etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Proyecto Final de Carrera

Ingeniería Informática

Desarrollo de Editores MOSKitt-FEFEM sobre modelos EMF persistidos en Base de Datos haciendo uso de CDO y FMF.

Código:

DSIC-142

Autor:

Carlos Sánchez Rodríguez

Director del Proyecto:

Joan Josep Fons Cors

Índice de contenido

1. Introducción

Este bloque de introducción se divide en seis apartados.

En el primer apartado se describirá el entorno de desarrollo donde se desarrolla el proyecto, a continuación del primer apartado en el segundo, se presentara las motivaciones que me han llevado a la realización del proyecto. En el tercer apartado se presentan los objetivos que se pretenden alcanzar con este trabajo siguiéndolo de una lista de palabras claves que se utilizaran durante la memoria. En el quinto apartado se pretende explicar en que situación me encontré el entorno de desarrollo del proyecto y como se pretende que funcione a partir del proyecto. El ultimo apartado de la introducción lo que se pretende es explicar como va a ser la estructura de esta memoria.

1.1 Ámbito

El trabajo realizado en este proyecto forma parte de MOSKitt (*Modeling Software KITT*), una herramienta CASE *Open Source* gratuita, desarrollada por la *Consellería de Infraestructuras y Transporte* (CIT). Está basada en Eclipse y su fin es dar soporte a la metodología *gvMétrica* (una adaptación de *Métrica III*). Entre otras funcionalidades, nos permite definir y transformar modelos.

Basada en una arquitectura de plugins, se llevan a cabo continuas iteraciones para ir obteniendo versiones estables/correctas de la herramienta, añadiendo nuevas funcionalidades y solventando errores existentes al final de cada una de ellas.

1.2 Motivación

Desde que existe la informática ha existido la necesidad de compartir la información. Ponemos por ejemplo el caso de una empresa en la que un empleado en la 5ª planta ha redactado un documento que lo necesita uno de la 1ª planta, al principio con la inexistencia de internet el empleado de la 5ª planta necesitaba copiar esa información a un elemento externo que permitieran el intercambio de la información (Diskettes, CD, DVD, USB Pen Drive ,...).

El mecanismo de intercambio de la información de un ordenador a otro a través de ese elemento externo no se producía con la rapidez esperada, ya que requería pasar primero al elemento externo y del elemento externo al ordenador del otro empleado, además de carecer de la edición simultanea del mismo documento.

Ahora pongamos el caso de que en vez de ser un documento, sea un formulario que almacena el registro de personas que entran dentro de una habitación, cada vez que una persona entra en la habitación otra persona registra la entrada en el formulario y cuando sale da de baja en el mismo formulario.

Con el ejemplo anterior, solo una única persona puede dar de alta y de baja las personas que entran el la habitación por lo que puede llegar a ser un problema cuando varias o muchas personas quieren entrar a la vez en la habitación, lo que se pretende con este proyecto es la posibilidad de añadir mas personas que den de alta y de baja a las personas que entran en la habitación editando el mismo formulario de manera simultanea.

1.3 Objetivos

El objetivo principal es la creación de un nuevo modulo en MOSKitt, este nuevo modulo gestionara la persistencia de recursos que han sido editados a través de editores de formularios, estos recursos además tendrán la capacidad de ser editados simultáneamente desde distintos clientes.

Este objetivo se dividirá en tres grandes bloques, el primer bloque sera la construcción de editores de formulario donde los datos se puedan almacenar dentro de una base de datos, el segundo bloque incluye la creación de una vista nueva dentro de MOSKitt capaz de gestionar esos datos almacenados y como tercer bloque incluye la gestión entre los distintos clientes que aceden al recurso y como controlar la simultaneidad entre los distintos clientes que se conectan a la vez al recurso para editarlo.

Para el primer bloque, MOSKitt tiene ya un sistema para la creación de editores de formulario dentro de un modulo llamado MOSKitt-FEFEM, este sistema requiere una alta capacidad en conocimientos de programación, ya que requiere la continua edición de código. Por ello, una empresa externa al proyecto MOSKitt ha implementado un nuevo modulo dependiente de FEFEM llamado MOSKitt-FMF capaz de crear un

modelo que contendrá los elementos que forma el editor (paginas, secciones, …) y a partir de este modelo generar el código del editor de manera automática, con el código generado de manera automática podremos editarlo para dar las ultimas pinceladas al editor. El resumen del primer bloque es crear editores de formulario basándose en la tecnología actual FMF y que además soporten la persistencia en una base de datos.

Para el segundo bloque, debemos de crear una nueva vista, llamada MOSKittDataStore que sea capaz gestionar el acceso a los recursos almacenados en la base de datos, por ello gestionara el acceso a las distintas bases de datos donde se encontraran los recurso, además deberá de implementar la manera de abrir un recurso con su editor de formulario correspondiente.

En el tercer bloque, se deberá de gestionar las diferentes notificaciones recibidas cuando un recurso a sido modificado y de esta manera invocar automáticamente simultaneidad de los datos, como objetivo del tercer bloque se incluye tambien controlar y gestionar cuando dos clientes editan simultáneamente la misma propiedad en el recurso.

1.4 Palabras clave

- **Eclipse:** Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido" .

- **MOSKitt:** Herramienta CASE libre, basada en Eclipse. Se utiliza para dar soporte a la metodología gvMétrica.

- **gvMétrica:** Es una metodología de desarrollo de software dirigido por modelos definida por la Consellería de Infraestructura y Transportes con el objetivo de agilizar sus proyectos de desarrollo de software. Es una adaptación de la metodología Métrica III.

- **Ingeniería dirigida por modelos (MDE):** Se trata de una metodología de desarrollo de software que se centra en la creación y explotación de modelos de dominio. Tiene por objetivo aumentar la productividad mediante la maximización de la compatibilidad entre los sistemas, lo que simplifica el

proceso de diseño y promueve la comunicación entre los usuarios y equipos que trabajan en el sistema.

- **Metamodeling:** Es la construcción de una colección de “conceptos” dentro de un determinado dominio. Un modelo es una abstracción de los fenómenos en el mundo real; un metamodelo es otra abstracción que destaca las propiedades del propio modelo. En definitiva, un metamodelo es un modelo cuyas instancias son modelos.

- **Object Management Group (OMG):** Es un consorcio dedicado al establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML o XMI. Es una organización sin ánimo de lucro que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para las mismas.

- **Plug-in (o plugin):** Es una aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy especifica. Esta aplicación adicional es ejecutada por la aplicación principal e interactúan por medio de la API.

- **View:** En la plataforma Eclipse una vista es un elemento dentro de Eclipse que se usa particularmente para navegar con información que suele estar jerarquizada, abrir un editor, o visualizar las propiedades de un editor activo.

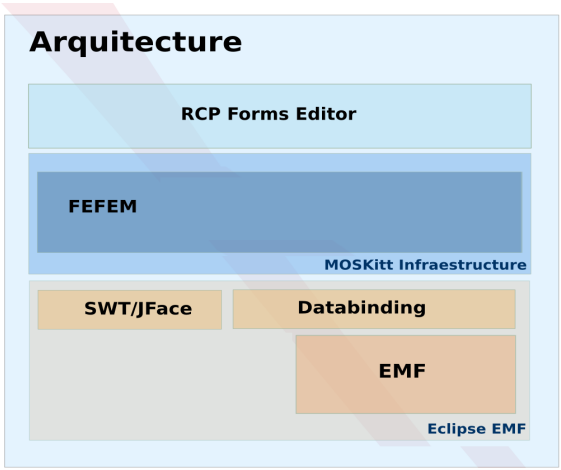
- **Eclipse Modeling Framework (EMF):** Es un framework de modelado y facilidad de generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurado.

1.5 Antecedentes

Anteriormente en MOSKitt para la creación de editores de formulario se utilizaba un modulo llamado MOSKitt-FEFEM, este sistema requiere una alta capacidad en conocimientos de programación, ya que requiere la continua edición de código. MOSKitt-FEFEM lo que nos proporcionaba es una serie de clases java que proporciona los distintos contenedores que tiene un editor(paginas, secciones, grupos, …) así como de los posibles composites. Un composite es una clase java que dota de funcionalidad y diseño a un elemento del editor(tablas, chekbox, textbox, Dialogs....).

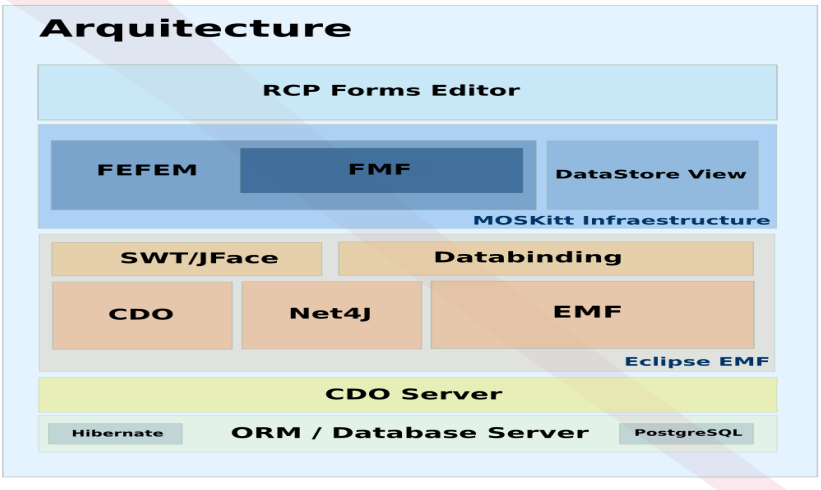
Una vez realizado el formulario, los recursos editados por este se almacenaban en ficheros XMI, fichero con una estructura similar a un XML. Esto impedía la edición del recurso por dos clientes de manera simultanea. Si otro cliente quería editar el recurso, debería pedírselo al cliente que lo creo o este haberlo compartido.

A continuación se muestra en detalle como era la arquitectura utilizada anteriormente en MOSKitt para la creación de editores de formulario.



Una arquitectura simple, donde el modulo FEFEM, dependiente de SWT/JFace y Databinding proporcionaban los recursos necesarios para creación de editores de formulario donde los recursos que editaban se guardaban en ficheros XMI.

Lo que se pretende con este proyecto es transformar la arquitectura anterior a esta nueva, en la creación de editores de formulario.



En esta arquitectura nueva que se pretende conseguir vemos la incorporación de FMF un modulo de MOSKitt realizado por una empresa externa al proyecto y que pretende facilitar la generación de código en la creación de los editores de formulario. También incorporamos en el mismo nivel una vista DataStore que permita la gestión de los recursos creados a partir de los editores de formulario. En la capa del medio incorporamos CDO y Net4j, con CDO nos permitirá la persistencia de los recursos. CDO contiene las clases y métodos necesarios para la comunicación con el servidorCDO, Net4j nos proporciona también parte de la comunicación con el servidor. La capa inferior, el servidorCDO nos permitirá la comunicación con la base de datos, sera el servidorCDO quien ejecute las ordenes de los clientes frente la base de datos.

1.6 Estructura de la memoria

La memoria contiene una introducción donde se explica el ámbito del proyecto, los objetivos que lo forman así como la planificación del proyecto. Se ha intentado ir completando la planificación a la vez que se realizaba las practicas.

En la sección 2 analizaremos las distintas tecnologías utilizadas en este proyecto, mencionando qué son y cómo las hemos utilizado, haciendo una breve descripción de su funcionamiento.

En la sección 3 se plantea la planificación llevada durante todo el proyecto describiendo cada una de sus fases y finalizando con un cronograma donde se pueden ver los periodos que engloban cada una de las tareas programadas en el proyecto.

En la sección 4 se encontrara la especificación de requisitos basándose en SW IEEE-830, se detallaran los requisitos de las interfaces externas, los requisitos específicos donde se dividirán en función del usuario y se detallaran los requisitos funcionales, se hablara también de las restricciones en el diseño y de los requisitos de rendimiento, por ultimo en esta sección se describirán los atributos del proyecto.

En la sección 5 y basándose en la sección anterior se describirá la parte del análisis y diseño del proyecto. En esta sección se incluirá el diagrama de casos de uso obtenido basándose en la especificación de requisitos describiendo dos de ellos que nos

acompañaran durante todo el ciclo del software. En esta sección se mostrara el diagrama de clases de nuestro proyecto explicando aquellas secciones mas importantes, diagramas de secuencia de las acciones que puede realizar nuestra vista y diagramas de actividad de la parte de sincronización de nuestro proyecto.

En la sección 6 se detalla la implementación de los dos casos de uso escogidos en la sección anterior, mostrando partes del código importantes para el correcto funcionamiento de nuestra vista DataStore. En esta sección también se explica como esta implementado la parte de la edición simultanea del mismo recurso entre distintos clientes.

En la sección 7 es la parte donde se encontraran la documentación aportada al proyecto, manuales de la vista MOSKittDataStore, como levantar un servidor CDO en nuestro ordenador, como preparar nuestro metamodelo para que sea soportado por CDO y como construir nuestro propio editor a través de FMF y este sea contribuido a la vista MOSKittDataStore.

En la sección 8 se explicaran las pruebas que se han hecho a la vista y a los editores creados, como se han implementado esas pruebas y como se han ejecutado.

En la sección 9 y antes de la conclusión se va hacer una reflexión sobre la continuidad de este proyecto y que perspectiva tiene el proyecto de cara al futuro.

Finalmente encontramos dos secciones que contendrán las conclusiones personales del proyecto y las referencias que han servido de gran utilidad para la elaboración de este proyecto.

2. Contexto Tecnológico

Este proyecto integra diferentes tecnologías, cada una de ellas utilizada para una labor específica. En este apartado las analizaremos cada una de ellas, algunas de ellas engloban otros proyectos internamente que también comentaremos en el caso de que hayan sido utilizados.

2.1 Java

Java es un lenguaje de programación orientado a objetos, desarrollado por la empresa Sun Microsystems (recientemente adquirida por otra empresa del sector, Oracle). Es multiplataforma y está desarrollado bajo la licencia GNU GPL (General Public License, Licencia Pública General).

Actualmente, este lenguaje de programación es uno de los más utilizados por su versatilidad, ya que Java se utiliza para crear todo tipo de aplicaciones, webs dinámicas, acceso a bases de datos, etc; por ser de código abierto y por ser independiente de la plataforma. Además, permite crear programas modulares y código reutilizable.

2.2 Eclipse Galileo

Eclipse es una plataforma de programación utilizada para crear entornos integrados de desarrollo (IDE, Integrated Development Environment). Este entorno de programación fue desarrollado originalmente por IBM (International Business Machines) como el sucesor de su familia de herramientas para VisualAge. Actualmente es desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta el código abierto. Eclipse Galileo es una de las versiones más recientes de este IDE (la más reciente recibe el nombre de Eclipse Helios),

2.2.1 EMF

Eclipse Modeling Framework (Framework de modelado Eclipse, EMF) es un framework de modelado y facilidad de generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurado. Desde una especificación del modelo descrita en XMI, EMF suministra herramientas y

soporte runtime para producir un conjunto de clases Java para el modelo, un conjunto de clases Adapter que permiten visualización y edición basándose en comandos del modelo, y un editor básico.

Los Modelos pueden ser especificados usando Anotación Java, documentos XML, o herramientas de modelado como Rational Rose, y después ser importados a EMF.

Lo más importante de todo, EMF suministra las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF. En resumen, EMF nos aporta un diseño previo, que una vez concretado, genera código completamente funcional y sin errores, basado en nuestro modelo.

2.2.2 CDO

CDO (Connected Data Objects) es un proyecto eclipse integrado en EMF, nos permite la persistencia de nuestro modelo, de manera que un mismo recurso puede ser consultado desde dos terminales al mismo tiempo.

CDO gestiona el acceso del recurso persistido mediante sesiones y transacciones. Esto es debido a que CDO implementa dos parte, una parte de cliente y una parte de servidor. La parte de cliente es aquella que gestiona, la creación de sesiones al servidor y la apertura de los recursos así como la actualización de los cambios en tiempo real y el gestionado de las transacciones sucias en ese momento.

El servidor se encarga de escuchar y proporcionar servicios de acceso a los datos a los diferentes clientes que se conectan a el a traves de sesiones y transaciones. El protocolo de comunicación por defecto entre cliente y servidor viene implementado por Net4j Signalling Platform.

2.2.3 Teneo

Teneo es la solución a la persistencia en base de datos de los modelo EMF usando Hibernate o EclipseLink. Teneo suporta la creación automática de los Relational Mappings de los modelos EMF. Los objetos de los modelos EMF son almacenados y recuperados mediante consultas avanzadas (HQL o EJB QL).

Teneo forma parte de del proyecto Eclipse Modeling Framework (EMF).

2.2.4 ATL

ATL (Atlas Transformation Language) es un lenguaje y una herramienta de transformación de modelos. En el campo del Model-Driven (MDE), ATL provee maneras de producir un conjunto de modelos de destino a partir de un conjunto de modelos de origen.

Desarrollado en la parte superior de la plataforma Eclipse, el entorno de desarrollo integrado ATL proporciona una serie de herramientas (resaltado de sintaxis, depurador, ...) que tienen como objetivo facilitar las transformaciones ATL.

2.3 MOSKitt

Modeling Software KIT (MOSKitt) es una herramienta CASE LIBRE, basada en Eclipse que está siendo desarrollada por la Conselleria de Infraestructura y Transporte (CIT) para dar soporte a la metodología gvMetrica (una adaptación de Metrica III a sus propias necesidades). gvMétrica utiliza técnicas basadas en el lenguaje de modelado UML.

Su arquitectura de plugins la convierte no sólo en una Herramienta CASE sino en toda una Plataforma de Modelado en Software Libre para la construcción de este tipo de herramientas.

MOSKitt se desarrolla en el marco del proyecto gvCASE, uno de los proyectos integrados en gvPontis, el proyecto global de la CIT para la migración de todo su entorno tecnológico a Software Libre.

2.3.1 FEFEM

FEFEM es un framework (un conjunto de clases Java relacionadas entre ellas con un objetivo común) que tiene como objetivo facilitar el desarrollo de editores Eclipse basados en formularios que manipulan modelos definidos mediante metamodelos

Ecore.

Habitualmente, los modelos se editan mediante editores gráficos donde los elementos se representan en diagramas que utilizan figuras relacionadas entre ellas (como los modelos UML, BPMN, etc.). Trabajando sobre la plataforma Eclipse se puede utilizar el Graphical Modeling Framework (GMF) para desarrollar este tipo de editores. En ocasiones estas metáforas gráficas no son las más adecuadas y resulta más sencillo o intuitivo editar el contenido de un modelo utilizando formularios (tablas, campos de texto, maestros-detalle, etc.).

Otro escenario habitual ocurre cuando se utilizan metamodelos Ecore para definir información estructurada ya que, aunque no representan en el sentido estricto un modelo (una representación abstracta y/o parcial de una entidad), la infraestructura proporcionada por el Eclipse Modeling Framework (EMF) para la gestión de modelos puede aplicarse de manera genérica para la gestión de datos estructurados.

En ambos casos pueden identificarse una serie de patrones cuya resolución utilizando Eclipse y su infraestructura para la implementación de editores basados en formularios es similar (por ejemplo, editar una propiedad de texto de un elemento del modelos, visualizar y editar colecciones de elementos, etc.). Por ello, en el contexto del proyecto MOSKitt, se ha desarrollado el framework FEFEM, de manera que la construcción de un editor pueda realizarse como una composición de los distintos patrones.

2.3.2 FMF

FMF (Forms Modeling Framework) que permite diseñar por medio de un DSL (Domain Specific Language) los editores basados en formularios deseados y generar de forma automática sobre FEFEM el editor correspondiente. FMF es una herramienta de OpenCanarias S.L que la ha donado a la CIT para el proyecto MOSKitt.

3. Planificación

En este apartado se va a presentar la planificación llevada durante todo el proyecto, la planificación ha sido divida por una serie de tareas que han sido planificadas previamente al proyecto. Finalmente en este apartado contendrá un cronograma gráfico de las tareas mostradas.

3.1 Tareas

El proyecto ha tenido siete meses de duración, comprendiendo varias fases o tareas:

1. **Aprendizaje**

En esta fase se dedicara un buen porcentaje de tiempo, ya que hay que adaptarse al modo de trabajo, conocer la herramienta que se utiliza, aprender las tecnologías empleadas y “trastear” un poco con ellas.

Incluye el conocimiento del método que ya existía sobre la creación de formularios y la creación de plugins dentro de eclipse. También incluye en la tarea el uso de la herramienta subversión dentro de Eclipse para el tratamiento de la versiones y la bajada de recursos del repositorio de la herramienta.

Tiempo total: 1 mes.

2. **Adaptación**

Después de realizar la parte de aprendizaje me incorporarán en el proceso de un nuevo proyecto ya comenzado, en el que yo me encargare de los diseños de los posibles editores que surjan del proyecto TechEnv dentro de la herramienta MOSKitt, el proyecto trata de la creación de un conjunto de editores capaz de mantener la información de los entornos tecnológicos usados dentro de un proyecto cualesquiera.

La tarea a desempeñar consistirá en el diseño de un nuevo editor de formulario que servirá como catalogo de todos los posibles entornos tecnológicos que se usan dentro de la Conselleria de Infraestructuras y Transporte (CIT) a partir de la herramienta MOSKitt-FMF.

Tiempo total: 1 mes.

3. **Investigación**

Finalizado el primer editor, se someterá a pruebas en CIT, siendo yo el responsable de resolver todos los bugs que surjan.

La tarea consiste en la investigación de herramientas EMF que permiten la persistencia de modelos en base de datos. Lo que se pretende es poder persistir los datos creados a partir de los editores del proyecto TechEnv.

Las herramientas a investigar serán: Teneo y CDO.

Tiempo total: 1 mes.

4. **Construcción de la vista DataStore y adaptación de los editores.**

La tarea consiste en la creación de una vista en MOSKitt que de soporte a la persistencia con una de las herramientas anteriormente investigadas. La nueva vista tendrá la capacidad de gestionar las distintas conexiones a los recursos.

Esta vista soportara por debajo todo el peso de CDO (Creación de sesiones, transacciones y soporte de invalidaciones remotas del modelo actual). En esta parte se incluye la adaptación de los posibles editores, para que estos soporten la persistencia que nos proporciona CDO.

Tiempo total: 3 meses.

5. **Diseño de pruebas de los editores y ejecución de las mismas.**

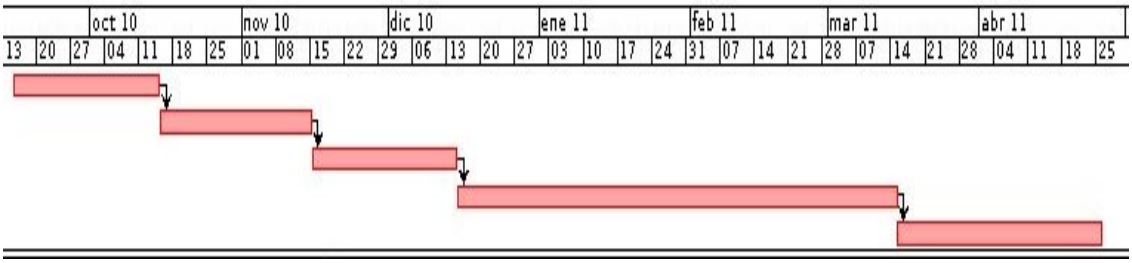
En esta parte del proyecto se generaran a partir de los modelos generados de los editores, modelos que contengan casos de pruebas, esto se hará a través de ATL que nos permitirá la transformación de un modelo a otro. Una vez generados los caso de test se deberán ejecutar.

Tiempo total: 1 mes.

3.2 Esquema cronológico

En esta parte se presenta el esquema cronológico de las tareas en el proyecto, como se puede apreciar en el diagrama las tareas van encadenadas y por lo tanto no se puede empezar una tarea si las tareas anteriores no han terminado.

	Nombre	Duración	Inicio	Terminado
1	Aprendizaje	22,125 d...	15/09/10 8:00	15/10/10 9:00
2	Adaptación	21 days	15/10/10 9:00	15/11/10 9:00
3	Investigación	22 days	15/11/10 9:00	15/12/10 9:00
4	Construcción de la vista DataStore y adaptación de los editores.	64 days	15/12/10 9:00	15/03/11 9:00
5	Diseño de pruebas de los editores y ejecución de las mismas.	30 days	15/03/11 9:00	26/04/11 9:00



4. Especificación de Requisitos SW IEEE-830

El análisis y desarrollo de requerimientos tiene como producto final: un acuerdo documentado entre el cliente y el grupo de desarrollo acerca del producto a ser construido. Estos documentos tienen por finalidad reunir los requisitos completos del cliente tal de desarrollar un software de acuerdo a las exigencias del mismo.

En este apartado lo que se va a exponer es ese acuerdo documentado siguiendo el Standard IEEE380.

4.1 Requisitos de interfaces externas

El plugin MOSKittDataStore, sera un plugin eclipse de la versión 3.5 es decir es un plugin para la versión GALILEO del eclipse, este requisito es debido fundamentalmente a que se pretende integrar este plugin en MOSKitt que actualmente trabaja con esta versión eclipse.

MOSKittDataStore estará compilado en la versión de Java 1.5 esto es debido a que MOSKitt esta compilado de esta manera y poder asegurar que el plugin funcionara correctamente para maquinas virtuales java 1.5 o superiores.

MOSKittDataStore tendrá dependencias sobre FEFEM y CDO, por lo que debe de estar instalado en eclipse. FEFEM se puede conseguir del repositorio del proyecto MOSKitt así como EMFT que también requeriremos.

Para que FEFEM funcione correctamente depende de EMF que tendrá que estar instalado en eclipse Para la persistencia del recurso usaremos CDO su versión 2.0 que es la que viene instalada en MOSKitt.

Para la creación de editores propios usaremos FMF también incluido en el repositorio de MOSKitt.

El servidor CDO que gestionara los recursos CDO persistidos, necesita de una base de datos donde almacenar los recursos, se propone una base de datos Postgres. Para el gestionado de la base de datos se utilizara la aplicación pgAdmin3.

Por ultimo para las transformaciones de un modelo FMF a un modelo de Casos de test requeriremos de la instalación de ATL en nuestro eclipse.

Resumen interfaces externas:

- Eclipse 3.5 Galileo
  - Java 1.5
  - Subversión
  - EMF 2.5
    - EMFT
      - FEFEM
        - FMF
  - CDO 2.0
    - Postgres
      - pgAdmin 3
  - ATL

4.2 Requisitos específicos

Dentro de los requisitos específicos podemos dividirlos según el usuario, en este caso tenemos tres tipos de usuarios:Cliente, Programador y Administrador del servidorCDO

4.2.1 Cliente

Es el que usa directamente la vista MOSkittDataStore, para acceder a los recursos y así poderlos editar. Sus requisitos especificos son:

4.2.1.1 Registro de conexiones a base de datos

Introducción: El usuario debe de ser capaz de registrar conexiones a distintas fuentes de datos, a estas fuentes de datos las llamaremos repositorios que es donde se encuentra los distintos recursos almacenados.

