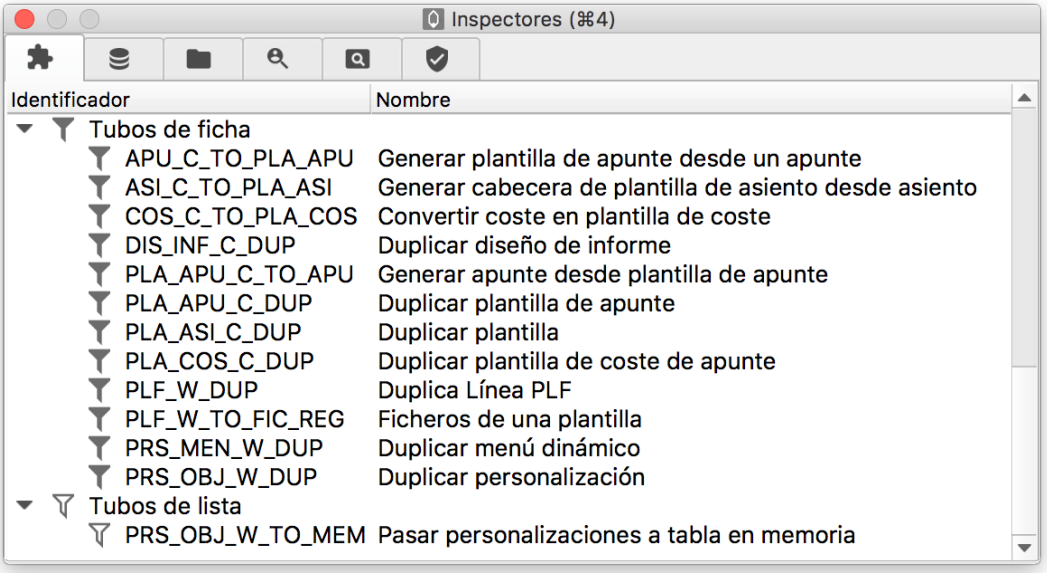


Aplicando el diccionario conseguimos tablas con identificadores de 1 abreviatura y otras 2 y hasta 3 abreviaturas de 3 caracteres. En general no conviene sobrepasar las 3 abreviaturas ya que acabaríamos teniendo identificadores demasiado largos.

Es habitual que haya tablas relacionadas bien por su funcionalidad o porque pertenecen al mismo submódulo, como por ejemplo "*COM*" para compras y "*VTA*" para ventas. En estos casos es conveniente que el dato "común" o agrupador sea el de más peso y se use como prefijo, en nuestro ejemplo es correcto poner *VTA_PED* para pedido de venta en lugar de *PED_VTA*. De esta forma conseguimos que alfabéticamente las tablas del mismo submódulo estén juntas. Si no aplicamos este criterio el orden alfabético producirá una organización más caótica.

Usa identificadores que combinen origen y destino para tubos y procesos

Existen objetos en los que es muy importante tanto su origen como su destino. Un caso claro son los tubos de ficha y lista. En estos objetos es conveniente que el identificador incluya ambas tablas.



En el ejemplo podemos apreciar como cuando la tabla de origen y destino son diferentes se separan con la abreviatura *TO*. Es cierto que está en inglés, pero es una abreviatura corta y fácil de leer e interpretar.

Podemos apreciar que cuando la tabla de origen y destino es la misma se está aplicando en este caso el sufijo *_DUP* para indicar que el objeto creará un duplicado del registro de origen.

En el último ejemplo el sufijo es *_MEM*, esto se utiliza para indicar que se generarán los registros de origen en la misma tabla de destino pero en memoria, en lugar de repetir el identificador completo de la tabla *PRS_OBJ_W_MEM* se utiliza solo el sufijo diferencial. Estos casos son bastante excepcionales por lo que si se aplica la norma aunque el identificador sería mucho más largo *PRS_OBJ_W_TO_PRS_OBJ_W_MEM* sigue siendo igual de válido.

Usa sufijos en los identificadores de las tablas, tablas estáticas y variables globales

El editor no permite que dos tablas tengan el mismo identificador en el mismo proyecto, pero sí es posible crear dos tablas con el mismo identificador en distintos proyectos. Cuando se trabaja con múltiples soluciones o múltiples proyectos heredados o incluso cuando se trabaja sobre un núcleo común a todas las aplicaciones hay que tener especial cuidado en conseguir que no se repita el identificador de una tabla.

El problema se produce cuando al instanciar ambos proyectos se realiza sobre la misma carpeta del disco produciéndose un conflicto al solo existir una tabla que tiene dos definiciones de estructura diferentes en los proyectos.

Para evitar esta duplicidad de identificadores es conveniente usar un sufijo diferenciador que permita poner identificadores a las tablas sin riesgo de caer en la duplicidad. Conviene que esos sufijos tampoco se repitan. Se puede utilizar el criterio de un sufijo diferente por aplicación, módulo, etc.

Como el número de aplicaciones o módulos no suele ser muy alto, se pueden utilizar sufijos con una sola letra, por ejemplo: "*_G*" para gestión, "*_C*" para contabilidad, "*_M*" para maestros generales, "*_W*" para configuración, etc. En caso que el nº de aplicaciones o módulos sea muy grande se pueden colocar 2 ó más letras.

Para mantener un criterio único, se aplicará el mismo criterio de las tablas a las tablas estáticas y también a las variables globales que tengan una relación directa con un módulo.

No uses el sufijo de la tabla en los identificadores de campos e índices

Aunque las tablas tengan un sufijo y cuando añadimos campos a una tabla se crean con el mismo identificador de la tabla tanto el campo como el índice correspondiente. Por mejorar la legibilidad de los subobjetos de la tabla: campos, índices y actualizaciones, quitaremos del identificador el sufijo correspondiente.

Por ejemplo, si la tabla de artículos tiene como identificador "*ART_M*", las diferentes tablas de líneas de detalle de compras y ventas tendrán un puntero al artículo cuyo campo, índice o actualización tendrá como identificador "*ART*" en lugar de "*ART_M*".

Excepciones para que los campos punteros a tabla maestra no usen su mismo identificador

Por regla general coincidirá el identificador del campo con el de la tabla o tabla estática apuntada. Es decir, el campo puntero al artículo se identificará como "*ART*" ya que su tabla maestra se identifica como

“*ART_M*”.

Sin embargo, se pueden dar circunstancias que no permitan usar el identificador exacto de la tabla:

Si en una misma tabla existen varios punteros a la misma tabla maestra, es lógico que el identificador sea más explícito, y por lo tanto diferente al de la tabla maestra. Por ejemplo si una entidad puede tener forma de pago para cobros y forma de pago para pagos, si la tabla de formas de pago se identifica como “*FPG_M*”, los campos podrían identificarse como “*FPG_COB*” y “*FPG_PAG*” respectivamente.

En ocasiones hay tablas que contienen múltiples tipos de registros, por ejemplo el caso de la tabla de entidades o contactos que puede servir para almacenar diferentes tipos de registros como clientes, proveedores, comerciales, etc. En estos caso se podrían utilizar los siguientes identificadores en la tabla de factura de venta:

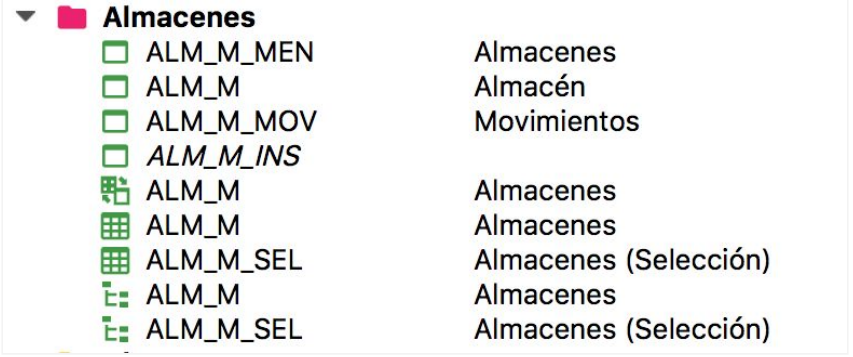
Tipo	Identificador	
ENT	Cliente	Desaconsejable si en la tabla pueden existir otros campos punteros a la entidad como el comercial.
ENT_CLT	Cliente	Es válido ya que permite que el comercial tenga como identificador *ENT_CMR*.
CLT	Cliente	Este es el identificador más corto, pero además es el más explícito ya que indica el uso del dato y no el origen de la tabla. Para campos de uso masivo como el de clientes, proveedores, etc. Este identificador puede ser el más conveniente.

En cualquier caso debe existir un consenso en el equipo de cuál de los 2 últimos utilizar.

No te preocupes por los identificadores repetidos en el proyecto

Es cierto que si miramos los identificadores de la carpeta de una tabla encontraremos muchas repeticiones. Sin embargo, esto es algo permitido por el editor de Velneo ya que el identificador “completo” de un objeto viene dado por: El proyecto + el tipo de objeto + el identificador.

De esta forma para un mismo proyecto podemos tener objetos con el mismo identificador siempre que sean de diferente tipo. Esto nos permite utilizar el mismo criterio para todo los objetos sin necesidad de recurrir a un prefijo o sufijo que lo indique el tipo de objeto.



Base de datos

La pieza más importante en el análisis de una aplicación es sin duda la base de datos. Podríamos afirmar que un buen diseño de base de datos garantiza rendimiento y mantenibilidad mientras que un mal diseño nos garantiza problemas que se irán agravando con el paso del tiempo.

Demos tratar de diseñar nuestra base de datos con la mayor sencillez posible, de lo contrario cualquier corrección, mejora o evolución se convierte en una tarea compleja y por lo tanto mucho más costosa. A continuación vamos a ver algunas buenas prácticas a la hora de diseñar la estructura de base de datos de nuestra aplicación.

Una base de datos, un responsable

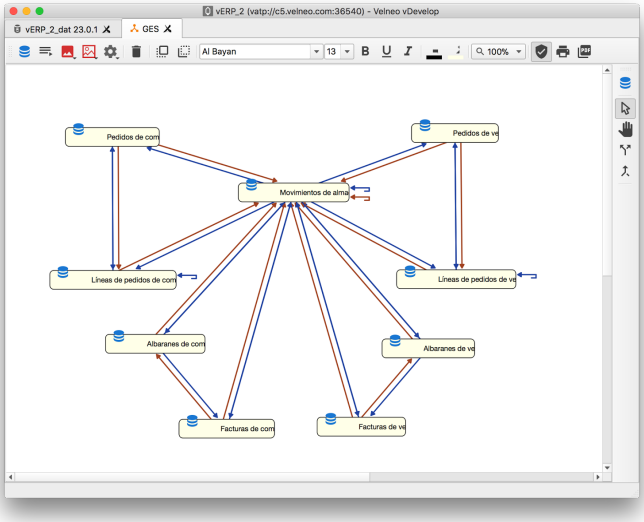
Dada la importancia de la base de datos es fundamental que esté bajo la tutela de un único responsable. Esto no significa que solo una persona pueda hacer cambios, que también puede ser una buena práctica, sino que no debería de realizarse ningún cambio en la base de datos sin que el responsable esté informado y valide dicho cambio. Ya que de no hacerse así corremos el riesgo de que la base de datos contenga campos que ya no se utilizan pero que nadie se atreve a borrar, índices duplicados en tablas muy grandes donde es más difícil controlar todo lo que ya existe, etc. En definitiva, cada base de datos debe tener un responsable único.

Esquemas

Crea esquemas para documentar las tablas

Cuando comenzamos a desarrollar una aplicación lo haremos desarrollando la estructura de base de datos, para realizar esa tarea es recomendable crear un objeto esquema que nos permitirá crear las tablas de forma visual y además dejarlo documentado para en el futuro poder recordar de un vistazo las relaciones entre las diferentes tablas. El objeto esquema hace bueno el dicho “Una imagen vale más que mil palabras”.

Conviene crear las tabla directamente desde el esquema ya que además de crear el objeto ya lo dejamos incrustado en el esquema lo que nos facilitará crear las relaciones de forma visual.



Crea múltiples esquemas

A medida que vamos añadiendo tablas a nuestro proyecto conviene crear múltiples esquemas con el doble objetivo de evitar tener un esquema con tantas tablas y relaciones que resulta muy complicado ver la estructura y entenderla, y por otro lado nos permite tener esquemas específicos con la estructura de sub módulos o funcionalidades específicas, consiguiendo que otros desarrolladores entienda la estructura de tablas y sus relaciones rápidamente.

Número de tablas y tamaño de registros

¿El número de tablas influye en el rendimiento?

Salvo que estemos hablando de miles de tablas, en cuyo caso podría afectarnos en el tiempo de reinicio de la instancia en el servidor, a nivel de ejecución hay que tener en cuenta que nos afectan las tablas en uso, no las declaradas en el proyecto.

¿El tamaño de registro de una tabla como influye?

Influye en el tamaño de las transacciones, en el nº de conexiones que un cliente tiene que establecer con el servidor para obtener los datos de una lista de registros y en los tiempos de regeneración de la tabla ante un cambio de estructura.

Por lo tanto debemos intentar reducir el tamaño de registro de una tabla en la medida de lo posible. Unas buenas prácticas podrían ser:

- Evita crear campos que no se usan.
- Si se necesitan campos alfabéticos muy largos (>100 caracteres) que se usan en un porcentaje bajo de registros, puede ser más óptimo crearlo de tipo objeto texto, de esta forma en el registro ocupa 8 bytes y en el contenedor las celdas que necesite de 512 bytes.
- Evita la información repetida, por ejemplo, intenta no duplicar el nombre de los artículos en las líneas de movimientos.
- Extrema el tamaño de registros en las tablas que vayan a contener millones de registros.

¿Es mejor tener muchas tablas con un único tipo de registro o es mejor tener una única tabla con múltiples tipos de registro?

Si el número de registros no es elevado, es decir no contendrá la tabla millones de registros) será más cómodo crear una única tabla con un campo que identifique el tipo de registro.

El campo de tipo de registro para que esté bien documentado debería apuntar a una tabla estática evitando así tener que documentar los valores del mismo por los diferentes lugares de la aplicación donde se use.

Otro factor a tener en cuenta es el número de índices que se van a crear. Si por ejemplo vamos a crear los típicos índices de código, nombre, palabras y trozos y tenemos 5 tipos de registros vamos a crear 20 índices. Este valor no es un problema, pero si vamos a tener que crear un número alto de índices (>200) tal vez debemos replantearnos el uso de tablas independientes.

Hay que tener en cuenta que aunque esta tabla tenga muchos índices, estarán condicionados de tal forma que en cada índice solo encontraremos los registros de un determinado tipo.

Tipos de tablas

¿Cuándo es conveniente usar una tabla de tipo maestro arbolada?

Cuando tengamos que representar los registros de dicha tabla en un árbol en el que cada nivel representa una relación de herencia entre los registros.

Ejemplo habituales son el plan de cuenta de contabilidad, clasificaciones de familias y subfamilias, etc.

¿Qué tamaño de campo ID debo usar en una tabla arbolada?

El menor posible que te permita contener el mayor código que se necesita grabar. Es decir, debemos evitar poner un campo código en el que dejemos un tamaño mayor “por si acaso”.

Hay que tener en cuenta que el tamaño del ID influye en los índices y también en el tamaño de los campos que apuntan a esta tabla como maestra. Por lo tanto afecta al rendimiento, cuanto menor sea el código más rápido se manejará la tabla.

Por este motivo se suelen usar campos alfabéticos “comprimidos” como el *Alfa 64* (Ahorro del 25%) y *Alfa 40* (Ahorro del 33%) que nos permiten contener más caracteres en el código con una menor ocupación en disco.

¿Cuándo es conveniente usar tablas de tipo histórico?

Cuando se den las 2 siguientes circunstancias:

1. La tabla no tiene un código único que identifique al registro sino que almacena información que no está codificada, o tiene múltiples maestros relacionados, todos ellos con el mismo peso.
2. La tabla nunca será maestra de otra tabla plural. Esto se debe tener en cuenta para que en el caso de que exista un plural no nos veamos obligados a incluir múltiples campos punteros para resolver la clave única que se haya creada en esta tabla histórica.

¿Y si creo siempre todas las tablas maestras?

Es cierto, que evitar crear tablas históricas y en su lugar crearlas siempre como maestras nos evita la 2ª circunstancia de la pregunta anterior, y es cierto que en la mayoría de los casos nos servirá aplicar este criterio de todas maestros.

Sin embargo, existen algunas excepciones que debemos tener en cuenta.

Si tengo que apuntar a una tabla con punteros indirectos reales usando campos para resolver el índice de clave que no es el *ID*, no me sirve de nada que la tabla sea maestra, al contrario me obliga a mantener un campo y un índice innecesarios.

Si creamos la tabla como maestra y apuntamos a ella a través del código (*ID*) obteniéndolo con una búsqueda por otro de sus índices de clave única compuesto por uno o varios campos que no son el *ID*, tendré problemas de refactorización de datos en el caso de que cambien los campos que componen el índice de clave, esto me obligaría a tener que programar el control del cambio de valor de dichos campos. Por ese motivo es preferible apuntar los registros de este tipo de tablas con punteros indirectos que reaccionan automáticamente al cambio de valores con los que se resuelve el puntero al índice de clave única.

Por ejemplo, una tabla de estadística cuyo índice de clave única viene dado por los campos empresa, año, mes y cliente, deberíamos apuntar desde la tabla que actualiza sus datos a través de un puntero indirecto resuelto con campos de la tabla. En este caso definir la tabla como histórica puede ser una buena práctica.

¿Cuándo es conveniente usar tablas de extensión?

Esta tipo de tabla debemos verla siempre como una solución a un problema que no tenga otras alternativas y crearla solo cuando no nos quede más remedio.

Los casos más habituales son:

1. Tengo que añadir campos a una tabla que está en un proyecto (núcleo estándar) que no puedo o quiero modificar porque cuando se vuelva a actualizar estaría obligado a repetir los cambios. En general son personalizaciones para un cliente concreto sobre una tabla estándar para todos mis clientes o los de un sector.
2. Cuando tengo una tabla con cientos de miles o millones de registros y hay un grupo de campos que se usan en un % bajo de registros (<20%) y que hacen crecer el tamaño del registro de forma significativa, por ejemplo pasamos de un tamaño de registros de 400 bytes a 3.000 bytes.

¿Por qué hay que evitarla en la medida de lo posible? Fundamentalmente para aplicar el principio de sencillez que facilite su desarrollo y posterior mantenibilidad. Pero no debemos sacar la conclusión de que no debemos usarla, simplemente usarla con rigor y en los casos en los que sea estrictamente necesario.

¿Cuándo es conveniente usar tablas submaestras?

Como su nombre indica es conveniente cuando una tabla tiene una dependencia directa de una tabla maestra, de tal forma que podemos asegurar que no tiene sentido que exista un registro en la tabla submaestra sin que exista previamente el registro de la maestra.

Un caso típico de esta tabla son las tablas de líneas de detalle de las tablas de documentos de compra y ventas.

La ventaja de declarar esta tabla es que el índice *ID* está formado por el código de la maestra y el código numérico de la submaestra que se numera automáticamente. Si el código de la submaestra no es numérico entonces perdemos esta ventaja y no merece la pena hacerla submaestra.

El otro motivo por el que se desaconseja su uso es que si esta tabla va a tener plurales es mejor usarla de tipo maestra ya que de lo contrario nos encontraremos que para apuntar a un registro desde otra tabla plural vamos a necesitar mínimo 2 campos (maestro y código).

Por el mismo motivo tampoco es cómodo crear tablas submaestras de múltiples niveles ya que cada vez el índice ID tiene más partes y al relacionar otras tablas con esta submaestra se necesitan tantos campos como partes componen el índice, en cambio con una tabla maestra sabemos que podemos resolver la

relación con un solo campo.

Campos

A continuación se detalla en una tabla el uso recomendado de campos de tipo alfabético.

Tipo de campo	Ejemplo de uso
Alfa 256	Almacenar campos alfanuméricos permitiendo todo tipo de caracteres y longitud específica. Hay que tener en cuenta que aunque soporta hasta 65.535 caracteres no tiene mucho sentido guardar tanta información en un campo de este tipo, siendo recomendable usar campos objeto texto o texto enriquecido en su lugar.
Alfa 128	Para guardar datos alfanuméricos como nombres, direcciones, etc. Hay que tener en cuenta que no soporta todos los caracteres ASCII por lo que debemos evitar usarlo para almacenar URLs o eMails por ejemplo, en ese caso es mejor usar el Alfa 256
Alfa 64	Para guardar datos alfanuméricos en mayúsculas independientemente de como los escriba el usuario. Se usa para guardar textos en mayúsculas, referencias o códigos que usan caracteres no soportados en los Alfa 40.
Alfa 40	Muy recomendable para códigos alfanuméricos, referencias, código de barras, etc. Hay que tener en cuenta que el tamaño mínimo siempre debe ser múltiplo de 2 y que su contenido que se puede grabar en el campo será múltiplo de 3, es decir, que si quiero poner en un formulario un campo para guardar 4 caracteres este tipo de campo puede no ser óptimo.
Alfa Latin1	Cuando el contenido del campo debamos enviarlo a un servicio o software externo que tenga ese requisito de codificación. También podemos resolver la codificación en el momento de la exportación, por este motivo no es habitual usar este tipo de campo.
Alfa UTF-16	Cuando tengamos que incluir contenido en idiomas de doble byte como el Chino. Hay que tener en cuenta que estos campos ocupan el doble que uno normal, por lo tanto no debemos poner los campos de las tablas de este tipo si creemos que en el futuro instalaremos una versión para usuarios en Chino ya que estaremos perjudicando el tamaño de la base de datos y el rendimiento por algo que puede no llegar a concretarse nunca. Es preferible que llegado el momento hagamos el cambio de tipo de los campos ya que la refactorización automática hará que el cambio no produzca ninguna pérdida de datos.

¿Son todos los campos Alfa igual de rápidos?

No, realmente el campo *Alfa 256* es el más rápido en su uso debido a que su contenido se procesa de

forma directa, sin embargo los *Alfa 128*, *Alfa 64* y *Alfa 40* utilizan datos comprimidos a nivel de bits. Realmente el proceso de compresión y descompresión es muy rápido pero a la hora de realizar miles o millones de operaciones con cadenas es preferible el uso de campos Alfa 256.

¿Puedo usar campos de tipo tiempo para acumular horas, minutos y segundos?

Sí, esa es su función, pero debemos tener en cuenta que el campo admite hasta un máximo de 24 horas (un día), por lo tanto si queremos almacenar tiempo superior a un día debemos utilizar un campo numérico donde guardemos los segundos, minutos u horas según el caso y luego utilizar campos para convertir esos tiempos a horas:minutos:segundos.

¿Cuándo debo utilizar campos de tipo fórmula?

Los campos fórmula son muy recomendables para ahorrar espacio en el tamaño de registro, ya que su ocupación en disco es cero.

Debemos tener en cuenta que la fórmula se calcula allí donde se solicita el valor del campo, por ese motivo si vemos que queremos hacer un uso intensivo de ese campo en rejillas o si hay muchas dependencia de contenidos iniciales como puede ocurrir en tablas de estadísticas con acumulados mensuales (saldos, existencias, estadísticas, etc.) podemos optar por utilizar campos con persistencia en disco, que aunque ocupan espacio y aumentan el tamaño del registro, evitan el recálculo del dato ya que se calcula una única vez a través o bien del contenido inicial, de algún evento de tabla o directamente en código del objeto visual, y nos ahorra volver a calcular en el momento en que se muestra el datos en un informe, rejilla, formulario, etc.

¿Cuándo es recomendable usar campos objeto texto?

Los campos objeto tienen una ocupación en el registro de 8 bytes en los que se almacena la referencia al primer bloque de 512 bytes del contenedor de objetos. En el contender todos los contenidos sean texto, imágenes o ficheros se almacenan en celdas de 512 bytes, cuando un objeto ocupa más de 512 bytes va ocupando más celdas de este tamaño hasta almacenarse en su totalidad, todas las celdas de un objeto quedan relacionadas e indexadas para un rápido acceso.

Por lo tanto tenemos que tener claro que si vamos a almacenar un texto con un tamaño fijo menor de 512 bytes, a priori podríamos pensar que un campo alfabético será más recomendable, sin embargo eso dependerá del número de registros que estén ocupados. Si por ejemplo creamos un campo de 100 bytes que solo es ocupado en el 10% de los registros de una tabla que tiene 1.000.000 de registros, estaríamos ocupando 100 MB de disco de los que el 90% estaría vacío. Sin embargo, si usamos un campo objeto tendríamos 8 MB de ocupación de los 8 bytes del campo más 100.000 celdas de 512 bytes lo que daría un total de 520 MB, es decir, la mitad de ocupación en disco más la ventaja de que el registros es 92 bytes más ligero. Aunque no supone ningún inconveniente tenemos que tener claro que la carga de objetos se realiza en hilos secundarios ya que no viaja con la información del registro.

Otra característica muy interesante de la base de datos de Velneo es que puedes indexar por trozos y/o palabras los campos objeto texto y objeto texto enriquecido (en este caso la base de datos se encarga de eliminar las etiquetas HTML de la indexación). Esta es una característica muy potente, aunque hay que gestionarla bien ya que la indexación de textos largos pueden generar millones de entradas en el índice de los contenedores.

Existen diferentes circunstancias en las que el uso de objeto texto nos va a ayudar a optimizar la ocupación de espacio en la base de datos, y por lo tanto el rendimiento en ejecución de nuestra aplicación.

- Cuando la ocupación de registros es baja, por ejemplo <10% para tamaños de >100 bytes.
- Para evitar crear campos alfabéticos muy grandes (>100 bytes).
- Para almacenar contenido variable que puede ser de miles de KB o cientos de MB.
- Para evitar la creación de campos adicionales configurables. Se explica más abajo.
- Para almacenar contenido HTML usa el objeto texto enriquecido.

Si tengo miles de objetos dibujo o texto ¿Los guardo en la base de datos?

Aunque los objetos texto son muy cómodos de usar si lo que queremos almacenar es una gran cantidad de información como puede ser el caso de un gestor de documental en el que podremos almacenar cientos de miles de documentos de gran tamaño, el mejor planteamiento puede ser no utilizar campos objeto y en su lugar almacenar de forma externa los ficheros guardando en un campo del registro la senda o URL de acceso a dicho fichero.

Hacerlo de forma externa nos permite mayor flexibilidad a la hora de almacenar los ficheros clasificados y organizados en disco por carpetas, a la vez que minimiza el tamaño de nuestra base de datos lo que facilita su gestión y reindexación.

El único hándicap en este caso es que perdemos la posibilidad de indexar por trozos o palabras los textos, aunque se pueden usar alternativas como almacenar solo palabras claves en un objeto texto del registro que nos facilite la localización del fichero sin engordar nuestra base de datos con su contenido.

Guarda el contenido de diferentes campos en un solo campo objeto texto

En ocasiones hay aplicaciones muy configurables que permiten a los clientes finales o usuarios añadir campos personalizados en algunas tablas. La base de Velneo es estática en cuanto a su definición, es decir, no podemos cambiar en tiempo de ejecución la estructura de una tabla añadiendo nuevos campos.

Para poder simular los campos personalizables se podría pensar en dejar creados, por ejemplo 3 campos alfabéticos de 50 caracteres, 3 campos numéricos y 3 campos de tipo fecha. Además de poco práctico ya que en un momento dado el usuario podría necesitar 4 campos de un determinado tipo y ninguno del resto supone un gran desperdicio de espacio en disco con múltiples campos vacíos.

En su lugar de puede optar por usar un campo objeto texto en el que almacenemos los contenidos de todos los campos con algún tipo de separador, por ejemplo:

- Formato XML. <nombre campo>Contenido del campo</nombre campo>
- Formato JSON. { “nombre campo” : “contenido campo” }
- Formato CSV. En la primer línea los nombres de campo “nombre campo 1”|”nombre campo 2”|”nombre campo 3” y en la 2ª línea los datos “contenido campo 1”|”contenido campo 2”|”contenido campo 3”
- Salto de línea. El nombre del campo estaría configurado en un campo objeto texto a nivel de aplicación o empresa y el contenido de los se guarda cada uno en una línea añadiendo un salto de línea después del dato.

De esta forma podemos almacenar múltiples valores en un único campo objeto texto. Evidentemente hay un trabajo de programación adicional para poder visualizar esta información de forma dinámica en un formulario o rejilla (usando una tabla en memoria, por ejemplo). Además, la indexación por los valores de

estos campos de forma individual resulta más compleja.

Contenidos iniciales

Los contenidos iniciales de los campos se evalúan cuando damos el alta de un registro y cuando hacemos modificaciones de los datos del registro. Es un gran recurso para el programador y debemos usarlo con cuidado para no abusar de sus bondades perjudicando el rendimiento de nuestra aplicación.

Minimiza las dependencias en contenidos iniciales

Una de las grandes ventajas es que el valor de un campo se calcula automáticamente en base al de otros campos. Esta característica es buena siempre y cuando no abusemos de ella, es decir, si tenemos una tabla con cientos de campos y creamos unos contenidos iniciales muy dependientes entre sí de tal forma que cualquier cambio en un campo produzca el recálculo de muchas decenas o incluso un centenar de contenidos iniciales en otros campos podemos detectar lentitud en nuestra aplicación. Para evitar estos casos excepcionales podemos renunciar al contenido inicial y hacer los cálculos bien en el objeto visual en 1º plano o también en los triggers anterior al alta o modificación, evitando que se recalculen de forma constante y en su lugar conseguir que los cálculos solo se realicen una vez.

Cuidado con los contenidos iniciales que dependen de punteros a hermanos contiguos

Cuando usamos hermanos contiguos en los contenidos iniciales debemos tener la precaución de evitar cálculos en cascada incontrolados. En principio esto no debería de producirse con contenidos iniciales ya que solo afectan al registro en curso, sin embargo sí que tenemos que tenerlo en cuenta si en lugar de un campo con persistencia en disco y contenido inicial usamos un campo fórmula que utiliza un campo obtenido a través de un puntero a un hermano contiguo, ya que en ese caso si el campo del registro apuntado a su vez es una fórmula que tira del hermano contiguo lo que estamos provocando es que el cálculo de un campo realiza lecturas y cálculos en un número de registros incontrolado que puede dar lugar a cálculos de miles de registros.

