UNIVERSIDAD POLITÉCNICA DE VALENCIA
ESCUELA TÉCNICA SUPERIOR DE INFORMÁTICA APLICADA

Implantación de Metodología de desarrollo en departamento informático

PROYECTO FIN DE CARRERA

Autor
David López Carrasco

Director
Manuel Herrero Mas

Mayo 2010

Contenido

Agradecimientos

Este proyecto se ha dilatado en el tiempo mucho más de lo que me hubiera gustado, cuando tratas de compaginar trabajo y estudios normalmente hay un perjudicado y son los estudios. Y cuando tratas de compaginar trabajo, estudios y tu vida fuera de ellos, quien sale dañado es tu vida fuera de ellos. Por ese motivo he de dar las gracias a las personas que han permitido que durante tanto tiempo, haya estado entre el trabajo y el proyecto y dándoles de lado tantas horas.

A Manolo por aceptar dirigir este proyecto, ha sido un placer que una de las personas que más respecto te causa en el sector de la informática, sea quien dirija tu proyecto fin de carrera.

A Pedro, porque el trabajo que realicé hace años junto a él inspiró este proyecto y mucho de lo aprendido entonces está aquí dentro.

A mi compañero José, que a su particular manera me enseño gran parte de lo que sé hoy día.

A Esteban, por las horas de biblioteca en las que me ha acompañado, de no ser por eso, aun quedaría un largo período de tiempo para acabar.

A mis padres, sin su esfuerzo ahora no tendría la posibilidad de escribir estas líneas.

A Ana, por esconder su soledad bajo una sonrisa los días que he pasado en la Universidad.

A todos GRACIAS.

Capítulo 1. Introducción al proyecto

1.1. Motivación

Existen grandes organizaciones, tanto públicas como privadas, que poseen su propio departamento de sistemas de información. Estos departamentos están dedicados tanto al mantenimiento de los sistemas, como al desarrollo de nuevas aplicaciones para cubrir las demandas de los diferentes departamentos de la empresa u organismo.

Es bastante habitual, que el personal informático lleve mucho tiempo en la compañía y no se haya renovado tecnológicamente. Este problema normalmente suele ser causado por la reticencia de las empresas a dedicar tiempo de su personal a la formación en nuevas tecnologías. Una de las consecuencias del problema, es que se acaban empleando metodologías de desarrollo anticuadas, lenguajes de programación poco obsoletos y otro muy importante, en el caso de usar tecnologías cerradas, las empresas creadoras abandonan el soporte a los productos.

El auge de la programación orientada a objetos, y los problemas mencionados en el párrafo anterior, crean la necesidad de adaptar las aplicaciones de la organización a una nueva realidad, empezando por formar a sus empleados en estos nuevos lenguajes y metodologías ágiles de desarrollo.

1.2. Objetivo

El presente documento, pretende ofrecer un modelo de programación para organizaciones que deseen comenzar a desarrollar aplicaciones haciendo uso del lenguaje Java y abandonar lenguajes como Visual Basic 6.0, del que Microsoft ha finalizado su soporte.

Se intenta enseñar desde cero, a un programador experimentado en cualquier otro lenguaje, a crear aplicaciones con un nuevo entorno de desarrollo, se mostrará como instalar y usar las herramientas necesarias para crear una aplicación modelo en la organización. Una aplicación con mantenimiento de datos (inserción, modificación y borrado) y explotación de éstos (generación de informes).

De esto modo, siguiendo las directrices mostradas, se llegará a construir una aplicación básica de mantenimiento, basada en una arquitectura de tres capas.

Capítulo 2. Introducción a la arquitectura

2.1. Arquitectura multicapa

En las arquitecturas multicapa, se divide el sistema en diferentes módulos o capas dependiendo de su funcionalidad. Cada una de estas capas representa una funcionalidad claramente diferenciada de las demás.

Dependiendo de las arquitecturas, es posible dividir la aplicación en un número diferente de capas, a continuación se muestra la figura de la arquitectura que se va a presentar durante esta lectura, la **arquitectura en tres capas**.

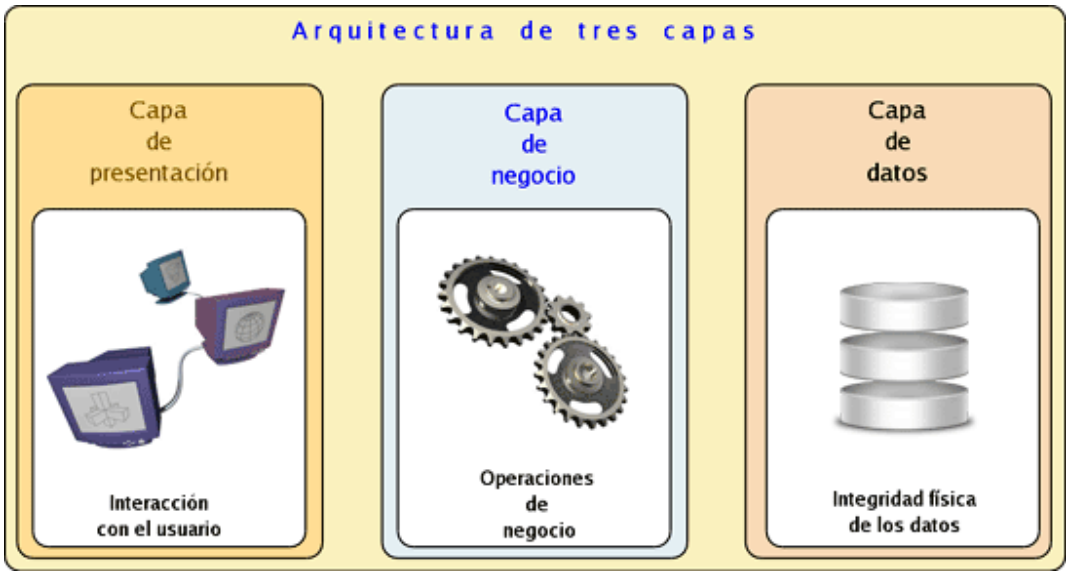


Figura 1. Diagrama de una arquitectura multicapa

En el caso de las arquitecturas basadas en tres capas, se habla de los siguientes niveles o módulos en la aplicación:

- **Capa de datos**: Representa el almacenamiento físico de los datos de la aplicación. Normalmente está implementada dentro de un SGBD[1], aunque pueden ser ficheros, otras aplicaciones, robots, etc.
- **Capa de negocio**: En esta capa se encuentra implementada toda la lógica de la aplicación. Es el nivel intermedio entre la capa de datos y la capa de presentación. Es aquí donde se llevan a cabo los cálculos o la transformación de datos entre otras operaciones. Quizás, sea el nivel que mayor carga de trabajo conlleva, puesto que el código desarrollado en esta parte, deberá

[1] Sistema de Gestión de Base de Datos

comunicar de un modo transparente a la capa de datos y a la capa de presentación.

- **Capa de presentación**: La encargada de presentar al usuario la información devuelta por la capa de negocio y de recoger los datos que se introduzcan en la aplicación. Representa a los formularios, informes, listados que el usuario ve durante su interacción con el sistema.

Para facilitar el trabajo y ahorrar tiempo en el desarrollo, ya existen herramientas y librerías que intervienen en la creación de cada uno de los tres niveles presentados. Más adelante se describe que herramientas o librerías ayudan en la implementación de cada una de las capas.

Capítulo 3. Introducción a las herramientas

3.1. Eclipse

Eclipse es un entorno de desarrollo de software (IDE) creado por IBM, principalmente escrito en Java y empleado también principalmente para el desarrollo de aplicaciones en Java. Mediante el uso de plugins se puede ampliar su funcionalidad para entre otras cosas, ser capaz de desarrollar en un diverso número de lenguajes de programación diferentes.

Actualmente, Eclipse es junto a NetBeans (Sun), la interfaz más popular entre los desarrolladores Java y se utilizará a lo largo de este proyecto para llevar a cabo el desarrollo.

3.2. Spring

Es lo que se conoce con el término de ***framework de trabajo.*** Este término es muy utilizado por los programadores y no tiene una fácil traducción. Se podría resumir en que es un esquema o patrón con el que trabajar. Spring propone el patrón MVC (Modelo-Vista-Controlador), es decir separar la aplicación en capas como se explica en el apartado anterior.

Se compone de muchos módulos que ofrecen multitud de servicios diferentes. A lo largo de los ejemplos se utilizará el framework para el acceso a datos (integrándolo con Hibernate), para la inversión del control (inyección de dependencias) y gestión de la transaccionalidad de la aplicación. También incluye módulos para mensajería, autenticación, acceso a LDAP, etc. Es muy recomendable profundizar en esta herramienta puesto que su uso está increíblemente extendido entre toda la comunidad de programadores Java e incluso a sido portado a otros lenguajes como .NET.

3.3. Hibernate

Es una herramienta o framework, que permite abstraerse como programador del motor de bases de datos que se desee atacar. Cuando se programa en un lenguaje orientado a objetos, el programador acaba pensando en clases, atributos y métodos y no en tablas, campos y consultar.

La herramienta permite mediante ficheros de configuración Xml, mapear o establecer la relación entre los objetos o beans de la aplicación y las tablas de la base de datos. Estos ficheros de configuración pueden ser generados mediante ingeniería inversa sobre la base de datos.

3.4. HSQLDB

Sistema de bases de datos relacional escrito integramente en Java. Se usa en muchos proyectos opensource, pero no tiene la robustez de otros motores para grandes aplicaciones como Oracle o IBM DB2. Habitualmente se utiliza en el mundo de la

programación como base de datos local para pruebas en fases tempranas del desarrollo. Goza de un amplio éxito gracias a su bajo consumo de recursos en la máquina del desarrollador.

3.5. PostgreSql

Base de datos relacional de código abierto muy extendida. Es habitual que se emplee en sistemas en producción, puesto que a diferencia de HSQLDB sí es un sistema robusto y se acerca a las prestaciones de los sistemas de pago. Es el sistema que se empleará para pasar a producción las aplicaciones que se desarrollen con la metodología a mostrar.

3.6. SqlExplorer

Es un cliente de bases de datos basado en Eclipse. Permite realizar consultas y explorar cualquier base de datos que disponga de driver JDBC. Se puede encontrar como plugin para embeber dentro del IDE de desarrollo o bien como aplicación independiente.

3.7. JUnit

Nuevamente, se habla de un framework, en este caso para realizar pruebas al código de la aplicación. Consiste en un conjunto de herramientas que permiten probar los métodos y clases de los que se compone la aplicación desde el mismo entorno de desarrollo entre otras formas de integrarlo. Se conocen como pruebas unitarias.

El uso de este tipo de testeo se ha vuelto muy popular, hasta el punto de que han aparecido metodologías de desarrollo ágiles cuyo punto neurálgico son las pruebas unitarias (Test Driven Driven Development).

3.8. SWT

El Standard Widget Toolkit es un conjunto de componentes para construir interfaces gráficas. Está creado por el proyecto Eclipse e intenta que las aplicaciones desarrolladas mediante su uso brinden al usuario el aspecto de su sistema operativo.

3.9. JasperReports

Es una herramienta de creación de informes libre escrita completamente en Java. Permite crear documentos que puedan ser fácilmente impresos o exportados a diferentes formatos portables.

3.10. iReport

Es la herramienta que permite diseñar informes en JasperReports de una manera sencilla y visual. Juntos se convierten en una potente herramienta de generación de informes como podría ser Crystal Reports.

Capítulo 4. Preparación del entorno de desarrollo

4.1. Instalación Java

En primer lugar, se debe de bajar el paquete de instalación de la web de Sun[2]. En el sitio web, se pueden encontrar multitud de empaquetados diferentes, como puede ser el entorno para ejecución exclusivamente, al kit de desarrollo, kits de desarrollo para empresas, paquetes que incluyen el servidor de aplicaciones de Sun, paquetes con NetBeans, etc. En el caso del desarrollo empleando eclipse, será necesario descargar la última versión del entorno el JDK sin extras.

La instalación del paquete es muy sencilla y por ello no se entrará al detalle de los pasos a seguir para su instalación. Únicamente se debe recordar el directorio de instalación para los pasos que se describen a continuación.

Es muy común en las herramientas de desarrollo, que se deba introducir ciertas ***variables de entorno*** en el sistema operativo y añadir algunas rutas al ***PATH***. En el caso de Java y algunas de las herramientas que se describen a lo largo del documento, no podía ser menos.

En el caso de la máquina virtual, se debe crear la variable de entorno ***JAVA_HOME***, cuyo valor corresponde con el directorio de instalación del paquete JDK. En el caso de una instalación sobre Windows este directorio se encuentra bajo el directorio "C:\Archivos de programa\Java\jdk-***version***". En consecuencia, la variable resultante es la siguiente:

Nombre Variable	**Valor Variable**
JAVA_HOME	C:\Archivos de programa\Java\jdk-version

Por otro lado, la variable de entorno PATH, ha de ser modificada para añadir el directorio con los ejecutables de la máquina virtual. El valor resultante de la variable PATH ha de ser el que se muestra a continuación

Sistema Operativo	**Nombre Variable**	**Valor Variable**

[2] Sun Microsystems – http://java.sun.com

Windows	PATH	%PATH%;%JAVA_HOME%\bin
Linux	PATH	$PATH;$HOME\bin

Para obtener más información acerca de cómo crear y modificar variables de entorno, se recomienda leer el **Anexo A** del presente documento.

Una vez realizados los pasos anteriores, únicamente queda comprobar que las variables han sido correctamente configuradas. Para ello se debe abrir una consola del sistema e introducir el comando "javac -version" (java compile).

4.2. Instalación Eclipse

La instalación de Eclipse es muy sencilla, y normalmente basta con bajar un paquete comprimido de la Web del proyecto y descomprimirlo en el disco duro del equipo donde se va a realizar el desarrollo.

En primer lugar, se debe obtener el software en la sección de descargas de la página Web de la fundación[3]. En la sección de descargas se pueden encontrar múltiples versiones del producto dependiendo del tipo de aplicaciones a desarrollar. Para el caso a exponer en este proyecto, se debe seleccionar la IDE (Entorno de desarrollo integrado) para desarrolladores Java.

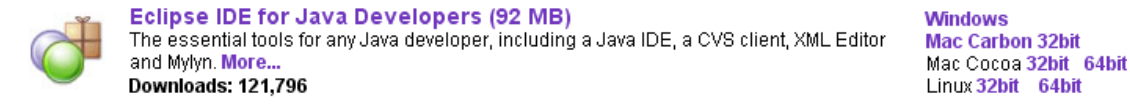


Figura 2 - Descarga de Eclipse

Se recomienda dedicar un directorio común a la instalación de todas las aplicaciones que se van a emplear en el desarrollo. Por ejemplo ***"C:\Java"*** en el caso de

[3] Eclipse Foundation: http://www.eclipse.org/downloads

Windows o ***"/home/user/java"*** en el caso de Linux. Tras la descarga de Eclipse, se debe descomprimir la aplicación en el directorio elegido para el software desarrollo.

Tras ejecutar Eclipse, se solicita la creación de un *workspace*. Al igual que se recomienda agrupar las aplicaciones bajo un directorio común, se recomienda agrupar los diferentes workspaces dentro de un mismo directorio bajo la ruta elegida para las aplicaciones. Esto facilita la rápida localización del código y de las aplicaciones cuando se desea acceder desde un explorador de archivos como "Dolphin" o "Explorer".

4.3. Instalación de Plugins

Existen dos opciones a la hora de realizar la instalación de un plugin, dependiendo del desarrollo que haya realizado el autor.

Por un lado puede que la instalación sea manual y por otro lado, se puede realizar una instalación desde la gestión de plugins que integra Eclipse.

En el caso de la instalación manual, normalmente se debe copiar el plugin dentro de la carpeta de instalación de Eclipse, bajo la carpeta ***plugins***. Se recomienda siempre leer la documentación de instalación ya que puede que aparezcan ligeros cambios a esta norma.

En el caso de la instalación a través del IDE de desarrollo, se ha de añadir el site del actualización del plugin a Eclipse. Se recomienda siempre que sea posible, realizar este tipo de instalación puesto que en caso de aparecer actualizaciones del plugin, el propio entorno avisará de que hay disponibles actualizaciones y podrán ser descargadas directamente.

A continuación se puede ver como realizar ambos tipos de instalación, ya que debido a los plugins seleccionados para la metodología de desarrollo se van a realizar ambos tipos de instalación.

4.3.1 M2Eclipse (Maven para Eclipse)

Este plugin permite trabajar desde Eclipse con la herramienta Maven, sin necesidad de emplear la línea de comandos de la consola. El corazón de Maven es un fichero de configuración en XML, el **pom**. Gracias al plugin, se puede editar el fichero desde la propia interfaz de desarrollo y haciendo uso de herramientas gráficas, lo que permite a las personas menos diestras con Maven trabajar sin necesidad de conocimientos extensos de la herramienta.

El plugin permite la instalación a través del asistente de Eclipse, con lo cual se va a realizar de este modo.

En primer lugar, se debe seleccionar la opción de "Instalar nuevo software" desde el punto de menú "Ayuda". Al seleccionarla, aparece la ventana que muestra en la figura:

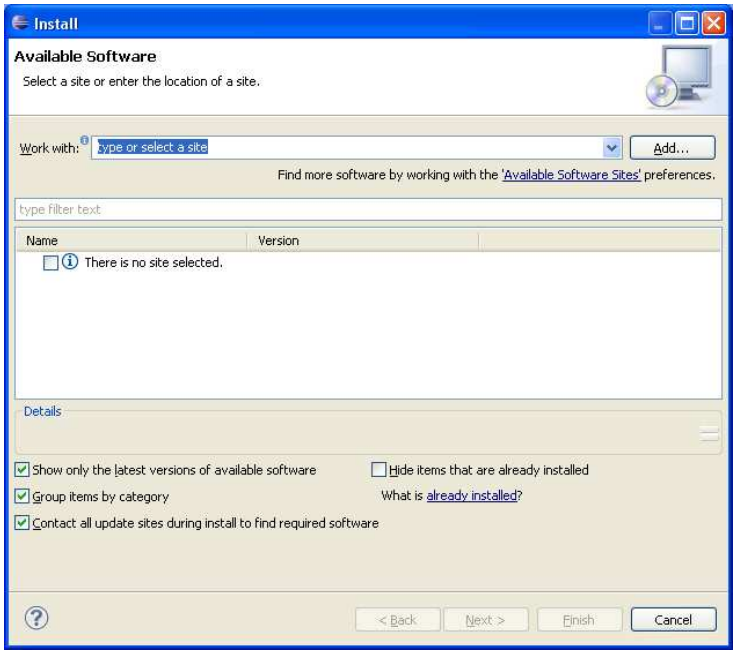


Figura 3 - Instalación de plugins en eclipse

Ahora se debe añadir la URL del site de instalación del plugin, la cual se ha de buscar en la página[4] de desarrollo del plugin. Una vez obtenida la dirección de instalación, pulsar sobre el botón "Add". En el diálogo que se obtiene a continuación se ha de dotar de un nombre a la nueva localización, además de la ruta. Se puede elegir el nombre de la figura que se muestra continuación.

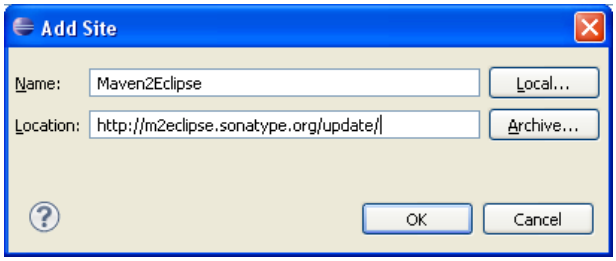


Figura 4 - Añadir Site de instalación de plugins

Al aceptar el cuadro de diálogo, Eclipse se encarga de explorar el site en busca de software disponible para instalar. Tras recopilar la información necesaria, se muestra una ventana donde seleccionar las opciones a instalar. Para el tipo de desarrollo que se propone en este documento, basta con seleccionar los componentes de integración básicos e ignorar los componentes opcionales. La figura mostrada tras estas líneas representa la configuración propuesta, en la que se omite la documentación del producto.

[4] Sonatype: http://m2eclipse.sonatype.org/

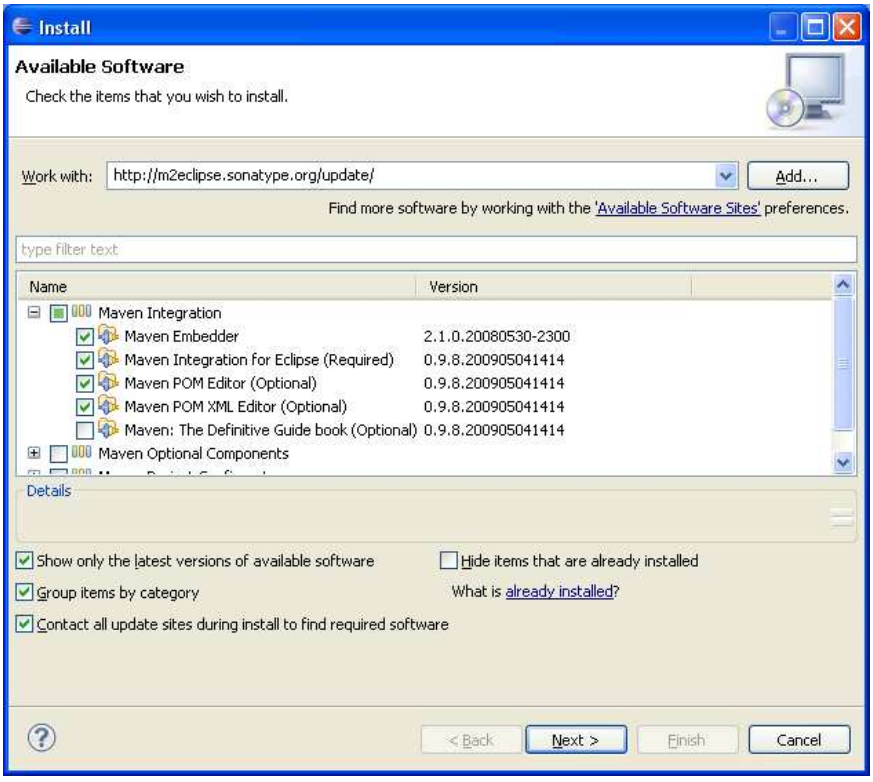


Figura 5 - Plugins disponibles para instalación

4.3.2 JUtils

JUtils, es una extensión de código abierto orientada a la generación de métodos toString. Este tipo de métodos, a priori no parecen útiles, ya que en la programación de aplicaciones de mantenimiento, raramente se encuentra en la capa de presentación datos provenientes de este tipo de métodos. Pero sí son especialmente útiles a la hora de depurar la lógica de la aplicación contenida en la capa de lógica.

En este caso, la instalación del plugin es manual. Para ello se ha de descargar el software la página del producto[5]. Tras la descarga, se obtiene un archivo comprimido, cuyo contenido se ha de copiar a la carpeta plugins de eclipse.

Para finalizar la instalación, se debe reiniciar el entorno de desarrollo, con el fin de recargar los plugins existentes.

[5] http://eclipse-jutils.sourceforge.net/

4.3.3 Azzurri Clay Mark

Permite trabajar desde el entorno de eclipse en el diseño de la base de datos de la aplicación de manera gráfica. De este modo, se simplifica la tarea de creación de la base de datos, ya que mediante esta herramienta, se puede generar el script necesario para crear la base de datos independientemente del SGBD elegido.

La instalación se realiza desde el entorno de eclipse, del mismo modo que m2eclipse, cuya instalación se explica previamente a estas líneas.

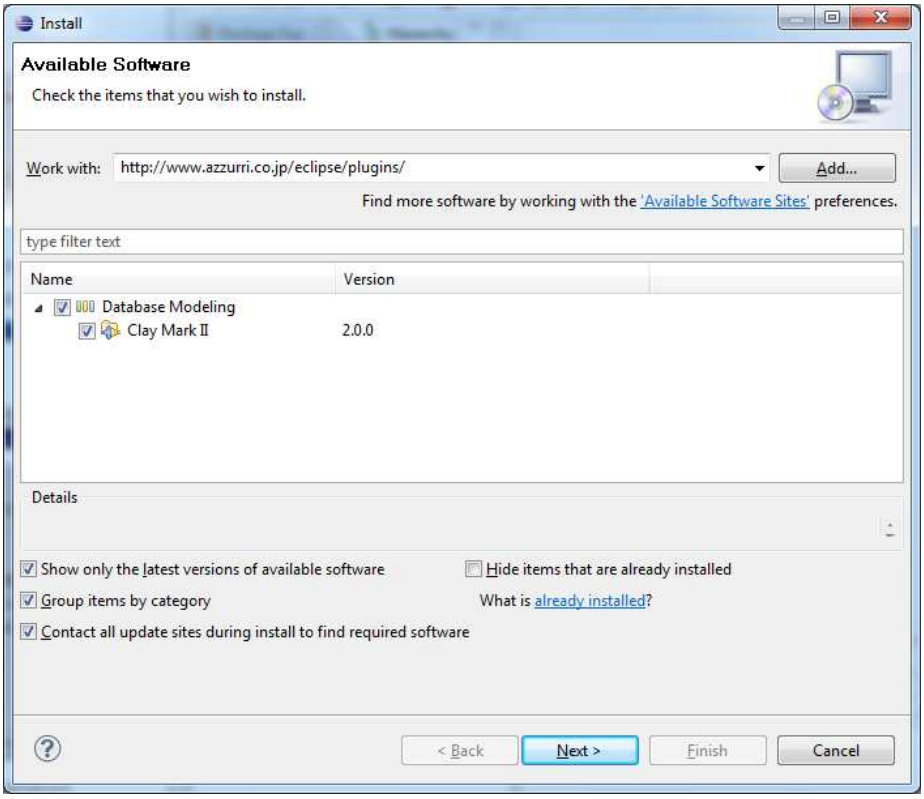


Figura 6 - Instalación de Azzurri Clay

El site de instalación, se puede encontrar fácilmente en el sitio web[6] de la empresa japonesa que lo mantiene.

