

Formulario en Arena

Autor: Carlos Lombardi

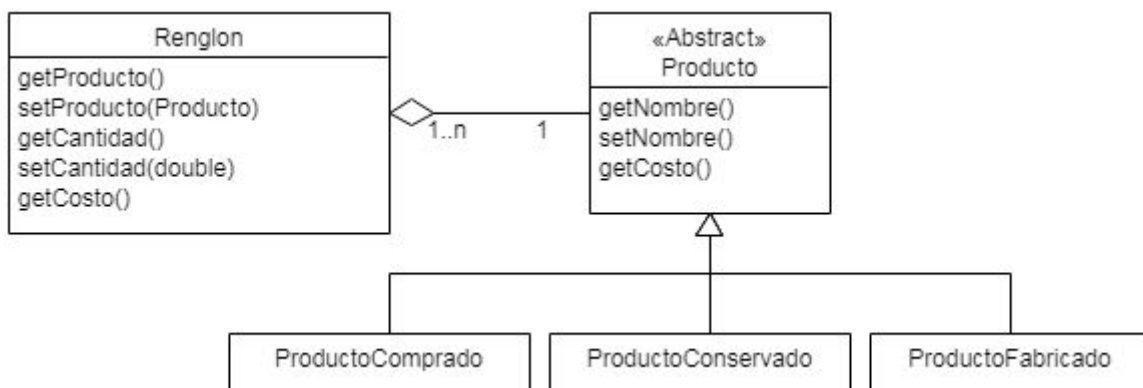
Fecha: 18/05/2017

El objetivo de este documento es contar una forma de armar, en Arena, una pantalla de edición de un renglón de pedido, que tenga el aspecto que se muestra al costado.

En la jerga, se suele usar la palabra **formulario** para referirse a una pantalla, o un panel, en el que se edita un objeto.

Entonces, lo que vamos a desarrollar es un formulario para editar un renglón.

Las clases de modelo involucradas se muestran en el diagrama de clases de abajo.



El modelo incluye un objeto store, que tiene registrados los productos que pueden formar parte de un producto. Este objeto es el Singleton de la clase PedidoAppStore, lo obtenemos así:

`PedidoAppStore.theStore()`.

Lo que nos va a interesar del store es que le puedo pedir la lista de productos ordenados por nombre, para eso le envío `getProductosPorNombre()`.

Este documento también trae información que puede ser útil para armar distintos tipos de pantalla en Arena, y quién te dice, en otras tecnologías también.

En particular se habla de

- mostrar información vinculada a una propiedad.
- cuáles propiedades se actualizan automáticamente en pantalla y cuáles no. Cómo forzar la actualización de las que no se actualizan solas.
- configuración de botones.
- combinación de paneles con distintos layouts para que los controles queden organizados de acuerdo a lo que se diseña.
- qué hacer si se quiere bindear un control a una propiedad de un objeto que no es el modelo de la ventana.

Creación de la clase

Cuando una ventana es un formulario, conviene que sea subclase de la clase `Dialog`¹. Esto nos va a ayudar a dar el comportamiento adecuado a los botones de aceptar y cancelar. Por ahora creemos la clase, más adelante veremos cómo manejar los botones.

Como todas las clases que sirven para representar ventanas en Arena, `Dialog` es genérica. En el parámetro de tipo se indica la clase del modelo de la ventana. En este caso vamos a tener

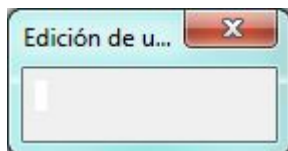
```
public class RenglonForm extends Dialog<Renglon> {
    public RenglonForm(WindowOwner owner, Renglon model) {
        super(owner, model);
    }

    @Override
    protected void createFormPanel(Panel mainPanel) {
        this.setTitle("Edición de un renglón");
    }
}
```

Este código inicial incluye:

- la definición de la clase,
- el constructor, que hay que redefinirlo obligatoriamente por cómo maneja Java los constructores,
- el método que completa el panel, para los `Dialog` se llama `createFormPanel` en lugar de `createContents`, como es para las `Window` y `MainWindow`. Por ahora lo único que hicimos fue setear el título de la ventana (fíjense que `this` es el `Dialog`, el panel es `mainPanel`).

La ventana que obtenemos hasta ahora es bastante modesta ...



... pero por algo se empieza.

¹ En rigor, lo mejor-mejor es `TransactionalDialog`. Pero eso lo vamos a ver bastante más adelante.

Selección del producto

En el formulario se cargan dos propiedades del renglón: el producto y la cantidad.

