# Capítulo 1. MOSKitt: Generación de código OpenXava (v.0.3.0)

## Tabla de contenidos



Versión pdf de este documento.

# Introducción

El módulo Codgen-OpenXava v.0.3.0 de MOSKitt proporciona la generación semi-automática de aplicaciones OpenXava [http://www.openxava.org].

Hablamos de generación semi-automática para indicar que el código de la aplicación OpenXava que se obtiene tras la generación deberá ser completado por los desarrolladores.

Desde una visión global, la generación de código propuesta por MOSKitt consiste en un conjunto de transformaciones de modelos hasta llegar al código final siguiendo una estragegia de Desarrollo de Software Dirigido por Modelos (DSDM).

# ¿Qué es OpenXava?

Tal y como se indica en su página del proyecto OpenXava [http://www.openxava.org]:

"OpenXava es una herramienta para el desarrollo rápido de aplicaciones web, apropiada para aplicaciones de gestión y orientadas a bases de datos.

Las aplicaciones generadas por OpenXava son aplicaciones Java EE/J2EE estándar. OpenXava usa JPA, Hibernate o EJB CMP2 (hasta OX3.1.4) para persistencia, y algo de estado con alcance de sesión a nivel de servlets.

Con respecto al sistema de gestión de usuarios y navegación OpenXava cuenta con los módulos. Los modulos generados por OpenXava son portlets estándar, por lo tanto son desplegables en cualquier portal Java compatible con su propias características de navegación y seguridad. El sitio de OpenXava usa Liferay Portal , por lo tanto en este caso se usa la seguridad de Liferay, pero se puede escoger cualquier otro portal o crear nuestro propio sistema de seguridad y navegación para los módulos OpenXava".

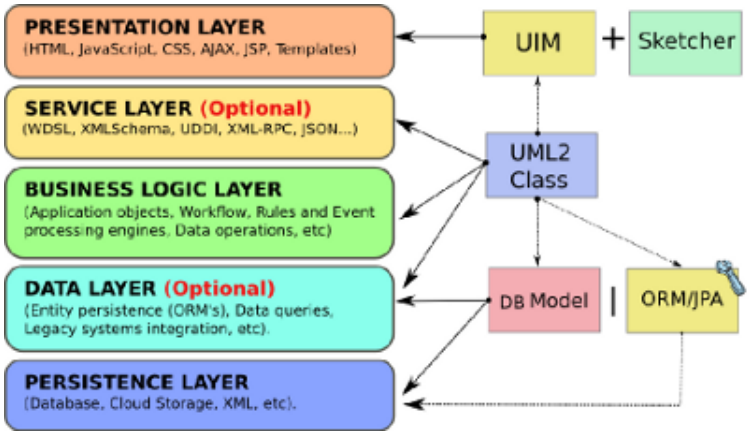
OpenXava es un proyecto Open Source con licencia LGPL (GNU Lesser General Public License). Para más información sobre OpenXava consultar su portal en www.openxava.org.

# Aplicaciones Web: Arquitectura y modelos

Puesto que estamos hablando de aplicaciones web vamos a analizar desde el punto de vista del modelado de qué se componen este tipo de aplicaciones. Una aplicación web sigue una arquitectura N-Capas, cuyos componentes, de forma genérica, son:

- Capa de Presentación
- Capa de Lógica de Negocio (puede incluir servicios).
- Capa de Persistencia de Datos (puede incluir una capa de gestión de los datos).

Especificar una aplicación web consiste en especificar cada uno de estos componentes. En la siguiente figura se muestran qué modelos propone MOSKitt para especificar cada uno de ellos:




Las aplicaciones OpenXava, por ser aplicaciones web seguirán el esquema anterior con respecto a su modelado. Siguiendo este esquema, los modelos necesarios para especificar una aplicación en OpenXava y que por tanto serán la entrada del generador de código en MOSKitt son:

- Modelo UML2: para la definición de la Lógica de Negocio y sus Entidades.
- Modelo JPA: para la definición de la persistencia de las entidades definidas en UML2.
- Modelo Sketcher+ Modelo UIM (User Interface Model): para la definición de la Interfaz de Usuario.

El responsable de construir todos estos modelos es el analista/diseñador, sin embargo, el principal consumidor del código Java que se obtiene como resultado es el desarrollador, que es quien finalmente deberá completarlo.

# Descripción de un caso de estudio: Desarrollo de una aplicación de facturación.

Vamos a seguir los pasos necesarios para desarrollar una aplicación OpenXava con MOSKitt siguiendo una aproximación DSDM (Desarrollo de Software Dirigido por Modelos). Para ello, en este manual se acompañará cada paso del proceso con su correspondiente ejemplo.

En concreto, vamos a desarrollar una aplicación de facturación para una pequeña empresa. En el modelado de la aplicación aparecen los siguientes elementos principales:

- Cliente (Customer)
- Albarán (Delivery)
- Factura (Invoice)
- Producto (Product)
- Vendedor (Seller)
- Almacén (WhareHouse)

De forma muy sencilla podemos resumir los requisitos del siguiente modo:

- La aplicación deberá permitir el mantenimiento de todas estas entidades.
- Un cliente puede tener una o más facturas y uno o más albaranes.
- Un albarán sólo puede tener una facura.
- Cada factura, además de información específica de la factura, indicará en cada una de sus líneas la cantidad facturada de un producto concreto.
- Un producto puede aparecer en más de una factura.
- Un producto lo sirve un determinado proveedor.

A lo largo del documento vamos a aplicar sobre este caso de estudio el proceso de desarrollo de software propuesto por MOSKitt.

# Presentación del Proceso DSDM propuesto por MOSKitt.

Vamos ahora a dar un repaso al proceso de desarrollo de software propuesto por MOSKitt para construir aplicaciones, en este caso aplicaciones OpenXava. Proceso basado en la estrategia DSDM (Desarrollo de Software Dirigido por Modelos).

## Módulos de MOSKitt que participan en el proceso.

Los principales módulos que intervienen en el proceso y que por tanto tendremos que tener instalados son:

- **MOSKitt-UML2**: Este módulo permite especificar modelos UML2 haciendo un uso intensivo de los Diagramas de Clases (también proporciona otros diagramas como son los de secuencia, actividad y estados, sin embargo, estos últimos no son utilizados durante la generación de código).

- **MOSKitt-Sketcher**: Módulo que permite la definición de interfaces de usuario guiadas por el diseño de su aspecto. Es algo más que un típico "pintador" de interfaces ya que también permite enlazar los diferentes elementos del Sketcher (widgets) con elementos del modelo de clases de UML2 (propiedades ó métodos).

- **MOSKitt-UIM**: Módulo que permite la definición de interfaces de usuario de una forma abstracta. Un modelo inicial transformable en código OpenXava puede obterse a partir de un modelo de Sketcher que cumpla ciertos requisitos. UIM es la entrada de la generación de código y a través de él se alcanzan el resto de los modelos utilizados para especificar la aplicación.

- **MOSKitt-codgen-OpenXava**: Es el módulo que proporciona la generación de código OpenXava. Este módulo está incluido en el módulo *MOSKitt.cogden.JEE* publicado en el apartado de descargas de MOSKitt http://www.moskitt.org/cas/moskitt-descargas/.

Además de estos módulos, existen otros módulos auxiliares que también deben estar instalados en MOSKitt para conseguir la generación. Estos módulos son:

- **MOSKitt-Sketcher2UIM**: Proporciona las transformaciones necesarias para, a partir de un modelo de Sketcher obtener un modelo de UIM equivalente e interpretable por el generador de OpenXava.

- **MOSKitt-UITemplates**: Contiene plantillas de componentes de interfaz predefinidos para facilitar al usuario el modelado de aplicaciones web. Estos componentes son interpretables por el generador.

Para más información sobre cómo instalar estos módulos y las dependencias existentes entre ellos consultar la web de MOSKitt [http://www.moskitt.org] apartado de descargas [http://www.moskitt.org/cas/moskitt-descargas/].

Para más información sobre los módulos funcioneales que proporciona MOSKitt consultar el apartado Módulos [???] en la web MOSKitt, en el que se describen cada uno de ellos).

# Roles involucrados en el proceso: Analista y Desarrollador

Los perfiles a quienes va dirigido este manual son los de *Analista* y *Desarrollador* para que:

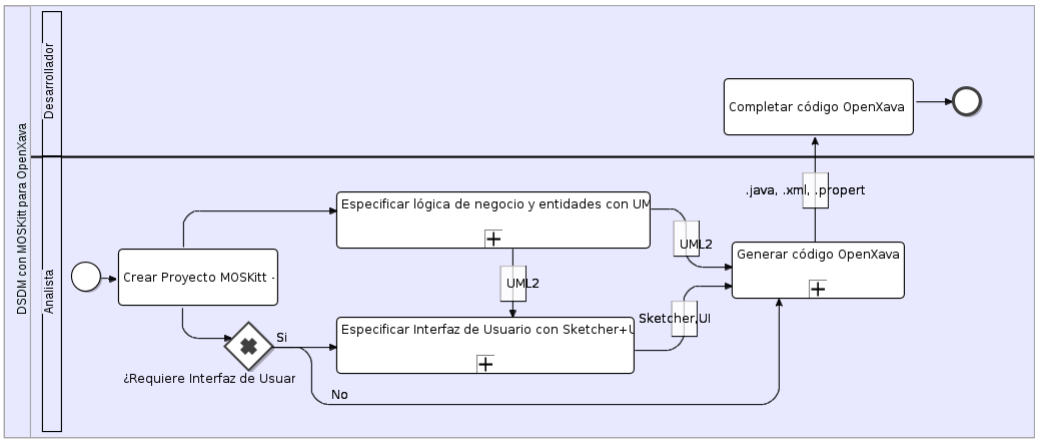
- Los *analistas* conozcan bien cómo deben construir los modelos para que el generador sepa interpretarlos.

- Los *desarrolladores* (1) sepan de dónde proviene cada una de las partes del código generadas (de qué modelo ha extraido la información que le ha servido de base), (2) qué partes del código son susceptibles de ser completadas o revisadas en función del modelo definido por el analista, (3) qué mecanismos les proporciona MOSKitt para explotar la información contenida en los modelos y, por último, (4) qué mecanismos se deben utilizar para que MOSKitt respete los cambios realizados sobre el código por los desarrolladores para que, cuanto los analistas modifiquen los modelos, sus cambios sean respetados.

**MOSKitt-codgen-OpenXava** es el módulo que hace de puente entre ambos roles y es muy importante que los dos conozcan el proceso completo para comprender mejor el trabajo y realizarlo así de una forma más cómoda. Aunque en este manual nos centramos principalmente en este módulo, consideramos oportuno incorporar información del resto de los módulos que intervienen en el proceso para que quede más claro. Por este motivo, aunque en el manual de usuario de cada uno de los módulos MOSKitt puede encontrarse la ayuda completa, en este manual vamos a hacer un pequeño repaso al desarrollar el tutorial sobre "Tutorial: Cómo desarrollar aplicaciones OpenXava con MOSKitt paso a paso".

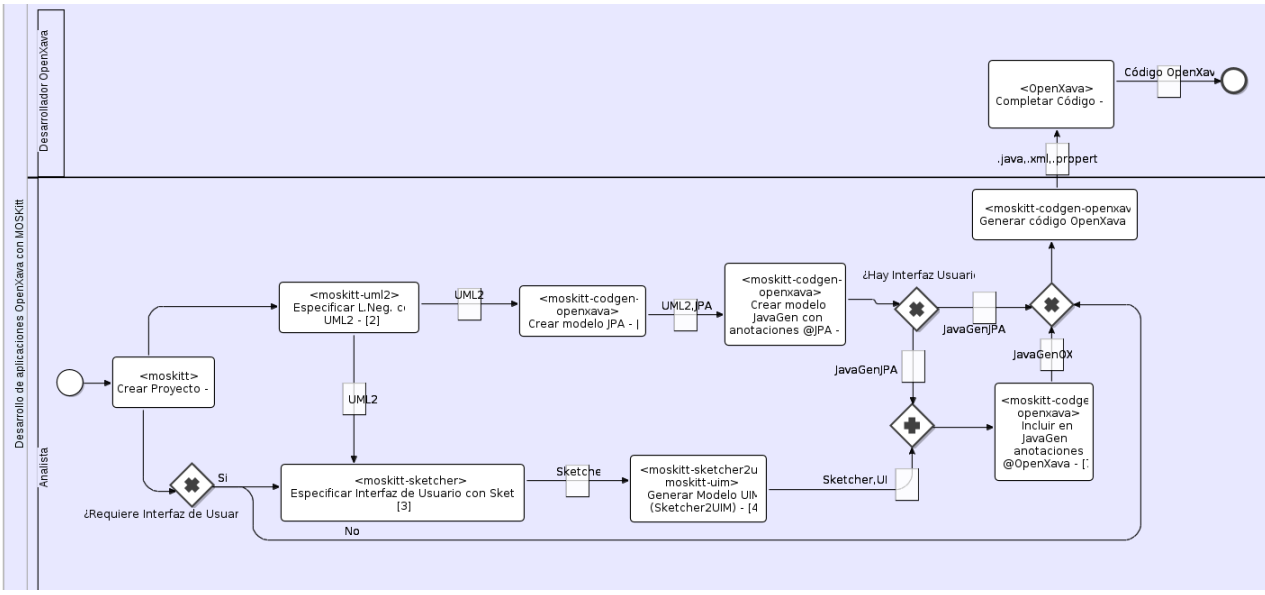
# Presentación del proceso de construcción de una aplicación OpenXava con MOSkitt

En este punto vamos a describir a través de un diagrama BPMN (Bussines Process Modeling Notation) los pasos que se deben seguir para obtener una aplicación OpenXava con el generador MOSKitt. En el diagrama también se indica el reparto de tareas entre el rol de Analista y el de Desarrollador.




Siguiendo el argumento expuesto en el punto relativo a Arquitectura y Modelos, el diagrama anterior indica que el analista debe especificar la lógica de negocio definiendo el modelo UML2 correspondiente. En caso de ser necesario, también deberá especificar la interfaz de usuario de la aplicación mediante los modelos Sketcher y UIM. A partir de todos estos modelos, el generador MOSKitt-codgen-OpenXava generará una versión inicial de la aplicación OpenXava la cual deberá ser completada por el Desarrollador.

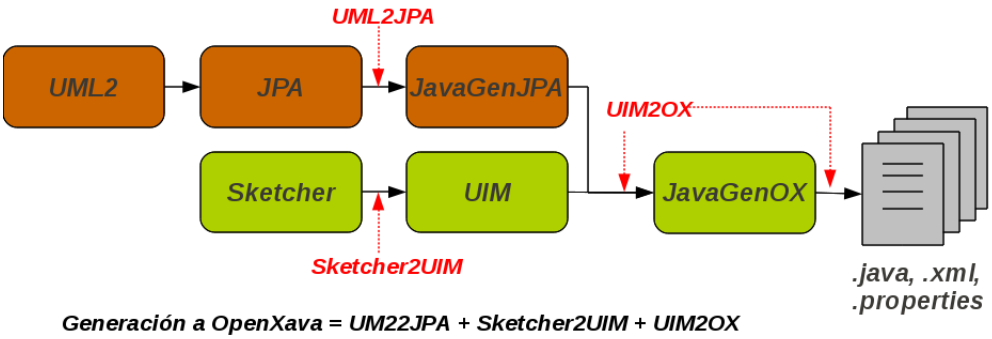
Vamos ahora a ver el proceso con un poco más de detalle, para lo cual podemos ver en la siguiente imagen como se han descompuesto los subprocesos del diagrama BPMN anterior. Además, en la cabecera de cada tarea se muestra el módulo MOSKitt que nos proporciona la funcionalidad necesaria para llevarla a cabo (Ejemplo: `<moskitt-uml2>`, `<moskitt-sketcher>` etc...).




Siguiendo con lo indicado en el apartado Arquitectura y Modelos, en este diagrama podemos ver cómo partimos de modelos abstractos, independientes de la plataforma:

- Modelo UML2: Lógica de negocio (operaciones y entidades del dominio).
- Modelo Sketcher + Modelo UIM: Interfaz de Usuario.

El modelo UML2 se apoya en un modelo JPA que le permite configurar cómo deben ser persistidas las entidades del dominio. A partir de todos estos modelos se genera un modelo `JavaGen` más cercano a la plataforma de implementación con la información correspondiente a todas las clases Java con sus anotaciones, tanto las anotaciones JPA (Java Persistence API) relativas a la persistencia de los datos como las relativas a la capa de presentación ya específicas de OpenXava. Es a partir de este modelo `JavaGen` a partir del cual se obtiene finalmente los ficheros con el código Java generado.




***Generación a OpenXava = UM22JPA + Sketcher2UIM + UIM2OX***
***(Cadena de Transformaciones)***

Como podemos ver en la figura anterior, la generación OpenXava consiste en realidad en una cadena de transformaciones entre modelos que dan como resultado el código OpenXava final.

En función de la naturaleza de la aplicación el analista puede o no requerir la especificación de la capa de interfaz. Para que la especificación de la interfaz de usuario realizada con el Sketcher esté completa, debe relacionarse con los elementos del modelo UML2. De este modo, el Analista puede expresar aspectos como son: qué propiedades de las clases se van a mostrar/editar desde la interfaz y qué métodos del negocio van a ser invocados desde ésta. El modelo de UIM generado de forma automática a partir del modelo de Sketcher servirá como entrada al generador.

La generación se lanzará a partir del modelo de UIM, desde el cual el generador podrá alcanzar tanto el modelo de Sketcher para averiguar la disposición de los widgets en los formularios como el modelo de UML2 con el que está relacionado de donde podrá obtener información relativa a los tipos de datos, cardinalidades, navegabilidad entre clases etc...

NOTA: Aunque no lo hemos indicado en el diagrama BPMN por motivos de simplicidad, en muchas ocasiones las cosas no se empiezan de cero y se dispone ya de un esquema de bases de datos inicial que puede servir de base en el análisis. En este caso, MOSKitt nos permite obtener el modelo UML2 correspondiente a partir de la Base de Datos física existente (para más información sobre cómo obtener el modelo UML2 a partir de una base de datos física consultar el Manual de Usuario de los módulos MOSKitt-DB y MOSKitt-dbtransformations). En la sección de anexos de este manual hay un apartado que explica cómo realizar los pasos más importantes de estas tareas.

En los siguientes puntos pasamos a revisar cada una de las tareas aquí descritas. Lo vamos a hacer siguiendo el caso de estudio de la aplicación de facturación.

# Tutorial: Cómo desarrollar aplicaciones OpenXava con MOSKitt paso a paso.

Vamos ahora a dar un repaso al proceso de desarrollo de software propuesto por MOSKitt para construir aplicaciones, paso a paso.

## Tarea 1: Preparar el entorno y crear el proyecto MOSKitt

Lo primero que tenemos que hacer es instalar MOSKitt con todos los plugins necesarios para ejecutar la transformación y a continuación, crear los proyectos eclipse necesarios para trabajar. Para ello seguiremos los siguientes pasos:

# Preparación del entorno MOSKitt

A continuación están enumerados los requerimientos necesarios para poder ejecutar transformaciones de generación de código a OpenXava en MOSKitt.

1. La RCP *MOSKitt 1.3.7*

2. Los siguientes proyectos de Eclipse:

   - **Eclipse Webtools**

   - **Model Discovery (MODISCO)**

3. Los siguientes módulos MOSKitt:

   a. **MOSKitt MDT Storage 0.9.3**

   b. **MOSKitt UIM (User Interface Modeling)**

   c. **MOSKitt User Interface Templates**

   d. **MOSKitt Sketcher 1.7.0 (User Interface Sketching)**

   e. **MOSKitt Sketcher2UIM transformations**

   f. **MOSKitt code generation for Java/JEE tools**

Si aún no tienes instalado MOSKitt puedes:

1. Descargar todo el entorno de ejecución ya preparado para la generación de OpenXava localizado en http://www.moskitt.org/cas/openxava_instalation/ (contiene la RCP 1.3.7 de MOSKitt con todos los plugins necesarios ya instalados).

2. Descomprimir el fichero `moskitt-1.3.7.v201108160830-jee_codgen-linux.gtk.x86.zip` para Linux ó `moskitt-1.3.7.v201108160830-jee_codgen-win32.win32.x86.zip en el caso de Windows.`

Sin embargo, si ya tienes MOSKitt 1.3.7 instalado y no quieres optar por la opción anterior, sobre esta instalación de MOSKitt debes:

1. Registrar los siguientes repositorios utilizando la opción `Help/Install new software...`:

   - **MOSKitt MDT Storage 0.9.3**: http://download.moskitt.org/moskitt/mdt.storage/updates-1.3.4

   - **MOSKitt UIM (User Interface Modeling)**: http://download.moskitt.org/moskitt/uim/updates-1.3.7

   - **MOSKitt User Interface Templates**: http://download.moskitt.org/moskitt/uitemplates.cit/updates-1.3.7

   - **MOSKitt Sketcher 1.7.0 (User Interface Sketching)**: http://download.moskitt.org/moskitt/sketcher/updates-1.3.7

   - **MOSKitt Sketcher2UIM transformations**: http://download.moskitt.org/moskitt/sketcher2uim/updates-1.3.7

   - **MOSKitt code generation for Java/JEE tools**: http://download.moskitt.org/moskitt/codgen.jee/updates-1.3.7

2. De nuevo, desde la opción `Help/Install new software...` instalar cada uno de los proyectos o módulos en el siguiente orden:

a. **MOSKitt MDT Storage 0.9.3**: seleccionar e instalar la feature *Artifacts Storage for MOSKitt*

b. **MOSKitt UIM (User Interface Modeling)**: seleccionar e instalar la feature *Functional module UIM*

c. **MOSKitt User Interface Templates**: seleccionar e instalar la feature *Functional module UiTemplates*

d. **MOSKitt Sketcher 1.7.0 (User Interface Sketching)**: seleccionar e instalar la feature *Functional module Sketcher*

e. **MOSKitt Sketcher2UIM transformations**: seleccionar e instalar la feature *Moskitt Sketcher to UIM generic Transformation feature*

f. **MOSKitt code generation for Java/JEE tools**: Activar la opción `Contact all update sites during install...`, seleccionar e instalar la feature *MOSKitt code generation for Java EE.*

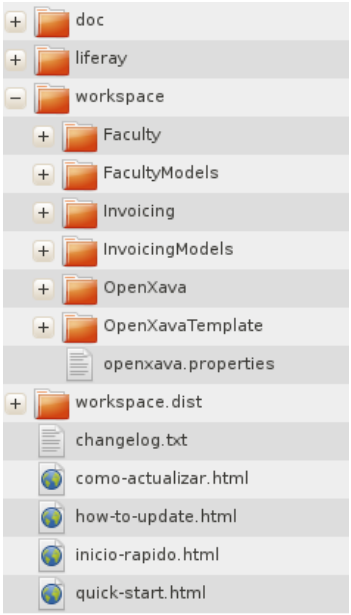
## Creación del proyecto MOSKitt

Una vez hemos instalado el entorno MOSKitt, debemos crear los proyectos Eclipse necesarios para trabajar: el proyecto MOSKitt para diseñar la aplicación y el proyecto Java para depositar allí el código generado por MOSKitt.

Para ello seguiremos los siguientes pasos:

1. Descargar el fichero `oxportal-4.2.2.zip` localizado en la web MOSKitt en el apartado Entorno de desarrollo OpenXava [http://www.moskitt.org/cas/openxava_devenv/] en la sección Generación de Código/OpenXava [http://www.moskitt.org/cas/openxava/].

2. Descomprimir el archivo en el sistema de ficheros obteniendo el siguiente árbol de directorios:




3. Abrir MOSKitt seleccionando como workspace el que se ha creado en `oxportal/workspace`.

4. Crear un nuevo `proyecto MOSKitt` en el cual vamos a localizar nuestros modelos: `File/New/MOSKitt Project`.

Para más información sobre el contenido de cada uno de los directorios proporcionados por MOSKitt en el archivo `oxportal-4.2.2.zip` consultar el apartado sobre "Ejecución del proyecto OpenXava".

Todos los modelos que aparecen en este tutorial están basados en los del proyecto InvoicingModels contenido en el fichero `oxportal-4.2.2.zip`. Una vez descomprimido el fichero, el proyecto está localizado en el directorio `oxportal/workspace/InvoicingModels`. Este proyecto contiene los mismos modelos descritos en este tutorial e, incluso en algunos casos los modelos has sido enriquecidos.

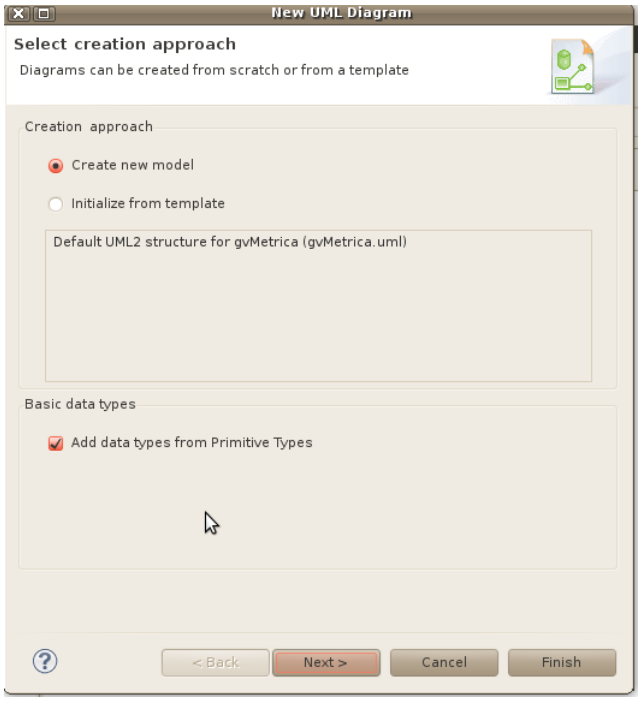
Si el usuario quiere comprobar el resultado de la generación de código OpenXava a partir de los modelos ya proporcionados sin construirlos él mismo, puede pasar directamente al punto "Tarea 4: Generar código OpenXava".

# Tarea 2: Especificar la Lógica de Negocio: Modelo UML2

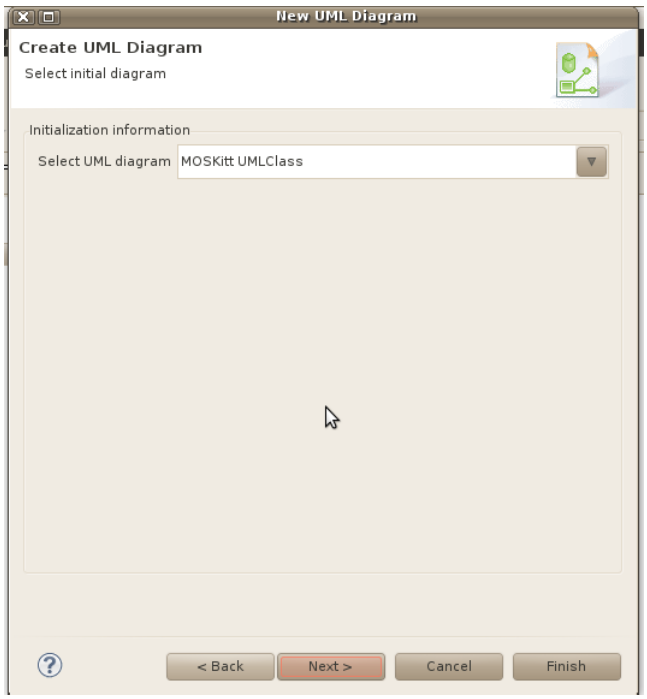
A partir de los requisitos establecidos para el caso de estudio, el analista, haciendo uso del módulo MOSKitt-UML2, define el diagrama de clases en el que describe las entidades que intervienen en el Sistema de Información de facturación, sus propiedades y sus métodos.

