Desarrollo basado en modelos de aplicaciones de gestión de datos para dispositivos móviles

Sistemas Informáticos

Universidad Complutense de Madrid

Facultad de Informática

Junio 2012
Carlos Moya Ruiz
David José Constanzo Oliver
Julián Martín Diez-Madroñero

DESARROLLO BASADO EN MODELOS DE APLICACIONES DE GESTIÓN DE DATOS PARA DISPOSITIVOS MÓVILES

- Autores:

- Carlos Moya Ruiz
- David José Constanzo Oliver
- Julián Martín Diez-Madroñero

- Profesor director:

- Manuel García Clavel

Memoria de Sistemas Informáticos

Facultad de Informática
Universidad Complutense de Madrid

Junio 2012

AUTORIZACIÓN DE DIFUSIÓN

Los abajo firmantes autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académico no comerciales, y mencionando expresamente a sus autores, tanto la propia memoria como el código, la documentación y/o el prototipo desarrollado.

Carlos Moya Ruiz **David José Constanzo Oliver** **Julián Martín Diez-Madroñero**

A nuestras familias y amigos, por aguantar nuestros momentos de mal humor, que no han sido pocos. También a Manuel y Gonzalo, por la ayuda y consejos que nos han proporcionado.

Los autores

RESUMEN

DESARROLLO BASADO EN MODELOS DE APLICACIONES DE GESTIÓN DE DATOS PARA DISPOSITIVOS MÓVILES

Las interfaces gráficas en la web cambian a un ritmo vertiginoso de forma constante. Si bien en sus inicios los diseños eran muy básicos, hoy en día una página web puede parecer perfectamente una aplicación de escritorio. Este hecho provoca que el diseño de la apariencia y la usabilidad tengan muchísima importancia. Al fin y al cabo, en muchas ocasiones, el factor diferencial por el cual un cliente escoge entre dos productos que intentan dar respuesta al mismo problema, es la apariencia. El cliente no sólo espera que el producto tenga una determinada funcionalidad, sino que además sea bonito y usable de manera intuitiva. Por todo ello es muy importante contar con buenas herramientas para diseñar interfaces gráficas.

En esta memoria comenzaremos describiendo ActionGUI, una tecnología para realizar aplicaciones web, y la necesidad de tener un editor de interfaces gráficas para ella. A continuación mostraremos los aspectos del editor desarrollado en este proyecto ofreciendo una visión tanto a nivel funcional como a nivel técnico.

DEVELOPMENT OF DATA MANAGEMENT APPLICATIONS FOR MOBILE DEVICES BASED ON MODELS.

The design of graphical user interfaces (GUIs) in the web is changing on a daily basis. In the beginning they were very simple but nowadays they look like a desktop application. As a result, the design of GUIs and usability has turned out very important. In the end, the user does not use to choose an application by its functionality but for its design. Because of that, having tools capable of designing appealing graphical user interfaces can be the key of success.

We will start this dissertation explaining the technology ActionGUI and the necessity of having an application on top of it to improve its visual interface. Then we will explain in depth how to interact with the application itself as well as the most interesting technical aspect.

PALABRAS CLAVE

- ✓ Vaadin
- ✓ Modelo
- ✓ CSS
- ✓ Widget
- ✓ Jasfag
- ✓ Maven
- ✓ GWT
- ✓ ActionGUI
- ✓ App-Móvil

ÍNDICE

1. ActionGUI

ActionGUI [10] es una tecnología para el desarrollo de aplicaciones web seguras (típicamente orientadas a la gestión de datos) a partir de modelos, basada en las investigaciones llevadas a cabo en el Modeling Lab del IMDEA Software Institute (http://software.imdea.org).

ActionGUI proporciona tres lenguajes de modelado para especificar, respectivamente, la estructura de los datos, la política de seguridad y el interfaz web de la aplicación que se desea desarrollar. A partir de estos tres modelos, y mediante transformaciones bien definidas, ActionGUI genera un único modelo para la aplicación, en el que la interfaz gráfica es ya inteligente con respecto a la política de seguridad; en adelante, a este modelo generado por ActionGUI le denominaremos "modelo jasfag" o simplemente “modelo”. Finalmente, a partir del modelo jasfag, ActionGUI genera el código de la aplicación, de forma automática y completa, listo para ser desplegado en cualquier servidor de aplicaciones web.

En su desarrollo actual, el lenguaje que ofrece ActionGUI para modelar las interfaces web es muy limitado. Simplemente, ofrece la posibilidad de posicionar los widgets (también denominados en adelante “componentes”) dentro de las ventanas y definir su tamaño. No es posible, en particular, definir los colores ni las fuentes que utilizan estos widgets, ni mucho menos situarlos dentro de "layouts" (o de combinaciones de "layouts"). Además, el editor de ActionGUI no ofrece actualmente ninguna ayuda para reutilizar modelos, con el fin, por ejemplo, de generar distintos interfaces para distintas pantallas de forma práctica y eficaz.

El objetivo del presente proyecto es precisamente solucionar las limitaciones actuales de la tecnología ActionGUI en lo que se refiere al modelado de interfaces gráficas y a la reutilización de estos modelos. En concreto, hemos implementado una aplicación web que, dado un modelo jasfag generado por ActionGUI, permite redefinir la posición, formato y tamaño de sus widgets, para distintas pantallas de dispositivos móviles, manteniendo, sin embargo, el comportamiento definido para estos widgets en el modelo original. La idea es que, a partir de modelo jasfag así "enriquecido" (o, más técnicamente, a partir de las hojas de estilo CSS que recogen los cambios de posición, formato y tamaño introducidos en el modelo jasfag original), el generador de código de ActionGUI producirá las nuevas interfaces web con las características de presentación deseadas.

2. La aplicación. Guía de uso.

2.1. La aplicación

2.1.1. Acceso a la aplicación

Para abrir la aplicación, al ser una aplicación web, necesitaremos un navegador. Aunque por las características de la aplicación se puede acceder a ella mediante cualquier navegador, recomendamos, por este orden:

- Google Chrome 19.0.1
- Mozilla Firefox 12.0
- Microsoft Internet Explorer 9
- Apple Safari 5.1.7

Una vez hayamos arrancado nuestro navegador web, podemos acceder a la aplicación escribiendo la url de la misma. Como ejemplo:

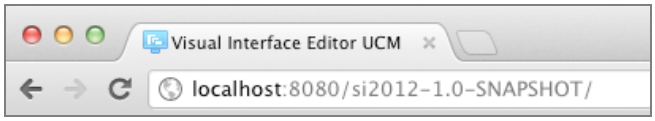


Imagen 1: URL local host

2.1.2. Explicación de la estructura de la aplicación

Una vez que la aplicación se ha cargado, podremos verla en pantalla:

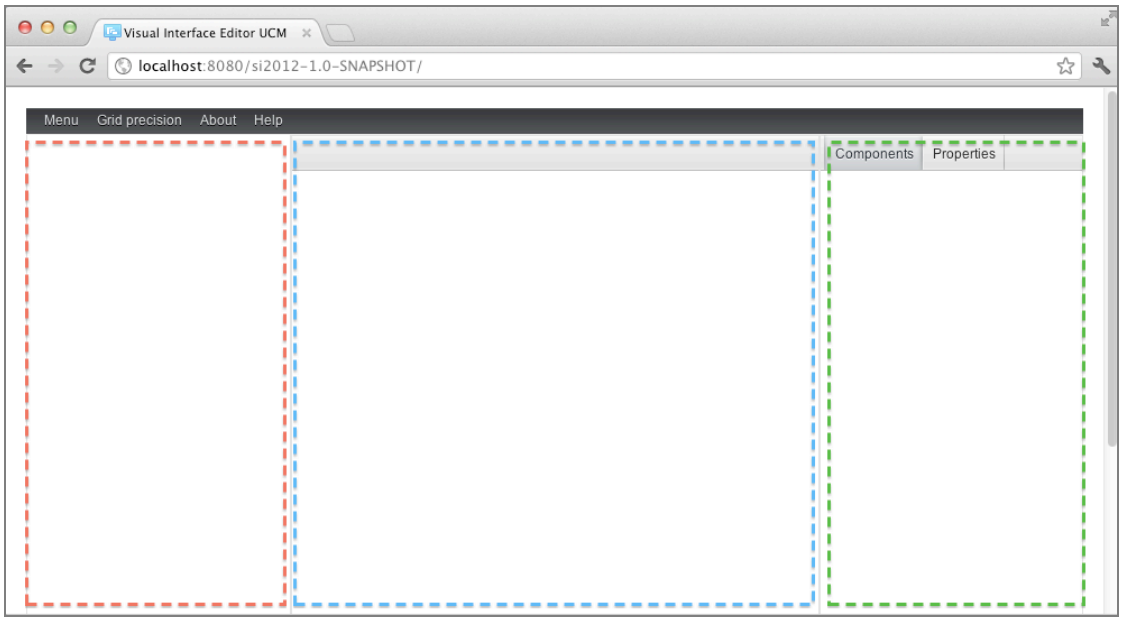


Imagen 2: Aplicación en el estado inicial

Como se puede apreciar, la aplicación está vacía, preparada para cargar un modelo. Consta de tres partes bien diferenciadas, aunque sincronizadas entre si:

- En la parte izquierda, bordeada en color rojo, tenemos el panel del *árbol* que indica los componentes que tenemos cargados en la vista. Además, desde el *árbol*, podemos realizar muchas otras acciones que se explicarán en los siguientes puntos.

- En la parte central, bordeada en color azul, tenemos la vista del *dispositivo* para el cual estamos modificando y enriqueciendo el modelo. Cuando se carga un modelo, en la parte superior de esta vista aparecerán tantas pestañas como ventanas tenga el modelo, pudiendo cambiar fácilmente entre una y otra. Cuando se realiza un cambio de pestaña (ventana en el modelo), se actualizará de manera automática tanto el *árbol* de la parte izquierda como el panel de *componentes y propiedades* de la derecha ya que cada pestaña tiene los suyos propios. En cada pestaña aparecerá un dispositivo genérico o uno específico, pudiendo elegir entre varios de los más populares actualmente. Para más información sobre los tipos de *dispositivo* y cómo cambiar entre ellos, ver el apartado 2.4.1.

- En la parte derecha, bordeada en color verde, tenemos el panel de *componentes y propiedades* en el que podemos ver dos pestañas. En la primera, *Components,* veremos los componentes que no están añadidos a la vista. En la segunda, *Properties*, veremos para cada componente añadido en la vista, que haya sido seleccionado mediante el *árbol* o el *dispositivo,* sus propiedades de estilo.

Para más información a cerca de cómo cargar un modelo, ver el apartado 2.2 de la guía de uso.

2.2. Cargar un modelo

Una vez que hemos accedido a la aplicación a través del navegador web, podemos cargar en ella un modelo. Antes de hacerlo, debemos tener el fichero del modelo en nuestro ordenador.

Para abrir el modelo, tenemos que seleccionar la opción *Menu* en la barra de herramientas:

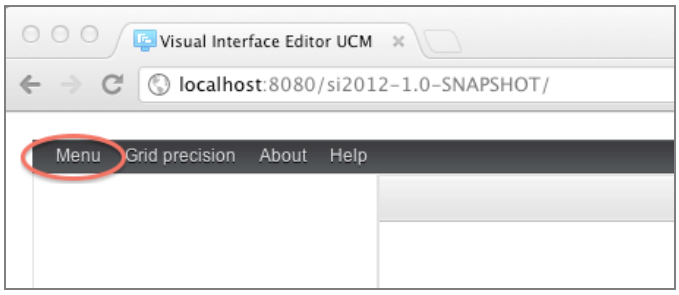


Imagen 3: Barra de opciones

Una vez seleccionada, se abrirá un menú desplegable con varias opciones, de la que elegiremos *Open:*

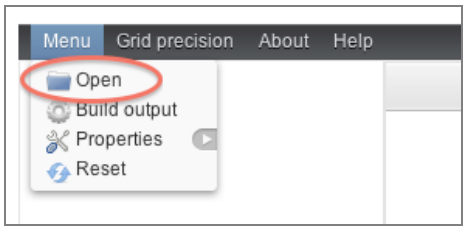


Imagen 4: Opción de menú desplegada

Al seleccionarla, se abrirá una ventana emergente desde donde podremos escoger el fichero que contiene el modelo mediante un selector de ficheros pulsando sobre el botón *Select file*:

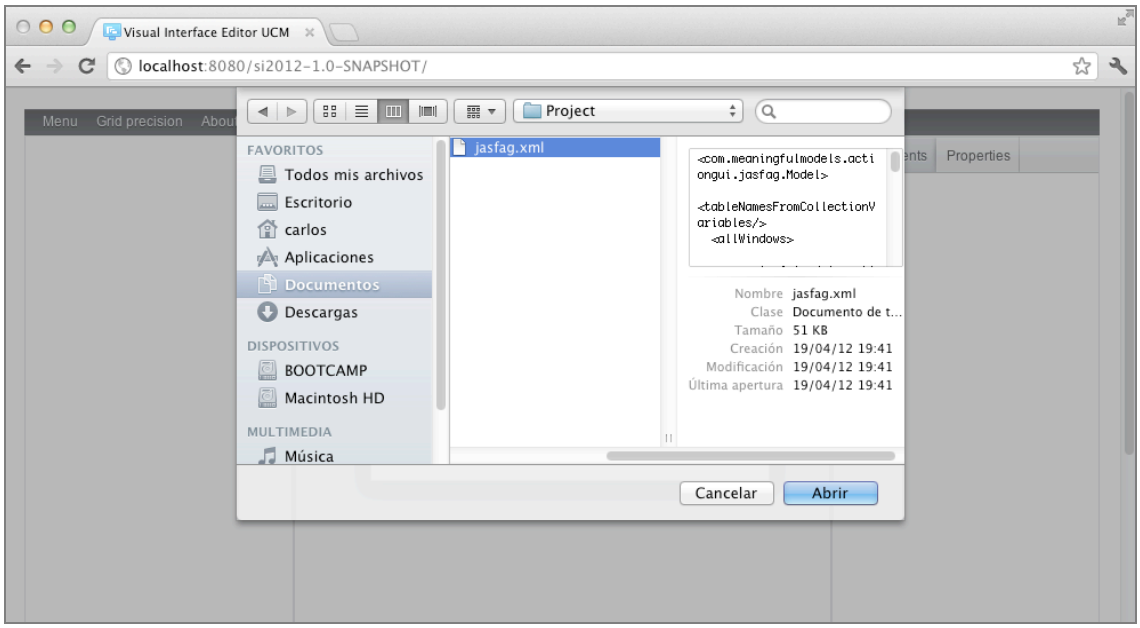


Imagen 5: Selección de fichero

Al pulsar *Abrir,* enviaremos a la aplicación una copia del modelo para que pueda abrirlo. Si durante el proceso de envío del modelo hubiese habido algún error, aparecería una ventana describiendo el mismo y no se produciría carga alguna del modelo en la aplicación. Si todo ha ido correctamente, tendremos nuestro modelo preparado para ser modificado.

Para más información sobre como modificar el modelo, ver el apartado 2.4 de la guía de uso.

Nótese que si ya hubiese un modelo cargado y quisiésemos cargar otro siguiendo los pasos explicados en este apartado, aparecería una venta para poder descargar el trabajo realizado hasta el momento antes de cargar el nuevo modelo:

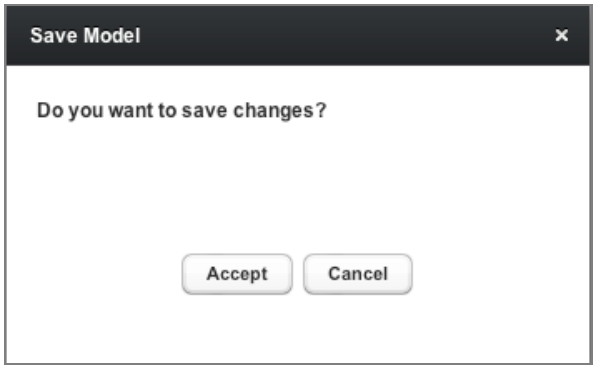


Imagen 6: Aviso para salvar el modelo

2.3. Entender el modelo una vez cargado en la aplicación

Una vez cargado un modelo en la aplicación, veremos lo siguiente:

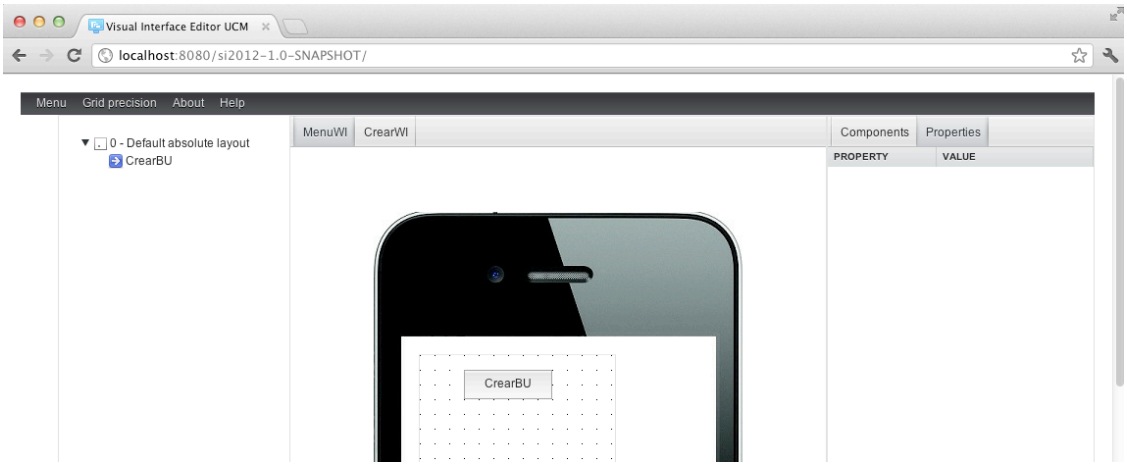


Imagen 7: Estado tras cargar un modelo

2.3.1. Ventanas

Como se puede apreciar, el modelo cargado en la aplicación tiene dos ventanas, representadas por dos pestañas en el panel de *dispositivo.* Para hacer más fácil el uso de la aplicación, el nombre de dichas pestañas es el mismo que el de las ventanas del modelo. Cada una de esas ventanas tiene asociado un *árbol* y un panel de *componentes y propiedades.*

Por defecto, todos los componentes de cada ventana se cargan en la pantalla del dispositivo (también nos referiremos a ella como *vista*) en las coordenadas leídas en el modelo.

Para cambiar de una ventana a otra, basta con seleccionar entre las distintas pestañas del panel central:

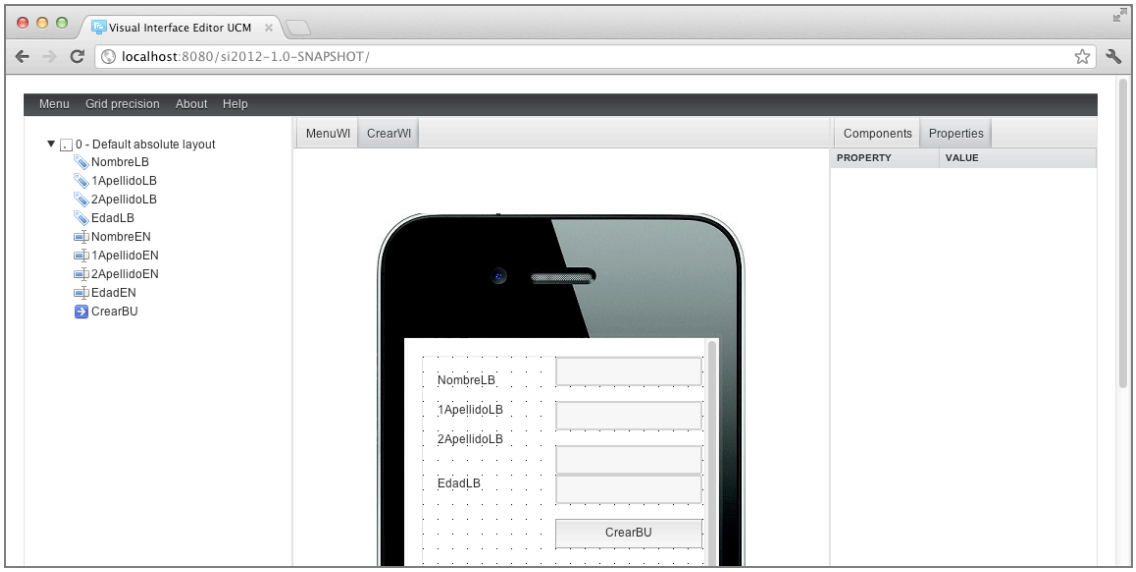


Imagen 8: Cambio de pestaña

2.3.2. Componentes

Una vez que hemos visto como cargar un modelo y movernos entre las diferentes ventanas, vamos a comentar cómo se representan en la aplicación los componentes leídos en el modelo.

2.3.2.1. Botón

El botón o *button* leído del modelo se representa en la aplicación mediante el icono  y su apariencia por defecto es:

2.3.2.2. Etiqueta

La etiqueta o *label* leído del modelo se representa en la aplicación mediante el icono  y su apariencia por defecto es:

2.3.2.3. Campo de texto

El campo de texto o *textfield* leído del modelo se representa en la aplicación mediante el icono  y su apariencia por defecto es:

2.3.2.4. Menú desplegable

El menú desplegable o *combobox* leído del modelo se representa en la aplicación mediante el icono  y su apariencia por defecto es:

2.3.2.5. Enlace

El enlace o *link* leído del modelo se representa en la aplicación mediante el icono  y su apariencia por defecto es:

2.3.2.6. Selección

La selección o *checkbox* leído del modelo se representa en la aplicación mediante el icono  y su apariencia por defecto es:

2.3.2.7. Tabla

La tabla o *table* leído del modelo se representa en la aplicación mediante el icono  y su apariencia por defecto es:

2.3.3. Árbol

En el árbol asociado a cada ventana o pestaña , se ve un resumen de los componentes que forman parte de la vista y el layout que los contiene.

Se verán identificados por el icono correspondiente a su tipo y por el nombre leído desde el modelo para, de ese modo, facilitar el uso de la aplicación.