Entradas:

1. **Repositorio ID**: Es el identificador universal del repositorio dentro del

servidor CDO.

2. **CDO Repositorio:** Nombre del repositorio.
3. **Tipo de conexión:** Protocolo de transporte que se utilizara en la conexión con el servidor CDO.
4. **Autentificado:** Indica si el servidor CDO requiere una autenticación de los clientes que quieren establecer la conexión.
5. **Usuario:** Nombre de usuario, este dato es opcional y depende de si se necesita autentificado o no el servidor .
6. **Contraseña:** Contraseña del usuario , este dato es opcional y depende de si se necesita autentificado o no el servidor
7. **Servidor CDO:** Dirección IP y puerto del servidor CDO donde se encuentra el repositorio.

Proceso: Con los datos anteriores se crea una conexión a una base de datos que sera utilizada para acceder a los recursos almacenados en ella.
Salida: Devuelve la conexión creada.

4.2.1.2 Quitar las conexiones a base de datos

Descripción: El usuario debe de ser capaz de eliminar las distintas conexiones a repositorios registrados, la eliminación de la conexión no implicara la eliminación de los recursos en sus interior.

Entrada: Una conexión creada anteriormente.

Proceso: Se eliminan los datos de la conexión y se realiza un refresco para volver a mostrar aquellas conexiones que no han sido eliminadas.

Salida: No devuelve ninguna salida. Solo se muestran de nuevo aquellas conexiones que han sido añadidas pe3ro no aquellas que han sido eliminadas.

4.2.1.3 Editar las conexiones a base de datos

Descripción: El usuario debe de ser capaz de editar los distintos atributos que contenga la conexión al repositorio, es decir sera capaz de editar una conexión sin tener que eliminarla.

Entrada: Una conexión creada anteriormente.

Proceso: Se mostraran las propiedades de la conexión, donde se podrán editar cualquiera de sus campos.

Salida: No devuelve ninguna salida. Mostrara la conexión editada junto a las demás.

4.2.1.4 Inicialización de un repositorio de datos

Descripción: El usuario debe de ser capaz de inicializar una conexión a un repositorio, al inicializar una conexión se mostraran los distintos recursos que contenga el repositorio.

Entrada: Una conexión creada anteriormente.

Proceso: Se selecciona la conexión de entrada y bien con el botón derecho y del menú abrir o haciendo doble-clic sobre ella se conectara al repositorio y se recogerá aquellos recursos que sean compatibles en ese momento se mostraran

Salida: Muestra los recursos almacenados en el repositorio que acaba de ser inicializado.

4.2.1.5 Añadir un recurso a un repositorio de datos

Descripción: El usuario sera capaz de añadir un nuevo recurso a una conexión

Entrada: Un nombre y seleccionara el metamodelo al que pertenece.

Proceso: La vista le pedirá al usuario la entrada, procesando esta para la creación de un nuevo recurso en el repositorio seleccionado.

Salida: Nos abrirá el editor de formulario del recurso nuevo.

4.2.1.6 Abrir un recurso a un repositorio de datos

Descripción: El usuario deberá de ser capaz de abrir un recurso con el editor adecuado.

Entrada: Un recurso de la lista de recursos que se muestra el la inicialización de la conexión.

Proceso: Al abrir un recurso comprobara el tipo de recurso que es para poder abrirlo con el editor adecuado.

Salida: Nos abrirá el editor de formulario del recurso.

4.2.1.7 Eliminar un recurso a un repositorio de datos

Descripción: El usuario deberá ser capaz de eliminar un recurso dentro de un

repositorio.

Entrada: Recurso a eliminar

Proceso: Se selecciona un recurso, se le da a eliminar la vista preguntara si realmente quieres eliminar el recurso, si es así, lo elimina no si antes comprobar si esta abierto para cerrarlo.

Salida: Ninguna, el recurso deja de mostrarse en la lista.

4.2.2 El Programador

El programador de los editores de formulario es aquella persona que construye un modelo y su editor(opcional) y quiere que los objetos de su modelo sean persistidos en una base de datos. Para ello tiene su requisito especifico.

4.2.2.1 Adaptación de los posibles editores a la vista

Descripción: El programador de un editor debe de ser capaz de registrar su editor y metamodelo al que pertenece el editor, para que la vista MOSKittDataStore lo tenga en cuenta a la hora de que un añadir un nuevo recurso.

Entrada: El programador deberá de proporcionar a la vista:

- MetamodelURI: URI del metamodelo a persistir.
- MetamodelName: Nombre con el que se identificara el metamodelo.
- InitialObjectName: Nombre del elemento raíz de nuestro modelo
- EditorId(opcional): ID del editor con el que abrirá el modelo, en el caso de no proporcionárselo abrirá con el editor por defecto de CDO.

Proceso: Este proceso se hará a través de un punto de extensión donde los programadores de los metamodelos que quieran ser persistidos contribuirán con sus datos.

Salida: Ninguna, solo que al intentar abrir un recurso, lee las distintas contribuciones para abrirlo con el editor adecuado.

4.2.3 El administrador del ServidorCDO

Sera el encargado de la gestión del servidorCDO que puede estar instalado en la

misma maquina o en otra distinta que la del cliente. Para ello tiene los siguientes requisitos específicos:

4.2.3.1 Inicializar el servidorCDO

Descripción: El administrador del servidor CDO debe de ser capaz de inicializar el servidor. Todas la gestión de los recursos pasan de la vista MOSKittDataStore al servidorCDO quien es el que se comunica con la base de datos directamente.

Entrada: Ninguno, se necesita que el servidor este instalado sobre una maquina.

Proceso: Con el terminal te sitúas el la carpeta /bin del servidor y lo ejecutas invocándolo.

Salida: El servidor se inicializa.

4.2.3.2 Apagado del servidorCDO

Descripción: El administrador del servidor CDO debe de ser capaz de cerrar el servidor.

Entrada: Ninguno, se necesita que el servidor este instalado sobre una maquina.

Proceso: Con servidor inicializado se ejecuta la orden de cerrado del servidor

Salida: El servidor se apaga.

4.2.3.3 Configuración del servidorCDO

Descripción: El administrador del servidor también tendrá acceso a la base de datos, sera capaz de gestionar la eliminación de repositorios, de la creación de nuevos repositorios y la edición de los mismos, así como de gestionar donde se van a guardar los recursos de los distintos repositorios.

Entrada: Fichero de configuración del servidorCDO

Proceso: Antes de inicializar el servidorCDO, este leerá la información de configuración del fichero de configuración.

Salida: Ninguna.

4.3 Requisitos de rendimiento

El sistema debe de soportar la gestión de múltiples (mínimo 2) clientes conectados a la vez para acceder al mismo recurso, pero solo un cliente a la vez puede modificar el contenido de uno de los elementos del recurso, en caso de colisión, dos o mas clientes modifiquen el mismo elemento dentro de un recurso, se le dará prioridad al primer cliente que haga su commit en la base de datos, siendo los demás anulados a través de un rollback, para no perder la transacción. En ningún momento los clientes deben de perder la transacción mientras el recurso lo tengan abierto. Como debe de gestionar como mínimo dos clientes conectados a la vez, debe de soportar dos transacciones desde distintos puntos conectados al mismo recurso, pero un cliente deberá de tener una transacción por cada recurso que tenga abierto.

La base de datos, debe de ser lo suficientemente potente para almacenar miles de registros en un mismo repositorio, cada repositorio puede contener múltiples recursos que pertenezcan a diferentes editores de formulario.

4.4 Restricciones de diseño

Dentro de los requisitos de diseño tenemos los siguientes:

- La aplicación a generar sera una nueva vista en MOSKitt, es decir una nueva vista en eclipse 3.5.
- Una conexión a un repositorio, es identificado por UUID universal del repositorio, nombre del repositorio.
- Las conexiones pueden requerir autenticación (usuario y password).
- Cada editor abierto por un cliente podrá o no recibir notificaciones de manera que si otro cliente esta conectado al mismo recurso que tu y hace modificaciones en el recurso, tu puedas ver de manera simultanea esas modificaciones.
- La vista tiene que controlar las sesiones con el servidor y las posibles transacciones que tenga abierta, de manera que al cerrar un recurso se cierre la transacción y la sesión si no hay otro recurso del mismo repositorio abierto en ese momento.
- Las distintas conexiones a repositorios se deberán configurar desde una

pagina en el menú preferences de eclipse (window->preferences).

- La vista tendrá dos botones: refrescar, activar notificaciones para el editor actualmente abierto.
- El botón de activar notificaciones nos dice también si el editor en el que estamos tiene o no activadas las notificaciones.
- La vista debe de soportar la apertura del recurso con el editor propuesto con el programador del recurso, para ello la vista creara un punto de extensión, para que los programadores puedan conectar sus recursos con la vista, el punto de extensión como mínimo contendrá la información mas relevante: nombre del metamodelo, URI del metamodelo y identificador del editor.
- En caso de que el identificador del editor este vacío, el recurso se abrirá con el editor que incorpora CDO para abrir los recursos.
- La vista, los cuadros de dialogo, los menús de la vista, … tendrá que estar internacionalizado.
- La aplicación de manera global tiene que tener un alto poder de usabilidad, es decir debe de ser fácil de usar y de manipular.
- El código de la aplicación debe de ser lo mas legible posible, ya que sera una aplicación de código abierto de la Conselleria de Infraestructura y Transporte y otros usuarios podrán aportar cosas.

4.5 Atributos

Fiabilidad

El sistema debe de tener una alta fiabilidad, recordamos que el objetivo del PFC es la creación de un nuevo modulo para MOSKitt que es un proyecto de la Conselleria de Infraestructura y Transporte, por lo que el sistema lo van a usar un numero alto de personas, y por lo tanto el sistema debe de ser lo mas fiable posible. El sistema debe de detectar las posibles excepciones o fallos y resolverlos, el cliente no puede quedarse bloqueado bajo ninguna circunstancia. Para hacer un sistema lo mas fiable posible y tratándose de que este modulo sera una primera versión, debe de hacerse un numero considerable de pruebas, así como habilitar de una plataforma para que los usuarios puedan exponer sus problemas de manera que puedan resolverse lo antes posible, para ello se usara la misma plataforma que usa el proyecto MOSKitt.

Mantenibilidad

El código debe de ser mantenido por una gran numero de personas, por lo que el código debe de estar comentado en todo momento, también se requiere que el código este bien estructurado, la política de la CIT es: “Escribe código como si el que fuera a mantenerlo fuera un loco psicópata que conoce donde vives”

Portabilidad

El proyecto pertenece a un nuevo modulo de MOSKitt. MOSKitt, a fecha de hoy esta distribuido para la versión Ubuntu 10.04, aunque también esta construido para versiones Windows en modo alpha ya que no han sido lo suficientemente probadas, por lo que nuestro modulo debe de soportar ambas.

Seguridad

Es importante que se gestione el acceso a los recursos de la base de datos, por lo tanto, no todo el mundo debe de tener acceso a los recursos, solo las personas autorizadas podrán acceder a los recursos, en esta primera versión y debido a las limitaciones del soporte que tiene CDO 2.0 bastara con poder decir que usuarios tienen acceso a los recursos, impidiendo la conexión aquellos que no tengan acceso. El acceso permite la visualización, edición de cualquier recurso en ese repositorio.

5. Análisis y Diseño

Esta parte del proyecto se abordara el diseño estructural, se presentara por tanto el modelo esencial. El modelo esencial es una serie de diagramas que modelaran el modulo del sistema que se va a realizar.

El modelo esencial modela lo que nuestro modulo debe de hacer para satisfacer los requerimientos del usuario, en ningún caso en esta parte se especificara como se va a implementar, tampoco como se van a llevar a cabo las funciones, ni dónde ni qué o quién las hace.

Para la definición de los diagramas se ha empleado la propia herramienta MOSKitt ya que incorpora mecanismos para la creación de distintos diagramas UML, a parte gracias a la utilización de la propia herramienta se han resuelto una serie de bugs internos que fueron subsanados en la versión MOSKitt 1.3.x

5.1 Casos de Uso

Los diagramas de casos de uso documentan el comportamiento de un sistema desde el punto de vista del usuario. Por lo tanto los casos de uso determinan los requisitos funcionales del sistema, es decir, representan las funciones que un sistema puede ejecutar.

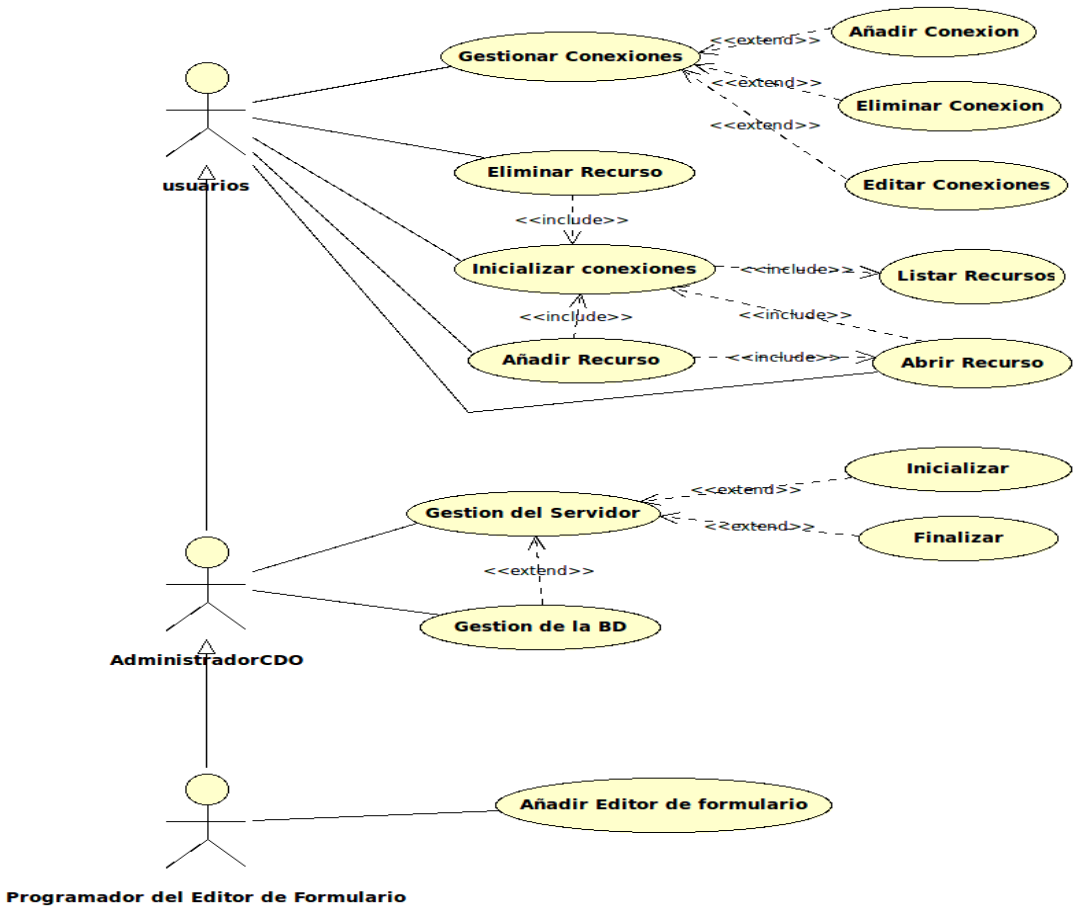
Su ventaja principal es la facilidad para interpretarlos, lo que hace que sean especialmente útiles en la comunicación con el cliente.

Los principales elementos que intervienen en un diagrama de casos de uso son los actores, los casos de uso, las asociaciones , las generalizaciones y los limites del sistema.

Los actores son los diferentes roles desde donde se accederá al sistema, normalmente suelen representar a las personas humanas pero también pueden representar colectivos de estas o servidores, en definitiva son cualquier cosa externa que interactúa con el sistema. Los casos de uso son las tareas que desarrollara el sistema, las asociaciones son las relaciones entre actores-casos de uso y casos de

uso-casos de uso. Cuando se producen relaciones entre dos casos de uso pueden ser de dos tipos <includes> o <extends>.
Los includes es cuando una tarea incluye además a otra tarea, es decir se van a realizar a la vez, los extends quieren decir que la tarea puede o no realizarse conjuntamente. Las generalizaciones son relaciones entre actores, pretenden expresar herencias de estos. Los limites del sistema muestran las diferentes partes que engloban un sistema. El siguiente diagrama mostrara los diferentes casos de usos del modulo a desarrollar.

A continuación se muestra el diagrama de casos de usos de nuestro proyecto, para poder conseguir este diagrama se han atendido a la organización de los distintos requisitos específicos de los usuarios en el sistema.



Es por eso por lo que se aprecia los tres tipos de usuario: Cliente(usuarios), AdmistradorCDO y Programador.

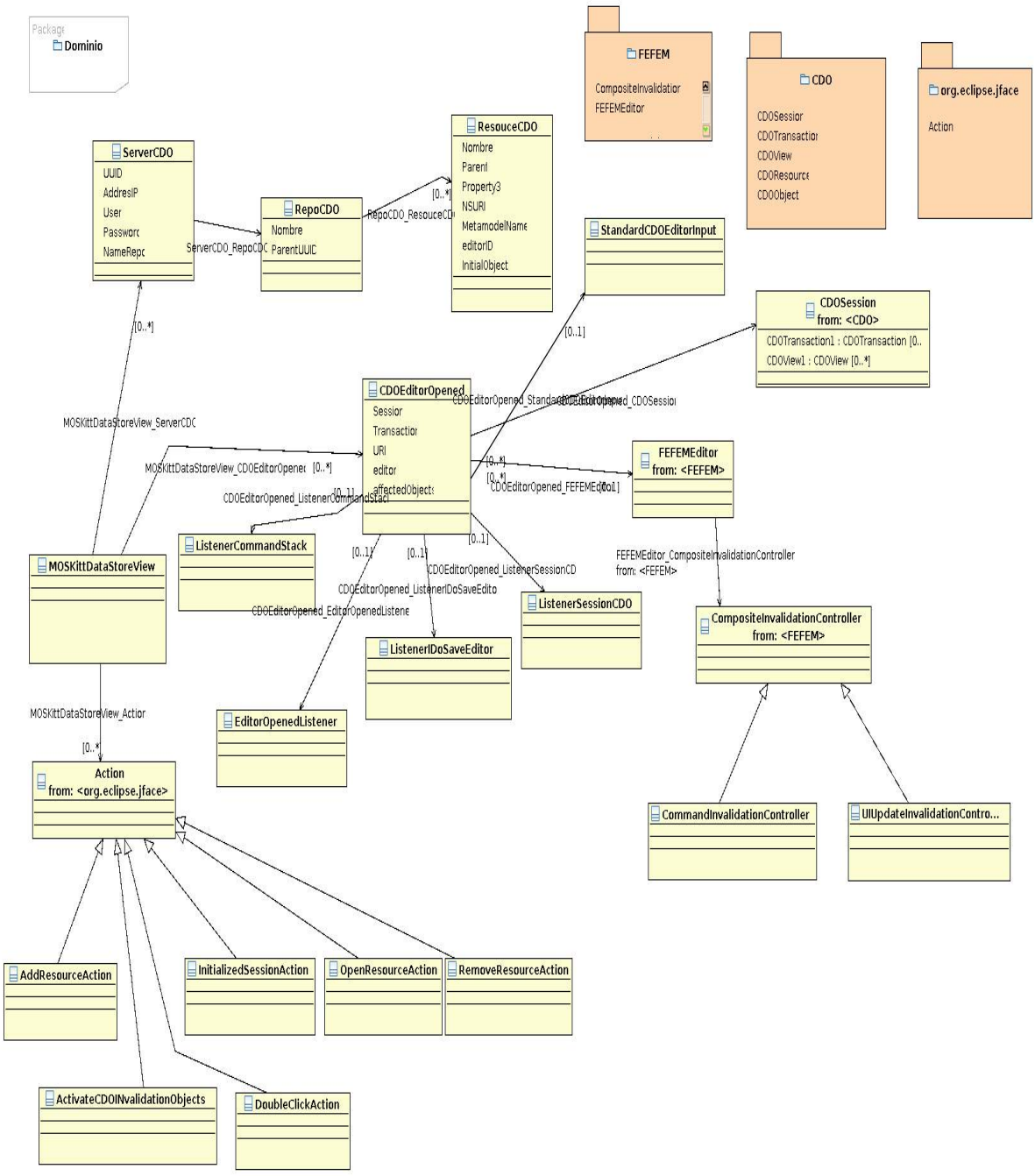
Cada caso de uso hace referencia a las distintas funciones que asume cada role. A continuación se van a abordar en detalle dos casos de uso que nos acompañaran durante todo el proceso de analisis, diseño y implementacion.

Nombre:	Añadir Recurso CDO
Autor:	Carlos Sanchez Rodriguez
Fecha:	01/11/10
Descripción: Se añade un nuevo recurso al repositorio para que sus objetos sean persistidos en una base de datos.	
Actores: Cliente, Programador, Administrador. (Programador y Adminitrador son especializacion de Cliente)	
Precondiciones: Antes de poder empezar el proceso, la conexión debe de estar inicializada previamente.	
Flujo Normal: 1. El usuario inicializa la conexión donde quiere insertar el nuevo recurso. 2. El usuario abre el menú contextual de la conexión inicializada. 3. El usuario hace clic sobre añadir nuevo recurso. 4. El usuario inserta un nombre del recurso y el tipo de metamodelo que pertenece el recurso que va a ser añadido. 5. El sistema valida el nombre y crea un nuevo recurso con el nombre y metamodelo elegidos anteriormente por el usuario. 6. El sistema abre el nuevo recurso creado con el editor correspondiente.	
Flujo Alternativo: 5. El sistema valida el nombre y el nombre del recurso ya existe. 6. El sistema abre el recurso.	
Poscondiciones: Una vez abierto el recurso, queda registrada la apertura del editor en el workbench.	

Nombre:	Abrir Recurso CDO
Autor:	Carlos Sánchez Rodríguez
Fecha:	01/11/10
Descripción: Se abre un recurso que esta persistido en la base de datos con su editor correspondiente.	
Actores: Cliente, Programador, Administrador.(Programador y Administrador son especialización de Cliente)	
Precondiciones: Antes de poder empezar el proceso, la conexión debe de estar inicializada previamente.	
Flujo Normal: 1. El usuario inicializa la conexión donde quiere insertar el nuevo recurso. 2. El usuario hace doble clic sobre el recurso que se quiere abrir. 3. El sistema se trae el recurso. 4. El sistema analiza el recurso y obtiene el editor con el cual abrir el recurso. 5. El sistema lanza la orden de abrir el recurso.	
Flujo Alternativo: 4. El sistema analiza el recurso y no obtiene el editor con el cual abrir el recurso. 6. El sistema abre el recurso con el editor con el editor por defecto de CDO.	
Poscondiciones: Una vez abierto el recurso, queda registrada la apertura del editor en el workbench.	

5.3 Diagramas de clases

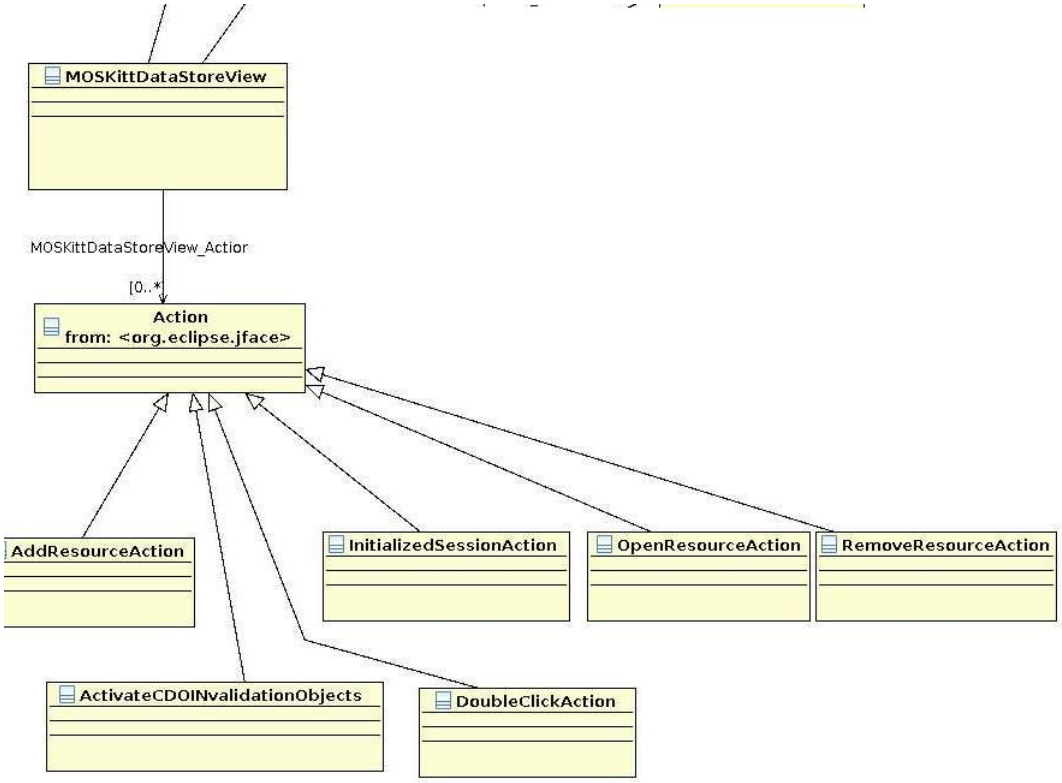
Partiendo del diagrama de casos de uso anterior y la especificación de requisitos del capitulo anterior, podemos definir el diagrama de clases de nuestro modulo:



El diagrama anterior muestra el complejo diagrama de clases de nuestro modulo, a continuación lo que se va a explicar las diferentes partes en las que se puede dividir el diagrama de clase anterior.

5.3.1 La vista

En este apartado se va a centrar en la explicación de la parte de la vista del diagrama de clase anterior. La parte de la vista en el diagrama de clases anterior seria :



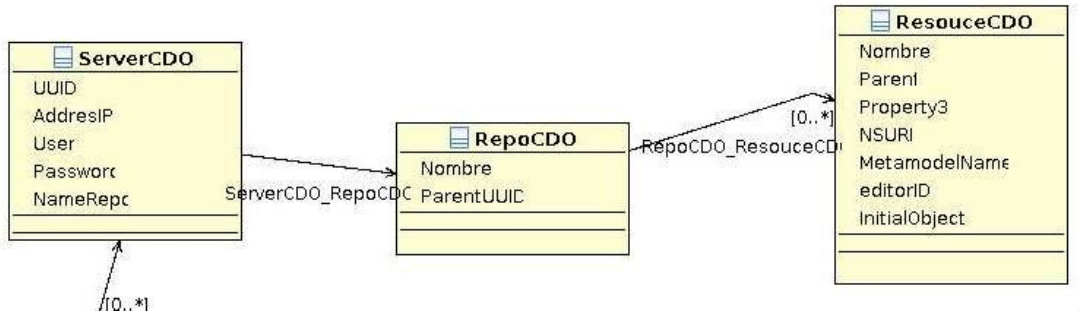
La clase MOSKittDataStoreView, esta clase representa la nueva vista que crearemos en MOSKitt para la gestión de los recursos CDO persistidos en la base de datos. La vista incorporara una serie de acciones a realizar por el usuario, todas la acciones de la vista heredaran de Action de <org.eclipse.jface>. A continuación se muestra una tabla donde representa la acción hacer por que clase viene representada. Esta marcada la clase **AddResourceAction** y **OpenResourceAction** porque son las clases que representan los casos de uso anteriormente descritos en detalle y que veremos su implementacion también en detalle.