Para ello crearemos un nuevo modelo UML2 (`File/New/MOSKitt UML Diagram`) al cual nombraremos como `invoice_domain` tal siguiendo los pasos que muestran las figuras que aparecen a continuación.

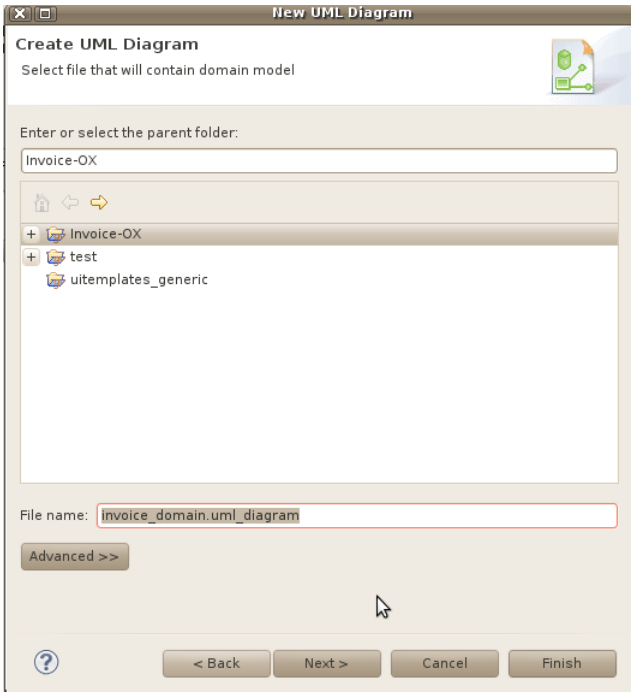
1. Creación de un modelo nuevo con la opción `Create new model` marcada. Indicamos también que el modelo incluya los tipos primitivos de UML2 por defecto (`Add data types from Primitive Types`):



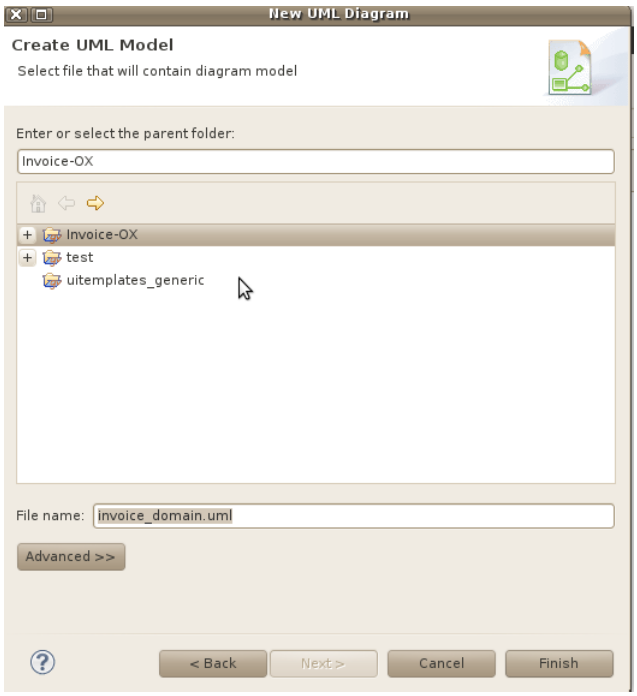

2. Seleccionamos como Diagrama principal un Diagrama de Clases pues sólo vamos a hacer uso de este tipo de diagramas UML2:




3. Seleccionamos el proyecto que debe incluir el diagrama del modelo (`Enter or select the parten folder`) y le asignamos un un nombre (`File name:`):

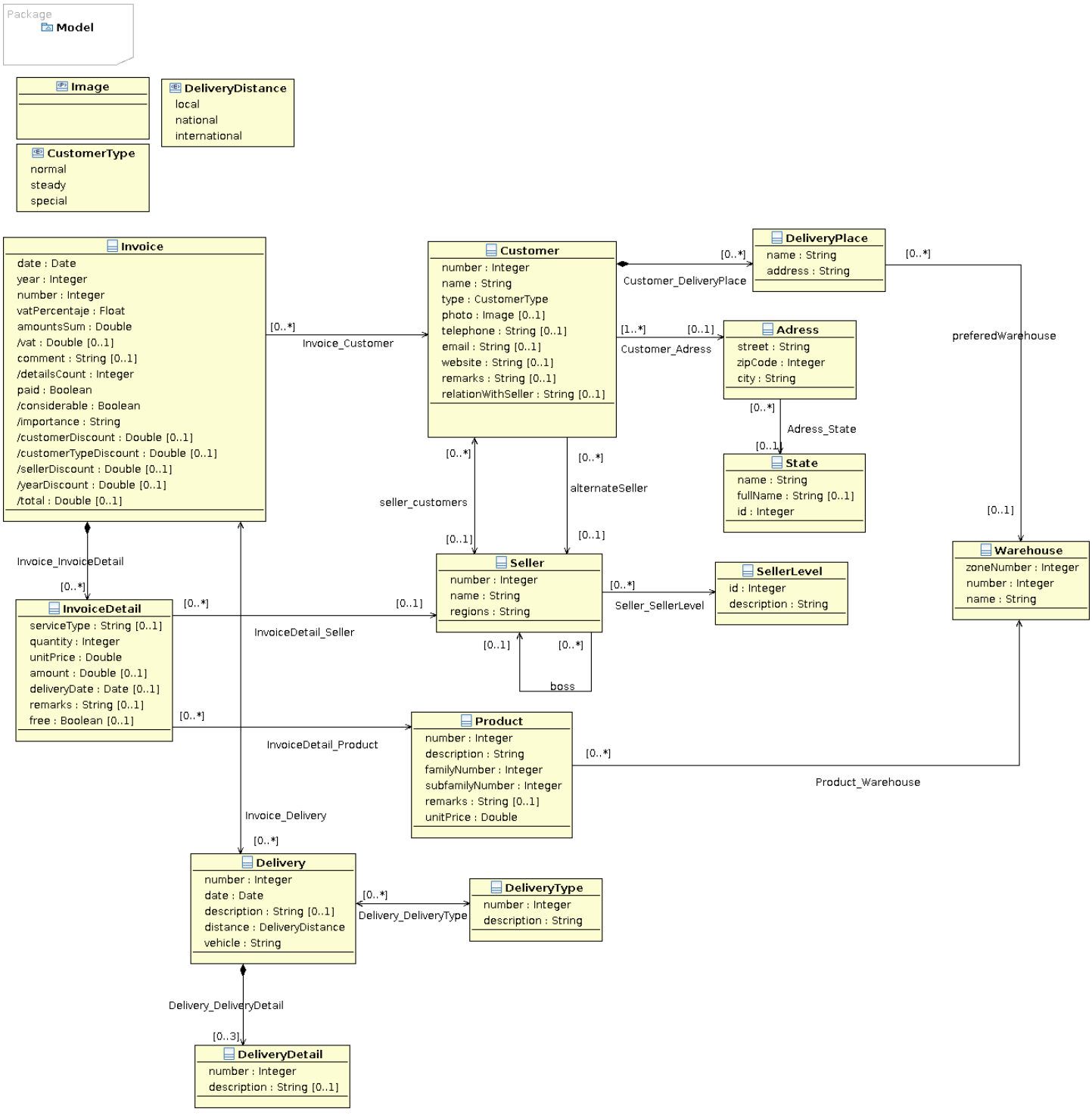



4. Seleccionamos el proyecto que debe incluir el modelo (`Enter or select the parten folder`) y le asignamos un un nombre (`File name:`). Por defecto MOSKitt le asignará el mismo nombre que al diagrama:




A continuación, se abrirá un diagrama de clases vacío para empezar a modelar nuestra aplicación.

Asumimos el siguiente modelo de clases (modelo UML2) para modelar la lógica de negocio de la aplicación de facturación que queremos generar:




Notese que no se han definido métodos en el diagrama de clases. Esto es debido a que en su primera versión, la aplicación de facturación contendrá sólo los mantenimientos de todas sus entidades sin ninguna necesidad adicional: sólo inserciones, modificaciones y borrados (vamos a considerar éstas como operaciones básicas proporcionadas por el framework).

Recursos de Entrada de la tarea:

- Catálogo de Requisitos Funcionales.

Recursos de Salida de la tarea:

- Modelo de UML2 con las clases del sistema de facturación (`invoice_domain.uml`).

- Diagrama de clases (`invoice_domain.uml_diagram`).

Para más información sobre cómo modelar con MOSKitt-UML2 consultar el manual de usuario en la ayuda de MOSKitt.

# Tarea 3: Especificar la Interfaz de Usuario: Modelos de Sketcher y UIM

Vamos a ver ahora cómo definir la interfaz de la aplicación de facturación con el módulo **MOSKitt-Sketcher** y cómo obtener a partir de este el modelo UIM.

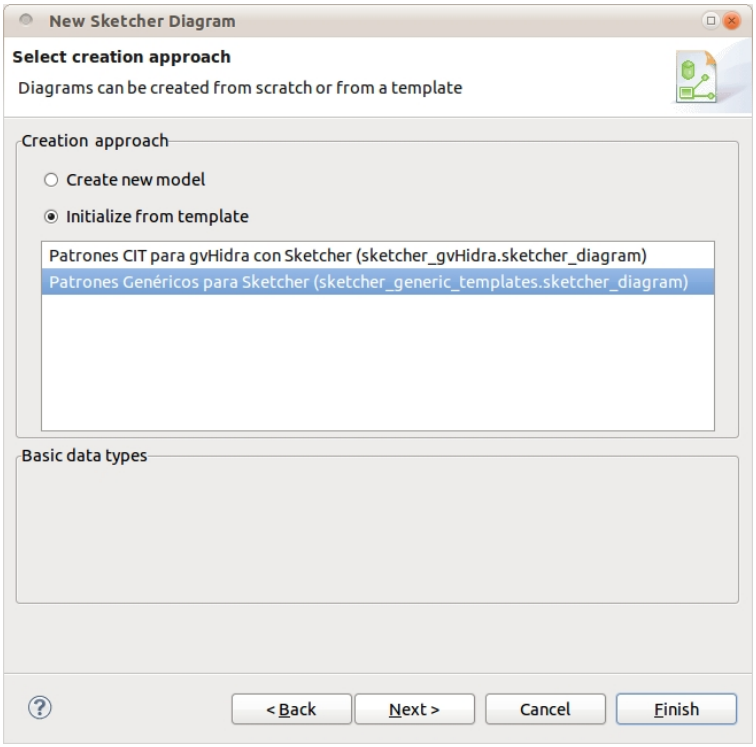
## Tarea 3.1.1: Creación de un nuevo modelo de Sketcher.

Para crear un nuevo modelo de Sketcher debemos seleccionar la opción `File/New.../MOSKitt Sketcher Diagram` la cual abrirá el asistente para la creación de modelos Sketcher para que le podamos asignar un nombre al modelo.


La especificación en el Sketcher se hace partiendo de componentes básicos web los cuales ya están modelados en MOSKitt y listos para ser utilizados por los analistas. A partir de estos componentes básicos pueden ser construidos componentes más complejos que van a representar a cada uno de los módulos (*portlets*) de los que se compone la aplicación OpenXava.

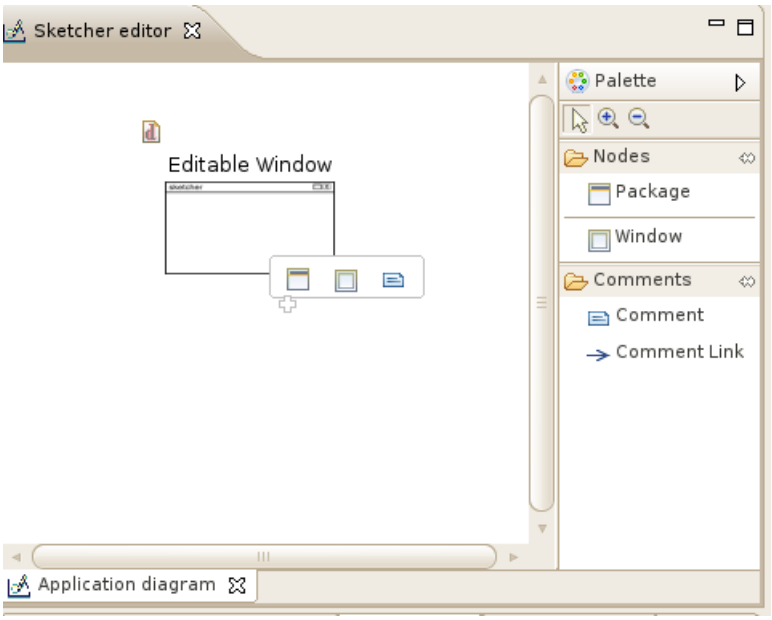
Estos componentes son bastante genéricos, además están implementados por OpenXava y son reconocibles por el generador. El hecho de que sean tan genéricos facilita que con ellos se pueda modelar cualquier aplicación web sea cual sea el framework destino y que en un futuro sean reutilizados para generar en cualquier otro framework diferente de OpenXava.

Para poder usar estos componentes en el Sketcher, al crear un nuevo diagrama (`File/New.../MOSKitt Sketcher Diagram`) se debe seleccionar la opción `Inicialize from template` y en concreto, de la lista que aparece en el asistente, la opción *`Patrones Genéricos para Sketcher (sketcher_generic_templates.sketcher_diagram)`* tal y como se muestra en la figura que hay a continuación:




Al finalizar el asistente, en el editor aparecerá un nuevo diagrama con un único elemento denominado `Editable Window` sobre el que podemos empezar a especificar el primer módulo de nuestra aplicación.

¡Ojo! El *nombre de la aplicación* se debe especificar en la propiedad 'Id' del elemento raíz (canvas) del modelo de Sketcher.




Para más información sobre las plantillas proporcionadas por MOSKitt para el modelado de aplicaciones interpretables por moskitt-codgen-openxava consultar el apartado Plantillas genéricas para el modelado de aplicaciones web.

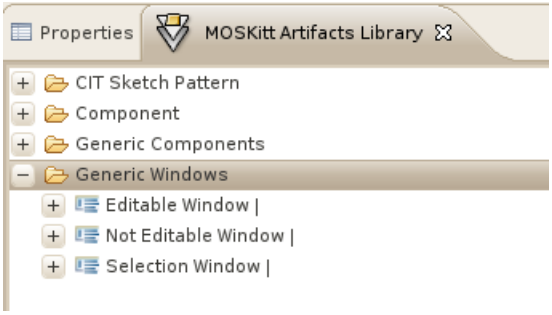
# Tarea 3.1.2: Creación de un nuevo módulo OpenXava para el mantenimiento de Facturas (Invoice)

En el modelo Sketcher un módulo en OpenXava está representado por un elemento de tipo `Window`. Para facilitar el modelado de los módulos OpenXava, MOSKitt incorpora ya dos tipos de módulos predefinidos representados por las plantillas:

- `Editable Window`.
- `Not Editable Window`.

Para más información sobre las plantillas que proporciona MOSKitt para el modelado de módulos OpenXava consultar el apartado Plantillas genéricas para el modelado de aplicaciones web localizado en las sección final de anexos.

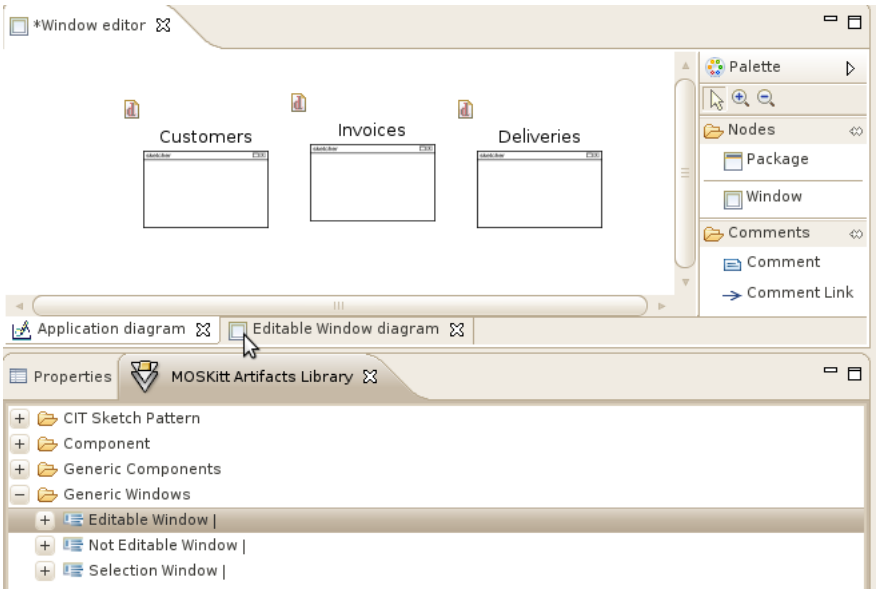
A continuación se muestra una imagen en la que se muestran las plantillas que proporciona MOSKitt para modelar módulos OpenXava:




Para crear un nuevo módulo sólo tenemos que abrir la vista `MOSKitt Artifacts Library` y arrastrar la plantilla que más nos interese al canvas inicial del Sketcher en el cual aparecerá una imagen representando a cada módulo de nuestra aplicación.

¡Ojo! La plantilla `Selection Window` representa a una ventana de selección y, aunque en Sketcher es un elemento `Window`, no está representando un módulo en OpenXava.

Si vamos arrastrando windows desde la carpeta `Generic Windows` de la vista `MOSKitt Artifacts Library` obtenemos el siguiente diagrama como resultado:



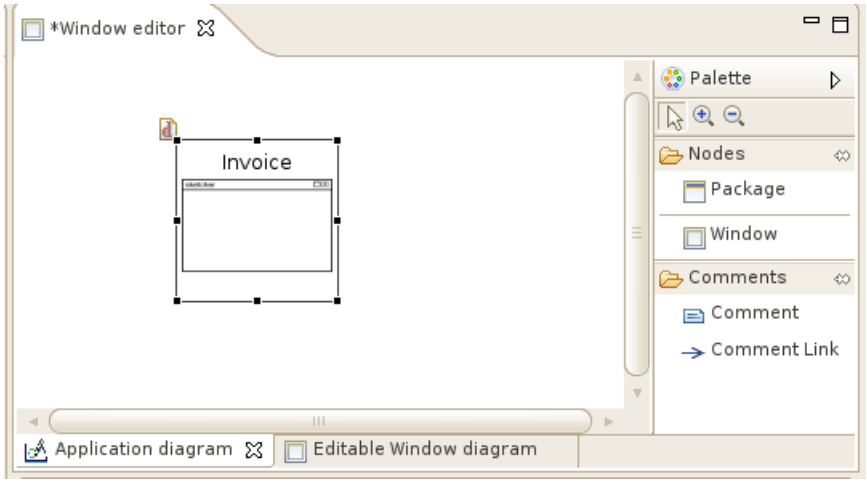

En el diagrama principal (inicialmente denominado `Application Diagram`) disponemos de una representación por cada módulo de la aplicación.

# Tarea 3.1.3: Modelado del módulo OpenXava para el mantenimiento de Facturas (Invoice)

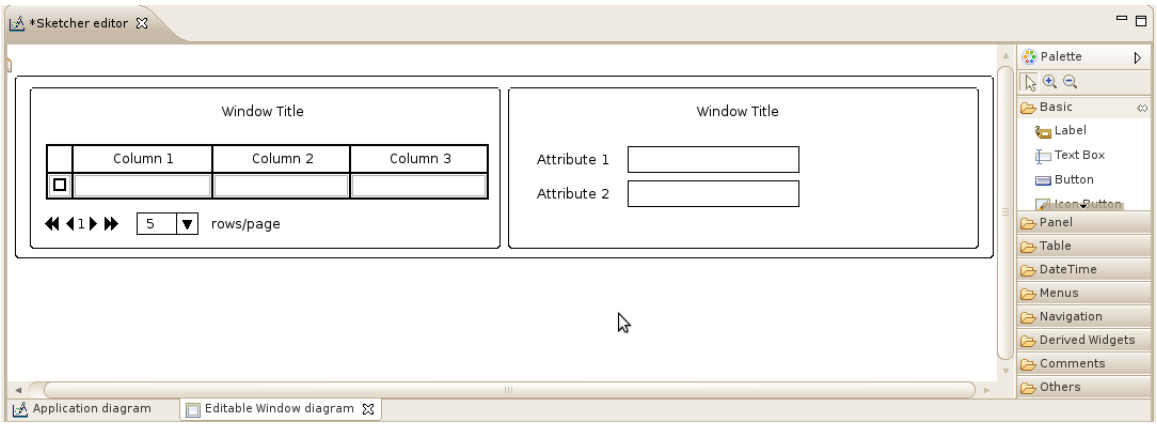
Vamos a modelar un primer módulo para el mantenimiento de Facturas (`Invoice`) en nuestra aplicación de Facturación. El módulo que vamos a modelar con Sketcher está basado en el que se puede consultar en la web de OpenXava en la dirección http://www.openxava.org/web/guest/invoice.

Puesto que nos interesa disponer de las operaciones básicas de mantenimiento (inserción, edición y borrado), haremos uso de la plantilla `Editable Window` (como es el primer módulo podemos aprovechar la `Editable Window` que nos proporciona Sketcher al crear un modelo nuevo).

Para editar la ventana debemos hacer doble click sobre la figura que representa al módulo en el diagrama principal (con el nombre por defecto de `Application diagram`):




Al hacer doble click se abrirá una nueva pestaña en el editor para poder modificar el contenido de la ventana tal y como se muestra en la figura que aparece a continuación:



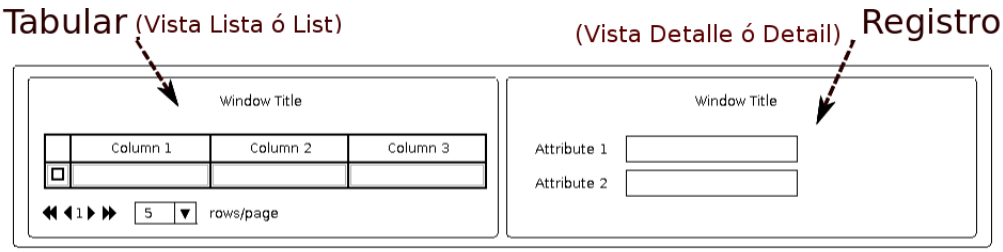

Como podemos ver en la figura anterior, no empezamos de cero, sino que partimos ya de un modelo inicial que nos proporciona la plantilla `Editable Window`. Ésta plantilla contiene dos elementos principales de tipo `Panel`:

- Un `Panel` que representa la vista en formato `Lista` (o `List`) del módulo.
- Un `Panel` que representa la vista en formato `Detalle` (o `Detail`) del módulo.

Para que el generador de código reconozca en estos paneles a las correspondientes vistas OpenXava, estos elementos `Panel` están etiquetados respectivamente con los tags:

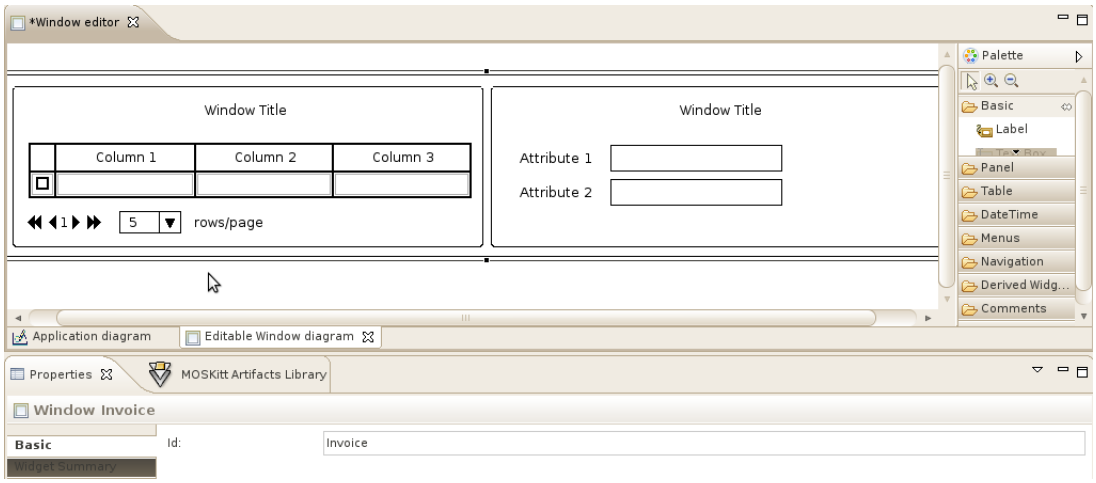
- `Tabular`: Tag utilizado para marcar la vista en formato `Lista` (`List`).
- `Registro`: Tag utilizado para marcar la vista en formato `Detalle` (`Detail`).

A continuación se muestra qué elemento gráfico está etiquetado con cada uno de estos tags:



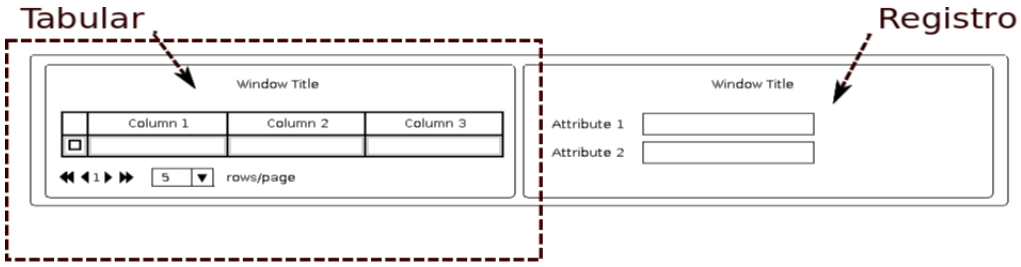

De esta ventana vamos a especificar la siguiente información:

- Propiedad *Id* de la `Window`: está localizada en la pestaña de propiedades, subpestaña `Basic` de la `Window`. En esta propiedad indicaremos el nombre el módulo, en este caso, `Invoice`.