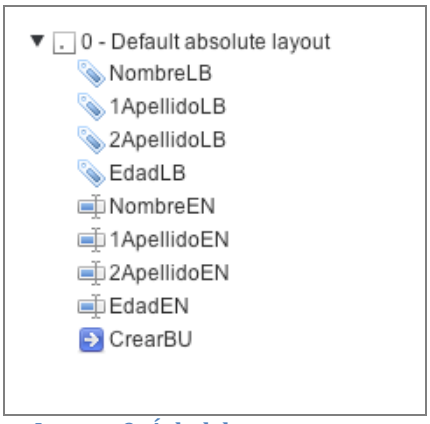


Imagen 9: Árbol de componentes

2.3.4. Componentes y Propiedades

Al seleccionar un componente en el *árbol* o en la vista del *dispositivo* se verán automáticamente sus propiedades en la parte derecha de la aplicación bajo la pestaña *Properties.*

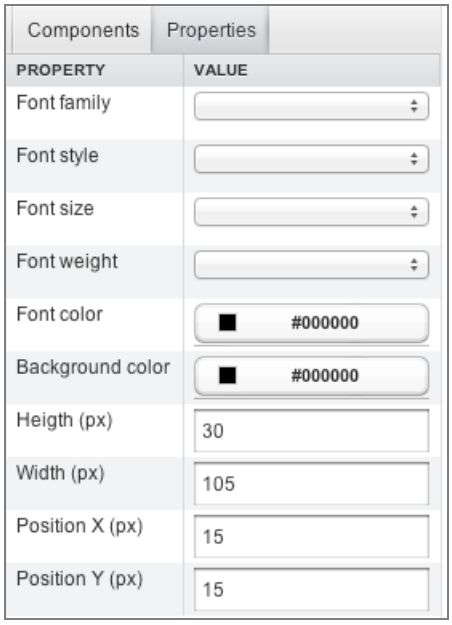


Imagen 10: Tabla de propiedades

En el capítulo 2.4 explicaremos el significado de las diferentes propiedades.

Bajo la pestaña *Components* encontraremos los componentes que pertenecen a la ventana pero que no están en la vista. En el capítulo 2.4.2 explicaremos cómo añadir y eliminar componentes de la vista.

2.4. Modificar el modelo

En los lo siguientes apartados veremos qué se puede hacer y cómo podemos hacerlo a la hora de modificar el modelo.

2.4.1. Modificar el dispositivo

Uno de los elementos fundamentales de la aplicación es el dispositivo móvil para el cual se está diseñando la interfaz. El dispositivo ocupa la parte central de la aplicación, siendo el elemento más visible. Soportamos cuatro tipos diferentes de dispositivos, uno genérico y otros tres específicos. Para seleccionar un tipo de dispositivo, accedemos a *Menu -> Properties -> Device* :

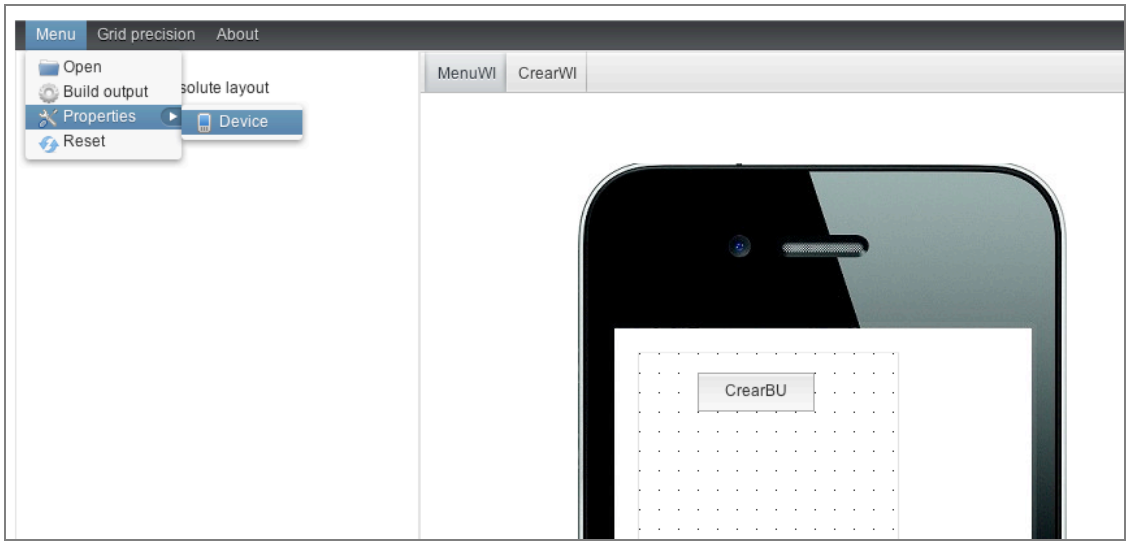


Imagen 11: Modificación del tipo de dispositivo

2.4.1.1. Dispositivo genérico

Es el nombre que recibe en nuestra aplicación la pantalla genérica. Podemos ajustar su altura y su anchura tanto en pixeles como en centímetros para adecuarse al tamaño de cualquier tipo de dispositivo:

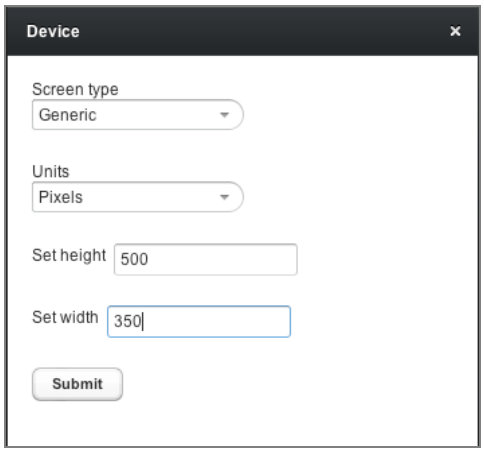


Imagen 12: Configuración del dispositivo

El marco del dispositivo que acabamos de crear aparecerá delimitado en la aplicación por una línea gris:

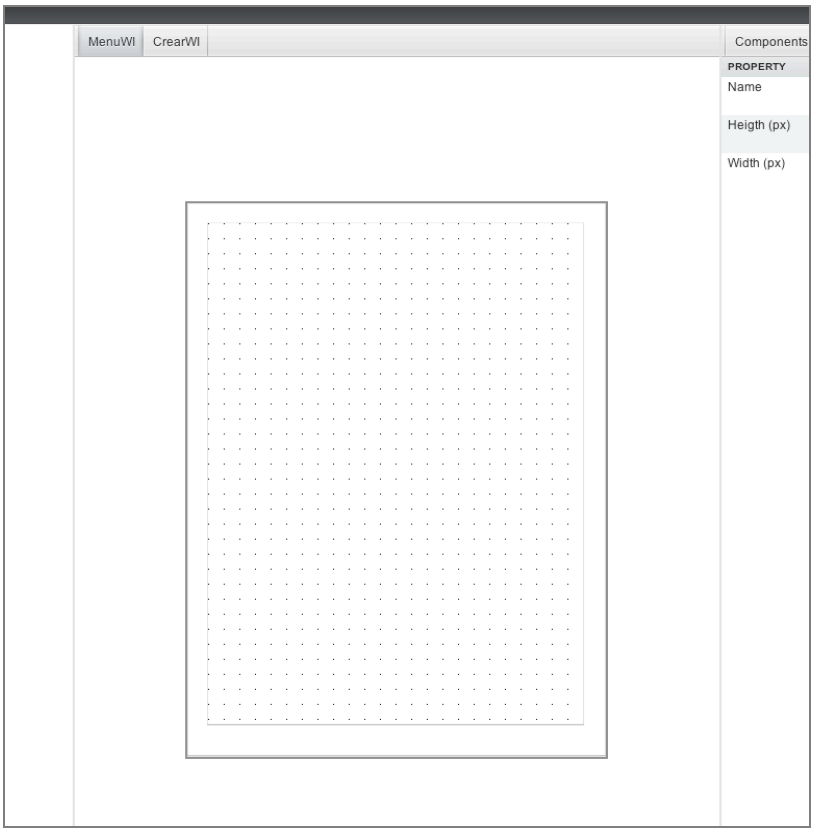


Imagen 13: Marco del nuevo dispositivo

2.4.1.2. Apple IPhone 4

Uno de los tres dispositivos propietarios que soporta la aplicación es el iPhone 4 de Apple Inc. Cuando seleccionamos el iPhone Screen no podemos elegir el tamaño pues se le asigna por defecto el tamaño que tiene en la realidad:

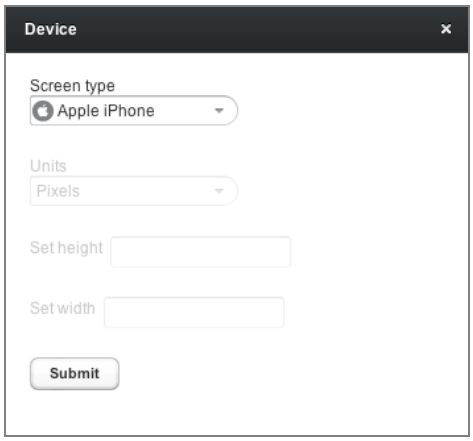


Imagen 14: Selección de iPhone

Una vez seleccionado pulsamos sobre el botón *Submit* para realizar el cambio. El iPhone aparecerá en la parte de edición como se muestra a continuación:

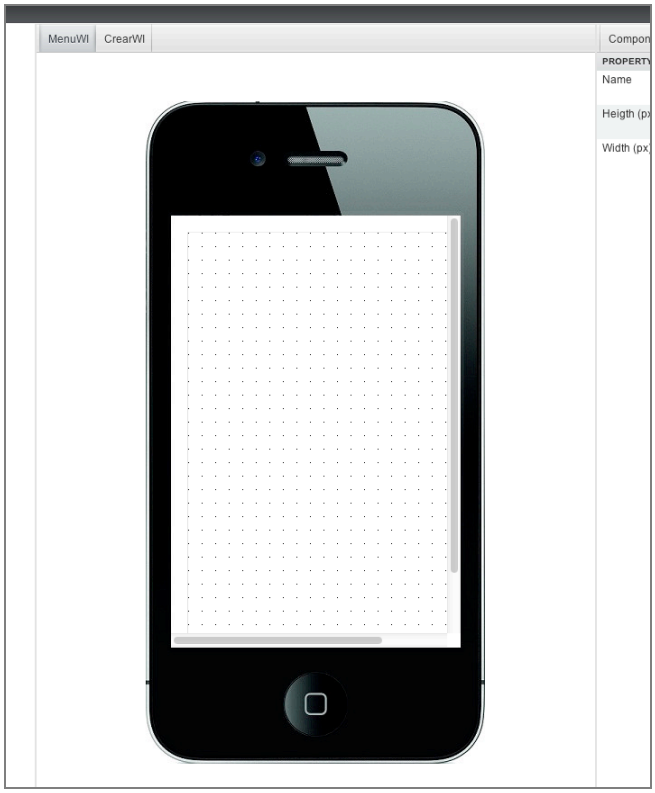


Imagen 15: Terminal Apple iPhone

2.4.1.3. BlackBerry Curve 8520

La BlackBerry de RIM también está presente en la aplicación en su versión Curve 8520. Al seleccionarla tampoco podremos modificar su tamaño pues este viene marcado por el modelo elegido:

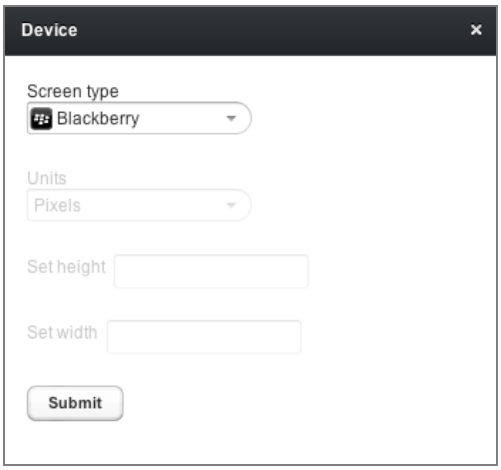


Imagen 16: Selección de BlackBerry

Una vez seleccionado pulsamos sobre el botón *Submit* para realizar el cambio. La BlackBerry aparecerá en la parte de edición como se muestra a continuación:

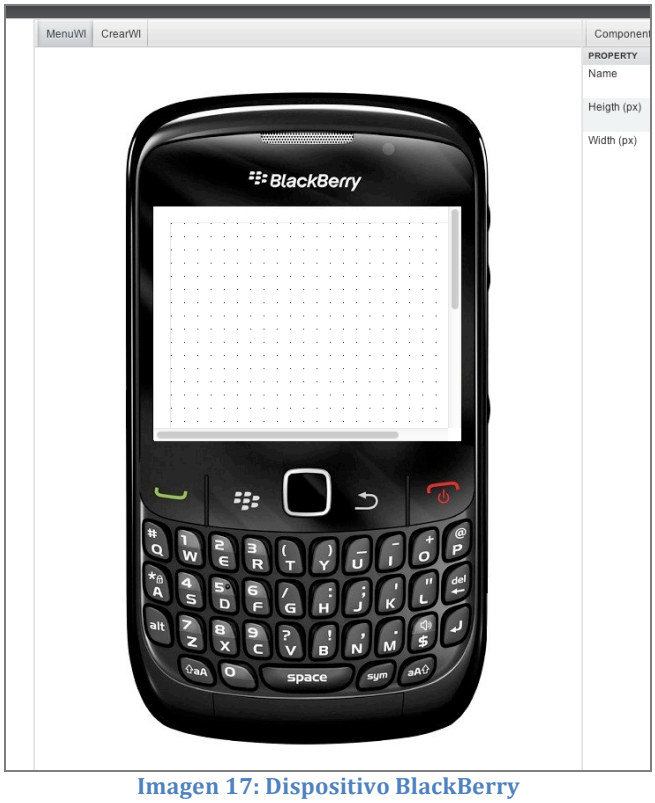


Imagen 17: Dispositivo BlackBerry

2.4.1.4. Samsung Galaxy SII

Por último hemos introducido en la aplicación uno de los dispositivos Android más populares, el Galaxy SII de Samsung. Al igual que con el resto de modelos específicos, los campos para elegir el tamaño del dispositivo permanecen desactivados:

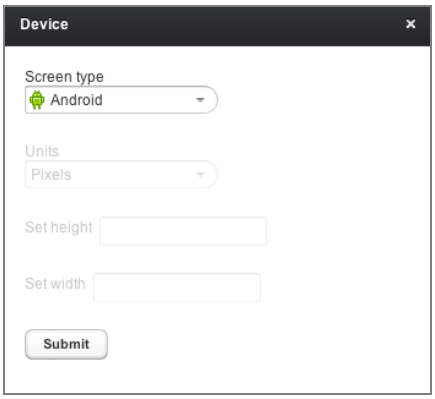


Imagen 18: Selección de dispositivo Android

Una vez seleccionado pulsamos sobre el botón *Submit* para realizar el cambio. El dispositivo aparecerá en la parte de edición de la siguiente manera:

Imagen 19: Dispositivo Android

2.4.2. Cambiar la rejilla del dispositivo

Una de las características más útiles del dispositivo es la rejilla, que tiene un doble uso. Por una parte es capaz de ayudar al usuario a mover mejor los componentes dentro del Layout y por otra, autoajusta la posición de los componentes a las líneas punteadas más cercanas, permitiendo así una precisa alineación de los mismos. El tamaño de la rejilla puede modificarse dependiendo de la precisión que desee el usuario a la hora de ajustar los componentes: el rango de posibilidades abarca desde no tener rejilla en absoluto pudiendo entonces mover libremente los componentes por toda la vista, hasta tener rejillas de 10, 15 y 20 píxeles. Para cambiar la precisión de la rejilla, seleccionamos en la barra de herramientas la entrada *Grid precision* y en el menú que aparecerá, la opción deseada:

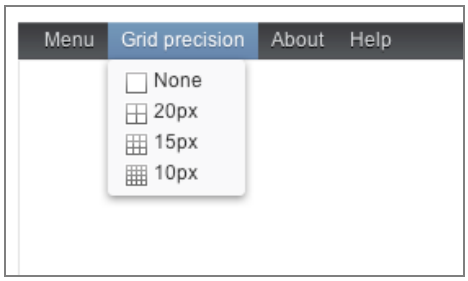


Imagen 20: Precisión de la rejilla

La vista quedará como sigue:

Imagen 21: Grid precision none

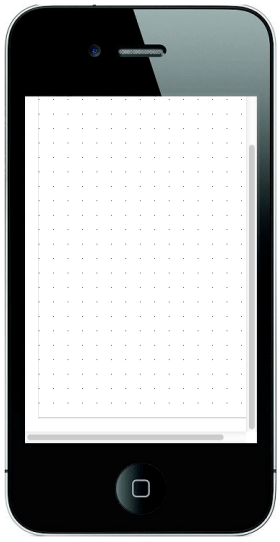


Imagen 22: Grid precision 10px

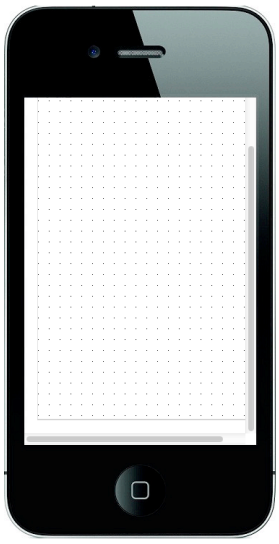


Imagen 23: Grid precision 15px

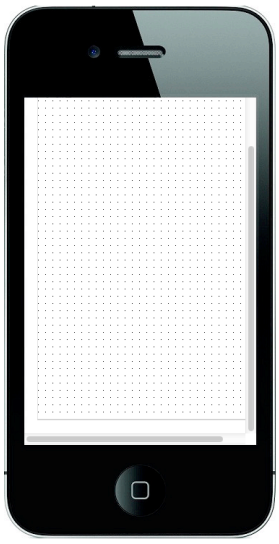


Imagen 24: Grid precision 20px

2.4.3. Agregar y eliminar componentes de la vista

Los componentes leídos en el modelo por defecto aparecen cargados en la vista del dispositivo en las posiciones indicadas en el propio modelo. Una vez tenemos los componentes cargados podemos eliminarlos de la vista. Cuando se elimina un componente de la vista no se está eliminando del modelo, sino que pasa a estar en el panel de *componentes y propiedades* bajo la pestaña *Components*, listos para ser incluidos en la vista de nuevo en cualquier momento.

2.4.3.1. Eliminar uno

Para quitar un componente de la vista, hacemos *click* derecho sobre él y pulsamos sobre la opción *(-) Remove component:*

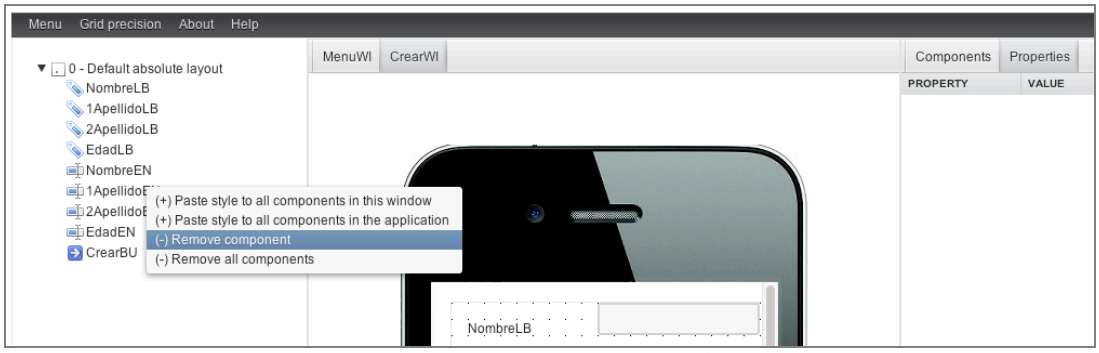


Imagen 25: Eliminar componente del árbol

El componente se moverá hasta la parte derecha de la aplicación bajo la pestaña *Components* y no se verá ni en el dispositivo ni en el árbol:

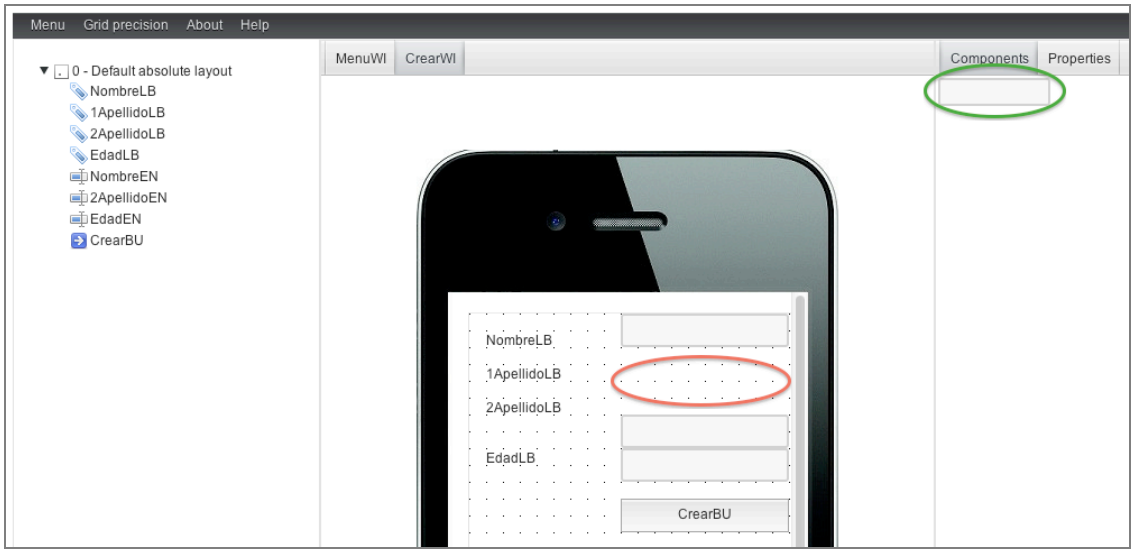


Imagen 26: Componente eliminado

2.4.3.2. Eliminar todos

Para quitar todos los componentes de la vista, hacemos *click* derecho sobre cualquiera de ellos y pulsamos sobre la opción *(-) Remove all components:*

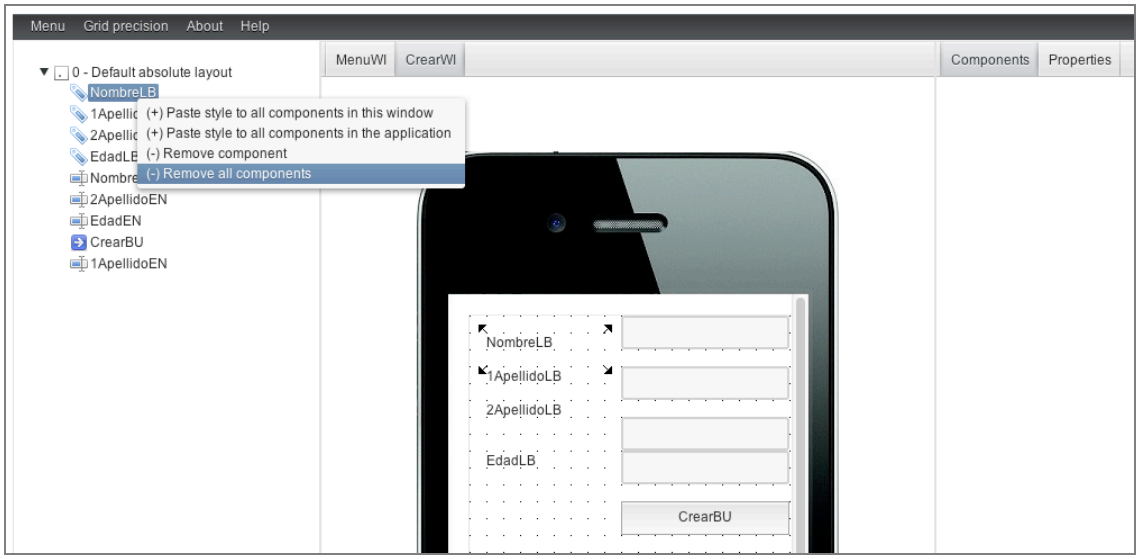


Imagen 27: Eliminar todos los componentes

Los componentes se moverán hasta la parte derecha de la aplicación bajo la pestaña *Components* y no se verá ninguno en el dispositivo ni en el árbol:

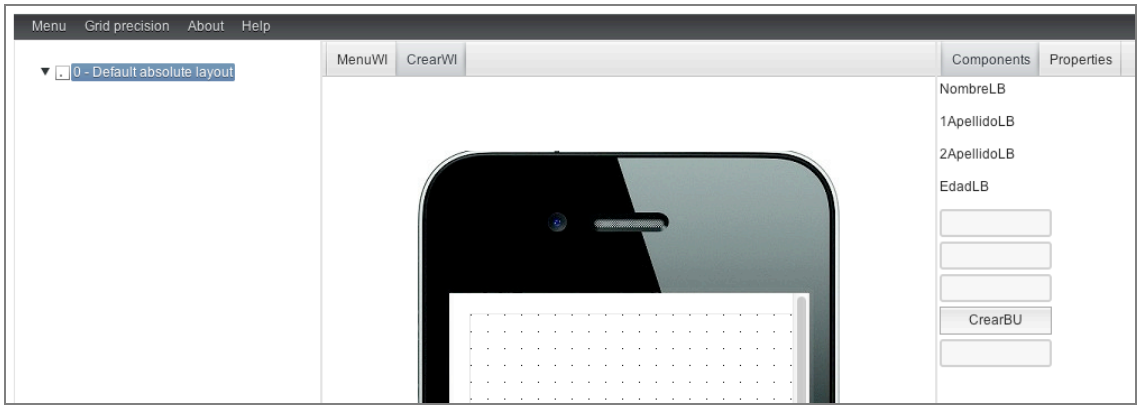


Imagen 28: Componentes borrados

2.4.3.3. Agregar componentes

Para visualizar un componente en el dispositivo hay que añadirlo previamente al árbol. Para añadir un componente al árbol tenemos dos opciones.