[6] http://www.azzurri.co.jp/

Capítulo 5. El primer proyecto en Eclipse

En el caso del desarrollo de software, normalmente siempre vale más un ejemplo que mil palabras. Por ello, a lo largo del documento, se sigue un pequeño ejemplo que abarca todas las fases del desarrollo y que cubre la gran mayoría de las situaciones o problemas en los que se puede encontrar el lector a la hora de desarrollar su propio proyecto.

El ejemplo se basa en una aplicación de facturación, donde se cubren las operaciones relativas al mantenimiento de las facturas, desde su creación/edición hasta su impresión en forma de informe.

5.1. Creación del proyecto

La tarea de creación de la estructura inicial, la debe realizar necesariamente una única persona del equipo de desarrollo. No es un requisito que sea la misma persona para todos los proyectos de la organización, es decir, no hay porque tener una persona en el equipo especializada en esta tarea. Esta necesidad nace de la orientación al trabajo en equipo en un entorno compartido mediante **Subversion**.

Gracias a los plugins disponibles en la actualidad, la tarea de crear la estructura inicial adecuada para **Maven** es realmente sencilla. Simplemente se deben realizar unas cuantas pulsaciones de ratón para llevarla a cabo.

En primer lugar se deberá ir al menú superior y seleccionar la opción ***File>New>Other…***

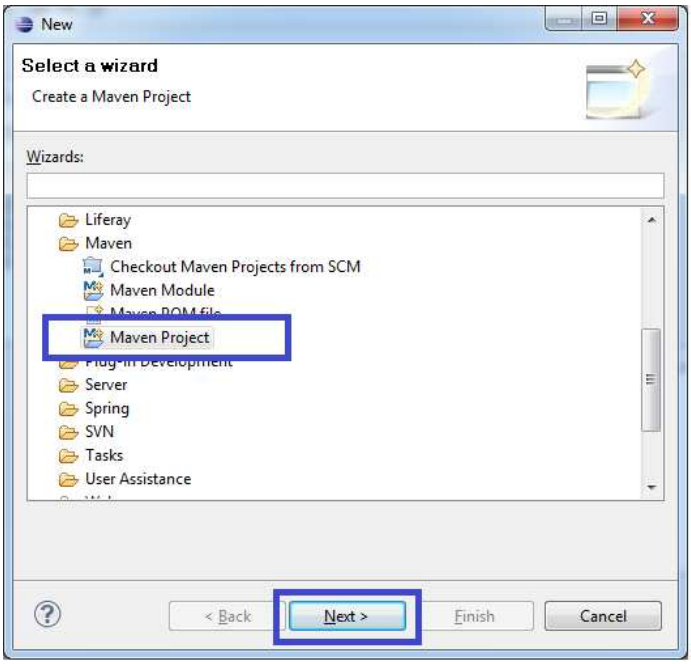


Figura 7 - Diálogo para nuevo proyecto

En el cuadro de diálogo de selección, se deberá marcar "***Maven Project***" y seleccionar siguiente.

En el caso de no seleccionar la opción de un proyecto simple, se solicita al usuario posteriormente un arquetipo de proyecto. Según el "archetype", se crea una estructura u otra diferente. En el caso del proyecto que se va a mostrar, con seleccionar la opción de crear un proyecto simple, es suficiente. Posteriormente se podrá añadir las opciones necesarias y modificar la estructura a las necesidades concretas del tipo de proyecto que se va a desarrollar.

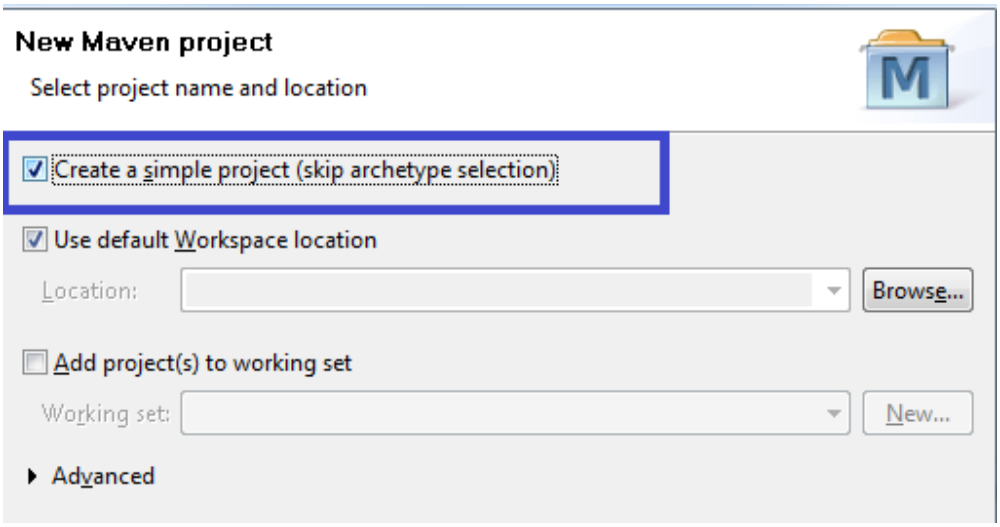


Figura 8 - Nuevo proyecto Maven

El siguiente y último paso, para finalizar con el asistente, es rellenar el asistente que generar el fichero POM de la aplicación. Toda la configuración de Maven, reside en este fichero y mediante los asistentes, los usuarios menos expertos no deben encontrar demasiados problemas en su uso.

En lo referente a Maven, se podría considerar un proyecto o una librería como un "artifact". Los datos que se rellenan en el asistente tienen que ver con cómo se conoce al paquete resultante en el argot Maven. Los datos a rellenar son los siguientes:

Group Id: Normalmente tiene que ver con la empresa o equipo que desarrolla el paquete o el proyecto. Si el sitio web de la empresa es www.dalocar.es (sitio ficticio). Un id adecuado es el siguiente:

- es.dalocar.software
- com.ibm.db2 (Empresa + producto)

Artifact Id: Está relacionado con el nombre concreto de la librería o del proyecto en sí, existen casos/desarrolladores, en los que incluyen el groupid de nuevo. En este

caso, en primer lugar se va crear un proyecto para gestionar toda la lógica de la aplicación. De este modo un **artifactId** válido puede ser **facturas-business**.

En cuanto a la **Versión**, se dejará el texto que viene por defecto en el asistente, 0.0.1-SNAPSHOT. En Java, está sería la versión inicial de cualquier proyecto. Snapshot indica que es una versión que todavía está en desarrollo. Maven guarda, a la hora de empaquetar, la fecha y la hora en la que se ha realizado.

Cuando un proyecto o librería alcanza su madurez y es estable, se puede quitar del POM la coletilla de Snapshot e ir incrementando la numeración de la versión en cada actualización. La primera cifra se emplea para cambios muy importantes en la librería y que realmente difieren sustancialmente de la versión anterior. Para actualizaciones con pequeñas mejoras o corrección de errores, se incrementan las cifras a la derecha del primer punto.

En cuanto al empaquetado o **packaging,** se debe seleccionar la opción de jar de entre las opciones propuestas por el asistente. Con esta opción lo que se consigue es que cuando se realice el empaquetado la librería o proyecto quede contenido en un archivo jar.

Name y **Description,** como el nombre indica, son para dotar al proyecto de nombre y de una descripción más exhaustiva que el propio título, donde indicar la funcionalidad y finalidad del proyecto.

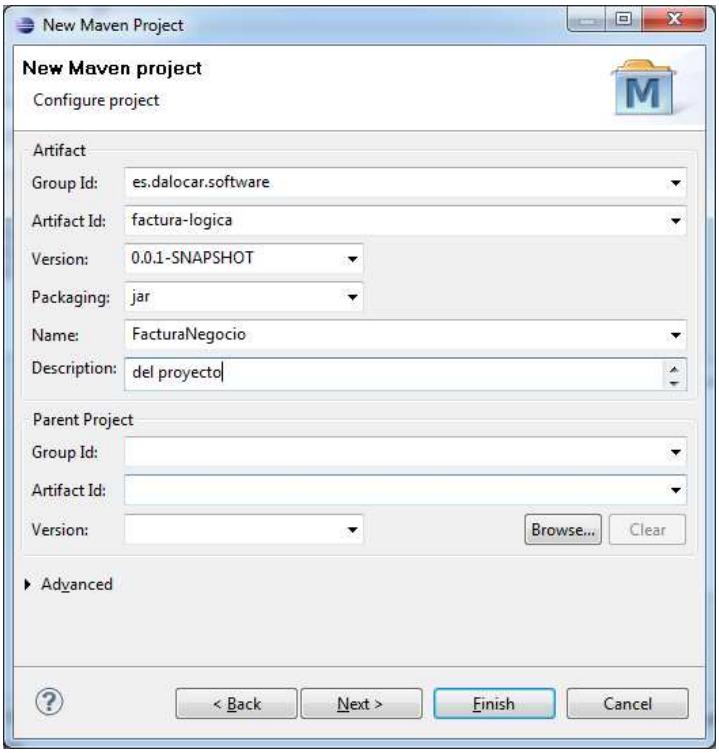


Figura 9 - Nuevo proyecto Maven, configuración

Tras **finalizar** el asistente, se puede encontrar el proyecto generado en eclipse con la estructura de carpetas necesaria y el fichero **pom.xml** necesario para el trabajo con Maven.

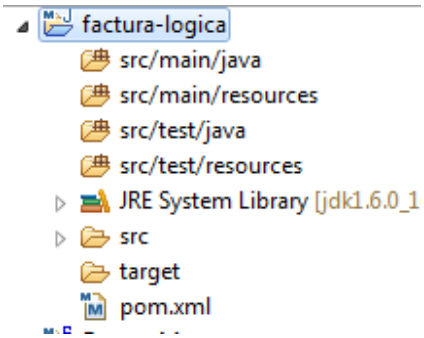


Figura 10 - Vista Package Explorer

- src/main/java

  Bajo este "Source Folder", se alberga todo el código fuente (agrupado por paquetes) del proyecto a excepción de las clases cuya finalidad es lade implementar los casos de test de la aplicación.

- src/main/resources

  Bajo esta carpeta, deben colgar todos los recursos de la aplicación. Se entiende por recurso, todo aquel fichero a excepción de los .java, que es necesario para el correcto funcionamiento de la aplicación. Puede ser una imagen, un fichero de configuración, de mapeo de hibernate, ficheros de internacionalización, etc.

- src/test/java

  Bajo ella, cuelga todo el código fuente destinado a las pruebas unitarias de la aplicación.

- src/test/resources

  Bajo la carpeta, se encuentra todo recurso necesario para la ejecución de los test.

Además de los "folders" anteriores, se puede comprobar que también ha sido creado un directorio llamado target. Bajo este directorio, almacenará Maven los paquetes generados tras la ejecución de cualquiera de sus **goals**.

El fichero pom, debe contener código XML, muy igual en forma al siguiente:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>es.dalocar.software</groupId>
  <artifactId>factura-logica</artifactId>
  <name>FacturaNegocio</name>
  <version>0.0.1-SNAPSHOT</version>
  <description>Este proyecto contiene toda la operativa y beans con la lógica de negocio
del proyecto</description>

</project>
```

Con estos sencillos pasos, la estructura del proyecto queda creada. Con la evolución de los plugins para integrar Maven con Eclipse y la expansión de Maven entre los desarrolladores Java, la creación de proyectos de este tipo en el IDE escogido, se ha vuelto muy sencilla y transparente al usuario, que antiguamente debía de configurar manualmente el proyecto creando las carpetas y el fichero **pom** manualmente.

A lo largo del documento, se realizan ciertos cambios en la configuración de Maven, como añadir nuevas dependencias o repositorios o configurar el empaquetado y despliegue de la aplicación. Estas tareas se tratan de manera más concreta en el apartado de la lectura actual correspondiente.

5.2. Subida del proyecto a Subversion

Como se comenta en el apartado anterior, la primera tarea después de crear la estructura inicial del proyecto, es la de sincronizar con el servidor de versiones (svn). Es de éste, de donde el resto de integrantes del equipo de desarrollo bajan el proyecto a su entorno para comenzar el trabajo en paralelo.

En este apartado, se enseña al lector los pasos a seguir para realizar esa primera sincronización del proyecto al sistema de control de versiones.

5.2.1 Crear la conexión

Para realizar la conexión al repositorio SVN, previamente se debe disponer de los siguientes datos:

- Nombre del proyecto (rama de código)
- URL de conexión al repositorio.
- Proyecto de Eclipse que que desea instalar.

La primera tarea a realizar en el entorno de Eclipse, es la de crear el repositorio SVN. Para lo cual, se debe cambiar la perspectiva de trabajo y seleccionar la perspectiva "SVN Repository Exploring". Desde la vista "SVN Repositories" haciendo clic derecho de ratón se puede obtener la opción "Repository Location".

Al seleccionar la opción aparecerá una ventana similar a la actual.

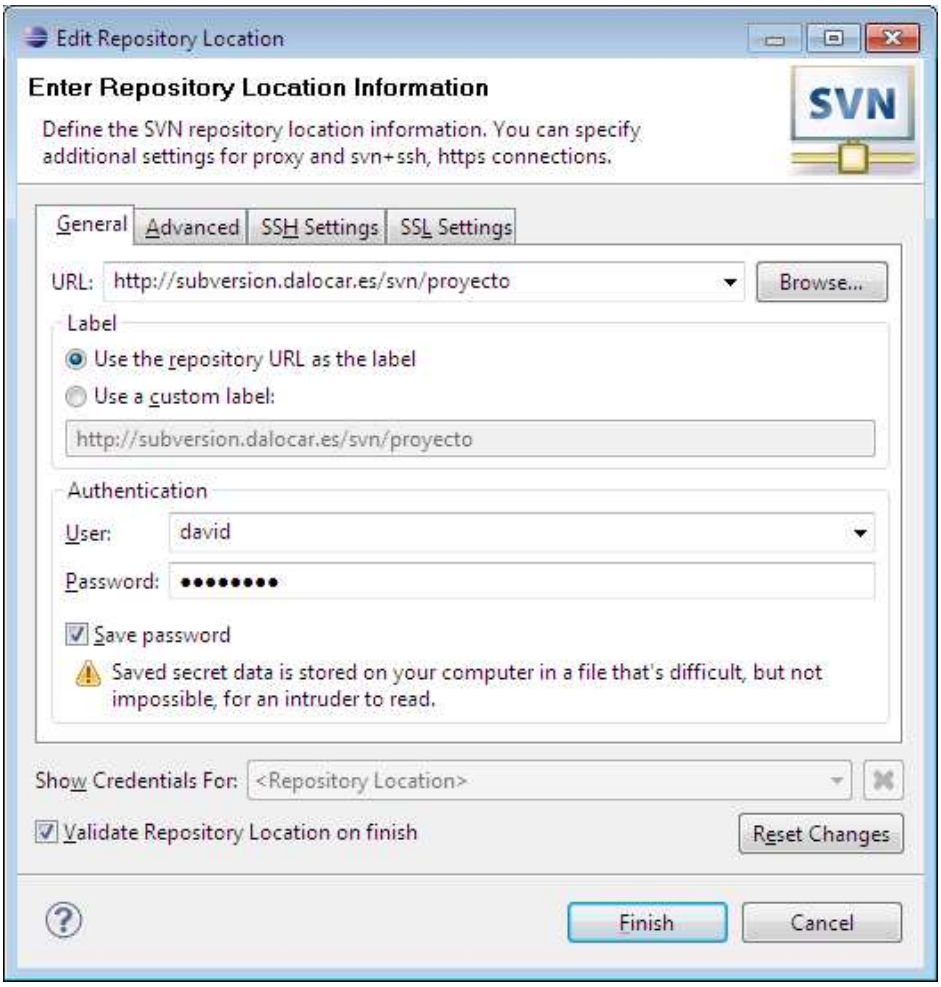


Figura 11 - Configuración de un repositorio SVN

Normalmente, el programador trabaja en un entorno controlado por un técnico de sistemas. El técnico de sistemas debe proporcionar al equipo la URL creada para bajo la cual cuelgan todos los proyectos que cree el equipo de desarrollo. Además de la dirección del repositorio, debe encargarse de la creación de usuarios y de dotar al usuario de su login y contraseña para un acceso seguro al repositorio.

En la ventana se muestran más opciones avanzadas, pero que de cara a trabajar con un repositorio SVN instalado de manera estándar, poca relevancia tienen de cara al programador.

5.2.2 Compartir el proyecto en SVN

Una vez creada la localización del repositorio (apartado anterior), el siguiente paso, pasa por compartir el proyecto en el servidor. Para ello, se debe volver a la perspectiva con la que se trabaja habitualmente, la perspectiva "Java".

Una vez en la perspectiva adecuada, se debe seleccionar el proyecto a compartir, y desplegar el menú contextual con el botón derecho del ratón. En el menú desplegado, se selecciona la opción "Share Project" bajo la categoría "Team".

En primer lugar se solicita al usuario el tipo de repositorio con el que se desea trabajar. En este caso y en adelante, SVN. Tras seleccionar la opción correcta, se debe pulsar sobre "Next" para ver la siguiente pantalla.

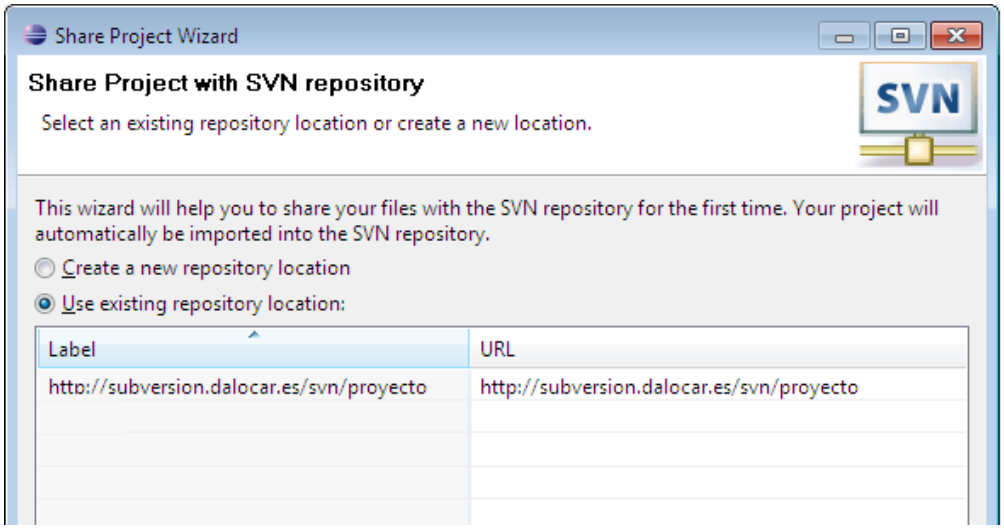


Figura 12 - Compartir proyecto SVN

En la siguiente ventana, aparecen las opciones relativas al proyecto y las que se recomiendan a la hora de compartir un proyecto en SVN.

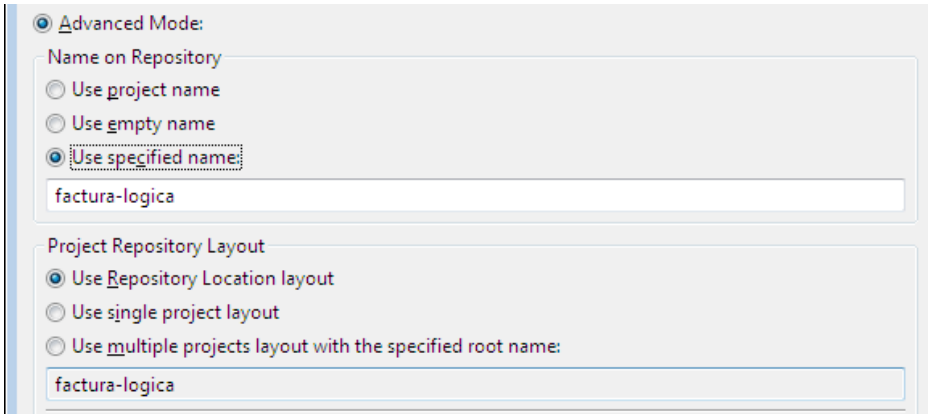


Figura 13 - Compartir proyecto SVN 2

El nombre en el repositorio, es importante ya que es el nombre que adopta el proyecto bajo la raíz del repositorio y más importante aún, es el nombre con el que normalmente el resto del equipo de desarrollo realiza la descarga del proyecto a su entorno de desarrollo.

La plantilla o layout de proyecto, es la opción que indica al servidor la estructura de carpetas a crear. Dejando seleccionada la primera entrada, "Repository Layout", se crean los directorios típicos de SVN: *trunk*, *branches* y *tags*.

Tras comprobar por última vez que las opciones marcadas son las correctas, se emplea el botón ***Next*** para acceder a la ventana de "Commit".

En este punto es interesante que el lector sepa de las tres operaciones básicas al trabajar contra un repositorio de código en SVN.

- **Commit**: La operación de subir código desde el IDE de desarrollo hacia el servidor. Únicamente se suben archivos más recientes o actualizados que los disponibles en el servidor.
- **Merge**: La operación Merge, permite mezclar el código local con el disponible en el servidor. El programador debe utilizar esta operación en los casos en los que se encuentra con **conflictos**. Con esta operación, se crea una nueva versión en el servidor incluyendo el código de las dos versiones en conflicto, normalmente debido al trabajo de dos programadores en paralelo sobre el mismo fragmento de código.
- **Update:** La operación para bajar versiones de código más recientes a las disponibles en local.

Lo que propone Eclipse a la hora de subir por primera vez el proyecto, es subir el proyecto completo con todos los directorios y ficheros. Esta opción **no es interesante**, puesto que existen una serie de directorios que no interesa que existan bajo el árbol del proyecto. Ante este problema, se presentan dos soluciones posibles:

1. Desactivar la opción "***Launch commit dialog***…" y realizar la subida de directorios posteriormente.
2. Dejar la opción y seleccionar en el siguiente diálogo las opciones que interesen.

Se recomienda la primera opción, debido a su sencillez y a que permite a usuarios no avanzados en SVN entender el funcionamiento. Tras seguir estas doctrinas, seleccionar la opción "***Finish"*** recordando desmarcar la opción vista en el párrafo anterior.

5.2.3 Subir los directorios del proyecto

En el momento de subir un proyecto por primera vez a Subversion, se debe tener en cuenta qué archivos y qué directorios, es interesante que residan en SVN y cuáles no. Es una práctica común a cualquier proyecto gestionado mediante un sistema de control de versiones concurrente, que no se sincronice todo el contenido del proyecto, si no aquello que es interesante controlar por un sistema de versiones.

Es decir, cualquier fichero susceptible de ser generado de manera indirecta por Eclipse, por Maven o por el proceso de compilación, no debe ser compartido en el repositorio. Aquí entran directorios como el ***target*** de Maven, dónde se compilan las clases Java y se guardan los empaquetados, documentación, etc. De Maven. Ficheros de configuración de Eclipse como el ***.project*** o ***.classpath*** o los directorios de settings de Eclipse.

El no subir este tipo de ficheros, hace que el proyecto se convierta en independiente del IDE y no se obliga a cualquier persona que desee descargar el código a trabajar con el mismo entorno que se emplea en esta bibliografía (Eclipse).

Antes de subir los directorios por primera vez al repositorio, estos aparecen junto a un interrogante, lo que indica que el servidor no tiene información acerca de ellos y que son nuevos para él. Esta información se la debe aportar el programador al realizar la operación de ***commit.***

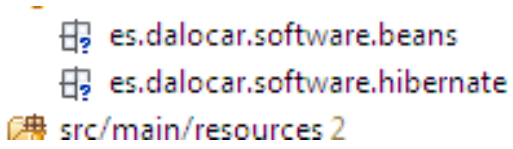


Figura 14 - Package explorer - Proyecto sincronizado

I

Para indicar a SVN que se ignore un determinado recurso, ya sea un directorio o un fichero, se debe seleccionar el directorio y con el botón derecho del ratón, elegir la opción "**Team > Add to svn:ignore**".

Figura 15 - Ignorar elementos en SVN

Al seleccionar esta opción, Eclipse pregunta al usuario que regla se quiere aplicar para el recurso, esto es así porque se permiten expresiones regulares para el nombre del directorio o fichero a ignorar. En este caso, se selecciona la primera opción, para ignorar únicamente mediante el nombre exacto y no mediante un patrón de expresión regular.

A continuación se detalla los recursos que deben ser omitidos a la hora de subir a **Subversion**, con el fin de ofrecer mayor libertad en la elección del entorno de desarrollo.

- El directorio **target**
- El directorio **Settings** y los archivos de preferencias que cuelgan de él.
- El fichero **.classpath**
- El fichero **.project**

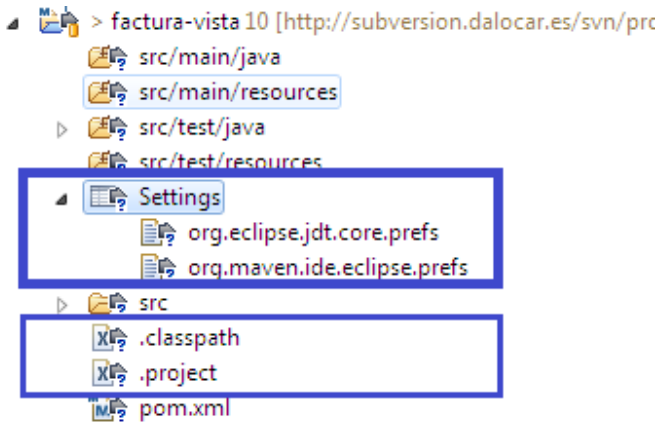


Figura 16. Lista de ficheros a omitir en Subversion

Una vez añadidos los diferentes recursos a **svn:ignore**, ya se puede proceder a subir el resto del proyecto a SVN.

Para ello se debe seleccionar el proyecto y mediante botón derecho buscar la opción etiquetada como "**Team> Synchronize with Repository**", la cual de manera habitual aparece la primera bajo el submenú.

La primera vez Eclipse siempre pregunta si se desea cambiar de perspectiva y emplear la llamada "**Team Synchronizing**", en caso de que esto ocurra, se debe responder afirmativamente.

El último paso, consiste en hacer "**commit**" de todos los elementos marcados con el signo "+". Eclipse muestra un cuadro de diálogo para facilitar la tarea y permite añadir un comentario para el registro de cambios. Finalizada esta operación, ya se puede volver a la perspectiva de desarrollo "**Java**" y permitir al resto de usuarios que descarguen el proyecto. Se puede observar en esta perspectiva, como el interrogante junto a los nuevos elementos ha desaparecido y ahora aparece un cilindro amarillo a la izquierda del recurso y un número que indica la versión a su derecha.

Capítulo 6. Diseño y creación de la base de datos

A lo largo de este capítulo se explicará al lector, las herramientas que intervienen y que ayudan en la creación de la base de datos. Esta fase del desarrollo, es clave en el futuro exitoso o no de la aplicación, ya que en gran medida el rendimiento de la aplicación depende directamente de un buen diseño.

Aquí no se darán las claves para un buen diseño de la base de datos, la elección de la estructura pasa por un buen modelo Entidad-Relación y su conversión al diseño lógico. De este tema el lector puede encontrar mucha y diversa literatura en las librerías, bibliotecas o en la red, más detallada y extensa que el actual documento no destinado a este tema exclusivamente.

6.1. HSQLDB - BBDD desarrollo local

HSQL es un sistema de gestión de base de datos muy liviano, el cual se recomienda para realizar pruebas en local (máquina del desarrollador). Cómo se comentó anteriormente su principal ventaja es la de ser fácilmente integrable con muchos de los plugins que amplían las funcionan con Eclipse.

6.1.1 Creación de la base de datos vacía

En un sistema de trabajo como la organización para la que se ha basado este texto, la arquitectura de la base de datos, viene dada por el responsable de base de datos de la organización. No está demás conocer herramientas como HSQLDB que facilitan la labor del programador, al poder trabajar en local sin interferencias de otros miembros del equipo de trabajo en sus pruebas de acceso a datos.

A continuación se dan los pasos a seguir para crear una base de datos vacía sin tablas. En apartados posteriores, se explica al lector como generar un script de creación de tablas de manera sencilla haciendo uso de una interfaz gráfica.

6.1.1.1 Descarga e instalación

Tras descargar y descomprimir la última versión estable[7] de HSQLDB en la máquina del desarrollador, se debe entrar en el directorio creado y crear una nueva carpeta para albergar la base de datos a crear.

Bajo este directorio, se ha de crear un archivo ejecutable (.bat o .sh) para el sistema operativo sobre el cual se trabaje. El contenido de este fichero ha de ser la orden mostrada a continuación:

[7] HSQLDB: http://hsqldb.org/ - Latest Stable version