El producto es una instancia de (alguna subclase de) `Producto`, no es un “dato simple” (número, `String`, booleano). Hay que elegir alguno de los productos que están configurados, que son los que se obtienen pidiéndole `getProductosPorNombre()` a `PedidoAppStore.theStore()`.

En la jerga de las UI se usa la palabra **selección** para estos casos, en los que hay que (justamente) seleccionar una opción de una lista que viene dada. El formulario debe incluir un control para la selección del producto.

Arena incluye un tipo de control, llamado `List`², que es una lista de selección. Una `List` maneja una lista de objetos, vinculando el objeto que se seleccione en la lista con una propiedad de algún objeto.

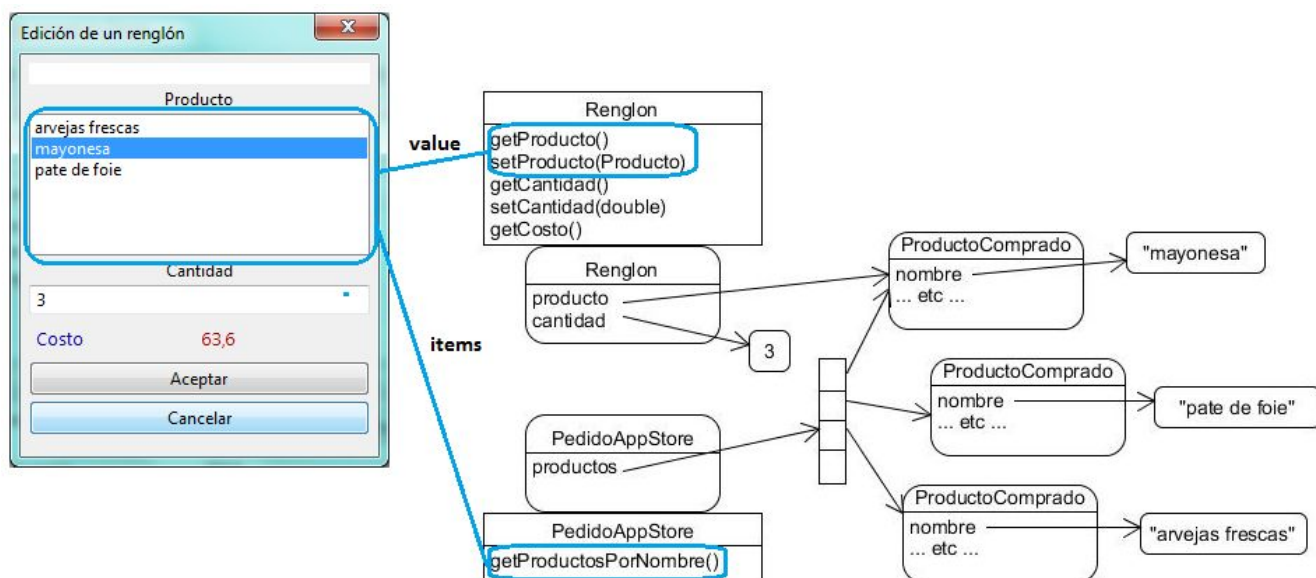
Atención: una `List` tiene **dos** vínculos:

- uno para obtener la lista de opciones, lo que se llaman los **items**. En Arena³, este vínculo es read-only, o sea, no hay posibilidad de modificar la lista desde el control.
- el otro para el objeto que se seleccione en la lista. Este es el **value** (en castellano valor) que funciona para muchos otros controles (`Label`, `TextBox`, `NumericField` entre ellos).

En nuestro ejemplo,

1. los **items** se vinculan con la propiedad `productosPorNombre` del objeto `PedidoAppStore.theStore()`, mientras que
2. el **value** hay que vincularlo con la propiedad `producto` del modelo de la ventana.

Este gráfico muestra los dos vínculos de nuestra lista de selección.



Observar que

1. la lista que muestra es la de los productos que maneja el store, ahí se ve el vínculo de items.
2. el producto seleccionado es el producto que tiene asignado el renglón, ahí se ve el vínculo de value.

² OJO que van a encontrar muchas clases de nombre `List`. La que hay que usar es `org.uqbar.arena.widgets.List`. Varios de (si no todos) los controles de Arena están en el package `org.uqbar.arena.widgets`, “widget” es un sinónimo (tal vez aproximado) de “control”.

³ Otras tecnologías incluyen controles de selección que permiten agregar opciones.