2.4.3.3.1. Lista desplegable

En el panel de la derecha bajo la pestaña *Components,* hacemos click derecho sobre el componente que deseamos añadir. Aparecerá un menú contextual con un listado de todos los posibles layouts donde poder añadirlo. Hacemos click en el que queramos y automáticamente aparecerá en el árbol y en el dispositivo:

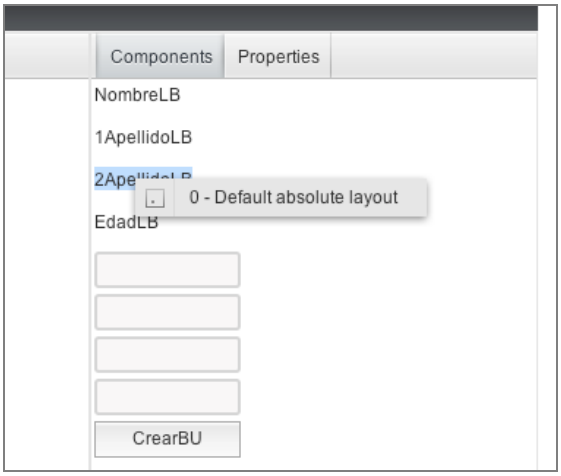


Imagen 29: Menú contextual

2.4.3.3.2. Arrastrar y soltar

En el panel de la derecha, bajo la pestaña *Components,* podemos arrastrar un componente hasta un elemento de tipo layout en el árbol de la izquierda.

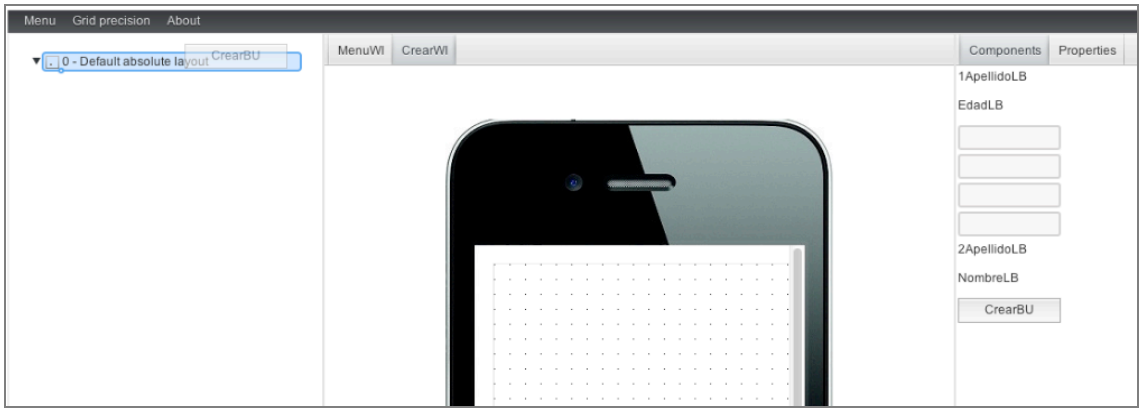


Imagen 30: Añadir componnete con *drag&drop*

Después de soltar el elemento sobre el layout, este se añadirá debajo de él y lo veremos en el dispositivo:

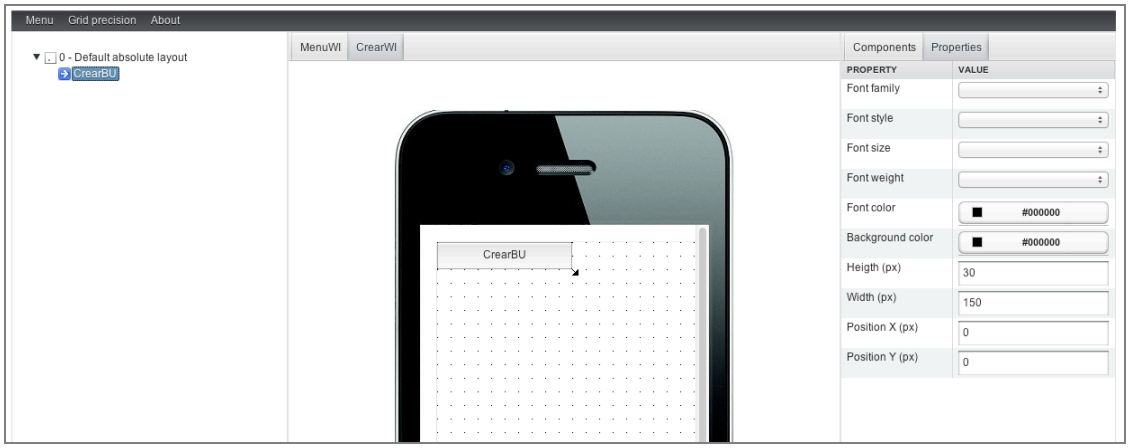


Imagen 31: Componente añadido

2.4.4. Mover componentes en la vista

Para mover un componente que ha sido añadido a la vista podemos seleccionarlo dentro del dispositivo y arrastrarlo hasta la posición deseada.

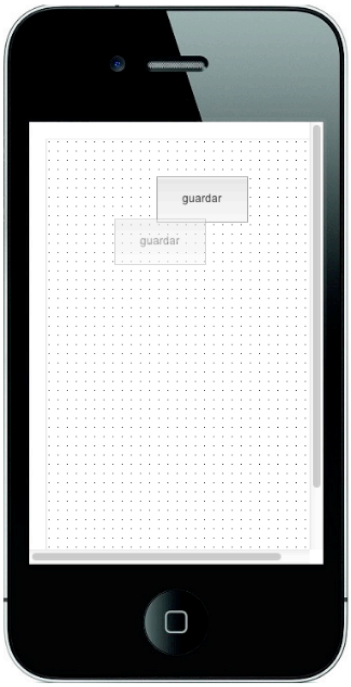


Imagen 32: Arrastar componente

En el apartado 2.4.6.2.8 mostramos otra forma de mover componentes más precisa.

2.4.5. Modificar tamaño de los componentes en la vista

Los componentes contenidos en la vista pueden ser redimensionados. Al seleccionar un componente en el árbol o directamente sobre la vista, aparecerán 4 flechas, una en cada esquina del componente, indicando en qué dirección se puede redimensionar.

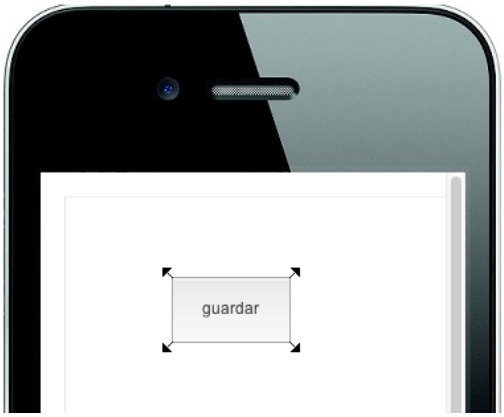


Imagen 33: Modificar tamaño

Seleccionando una de las 4 flechas y arrastrándola hasta donde queremos que se redimensione conseguiremos que tanto el alto como el ancho del componente se actualicen a los valores deseados:

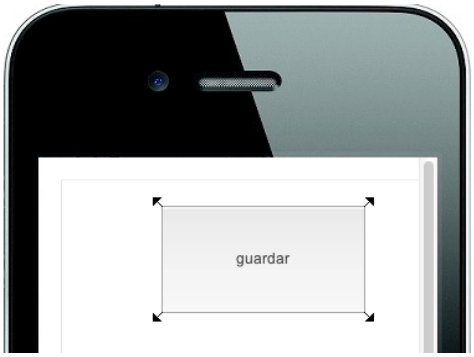


Imagen 34: Componente agrandado

En el apartado 2.4.6.2.7 mostramos otra forma de redimensionar componentes más precisa.

2.4.6. Modificar estilo de los componentes

Podemos modificar el aspecto de los componentes del modelo para ajustarlo a nuestras necesidades. En los siguientes puntos se explica cómo acceder a las propiedades los componentes y como modificarlas.

2.4.6.1. Cómo cambiar el estilo

Para acceder a las propiedades de un componente o de su contenedor basta con hacer *click* izquierdo sobre el en el árbol o en la vista. Esto resaltará el elemento en el dispositivo y cargará sus propiedades en la parte derecha de la aplicación bajo la pestaña *Properties*:

Imagen 35: Cambio de propiedades

2.4.6.2. Atributos de estilo

Cada componente tiene asociadas unas propiedades que le dan el aspecto visual. Podemos, para cada una de ellas, elegir entre varias opciones. A continuación explicamos las propiedades que podemos modificar y sus opciones.

2.4.6.2.1. Tipo de fuente

Esta propiedad nos permite elegir el tipo de fuente que tiene el texto del componente. Podemos elegir entre los siguientes:

- Serif
- Sans Serif
- Cursive
- Fantasy
- Monospace
- Arial
- **Arial Black**
- Comic sans
- Courier
- Florence
- Helvética
- Lucida
- Palatino
- Times
- Times New Roman
- Verdana

2.4.6.2.2. Estilo de la fuente

Esta propiedad nos permite elegir el estilo de la fuente que tiene el texto del componente. Podemos elegir entre:

- Normal
- Italic
- Oblique
- Inherit

2.4.6.2.3. Tamaño de fuente

Esta propiedad nos permite elegir el tamaño de la fuente que tiene el texto del componente. Podemos elegir un tamaño en el rango de 2 a 50.

2.4.6.2.4. Peso de la fuente

Esta propiedad nos permite elegir el peso de la fuente que tiene el texto del componente. Podemos elegir entre:

- Normal
- Bold
- Lighter

2.4.6.2.5. Color de la fuente

Esta propiedad nos permite elegir el color de la fuente que tiene el texto del componente. Al pinchar sobre el botón se abrirá un selector:

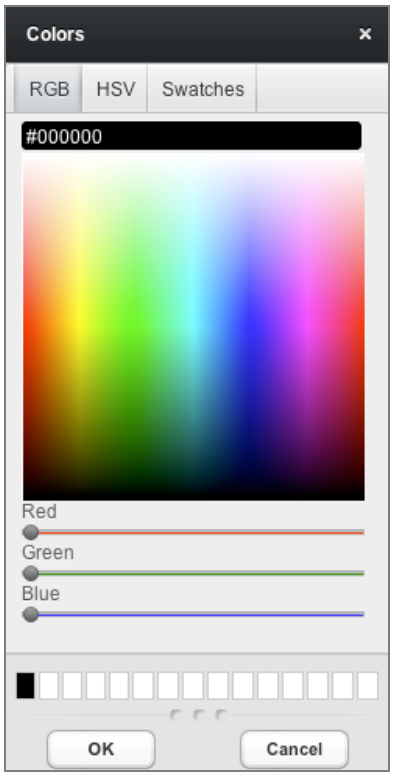


Imagen 36: Selector de color

Este selector nos permite escoger entre diferentes colores y además guarda los últimos colores seleccionados en memoria para poder aplicarlos de nuevo.

2.4.6.2.6. Color de fondo

Esta propiedad nos permite elegir el color de fondo de la fuente que tiene el texto del componente. El funcionamiento es exactamente igual que el del apartado anterior.

2.4.6.2.7. Tamaño del componente

Esta propiedad nos permite elegir el tamaño del componente tanto en altura como en anchura. Vimos en el apartado 2.4.5 cómo redimensionar un componente mediante drag&drop. Sin embargo, este método nos permite dar un tamaño exacto en píxeles.

2.4.6.2.8. Posición en la vista

Esta propiedad nos permite elegir la posición del componente en la vista. Vimos en el apartado 2.4.4 cómo mover un componente mediante drag&drop. Sin embargo, este método nos permite dar una posición exacta en píxeles.

2.4.7. Atributos que se puede cambiar de cada componente

En la siguiente tabla se resume qué atributos de estilo se pueden cambiar para cada componente.

	Tipo de fuente	Estilo de fuente	Tamaño de fuente	Peso de fuente	Color de fuente	Color de fondo	Tamaño	Posición
Botón	✓	✓	✓	✓	✓	✓	✓	✓
Etiqueta	✓	✓	✓	✓	✓	✓	✓	✓
Campo de texto	✗	✗	✗	✗	✗	✓	✓	✓
Menú desplegable	✗	✗	✗	✗	✗	✓	✓	✓
Enlace	✓	✓	✓	✓	✓	✓	✓	✓
Selección	✗	✗	✗	✗	✗	✓	✓	✓
Tabla	✗	✗	✗	✗	✗	✓	✓	✓

Tabla 1: Propiedades soportadas

2.4.8. Copiar estilo entre componentes

Otra de las funcionalidades que tiene la aplicación es la de poder copiar y pegar el estilo de un componente en otro u otros, no solo de la misma ventana si no también entre ventanas. Esta funcionalidad permite reducir el tiempo de diseño cuando se quiere que dos o más componentes tengan el mismo aspecto. Se copiarán todas las propiedades menos la posición en la vista.

2.4.8.1. Copiar estilo de un componente a otro

Para copiar el estilo de un componente a otro del mismo tipo, podemos arrastrar el componente con el estilo deseado y dejarlo caer encima del componente cuyo estilo queremos cambiar:

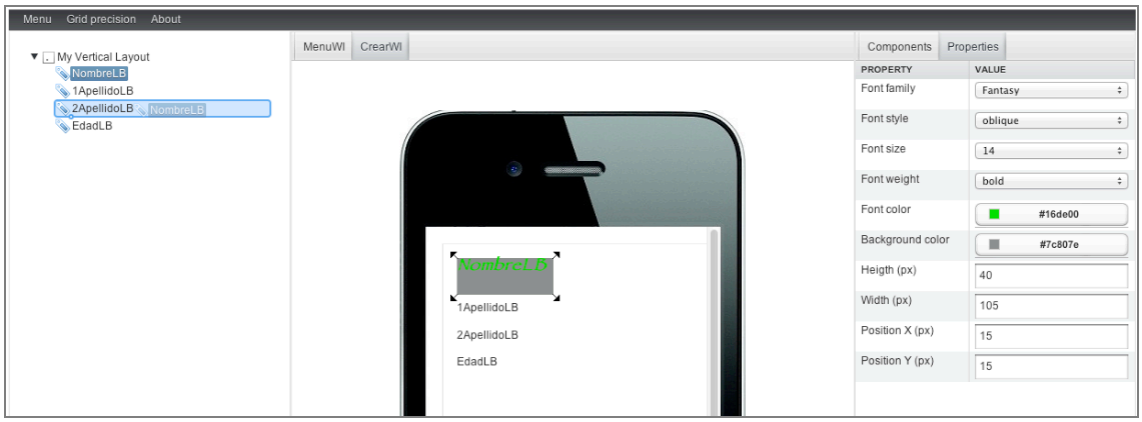


Imagen 37: Copia de estilos

Una vez que hemos soltado un componente en otro, aparecerá una ventana modal confirmando si queremos copiar el estilo del componente origen en el componente destino:

Imagen 38: Ventana de aviso

2.4.8.2. Copiar estilo a todos los componentes de la ventana

Para copiar el estilo de un componente a todos los componentes del mismo tipo dentro de la misma ventana, podemos hacer click derecho sobre el componente del que queremos copiar el estilo. Se abrirá un menú contextual en el que tendremos que escoger la opción *(+) Paste style to all components in this window*:

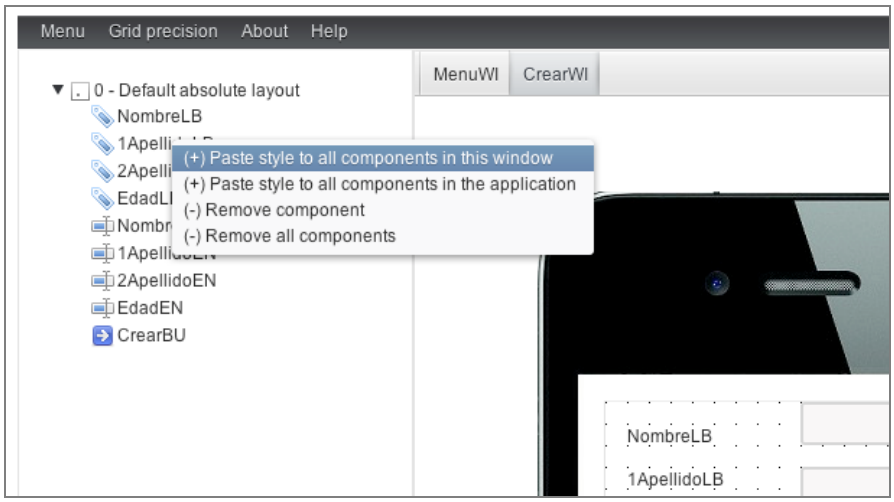


Imagen 39: Copia de estilos en una pestaña

Una vez que hemos hecho click, aparecerá una ventana modal confirmando si queremos copiar el estilo del componente origen en el resto de componentes de la misma ventana:

Imagen 40: Ventana de aviso

2.4.8.3. Copiar estilo a todos los componentes de la aplicación

Para copiar el estilo de un componente a todos los componentes del mismo tipo en toda la aplicación, podemos hacer click derecho sobre el componente del que queremos copiar el estilo. Se abrirá un menú contextual en el que tendremos que escoger la opción *(+) Paste style to all components in the application*:

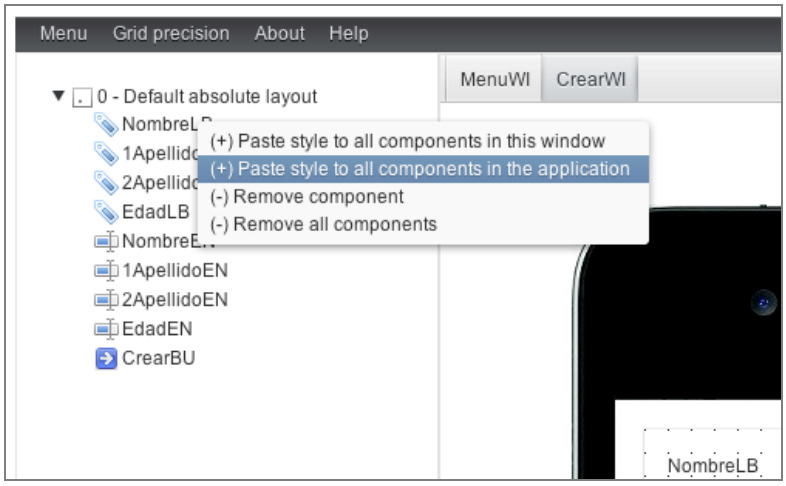


Imagen 41: Copia de estilos a todas las pestañas

Una vez que hemos hecho click, aparecerá una ventana modal confirmando si queremos copiar el estilo del componente origen en el resto de componentes de la aplicación:

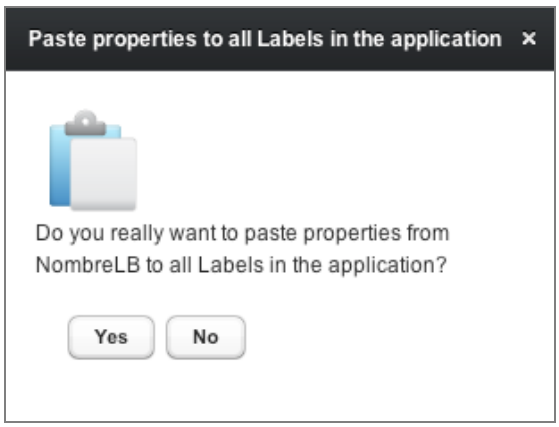


Imagen 42: Ventana de aviso

2.5. Generar nuevo modelo

Una vez hemos modificado el modelo debemos generar la salida. Para ello, tenemos que seleccionar la opción *Menu* en la barra de herramientas:

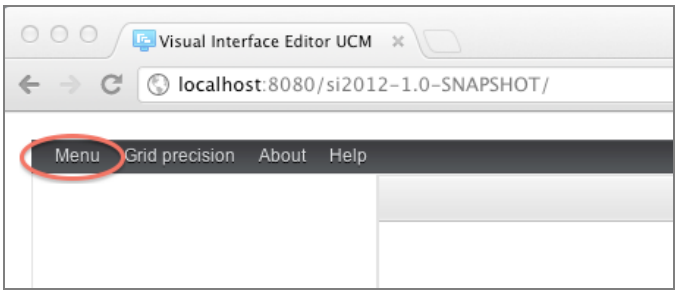


Imagen 43: Barra de menú

Una vez seleccionada, se abrirá un menú desplegable con varias opciones, de la que elegiremos *Build output:*

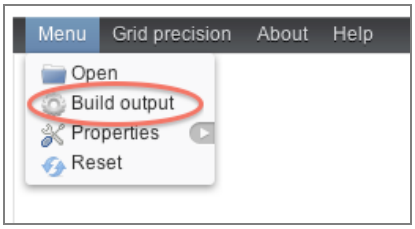
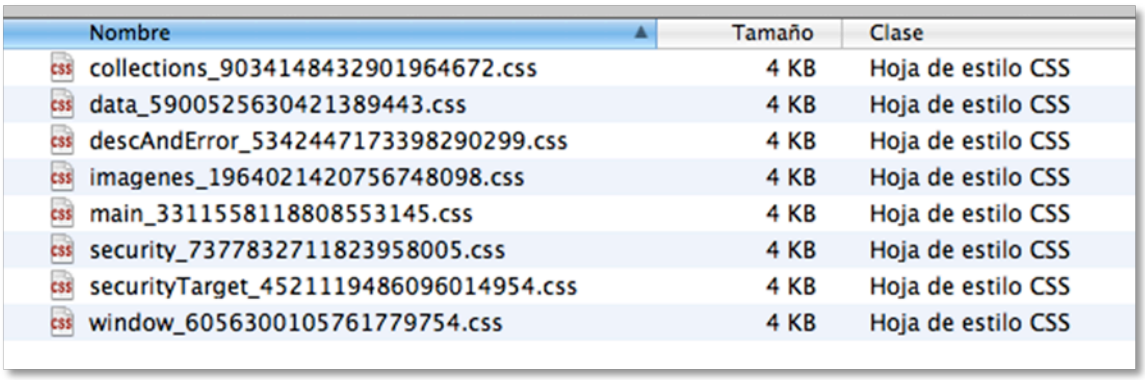


Imagen 44: Menú desplegado

Una vez seleccionada, se abrirá una ventana modal que nos permitirá descargar el nuevo modelo modificado. Para ello, hay que pulsar el botón *Build and download*:

Imagen 45: Construir la salida

Se descargará un fichero con el nombre *OutputCss.zip* que contendrá los estilos que hemos añadido sobre el modelo. Se trata de un archivo único comprimido de extensión ZIP. Dicho archivo contiene tantos archivos CSS como número de ventanas hubiese inicialmente en el modelo jasfag o lo que es lo mismo, tantos archivos como pestañas se muestren en el área de edición. Cada uno estos archivos CSS tiene el nombre del identificador de ventana que proviene del modelo jasfag seguido de una barra baja más un número que se adjudica automáticamente y de forma unívoca por el sistema de ficheros del servidor, para evitar posibles conflictos con los nombres de los archivos entre posibles usuarios. Un posible ejemplo del contenido que vería el usuario al abrir el archivo zip sería el siguiente.



Nombre	Tamaño	Clase
collections_9034148432901964672.css	4 KB	Hoja de estilo CSS
data_5900525630421389443.css	4 KB	Hoja de estilo CSS
descAndError_5342447173398290299.css	4 KB	Hoja de estilo CSS
imagenes_1964021420756748098.css	4 KB	Hoja de estilo CSS
main_3311558118808553145.css	4 KB	Hoja de estilo CSS
security_7377832711823958005.css	4 KB	Hoja de estilo CSS
securityTarget_4521119486096014954.css	4 KB	Hoja de estilo CSS
window_6056300105761779754.css	4 KB	Hoja de estilo CSS

Imagen 46: Contenido del fichero ZIP de salida

2.6. Recargar la aplicación

Ante cualquier posible error o si la sesión ha expirado por inactividad, podemos recargar la aplicación. Esto hará que toda la aplicación vuelva al estado original, como la primera vez que accedimos a ella. Para ello, tenemos que seleccionar la opción *Menu* en la barra de herramientas:

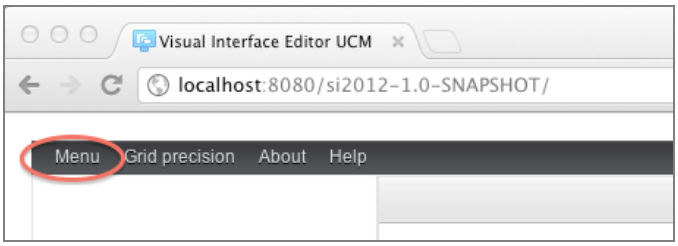


Imagen 47: Barra de menú

Una vez seleccionada, se abrirá un menú desplegable con varias opciones, de la que elegiremos *Reset:*

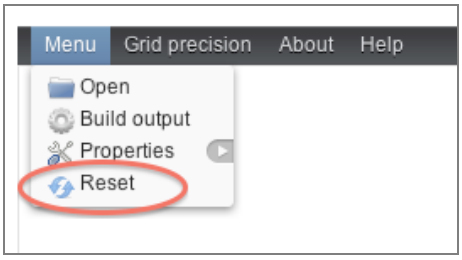


Imagen 48: Menú desplegado

A continuación, se abrirá una ventana modal confirmando si realmente queremos recargar la aplicación:

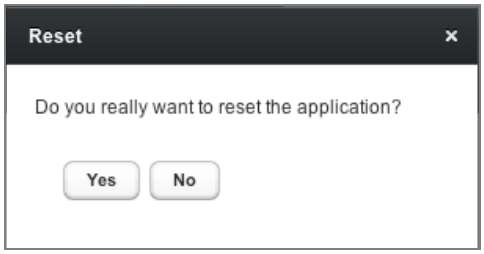


Imagen 49: Aviso

2.7. Acceder a la guía de usuario desde la aplicación

Para acceder a este manual desde la aplicación, tenemos que seleccionar la opción *Help* en la barra de herramientas:

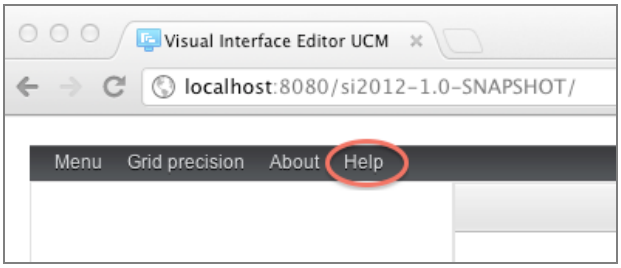


Imagen 50: Barra de menú - Help

Esto abrirá una ventana interna a la aplicación con el manual en formato pdf:

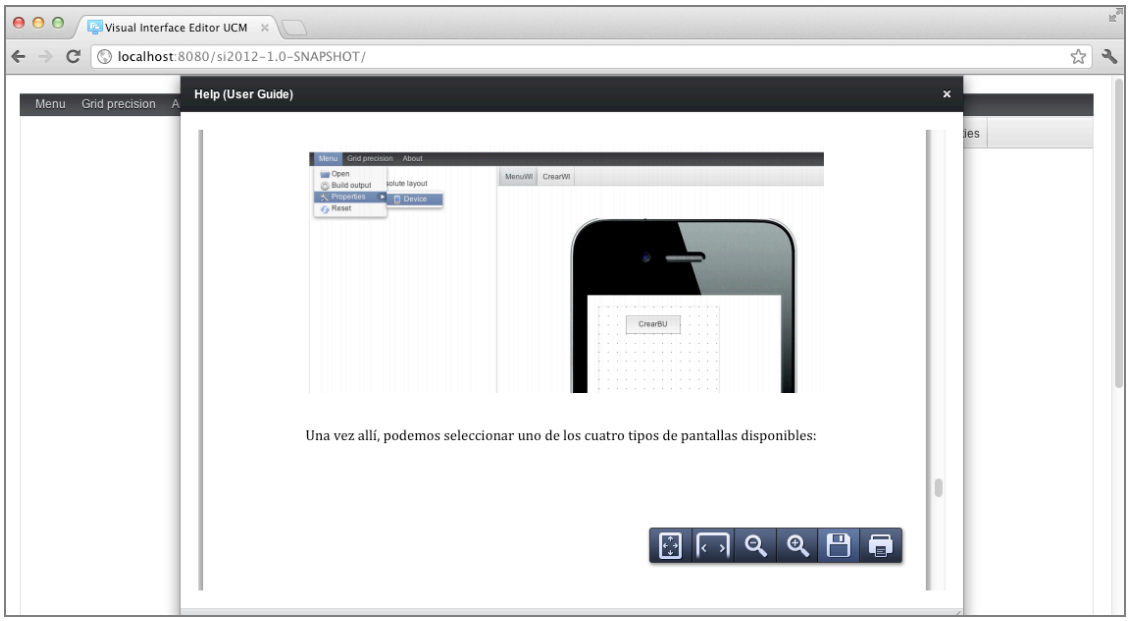


Imagen 51: Manual de usuario

3. La aplicación. Diseño e implementación.

3.1. Descripción general

La parte central de la aplicación es la que engloba el uso, gestión y transformación de componentes gráficos. Para poder realizar dichas funciones necesitamos interactuar con tres tipos de entidades diferentes.

La primera de ellas proviene de los modelos que sirven como *input* para la aplicación, los componentes gráficos del modelo jasfag. Dichos modelos se presentan en formato de ficheros XML y son leídos por nuestra aplicación, de dónde extraemos propiedades tales como el número de ventanas, identificadores unívocos de los *widgets*, tamaño, posición, etc. Una vez se obtiene el modelo y es interpretado por nuestra aplicación, el usuario es capaz de modificar dinámicamente las propiedades con el fin de obtener el *output* deseado, siendo este un conjunto de ficheros CSS. Estas ideas se representan de forma gráfica en la imagen mostrada a continuación.

Imagen 52: Esquema conceptual de la aplicación

La siguiente entidad implicada son los propios *widgets* de Vaadin, que proporcionan una gran riqueza visual además de una funcionalidad básica de la que partir a la hora de realizar las transformaciones deseadas por el usuario. No obstante y pese a ser muy útiles para realizar aplicaciones web tradicionales, presentan algunos problemas y limitaciones a la hora de manipular ciertos aspectos gráficos.

Una vez explicados estos dos elementos podemos entonces introducir el tercer componente, los propios *widgets* de nuestra aplicación a los cuales nos referiremos como *SiComponents*. Dichos componentes se encuentran como intermediarios entre los elementos gráficos provenientes del modelo jasfag y los *widgets* de Vaadin. Nos sirven como contenedores (modelo de datos) para albergar la información que el usuario genera, además de cómo elemento presentacional sirviéndonos de la riqueza visual que proporciona Vaadin.

La forma por la cuál se modificará la apariencia de los *widgets* será mediante el uso de hojas de estilos CSS. El usuario seleccionará que propiedades quiere modificar mediante la interfaz gráfica de usuario que proporcionamos. Dichos cambios se modificarán dinámicamente, mostrando la apariencia final que tendrán los *widgets* dentro de unos terminales móviles genéricos o una pantalla definida previamente. La gestión de las hojas de estilos se realizará mediante lo que denominamos *CSSManager*, siendo explicado en detalle más adelante. La interacción del usuario con los *SiComponets* y la modificación de las propiedades relativas de posición y tamaño (así como la funcionalidad de la cuadrícula de ajuste del área de edición) se realiza mediante lo que llamamos *DropPanel*, que al igual que el *CSSManager* será explicado en este mismo apartado.

3.2. SiComponent

A continuación intentaremos explicar en mayor detalle la relación de las tres entidades (elementos leídos de los modelos jasfag, los *widgets* de Vaadin y nuestros componentes, a los que denominados *SiComponents)*. Lo primero de todo es explicar la obtención de información a partir de los modelos jasfag, que son el elemento inicial para la creación de todos los *widgets* de la familia *SiComponent*. El proceso que se sigue es el siguiente:

1. Leer el fichero jasfag.xml que contiene el modelo.

2. Recuperar la información relevante para su representación gráfica hacia el usuario (tipo de widget, nombre, tamaño, posición, etc).

3. Extender la funcionalidad de los componentes Vaadin para permitir modificar su tamaño desde la interfaz gráfica de usuario, cambiar su posición mediante drag&drop, etc.

4. Utilizar hojas de estilo, transparentes al usuario, creadas dinámicamente para definir el resto de aspectos visuales que Vaadin no permite modificar, e inyectarlas en la capa del cliente.

5. Gestionar y agrupar la información, para que finalmente el usuario pueda generar la salida con todas las propiedades modificadas.

Para ejemplificar de manera gráfica, mostramos a continuación el esquema conceptual que describe la relación existente entre los *widgets* de Vaadin, el modelo de jasfag y nuestros componentes junto con las hojas CSS.

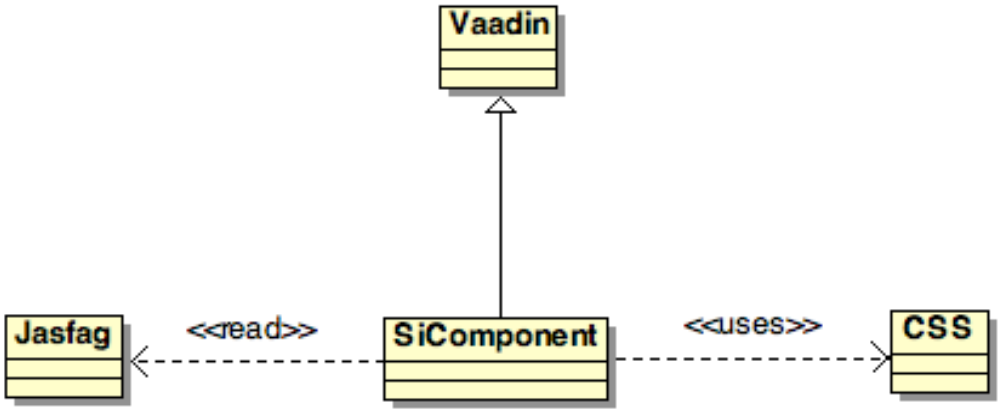


Diagrama 1: Relaciones de SiComponent

Antes de detallar cada uno de los componentes gráficos o *widgets* soportados por nuestra aplicación, mostraremos un diagrama de clases para describir las principales entidades que intervienen en la jerarquía de Vaadin. En naranja aparecen las clases abstractas, en gris las interfaces y en azul las clases regulares que conforman los *widgets* de los cuales nos servimos para dos propósitos.

Por una parte utilizamos componentes Vaadin para construir la propia aplicación y por otra, utilizamos sus *widgets* para extender a nuestros componentes gráficos, añadiendo propiedades extra para el diseño de interfaces a partir de modelos jasfag,

En la imagen que se muestra a continuación aparece la relación y jerarquía de clases de los componentes gráficos de Vaadin. Los *widgets* utilizados a partir de los cuales se crean nuestros propios componentes gráficos aparecen enmarcados dentro de un rectángulo rojo discontinuo.

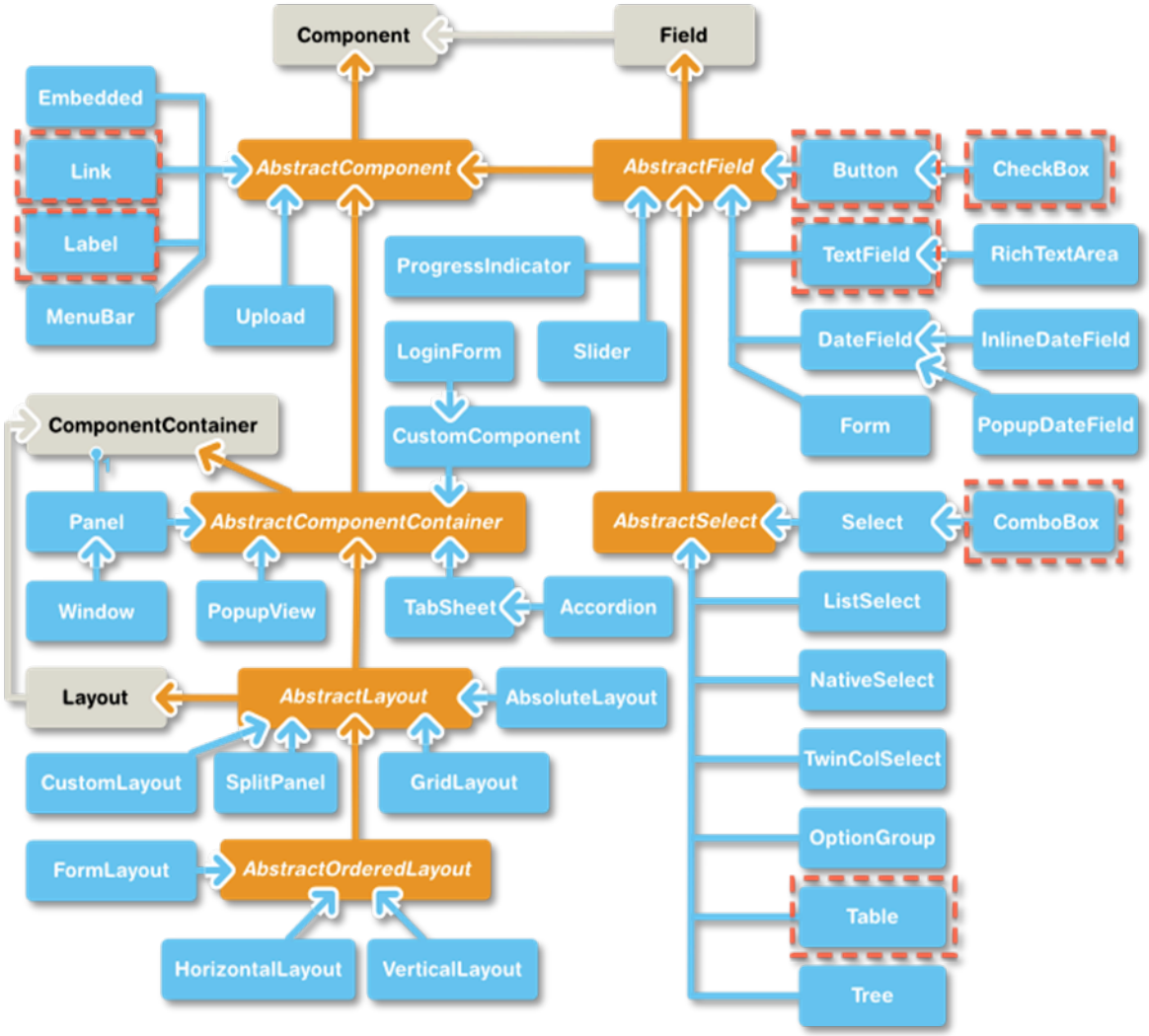


Diagrama 2: Jerarquía de clases de Vaadin

Un punto importante a destacar es que debido al desacoplamiento que utliza Vaadin entre la capa de presentación y la lógica de la aplicación, la manipulación de propiedades visuales como el cambio de color, fuente, etc. se vuelve algo compleja ya que se necesita hacer mediante hojas de estilo CSS, lo cual dista bastante del uso que se hace en Java (por ejemplo, cuando se programa en *Swing*). Esto no supondría un problema si se realizase de forma estática y se pudise introducir los estilos al generar el archivo *war* de la aplicación. Sin embargo, este no es nuestro caso puesto que queremos generar estilos personalizados por el usuario en tiempo de ejecución. Este problema y la solución tomada se abordarán más adelante.

A continuación describiremos brevemente cada unos de los *widgets* de la familia *SiComponent* soportados por nuestra aplicación, además de algunas de las complicaciones y limitaciones intrínsecas de Vaadin a la hora de hacer *widgets* avanzados de cierta complejidad. No obstante, si se quiere información más detallada acerca de los *widgets* de la familia *SiComponent*, sugerimos leer el Anexo II al final de este documento.

1. SiButton: Tiene la funcionalidad clásica del botón. En lo relativo a la modificación de la apariencia visual presentaba problemas con los CSS que Vaadin tiene por defecto, por lo que tuvimos que modificarlos. Además de eso, presentaba problemas con los eventos de pulsado, teniendo que redefinirlos para conseguir el comportamiento deseado.

2. SiLabel: Los *labels* se utilizan para presentar información textual al usuario, permitiendo también incluir imágenes de forma adicional. Son elementos análogos a los *labels* existentes en Swing o en otras APIs de interfaces gráficas de usuario.

3. SiLink: Los links son elementos textuales muy similares a los *labels*, con la particularidad de que el usuario puede hacer click sobre ellos para dirigirse a otra URL. En nuestro caso particular, no existen peculiaridades significativas que comentar.

4. SiCheckBox: Los *checkboxes* son elementos de valor binario (marcado o no marcado) que pueden ir acompañados de texto y de alguna imagen. Con respecto a este componente cabe comentar que se tuvo que deshabilitar los eventos de pulsación sobre él para poder obtener el comportamiento deseado, evitando su activación y desactivación a la hora de hacer click sobre él.

5. SiTextfield: Los *textfield* son elementos de entrada, y ocasionalmente de salida, de texto. Se produjeron problemas a la hora de redimensionar estos componentes, relativos a la apariencia visual que proporcionan sus CSS por defecto. Para solucionarlo se tuvo que reescribir la hoja de estilos. También hubo que desactivar su entrada de texto y algunos eventos de pulsación.

6. SiComboBox: El *combobox* es un elemento que permite seleccionar un elemento concreto entre un conjunto de valores posibles, mediante una lista desplegable. Para este componente encontramos limitaciones tanto en su apariencia visual a la hora de modificar su tamaño, como en lo relativo a la captura de eventos predefinida, lo que nos obligó a redefinir estas características.

7. SiTable: Las tablas son elementos que permiten la presentación de información textual al usuario, contener otras tablas dentro de sí mismas, imágenes, botones y un largo número de componentes gráficos.

En este caso tuvimos también problemas con los eventos a la hora de desplazar con *drag&drop*. Además de esto, se tuvo que crear un CSS nuevo para poder representar su información gráficamente sin sufrir distorsión al modificar su tamaño.

3.3. Manipulación de propiedades gráficas - CSSManager

Como ya se ha mencionado, la parte central de la aplicación aparte del uso y manejo de los componentes gráficos, es la funcionalidad relativa a las propiedades visuales de los *widgets*. Estas propiedades hacen referencia al color de fondo, el tamaño de fuente, color del texto, tamaño y posición del *widget*, etc. La mayoría de estas propiedades se modifican mediante hojas de estilo generadas dinámicamente por el usuario.