```
java -cp ..\lib\hsqldb.jar org.hsqldb.Server -database.0 file:facturacion -dbname.0 facturacion
```

Este comando, únicamente realiza una llamada a la librería principal de hsqldb con una serie de parámetros. El tipo de instancia que se desea, el nombre de la base de datos y el nombre del fichero a crear para la base de datos. En el caso mostrado, se creará una base de datos llamada facturación en el momento de su primera ejecución.

6.1.1.2 Ejecución

Tras ejecutar el script creado por primera vez, se abre la consola de comandos que muestra la salida estándar del servidor y además se puede observar cómo se generan una serie de ficheros con el nombre dado al parámetro **file**.

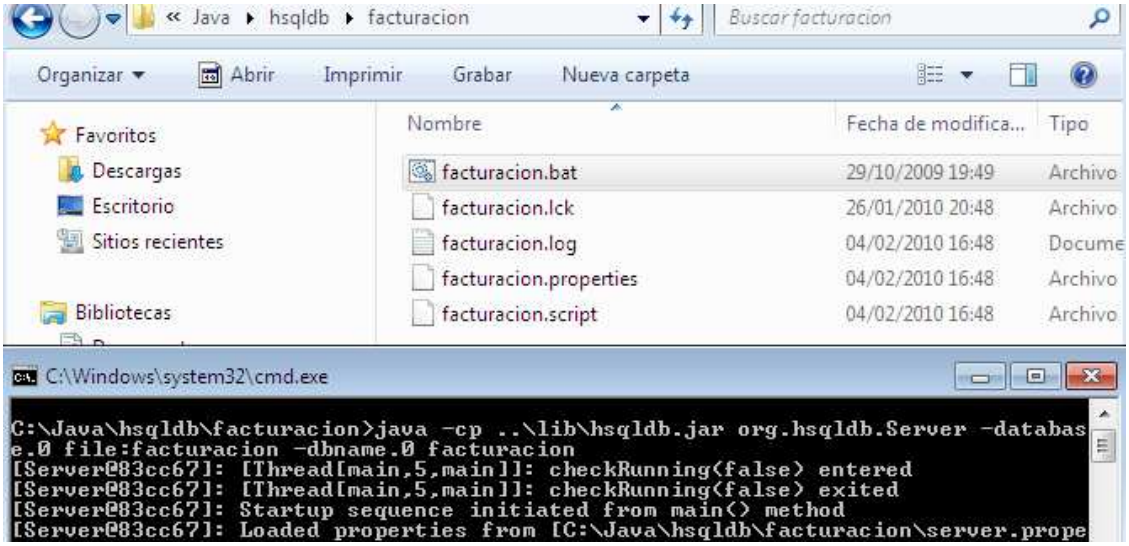


Figura 17 Primera ejecución de hsqldb

En lo sucesivo, cada vez que se desee arrancar contra la base de datos de test, se debe tener arracada a ésta. El modo de poner en marcha el servidor de bases de datos, es mediante el mismo script empleado para su creación. En la primera ejecución genera la estructura necesaria de ficheros y en las siguientes simplemente se aprovecha de ella.

6.1.1.3 Testeo

El siguiente paso es comprobar la conexión con la base de datos. Para ello se aconseja hacer uso de la herramienta SqlExplorer[8].

La instalación de la versión standalone, es muy sencilla, y basta con descomprimir el fichero que se descarga de la página web del proyecto.

[8] SqlExporer - http://www.sqlexplorer.org/ > Downloads

El primer paso para trabajar contra la base de datos desde este entorno es **muy importante** puesto que es el mismo para cualquier sistema de gestión de bases de datos contra el que se quiera trabajar. Este paso consiste en la instalación del **driver** para Java.

Dentro de la ventana de la aplicación, se debe seleccionar el subitem de menú **Preferences** bajo la opción **File.** En la ventana que se muestra a continuación se debe buscar la opción **JDBC Drivers.**

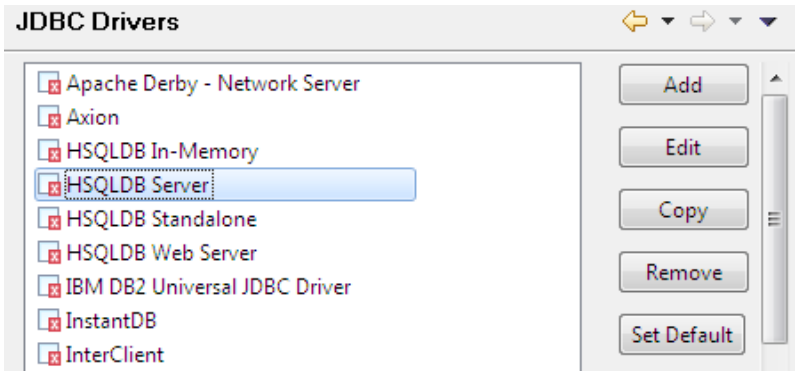


Figura 18 - Instalación drivers JDBC en SQLExplorer

La herramienta trae preconfiguradas una serie de configuraciones para diferentes bases de datos. Estas configuraciones tienen que ver con la cadena de conexión y el patrón que las define. Lo que no incluye el paquete de instalación es el driver para cada una de las configuraciones, esto es debido a que, aún siendo muy atractivo para el lector está opción aumentaría muchísimo el tamaño del paquete de instalación y por otro lado, existen drivers que no son de descarga libre o no se permite su empaquetado en terceras aplicaciones.

Tras la explicación, el siguiente paso a dar. Se debe seleccionar el driver mostrado en la figura anterior (HSQLDB Server) y pulsar sobre **Edit**.

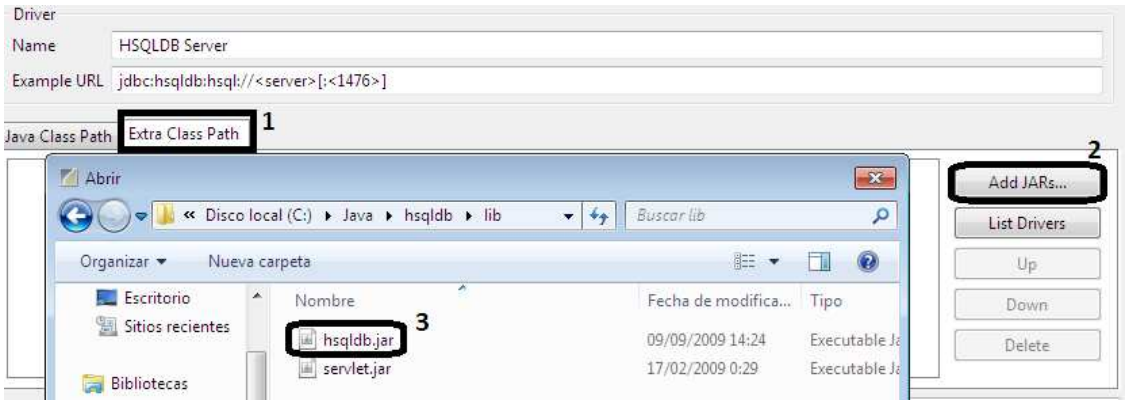


Figura 19 - Configuración del driver JDBC

Los parámetros **Name** y **Example URL**, permanecen intactos puesto que son acertados. Únicamente basta con indicar donde se encuentra el jar con el driver. Para ello, se debe seleccionar la pestaña **"Extra ClassPath"** y dentro del panel de la pestaña pulsar

el botón "**Add Jars**"**.** En el diálogo de selección de fichero, se deberá buscar el directorio donde se ha descomprimido HSQLDB y bajo él, localizar la carpeta **lib**. Bajo esta carpeta se encuentra el jar que contiene el driver adecuado para este tipo de base de datos.

Tras seleccionar el jar y cerrar la ventana, se debe pulsar sobre el botón **"List Drivers"**, con lo que se rellena el combo de la parte inferior. Una vez poblado, se selecciona la única opción disponible **org.hsqldb.jdbcDriver**, y se puede aceptar y cerrar las ventanas hasta llegar a la ventana principal de la aplicación.

Una vez configurado el driver, es el momento de crear una conexión contra la base de datos. La creación de conexiones se realiza en la vista de **Connections.** En esta vista se encuentra un botón acompañado de un símbolo de suma. Acto seguido a su utilización, se puede ver una ventana como la siguiente (sin datos) para configurar el perfil de conexión:

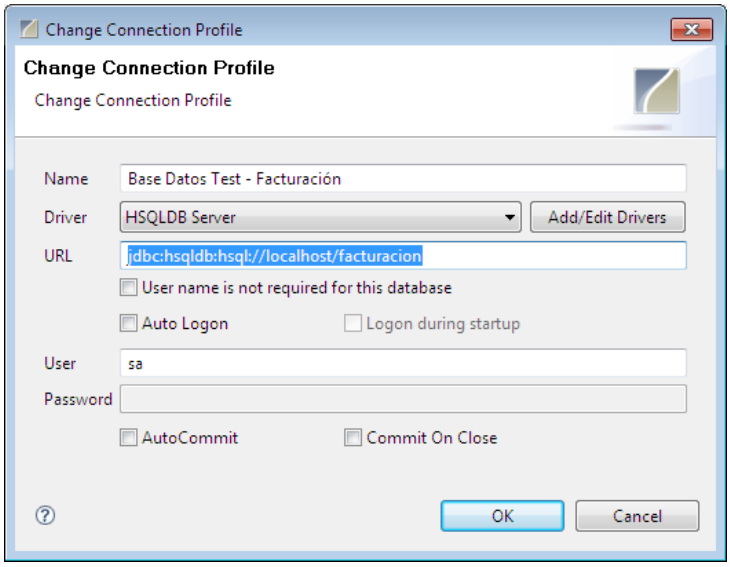


Figura 20 - Creación del perfil de conexión en SqlExplorer

El campo **Name** indica el nombre que se mostrará en la vista de conexiones para su identificación. En el desplegable de **Driver**, se selecciona el driver recién configurado. En cuanto a la **URL** de conexión se debe modificar por la siguiente:

jdbc:hsqldb:hsql://localhost/facturación

La última parte de la cadena de conexión ("facturación"), se corresponde con el nombre de la base de datos elegido en el script de arranque. Es resto de parámetros se mantienen como los muestra la herramienta, previa comprobación de que el usuario es "**sa**" y la contraseña no existe.

Tras guardar los cambios, haciendo doble clic sobre la conexión, se muestra el siguiente cuadro de diálogo.

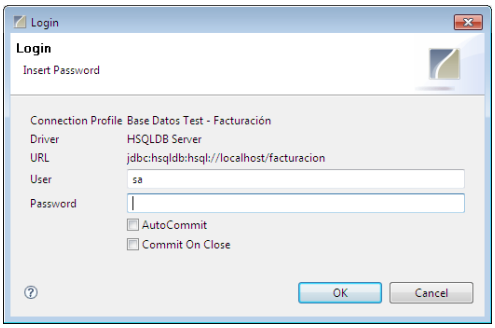


Figura 21 - Cuadro de diálogo para conexión a la BBDD

Al aceptar, se abre la conexión de la base de datos, y en la vista "**Database Structure**" se muestra la estructura de la base de datos. Esto significa que la conexión ha funcionado y en consecuencia, que la base de datos está correctamente creada y arrancada.

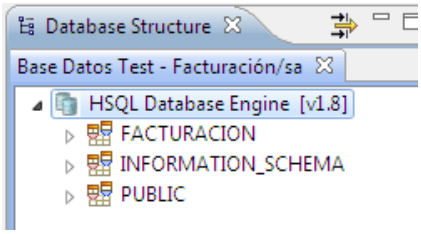


Figura 22 - Vista en SqlExplorer de la estructura

6.1.2 Diseño de la BBDD mediante herramientas gráficas

Creada la base de datos, es la hora de crear el esquema y tablas relativos al modelo de datos de la aplicación necesarios para trabajar. Para esta labor, se ha decidido la utilización de **Azzurri Clay Mark**, cuya instalación el lector ya conoce de apartados anteriores.

Lo primero, es crear un diagrama de modelado. Para ello, se selecciona "**New>Other…**" en el menú de eclipse. En el asistente se selecciona la opción "**Clay Database Diagram**".

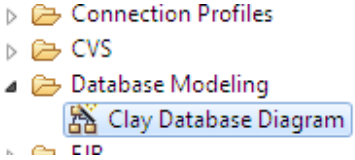


Figura 23 - Herramienta Clay en Eclipse

El asistente es muy sencillo y no merece la pena dedicar líneas a su detalle. Simplemente, se debe saber que durante el asistente, se solicitarán dos datos: un nombre

para el fichero a generar y un dialecto de base de datos. El fichero a generar lo puede elegir el usuario, se recomienda guardarlo bajo la carpeta **conf** del proyecto. Como dialecto se debe elegir HSQLDB 1.8, o la versión de éste equivalente a la versión que se ha descargado y configurado en el apartado anterior.

Tras finalizar el asistente se abre un espacio de trabajo donde se permite trabajar con tablas y relaciones de manera gráfica.

Se recomienda explorar todas las opciones disponibles mediante el botón derecho sobre el espacio reservado para el dibujo y el menú lateral. Su manejo es muy sencillo y dado que se presupone al lector, ciertos conocimientos de bases de datos, se da por sentado que no tendrá dificultad alguna en su manejo.

En primer lugar es recomendable, aunque no obligatorio, crear un esquema para en la base de datos para el proyecto.

Para ello, empleando el botón derecho del ratón y seleccionando la opción "**Edit Schemas**", se puede crear un nuevo esquema o mantener los existentes.

A continuación se muestra al lector el formulario de configuración para una de las tablas elegidas para el ejemplo, la tabla **Factura**:

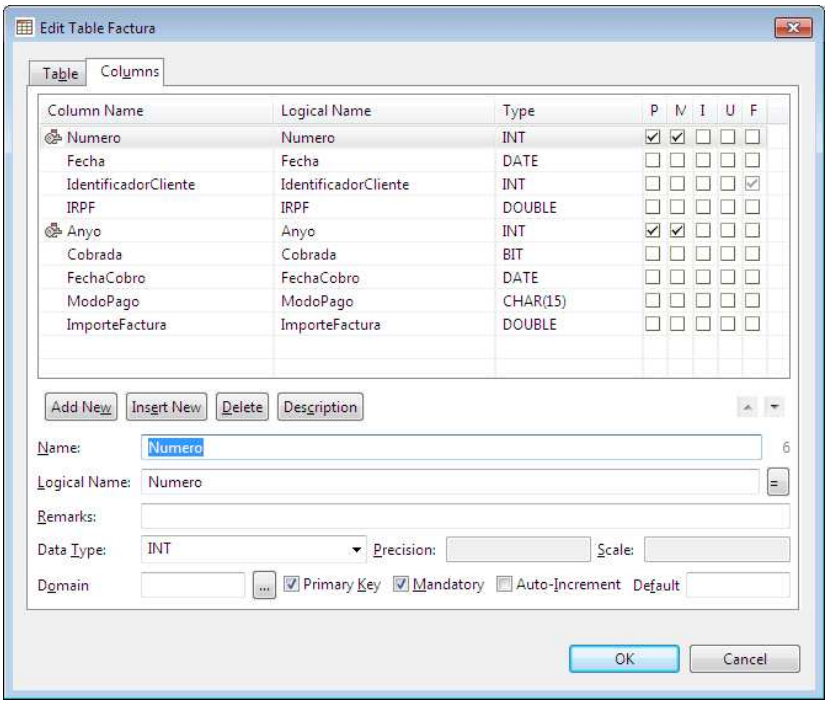


Figura 24 - Diseño de la tabla 'Factura'

A modo de resumen, en este asistente muestra dos pestañas. La primera permite el mantenimiento de los datos relativos a la tabla (nombre, esquema, etc) y el segundo permite elegir crear campos de la tabla eligiendo opciones como el nombre del campo, el tipo de datos, si forma parte del identificador de la tabla, la precisión, la longitud, etc.

Para la creación de relaciones, se elige la herramienta "**Foreign Key**" en el menú lateral. Esta opción, es muy fácil de reconocer por su icono . Con la herramienta seleccionada, se debe pinchar en primer lugar en la tabla "hija" y después en la tabla maestra para indicar el sentido de la relación. Tras crear la relación, se elige la herramienta de selección y se hace doble clic sobre la relación para mostrar la ventana de configuración de la relación.

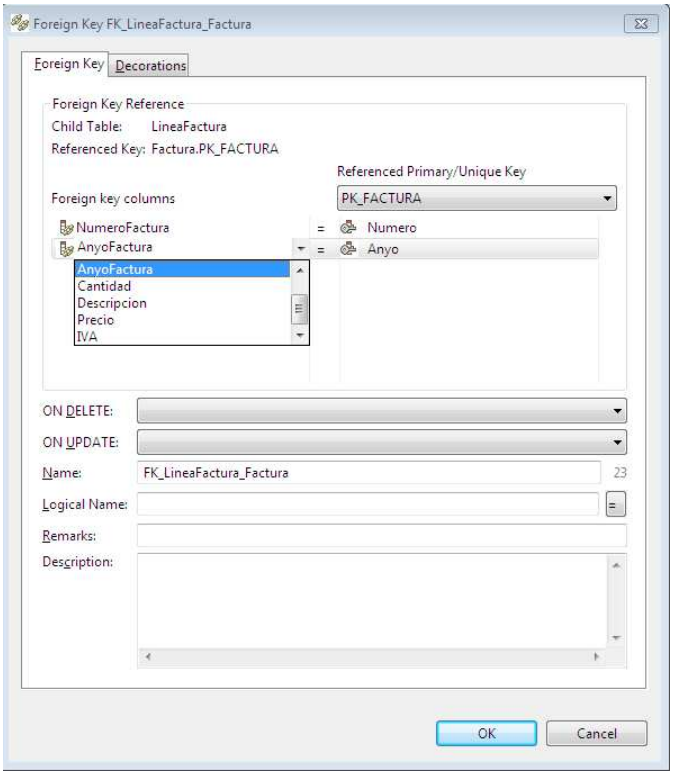


Figura 25- Configuración de la relación Factura - LineaFactura

Clicando sobre la columna "**Foreign Key/columns"** se debe elegir las columnas de la tabla implicadas en la relación.

Tras finalizar el diseño de la estructura, ya se puede proceder al siguiente paso.

6.1.3 Creación de la estructura de la BBDD

Finalizado el diseño de la base de datos, es posible generar un script de SQL, para la creación de las tablas. Para la creación del script, es necesario mostrar la vista "**Outline**", mientras el diseño se encuentra en la vista principal.

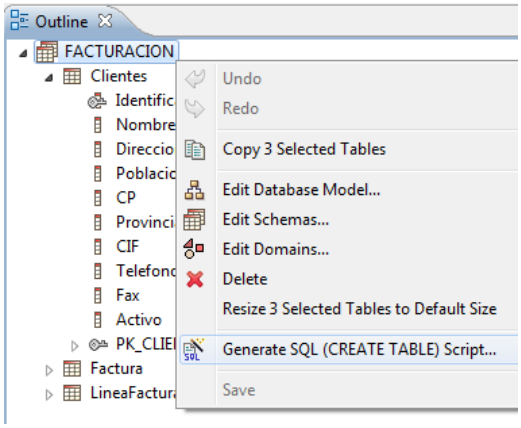


Figura 26 - Creación de las tablas desde Azzurry Clay

El plugin permite la exportación o la generación del script objeto por objeto (tabla por tabla). En este caso, se recomienda seleccionar el esquema completo para su creación. Para ello, con el esquema correspondiente seleccionado y mediante clic derecho del ratón, se selecciona la opción "***Generate SQL (CREATE TABLE)"***.

Al usar esta opción Eclipse muestra un cuadro de diálogo que queda a la espera de la ruta dónde almacenar el script y el nombre para el fichero. En este caso, se recomienda almacenar el fichero bajo la ruta **/src/conf/dev** del proyecto. Como nombre, se puede usar ***create_tables.sql*** por ejemplo.

En la configuración que aquí se recomienda, se emplea **dev** para almacenar los ficheros referentes al entorno de desarrollo y **prod** para el entorno de producción.

El siguiente paso, consiste en lanzar el **script** que acaba de generar el plugin, contra la base de datos. Para realizar esta operación, se puede emplear cualquier herramienta de gestión de base de datos. En este caso, se hace uso de la herramienta que se configuró anteriormente, ***SQLExplorer.*** Si se han seguido los pasos hasta este punto, se debería de tener creada conexión contra la base de datos de la aplicación.

Tras el arranque y apertura de la conexión, se debe abrir una ventana del **Editor de consultas.** Desde esta ventana se permite lanzar consultas escritas directamente en el editor o bien cargar scripts ***.sql***. En este caos se va a optar por la opción de cargar un fichero, el fichero que se acaba de crear a través del plugin de Eclipse.

Figura 27 - SQLExplorer, carga de un script

Tras pulsar el icono de abrir fichero, se debe buscar el fichero generado dentro del directorio del proyecto. Tras abrirlo, aparece en el editor, todo el código SQL generado.

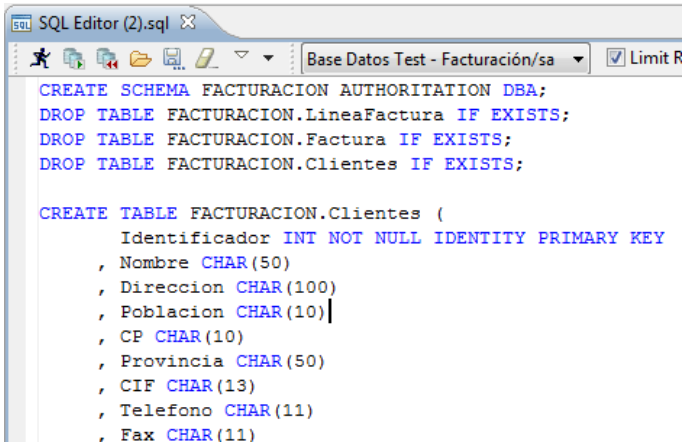


Figura 28 - Código generado

La ejecución del script se puede realizar mediante la combinación de teclas Ctrl+Enter o bien mediante el icono de ejecución  .

La ejecución crea la estructura de tablas necesaria, para comprobarlo, se debe visualizar la vista de "***Database Structure***", y refrescar la entrada ***Tables*** bajo el esquema **FACTURACION** *(Este será el esquema en el caso de que se siga el ejemplo al pie de la letra)***.**

6.2. Capa de Persistencia (Hibernate)

6.2.1 Introducción

Para facilitar el acceso a base de datos, se hace uso del framework Hibernate y Spring junto a algunos plugins que facilitan la generación del código necesario para el trabajo con el patrón DAO (Data Access Objects). En este apartado se realiza una introducción al trabajo con Hibernate y Spring, en la que se muestra mediante el ejemplo seleccionado los escenarios típicos del trabajo con base de datos, pero dónde no se cubre aspectos específicos que puedan aparecer al realizar aplicaciones complejas y para los cuales se recomienda la lectura de los manuales y tutoriales oficiales tanto de Spring como de Hibernate.

6.2.2 Dependencias necesarias

Para el trabajo con Hibernate y Spring, es necesario añadir al proyecto las dependencias correspondientes. Para añadir las dependencias, es necesario editar el fichero ***pom.xml.*** En las últimas versiones del plugin ***m2eclipse***, ya se permite la edición mediante interfaz gráfica, lo cual es interesante para los menos expertos.

La tarea de añadir dependencias es muy simple, de modo que no se va a detallar cómo realizar esta operación de manera gráfica y simplemente se dan las dependencias a añadir en el fichero.

En el caso de ***Hibernate***, se debe añadir la siguiente dependencia:

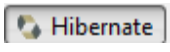
```
<dependency>
        <groupId>hibernate</groupId>
        <artifactId>hibernate3</artifactId>
        <version>3.2.3.GA</version>
</dependency>
```

En el caso de ***Spring***, se deben añadir las siguientes dependencias:

```
<!-- Spring -->
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>2.5.6</version>
</dependency>
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>2.5.6</version>
</dependency>
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>2.5.6</version>
</dependency>
```

Existen algunas dependencias transitivas (toda aquella dependencia no directa del proyecto), pero la ventaja que aporta Maven es que automáticamente se encarga de añadirlas al proyecto.

6.2.3 Configuración Hibernate

Tras añadir las dependencias necesarias y suponiendo que ***Hibernate Synchronizer*** ya está instalado en Eclipse, se debe cambiar la perspectiva por la perspectiva Hibernate 

.

En esta perspectiva, existe la vista "***Hibernate Configuration***", la cual permite crear nuevas configuraciones para realizar ingeniería inversa sobre la base de datos.

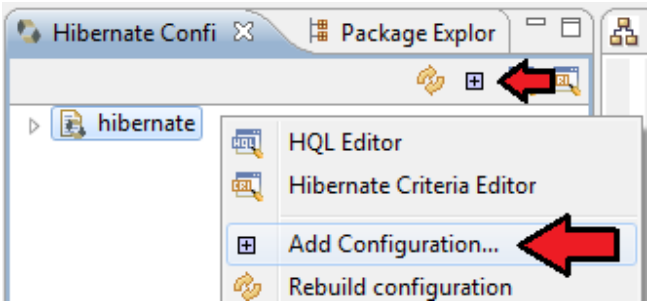
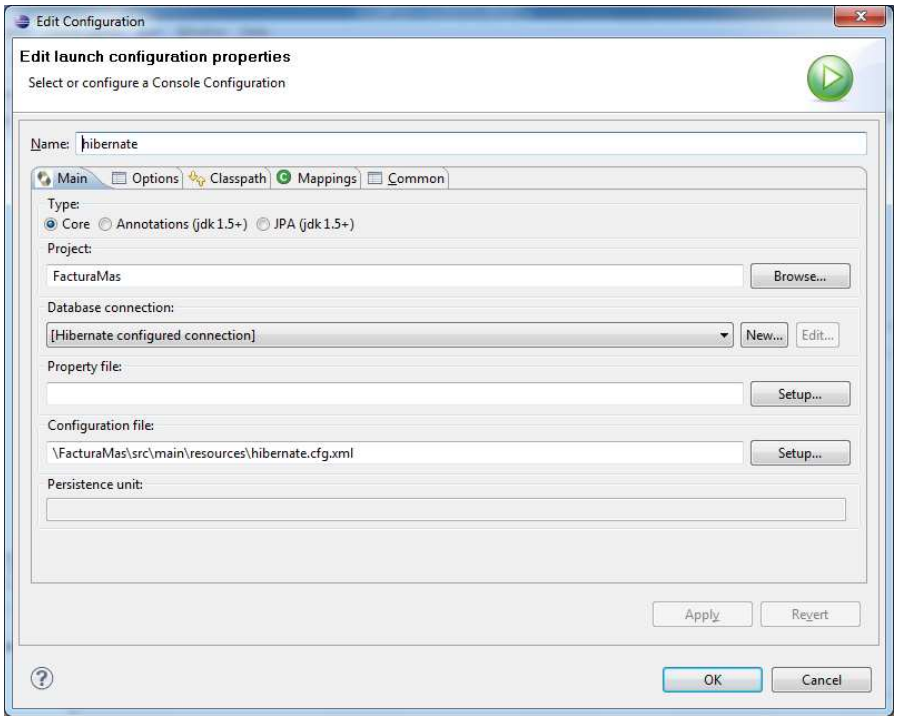


Figura 29 - Hibernate Synchronizer, Nueva configuración

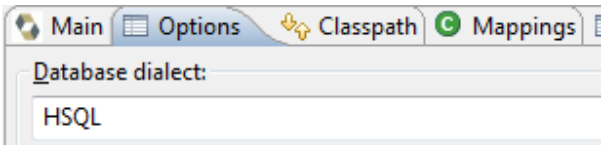
En la figura anterior, se muestran las dos opciones disponibles para crear una nueva configuración. Tras usar cualquiera de ellas, Eclipse muestra un diálogo con cinco pestañas para la configuración de Hibernate.

A continuación se repasan paso a paso las opciones a configurar y los valores necesarios para el ejemplo:

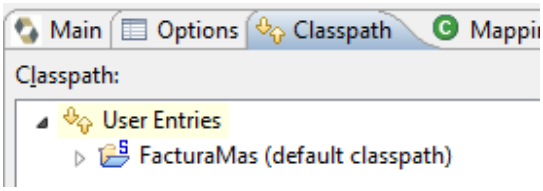
- La primera de las pestañas, principal, permite seleccionar el proyecto con el que se desea trabajar y el fichero de configuración a generar. En este caso, se selecciona el proyecto de ejemplo que se está siguiendo en este texto y el fichero de configuración se guarda bajo el directorio ***/src/main/resources/*** con el nombre ***hibernate.cfg.xml***



- En la pestaña de ***Opciones***, se debe elegir el dialecto de la base de datos que se va a atacar. En este caso HSQL.

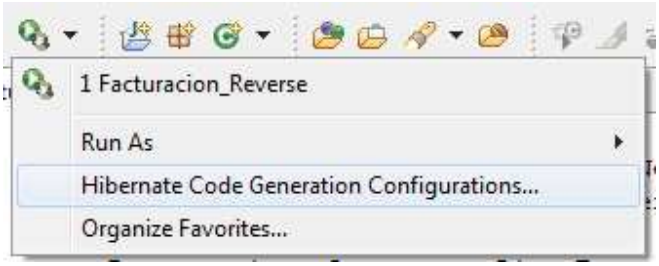


- La tercera pestaña, ***Classpath***, permite configurar cómo su nombre indica, el classpath para la configuración. Se añade el proyecto actual con el botón ***Add Project.***



Con estas opciones, por el momento es suficiente.

A continuación se procede a crear la configuración de ejecución para la generación de código o ingeniería inversa sobre la base de datos.



Se hace clic en la opción "***Hibernate Code Generation Configurations...***" bajo el icono de Hibernate en la barra de tareas. También es posible encontrar está opción ***Run*** del menú.

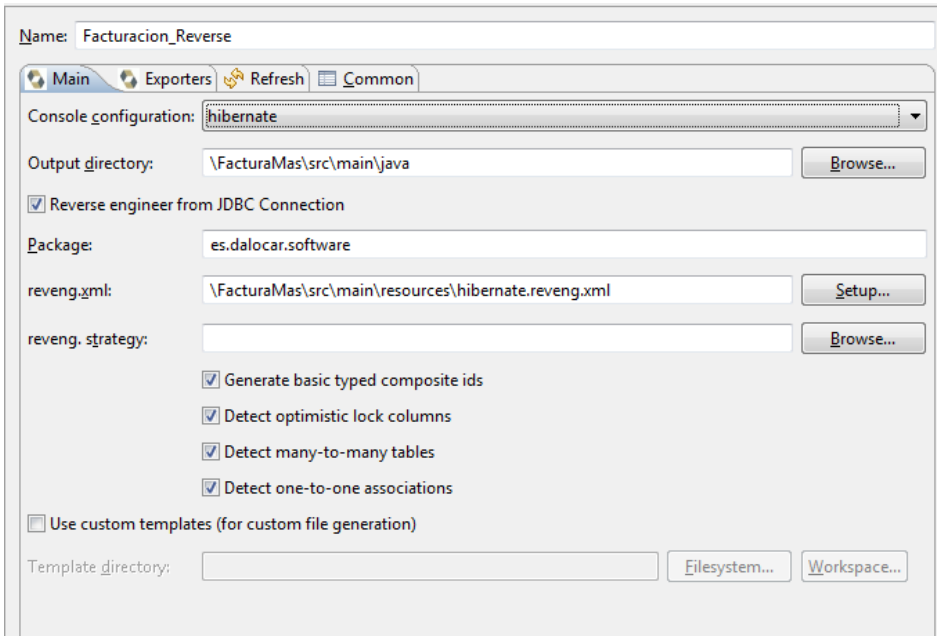
Al igual que ocurre con la configuración vista para la conexión de Hibernate, aquí también se muestra un diálogo con varios paneles o pestañas.

Solamente es necesario modificar opciones bajo las dos primeras pestañas. En las otras dos no hace falta cambiar ningún dato puesto que las opciones que existen por defecto son correctas.

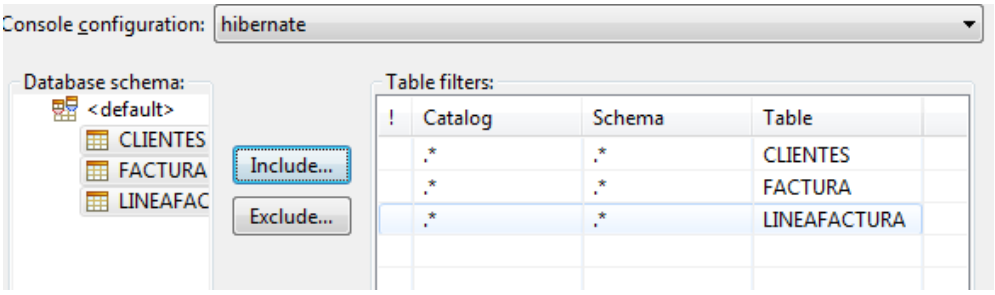
Bajo la pestaña ***Main*** se debe optar por las siguientes opciones:

- **Console configuration:** Permite elegir la configuración creada anteriormente.
- **Directorio salida:** Es el directorio bajo el cual se genera el código y los ficheros necesarios.
- Se debe activar la opción de ingería inversa desde una conexión JDBC.
- El paquete bajo el cual se genera el código. En este caso ***es.dalocar.software.***

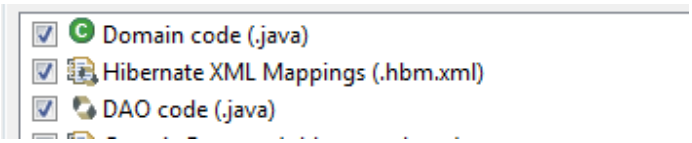
- Generar el fichero **Reveg.xml**. Este punto es quizás el más importante a la hora de generar el código, puesto que es quien lee la estructura de las tablas. Se debe hacer click sobre "**Setup"** para crear un nuevo fichero. Tras pulsar el botón se muestra un diálogo donde introducir la ubicación y el nombre del fichero a generar. El nombre que propone por defecto, es válido y la localización para el fichero debe ser bajo ***src/main/java*** del proyecto en el que se está trabajando.



En este momento es importante que la base de datos esté arrancada para realizar el proceso de lectura de la estructura de las tablas.

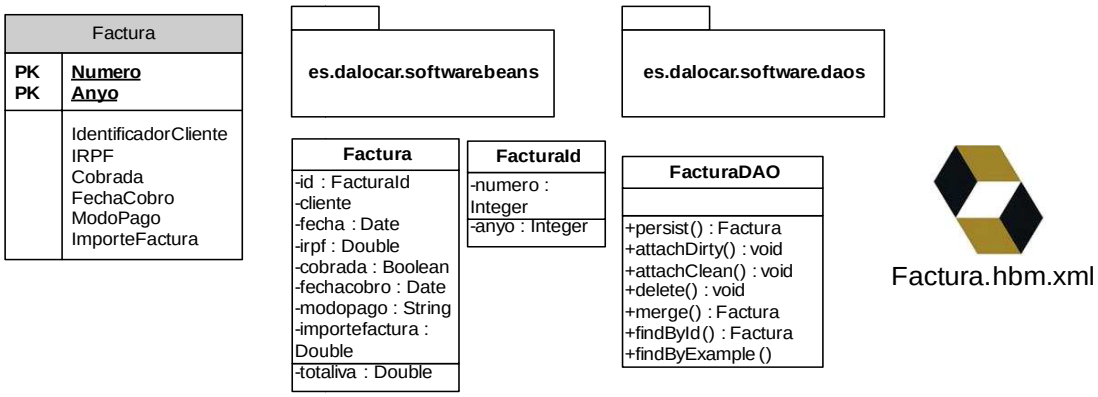


Para que aparezcan las tablas y el esquema, se debe pulsar sobre el botón "Refresh" bajo el panel de la izquierda. Tras seleccionar las tablas se pulsa sobre "***Include"*** para añadir todas ellas al listado de filtros. Si se sigue el ejemplo, la estructura debe quedar como se muestra en la figura anterior. Tras aceptar se vuelve de nuevo al diálogo de configuración anterior donde se ha de seleccionar la segunda pestaña, "***Exporters***". En esta pantalla se dejan marcadas exclusivamente las opciones de generar código java, mapeos y DAOs.



Tras esto, ya es posible pulsar el botón "***Run***", para comenzar la generación de código.

Cuando se ejecuta la configuración creada, por cada tabla se crean al menos tres objetos, un fichero de mapeo, una clase de acceso a datos y un bean de Java.



En la figura anterior, se puede ver el código que se espera que se genere para el caso de la tabla **Factura**.

- Un Javabean, ***FacturaId***, que representa a la clave primaria. Cuando el identificador de la tabla es compuesto, el generador de código crea un objeto agregado para la clave.
- Un Javabean, ***Factura,*** que representa a la tabla factura al completo y que contiene un objeto identificador.

Las dos clases anteriores se mueven bajo el paquete **es.dalocar.software.beans**

- Una clase de acceso a datos, ***FacturaHome***, la cual deberá ser renombrada a FacturaDAO y que contiene todos los métodos que facilitan el acceso a datos.

Esta clase se mueve bajo el paquete es.dalocar.software.daos

- Un fichero, ***Factura.hbm.xml,*** que define los mapeos para hibérnate entre el modelo relacional y el modelo objetual.

Este fichero, se moverá al source folder **src/main/resources** bajo el paquete **es.dalocar.software.beans**.

6.2.4 Pruebas del código generado

Una vez generado el código, una buena práctica es proceder a realizar un test de JUnit para comprobar que el acceso a datos se hace correctamente.

Toda clase de test debe estar bajo la carpeta de fuentes **src/test/java.** Acto seguido se explica cómo crear un test para la clase de acceso a datos del ejemplo anterior.

6.2.4.4 Dependencias necesarias

Al igual que en el caso de Hibernate y Spring, existen dependencias que deben ser incluidas para los test unitarios. En este caso son las dependencias de Log4j y de JUnit.

Se recomienda leer la documentación[9] sobre Log4j y su uso. Simplemente se debe añadir el fichero **log4j.properties** anexo a este documento al proyecto bajo la raíz de la carpeta de fuentes **src/main/resources.** En Internet hay mucha literatura acerca del uso de **log4j** y cubrir en proyecto todas sus funcionalidades resulta imposible.

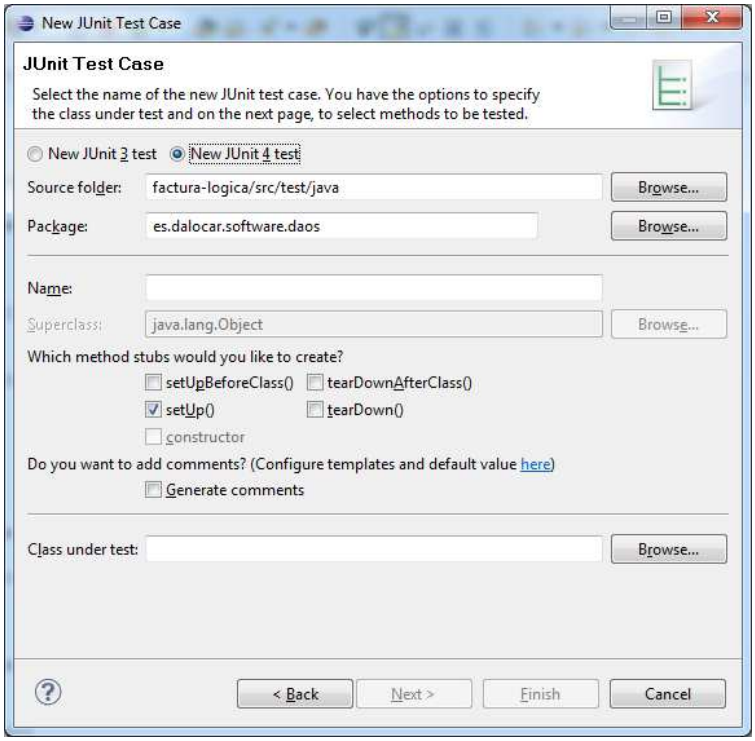
Las dependencias a añadir son las siguientes:

```
<dependency>
	<groupId>log4j</groupId>
	<artifactId>log4j</artifactId>
	<version>1.2.15</version>
</dependency>
```

6.2.4.5 Creación de una clase de Test

Para realizar un test de JUnit, se debe de crear una nueva clase Java que extienda de la clase TestCase. Eclipse provee de un asistente que crea este tipo de clases por defecto. Para ello, lo más sencillo es acudir al menú "**File>New…"** y en el asistente que aparecerá seleccionar la opción "Junit>Test Case". Tras esto se muestra un asistente que permite introducir las opciones para la clase Java a crear.

[9] Log4j: http://logging.apache.org/log4j/1.2/index.html



Como "Source folder" se deberá seleccionar la carpeta **src/test/java** del proyecto. Cómo paquete, es una buena práctica el que coincida con el paquete de la clase para la que se va a realizar el test. En el nombre de la clase, se introduce normalmente el nombre de la clase a testear finalizado en ***Test***. Por ejemplo, para Factura.java, el nombre de clase sería FacturaTest.

Además de ello, se deja marcado la opción de generar un método **setUp()**. Es en este método dónde se deben inicializar las clases de contexto para **Spring,** o el log4j por ejemplo.

Tras introducir estos datos es momento de finalizar el asistente.

En la clase creada se debe crear un nuevo método, que realice llamadas a la base de datos y en caso de devolver datos que realice un assert para indicar que el resultado ha sido correcto.

Capítulo 7. La lógica de negocio

Cómo se dijo en la introducción a las arquitecturas multicapa, esta parte es la que quizás lleve más trabajo a la hora de realizar un desarrollo. Esto es así, porque como su nombre indica, contiene la lógica del negocio. Cada negocio es un mundo y tiene el personal tiene sus propios hábitos de trabajo adquiridos, los cuales han de ser llevados al código. Se han de definir búsquedas sobre la base de datos bajo determinados parámetros, se han de realizar operaciones y cálculos que en cada empresa u organización poco tienen que ver.

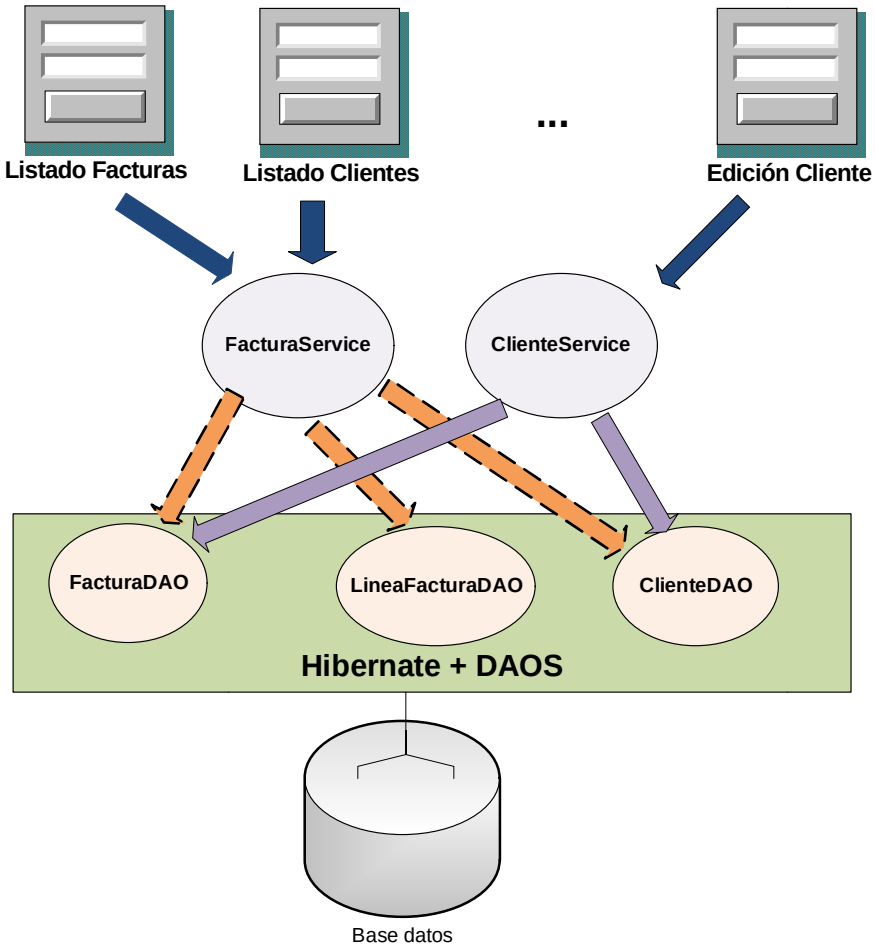
Este es quizás el motivo, por el que no existe un abanico tan amplio de herramientas que ayuden a acelerar el desarrollo como puedan existir para el resto de niveles. No obstante, sí que existen algunas y se hará uso de ellas para facilitar en la medida de lo posible el desarrollo y acortar su expansión en el tiempo.

Bajo esta capa, se encuentran las clases encargadas de encapsular el acceso a la base de datos. De este modo, se aísla al programador de la interfaz gráfica de programar cualquier método complejo. Simplemente debe realizar llamadas a los métodos de las clases contenidas en esta capa sin preocuparse de realizar cálculos.

Con la aparición de herramientas como Hibernate, el límite entre la capa de datos y la de lógica de negocio queda un tanto desdibujado. Se puede considerar que los DAO's también forman parte de la capa de lógica y que las librerías de Hibernate cubren todo el acceso a base de datos. Aquí se va a considerar que los DAOs cubren las funcionalidad de acceso a datos en su mayoría y que la lógica de negocio queda encapsulada en los servicios.

A continuación se muestra una figura que da una idea de la funcionalidad de los servicios. Lo que se pretende es presentar el papel que juegan las clases de servicio. En la parte superior de la figura, se encuentran la vista de la aplicación. Únicamente se ha pintado una flecha por cada formulario hacia uno de los servicios, puesto que lo habitual es que desde cada formulario se ataque a un único servicio.

De los servicios hacia los DAO's se puede observar que salen varias flechas desde cada uno de los servicios. Se pretende mostrar una relación de uno a muchos de los servicios y los DAOs. Un servicio, puede contener en su implementación llamadas a varios DAOs diferentes.



Por ejemplo, sea el caso de guardar en la base de datos una nueva factura. En el momento de apretar el botón de guardar se deben realizar distintas operaciones. Una factura, está identificada por un número de factura dentro de un año.

Este tipo de identificador compuesto es inviable que sea autogenerado por la base de datos. Las clases de servicio, deben aislar al programador de la vista de esta problemática y dotar a éste de un método de guardar factura, donde únicamente tenga que pasar como parámetro la factura a guardar.

Dentro de este método, se debe de realizar la búsqueda de la última factura guardada durante el año de ésta, calcular el nuevo identificador, validar los datos y realizar el guardado a través del DAO.

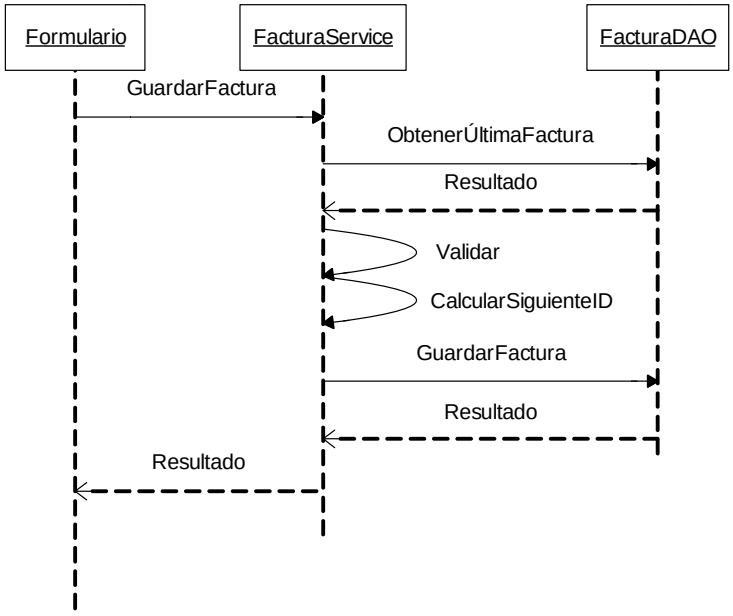


Figura 30 - Diagrama de secuencia del guardado

En los anexos, se puede consultar todas las clases generadas para la el ejemplo.

Capítulo 8. Capa de presentación

Esta parte del documento, se centra en la vista o presentación de la aplicación. Esta capa, es la que interacciona con el usuario, es decir recibe los datos introducidos por éste y muestra el resultado de sus peticiones por pantalla.

Para esta capa, se podrá optar por crear un nuevo proyecto que tenga como dependencia al que contiene la capa de datos, o bien por añadir las nuevas clases al proyecto desarrollado hasta ahora. Para el presente ejemplo se va a optar por desarrollar de manera separada las dos capas y dejar el código desarrollado hasta el momento como librería auxiliar para un nuevo proyecto

De este modo, se consigue independizar totalmente de la visualización elegida. Con la misma dependencia (el código creado hasta ahora), se puede desarrollar una interfaz basada en Web, en Swing, en SWT, etc.

Para la implementación de los formularios, se hace uso de las librerías SWT (Stándar Widtget Tookit). Por ello, es necesario disponer de las librerías necesarias en el path de la aplicación, tarea realmente sencilla incluyendo toda dependencia necesaria en el pom de la aplicación. Se puede consultar las dependencias en el fichero ***pom*** disponible entre los ficheros anexos al documento.

Por último recordar al lector la necesidad de disponer del plugin **jigloo** para Eclipse**,** que permite diseñar los formularios de manera gráfica, al igual que las herramientas de otros entornos basados en Visual Basic por ejemplo.

8.1. Formularios o composites

Un formulario en SWT, está representado por un conjunto de widgets o herramientas (campos de texto, combos, etc.) contenidas en una clase contenedora llamada Composite. De este modo, la mayor parte de los formularios a crear son objetos de la clase Composite de las liberías SWT.

8.1.1 Creación del menú inicial

El menú inicial es un Composite o clase más cualquiera dentro de la aplicación de no ser porque es la encargada de inicializar la configuración de la aplicación, con tareas como la inicialización del sistema de logging, de manejo de las excepciones y de contener el punto de entrada (método main de la aplicación).

Para su comprensión se recomienda visualizar el código de ejemplo proporcionado de manera conjunta a este texto. De hecho, basta con realizar una copia directa de la clase MainComposite.java que se adjunta y adaptar el fichero de propiedades y los puntos del menú a las necesidades concretas de la aplicación que se esté programando.

Aún siendo necesaria únicamente una copia exacta del código proporcionado, a continuación se repasan los métodos más importantes de la clase de cara a mejorar la comprensión del código y la forma de añadir nuevas entradas al menú de la aplicación.

- **configureLog4j().** Este método carga la el fichero de propiedades y configura el sistema de trazas **log4j**. En la clase ***MainComposite*** se busca el fichero de configuración en la ruta relativa *conf/log4j.properties*. Por tanto, a partir del momento en que se realiza la copia de la clase al proyecto en desarrollo también se debe crear el fichero de propiedades en la ruta indicada para poder utilizar el sistema de trazas.

- **handleError().** Se encarga de capturar las excepciones no controladas que ocurren durante la aplicación y mostrar un mensaje al usuario en caso de que se produzca alguna de ellas.

- **addListeners().** En este método se programa todos los eventos del Composite. Básicamente se trata de añadir los eventos de selección del menú de la aplicación, lo cual se muestra a continuación. Este método es invocado en el constructor del Composite.

- **main().** Método principal de la aplicación y que la convierte en ejecutable. A continuación se realiza un análisis del contenido del método donde se colocará la mayor parte de la personalización del menú.

El secreto del Composite de menú está en este método. Realmente este método se convierte en el punto de partida de la aplicación una vez creado adecuadamente con todas sus opciones y enlazado con los menús de lista y los listados de impresión.

Por tanto, es en este método donde ocurre la inicialización del programa y dónde se configura el sistema de trazas, donde se incluye una posible verificación de seguridad, el manejo genérico de excepciones, donde se muestra la pantalla inicial, etcétera. A continuación se explica el contenido (sin tener en cuenta las trazas), por lo que se recomienda la lectura de los siguientes párrafos mientras se dispone del código anexo al proyecto abierto.

Como se puede observar, en primer lugar se realiza la configuración del sistema de trazas log4j con el método ***configureLog4j***().

A continuación, se puede ver un fragmento de código de inicialización propio de la API SWT y que no merece de demasiado comentarios puesto que es en gran parte autogenerado por la herramienta ***jigloo***.