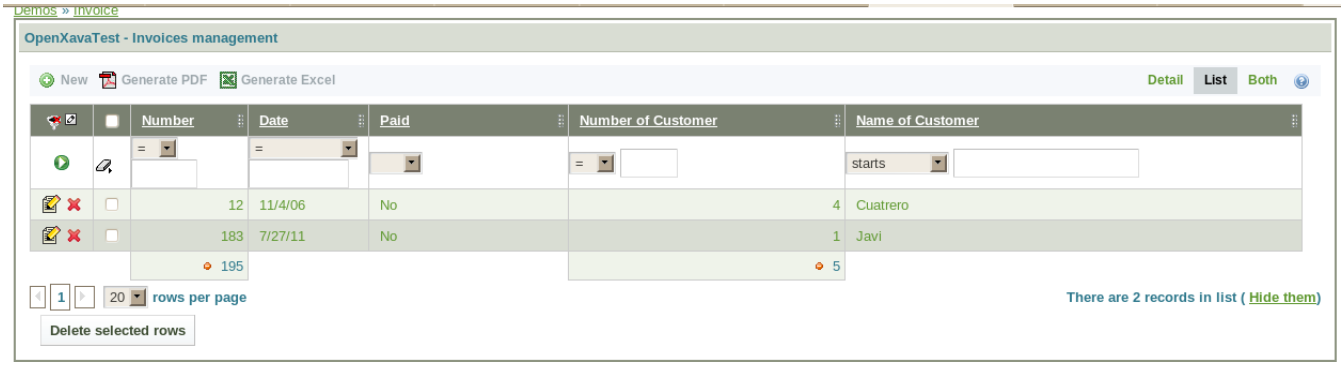



## Tarea 3.1.4: Modelar la vista Lista/List del mantenimiento de Facturas (Invoice)

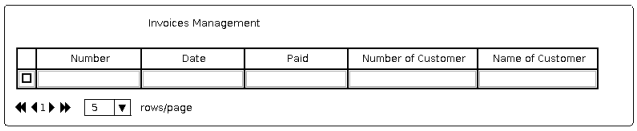
El siguiente paso es modelar modelar la vista en formato `Lista` (`List`) del módulo de mantenimiento de Facturas (`Invoice`). Esta vista se corresponde con uno de los componentes que forman parte de la plantilla `Editable Window`, el Panel etiquetado como `Tabular`:




Siguiendo el ejemplo de la página web de OpenXava, la ventana que vamos a modelar tiene el siguiente aspecto:



Haciendo uso de los elementos de la paleta que nos proporciona MOSKitt-Sketcher completamos el panel anterior para que tenga el siguiente aspecto:




Básicamente, lo único que hemos tenido que hacer es:

- Añadir 2 nuevas columnas a la tabla ya que la tabla que nos proporciona la plantilla tiene 3 columnas y nosotros necesitamos 5. (son elementos `Table Column` localizados en la paleta del Sketcher, en la pestaña `Table`).

- Indicar el título de la cabecera de cada una de las columnas de la tabla. Para hacer esto le hemos dado valor a la propiedad *Text* localizada en la subpestaña `Basic` de la pestaña de `Properties` del elemento `Table Column` correspondiente.

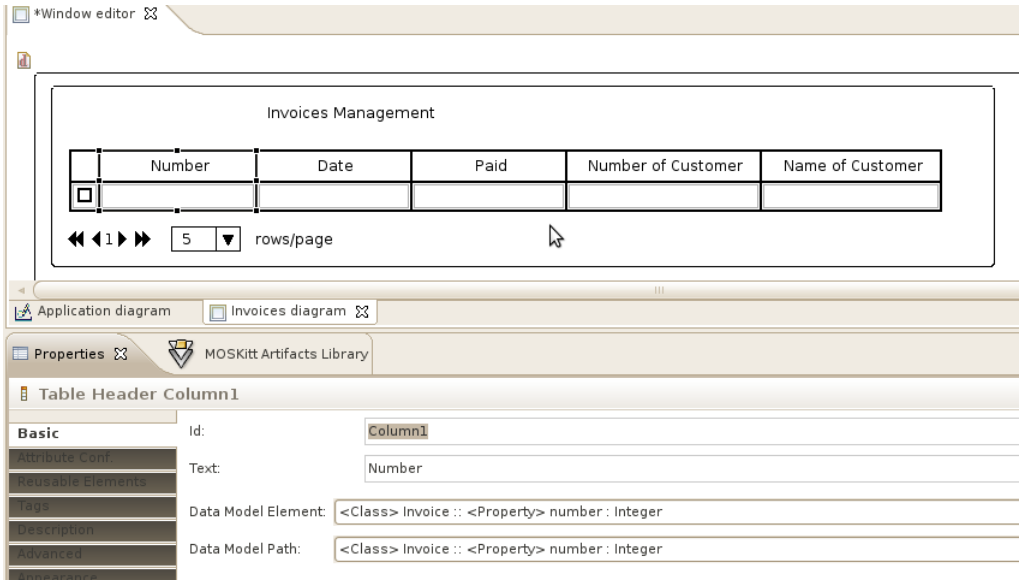
- Enlazar los widgets (elementos `Table Column` en este caso) con las correspondientes propiedades de UML2 completando las propiedades: *Data Model Element* y *Data Model Path*.

  - Enlazar las columnas etiquetadas como `Number`, `Date` y `Paid` con las propiedades `Number`, `Date` y Paid de la clase **`Invoice`**.

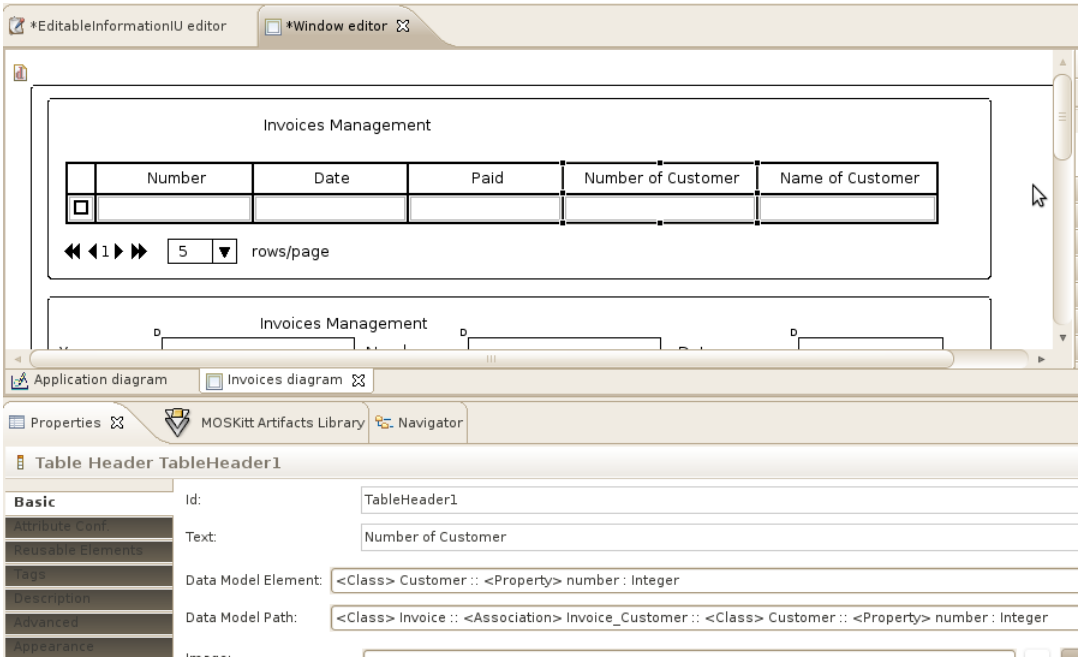
  - Enlazar las columnas etiquetadas como `Number of Customer` y `Name of Customer` con las propiedades `Number` y `Name` de la clase **`Customer`** a través de la asociación **`Invoice_Customer`** (¡ojo!, esto es importante).

    Para más información sobre cómo relacionar los modelos de Sketcher y UML2 consultar el apartado que aparece más adelante en este manual y que trata sobre Cómo relacionar elementos del Sketcher con elementos de UML2.

En la siguiente figura podemos ver cómo se han completado estas propiedades para la columna referente al Número de la Factura (`Number`):




En la siguiente figura podemos ver cómo se han completado estas propiedades para la columna referente al Número del Cliente de la factura (`Number of Customer`). Observar la diferencia con respecto a la propiedad *Data Model Element Path* de la figura anterior:

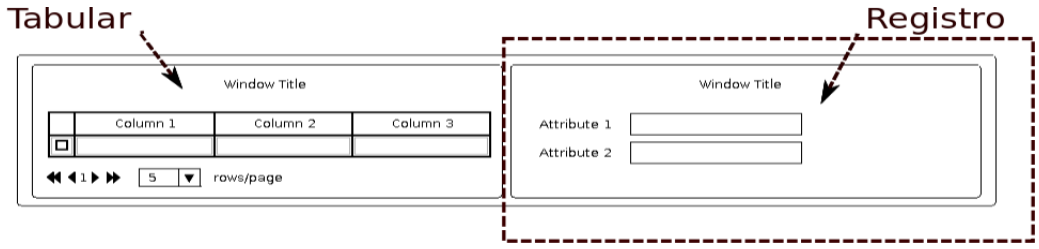



Para más información sobre cómo relacionar los modelos de Sketcher y UML2 consultar el apartado que aparece más adelante en este manual y que trata sobre Cómo relacionar elementos del Sketcher con elementos de UML2.

Para más información sobre cómo utilizar el editor MOSKitt-Sketcher consultar el Manual de Usuario en la ayuda de MOSKitt.

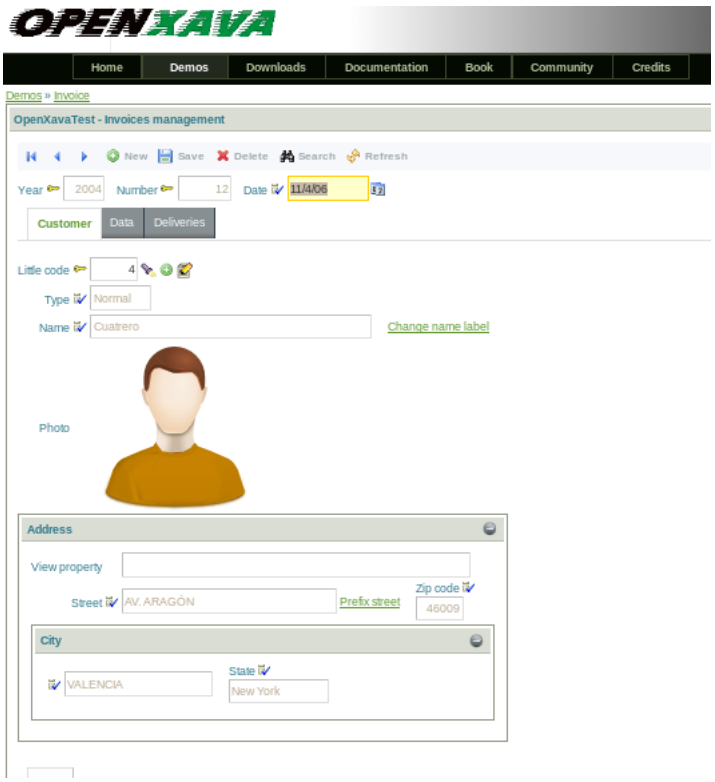
## Tarea 3.1.5: Modelar la vista Detalle/Detail del mantenimiento de Facturas (Invoice)

El siguiente paso es modelar modelar la vista en formato `Detale` (`Detail`) del módulo de mantenimiento de facturas (`Invoice`). Esta vista se corresponde con uno de los componentes que forman parte de la plantilla `Editable Window`, el `Panel` etiquetado como `Registro`:



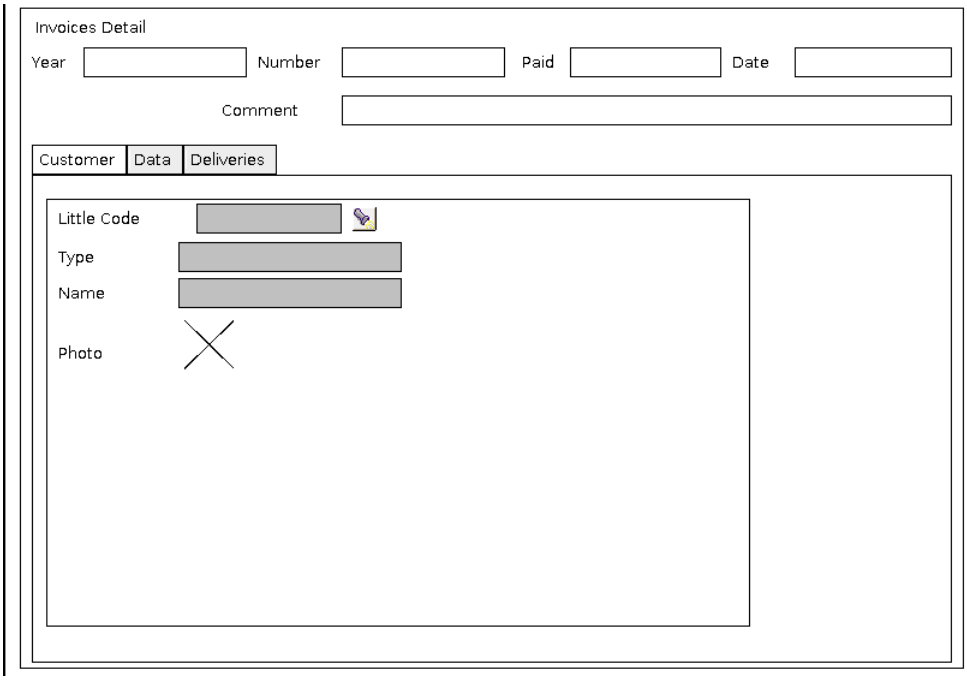

Siguiendo el ejemplo de la página web de OpenXava, la ventana que vamos a modelar es más compleja que la anterior ya que contiene varios paneles anidados, así es que vamos a ir por partes.

El primer panel que nos interesa modelar tiene el siguiente aspecto:




En él aparece una cabecera con datos de la factura y tres pestañas. En la primera de ellas, etiquetada como `Customer`, se muestra sólo información del cliente de la factura.

Haciendo uso de los elementos de la paleta que nos proporciona el Sketcher completamos el panel original para que tenga el siguiente aspecto:




En este caso ya hemos tenido un poco más de trabajo ya que hemos tenido que:

- Reubicar los `2 TextBox` que la plantilla proporcionaba por defecto para alinearlos en horizontal (se corresponderían con los campos `Year` y `Number`).
- Añadir un nuevo `TextBox` para el campo `Date`.

- Modificar la propiedad *Text* de cada una de las `Label` que acompañan a los `TexBox` para poner la etiqueta requerida en cada caso (`Year`, `Number` y `Date`).

- Enlazar todos los campos con las correspondientes propiedades de la clase **`Invoice`** del diagrama de UML2 (para más información sobre la relación entre los modelos de Sketcher y UML2 consultar el apartado sobre Cómo relacionar elementos del Sketcher con elementos de UML2). Para ello debemos completar las propiedades: *Data Model Element* y *Data Model Path*.

- Añadir un elemento `TabPanel` (panel de pestañas) con 3 `TabItems` (pestañas). Estos dos elementos se pueden arrastrar desde la subpestaña `Panel` de la paleta del Sketcher.

Para cada uno de los `TabItems` contenidos en esta vista hemos tenido que:

- Modificar la propiedad *Text* para poner la etiqueta requerida en cada caso (`Customer`, `Data` y `Deliveries`).

- Incluir los widgets necesarios.

Vamos a ir recorriendo cada uno de ellos para ver qué hemos tenido que hacer en cada caso para modelarlo:

- **`TabItem Customer`**: Esta pestaña muestra los datos del cliente de la factura tal y como podemos ver en la figura anterior. Para modelar esta pestaña hemos tenido que:

  - Incluir un componente `Single Selection Group Panel` arrastrándolo dede la carpeta `Generic Components` de la vista `MOSKitt Artifacts Library` (para más información sobre las plantillas que proporciona Sketcher para facilitar el modelado consultar la sección Plantillas genéricas para el modelado de aplicaciones web).

  - Añadir tres nuevos TextBox para los campos `type, name` y photo de la clase **Customer**.

  - Modificar la propiedad *Text* de cada `Label` que acompaña a los `TexBox` para poner la etiqueta requerida en cada caso (`Litle Code`, Type, Name y Photo).

  - Enlazar todos los campos con las propiedades `number`, `type, name` y photo de la clase **`Customer`** del diagrama de UML2 (para más información sobre la relación entre los modelos de Sketcher y UML2 consultar el apartado sobre Cómo relacionar elementos del Sketcher con elementos de UML2). Para ello debemos completar las propiedades: *Data Model Element* y *Data Model Path* de cada widget.

- **`TabItem Data`**: Esta pestaña contiene de nuevo dos pestañas, una para mostrar las líneas de la factura (`Details`) y otra para mostrar información de la factura relativa a los importes (`Amounts`). Vamos a ir recorriendo cada uno de los `TabItems` que aparecen en esta pestaña para ver qué hemos tenido que hacer en cada caso para modelarlo:

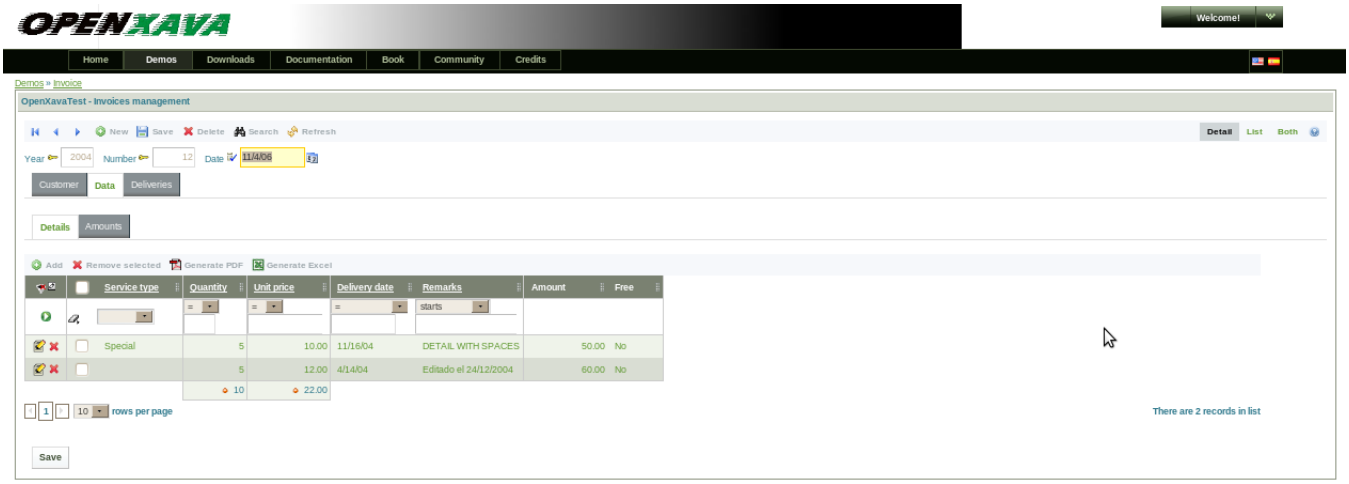
  - **`TabItem Details`**: Esta pestaña muestra las líneas de la factura en formato tabular tal y como podemos ver en la figurad el modelo de sketcher que aparece a continuación. Para modelar esta pestaña hemos tenido que:

    - Incluir un componente `Detail-Tabular Panel` arrastrando desde la carpeta `Generic Components` de la vista `MOSKitt Artifacts Library` (para más información sobre las plantillas que proporciona Sketcher para facilitar el modelado consultar la sección Plantillas genéricas para el modelado de aplicaciones web).

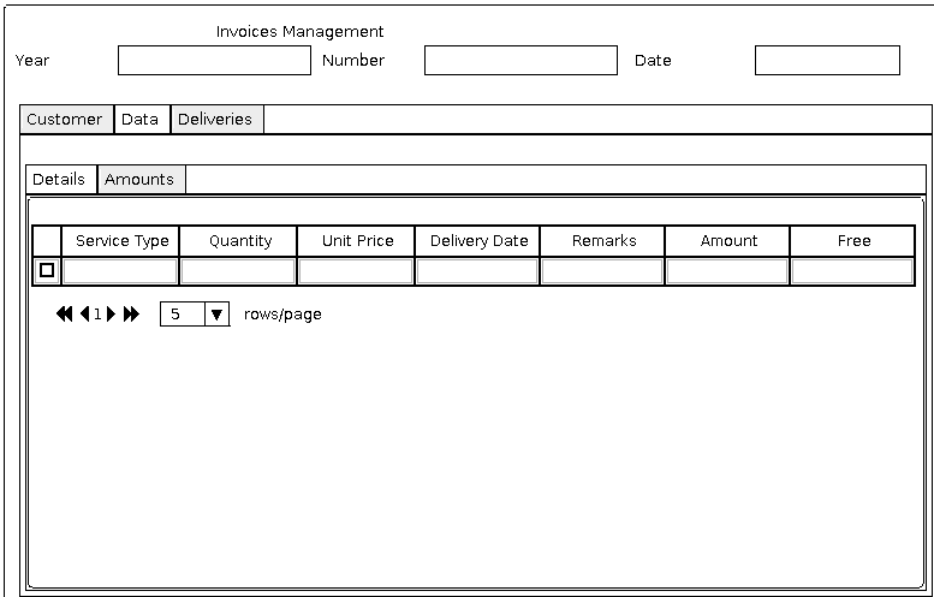
    - Añadir nuevas columnas a la tabla ya que la tabla que nos proporciona la plantilla tiene 4 columnas y nosotros necesitamos 7 (son elementos `Table Column` localizados en la pestaña `Table` de la paleta del Sketcher).

    - Indicar el título de la cabecera de cada una de las columnas de la tabla dando valor a la propiedad *Text* localizada en la subpestaña `Basic` de la pestaña de `Properties` de cada elemento `Table Column`.

- Enlazar todos los campos con las correspondientes propiedades de la clase **`Invoice_Detail`** del diagrama de UML2 (para más información sobre la relación entre los modelos de Sketcher y UML2 consultar el apartado sobre Cómo relacionar elementos del Sketcher con elementos de UML2). Para ello debemos completar las propiedades: *Data Model Element* y *Data Model Path* de cada widget.




Este formulario se modela en Sketcher tal y como aparece en la siguiente figura:




- **`TabItem Amounts`**: Esta pestaña muestra información relativa a los importes de la factura agrupada de nuevo en dos pestañas. En una pestaña se muestra información del IVA (`V.A.T`) y en la otra sumatorios de los importes (`Amounts Sum`) tal y como podemos ver en la figura del modelo de sketcher que aparece a continuación. Para modelar esta pestaña hemos tenido que:

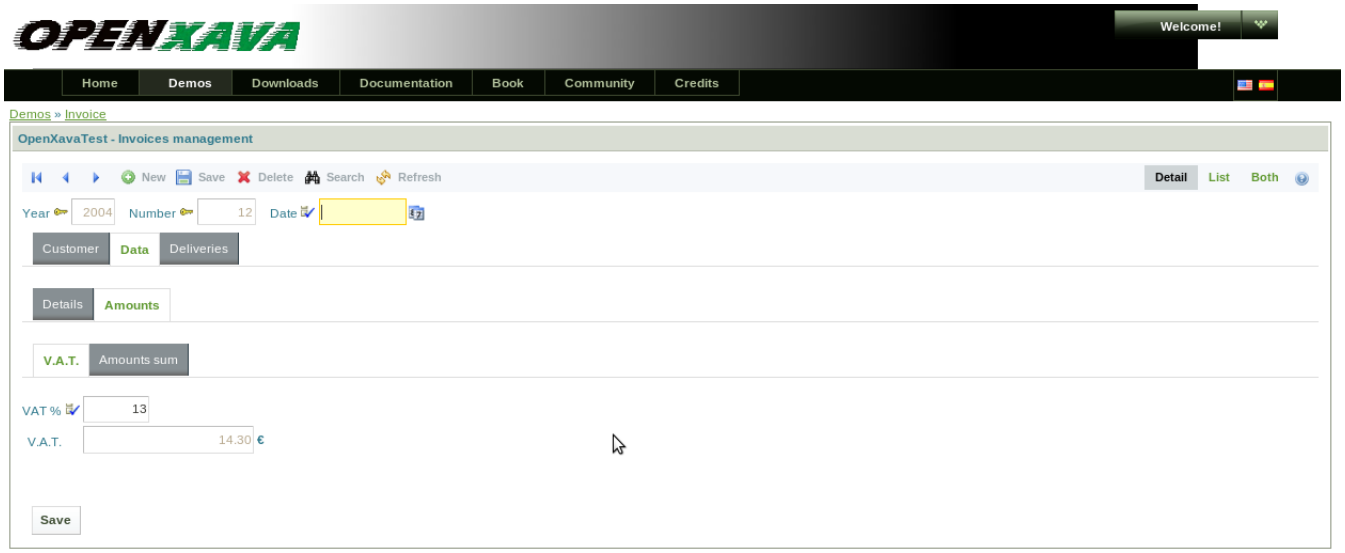
  - Añadir un elemento `TabPanel` (panel de pestañas) con 2 `TabItems` (pestañas). Estos dos elementos se pueden arrastrar desde la subpestaña `Panel` de la paleta del Sketcher.

  - Modificar la propiedad *Text* de cada pestaña para poner la etiqueta requerida en cada caso (`V.A.T.` y `Amounts Sum`).