A continuación mostramos la vista de propiedades en el contexto de la aplicación, resaltada en un recuadro azul discontinuo en la parte derecha de la imagen.

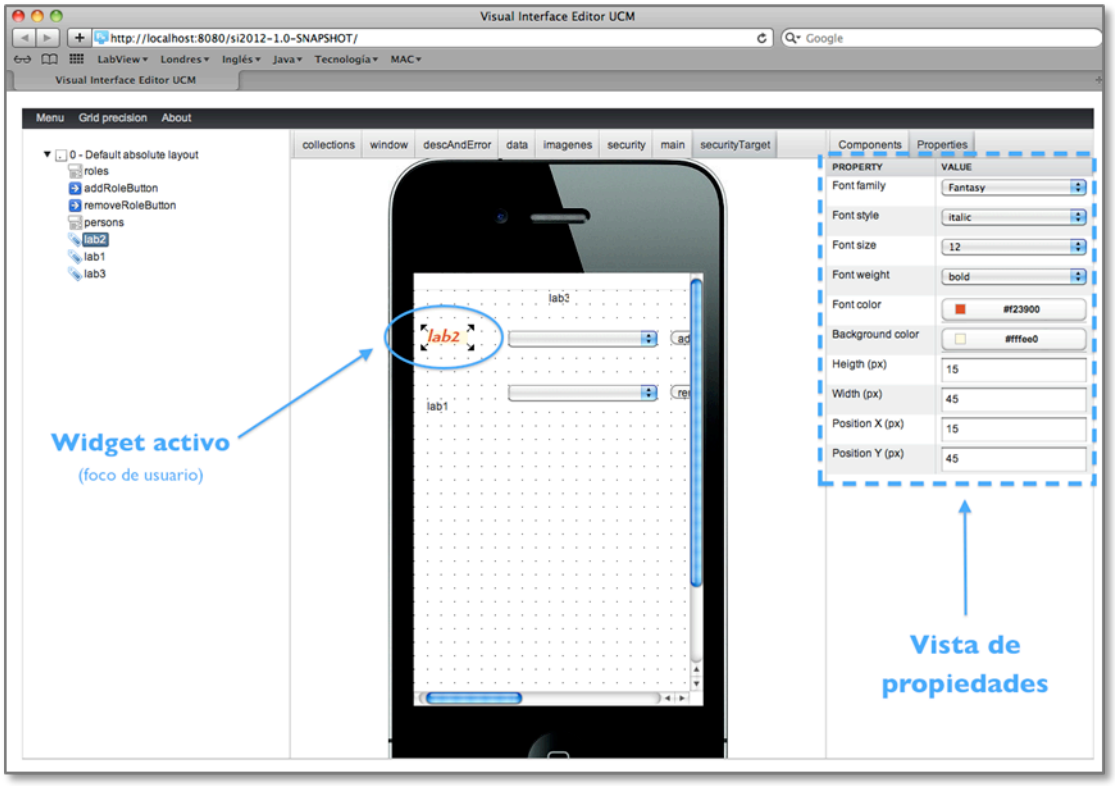


Imagen 53: Propiedades del componente

Como ya hemos comentado, los propiedades de los *widgets* se modifican mediante el uso de CSS. Esto es debido a que en Vaadin la lógica de la aplicación está completamente desacoplada de la capa de presentación.

Esto puede suponer muchas ventajas a la hora de realizar aplicaciones convencionales pero en nuestro caso, supuso un problema a la hora de aplicar el cambio de propiedades y su generación de forma dinámica. Por ejemplo, algunas de las dificultades y consideraciones que se tuvieron que tener en cuenta fueron las siguientes:

- ✓ Cada *widget* ha de tener sus propias reglas CSS.
- ✓ Las propiedades de cada *widget* se han de mostrar en la vista de propiedades cuando obtiene el foco de usuario.
- ✓ Las propiedades se deben modificar dinámicamente desde la vista de propiedades.
- ✓ La información de las hojas de estilo se debe poder generar de diferentes formas y poder serializarse.
- ✓ Cada vez que se realiza un cambio en una propiedad, ha de verse su resultado automáticamente en el área de edición. Esta actualización ha de hacerse en cada pestaña por separado.
- ✓ Se debe saber en que ventana (pestaña de edición) se encuentran los *widgets* para generar los CSS.

No obstante, antes de comentar dichas dificultades intentaremos explicar en detalle el modelo de datos y el proceso que se sigue con la gestión de las propiedades. Antes de nada mostraremos un pequeño diagrama que ayude a entender la explicación.

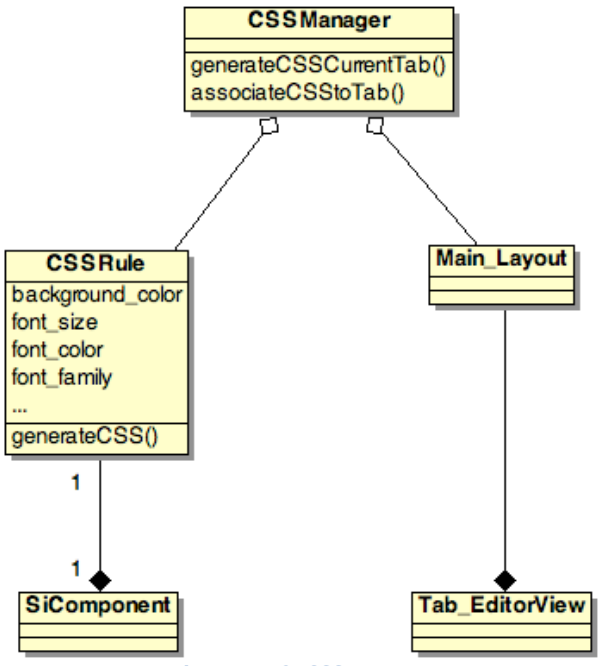


Diagrama 3: CSSManager

La clase CSSRule es la encargada de almacenar y gestionar la información individual de los SiComponents. Tiene operaciones para introducir cada una de las propiedades soportadas y otras para poder construir reglas específicas que no hayan sido previstas en un principio. También genera las propiedades CSS para un SiComponent concreto en los diferentes formatos (formato reducido para realizar las actualizaciones de forma dinámica y el formato humanizado, que está formateado para que sea legible por una persona).

Los widgets están agrupados por ventanas según el modelo jasfag. En nuestro caso, esas ventanas tienen una correspondencia directa con las pestañas de edición: cada pestaña tiene un conjunto de widgets. Por lo tanto, cada vez que se hace un cambio en un SiComponent hay que saber en qué pestaña está contenido. Esta relación se lleva a cabo mediante el CSSManager. Esta clase se encarga de llevar una correspondencia entre las propiedades CSS de los componentes gráficos y los layouts de las pestañas en las que han de mostrar. Además de eso se encarga de la generación de las hojas de estilos completas para cada una de esas pestañas. No hace falta decir que estas funcionalidades son sumamente importantes, además de ser un aspecto crítico de rendimiento ya que se debe modificar la información en el SiComponent correspondiente, volver a generar todos las propiedades CSS para esa ventana de edición e inyectar la hoja de estilos dinámicamente, para poder presentar la información actualizada al usuario.

Una vez tenemos todo esto, resulta más sencillo mostrar los valores de las propiedades de un SiComponent en su vista correspondiente, al igual que modificar sus valores mediante el controlador de la aplicación. A la hora de mostrar las vistas de propiedades correspondientes a un SiComponent utilizamos otro manager. Este se encarga de decidir qué vistas se han de mostrar en función del widget que recibe y de fijar sus nuevos valores. De esta forma se pueden ampliar las propiedades de los widgets sin tener que modificarlos. Además permite realizar diferentes operaciones de configuración sobre estas, si fuese necesario.

Otro problema principal al que tuvimos que hacer frente no guarda relación con la arquitectura de aplicación ni con su modelo de datos. Es una barrera a nivel tecnológico, propia de Vaadin y de la arquitectura Cliente-Servidor. El problema surge cuando el usuario modifica alguna propiedad que se ha de reflejar mediante una nueva regla CSS, ya que ésta se ha de mostrar automáticamente para que el usuario vea su efecto. Las hojas de estilos son descargadas por el navegador para interpretarlas y así poder presentar la información visual correspondiente. El problema viene cuando esas hojas de estilo se han de generar de manera dinámica, ya que no se puede manejar información en el sistema de ficheros de la máquina cliente por motivos de seguridad. La opción de manipular ficheros dentro no es posible ya que al generar el *war* de la aplicación no se puede manejar su contenido. La opción de trabajar con ficheros en el servidor planteaba un problema de escalabilidad a la hora de aumentar el número de usuarios; además podría generar retardos importantes en los tiempos de respuesta.

Este problema se podía evitar, lógicamente, limitando las opciones de selección al usuario de manera que sólo pudiese elegir entre unas opciones fijadas de antemano. Esta solución, como se puede suponer, no era muy satisfactoria. Se optó finalmente por investigar más a fondo la cuestión hasta dar con un Add-on (los *addons* son extensiones o librerías que se crean como complemento, para suplicar carencias o mejorar funcionalidades de Vaadin) que justamente solucionaba este problema. El *addon* se llama CSSInject y como su propio nombre indica, lo que hace es inyectar hojas de estilo CSS en las etiquetas HTML de los *widgets* para poder mostrar estilos dinámicamente.

En la siguiente imagen mostramos información del *addon* utilizado y de su autor.

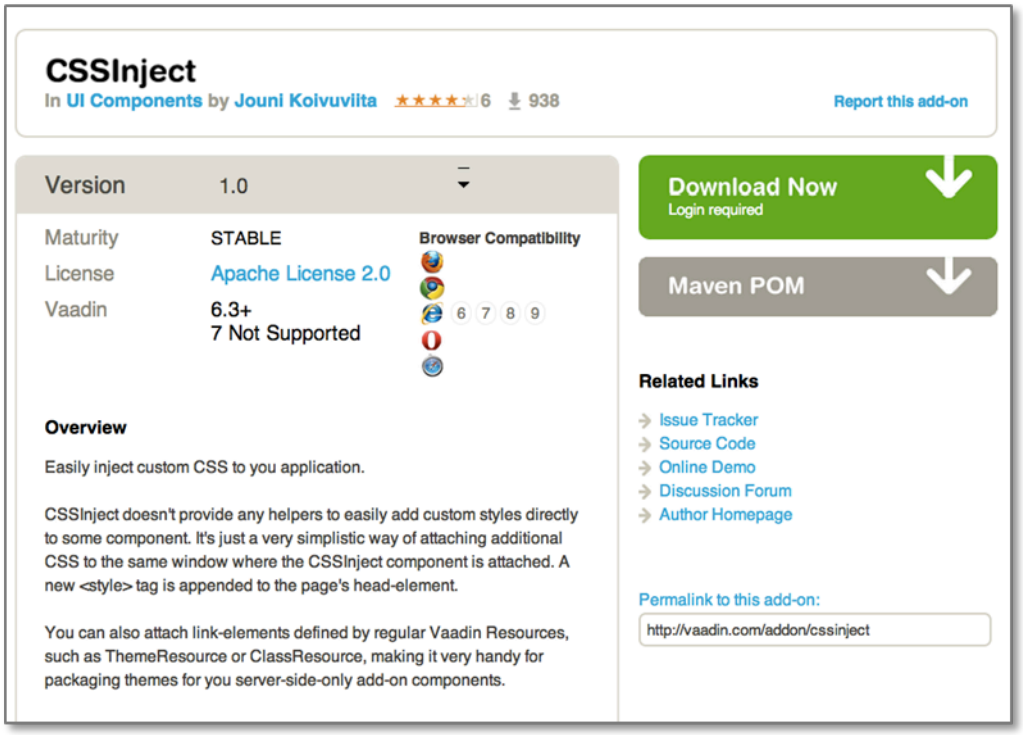


Imagen 54: Addon CSSInject

3.4. Manipulación de propiedades gráficas - DropPanel

Como se ha mencionado en el apartado anterior, la edición de las propiedades gráficas o estilos de los widgets, es una de las partes centrales de la aplicación. Es decir, necesitábamos de algún modo, ser capaces de modificar y visualizar dinámicamente las diferentes propiedades de estilos que puede poseer un componente. Como ya se ha mencionado, estas eran el color, el tamaño de fuente, el tipo de fuente, etc.

No obstante, para que la edición de un determinado grupo de componentes gráficos sea completa, a parte de modificar sus propiedades de estilo, deberemos ser capaces de modificar sus propiedades de tamaño y posición. Además, necesitamos que esta edición sea dinámica de cara al cliente. Es decir, cada vez que cambiemos alguna de estas propiedades, queremos que, inmediatamente, se muestren los cambios en el área de edición.

Para facilitar la edición del tamaño y posición de los widgets por parte del cliente, decidimos que esto se realice haciendo click y arrastrando el componente por el área de edición. Es en este momento, donde nace el DropPanel, un Panel convencional, que además de permitir añadir y quitar componentes gráficos (como cualquier panel), permite editar el tamaño y la posición mediante el uso exclusivo del ratón.

Para implementar este módulo, hemos utilizado el drag and drop en diversos elementos, tal como se describe en el siguiente diagrama de clases, donde comentaremos sus características más importantes.

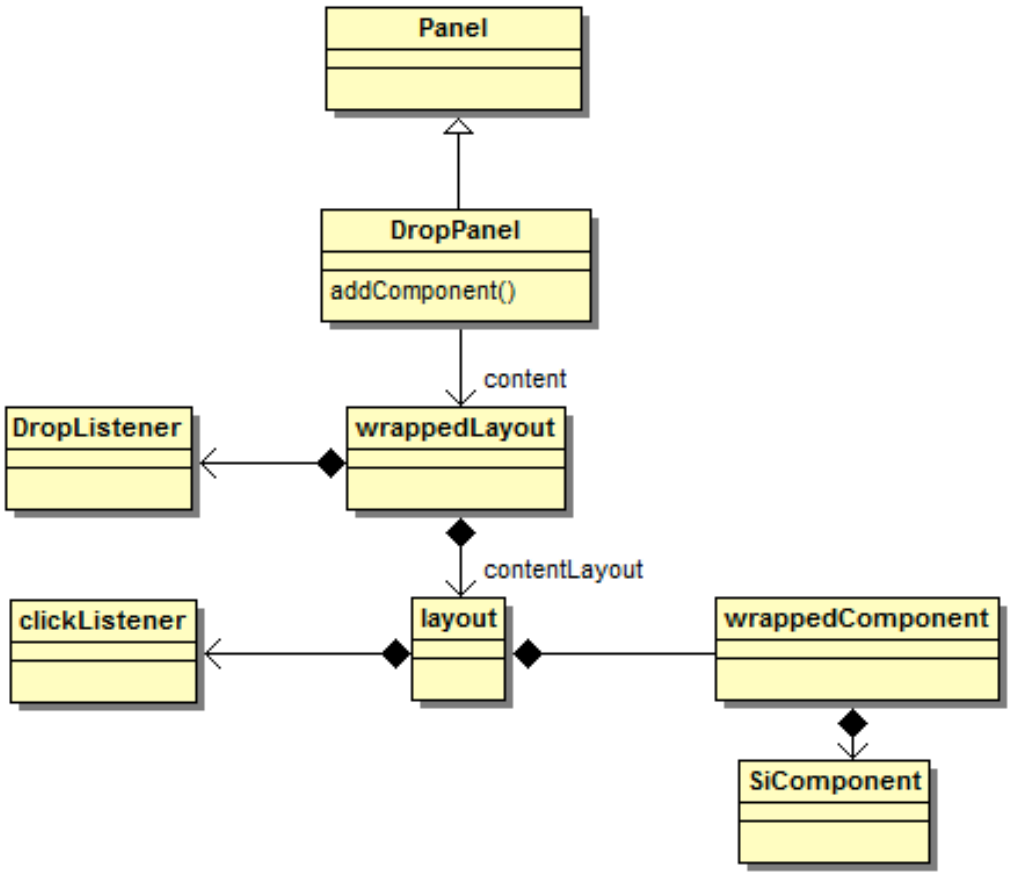


Diagrama 4: Relaciones de DropPanel

3.4.1 DropPanel

El DropPanel es la clase principal de esta jerarquía, ya que es la clase que contiene al resto de clases. Es decir, la instancias del resto de las clases no tienen sentido si no pertenecen a un DropPanel.

El DropPanel proporciona toda la interfaz pública para gestionar todo este entramado de objetos y así permitir de una forma sencilla el redimensionado y el cambio de posición de los objetos en el mismo.

Estos mecanismos, se podrían resumir en métodos para añadir y retirar SiComponents(Widgets), así como modificarlos. El DropPanel será el encargado de envolver estos componentes en un DragAndDropWrapper de manera transparente para el usuario. De esta forma, cuando se añade un componente normal, este se transforma automáticamente en un componente que se puede arrastrar y soltar por los diferentes elementos que conforman el drag and drop.

El contenido de este panel será de un Layout que está envuelto por un DragAndDropWrapper. Esta operación se realiza para que el contenido del panel permita drag&drop. Es decir, como queremos poder arrastrar diferentes elementos por el contenido del citado DropPanel, utilizamos este mecanismo para conseguirlo.

3.4.2 WrappedLayout

Como se ha comentado anteriormente, este componente de la jerarquía es un layout contenido en un DragAndDropWrapper para permitir el drag and drop. Como se puede observar en el diagrama de clases, este elemento está relacionado con un DropListener y con un Layout (que en nuestro caso, es un AbsoluteLayout)

3.4.3 DropListener

El DropListener es el encargado de gestionar los diferentes elementos que se van arrastrando y soltando sobre el WrappedLayout. De este modo, se permite gestionar de forma diferente a los elementos que se están soltando desde un panel diferente, los elementos que se están arrastrando desde el mismo panel, o incluso los elementos especiales que se usan para la redimensión de componentes. Dependiendo del elemento que sea soltado, el dropListener será el encargado de decidir si no se hace nada porque es una operación inválida, se añade un componente, o se le cambia la posición o el tamaño.

3.4.4 ClickListener

Como se puede comprobar mediante el uso de la aplicación, el hecho de realizar un click sobre uno de los componentes o widgets, provoca que este quede seleccionado.

La tecnología utilizada no implementaba esto de forma nativa, por lo que tuvimos que registrar un listener para los clicks.

Cada vez que se efectúa un click en el dropPanel, se produce un evento de click. Dicho evento es propagado hasta el layout contenido en el wrapper que a su vez está contenido en el DropPanel. El Layout, al registrar el evento, llama al clickListener, indicándole que se ha realizado un click.

De este modo, la tarea del clickListener es la de gestionar cada click, y, en caso de que las coordenadas del click coincida con las coordenadas de algún componente, indicar al dropPanel que se le ha hecho click sobre un componente, e indicarle cuál.

3.4.5 Wrapped Component

Como se ha comentado anteriormente, para permitir que los elementos tengan las propiedades de drag and drop, se necesita que estos estén contenidos dentro de un DragAndDropWrapper. Así, el wrappedComponent no es más que un DragAndDropWrapper que contiene a un Componente para permitir así que este pueda ser arrastrado por el DropPanel.

4. Entorno de desarrollo y tecnologías utilizadas

4.1. Introducción

La elección de las tecnologías para llevar a cabo este proyecto, se fundamentan en un pilar básico: La completa integración de nuestra aplicación con la tecnología ActionGUI. El hecho de que ActionGUI se estuviera desarrollando en VAADIN fue la razón principal para que nosotros adoptáramos esta tecnología. No debemos olvidar que VAADIN trabaja sobre la plataforma J2EE. Esto determinó el resto de tecnologías que hemos empleado.

4.2. Entornos de desarrollo

4.2.1. Introducción

Actualmente, existen numerosos IDE's para el desarrollo en JAVA. Nosotros elegimos utilizar Netbeans y Eclipse por diversas razones. Además de ser un software gratuito, son entornos en los que ya estamos acostumbrados a trabajar. Además, el resto de tecnologías que íbamos a utilizar estaban completamente integradas con ambos entornos mediante multitud de *plugins*.

4.2.2. Eclipse

Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados.

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

Eclipse fue liberado originalmente bajo la Common Public License, pero después fue re-licenciado bajo la Eclipse Public License. La Free Software Foundation ha dicho que ambas licencias son licencias de software libre, pero son incompatibles con Licencia pública general de GNU (GNU GPL).

4.2.3. NetBeans

NetBeans es un entorno de desarrollo integrado libre, hecho principalmente para el lenguaje de programación Java. Existe además un número importante de módulos para extenderlo.

NetBeans es un proyecto de código abierto de gran éxito con una gran base de usuarios, una comunidad en constante crecimiento, y con cerca de 100 socios en todo el mundo. Sun MicroSystems fundó el proyecto de código abierto NetBeans en junio de 2000 y continúa siendo el patrocinador principal de los proyectos.

La plataforma NetBeans permite que las aplicaciones sean desarrolladas a partir de un conjunto de componentes de software llamados módulos. Un módulo es un archivo Java que contiene clases de java escritas para interactuar con las APIs de NetBeans y un archivo especial (manifest file) que lo identifica como módulo. Las aplicaciones construidas a partir de módulos pueden ser extendidas agregándole nuevos módulos. Debido a que los módulos pueden ser desarrollados independientemente, las aplicaciones basadas en la plataforma NetBeans pueden ser extendidas fácilmente por otros desarrolladores de software.

4.3. Sistema de control de versiones (RCS)

4.3.1. Introducción

Como en la mayoría de proyectos software, pensamos que era necesario un sistema de control de versiones para que gestionara todos los cambios que íbamos realizando en el proyecto. Entre las diferentes alternativas que valoramos, como Git o Mercurial, acabamos eligiendo Subversion, ya que, aunque es más antigua que las otras dos, y tiene menos potencia, era con la que estábamos más familiarizados. Finalmente, de entre todas las ofertas de repositorios que existen, acabamos eligiendo xp-dev, ya que además de gratuita, se puede elegir si el repositorio es privado.

4.3.2. Repositorio

Un repositorio, depósito o archivo es un sitio centralizado donde se almacena y mantiene información digital, habitualmente bases de datos o archivos informáticos.

4.3.3. Sistema de control de versiones

Revisión Control System o RCS es una implementación en software del control de versiones que automatiza las tareas de guardar, recuperar, registrar, identificar y mezclar versiones de archivos. RCS es útil para archivos que son modificados frecuentemente, por ejemplo programas informáticos, documentación, gráficos de procedimientos, monografías y cartas. RCS también

puede ser utilizado para manejar archivos binarios, pero con eficacia y eficiencia reducidas.