```
Display display = Display.getDefault();
Shell shell = new Shell(display);
MainComposite inst = new MainComposite(shell, SWT.NULL);
…
Más codigo autogenerado
…

// Establecer el icono de la aplicación desde recursos
```

Las siguientes dos líneas se encargan de establecer el icono de la ventana principal de la aplicación, en este caso se lee el valor desde recursos (archivo ***resources.properties***, el cual se debe encontrar en el classpath de la aplicación). La imagen a utilizar como icono de la ventana principal también debe encontrarse en el classpath.

```
// Establecer el icono de la aplicación desde recursos
String iconPath = ResourceUtils.getString(ResourceUtils.IMAGE_APP_ICON);
shell.setImage(SWTResourceManager.getImage(iconPath));
```

En la clase **ResourceUtils**, se encuentran las claves que identifican a los recursos dentro del fichero **resource.properties**.

```
public static final String IMAGE_APP_ICON = "image.app.icon";
```

De modo que la clave "image.app.icon" debe existir en el fichero de propiedades y su valor indica la imagen a emplear como icono.

```
image.app.icon=es/dalocar/facturamas/res/favicon.ico
```

La siguiente sentencia hace que la ventana aparezca maximizada durante la carga inicial. Esta llamada puede estar aquí o también en el propio constructor de la clase. Tras esta línea se ve que la ventana principal se muestra por fin al usuario con ***shell.open***().

El código que aparece entre comentarios "**$hide$**" no es procesado por el editor visual ***Jigloo*** (esto es así, con el fin de que no se carguen los recursos de datos ni la lógica de negocio mientras se está desarrollando visualmente, de lo contrario la carga sería muy lenta y se producirían errores si los recursos necesarios, como puede ser la base de datos, no están disponibles).

- **MainComposite().** Constructor de la clase. Es en el constructor donde se realiza la llamada a varios métodos de inicialización, entre ellos ***addListeners***().

En el constructor de la clase "***MainComposite***" se realizan varias operaciones de inicialización, entre ellas la llamada al método ***addListeners***() que configura los eventos de la clase. La línea que aparece comentada corresponde a una parte de código de lógica de negocio, la cual se explica en capítulos posteriores. A continuación se muestra el código de ejemplo.

```
public MainComposite(Composite parent, int style) {
        super(parent, style);
        initGUI();

        // $hide>>$
        manager = ApplicationManager.getManager();
        // $hide<<$

        getShell().setText("Menú Principal");
        createAccelerators();
        addListeners();
```

```
}
```

Si se ejecuta la clase "***MainComposite***" dentro de Eclipse, ya sea como "*Java Application*" o usando el botón de acción "*Build an run*" de Jigloo, aparece una ventana similar a la de la imagen, pero maximizada (en caso de aparecer textos entre signos de exclamación, se deben introducir las claves de internacionalización necesarias en el fichero ***messages.properties*** de la aplicación).

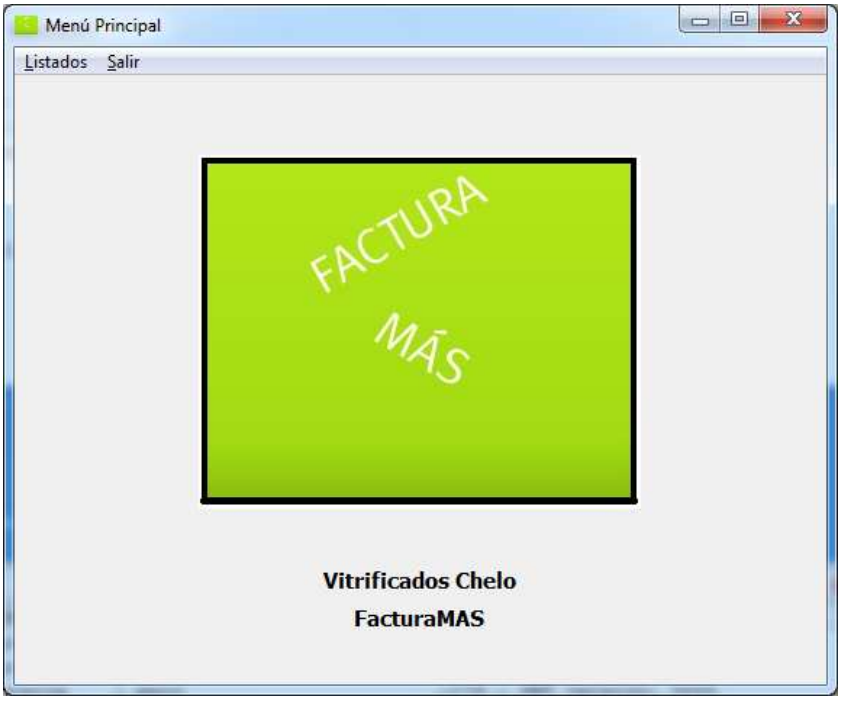


Figura 31 - Ventana principal de la aplicación

8.1.1.6 Los elementos del menú

Una vez creado el esqueleto de la ventana principal es el turno de crear los controles o elementos del menú. Como ya es sabido, un menú consta de una barra superior horizontal en la que se disponen los elementos raíz de cada submenú, y dentro de cada uno de ellos cuelgan a su vez otros submenús. La profundidad puede alcanzar el nivel que se desee, aunque lo habitual es no pasar de más de tres de ellos. SWT utiliza dos tipos de objeto, ***Menu*** y ***MenuItem***, para crear cualquier combinación de menú que se desee.

El tipo SWT habitual de un *MenuItem* es ***CASCADE***, lo cual significa que pueden colgar más submenús dentro de él. Aunque existe otros tipos:

- **PUSH**. Similar al tipo CASCADE, pero que no admite colgar más elementos dentro de él. Ambos, sin embargo, responden al evento de selección por tanto se pueden utilizar según convenga.

- **CHECK**. Un submenú que actúa como un pulsador ON/OFF. Cada vez que se selecciona, su propiedad selection alterna entre los valores true/false. Si está seleccionado muestra una marca "v", similar a la de un cuadro de selección (checkbox). Este tipo de controles sirven para mantener propiedades de activo/inactivo dentro de la aplicación.

- **RADIO.** Un submenú que actúa dentro de un grupo de elementos, dentro de dicho grupo únicamente un elemento puede tener la propiedad ***selection*** con el valor *verdadero* (solamente uno puede estar seleccionado a la vez).

- **SEPARATOR**. Actúa como separador entre elementos de un menú y no responde ante acciones del usuario.

Para la creación del menú de manera gráfica se emplea el GUI Editor de Jigloo. En la vista "Outline" de Eclipse se observa en la carpeta con el nombre "Extra components" que aparece el elemento superior de tipo Menu, BAR. Este es el punto inicial y representa la barra de menús del Shell principal.

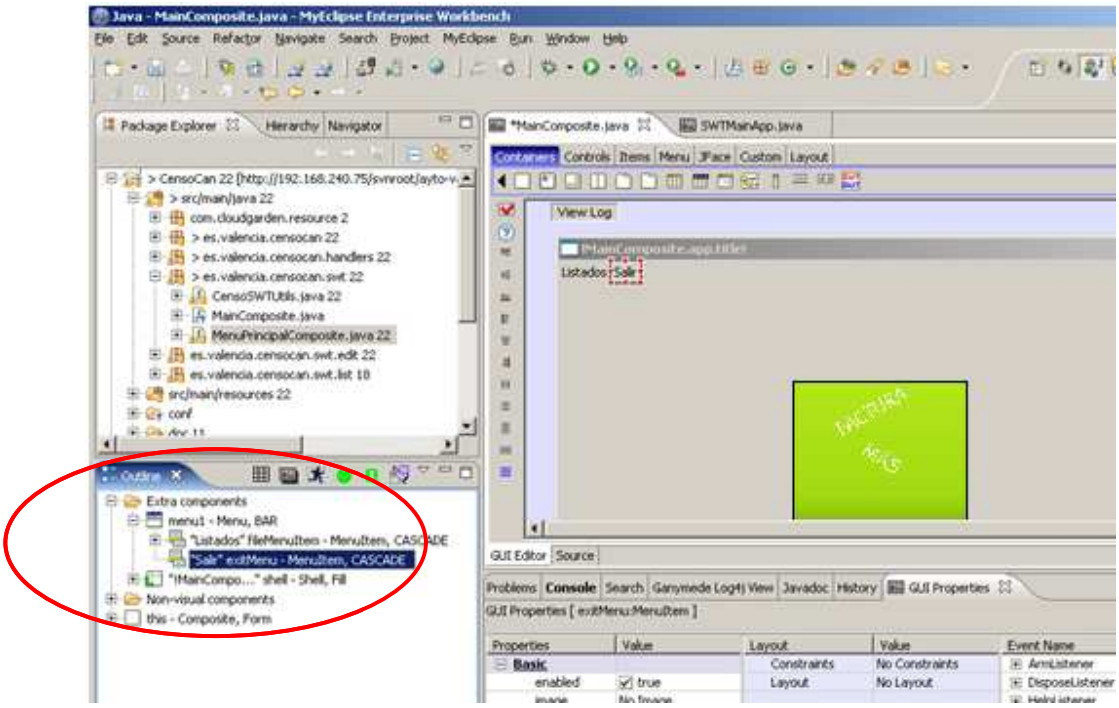


Figura 32 - Creación del Menú Principal

El primer nivel de un menú típico consta de varios objetos MenuItem - CASCADE. Dentro de cada uno de ellos se cuelga un objeto de tipo Menú - DROP_DOWN que representa la raíz de un submenú, a partir de éste se van colgando nuevos objetos MenuItem - PUSH o MenuItem - CASCADE y así sucesivamente.

Para el ejemplo, se crean dos MenuItem, dentro del elemento raíz del menú, uno será para colgar los listados y otro para salir de la aplicación, por tanto el primero podrá ser de tipo CASCADE y el segundo de tipo PUSH (aunque realmente esto es indiferente, el segundo MenuItem también podría ser de tipo CASCADE).

Figura 33 - Resultado del menú propuesto

Para crearlos, se puede borrar el contenido actual del menú y empezar desde cero. Se debe seleccionar el elemento "menu1" en la vista "Outline" y con el botón derecho se selecciona la opción "Add MenuItem – CASCADE". En las propiedades del MenuItem se ha de indicar el nombre y texto que se quiere que aparezca, como se muestra en la figura.

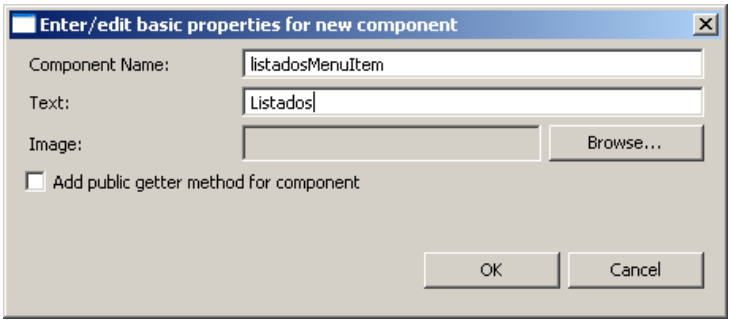


Figura 34 - Creación de un nuevo elemento de menú

Se observa que Eclipse crea automáticamente un objeto de tipo Menu, colgando del nuevo MenuItem, esto es debido a que se ha seleccionado CASCADE como su tipo. Lo que se hace a continuación es renombrar el objeto Menu para que se llame por ejemplo, "listadosMenu".

A continuación, se cuelga de él un nuevo MenuItem, "listadoFacturasMenuItem" que será el encargado de abrir el formulario de listado de facturas.

Por otra parte y colgando de la raíz del menú, se crea un MenuItem de tipo PUSH, "exitMenuItem" y se le añade el texto "Salir". El aspecto final del menú se muestra en la figura.

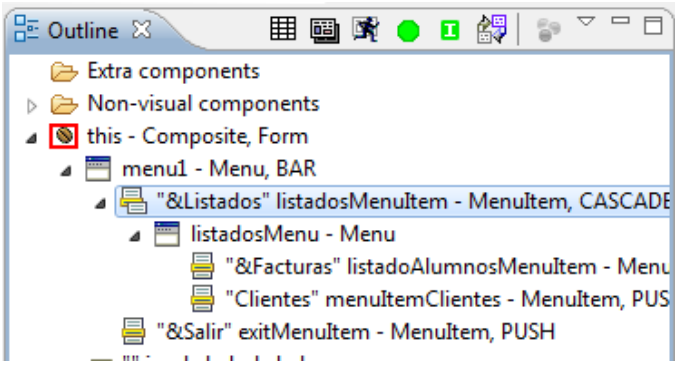


Figura 35 - Vista outline del menú generado

8.1.2 Creación de un composite de tipo edición

En esta parte del documento, se explica cómo crear un formulario de edición. Con el fin de organizar el código de una manera más óptima, se ha de crear un paquete para almacenar los composites de edición llamado, en el caso del ejemplo que se sigue en esta lectura, **es.dalocar.software.swt.edit.**

A continuación se explican los pasos a seguir para la creación de un Composite para editar los campos de la tabla CLIENTES del ejemplo. Es un ejemplo sencillo, que

muestra de un modo simple la metodología y se obvia los detalles más minuciosos de la programación en SWT.

Si se ha seguido el ejemplo desde su inicio, ya se dispone de las dependencias necesarias en el fichero de configuración de Maven para esta parte del documento.

8.1.2.7 Creación de la clase

Para la creación desde cero de este composite, se hace uso del asistente de Eclipse. Desde la perspectiva "Java", se deja seleccionado el proyecto en el cual se está trabajando y se acude al menú de Eclipse "File>New>Other". En el cuadro de diálogo aparecido se ha de buscar la opción "GUI Forms>SWT>SWT Composite", como se muestra en la siguiente figura.

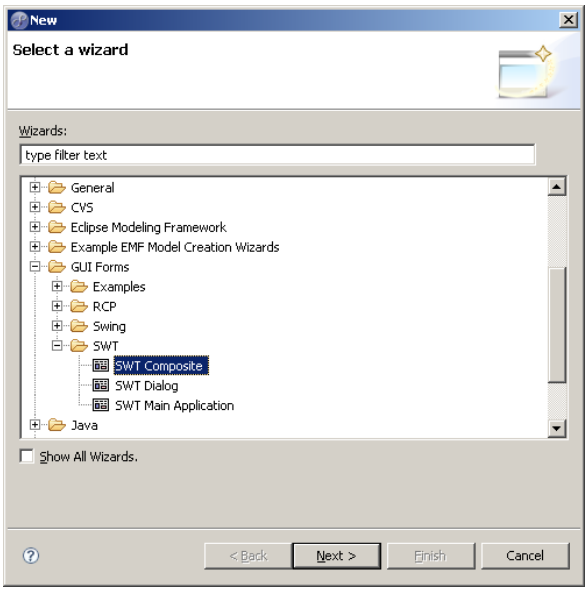


Figura 36- Creación de un composite

Tras seleccionar "Next", se solicitan los datos para la clase Java que se creará para el nuevo formulario.

Como paquete se debe seleccionar el que se ha creado al inicio de la sección para agrupar los composites de edición. En el nombre de clase se recomienda emplear un nombre del tipo **TablaEditComposite**, en el ejemplo ClienteEditComposite. Cómo superclase se debe utilizar la clase Composite de las librerías SWT como muestra la siguiente figura.

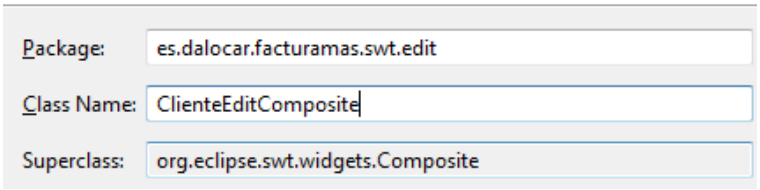


Figura 37 - Parámetros configuración EditComposite

Tras finalizar, Eclipse abre el nuevo formulario con el editor gráfico ***Jigloo***. En el caso de que esto no ocurra, lo que se hace es seleccionar la clase que se acaba de crear en la vista "***Package Explorer***" y mediante clic derecho se selecciona la opción "Open With>Form Editor".

A continuación, en la vista "***GUI Properties***", se cambia el layout creado por defecto, de ***Grid*** a ***Absolute***. Con esta operación se modifica el Composite para que utilice el AbsoluteLayout de SWT y se pueden posicionar los controles en el lugar deseado. (Recordar guardar la clase cada vez que realicemos modificaciones).

Lo siguiente es dimensionar adecuadamente el Composite. Para ello se selecciona el elemento "*this – Composite, Absolute*" en la vista "***Outline***", y desde el "*Form Editor*" se arrastra el cuadro desde los recuadros rojos para cambiar el tamaño con el ratón hasta que se desee.

Una vez hecho esto, se coloca un nuevo contenedor SWT dentro del Composite, para que el formulario logre el aspecto deseado. Para ello, se selecciona el elemento "***Group***" en la ficha "***Containers***" que se puede ver en la paleta de componentes del "*Form Editor*".

En la pantalla de propiedades de nuevo componente, se le da un nombre y un texto al nuevo elemento. Es necesario asimismo indicar el layout como "Absolute". Se le puede indicar a Jigloo que el layout elegido sea a partir de ese momento el que se utilice por defecto.

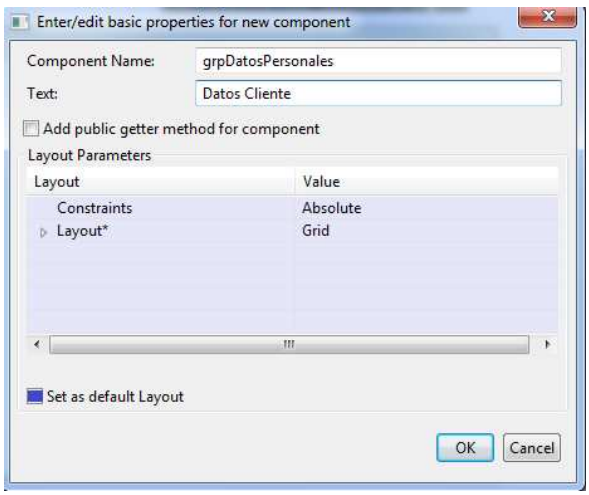


Figura 38 - Propiedades de un grupo de elementos

Estos son los métodos generados por el plugin Jigloo:

Los métodos que Jigloo ha generado son:

- **main()**. Jigloo crea este método para que la clase sea ejecutable. De esta forma es posible visualizar el aspecto del formulario desde el editor, e incluso lanzarlo para que se ejecute. Sin la ayuda de Jigloo esto no sería posible a menos que se creen métodos similares de forma manual. Este método no tiene utilidad posteriormente en la aplicación, es una ayuda para el desarrollo.
- **showGUI()**. El método m
- ain() descrito anteriormente realiza una llamada a éste método también generado por Jigloo, el cual se encarga de crear un Shell() que contendrá al Composite que se acaba de crear. Al igual que el método anterior, este método es una ayuda para el desarrollo y no tiene utilidad para la aplicación.
- **Constructor**. El constructor del Composite es el punto inicial de carga del formulario. Servirá para contener todas las inicializaciones que se necesite incluir en él. En el constructor hay siempre una llamada al método ***initGUI***().
- **initGUI()**. Este método es generado por completo por Jigloo, se encarga de la creación de todos los controles y es aconsejado incluir código propio en él. El constructor llama a este método para crear todos los elementos del Composite. Cada vez que se añade un nuevo control, se modifica uno existente o se crea un evento, este método se ve modificado.

8.1.2.8 Añadir controles

Como se puede ver en el apartado anterior, Jigloo facilita la tarea de desarrollo visual con los métodos generados, de lo contrario no sería posible visualizar el formulario. Se ha de tener especial cuidado en no modificar los métodos generados por Jigloo de manera automática. En su lugar, se debe emplear el cuerpo del contructor del Composite.

A continuación se deben añadir al composite los controles para la edición de los datos. Se añade una etiqueta y un campo de texto por cada uno de los atributos a modificar. Para ello, se emplea la pestaña "*Controls*" de la paleta de elementos del editor. Para las etiquetas se usa el control "***Label***" mientras que para los campos de texto se emplea el control de tipo "***Text***".

En el caso de los campos texto, se debe especificar que tengan borde, mediante el botón derecho de ratón sobre el campo de texto, "**Set/Change Style… > BORDER**". Además de esto, se recomienda limitar el número de caracteres admitidos por el campo de texto con el fin de evitar incongruencias y excepciones con los tamaños aceptados por los campos en la base de datos. Para este se utiliza la propiedad "***TextLimit***".

Además de los campos de texto, se añaden dos botones, Aceptar y Cancelar, para guardar o descartar los cambios realizados en el objeto.

El resultado del formulario debe ser similar al de la figura siguiente:

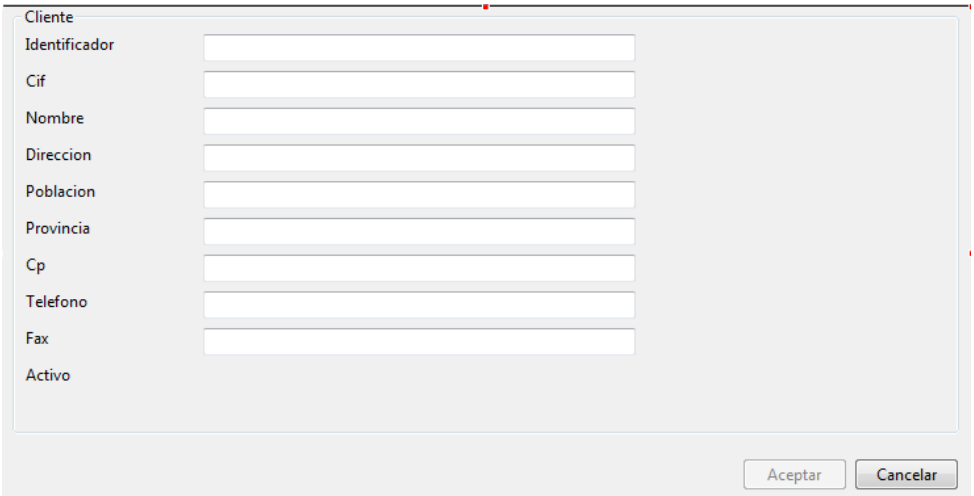


Figura 39 - Composite de edición

8.1.2.9 Programación de los eventos

La programación de los eventos en SWT es una tarea muy flexible. El código que controla los eventos puede ser creado en el constructor de la clase, en un método personalizado, ayudándose del editor Jigloo, de manera dinámica, etc.

Las opciones de Jigloo y la creación de métodos personalizados son las más interesantes.

La ventaja de emplear Jigloo es que la propia herramienta se encarga de generar el código necesario con varios clics de ratón. La desventaja es que es posible que se sufran pérdidas de código si no se tiene cuidado ya que la herramienta modifica el código cuando se realizan cambios en la parte gráfica. La ventaja en caso de optar por crear métodos personalizados radica en que el programador tiene mayor control sobre la creación del evento pero pierde la capacidad de generar el código ayudado de una herramienta gráfica.

En este caso, se opta por la opción de generar el código de manera manual, añadiendo al formulario un método llamado "***addListeners***" dónde el programador añade

todos los eventos necesarios para el funcionamiento de ese ***Composite***. Este método será invocado desde el constructor de la clase para que se creen todos los manejadores de eventos necesarios en el momento de instanciar la clase.

8.1.2.9.1 Ejemplo evento: Filtrado de caracteres

Para ejemplarizar la utilización de un evento, se va a controlar los caracteres admitidos por un campo de texto. En este caso, se va a filtrar el campo de código de cliente para que únicamente admita caracteres numéricos. Para no admitir cualquier otro carácter, se programa un evento que compruebe el texto que se introduce, de forma que invalide la operación si el texto no es válido.