Clase	***Acción***
AddResourceAction	**Añadir recurso**
InitializedSessionAction	Inicializar la sesión
OpenResourceAction	**Abrir un recurso**
RemoveResourceAction	Eliminar un recurso

ActivateCDOInvalidationObject	Activar el mecanismo de sincronización
DoubleClickAction	Acción de dobleclick

5.3.2 Gestión de Elementos

A continuación se muestra las clases donde se almacenara la información de las diferentes sesiones, repositorios y recursos que tengamos almacenados, una vez consultados se almacenara la información en las siguientes clases para no tener que repetir sucesivas consultas al servidorCDO para la obtención de los recursos.

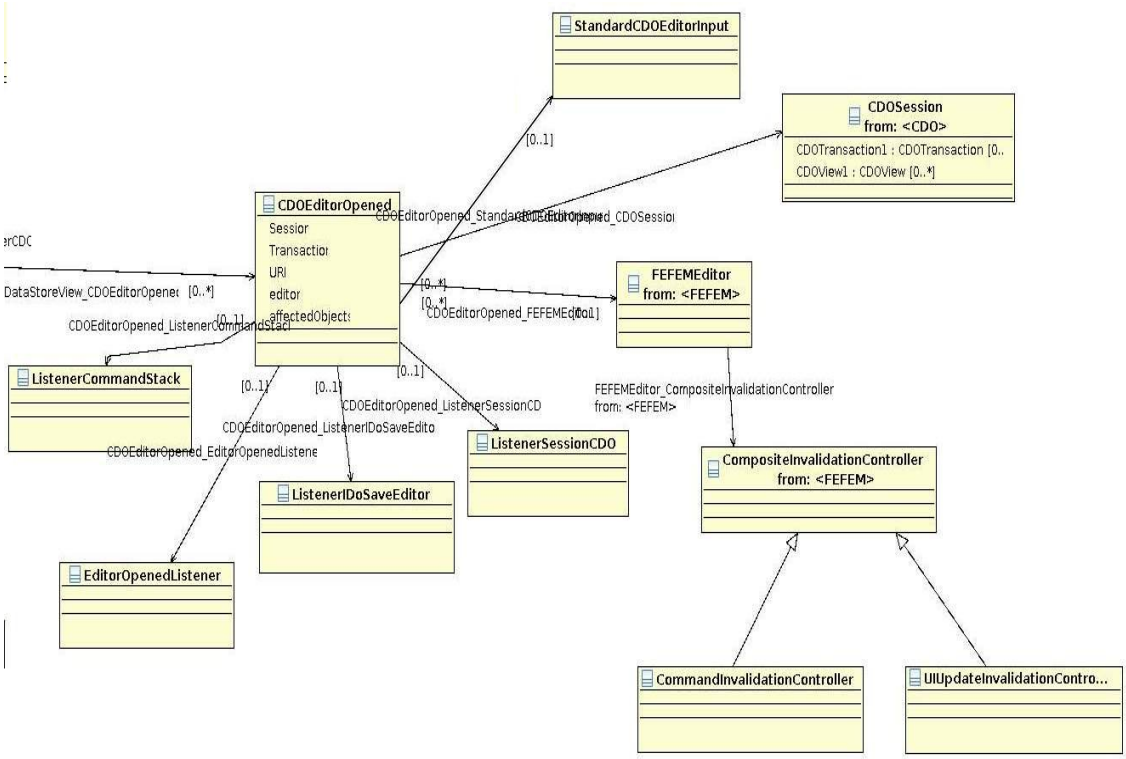


La vista MOSKittDataStore podrá tener 0 o mas conexiones a repositorios. Un repositorio es como un almacén de recursos. Una conexión con el servidor puede tener uno o mas repositorios. Nosotros las conexiones se harán sobre esos repositorios, pero debemos de almacenar la información sobre la sesión a la que pertenecen.

Si observamos un servidor tendrá un UUID, un identificador único y universal que identificara el servidor CDO. Los repositorios dentro de la sesión se identificaran por su nombre y un recurso tendrá la siguiente información importante: Nombre, URI del metamodelo al que pertenece, Nombre del metamodelo al que pertenece y Id. editor con el cual se abre el recurso.

En el momento de obtener los recursos (Inicializar la sesión) se calcularan los distintos atributos que quedaran almacenados.

5.3.3 Recurso abierto



Cada recursos que se abra, se abrirá con un editor de formulario especificado previamente por el programador del editor, ese recurso abierto vendrá representado por la clase CDOEditorOpened, esa clase viene instanciada con una serie de listeners:

ListenerCommandStack: Escuchara la pila de comandos almacenado los identificadores de los objetos alterados por el propio usuario.

ListenerIDoSaveEditor: Escuchara cuando el editor es guardado, lo que hará sera vaciar esa lista de identificadores de objetos modificados.

ListenerSessionCDO: Escucha los posibles cambios en el recurso hechos remotamente.

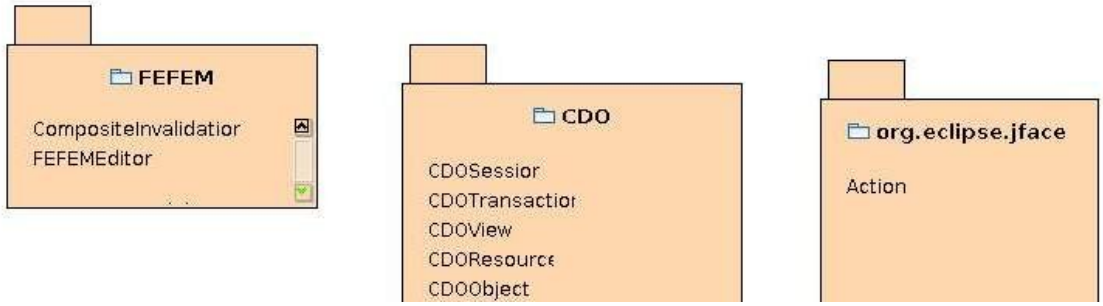
EditorOpenedListener: Se encarga principalmente de escuchar los cambio del editor sobre el Workbench, en especial el cerrado del editor para eliminar la instancia de la clase eliminado los listener enganchados a la clase CDOEditorOpened .

Los editores de formularios están basados en FEFEM, aunque estarán hechos

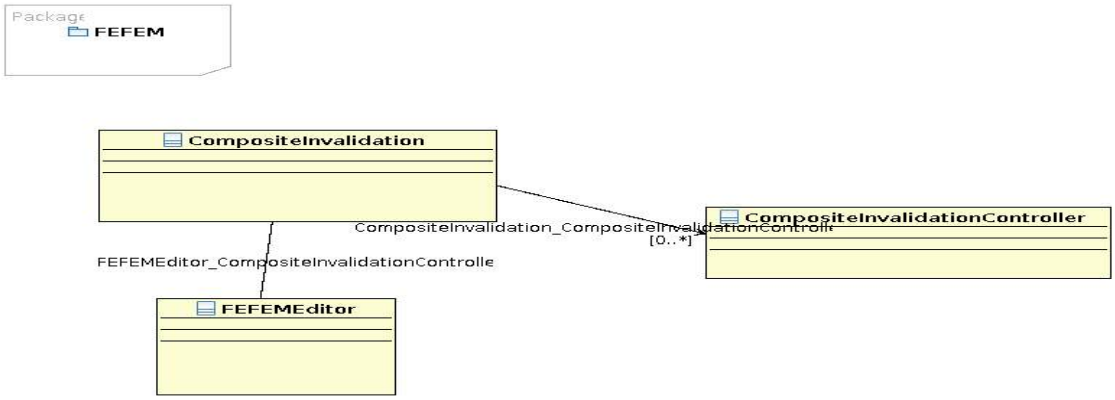
utilizando la técnica de FMF. Los editores de formulario basados en FEFEM extienden de la clase FEFEMEditor, esta clase contendrá mecanismos para saber exactamente que elementos están instanciados actualmente en el editor.
Las clases CommandInvalidationControler y UIUpdateInvalidationControler serán las clases de tipo manejador que serán ejecutadas al recibir un evento de alteración del recurso de forma remota. La primera se encargara de la pila de comandos y la segunda de actualizar la interfaz.

5.3.4 Paquetes Externos

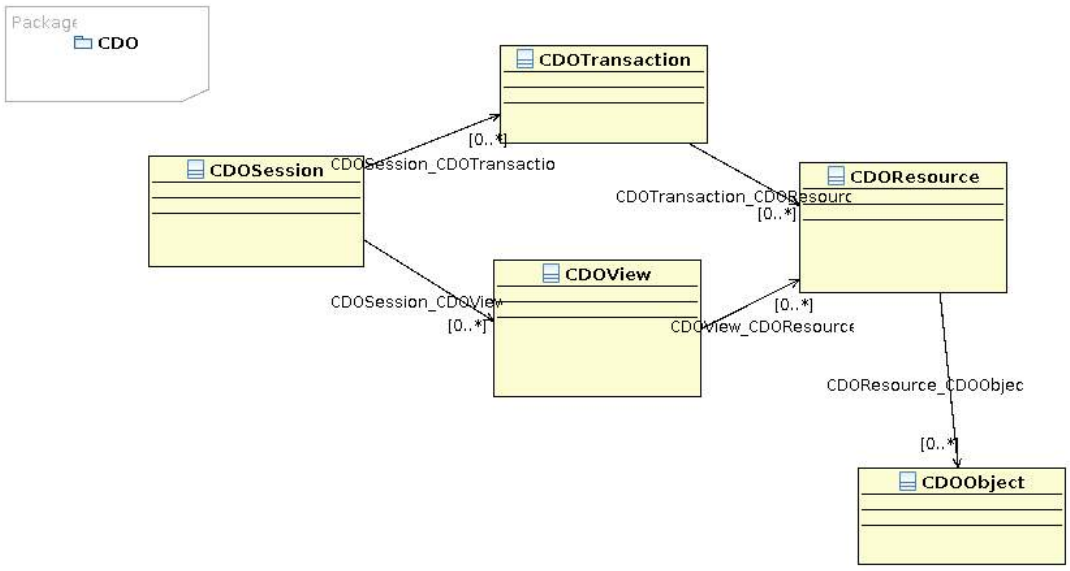
En esta parte se describirá brevemente los paquetes externos que aparecen en el diagrama de clases.



Como hemos comentado anteriormente, los editores de formulario basados en FEFEM extienden de la clase FEFEMEditor, un editor de formulario esta compuesto de composites, estos composites son componentes que representan elementos en un formulario (campos de texto, tablas …) FEFEMEditor contendrá la información de que composites tiene instanciados en cada momento.
A continuación se muestra el diagrama de clases resumido de FEFEM que hacen uso en nuestro modulo.



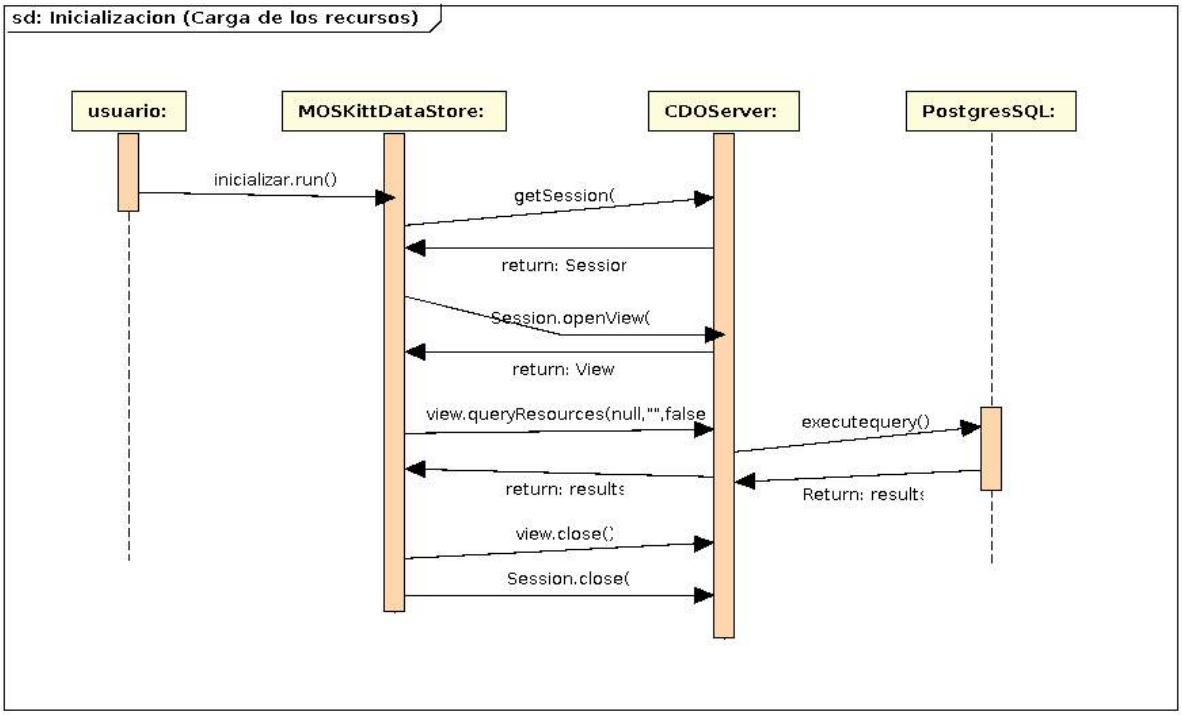
Cada conexión con un repositorio vendrá representada por la clase CDOSession, un repositorio contiene distintos recursos, el acceso al recurso es controlado por medio de transacciones representada por la clase CDOTransaction, los recurso de CDO vienen representados por la clase CDOResource un recurso contendrá objetos CDO representados por su clase CDOObjects. El diagrama de clases siguiente muestra de manera breve como esta compuesto el diagrama de clases de CDO.



5.4 Diagramas de Secuencia

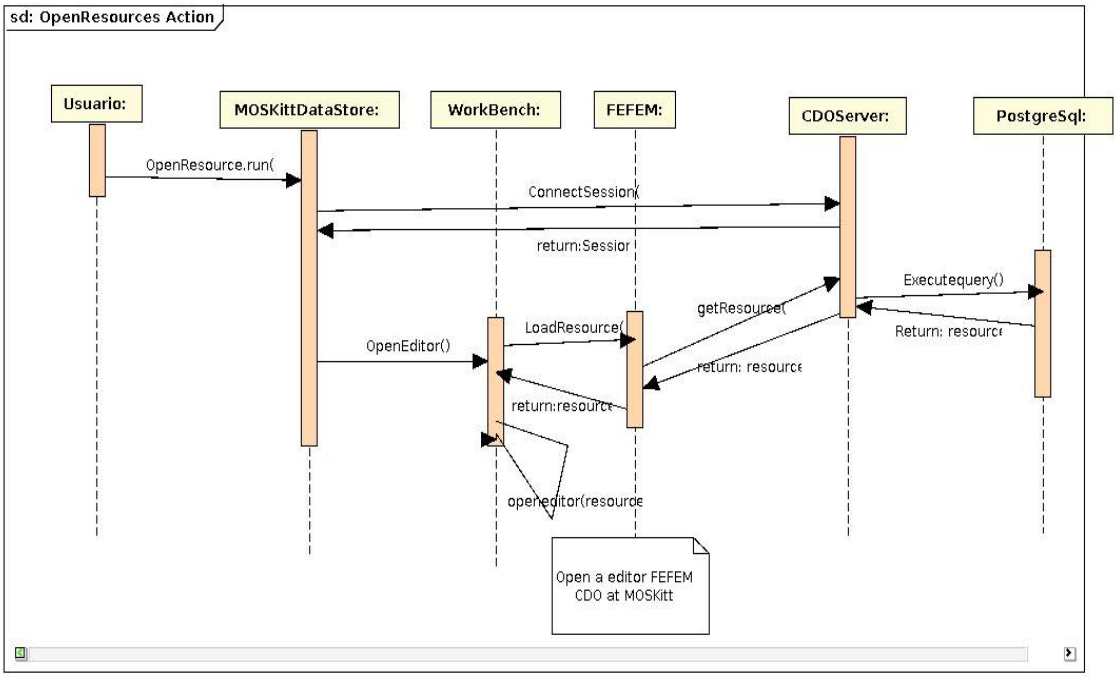
A continuación se va a detallar los diagramas de secuencia de las acciones que puede realizar el usuario al sistema.

1. **Diagrama de Secuencia del Proceso de inicialización.**



El usuario realiza la acción inicializar sobre la clase MOSkittDataStore, lo primero que hace la clase sera pedir la sesión al servidor CDO, una vez obtenida la sesión, abre una vista de la sesión, es decir no va a poder modificar los recursos solo los va a poder consultar, con la vista ejecuta una query para obtener los recursos, el servidor consulta los recursos en la base de datos y se los devuelve a la clase. En este momento la clase MOSkittDataStore tiene una visión de los recursos instanciados en la base de datos, en esa sesión.

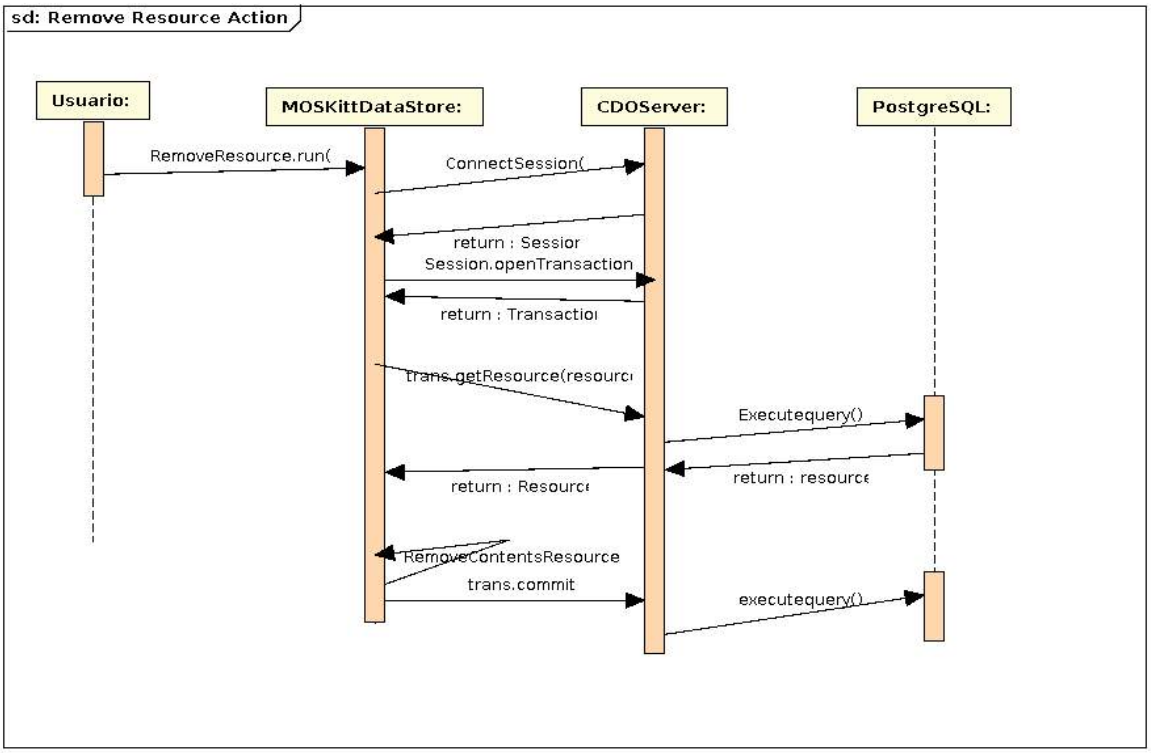
2. Diagrama de Secuencia del Proceso de apertura del recurso



El siguiente diagrama muestra la secuencia de la apertura de un recurso que se encuentra persistido en una base de datos. Antes de realizar este proceso, el usuario ha ejecutado una inicialización con este obtiene los recursos en una sesión. El procedimiento para abrir un recurso con su editor correspondiente es el siguiente:

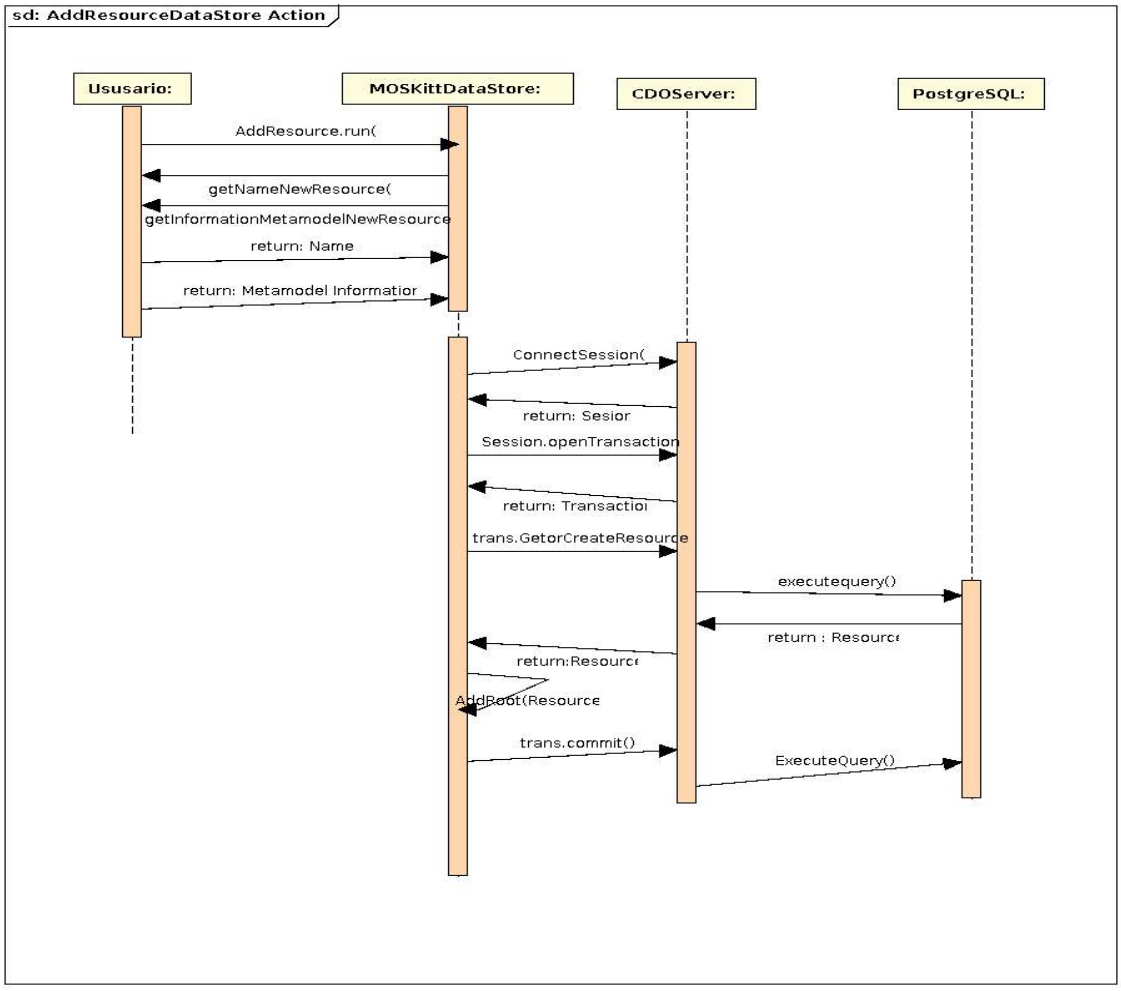
El usuario ejecuta la acción openResource sobre un recurso en la clase MOSkittDataStore, lo primero que hace es la conexión nuevamente sobre el servidor CDO en este caso, también abrirá una transacción, esta vez queremos poder modificar un recurso, con la transacción abierta hacemos un OpenEditor en el WorkBench que se encargar de cargar el recurso, como son recursos de FEFEM sera este quien le pida el recurso al servidor nuevamente, esta vez tenemos una transacción abierta por lo que no abra ningún problema, con el recurso obtenido es el WorkBench quien localiza con que editor abre el recurso y lo abre.

3. Diagrama de Secuencia del Proceso de borrado de un recurso



En este proceso lo que se pretende es la eliminación del recurso, para ello previamente el usuario debe de haber realizado una inicialización de los recursos, seleccionado el recurso a eliminar , el usuario realiza la acción RemoveResource sobre MOSKittDataStore, como siempre MOSKittDataStore conecta una sesión y como también se va a modificar los recursos, abre una transacción obtiene el recurso, y la vista elimina los contenidos del recurso, para finalizar realiza commit para que se guarden los cambios realizados.