El código (dentro de `createFormPanel`) nos queda así:

```
List<Producto> selectorDeProducto = new List<Producto>(mainPanel);
selectorDeProducto.bindItems(
    new ObservableProperty<>(PedidoAppStore.theStore(), "productosPorNombre"));
selectorDeProducto.bindValueToProperty("producto");
selectorDeProducto.setWidth(220);
selectorDeProducto.setHeight(80);
```

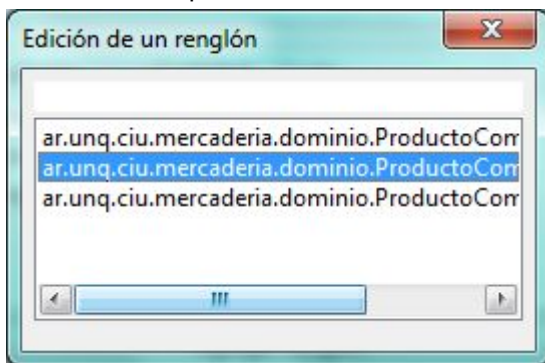
Notamos que la clase `List` es genérica, el tipo parámetro es de qué clase son los elementos de la lista. En este caso vamos a usar una `List<Producto>`.

¿Por qué se usó `bindItems` en lugar de `bindItemsToProperty`? Porque la propiedad que hay que vincular para los items no es una propiedad del modelo de la ventana (que es el renglón), es una propiedad de otro objeto.

El mensaje `bindItemsToProperty` es solamente para vincular con una propiedad del modelo de la ventana. Para armar vínculos con propiedades de otros objetos, hay que usar `bindItems` (sin "ToProperty") y pasarle un `ObservableProperty`.

Lo mismo pasa con `bindValueToProperty` / `bindValue`, y cualquier otro aspecto de un control (tipo de letra, color, y varios etc.) que se pueden vincular con el modelo. P.ej. para foreground, que es el color de frente, tenemos `bindForeground` y `bindForegroundToProperty`.

Veamos cómo queda nuestro formulario con el control de selección.

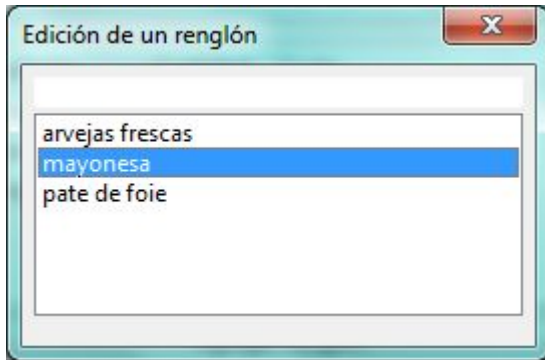


Tenemos una lista con tres elementos, hasta ahí vamos bien. Pero ... lo que se ve de cada producto no es lo que queremos.

Lo que queremos es que se muestre el nombre, o sea el valor de la propiedad nombre de cada producto. Pero eso ¡no lo dijimos cuando configuramos la `List`! Arena no tiene cómo adivinarlo. Para decirle a la `List` qué propiedad queremos que muestre de cada elemento, hay que decirle `setAdapter` ... pero no a la `List`, sino al objeto que devuelve `bindItems`, que es un `Binding` (o sea vínculo). Nos queda así:

```
List<Producto> selectorDeProducto = new List<Producto>(mainPanel);
selectorDeProducto
    .bindItems(
        new ObservableProperty<>(PedidoAppStore.theStore(), "productosPorNombre"))
        .setAdapter(new PropertyAdapter(Producto.class, "nombre"));
selectorDeProducto.bindValueToProperty("producto");
selectorDeProducto.setWidth(220);
selectorDeProducto.setHeight(80);
```

Ahora sí nuestra lista de selección se ve bonita.



Completamos el formulario de carga

El otro dato que se carga del renglón es la cantidad. Eso lo podemos manejar con un `NumericField`. También nos están faltando los nombres de los campos, que son `Label` de texto fijo. ¿Qué quiere decir “de texto fijo”? Que el valor se pone directamente usando `setText`. Un `Label` también puede ser de texto variable, para eso vinculamos su valor usando `bindValueToProperty` o `bindValue`.