```
/**
 * Metodo que añade a todos los campos de texto un listener para
 * escuchar los cambios
 */
private void addListeners() {

    txtIdentificador.addVerifyListener(new VerifyListener() {
      public void verifyText(VerifyEvent evt) {
            if (DEBUG)
                  log.debug("Identificador.verifyText, event=" + evt);

            evt.doit = SWTUtils.filterInteger(evt.text);
      }

});
```

La clase ***SWTUtils*** se encuentra en el proyecto adjunto y contiene multitud de métodos estáticos para ayudar en el desarrollo de aplicaciones gráficas. Para realizar los diferentes filtrados se pueden utilizar los métodos definidos en ella, como por ejemplo:

- **filterAlphaBetic().** Admite únicamente letras.
- **filterAlphaNumeric**(). Admite únicamente números y letras.
- **filterNumeric**(). Admite únicamente números con puntos y comas.
- **filterLowCase**(). Admite únicamente letras en minúsculas
- **filterUpperCase**(). Admite únicamente letras en mayúsculas.
- **filterNoSpace**(). Admite únicamente texto sin espacios.
- **filterChars**(). Admite únicamente los caracteres indicados como parámetro.

8.1.2.9.2 Ejemplo evento: Control de cambios

Otro evento que programar en los formularios es el de control de cambios. Lo que se prentende es habilitar o deshabilitar el botón de aceptar dependiendo de si el usuario ha cambiado el contenido del formulario. Los pasos a realizar, conceptualmente, son los siguientes:

1) Calcular el contenido inicial
2) Programar eventos de modificación
3) Verificar si ha habido cambios mediante una función

1) Calcular el contenido inicial

Para saber cuál es el contenido inicial de los controles del formulario, se emplea el método `SWTUtils.calculateState(Composite)`, que obtiene una cadena de texto con el contenido de los controles. En el constructor de los composites de edición se almacena este valor en una variable de tipo String para hacer comprobaciones posteriormente.

El estado inicial del botón "Aceptar" es "inhabilitado", por lo que se configura como "enabled = false" en la vista "GUI Properties" o bien se añade una sentencia en el constructor que lo inhabilite:

aCButtonsComposite1.getButtonAceptar().setEnabled(false);

2) Programar los eventos de modificación

Para cada control del formulario que admita entrada de datos es necesario programar el evento de modificación que calcule el nuevo estado y habilite/deshabilite el botón de aceptar. Esto se traduce en la creación de un evento por control y la llamada en dicho evento a la función que realice el cálculo.

El ejemplo de código muestra lo necesario para programar los eventos de modificación en los controles de texto del formulario `ClienteEditComposite`.

```
txtNombre.addModifyListener(new ModifyListener() {
     public void modifyText(ModifyEvent evt) {
           checkModify();
      }
});
```

Verificar si ha habido cambios mediante una función

Por último queda la comprobación de cambios. La operación consiste en obtener de nuevo el estado del contenido de los controles del formulario y compararlo con el inicial. El código de ejemplo muestra el código del método `checkModify()` que es invocado en el ***listener*** anteriores.

```
	/**
	 * Comprobar el estado del formulario
	 *
	 * @return Devuelve true si se ha producido alguna modificación
	 * en el formulario, false en caso contrario
	 *
	 */
	private boolean checkModify() {
		String currentState = SWTUtils.calculateState(this);
		boolean modified = !initialState.equals(currentState)
				&& comprobarObligatorios();
		buttons.getButtonAceptar().setEnabled(modified);
		return modified;
	}
```

Para comprobar que funciona correctamente, se ejecuta el formulario y se escribe texto en los controles. Se puede observar que el botón de aceptar se habilita y deshabilita en función de si se modifica el contenido inicial.

8.1.2.9.3 Ejemplo evento: Cierre del formulario

Existen dos eventos de salida en un formulario de edición, uno en el botón de "Aceptar" y otro en el botón de "Cancelar".

Al aceptar, se realiza la validación de los datos introducidos, para asegurar que son correctos. Además, el código del formulario debe detectar si ha habido cambios y pedir confirmación al usuario para guardarlos.

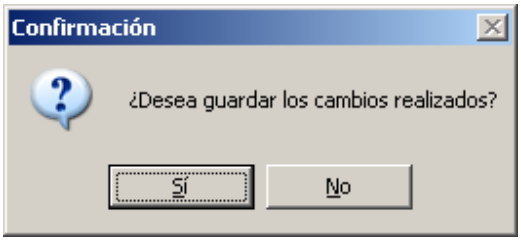
Al cancelar, el código del formulario debe verificar si ha habido cambios y pedir confirmación al usuario antes de cerrar.

El evento que se utiliza para ambas acciones es el `SelectionListener`, que actúa tanto ante una pulsación de tecla como mediante un clic de ratón. En el ejemplo de código se aprecia cómo se crean las ventanas de mensajes.

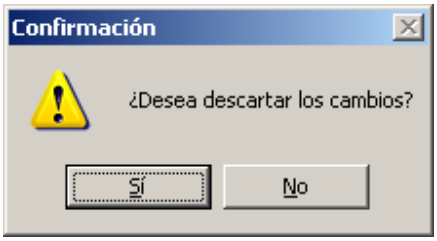
```
buttons.getButtonAceptar().addSelectionListener(new SelectionAdapter()
{
	public void widgetSelected(SelectionEvent evt) {
		populate();
```

```
            if (validate()) {
                  save();
                  getShell().dispose();
            }
      }
});

buttons.getButtonCancelar().addSelectionListener(new SelectionAdapter()
{
      public void widgetSelected(SelectionEvent evt) {
            if (checkModify()) {
                  MessageBox msgBox = new MessageBox(getShell(),
SWT.YES | SWT.NO | SWT.ICON_WARNING);
                  msgBox.setText("Confirmación");
                  msgBox.setMessage("¿Desea descartar los cambios?");
                  int ret = msgBox.open();
                  if (ret == SWT.YES) {
                        getShell().dispose();
                  }
            }
            getShell().dispose();
      }

      });
```

Cuando se ejecuta el formulario con los eventos de salida y se introduce algún valor en los campos de texto el formulario muestra una salida como la que se visualiza la figura siguiente.



De igual forma, al introducir cambios en el formulario y pulsar el botón de cancelar, se observa en la imagen que aparece la correspondiente ventana de aviso.



En el primer caso, si se elige la opción “Sí” el formulario, valida los datos y si son correctos se cierra y guarda los cambios. En el segundo caso, si se elige la opción “Sí” el formulario se cierra y se descartan los cambios. En cualquier otro caso la ventana de formulario permanece abierta.

El comportamiento puede cambiarse en función de las necesidades de cada aplicación, aquí se ha dado una visión de un comportamiento típico.

8.1.2.10 Otras características

Para completar la programación visual de un formulario de edición, faltan algunos conceptos que se introducen a continuación como son la definición de que campos son obligatorios al rellenar el formulario y parametrizar el formulario para aprovechar el código del formulario tanto para altas, como para modificaciones.

8.1.2.10.1 Definición de campos requeridos

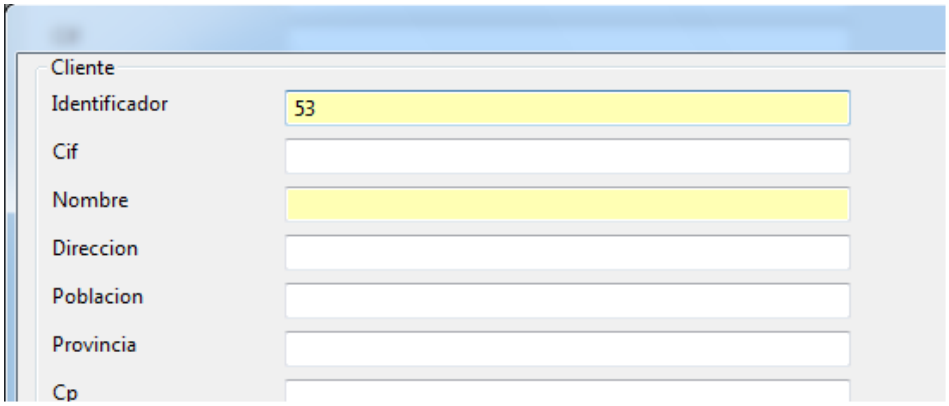
Una práctica aconsejable a la hora de desarrollar la interfaz, es resaltar o marcar de algún modo aquellos campos que son de carácter obligatorio. Normalmente estos campos de texto, coinciden con los campos de la base de datos que tienen restricción de valor no nulo. Este tipo de campos son los que se encuentran en los formularios web marcados con un asterisco o que aparecen marcados en negrita.

Esta tarea a priori, puede ser bastante artesanal, es decir, se debería colorear los controles que tienen carácter obligatorio de manera manual. Para un trabajo repetitivo, se puede hacer uso de una función de personalización a la que se llame desde el constructor y realice esta tarea.

A continuación se muestra la función que realiza esta tarea en el formulario de edición de clientes del ejemplo.

```
private void customize() {
	// Establecer los controles que sean requeridos
	txtIdentificador.setData(SWTUtils.CONTROL_REQUIRED, Boolean.TRUE);
	txtNombre.setData(SWTUtils.CONTROL_REQUIRED, Boolean.TRUE);
	SWTUtils.customize(this);
}
```

Si se ejecuta el formulario, se observa que ahora los campos marcados como "***required***" aparecen de distinto color como se muestra en la figura.



8.1.2.10.2 Parametrización del alta/baja

Una funcionalidad realmente interesante consiste en parametrizar los formularios de edición de datos de modo que puedan ser empleados tanto para realizar inserciones como para modificar registros existentes sin la necesidad de duplicar el formulario y quedando un código mucho más sencillo.

El comportamiento de alta o modificación se programa en el constructor de la clase añadiendo un tercer parámetro "esAlta" de tipo *boolean*. Si el valor del parámetro es ***verdadero***, indica que se trata de una inserción, mientras que si es ***false*** indica que se trata de una modificación de un registro existente. La declaración del método en el caso del ejemplo es la siguiente:

```
public ClienteEditComposite(Composite parent, int style, boolean esAlta)
```

En el cuerpo del constructor, se almacena el valor del parámetro en una variable de clase, cuyo nombre es el mismo al del parámetro.

```
this.esAlta = esAlta;
```

Además, se debe sobrecargar el constructor para que exista uno que acepte dos parámetros y se pueda seguir empleando el método ***initGui*** autogenerado por Jigloo. En el nuevo método se realiza una llamada al que se acaba de modificar.

```
public ClienteEditComposite(Composite parent, int style) {
	this(parent,style,true);
}
```

Como último cambio a realizar, dependiendo del comportamiento deseado, se debe elegir sí se deshabilita el campo identificador de la edición en caso de modificación o no. Para ello se añade la siguiente línea en el constructor, tras la llamada al método ***initGui()***.

```
this.txtIdentificador.setEnabled(esAlta);
```

8.1.3 Creación de un composite de tipo lista

En esta parte del documento, se explica cómo crear composite para mostrar listas de elementos y operar e invocar operaciones de mantenimiento desde éste. Con el fin de organizar el código de una manera más óptima, se crean dos paquetes, uno para almacenar los composite de tipo listado (ej: **es.dalocar.software.swt.edit**) y otro para almacenar las clases que actúan de manejadores para los composites (ej: **es.dalocar.software.swt.handlers**).

A continuación se ofrecen los pasos más importantes para la creación de un listado de clientes que se rellene desde la base de datos y como dotar hacer que se puedan invocar las operaciones de mantenimiento (CRUD[10]).

8.1.3.10.1 Creación de un listado base

Una buena idea pasa por crear un listado genérico para reutilizarlo en cualquier otro listado de la aplicación gracias al manejador. Este listado genérico, puede contener un campo y un botón de búsqueda en la parte superior, una tabla donde mostrar los resultados y el número de éstos además de los botones para las diferentes operaciones en la parte central y una botonera inferior para cerrar.

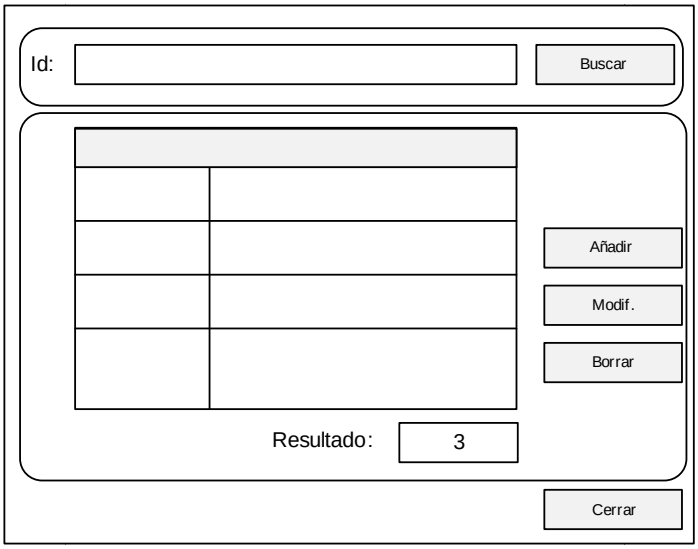


Figura 40 - Dibujo de un listado genérico.

En el apartado anterior (Ver 8.1.2.7) se dan los pasos a seguir para la creación de la clase además de enseñar el modo de añadir controles, de este modo a continuación simplemente se ofrecen los tipos de componentes añadir desde el asistente. Únicamente se debe tener en cuenta al crear la nueva clase, seleccionar cómo destino de ésta el paquete elegido para los listados y no el que se muestra en el apartado anterior.

[10] Create-Retrieve-Update-Delete – Creación, obtención, actualización y borrado

Para la parte superior, se emplean una etiqueta y un campo de texto por cada uno de los atributos por los que se desee implementar la búsqueda y un botón de búsqueda bajo un contenedor de tipo ***Group***.

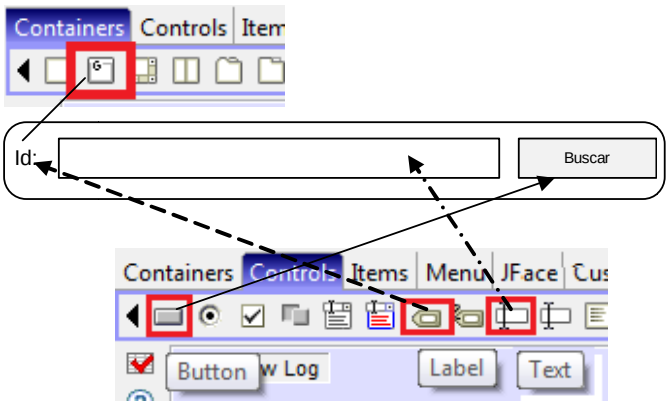


Figura 41- Relación entre componentes y esquema

Para la parte central se emplea un contenedor de tipo tabla junto a tres botones laterales para las diferentes operaciones. Bajo la tabla se ubica una etiqueta junto a un campo de texto no editable para mostrar el número de resultados. Todos estos componentes, se agrupan bajo un contenedor Group.

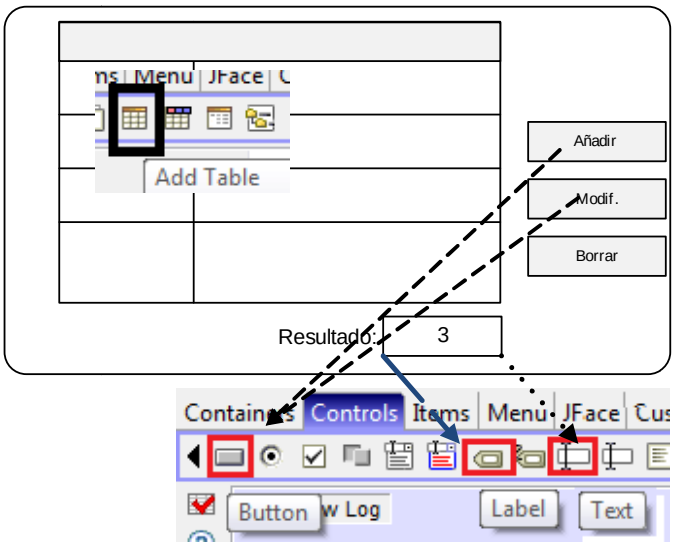


Figura 42 - Mapeo componente - esquema

Una idea incluso mejor, es la de separar el grupo superior del inferior en dos composites, de modo que se pueden reutilizar el componente de lista independientemente del número de campos por los que se desee buscar en diferentes composites y haciendo más flexible su uso.

De este modo se divide el composite en otros tres más básicos:

- **TopListComposite**

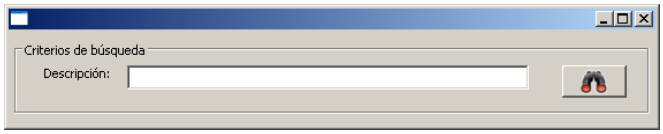


Figura 43 Composite para la parte superior de listas

- **CenterListComposite**

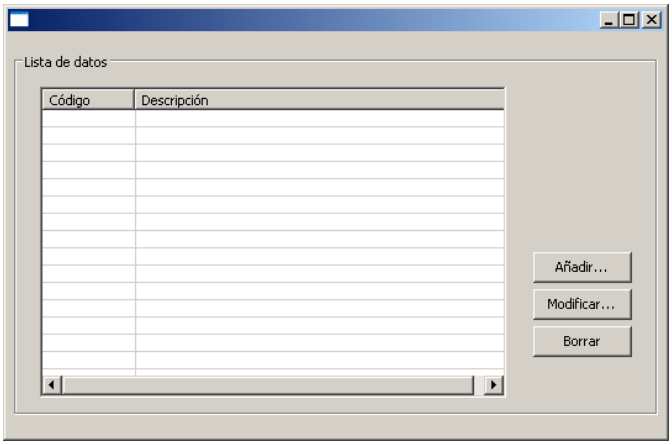


Figura 44 – Composite para los datos de la lista

- **BottomListComposite**

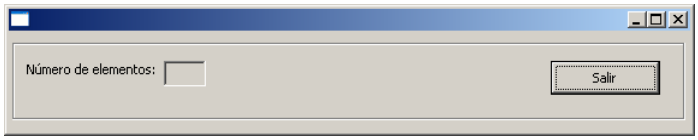


Figura 45 – Composite para el número de resultados

Creados los componentes básicos, la idea es crear el composite base final a partir de estos tres componentes.

Para ello se crea un nuevo Composite vacío llamado BasicListComposite y se le añaden los que se acaban de crear.

Para esta tarea, se emplea el asistente de ***Jigloo***. En la barra de herramientas de ***Jigloo***, se selecciona el icono  bajo la pestaña ***Custom***. Tras pulsar el icono aparece un diálogo para seleccionar la clase del composite a añadir. Se debe escribir el nombre que se le ha dado al composite de búsqueda en el apartado anterior, **TopListComposite.**

El siguiente paso a realizar es dotar de un nombre al elemento recién añadido al formulario. Se introduce "***top***" como nombre y además se marca la opción de "Add public getter method" para que se cree un método getter para el componente. Tras añadir el composite superior se debe hacer lo mismo para los composite centrales e inferior.

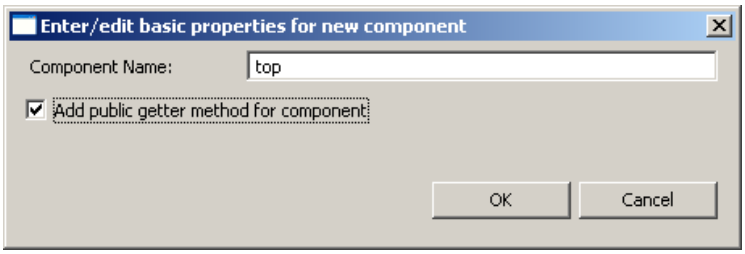


Figura 46 – Composite superior de la búsqueda

El aspecto final debe ser similar al que se muestra a continuación:

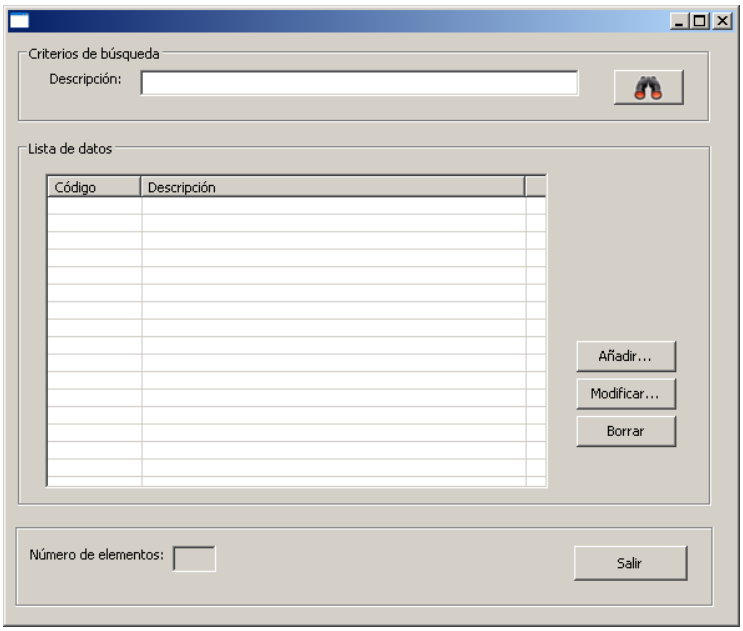


Figura 47 - Aspecto final Composite de listado

8.1.4 Reutilización de composites de tipo lista (Handlers)

En el desarrollo de aplicaciones de escritorio, se implementan una elevada cantidad de Composites de tipo listado con un aspecto prácticamente idéntico y cuya única diferenciación reside en los datos mostrados.

Es por esto que en el **BasicListComposite** que se acaba de crear, no se ha añadido ningún tipo de comportamiento ante eventos. Esto es debido a que únicamente se programará el aspecto una vez y se programarán tantos objetos ***handler,*** como listados diferentes sean necesarios.

El manejador o handler, contiene el código necesario para enlazar los eventos al Composite genérico, además del código para tareas como rellenar el listado o modificar o customizar el Composite.

El objetivo es trasladar cualquier código que pueda variar entre listados a una clase externa sin volverse a preocupar de modificar el formulario para cualquier otro fin que no sea el aspecto visual.

Se puede consultar a fondo el código del manejador en el código anexo a la lectura, a continuación se explican los fragmentos más importantes de los que consta el código.

La clase a crear, contiene un atributo del tipo ***BasicListComposite*** recién creado y un método ***run()*** donde se instancia el objeto lista y se invoca a los diferentes métodos para añadir eventos, rellenar la lista, cambiar el título de la ventana, etc.