Para evitar estas circunstancias es preferible usar campos con persistencia en disco y contenidos inciales o cuyo valor se calcula una única vez en un evento de tabla o proceso. Aunque tenemos mayor ocupación en disco a cambio obtener un mejor rendimiento de la aplicación.

Evita el uso de funciones largas o complejas en contenidos inciales

Si tenemos una función que realiza un cálculo complejo que, por ejemplo requiere la lectura de múltiples registros, y usamos esta función en contenidos iniciales de campos debemos revisar que no afecta al rendimiento. Hay que tener en cuenta que los contenidos iniciales aunque están definidos en la tabla no siempre se ejecutan en el servidor, al dar el alta desde un formulario se ejecutan en 1º plano, y en el caso comentado puede producir lentitud en la apertura del formulario o la aparición del icono de espera cuando se esté ejecutando el cálculo del valor del campo.

Si además, a la función se le pasan como parámetros valores de otros campos, podemos encontrarnos con que la función se ejecuta múltiples veces al estar en un contenido inicial, para evitar esta circunstancia debemos evitar su uso en un contenido inicial moviendo la ejecución de la función a los eventos de tabla anterior a alta y modificación, o si la aplicación lo permite directamente en el objeto visual como puede un formulario de edición.

Evita siempre que puedas el uso de contenido inicial JavaScript

En los contenidos iniciales de los campos de una tabla podemos utilizar fórmulas de código Velneo y también fórmulas JavaScript. Debemos saber que cada vez que se ejecuta una fórmula JavaScript es necesario lanzar un motor de ejecución y alimentarlo con las clases generales para que disponga de la información del entorno, aunque esta operación es rápida en términos generales es lo suficientemente lenta como para notar retardo respecto al cálculo de fórmulas de código Velneo, por lo tanto debemos usar fórmulas JavaScript con la precaución de saber que solo se calculará una vez.

Si tenemos varios campos que necesitamos calcular con una fórmula JavaScript podemos optimizarlo no usando la fórmula en el contenido inicial y en su lugar ejecutarla en los eventos de tabla anterior al alta y modificación lanzando un proceso de código JavaScript de origen ficha. De esta forma aseguramos que se ejecute una única vez y además podemos calcular el valor de múltiples campos en el mismo script con lo que optimizar todos los cálculos en una única ejecución del motor de JavaScript.

En las importaciones de millones de registros optimiza el cálculo de contenidos iniciales

Cuando estamos importando miles o incluso millones de registros en las tabla Velneo es habitual que los datos que estamos importando no requieran que se disparen los contenidos iniciales ya que nos llegan calculados.

Con el fin de optimizar la importación, debemos sustituir el uso del comando de instrucción “Modificar campo” por el comando de instrucción “Modificar campo solamente” que se encarga de modificar el valor del campo pero evitando que se disparen los contenidos iniciales. Si en algún momento necesitamos que se ejecuten los contenidos iniciales podremos forzarlo ejecutando el comando de instrucción “Calcula campos dependientes”, este comando se puede ejecutar múltiples veces antes de grabar el registro.

Índices

Crea siempre los índices de campos punteros a maestros

La base de datos Velneo tiene algunos automatismos realmente interesantes, uno de ellos es la creación automática de los enlaces plurales, este subobjeto es totalmente dinámico y se crea en tiempo de ejecución en base a los índices existentes en las tablas, de tal forma que si las primeras partes de un índice coinciden con el índice *ID* de una tabla maestra se crea automáticamente el subobjeto enlace plural. Este subobjeto permite navegar por la información desde la tabla maestra a su plural.

Además de la navegación desde el maestro a su plural este subobjeto permite el funcionamiento a otro automatismo de la base de datos, el despliegue del cambio de código del maestro a sus plurales. Es decir, si un maestro tiene el código 100 y por algún motivo necesitamos ponerle el código 200, al hacerlo la base de datos de Velneo se encarga de cambiar el código en todas las tablas plurales que apuntan a este registro maestro.

Para que el cambio de código funcione bien y no nos llevemos ninguna sorpresa es necesario que existe un enlace plural. Por este motivo es fundamental que creemos siempre un índice en la tabla plural a través del campo puntero a maestro, en el caso de las tablas submaestras el índice tiene varias partes pero el funcionamiento es similar. Este índice se crea automáticamente cuando creamos el campo puntero a maestro con las herramientas del editor de esquemas o con el selector de tabla maestro en el editor de tabla. Si añadimos el campo manualmente o copiando de otro campo debemos tener la precaución de

crear el índice.

Debemos tener en cuenta que en muchas ocasiones tenemos índices condicionados, es decir que indexan a través del campo puntero a maestro pero solo indexan algunos registros, los que cumplen la condición. Esto debemos tenerlo presente ya que en ese caso el cambio de código solo se aplicaría en los registros que cumplen la condición, pero nos quedarían otros registros con el código antiguo lo que supondría un gran problema. Por ejemplo, si tenemos la familia A100, y en la tabla de artículos solo tenemos un índice por familia que indexa los que tienen existencia. Si cambiamos el código de la familia a B200, solo se cambiaría en los artículos con existencia, quedando erróneamente otros artículos con el código de familia A100, que en caso de ser reutilizado supondría tener una base de datos errónea. Por este motivo, si tenemos que crear índices condicionados es conveniente también tener un índice al maestro sin ninguna condición, es decir que indexe todos los registros.

Evita el cambio de código de maestro siempre que sea posible

Es cierto que no es habitual cambiar los códigos *ID* de las tablas maestras, y en muchos casos ese dato ni se visualiza en pantalla ni se deja cambiar al usuario, pero en una base de datos existen muchas tablas con circunstancias “especiales”, por ejemplo tablas cuyos registros nacen de la importación de información de otro sistema o que son claves que cambian con el tiempo.

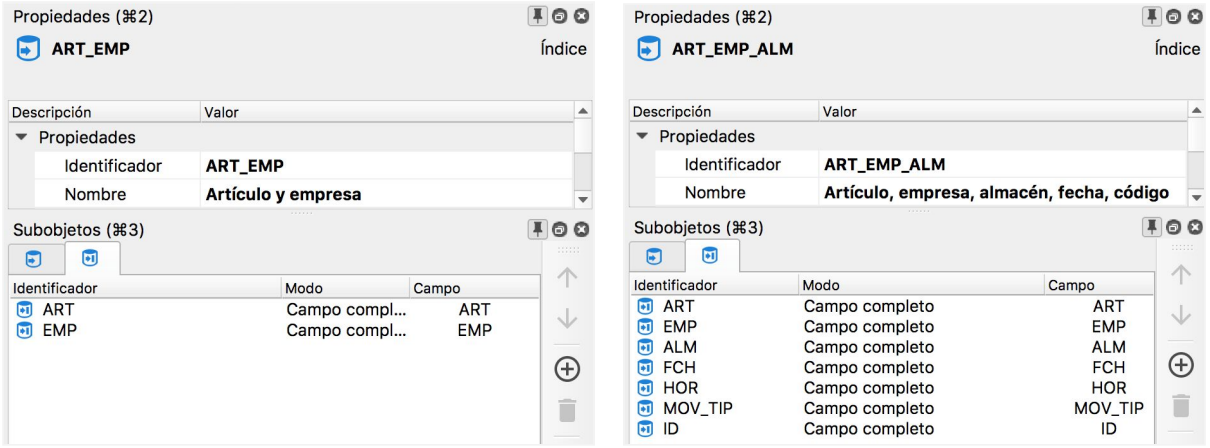
Lo más recomendable en Velneo es crear siempre la tabla maestra con el campo *ID* y añadir otros campos de códigos externos que pueden cambiar, pero dejando siempre como enlace entre el maestro y sus plurales a través del *ID* que genera Velneo. Esto es lo más habitual y recomendable, porque aunque los artículos tengan referencias o códigos de barras y los clientes tengan un *CIF* o un *DNI* siempre será más óptimo indexar y apuntar a tablas del campo *ID* numérico que tendrá un máximo de 4 bytes. De esta forma ganamos espacio y rendimiento.

Si en alguna tabla, por ejemplo de tipo arbolada, necesitamos usar códigos que nos vienen dados por terceros, debemos tener en cuenta lo comentado en el punto anterior sobre tener siempre índices en las tablas plurales sin condicionar que nos aseguren que cualquier cambio en el código del maestro se aplicará en todos sus plurales.

Evita los índices “duplicados” que tienen la parte izquierda común

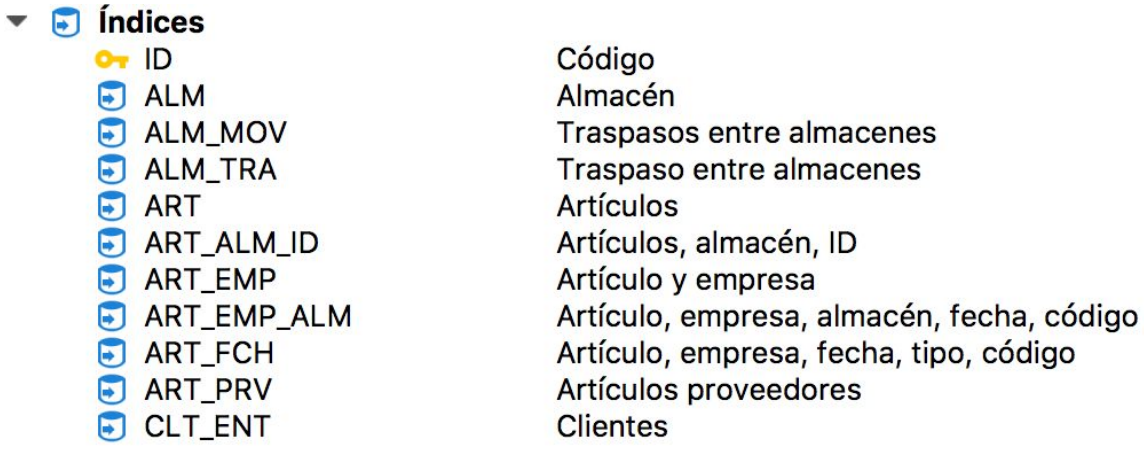
En tablas grandes con muchos campos y muchos índices hay que tener especial precaución con los índices que se crean ya que es muy fácil crear índices “duplicados” si no tomamos medidas para evitarlo.

¿Qué es un índice duplicado? Obviamente el primer caso de índices duplicados es aquél en que ambos índices son exactamente iguales, pero también podemos considerar que un índice está duplicado cuando sus partes coinciden con las primeras partes de otro índice. Veamos un ejemplo.



Es muy habitual que a medida que va creciendo el proyecto se vayan creando nuevos índices, en el ejemplo anterior es posible que inicialmente se haya creado el índice *ART_EMP* con esas 2 partes y posteriormente se creó el índice *ART_EMP_ALM* con 7 partes, si el responsable de base de la base de datos no tiene cuidado quedarían los 2 índices creados cuando realmente no es necesario ya que podemos utilizar el índice *ART_EMP_ALM* buscando por parte izquierda resolviendo solo el artículo y empresa, obteniendo de esta forma el mismo resultado que si usamos el índice *ART_EMP*. Cuando encontremos un caso de estos nos quedaremos con el índices de más partes y refactorizaremos los objetos que usaban *ART_EMP* para que usen *ART_EMP_ALM* por parte izquierda o incluso también se puede usar entre límites.

La mejor forma de evitar tener índices duplicados es poner buenos identificadores a los índices para que expresen bien sus partes y organizar los índices de las tablas por orden alfabético. De esta forma detectamos fácilmente las duplicidades de un vistazo, como se puede apreciar en la siguiente captura.



Es evidente que ver un índice *ART_EMP* al lado de otro que se llamada *ART_EMP_ALM* es un claro indicativo de que puede haber una duplicidad, aunque podría darse la circunstancia de que tengan diferente condición de indexación, algo que debería reflejarse en el identificador.

¿Cuándo usar índices condicionados?

Los índices condicionados son una gran herramienta para el programador. En principio su uso es totalmente aconsejable ya que con ellos mejoramos el rendimiento de nuestras aplicaciones al evitar búsquedas más complejas o filtrados.

Es cierto que el tiempo de indexación de un índice condicionado es aproximadamente un 30% superior a un índice sin condicionar, al tener que evaluarse la fórmula de la condición, pero este tiempo además de que solo nos penaliza una única vez en el alta, baja o modificación, es muy pequeño, por lo que podemos asumirlo sin ningún problema dadas las ventajas que nos aporta.

Siempre que tengamos estados de registros, los índices condicionados son un gran aliado ya que podemos obtener de forma directa los registros adecuados ordenados en función de las partes definidas. Sin duda alguna una herramienta a tener en cuenta y usar de forma constante.

Hay dos casos en los que no merece la pena crear índices condicionados:

1. Si un índice condicionado solo se usa una vez al año para un informe concreto, no tiene mucho sentido crear en la tabla un índice que estará infrautilizado para ganar unos segundos en un informe que apenas se utiliza.
2. Si tengo decenas de estados, en lugar de crear decenas de índices condicionados tiene más sentido crear un índice por el campo estado y buscar de forma directa un estado.

Los índices acepta repetidas ocupan 4 bytes más

El tamaño de un índice viene dado por la suma de tamaño de las partes que lo componen, sin embargo en un índice de tipo acepta repetidas debemos tener en cuenta que Velneo añade 4 bytes al tamaño del índice. Esto lo hace porque aunque acepte claves repetidas la base de datos necesita poder apuntar a cada registro de forma única. Al añadir 4 bytes Velneo permite hasta 4.000 millones de repeticiones de una clave. Es decir que aunque para nosotros a nivel de programación se aceptan claves duplicadas, internamente se comporta como si fuese un índice de clave única, aunque nosotros como programadores nunca veremos los 4 bytes adicionales que componen el índice.

Los índices de clave única son más rápidos

A la hora de regenerar una tabla o indexar alguno de sus índices podemos observar que los índices de clave única son más rápidos en estas operaciones que los de acepta repetidas, lógicamente en este proceso influye lo comentado en el apartado anterior del control de claves repetidas. Además, cuanta menos repetición de claves tengamos más rápido se indexa un índice.

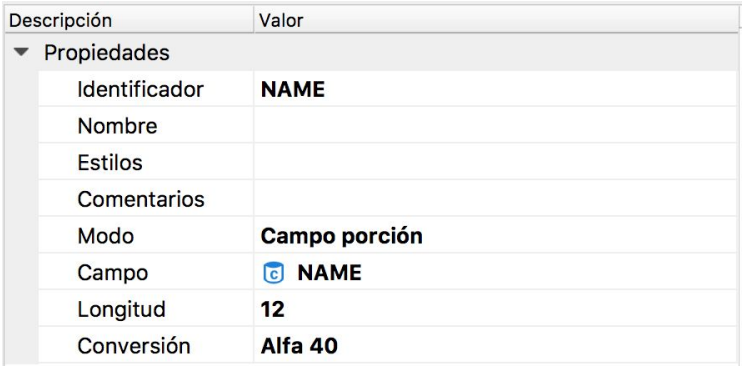
En un índice acepta repetidas el orden de los registros vendrá dado por el orden de creación de dicho registro, este comportamiento puede ser deseado o no. En caso de que queramos garantizar un orden específico conviene añadir más partes a nuestro índice, intentando siempre en la medida de los posible crear el índice con el menor tamaño posible. Por ejemplo, en la tabla de facturas podemos crear un índice por el cliente de tipo acepta repetidas, pero puede ser mucho más interesante crearlo con las partes cliente y fecha, de esta forma cuando carguemos plurales de facturas del cliente nos aparecerán ordenadas por fecha, si además en el índice añadimos el *ID* o el número de la factura y podemos poner el índice de tipo clave única, además de ser un índice más rápido para la reindexación conseguiremos que en caso de que un cliente tenga más de una factura en la misma fecha salgan ordenadas por número.

En definitiva, que es más recomendable tener índices de clave única para lo cual en las tablas maestras siempre podremos conseguirlo de forma sencilla añadiendo el *ID* como última parte del índice.

Usa la longitud y conversión de la parte del índice para reducir el tamaño

Cuando tenemos que indexar un campo alfabético con un tamaño grande (>50 caracteres) puede ser muy

buena opción aplicar una indexación parcial. Salvo que sea necesario indexar de clave única, podemos utilizar la propiedad longitud para reducir el tamaño del índice.



En el ejemplo anterior vemos como al especificar la propiedad longitud en el campo *NAME* nos permite reducir el tamaño del índice a 12 bytes. De esta forma solo se indexarán los primeros caracteres del campo, algo que en la mayoría de las ocasiones no supone ningún problema ya que no es habitual que coincidan, y en el caso de que coincidan estarían juntos en la lista.

Por otro lado la propiedad conversión nos permite indicar que aunque el campo sea de tipo *Alfa 256*, a la hora de indexarlo, en el índice se indexe como *Alfa 128*, *Alfa 64* o *Alfa 40* consiguiendo de esta forma que se puedan encontrar los registros tanto en minúsculas como en mayúsculas y sobre todo que con una longitud de 12 bytes en el índice estemos indexando por los 18 primeros caracteres del campo *NAME*.

Índices de trozos y palabras

Sin duda son los índices más potentes de la base de datos de Velneo, su gran virtud es la potencia de búsqueda su mayor problema es el tamaño en disco y la reindexación. Por este motivo hay que equilibrar su uso.

En tablas con pocos registros no hay ningún problema generar ambos índices, pero en tablas con millones de registros tenemos que tratar de evitar que el índice nos cause problemas de rendimiento, en algunos casos puede ser conveniente generar solo el índice por palabras ya que es mucho más reducido que el de trozos, pensemos que la palabra “Amortiguador” generaría una única entrada en el índice de palabras, pero 10 entradas (Amo, mor, ort, rti, tig, igu, gua, uad, ado, dor) en el índice por trozos, lo que supone una gran ocupación en disco y un mayor tiempo de reindexación.

Debemos evitar siempre crear, siempre que sea posible, varios índices de trozos y palabras. Es decir, no tiene sentido crear el índice por palabras para el campo nombre y otro índice por palabras para el campo dirección, en ese caso debemos crear un único índice por palabras añadiendo ambos campos como partes del mismo índice, además de tener menos índices lo que mejora el tiempo de reindexación ya que solo se lee el registro una vez para reindexar ambos campos sino que además nos permite que el usuario busque por cualquier de los dos datos a la vez sin tener que pedirle dos datos en pantalla o tener que hacer 2 búsquedas y cruzarlas.

Hay que tener en cuenta que podemos incluir en los índices por trozos y palabras campos de tipo objeto texto y objeto texto enriquecido, en este último caso Velneo se encarga de quitar las etiquetas HTML e indexar solo el contenido del campo. Debemos ser precavidos a la hora de indexar este tipo de campos por trozos o palabras ya que el número de entradas en el índice puede ser gigantesco dependiente de lo que grabemos en dichos campos ya que debemos recordar que son de longitud variable y si el usuario quiere puede meter en un campo el contenido de un libro. Además de la ocupación en disco, dar de alta un

registro que tenga que indexar un gran volumen de palabras o trozos de palabras puede suponer un retardo que produzca una mala experiencia para el usuario.

Índices complejos

Este tipo de índice como su nombre indica es un objeto sencillo de definir pero con una funcionalidad realmente compleja que resuelve casos que requieren mucha programación o que gracias al uso de este tipo de índice se consiguen unos rendimientos que no podemos alcanzar mediante programación.

Por cada índice complejo crea código para regenerarlo la primera vez que se instancia

Es muy importante tener en cuenta que aunque los índices complejos se reindexan automáticamente al cambiar las partes tienen el hándicap de que no se indexan la primera vez que se crean, algo que debemos tener en cuenta si creamos un índice complejo sobre tablas que contienen datos. Una buena práctica consiste en crear el código necesario para forzar su indexación inicial cuando instalamos la versión de nuestra aplicación.

```
// Regenerar índices complejos añadidos en la 7.16.1
theApp.regenComplexIndex("velneo_verp_2_dat/COM_ALB_PRV_NOM", false);
theApp.regenComplexIndex("velneo_verp_2_dat/COM_FAC_PRV_NOM", false);
theApp.regenComplexIndex("velneo_verp_2_dat/COM_PED_PRV_NOM", false);
theApp.regenComplexIndex("velneo_verp_2_dat/VTA_ALB_CLT_NOM", false);
theApp.regenComplexIndex("velneo_verp_2_dat/VTA_FAC_CLT_NOM", false);
theApp.regenComplexIndex("velneo_verp_2_dat/VTA_PED_CLT_NOM", false);
theApp.regenComplexIndex("velneo_verp_2_dat/VTA_PRE_CLT_NOM", false);

theApp.regenComplexIndex("velneo_verp_2_dat/ENT_CTT_M", false);
theApp.regenComplexIndex("velneo_verp_2_dat/ENT_DIR_M", false);
theApp.regenComplexIndex("velneo_verp_2_dat/ENT_PAI_M", false);
theApp.regenComplexIndex("velneo_verp_2_dat/ENT_REL_M", false);
```

¿Cuándo debo usar un índice complejo?

Poder indexar registros de una tabla por datos que se encuentran almacenados en otras tablas nos ayuda a reducir el tamaño de las tablas al no tener que duplicar información redundante para poder indexarla, nos evita programación adicional para reflejar los cambios de datos en la tabla donde queremos indexar, sin embargo, también tiene como pro que regenerar índices complejos de tablas grandes va a requerir tiempo y puede que mucho espacio en disco, en función del tamaño de las partes a indexar.

Ejemplos típicos de índices complejos son:

- Indexar contactos por sus direcciones, teléfonos, emails.
- Indexar ventas por las palabras del artículo.
- Indexar facturas por los trozos del nombre del cliente.

Por este motivo hay que tener precaución a la hora de generar índices complejos de tablas con millones de registros con un índice por trozos o palabras ya que estaríamos creando un índice enorme en tamaño y con un tiempo de reindexación muy elevado. Esto no quiere decir que no podamos crear un índice complejo por trozos o palabras del nombre del artículo indexando las líneas de venta, pero sí debemos tener en cuenta el tamaño y la ocupación para decidir si por ejemplo solo lo generamos por palabras que será mucho más pequeño que si lo hacemos por trozos.

Actualizaciones

Es la característica estrella de las tablas Velneo. Poder actualizar valores en tablas maestras desde sus plurales sin programar código es muy atractivo, sin duda, pero todavía lo es más la rapidez de los cálculos que al estar automatizados en el propio sistema Velneo son más rápidos que si los programamos nosotros en procesos o funciones y además la fiabilidad de que funcionan bien en todos los casos, aunque en la definición solo le digamos lo que tienen que hacer en el alta.

Utiliza actualizaciones siempre que puedas

Dadas sus virtudes no hay duda, siempre que puedas hacer una actualización no escribas código en los eventos de tabla. Es más, deberías pensarlo al revés, siempre que vayas a escribir código en un evento de tabla piensa si puedes hacerlo mediante una actualización.

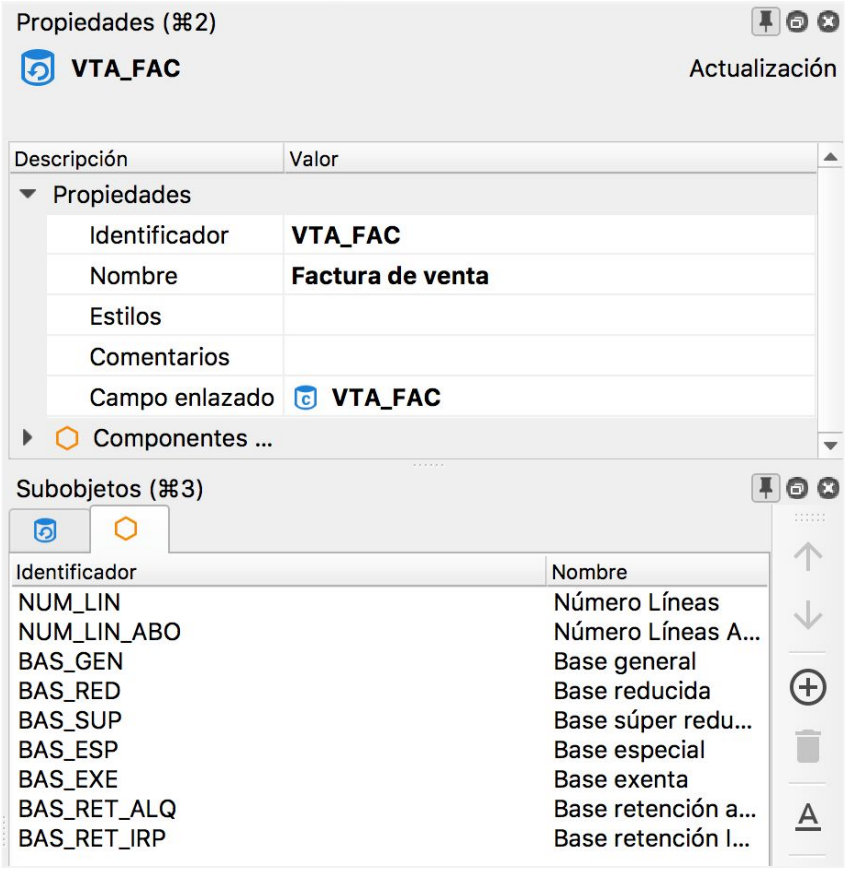
Hay que tener en cuenta que a la facilidad de configuración de una actualización se le une la posibilidad de condicionarla lo que facilita la realización de cálculos más complejos. Aplicando estos criterios, en muchos casos es preferible hacer actualizaciones en tablas maestras acumulando líneas totales, líneas servidas, etc. que nos permiten declarar en la tabla maestra un campo que nos indique si ya está servido o no en base a los valores de los campos acumulados en vez de escribir código en ningún evento de tabla.

En las actualizaciones por valor absoluto hay que tener en cuenta las bajas

Es habitual usar actualizaciones para almacenar en una tabla maestra los últimos valores, por ejemplo en el cliente podríamos guardar la fecha del último pedido. En estos casos tanto en alta como en modificación no hay problema a la hora de condicionar la actualización y dejar el valor correcto en el maestro, sin embargo cuando damos la baja de un pedido que era el último de un cliente nos encontraremos de que no podemos actualizar la fecha del último pedido, salvo que tengamos un puntero a hermano contiguo o un singular de plural que nos facilite obtener dicho dato. Este caso debemos tenerlo en cuenta para en el trigger posterior a la baja ejecutar un código que se encargue de buscar el último pedido del cliente y actualizar su fecha.

Crea solo una actualización por tabla

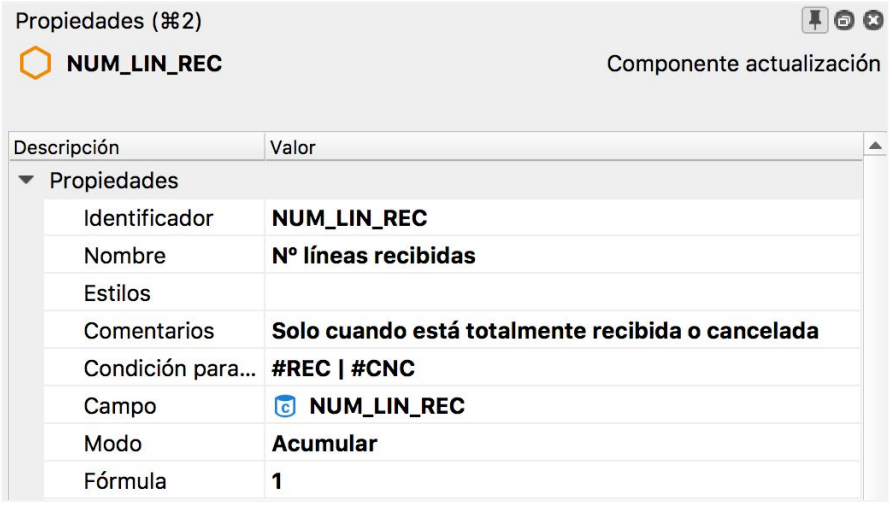
Si tenemos que actualizar más de un campo en la tabla maestra no tiene ningún sentido crear una actualización para cada campo. Esto además de hacer crecer el tamaño de nuestro proyecto es peor a nivel de rendimiento porque obliga a ejecutar varias actualizaciones contra el mismo registro. Por lo tanto siempre que tengamos que hacer actualizaciones a una tabla maestra debemos incluir en la misma tantos componentes de actualización como sean necesarios.



Utiliza actualizaciones condicionadas

La versatilidad de las actualizaciones se ve potenciada con la posibilidad de utilizar condiciones. El principal motivo es que Velneo es capaz de actualizar en función de la condición de forma automática, es decir, que si se cumple la condición aplica la actualización y si deja de cumplirse aplica la actualización contraria, y lo más importante sin programar lo que reduce la posibilidad de errores del programador.

Por ejemplo si condicionamos una actualización del nº de líneas recibidas de un pedido a que la línea esté recibida o cancelada, cuando se cumple la condición se suma 1 al campo de la tabla maestra, sin embargo, al cambiar la condición si ya no se cumple se resta 1.



Por este motivo es conveniente pensar si podemos crear una actualización antes de escribir código.

No utilices variables locales en la condición o fórmula de las actualizaciones

Aunque las tablas permite declarar variables locales cuyo valor podemos alterar y usar en todos los eventos de tabla, por el momento Velneo no es capaz de usar el valor de esas variables locales en las actualizaciones, ni en la fórmula del valor ni en la condición para modificar.

Hay que tenerlo en cuenta porque el editor sí nos permite usarlas en las fórmulas, pero en ejecución no funcionará.

Evita complejas actualizaciones encadenadas que puedan ocasionar conflictos por bloqueo

Con las grandes virtudes que tienen las actualizaciones es lógico usarlas masivamente y con total tranquilidad.

Sin embargo, de la misma forma que nos puede ocurrir con los contenidos iniciales de campos donde podemos por mala definición crear un cálculo recursivo, en las actualizaciones nos puede pasar lo mismo. Por ejemplo, podríamos cometer el error de que la tabla A actualiza la tabla B, la tabla B actualiza la tabla C y la tabla C actualiza la tabla A produciendo un error por recursividad ya que tras la modificación de la tabla C a la A volvería a empezar el ciclo. Sin duda se trataría de un error de programación que seguramente podemos evitar aplicando condiciones a las actualizaciones para evitar que ejecute más de un ciclo.