4.3.4. Subversion

Subversion es un sistema de control de versiones diseñado específicamente para reemplazar al popular CVS. Es software libre bajo una licencia de tipo Apache/BSD y se le conoce también como svn por ser el nombre de la herramienta utilizada en la línea de comando.

Subversion puede acceder al repositorio a través de redes, lo que le permite ser usado por personas que se encuentran en distintas computadoras. A cierto nivel, la posibilidad de que varias personas puedan modificar y administrar el mismo conjunto de datos desde sus respectivas ubicaciones fomenta la colaboración. Se puede progresar más rápidamente sin un único conducto por el cual deban pasar todas las modificaciones.

4.4. Gestión del proyecto

4.4.1. Introducción

El hecho de desarrollar en VAADIN, nos suponía diversos problemas que íbamos a tener que solucionar:

-La necesidad de un servidor para poder desplegar la aplicación que íbamos a desarrollar.

-La necesidad de un plugin para Eclipse o Netbeans que nos permitiera crear proyectos VAADIN, así como añadir y borrar dependencias, ya que probablemente tuviéramos que extender la propia funcionalidad de VAADIN con diversos ADDONS.

-Algunos de los módulos de nuestra aplicación (como el parseador del fichero jasfag), se encontraban en repositorios.

Por estos motivos, consideramos necesario el uso de una herramienta que nos simplificase todas estas tareas de gestión del proyecto, y, finalmente, decidimos utilizar Maven.

4.4.2. Maven

Maven es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Es similar en funcionalidad a Apache Ant (y en menor medida a PEAR de PHP y CPAN de Perl), pero tiene un modelo de configuración de construcción más simple, basado en un formato XML. Estuvo integrado inicialmente dentro del proyecto Jakarta pero ahora es ya un proyecto de nivel superior de la Apache Software Foundation.

Maven utiliza un Project Object Model (POM) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Viene con objetivos predefinidos para realizar ciertas tareas, como la compilación del código y su empaquetado.

Una característica clave de Maven es que está listo para usar en red. El motor incluido en su núcleo puede dinámicamente descargar plugins de un repositorio, el mismo repositorio que provee acceso a muchas versiones de diferentes proyectos Open Source en Java, de Apache y otras organizaciones y desarrolladores. Este repositorio y su sucesor reorganizado, el repositorio Maven 2, pugnan por ser el mecanismo de facto de distribución de aplicaciones en Java, pero su adopción está siendo muy lenta.

Maven provee soporte no sólo para obtener archivos de su repositorio, sino también para subir artefactos al repositorio al final de la construcción de la aplicación, dejándola al acceso de todos los usuarios. Una caché local de artefactos actúa como la primera fuente para sincronizar la salida de los proyectos a un sistema local.

Maven está construido usando una arquitectura basada en plugins que permite que utilice cualquier aplicación controlable a través de la entrada estándar. En teoría, esto podría permitir a cualquiera escribir plugins para su interfaz con herramientas como compiladores, herramientas de pruebas unitarias, etcétera, para cualquier otro lenguaje. En realidad, el soporte y uso de lenguajes distintos de Java es mínimo en la actualidad.

4.5. Tecnologías web

4.5.1. Google Web Toolkit

Conscientes de los problemas inherentes de JavaScript, Google lanzó Google Web Toolkit (GWT) en 2006. Como es habitual en el desarrollo de aplicaciones web, GWT impone diferenciar la parte de la aplicación que se ejecutará en el servidor y la parte que se ejecutará en el cliente. A diferencia de otros frameworks, GWT permite que ambas partes estén programadas en Java.

Durante el compilado, la parte del servidor sigue el proceso común de cualquier aplicación Java, generando el correspondiente Java bytecode que interpretará la máquina virtual Java del servidor. Sin embargo los navegadores web no son capaces de interpretar Java bytecode, sino HTML y JavaScript. GWT primero compila las clases Java del cliente a bytecode y, partiendo de este código, genera páginas HTML con sus correspondientes funciones JavaScript. Si el proceso acabase aquí realmente no ofrecería grandes ventajas respecto a otros frameworks, pero la aplicación del lado del cliente generada cumple las siguientes propiedades:

1. Garantiza que el código JavaScript funcionará en cualquiera de los grandes navegadores web tanto de escritorio como de dispositivos móviles.

2. Garantiza que el código JavaScript generado será eficiente.

3. Tiene varios niveles de compilación, pasando de los más descriptivos a los más ofuscados. Los primeros son útiles mientras se desarrolla la aplicación para comprender qué tareas realiza cada script, mientras los segundos dificultan la lectura a un cliente malicioso y además reducen sustancialmente el número de caracteres de código generado, lo cual aumenta el rendimiento del ancho de banda del servidor.

4. Permite hacer llamadas al servidor mediante llamadas a procedimientos remotos (Remote Procedure Call o RPC). El servidor recibe una petición, realiza los cálculos pertinentes y devuelve la información.

5. Existen extensiones para añadirlo a cualquiera de los grandes entornos de desarrollo Java, prestando soporte opcional para Eclipse.

6. Aunque en la aplicación terminada la parte del cliente esté escrita en HTML y JavaScript, durante la fase de desarrollo la aplicación puede ser depurada sobre el código Java, usando para ello las herramientas que el programador considere más oportunas incluyendo la depuración paso a paso.

Además, al igual que AWT y Swing, las librerías gráficas de GWT están basadas en componentes que disparan eventos que a su vez son atendidos por oyentes, por lo que desarrollar aplicaciones web con GWT resulta familiar a un desarrollador de aplicaciones de escritorio.

Una prueba del éxito de GWT son los frameworks de terceros que se basan en él. Algunos de ellos extienden GWT con nuevos componentes, como en el caso de SmartGWT. Otros, como Vaadin, añaden una nueva capa de abstracción.

4.5.2. Vaadin

Vaadin es un framework de desarrollo de aplicaciones web que, al contrario que GWT, se caracteriza porque toda la lógica de la aplicación está en el servidor. En GWT, cada vez que del lado del cliente se produce un evento (pulsar una tecla, hacer un click, etc) el cliente realiza la lógica asociada y sólo en caso de necesitarlo, recurre al servidor con una llamada RPC. En Vaadin el cliente no mantiene la lógica de la aplicación, sino que únicamente sabe qué componentes gráficos se muestran y cómo comunicar los eventos al servidor. El servidor, por otra parte, almacena toda la lógica y una copia del estado de la interfaz gráfica del usuario. Cuando se produce un evento, el cliente notifica al servidor que este se ha producido, el servidor ejecuta la lógica, modifica el estado de visualización de la aplicación y se lo comunica al cliente, que actualiza su estado si es necesario.

El modelo centrado en el cliente que usa GWT tiene las ventajas de consumir un menor ancho de banda y que la carga computacional del servidor es menor, puesto que delega gran parte del cálculo en el cliente. Por otro lado el modelo centrado en el servidor facilita la implementación de una política de seguridad y, de cara al programador, no es necesario preocuparse en absoluto de la sincronía entre el cliente y el servidor, pudiendo ver el sistema como una aplicación que se ejecutará en una misma máquina.

Las versiones modernas de Vaadin usan GWT en el lado del cliente. Por cada componente Vaadin en el servidor existe un componente en el lado del cliente. Estos componentes de cliente tienen un núcleo GWT y una capa Vaadin que se encarga de implementar la sincronización entre el estado en el cliente y el servidor.

4.5.3. Hojas de estilo CSS

El lenguaje HTML está limitado a la hora de aplicar forma a un documento. Esto es así porque fue concebido para otros usos (científicos sobre todo), distinto a los actuales, mucho más amplios.

Para solucionar estos problemas los diseñadores han utilizado técnicas tales como la utilización de tablas e imágenes transparentes para ajustarlas, utilización de etiquetas que no son estándares del HTML y otras muchas. Estas "trampas" han causado a menudo problemas en las páginas a la hora de su visualización en distintas plataformas. Además, los diseñadores se han visto frustrados por la dificultad que tenían (aun utilizando estos trucos) para maquetar las páginas, ya que muchos de ellos venían maquetando páginas sobre el papel, donde el control sobre la forma del documento es absoluto.

Otra solución propuesta consiste en hacer que las páginas web tengan mezclado en su código HTML el contenido del documento con las etiquetas necesarias para darle forma. Esto tiene sus inconvenientes ya que la lectura del código HTML se hace pesada y difícil a la hora de buscar errores o depurar las páginas.

Características

Las hojas de estilo en cascada (Cascading Style Sheets - CSS) es un lenguaje estructurado usado para definir la presentación de un documento escrito en HTML o XML. El organismo encargado de desarrollar la especificación de las hojas de estilos, que servirán como estándar para los navegadores, es el W3C (Wold Wide Web Consortium).

El modo de funcionamiento de las CSS consiste en definir, mediante una sintaxis especial, la forma de presentación que le aplicaremos a:

- Una web entera, de modo que se puede definir la forma de todo la web de una sola vez.
- Un documento HTML o página, se puede definir la forma, en un pequeño trozo de código en la cabecera, de toda la página.
- Una porción del documento, aplicando estilos visibles en un trozo de la página.
- Una etiqueta en concreto, llegando incluso a poder definir varios estilos diferentes para una sola etiqueta.

Además, la sintaxis CSS permite aplicar al documento formato de modo mucho más exacto. Si antes el HTML se nos quedaba corto para maquetar las páginas y teníamos que utilizar trucos para conseguir nuestros efectos, ahora tenemos muchas más herramientas que nos permiten definir esta forma:

- Podemos definir la distancia entre líneas del documento.
- Se puede aplicar indexado a las primeras líneas del párrafo.
- Podemos colocar elementos en la página con mayor precisión, y sin lugar a errores.
- Y mucho más, como definir la visibilidad de los elementos, márgenes, subrayados, tachados...

En esta línea, mientras que con HTML tan sólo podíamos definir atributos en las páginas con pixeles y porcentajes, con el uso de CSS podemos definir muchas características utilizando unidades como:

- Pixels (px) y porcentaje (%), como antes.
- Pulgadas (in)
- Puntos (pt)
- Centímetros (cm)

El lenguaje *CSS* se basa en una serie de reglas que rigen el estilo de los elementos en los documentos estructurados, y que forman la sintaxis de las hojas de estilo.

Cada regla consiste en un selector y una declaración. Esta última va entre corchetes y consiste en una propiedad o atributo, y un valor separados por dos puntos.

Ejemplo:

Regla: `h2{color: green;}`

- **h2** ---> es el selector
- **{ color : green; }** ---> es la declaración
- **color** ---> es la propiedad o atributo
- **green** ---> es el valor

➢ Selector

El *Selector* especifica que elementos HTML van a estar afectados por esa declaración, de manera que hace de enlace entre la estructura del documento y la regla estilística en la hoja de estilo.

- **Declaración**

La *Declaración* que va entre corchetes es la información de estilo que indica cómo se va a ver el selector. En caso de que haya más de una declaración se usa punto y coma para separarlas.

- **Propiedad o Atributo y Valor**

Dentro de la declaración, la *Propiedad* o *Atributo* define la interpretación del elemento asignándosele un cierto *Valor*, que puede ser color, alineación, tipo de fuente, tamaño..., es decir, especifican qué aspecto del selector se va a cambiar.

La información CSS se puede proporcionar por varias fuentes, ya sea adjunto como un documento por separado o incorporado en el documento HTML, y dentro de estas posibilidades destacan tres formas de dar estilo a un documento web:

- **Hoja de Estilo Externa**

La *Hoja de Estilo Externa* se almacena en un archivo diferente al del archivo con el código HTML al que está vinculado a través del elemento *link*, que debe ir situado en la sección *head*. Es la manera de programar más eficiente, ya que separa completamente las reglas de formato para la página HTML de la estructura básica de la página.

- **Hoja de Estilo Interna**

La *Hoja de Estilo Interna* está incorporada a un documento HTML, a través del elemento *style* dentro de la sección *head*, consiguiendo de esta manera separar la información del estilo del código HTML.

- **Estilo en Línea**

El *Estilo en Línea* sirve para insertar el lenguaje de estilo directamente dentro de la sección *body* con el elemento *style*. Sin embargo, este tipo de estilo no se recomienda pues se debe intentar siempre separar el contenido de la presentación.

Ventajas del uso de CSS

A modo de resumen, podemos indicar las siguientes ventajas:

- El estilo se puede guardar completamente por separado del contenido siendo posible, por ejemplo, almacenar todos los estilos de presentación para una web de 10.000 páginas en un sólo archivo de CSS.

- CSS permite un mejor control en la presentación de un sitio web que los elementos de HTML, agilizando su actualización.

- Aumento de la accesibilidad de los usuarios gracias a que pueden especificar su propia hoja de estilo, permitiéndoles modificar el formato de un sitio web según sus necesidades, de manera que por ejemplo, personas con deficiencias visuales puedan configurar su propia hoja de estilo para aumentar el tamaño del texto.

- El ahorro global en el ancho de banda es notable, ya que la hoja de estilo se almacena en cache después de la primera solicitud y se puede volver a usar para cada página del sitio, no se tiene que descargar con cada página web. Por otro lado, quitando todo lenguaje de marcado en la presentación en favor del uso de CSS reduce su tamaño y ancho de banda hasta más del 50%, esto beneficia al dueño del sitio web con menos ancho de banda y costes de almacenamiento, as como a los visitantes para los cuales las páginas se van a cargar más rápido.

- Una página puede tener diferentes hojas de estilo para mostrarse en diferentes dispositivos, como pueden ser impresoras, lectores de voz, o móviles.

4.5.3. Limitaciones de las tecnologías web

Gracias a frameworks como los aquí mencionados y a nuevas tecnologías y protocolos, las aplicaciones web se encuentran muy cercanas a las aplicaciones de escritorio.

Sin embargo todavía existen ciertas limitaciones impuestas por la propia arquitectura cliente-servidor en la que estas aplicaciones se basan. El caso más claro es el de las llamadas notificaciones push o eventos disparados desde el servidor.

La arquitectura cliente-servidor impone que sea el cliente el que realice una petición y que sea el servidor el que la conteste. Esto es debido a que en esta arquitectura, un cliente es aquel que hace peticiones y nunca las espera.

Sin embargo no es raro que en una aplicación se produzca un cambio de estado sin que intervenga el cliente. Por ejemplo, en una aplicación de chat en la cual se comunican dos clientes (cliente A y cliente B) a través de un servidor, el funcionamiento más intuitivo sería que al recibir un mensaje del cliente A, el servidor enviase el mensaje al cliente B. Sin embargo en la práctica, hasta que el cliente B no comienza una nueva petición, el servidor no puede comunicarle el cambio. Las aplicaciones web que se enfrentan a estas situaciones suelen solventarlas de una de estas dos maneras:

-Mediante polling, es decir el cliente hace uso de un temporizador que, cada cierto tiempo, hace peticiones al servidor, que contesta con los cambios que se hayan producido. En general estas peticiones y sus respuestas son vacías, lo cual sobrecarga la red de manera innecesaria.

-Técnicas artificiales como posponer la respuesta del servidor hasta que o bien pase un tiempo o bien se tenga información, lo cual simula que el cliente se encuentra esperando un mensaje.

Debido al auge de las aplicaciones web estas situaciones son cada vez más frecuentes, por lo que se están desarrollando tecnologías para implementar las notificaciones push, como por ejemplo WebSockets.

5. Mejoras y futuros desarrollos

El estado actual de la aplicación es estable, con un nivel de usabilidad y experiencia de usuario bastante buena para no ser una aplicación comercial. No obstante, existen algunos aspectos dónde nos gustaría haber profundizado de haber podido disponer de más tiempo de desarrollo.

Algunas de las mejoras que más se podrían echar en falta para herramientas de edición gráficas o "WYSIWYG" (*What You See Is What You Get*) son las siguientes:

- Salvar estados parciales: Actualmente el usuario tan sólo puede generar el estado final de salida. Esto no permite guardar estados parciales ni cargar otros trabajos realizados previamente. Una mejorar inmediata sería cubrir este aspecto permitiendo guardar y cargar ficheros de estados intermedios.

- Permitir la multiselección de widgets: Si se permitiese seleccionar varios widgets en el área de edición, se podría modificar su posición en conjunto o cambiar propiedades a todos ellos a la vez, en vez de la funcionalidad soportada actualmente de copiar estilos.

- Incluir más widgets: Actualmente soportamos algunos de los widgets más básicos. Una buena idea sería soportar otros nuevos como el uso de imágenes, calendarios, barras de menú, etc.

- Ampliar el rango de propiedades predefinidas: El espectro de opciones que proporcionan las hojas de estilo CSS es sumamente amplio, por ello sería bueno poder soportar algunas más de ellas en la sección de propiedades.

En lo referente a futuros desarrollos, nos vienen a la mente dos ideas. Una de ellas sería menos invasiva de cara a la modificación del entorno de nuestra aplicación. La otra, supondría un enfoque diferente y conllevaría probablemente, la realización de un nuevo proyecto reutilizando partes de lo que ahora se tiene. Procederemos primero a explicar la primera idea para después introducir la opción más intrusiva.

❖ Editor CSS

Debido a la importancia que tienen las hojas de estilo en el contexto de nuestra aplicación y el uso intensivo que hacemos de ellas, parece interesante y nada descabellado introducir una nueva funcionalidad que permita editar propiedades CSS de manera avanzada aparte de las propiedades predefinidas que ya soportamos.

La idea sería introducir algún tipo de editor que permita crear estilos propios definidos por el usuario. Esto supondría la creación de algún tipo de intérprete textual para que un usuario con conocimientos en CSS, pueda realizar de manera similar a la de un programador, la edición de las hojas de estilo. Por otra parte, se permitiría también una paleta de opciones y estilos predefinidos que sirvan de guía para los usuarios con menos conocimientos en este campo.

De esta manera, se podría cubrir por completo las necesidades de edición de propiedades, dotando nuestra aplicación de una flexibilidad y potencial tan amplio como el propio permitido por el estándar de las hojas de estilo CSS. Además de esto, supondría una gran ayuda de cara al usuario que podría reutilizar hojas de estilos creadas en otros proyectos importándolas y posteriormente, utilizar nuestro editor para mejorarlas lo máximo posible.

❖ Generar páginas web como output final

Actualmente nuestra aplicación es totalmente dependiente de los modelos jasfag. A modo de recordatorio mostramos de nuevo el esquema general de nuestra aplicación.

Imagen 55: Esquema conceptual de la aplicación

Lo que se pretendería en un posible futuro desarrollo es seguir manteniendo esta posibilidad de editar a partir de modelos, pero además de esto, que sea posible editar aplicaciones web desde cero. La idea sería realizar un editor de interfaces gráficas para aplicaciones web, totalmente independiente.

La funcionalidad básica del manejo de widgets ya está implementada y gran parte de la modificación visual sobre estos.

Habría que profundizar más en los widgets y elementos gráficos soportados, en la distribución entre las diferentes ventanas, pensar sobre cómo modificar la apariencia permitiendo posteriormente realizar la lógica pero sobretodo, habría que profundizar bastante en cómo sería el formato final del output.

Una posible idea sería generar la salida en forma de clases Vaadin que al fin y al cabo sería como generar documentas HTML más JavaScript, acompañados de hojas de estilos CSS para modificar la apariencia gráfica de los widgets. El nuevo esquema de aplicación podría ser algo similar a lo que mostramos a continuación.

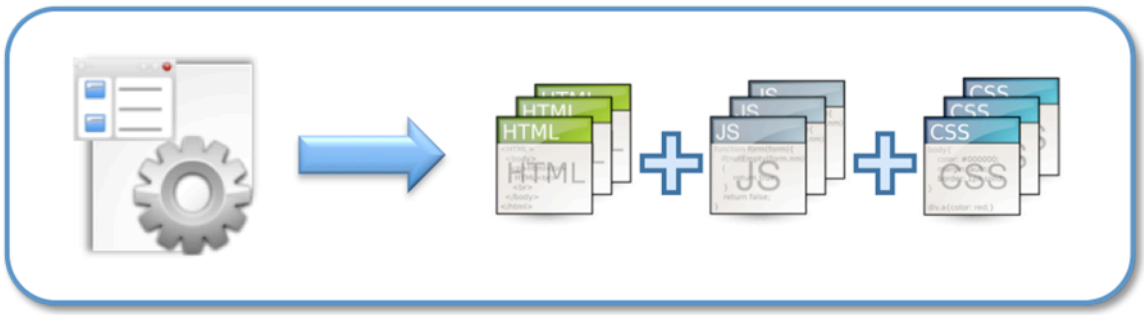


Imagen 56: Esquema conceptual de la nueva aplicación

Anexo I

1. Subida de ficheros

La subida y descarga de archivos se podrían identificar como la parte inicial y final respectivamente, dentro del flujo de usuario de nuestra aplicación. Para poder utilizar nuestra aplicación, es necesario subir un modelo jasfag, previamente serializado en un fichero XML. Como recordatorio del flujo de la aplicación mostramos un pequeño esquema conceptual al respecto, resaltando el input y el output que se ven involucrados.

Imagen 57: Esquema conceptual de la aplicación

Para realizar esto es necesario disponer de algún elemento que nos permita subir ficheros desde el cliente hasta el servidor. Vaadin nos proporciona un componente llamado "Upload" que se encarga de esto. Vaadin separa la parte gráfica de los botones de aceptar, cancelar, etc y el selector de ficheros, de la parte lógica que se encarga de manejar el flujo de datos entre cliente y servidor. Esta parte se ha de codificar implementando la interfaz Receiver, que proporciona métodos para saber cuando comienza la descarga, cuando acaba, si se produce algún error, etc. Por último, el resultado de la descarga dará lugar a un stream de datos o si se desea, al propio fichero subido por el usuario.

Como optimización, cara a posibles errores de usuario, se comprueba en el lado del cliente que la extensión del archivo sea correcta, para evitar tráfico innecesario y el tiempo que esto conlleva. Las posteriores comprobaciones sobre la corrección del formato del archivo, se han de realizar en el lado del servidor una vez ha finalizado el proceso de subida.

A continuación mostramos un ejemplo del aspecto que tiene el selector de ficheros de nuestra aplicación, cuando se pretende subir algún modelo jasfag para comenzar el proceso de edición de los widgets.