```
public void run(Widget parent) {
	Shell shell = new Shell( parent.getDisplay().getActiveShell(),
					SWT.APPLICATION_MODAL | SWT.SHELL_TRIM);
	shell.setText("Mantenimiento de Cliente"); //$NON-NLS-1$
	listaComposite = new ClienteListComposite(shell, SWT.NONE);

	customize();
	addListeners();
	// $hide>>$
	search();
	refresh();
	// $hide<<$
	SWTUtils.openCentered(listaComposite, shell);
}
```

- **Customize()** - Realiza los cambios necesarios al formulario para cambiar su aspecto en cuanto a títulos, número de columnas, etc. En el siguiente apartado se explica cómo realizar este proceso.
- **addListeners()** - Añade los "escuchadores" para los eventos de los botones o del doble clic en la lista.
- **Search() –** Recoge los valores de los campos de búsqueda y realiza una búsqueda para rellenar el atributo de clase ***lista,*** que emplea el método refresh().
- **refresh() –** Refresca la lista.

8.1.5 Modificar el aspecto del listado

Como ya se ha comentado, la ventaja de esta forma de trabajo es que se consigue independizar la lógica que se esconde tras el Composite de su aspecto, de modo que puede ser reutilizado.

Para ilustrar este concepto, imagineque en lugar de dos columnas se desea que aparezcan tres columnas en la tabla y que además se quiere cambiar el rótulo de las cabeceras de ésta.

Sin modificar el código de la clase ***BasicListComposite,*** se puede realizar esta operación directamente desde el handler creando un método customize(), que realice dichas modificaciones e invocarlo desde run() previa llamada a la carga de los listeners. Un código que refleje este concepto puede ser el siguiente:

```
private void customize() {

	Table tabla = listaComposite.getBrowseComposite().getCenter()
                                                        .getTabla();

	listaComposite.getBrowseComposite().getCenter().getGrupoTabla()
		.setText("Lista de Clientes");
	tabla.getColumn(0).setText("Identificador");
	tabla.getColumn(0).setWidth(25);
	tabla.getColumn(1).setText("Cif");
	tabla.getColumn(1).setWidth(65);

	TableColumn column;
	column = new
TableColumn(listaComposite.getBrowseComposite().getCenter().getTabla(), 0, 2);
	column.setText("Nombre");
	tabla.getColumn(2).setWidth(500);
	if (esSeleccion.booleanValue()) {
		listaComposite.getBrowseComposite().getCenter().getAmb1()
				.setVisible(false);
	}

}
```

En el código anterior, se modifican los textos de las cabeceras, se cambian los anchos de columna y se añade una tercera columna *nombre*. Además por último se ocultan los botones de añadir/modificar/borrar en caso de que se quiera como composite de selección.

8.2. Creación de un informe (Reporting)

Hasta ahora se ha mostrado la forma en que se pueden mostrar los datos de la aplicación para su mantenimiento de forma cómoda por parte del usuario. Por último se va a mostrar como recoger esos datos y mostrarlos de un modo más atractivo mediante informes en los que se pueden realizar agrupaciones, estadísticas o añadir gráficas. En el mercado existen multitud de herramientas para realizar esta tarea como pueden ser ***Crystal Reports*** o ***ActiveReports***, aplicaciones de pago. En el mundo Java existe una librería opensource muy extendida cuya finalidad es la misma que las mencionadas, con la que se pueden lograr resultados excelentes de manera gratuita. Jasper Reports trabaja con ficheros XML siguiendo determinado esquema que definen el aspecto y datos del

reporte, para facilitar el trabajo con esta librería y evitar tener que conocer el etiquetado y la estructura XML de Jasper, se puede hacer uso de la herramienta gratuita ***iReport***, que permite trabajar con informes de manera sencilla y visual.

8.2.1 Creación de un informe

8.2.1.11 Informe base vacío

Una vez instalado iReport en la máquina, se arranca la aplicación de modo que se presenta al usuario una ventana similar a la que muestra la figura (depende de la versión descargada, en este documento 3.7):

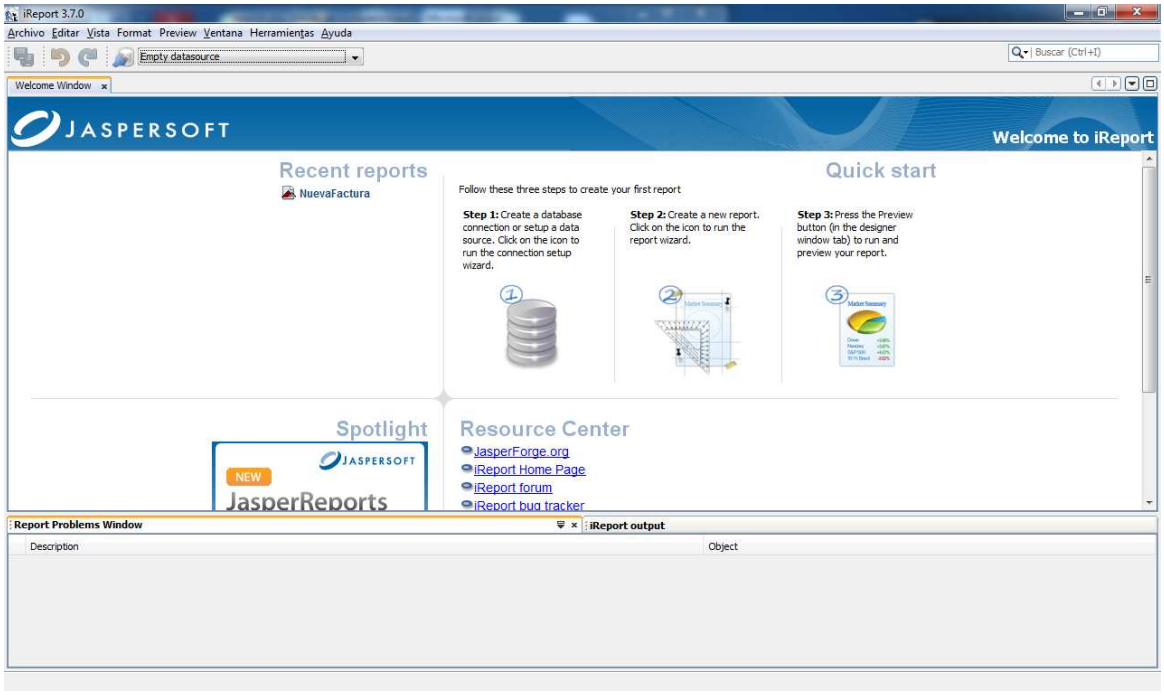


Figura 48 - Pantalla entrada iReport

En las últimas versiones de IReport, se ofrecen una serie de informes prediseñados de manera que únicamente se ha de configurar el origen de datos para el reporte y el diseño visual ya viene definido en la plantilla. A continuación se muestra como crear un informe desde cero, es decir desde una página en blanco. Para ello, se hace uso del ejemplo empleado a lo largo de todo el documento, la aplicación de facturación. En este caso se crea una factura que se podrá visualizar para imprimir o bien para exportar en Pdf.

En primer lugar, se selecciona en el menú la opción "**Archivo>Nuevo**". Con ello aparecerá un asistente para la creación de los diferentes tipos de elementos. En este caso se emplea la opción ***Report*** y tras seleccionarla la opción "***Blank A4***". Una vez

seleccionada se pulsa sobre el botón "***Open this template***" abriendo el asistente mostrado en la figura:

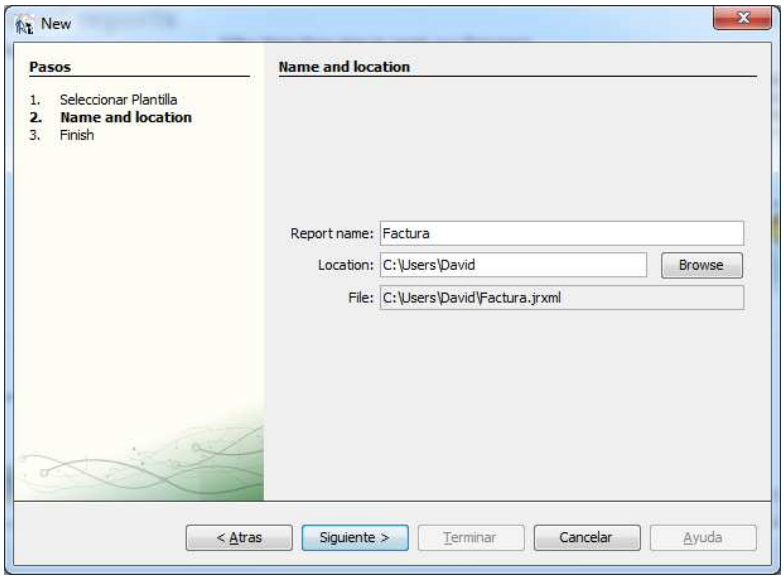
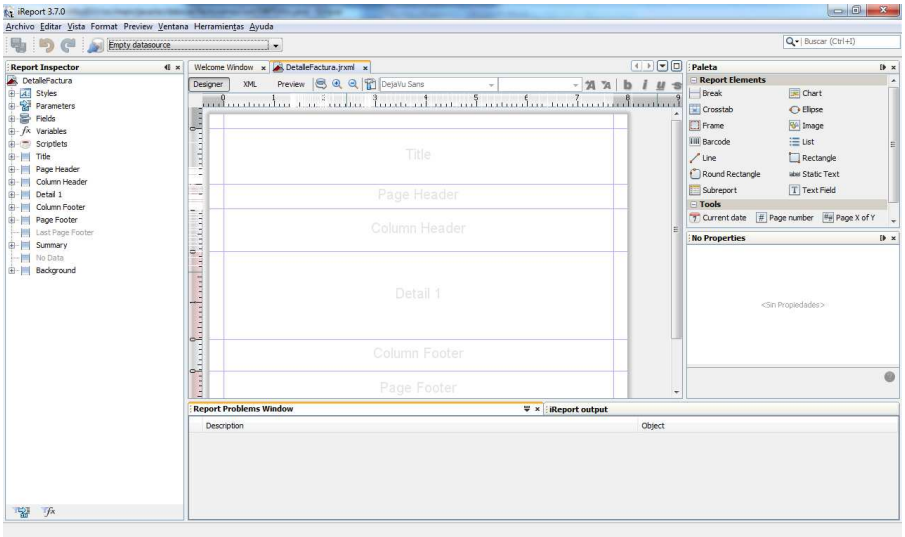


Figura 49 - Factura en blanco

En el nombre se ha de introducir algo relacionado con el informe, en este caso DetalleFactura. Sobre la localización del fichero se puede dejar para más tarde, y emplear la opción de "guardar como" para cambiar la ruta. Estos ficheros se guardan dentro del propio proyecto, dentro de un paquete bajo la carpeta de recursos. En el caso del ejemplo bajo el source path **/src/main/resources** y dentro del paquete creado para los fuentes de los informes **es.dalocar.facturamas.reports.** Tras introducir los datos necesarios se pulsa sobre el botón "***Siguiente"*** y en la pantalla de confirmación sobre ***"Terminar"***, con lo que se obtiene una situación similar a la figura siguiente:



Como se observa en el informe en blanco, un informe en Jasper se divide, como en la mayoría de sistemas de reports, en varias secciones.

- **Título** (Title) → Su contenido parece únicamente al inicio del informe.
- **Encabezado de página** (PageHeader) → Aparece al inicio de cada una de las páginas de las que conste el informe, es decir, si tiene 20 páginas se imprimirá 20 veces.
- **Encabezado de columna** (ColumnHeader) → Aparece al inicio de cada columna en el documento generado.
- **Detalle** (Detail) → El motor de Jasper genera una entrada en esta sección por cada registro o elemento que contenga el datasource pasado al informe.
- **Pie de columna** (ColumnFooter) → Aparece debajo de la sección de detalle, bajo cada columna. No se ajusta al contenido, se mantiene siempre con la altura fija que se le indica en la configuración.
- **Pie de página** (PageFooter) Aparece en la parte inferior de cada página generada en el informe. No se ajusta al contenido.
- **Otros**
    - **Última página** (LastPage)
    - **Resumen** → Aparece al final del documento. Se puede forzar a que empiece en una nueva página activando **isSummaryNewPage**. Aunque no esté activo, si no cabe la sección al final del documento empezará en una nueva página.
    - **Sin datos** → Sección que se imprime cuando no existen datos a mostrar en la sección de detalle.

8.2.1.12 Importación de campos al informe

Existen varios modos de pasar los datos al informe. Creación de un servicio para que se consulte, pasar una SQL al informe para que devuelva directamente desde base de datos los registros, pasar un listado de objetos al informe y algunos otros más.

Para la metodología escogida, la manera más cómoda de pasar estos datos es mediante el uso de colecciones de objetos que se pasan al informe. La primera tarea es conseguir trabajar de manera gráfica con los atributos de estos objetos y emplearlos como campos de texto para el informe.

Para la importación de campos al informe, previamente se ha de configurar un **classpath** de manera que apunte al directorio donde se encuentran los ficheros compilados de la aplicación. Para ello se ha de buscar la pestaña ***Classpath*** en el diálogo que se muestra tras seleccionar la opción de menú ***"Herramientas>Opciones"***.

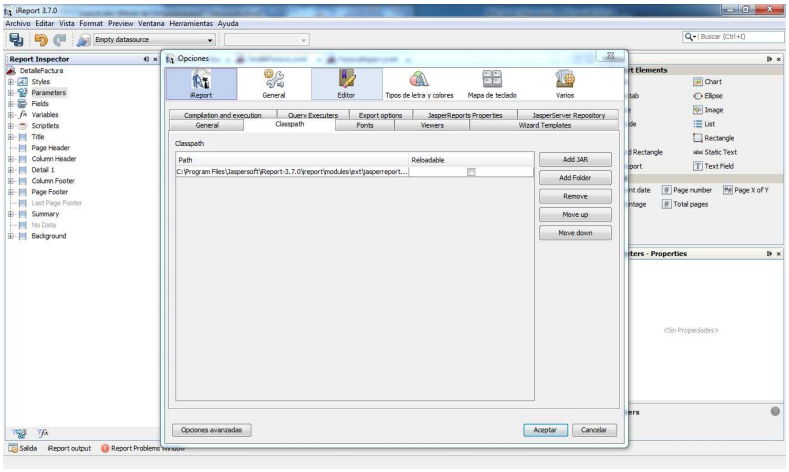


Figura 50 - Configuración del Classpath en IReport

En el diálogo se selecciona la opción de ***"Añadir Carpeta"***, y se busca la carpeta **target/classes** de la aplicación.

Una vez configurado el classpath, el siguiente paso es importar los atributos de los beans necesarios para el informe. En el caso de una factura, se muestran los datos del cliente, como el nombre, dirección y cif y datos propios de la factura como importe, número y fecha entre otros. Por ello se necesita importar al informe los beans Factura y Cliente de la lógica de negocio.

Para importar los campos, se debe hacer del diálogo que se muestra al pulsar sobre el icono de consulta de informe  , que se encuentra sobre la barra de herramientas que está justamente sobre le informe. Tras pulsar sobre el icono se abre la ventana de consulta de informe dónde se seleccionar la pestaña "**JavaBean Datasource**". En el campo ***ClassName,*** se introduce el nombre completo de la clase a importar, en este caso Factura. Si el classpath ha sido correctamente configurado en la sección anterior, al pulsar sobre el botón "**Read Attributes**", aparecen automáticamente todos los atributos de la clase.

Se selecciona todos aquellos que se desea que aparezca información en el informe y se pulsa sobre el botón "Add selected fields". El resultado será parecido al que se muestra en la figura a continuación:

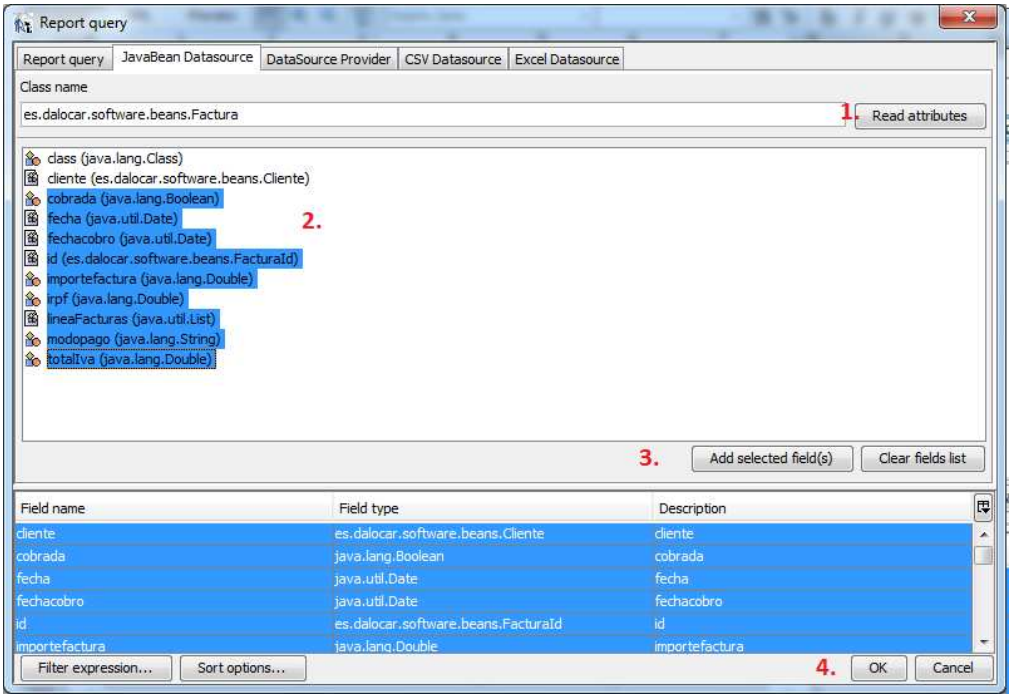


Figura 51 – Pasos a realizar para importar campos al informe

Tras cerrar el diálogo en la estructura de campos del informe aparecen todos los campos importados.

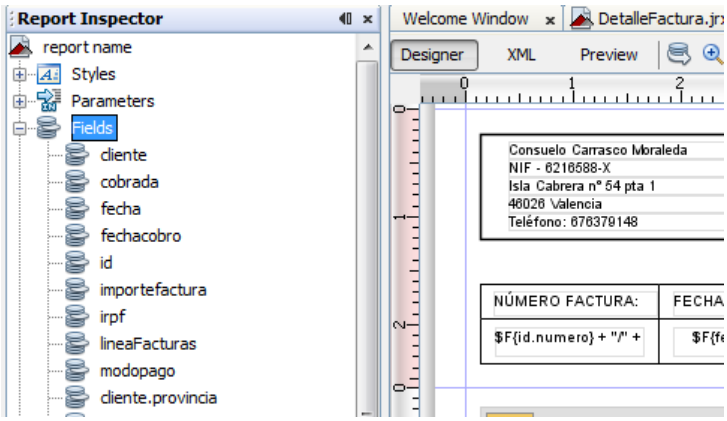


Figura 52 - Resultado de la importación de campos

Especial atención merecen los campos "agregados". Se ha de tener en cuenta que se puede acceder a cualquier atributo de la clase que disponga de un método ***get***, sea del tipo que sea, lo cual permite tener acceso a atributos de otras clases referenciadas por la clase con la que se está trabajando actualmente.

Ejemplo:

```
public class Factura {
      private FacturaId id;
```

```
    private Cliente cliente;...
      }

      public class FacturaId {
            private Integer numero;
            private Integer anyo;
      …
      }

      public class Cliente {
            private String nombre;
            private String direccion;
      …
      }
```

El modo de importar los campos que se han descrito es muy similar al que se ha empleado con los atributos simples. En este caso se hace doble clic sobre "***id***" para que muestre los atributos simples de los que se compone y se seleccionan para su importación.

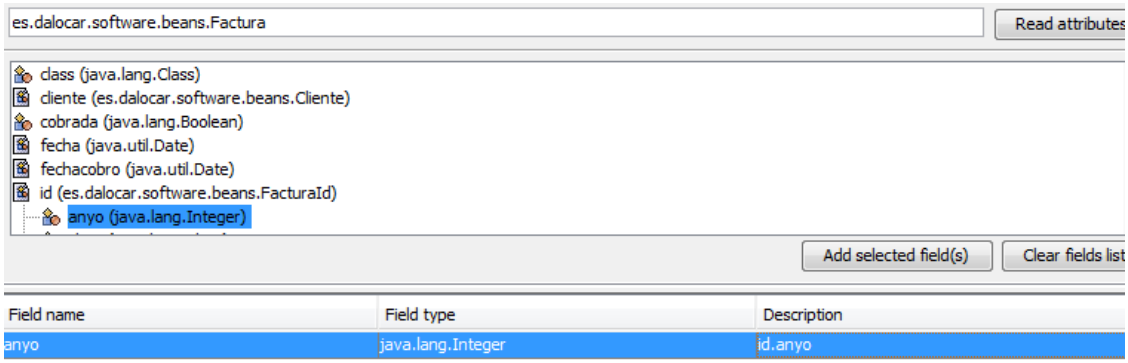


Figura 53 - Agregar un campo anidado al informe

8.2.1.13 Añadir elementos de diseño al informe

Elementos dinámicos

Para añadir elementos de tipo dinámico (variables, parámetros y campos) basta con arrastrarlos a la ventana de diseño del formulario.

Para añadir campos al formulario basta con arrastrar desde la ventana de campos hacia el lugar donde queremos que aparezca el campo.

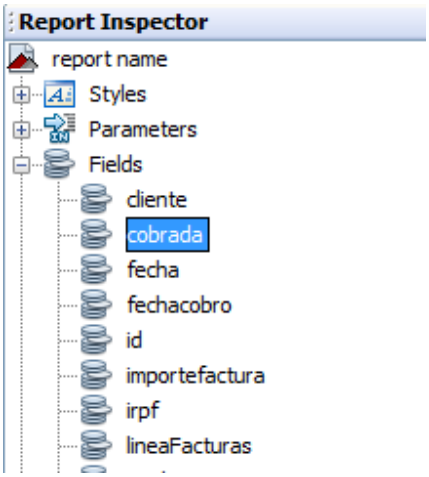


Figura 54 - Agregar elementos dinámicos al informe

Elementos estáticos

Se entiende por elementos estáticos, aquellos que son independientes de los datos de la aplicación y que no cambiarán en el informe. Elementos tales como etiquetas, líneas de separación, rectángulos para marcos, etc. Para ello se selecciona el icono de alguno de los elementos estáticos en la paleta de herramientas de iReport y después se selecciona la localización que se desea en el informe para el elemento.

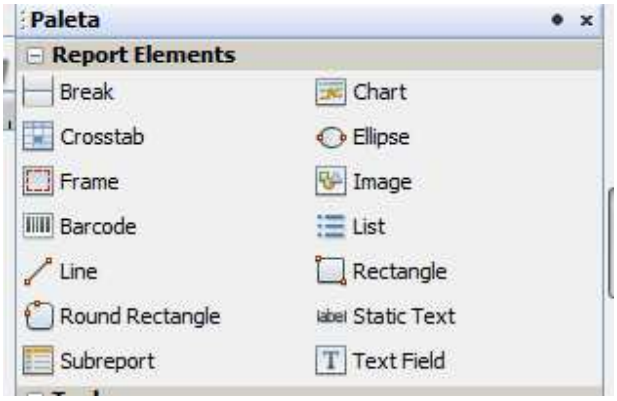


Figura 55 - Paleta de herramientas de iReport

8.2.1.14 Crear agrupaciones

Como ya se explica en el inicio del presente capítulo Los informes en JasperReports se dividen en una serie de bandas, porciones horizontales de la página que se imprimen y modifican en altura dependiendo de sus propiedades y de su contenido.

Además de las secciones comentadas se pueden añadir dos más mediante la creación de agrupaciones, el **groupHeader** y el **groupFooter.**

Un grupo es una forma flexible de organizar la información en un informe. Consiste en una secuencia de registros con algo en común, como el valor de un cierto campo.

Las agrupaciones se definen a través de una expresión. Jasper evalúa esa expresión y en el momento que la expresión cambia, inicia un nuevo grupo.

Creación de un nuevo grupo

Para crear un nuevo grupo, se hace clic derecho sobre el informe en la vista "Report Inspector". Tras pulsar sobre la opción aparece un asistente en el que solicita el campo de los disponibles por el que se desea agrupar y el nombre que se desea para el grupo. Se ha de tener en cuenta que se pueden crear tantas agrupaciones como de campos dispone el informe, por ello dar un nombre descriptivo es importante a la hora de identificar en el diseñador las agrupaciones.

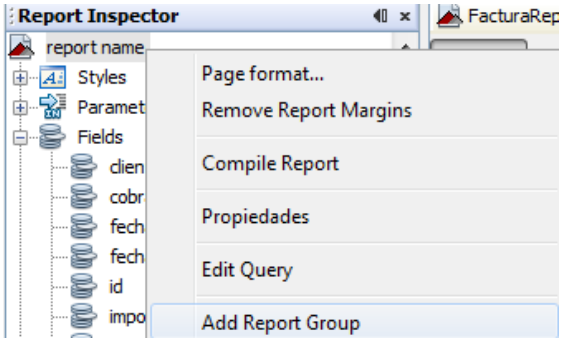
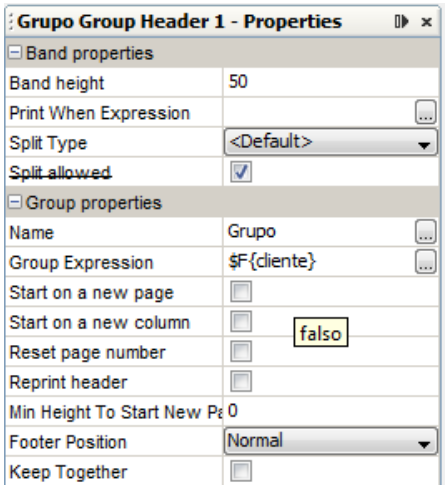


Figura 56 - Creación de un nuevo grupo en iReport

Al seleccionar la cabecera de grupo en la vista del inspector de informes, en la ventana de propiedades aparece lo siguiente:



Algunas de las propiedades a destacar son las siguientes:

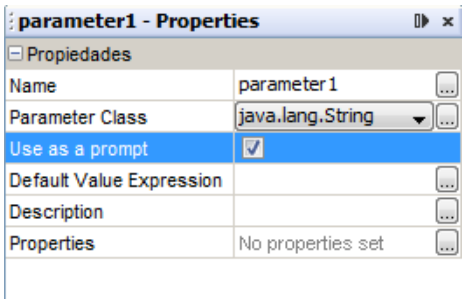
- **Start on a new column**: Fuerza una ruptura de página al inicio de cada nuevo grupo.
- **Start on a new page:** Fuerza una ruptura de página al final del grupo.
- **Reset page number:** Inicializa la variable número de páginas al inicio de cada grupo.
- **Reprint header:** Si el contenido del grupo queda en dos o más páginas diferentes, se imprimirá de nuevo el encabezado en cada una de ellas.
- **Min height to start new page:** Si es distinto de 0, Jasper empieza a imprimir en una nueva página si el espacio inferior es menor al especificado.
- **Band Height (Cabecera y pie):** Tamaño de las bandas de cabecera y pie.

8.2.1.15 Paso de parámetros

Los parámetros son valores que usualmente se pasan al informe desde el programa que crea el informe y se pueden usar para guiar el comportamiento del informe o bien para pasar información adicional al informe como el título del informe o una imagen.

Creación de un nuevo parámetro

Para crear un nuevo parámetro, se va a la vista de la estructura del documento y se hace clic derecho sobre *Parameters* y *Add Parameter*. Por defecto se crea un parámetro con nombre ***parameter1.*** Se selecciona este parámetro y se cambia su nombre y tipo en la ventana de propiedades a la derecha de la ventana de diseño.



Paso de parámetros

Cuando se crea un nuevo informe, se debe proporcionar tanto un Mapa de parámetros como una colección con los beans a mostrar en el informe. Como clave para el parámetro se usa el nombre proporcionado en el asistente del iReport. Se introduce uno por cada uno de los parámetros que se vaya a emplear en el informe.

```
Map map = new HashMap();
map.put("parameter1", param1);
map.put("parameter1", param2);
```

8.2.1.16 Mostrar el informe

En el proyecto de ejemplo, se han implementado las tres clases básicas que se necesitan para crear y representar los informes. Estas clases se encuentran bajo el paquete ***es.dalocar.facturamas.reports***:

Report.java: Interfaz a implementar por todos los informes de la aplicación.

ReportBase.java: Implementación básica de la interfaz y clase a la que se deberá extender con cada uno de los informes que se crean para añadir la información de nombres de parámetros y del fichero .**jasper** del report compilado a mostrar.

Reporter.java: Provee el método ***view***, que acepta como parámetro un Report y se encargará de abrir un ***JasperViewer*** para mostrarlo.

Además de estas tres clases, se crea una nueva clase por cada fichero jasper creado. A esta clase se le añade atributos estáticos y finales para indicar el nombre del fichero compilado con el informe y otros tantos como parámetros se requieran el informe.

A continuación se muestra un fragmento de la clase creada para mostrar el detalle de una factura en la aplicación ejemplo.