Eventos de tabla o triggers

No modifiques datos en el trigger posterior

Aunque parezca de perogrullo, lo cierto es que a veces ocurre que por despiste o por copia/pega puedes ver código en un trigger posterior al alta o modificación tratando de modificar el registro que acaba de ser creado o modificado.

Lo peor de todo es que si el programador trata de ver el valor de los campos modificados obtendrá que la ficha en memoria ha cambiado y puede considerar que la programación es correcta, sin embargo, debemos tener presente que en el trigger posterior ya que no se cambia la ficha en disco, por mucho que cambiemos los valores de los campos en la ficha en memoria.

No dejes eventos de tabla vacíos

Existen hasta 9 posibles eventos de tabla diferentes y en ocasiones se crean con un código que posteriormente se modifica o incluso se elimina. Debemos tratar de dejar siempre nuestro código lo más limpio posible, y si quitamos todas las líneas de un evento de tabla, debemos eliminarlo ya que de lo contrario estamos dejando un subobjeto que además de ocupar espacio también consume tiempo de ejecución al tener que evaluarlo al producirse una operación transaccional en la tabla.

Variables globales

Este objeto de datos puede ser de dos tipos según su persistente, en disco o en memoria.

Uso controlado de las variables globales en disco

La gran virtud de una variable global en disco es la sencillez con la que se declara y que está accesible a todos los ámbitos de la aplicación.

Debido a que su funcionamientos es similar al de una tabla por lo que cada vez que hacemos referencia a una variable global en disco en una fórmula o comando de instrucción ejecutados en el cliente estamos provocando una conexión al servidor para solicitar el valor actual. Por lo tanto debemos usarla con mucha precaución sobre todo en aplicaciones que se van a ejecutar en el Cloud.

Esto no es óptimo por lo que en muchos casos es preferible usar una tabla de configuración con un solo registro en el que incluimos los campos que deseamos compartir por todos los usuarios. La ventaja de la tabla es que una vez cacheada en memoria su lectura no requiere conexión al servidor y si hay cambios el refresco en tercer plano se encarga de actualizarla.

Otra posible optimización es crear una variable global en memoria que rellenamos al arrancar la aplicación con el valor de la variable global en disco, por lo que reducimos el número de conexiones al servidor a una. Sin embargo, esto solo es válido si no necesitamos tener su valor actualizado en caso de que haya cambiado.

Las variables globales son compartidas

Las variables globales en disco con compartidas por todos los usuarios, sin embargo las variables globales en memoria son compartidas exclusivamente por el cliente que las ejecuta.

Si en una misma máquina ejecutamos varios vClient, cada vClient tendrá su propia instancia de las variables en memoria, esa instancia de la variable es compartida para todos los objetos de la aplicación de ese vClient, pero no será visible para el resto de vClient.

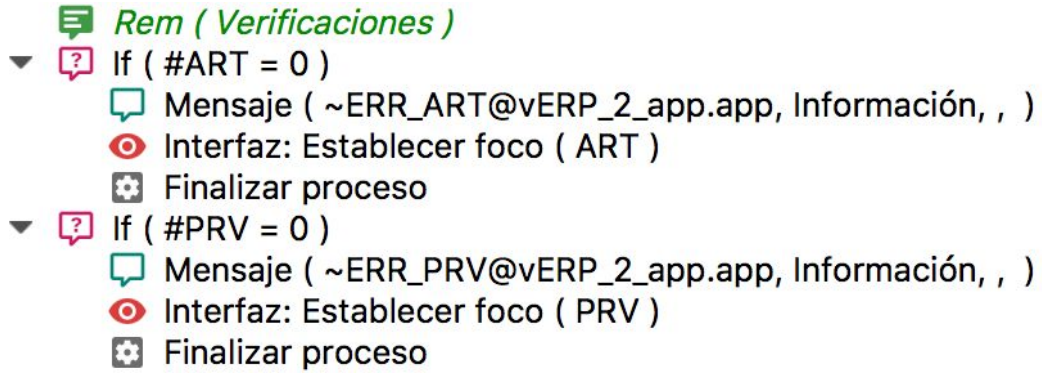
Debemos tener en cuenta que en el servidor también se crean variables globales en memoria por lo que podemos usarlas para compartir información entre todos los usuarios teniendo siempre presente que el valor de esa variable se perderá en el reinicio del Velneo vServer. Estas variables pueden ser interesantes para contener información en curso como podría ser el caso de las sesiones web conectadas en una aplicación que devuelva contenido para web.

Constantes

Como su nombre indica este objeto está destinado a almacenar valores fijos que no podrán alterarse en tiempo de ejecución.

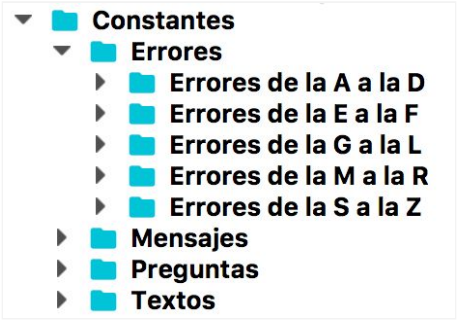
Usa constantes para todos los textos que puedan requerir traducción

Cuando escribimos textos en las propiedades de los objetos, subobjetos y controles de nuestra aplicación dependiendo del tipo de propiedad se pueden traducir directamente con el componente de la plataforma Velneo vTranslator, sin embargo los textos escritos en fórmulas no se pueden traducir, con el fin de facilitar la traducción de todos los textos es recomendable utilizar constantes en todos los textos usados en fórmulas.



Organiza las constantes por su uso

A medida que va creciendo una aplicación se hace necesario organizar las constantes que vamos declarando, una posible organización es la que vemos en la siguiente captura.



En la tabla siguiente se muestran las agrupaciones más habituales de constantes así como el prefijo utilizado:

Tipo	Prefijo	Descripción
Errores	*ERR_*	Utilizadas en los mensajes de error de las diferentes verificaciones.
Mensajes	*MSG_*	Utilizadas para los textos que visualizan en mensajes informativos.

Preguntas	*PRG_*	Utilizadas para contener los textos usados en preguntas y confirmaciones.
Textos	*TXT_*	Utilizadas para contener textos de uso general como nombres de tablas u otros términos.

Dentro de cada carpeta las constantes se organizan alfabéticamente. En el caso de que tenemos muchas (>30) constantes podemos crear subcarpetas como vemos en la imagen superior para agruparlas según su letra inicial del identificador.

Imágenes

Lo primero que tenemos que tener claro es que en muchos proyectos el mayor tamaño viene dado por las imágenes incluidas en el mismo. Las imágenes son un gran recurso, pero mal utilizado puede ser un gran enemigo a la hora de tener proyectos con un tamaño reducido.

Reduce el número

Como programadores nos gusta disponer de recursos gráficos para aplicar como iconos para botones, toolbars, etc. Por ese motivo tendemos a añadir a nuestros proyectos todos los iconos que consideramos de uso habitual en las aplicaciones. El objetivo es noble, pero la realidad es bien distinta, todo lo que no se usa sobra, por lo tanto deja solo en tus proyectos las imágenes que realmente usas y elimina las que no utilices. La excusa del “por si acaso” no es válida, no hay ningún problema en añadir una imagen o icono en el momento que la necesites.

No incluyas las imágenes a través del portapapeles

Cuando incluimos imágenes en nuestros proyectos debemos tratar de importarlas siempre directamente de un fichero en disco, es la mejor forma de garantizar que la estamos importando con las optimizaciones adecuadas. Si por ejemplo copiamos una imagen al portapapeles lo más probable es que su formato sufra una conversión que nos haga perder toda optimización que hayamos realizado.

Optimiza las imágenes antes de importarlas

Cuando vayamos a importar una imagen o icono en nuestro proyecto antes de importarla es recomendable pasarla previamente por un sistema de optimización que reduzca su paleta de colores o tamaño. De esta forma podemos ganar cientos de bytes que siempre son de agradecer para conseguir proyectos del menor tamaño posible lo que agiliza su almacenamiento y el envío del mismo por Internet. Existe multitud de aplicaciones y servicios online para hacerlo como por ejemplo tinypng.com que nos permite arrastrar y soltar múltiples imágenes de diferentes formatos y que son optimizadas individualmente para su descarga.

¿Dónde ubicar los objetos dibujo?

Si son imágenes o iconos que vamos a usar en esquemas o acciones definidas en el proyecto de datos no nos queda más remedio que ubicarlas en el proyecto de datos.

Si son imágenes o iconos que vamos a usar la interfaz parece más lógico ubicarlas en el proyecto de aplicación ya que de forma natural trataremos de localizarlas en el mismo proyecto donde estamos creando la interfaz. Si el número de imágenes es reducido no merece la pena pensar en ubicarlo en el proyecto de datos.

Si nuestro proyecto requiere el uso de cientos de imágenes o iconos y queremos “adelgazar” nuestro proyecto de aplicación, podemos almacenarlas en el proyecto de datos que habitualmente ocupa una décima parte del tamaño del proyecto de aplicación, consiguiendo así reducir el tamaño del proyecto de aplicación sin que la penalización del proyecto de datos que suele cambiar mucho menos sea un problema.

Evita la información redundante, icono y texto juntos no siempre tienen sentido

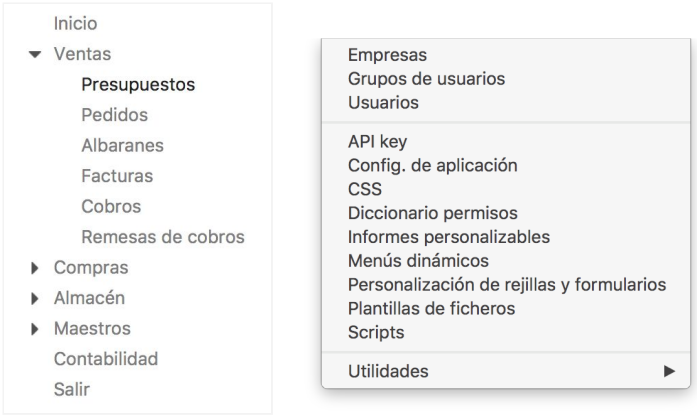
Cuando aplicamos un sistema de diseño conseguimos unicidad en nuestra aplicación, es decir, que el usuario vea que toda la aplicación se comporta igual y tiene una interfaz homogénea. Uno de los aspectos a considerar por el diseñador es en que casos se aplicarán iconos, cuando llevarán solo texto y cuando deben llevar ambos datos.

Si un icono es muy representativo no necesita de texto, por ese motivo en la toolbars se pueden llegar a utilizar solo iconos sin que el usuario tenga necesidad de más explicaciones para reconocerlos. Debemos tener en cuenta que un texto se lee y se entiende mientras que una imagen requiere interpretación.

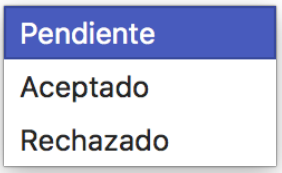
En el caso de los botones es habitual tener que usar texto ya que no siempre es fácil representar su significado mediante iconos, por ese motivo puede ser más coherente no usar iconos en ningún botón

aunque muchos podrían tener un icono fácilmente reconocible.

En los menús el texto se hace necesario por lo que el uso del icono debería estar bastante justificado ya que de lo contrario estaría metiendo ruido al ser información redundante respecto al texto.



En los combobox se pueden usar iconos cuando representan información rápida de leer para el usuario, ya que de lo contrario también caemos en la redundancia, por ese motivo entre poner solo icono o solo texto, tiene más sentido usar solo texto.



Es cierto que para representar información de estados en una rejilla si puede ser más útil el icono debido a que ocupa menos espacio que el texto, pero siempre y cuando el icono no requiera ninguna explicación. Por ejemplo, un círculo verde = servido y un círculo, rojo = pendiente es algo que como programadores nos parece lógico, pero que el usuario debe interpretar, es evidente que con el tiempo se acostumbrará a los colores y su significado pero no es algo estándar que ya esté preestablecido.

Utiliza una librería de iconos homogénea

La iconografía de la aplicación debemos cuidarla tanto como cualquier otro aspecto de la interfaz. Debe ser homogénea, es decir, no debemos buscar iconos por internet y mezclar iconos de diferentes librerías porque se nota y queda muy mal, da una sensación de aplicación descuidada.

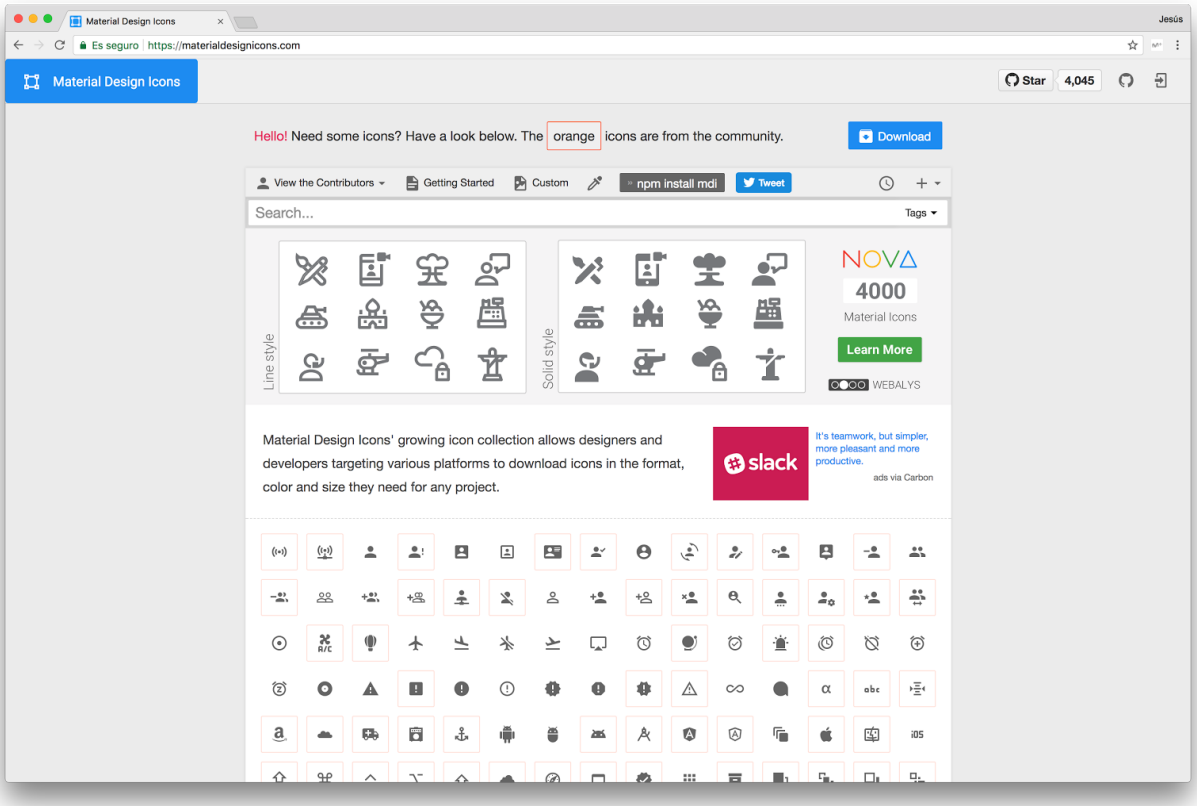
No utilices iconos de diferentes tamaños para los mismos contextos, si pones iconos en las toolbars todos deben tener el mismo tamaño, y lo mismo debes hacerlo en los botones, menús, pestañas, etc.

No te compliques la vida buscando librerías con miles de iconos espectaculares porque los iconos no deben ser “bonitos” sino que deben ser fáciles de interpretar y en este apartado los iconos más elaborados y con más colores cumplen peor esta función. Fíjate en las señales de tráfico como utilizan muy pocos colores y usan imágenes sencillas, fáciles de interpretar en muy poco tiempo.

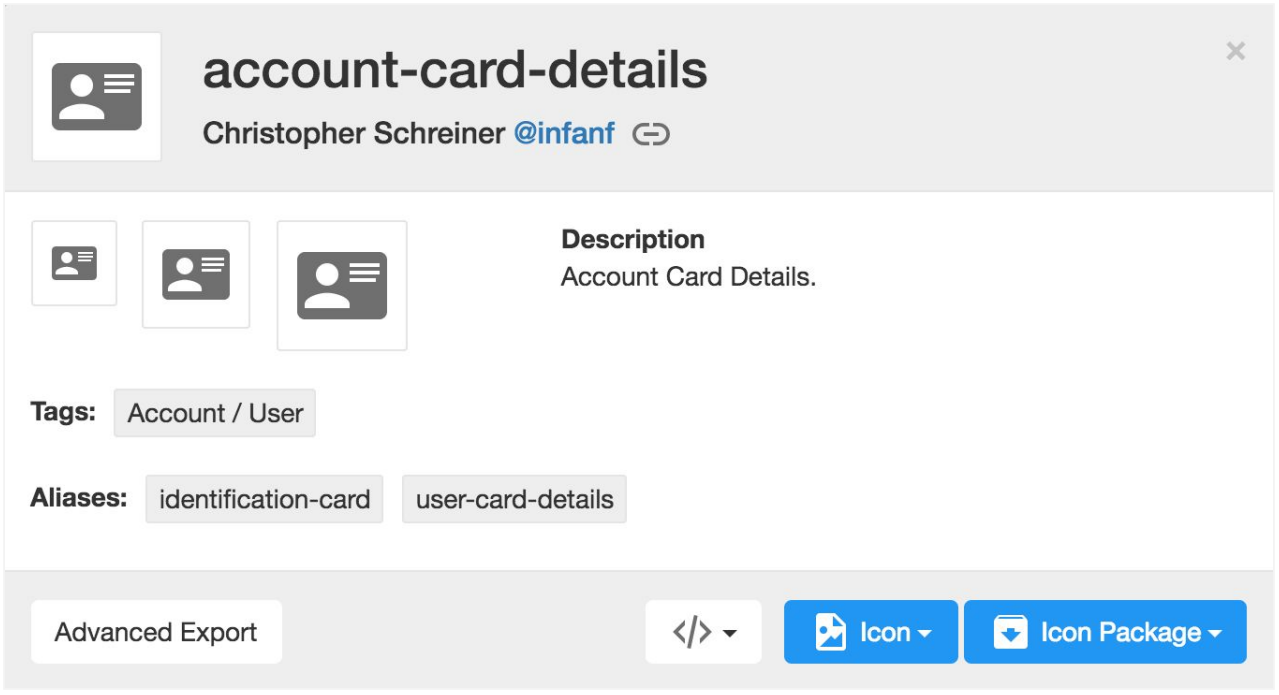
Si tenemos claro el objetivo que deben cumplir los iconos encontraremos que las librerías con iconos más sencillos y de un solo color están triunfando en la web, las aplicaciones para móviles y en las de escritorio más modernas.

Material Design Icons de Google es una gran librería que cuenta con miles de iconos y que además podremos extender con otras muchas librerías gratuitas y de pago que han sido desarrolladas con el mismo sistema y que por lo tanto podremos combinar sin que se aprecien diferencias de estilo.

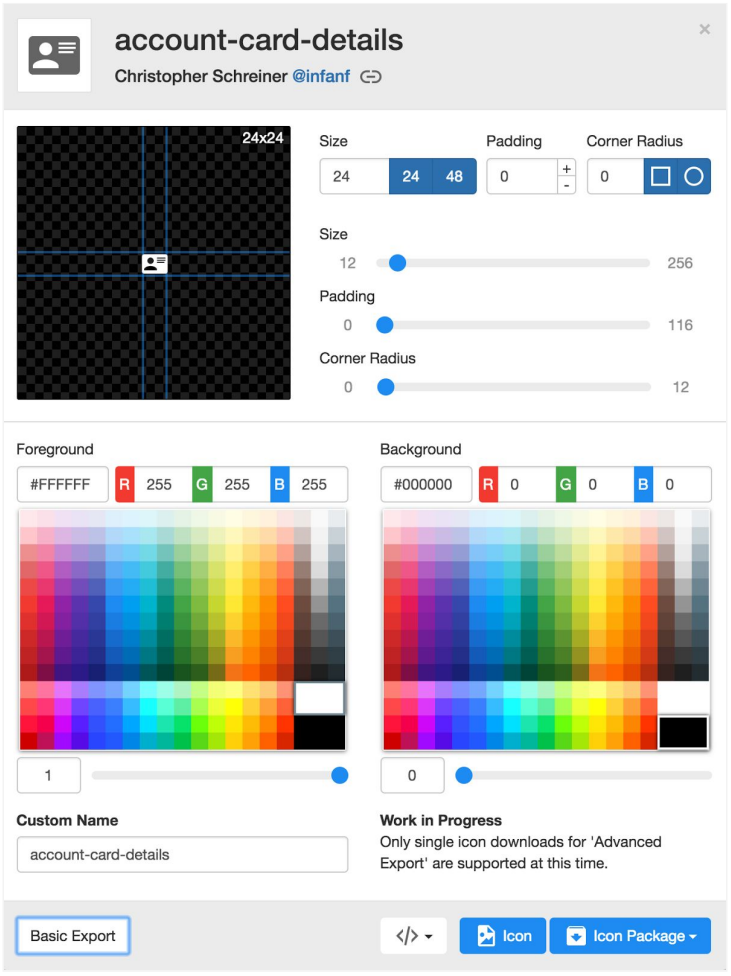
En esta página web contamos con un buscador que nos facilita localizar iconos. Una de las grandes ventajas de esta librería es que cualquier usuario que tenga un móvil Android o que use aplicaciones de Google en iOS se sentirá cómodo porque en muchos casos lo reconocerá de forma directa.



Una de las ventajas de esta página web es que cuando seleccionamos un icono accedemos a una página que nos permite exportarlo a diferentes formatos.



Y además con el botón Advanced Export podremos acceder a un editor que nos permitirá realizar múltiples configuración del icono tanto en tamaño como en colores de fondo, primer plano, padding y radio.



Tras configurar el icono a nuestro gusto podemos aplicarle un nombre al icono y exportar con el botón Icon, o incluso podemos exportarlo en formato SVG.

Utiliza iconos para dar soporte a High DPI

Un aspecto que debemos tener en cuenta en el desarrollo de nuestras aplicaciones es que los dispositivos actuales y más aún en el futuro tienen resoluciones mucho más altas que el FullHD (1920x1080), los dispositivos móviles, tabletas de alta resolución e incluso las pantalla con resolución 4K empiezan a ser más habituales, por este motivo no podemos incluir en nuestra aplicación iconos de 16x16 o 32x32 ya que en estas pantallas se verán pixelados. Para resolver este problema debemos incluir en nuestra aplicación iconos con una resolución de 64x64 o 96x96 para tener cubiertas futuras resoluciones.

CSS

Uno de los aspectos más importantes en el desarrollo de una interfaz de una aplicación es disponer de la posibilidad de aplicar cambios en el diseño de forma global, sin estar obligados a realizar cambios de forma manual en todos los objetos de interfaz de nuestra aplicación.

Las CSS nos permiten en Velneo cambiar de forma sencilla y rápida aspectos de la interfaz tan importantes como los colores, tipografía, tamaños, márgenes, iconos, etc.

Para crear unas CSS coherentes es necesario usar un sistema de diseño que nos facilite su creación. Así que lo primero que debemos hacer es crear nuestro propio sistema de diseño o seleccionar uno ya existente para aplicarlo en nuestros desarrollos.

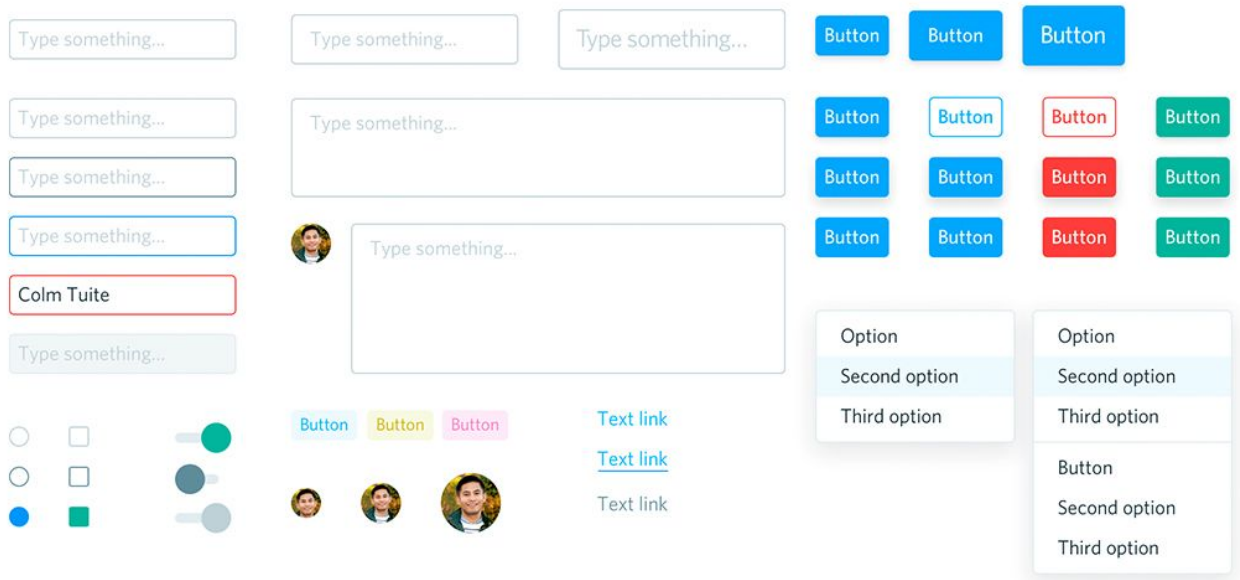
¿Qué es un sistema de diseño?

"Un sistema de diseño es un conjunto de reglas que organizan, dan consistencia y armonía a un entorno complejo y variable de contenido y funcionalidad.

Para que un sistema sea tal, es importante que cumpla algunas premisas: que sea escalable, que su unidad mínima se base en una certeza, que sea recursivo en sus formas y proporciones, que regule no sólo la forma y comportamiento de los objetos sino también las relaciones entre ellos, que sea eficiente, predecible y sometedor. Esto es, que una vez definido, obligue a todo contenido o funcionalidad a existir bajo sus propias reglas."

Javier Cañada (Director de diseño)

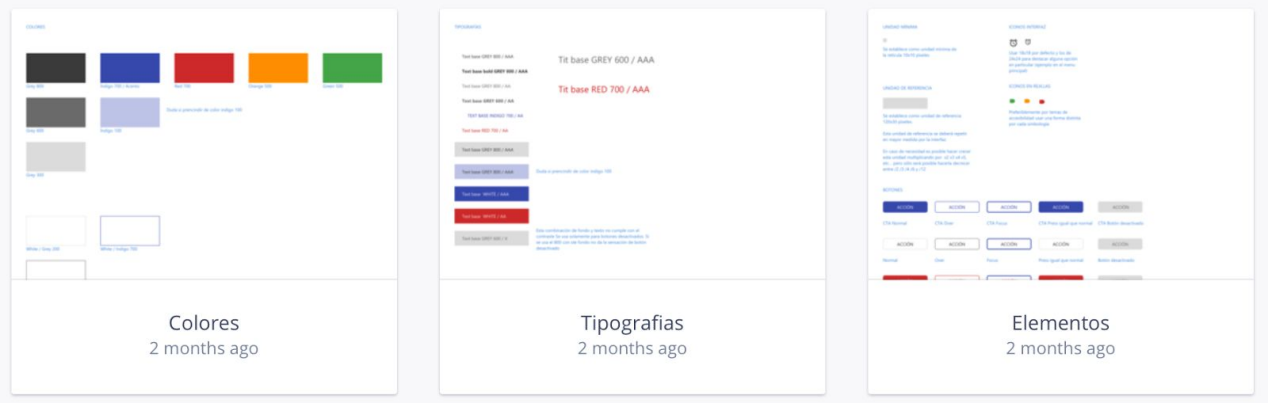
Una de las tareas más importantes de las personas que trabajan en diseño de producto, ya sean el gestor de producto o un diseñador de producto, es crear un sistema de diseño coherente para el producto que se adapte a los diferentes canales en los que va a vivir (producto físico, producto digital, cartelería física, banners para los diferentes canales digitales, etc…).



¿Por qué es tan importante tener un sistema de diseño?

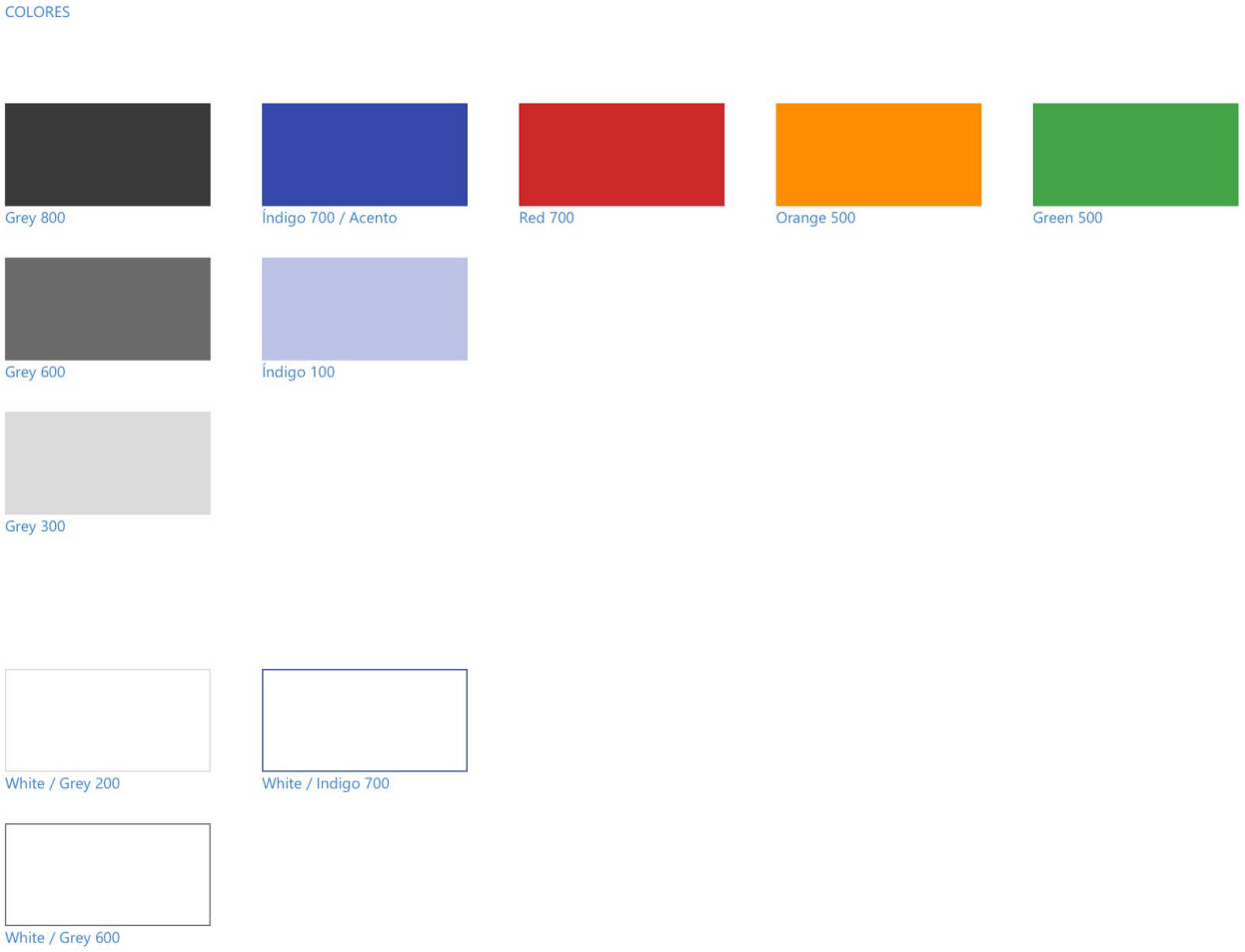
Tener un sistema de diseño coherente, ya sea de creación propia o adaptado a partir de alguno preexistente, es una buena manera tanto de tener un sistema consistente para el usuario, así como una manera de enganchar fácilmente a nuevos miembros del equipo de trabajo y que se adapten a nuestra forma de trabajar de una manera rápida y sencilla.