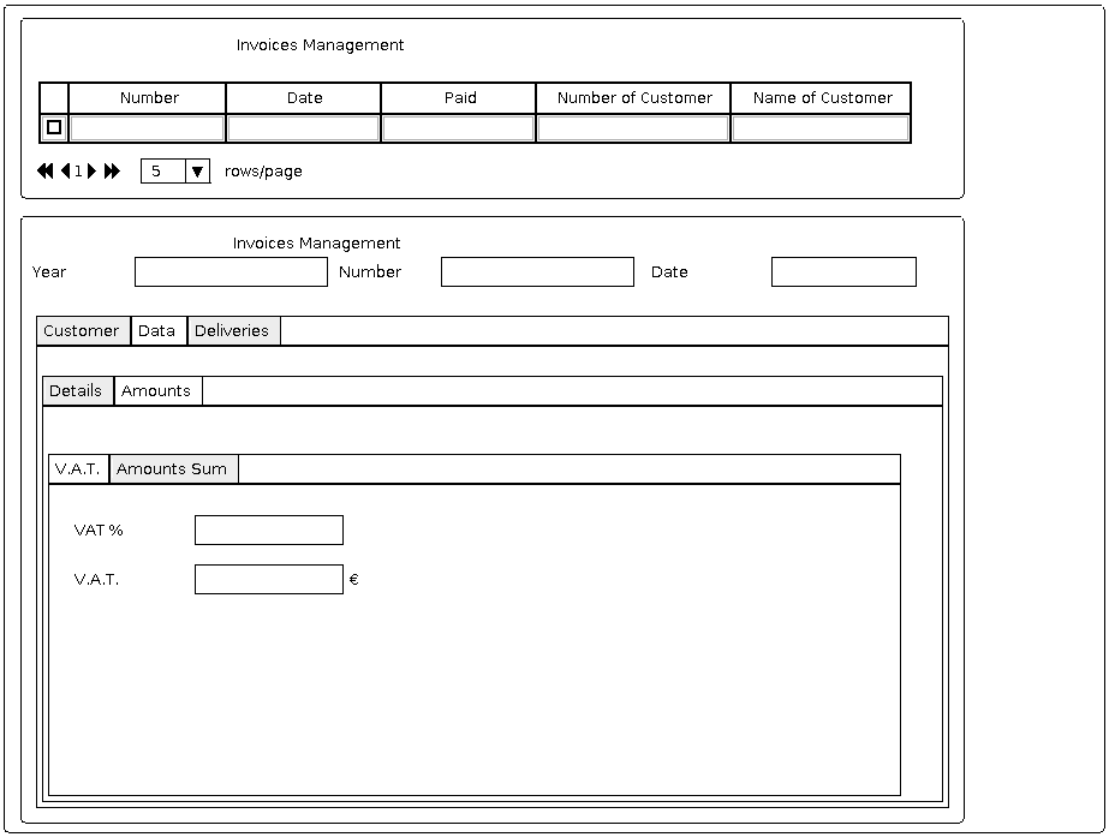
  - Incluir los widgets necesarios para completar cada una de las pestañas:

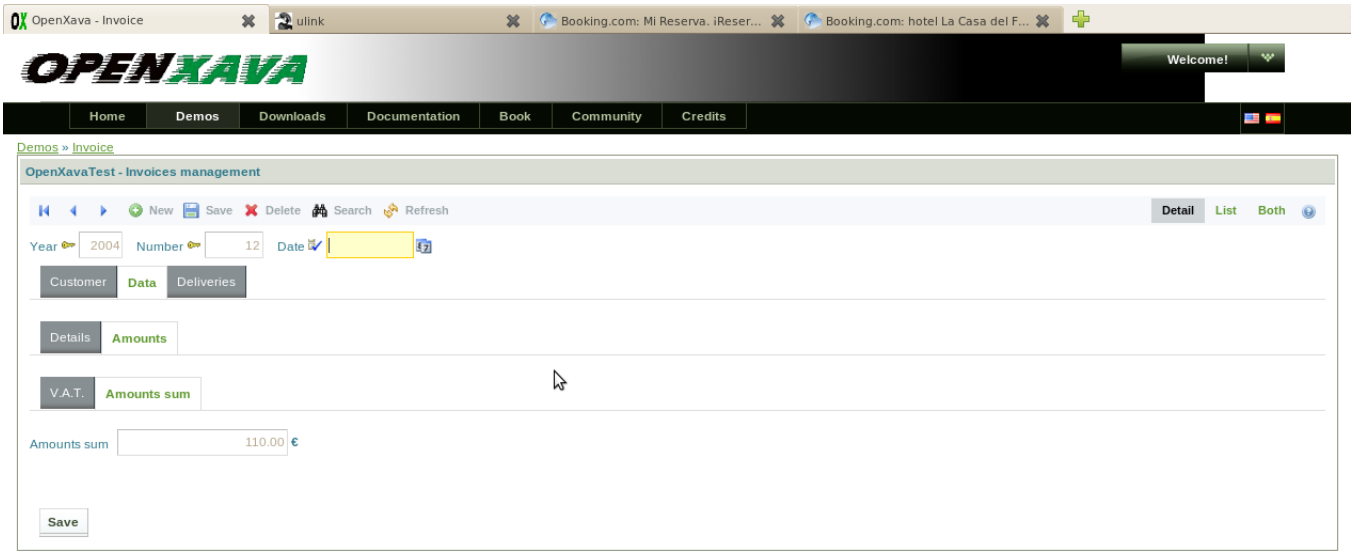
- En la pestaña V.A.T incluimos dos Text Box para las propiedades `vatPercentaje` y `/vat` de la clase **Invoice**.

- Enlazar todos los campos con las correspondientes propiedades de la clase **`Invoice`** del diagrama de UML2 (para más información sobre la relación entre los modelos de Sketcher y UML2 consultar el apartado sobre Cómo relacionar elementos del Sketcher con elementos de UML2). Para ello debemos completar las propiedades: *Data Model Element* y *Data Model Path* de cada widget.

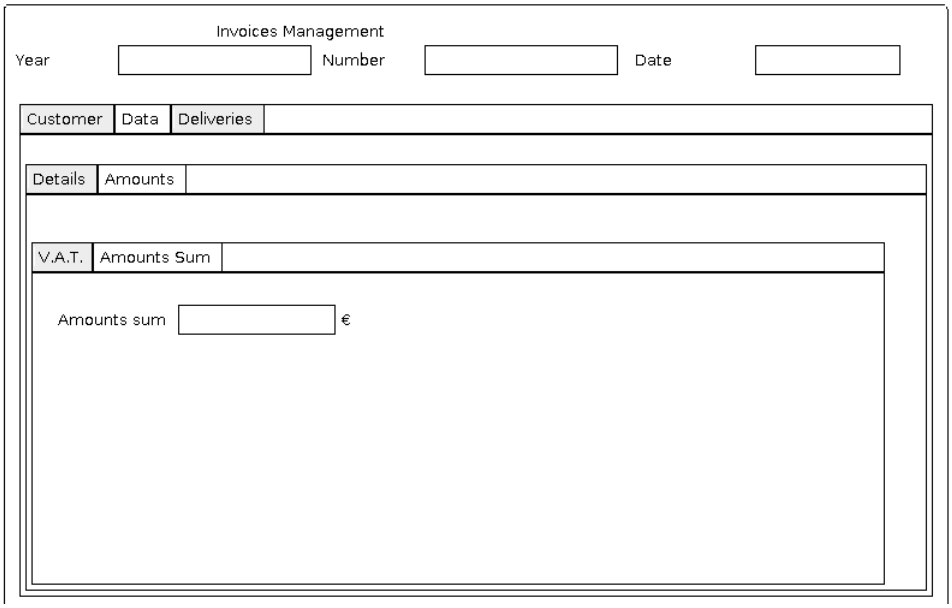



Este formulario se modela en el Sketcher tal y como aparece en la siguiente figura:






Este formulario se modela en el Sketcher tal y como aparece en la siguiente figura:

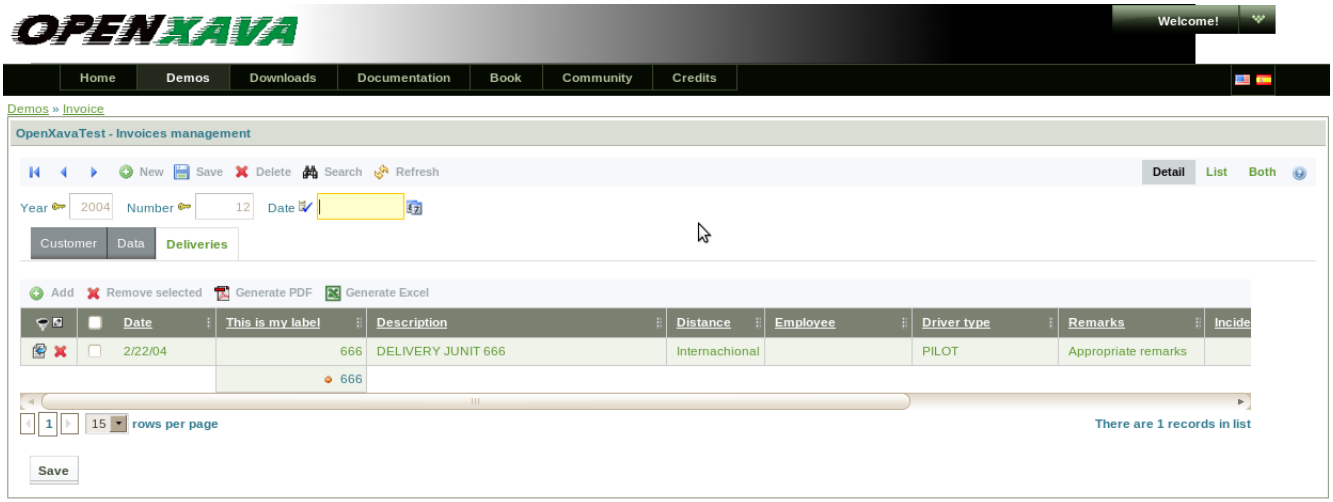



- **`TabItem Deliveries`**: Esta pestaña muestra la lista de albaranes asociados con la factura. No permite insertar nuevos albaranes, sólo relacionar albaranes existentes con la factura que tenemos activa en el módulo. Para modelar esta pestaña hemos tenido que:

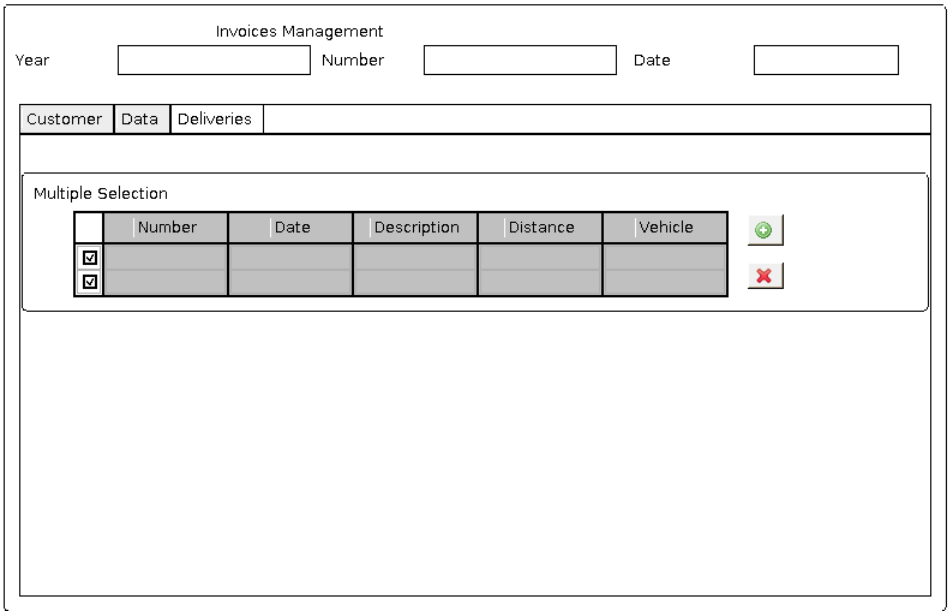
  - Incluir un componente `Multiple Selection Panel` localizado en la carpeta `Generic Components` de la vista `MOSKitt Artifacts Library` (para más información sobre las plantillas que proporciona Sketcher para facilitar el modelado consultar la sección Plantillas genéricas para el modelado de aplicaciones web).

  - Completar el componente del mismo modo que hemos procedido hasta ahora con el resto de paneles con los widgets correspondientes.

  - Enlazar todos los campos con las correspondientes propiedades de la clase **`Delivery`** a través de la relación `Invoicing_Delivery` del diagrama de UML2 (para más información sobre la relación entre los modelos de Sketcher y UML2 consultar el apartado sobre Cómo relacionar elementos del Sketcher con elementos de UML2). Para ello debemos completar las propiedades: *Data Model Element* y *Data Model Path* de cada widget.



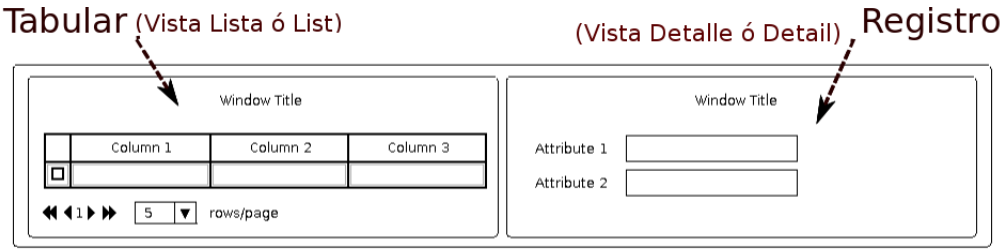

Este formulario se modela en el Sketcher tal y como aparece en la siguiente figura:



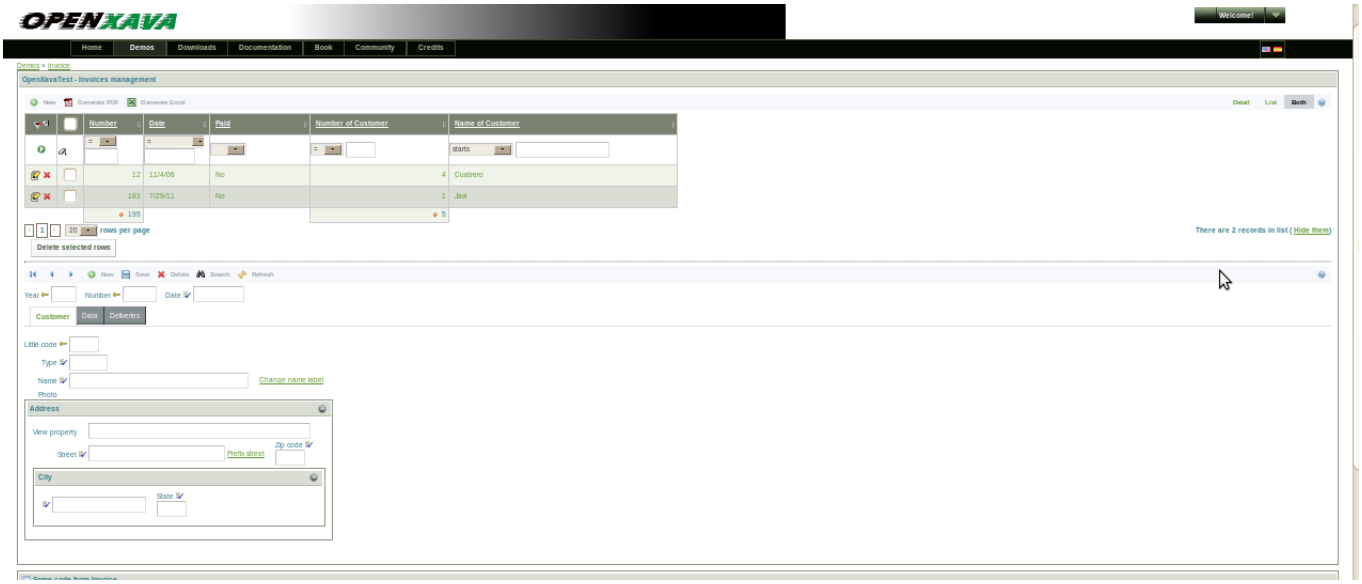

Para más información sobre cómo utilizar el editor MOSKitt-Sketcher consultar el Manual de Usuario en la ayuda de MOSKitt proporcionado por este módulo.

## Tarea 3.1: Modelar la vista Both del mantenimiento de Facturas (Invoice)

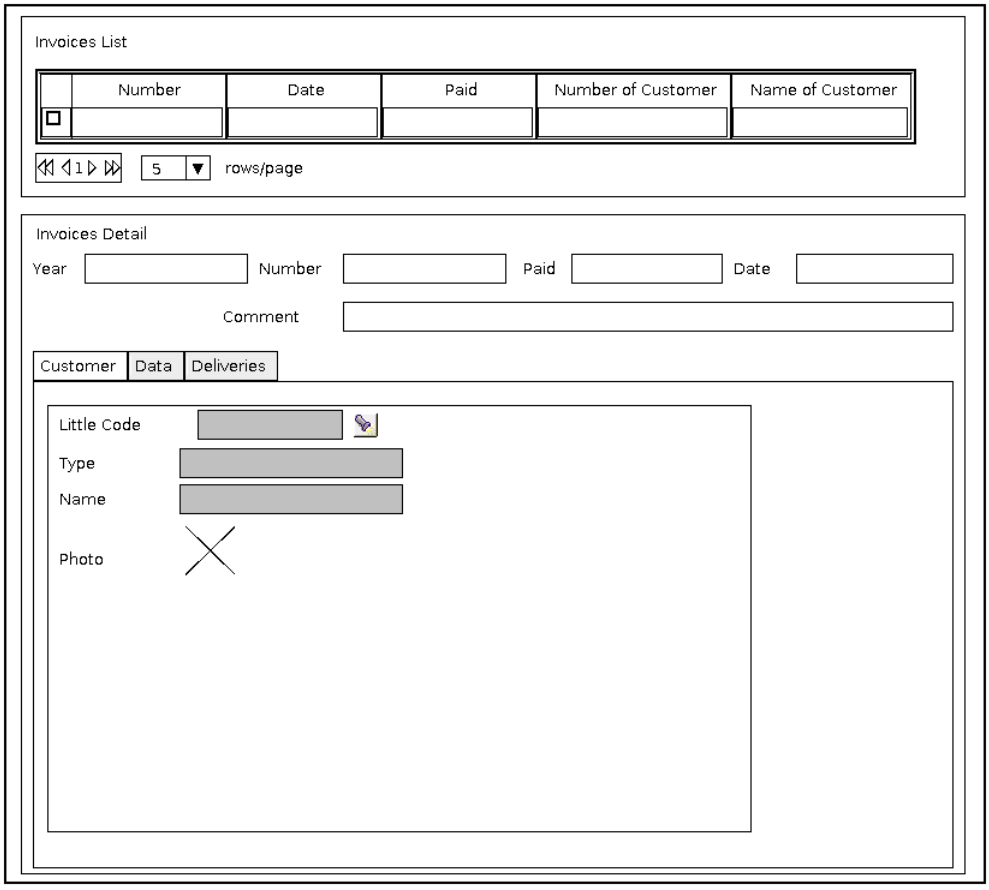
Esta vista no es necesario modelarla ya que lo hemos ido haciendo mientras hemos ido modelando cada una de las dos vistas del módulo: `Tabular` y `Registro`.




Siguiendo el ejemplo de la página web de OpenXava, la ventana que hemos vamos a modelar tiene el siguiente aspecto:



Haciendo uso de los elementos de la paleta que nos proporciona el Sketcher hemos ido completanto los paneles de Lista y Detalle y al final el componente completo tiene el siguiente aspecto:.




## Tarea 3.1: Cómo relacionar los elementos del Sketcher con los del Modelo UML2

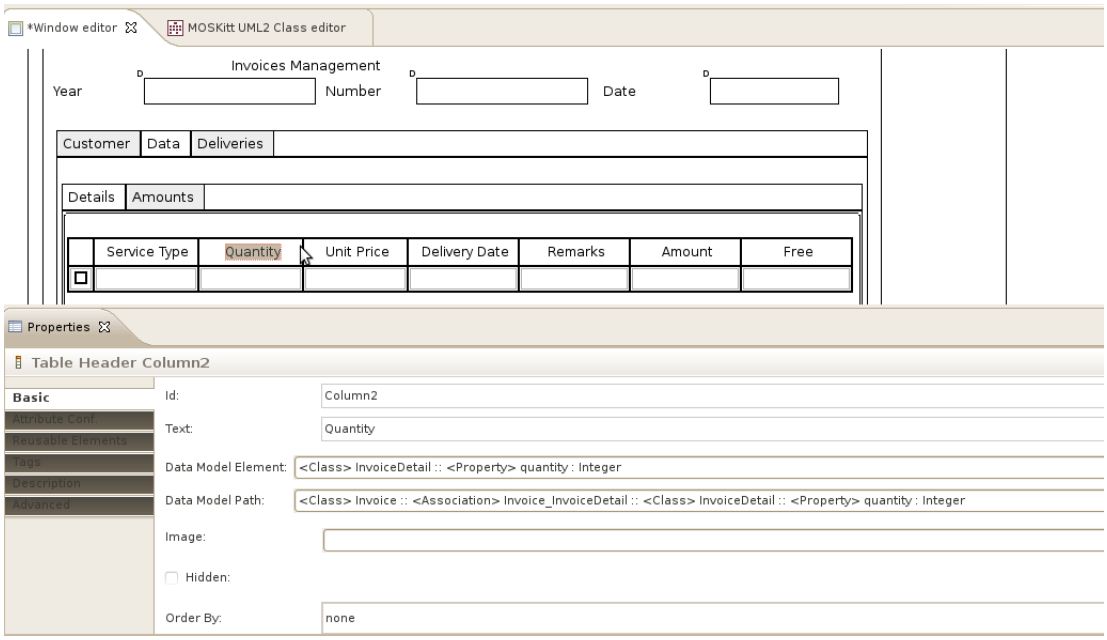
Para completar la especificación de la interfaz necesitamos enlazar muchos de los widgets con elementos del modelo de clases. De este modo el analista podrá indicar:

- Qué propiedades de las clases se deben mostrar en los widgets de la interfaz y si se deben poder editar o no.
- Qué métodos de las clases pueden ser invocados desde la interfaz (normalmente pulsando un botón).
- Qué enumerados deben ser ofrecidos en una lista de valores
- etc...

Para tener acceso a los elementos del modelo de UML2 en primer lugar tenemos que cargar este modelo como un nuevo recurso para el Sketcher. Para ello, debemos hacer click con el botón derecho sobre el lienzo del Sketcher y seleccionar la opción `Load Resource` del menú contextual.

En este momento aparecerá el siguiente asistente para cargar el modelo y, haciendo uso del botón `Browse Workspace...` debemos buscar el modelo de clases `invoice_domain.uml` (en nuestro caso de estudio) y finalmente pulsaremos `OK`.


A partir de este momento todos los elementos del modelo UML2 serán accesibles desde el modelo de Sketcher de la aplicación de facturación y podremos completar las propiedades necesarias de los elementos de cada panel.



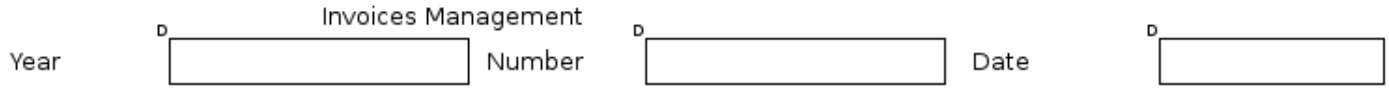

En la imagen anterior se muestran las propiedades de un campo de texto. Para cada elemento debemos dar valor a las siguientes propiedades:

- *Id*: Identificador del widget (normalmente por claridad se indica el nombre de la propiedad de UML2 con la que se relaciona).
- *Elemento Modelo*: muestra con qué elemento del modelo de UML2 se relaciona el widget.
- *Ruta Elemento Modelo*: indica cómo se debe navegar en el modelo UML2 para alcanzar la información a mostrar. Vamos a ver su utilidad con un ejemplo:

- Si selecciono la propiedad `Quantity` de la clase `Invoice_Detail` estaremos indicando que en la interfaz se van a mostrar todas las líneas de todas las facturas y, para cada una de ellas su propiedad `Quantity` .

- Sin embargo, si selecciono la misma propiedad, pero navegando desde la clase `Invoice` (a través de la asociación que tengo definida `Invoice_InvoiceDetail`), estaré indicando que lo que voy a mostrar es sólo la propidedad `Quantity` de las líneas de cada factura.

En la siguiente figura podemos observar en el extremo superior izquierdo de los campos de texto que aparece el símbolo **D** para indicar que ya está definida la relación con el modelo UML2 (modelo de Datos). Para saber con qué elemento de UML2 se ha relacionado el widget debemos ir a la pestaña de propiedades y consultar las propiedades *Elemento Modelo* y *Ruta Elemento Modelo* descritas anteriormente.




Para más información sobre todos los elementos disponibles en la paleta del Sketcher y sus propiedades consultar el manual de usuario del Sketcher en la ayuda de MOSKitt.

# Tarea 3.2: Modelo UIM (User Interface Model)

Para generar el código OpenXava con las anotaciones relativas a los módulos y a JPA (JavaGen), el generador de código tiene en cuenta la información de los siguientes modelos abstractos:

- El modelo UML2 referenciado desde el Sketcher.

- El modelo Sketcher

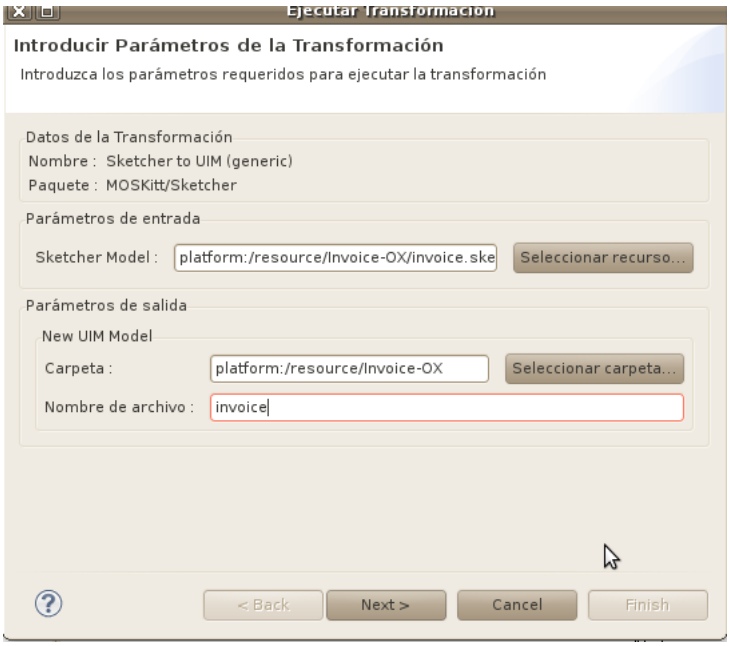
- El modelo UIM.

Por el momento disponemos de los dos primeros y necesitamos disponer también del Modelo UIM (User Interface Model). Vamos a ver en este apartado cómo obtenerlo.

## Tarea 3.2: Obtener el modelo UIM: Transformación Sketcher2UIM

En la mayoría de los casos el analista sólo definirá el modelo del Sketcher y el de UIM lo generará de forma automática lanzando una transformación específica para ello: Sketcher2UIM (generic).

Para hacerlo nos tenemos que situar sobre el modelo de Sketcher (`invoice.sketcher_diagram`) y en el menú contextual seleccionar el submenú MOSKitt *Transformations/Sketcher to UIM (generic)* (para más información sobre la transformación ver Manual de Usuario en la ayuda de MOSKitt). MOSkitt lanzará el asistente que aparece en la siguiente figura para lanzar la transformación:




Al completar la transformación aparecen tres archivos nuevos en el proyecto, el modelo y diagrama de UIM, y el modelo de trazas entre UIM y Sketcher.

Recursos de entrada de la tarea:

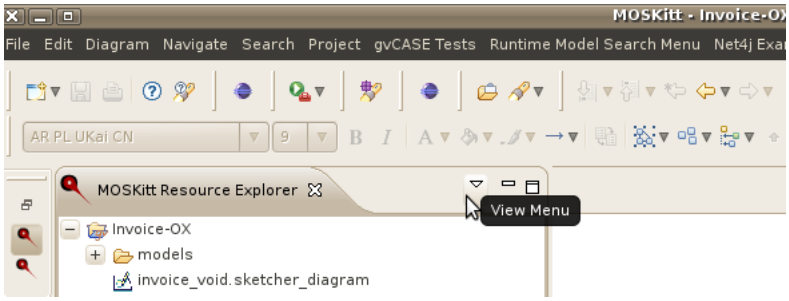
- Modelo de Sketcher (`invoice.sketcher_diagram`).
- Modelo de UML2 (`invoice_domain.uml`). Es necesario que este modelo esté en el proyecto aunque no sea un parámetro de entrada que haya que indicar en el Asistente. MOSKitt lo obtiene a partir de las referencias que explícitamente ha definido el Analista desde el Sketcher hacia él.