4. Diagrama de Secuencia del Proceso de añadido de un nuevo recurso

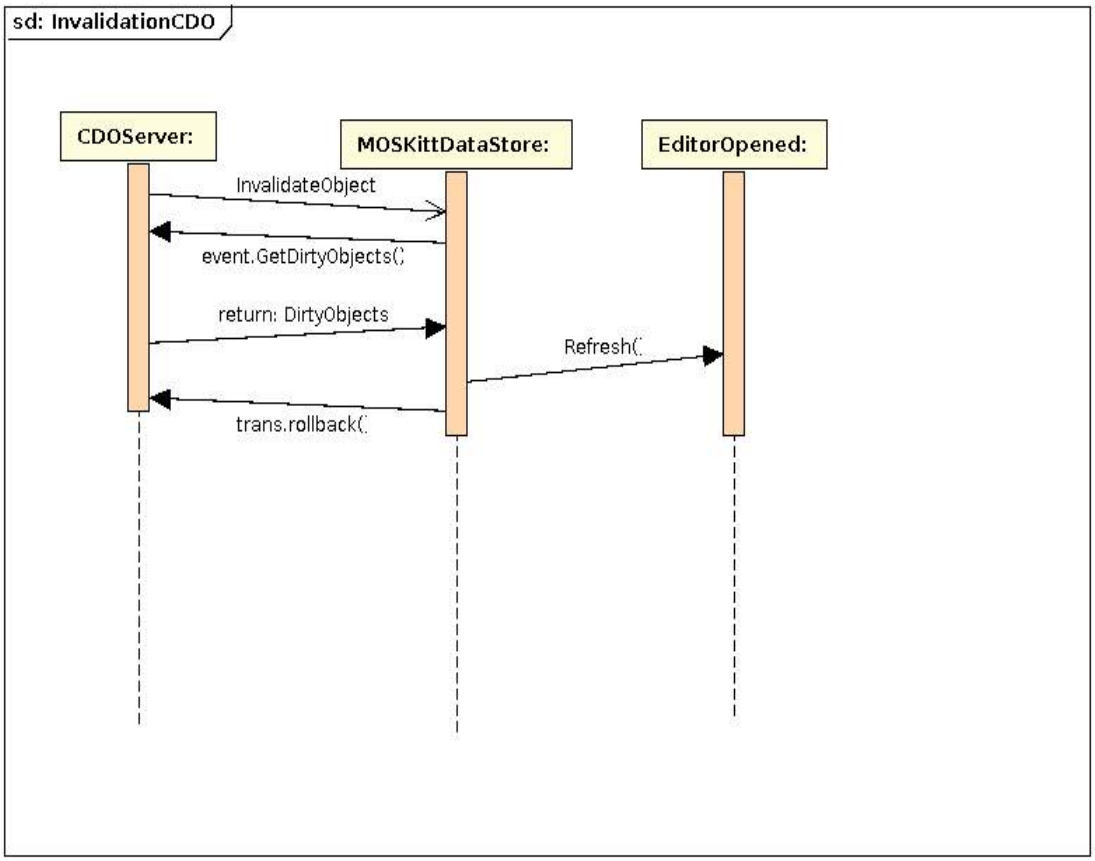


Este proceso se encargara del añadido de un nuevo recurso para que sea persistido en una base de datos. El usuario en este proceso realiza la acción de AddResource, en ese momento , MOSKittDataStore le pedirá Nombre del recurso y seleccionara a que metamodelo pertenece el nuevo recurso, sera el programador de los editores de formulario, quien añada su editor de formulario a esa lista. Solo se podrán añadir nuevos recursos aquellos editores de formulario que se hallan adherido a esta lista, una vez obtenida la información del usuario, MOSKittDataStore conecta a la sesión y abre la transacción.

Con la transacción obtenida invoca a GetorCreateResource indicándole los datos proporcionados por el usuario, el servidor ejecutara la query y devolverá un recurso vacío en el caso de que no exista, si el recurso es vacío, se le deberá de añadir el elemento raíz del metamodelo, una vez añadido se hace el commit.

Este proceso no se realiza solo, cuando termina de añadir el recurso ejecuta OpenResource para abrir el recurso nuevo.

5. Diagrama de Secuencia del Proceso de invalidación de una transacción



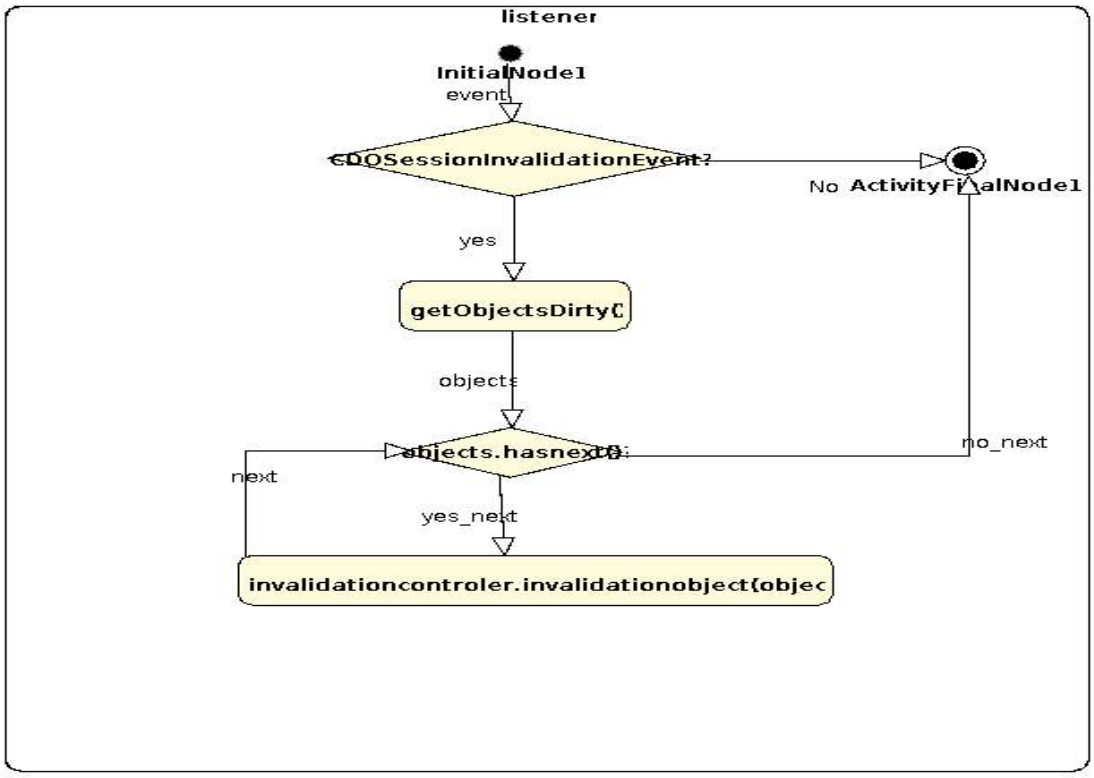
Este diagrama refleja el procedimiento a realizar en el caso de que llegue un evento de invalidación de la transacción esto se produce cuando otro usuario esta editando el mismo recurso que tu y los datos no reflejan la realidad, es decir no datos actuales no reflejan los cambio realizados por el otro usuario. El proceso es el siguiente el Servidor CDO emite un evento de invalidación, en ese momento MOSKittDataStore solicitara aquellos objetos que han sido modificados, según los objetos, estos refrescaran o además de un refresco tirara atrás la transacción haciendo un rollback y perdiendo los cambios no guardados del usuario. Este proceso se entiende mejor en los diagramas de actividad de a continuación.

5.5 Diagramas de Actividad

En esta sección se presentar los diagramas de actividad, los siguiente diagramas de actividad muestran el procedimiento a realizar cuando se recibe del Servido de CDO un evento que indica que se ha producido un cambio en el recurso abierto, esto es debido a que otro usuario accedido al mismo recurso y ha realizado una modificación, guardando los cambios.

Los siguiente diagramas vienen encadenados y muestra toda la actividad realizada desde que se recibe el evento hasta que es procesado.

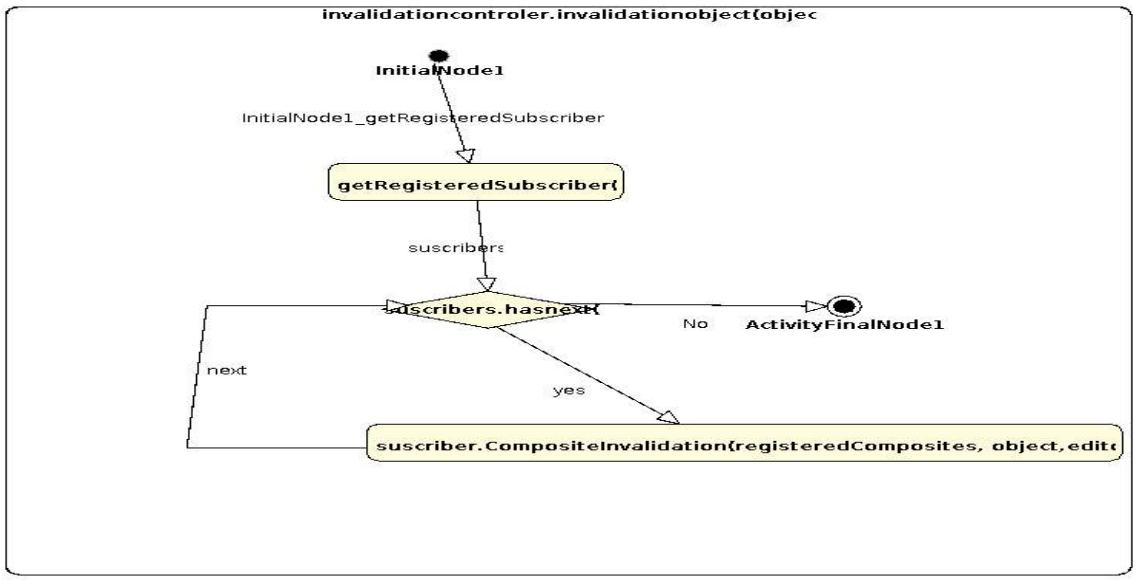
1. **Recepción del evento CDOSessionInvalidationEvent.**



En este diagrama muestra la recepción del evento y como se recogen aquello objeto que han sido modificados iterando por ellos y ejecutándose el invalidador, pasando el

objeto como parámetro.

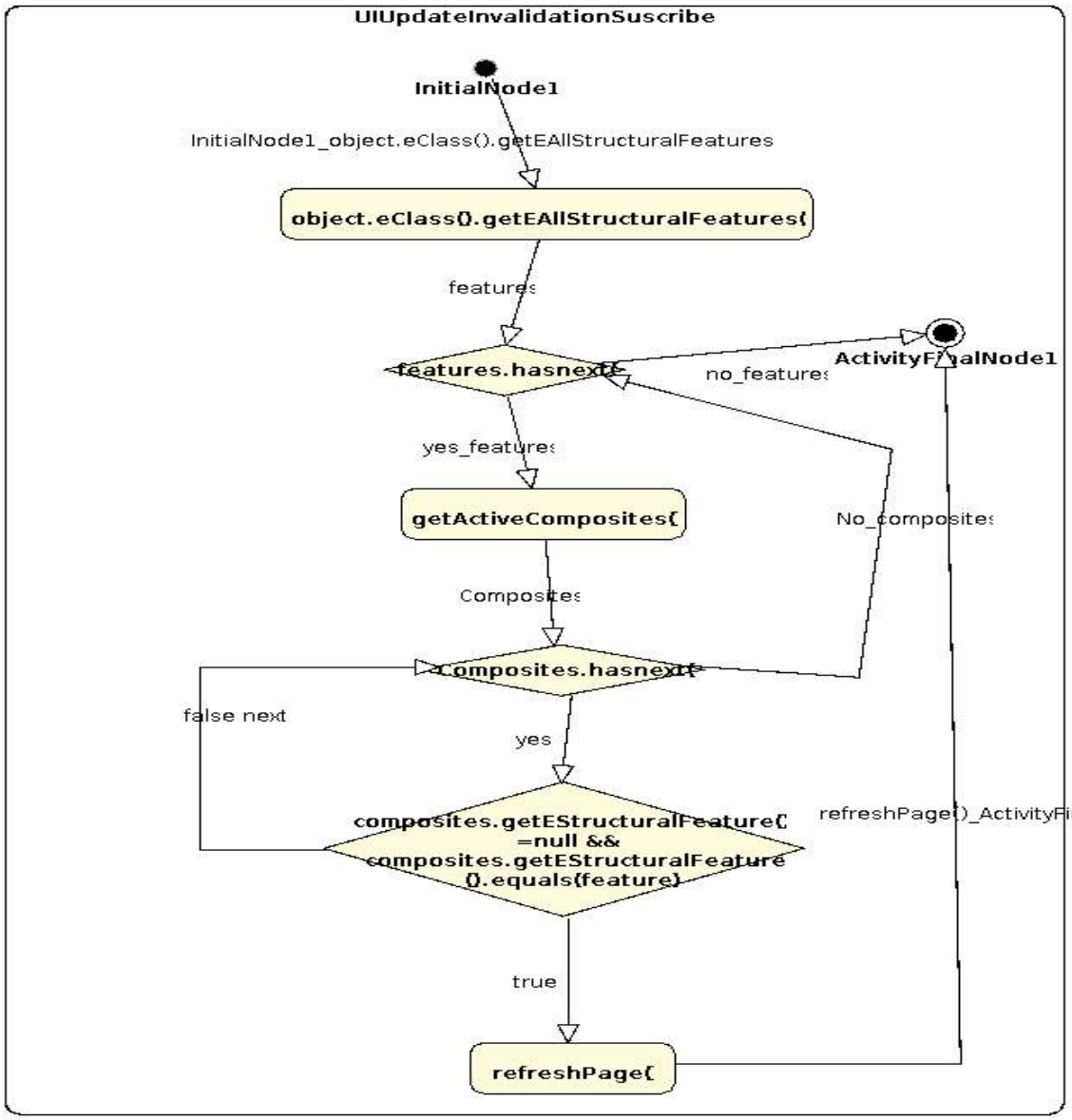
2. El invalidador y llamada de los subscriptores de modificación de objeto.



El diagrama anterior muestra el invalidador este lo que hace es recoger los suscriptores, los suscriptores son clases que se encargaran del proceso de los objetos cuando son invalidados, al suscriptor se le debe de pasar, el objeto, los composites registrados y el editor, la lista de registeredcomposite es una lista de aquellos composites que están instanciados en ese momento por el usuario cabe recordar que los composites son los elementos que forman un editor de formulario (textbox, checkbox, tables ..)

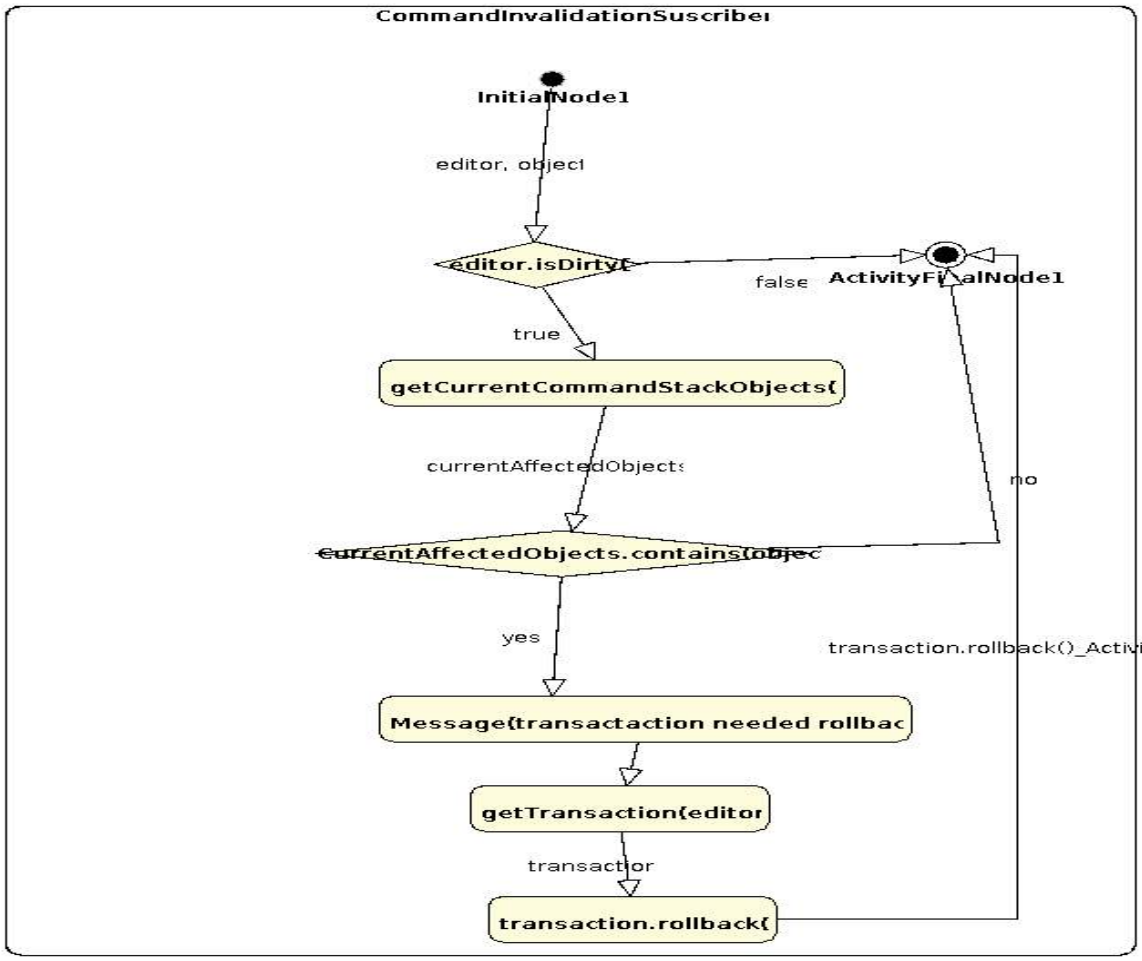
Actualmente solo hemos registrado dos suscriptores, un suscriptor para la actualización de la interfaz gráfica y otro para en el caso de que tenga el usuario comandos realizados pero no guardados sobre objetos invalidados.

3. UIUpdateInvalidationSuscribe



Este suscriptor es el encargado de actualizar la interfaz del editor en el caso de que el objeto invalidado haga referencia a alguno de los composites, para ello, a través del objeto invalidado extraemos las features que le afectan, recorremos los composites activos, si encontramos algún composite activo que de soporte a una feature y esta corresponde con alguna feature del objeto entonces actualizaremos el editor. Al actualizar el editor se verán los cambios recientes de manera simultanea.

4. CommandInvalidationSuscriber.



Este suscriptor comprueba si el usuario tenia algún cambio no guardado cuando le ha llegado el evento, si es así, va a mirar la pila de comandos buscando el objeto que le ha llegado al suscriptor si encuentra el objeto quiere decir, que el usuario al que le llega el evento ha modificado un objeto que ha sido modificado por por otro usuario que a guardado los cambios. En el caso de que esto ocurra la transacción queda invalidada y por lo tanto hay que hacer un rollback a la transacción para poderla recuperar, sin este rollback la transacción fallaría quedando muerto el editor. El problema que lleva el rollback es que el usuario pierde los cambios no guardados.

6. Implementación

En este apartado describiremos la implementación de parte del modulo, en concreto nos vamos a centrar en la implementación de los dos casos de prueba vistos en capitulo anterior en detalle: Añadir Recurso CDO y Abrir Recurso CDO.

Tanto en uno como en el otro, la acción ha debido de ser iniciada por el usuario, aquí lo que se pretende describir es la implementación de las acciones de Añadir Recurso CDO y Abrir Recurso CDO, no de la implementación de la invocación de las acciones.

6.2 Implementación Añadir Recurso CDO

Cuando el usuario invoca la acción se ejecuta el método run() de la acción. En la ejecución del método run() lo primero que hace es llamar al constructor del cuadro de dialogo donde aparecerá un textbox para la inserción del nombre y un combo box para que el usuario elija el tipo de metamodelo que tendrá el recurso que va a añadir.

La siguiente imagen muestra el código de lo anteriormente descrito.

```
@Override
	public void run() {
		RunAddResourceAction();
	}
	/**
	 * This method do the procces to add a new CDO Resource.
	 */
	private void RunAddResourceAction() {
		MoskittDataStoreAddResourceDialog moskittdatastoreaddResourceDialog;
		moskittdatastoreaddResourceDialog = OpenAddResourceDialog();

		if (moskittdatastoreaddResourceDialog.open() != InputDialog.OK) {
			return;
		}
		if (moskittdatastoreaddResourceDialog.getReturnCode() == 0) {
```

MoskittdatastoreaddResourceDialog es la clase que contendrá el cuadro de dialogo y los métodos de retorno de la información añadida por el usuario.

Una vez abierto el cuadro de dialogo la acción, se esperara hasta que el usuario le de a aceptar.

Cuando el usuario completa los campos y le da al botón aceptar se recogen los datos

que se van a necesitar para establecer la conexión. El siguiente fragmento de código muestra la recogida de los datos para establecer la conexión.

```
server = getServer();
ResourceCDO res = CreateResourceCDO(moskittdatastoreaddResourceDialog, server);
reporres=getRepositorie(server);
reporres.addResourceCDO(res);
GetInformationContributors(res);
```

CreateResourceCDO extraerá los datos del cuadro de dialogo y del servidor para poder crear el recurso posteriormente. GetInformationContributors coge la información referente a la contribución de los programadores del metamodelo seleccionado por el usuario.

A partir de ahí ya tenemos toda la información necesaria para preparar la sesión, conectar la sesión, abrir una transacción y crear el nuevo recurso dentro de la sesión.

```
String description=PrepareSession();
view.session=ConnectSession(description);

if (view.session == null) {
        return;
}

view.session.options().setGeneratedPackageEmulationEnabled(true);
CDOResource rcdo = null;
view.transaction=getTransactionSession(view.session);
try {
        rcdo = view.transactio.getOrCreateResource(res.getNombre());
        } catch (Exception e) {
                OpenMessageError();
                return;
        }
```

Lo primero que se hace es preparar la sesión , esto no es mas que crear una descripción con el siguiente formato:

tcp://<direccionIPServer>repositoryName=<NombreRepositorio>&automaticPackageRegistry= true";

También en la preparación de la sesión se preguntara si esta autentificada para el creado de la credencial que autorice el acceso al servidor.

Obtenida la descripción, ya podremos conectarnos a la sesión. Durante la conexión de la sesión se controlaran posibles problemas de conexión que en caso de no realizarse correctamente sera notificado mediante cuadros de dialogo al usuario.
Una sesión puede no ser establecida principalmente por tres causas:

1. Fallo de autenticación
2. Servidor desconectado
3. Contenido de la descripción de la sesión incorrecto.

Con la sesión creada, para poder acceder en modo lectura y escritura para insertar el recurso debemos de crear una transacción, para ello basta con invocar sobre la sesión el método `session.openTransaction();`

Una vez tenemos creada la transacción ejecutaremos el getOrCreateResource(nombre del recurso) . Con el método anterior, en el caso de que el recurso exista no se creara sino que se obtendría.
Con el método anterior ya tendríamos creado el recurso el recurso con el nombre indicado, pero este recurso es un recurso vacío, para que sea un recurso con un modelo dentro debemos de insertar el elemento raíz del metamodelo seleccionado por el usuario , para ello primero comprobaremos de que el recurso no tiene ese elemento intentando acceder a el, si no podemos acceder a ese elemento entonces insertaremos el elemento raíz.

```
try {
        rcdo.getContents().get(0);
} catch (Exception exc) {
        AddRootObject(res, rcdo);
}
```

Con el recurso creado ya correctamente solo nos faltaría abrirlo con su editor correspondiente. Esta parte sera explicada en detalle en el apartado siguiente.

6.3 Implementación Abrir Recurso CDO

En este apartado explicaremos el procedimiento a la hora de abrir un recurso CDO, es

decir que esta persistido en la base de datos. No va a ser solo el proceso de abrir el recurso, también abra que incorporar una serie de mecanismos para controlar el editor una vez abierto.

Al igual que el anterior se va a empezar desde el momento en el que el usuario seleciona uno de los recursos listados y selecciona la accion de abrir.

En ese momento, se invocara al metodo run() de la clase `OpenResourceAction`. En ese método se invoca al método principal de la clase llamado `RunOpenResourceAction()` y se decide también si se ha de añadir a la pagina del workbench el listener que nos informara del cerrado del editor para poder quitar los posibles listeners que instanciemos sobre el editor abierto.

```
@Override
        public void run() {
                RunOpenResourceAction();
                if(!ListenerOn){
                        page.addPartListener(editorListen);
                }
        }
```

El metodo RunOpenResourceAction() lo primero que hara sera obtener el recurso CDO selecionado en la vista. Para ello lo unico que hace es una consulta sobre el viewer de la vista.

```
private void RunOpenResourceAction() {
                if (((IStructuredSelection) view.treeViewer.getSelection())
                                .getFirstElement() instanceof ResourceCDO) {

                        ResourceCDO dsc = getResourceSelected();
                        String NsUri = dsc.getNSUri();
                        String editorid = getEditorId(NsUri);
```

A través de la URI del Metamodelo del recurso, podemos acceder al identificador del editor, esto se hace principalmente consultando a los editores que han contribuido su información a la vista, la manera de contribuir a la vista los editores creados por los programadores se explica en detalle en el capitulo siguiente.

Muchas veces el programador de un Metamodelo no contribuye con la información del identificado del editor, en estos casos el editor a abrir sera el que viene por defecto en CDO2.0.

Con el recurso y el editorid, solo necesitaremos preparar la sesión , conectar la sesión y con la sesión abierta invocar la apertura del editor. La preparación de la sesión y la conexión de la sesión se sigue el mimo procedimiento empleado en el apartado anterior.

A continuación se muestra el fragmento de código donde se prepara la sesión y se conecta a la sesión, si todo ha ido bien es decir la sesión se abierto correctamente se invoca al método openEditor pasándole como parámetros la sesión, el recurso y el identificador del editor.

```
getServerCDO(dsc);
String description=PrepareSession();
view.session=ConnectSession(description);
//This is used when in the same repository contains diferrentes package
// if this option is true don't needed to install all Epackage.
view.session.options().setGeneratedPackageEmulationEnabled(true);
        if(view.session != null){
        IEditorPart editor = openEditor(dsc,editorid,view.session);
```

El método openEditor sera el encargado de abrir el editor sobre Workbench de MOSKitt, para abrir un editor lo único que necesitamos es la URI del recurso y el identificador del editor abrir.

Para abrir un recurso se invoca al método openEditor de la pagina activa del workbench.

IEditorPart org.eclipse.ui.IWorkbenchPage.openEditor(IEditorInput input, String editorId)

Como hemos indicado anteriormente para abrir el recurso necesitamos su URI, para crear la URI sobre el recurso seleccionado hacemos uso del siguiente código, donde getParentUUID() es el identificador universal único del repositorio donde se encuentra el recurso y getNombre() devuelve el nombre del recurso.

```
private IEditorPart openEditor(ResourceCDO dsc, String editorid, CDOSession session) {

        page = PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage();

        URI uri = URI.createURI("cdo://" + reporres.getParentUUID() + "/" + dsc.getNombre());
```

Con el código anterior tenemos la URI del recurso ahora hay que diferenciar dos casos:

1. Cuando se ha especificado un editor concreto para abrir el recurso.
2. Cuando hay que abrir el recurso con el editor por defecto de CDO2.0

En el primer caso la implementación del método openEditor de la pagina activa del workbench es trivial.

```
try {
        return page.openEditor(new URIEditorInput(uri),editorid);
} catch (PartInitException e1) {
        e1.printStackTrace();
}
```

Para el segundo caso necesitamos la clase `StandardCDOEditorInput` con los parametros: transaccion, Nombre del recurso, URI del recurso, esta clase nos devuelve el `CDOEditorInput`. Con el `CDOEditorInput` y el identificador del editor `CDOEditor.EDITOR_ID` ya se podría abrir el recurso.

```
try {
        return page.openEditor(new StandardCDOEditorInput(trans,dsc.getNombre(), false,
                                uri.toString()),CDOEditor.EDITOR_ID);
} catch (PartInitException e1) {
        e1.printStackTrace();
}
```

A continuación se muestra el código completo del método para abrir el editor.

```
private IEditorPart openEditor(ResourceCDO dsc, String editorid, CDOSession session) {

        page = PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage();

        URI uri = URI.createURI("cdo://" + reporres.getParentUUID() + "/" + dsc.getNombre());

        if(editorid.equals("")){
                CDOTransaction trans = getTransaction(session);
                if (page != null)
                        try {
                        return page.openEditor(new StandardCDOEditorInput(trans,
                                                dsc.getNombre(), false, uri.toString()),
                                                        CDOEditor.EDITOR_ID);
                                } catch (PartInitException e1) {
                                        e1.printStackTrace();
                                }
                        return null;
                }
        else{
                if (page != null)
                        try {
                                return page.openEditor(new URIEditorInput(uri),editorid);
                        } catch (PartInitException e1) {
                                e1.printStackTrace();
                        }
                return null;
                }
        }
```

Con lo anterior descrito el editor se encontrara abierto pero necesitamos registrar el editor abierto para poder capturar los diferentes eventos que se produzcan sobre el. Los eventos que tenemos que capturar sobre el para que se produzca la sincronización simultanea son los siguientes.