En Velneo actualmente estamos utilizando el siguiente sistema.



Sistema de diseño. Colores

El sistema especifica los colores que podemos usar en nuestra aplicación, no debemos salirnos de esta paleta de colores y hay que combinarlos de la forma adecuada para conseguir una interfaz limpia, sencilla y a la vez elegante para el usuario.

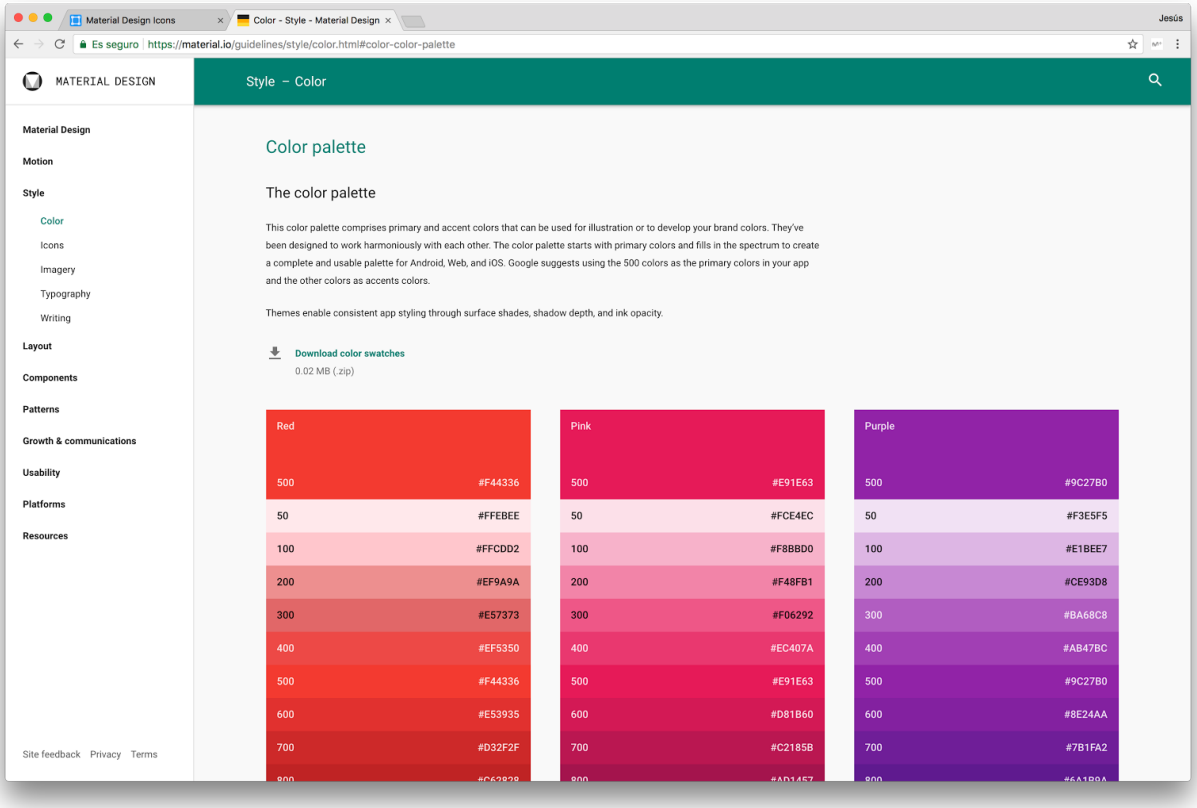


A continuación se detallan los colores con su valor hexadecimal.

Nombre color	Color hexadecimal
Grey 800	#424242
Grey 600	#757575
Grey 300	#E0E0E0
Indigo 700 / Acento	#3F51B5
Indigo 100	#C5CAE9
Red 700	#D32F2F
Orange 500	#FF9800
Green 500	#4CAF50
White / Grey 200	border: 1px solid #E0E0E0;
White / Indigo 700	border: 1px solid #3F51B5;

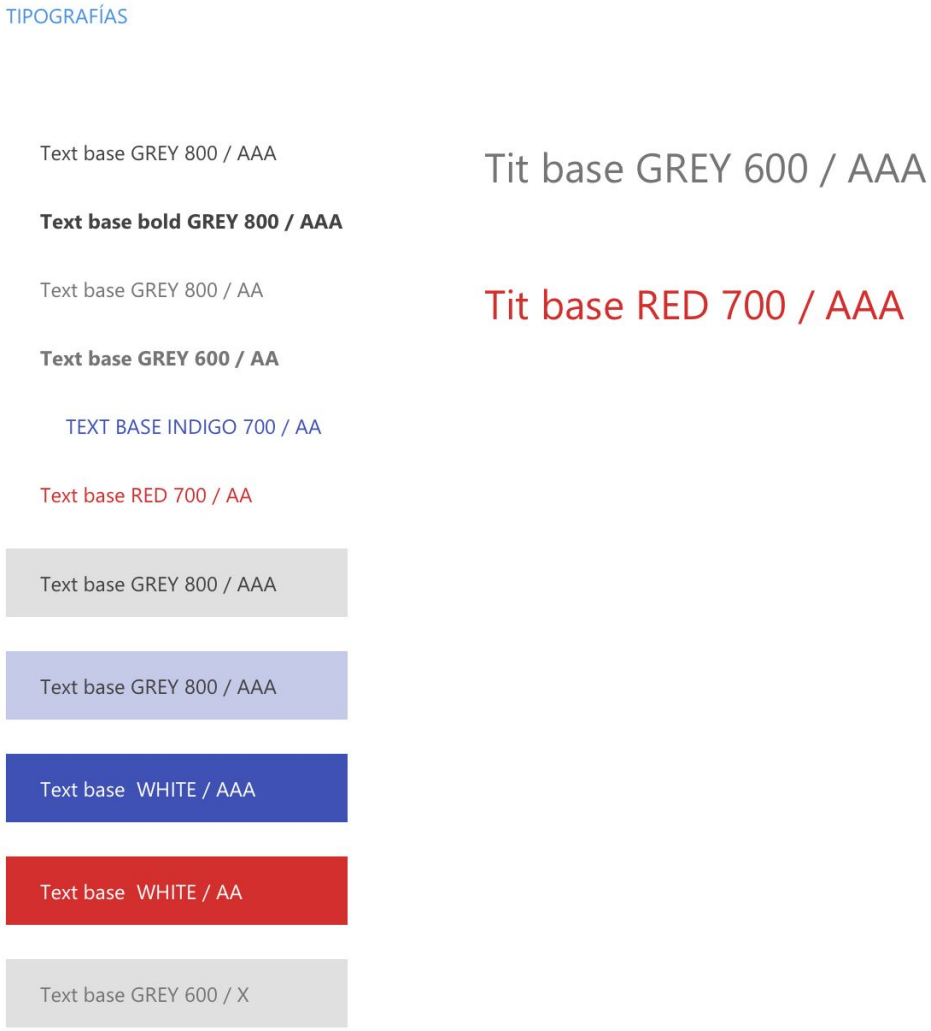
White / Gray 600	border: 1px solid #757575;

Como podemos apreciar se usan las [paletas de colores de Material design](#) para combinar colores con coherencia.



Sistema de diseño. Tipografía

El sistema también define las tipografías que podremos usar en la interfaz de la aplicación.



Si nos fijamos el sistema no especifica una tipografía en concreto, esto es debido a que como Velneo es multiplataforma utilizaremos en la fuente de sistema, que es la propuesta por defecto. La ventaja es que esa fuente siempre existe en el sistema del usuario final y no es necesario realizar ninguna instalación adicional. Además, conseguimos que la interfaz se verá igual en desarrollo que en producción.

Los tamaños especificados en el sistema son:

- Los textos tanto estáticos como de edición son de 12px.
    - Se usa el color GREY 600 para los textos estáticos.
    - Se usa el color GREY 800 para la edición.
- Los títulos (Tit) utilizarán un tamaño de 24px.

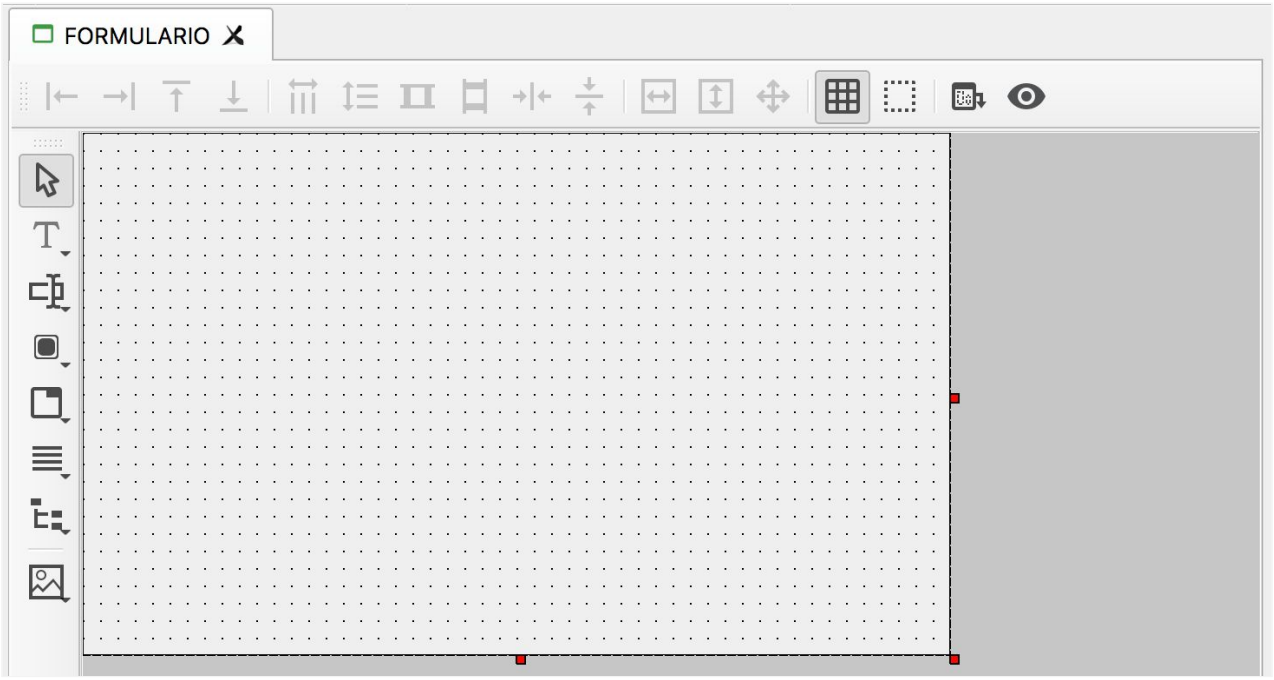
Sistema de diseño. Unidad mínima

Uno de los aspectos clave del sistema es la unidad mínima. Es decir, el tamaño base para su aplicación en el tamaño de todos los controles. En el sistema Velneo la unidad mínima es de 10x10 píxeles lo que significa que todos los controles, formularios, columnas, etc. serán múltiplos de 10 en su alto o ancho.

UNIDAD MÍNIMA

Se establece como unidad mínima de la retícula 10x10 píxeles

La ventaja de usar un múltiplo de 10 es que es muy sencillo de calcular y también de aplicar ya que la cuadrícula del editor de formularios está diseñada en base a esa misma unidad de referencia 10x10.

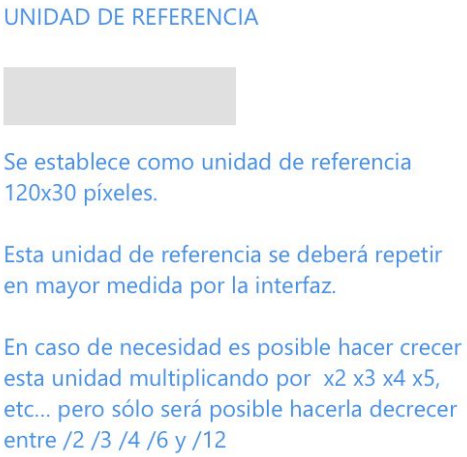


Por este motivo siempre recomendamos diseñar los formularios con la cuadrícula activa, de esta forma todos los controles se pueden añadir y ubicar fácilmente manteniendo las alineaciones correctas y precisas.

Sistema de diseño. Unidad de referencia

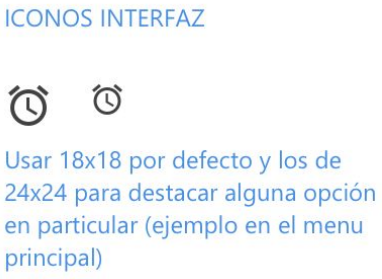
La unidad de referencia es la base para el cálculo de las dimensiones de los controles, formularios y columnas de nuestra aplicación.

En concreto el sistema Velneo utiliza la unidad 120x30, es decir 120 píxeles de ancho por 30 de alto. Si vemos las aplicaciones desarrolladas a partir de este sistema encontraremos que los botones tienen todos estas dimensiones. Sin embargo, podemos encontrar botones que tengan un tamaño diferente debido a que el texto a mostrar es largo y necesita un tamaño mayor o más pequeños si hay que ubicar muchos botones en un mismo área. Para esas excepciones se aplicarán tamaños basados en la referencia aplicando un factor de multiplicación división según queramos hacerlo más grande o más pequeño, es decir que podemos tener botones de tamaño 60, 30 o 240 que siguen aplicando el criterio de la unidad de referencia.

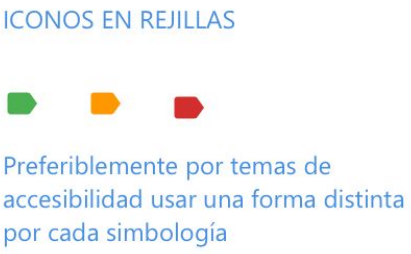


Sistema de diseño. Iconos

En el caso de las toolbars se utilizan icono con tamaño de 24x24 píxeles. Aún así hay que tener en cuenta el High DPI, por ese motivo los iconos deben tener tamaños superiores como 48x48, 64x64 o 96x96 ya que al reducirlos a 24x24 se verán con buena calidad, sin embargo el efecto contrario genera pixelación.



En las rejillas se pueden usar iconos para representar información, tanto con campos objetos dibujo o con iconos de tablas estáticas. En ambos casos el tamaño debería ajustarse a esos 18x18 o 24x24 y aplicar el criterio de forma distinta para cada simbología, es decir combinar color con forma ayuda al usuario a identificar el significado del color.



Sistema de diseño. Campos

A la hora de crear cajas de edición aplicaremos la unidad de referencia como tamaño base, sobre todo en campos de ancho o alto fijo, sin embargo en campos con ancho por defecto o proporcional no tendremos que ajustarnos a la unidad de referencia pues su tamaño dependerá del área disponible en el momento del pintado, sin embargo sí que es conveniente que su tamaño en el editor de formularios se ajuste al máximo posible a los valores de la unidad de referencia.

En cuanto a los colores de borde los controles de edición se aplican con la siguiente CSS.

```
QLineEdit, QTextEdit {
    background-color: #FFFFFF;
    border: 1px solid #E0E0E0;
    color: #212121;
    height: 30px;
    padding-left: 2px;
    selection-background-color: #3F51B5;
    selection-color: #FFFFFF; }

QLineEdit:hover, QTextEdit:hover {
    border: 1px solid #757575; }

QLineEdit:focus, QTextEdit:focus {
    border: 2px solid #3F51B5; }

QLineEdit:disabled, QTextEdit:disabled {
    background-color: #BDBDBD; }
```

Sistema de diseño. Botones y toolbars

En la imagen podemos ver el sistema aplicado a los botones y toolbar en Velneo.

Para la aplicación del estilo de botones utilizamos la siguiente CSS:

```
QPushButton {
    background-color: #FFFFFF;
    border: 1px solid #E0E0E0;
    border-radius: 5px;
    color: #212121;
    font-size: 12px;
    height: 30px;
    line-height: 16px;
    text-align: center;
    qproperty-iconSize: 18px; }

QPushButton:hover {
    border: 1px solid #3F51B5; }

QPushButton:focus {
    border: 2px solid #3F51B5; }

QPushButton:pressed {
    border: 2px solid #3F51B5; }
```

```
QPushButton:disabled {
    background-color: #9E9E9E;
    border: 1px solid #9E9E9E;
    color:  #FFF; }

QPushButton::menu-indicator {
    image: none;
    width: 0px; }
```

Hay que tener en cuenta que hay botones con una CSS diferente como el caso de botón de llamada a la acción “CTA” o de atención “ATN” :

```
QPushButton#BTN_ACE {
    background-color: #3F51B5;
    color: #FFF; }

QPushButton:hover#BTN_ACE {
    background-color: #FFFFFF;
    color: #3F51B5; }

QPushButton:focus#BTN_ACE {
    background-color: #FFFFFF;
    color: #3F51B5; }

QPushButton:pressed#BTN_ACE {
    background-color: #3F51B5;
    color: #FFF; }

QPushButton:disabled#BTN_ACE {
    color:  #727272; }
```

Para la aplicación del estilo de las toolbar utilizamos la siguiente CSS:

```
QToolBar {
    background-color: transparent;
    border: 0px;
    padding: 3px;
    spacing: 10px;
    qproperty-iconSize: 18px; }

QToolButton {
    background-color: #FFF;
```

```
    border: 1px solid #E0E0E0;
    border-radius: 5px;
    color: #727272;
    margin-right: 1px;
    min-height: 18px;
    min-width: 18px;
    padding: 5px;
    qproperty-iconSize: 18px; }

QToolButton:hover {
    border: 1px solid #3F51B5;
    border-radius: 5px; }

QToolButton:focus {
    border: 2px solid #3F51B5;
    border-radius: 5px; }

QToolButton:disabled {
    background-color: #CECECE;
    border: 1px solid #727272;
    color: #727272; }

QToolButton:pressed {
    border: 2px solid #3F51B5;
    border-radius: 5px; }

QToolButton::menu-indicator {
    background-color: transparent;
    color: transparent; }
```

Sistema de diseño. Etiquetas

Las etiquetas suelen representarse con controles de tipo texto estático. Su tamaño viene predefinido por la unidad de referencia y sus colores por la paleta del sistema.

A continuación vemos las CSS que se utilizan para aplicar el sistema:

```
QLabel {
    background-color: transparent;
    color: #757575;
    font-size: 11px; }
```

¿Cuál es la clase para cada tipo de objeto, control o subcontrol?

Cada objeto, subobjeto o control puede disponer de una clase en el CSS que nos permite alterar su estilo visual. A continuación se relacionan las clases de Qt o propias de Velneo más utilizadas.

Clase CSS	Objeto, control o subcontrol
QCheckBox	Botón check
QComboBox	Combobox
QDateEdit	Caja de edición de campo fecha
QDateTime	Caja de visualización de campo fecha y hora
QDateTimeEdit	Caja de edición de campo fecha y hora
QDialog	Ventana en cuadro de diálogo
QDockWidget	Dock
QDoubleSpinBox	Caja de edición de campo numérico con botones arriba y abajo
QFrame	Marco
QGroupBox	Caja de grupo
QHeaderView	Cabecera de rejillas y árboles
QLabel	Etiqueta de texto
QLineEdit	Caja de edición de texto en una línea
QMainWindow	Ventana principal
QMenu	Menú contextual
QMenuBar	Barra de menú (solo afecta a Windows)
QMessageBox	Ventana de mensaje
QNumberSpinBox	Caja de edición de campo numérico con botones arriba y abajo
QProgressBar	Barra de progreso
QPushButton	Botón
QRadioButton	Botón de radio
QScrollBar	Barra de scroll vertical y horizontal

QSlider	Deslizador
QSpinBox	Caja de edición de campo numérico con un botón
QSplitter	Splitter
QStatusBar	Barra de estado
QTabWidget	Separador de formularios (pestañas)
QTableView	Rejilla
QTextEdit	Caja de edición de texto multilínea
QTimeEdit	Caja de edición de hora
QToolBar	Barra de herramientas
QTooltip	Tooltip
QTreeView	Árbol visor de tabla y menú arbolado
QWidget#qt_calendar	Calendario
VBoundFieldEdit	Caja de edición de campo puntero a maestro
VCFootView	Pie de rejilla
VListBox	Listbox

Aplicar propiedades en las CSS

Una de las características especiales de las CSS de Qt usadas en Velneo es que podemos aplicar algunas propiedades de los controles u objetos directamente en la CSS. Esto nos permite cambiar el comportamiento del control de forma genérica sin programación adicional, directamente en la hoja de estilo. A continuación vemos algunos ejemplos:

Fijamos el texto a mostrar en el control cuando no tenga contenido.

```
QLineEdit#TXT_BUS {
    qproperty-placeholderText: 'Texto a buscar'; }
```

Fijamos el tamaño del icono en los botones.

```
QPushButton {
    background-color: #FFFFFF;
    border: 1px solid #E0E0E0;
    border-radius: 5px;
    color: #212121;
    font-size: 12px;
    height: 30px;
    line-height: 16px;
    text-align: center;
    qproperty-iconSize: 18px; }
```

Ocultamos la barra de estado.

```
QStatusBar {
    background-color: #FFF;
    border-top: 1px solid #CECECE;
    qproperty-visible: false; }
```

Ocultamos las líneas del grid de rejillas.

```
QTableView {
    alternate-background-color: transparent;
    background-color: #FFF;
    border: none;
    border-bottom: 1px solid #BDBDBD;
    font-weight: normal;
    gridline-color: transparent;
    selection-background-color: #3F51B5;
    selection-color: #FFF;
    qproperty-showGrid: false; }
```

Ocultamos las líneas del grid en el calendario.

```
QWidget#qt_calendar_navigationbar {
    background-color: #CECECE;
    qproperty-gridVisible: false; }
```

Fijamos el icono que se visualizará en el botón de siguiente mes del calendario.

```
QWidget#qt_calendar_nextmonth {
    background: #CECECE;
    border: 1px solid #CECECE;
    qproperty-icon: url(SENDA_ICONOS_DER.png); }
```

Aplicar iconos en las CSS

Las CSS nos permiten aplicar iconos en algunos de sus controles. Esto nos proporciona dinamismo y grandes posibilidades de personalización de la aplicación directamente aplicadas en la CSS. En el caso de los iconos tenemos la posibilidad de indicar una URL para mostrar imágenes e iconos, por lo tanto estas imágenes podrían estar en un Internet o en local.

Es más óptimo disponer de las imágenes en local. Por ese motivo hemos implementado en las CSS una senda virtual basada en un texto que reemplazaremos por la senda real en disco más el nombre de la imagen e icono. Vemos ejemplos de la CSS aplicando los iconos a diferentes botones.

En este ejemplo podemos apreciar como podemos utilizar la coma como separador de múltiples controles a los que vamos a aplicar la misma CSS, igual que sucede en las CSS de las páginas web.

```
QDateEdit::up-button, QDateTime::up-button, QDateTimeEdit::up-button,
QTimeEdit::up-button, VBoundFieldEditBrowser::up-button, VBoundFieldEdit::up-button {
    border: none;
    image: url(SENDA_ICONOS_ARR.png); }
```

```
QDateEdit::down-button, QDateTime::down-button, QDateTimeEdit::down-button,
QTimeEdit::down-button, VBoundFieldEditBrowser::down-button,
VBoundFieldEdit::down-button {
    border: none;
    image: url(SENDA_ICONOS_ABA.png); }

QComboBox::drop-down, QDateEdit::drop-down, QDateTime::drop-down,
QDateTimeEdit::drop-down, VBoundFieldEditBrowser::drop-down,
VBoundFieldEdit::drop-down, QTimeEdit::drop-down {
    border: none;
    image: url(SENDA_ICONOS_ABA.png);
    width: 13px; }

QNumberSpinBox:up-button, QDoubleSpinBox:up-button, QSpinBox:up-button {
    border: none;
    image: url(SENDA_ICONOS_ARR.png); }

QNumberSpinBox:down-button, QDoubleSpinBox:down-button, QSpinBox:down-button {
    border: none;
    image: url(SENDA_ICONOS_ABA.png); }
```

A partir de esta CSS, la técnica que utilizamos para aplicar ese icono es la siguiente. Al arrancar la aplicación descargamos los iconos que tenemos como objeto dibujo en nuestra aplicación. No es necesario utilizar ficheros adjuntos. Estas imágenes o iconos se descargan al directorio caché del client que nos garantiza acceso con capacidad de lectura y escritura.

```
// Guardar iconos en disco para usarlos en las CSS
importClass("VFile");
importClass("VImage");

// Preparar variables de trabajo
var fichero = new VFile();
var icono   = new VImage();
var iconos  = ["ABA", "ABA_BLA", "ARR", "ARR_BLA", "CRR", "DER", "DER_BLA", "IZQ", "IZQ_BLA"];
var alias   = "velneo_verp_2_app/";
var senda   = theApp.clientCachePath();

// Verificamos si el icono ya existe en el directorio del cacherun, en caso contrario se crea
for (var numIcono = 0; numIcono < iconos.length; numIcono++) {
    var fichero = new VFile(senda + iconos[numIcono]);
    if (fichero.exists() === false) {
        icono.loadResource(alias + iconos[numIcono]);
        icono.save(senda + iconos[numIcono] + ".png", "PNG");
    }
}
```

A la hora de aplicar la CSS hacemos la sustitución del texto *SENDA_ICONOS_* por la senda real del usuario *sysCacheClientPath*.

Aplicar a controles con identificadores específicos

Otra de las virtudes de las CSS es que además de poder hacer cambios generales a toda la aplicación también nos permite aplicar cambios específicos a controles concretos. Aquí cobra mayor relevancia el ser estrictos en la aplicación de los mismo identificadores para los mismos controles en todos los objetos. De esta forma podemos conseguir:

Aplicar el texto “Texto a buscar” solo en los controles cuya identificador sea *TXT_BUS*, que normalmente se utiliza en los menús.

```
QLineEdit#TXT_BUS {
    qproperty-placeholderText: 'Texto a buscar'; }
```

Aplicar un estilo diferente a los botones ampliar, reducir, buscar y menú.

```
/* BOTÓN AMPLIAR, BOTÓN BUSCAR, BOTÓN MENU y BOTÓN REDUCIR */
QPushButton#BTN_AMP, QPushButton#BTN_BUS, QPushButton#BTN_MEN, QPushButton#BTN_RED {
    background-color: transparent;
    border: 1px solid #E0E0E0;
    border-radius: 5px;
    qproperty-iconSize: 18px; }
```

Aplicar un estilo diferente a los botones ampliar, reducir, buscar y menú cuando el ratón está encima, gana el foco, está presionado o desactivado.

```
QPushButton:hover#BTN_AMP, QPushButton:hover#BTN_BUS, QPushButton:hover#BTN_MEN,
QPushButton:hover#BTN_RED {
    border: 1px solid #3F51B5; }

QPushButton:focus#BTN_AMP, QPushButton:focus#BTN_BUS, QPushButton:focus#BTN_MEN,
QPushButton:focus#BTN_RED {
    border: 2px solid #3F51B5; }

QPushButton:pressed#BTN_AMP, QPushButton:pressed#BTN_BUS, QPushButton:pressed#BTN_MEN,
QPushButton:pressed#BTN_RED {
    border: 2px solid #3F51B5; }

QPushButton:disabled#BTN_AMP, QPushButton:disabled#BTN_BUS,
QPushButton:disabled#BTN_MEN, QPushButton:disabled#BTN_RED {
    border: 1px solid #9E9E9E; }
```

Aplicar un tamaño fijo a determinados controles, en este caso los botones aceptar, cancelar, suprimir y opciones. Esto por ejemplo nos permite aplicar nuestro sistema con la unidad de referencia de 120x30 incluso aunque no cambiemos el control en los formularios, lo que supone un importante ahorro de tiempo.

```
/* BOTÓN CON TAMAÑO FIJO */
QPushButton#BTN_ACE, QPushButton#BTN_CNC, QPushButton#BTN_SUP, QPushButton#BTN_OPC {
    width: 120px; }
```

Codificación

La principal labor de un desarrollador es escribir buen código. Las mejores aplicaciones siempre tienen bajo el capó buen código. No es posible construir buenas aplicaciones, con buena nota en funcionalidad, usabilidad y rendimiento escribiendo mal código.

Cuando estamos escribiendo código debemos ser conscientes de que ese código debe durar muchos años, en ocasiones decenas de años. Además, ese código va a ser mantenido por nosotros mismos o por otros desarrolladores. Por lo tanto debemos mimarlo para que sea fácil de entender, mantener y mejorar.