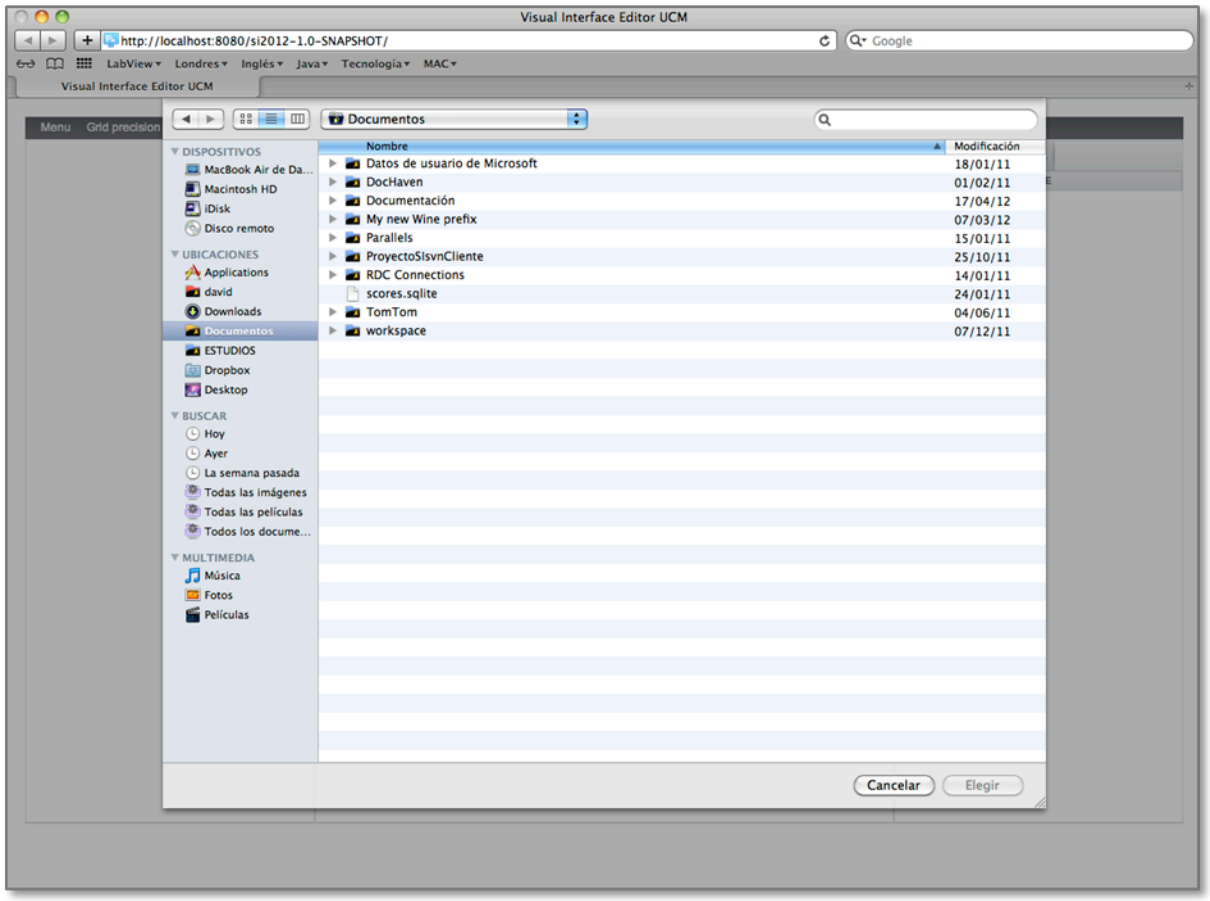


Imagen 58: Selección de fichero a subir

El siguiente ejemplo, muestra la barra de progreso que nos ofrece Vaadin, cuándo se comienza el proceso de subida de ficheros al servidor.

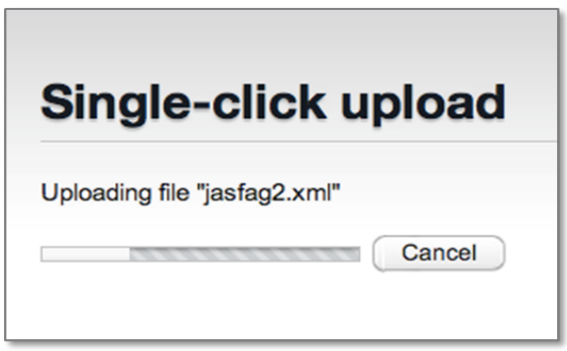


Imagen 59: Ejemplo de barra de progreso

Anexo II

1. Button

1.1. Presentación del widget

Los botones son unos de los elementos más importantes tanto en aplicaciones de desktop como en aplicaciones web. Nos centraremos en sus propiedades visuales y no en su lógica (control de eventos).

Con respecto al aspecto de los botones, en la imagen que se muestra a continuación podemos ver la diferencia de apariencia entre los botones de Vaadin a la izquierda y los utilizados en nuestra aplicación, los botones nativos a la derecha.

Imagen 60: Ejemplos de botones Vaadin

1.2. Descripción técnica y problemas encontrados

En un principio se optó por utilizar la representación gráfica del botón de Vaadin. Esto suponía problemas a la hora de manipular el tamaño del wiget, que distorsionaba su apariencia. Debido a esto, se optó por utilizar otro componente de Vaadin, el botón nativo. Sobre este componente no se producen tales problemas a la hora de editar sus propiedades de dimensión, la única diferencia es que su apariencia visual difiere ligeramente en función del sistema operativo que se esté utilizando.

El gran número de estilos con que cuenta Vaadin y su complejidad, puede suponer una ventaja a la hora de realizar aplicaciones web convencionales puesto que enriquece mucho la apariencia visual, con las ventajas que esto supone de cara al usuario. En nuestro caso particular supuso un problema en varias ocasiones, especialmente con los botones.

Más concretamente con las propiedades relativas al caption de estos. Debido al anidamiento de reglas que permiten las hojas de estilos CSS, se producía una "colisión" de estas con la forma inicial de generación de reglas que se optó en un principio. Esto obligo a modificar los modelos de reglas para poder añadir más niveles de anidamiento para poder sobrescribir las reglas de estilo relativas al caption de los botones.

A continuación mostramos un diagrama para explicar de manera gráfica las relaciones entre algunos componentes involucrados en las situaciones anteriormente comentadas.

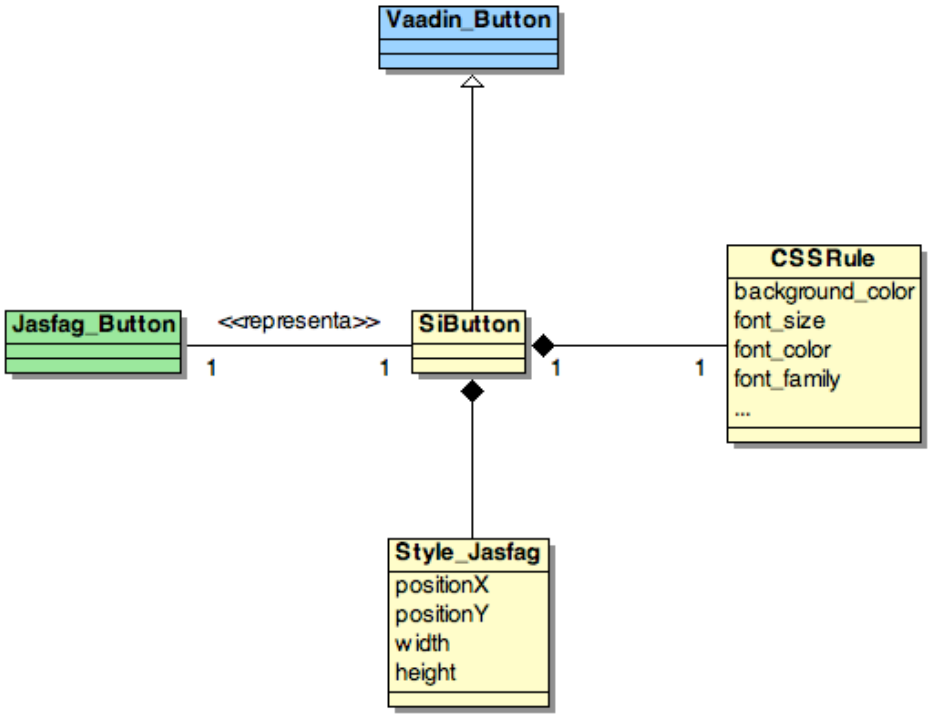


Diagrama 5: Relaciones de SiButton

En azul aparece la clase de Vaadin de la cual extiende nuestro botón. En verde el botón del modelo de jasfag leído, del cual se obtiene la información relativa a la posición y tamaño del widget. Está información será modificada por el usuario y transformada por la aplicación a propiedades CSS en los ficheros de salida (inicialmente se modificaba el modelo original, opción que se descartó posteriormente para hacer más independiente la aplicación).

Las clases en amarillo son con las que trabaja la aplicación. Por una parte el botón con el que se representa al componente gráfico y por otra, la clase CSSRule. Está ultima es la encargada de recolectar las propiedades que configura el usuario sobre el widget y su transformación en formato CSS para poder modificar la apariencia de los componentes gráficos dinámicamente y también, en el formato de salida en forma de hojas de estilos que puedan ser leídas por las personas.

2. Label

2.1. Presentación del widget

Los labels o etiquetas se utilizan en Vaadin principalmente para mostrar información textual al usuario. Es un elemento muy útil y versátil ya que admite de forma sencilla multitud de propiedades CSS, pudiendo utilizar un label y darle la apariencia de un botón, checkbox y otros muchos componentes.

Pare ejemplificar lo versátiles que pueden resultar los labels, mostraremos algunos de los formatos de apariencia que pueden tomar. Algunos de estos incluyen imágenes y en el caso del loading, es una imagen animada aunque no se aprecie en la captura.

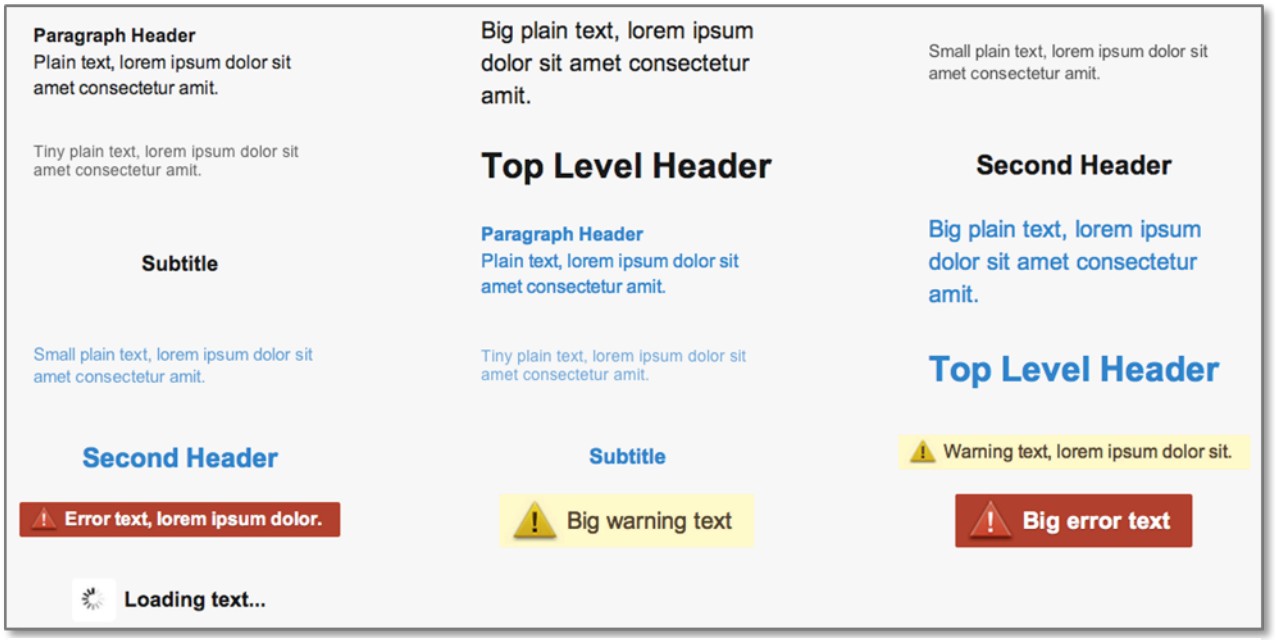


Imagen 61: Ejemplo de Labels de Vaadin

2.2. Descripción técnica y problemas encontrados

Una característica importante de los elementos Label, es que admiten textos en varios formatos, como por ejemplo HTML, lo que permite darle una apariencia diferente sin necesidad de utilizar hojas de estilo. El texto que se quiere representar se escribe directamente sobre el elemento con los tags HMTL que se desee, por ejemplo de la siguiente manera: `Label label = new Label("<b>This a label with bold style</b>");`

En cuanto a la relación con otras clases implicadas en la estructura de los labels, es prácticamente idéntica a la existente con los botones. No obstante en este caso, no hubo problemas con los niveles de anidamiento de los CSS. Como ya se mencionó es un de los elementos más flexibles y versátiles desde el punto de vista de la modificación de propiedades gráficas, lo que en cierta forma facilitó las pruebas iniciales para la realización del modelo de datos de las propiedades y cómo generar las reglas de las hojas de estilos.

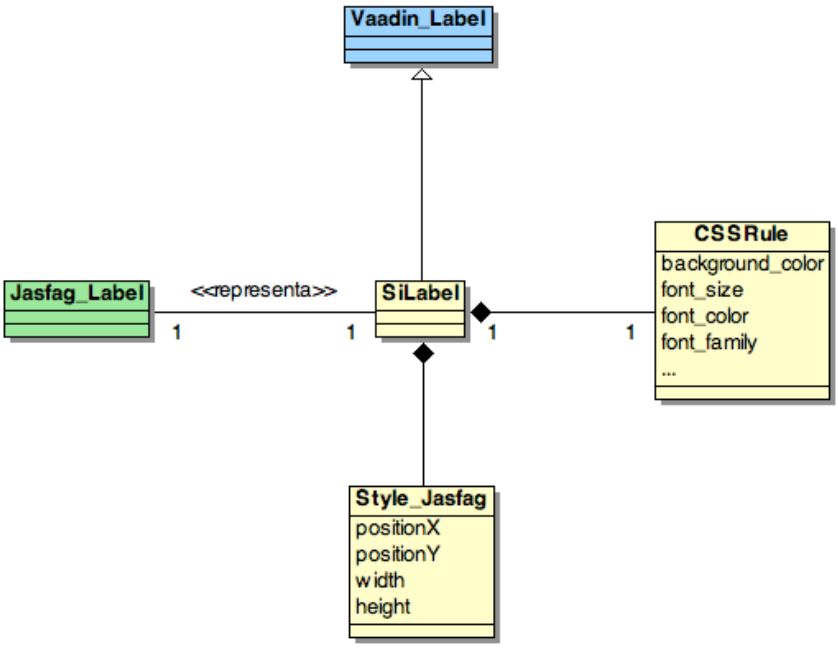


Diagrama 6: Relaciones de SiLabel

La relación de colores para explicar el modelo son iguales que en las del botón. La entidad de Vaadin de la cual se hereda aparece en azul, en verde el elemento de jasfag que tiene equivalente directa con el Label de nuestra aplicación. En amarillo aparecen las clases que utilizamos para la edición y gestión de nuestros propios widgets.

3. Link

3.1. Presentación del widget

No hace mucha falta explicar lo que es un link y su función, por tanto nos centraremos más en las peculiaridades que nos encontramos a la hora de abordar la representación de este componente.

A continuación queremos mostrar un pequeño ejemplo de la apariencia básica que tienen los links en Vaadin.

Imagen 62: Ejemplos de Links de Vaadin

Se puede observar un link normal, y otros dos. Uno de ellos con texto resaltado de otro color al situar el cursor encima e icono. Además de esto aparece un tag o tooltip con el link y un comentario. Por último, se muestra un link con icono y sin texto.

3.2. Descripción técnica y problemas encontrados

En lo relativo a las propiedades de las hojas de estilo CSS no hay nada nuevo que comentar ; es un comportamiento idéntico al explicado anteriormente con los labels. La peculiaridad reside a la hora de identificar un link dentro del modelo jasfag.

En los elementos gráficos de los modelos jasfag no existe una entidad como tal que sea un link, tan sólo existen labels. Lo que diferencia un label de un link es que este último lleva asociado un evento, más concretamente un evento “onCreate” asociado a la creación del elemento. De esta manera es como podemos diferenciar si estamos ante un widget del tipo label o por el contrario, ante un link. En función de esto crearemos un componente gráfico diferente al leer el fichero de entrada.

Para ejemplificar esto, mostramos un diagrama que aclara de forma gráfica la peculiaridad citada más arriba: a saber, la concerniente a la existencia de dicho evento en los componentes de tipo label de los modelos jasfag.

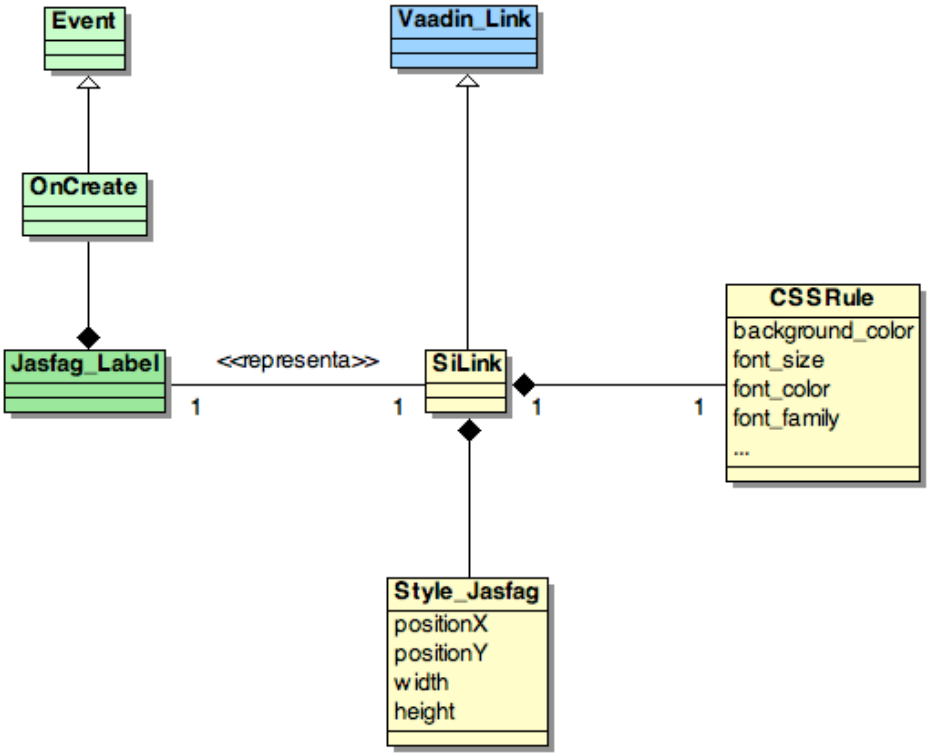


Diagrama 7: Relaciones de SiLink

De nuevo, la leyenda de colores se mantiene igual. En este caso, para representar los eventos que diferencian a los links de los labels se ha utilizado un color verde de una tonalidad más clara. Por lo demás, no hay nada especialmente digno de reseña diferente a lo citado para el widget de tipo Label.

4. CheckBox

4.1. Presentación del widget

Los widgets tipo checkbox junto con los radiobutton, son comúnmente utilizados en aplicaciones de escritorio, pero más aún en aplicaciones web formando parte de formularios. Es un elemento que permite seleccionar mediante un "tick" el valor binario de un determinado campo, que suele ir acompañado de una descripción textual.

Lo mejor que se puede hacer para ejemplificar el componente, es añadir algunos ejemplos como los de la imagen a continuación.

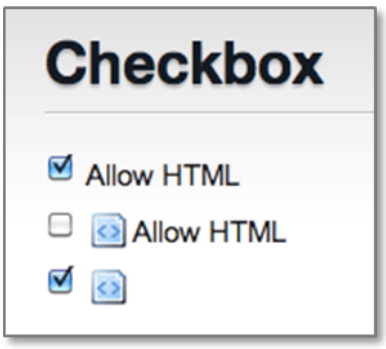


Imagen 63: Ejemplos de CheckBox de Vaadin

4.2. Descripción técnica y problemas encontrados

Como puede verse, el checkbox es una mezcla entre un botón y un label. De cara a las propiedades CSS no presentó ninguna incompatibilidad al nivel de jerarquías de reglas. Lo único digno de destacar es que se tuvo que desactivar los eventos sobre dicho componente ya que cada vez que el usuario lo seleccionaba, situando el foco sobre él, se marcaba y desmarcaba su indicador de selección (el tick). En lo referente a la relación del diagrama de clases, al igual que el resto de componentes, su estructura permanece homogénea. Se adjunta un esquema gráfico para mayor aclaración.

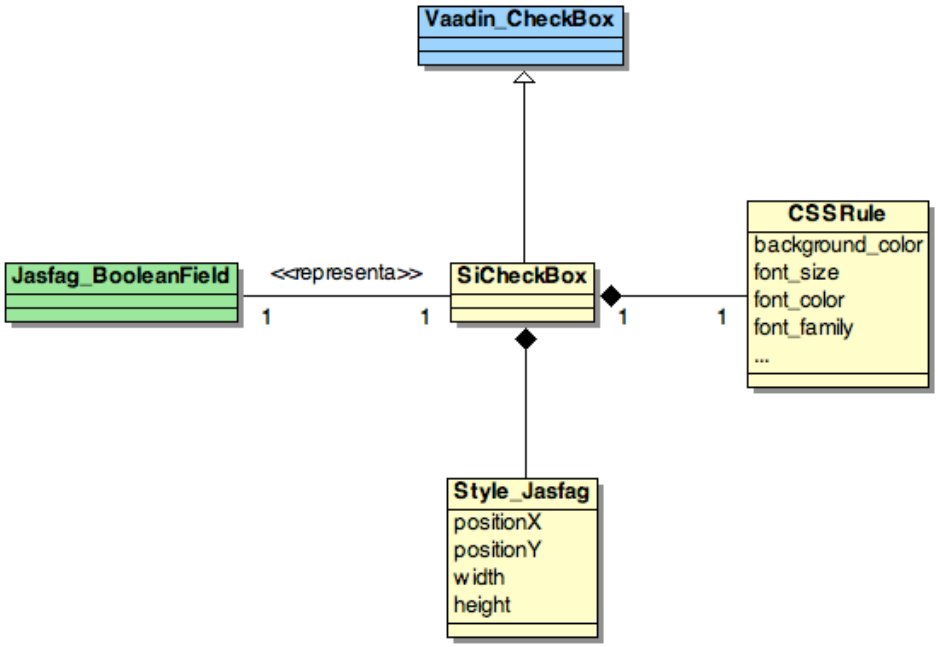


Diagrama 8: Relaciones de SiCheckBox

5. Textfield

5.1. Presentación del widget

Los elementos gráficos tipo Textfield son campos en los que está habilitada la entrada de texto, aunque también son utilizados en menor medida como output hacia el usuario. Sirven para que el usuario rellene valores en formularios, realice búsquedas y cualquier otro tipo de acción asociada a elementos textuales. Vaadin proporciona unos estilos por defecto que nuevamente, mediante la modificación de las propiedades de las hojas de estilo CSS, se pueden conseguir. Para ilustrar algunas de las posibilidades que presenta este widget en Vaadin, mostramos una imagen con algunos ejemplos.