1. Cambios en la pila de comandos del editor
2. Guardado del editor
3. Edición remota del recurso.

El primero lo usamos para registrar los cambios locales que producimos nosotros guardándonos el identificador del objeto editado en una pila que podremos consultar.

El segundo sirve para vaciar esa pila de objetos editados, que consultamos cuando recibimos un evento de edición remota del recurso.

El tercero es la recepción de un evento de edición remota del recurso abierto, cuando eso sucede preguntamos que objetos han sido modificados remotamente y puede ocurrir dos cosas.

1. Que modifiquen un objeto que he modificado yo sin llegar a guardar los cambios y por tanto se encuentra en la pila de objetos modificados
2. Que no se encuentre los objetos modificados en la pila.

Cuando ocurre el primer caso nuestra transacción no sirve por lo que nos obligamos a perder los datos no guardados y hacer un rollback a la transacción, a continuación del rollback haremos un refresco de los datos del recurso en el editor. En el segundo caso solo tendremos que hacer el refresco de los datos del recurso en el editor.

Es de esta manera como se mantienen los datos sincronizados y visualizados simultáneamente con los datos que se encuentran en la base de datos.

7. Ejecución y Puesta en Marcha

En este apartado se mostraran manuales de la vista MOSKittDataStore, como levantar un servidor CDO en nuestro ordenador, como preparar nuestro metamodelo para que sea soportado por CDO y como construir nuestro propio editor a través de FMF y este sea contribuido a la vista MOSKittDataStore.

7.1 Manual MOSKittDataStore

Descripción

La vista MOSKitt DataStore nos permite gestionar recursos que se encuentran en una base de datos. Estos recursos contienen datos de un metamodelo concreto. Con la vista podremos instanciar lo recursos que se encuentran en la base de datos y ver a que metamodelo pertenecen, añadir nuevos recursos para un metamodelo concreto, eliminar recursos existentes y abrir los recursos para editarlos con el editor correspondiente.

Antes de empezar a ver la partes debemos de conocer el significado de repositorio.

Se llama repositorio a un conjunto de recursos almacenados en una base de datos accesible desde cualquier lugar. Un repositorio se identifica por un ID único en el servidor desde el cual se accede, también tiene un nombre ya que muchas veces el ID no nos da pistas sobre los recursos que contiene el repositorio. Es posible almacenar recursos de distintos metamodelos en un
mismo repositorio, aunque no recomendable, ya que la idea del repositorio es tener almacenado los recursos de un mismo tipo en un mismo repositorio.

Partes

El MOSKitt DataStore podemos dividirlo en dos partes:

1. Las Preferencias MOSKittDataStore

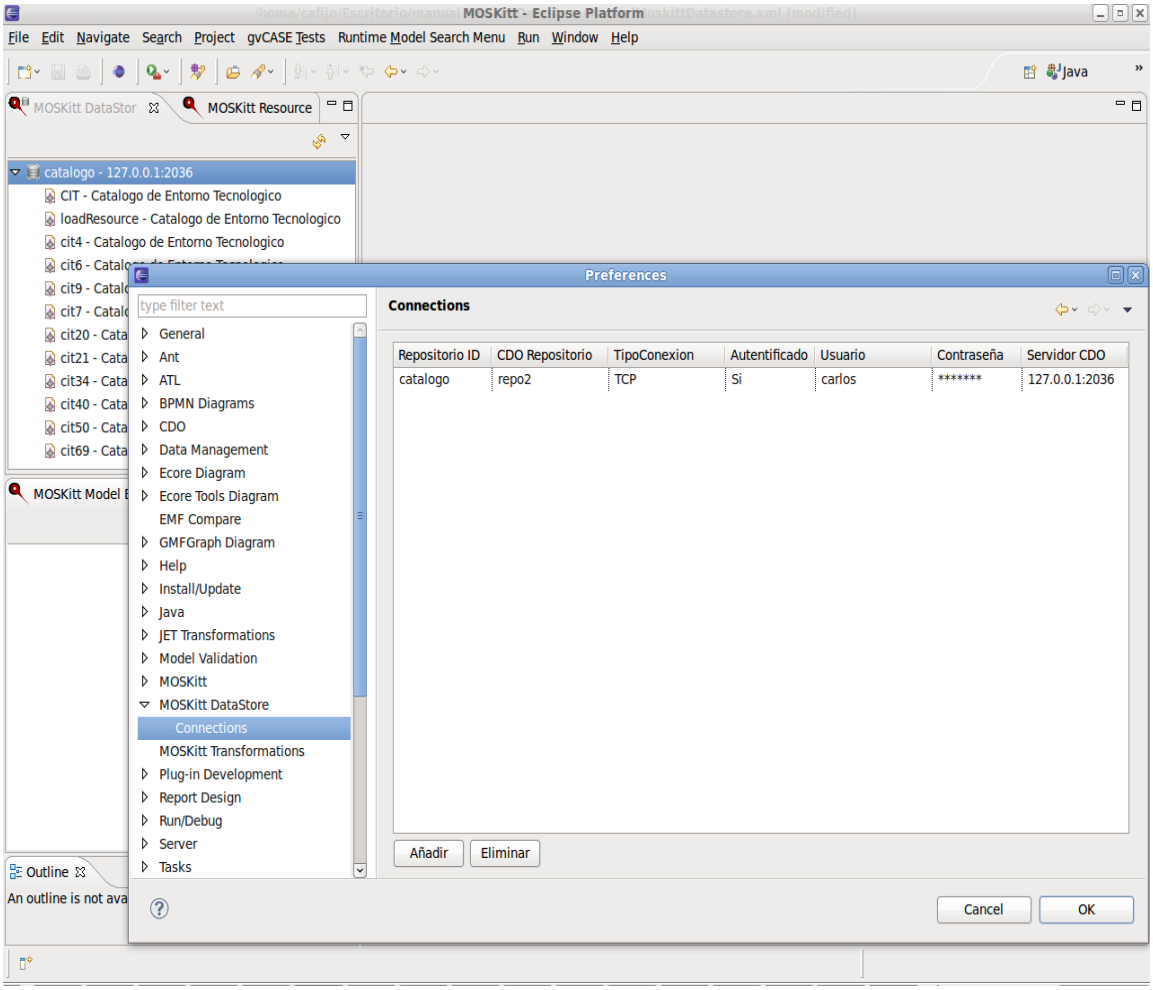
Desde aquí estableceremos la configuración de conexión al servidor CDO y sus distintos repositorios.

2. La vista MOSKittDataStore

Desde aquí podremos acceder a los recursos de los distintos repositorios y abrirlos con su editor para ser modificados y guardados posteriormente.

1. Las Preferencias MOSKittDataStore

Para acceder a las preferencias del MOSKittDataStore hay que ir al menú **Window → Preferences**

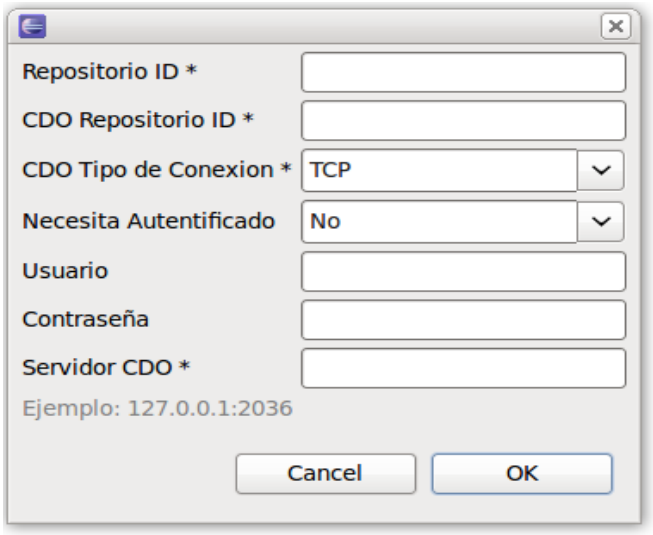


Desde aquí establecemos la configuración de la conexión con el servidor CDO, la información necesaria es la siguiente:

1. **Repositorio ID:** Es el identificador universal del repositorio dentro del servidor CDO.

2. **CDO Repositorio:** Nombre del repositorio.
3. **Tipo de conexión:** Protocolo de transporte que se utilizara en la conexión con el servidor CDO.
4. **Autentificado:** Indica si el servidor CDO requiere una autenticación de los clientes que quieren establecer la conexión.
5. **Usuario:** Nombre de usuario, este dato es opcional y depende de si se necesita autentificado o no el servidor .
6. **Contraseña:** Contraseña del usuario , este dato es opcional y depende de si se necesita autentificado o no
   el servidor
7. **Servidor CDO:** Dirección IP y puerto del servidor CDO donde se encuentra el repositorio.

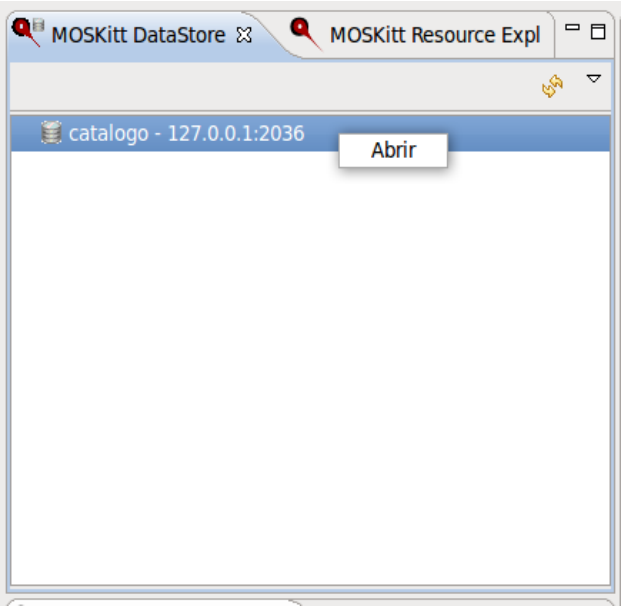
Para añadir una nueva conexión a un repositorio, es suficiente con hacer click en el botón añadir, este abrirá un cuadro de dialogo como el siguiente:



Desde aquí podremos escribir la nueva configuración para el acceso de un repositorio en una conexión a un servidor CDO, si queremos eliminar una conexión a repositorio, solo tenemos que señala el repositorio que deseamos eliminar y pulsar el botón "**Eliminar**"

Podremos editar una conexión señalando la conexión a editar y pulsando el botón "**Editar**", se nos abrirá un cuadro de dialogo similar al anterior desde el cual podremos modificar los valores de la conexión seleccionada.

2. La vista MOSKittDataStore

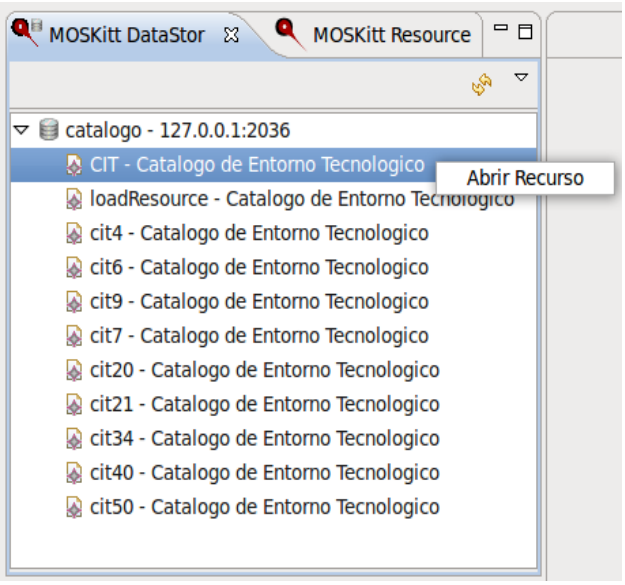


La vista MOSKittDataStore nos mostrara los distintos repositorios configurados previamente en la pagina de preferencias de MOSKittDataStore

La vista muestra una lista de repositorios en concreto nos mostrara por cada repositorio el nombre y la dirección IP de la maquina junto con el puerto donde está almacenado el repositorio.

Como hemos comentado anteriormente un repositorio puede contener distintos recursos de diferentes metamodelos. Para poder ver los distintos recursos que contiene un repositorio, es suficiente con hacer dobleclick sobre el repositorio que queremos ver. Otra opción de abrir el repositorio es situarnos con el ratón sobre el repositorio abrir y desplegar su menú contextual con el botón derecho a continuación pulsamos sobre "**Abrir**"

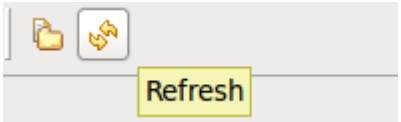
"Podemos ver cada repositorio como una carpeta compartida por muchos usuarios la cual contiene recursos compartidos, para ver los diferentes cambios producidos en la carpeta es suficiente con "Refrescar" la vista y abrir de nuevo el repositorio, con esto conseguiremos tener los recurso actualizados."



Los recursos dentro del repositorio se muestran como hijos del repositorio como si de un archivo dentro de una carpeta se tratase. Para diferenciar unos recursos con otros estos se muestran en la vista primero con su nombre, este nombre debe de ser único dentro del repositorio y a continuación se muestra el nombre del metamodelo al que pertenece el recurso.

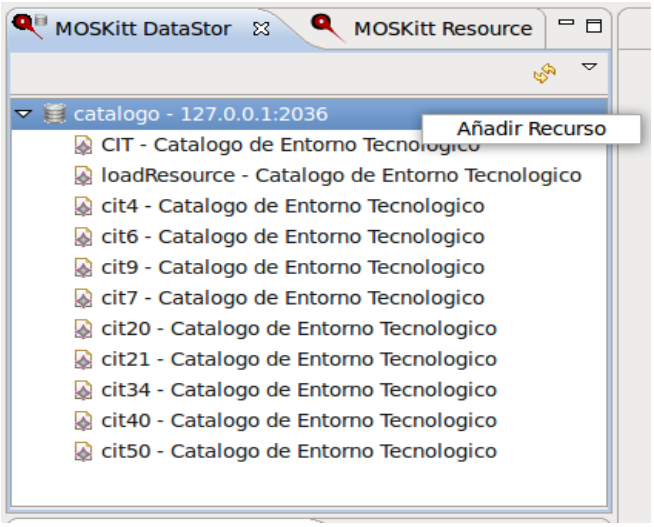
Para poder abrir el recurso es suficiente con hacer dobleclick sobre el recurso que se desea abrir, en ese momento se abre el editor de formulario correspondiente con el metamodelo del recurso. Otra opción de abrir el recurso es situarnos con el ratón sobre el recurso, desplegar su menú contextual con el botón derecho y a continuación pulsar sobre "**Abrir Recurso**" .

Desde el editor abierto es posible modificar la información del recurso para el posterior guardado del recurso. Es posible que desde otro lugar estén modificando el recurso por eso desde la barra de herramientas junto al "**Load Resource**" disponemos de la herramienta "**Refresh**" del propio editor.



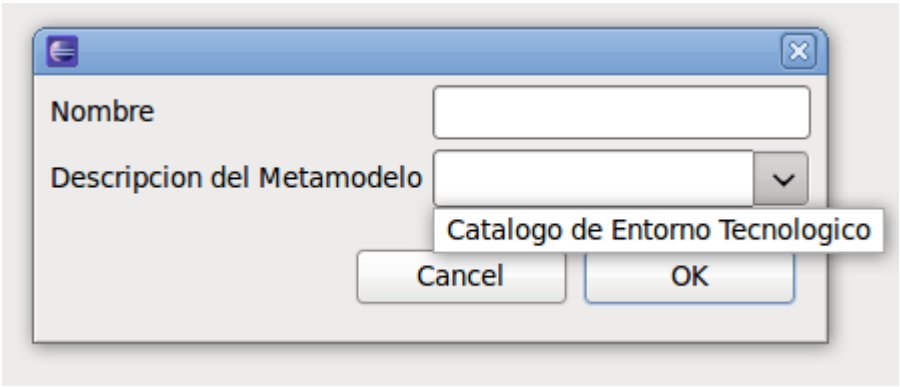
Es posible cargar recursos dentro de un recurso usando la función Load Resource con la siguiente URI:

cdo://<Repositorio ID>/<NombreRecurso>



Es posible añadir recursos al repositorio, seleccionamos el repositorio con el botón derecho y pulsamos sobre "**Añadir Recurso**" .

Con esta acción se nos ejecuta un cuadro de dialogo como el siguiente en el que tendremos que especificar el nombre del recurso y el metamodelo al que pertenece de una lista, solo los metamodelos adaptados a CDO aparecerán en esta lista.



Pulsando OK el nuevo recurso de guardara en el repositorio y nos lo abre con editor adecuado. Si añadimos un nuevo recurso ya existente lo que hace es abrirnos el recurso solamente.

7.2 Manual Servidor CDO

Instalación del servido CDO, configuración del servidor, arranque del servidor y parado, configuración de los usuarios que aceden al servidor.

Contenidos

1.Instalación del servidor de CDO.

Para la instalación y ejecución del servidor CDO necesitamos:

1. El CDOServer.zip
2. Una maquina virtual de Java instalada y referenciada.
3. Una base de datos Postgres

Una vez tenemos las dos cosas solo tenemos que extraer el CDOServer.zip. Con esto tendríamos instalado el ServidorCDO. Antes de arrancar el servidor necesitamos configurarlo.

2.Configuración del servidor

Para configurar el ServidorCDO necesitamos crear un archivo llamado cdo-server.xml dentro de la carpeta CDOServer/eclipse/config

El archivo cdo-server.xml tiene el siguiente formato:

```
<?xml version="1.0" encoding="UTF-8"?>

<cdoServer>

<acceptor type="tcp" listenAddr="0.0.0.0" port="2036">
        <!-- <negotiator type="challenge" description="/temp/users.db"/> -->
</acceptor>

<repository name="repo1">
        <property name="overrideUUID" value="1ff5d226-b1f0-40fb-aba2-0c31b38c764f"/>
        <property name="supportingAudits" value="true"/>
        <property name="supportingRevisionDeltas" value="true"/>
        <property name="verifyingRevisions" value="false"/>
        <property name="currentLRUCapacity" value="10000"/>
        <property name="revisedLRUCapacity" value="100"/>

        <store type="db">

                <!-- type: horizontal | vertical | <any user-contributed type>-->
                <mappingStrategy type="horizontal">
                <!--  ONE_TABLE_PER_REFERENCE | ONE_TABLE_PER_CLASS | ONE_TABLE_PER_PACKAGE |
ONE_TABLE_PER_REPOSITORY -->
                <property name="toManyReferences" value="ONE_TABLE_PER_REFERENCE"/>
                <!-- LIKE_ATTRIBUTES -->
                <property name="toOneReferences" value="LIKE_ATTRIBUTES"/>
                <!-- MODEL | STRATEGY-->
                <property name="mappingPrecedence" value="MODEL"/> </mappingStrategy>

                <!--<dbAdapter name="derby-embedded"/>
                <dataSource class="org.apache.derby.jdbc.EmbeddedDataSource"
                databaseName="/temp/cdodb1"
                createDatabase="create"/>-->

                <!--<dbAdapter name="hsqldb"/>
                <dataSource class="org.eclipse.net4j.db.hsqldb.HSQLDBDataSource"
                database="jdbc:hsqldb:mem:cdodb1"
                user="sa"/>-->

                <!--<dbAdapter name="mysql"/>
                <dataSource class="com.mysql.jdbc.jdbc2.optional.MysqlDataSource"
                url="jdbc:mysql://localhost/cdodb1"
                user="sa"/>-->

                <dbAdapter name="postgresql"/>
                <dataSource class="org.postgresql.ds.PGSimpleDataSource"
                url="jdbc:postgresql://localhost:5432/cdo"
                databaseName="cdo"
                user="cdo"
                password="cdo"/>

        </store>
</repository>

</cdoServer>
```

Las siguientes secciones explican cada uno de los elementos XML.

<cdoServer>

Es elemento raíz: Puede contener uno o varios elementos <acceptor> y cero,

uno o varios elementos <repository>.

Ejemplo:

```
<cdoServer>
        <acceptor>
        </acceptor>
        <repository>
        </repository>
        <repository>
        </repository>
</cdoServer>
```

El servidor anterior contendría 1 acceptor y dos repositorios.

<acceptor>

Aquí es donde le especificamos en que dirección y puerto donde vamos a escuchar las peticiones del servidor también especificamos el protocolo actualmente soportado solo TCP.

```
<acceptor type="tcp" listenAddr="0.0.0.0" port="2036">
```

Dentro del acceptor podemos especificar o no un negotiator para tener un control de los usuarios a la hora de acceder. El control de acceso lo explicamos mas adelante.

<repository>

El repositorio contiene un elemento nombre. El elemento repositorio puede contener varias propiedades y debería **contener un único <store>**

```
<repository name="repo1">
```

Pero lo que realmente identifica el repositorio es la propiedad:

```
<property name="overrideUUID" value="1ff5d226-b1f0-40fb-aba2-0c31b38c764f"/>
```

Este es el identificador universal del repositorio debe de ser único. Normalmente el value contiene un valor mas cómodo ya que un recurso dentro del servidor se identifica de la siguiente manera.

Algunas propiedades mas son:

```
<property name="supportingAudits" value="true"/>
<property name="supportingRevisionDeltas" value="true"/>
<property name="verifyingRevisions" value="false"/>
<property name="currentLRUCapacity" value="10000"/>
<property name="revisedLRUCapacity" value="100"/>
```

Propiedad supportingAudits

Especifica si el repositorio apoyará puntos de vista de auditoría o no.

Propiedad supportingRevisionDeltas

Especifica si el depósito pasará cambiado las revisiones de su tienda en forma de deltas de revisión (en que se recibieron de la capa de protocolo) o como revisiones de conjunto

Propiedad verifyingRevisions

Especifica si el repositorio se compruebe la versión en caché o no.

Propiedad currentLRUCapacity

Especifica la capacidad de la memoria caché de tamaño fijo para la versión actual en el repositorio. La revisión se llama actual si no se revisa.

Propiedad revisedLRUCapacity

Especifica la capacidad de la memoria caché de tamaño fijo para las revisiones revisado en el repositorio. La revisión se llama revisado en caso de una o más revisiones más recientes del mismo objeto existen.

<store>

El store debe de ser del tipo db.

```
<store type="db">
```

Este tipo nos permite tener los siguiente elementos:

- **mappingStrategy**
- **dbAdapter dataSoure**

  En el mappingStrategy establecemos la estrategia en el creado de las tablas. El tipo que admite esta versión es horizontal y podemos establecer las propiedades para la relaciones "toManyReferences" y "toOneReferences" tal y como se muestra en el ejemplo.

```
<mappingStrategy type="horizontal">
<!-- ONE_TABLE_PER_REFERENCE | ONE_TABLE_PER_CLASS | ONE_TABLE_PER_PACKAGE | ONE_TABLE_PER_REPOSITORY -->
```

```
<property name="toManyReferences" value="ONE_TABLE_PER_REFERENCE"/>

<!-- LIKE_ATTRIBUTES -->
<property name="toOneReferences" value="LIKE_ATTRIBUTES"/>

<!-- MODEL | STRATEGY-->
<property name="mappingPrecedence" value="MODEL"/>
</mappingStrategy>
```

Propiedad toManyReferences

Especifica cómo se construye el O / mapper R que se encargará de las ManyReferences del modelo . Los valores disponibles son:

ONE_TABLE_PER_REFERENCE: Cada referencia a varios del modelo tendrá su propia tabla de DB.

ONE_TABLE_PER_CLASS: Todas las referencias a varios de una clase del modelo compartirán una misma tabla en la base de datos.

ONE_TABLE_PER_PACKAGE: Todas las referencias a varios de un paquete del modelo compartirán una misma tabla en la base de datos.

ONE_TABLE_PER_REPOSITORY: Todas las referencias a muchos de todo el modelo de clases del repositorio compartirán una misma tabla en la base de datos.

Propiedad toOneReferences

Especifica cómo se construye el O / mapper R que se encargará de las OneReferences del modelo. Los valores disponibles son:

LIKE_ATTRIBUTES: Referencias individuales se almacenan en una columna CDOID en la misma tabla de la base de datos igual que los atributos de la clase del modelo . Esta opción es la opción por defecto.

Dentro del <store> también tenemos que tener los datos de la base de datos donde queremos almacenar el repositorio, para ello utilizamos las tags <dbAdapter> y <dataSource> . Por ejemplo en postgress utilizaríamos la siguiente configuración.

```
<dbAdapter name="postgresql"/>
```

```
<dataSource class="org.postgresql.ds.PGSimpleDataSource"
url="jdbc:postgresql://localhost:5432/cdo"
databaseName="cdo"
user="cdo"
password="cdo"/>
```

El dbAdapter en el caso de postgres utilizamos el postgresql, con esto se especifican las palabras reservada que utiliza una base de datos postgres. En el dataSource establecemos la información de la base de datos: dirección, nombre, usuario, contraseña.

Con todo los datos anteriores construiríamos nuestro archivo de configuración del servidor CDO que tiene que tener el nombre cdo-server.xml.

3.Configuración del control de acceso al servidor

Anteriormente hemos comentado que dentro del acceptor podemos tener un control de los usuarios que acceden al servidor para ello usamos la etiqueta negotiator tal y como se muestra a continuación.

```
<acceptor type="tcp" listenAddr="0.0.0.0" port="2036">
        <negotiator type="challenge" description="/temp/users.db"/>
</acceptor>
```

El negotiator es el que se encarga de negociar la entrada de peticiones de clientes al server. El tipo de negotiator que
usamos es el challenge que es un Negociador simple, pero eficaz y criptográficamente seguro. En negociator contiene una description donde escribiremos la ruta absoluta de nuestro fichero de usuarios.

Este fichero de usuarios tendrá el siguiente formato

```
tom=myverysecretpassword
```

Una vez establecido un negotiator todos los clientes necesitaran esta autentificados con usuario y password.