Hasta acá, el código del `createFormPanel` queda así:

```
@Override
protected void createFormPanel(Panel mainPanel) {
    this.setTitle("Edición de un renglón");

    Label labelProducto = new Label(mainPanel);
    labelProducto.setText("Producto");
    List<Producto> selectorDeProducto = new List<Producto>(mainPanel);
    selectorDeProducto
        .bindItems(
            new ObservableProperty<>(PedidoAppStore.theStore(), "productosPorNombre"))
        .setAdapter(new PropertyAdapter(Producto.class, "nombre"));
    selectorDeProducto.bindValueToProperty("producto");
    selectorDeProducto.setWidth(220);
    selectorDeProducto.setHeight(80);

    new Label(mainPanel).setText("Cantidad");
    NumericField campoCantidad = new NumericField(mainPanel);
    campoCantidad.bindValueToProperty("cantidad");
}
```

Acá un detalle del código; observar cómo se configuró el label de cantidad. En lugar de

```
Label labelCantidad = new Label(mainPanel);
labelProducto.setText("Cantidad");
```

se hizo

```
new Label(mainPanel).setText("Cantidad");
```

derecho viejo.

¿Cómo se entiende esto? El enganche entre panel y control se hace en el constructor. Veamos el constructor de `Widget`, que es un antecesor de todos los controles de `Arena`:

```
public Widget(Container container) {
    this.container = container;
    container.addChild(this);
}
```

`Container` es una interface que es implementada por `Panel`. O sea, al crearse cualquier widget, ya se

agrega como “hijo” (o sea componente) del contenedor.

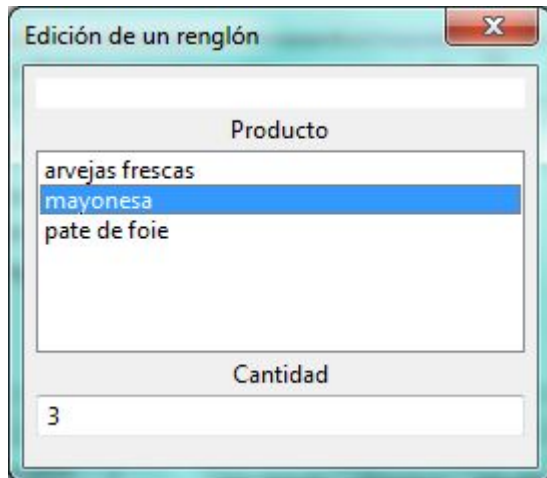
Después al label hay que decirle setText, y nada más. Eso se puede hacer como se ve en el código: el mensaje setText le llega al resultado de x, que es justamente el nuevo Label.

Nunca hizo falta asignar una variable. Y anda.

El otro Label se podría haber hecho así, y también el NumericField:

```
new NumericField(mainPanel).bindValueToProperty("cantidad");
```

A ver cómo andamos:



El costo – panel dentro de panel

Lo siguiente es agregar el costo del pedido. Este dato no se edita, es un cálculo. Este es el método en Renglon:

```
public double getCosto() {  
    return this.getProducto().getCosto() * this.getCantidad();  
}
```

Esto parece fácil: dos Labels, uno fijo con la palabra “Costo”, el otro vinculado a la propiedad costo del modelo de la ventana, que es el renglón. A estos Labels les ponemos color y un tamaño de letra un poco más grande. Vamos:

```
Label labelCosto = new Label(mainPanel);  
labelCosto.setText("Costo");  
labelCosto.setFontSize(10);  
labelCosto.setForeground(Color.BLUE.darker());  
  
Label valorCosto = new Label(mainPanel);  
valorCosto.bindValueToProperty("costo");  
valorCosto.setFontSize(10);  
valorCosto.setForeground(Color.RED.darker());
```

Para los colores usar java.awt.Color. Color.RED es una instancia de Color. Si a un color le envío el mensaje darker(), ¿qué piensan que devuelve? Otro color, un poco más oscuro.

Preguntas sobre esto:

1. ¿qué será Color.RED.lighter()?
2. ¿qué será Color.RED.darker().darker().darker()?

Otro desafío sobre esto: lograr una definición similar a la que hicimos arriba del label que pone la palabra “Cantidad”, o sea, que no sean necesarias las variables. OJO que no es tan fácil, si no sale preguntar.

A ver cómo queda el formulario con estos controles agregados. A la derecha se copia el formulario completo como está al principio de este documento.