Recursos de salida de la tarea:

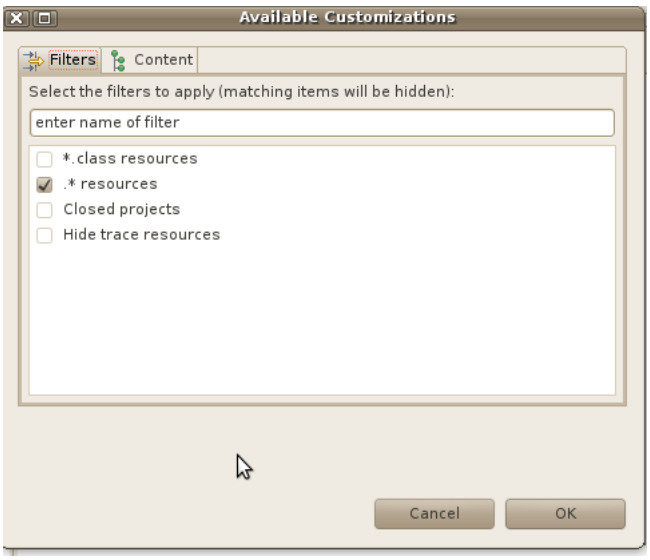
- Modelo de UIM (`invoice.uim`).
- Diagrama de UIM (`invoice.uim_diagram`).
- Modelo de trazas entre el modelo de Sketcher y el modelo de UIM generado (`invoice!_!invoice!-!sketcher2uim.gvtrace`).

**NOTA:**

Si no es visible el modelo de trazas en la vista `MOSKitt Resource Explorer` es porque está activado el filtro para que este tipo de recursos no se muestren en ella. Para hacerlos visibles hay que acceder al `View Menu` de esta vista:




Desde el desmarcar el check para indicar que se deben ocultar estos ficheros (opción Hide trace resources ):




En el siguiente apartado se va a explicar el por qué de la existencia del modelo UIM y su relación con el modelo Sketcher en la especificación de la Interfaz de Usuario.

## Tarea 3.3: Completar el modelo UIM

En este apartado vamos a exponer la importancia del Modelo UIM en el marco de la definición de interfaces de usuario. Si aún estáis viendo por primera vez MOSKitt no es necesario que entréis con detalle a conocer este modelo.

De hecho, para seguir este tutorial no necesitáis leer esta sección, no obstante os recomdendamos que más adelante, cuando estéis más familiarizados con el proceso propuesto por MOSKitt volvais a aquí y seguramente entonces veáis que os aclara muchas cosas respecto a Sketcher/UIM.

Antes de empezar vamos a hacer algunas puntualizaciones sobre cómo MOSKitt propone que se aborde el uso de ambos modelos:

- Cuando un usuario empiece a utilizar MOSKitt es muy probable que sólo haga uso del Sketcher pues la definición de interfaces de usuario con este módulo es mucho má intuitiva (pinta las ventanas tal y como espera que se visualicen después). Sin embargo, es importante entender qué limitaciones plantea el Sketcher a la hora de definir las pantallas/formularios de las aplicaciones y cómo UIM resuelve en la mayoría de los casos estos problemas de expresividad.
- De todos modos, a pesar de que MOSKitt-Sketcher nació como un pintador de interfaces de usuario, se ha enriquecido su expresividad para que el uso de UIM no sea imperativo en la definición de interfaces (al menos que no lo perciba así el usuario). La idea es:

  *Con UIM lo puedes expresar casi todo, pero usar Sketcher es más fácil. Vamos a intentar llegar a un porcentaje bastante elevado en la definición de las interfaces con el sketcher y dejamos UIM para los casos más complejos.*

  - Al principio de la implantación de MOSKitt, los analistas prefieren completar el modelo de Sketcher con comentarios y anotaciones. Recomendamos que, de decidir hacerlo así, estas notas se incluyan en el campo `Descripción` de los elementos del Sketcher (disponible en la pestaña de propiedades) siguiendo unas reglas comunes a toda la organización.
  - Sin embargo, a medida los usuarios vayan conociendo la herramienta llegará un momento en el que el uso de UIM no les resulte tan costoso como al principio y se animen a abordar el completar las interfaces con este lenguaje.

UIM no es ambigüo como sucede con cualquier descripción textual, por tanto UIM puede ser procesado con el generador y, por este motivo, todo aquello especificado en UIM es susceptible de ser incorporado antes o después en la generación de código.

# Diferencias entre Sketcher y UIM

Os hemos explicado que es mejor incorporar UIM cuando los usuarios estén más familiarizados con MOSKitt. Ahora os queremos aclarar qué diferencias hay entre ambos modelos:

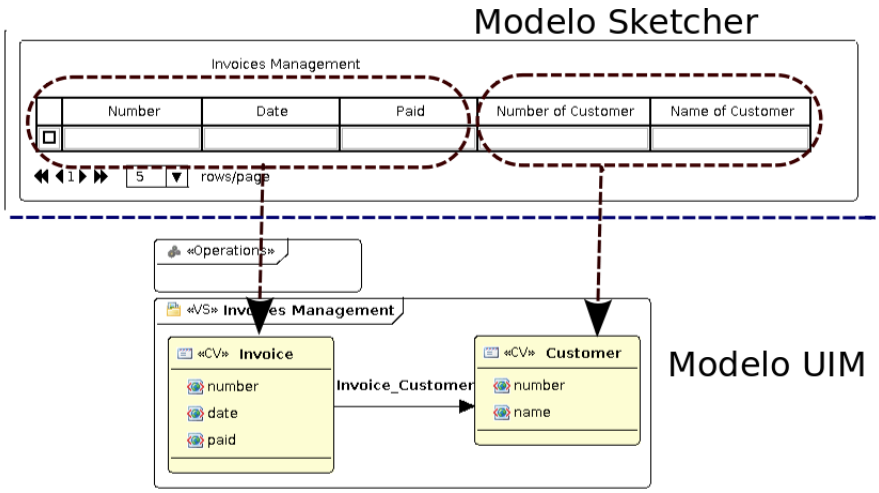
UIM permite definir una interfaz desde un punto de vista abstracto, por eso las figuras que se usan para representar los elementos de la interfaz no aportan ninguna información sobre cual debe ser su apariencia en la interfaz de la aplicación final.

Un diagrama de UIM es similar a un diagrama de clases y en él podemos ver sobre todo "cajas" etiquetadas que contienen elementos y se relacionan unas "cajas" con otras. Una interfaz especificada con UIM se puede representar de muy diversas formas/aspectos, o lo que es lo mismo, puede implementarse en diferentes plataformas (un cliente ligero, un cliente rico, un móvil etc...).

Por el contrario, con Sketcher podemos especificar una interfaz de forma muy concreta. Esto significa que cuando definimos una interfaz con Sketcher ya estamos tomando decisiones sobre qué aspecto debe tener y qué comportamiento para que se ajuste a una plataforma concreta: ya estamos pensando en un navegador, en un teléfono móvil, en un cliente rico etc..... Estamos decidiendo qué widgets va a contener mi interfaz y dónde los voy a colocar y además estoy obteniendo una representación bastante fiel de cómo va a ser mi interfaz definitiva.

Por todo esto, especificar interfaces con el Sketcher resulta muy intuitivo y los usuarios casi sin consultar el manual se sienten cómodos haciendo esta parte del trabajo.

En la figura que aparece a continuación podemos ver el aspecto que tiene una interfaz descrita con UIM y con Sketcher. La interfaz es la correspondiente a la vista `Lista`/`List` de un modulo OpenXava:




Sin embargo, una vez hayamos terminado de "pintar" la interfaz nos daremos cuenta de que nos hace falta decir algunas cosas más. Es decir, si como analistas, le pasamos una interfaz "pintada" con el Sketcher a uno de nuestros desarrolladores, estamos seguros de que les tendremos que dar más información para que puedan desarrollar de esa pantalla o informe. Algunas de las cosas que tendremos que decirles son:

- ¿Esta interfaz se ajusta a alguna plantilla?
- Cada text box, ¿qué propiedad de qué clase tiene que mostrar y/o permitir editar?.
- ¿Todos los campos son siempre editables?.
- Un botón de la interfaz ¿a qué metodo de negocio debe invocar? ¿cuales son sus parámetros de entrada? ¿dónde debe mostrar el resultado si tiene parámetros de salida?
- Seleccionar un determinado valor en una lista ¿tiene algún comportamiento asociado en la interfaz?

- etc...

Bueno, pues estos son los tipos de cosas que puedo decir con UIM y que habitualmente no se puede decir en un simple "pintador".

No obstante, Sketcher no es un simple pintador sino que ha sido diseñado para que haga de puente entre el usuario y UIM.

¿Cómo?: permitiéndo especificar desde Sketcher algunas de estas características más propias de UIM, para facilitar de este modo al analista el uso de UIM.

¿Cuál es la información propia de UIM que ya puedo adelantar desde el Sketcher?. Principalmente la siguiente:

- A qué plantilla se debe ajustar mi interfaz.
- El valor de qué propiedades muestro en los widgets y debo permitir editarlos.
- Qué métodos de negocio invoco desde la interfaz (normalmente al pulsar un botón).
- Qué widgets son editables/visibles/obligatorios siempre.
- Qué widgets son editables/visibles/obligatorios para cada operación (en la inserción, borrado, edición etc...).
- Navegación entre ventanas.

Entonces.... ¿para qué quiero UIM?. UIM es necesario porque es quien realmente aporta todos los conceptos para especificar una interfaz de usuario: su estructura y su comportamiento. Aunque mucha de la información necesaria ya se ha dicho en Sketcher, no es suficiente.

Aún así, podemos generar aplicaciones sin que el usuario tenga que llegar a abrir un modelo UIM.

| | |
|---|---|
| Recursos de entrada de la tarea: | • Modelo de UIM (`invoice.uim`).<br>• Diagrama de UIM (`invoice.uim_diagram`). |
| Recursos de salida de la tarea: | Como resultado de la tarea se obtienen (si procede) los mismos recursos pero actualizados por el usuario:<br>• Modelo de UIM (`invoice.uim`).<br>• Diagrama de UIM (`invoice.uim_diagram`). |

# Tarea 4: Generar el código OpenXava

Una vez hemos modelado la aplicación ya podemos generar el código con las anotaciones relativas a:

- La persistencia de las entidades definidas en el modelo UML2 según el estándar JPA.
- La interfaz de usuario que nos va a mostrar la información de estas mismas entidades según OpenXava.

Para ello, seguiremos los siguientes pasos:

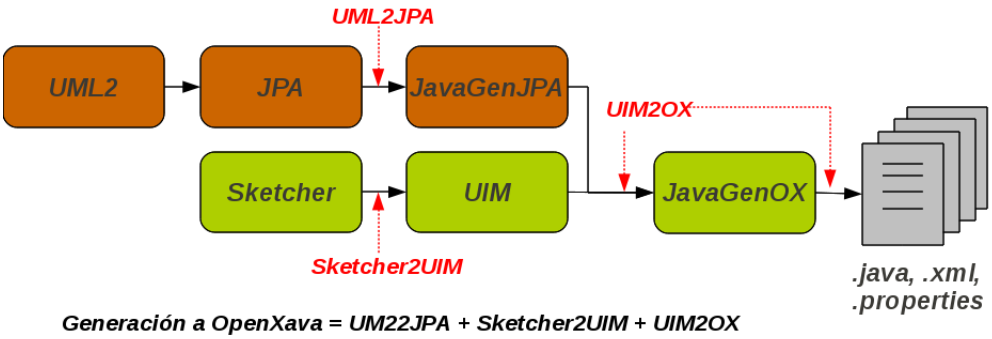
1. Paso 1: Generar las clases Java con las anotaciones JPA para la persistencia de las entidades UML2.

   Para ello lanzaremos la transformación UML2JPA a partir del modelo UML2 (`invoice_domain.uml`).

2. Paso 2 [Opcional]: Generar las clases Java con anotaciones OpenXava y JPA.

   Para ello lanzaremos la transformación UIM2OpenXava a partir del modelo UIM (`invoice.uim`).

En ambos casos se tienen en cuenta más modelos, tal y como se muestra en la figura que recordamos ahora:




***Generación a OpenXava = UM22JPA + Sketcher2UIM + UIM2OX***
***(Cadena de Transformaciones)***

A continuación vamos a ver con un poco más de detalle estos dos pasos.

## Creación y configuración de un proyecto nuevo OpenXAVA en MOSKitt

Vamos a ver qué pasos tenemos que dar para crear un proyecto nuevo basado en OpenXAVA y que permita ser empaquetado, desplegado y ejecutado en el portal Liferay de la distribución.

1. Abrir MOSKitt y apuntar el workspace al directorio *oxportal/workspace*

2. Aunque no es necesario se aconseja cambiar a la perspectiva <Java EE>

3. Ejecutar la acción de menú <Run>.<External Tool>.<New OpenXava Project>

4. Introducir el nombre del proyecto en el cuadro de diálogo que aparece (¡Ojo! Este nombre deberá ser igual que el valor de la propiedad 'Id' del elemento Sketcher del modelo de Sketcher o la propiedad 'nombre' del elemento UserInterfaceModel del modelo de UIM).

5. Ya tenemos creado nuestro proyecto en el disco, pero no lo tenemos accesible desde la vista de proyectos. Para hacerlo accesible hacemos lo siguiente:

   a. Ejecutar la acción del menú <File>.<Import>

   b. Seleccionar "Existing Projects into Workspace" y pulsar <Next>

   c. Seleccionar en la opción "Select root directory" la ruta del workspace: *oxportal/workspace*

   d. En la lista de proyectos nos aparecerá nuestro nuevo proyecto seleccionado. Pulsamos <Finish> y nuestro proyecto estará disponible en la vista de Proyectos.

## Tarea 4.1: Lanzar la transformación UML2JPA

La transformación UML2JPA de MOSKitt permite generar código Java a partir de elementos de un modelo UML2. Estos últimos son aquellos representados en el diagrama de clases: clases, interfaces, propiedades, operaciones, asociaciones, especializaciones, etc.

El código java contempla, además de la declaración de tipos java estándard, la anotación de las clases y/o sus atributos y métodos. Estas anotaciones permiten expresar y configurar el código generado con elementos específicos de estándares JSR. Los conjuntos de anotaciones soportados actualmente son:

- JSR-220: Enterprise JavaBean 3.0. Java Persistent API. Nos permite definir la configuración de persistencia de nuestras entidades.

- JSR-303: Bean Validation API. Nos permite especificar restricciones de validación en nuestras entidades que luego pueden ser usadas desde distintas capas en la arquitectura de nuestra aplicación.

Si vemos con un poco más de detalle el proceso de transformación del modelo UML2 al código Java correspondietne, nos daremos cuenta de que entran en juego los siguiente recursos:

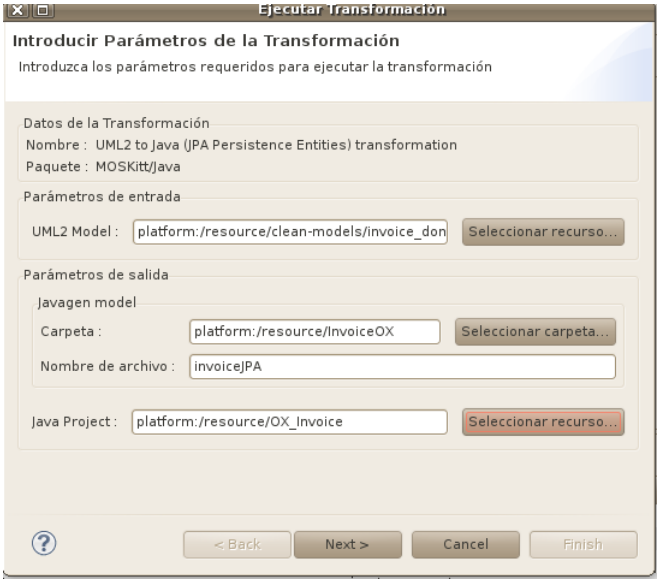
Recusos de Entrada:
- Modelo UML2 (`invoice_domain.uml`).

Recursos Intermedios:
- Modelo de configuración de la transformación (`invoice_domain.transformationconfiguration`) en el cual se indica, para cada entidad UML2 qué anotación JPA se debe aplicar.
- Modelo JPA (`invoice_domain.jpadefinitions`) que complementa al anterior indicándole la información requerida por cada una de las anotaciones seleccionadas.

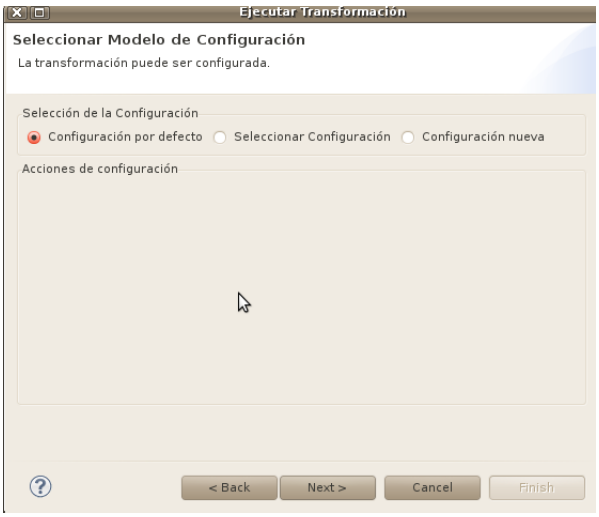
Recursos de Salida:
- Modelo JavaGen con las anotaciones JPA que se van a incluir en las clases Java (`invoiceJPA.javagen`).
- Clases Java con anotaciones JPA.

Para lanzar la transformación nos debemos situar sobre el modelo UML2 (`invoice_domain.uml`) y seleccionar del menú contextual el submenú *MOSKItt Transformations* y de éste la opción *UML2 to Java (JPA Persistence Entities) transformation*.

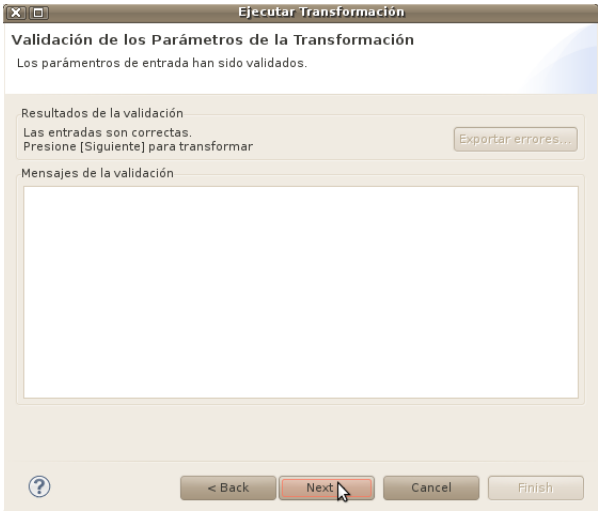
1. El asistente pide como parámetros de entrada el modelo UML2 y como parámetros de salida nos pide una carpeta dónde dejar el modelo JavaGen generado, el nombre que le debe dar y un proyecto Java donde dejar el código fuente generado:



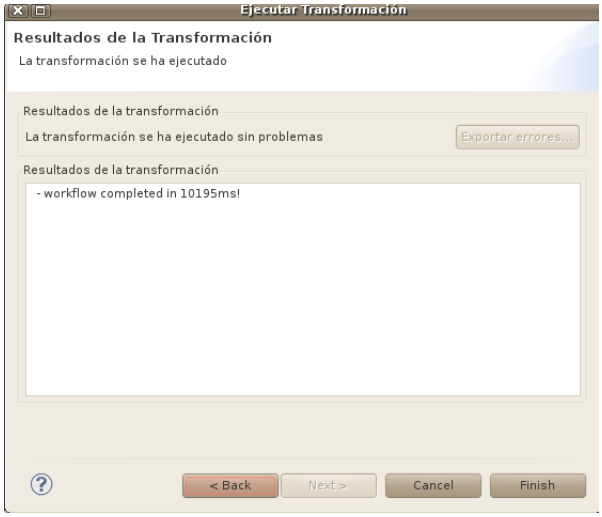

2. Todas las transformaciones MOSKitt se pueden configurar para alterar el resultado en función de la aplicación que nos ocupa en cada momento. Sin embargo, MOSKitt proporciona una configuración por defecto con las reglas de transformación más comunente utilizadas en cada caso y es esta la que vamos a utilizar por el momento (ma# adelante, en secciones posteriores veremos cómo modificar esta configuración):



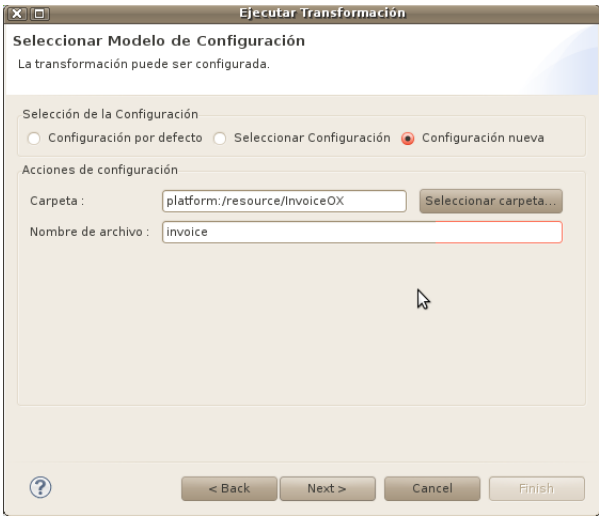

3. Antes de lanzar la transformación se comprueba que los parámetros de entrada son correctos. Es decir, que los modelos de entrada son los que deben de ser y que su contenido es el esperado (cuando se detectan errores en este punto, la transformación sólo continuará si los errores son considerados como warnings):



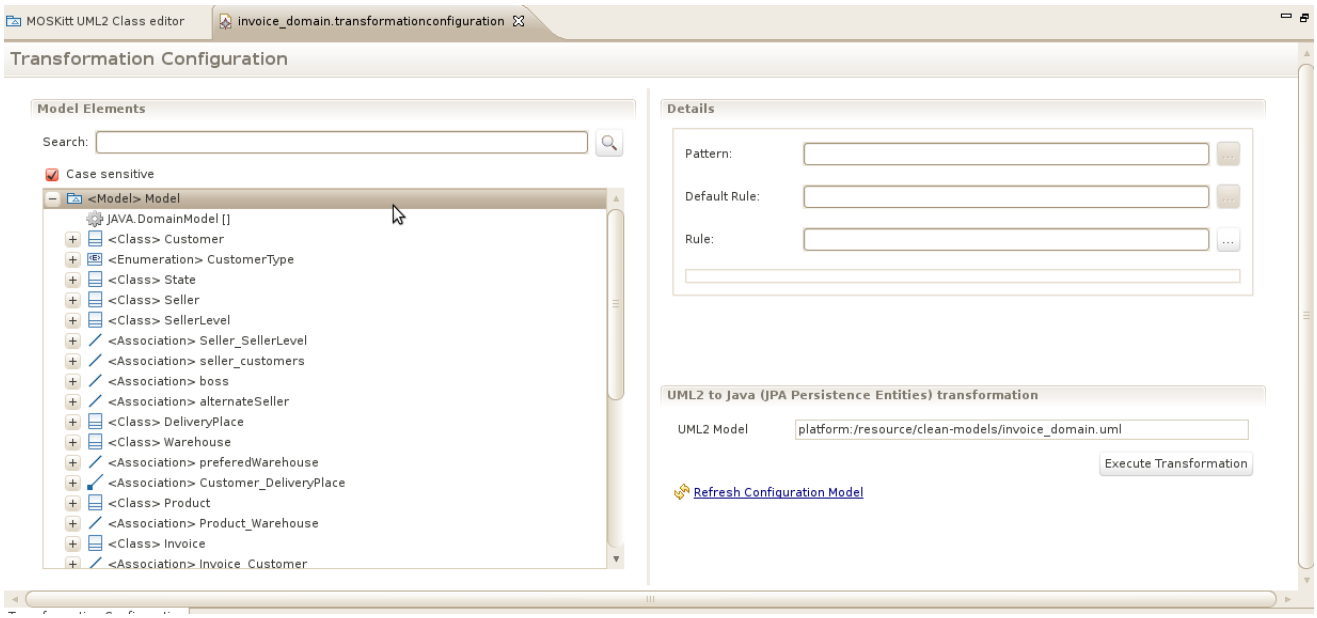

4. Una vez ha finalizado la transformación el asistente nos indica que todo ha ido bien:



Si estuviésemos interesados en configurar alguna de las etiquetas JPA a aplicar en la transformación, en el paso 2 debemos seleccionar la opción Configuración nueva tal y como se muestra en la siguiente figura:




En este caso, al pulsar el botón `Next`, se abrirá el Editor de Configuración de la transformación para que podamos configurar las etiquetas JPA que consideremos necesarias. El editor tiene el aspecto que muestra la siguiente figura:




Para más información sobre las etiquetas JPA configurables ó sobre el funcionamiento del editor de configuración consultar el Manual de Usuario de MOSKitt.

## Tarea 4.2: Lanzar la transformación UIM2OX

La transformación UIM2OX de MOSKitt permite generar código Java a partir de elementos de un modelo abstacto de interfaz de usuario UIM y un modelo JavaGenJPA (un modelo que representa clases Java preparadas para manejar su persistencia con anotaciones JPA) . Las entidades a persistir están modeladas en un modelo UML2 que a su vez está enlazado con el modelo UIM ya que en éste se indica qué propiedades de qué clases van a ser mostradas desde la interfaz.

Por tanto, los recursos utilizados por la transformación son:

Recursos de Entrada:

- Modelo UIM (`invoice.uim`).

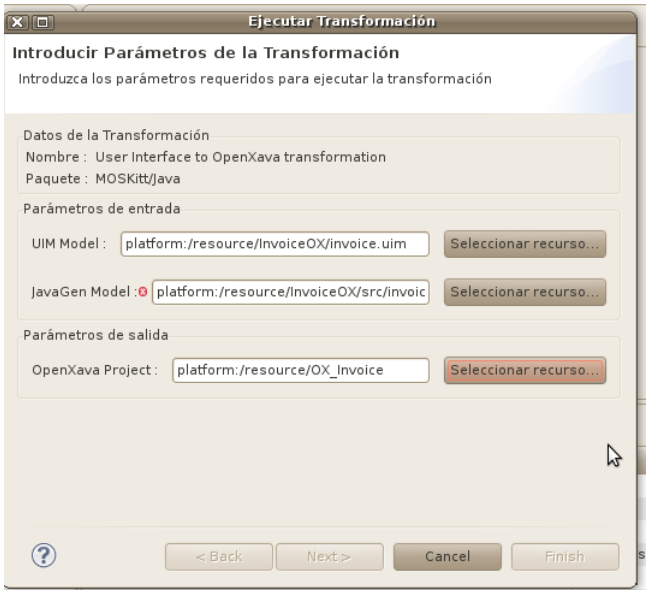
- Modelo JavaGen obtenido en el paso 1 (`invoice.javagen`).
- A partir del modelo UIM de entrada, MOSKitt alcanzará los modelos Sketcher y UML2 para obtener de ellos cierta información necesaria para completar la transformación.