Por estos motivos no debemos correr a la hora de escribir código y debemos emplear el tiempo necesario para hacer un buen naming, buenos comentarios y código de calidad. A igualdad de rendimiento el mejor código es el más sencillo de entender y mantener.

Usa una descripción del objeto clara, precisa y lo más breve posible

En la codificación todo es importante, pero para facilitar su legibilidad debemos escribir buenas descripciones de objetos que nos faciliten entender que hace el objeto o para que ha sido creado, utilizando el menor número de palabras posibles.

Tras copiar un objeto el siguiente paso debería ser modificar su descripción, cuando no lo hacemos nos con objetos que teniendo diferentes identificadores tienen la misma descripción lo que dificulta su mantenibilidad.

Comenta bien tu código

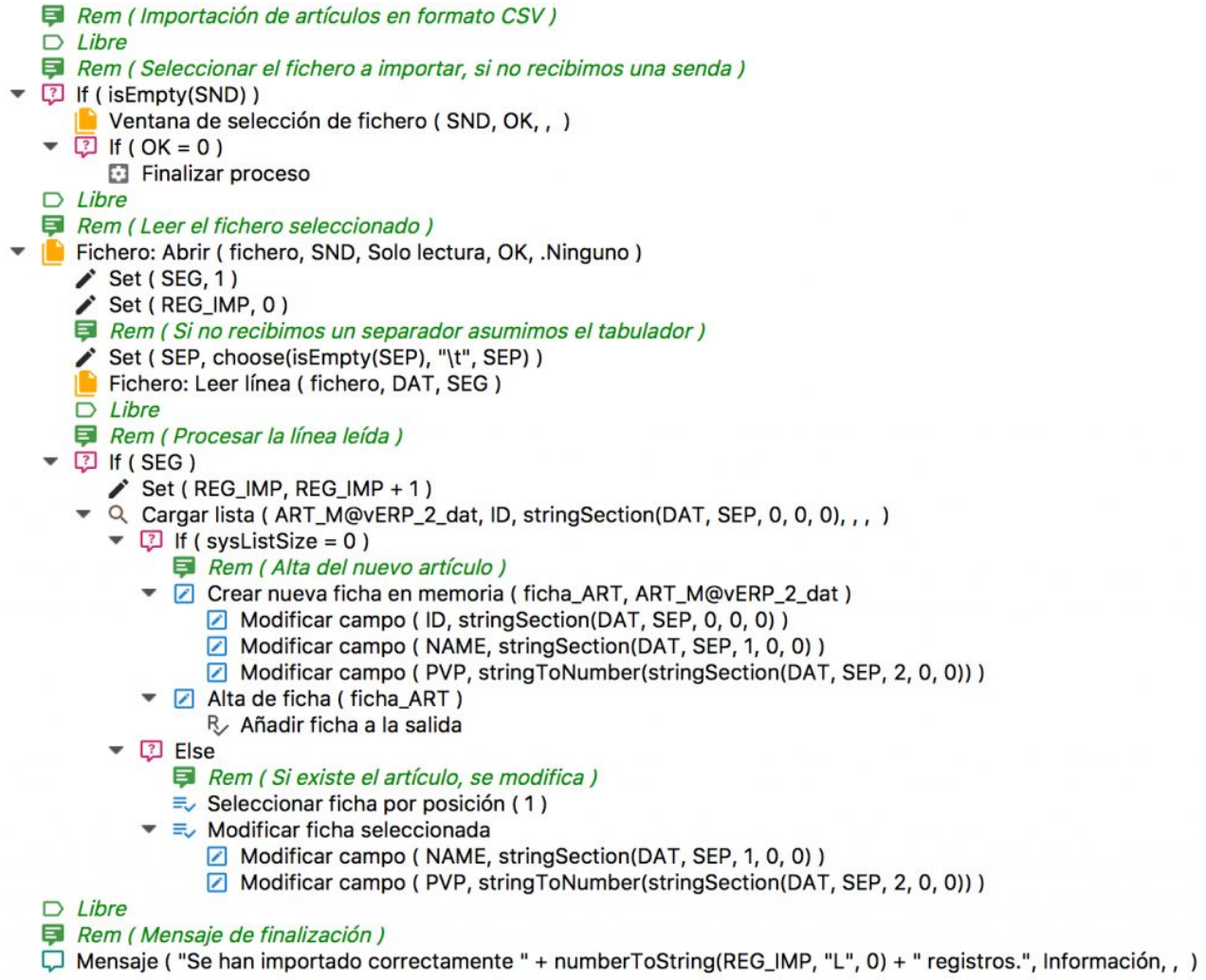
Un código sin comentarios nos obliga a leer todo el código para entender qué hace. Además de ser más lento para el programador que lo mantiene, es muy fácil que no se llegue a conclusiones acertadas ya que no siempre es obvio todo lo que se programa. Por este motivo es muy importante comentar código y, sobre todo comentarlo bien.

Comentar bien el código implica no escribir comentarios obvios que no aportan valor, ni comentarios tan extensos que cuesta lo mismo leerlos que leer e interpretar el código. Un buen comentario debe ser preciso, conciso y estar ubicado en el lugar adecuado.

Todo proceso, función o manejador de evento debería comenzar con un comentario que describa lo que hace. Es cierto que es redundante en muchos casos con la propiedad descripción de propio objeto, pero debemos entender que no siempre tenemos a la vista el código y sus propiedades, por eso es importante disponer del comentario en el inicio del código.

Aplica el mismo estilo de comentarios en todo el código

En un equipo de desarrollo no hay nada mejor que conseguir que todos los programadores escriban el código aplicando los mismos criterios para el naming, descripciones, comentarios y codificación. Con el fin de facilitar esta homogeneidad es conveniente disponer de un estilo de comentarios. A continuación se describe un estilo sencillo y fácil de recordar. A continuación se muestra un ejemplo de cómo queda el código aplicando el estilo de comentarios.



Criterios base para aplicar a los comentarios y algunas matizaciones

Los criterios base a aplicar son los siguientes:

- Los comentarios se escriben con líneas aplicando el comando *Rem*.
- Las líneas de comentarios se “comentarán” para que queden de color verde destacando del resto del código y evitando su evaluación en ninguna circunstancia.
- Si el texto del comentario es muy largo y no se ve por completo en pantalla se dividirá en varias líneas *Rem*.
- Las separaciones de código o comentarios se conseguirán empleando líneas libre antes de la línea de comentario *Rem*.
- Las líneas libres también se “comentarán” para facilitar su lectura y creación del concepto de bloque.

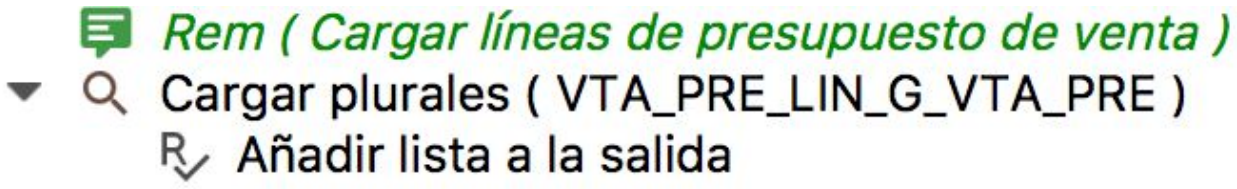
Sobre estos criterios base debemos tener en cuenta diferentes excepciones o matizaciones a la hora de aplicarlos en función de la localización.

A vamos a reparar los diferentes tipo de líneas de comentarios que espero te resulten lógicos y fácil es de aplicar si deseas usarlos en tu código.

Comentario de inicio de código

Es conveniente que el código comience con una descripción general del mismo. En muchos casos puede coincidir con la descripción del objeto: proceso, función, manejador de evento, etc.

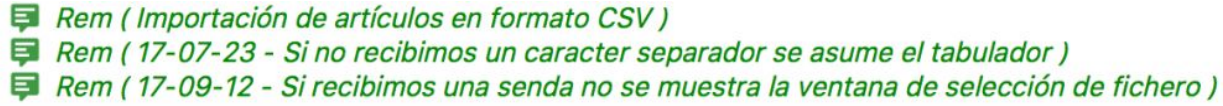
Esta línea *Rem* no requiere ninguna línea libre anterior ni posterior.



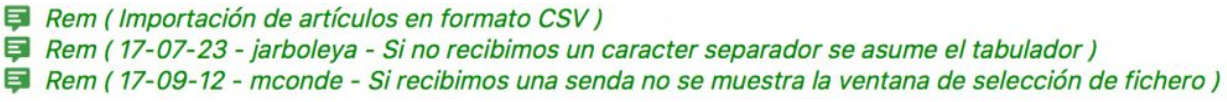
Comentario de log de cambios

Si el cambio de un código requiere ser documentado y declarado de forma explícita se añadirá tras el comentario descriptivo de inicio de código una o varias líneas de log. Estas líneas estarán separadas de la descripción inicial por una línea libre.

El formato de la línea de log será:

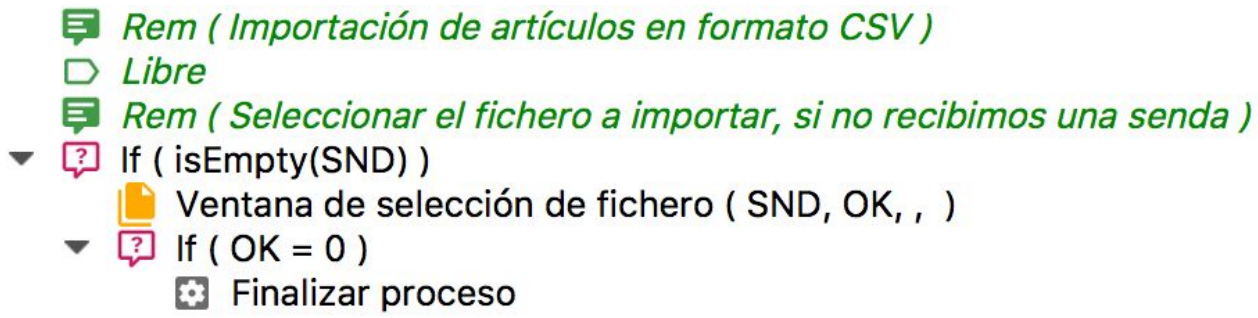


Aunque en Velneo vERP no es necesario indicar el nombre o nick del programador, si se considera importante para el desarrollo en equipo se aplicará el siguiente formato:



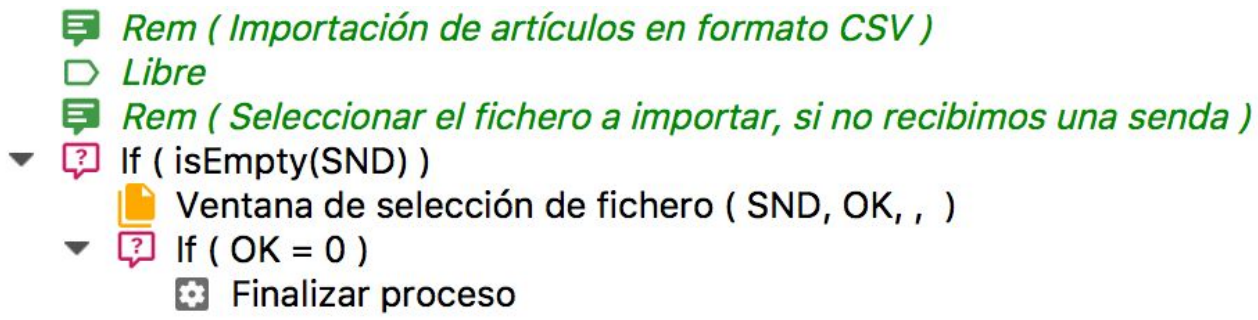
Comentario antes del código y después de la descripción

Si una vez añadida la línea *Rem* de la descripción general es necesario poner un comentario antes de la primera línea de código se separarán ambas líneas de comentarios por una libre.



Comentario inicial de un nuevo bloque en el mismo nivel

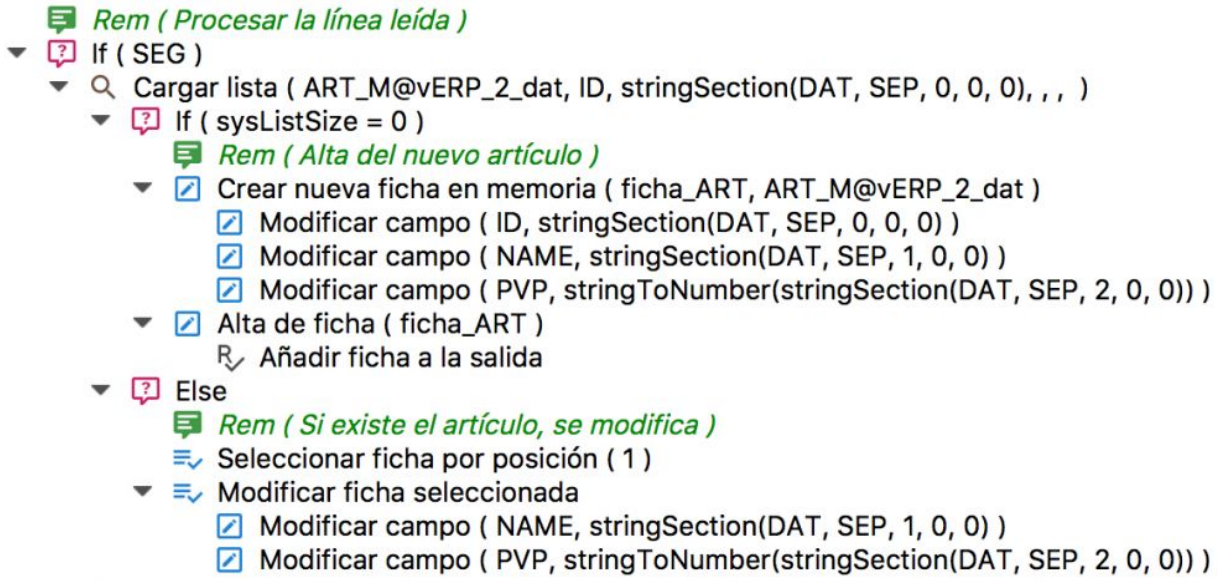
Para conseguir que ambos bloques de código queden claramente separados visualmente se aplicará una línea libre antes del comentario consiguiendo que el espacio en blanco ayude a separar ambos bloques.



Comentario en primera línea de un bloque sangrado

Cuando hay bloques de código que se escriben con sangrado debido a comandos de instrucción que generan subprocesos como ocurre con los comandos if, cargar lista, recorrer lista, etc. No será necesario poner una línea libre ya que el sangrado consigue el efecto de separación de bloques y una línea libre genera demasiada separación.

En el caso de los comando if, else y elseif las líneas de sus subprocesos si empiezan con un comentario lo harán siempre sin necesidad de incluir anteriormente una línea libre.



Comentario en primera línea tras finalizar un sangrado

Aunque la finalización de un sangrado ya genera separación visual del código, la primera línea tras recuperar el nivel de código anterior conviene que si comienza con comentario tenga una línea libre anterior ya que nos ayudará a comprender que existe código anterior al mismo nivel.

```
Else
    Rem ( Si existe el artículo, se modifica )
    Seleccionar ficha por posición ( 1 )
    Modificar ficha seleccionada
        Modificar campo ( NAME, stringSection(DAT, SEP, 1, 0, 0) )
        Modificar campo ( PVP, stringToNumber(stringSection(DAT, SEP, 2, 0, 0)) )
Libre
Rem ( Mensaje de finalización )
Mensaje ( "Se han importado correctamente " + numberToString(REG_IMP, "L", 0) + " registros.", Información, , )
```

Comentario local a un línea dentro de un bloque

Cuando un comentario se utilice para documentar la línea o línea siguientes, pero no a todas las líneas del bloque, este comentario no incluirá una línea libre anterior, ya que su función no es separar bloques de código.

```
Fichero: Abrir ( fichero, SND, Solo lectura, OK, .Ninguno )
    Set ( SEG, 1 )
    Set ( REG_IMP, 0 )
    Rem ( Si no recibimos un separador asumimos el tabulador )
    Set ( SEP, choose(isEmpty(SEP), "\t", SEP) )
    Fichero: Leer línea ( fichero, DAT, SEG )
```

No dejes líneas en blanco

Cuando editamos código en un manejador de evento, proceso, función o evento de tabla hay muchos programadores que tienen el hábito de añadir líneas vacías para luego ir rellenando el código, eso está bien siempre y cuando una vez terminado de escribir el código eliminemos las líneas “Libre” no comentadas.

```
Rem ( Si no existe el registro de existencias, se crea )
Set ( OK, fun:EXS_ALT@vERP_2_dat.dat( #ALM, #ART, #EMP ) )
Libre
Libre
Libre
Libre
Rem ( Si no existe el registro de Artículo + Proveedor, se crea )
Set ( OK, fun:ART_PRV_ALT@vERP_2_dat.dat( #ART, #PRV, #REF_PRV ) )
Libre
Libre
Libre
```

El motivo de no dejar líneas libres es doble, por un lado porque una línea no comentada se evalúa aunque sea para saber que no hay ningún comando de instrucción a ejecutar y por otro lado da la sensación de código incompleto no teniendo claro si el código está terminado o queda algo por programar.

¿Qué pasa con el código que ya tengo escrito?

Te puedes preguntar si merece la pena repasar todo el código que ya tengas escrito en una aplicación para aplicar un nuevo criterio de comentarios. En principio no es necesario invertir ese tiempo, pero lo que sí es conveniente es aplicar el nuevo criterio cada vez que edites código antiguo. Esto ayuda a saber que ese código ha sido modificado y con el paso del tiempo podrás conseguir que la mayoría de los procesos más importantes de la aplicación tengan el nuevo criterio aplicado.

Procesos

Sin duda es el objeto más poderoso de Velneo a la hora de crear funcionalidad en nuestras aplicaciones. Tiene la capacidad de ejecutarse en cualquier plano, admite cualquier origen (ninguno, ficha o lista) y cualquier destino (ninguno, ficha o lista), puede recibir un número ilimitado de parámetros y además puede devolver cualquier valor de cualquier variable local declarada en el objeto como si se tratase de parámetros de retorno. Tanta potencia requiere control para no hacer un mal uso de los procesos.

Aplica el criterio de responsabilidad única

Cuando estamos desarrollando una funcionalidad es fácil caer en la tentación de escribir un proceso largo que contiene toda la funcionalidad. Sin embargo esa es un mala praxis. Cuando más largo es un proceso más complicado es de leer, entender y mantener. Por ese motivo es conveniente usar el criterio de responsabilidad única. En lugar de tener un mega proceso es mejor:

- Crear un proceso principal que se encargue de llamar a otros procesos.
- Cada uno de los procesos llamados debería realizar una único función. No debemos confundir función con cálculo, es decir un proceso puede calcular muchos valores pero siempre que se realicen sobre la misma información.

Tampoco debemos caer en el error opuesto, es decir, atomizar tanto nuestros procesos que al final tengamos un grupo de procesos encadenados difíciles de analizar y comprender. Por ejemplo, no es fácil de mantener un proceso A que llama a un proceso B que a su vez llama a los procesos C1 y C2 y cada uno de estos llamada otros procesos. Esta jerarquía de procesos hace complicado seguirlo y mantenerlos.

Por lo tanto nuestro objetivo debe ser siempre buscar el equilibrio entre responsabilidad única y evitar el exceso de atomización, para ellos podemos recurrir a combinaciones de procesos y funciones que faciliten la legibilidad del código.

Otro problema que plantea la aplicación de la responsabilidad única es la necesidad de pasar información de un proceso a otro, algo que se evita cuando todo está en el mismo proceso. En este punto volvemos a repetir la palabra equilibrio, es decir debemos aplicar el criterio de responsabilidad única cuando un proceso va a ser llamado por otros y es mejor tener pequeñas piezas de código que realizan funciones concretas con un bajo nivel jerárquico y sin complejidades a la hora de pasar información.

Separa interfaz de proceso

Uno de los aspectos más importantes a la hora de optimizar un proceso es separar la parte de interacción con el usuario a través de la interfaz de la aplicación de reglas de negocio, cálculos y otras operaciones transaccionales automáticas que no requieren interacción.

El problema de que todo esté junto es que nos imposibilita la ejecución de un proceso en 3º plano, perdiendo la posibilidad de optimizar la parte de aplicaciones de reglas de negocio, cálculos y otras operaciones transaccionales.

Por este motivo y aunque requiera algo más de programación siempre es conveniente tener separada en un proceso independiente la parte de interfaz. Un ejemplo de buena práctica podría ser el siguiente esquema de ejecución:

- Un proceso *LLAMADOR* lanza la interfaz donde se pide la información al usuario.
- El proceso *LLAMADOR* realiza las verificaciones oportunas avisando al usuario en caso de error.
- Si todo es correcto lanza en 3º plano un proceso *CALCULADOR* que realiza las operaciones transaccionales.
- Al finalizar el proceso *CALCULADOR* en 3º plano el proceso de interfaz recupera la información relevante como el estado final, errores en caso de que los haya, registros creados, etc.
- El proceso *LLAMADOR* muestra al usuario el resultado final del proceso ejecutado.

En el ejemplo anterior solo hay 2 procesos *LLAMADOR* y *CALCULADOR*, el primero se encarga de la interacción con el usuario a través de la interfaz tanto antes como después de que finalice la transacción, mientras que el segundo proceso se ejecuta de forma optimizada en el servidor ya que no utiliza nada de interfaz.

Este mismo esquema podemos realizarlo de forma similar sustituyendo el proceso *LLAMADOR* por un formulario que realizar toda la parte de interfaz con manejadores de evento del formulario.

Evita la complejidad ciclomática

La complejidad ciclomática es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Es una de las métricas de software de mayor aceptación, ya que ha sido concebida para ser independiente del lenguaje.

Traducido a lenguaje Velneo es un valor que se calcula en base a la cantidad de niveles que se establecen en un proceso. Veamos un ejemplo:

```
Rem ( Preparación de la auxiliar )
If ( #PLA_APU_AUX= "_" )
    Set ( AUX, #AUX )
Else if ( #PLA_APU_AUX= "A" )
    Set ( AUX, ANT_AUX )
Else if ( #PLA_APU_AUX= "B" )
    Set ( AUX, VTO_AUX_CLT_CUR )
Else if ( #PLA_APU_AUX= "C" )
    Set ( AUX, VTO_AUX_CLT_ORI )
Else if ( #PLA_APU_AUX= "D" )
    Set ( AUX, VTO_AUX_BCO )
Libre
Rem ( Si no existe y está configurado, se genera la cuenta auxiliar )
If ( ( VTO_ID ) & ( #ADD_AUX_NO_EXI ) )
    Set ( AUX_OPC, #PLA_APU_AUX )
    If ( COB_PAG = "C" )
        Rem ( Vencimiento a cobrar )
        Cargar lista ( VTO_COB_C@vERP_2_dat, ID, VTO_ID, , , )
            Seleccionar ficha por posición ( 1 )
            Leer ficha seleccionada
                If ( AUX_OPC = "D" )
                    Rem ( Banco )
                    Set ( NOM, #AUX_BCO.NAME )
                    Crear nueva ficha en memoria ( ficha_AUX_C, AUX_C@vERP_2_dat )
                        Modificar campo ( PGC, PGC )
                        Modificar campo ( ID, AUX )
                        Modificar campo ( NAME, NOM )
                    Alta de ficha ( ficha_AUX_C )
                        Libre
                Else
                    Rem ( Auxiliar )
                    Set ( NOM, #AUX.NAME )
                    Crear nueva ficha en memoria ( ficha_AUX_C, AUX_C@vERP_2_dat )
                        Modificar campo ( PGC, PGC )
                        Modificar campo ( ID, AUX )
                        Modificar campo ( NAME, NOM )
```

Sin duda alguna cuando vemos un proceso así no es fácil saber que hace cada línea del proceso ya que

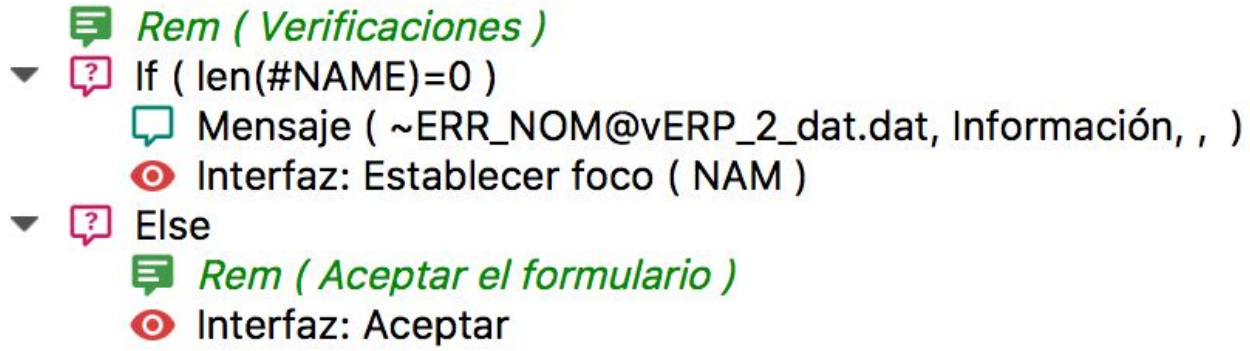
cuando estamos en 1º, 2º o 3º nivel de jerarquía todavía podemos controlarlo, pero cuando los niveles siguen creciendo nos obliga a leer todo el código secuencialmente para saber bajo qué condiciones se ejecuta las líneas de ese nivel.

Los comandos *if* y los subprocesos que generar muchos comandos de instrucción nos añaden complejidad ciclomática a los procesos, por ese motivo debemos tratar de simplificarlos al máximo y en estos casos aplicar el criterio de responsabilidad única puede ser de gran ayuda, así como el uso de funciones que simplifican la lectura del código.

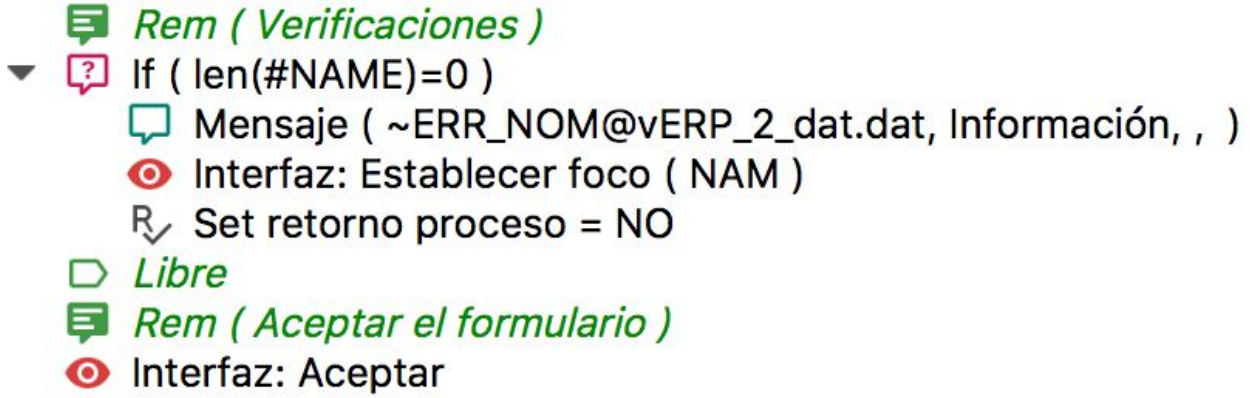
Las verificaciones primero

Cuando tenemos que hacer verificaciones para decidir si vamos o no a ejecutar un parte del código del proceso, siempre que sea posible aplica el criterio de las verificaciones primero y en caso de error finaliza el proceso. Vamos a ver 2 ejemplos de código Velneo que hacen lo mismo funcionalmente:

El primero verifica y si no hay error aceptar el formulario:



El segundo verifica, si hay error termina y en caso contrario continuar con el proceso que también acepta.

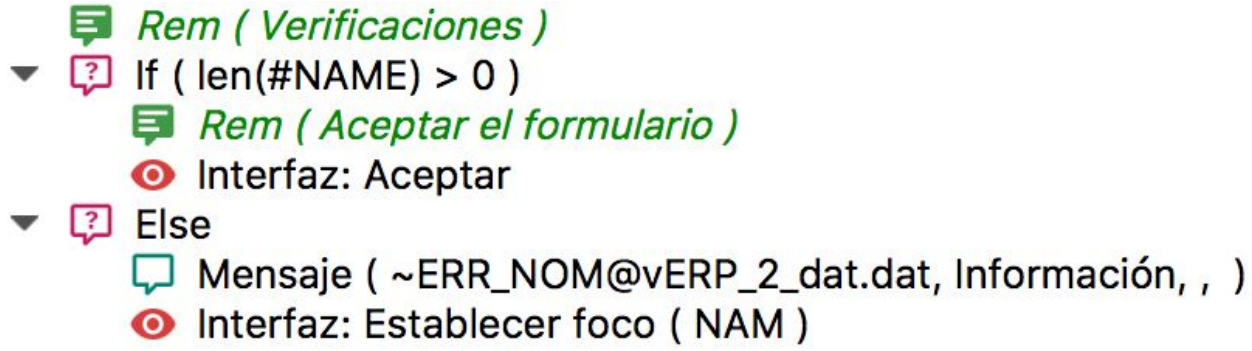


Aunque ambos 2 hacen lo mismo para el usuario final, el primero tiene más complejidad ciclomática ya que el aceptar está en un 2º nivel dentro de un else, además si mañana queremos hacer algo más dentro del else es posible que tengamos que crear más complejidad ciclomática al crear un if de 2º nivel.

La ventaja del segundo código es que verifica todo lo que tenga que verificar y si hay error muestra el mensaje y termina, y el resto del proceso ya continua en el nivel principal sin ninguna complejidad ciclomática, lo que permite añadir un if en el primer nivel.

Aunque en un proceso tan pequeño no se nota en exceso la diferencia a medida que hagamos más cosas en nuestro proceso podremos apreciar como verificar y finalizar al principio mejora la legibilidad y mantenibilidad de nuestro código.