4.Comandos de arranque y parada del servidor

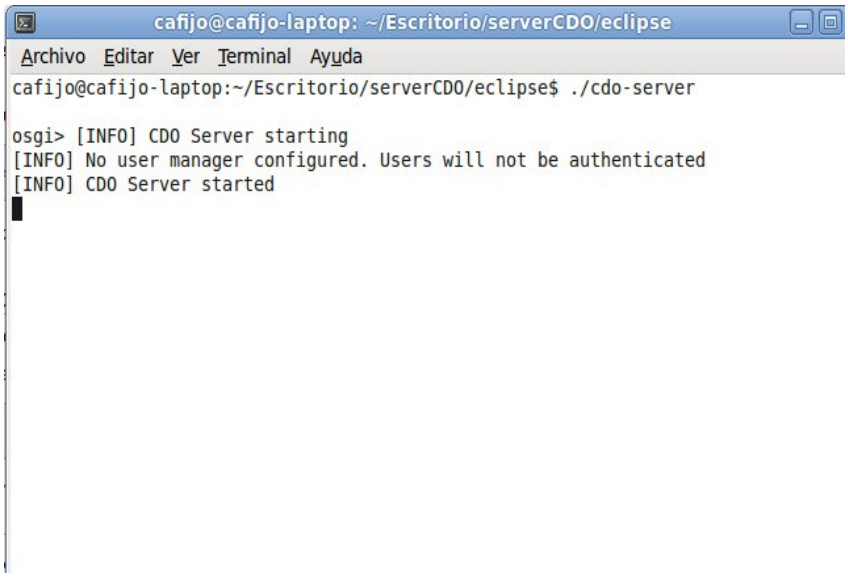
Configurado el archivo de configuración del server y guardado en la carpeta serverCDO/eclipse/config podemos arrancar el servidor. Para arrancar el servidor solo tenemos que acceder a la carpeta descomprimida ServerCDO/eclipse y ejecutamos el server con el siguiente comando.

./cdo-server

Para el arranque del servidor es necesario tener instalada una maquina virtual java. Para mas información de como instalar la maquina virtual java en tu ordenador siga los pasos de la documentación siguiente.

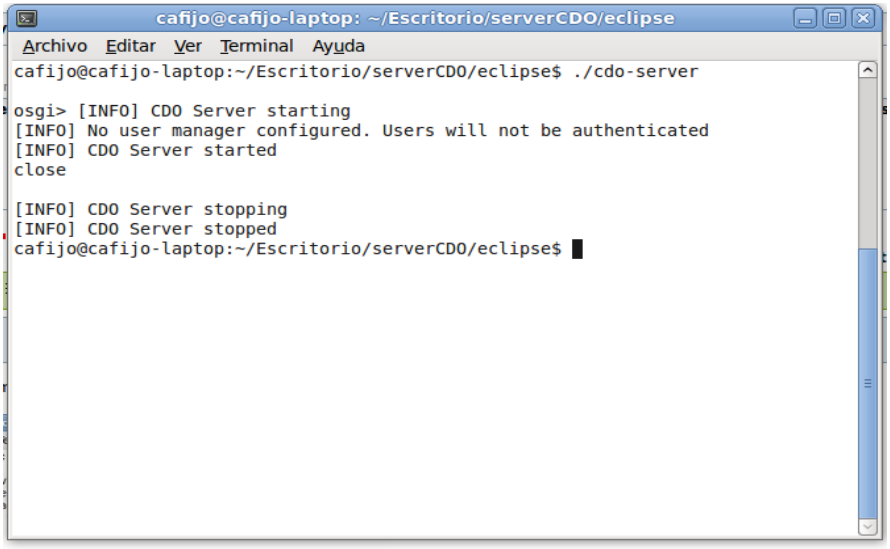
http://www.guia-ubuntu.org/index.php?title=Java

Con esto arrancamos el servidor cdo.



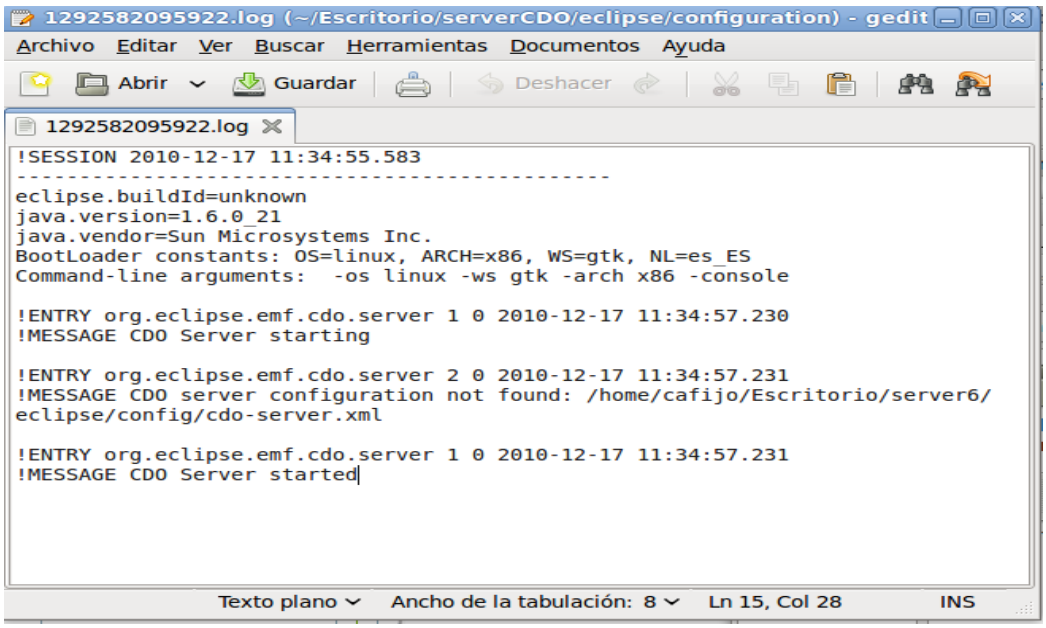
Para finalizarlo solo debemos escribir el siguiente comando.

close



5. Log File

El servidor contiene un .logfile que registrara todos los errores de las peticiones de los clientes. Para ver ese .log se
encuentra en la carpeta ServerCDO/eclipse/configuracion



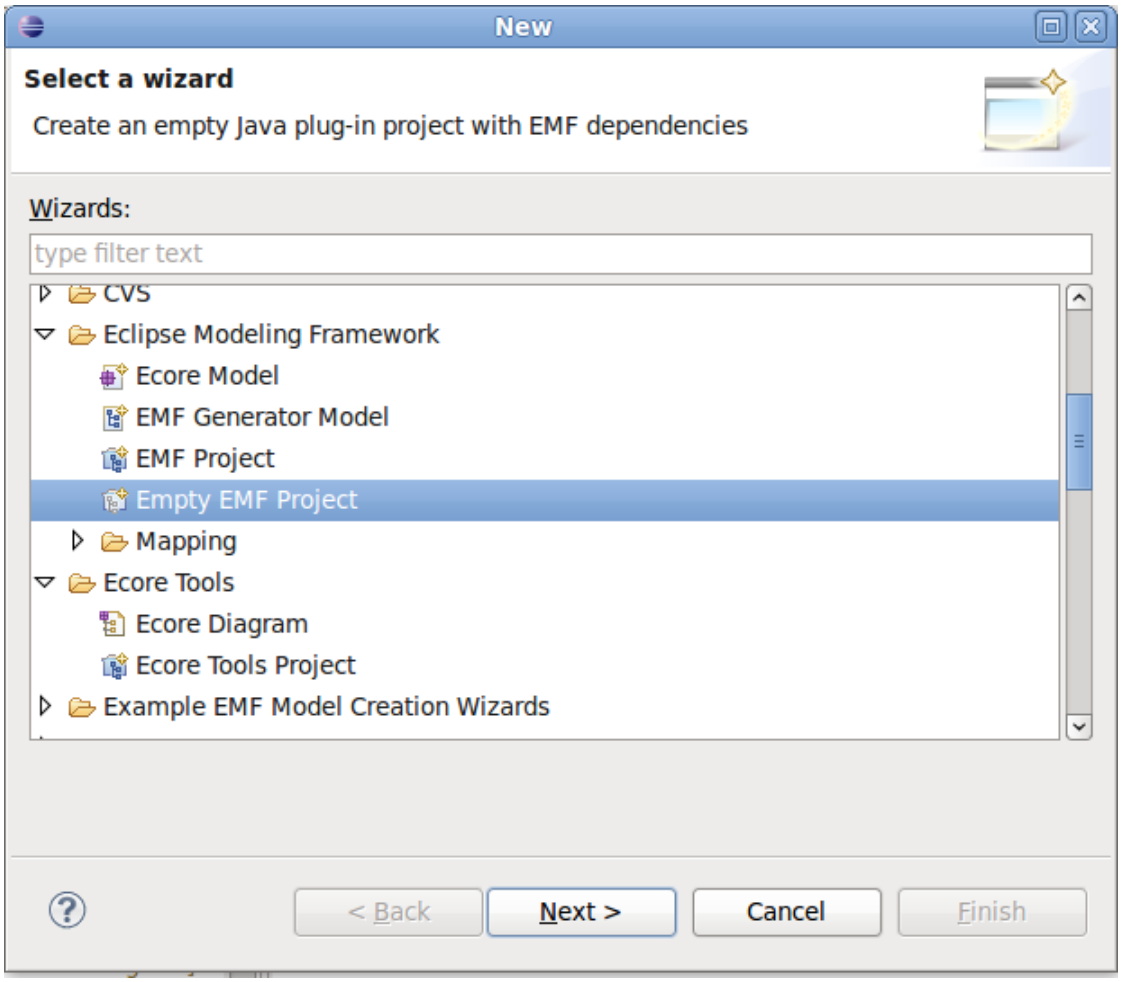
El aspecto de ese logFile es el siguiente.

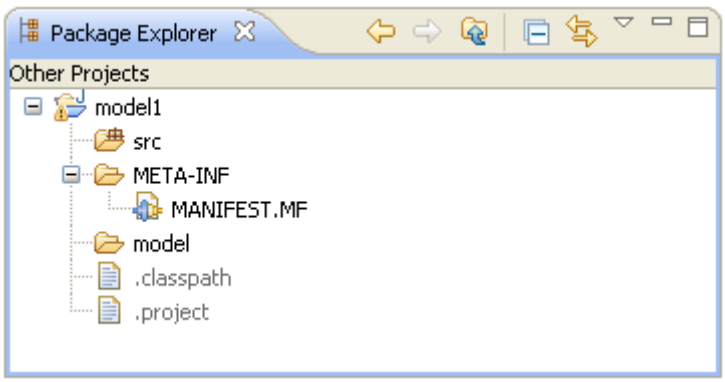
En este caso se puede contemplar un problema en la ruta de su fichero de configuración.

7.3 Preparar modelos EMF para CDO

Creando un Modelo Ecore

Para crear un modelo ecore de EMF, solo tenemos que crear a través del wizard un nuevo proyecto vació de EMF, a partir de ahí dentro de la carpeta model crearemos un Ecore model (.ecore). Los archivos .ecore para modelos CDO es el mismo que para modelos EMF puros.

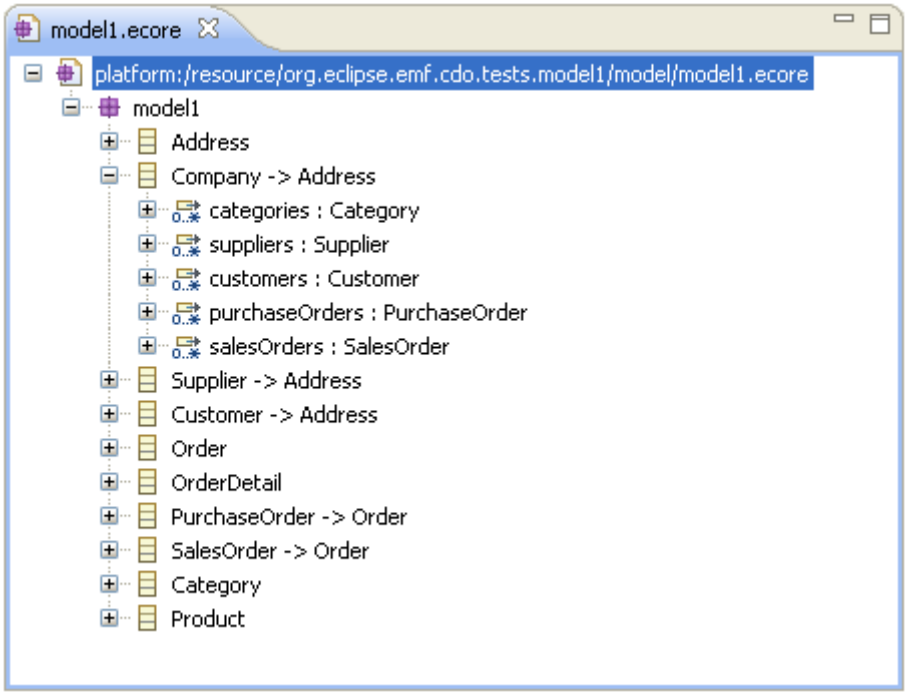




Un modelo ecore es la representación de un modelo de clases, es decir, el

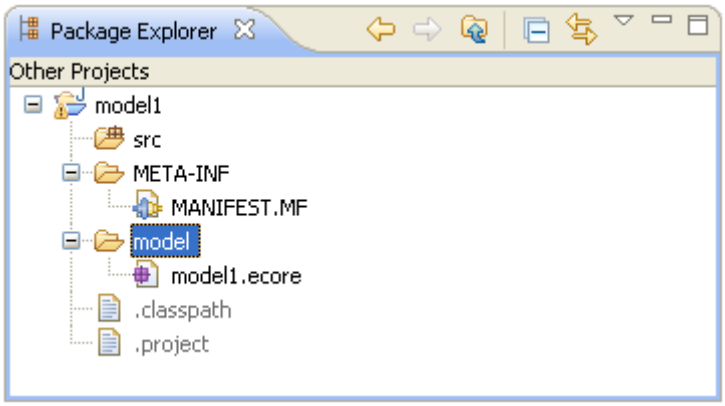
modelo ecore contendrá clases, atributos de las clases y relaciones de las clases, también contendrá un elemento raíz que a traves de el podrán ser alcanzadas todas las clases.

A continuación se puede ver un ejemplo de un modelo ecore con sus clases, atributos y relaciones entre clases.



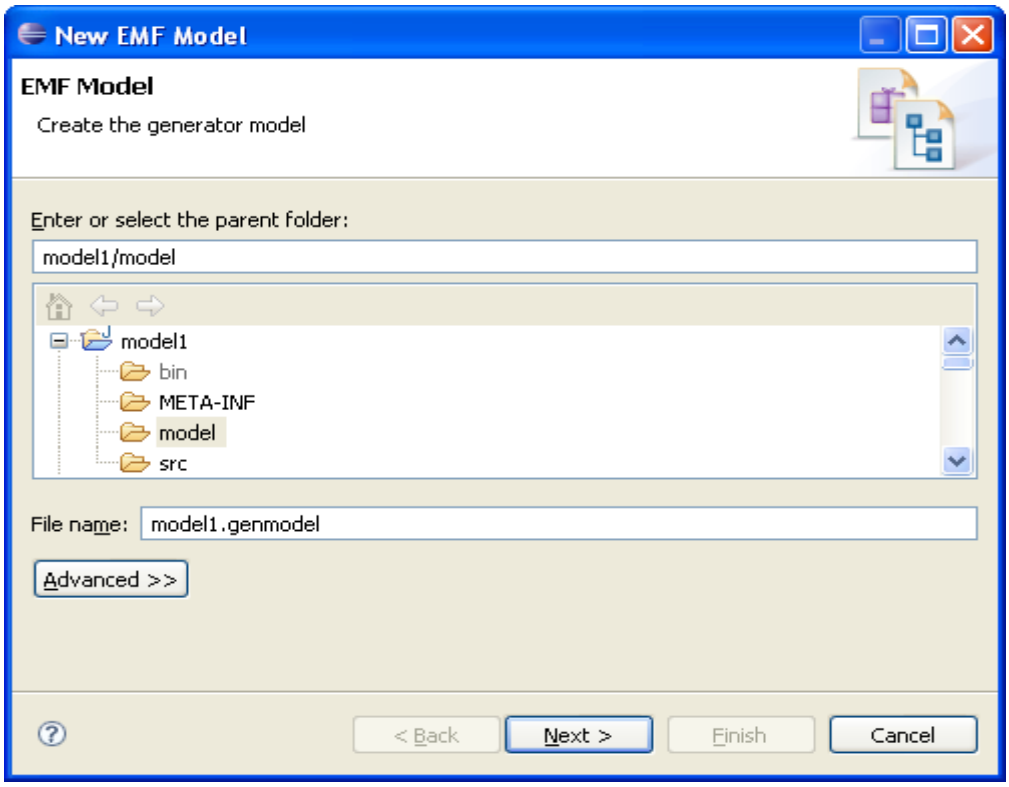
Se puede utilizar herramientas gráficas de ecore para crear el diagrama de clases, por ejemplo Ecore Diagram proporciona las herramientas visuales para el dibujado del diagrama, solo habría que relacionar el Ecore Diagram con su archivo .ecore.

Una vez creado el modelo ecore el proyecto debe de verse de la siguiente manera:

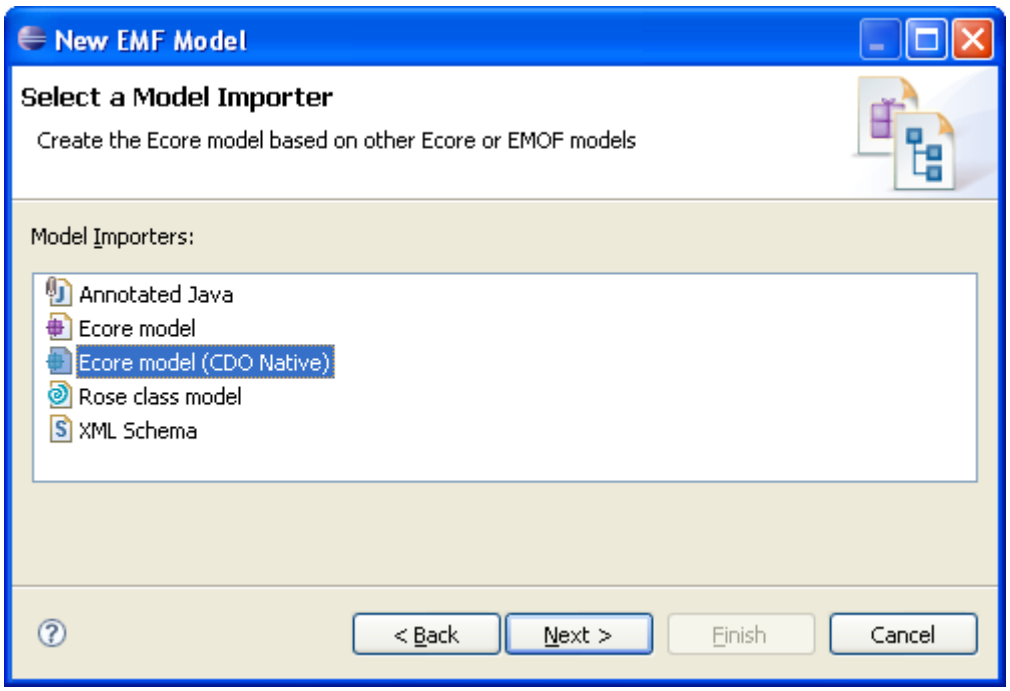


Generar un modelo Ecore para CDO

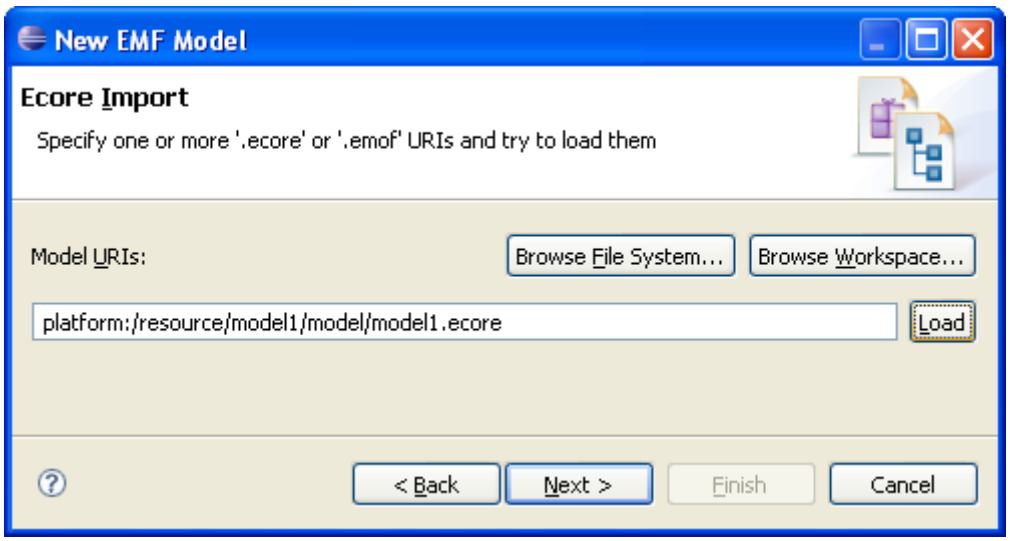
Una vez creado el modelo ecore, el siguiente paso es crear un modelo generador, para crear ese modelo generador solo tenemos que hacer clic con el botón derecho sobre el modelo ecore y seleccionar nuevo EMF Generador Model



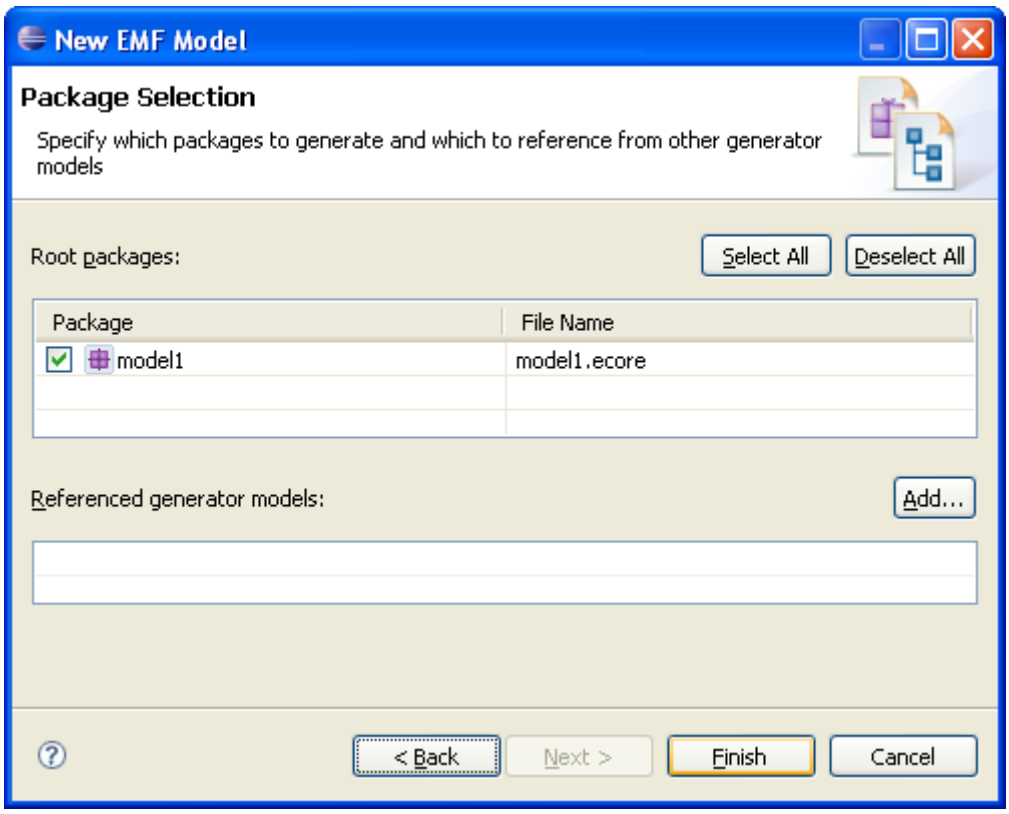
En la pagina siguiente, *Select a Model Importer*, seleccionaremos la opción Ecore model (CDO native).



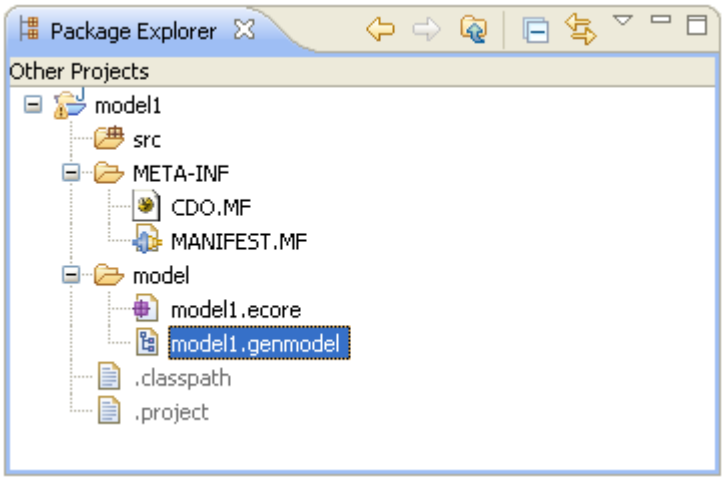
En la siguiente página, *Ecore Import*, aquí es donde se le indica el modelo .ecore, al hacerlo desde el botón derecho sobre el ya vendrá indicado y solo tendremos que hacer clic en el botón *Load*:



En la siguiente pagina, *Package Selection*, seleccionamos los paquetes raíces del modelo y importamos modelo que puedan hacer referencia.