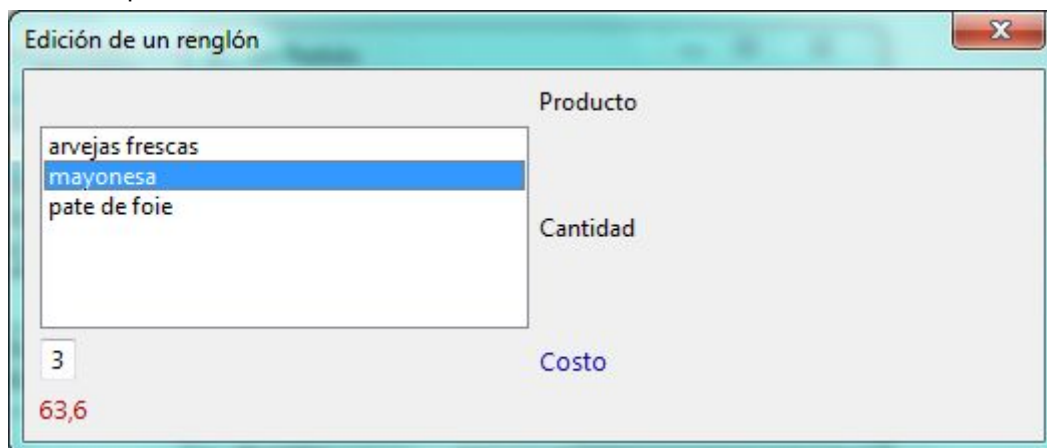
Auch, no están iguales (aparte de que faltan los botones, que vamos a agregar después). ¿Se ve dónde está la diferencia? En dónde aparece el costo de 63,6. Como lo hicimos nosotros, aparece debajo de la palabra “Costo”. En el formulario terminado, aparece a la derecha.

Ya hablamos de que para organizar los controles en un panel, usamos Layouts. La forma en que están la palabra “Costo” y el costo en el formulario de la derecha, sugiere un layout a dos columnas.

Uno podría tentarse de ponerle layout dos columnas al panel, así:

```
@Override
protected void createFormPanel(Panel mainPanel) {
    this.setTitle("Edición de un renglón");
    mainPanel.setLayout(new ColumnLayout(2));
    ... el seteo de todos los controles ...
}
```

Pero nos queda horrible⁴:



⁴ Además tira un error, porque a las listas de selección de Arena no les gusta meterse en un layout de columnas.

Lo que pasa es que **no queremos aplicar layout dos columnas a todo el formulario**. Lo de dos columnas tiene que ser **solamente** para el label “Costo” y el costo.

Esto lo logramos metiendo un panel solamente para estos dos controles, y a este panel interno darle el layout de dos columnas. O sea:

```
Panel panelCosto = new Panel(mainPanel);
panelCosto.setLayout(new ColumnLayout(2));

Label labelCosto = new Label(panelCosto);
labelCosto.setText("Costo");
labelCosto.setFontSize(10);
labelCosto.setForeground(Color.BLUE.darker());

Label valorCosto = new Label(panelCosto);
valorCosto.bindValueToProperty("costo");
valorCosto.setFontSize(10);
valorCosto.setForeground(Color.RED.darker());
```

Observar que los dos Label van al panelCosto, no al mainPanel. Al crear el panelCosto le pasamos quién es su “panel padre”. O sea que el panelCosto va a ser una “cajita”, que el mainPanel va a acomodar como una unidad, sin meterse en lo que hay adentro.

Ahora sí queda bonito⁵:

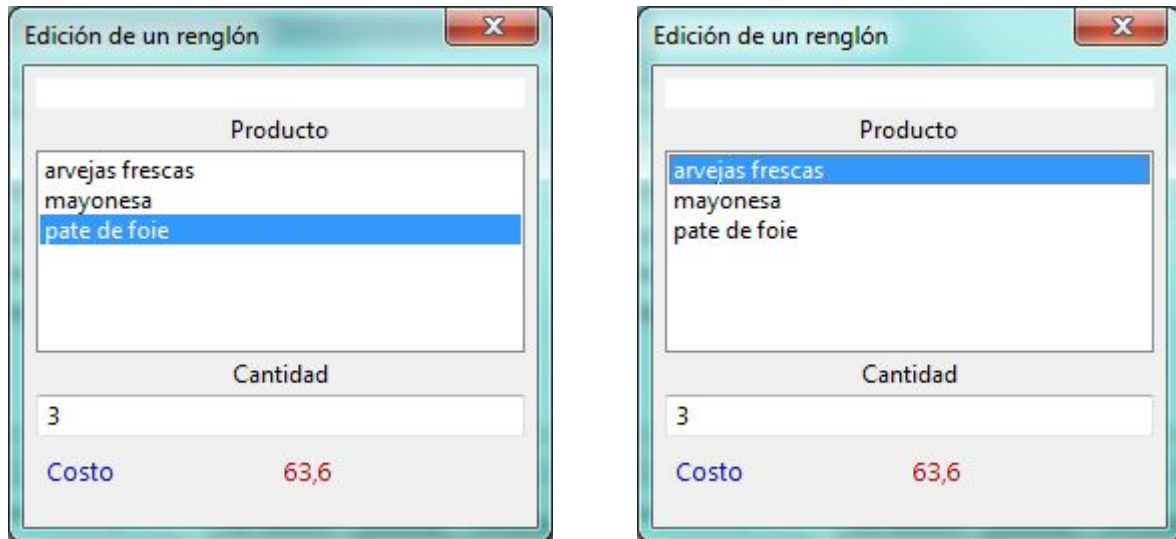


⁵ No olvidarse de borrar el seteo de layout al mainPanel, al principio de createFormPanel.

Recalculando ...

Se supone que el número que está en rojo está vinculado con la propiedad costo del producto. Entonces, debería cambiar cuando cambio el producto, porque como cada producto tiene un costo unitario distinto, el costo total del renglón (que, recordemos, es costo unitario * cantidad) también se modifica.

Pero ¡no anda!, ufa



Cambio el producto, y el costo siempre 63,6, como si fuera siempre mayonesa.

¿Qué pasa? Que el vínculo entre el control y la propiedad no es bidireccional en forma automática. Algunos vínculos Arena los puede manejar en forma totalmente automática, otros no.

Seamos un poco más precisos.

Si una propiedad (getter / setter) se corresponde directamente con un atributo (variable), entonces Arena puede mantener el vínculo bidireccional en forma automática. En el renglón, esto pasa con las propiedades producto y cantidad.

Por otro lado, la propiedad costo es *calculada*. Para las propiedades calculadas, el valor en pantalla no se actualiza automáticamente, porque Arena no puede saber cuándo cambia.

En el ejemplo, notamos que el valor de la propiedad costo cambia como consecuencia de haberse modificado el producto. Arena no tiene forma de saber qué eventos implican un cambio en el valor del costo, por lo tanto, no sabe en qué momentos tiene que redibujar el control bindeado a la propiedad costo. Por eso, la actualización no es automática.

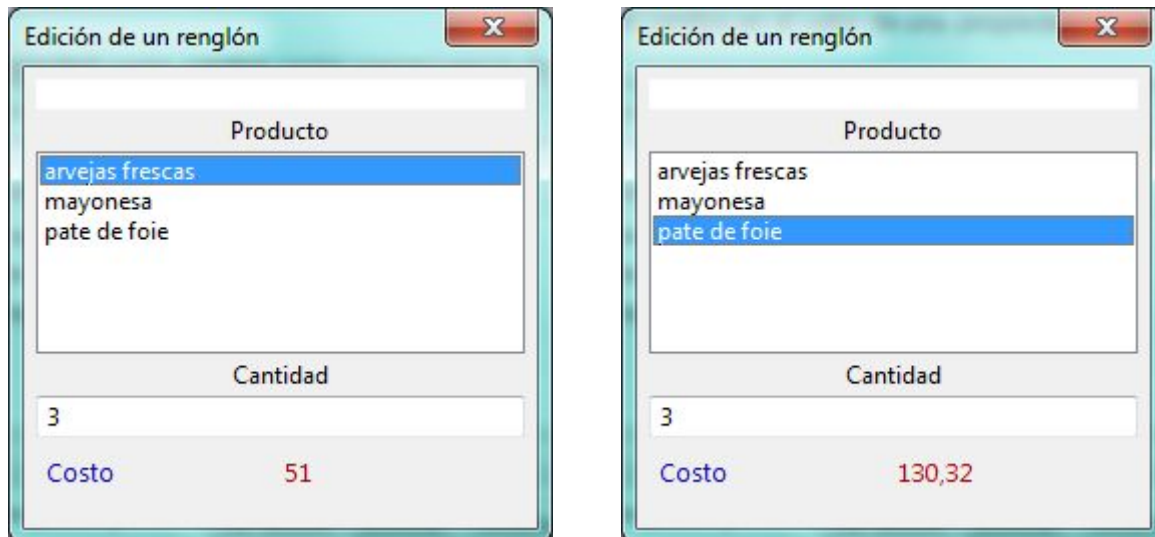
Para lograr que se actualicen los controles bindeados a propiedades calculadas, hay que avisarle a Arena ante cada evento que pueda implicar el cambio en el valor de una propiedad. Esto se hace lanzando un aviso de "propiedad cambiada".