Recursos de Salida

- Modelo JavaGen en el que se incorpora información sobre las anotaciones OpenXava que se van a incluir en las clases Java (`invoice.javagen`).
- Ficheros con las clases Java con anotaciones JPA y OpenXava (.java), ficheros .xml para definir la aplicación OpenXava y ficheros para la internacionalización de la aplicación (.properties).

Podemos lanzar la transformación desde el modelo de UIM. Para ello, sobre el modelo (`invoice.uim`) seleccionamos del menú contextual el submenú *MOSKItt Transformations* y de éste seleccionamos la opción *User Interface to OpenXava transformation*.

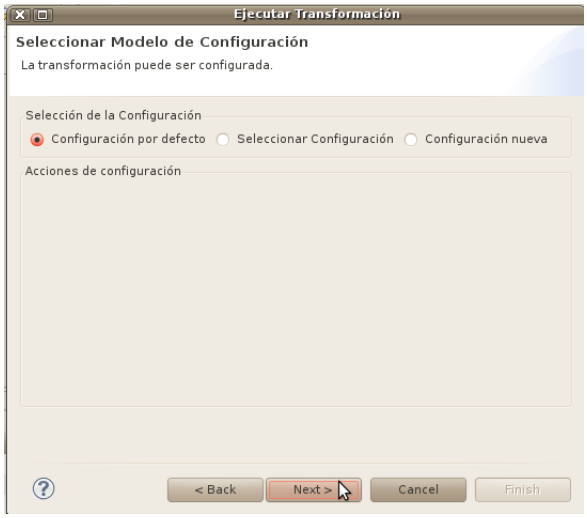
El asistente pide como parámetros de entrada los modelos UIM y JavaGen y un proyecto Java donde dejará el código generado:




Hay que tener en cuenta que hay dos proyectos diferentes:

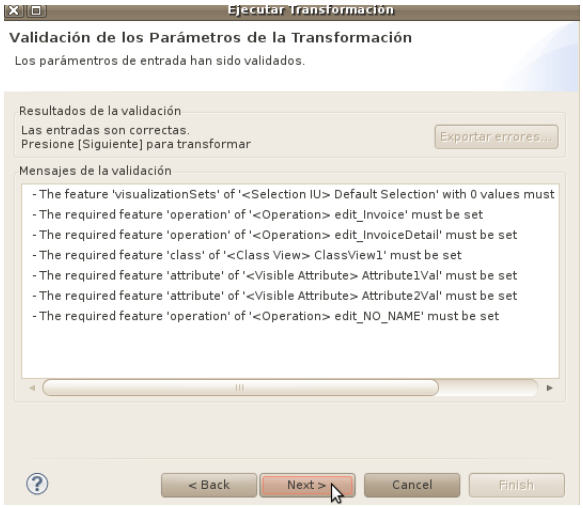
- El proyecto MOSKitt en el cual se localizan nuestros modelos.
- El proyecto Java en el cual se localiza el código generado por la transformación.

Todas las transformaciones MOSKitt se pueden configurar para alterar el resultado en función de la aplicación que nos ocupa en cada momento. Sin embargo, MOSKitt proporciona una configuración por defecto con las reglas de transformación más comunente utilizadas y es esta la que vamos a utilizar por el momento, ma# adelante, en secciones posteriores veremos cómo modificar esta configuración:

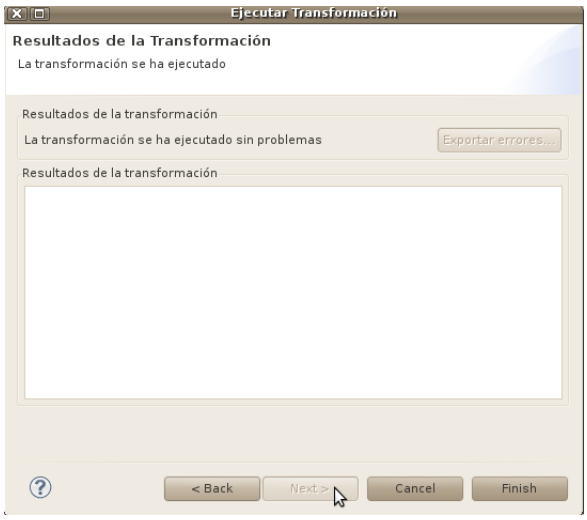



En este caso, la configuración por defecto genera el código sin Zonas Protegidas (para más información sobre qué son y para qué sirven las zonas protegidas ir al apartado correspondiente, más adelante en este mismo manual) y un módulo por componente con los controladores por defecto (para más información sobre los módulos y controladores ir al apartado correspondiente).

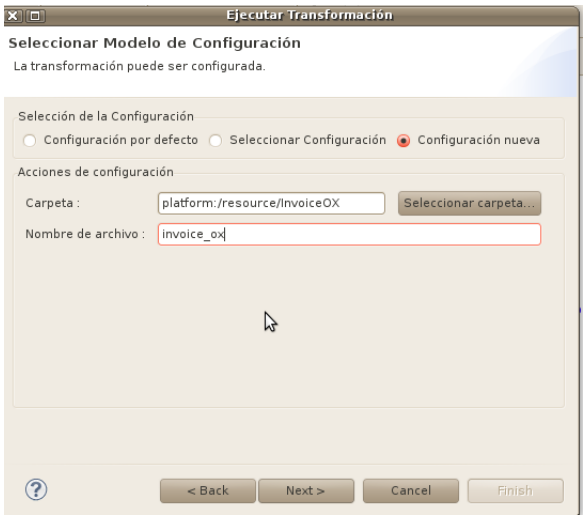
Antes de lanzar la transformación se comprueba que los parámetros de entrada son correctos. Es decir, que los modelos de entrada son los que deben de ser y que su contenido es el esperado. Cuando se detectan errores en este punto, la transformación sólo continuará si los errores son considerados como warnings (como es el caso que se muestra en al figura que aparece a continuación):



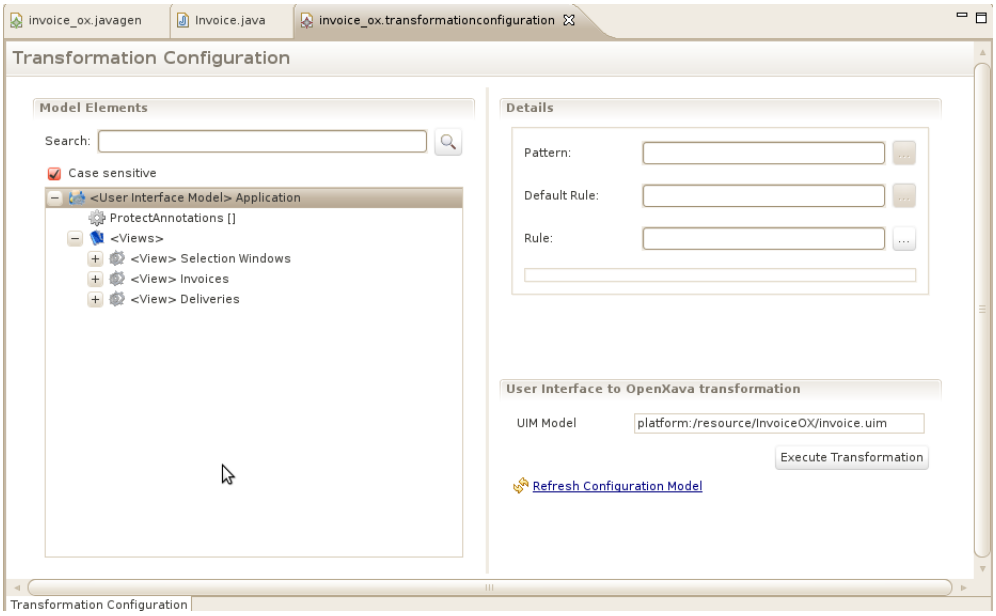

Una vez ha finalizado la transformación el asistente nos indica que todo ha ido bien:




Si estamos interesados en configurar la transformación, en el segundo paso debemos seleccionar la opción `Configuración nueva` tal y como se muestra en la siguiente figura:

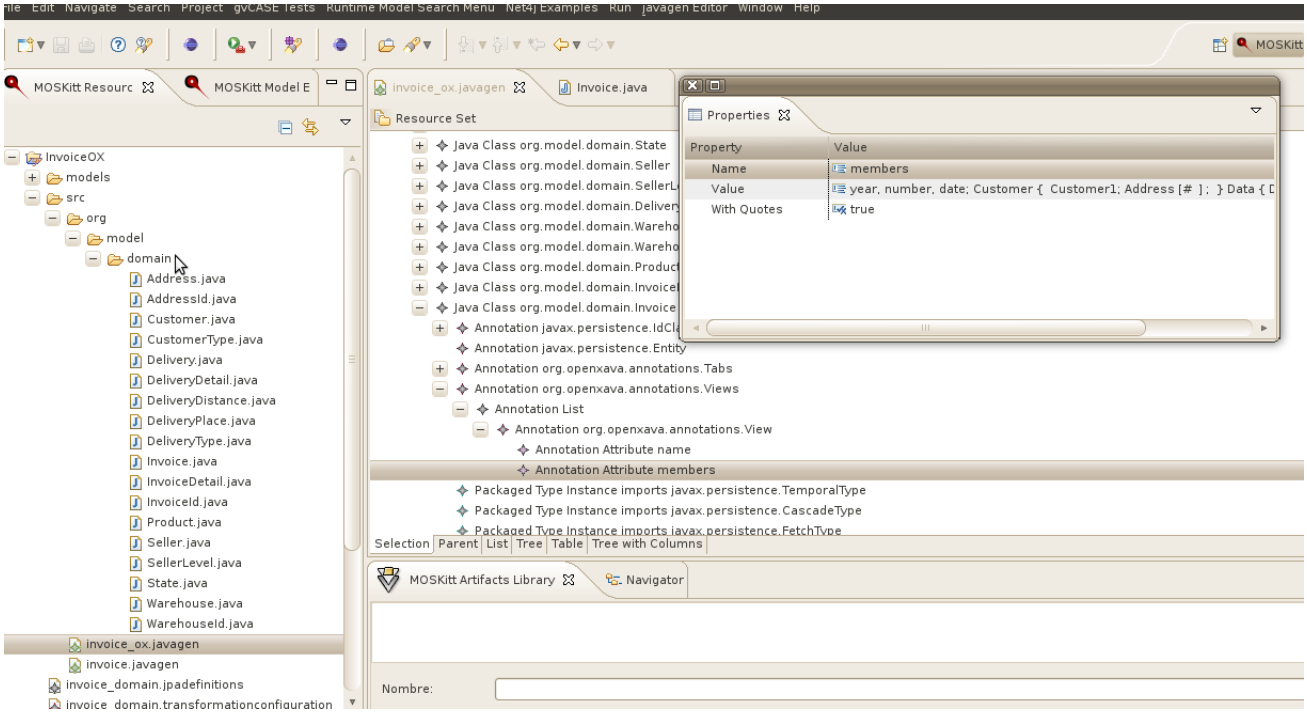



En este caso, al pulsar el botón `Next`, se abrirá el Editor de Configuración de la transformación. El editor tiene es aspecto que muestra la siguiente figura:

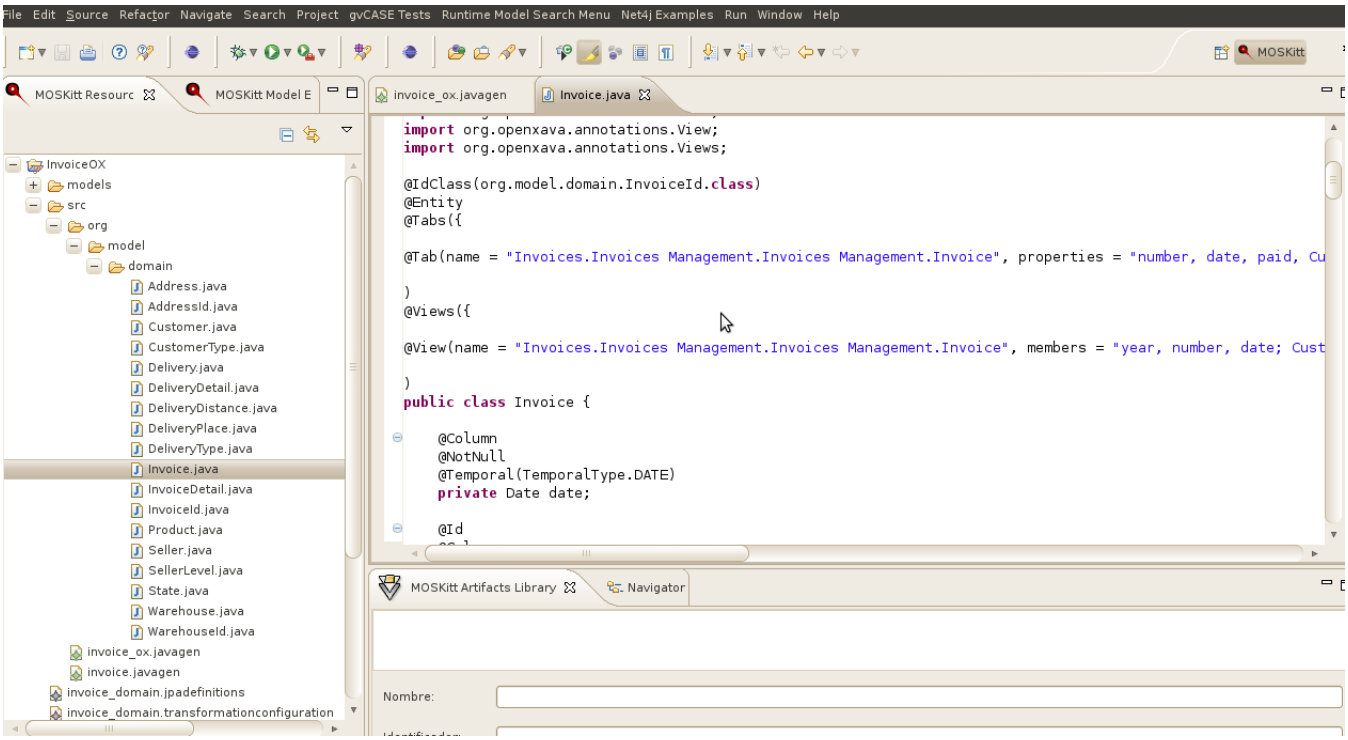


Para más información sobre el funcionamiento del editor de configuración consultar el Manual de Usuario de MOSKitt.

En la siguiente figura se muestra el resultado de la transformación. En concreto el contenido del modelo JavaGen en el cual podemos ver las clases Java que luego aparecen en el código generado:




A continuación podemos ver cómo al pulsar doble click sobre cualquiera de los ficheros Java se muestra su contenido en el editor con las correspondientes anotacioens JPA y OpenXava:




Para más información sobre los recursos generados por la transformación consultar más adelante, en este mismo manual.

# Entorno de desarrollo, despliegue y ejecución de una aplicación OpenXava

Existen dos modos de despliegue de una aplicación OpenXAVA. El primero de ellos

- **Empaquetado y despliegue como aplicación Web esta#dard (.WAR):** nos permite empaquetar la aplicación como un archivo .WAR (Web application Archive) el cual puede ser desplegado en cualquier contenedor de Servlets estándard, por ejemplo Apache Tomcat, Jetty, etc. El resultado nos permite acceder a cada módulo de la aplicación especificando una URL concreta, pero esto no resulta muy cómodo de cara al usuario final.

- **Empaquetado y despliegue como aplicación Web con portlets (JSR-286):** nos permite desplegar la aplicación en un portal que soporte el estándard Portlets 2.0 (JSR-286). Cada módulo de la aplicación se transforma en un portlet. El portal nos permite configurar el acceso a todos los portlets estableciendo políticas de acceso a los mismos para los usuarios del portal. Además nos permite utilizar componentes que mejoran la accesibilidad (menús) para que el usuario pueda acceder a todos los módulos de forma intuitiva. Esta es la forma que hemos elegido para desplegar y ejecutar nuestra aplicación de ejemplo.

## Descripción del entorno OXPortal

Denominamos OXPortal al entorno de desarrollo, despliegue y ejecución de nuestra aplicación OpenXAVA. Básicamente es un paquete comprimido con los componentes que se describen a continuación:

- **Contenedor de Servlets/JSP Apache Tomcat 6.0.29:** pondrá en ejecución las aplicaciones Web que despleguemos (incluida el propio portal).
  - Ubicación principal: oxportal/liferay/tomcat-6.0.29
  - Ubicación scripts de arranque/parada: oxportal/liferay/tomcat-6.0.29/bin
  - Ubicación ficheros de configuración: oxportal/liferay/tomcat-6.0.29/conf
  - Ubicación directorio despliegue: oxportal/liferay/tomcat-6.0.29/webapps
- **Portal JSR-286 Liferay 6.0.6 Community Edition:** pondrá en ejecución todos los módulos de nuestra aplicación como portlets.
  - Ubicación principal: oxportal/liferay
  - Ubicación directorio despliegue: oxportal/liferay/deploy
- **Base de datos HSQLDB (1.8.1):** mantiene los datos que necesit a el portal Liferay además de proporcionarnos almacenamiento para la base de datos de nuestra aplicación.
  - Ubicación del driver JDBC: oxportal/liferay/tomcat-6.0.29/lib/ext/hsql.jar
  - Ubicación base de datos del portal Liferay: oxportal/liferay/data
  - Ubicación base de datos de aplicación: oxportal/liferay/tomcat-6.0.29/data
- **Workspace para MOSKitt con la distribución OpenXAVA 4.2.1:** entorno que nos permite desarrollar aplicaciones OpenXAVA. Si desde MOSKitt conectamos con este workspace tendremos a disposición todo lo necesario para generar y desplegar nuestra aplicación en el portal.
  - Ubicación workspace: oxportal/workspace

Para tener listo nuestro entorno sólo tenemos que descomprimir el paquete *oxportal.zip* en un directorio a nuestra elección.

# Creación de la instancia de base de datos y configuración de acceso en un proyecto OpenXAVA.

Ya tenemos nuestro proyecto OpenXAVA en nuestro workspace listo para ser completado. El siguiente paso es crear una instancia de base de datos HSQLDB en la cual estarán las tablas correspondientes y configurar el proyecto para que pueda acceder a la misma.

Para **crear la instancia de la base de datos** realizaremos los siguientes pasos:

1. Abrir una terminal del intérprete de órdenes (shell).

2. Nos ubicamos en el directorio oxportal/liferay/tomcat-6.0.29/bin

3. Ejecutamos la orden de creación de instancia de base de datos de HSQLDB: *./start-hsqldb.sh <nombre_instancia> <puerto>*

   Debemos tener en cuenta qué nombre damos a nuestra instancia y qué puerto seleccionamos. Ejemplo: *./start-hsqldb.sh invoicingDB 9001*

4. A partir de este momento ya tenemos una instancia de base de datos vacía que podemos acceder a ella vía JDBC usando esta URL: *jdbc:hsqldb:hsql://localhost:<puerto>/<nombre_instancia>*

   Otros parámetros de acceso necesarios son los siguientes:

   - **username:** sa
   - **password:**

Para configurar el **acceso a la base de datos desde MOSKitt para poder crear y actualizar las tablas del esquema de base de datos** realizaremos los siguientes pasos:

1. Editamos el fichero <mi_proyecto>/persistence/META-INF/persistence.xml

2. Buscamos la "<persistence-unit>" llamada "junit". Tenemos que completar la propiedad "hibernate.connection.url". Especificamos la url de acceso JDBC a nuestra base de datos: jdbc:hsqldb:hsql://localhost:<puerto>/<nombre_instancia_db>

3. Guardamos el fichero.

Para configurar el **acceso a la base de datos desde el portal Liferay para que nuestra aplicación acceda a los datos una vez desplegada y en ejecución** realizaremos los siguientes pasos:

1. Verificamos que el servidor Tomcat lo tenemos parado. Si no es así debemos pararlo (`./stop-tomcat.sh`).

2. Editamos el fichero oxportal/liferay/tomcat-6.0.29/conf/context.xml

3. Añadimos al tag <Context> la siguiente entrada:

   *<Resource name="jdbc/<mi_proyecto>DS" auth="Container" type="javax.sql.DataSource" maxActive="20" maxIdle="5" maxWait="10000" username="sa" password="" driverClassName="org.hsqldb.jdbcDriver" url="jdbc:hsqldb:hsql://localhost:<puerto>/<nombre_instancia_db>"/>*

4. Una vez el servidor Tomcat se inicie las aplicaciones en él desplegadas tendrán disponible el recurso JNDI con nombre "jdbc/<mi_proyecto>DS" que hemos registrado en el contexto global

del contenedor. Este recurso es un DataSource que nuestra aplicación usará para acceder a la base de datos configurada.

# Creación de las tablas de la base de datos a partir de entidades JPA de un proyecto OpenXAVA.

Una vez tenemos el código de nuestras entidades JPA en nuestra aplicación (ya sea por haber sido generado con MOSKitt o creado a mano) es el momento de crear las tablas en la base de datos. Debemos cerciorarnos que tenemos en marcha la instancia de la base de datos tal y como explica el punto anterior.

Desde MOSKitt ejecutaremos los siguientes pasos:

1. Abrir el fichero <mi_proyecto>/build.xml

2. En la vista "Outline" veremos la lista de targets disponibles en este automatismo Ant. Seleccionamos la target llamada "updateSchema". Abrimos el menú contextual desde ella y ejecutamos la opción <Run As>.<Ant Build>.

3. En la vista "Console" veremos una traza de la ejecución del automatismo seleccionado. En ella veremos cómo se crean nuestras tablas y si el proceso ha ido bien o no. El siguiente paso es inspeccionar dichas tablas. Para ello usaremos un gestor de base de datos compatible JDBC (ejemplo, SquirrelSQL).

# Puesta en marcha del entorno de ejecución. Arranque del portal Liferay.

Antes de desplegar nuestra aplicación necesitamos poner en marcha el portal para que una vez se realize el despliegue podamos tener la aplicación en ejecución.

Para ello desde una terminal de órdenes nos ubicaremos en la carpeta o*oxportal/liferay/tomcat-6.0.29/bin y ejecutaremos la orden:*

*./startup.sh.*

Si queremos ver los mensajes de información que el contenedor Tomcat genera para poder ser informados de posibles errores tanto en el despliegue como en la ejecución de nuestra aplicación, podemos ejecutar la siguiente orden:

*tail -f ../logs/catalina.out.*

Esta orden muestra por la terminal todo lo que ocurre en el contenedor.

# Empaquetado y despliegue de un proyecto OpenXAVA como portlets

Ya tenemos nuestra base de datos disponible y el portal en ejecución. Nuestra aplicación está lista para ser desplegada y ejecutada. En el fichero build.xml que acompaña a nuestro proyecto tenemos un target llamado "**deployPortlets**" que hace todas las tareas de una vez. Estas tareas son:

- Compilación del código fuente de la aplicación.

  Generación de los ficheros descriptores necesarios: web.xml y portlet.xml

  Empaquetado de la aplicación como fichero .WAR

Desplegado de la aplicación en el portal (Liferay).

Desde MOSKitt ejecutaremos los siguientes pasos:

1. Abrir el fichero <mi_proyecto>/build.xml

2. En la vista "Outline" veremos la lista de targets disponibles en este automatismo Ant. Seleccionamos la target llamada "deployPortlets". Abrimos el menú contextual desde ella y ejecutamos la opción <Run As>.<Ant Build>.

3. En la vista "Console" veremos una traza de la ejecución del automatismo seleccionado. En ella veremos cómo compila, se empaqueta la aplicación y se copia a la carpeta de despliegue de aplicaciones con portlets en el portal.

Si tenemos una terminal con los mensajes de salida del contenedor Tomcat tal y como se sugiere en el punto anterior, podremos ver el proceso de despliegue de la aplicación en el portal Liferay. Ello nos ayudara a detectar posibles errores en el despliegue así como a adivinar cuando nuestra aplicación está lista para ser usada.

# Acceso a la aplicación en ejecución desde el portal Liferay

Ya tenemos desplegada nuestra aplicación en el portal. Ahora nos toca acceder al mismo y configurarlo para que los portlets de nuestra aplicación OpenXAVA sean accesibles por el usuario final.

La instancia de Liferay que se distribuye con el paquete oxportal.zip ya viene configurada con un usuario administrador con privilegios para crear páginas que nos permitan ejecutar los portlets o módulos de nuestra aplicación. Vamos pues a crear una página en el portal que sea capaz de darnos acceso a los portlets de nuestra aplicación. Para ello seguiremos los siguientes pasos:

1. Abrir nuestro navegador y apuntar a la URL: http://localhost:8080

2. En el portlet de Login introduciremos los datos de acceso del usuario administrador del portal:

   - Usuario: test@moskitt.org

   - Password: test

3. Desde el menú ubicado en la parte superior, ejecutaremos la acción <Administrar>.<Panel de Control>

4. En el panel colapsable localizado a la izquierda, en la sección llamada "MOSKitt Web Application Generation", accedemos a la opción "Páginas".

5. En el formulario actual vamos a crear una nueva página. Para ello introduciremos el nombre de la página y le diremos que es de tipo "Panel". Pulsaremos después el botón "Añadir página".

6. Ya tenemos nuestra página creada. Ahora tenemos que configurarla. Para ello seleccionaremos la página en el menú. Después tenemos que seleccionar la opción tabular llamada "Página". Entonces nos aparecerá un formulario con la configuración de la misma.

7. En el formulario de configuración de la página veremos una sección llamada "Seleccione los portlets que estarán disponibles en el panel. ". En ella debemos tener disponible nuestra aplicación con todos sus portlets. Seleccionaremos aquellos que necesitemos sean accesibles por el usuario final. Después pulsaremos el botón "Guardar".

8. Ya tenemos nuestra página configurada. Para acceder a la aplicación desde el menú superior pulsaremos la opción "Volver a MOSKitt Web Application Generation". En este momento veremos una opción más tabular con el nombre de nuestra página. Si accedemos nos aparecerá a la izquierda

un portlet que nos muestra la lista de módulos de nuestra aplicación. Seleccionando cualquiera de ellos nos aparecerá como un portlet a la derecha la ejecución del módulo correspondiente.

# Zonas protegidas: Estrategias utilizadas por MOSKitt para respetar los cambios realizados por el desarrollador.

Uno de los principales temas a resolver cuando se genera código de forma automática es el tratamiento de las modificaciones manuales que ha realizado el desarrollador cuando, debido a cambios en los modelos, debemos regenerar el código.

Como hemos dicho al principio del manual, MOSKitt proporciona una generación semi-automática de código, lo cual implica que en la mayoría de los casos, los desarrolladores deberán completar el código para terminar la aplicación.

En la gran mayoría de los casos, éstos cambios deben mantenerse a la vez que se debe incorporar en el código los cambios originados por los cambios en el modelo. Para hacer esto, MOSKitt utiliza la técnica de las Zonas Protegidas: bloques de código delimitados por dos instrucciones, una de apertura y otra de cierre. Las líneas comprendidas entre ellas serán respetadas por el generador siempre.