Por último habría una 3º forma de hacerlo que aún resulta peor



En este último ejemplo además de crear más complejidad ciclomática como en el primer ejemplo, emplea una lógica inversa es decir, verifica si está bien y entonces acepta, pero en caso de error hace la parte final del código. Esto en un proceso corto todavía se puede llegar a leer, pero si el proceso ocupa más de lo que se ve en pantalla al abrir el editor sería muy complicado deducir que en la parte final del código hay un mensaje de error correspondiente a una verificación.

En definitiva, primero verificamos todo lo necesario y si todo está correcto ejecutamos la transacción. De esta forma evitamos la mala de idea de primero transacciono y si algo ha ido mal deshago transacción.

¿Cuándo es mejor un proceso que una función?

Existen diferentes motivos por los que un proceso puede ser más conveniente que una función:

- Cuando queremos ejecutar un código con un origen ficha o lista.
- Cuando queremos recuperar una ficha o lista de retorno.
- Si queremos que el código se puede ejecutar en un plano diferente al del código lanzador.
- Cuando no queremos tener límite de parámetros.
- Cuando queremos que el orden de los parámetros no influya.
- Cuando queremos que existan parámetros opcionales independientemente de su posición.
- Cuando queremos poder recuperar no un único valor de retorno sino todos los valores que sean necesarios.
- Cuando queremos que el código quede integrado en la transacción en curso aunque estemos ejecutando en 1º plano.

¿Cuándo debo usar el comando *ejecutar proceso*?

El comando de instrucción es más limitado que disparar objeto, sin embargo cuenta con la ventaja de la sencillez.

- Cuando ya estoy en el origen ficha o lista y no necesito pasarle parámetros.
- Cuando necesito ejecutar el proceso en 2º plano.

¿Cuándo debo usar el comando *disparar objeto* con un proceso?

El comando disparar objeto requiere más líneas de código que ejecutar proceso sin embargo cuenta con ventajas funcionales que nos motivan a usarlo cuando:

- Cuando quiero pasarle parámetros al proceso.
- Cuando necesito recuperar parámetros o valores calculados en el proceso ejecutado.

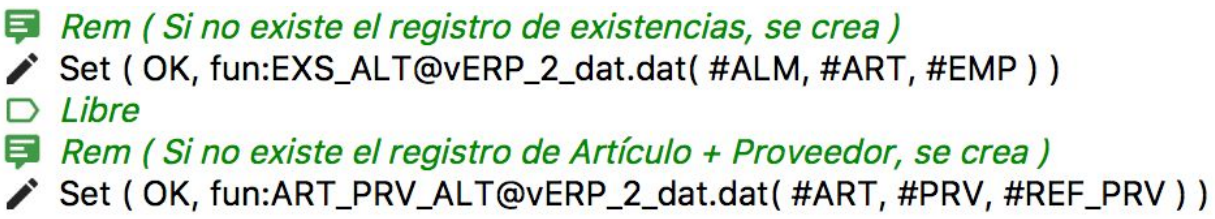
Funciones

La función es un contenedor de código sin origen. Podríamos decir que una función es como un proceso sin origen, pero la gran diferencia es que mientras el proceso puede ser ejecutado desde una acción, otro proceso, función, manejador o trigger, la función se puede ejecutar en cualquier fórmula lo que le da una potencia de ejecución que no tiene el proceso. Podríamos que una función puede ser ejecutada en cualquier ámbito de nuestra aplicación.

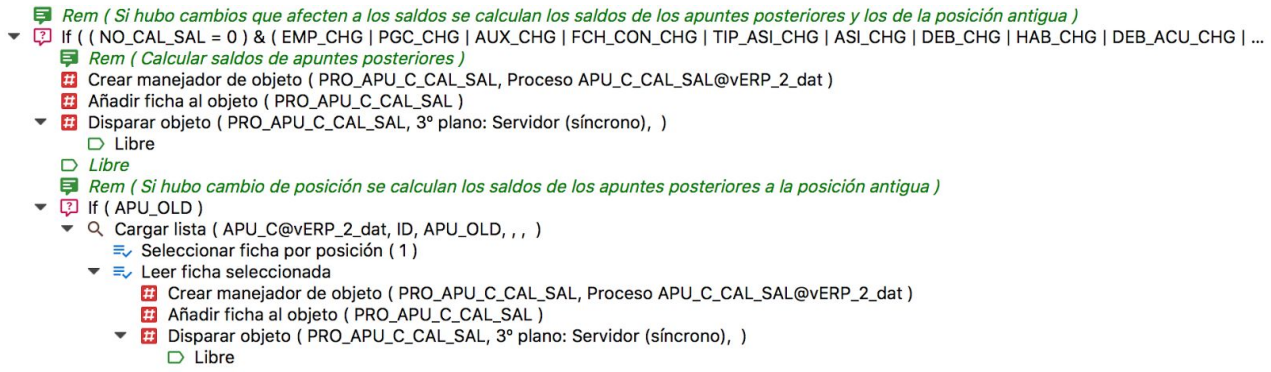
Acorta código

Uno de sus usos más interesantes es la posibilidad de evitar código repetido. Una función permite lanzar código pasándole parámetros para que ejecute una funcionalidad retornando un valor que podemos capturar para su reutilización.

Esto nos permite mover código repetido en un proceso, función, manejador de evento o evento de tabla a una función que será llamada desde diferentes puntos. La ventaja es que la llamada a una función se realiza con un única línea de código, en el siguiente ejemplo se ve la llamada a 2 funciones.

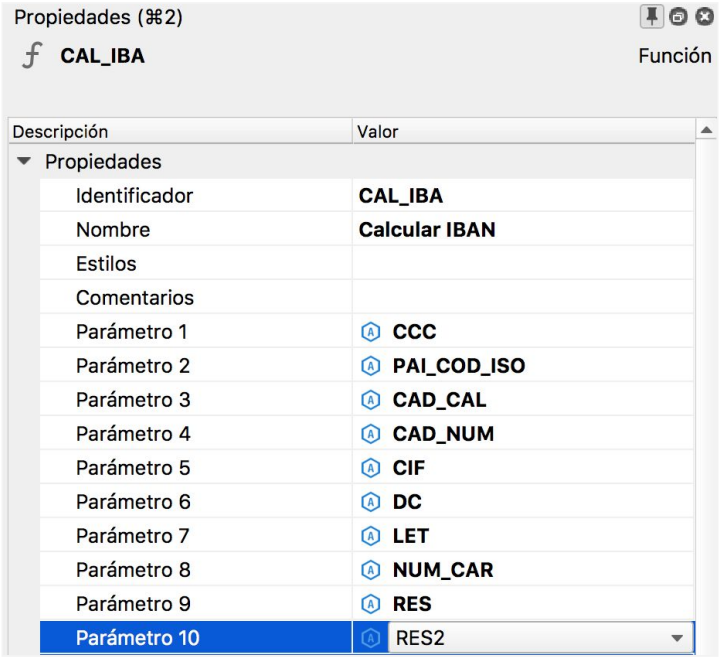


Sin embargo, ejecutar un proceso con paso de parámetros requiere varias líneas de proceso. En el siguiente ejemplo vemos la llamada a 2 procesos con los comandos de instrucción de manejador de objeto.



Ten en cuenta el número limitado de parámetros

Una de las limitaciones de las funciones es que admite un máximo de 10 parámetros. No es una gran limitación, pero debemos tenerla en cuenta a la hora de establecer la estrategia de paso de muchos parámetros a una función.



Tener una función con muchos parámetros no es cómodo, por lo que en la medida de los posible es mejor crear funciones con pocos parámetros.

Si tenemos que pasar más de 10 parámetros y no podemos hacerlo con un proceso tenemos 2 opciones, utilizar el 10º parámetros para pasar muchos valores o pasar un único parámetro con todos los valores. Esta segunda opción tiene la ventaja de que es más homogénea, es decir, no hay unos parámetros que se pasan directamente y otros agrupados sino que todos se pasan agrupados.

Ese parámetro con múltiples valores puede tener los valores aplicando un formato JSON o XML o CSV, por ejemplo. Una vez recibido el parámetro la función comienza descomponiendo dichos valores en las diferentes variables locales o en una variable global de tipo array.

Documenta los parámetros en el inicio de la función

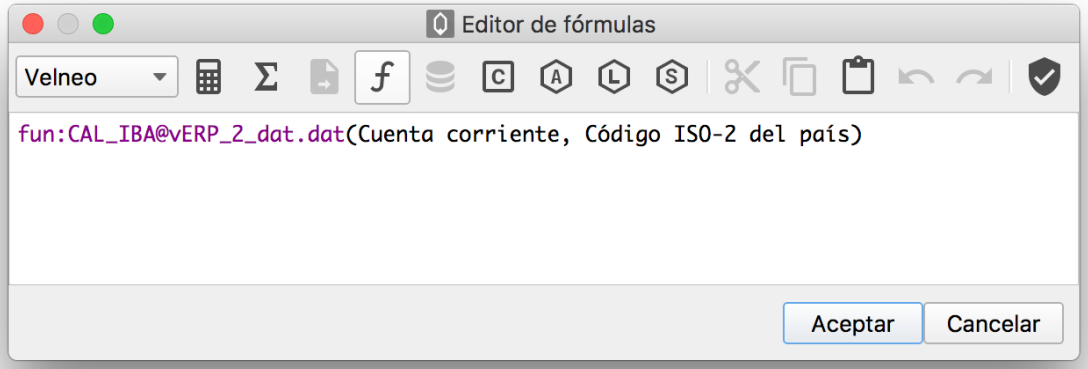
Pensando siempre en la mantenibilidad del código y que cualquier desarrollador puede necesitar usar la función es importante describir correctamente los parámetros que recibe la función y el valor que devuelve.

```
Rem ( Calcular IBAN )
Libre
Rem ( Parámetros: )
Rem ( CCC: Código de la cuenta que incluye la entidad, oficina y nº de cuenta )
Rem ( PAI_COD_ISO: Código ISO del país de origen de la cuenta )
Rem ( Retorno: Devuele el IBAN de la cuenta formado por los 2 caracteres del páis + 2 del código calculado )
Libre
Rem ( Preparar variables de trabajo )
Set ( PAI_COD_ISO, choose( isEmpty( PAI_COD_ISO ) = 0, PAI_COD_ISO, "ES" ) )
```

Usa buenas descripciones en las variables locales que sean parámetros

Cuando usamos una función tras seleccionarla de la lista nos encontraremos que en el lugar donde tenemos que escribir los parámetros nos aparecerán unos textos correspondiente a las descripciones de

las variables locales de la función. Es fundamental que esas descripciones sean lo más cortas posibles a la vez que cumplan la función de describir con precisión el dato que debemos pasar a la función. Si podemos utilizar una palabra es mejor que dos o más, pero lo más importante es que se describa bien el parámetro.



Ten en cuenta que en 1º plano genera una transacción independiente

Velneo tiene un sistema transaccional automático que se encarga de englobar en una única transacción todas las operaciones realizadas a partir de que ya exista una transacción abierta. Esto es totalmente aplicable a las funciones cuando se ejecutan en el servidor. Sin embargo, cuando una función transacciona y se ejecuta en 1º plano, su transacción no queda agrupada con la que ya estuviese en abierta en curso, sino que se crea una independiente.

Este funcionamiento debemos tenerlo en cuenta para evitar cuando sea preciso, cambiando en ese caso la función por un proceso o para forzarlo cuando nos interese cambiando un proceso por una función.

¿Cuándo es mejor una función que un proceso?

Existen diferentes motivos por los que una función es más conveniente que un proceso:

- Cuando queremos lanzar código desde una fórmula debemos usar una función.
- Si queremos que se puede ejecutar el código remotamente desde otro servidor a través de una función remota.
- Cuando queremos reducir el código de llamada a una línea.
- Cuando el código no tiene origen y necesitamos pasarle parámetros.
- Cuando queremos que genere una transacción independiente al ejecutarlo en 1º plano.

Conexiones de evento

Una gran parte de la potencia y funcionalidad de la interfaz de una aplicación viene dada por el uso de señales que nos permiten lanzar código en un momento determinado de la aplicación. Las conexiones de evento son muy potentes, pero también debemos usarlas con precaución para no abusar de ellas y producir el efecto no deseado en nuestra interfaz.

Evita el uso de la conexión pérdida de foco

Aunque es una tendencia natural usar esta señal, no es la más recomendable ya que existen muchas formas de perder el foco, cambiar de control con tabulación, con intro, pulsar una opción del botón de menú del control, pulsar una tecla de función que activa un botón, cambiar de aplicación, etc.

El problema es que no siempre nos vamos a encontrar con que el funcionamiento es el esperado, aunque detrás del comportamientos siempre hay una explicación lógica. Por este motivo es necesario trabajar con esta señal con precaución. Funciona y funciona bien, pero hay mucha casuística que se debe tener en cuenta.

Por ejemplo al pulsar un botón del formulario utilizando una tecla aceleradora, aunque se ejecuta el botón nuestro control no pierde foco ya que así es el funcionamiento de las señales en Qt. Por este motivo en ocasiones no es suficiente con la señal de pérdida de foco, además hay que hacer controles adicionales al aceptar o cerrar el formulario.

Value changed es una buena opción

Habitualmente es más recomendable usar la señal value changed que la de pérdida de foco para detectar si se han realizado cambios en los datos de un control. Es una señal que nos garantiza detectar cuando el valor del campo ha cambiado tanto si es con una opción de localidad o alta de maestro a través del botón de menú del control, como si es por una acción del usuario con el teclado a escribir un nuevo valor o con el ratón al pulsar algún botón arriba o abajo o de selección de una lista en vista de datos.

Lo que tenemos que tener presente es que si el cambio de valor del control se realiza mediante programación la señal no se disparará. Es decir, si el usuario cambia el valor manualmente si se dispara, pero si el cambio es realizado por un manejador de evento programado la señal no se va a disparar. Es fácil de gestionar, pero siempre que tengamos claro su funcionamiento.

Mejor usar “Ratón: botón soltado” que “Ratón: botón pulsado”

Estas señales aunque parezcan similares tienen una gran diferencia. El botón pulsado se dispara cuando el usuario pulsa el botón, aunque pulse y no suelte el botón del ratón la señal se habrá disparado, sin embargo si antes de soltar el botón se desplaza fuera del botón la señal ya se habría disparado cuando el usuario realmente a cambiado de opinión al tratar de desplazar el ratón fuera del botón.

Por este motivo es más recomendable utilizar la señal botón soltado que garantiza que el usuario pulsó y soltó el botón de ratón sobre el control. En el caso de controles de tipo botón ya existe una señal específica con el nombre “Botón pulsado”.

Incompatibilidad entre "Ítem: simple clic" e "Ítem: doble clic"

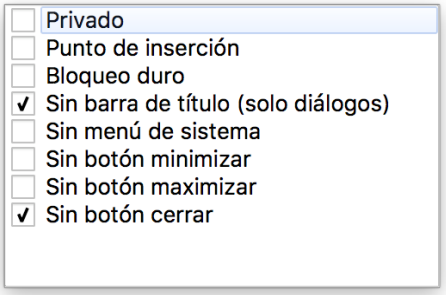
En las rejillas, por ejemplo, nos encontramos que podemos aplicar ambas señales, sin embargo debemos tener en cuenta que si declaramos las 2 señales nos vamos a encontrar con que al hacer simple clic se dispara la señal correspondiente, sin embargo hacer doble clic también se va a disparar la señal de simple clic, algo que puede no ser lo esperado, pero que debemos tenerlo en cuenta.

Onclose solo está disponible en el AUTOEXEC

Cuando tratamos de controlar el cierre de la aplicación contamos con la señal Onclose disponible en el objeto Marco denominado *AUTOEXEC*. Esta señal nos permite cancelar su cierre con el comando de instrucción "*Set retorno = NO*".

Controlar el cierre de un formulario en cuadro de diálogo

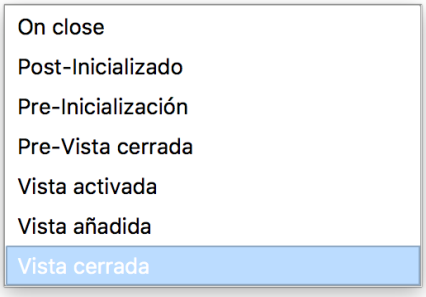
En el caso de los formularios en cuadro de diálogo aunque no disponemos de la señal podemos evitar su cierre quitando la barra de título de la ventana, o quitando el icono de cerrar



Una vez que no hay botón cerrar en el título de la ventana podemos poner un botón "*Cerrar*" o "*Cancelar*" en el formulario con el que tendremos control absoluto sobre la acción del usuario.

Controlar el cierre de un formulario en vista

En el caso de que queramos controlar el cierre de un formulario abierto en vista, podemos controlarlo a través de la señal "*Vista cerrada*".



Cuando se dispara la señal podemos utilizar funciones del API de Velneo a través de JavaScript para saber que formulario es el que está activo y por lo tanto el que está tratando de cerrar el usuario.

Manejadores de evento

Los manejadores de evento tienen la ventaja de ser código “conectado” al objeto al que pertenece, de tal forma que un manejador de evento de un formulario tiene control sobre el registro editado y todos los controles de la interfaz, y un manejador de evento de una rejilla sobre la lista de registros y sus columnas.

Al estar conectado el manejador de evento es usado para aplicar funcionalidades de avanzadas de interfaz que no podríamos lograr con procesos o funciones.

Un manejador puede llamar a otro del mismo objeto salvo en el marco *AUTOEXEC*

Un comando usado habitualmente y que nos ayuda a tener código de responsabilidad única es “*Interfaz: Ejecutar manejador de evento*”, este comando permite hacer llamadas de un manejador a otro teniendo siempre presente que comparten el registro o la lista de origen del objeto así como las variables locales y las cestas.

Sin embargo, hay una excepción, el marco *AUTOEXEC* aunque permite la creación de conexiones y manejadores de evento no permite que un manejador de evento llame a otro. En este caso particular tendremos que hacer uso de funciones o procesos para evitar código repetido.

Las variables locales son compartidas entre los manejadores

Una funcionalidad muy cómoda cuando trabajamos con los manejadores de objetos es que las variables locales declaradas en el objeto son compartidas por todos los manejadores, eso significa que podemos almacenar valores en variables locales para posteriormente utilizarlas en otro manejador. Esta funcionalidad es aplicable dentro del objeto, es decir a nivel de una tabla, un formulario, una rejilla, etc.

Debemos tener en cuenta que si un objeto está instanciado más de una vez, por ejemplo el usuario abre el formulario de dos clientes distintos, aunque el objeto es el mismo cada formulario tiene su propio ámbito de ejecución, y por lo tanto las variables de un formulario son comunes para todos sus manejadores, pero las variables locales de un formulario no son accesibles para los manejadores que están asociados al otro formulario.

Las cestas locales son compartidas entre los manejadores

Las cestas locales tienen un ámbito y una persistencia asociada a la ejecución del manejador que la crea, sin embargo sin un manejador de objeto crea una cesta local y llamamos desde ese manejador a otro manejador que utiliza un cesta con el mismo identificador, la cesta es compartida por ambos manejadores. Al finalizar la ejecución del manejador que creó la cesta el objeto será destruido de tal forma que al volver a lanzar el mismo manejador se creará una nueva cesta local.

Aplica el criterio de responsabilidad única y evita código repetido

Los manejadores de evento al igual que las otras piezas de código en Velneo permiten escribir todo el código que necesites, aunque no es recomendable hacer código largo ya que dificulta su legibilidad y mantenibilidad.

Gracias a la compartición del origen, variables y cestas, es muy sencillo evitar el código repetido en los manejadores de evento, ya que podemos hacer que un manejador llame a otro. Aplicando el mismo criterio

podemos evitar que los manejadores hagan múltiples cosas, por ejemplo verificaciones, transacciones, cambiar el estado de la interfaz, etc. Es recomendable crear pequeños manejadores de evento con responsabilidad única que son llamados desde otros manejadores de evento.

Barra de menú

La barra de menú es un objeto importante de la aplicación que en muchos casos requiere cierto grado de personalización.

No se pueden añadir o quitar opciones, pero sí limpiar y volver a construir

Para personalizar en tiempo de ejecución la barra de menú no podemos añadir o quitar opciones, la única posibilidad es limpiar la barra de menú y añadirle las opciones deseadas. Esto se puede realizar utilizando funciones de las clases API de Velneo.

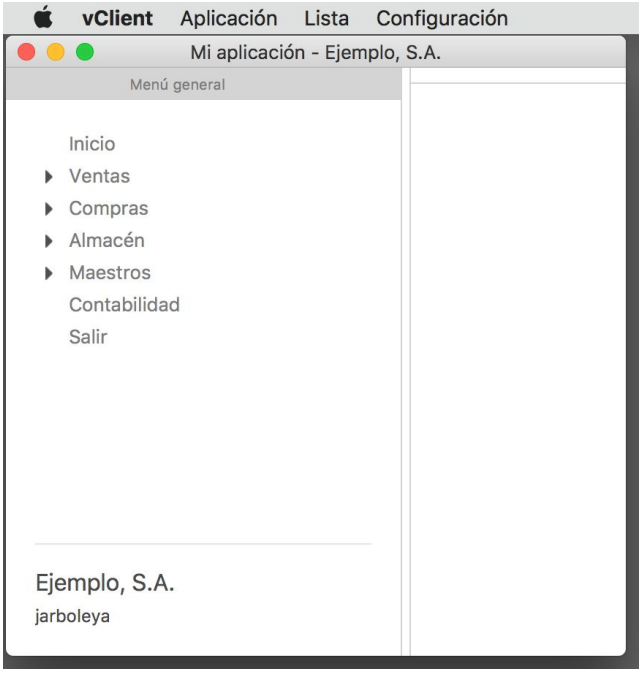
```
// Personalización de la barra de menú

// Se limpia la barra de menú
theMainWindow.clearMenuBar();

// Se añaden las opciones generales comunes para todos los usuarios
theMainWindow.addMenuToMenuBar("velneo_verp_2_app/PRN_APL");
theMainWindow.addMenuToMenuBar("velneo_verp_2_app/PRN_LST");

// El menú configuración solo para supervisores
if (theApp.isAdministrator()) {
    theMainWindow.addMenuToMenuBar("velneo_verp_2_app/PRN_SUP");
}

// Se añaden una opción "..." como punto de inserción
theMainWindow.addMenuToMenuBar("velneo_verp_2_app/PRN_INS");
```



Menús

Los menús y sus opciones son los elementos principales de interacción del usuario, ya que disparan las acciones que quieren ejecutar para llevar a cabo las tareas. Como siempre cuantas menos opciones tenga un menú es mejor para el usuario sin que por ello debamos incrementar el número de niveles del menú por no tener demasiadas opciones en cada nivel.

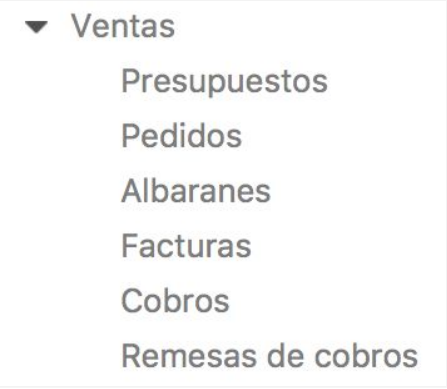
Minimiza las opciones de tus menús

Tanto en los menús arbolados como en las barras de menú hay que tratar de utilizar el menor número de opciones posibles. No es necesario establecer un número mínimo o un número máximo, pero sí que es conveniente que el nº de opciones no obligue al usuario a leer demasiado para encontrar la opción deseada.

El orden de las opciones de menú es la clave

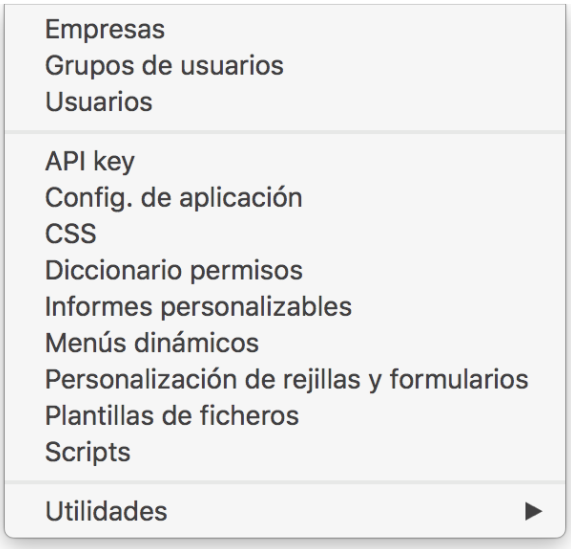
En el menú principal arbolado de la captura vemos el orden ventas, compras, almacén, maestros y contabilidad. Este orden sigue el criterio de uso, es decir, la opción más usada al principio y al final la menos usada.

En el menú de Ventas el orden aplicado sigue el criterio de ciclo funcional, es decir el orden que tiene un ciclo de ventas desde el presupuesto hasta el cobro de la factura mediante remesa.



En el menú de maestros, por ejemplo se sigue el criterio alfabético, ya que hay muchas opciones.

En el submenú de configuración de la barra de menús se sigue el criterio alfabético pero separando en 2 grupos las opciones, en primer lugar las más usadas y luego las de uso menos frecuente.



En definitiva, podemos usar diferentes criterios de ordenación, pero siempre siguiendo una lógica comprensible por el usuario de forma sencilla y lógica.

Crea menú de botón para cada maestro

En los campos de edición punteros a maestro debemos declarar siempre un menú de botón que permita

realizar las funciones estándar de localización, alta y edición.

Un aspecto fundamental de estos menús es que utilicen teclas aceleradoras que permitan al usuario ejecutar las opciones de forma rápida y directa a través del teclado sin usar el ratón. Este menú se puede desplegar con *Mayúscula+F4* y luego usar las teclas para seleccionar la opción y finalmente pulsar *Intro*, sin embargo es mucho más sencillo y directo usar la tecla aceleradora *F5* para localizar, *F6* para crear un nuevo registro y *F7* para editar el registro seleccionado.

Es fundamental para el usuario saber que las mismas teclas aceleradoras ejecutarán la misma acción en todos los casos.

Utiliza el mismo icono en todos los botones de menú

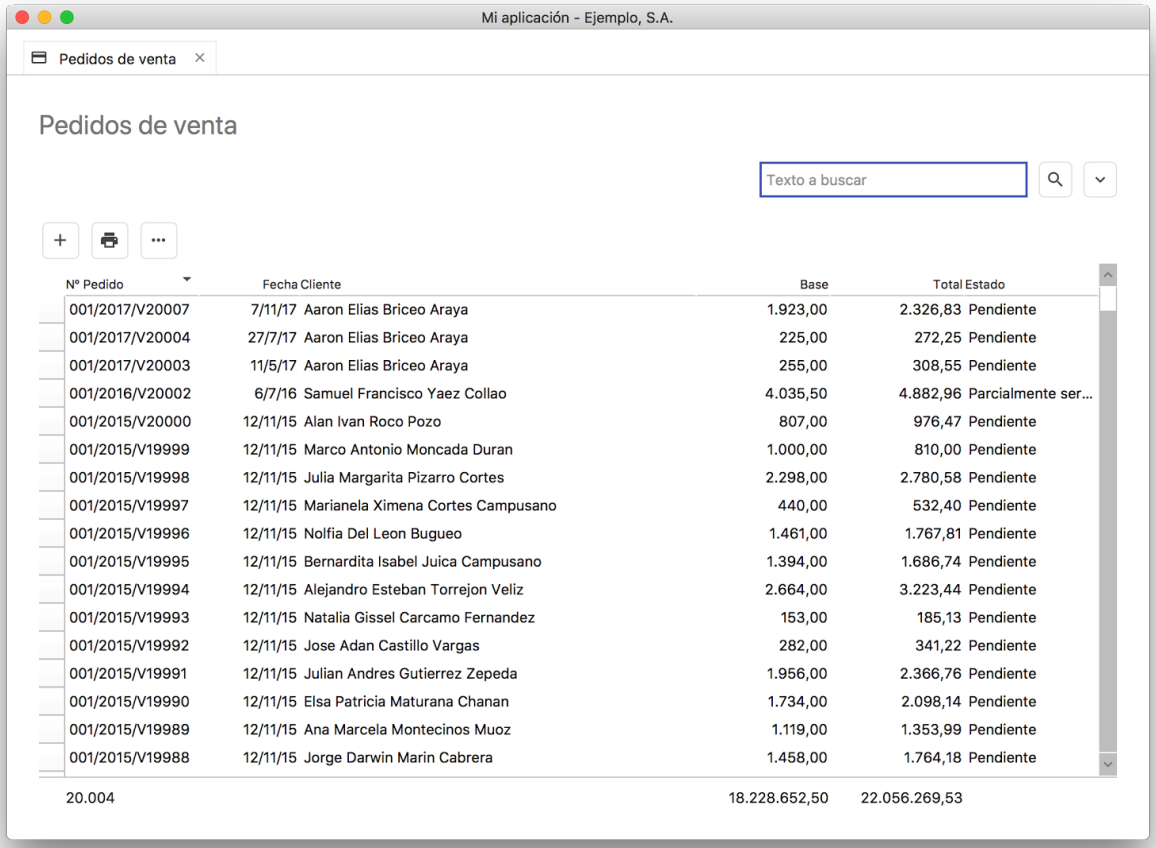
Para facilitar al usuario la comprensión de la interfaz recomendamos usar siempre el mismo icono que represente el menú de botón contextual en todos los controles, evitamos al usuario pensar que hará el botón.

Toolbars

Las barras de herramientas son muy utilizadas para aportar funcionalidad adicional en los objetos de lista como alternadores y rejillas, además de la toolbar general que se pueden añadir a los docks del marco.

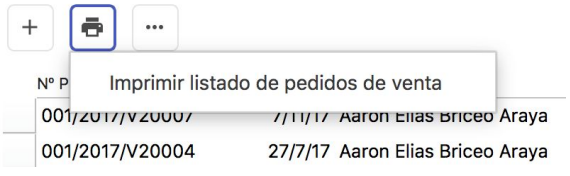
Utiliza iconos

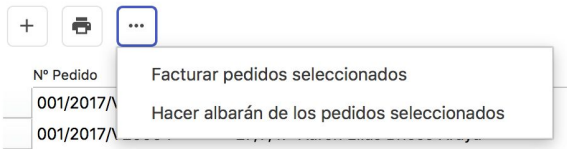
Las toolbar suelen tener opciones relacionadas con operaciones transaccionales como altas, bajas y modificaciones, impresión de informes o procesos que realizan acciones específicas como cálculos, generación de documentos, etc.