```
public class FacturaDetailReport extends ReportBase {

	/** Parámetro resultado de la petición: ADMITIDO/DENEGADO */
	public static final String PARAM_RESULTADO = "PARAM_RESULTADO";

	/** Nombre de archivo del informe */
	private static final String FILENAME = "FacturaReport.jasper";
```

El último requisito para visualizar el informe es rellenar la colección de beans y el mapa de parámetros y tras ello, llamar a la clase ***Reporter*** para que muestre el informe.

El ejemplo implementado para esta acción se puede encontrar en la clase ***EditFacturaComposite*** del ejemplo bajo el método ***doPrint().*** En caso de no necesitar parámetros, únicamente se pasa un mapa vacío.

8.2.1.17 Informes avanzados (Subreports)

El uso de subreports permite mostrar colecciones que se relacionan con el bean principal que está manejando el informe. Un ejemplo muy claro aparece en el detalle de una factura, donde cada una de sus líneas forma parte de una colección:

```
	public class Factura implements java.io.Serializable {
		// ……
```

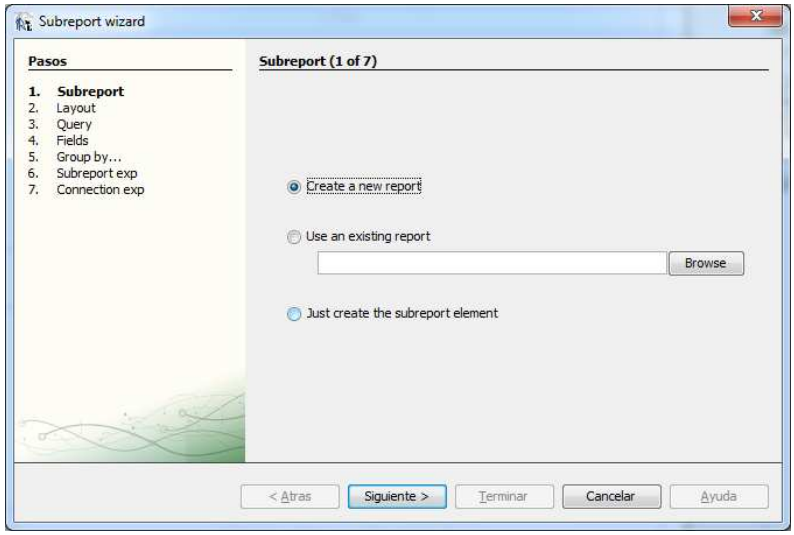
```
private List lineaFacturas = new ArrayList();
```

En primer lugar, se añade un campo al informe, del mismo modo que se hace con los tipos simples (Consultar al inicio de este capítulo). Este campo se corresponde con la colección de beans que se desea mostrar en el subreport.

Para crear un nuevo subinforme dentro del report principal, se debe hacer clic sobre el icono de la paleta de herramientas de subinforme 

, tras esto se puede crear en el informe un nuevo subreport del mismo modo que se crean el resto de componentes, pinchando y arrastrando sobre el report para definir el tamaño del elemento.

Una vez creado el recuadro del subreport, aparece el asistente de creación de subreports. En la primera pantalla, se elige la opción “Crear un nuevo informe”.



En la siguiente pantalla se pide un diseño o plantilla para el nuevo informe, se creará un informe en blanco al igual que se crea para el informe padre. Cómo ***datasource*** únicamente se permite en este modelo, uno en blanco. Se pulsa en siguiente hasta llegar a la pantalla del asistente donde se solicita un nombre para el subreport y que fichero se genera. Aquí además se selecciona que se desea una ruta estática para el fichero.

El siguiente paso es configurar el subreport. Se selecciona y se modifican sus propiedades en la vista de ***Properties***. la Lo siguiente que haremos será hacer clic derecho con el ratón sobre el subreport y elegir propiedades.

A continuación se detallan las propiedades para el informe de ejemplo, el detalle de una factura.

Parámetro	**Valor**
Subreport Expression	$P{SUBREPORT_DIR} + "LineaFacturaSubReport.jasper"
Connection type	Use a datasource expression
Datasource	new

Expression	net.sf.jasperreports.engine.data.JRBeanCollectionDataSource(**$F{lineaFacturas}**)

Ahora se puede editar el subreport como si se tratará de un report normal haciendo doble clic sobre él. Para visualizar el informe creado, se hace clic derecho sobre el informe en el diseñador y se selecciona la opción open ***Report***.

Capítulo 9. Empaquetado y despliegue

Finalizado el desarrollo y llegado a una versión estable de la aplicación hay que distribuirla. Una aplicación seria **no** se ejecuta desde el entorno de desarrollo y es necesario crear paquetes de manera que se pueda usar fácilmente por parte del usuario final sin requerir de un desarrollador para su instalación.

Cada vez que se crea una nueva versión de la aplicación esta debe ser empaquetada y distribuida a los usuarios de modo que es conveniente el hacer uso de herramientas que faciliten esta tarea.

Las personas que provienen del mundo visual basic, están acostumbras a que al compilar su aplicación se genere un ejecutable para el sistema operativo Windows. En el caso de lenguajes como java no es posible crear ejecutables para windows de manera directa, se ha de acudir a envoltorios o wrappers que se encargan de crear un ejecutable que internamente lanza la aplicación a ejecución sobre la máquina virtual.

Existen varias herramientas para facilitar esta tarea, a continuación se comentan algunas de ellas.

9.1. JSmooth

JSmooth es un envoltorio para ejecutables Java (.jar). Se encarga de crear ejecutables nativos de Windows (.exe) para aplicaciones Java. Evita problemas en el desarrollo java y lo hace más amigable. JSmooth es capaz de detectar una máquina virtual Java por sí mismo, en caso de no existir, el wrapper puede bajar e instalar automáticamente una máquina virtual apropiada o simplemente mostrar un mensaje o redirigir al usuario al sitio web.

Provee una serie de envoltorios para las aplicaciones Java, cada uno de ellos con su propio comportamiento.

IZPack

IZPack es una solución para empaquetado, distribución y despliegue de aplicaciones. Permite generar un único instalador para múltiples plataformas, ofreciendo una alternativa a soluciones nativas que son específicas para una única plataforma.

El único requisito para su uso es disponer de una máquina virtual corriendo sobre el sistema.

Launch4J

Launch4j es una herramienta multiplataforma para envolver aplicaciones java distribuidas como jars en ejecutables nativos de Windows. El ejecutable puede ser configurado para buscar una versión concreta de JRE o usar una empaquetada, además de configurar opciones como el tamaño de heap máximo e inicial. Provee otras opciones

como cambio de icono de la aplicación, nombre de proceso personalizado o llevar a la página de descarga de Java en caso de no encontrar una versión apropiada de la JRE.

Para el proyecto que se viene desarrollando en esta lectura, se elige JSmooth por su fácil integración con Ant y Maven. A continuación se detallan los pasos que se siguen para conseguir empaquetar y generar el ejecutable de la aplicación.

9.2. Instalación de JSmooth

El primer paso es realizar la descarga de la herramienta desde la página web de los desarrolladores[11]. Existen varios archivos disponibles y se escoge el instalador ejecutable para la última versión (jsmooth-xxx-.exe).

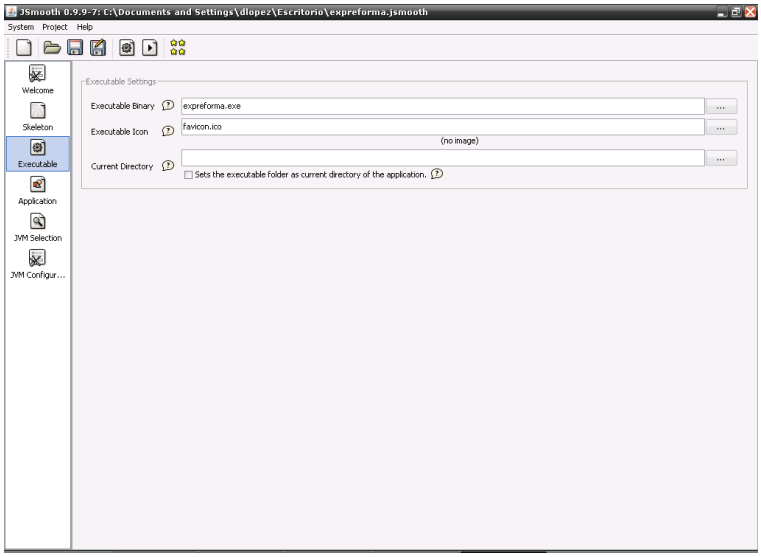
La instalación es muy sencilla y no se va a detallar los pasos a seguir, únicamente se ha de aceptar la licencia (de tipo GNU) y pasar a través del asisntente.

Uso de JSmooth

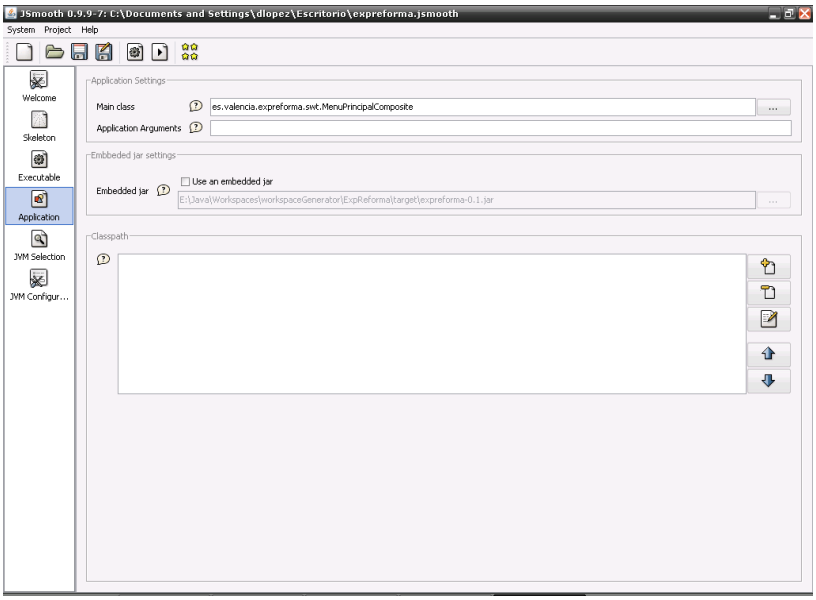
JSmooth permite guardar la configuración en un fichero *.jsmooth con formato xml. Este fichero de configuración es el que se incluye en el proyecto. En caso de tener un fichero de jsmooth ya disponible en otras aplicaciones se puede hacer uso de él y cambiar la configuración desde el editor de XML de Eclipse. En este caso se va a generar uno nuevo para la aplicación de ejemplo a través de la herramienta gráfica. Este fichero puede ser usado posteriormente como base.

En la pestaña Skeleton, se selecciona el tipo de esqueleto que se desea para la aplicación, normalmente son aplicaciones de escritorio, por lo que se deja seleccionada la opción **Windowed Wrapper**. Se puede cambiar el mensaje a mostrar al usuario en caso de que no tenga instalada una máquina virtual en su equipo.

[11] http://jsmooth.sourceforge.net/download.php



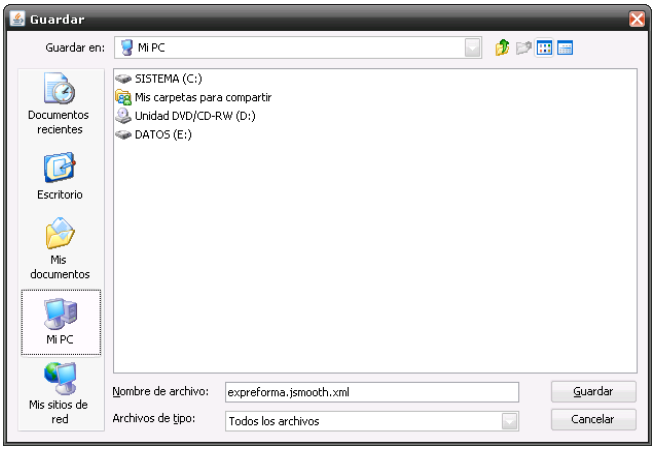
En la pestaña **Executable**, se encuentran las opciones referentes al fichero generado: nombre, icono y directorio de la aplicación. Como icono se emplea favicon.ico que se encuentra en el mismo directorio donde va a parar el fichero de configuración resultante.



En la pestaña **Application** se debe modificar la Main Class, e indicar la clase que contiene el punto de entrada a la aplicación. En la aplicación que se sigue como ejemplo es **MenuPrincipalComposite** contenida en el paquete **es.dalocar.software.swt.** Se ha de rellenar el campo incluyendo el nombre de paquete.

Con estos parámetros de configuración es suficiente para generar el fichero de configuración.

Se elige la opción **Save** y se le da nombre al fichero con extensión xml. En este caso **facturamas.jsmooth.xml.**



Este archivo debe ser copiado al directorio **src/main/exe** del proyecto. Posteriormente se debe añadir al archivo las entradas correspondientes al classpath de la aplicación.

```
<classPath>facturamas-0.1.jar</classPath>
```

Se puede hacer que se muestre una consola de debug dando valor 1 a la entrada Debug. Esta opción es recomendable a la hora de depurar problemas con el arranque de la aplicación a partir del ejecutable.

```
        <key>Debug</key>
        <value>0</value>
```

9.3. Generación del ejecutable con Ant

Apache Ant es una herramienta empleada normalmente en fase de compilación y construcción (build). Es parecido a la herramienta make, usada con C pero sin depender del sistema operativo. No depende de órdenes de shell si no que se basa en archivos de configuración XML y clases Java para realizar las diferentes tareas.

Para trabajar con Ant, se ha de descargar la herramienta de la página de apache correspondiente[12]. Una vez descargada se descomprimen los ficheros en un directorio del sistema, y se incluye en el path la ruta hacia el directorio bin de Ant. Con ello ya se puede trabajar desde la línea de comandos.

Simplemente se debe de ejecutar la tarea **ant**, con el target del fichero **build.xml** que queramos ejecutar.

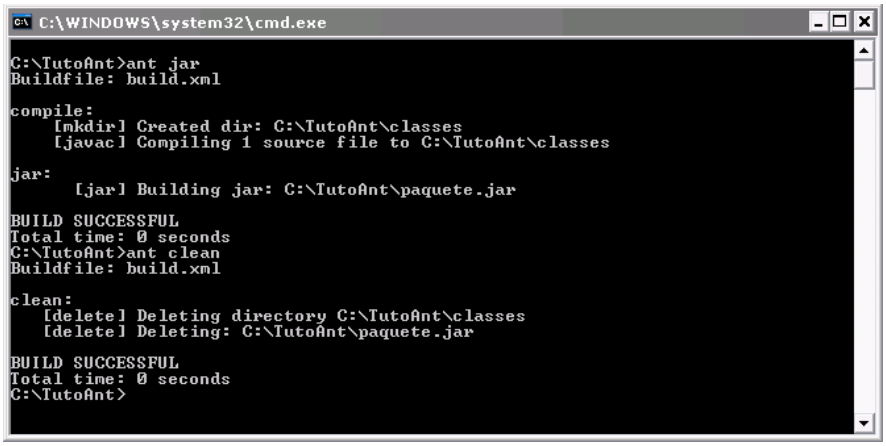
Ejemplo de fichero **build.xml**.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<project name="Ejemplo" default="compile">
```

[12] **http://ant.apache.org**

```
    <target name="clean" description="borrar archivos temporales">
        <delete dir="classes"/>
        <delete file="paquete.jar"/>
    </target>
    <target name="compile"
     description="compilar el codigo java a un archivo class">
        <mkdir dir="classes"/>
        <javac srcdir="." destdir="classes"/>
    </target>
    <target name="jar" depends="compile"
     description="crear un archivo Jar para la aplicación">
        <jar destfile="paquete.jar">
            <fileset dir="classes" includes="**/*.class"/>
            <manifest>
                <attribute name="Main-Class" value="Principal"/>
            </manifest>
        </jar>
    </target>
</project>
```

Resultado de la ejecución de la orden **ant jar** y **ant clean.**



Mediante el uso de los plugins de Maven, se consigue integrar Ant con Maven evitando la instalación Ant.

El primer paso es crear un fichero **build.xml** y definir una nueva etiqueta que provea a ant la información necesaria para crear tarea de **jsmooth**. Además se emplea un fichero de **properties** donde indicarle el directorio donde se encuentra la librería jsmooth para ant, dónde se encuentra los skeletons de jsmooth, el nombre del ejecutable a generar y el fichero de configuración de jsmooth para la aplicación.

```
    <property file="custom.properties" />

    <!-- JSmooth task definition -->
```

```
      <taskdef name="jsmoothgen"
                  classname="net.charabia.jsmoothgen.ant.JSmoothGen"
               classpath="${jsmooth.ant.jar}"/>
```

Fichero **custom.properties.**

```
jsmooth.skeletons.dir=C:/Program Files/JSmooth 0.9.9-7/skeletons
jsmooth.ant.jar=C:/Program Files/JSmooth 0.9.9-7/lib/jsmoothgen-ant.jar
jsmooth.metadata.file=facturamas.jsmooth.xml

jsmooth.exe.file=facturamas.exe
```

En el fichero **build.xml** se añade el **target.exe** que es el encargado de generar el ejecutable.

```
<!-- ================================
                target: exe
               ================================ -->
<target name="exe">
      <jsmoothgen project="${jsmooth.metadata.file}"
            skeletonroot="${jsmooth.skeletons.dir}" verbose="true" />

      <move file="${jsmooth.exe.file}" todir="../../../target"/>
</target>
```

9.4. Integración JSmooth, Ant y Maven2

Como se ha comentado, es posible integrar tareas de ***Ant***, dentro del fichero de configuración de ***Maven***. Para ello se hace uso de los plugins de Maven.

En este caso se usa el plugin **maven-antrun-plugin**. Se incluyen las siguientes líneas en el fichero **pom.xml.** Con estas líneas se le indica a ***Maven*** que durante la fase de empaquetado deberá ejecutarse el plugin. Además se le indica que el fichero **build.xml** está en el directorio **src/main/exe** y debe ejecutar el target **exe**.

```
<!-- Creación del archivo exe -->
<plugin>
<artifactId>maven-antrun-plugin</artifactId>
  <executions>
    <execution>
      <phase>package</phase>
        <configuration>
          <tasks>
            <ant dir="src/main/exe" inheritRefs="true">
              <target name="exe" />
            </ant>
          </tasks>
        </configuration>
        <goals>
          <goal>run</goal>
        </goals>
```

```
    </execution>
  </executions>
</plugin>
```

Con ello conseguiremos que en el directorio **target**, tras ejecutar el goal **package** se cree el ejecutable de la aplicación. El siguiente paso es crear un zip con la aplicación y el ejecutable generado para distribuirla.

9.5. Empaquetado de la Aplicación

9.5.1 Creación del jar de la aplicación

La creación del **.jar** se realiza mediante la ejecución del goal de Maven **package** y las acciones a ejecutar se definen en el fichero **pom.xml** en la sección correspondiente al **maven-jar-plugin**. Se puede ver a continuación un ejemplo correspondiente a la aplicación seguida como ejemplo a lo largo del proyecto:

```
<plugin>
 <artifactId>maven-jar-plugin</artifactId>
 <configuration>
  <archive>
   <manifestEntries>
      <Class-Path>classes/</Class-Path>
   </manifestEntries>
   <manifest>
      <mainClass>es.dalocar.facturamas.swt.MainComposite</mainClass>
      <addClasspath>true</addClasspath>
      <classpathPrefix>lib/</classpathPrefix>
   </manifest>
  </archive>
 </configuration>
</plugin>
```

Clave	Valor
manifestEntries	Lista de pares clave/valor que se deben añadir en el manifest
Manifest	Configuración para el fichero ***Manifest.mf*** que será contenido en el directorio ***META-INF*** del jar.
mainClass	Clase que contiene el punto de entrada a la aplicación.
addClasspath	Indica si se debe generar una entrada para el ***classpath*** o no. El valor por defecto es false, así que deberemos especificar explícitamente true.
classpathPrefix	Texto a añadir delante de todas las entradas del ***classpath***, en este caso "lib/" puesto que las dependencias de la aplicación irán a parar al directorio lib.

9.5.2 Creación del zip con la aplicación

El zip de la aplicación se genera durante la ejecución del goal **assembly**. En el fichero **pom.xml** se puede encontrar la sección correspondiente al plugin **maven-assembly-plugin**.

```
<plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
            <descriptorSourceDirectory>
              src/main/assembly
            </descriptorSourceDirectory>
      </configuration>
</plugin>
```

descriptorSourceDirectory	Directorio en el que buscar ficheros de descripción para el empaquetado.

Como se puede observar en el **pom.xml**, se ha especificado como directorio para los descriptores el directorio ***src/main/assembly***, es en este directorio donde se deben situar los ficheros xml de configuración.

Como ejemplo se puede tomar el fichero **bin.xml** contenido bajo ese directorio en la aplicación de ejemplo.

```
      <id>bin</id>
```

Con el tag **id** le indicamos a maven el tipo nombre que dará al paquete resultante, que será de la forma **artifactId-version-id.xxx.** xxx representa la extensión del archivo resultante y dependerá del formato elegido en la sección formats. En este caso se emplea formato zip, aunque se ofrecen otros para plataformas Linux como tar, gz, bz, etc.

```
      <formats>

            <format>zip</format>

      </formats>
```

En la sección **fileSets** se le indica mediante entradas **fileSet** los directorios y ficheros a incluir en el paquete. A continuación se explica cada una de las contenidas en **bin.xml.**

```
<fileSet>
	<directory>src/main/config/prod</directory>
	<outputDirectory>classes</outputDirectory>
	<includes>
		<include>**/*.*</include>
	</includes>
</fileSet>
```

En la entrada anterior, se le indica que se quiere que el contenido del directorio **src/main/config/prod**, sea incluido en el ***classes*** del ***zip*** resultante. En la sección *includes*, se pueden establecer filtros de los ficheros a incluir del directorio en cuestión. En el ejemplo s incluyen todos los ficheros y directorios bajo **prod.**

```
<fileSet>
	<directory>target</directory>
	<outputDirectory></outputDirectory>
	<includes>
		<include>FacturaMAS.jar</include>
		<include>*.exe</include>
		<include>classes</include>
	</includes>
</fileSet>
```

Con el fragmento de código anterior se le indica a Maven que se quiere que el contenido del directorio target del proyecto vaya a parar al directorio lib dentro del zip resultante y se establece un filtro de la forma **FacturaMAS.jar** para indicar que únicamente se quiere dentro del zip el jar que se habrá generado durante la fase de package

```
<fileSet>
	<includes>
		<include>*.dll</include>
	</includes>
</fileSet>
```

Esta entrada incluye las dll en el raíz del proyecto al zip de la aplicación. Esta entrada es necesaria para adjuntar las librerías de swt.

```
<dependencySets>
	<dependencySet>
		<outputDirectory>lib</outputDirectory>
		<unpack>false</unpack>
		<scope>runtime</scope>
	</dependencySet>
</dependencySets>
```

Con este tag se le indica que se quiere guardar las dependecias en el directorio lib dentro del paquete y que no se quieren como jar sin desempaquetar.

Se recomienda echar un vistazo los ficheros ***pom.xml*** y ***bin.xml*** del código anexo y tratar de comprender las entrañas de estos.

Tras ejecutar el goal de Maven ***package,*** se obtiene un fichero zip listo para descomprimir con la aplicación preparada para su ejecución.

Capítulo 10. Conclusiones y consejos

A lo largo de este documento se han dado las pinceladas de cómo construir una aplicación apoyándose en muchas herramientas. Para cada una de ellas, simplemente se ha dado una pincelada y una breve introducción a su uso. Lleva tiempo llegar a tener control sobre todas las herramientas y no siempre se conseguirá dominar todos los aspectos de ellas, sobretodo de Maven. Con el tiempo y paciencia, se puede acelerar todas las fases del desarrollo de una manera casi exponencial apoyándose en estas herramientas.

Durante este proyecto se ha realizado mucho hincapié en la parte práctica y aunque ya se ha comentado durante el proyecto es muy recomendable apoyarse en el código anexo, puesto que durante todo el documento se recurre a ejemplos que se encuentran completos en el código fuente y que combinando código y documento se puede facilitar enormemente la comprensión tanto del texto como del código.

El siguiente paso, es convertirse en todo un programador Java, ¡suerte!

Capítulo 11. Índice de figuras

Capítulo 12. Referencias

Entorno de desarrollo Eclipse. http://www.eclipse.org

Apache Ant. http://ant.apache.org/

Hibernate. http://www.hibernate.org

iReport. http://www.jasperforge.org/sf/projects/ireport

JasperReports. http://www.jasperforge.org/

Jigloo. http://www.cloudgarden.com/jigloo

JSmooth. http://jsmooth.sourceforge.net/

Maven 2. http://maven.apache.org

Spring Framework. http://www.springframework.org

Subversion. http://subversion.tigris.org

Subversive. http://www.polarion.org

SWT. http://www.eclipse.org/swt

Capítulo 13. Bibliografía

- **Arquitecturas multicapa**
  http://www.programacion.com/tutorial/jap_jsfwork/3/

- **Primeros pasos con Maven**
  http://www.chuidiang.com/java/herramientas/maven.php

- **SWT/JFace in Action**, GUI Design with Eclipse 3.0
  Matthew Scarpino, Stephen Holder, Stanford Ng, and Laurent Mihalkovic. Ed. Manning

- **SWT: A Developer's Notebook**
  Tim Hatton, Ed. O'Reilly Media, Octubre 2004

- **Pro Hibernate 3**
  David Minter. Ed. Apress. ISBN: 978-1-590-59511-4

- **Pro Spring**
  Rob Harrop. Ed. Apress. Enero 2005. ISBN: 1590594614