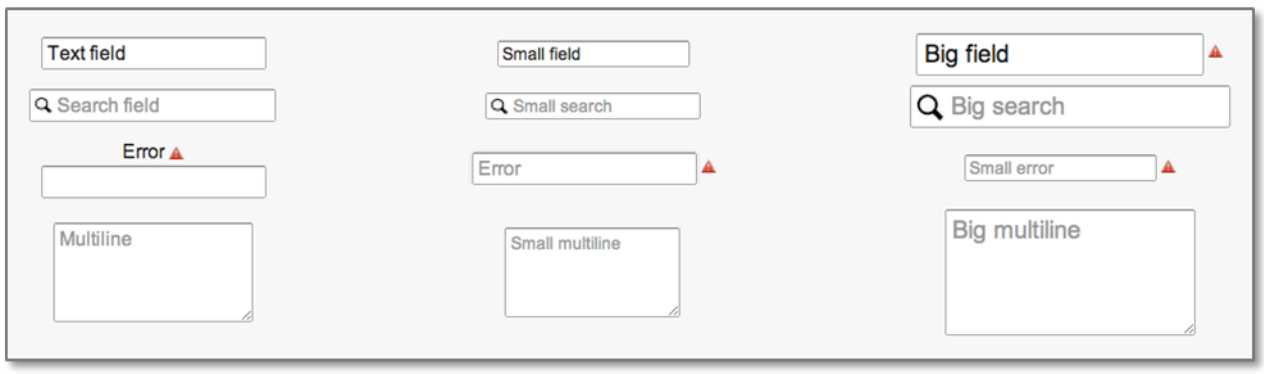


Imagen 64: Ejemplos de TextField de Vaadin

5.2. Descripción técnica y problemas encontrados

Una de los problemas que presentan los textfield de Vaadin es que al modificar su tamaño dinámicamente mediante la funcionalidad de usuario de nuestra aplicación, la imagen de la hoja de estilos que se define se veía deformada. Para solucionar esto, tuvimos que crear nuestro propio CSS para nuestro componente SiTextfield, el cuál no utiliza ningún tipo de imagen, tan sólo colores RGB. Un ejemplo de la apariencia final de este widget en nuestra aplicación es la que mostramos a continuación.

Imagen 65: Textfield de SiComponent

De nuevo y al igual que con el CheckBox, se tuvo que desactivar la posibilidad de selección, en este caso referente a la entrada de texto.

En lo referente al esquema de clases, nuevamente el diseño permanece homogéneo, todas las modificaciones necesarias para la adaptación y correcto funcionamiento de los componentes Vaadin, se realizan en nuestra clase propia. Nuevamente y para ejemplificar gráficamente lo mejor posible mostramos un diagrama de clases.

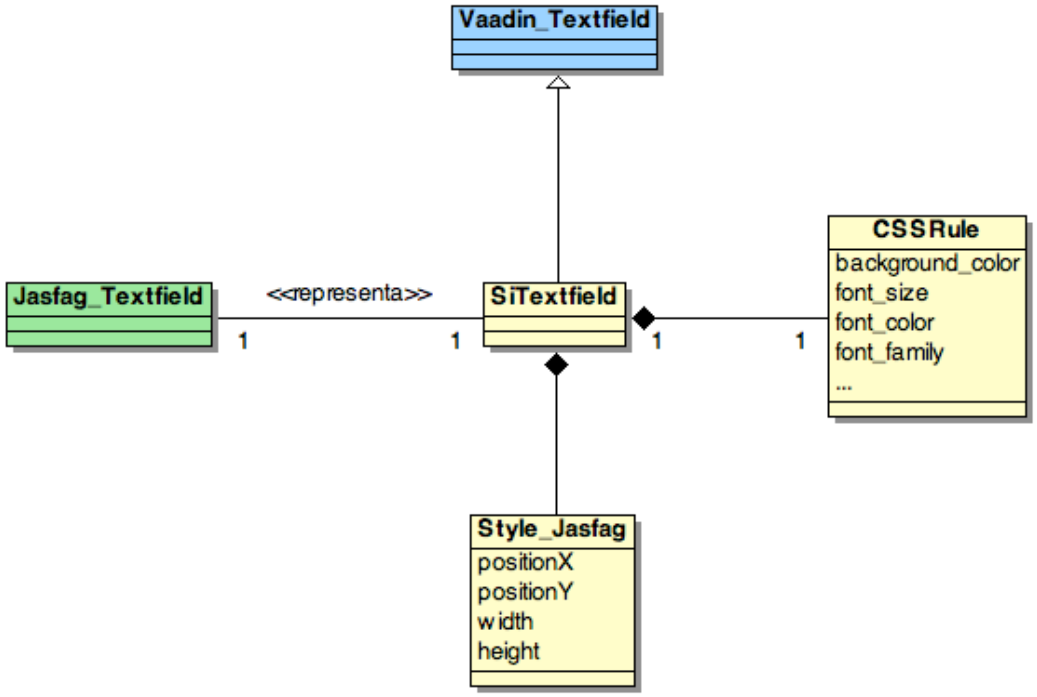


Diagrama 9: Relaciones de SiTextField

De nuevo, las clases en amarillo se corresponden con las utilizadas en nuestra aplicación, redefiniendo funcionalidades existentes en los widgets de Vaadin para adaptarlos a nuestra aplicación y extendiendo otros, como con el caso de los CSS.

6. ComboBox

6.1. Presentación del widget

El ComboBox es un elemento de selección dónde se muestra al usuario una serie de opciones para que elija entre ellas. También se puede habilitar que el usuario pueda introducir nuevas opciones a la selección o que por ejemplo, se puedan escoger varios valores habilitando la multiselección.

Nuevamente, no ahondamos demasiado en la funcionalidad de cara al uso de los widgets sino en su apariencia visual, modificación de algunas propiedades y el tratamiento de eventos para adaptarlos a nuestras necesidades. Antes de comentar las peculiaridades del ComboBox y las soluciones tomadas, mostraremos algunos ejemplos de la apariencia que puede tener dicho componente gráfico.

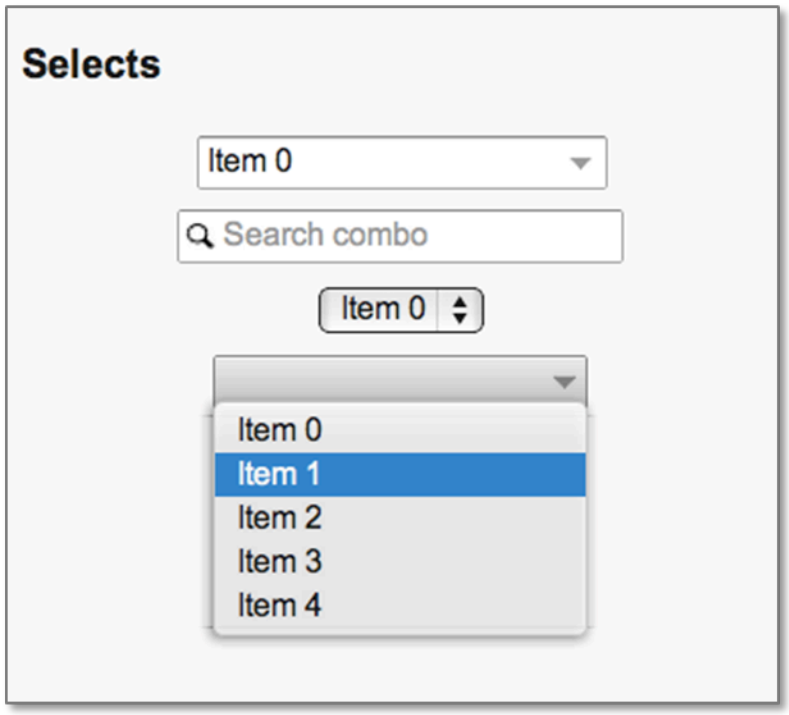


Imagen 66: Ejemplos de ComboBox de Vaadin

6.2. Descripción técnica y problemas encontrados

Debido a algunos de los problemas que daban los componentes que Vaadin posee, optamos por utilizar un componente nativo de ComboBox, que tiene una apariencia como la que mostramos a continuación.

Imagen 67: ComboBox de SiComponent

De cara a utilizar el combobox que Vaadin trae por defecto tuvimos problemas similares a los del textfield, es decir, que a la hora de dimensionar su tamaño en el canvas de edición mediante una acción del usuario su apariencia visual se veía distorsionada. Por tal motivo optamos por usar el componente nativo no obstante, se nos presentaba otro problema común en ambos casos.

A la hora de capturar los eventos de selección sobre el widget (tanto para arrastrar como para cambiar sus propiedades) ocurría que el propio ComboBox capturaba dichos eventos. Esto es lo más lógico para un uso normal de un combo, pero en nuestro caso teníamos que ampliar bastante la funcionalidad sobre el mismo. En Vaadin los eventos suelen ser capturados por los paneles que contienen a los widgets, buscando estos sobre todos sus elementos para después, pasar a realizar las acciones implementadas en los listeners.

Para solucionar tal problema se tuvo que desactivar la posibilidad sobre el widget de capturar eventos y así de esta forma, permitir al panel que lo contiene poder realizar las acciones pertinentes.

Los cambios anteriormente mencionados resultaron algo complicados de detectar y de proponer una solución. No obstante debido al buen diseño arquitectónico escogido en un principio, se pudo solventar sin modificar la jerarquía de clases manteniendo la homogeneidad citada en puntos anteriores. Nuevamente para ejemplificar esto gráficamente aportamos un diagrama de clases con la misma idea.

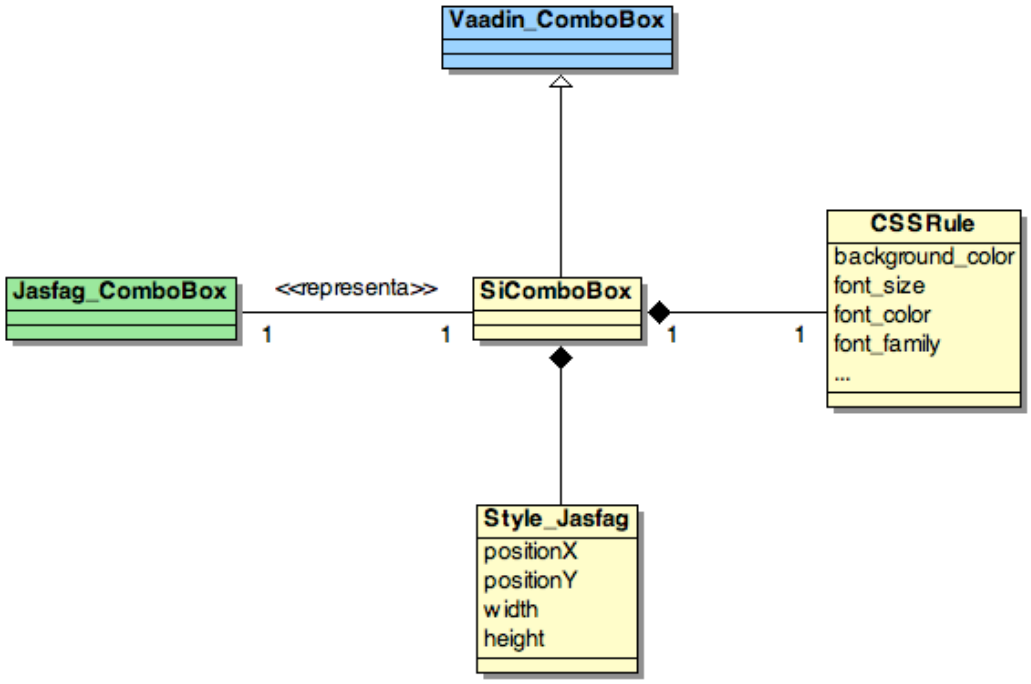


Diagrama 10: Relaciones de SiComboBox

La leyenda de colores es la misma que siempre. Las clases amarillas son las empleadas por nuestra aplicación. La de color verde se corresponde con el widget jasfag y la clase de color azul, con el elemento gráfico de Vaadin del cuál extendemos su funcionalidad para satisfacer las necesidades de la aplicación.

7. Table

7.1. Presentación del widget

El último de los elementos a comentar dentro de nuestros widgets, son las tablas. Una tabla es un elemento matricial con elementos característicos de fila, columna, celda y encabezado. Son unos componentes gráficos que pueden hacer cosas simples como mostrar texto y otras más complejas como ordenar elementos, incorporar imágenes, vídeos, otros widgets y un gran número de acciones y eventos complejos.

Antes de describir nuestro widget y comentar las dificultades encontradas, mostraremos algunos ejemplos de los estilos visuales que pueden tomar las tablas en Vaadin, los cuales pueden variar enormemente.

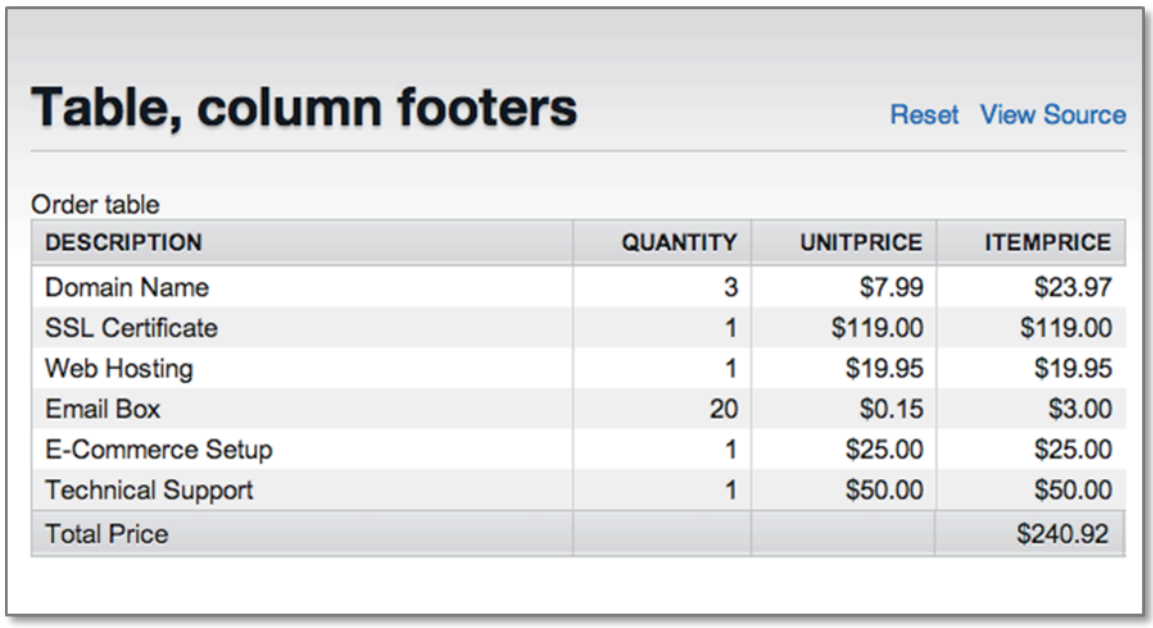


Imagen 68: Ejemplo de tabla de Vaadin sin imágenes

7.2. Descripción técnica y problemas encontrados

En nuestro caso particular, nuestro objetivo no es manipular elementos de fila, columna o celda. Por ser la tabla un elemento muy complejo y la posible dificultad de adaptar nuestro output, para posteriormente ser utilizado en las páginas webs de los modelo jasfag, decidimos centrarnos en los elementos visuales referentes al conjunto de la tabla, tratándola como una única entidad. Es decir, como una caja negra, sin importar lo que contenga dentro.

Este enfoque es adecuado para poder posicionar la tabla dentro del canvas y además, poder darle propiedades presentacionales al igual que a los otros widgets.

Con las tablas de Vaadin aparecieron de nuevo problemas en relación con los estilos CSS. Esta vez no pudimos optar por un elemento nativo, ya que no existen. La opción adoptada fue utilizar un textfield como elemento base y dibujar encima de él, mediante propiedades de las hojas de estilo, una cabecera para simular la apariencia de una tabla.

Esta estrategia es comúnmente utilizada con ciertos widgets cuando dan problemas. La solución más típicamente aceptada suele ser utilizar un elemento más básico y flexible, como los labels, y añadir reglas CSS y eventos. De esta forma se puede crear un widget totalmente diferente, siendo desde el punto de vista del usuario indistinguibles.

El resultado final resultante en nuestra aplicación, de cara a la presentación de las tablas tiene el siguiente aspecto.

Imagen 69: Ejemplo de tabla de SiComponent

Como puede observarse en la imagen, no se muestran ni las filas ni las columnas de la tabla. La idea es mostrar el tamaño que tendría la tabla que alberga los datos y modificar sus propiedades.

El siguiente problema que surge con las tablas no está relacionado con las propiedades de las hojas de estilo, sino con la gestión de los eventos. Las tablas en Vaadin tienen cierta cualidad para manejar eventos de tipo "Drag&Drop" para el intercambio el orden de las columnas. Esto suponía un problema a la hora de desplazar la tabla arrastrándola con el ratón. Si el punto destino dónde se quería desplazar la tabla se encontraba fuera de la misma no había problema. Sin embargo, cuando el punto de destino o posición donde se realiza la acción de drop está dentro de la propia tabla, esta recibía el evento y no se realizaba el desplazamiento.

Esta situación puede recordar a lo que ocurría con el ComboBox, aunque en el caso de este componente el problema no era realizar la acción de Drag&Drop, sino realizar cualquier tipo de click para poder seleccionar el elemento. Por tanto, el problema era diferente y la solución también. En este caso se tuvo que conseguir pasar los eventos de drop al panel que contiene la tabla para poder realizar los desplazamientos.

Nuevamente, estas modificaciones se pudieron hacer en la clase SiTable, sin necesidad de modificar el diseño inicial en la familia de nuestros componentes gráficos.

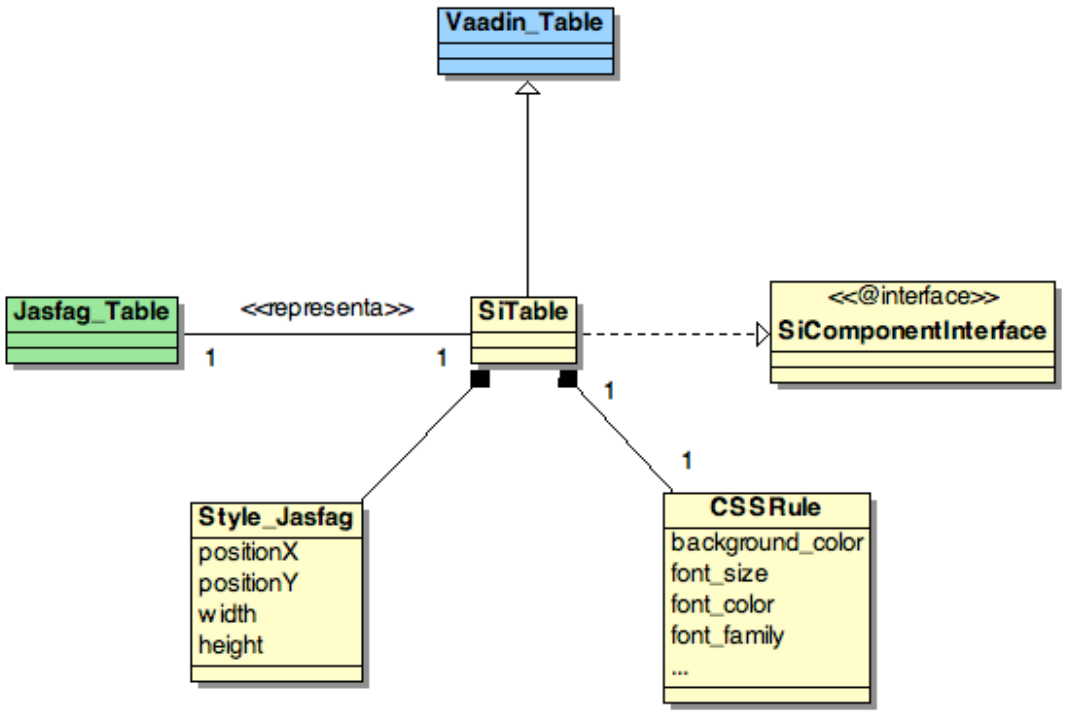


Diagrama 11: Relaciones de SiTable

En este nuevo diagrama se puede apreciar una entidad que no ha aparecido hasta ahora. Se trata de la interfaz que han de implementar todos los componentes o widgets de la familia *SiComponets*. En anteriores diagramas no se ha introducido este concepto por no ser excesivamente relevante y por no dificultar la información mostrada. En este caso optamos por introducir la idea de "*SiComponentInterface*". Dicha interfaz proporciona operaciones comunes a todos los widgets para poder aprovechar el polimorfismo y trabajar lo más cómodamente posible, desde el punto de vista del controlador.

De esta manera se evitan hacer castings innecesarios, utilizar el operador *instanceof* o cualquier otro tipo de argucia que nos permita Java, ya que normalmente, el uso de estos elementos suele significar una carencia de diseño e infrautilización de la orientación a objetos.

Bibliografía

[1] Book of Vaadin - https://vaadin.com/book

[2] Maven - http://maven.apache.org/

[3] W3C - http://www.w3.org/

[4] CSS Validator - http://jigsaw.w3.org/css-validator/

[5] CSS Manual - http://www.w3schools.com/css/default.asp

[6] Subversion - http://subversion.apache.org/

[7] GWT - https://developers.google.com/web-toolkit/

[8] CSS Inject - https://vaadin.com/directory#addon/cssinject

[9] Proyecto sistemas informáticos UCM, Junio 2011. "Diseño e implementación de un generador de código para modelos de interfaces gráficas en el lenguaje ActionGUI". Autor: Gonzalo Ortiz Jaureguizar. Director: Manuel García Clavel.

[10] David A. Basin, Manuel Clavel, Marina Egea, Miguel Angel García a de Dios, Carolina Dania, Gonzalo Ortiz, Javier Valdazo. "Model-Driven Development of Security-Aware GUIs for Data-Centric Applications". En "Foundations of Security Analysis and Design VI - FOSAD Tutorial Lectures", Lecture Notes in Computer Science, vol, 6858, pags. 101-124, 2011.

[11] Human Computer Interaction - http://hcibib.org/

[12] Scrum Manual - http://jeffsutherland.com/scrumhandbook.pdf

[13] Scrum Info - http://scruminfo.com/wp/