Nº Pedido	Fecha	Cliente	Base	Total	Estado
001/2017/V20007	7/11/17	Aaron Elias Briceo Araya	1.923,00	2.326,83	Pendiente
001/2017/V20004	27/7/17	Aaron Elias Briceo Araya	225,00	272,25	Pendiente
001/2017/V20003	11/5/17	Aaron Elias Briceo Araya	255,00	308,55	Pendiente
001/2016/V20002	6/7/16	Samuel Francisco Yaez Collao	4.035,50	4.882,96	Parcialmente ser...
001/2015/V20000	12/11/15	Alan Ivan Roco Pozo	807,00	976,47	Pendiente
001/2015/V19999	12/11/15	Marco Antonio Moncada Duran	1.000,00	810,00	Pendiente
001/2015/V19998	12/11/15	Julia Margarita Pizarro Cortes	2.298,00	2.780,58	Pendiente
001/2015/V19997	12/11/15	Marianela Ximena Cortes Campusano	440,00	532,40	Pendiente
001/2015/V19996	12/11/15	Nolfia Del Leon Bugueo	1.461,00	1.767,81	Pendiente
001/2015/V19995	12/11/15	Bernardita Isabel Juica Campusano	1.394,00	1.686,74	Pendiente
001/2015/V19994	12/11/15	Alejandro Esteban Torrejon Veliz	2.664,00	3.223,44	Pendiente
001/2015/V19993	12/11/15	Natalia Gissel Carcamo Fernandez	153,00	185,13	Pendiente
001/2015/V19992	12/11/15	Jose Adan Castillo Vargas	282,00	341,22	Pendiente
001/2015/V19991	12/11/15	Julian Andres Gutierrez Zepeda	1.956,00	2.366,76	Pendiente
001/2015/V19990	12/11/15	Elsa Patricia Maturana Chanan	1.734,00	2.098,14	Pendiente
001/2015/V19989	12/11/15	Ana Marcela Montecinos Muoz	1.119,00	1.353,99	Pendiente
001/2015/V19988	12/11/15	Jorge Darwin Marin Cabrera	1.458,00	1.764,18	Pendiente
20.004			18.228.652,50	22.056.269,53	

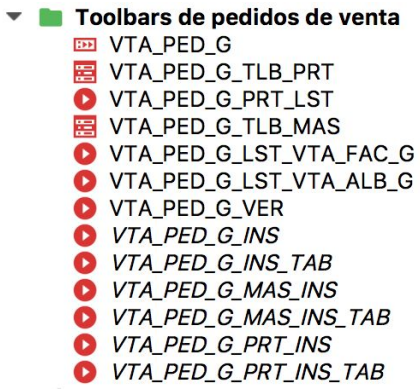
Como habitualmente se incluyen acciones estándar y para conseguir una interfaz sencilla en el sistema Velneo las toolbars se declaran con solo iconos y sin texto. Aunque al pulsar sobre el botón de la toolbar según sea de tipo informe o más opciones, por ejemplo se abrirá un menú con el detalle de opciones asociadas al botón. Veamos un par de ejemplos del menú que despliega al pulsar sobre el botón de informes o más opciones.



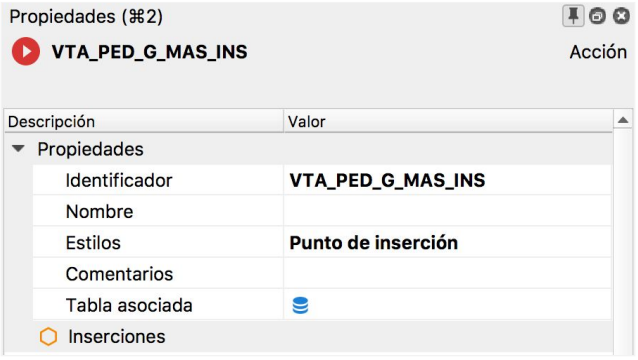


Si desarrollas una aplicación estándar, añade una acción con punto de inserción en cada menú

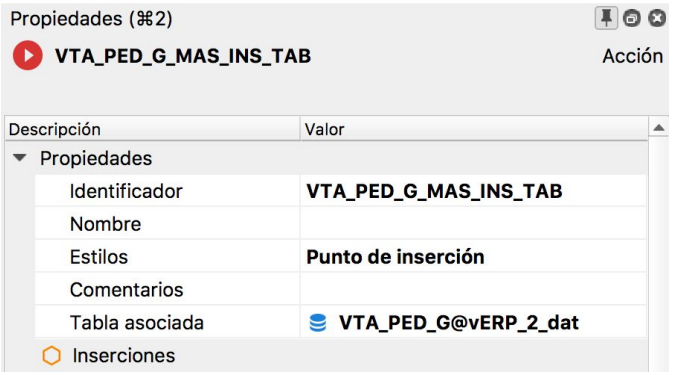
Si desarrollas aplicaciones a medida no tienes que preocuparte por facilitar la personalización desde otro proyecto que herede tu solución, sin embargo si desarrollas aplicaciones estándar o tienes un núcleo único para todos tus clientes, o una solución sectorial que luego personalizas para cada cliente, debes tener en cuenta que las toolbars sean “personalizables”. Es decir, que puedas añadirle opciones desde proyectos que lo heredan. En la siguiente captura se puede apreciar como existen opciones que aparecen en itálica que representan puntos de inserción que han sido añadidos en los menús que a su vez están incluidos en la toolbar.



En la captura se puede apreciar que existen 2 puntos de inserción para cada menú, uno de ellos solo lleva el sufijo *_INS* y representa un punto de inserción sin origen.

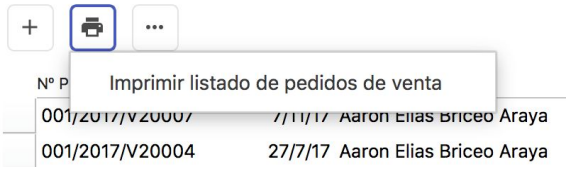


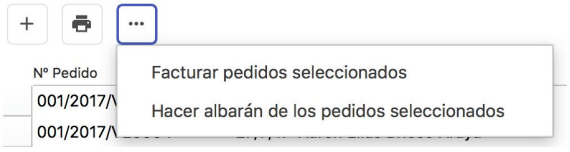
Si vamos a colocar la toolbar en un objeto con origen como puede ser un alternador o rejilla, conviene añadir una segunda opción de punto de inserción con origen de la tabla que es el que lleva el sufijo *_INS_TAB*.



Agrupa los botones por funcionalidad

Las toolbar con muchos botones son más complejas de usar y tienen más carga cognitiva para el usuario, es como tener un menú que contenga todas las opciones en el nivel principal, sin submenús. Aplicando el mismo criterio es preferible tener las opciones de informes o de más opciones que ejecutan cálculos o procesos agrupadas en un botón único que las unifica.





Con este sistema el usuario cada vez que ve una toolbar ya sabe con el primer vistazo si tiene alguna opción de impresión o de cálculos.

Acciones

Este objeto es usado habitualmente para añadir opciones en menús, barra de menús y toolbars, aunque también es posible ejecutarlo desde código. Es un objeto bastante simple que apenas tiene configuración, aunque sí debemos ser precisos con la descripción del objeto ya que se convertirá en el texto de las opciones de menú.

Evita el uso de iconos

Como ya hemos comentado en otros apartados, puede ser interesante aplicar algún icono siempre que aporte información relevante, sin embargo, las opciones de menús y toolbars suelen necesitar un texto más descriptivo por lo que puede ser redundante el uso de texto e iconos a la vez. En estos casos puede ser preferible evitar el uso de iconos.

Marco *AUTOEXEC*

Es el objeto de acceso a la interfaz de la aplicación.

Aplicar CSS en el evento *Pre-Inicialización*

El lugar adecuado para aplicar CSS generales a una aplicación es el marco *AUTOEXEC*. Y el lugar concreto es el manejador de *Pre-Inicialización*.

Si aplicamos las CSS en el evento Post-Inicializado la aplicación de CSS también funciona, pero tendremos un efecto secundario no deseado y es que primero se pintarán los controles con el estilo estándar de arranque y una vez pintados se ejecutará el manejador de evento Post-Inicializado que al aplicar las CSS provoca un repintado. Sin embargo, si lo hacemos en el Pre-Inicialización ya se aplica las CSS en el pintado inicial. Esto mismo es aplicable también en cualquier objetos de vista tanto de ficha como de lista.

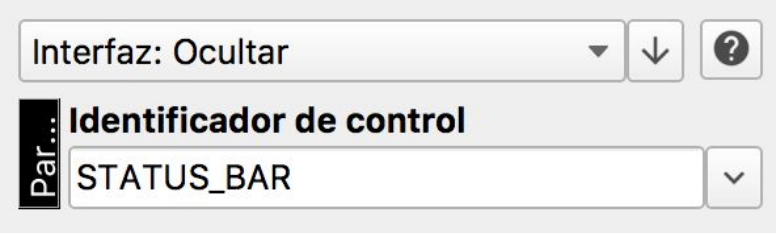
Permitir configurar que la barra de estado se puede mostrar u ocultar

La barra de estado es un espacio muy útil en algunas ocasiones para mostrar mensajes o barras de progreso, sin embargo podemos tener algunas circunstancias en las que no deseamos que sea visible, por ejemplo cuando queremos una interfaz lo más limpia posible y también cuando estemos trabajando con una resolución muy baja en formato 16:9 donde tenemos muy poco espacio vertical y cada píxel cuenta.

En esos casos puede interesarnos quitar la barra de estado, o hacerlo de modo temporal o incluso dejarlo configurable a nivel general de aplicación o de usuario. El siguiente código JavaScript lo permite de forma sencilla.

```
// Ocultar/Mostrar la barra de estado
if (theMainWindow.isStatusBarVisible()) {
    theMainWindow.hideStatusBar();
} else {
    theMainWindow.showStatusBar();
}
```

Aunque también podemos gestionarlo con código nativo.



Formularios de edición

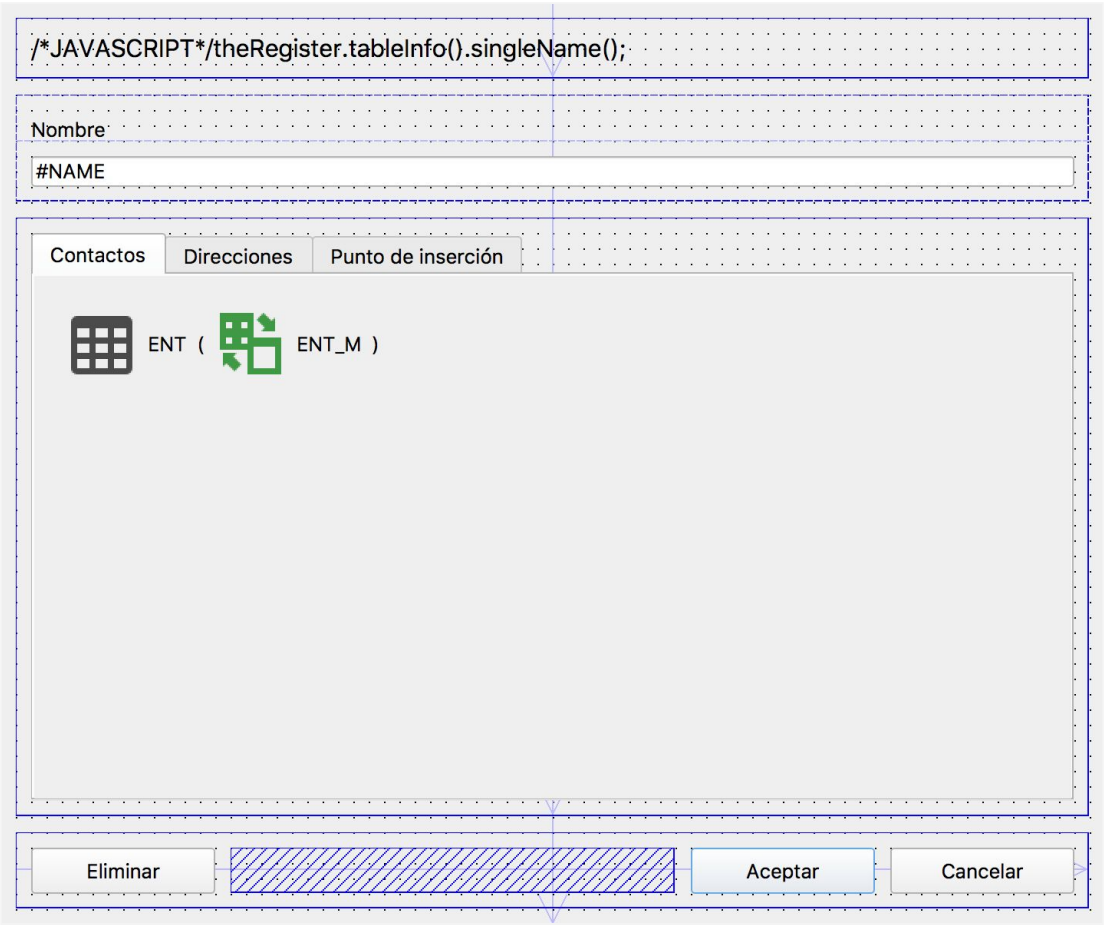
El objeto más importante a la hora de crear la interfaz de usuario de las aplicaciones. Debemos crearlos con el mayor mimo posible, cuando al máximo los detalles, porque para el usuario todo lo que tiene a la vista es muy importante.

Identificadores

Los controles incluidos en los formularios siguen los siguientes criterios de nomenclatura que podemos ver en la siguiente tabla que contiene los controles más habituales de un formulario.

Identificador	Descripción
BTN_AVA_CTL	Botón oculto que permite al usuario avanzar de control con la tecla *Intro*.
LAY_TIT	Layout del título.
TXT_TIT	Título del formulario.
LAY_CAB	Layout de cabecera.
TXT_ID	Texto estático o nombre del campo *ID*.
ID	Control de edición del campo *ID*.
TXT_NOM	Texto estático o nombre del campo *NAME*.
NOM	Control de edición del campo *NAME*.
LAY_DET	Layout detalle.
SEP	Control de tipo separador formularios.
LST	Control de vista de datos ubicado directamente en el formulario.
LAY_BTN	Layout botones (en el pie)
BTN_ACE	Botón aceptar.
BTN_CNC	Botón cancelar.
EXP_BTN	Expansor entre botones.
BTN_SUP	Botón eliminar.
BTN_OPC	Botón opciones.

Un formulario prototipo con estos controles sería el siguiente:



Resolución mínima

Diseñaremos los formularios para que los controles se vean correctamente con una resolución de 1366x768. Si hay problemas de espacio dividiremos la información en más subformularios.

Tamaño del formulario

Los formularios tendrán un ancho múltiplo de la unidad de referencia, es decir, 120, 240, 360, 480, 600, 720, 840 y 960. En principio trataremos de no maquetar formularios que superen este tamaño.

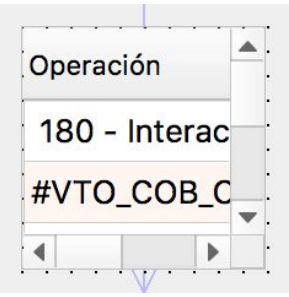
El alto de un formulario también será múltiplo de la unidad de referencia 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330, 360, 390, 420, 450, 480, 510, 540, 570, 600, 630, 660, 690 y 720. Trataremos de no superar los 720px en altura ya que la resolución mínima es de 768 y debe entrar la barra de título de la ventana más la barra de estado si está visible.

Tamaño de los subformularios

Los subformularios al visualizarse dentro del área del formulario principal no necesitar tener una dimensión específica asociada a la unidad de referencia, aunque siempre es conveniente aplicar los mismos criterios de tamaño.

Para subformularios que muestren una vista de datos lo mejor es utilizar una dimensión muy reducida ya que el control crecerá para ocupar todo el área. Por defecto ponemos el control de vista de datos de

100x100 y el formulario de 120x120.



Tamaños y alineamientos de los tipos de control

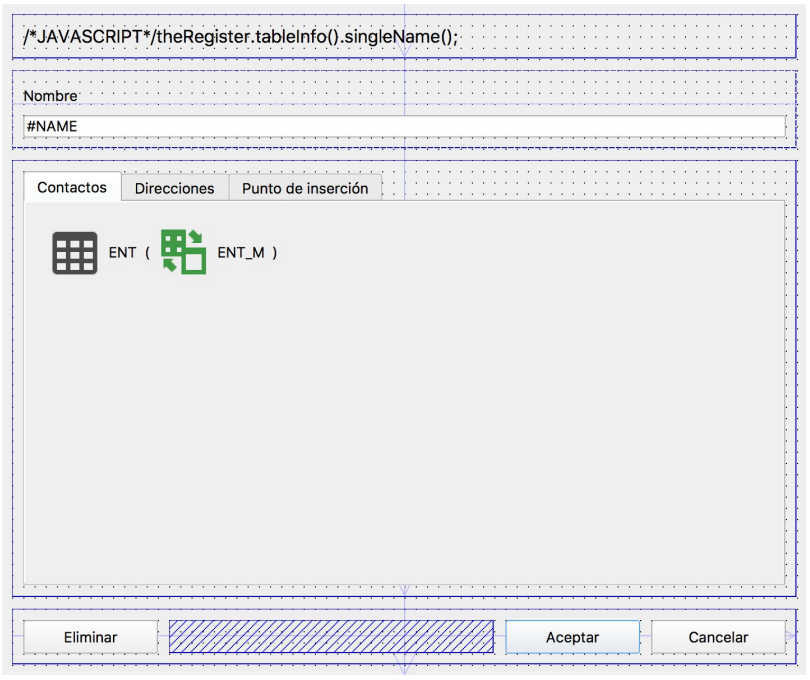
En la siguiente tabla se detallan los tamaños de anchos, altos, los anchos y altos de layout y la alineación de los diferentes tipos de control que podemos usar en los formularios. Estos son tamaños base pueden ajustarse para conseguir un mejor alineamiento de todos los controles en el formulario

Tipo de control	Ancho	Alto	Ancho layout	Alto layout	Alineación
Texto estático a la izquierda	120	20	Por defecto	Por defecto	Izquierda
Texto estático arriba	120	20	Por defecto	Por defecto	Según control
Nombre de campo a la izquierda	120	20	Por defecto	Por defecto	Izquierda
Nombre de campo arriba	120	20	Por defecto	Por defecto	Según control
Edición alfabética (10 caracteres)	120	20	Fijo	Por defecto	Izquierda
Edición alfabética (40 caracteres)	240	20	Por defecto	Por defecto	Izquierda
Edición numérica (2 enteros, 2 decimales)	60	20	Fijo	Por defecto	Derecha
Edición numérica (9 enteros, 2 decimales)	120	20	Fijo	Por defecto	Derecha
Edición fecha	120	20	Fijo	Por defecto	Derecha
Edición hora	90	20	Fijo	Por defecto	Derecha
Edición fecha y hora	180	20	Fijo	Por defecto	Izquierda
Botón	120	30	Fijo	Por defecto	Centrado
Combobox	120	20	Por defecto	Por defecto	Izquierda

Hay que tener en cuenta que estos tamaños se verán afectados por dos factores, el primero de ellos es que los tipos de ancho y alto por defecto y proporcional se ajustarán al área disponible en el formulario y el segundo es la aplicación de CSS que ajustará los anchos y altos de determinados controles.

Layouts

Para mostrar la configuración de los layouts vamos a utilizar un nuestro formulario prototipo:



El objeto formulario generalmente tiene definido un layout con la siguiente configuración:

Tipo de layout	**Vertical**
Alineamiento horizontal	**Izquierda**
Alineamiento vertical	**Arriba**
Espaciado	**0**
Margen izquierdo	**20**
Margen derecho	**20**
Margen superior	**20**
Margen inferior	**20**

El layout de título tiene la siguiente configuración:

Tipo de layout	**Vertical**
Espaciado	**10**
Margen izquierdo	**-1**
Margen derecho	**-1**
Margen superior	**-1**
Margen inferior	**10**
Ancho en layout	**Por defecto**
Alto en layout	**Por defecto**

El layout de cabecera tiene la siguiente configuración:

Tipo de layout	**Grid**
Espaciado	**10**
Margen izquierdo	**-1**
Margen derecho	**-1**
Margen superior	**-1**
Margen inferior	**10**
Ancho en layout	**Por defecto**
Alto en layout	**Por defecto**

El layout de detalle tiene la siguiente configuración:

Tipo de layout	**Grid**
Espaciado	**10**
Margen izquierdo	**-1**
Margen derecho	**-1**
Margen superior	**-1**
Margen inferior	**10**
Ancho en layout	**Proporcional**
Alto en layout	**Proporcional**

El layout de botones tiene la siguiente configuración:

Tipo de layout	**Horizontal**
Espaciado	**10**
Margen izquierdo	**-1**
Margen derecho	**-1**
Margen superior	**-1**
Margen inferior	**10**
Ancho en layout	**Por defecto**
Alto en layout	**Por defecto**

Título del formulario

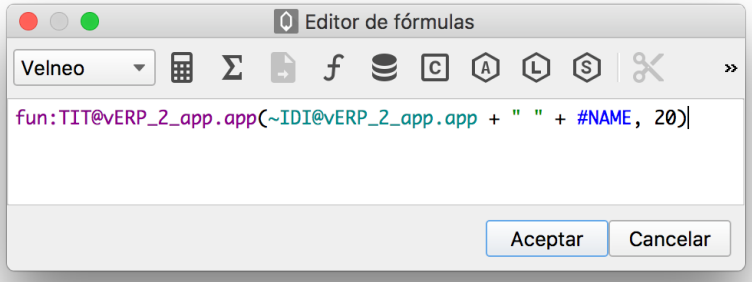
El control *TXT_TIT* de tipo texto estático situado en la parte superior del formulario asume automáticamente el nombre de singular de la tabla de origen del formulario, ya que ese es el valor por defecto más habitual. Para atrapar el nombre singular de la tabla se usa una fórmula JavaScript en la propiedad contenido del control.

Contenido	**/*JAVASCRIPT*/theRegister.tableInfo().singleName();**

Título de la pestaña

Para unificar el criterio del título del formulario que se muestra en las pestañas se utilizamos la propiedad título opcional.

Con el objetivo de que los título no sean demasiado grandes para conseguir que el tamaño de las pestaña permanezca reducido y que puedan visualizarse un buen número de ellas utilizaremos una función encargada de recortar al número de caracteres que le indiquemos el texto a mostrar.



La función es muy sencilla, se encarga de cortar el texto y añadirle unos puntos suspensivos "..." en el caso de que el texto sea mayor que longitud indicada.

```
Rem ( Devuelve el título recortado a la longitud máxima )
Set dato de retorno ( "" + choose( LON = 0, TXT,  choose(len(TXT) < LON, TXT, left(TXT, LON) + "...")) )
```

Como primer parámetro de la función le pasamos el texto. En el caso de tablas maestras se le pasa una constante cuyo texto puede ser fácilmente traducido.

Como segundo parámetro de la función le pasamos la longitud a la que debe cortar el texto, si se le pasa longitud 0 se aplica el texto completo.

¿Cuándo en vista o en cuadro de diálogo?

Como norma general deberíamos intentar que los formularios se muestre siempre en vista ya que tienen la ventaja de permitir al usuario interactuar con otros formularios y vistas sin necesidad de cerrar el previamente el formulario.

Cuando no nos interese que el usuario pueda hacer otras acciones en la aplicación sin antes cerrar el formulario en curso debemos poner a verdadero la propiedad *siempre cuadro de diálogo*, lo que nos garantiza que el usuario solo podrá trabajar en ese formulario.

Hay otros casos en los que al ser un formulario muy pequeño con poca información no queda bien si se muestra en vista, en esos casos conviene declararlos también como *siempre en cuadro de diálogo* verdadero.

Si lo disparas desde un proceso sale en cuadro de diálogo

Cuando ejecutamos un formulario o cualquier objeto de vista desde un proceso se nos visualizará en cuadro de diálogo independientemente de lo que tenga el formulario en la propiedad *siempre cuadro de diálogo*.

Que la ficha editada en el formulario esté bloqueada o no en el momento de mostrarse el formulario dependerá si hemos leído la ficha en modo de lectura/escritura o solo lectura. En caso de lectura/escritura se creará una transacción y la ficha estará bloqueada para otros usuarios de la misma forma que ocurriría si otro proceso estuviese transaccionando con ese registro.

Mostrar un formulario en vista lanzado desde un proceso

Existe una alternativa para visualizar objetos de vista lanzados desde un proceso en vista evitando que se muestren en cuadro de diálogo. Esto es útil por ejemplo si deseamos desde un proceso abrir varias pestañas en vista con diferentes registros en formulario o listas en rejilla.

El siguiente script es un ejemplo de como podemos conseguirlo.

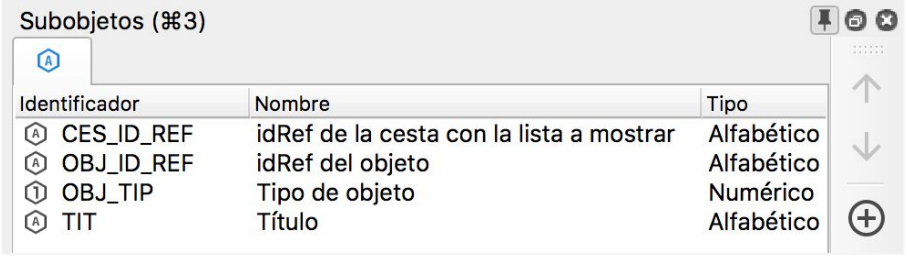
```
// ------------------------------------------------
// Mostrar una ficha o registro en una nueva pestaña
// ------------------------------------------------

// Cargar los parámetros recibidos por variables locales
var titulo      = theRoot.varToString("TIT");
var objetoTipo  = theRoot.varToString("OBJ_TIP");
var objetoIdRef = theRoot.varToString("OBJ_ID_REF");
var cestaIdRef  = theRoot.varToString("CES_ID_REF");

// Cargamos la lista de la cesta
var lista = new VRegisterList(theRoot);
theApp.getBasket(cestaIdRef, lista);

// Crear una nueva pestaña con el título y el objeto por cada registro recibido
for (var numRegistro = 0; numRegistro < lista.size(); numRegistro++) {
    registro = lista.readAt(numRegistro);
    var vista = theMainWindow.addDataView(objetoTipo, objetoIdRef, registro);
    vista.setTitle(titulo);
}
```

Podemos crear un proceso con este script JavaScript que recibirá estos 4 parámetros:



Hay que destacar que debemos pasar en un objeto cesta global el registro que queremos editar en el formulario.

Formularios de menú

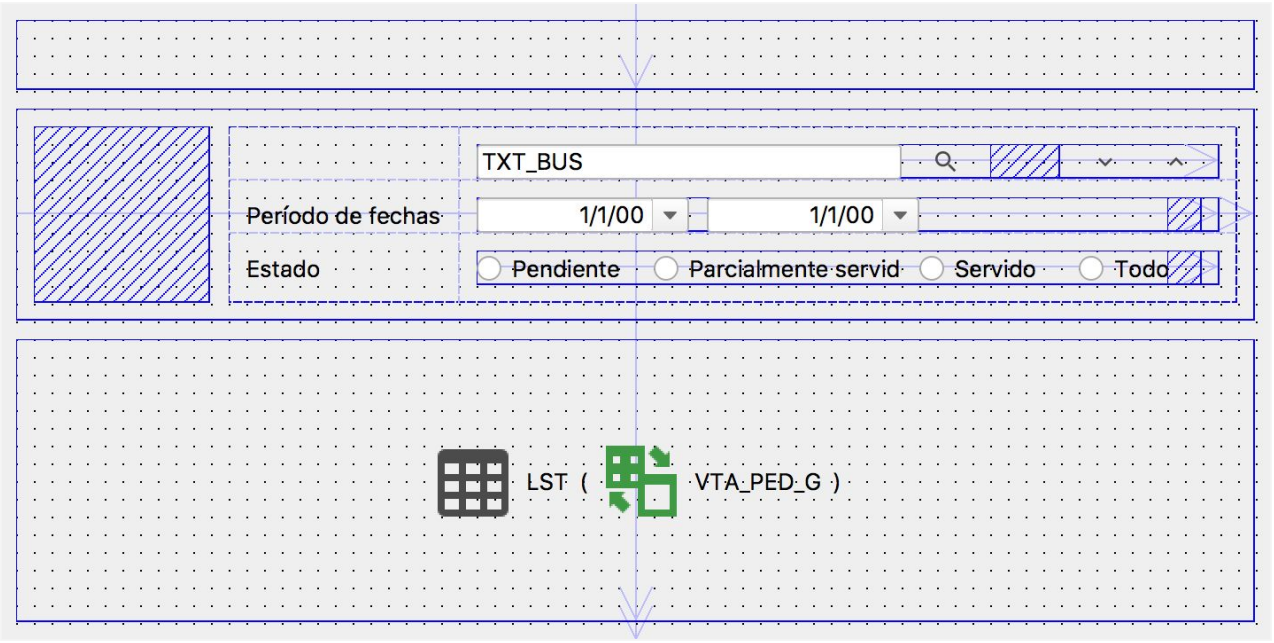
Los menús son formularios especializados en facilitar la búsqueda de información y visualizar en objetos de lista para facilitar la edición o procesado de dicha información.