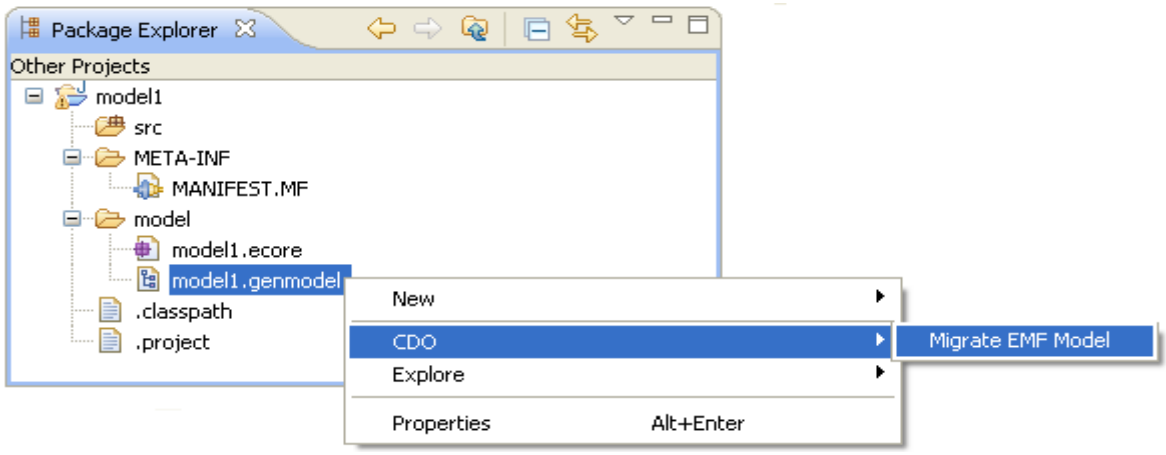


Para finalizar haremos clic en el botón Finish, nuestro proyecto debe de quedar de la siguiente manera:

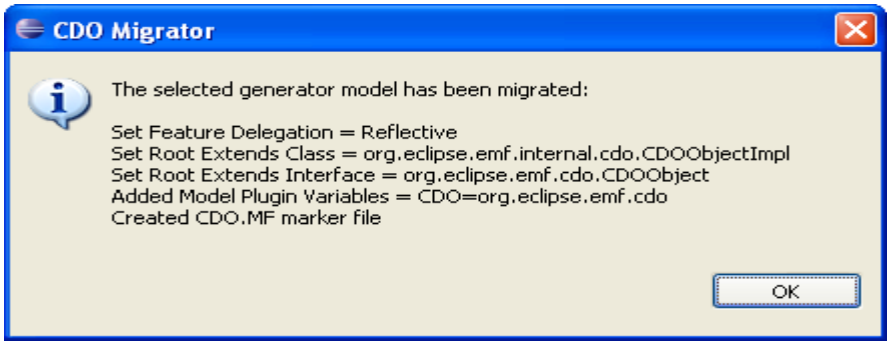


Migrar el modelo CDO

Antes de generar el código de nuestro modelo, tendremos que migrarlo para ello tendremos que hacer clic con el botón derecho del ratón sobre el .genmodel **CDO -> Migrate EMF Model**.

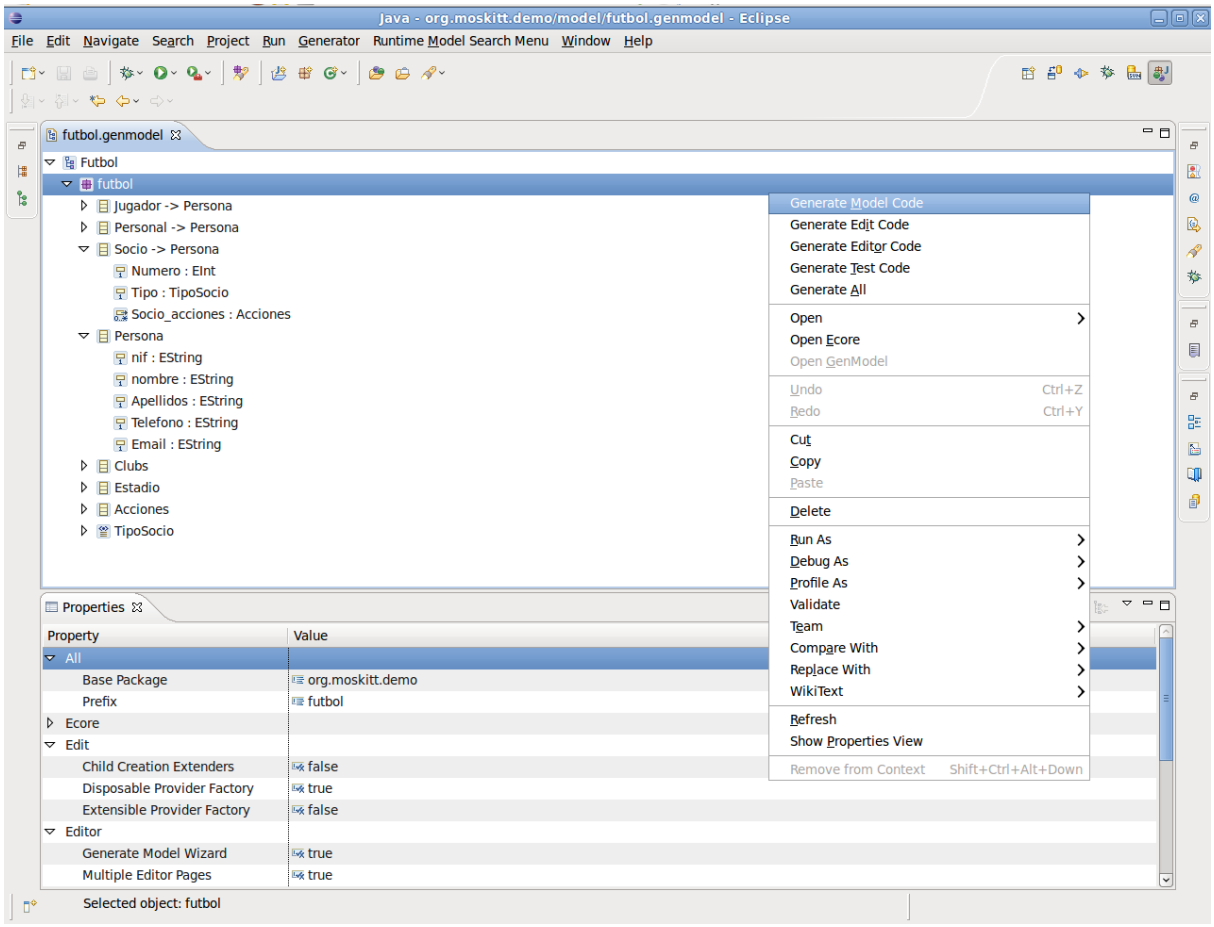


Si todo ha ido correctamente aparecerá un cuadro como el siguiente:



Generar el código del modelo Ecore

Llegamos al paso final es hora de generar el código que sustente el modelo ecore creado, para ello solo tenemos que abrir el .genmodel y sobre el paquete principal clic con el botón derecho, ahí tendremos las opciones para generar código del modelo, edit del modelo y editor del modelo así como test del modelo creado.



A parte del código del modelo, también genera 3 plugin extra mas, el primero es el edit, este plugin se encargar de dar etiquetado a todos los elementos del ecore creado. El otro plugin que genera es el editor, este genera un editor básico en forma de árbol para poder generar elementos del ecore. Por ultimo el plugin test genera código para realizar test en el editor de manera que se puedan generar todos los elementos del ecore correctamente.

7.4 Creación de un Editor FMF para MOSKittDataStore

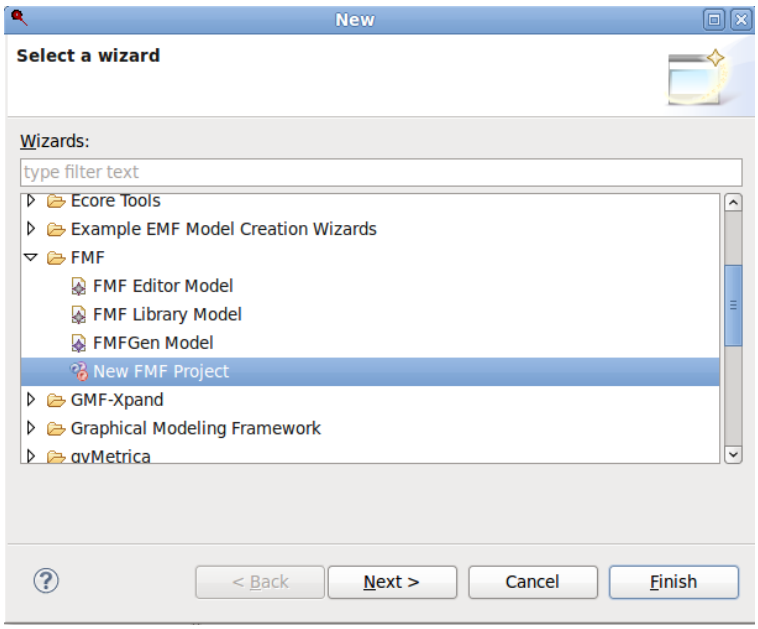
Este manual parte de la generación del editor por defecto basándose en un modelo EMF adaptado a CDO comentado en el manual anterior.
Lo que se pretende en este manual es partiendo del manual anterior, construir nuestro propio editor de formularios basándonos en la tecnología FMF y además adaptar nuestro editor para que funcione correctamente en la vista MOSKittDataStore. Esto quiere decir que este manual se dividirá en dos bloques: construcción de un editor de formulario basado en FMF y adaptación de nuestro editor a la vista MOSKittDataStore.

7.4.1 Construcción de un editor de formulario basado en FMF

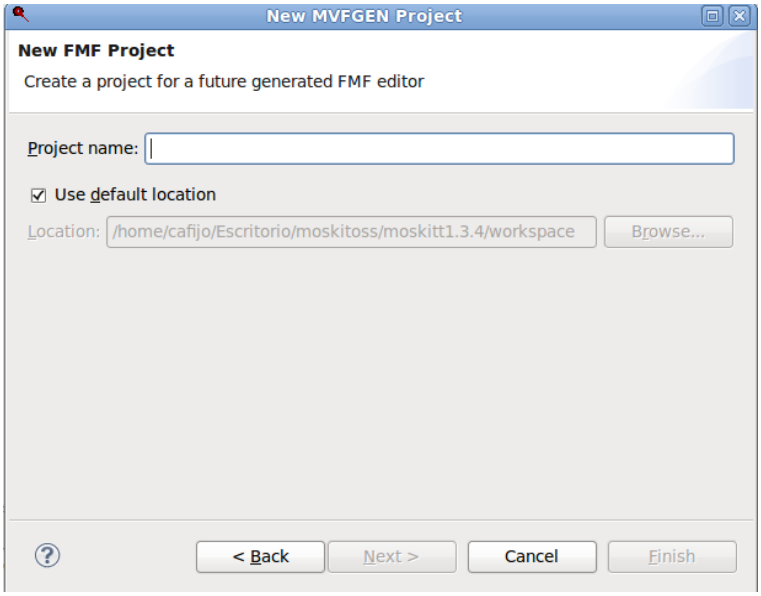
Esta parte del manual se puede dividir en tres partes: Creación de un nuevo proyecto FMF, Construcción del editor usando FMF y generación del código de nuestro editor.

Creación de un nuevo proyecto FMF

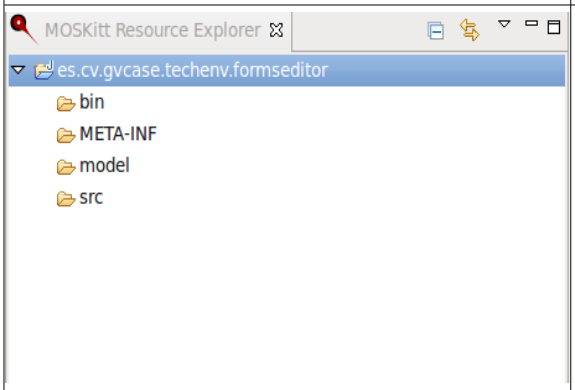
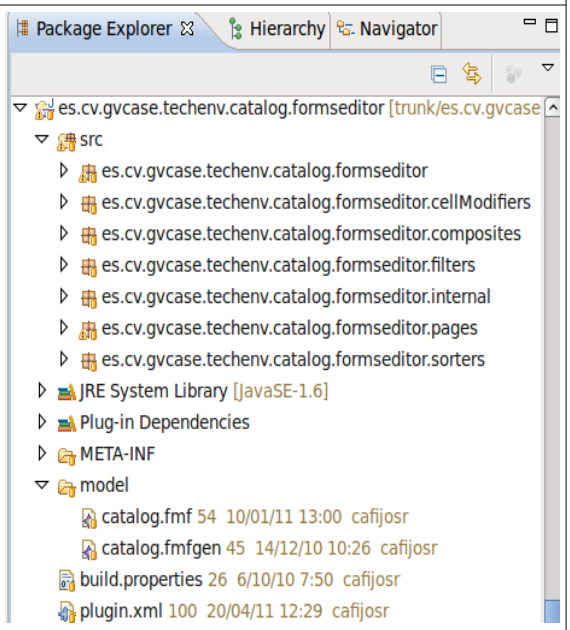
Para crear un nuevo proyecto solo tenemos que ir a File → New → Other, desde ahí seleccionar la carpeta FMF y seleccionamos New FMF Project.



Una vez seleccionado hacemos click sobre next para continuar con el proceso, en la siguiente pantalla nos pedirá el nombre de nuestro proyecto.



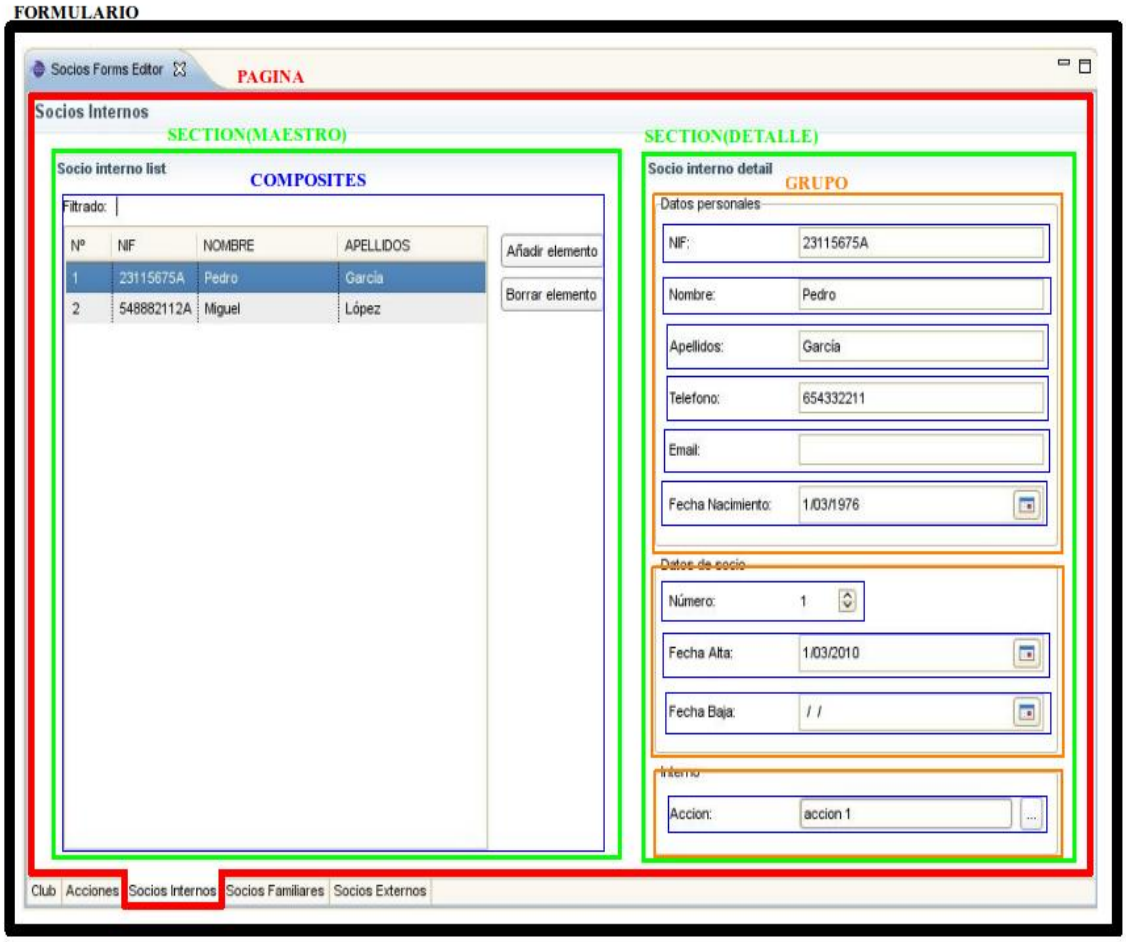
Al introducir el nombre finalizaremos, con esto nos creara un nuevo proyecto vacío de FMF, un proyecto vacío de FMF se diferencia de uno normal por la carpeta model. En la carpeta model situaremos los ficheros de creación (*.fmf) y los ficheros de generación del código (*.fmfgen) de nuestro futuro editor.

Vista de un proyecto FMF vacío	Vista de un proyecto finalizado
	

Se puede observar en el proyecto finalizado que la carpeta model contiene un modelo fmf y el modelo de generación de código de fmf. A continuación se describirá la creación de del modelo fmf.

Construcción del editor usando FMF

Una vez creado el proyecto vacío siguiendo las instrucciones anteriores empezamos la parte del modelado de nuestro editor de formulario. Antes de empezar a entrar en detalle debemos de tener claras las distintas partes dentro de un editor de formulario.



En la imagen anterior se muestran las diferentes partes por las que esta compuesta un editor, un editor esta compuesto por diferentes paginas en la imagen anterior vemos como en concreto este editor contiene 5 paginas. Una pagina puede contener una o mas secciones, lo normal es que tenga dos secciones un maestro y un detalle, la sección maestro es donde se encuentran aquellos elementos que hemos creado para pagina y la sección detalle mostrara el detalle de cada objeto del maestro. Una sección puede contener, grupos, composites y otras secciones , además los grupos pueden contener otros grupos y composites.

En el siguiente diagrama se puede observar padres y hijos de los elementos de un editor formulario.

1 FORMULARIO

1.1 PAGINAS

1.1.1 SECCION

1.1.1.1 SECCION

1.1.1.2 GRUPOS

1.1.1.2.1 GRUPOS

1.1.1.2.2 COMPOSITES

1.1.1.3 COMPOSITES

Teniendo claro este orden dentro solo nos falta conocimientos de GridLayout y LayoutData para poder empezar a editar nuestro modelo fmf.

GridLayout

El GridLayout nos permite especificar el numero de columnas que que creara un elemento(Page, Section, Group y Composite) para disponer de otros elementos en su interior, además de si dichas columnas tendrán o no el mismo ancho. Para ello incorporan dos propiedades dentro del modelo fmf.

1. Columns: Indica el numero de columnas que creara para disponer los elementos.
2. Same Size: Indica si las columnas tendrán el mismo tamaño.

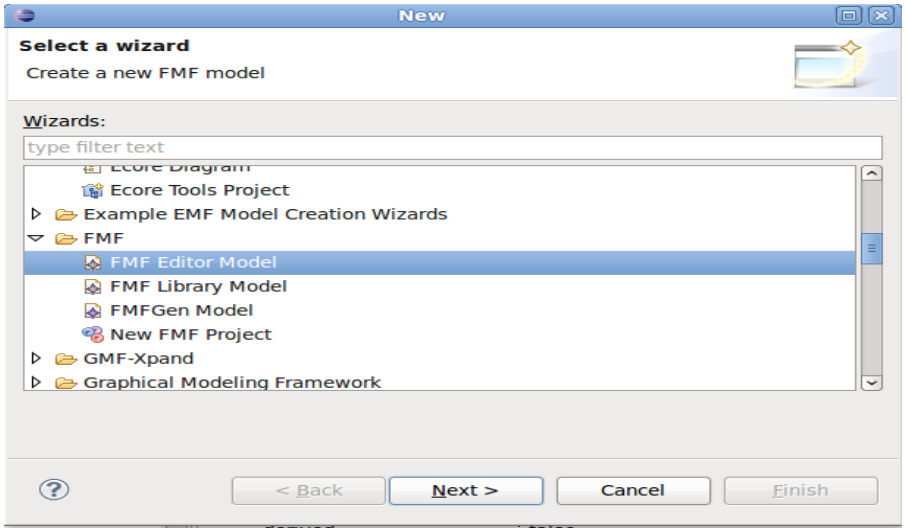
LayoutData

El LayoutData nos sirve para situar los elementos dentro de su padre. Se le podrá indicar de las columnas creadas por el padre cuantas ocupare, tanto a nivel horizontal como a nivel vertical, además de su disposición dentro de esas columnas (FILL, BEGINING, CENTER, END) tanto en horizontal como en vertical. Para ello incorpora las siguiente propiedades dentro del modelo fmf.

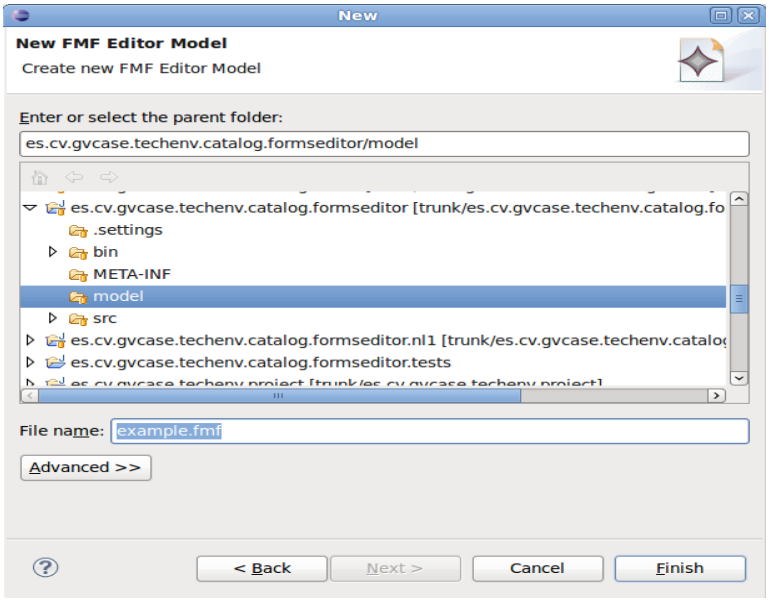
1. Grab Horizontal: Indica (true o false) si el elemento ocupara toda la columa en horizontal.
2. Grab Vertical: Indica (true o false) si el elemento ocupara toda la columa en vertical.

3. Horizontal Span: Indica el numero de columnas en horizontal que ocupara.
4. Horizontal Style: Indica el estilo del elemento en horizontal.
5. Vertical Span: Indica el numero de columnas en vertical que ocupara.
6. Vertical Style: Indica el estilo del elemento en vertical.

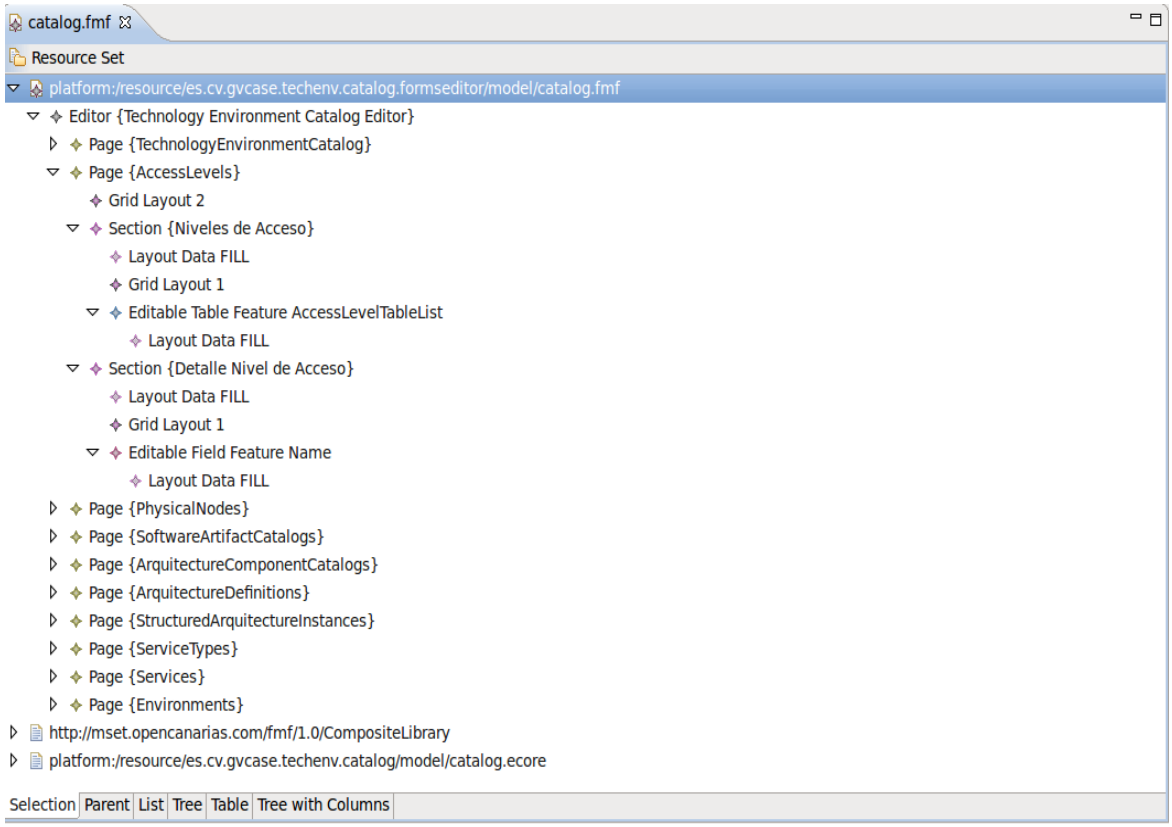
Una vez claro estos conceptos, ya solo tenemos que empezar a crear nuestro modelo fmf, para ello sobre la carpeta models de nuestro proyecto haremos click con el botón derecho new → other → FMF → FMF Editor Model



A continuación insertamos un nombre y finalizamos.

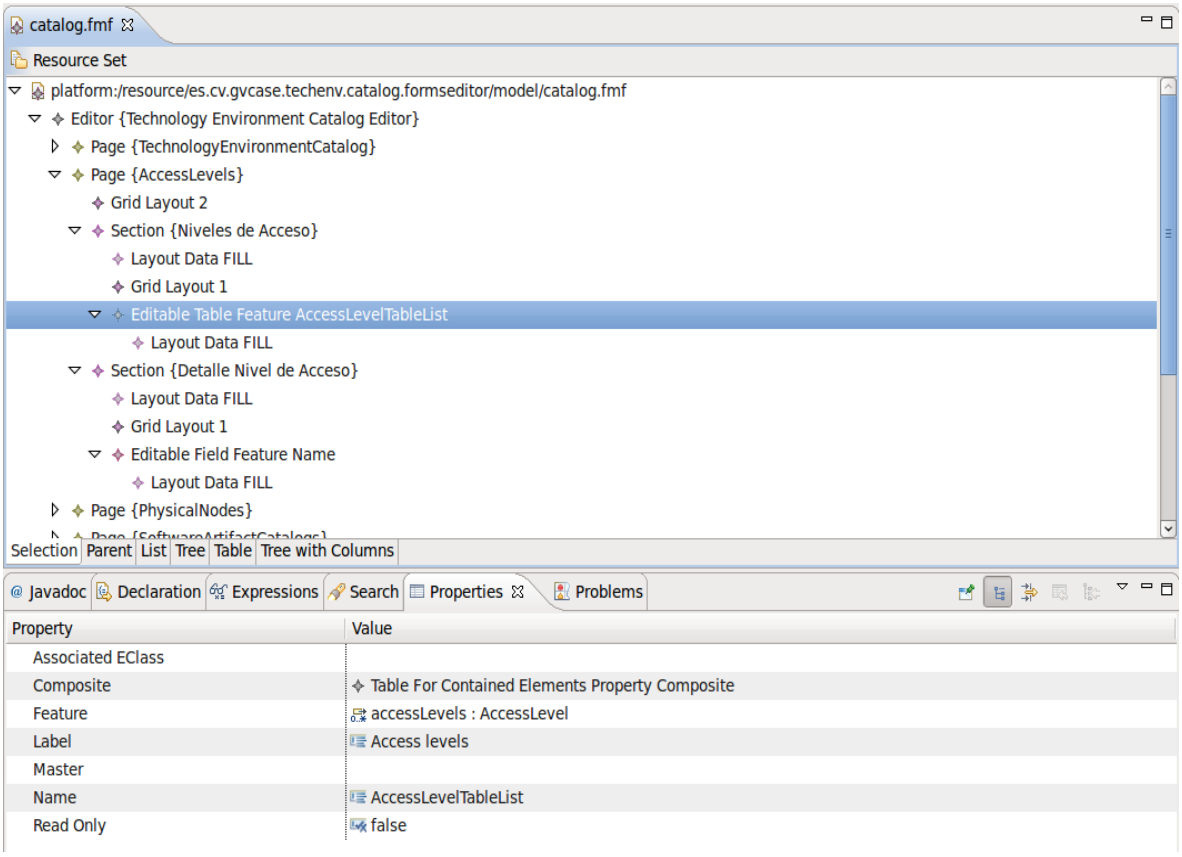


Una vez creado el nuevo modelo, debemos de cargar el modelo EMF ecore (representación del diagrama de clases de mi editor) realizado anteriormente, para ello click con el botón derecho sobre el modelo y pulsamos en *Load Resource*, una vez ahí seleccionaremos la ruta de nuestro modelo EMF creado anteriormente. Cuando lo tengamos cargados empezaremos el moldeado de nuestro propio editor, a traves del modelo FMF.