Podemos seleccionar el número y localización de las zonas protegidas requeridas en el código que vamos a generar, haciendo uso de la configuración de las transformaciones UML2JPA y UIM2OX.

1. **Sintaxis**

   Cada zona tiene el aspecto siguiente:

   */*PROTECTED REGION ID(preIniciarVentana__c7GjUIBSEeCc2cs1bFJTkw_PIU) ENABLED START*/*

   /* Escribe aquí el código de la función */

   /**********/

   /**********/

   /* Fin de la zona protegida */

   /**********/

   /*PROTECTED REGION END*/

   Las líneas con la palabra *PROTECTED* marcan el inicio y el final del bloque.

   La cadena que se encuentra encerrada entre los paréntesis de *ID* es el identificador de la zona. **Nunca se debe modificadar manualmente**.

2. **Activación y desactivación de las zonas protegidas**

   La palabra ENABLED indica que la zona debe protegerse. Si se quita esa palabra, la zona se reescribirá con lo generado.

   Por norma general, en una primera generación las zonas en las que el generador haya necesitado incluir código aparecerán desactivadas y aquellas en las que no haya incluido nada estarán activadas.

   El desarrollador podrá activar/desactivar las zonas protegidas a su conveniencia.

3. **Identificadores de las zonas protegidas.**

   Cada zona protegida estará etiquetada por un código único que la identifica. Este código se contruye basándose en el identificador de los objetos que determinan su contenido. Estos identificadores no pueden ser modificados por los usuarios y el generador los mantiene de una generación a otra.

   En la zona protegida que aparece en el punto 1, el identificador es: *ID(preIniciarVentana__c7GjUIBSEeCc2cs1bFJTkw_PIU)*.

   Si el usuario modifica el identificador de la zona protegida y el generador ya no la reconocerá en futuras generaciones.

# Configuración de la generación de módulos y controladores de la aplicación

Podemos especificar el comportamiento de los módulos de la aplicación haciendo uso de la configuración de la transformación UI2OX. En ella, existe un patrón 'Generate' por cada elemento View del modelo UIM en el cuál se puede especificar cómo pasar de la vista detalle a la vista lista, o bien definir un módulo que no tenga detalle y lista. Existe 5 modos de visualización distintos:

- DEFAULT - Genera un módulo típico ('detalle', 'lista' y 'ambos').
- DETAIL_ONLY - Genera un módulo con solo 'detalle'.
- LIST_ONLY - Genera un módulo con solo 'lista'.
- SPLIT_ONLY- Genera un módulo con 'ambos'.
- DETAIL_LIST - Genera un módulo con 'detalle' y 'lista' sin la vista 'ambos'.

La definición de módulos y controladores de una aplicación OpenXava queda fuera de nuestro alcance (http://openxava.wikispaces.com/application_es).

## Módulos de solo lectura

Si lo que queremos es especificar un módulo de solo lectura, tenemos que indicar el tag 'ReadOnly' para el elemento Window del Sketcher.

# Aspectos que se deben tener en cuenta al definir los modelos para la generación de código.

Este punto del manual está dirigido a aquellos analistas que están modelando sus aplicaciones con la intención de generar la mayor cantidad de código OpenXava posible con MOSKitt.

Con este objetivo vamos a describir cómo deben ser los modelos de entrada de la transformación para obtener el máximo partido a la generación de código ya que hemos de ser conscientes de que el generador espera que los modelos de entrada estén definidos de una forma muy concreta.

## Tarea 2: El modelo UML2

El generador sólo tiene en cuenta el Diagrama de Clases de un modelo UML2.

De este diagrama hace uso de los siguientes elementos:

- Los nombres de las clases definidas.
- Los nombres de las propiedades de las clases.
- Los tipos de datos de las propiedades de las clases con la restricción de que en esta versión no se soportan elementos DataType ni Class para definir los tipos.
- Las asociaciones entre las clases para navegar en el modelo.
- Las enumeraciones para definir las listas estáticas de valores.

Este modelo deberá estar trazado con el modelo JavaGen que se ha generado en la transformación UML2JPA, por tanto, debemos llevar la precaución de que el modelo de trazas esté en el proyecto:

- `invoice_domain!_!invoice!-!uml2jpa.gvtrace`

# Tareas 3 : Los modelos Sketcher y UIM

## Conceptos Previos: Generación de código basada en Plantillas/ Patrones

El generador de código de MOSKitt es un **generador basado en plantillas**, esto significa que si en los modelos de Sketcher/UIM hay definidas ventanas que no se basan en las plantillas que proporciona MOSKitt para el framework, estas ventanas no serán generadas.

¿Qué ocurre cuando tengamos que especificar una ventana que no está aún soportada por el framework o que su plantilla aún no esté disponible en el Sketcher?: En estos casos siempre podremos modelar la ventana usando todos los elementos que el Sketcher nos proporciona en su paleta, sin embargo, como no está basada en ninguna plantilla conocida por el generador, el código no se incorporará a la aplicación gvHidra.

*La idea es: "Siempre podremos modelar, aunque no podamos generar*".

Por lo tanto, en el caso de que un modelo contenga ventanas basadas en plantillas y otras que no lo estén, en la aplicación OpenXava generada sólo aparecerán aquellas ventanas que sí lo están.

Se pueden consultar las plantillas proporcionadas por MOSKitt en el apartado Plantillas genéricas para el modelado de aplicaciones web

### ¿Cómo reconoce el generador los elementos que contienen las plantillas?: Etiquetas

La pregunta que nos tenemos que hacer ahora es:

¿Cómo reconoce el generador las plantillas?:

- ¿Por su nombre?
- ¿Porque comprueba su contenido y lo deduce?
- ¿Por.....?

El generador reconoce tanto las plantillas como los elementos que las componen gracias a un conjunto de etiquetas (`tags`) que funcionan a modo de palabras reservadas. Gracias a estos tags el generador asume ya una estructura y un comportamiento concreto en cada elemento, lo cual le permite también validar los modelos de entrada que recibe.

Las etiquetas se pueden consultar en la pestaña de propiedades de cada elemento, subpestaña `Tags`.


Para más información sobre las etiquetas aplicadas a los elementos del Sketcher para construir las plantillas interpretables por el generador de OpenXava consultar más adelante el apartado "Lista completa de etiquetas que reconoce el generador".

## Modelo UIM

El modelo de UIM también tiene que cumplir una serie de propiedades para poder ser transformado a OpenXava.

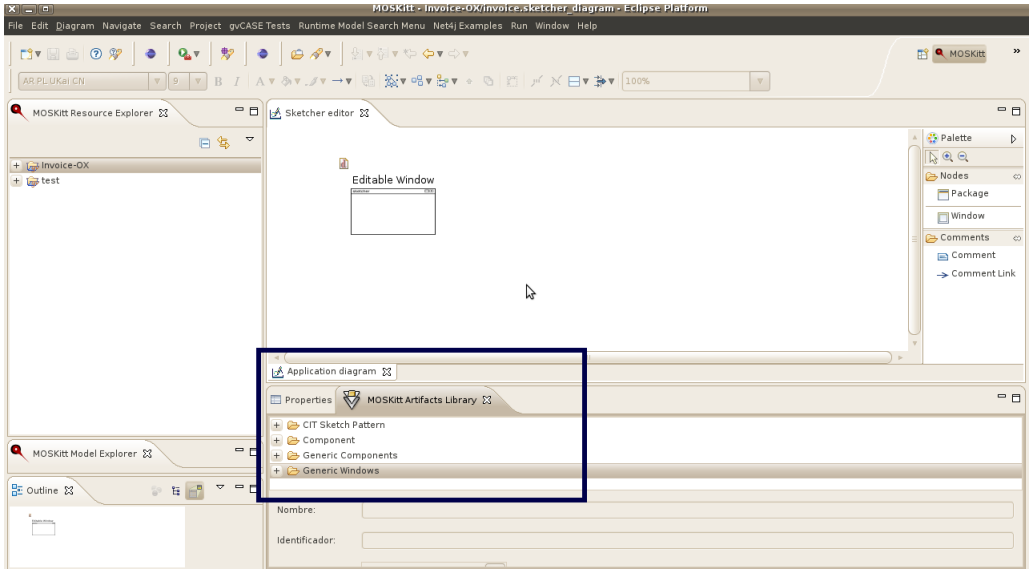
Como el modelo UIM que se genera a partir de la transformación Sketcher2UIM (generic) ya cumple con todas estas propiedades y por el momento, la versión 0.3.0 del generador codgen-OpenXava no hace uso de ningún elemento de UIM que no provenga de esta transformación no vamos a entrar en este punto.

# Anexos

# Plantillas genéricas para el modelado de aplicaciones web

En este apartado vamos a examinar con detalle los elementos que componen cada una de las plantillas proporcionadas por MOSKitt para el modelado de aplicaciones web interpretables por el generador de código moskitt-codgen-openxava..

Para tener disponibles las plantillas que proporciona MOSKitt, éstas deben haber sido cargadas previamente (ver sección sobre cómo crear un nuevo modelo de Sketcher). Una vez cargadas, las plantillas estarán disponibles en la vista `Window/Show View/Other.../MOSKitt Artifacts Library` tal y como aparece en la siguiente figura:




MOSKitt proporciona varios juegos de plantillas, preparadas para que sean interpretadas por los diferentes generadores que proporciona. Los juegos de plantillas interpretables por el generador de OpenXava de MOSKitt son:

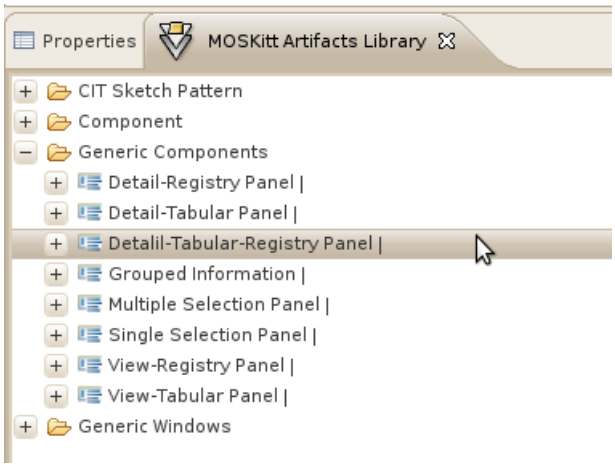
- **Generic Components**: contiene componentes genéricos que pueden ser incluidos en los elementos de tipo `Window` en el sketcher. Cada elemento de tipo `Window` en el Sketcher contendrá la definición de un módulo de OpenXava.

- **Generic Window**: contiene elementos `Window` ya preeditados para agilizar la definición de los módulos OpenXava.

A continuación vamos a ver con detalle cada una de estas plantillas.

# Plantillas de Componentes (Generic Component)

Estas plantillas aparecen en la vista `MOSKitt Artifacts Library` agrupadas en la carpeta `Generic Components`:




En esta carpeta disponemos de la siguiente lista de **Componentes**:

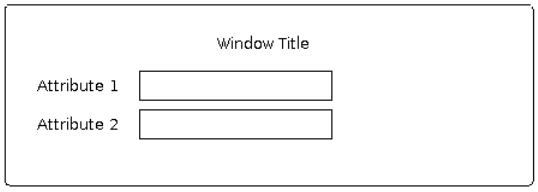
| Plantilla Sketcher | Anotaciones OpenXava | Se compone de... |
|---|---|---|
| View-Registry Panel | @View | |
| View-Tabular Panel | @Tab | |
| Detail-Registry Panel | @ColectionView + @View | |
| Detail-Tabular Panel | @ListProperties + @AsEmbedded | |
| Detail-Tabular-Registry Panel | @ColectionView + @View | Detail-Tabular Panel + Detail Registry Panel |
| Grouped Information | [] en @View.members | |
| Multiple Selection Panel | @ListProperties | |
| Single Selection Panel | @ReferenceView + @View | |
| Single Selection Group Panel | @ReferenceView + @View | |

Vamos a detallar ahora cada uno de estos componentes.

## View-Registry Panel

Representa a cualquier panel en formato registro en OpenXava siempre que no forme parte de una estructura Maestro/Detalle.

Para modelar la Vista de `Detalle`/`Detail` en un módulo OpenXava se utiliza precisamente este mismo componente, el cual tiene el siguiente aspecto:




Tags aplicados a este elemento:

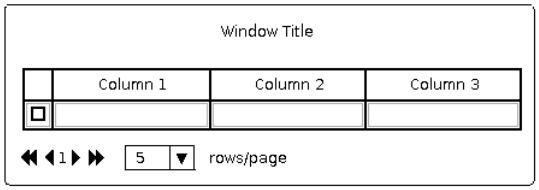
- `Registry`

Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual.

## View-Tabular Panel

Representa a cualquier panel en formato tabular en OpenXava siempre que no forme parte de una estructura Maestro/Detalle.

Para modelar la Vista de `Lista`/`List` en un módulo OpenXava se utiliza precisamente este componente, el cual tiene el siguiente aspecto:




Tags aplicados a este elemento siempre:

- `Tabular`

Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual.

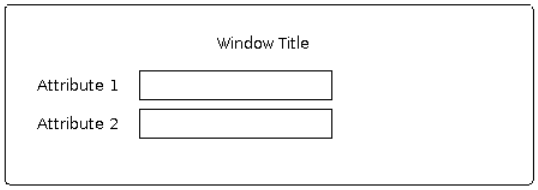
## Detail-Registry Panel

Representa a cualquier panel en formato *Registro* en OpenXava cuando es el elemento Detalle en una estructura Maestro/Detalle (no confundir con la vista *Detalle*, la cual está asociada con la etiqueta `Registry`).

Un panel se considera Maestro en una estructura Maestro/Detalle cuando contiene a un panel de este tipo.

Al igual que en el framework, se soporta cualquier nivel de anidamiento de jerarquías Maestro/Detalle ya que un panel puede ser Maestro (por contener un elemento con el tag `Detail-Registry Panel`,`Detail-Tabular Panel ó Detail-Tabular-Registry Panel`) y a la vez Detalle (por llevar el tag `Detail-Registry Panel`).

El aspecto de este componente es el mismo que el del componente `View-Registry Panel`:




Tags aplicados a este elemento:

- `Registry`
- `Detail`

Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual.

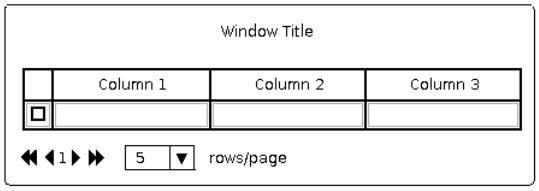
## Detail-Tabular Panel

Representa a cualquier panel en formato *Tabular* en OpenXava cuando es el elemento Detalle en una estructura Maestro/Detalle (no confundir con la vista Detalle, la cual está asociada con la etiqueta `Tabular`).

Un panel se considera Maestro en una estructura Maestro/Detalle cuando contiene a un panel de este tipo.

Al igual que en el framework, se soporta cualquier nivel de anidamiento de jerarquías Maestro/Detalle ya que un panel puede ser Maestro (por contener un elemento con el tag `Detail-Registry Panel,Detail-Tabular Panel ó Detail-Tabular-Registry Panel`) y a la vez Detalle (por llevar el tag `Detail-Tabular Panel`).

El aspecto de este componente es el mismo que el del componente `Detail-Tabular Panel`:




Tags aplicados a este elemento:

- `Tabular`
- `Detail`

Tags opcionales para este elemento:

- `NotInsert`
- `NotEdit`
- `NotDelete`

En función de los tags incluidos se generarán las anotaciones correspondientes en OpenXava: `@ReadOnly`, `@NoCreate`, `@NoModify` y `@EditOnly`.

Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual.

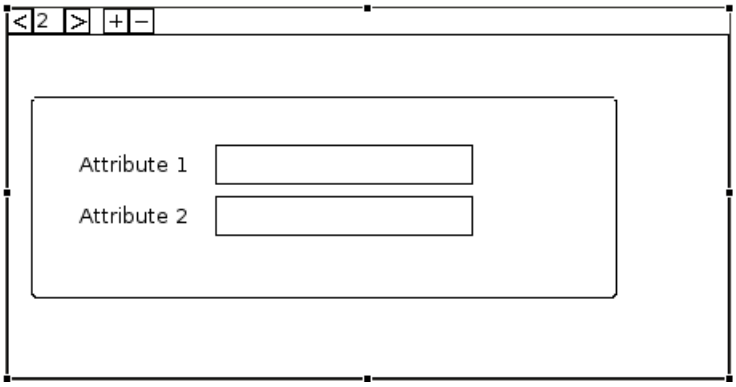
## Detail-Tabular-Registry Panel

Representa a cualquier panel en el cual se combinan los componentes:

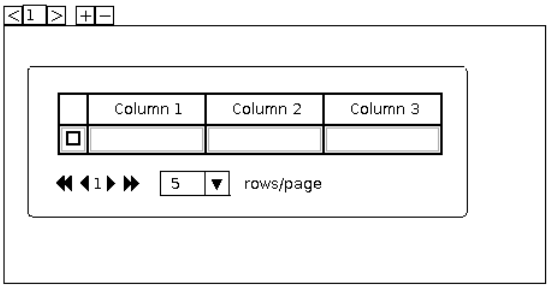
- `Detail-Tabular Panel`.
- `Detail-Registry Panel`.

Viene representado por el elemento `Multiview Panel` de Sketcher para expresar que se puede comnutar de un formato Tabular al formato Registro. En cualquiera de ellos se puede mostrar información distinta con respecto al mismo detalle.

Cada una de las vistas del componente viene represetnado por un `TabItem` distinto tal y como podemos v en las siguientes imágenes correspondientes al formato *Registro*:




Y al formato *Tabular*:




Conmutamos de una a otra haciendo uso de las pestañas < y >.

Tags aplicados a este elemento:

- `Tabular`
- `Registry`
- `Detail`

Tags opcionales para este elemento:

- `NotInsert`
- `NotEdit`
- `NotDelete`

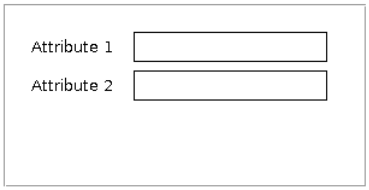
En función de los tags incluidos se generarán las anotaciones correspondientes en OpenXava: `@ReadOnly`, `@NoCreate`, `@NoModify` y `@EditOnly`.

Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual.

## Grouped Information

Permite agrupar información representándola visualmente a través de un típico `Group Box`.

El componente tiene el siguiente aspecto:

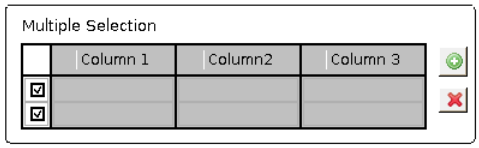


No hay Tags aplicados a este elemento.

## Multiple Selection Panel

Este control permite relacionar una instancia de una Clase1 con más de una instancia de otra Clase2. La Clase2 será alcanzable desde el diagrama de clases a través de una asociación entre la Clase1 y la Clase2 con cardinalidad 0:* ó 0:1 en el extremo de la relación correspondiente a la Clase2. La asociación además será navegable hacia la Clase2.

El aspecto de este componente es el siguiente:




Por contener un elemento Table permite mostrar más de un propiedad de las instancias seleccionadas (una propiedad por columna en la tabla).

Tags aplicados a este elemento:

- `MultipleSelection`

Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual.

## Single Selection Panel

Este control permite relacionar una instancia de una Clase1 con una sola instancia de otra Clase2.

La Clase2 será alcanzable desde el diagrama de clases a través de una asociación entre la Clase1 y la Clase2. En este caso la cardinalidad no es un aspecto crítico pues, aun existiendo en el diagrama de clases cardinalidad 0:* ó 1:* en el extremo de la relación correspondiente a la Clase2, en esta interfaz, el analista podría querer ser más restrictivo. La asociación además será navegable hacia la Clase2.

El aspecto de este componente es el siguiente:


Tal y como se muestra en la siguiente figura, este componente incorpora también un botón de búsqueda para seleccionar la instancia relacionada. Se asume que al pulsar el botón se invoca a una ventana de selección (representada por la plantilla `Selection Window` localizada en la carpeta `Generic Windows` de la vista `MOSKitt Artifacts Library`).

Por contener tan sólo un elemento `TexBox` no editable, permite mostrar tan sólo una propiedad de la instancia seleccionada.

Tags aplicados a este elemento:

- `SingleSelection`

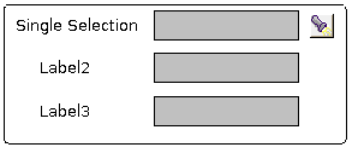
Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual. Para más información sobre cómo podemos asociar el botón de búsqueda con este tipo de Ventanas de Selección consultar más adelante en la sección de Plantillas correspondientes a las ventanas.

## Single Selection Group Panel

Al igual que el componente `Single Selection Panel`, este control permite relacionar una instancia de una Clase1 con una sola instancia de otra Clase2.

La Clase2 será alcanzable desde el diagrama de clases a través de una asociación entre la Clase1 y la Clase2. En este caso la cardinalidad no es un aspecto crítico pues, aun existiendo en el diagrama de clases cardinalidad 0:* ó 1:* en el extremo de la relación correspondiente a la Clase2, en esta interfaz, el analista podría querer ser más restrictivo. La asociación además será navegable hacia la Clase2.

El aspecto de este componente es el siguiente:




Tal y como se muestra en la siguiente figura, al igual que ocurría con componente`Single Selection Panel`, éste también incorpora un botón de búsqueda para seleccionar la instancia relacionada. Se asume que al pulsar el botón se invoca a una ventana de selección (representada por la plantilla `Selection Windows` localizada en la carpeta `Generic Window` de la vista `MOSKitt Artifacts Library`).

Por contener más de un elemento `TexBox` (no editables todos ellos), permite mostrar más de una propiedad de la instancia seleccionada (de hecho esta es la única diferencia con el componente `Single Selection Panel`. Es más, si arrastramos un componente de tipo `Single Selection Panel` y lo queremos convertir en un componente de tipo `Single Selection Group Panel` lo único que temos que hacer es añadir más `TextBox` dentro del elemento `Panel` que los contiene a todos.

Tags aplicados a este elemento:

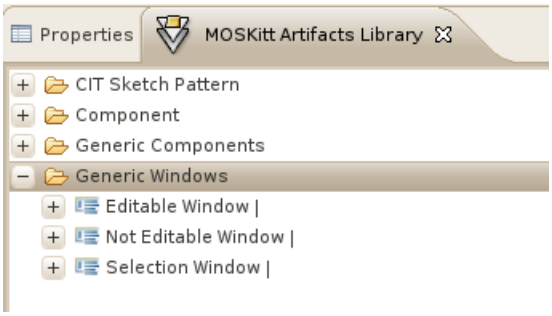
- `SingleSelection`

Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual.

Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual. Para más información sobre cómo podemos asociar el botón de búsqueda con este tipo de Ventanas de Selección consultar más adelante en la sección de Plantillas correspondientes a las ventanas.

## Plantillas de Ventanas

Estas plantillas aparecen en la vista `MOSKitt Artifacts Library` agrupadas en la carpeta `Generic Windows`:




En esta carpeta disponemos de la siguiente lista de **Ventanas**:

| Plantilla Sketcher | Anotaciones OpenXava | Se compone de... |
|---|---|---|
| `Editable Window` | module(Controler="Typical") | View-Tabular Panel + View-Registry Panel |
| `Not Editable Window` | module(Controler="Print",EnvVar="XavaEditable=false") | View-Tabular Panel + View-Registry Panel |

| Plantilla Sketcher | Anotaciones OpenXava | Se compone de... |
|---|---|---|
| `Selection Window` | @Tab | View-Tabular Panel |

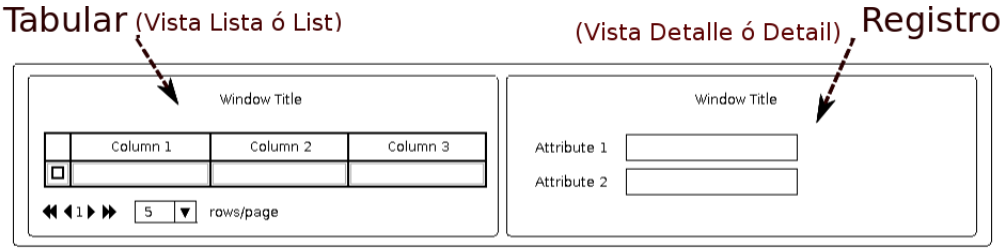
Vamos a detallar ahora cada uno de estas plantillas.

## Editable Window

Representa a cualquier módulo OpenXava cuyo controlador es el `Typical`. Incorpora operaciones de edición, inserción y borrado para las instancias de las clases que allí se muestran. Además incorpora por defecto dos vistas para la información que maneja:

- La vista `Lista`/`List` en formato *Tabular* modelada haciendo uso del componente `View-Tabular Panel`.
- La vista `Detalle`/`Detail` en formato *Registro* modelada haciendo uso del componente `View-Registry Panel`.

El aspecto de esta plantilla es el siguiente:



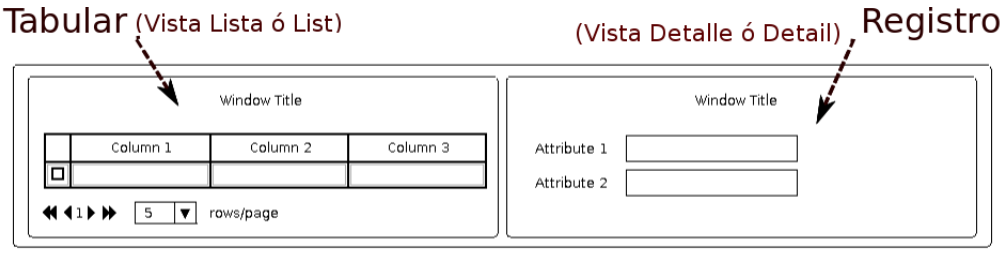

No hay Tags aplicados a este elemento.

## Non Editable Window

Representa a cualquier módulo OpenXava cuyo controlador es el `Print`. No incorpora operaciones de edición, inserción ó borrado para las instancias de las clases que allí se muestran. Al igual que la `Editable Window` incorpora por defecto dos vistas para la información que maneja:

- La vista `Lista`/`List` en formato *Tabular* modelada haciendo uso del componente `View-Tabular Panel`.
- La vista `Detalle`/`Detail` en formato *Registro* modelada haciendo uso del componente `View-Registry Panel`.

El aspecto de esta plantilla es el mismo que el de la `Editable Window`:




Tags aplicados a este elemento:

- `ReadOnly`

Este tag es la única diferencia a nivel de modelado entre la plantilla `Editable Window` y la plantilla `Non Editable Window`.

Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual.