Identificadores

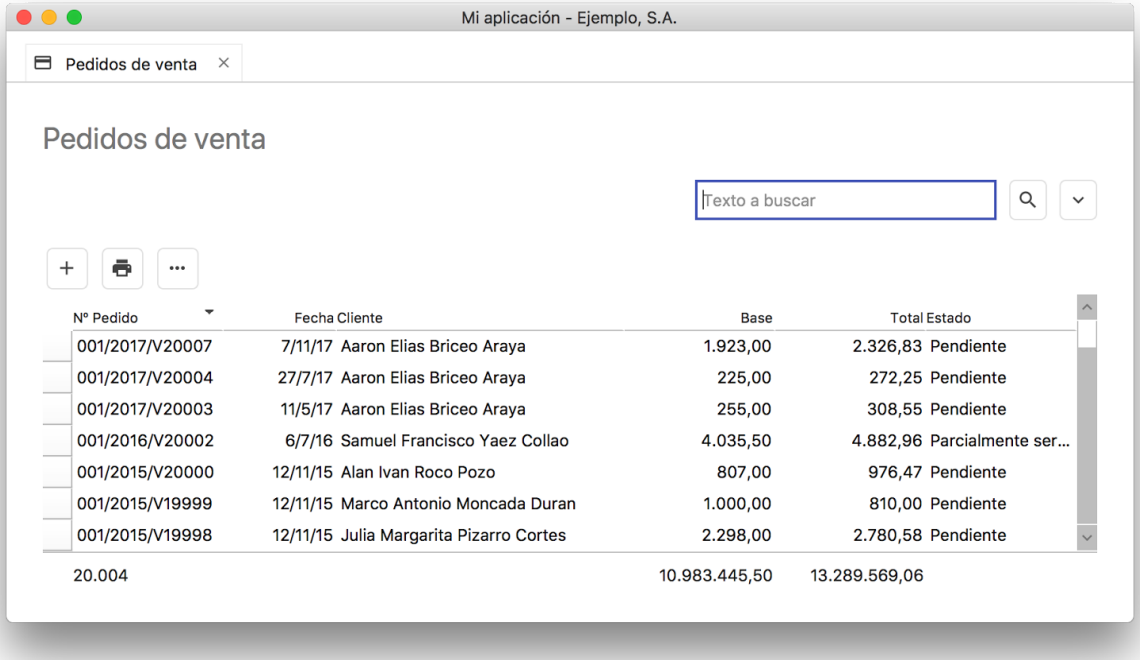
Los controles incluidos en los formularios de menú siguen los siguientes criterios de nomenclatura que podemos ver en la siguiente tabla que contiene los controles más habituales de un formulario.

Identificador	Descripción
BTN_AVA_CTL	Botón oculto que permite al usuario avanzar de control con la tecla *Intro*.
BTN_CNC	Botón oculto que permite cerrar el menú con la tecla *Escape*.
LAY_TIT	Layout del título.
TXT_TIT	Título del formulario.
LAY_CAB	Layout de cabecera.
ESP_CAB	Espaciador cabecera.
LAY_BUS	Layout búsqueda.
LAY_TXT	Layout texto a buscar.
TXT_BUS	Control de edición del texto a buscar (Con retardo señal ValueChanged)
BTN_BUS	Botón buscar.
ESP_TXT	Espaciador texto a buscar.
BTN_AMP	Para mostrar la búsqueda ampliada o avanzada.
BTN_RED	Para ocultar la búsqueda ampliada o avanzada.
TXT_FCH	Texto estático del período de fechas.
LAY_FCH	Layout fechas.
FCH_DES	Fecha desde.
FCH_HAS	Fecha hasta.
ESP_FCH	Espaciador fechas.
LAY_DET	Layout detalle.
LST	Vista de datos de lista.

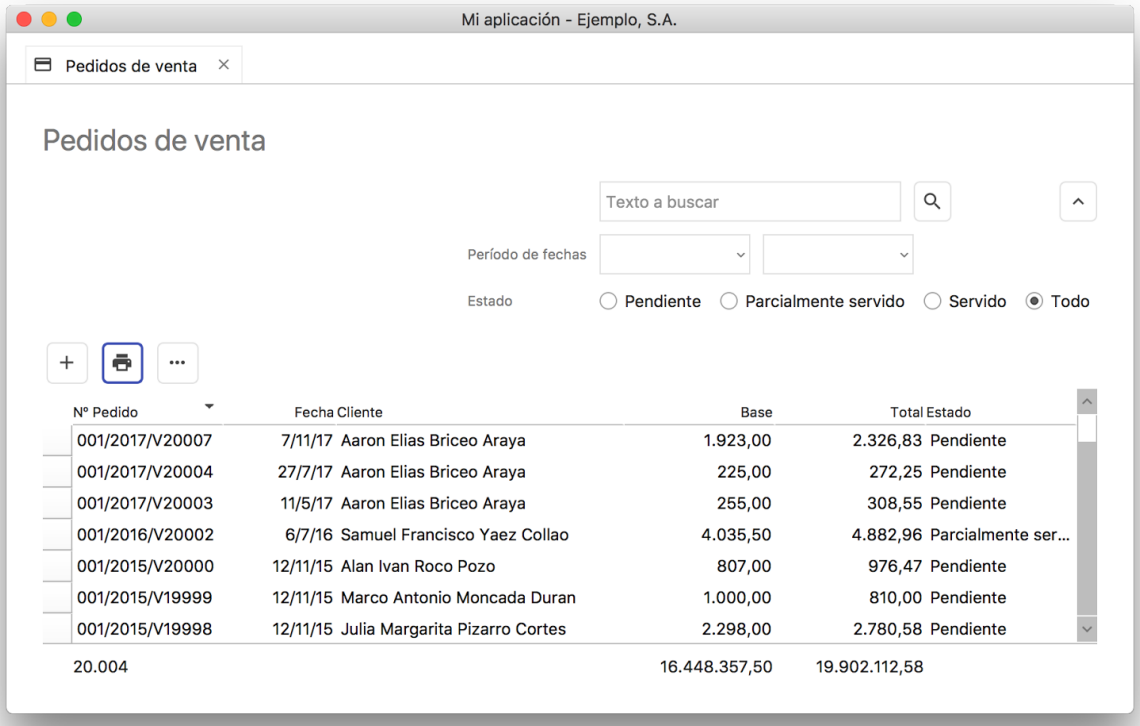
Un formulario de menú prototipo con estos controles sería el siguiente:



En ejecución se muestra por defecto con el siguiente diseño con búsqueda estándar:



y con búsqueda avanzada o ampliada queda así:



Layouts

En un formulario de menú prototipo utilizamos las siguientes configuraciones de layout:

El objeto formulario generalmente tiene definido un layout con la siguiente configuración:

Tipo de layout	**Vertical**
Alineamiento horizontal	**Izquierda**
Alineamiento vertical	**Arriba**
Espaciado	**0**
Margen izquierdo	**20**
Margen derecho	**20**
Margen superior	**20**
Margen inferior	**20**

El layout de título tiene la siguiente configuración:

Tipo de layout	**Vertical**
Espaciado	**10**
Margen izquierdo	**-1**
Margen derecho	**-1**
Margen superior	**-1**
Margen inferior	**10**
Ancho en layout	**Por defecto**
Alto en layout	**Por defecto**

El layout de cabecera tiene la siguiente configuración:

Tipo de layout	**Horizontal**
Espaciado	**10**
Margen izquierdo	**-1**
Margen derecho	**-1**
Margen superior	**-1**
Margen inferior	**10**
Ancho en layout	**Por defecto**
Alto en layout	**Por defecto**

El layout de búsqueda tiene la siguiente configuración:

Tipo de layout	**Grid**
Espaciado	**-1**
Margen izquierdo	**-1**
Margen derecho	**-1**
Margen superior	**-1**
Margen inferior	**-1**
Ancho en layout	**Por defecto**
Alto en layout	**Por defecto**

El layout de detalle tiene la siguiente configuración:

Tipo de layout	**Vertical**
Espaciado	**10**
Margen izquierdo	**-1**
Margen derecho	**-1**
Margen superior	**-1**
Margen inferior	**10**
Ancho en layout	**Por defecto**
Alto en layout	**Por defecto**

El resto de layouts que se usan para agrupar controles como el layout de texto a buscar y layout de fechas, tienen la siguiente configuración para ajustar al máximo los márgenes separando los controles internos:

Tipo de layout	**Horizontal**
Espaciado	**10**
Margen izquierdo	**0**
Margen derecho	**0**
Margen superior	**0**
Margen inferior	**0**
Ancho en layout	**Por defecto**
Alto en layout	**Por defecto**

Título de la pestaña

A diferencia de los formularios de edición el título de la pestaña de un menú se asume de la propiedad nombre del formulario.

Con el fin de evitar programación y el típico efecto de copiar/pegar que se produce cuando creamos un formulario de menú a partir de otro y se nos olvida cambiar la propiedad nombre se utiliza un manejador de evento de tipo JavaScript llamado *CHG_TIT* (cambiar título) que solo tiene esta línea de código.

```
// Cambiar el título del formulario aplicando el nombre del formulario al texto del control
theRoot.dataView().control("TXT_TIT").setText(theRoot.dataView().objectInfo().name());
```

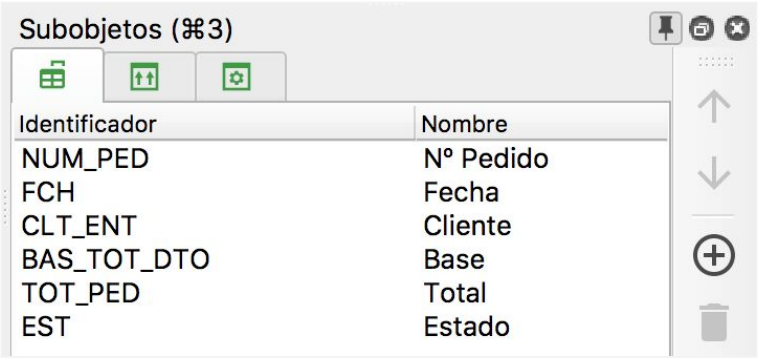
Podemos copiar este manejador en todos los menús y funcionará correctamente ya que se encarga de atrapar el valor de la propiedad nombre del formulario y ponerlo en el control *TXT_TIT* que es de tipo *Nombre de campo* sin ninguna resolución de campo, ya que este tipo de control si permite cambiar dinámicamente su contenido. El manejador *CHG_TIT* es ejecutado por el manejador *POS_INI* al construirse el formulario.

Rejillas

Se trata del objeto más usado en las aplicaciones Velneo ya que sabemos que como usuarios nos gusta recibir la información en modo lista para luego filtrar o entrar en el detalle de determinadas fichas.

Identificadores

Los identificadores de las columnas deben coincidir con su contenido. Si el contenido es un campo el identificador debe ser el mismo que el del campo, si es una fórmula deberíamos poner un identificador que represente el resultado de la fórmula.



Anchos y alineamientos de columnas en función del tipo de dato

En la siguiente tabla se describen los anchos, tipos de ancho y alineación de los tipos de datos más típicos utilizados en rejillas.

Tipo de dato	Ancho	Tipo de ancho	Alineación
Texto de tamaño fijo (3 caracteres)	30	Interactivo o fijo	Izquierda
Texto de tamaño fijo (9 caracteres)	90	Interactivo o fijo	Izquierda
Texto de tamaño fijo (12 caracteres)	120	Interactivo o fijo	Izquierda
Texto de tamaño variable (40 caracteres)	200	Máximo disponible	Izquierda
Fecha	90	Interactivo o fijo	Derecha
Número corto (2 enteros, 2 decimales)	60	Interactivo o fijo	Derecha
Número medio (6 enteros, 2 decimales)	90	Interactivo o fijo	Derecha
Número largo (9 enteros, 2 decimales)	120	Interactivo o fijo	Derecha
Icono pequeño	30	Interactivo o fijo	Centrado
Dibujo grande	120	Interactivo o fijo	Centrado

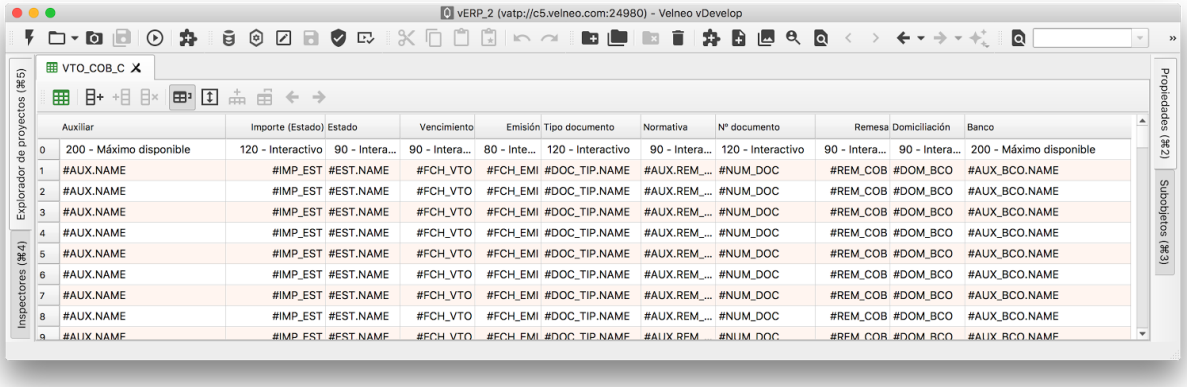
Si en una rejilla no hay ninguna columna de tipo de ancho "*máximo disponible*" pondremos a todas las columnas el tipo de ancho como "*máximo disponible*".

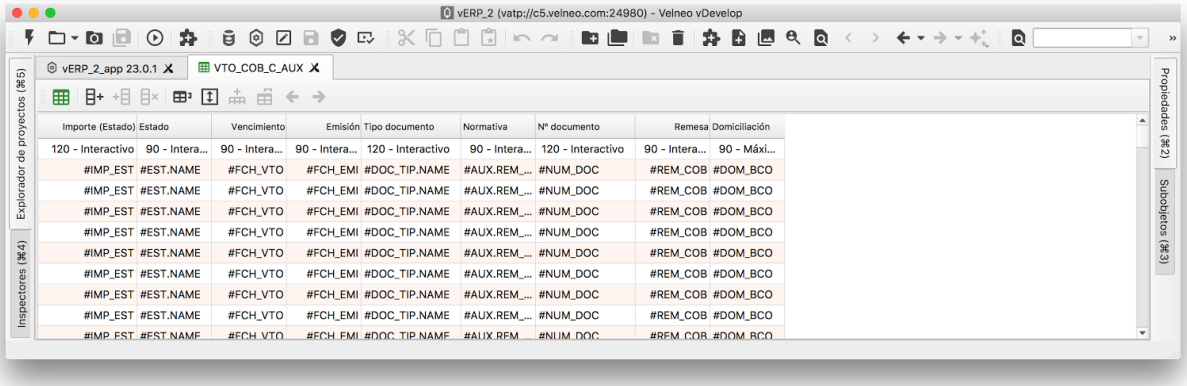
Crea rejillas específicas para uso en formularios de maestros

Dado que los objetos de vista se pueden utilizar tanto dentro de la interfaz de la aplicación por parte del programador como ser usados por los usuarios en caso de no ser privadas, conviene que por cada tabla se cree una rejilla “completa” con todos los campos o al menos los más significativos.

VTO_COB_C	Cobros
VTO_COB_C_APU	Cobros de un apunte
VTO_COB_C_AUX	Cobros de una cuenta auxiliar
VTO_COB_C_BCO	Cobros de un banco
VTO_COB_C_CBA	Cobros de una conciliación bancaria
VTO_COB_C_LOC	Cobros (Localizador)
VTO_COB_C_REM	Cobros de una remesa
VTO_COB_C_VTA_FAC	Cobros de una factura de venta
VTO_COB_C_VTO	Cobros de un vencimiento

Adicionalmente, debemos crear rejillas específicas que se visualizarán en las vistas de plurales de tablas maestras. Estas rejillas tienen la peculiaridad de que no contienen la columnas o columnas con información del maestro que ya es visible en el formulario que contiene la rejilla. A continuación vemos en primer lugar la rejilla de cobros *VTO_COB_C* con todas las columnas y debajo la rejilla de cobros de una cuenta auxiliar *VTO_COB_C_AUX* que podemos observar que no tiene la primera columna auxiliar.



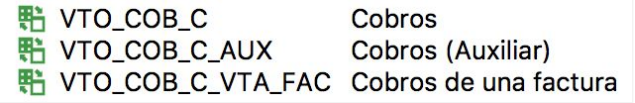


Alternadores de lista

El objeto alternador de lista es muy útil al permitir contener múltiples objetos de vista de datos facilitando al usuario final la posibilidad de ver la misma información con diferentes formatos o vistas (rejillas, informes, casilleros, etc.) además de forma optimizada. Por lo tanto su uso es recomendado.

Usa un alternador en lugar de poner la rejilla directamente

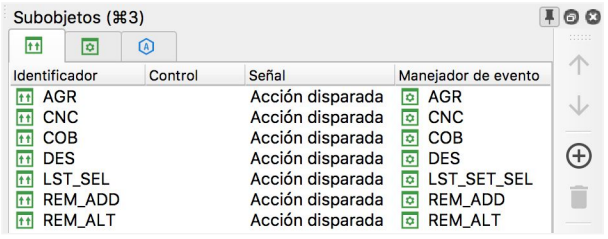
Una buena práctica consiste en no poner directamente rejillas en las vistas de datos y en su lugar poner siempre un alternador de lista. Aunque en muchos casos solo haya una rejilla, esta técnica permite que en el futuro se puedan añadir nuevas vistas de forma sencilla.

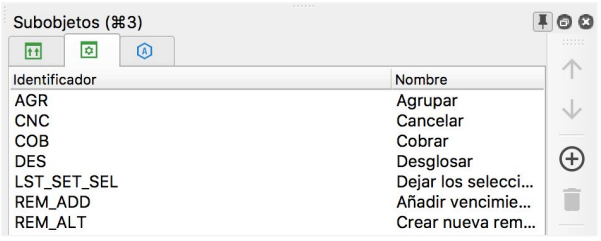


Puede ser normal que tengamos que crear múltiples alternadores de lista para la misma tabla.

Reducimos la cantidad de código

Una de las grandes ventajas de usar alternadores es que las conexiones y manejadores de evento que les declaremos serán funcionales para todas las vistas declaradas en el alternador. Por ejemplo, si tenemos una toolbar que usamos para dar funcionalidad a varias vistas al aplicarla a través del alternador conseguir que en lugar de declarar en todas las rejillas las conexiones y manejadores duplicados los tendremos declarados una única vez en el alternador ya que es capaz de lanzar los manejadores contra el objeto en curso. En las siguientes capturas podemos ver como en un alternador hemos declarado 7 conexiones de evento y sus manejadores que son lanzados por acciones disparadas desde una toolbar, si este alternador contiene 3 rejillas hemos conseguido que todas tengan la misma funcionalidad ahorrando repetir el código 3 veces.





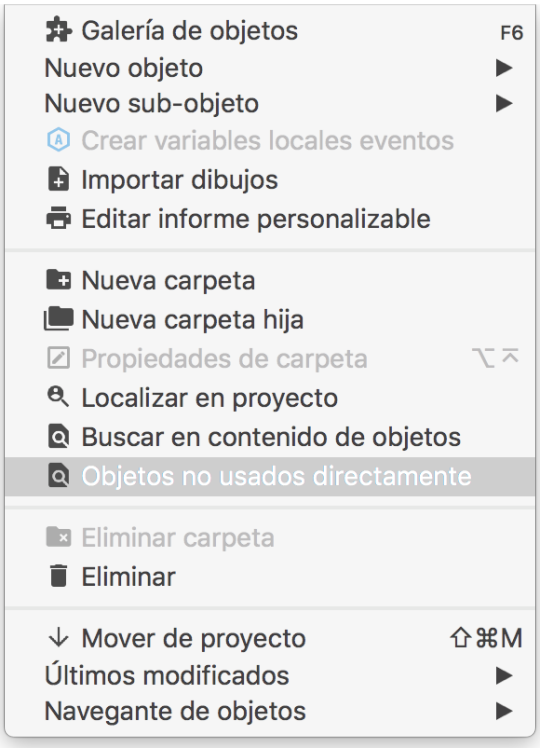
Calidad

La calidad de una aplicación es resultado de todo el trabajo realizado durante el ciclo completo de desarrollo. Al finalizar un sprint, revisión o versión y antes de su puesta en producción, es muy recomendable realizar siempre las siguientes opciones para garantizar la máxima calidad.

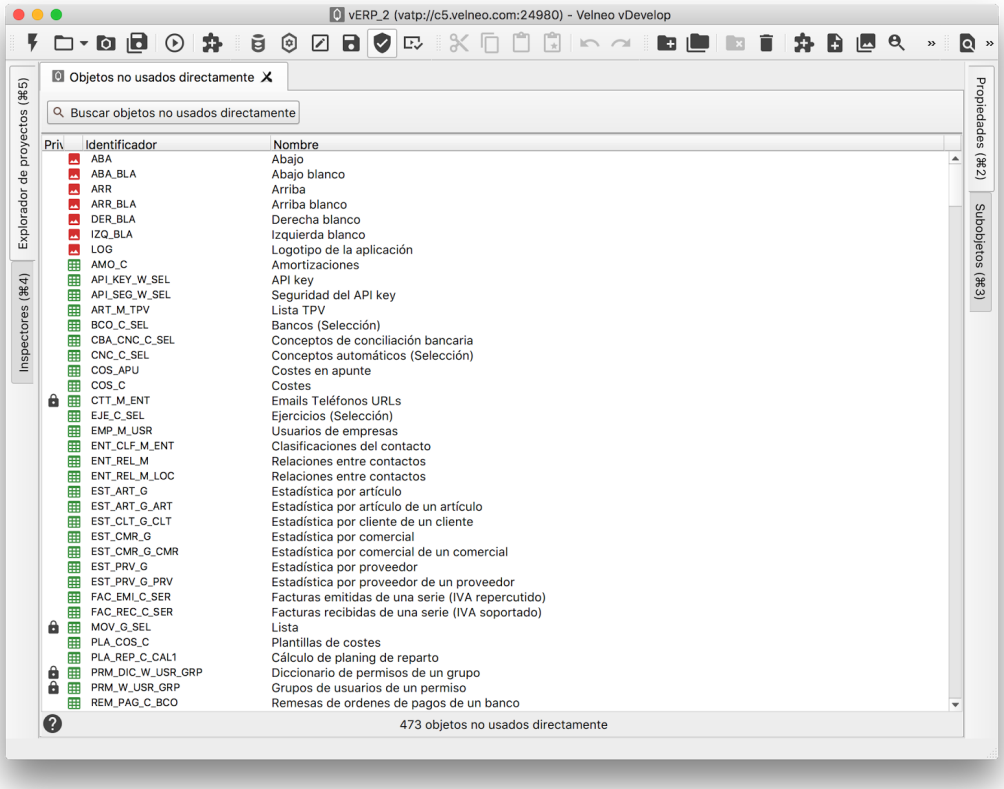
Revisa los objetos no usados directamente con la extensión

Otro aspecto crítico en la calidad del software es la limpieza del código, tener el proyecto objetos que no se utilizan es una mala praxis, y aún es más grave cuando disponemos de otra extensión que se encarga de realizar esa labor por nosotros mostrándonos en segundos la lista de objetos no usados en el proyecto.

En el menú de Objetos encontraremos la opción "*Objetos no usados directamente*". El nombre lo dice todo, son objetos que no son usados directamente dentro de los proyectos a través de código nativo Velneo.



Al lanzar esta opción nos aparecerá la extensión que podremos ejecutar simplemente pulsando el botón situado en la parte superior.



Hay que tener en cuenta que hay objetos que se usan directamente en ejecución como por ejemplo los objeto dibujo que aparecen al principio de la lista de la captura anterior se usan en un JavaScript que los exporta para usarlos en las CSS, por lo tanto antes de eliminar un objeto conviene buscarlo en los scripts (usando el check de herencia) para intentar asegurar que no es usado. Aún usando la búsqueda en scripts no podemos garantizar que el objeto no sea usado ya que podemos tener en nuestra aplicación scripts dinámicos almacenados en tablas de la base de datos, por lo que sería conveniente revisar que no usen dicho objeto dentro de scripts externos al proyecto.

También puede ocurrir que nos encontremos con objetos como acciones o procesos que aparentemente no se usan pero que en realidad son ejecutados de forma dinámica en tiempo de ejecución, como puede ocurrir con las opciones de menú cuya configuración puede estar almacenada en una tabla en disco.

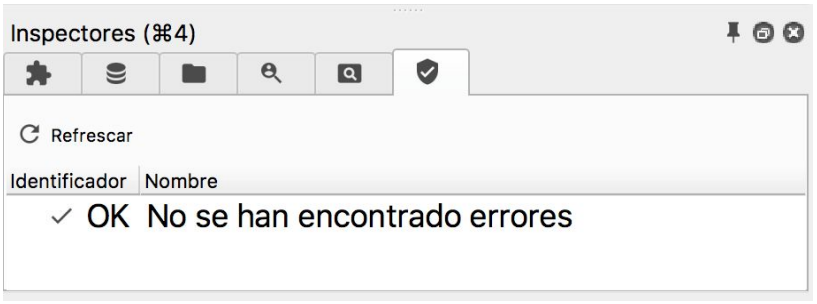
En definitiva, este inspector es de gran ayuda y nos simplifica los objetos que tenemos que revisar porque no tengan un uso directo, aunque tenemos que realizar una revisión posterior para antes de eliminarlos asegurarnos de que no son usados.

Es importante realizar siempre esta limpieza en nuestros proyectos periódicamente. Un buen momento puede ser al principio de una nueva versión o revisión, ya que en caso de eliminar un objeto que sí es usado será más fácil de detectar en las pruebas del desarrollador o de los testers.

Revisa los errores con el inspector en todos los proyectos

Realmente deberíamos usar el inspector de errores para revisarlos antes de cualquier ejecución. Si no lo hacemos siempre, sí que es conveniente revisarlo de vez en cuando durante una sesión de desarrollo y desde luego antes de cerrar el editor tras finalizar una sesión de trabajo. No deberíamos tener ningún error

detectado por el inspector de errores, salvo excepciones como algunos avisos por el uso de comandos de instrucción obsoletos, en cuyo caso lo recomendable es sustituirlo cuanto antes para evitar que aparezcan los errores en el inspector.

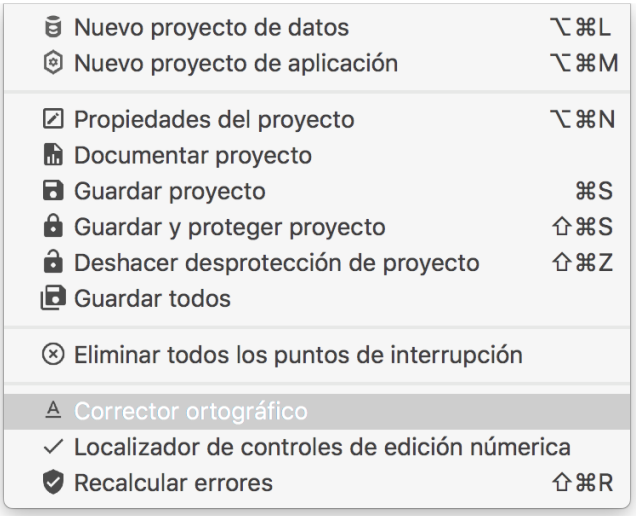


Siempre que vayamos a publicar una versión o a ponerla en un servidor de producción es “obligatorio” pasar el inspector de errores para obtener un resultado como el mostrado en la captura anterior.

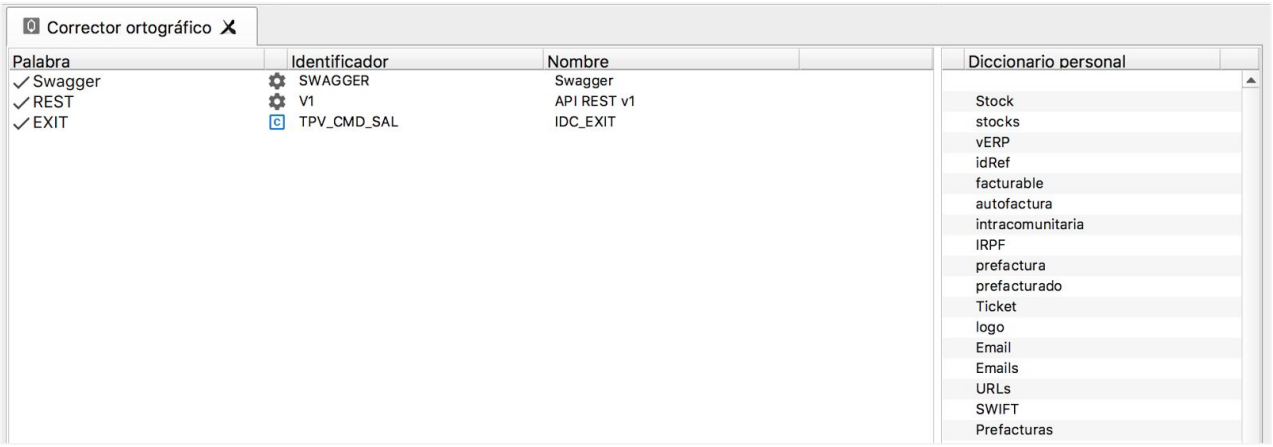
Revisa la ortografía con la extensión

Aunque los errores de programación nos parece importantes a los programadores, a los usuarios finales los errores ortográficos les resultan igual de molestos que cualquier error funcional. Los programadores tenemos una disposición a escribir igual que programamos, evitando el uso de acentos, escribir todas las palabras con la primera en mayúscula, etc. Esto que a priori nos puede parecer normal, es una falta de calidad en nuestro software que los usuarios finales detectan rápidamente.

Para evitar estos problemas tenemos que tomar 2 medidas. La primera es escribir siempre bien, acentuar correctamente las palabras, hacer buen uso de las mayúsculas y minúsculas, poner puntuación en las frases y no utilizar abreviaturas desconocidas para el usuario, en definitiva, escribir bien de la misma forma que cuidamos el texto en los correos electrónicos o si estuviésemos escribiendo un libro. La segunda medida es utilizar la extensión de vDevelop “Corrector ortográfico” disponible en el menú de Proyectos.



Al ejecutarla se nos mostrará en la parte izquierda la lista de palabras que no son válidas o si lo son no están identificadas como válida en nuestro diccionario personal. A la derecha se muestran todas las palabras que hemos añadido a nuestro diccionario personal.



Si hacemos doble clic sobre la palabra se nos abre el lugar donde se usa para que podamos corregirla, si consideramos que la palabra es correcta hacemos clic en el check situado a la izquierda de la palabra y se añadirá al diccionario personal dejando de aparecer como erróneamente, si esa palabra está repetida en más lugares al entrar en el diccionario personal desaparecen de la lista todas sus instancias.

Así pues, no hay excusa para entregar una aplicación con errores ortográficos. La calidad empieza aquí.