En este caso, los eventos que pueden hacer que cambie el costo son la modificación de producto y de cantidad. Por lo tanto, agregamos los avisos en los métodos `setCantidad(double)` y `setProducto(Producto)` en `Renglon`. Quedan así:

```
public void setCantidad(double cant) {
    this.cantidad = cant;
    ObservableUtils.firePropertyChanged(this, "costo");
}
public Producto getProducto() { return this.producto; }
public void setProducto(Producto prod) {
    this.producto = prod;
    ObservableUtils.firePropertyChanged(this, "costo");
}
```

El método estático `firePropertyChanged` toma dos parámetros: el objeto (en este caso el renglón), y la propiedad calculada que puede haber cambiado (en este caso `costo`).

Ahora sí actualiza el valor del costo:



Edición de un renglón	
Producto	
arvejas frescas mayonesa pate de foie	
Cantidad	
3	
Costo	51

Edición de un renglón	
Producto	
arvejas frescas mayonesa pate de foie	
Cantidad	
3	
Costo	130,32

Un poco de debate

Pero ... ¿no estamos rompiendo la separación entre modelo y vista, al tocar los métodos `setProducto` y `setCantidad` en `Renglon`?

La verdad ... que sí. Lamentablemente, Arena no da (hasta donde se me ocurre) una alternativa. En otras tecnologías, ante cada evento se actualiza toda la interfaz, por lo que no es necesario notificar los posibles cambios en el valor de propiedades.

También es cierto que los agregados en la clase `Renglon` no obligan a levantar una interfaz gráfica para usar dicha clase. P.ej. los tests que usan renglones siguen funcionando sin necesidad de “levantar Arena” o crear una interfaz. La única dependencia es que hay que incorporar la librería que incluye la clase `ObservableUtils`.

Aceptar y cancelar

Lo único que falta en el formulario son los botones. Estos son instancias de la clase `Button`, que conviene configurar en una forma un poco peculiar. Este es el código que debe agregarse en el método `createFormPanel`.

```
Button acceptButton = new Button(mainPanel);
acceptButton.setCaption("Aceptar");
acceptButton.onClick(() -> this.accept());

Button cancelButton = new Button(mainPanel);
cancelButton.setCaption("Cancelar");
cancelButton.onClick(() -> this.cancel());
```

El formulario tiene la forma que se espera.

Edición de un renglón

Producto

- arvejas frescas
- mayonesa**
- pate de foie

Cantidad

3

Costo 63,6

Aceptar

Cancelar

Lo peculiar son los closures que se evalúan al pulsar los botones: `this.accept()` y `this.cancel()`. Acá `this` es la instancia de `RenglonForm`, que es subclase de `Dialog`. Para entender estos closures, conviene ver cómo se abre un `RenglonForm` desde otra ventana.

Los dos usos de `RenglonForm`.

El formulario se abre desde una ventana que muestra un pedido, la clase es `PedidoWindow`.

Un Pedido

Producto	Cantidad	Costo
mayonesa	3.0	63.5999999...
pate de foie	5.0	217.2
arvejas frescas	1.0	17.0

Agregar renglón

Renglón 1 Ver detalle Eliminar **Modificar** 3.0

Costo total: 297,8

El modelo de esta ventana es una instancia de `Pedido`.

Hay **dos** botones que lanzan un `RenglonForm`: el de **Agregar renglón**, y el de **Modificar**. Hay dos diferencias entre las acciones de agregar y de modificar:

1. Para agregar, el modelo del `RenglonForm` debe ser un renglón nuevo. Para modificar, debe ser uno de los renglones que actualmente están en el pedido.
2. Qué hay que hacer cuando se acepta o se cancela el form. Para indicar esto desde la clase que usa `RenglonForm`, la clase `Dialog` entiende los mensajes `onAccept` y `onCancel`. Estos

dos mensajes tienen un parámetro que es un closure. El closure de `onAccept` se ejecuta cuando al `Dialog` le dicen `accept()`. Con `onCancel` es análogo respecto de `cancel()`.

Veamos el código que abre un `RenglonForm` para agregar un renglón. Este código está en la clase `PedidoWindow`.

```
private void agregarRenglon() {
    Renglon nuevoRenglon = new Renglon();
    RenglonForm form = new RenglonForm(this, nuevoRenglon);
    form.onAccept(() -> {
        this.getModelObject().addToRenglones(nuevoRenglon);
        ObservableUtils.firePropertyChanged(this.getModelObject(), "costo");
    });
    form.open();
}
```

En este código se pueden apreciar dos cosas:

1. El modelo del `RenglonForm` es un renglón recién creado.
2. El closure que se pasa como parámetro de `onAccept`, es la acción que se ejecuta cuando se pulsa el botón **Aceptar** en el formulario. El `firePropertyChanged` está para que se actualice el costo total del pedido, que aparece en la ventana de pedido debajo de todo.