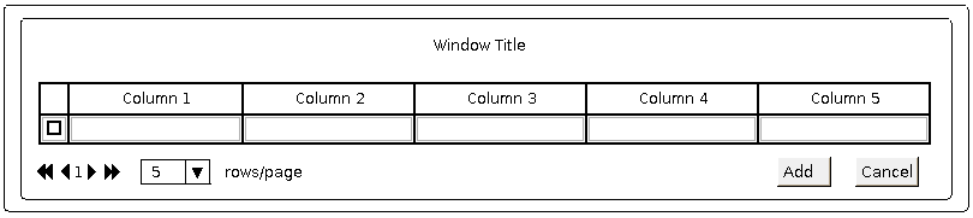
## Selection Window

A diferencia de las otras dos plantillas asociadas a elementos Window, esta no representa a un módulo OpenXava.

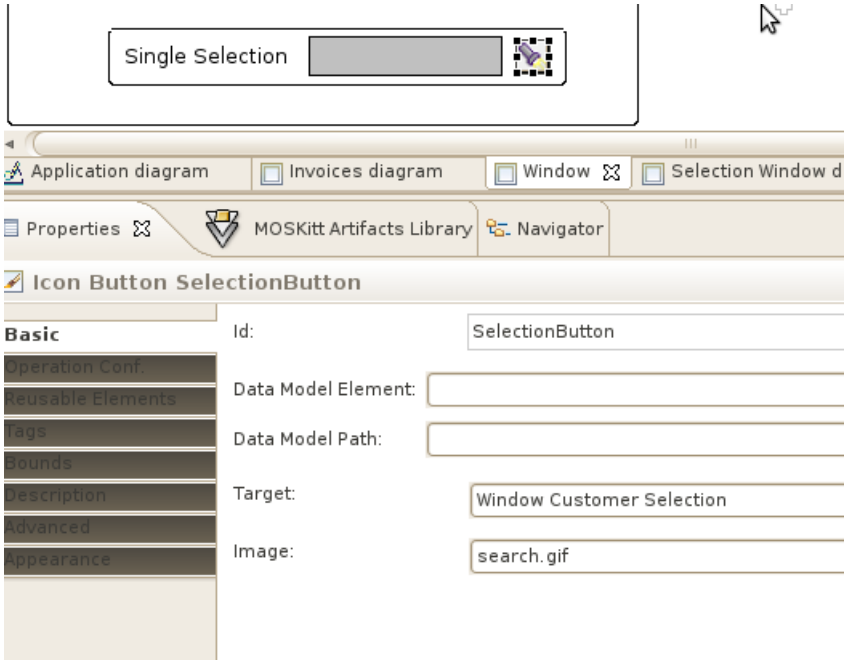
Esta plantilla permite mostrar una lista de instancias y seleccionar una de ellas. Normalmente va acompañada de un componente de selección de entre los que ofrece MOSKitt cuando se quiere que el generador complete el código de la aplicación. Estos elementos se encuentran en la carpeta `Generic Component` de la vista `MOSKitt Artifacts Library` y son:

- `Single Selection Panel`
- `Single Selection Group Panel`

El aspecto de este componente es el siguiente:



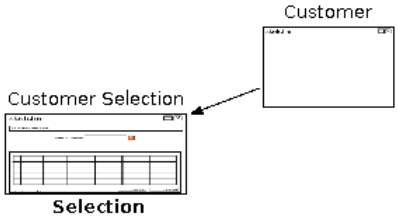

Los componentes de selección se conectan con `Windows` del tipo `Window Selection` al asignar valor a la propiedad *Target* del botón que representa la invocación de la ventana (en secciones anteriores de este manual han sido explicados con más detalle ambos componentes) tal y como muestra la siguiente figura:




Cuando se le da valor a la propiedad *Target*, en el diagrama principal aparecerán relacionados los siguientes elementos:

- La `Window` que contiene el componente de selección.
- La `Selection Window` indicada en la propiedad *Target* del botón de búsqueda.

Sketcher muestra esta relación tal y como se puede ver en la siguiente figura:




Por contener un elemento Table permite mostrar más de un propiedad de las instancias seleccionadas (una propiedad por columna en la tabla).

Tags aplicados a este elemento:

- `Selection`

Para más información sobre las etiquetas que es capaz de reconocer el generador MOSKitt-codgen-OpenXava consultar la sección correspondiente, más adelante en este mismo manual.

# Lista completa de etiquetas que reconoce el generador

En este apartado vamos a examinar con detalle los elementos que componen cada una de las plantillas.

Es bastante específico por lo que no se aconseja en una primera lectura del documento.

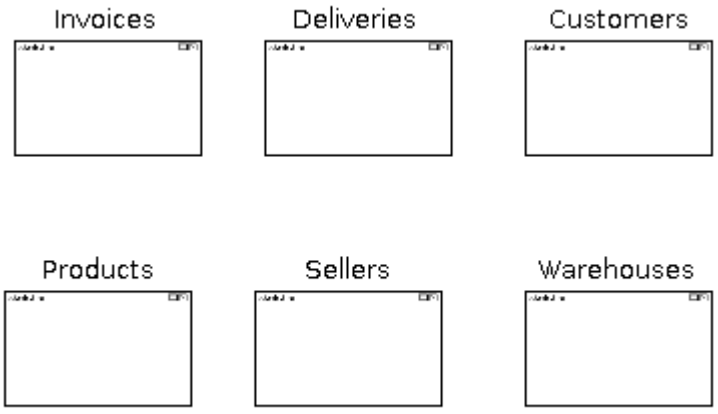
| Tag en Sketcher | Descripción | Elementos a los que se aplica el Tag en el Sketcher |
|---|---|---|
| `Tabular` | Identifica la disposición tabular de la información a mostrar. | Panel |
| Registry | Identifica la disposición en formato registro de la información a mostrar. | Panel |
| Detail | Identifica un Detalle de forma que su elemento contenedor se convertirá en su Maestro. | Panel |
| MultipleSelection | Identifica una disposición tabular de información para realizar una selección múltiple. | Panel |
| SingleSelection | Identifica una disposición en formato registro de información para realizar una selección única. | Panel |
| Title | Identifica a los títulos de los elementos gráficos. | Label |
| ReadOnly | Identifica cuándo un módulo es de sólo lectura, permitirá por defecto la impresión de los datos que muestra y la búsqueda de los mismos en la población de la clase involucrada. | Window |
| Selection | Identifica cuando una ventana en sketcher representa una selección y no un módulo. | Window |

| Tag en Sketcher | Descripción | Elementos a los que se aplica el Tag en el Sketcher |
|---|---|---|
| NotEdit | Identifica cuando en un formulario no es posible realizar la edición de los datos que en él se muestran. | Panel |
| NotInsert | Identifica cuando en un formulario no es posible crear instancias. | Panel |
| NotDelete | Identifica cuando en un formulario no es posible eliminar instancias. | Window |

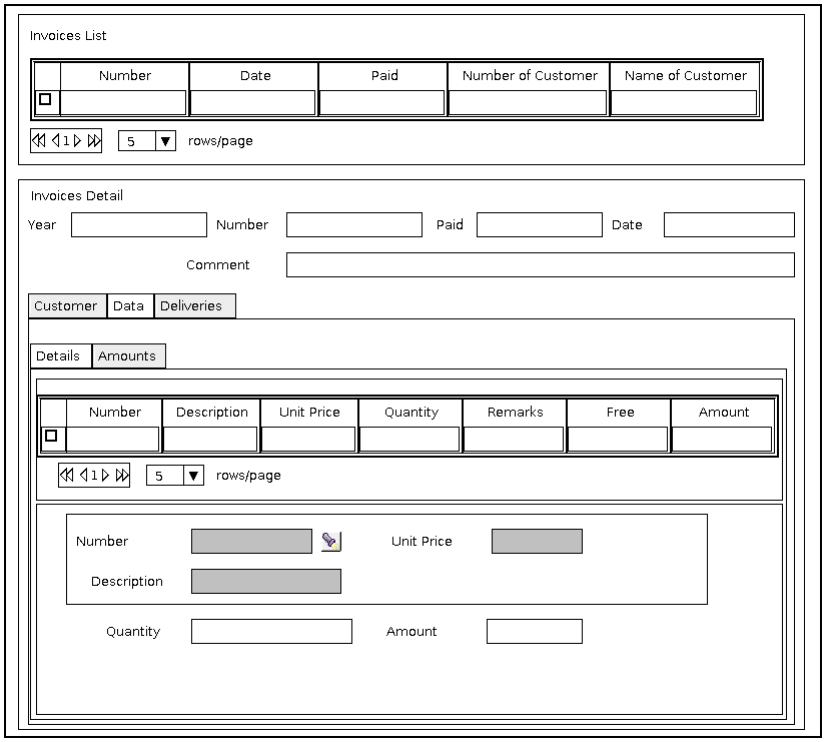
**Importante**: Para tener acceso a las Tags para OpenXava, es necesario iniciar el modelo de Sketcher con con las plantillas cargadas (ver sección sobre cómo crear un nuevo modelo de Sketcher).) o cargar el recurso que contiene las tags (`platform:/plugin/es.cv.gvcase.ui.templates.generic/models/sketcher_generic.sketcher`).

# Capturas de la aplicación de facturación con MOSKitt Sketcher

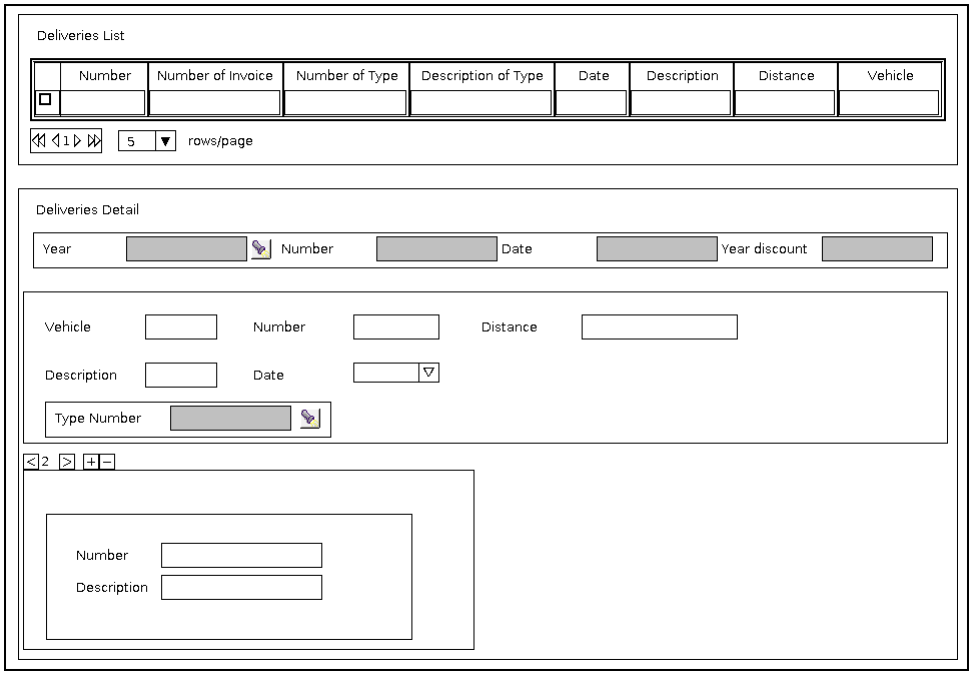
**Módulos de la aplicación**



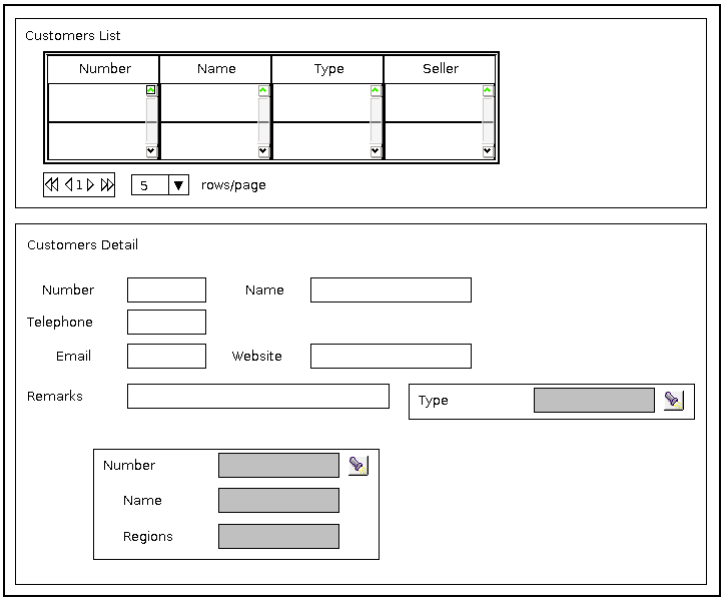

**Lista y detalle del módulo de Facturas**



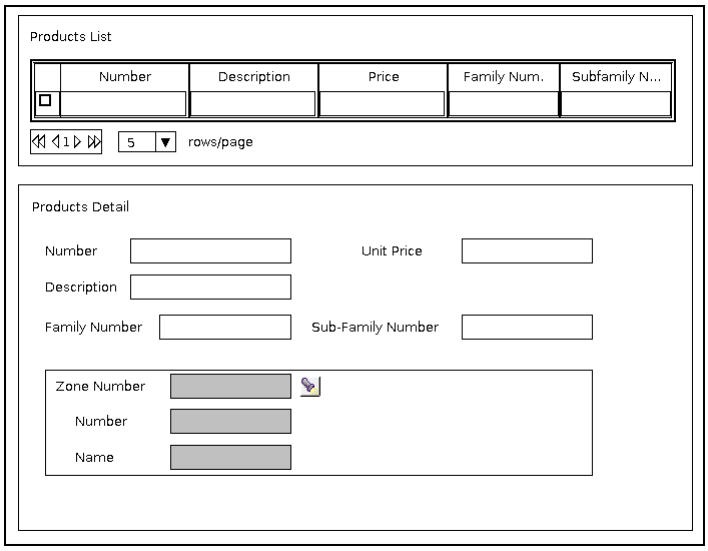

**Lista y detalle del módulo de Albaranes**



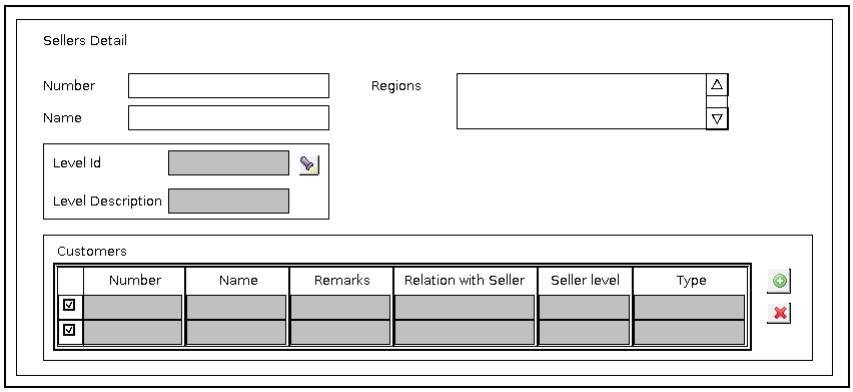

**Lista y detalle del módulo de Clientes**



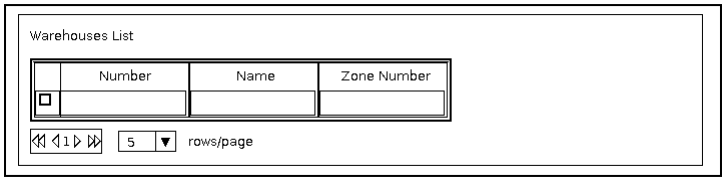

**Lista y detalle del módulo de Productos**




**Solo detalle del módulo de Vendedores**




**Solo lista del módulo de Almacenes**




# Otros escenarios: Cuando al empezar ya tenemos una Base de Datos

Volvemos ahora un poco al principio de la sección 3 en la que estamos describiendo cada uno de los pasos que se deben seguir para generar aplicaciones con MOSKitt. Hasta ahora hemos visto un camino en el proceso en el que al empezar no teníamos nada y empezábamos de cero a definir los modelos.

Sin embargo, pueden haber otro escenarios. En muchos casos es habitual disponer de una base de datos inicial a partir de la cual podemos llegar a obtener un modelo UML inicial para nuestro análisis. En este escenario los pasos a seguir para conseguir este modelo UML2 serán:

1. Obtener el modelo de BD a partir de Ingeniería Inversa de la Base de Datos física.

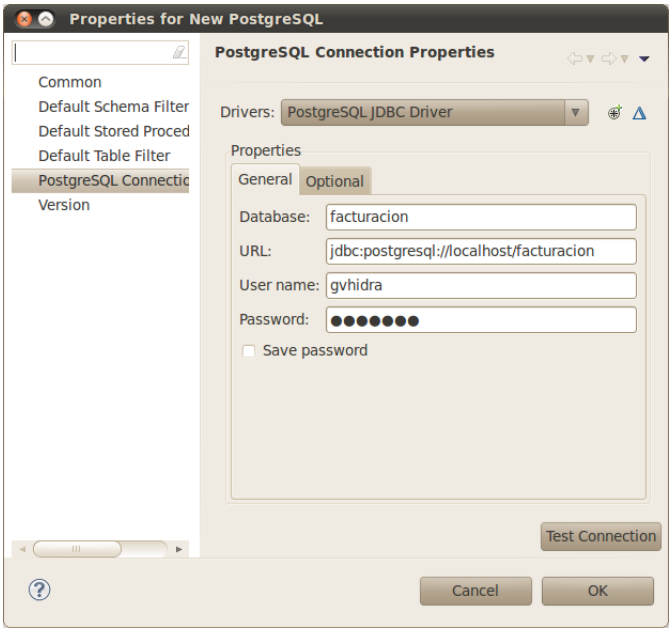
2. Lanzar la transformación BD2UML2 para obtener el modelo de UML2 inicial.

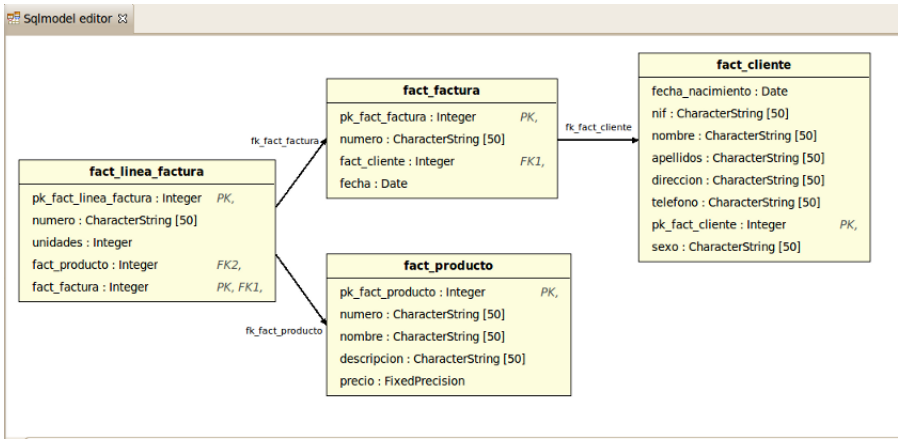
## Ingeniería Inversa de la Base de Datos

Es posible obtener el modelo UML2 correspondiente a partir de un modelo de base de datos resultado de realizar una ingeniería inversa (para más información consultar Manual Usuario de MOSKitt-DB).

Vamos a ver con un poco más de detalle este paso sobre el caso de estudio de la aplicación de facturación. Los pasos a seguir serían los siguientes:

1. En primer lugar es necesario crear en MOSKitt-DB, a través de la vista Data Source Explorer una conexión con la base de datos donde se encuentra el esquema del que queremos obtener la Ingeniería Inversa (vamos a suponer en este caso que se trata de una conexión a una base de datos local postgresql a través de JDBC).



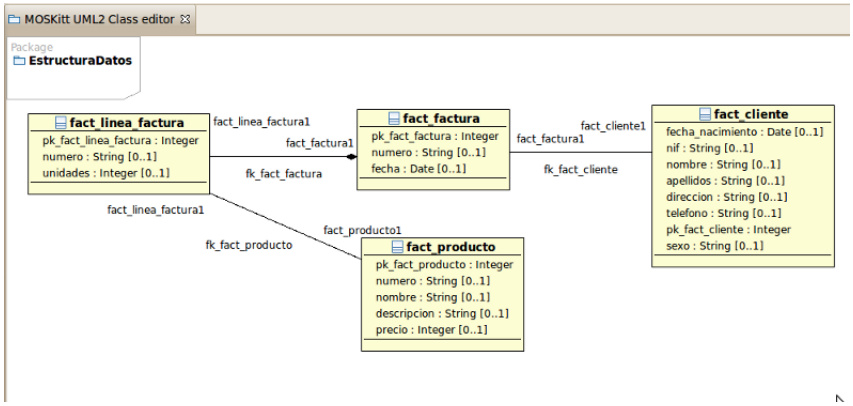



Nota: El nombre de las tablas es el que tienen en la base de datos, que en este caso incluyen un prefijo común.

## Transformación DB2UML : Obtener un modelo UML2 a partir del modelo de base de datos

A partir de un modelo de base de datos hay que construir un modelo de clases UML2. Esto se hará automáticamente a partir de la transformación de Base de datos a UML2 (DB2UML) (Ver Manual de Usuario para más detalles acerca de la transformación).

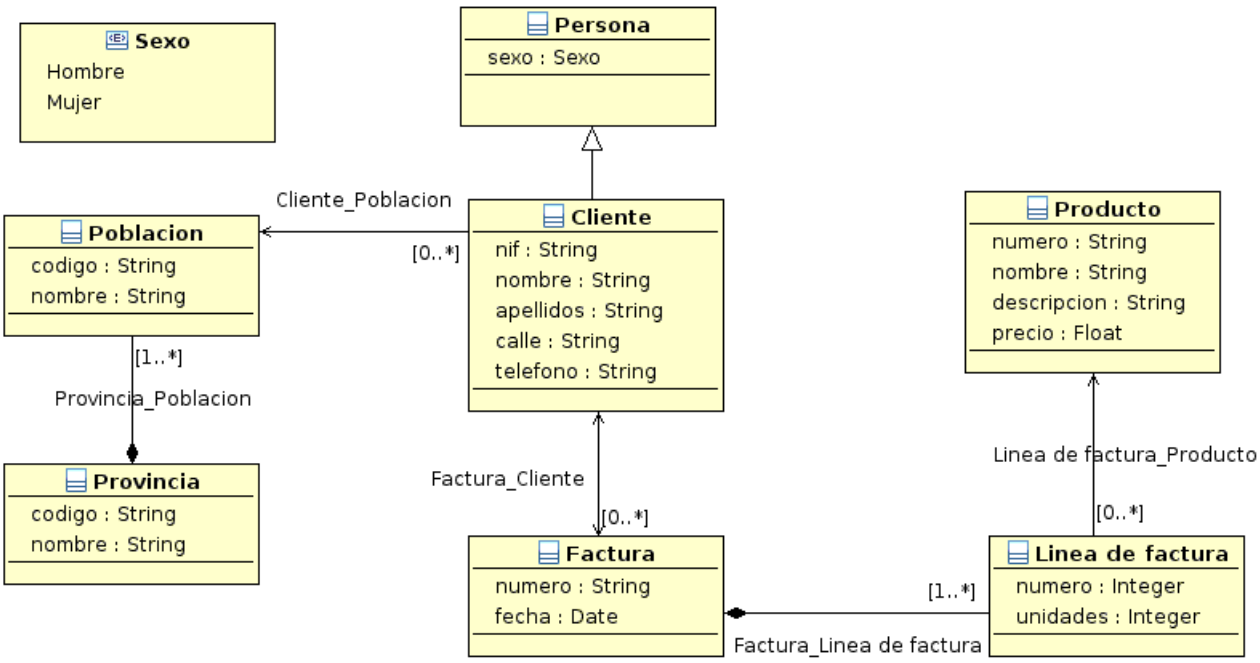
Como resultado se obtienen dos modelos, uno UML2 y otro que contiene las trazas entre los modelos de base de datos y UML2. ¡Atención!, el modelo de trazas puede estar oculto. El modelo UML2 será el utilizado para el análisis de la aplicación, mientas que el de base de datos y el de trazas son dispensables.



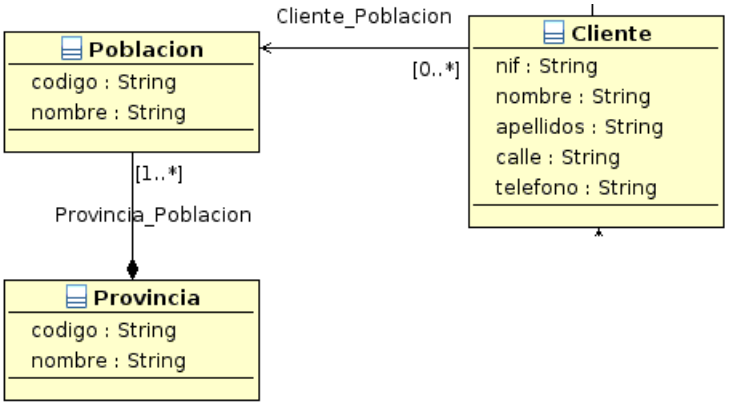

# Conceptos a tener en cuenta: Submodelo de Clases y Clase Principal

Toda ventana o panel lleva asociado un submodelo de clases formado por aquellas clases del modelo UML2 de las que se muestra alguna propiedad o se invoca algún método en esta interfaz.

Por ejemplo, supongamos el siguiente modelo clases para la aplicación de facturación:




Sin embargo, en el `Mantenimiento de Clientes` sólo vamos a mostrar información de las clases que aparecen en la siguiente figura:




Este sería el submodelo de clases que interviene en la interfaz de `Mantenimiento de Clientes`.

De todas las clases que pertenecen a cada uno de estos submodelos, siempre habrá una que será la que denominaremos **Clase Principal**, el resto de clases del modelo serán alcanzables desde ésta y participan en la interfaz por estar relacionada con la clase principal.

En ejemplo anterior, la Clase Principal es `Cliente` ya que en la ventana vamos a mostrar:

- De la clase `Cliente` las propiedades `nif`, `nombre` y `apellidos`.
- De la clase `Cliente::Poblacion` la propiedad `nombre` ( ¡ojo! aquí estamos indicando: "el nombre de la población del cliente" gracias a la navegación).
- De la clase `Clietne::Poblacion::Provincia` la propiedad `nombre` ( ¡ojo! aquí estamos indicando: "el nombre de la provincia de la población del cliente" gracias a la navegación).

Es importante observar que NO hemos dicho lo siguiente:

- De la clase `Cliente` las propiedades `nif`, `nombre` y `apellidos`.

- De la clase `Poblacion` la propiedad `nombre` ( ¡ojo! aquí estaríamos indicando: "De todas las poblaciones, su nombre" por no indicar navegación desde ninguna clase principal).
- De la clase `Provincia` la propiedad `nombre` ( ¡ojo! aquí estaríamos indicando: "De todas las provincias, su nombre" por no indicar navegación desde ninguna clase principal).

¿Donde indicamos la navegación? Como ya hemos visto anteriormente al explicar cómo modelizar la interfaz de usuario con el Sketcher, esa es una propiedad asociada a cada uno de los widgets del modelo del sketcher ( propiedad `Ruta Elemento Modelo`).