Ahora veamos qué se hace para modificar un renglón.

```
private void modificarRenglon(Renglon renglonActual) {
    RenglonForm form = new RenglonForm(this, renglonActual);
    form.onAccept(() -> {
        // que hay que hacer aca ??
        ObservableUtils.firePropertyChanged(this.getModelObject(), "costo");
    });
    form.open();
}
```

En este caso, el modelo del `RenglonForm` es `renglonActual`, que es el renglón “elegido” al poner el número de renglón correspondiente en el campo que está a la izquierda de los tres botones.

Cuando el formulario se acepta, lo que hay que hacer seguro que no es lo mismo que cuando se abre para agregar un renglón nuevo.

Conclusión: podemos usar el mismo formulario para crear y para modificar, porque la clase `Dialog` permite configurar las acciones de aceptar y cancelar. Ahora, **cuándo** se ejecutan estas acciones hay que configurarlo en la clase del formulario, poniendo `this.accept()` y `this.cancel()` cuando corresponda⁶.

¿Qué hay que hacer para modificar?

Fíjense que no está especificada la acción que hay que realizar cuando se acepta el formulario abierto para **modificar** un renglón. Está este comentario: `// que hay que hacer aca ??`.

Repasemos los bindings del formulario. La lista de edición está vinculada a la propiedad `producto`, el campo numérico a la propiedad `cantidad`. Estas dos propiedades corresponden a dos atributos del renglón, por lo tanto, `Arena` maneja el binding bidireccional en forma automática.

Eso quiere decir ... que **el renglón se modifica en el momento en que se cambian los valores** en el formulario. Los cambios no “esperan” a que se pulse el botón **Aceptar**.

⁶ En otras tecnologías, los formularios ya nacen con dos botones, el de aceptar y el de cancelar.

Abramos el formulario para modificar el primer renglón

Producto	Cantidad	Costo
mayonesa	3.0	63.5999999...
pate de foie	5.0	217.2
arvejas frescas	1.0	17.0

Agregar renglón

Renglón 1 Ver detalle Eliminar **Modificar** 3.0

Costo total: 297,8

Edición de un renglón

Producto

- arvejas frescas
- mayonesa**
- pate de foie

Cantidad

3

Costo 63,6

Aceptar

Cancelar

Al cambiar la cantidad en el form ...

Producto	Cantidad	Costo
mayonesa	4.0	84.8
pate de foie	5.0	217.2
arvejas frescas	1.0	17.0

Agregar renglón

Renglón 1 Ver detalle Eliminar **Modificar** 4.0

Costo total: 297,8

Edición de un renglón

Producto

- arvejas frescas
- mayonesa**
- pate de foie

Cantidad

4

Costo 84,8

Aceptar

Cancelar

En la primer línea de la tabla, la cantidad ya es 4, sin que se haya pulsado **Aceptar** en el formulario.

Una forma de solucionar este problema es modificar sobre una copia del renglón que está en el formulario, y cuando se pulsa aceptar, actualizar el renglón.

```
private void modificarRenglón(Renglón renglónActual) {
    Renglón renglónCopia = new Renglón();
    renglónCopia.setCantidad(renglónActual.getCantidad());
    renglónCopia.setProducto(renglónActual.getProducto());

    RenglónForm form = new RenglónForm(this, renglónCopia);
    form.onAccept(() -> {
        this.getModelObject().replaceInRenglones(
            renglónCopia, renglónActual
        );
        ObservableUtils.firePropertyChanged(this.getModelObject(), "costo");
    });
    form.open();
}
```

Hay que agregar el método `replaceInRenglones` en `Pedido`.

```
public void replaceInRenglones(Renglon nuevoRenglon, Renglon renglonActual) {  
    renglonActual.setCantidad(nuevoRenglon.getCantidad());  
    renglonActual.setProducto(nuevoRenglon.getProducto());  
}
```

También se puede sacar el renglón actual y poner el nuevo. Pero ¡OJO! hay que insertar el renglón nuevo en el mismo lugar, en la lista, en donde estaba el renglón que se está reemplazando.

```
public void replaceInRenglones(Renglon nuevoRenglon, Renglon renglonActual) {  
    int index = renglones.indexOf(renglonActual);  
    renglones.remove(renglonActual);  
    renglones.add(index, nuevoRenglon);  
}
```