En la imagen anterior tenemos un ejemplo de moldeado de nuestro propio editor, como podemos observar el modelo ecore de nuestro editor esta cargado:
platform:/resource/es.cv.gvcase.techenv.catalog/model/catalog.ecore

Además podemos ver que el editor tiene un nombre en este caso Technology Environment Catalog Editor y además estará compuesto por 7 paginas (Page), con su nombre respectivamente. La siguiente imagen muestra mas detenidamente una de las paginas AccesLevel.



Vemos que la pagina AccesLevel estará compuesta por dos columnas, en la primera columna contendrá la Section{Niveles de Acceso} esta sección tiene una columna y su disposición es FILL, también contiene un composite, en este caso es una Editable Table Feature, si examinamos un poco las propiedades tenemos los siguientes campos:

Associated EClass: Se le puede indicar la EClass a la que va asociada.
Composite: Tipo de composite que se va a construir
Feature: Feature(propiedad de la EClass que representa) que representa el

composite.

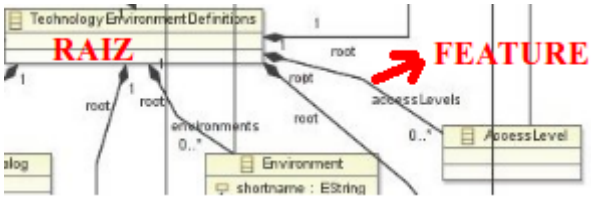
Label: Label del elemento

Master: Se le indica cual es su master en el caso de que exista.

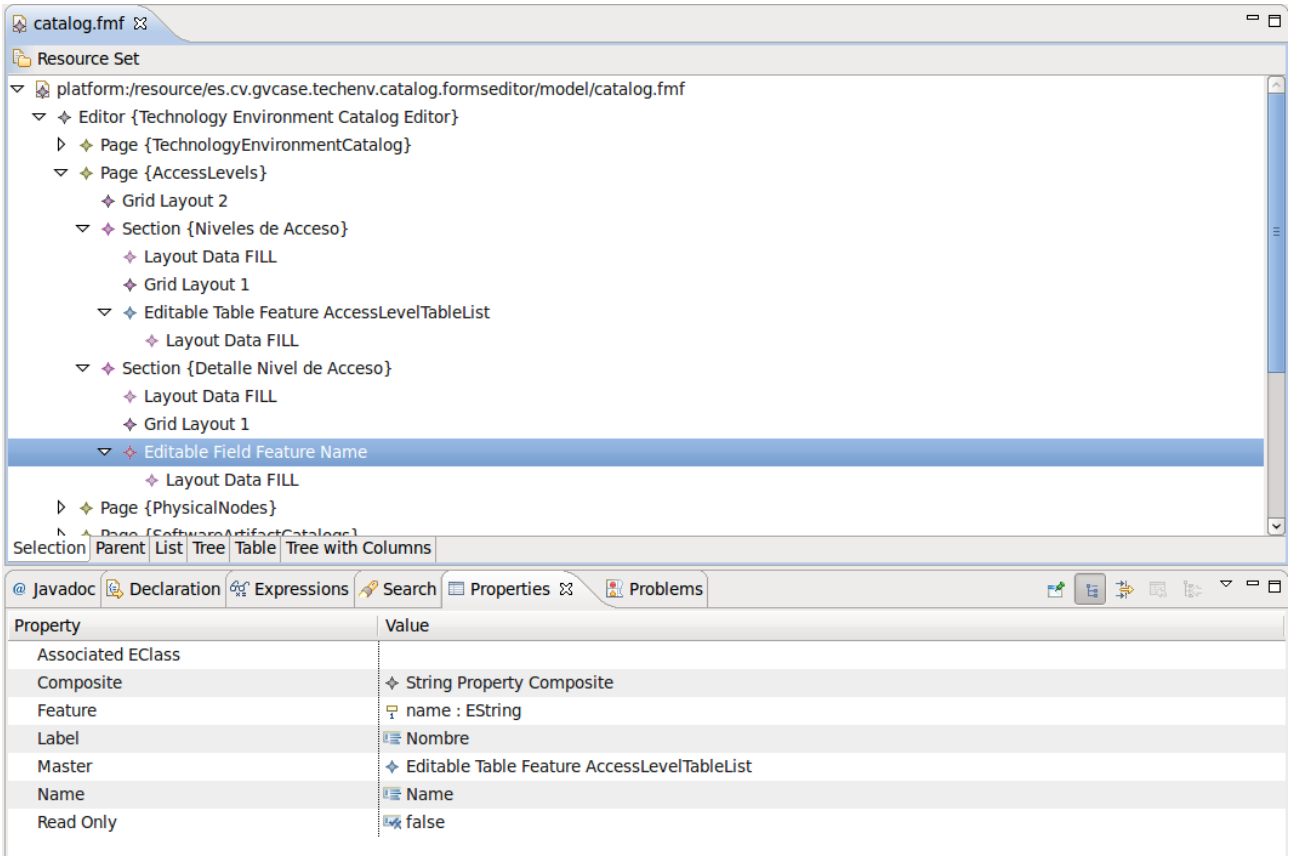
Name: Nombre del elemento

Read only: Indica si es solo lectura el composite.

En este ejemplo no se le indica la EClass Asociada ya que es una feature del elemento raíz:



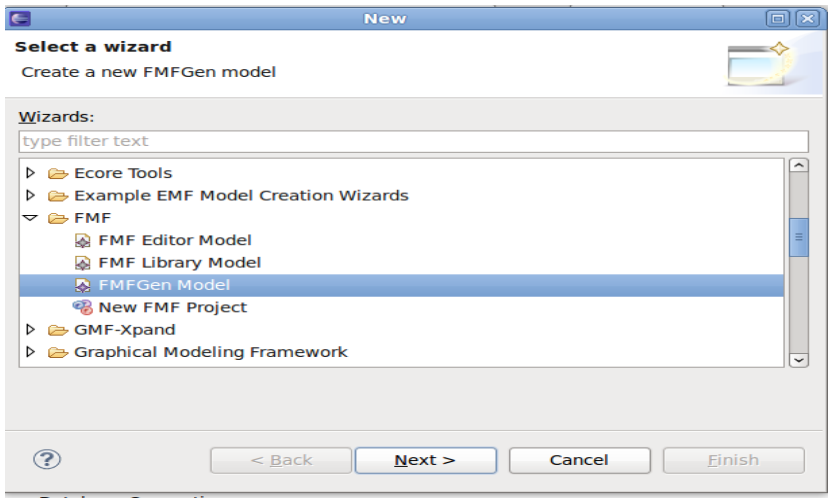
En la segunda Section{Detalle Nivel de Acceso} vemos que esta compuesta por una columna y que la sección esta dispuesta en FILL, vemos también que contiene un composite, en este caso es una Editable Field Feature, viendo las propiedades:



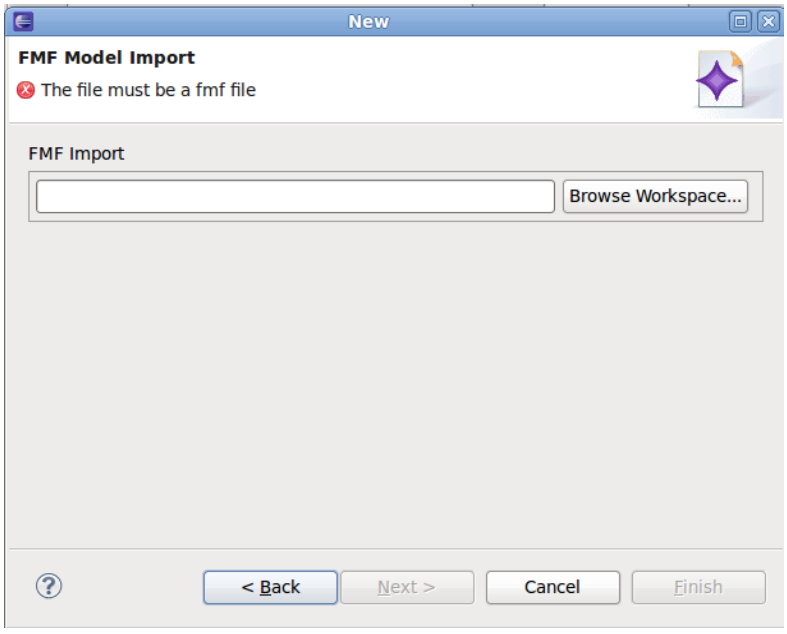
Vemos que representa un elemento esclavo del maestro que sera la tabla, es decir, el composite que represente el nombre dependerá del elemento seleccionado en la tabla anterior.

Generación del código del editor usando FMF

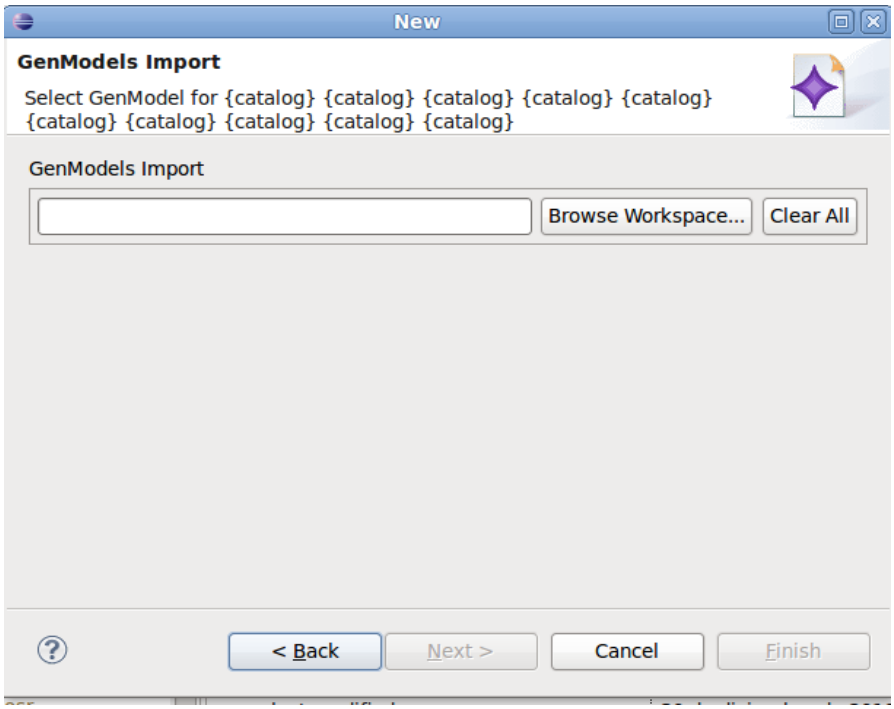
Una vez finalizado de editar el modelo FMF, nos preparamos para la generación del código para ello crearemos un modelo de generación FMF. New → other → FMF → FMFGenModel



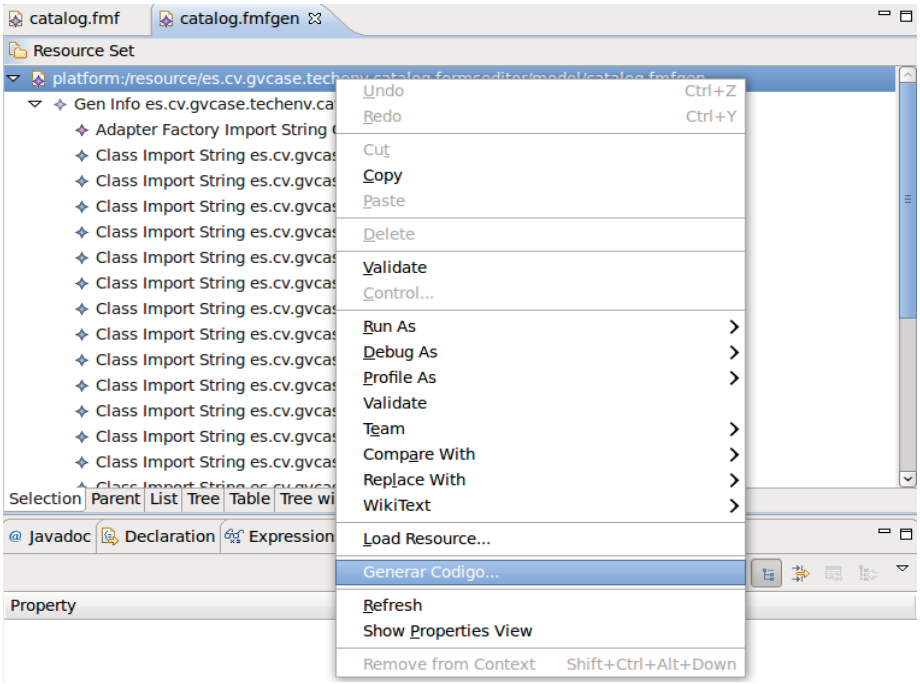
Continuamos con el proceso de creación del FMFGen model y nos pide que insertemos el modelo fmf anteriormente creado.



Se lo indicamos y continuamos el siguiente paso nos pide que insertemos el GenModel del modelo ecore de nuestro editor, creado en el apartado anterior.

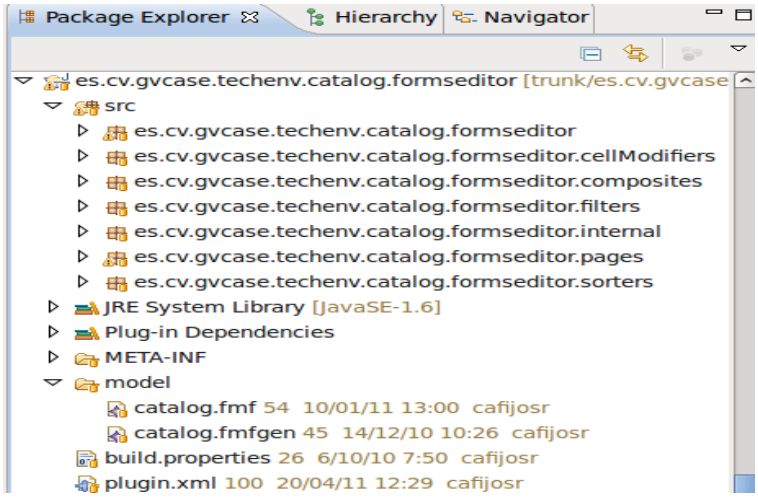


Una vez hecho esto le damos a finalizar y ya tendremos creado nuestro FMFGen model ahora solo faltaría lanzarlo para que genere el código de manera automática. Para ello botón derecho sobre el modelo FMFGen y después a Generar Bodigo.



Con esto y el la carpeta src de nuestro modelo generaría el código Java necesario de nuestro editor.

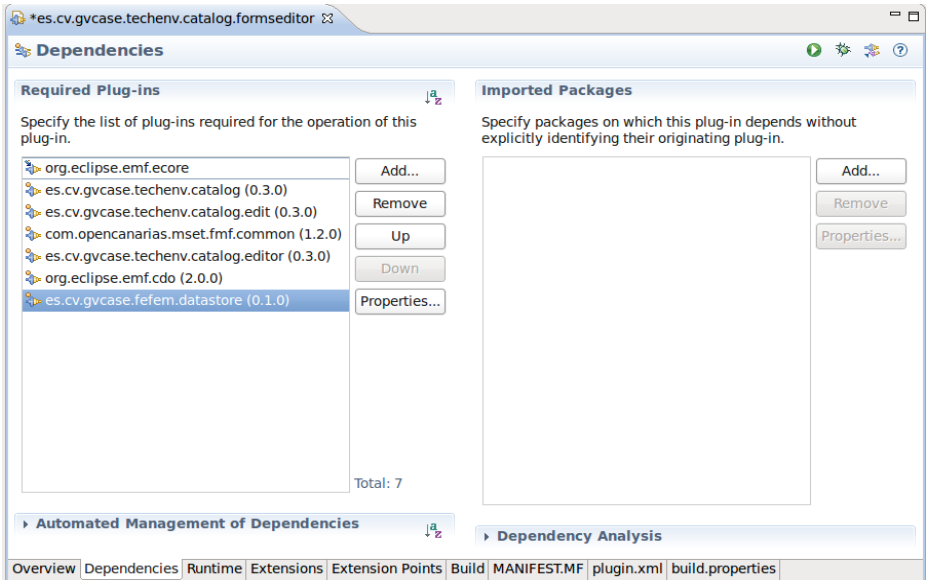
Cada una de las paginas son generadas en una clase java distinta, esta son almacenadas en el paquete *.pages.



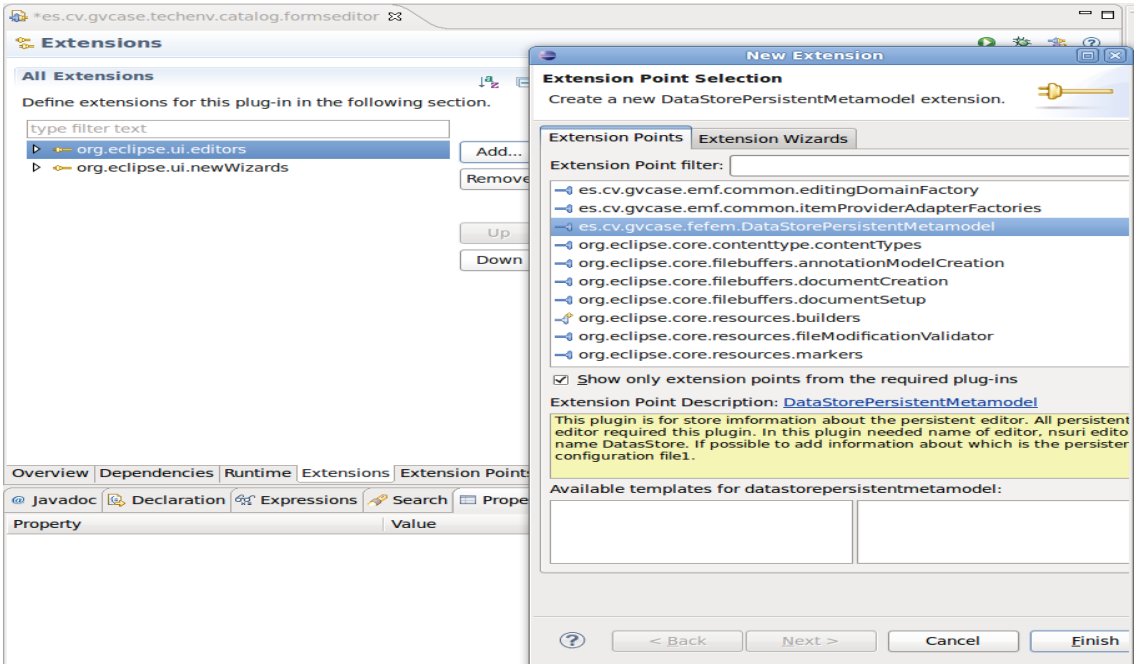
7.4.2 Adaptación del editor a la vista MOSKittDataStore.

En este manual lo que se va a presentar es una vez realizado nuestro propio editor que es lo siguiente que debemos de hacer para que la vista MOSKittDataStore lo reconozca, es decir, para que la vista MOSKittDataStore sea capaz de crear recursos del tipo del editor y sea capaz de abrir los recursos con su editor correspondiente.

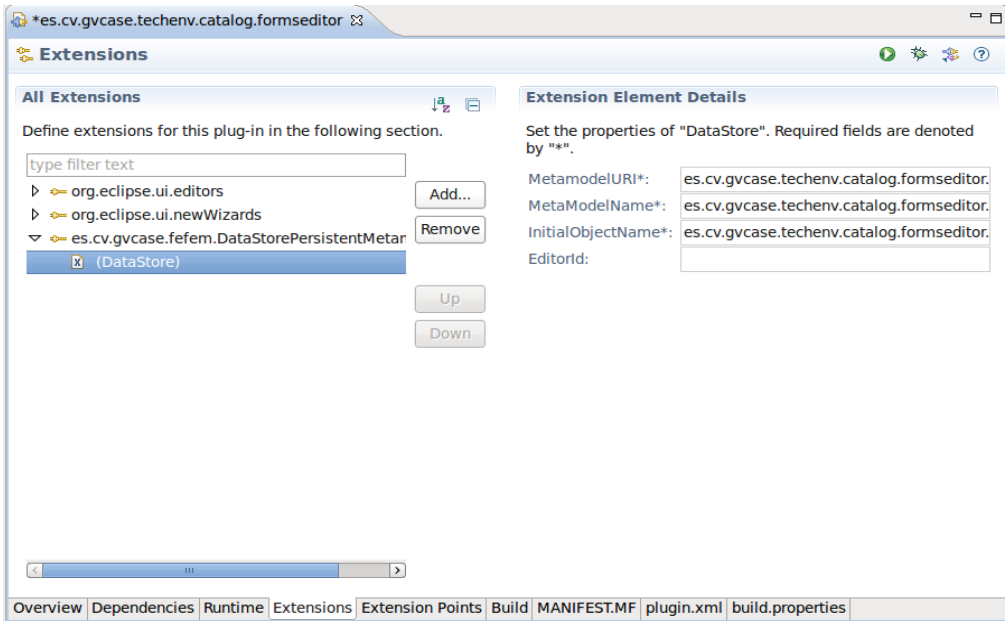
Para ello lo primero que tenemos que hacer es crear una dependencia sobre la vista MOSKittDataStore, para ello nos vamos al fichero plugin.xml de nuestro proyecto y en la pestaña Dependencies añadimos la dependencia a dataStore *es.cv.gvcase.fefem.datastore*



Una vez añadida la dependencia, nos situamos en la pestaña Extensions y añadimos una extensión a la vista, es.cv.gvcase.fefem.DataStorePersistentMetamodel.



Una vez añadida la Extension dentro de ella creamos un hijo llamado DataStore, ese hijo contendrá información sobre nuestro editor que debemos completar para que la vista conozca la existencia de ese nuevo editor y modelo a persistir.



La información requerida es:

MetamodelURI: URI del metamodelo a persistir.

MetamodelName: Nombre con el que se identificara el metamodelo.

InitialObjectName: Nombre del elemento raíz de nuestro modelo

EditorId(opcional): ID del editor con el que abrirá el modelo, en el caso de no proporcionárselo abrirá con el editor por defecto de CDO.

Con todo esto nuestro modelo podrá ser persistido y tendrá la capacidad de la sincronización automática.

8. Pruebas en los editores FMF

Durante el transcurso del proyecto uno de lo objetivos es incluir en el proyecto un sistema de creación y generación de pruebas para aquellos editores de formulario realizados en FMF.

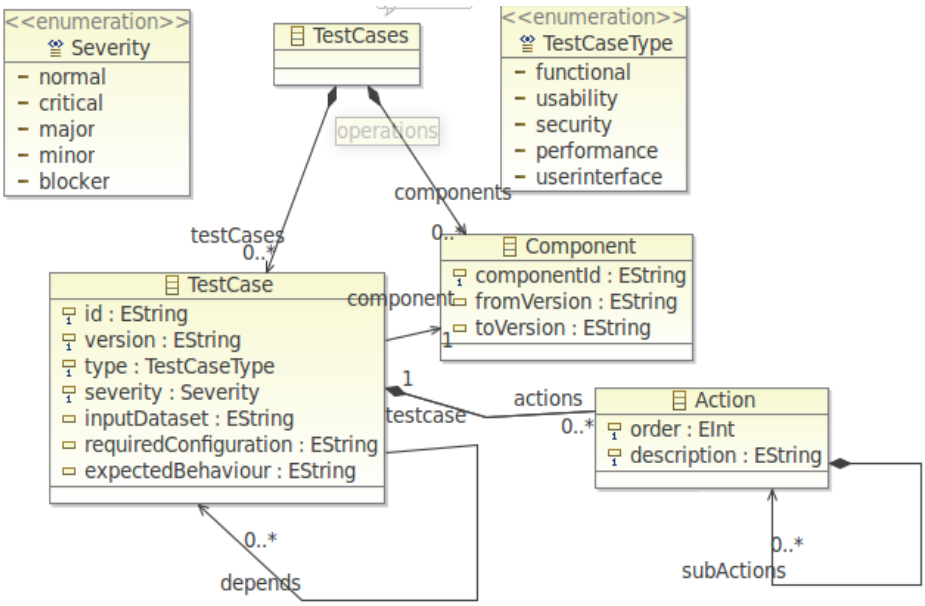
Hoy en día las pruebas automáticas en editores de formularios no esta muy bien lograda, aunque existe diferentes herramientas en el mercado no han logrado el objetivo de lo que requieren las empresas, en esta parte no se va a realizar ninguna herramienta automática que realice las pruebas, lo que se va a implementar es una manera sencilla de generar casos de test sobre un modelo FMF además de una manera de que quede constancia de la realización de los test.

Antes de continuar, se debe de tener conocimientos de transformaciones de modelo en MOSKitt. La transformación de modelos es una parte clave en la Ingeniería Dirigida por Modelos o Model Driven Engineering (MDE). Se define como la generación automática de un modelo a partir de otro modelo fuente, de acuerdo a unas reglas de transformación definidas. Dichas reglas especifican como se transforman los elementos del modelo fuente al modelo final.

Lo que se va implementar es dado un modelo FMF se transforme en un modelo de Casos de Test, además de un editor de formulario donde cargado un modelo de Casos de Test se capaz de gestionar la realización de las pruebas.

8.1 Implementación

Antes de proceder en la construcción de la transformación necesitamos el modelo destino a transformar, el modelo debe de contener implícita alguna manera de guardar los distintos casos de test así como las futuras versiones de un mismo producto, recogiendo estos datos se diseña el siguiente modelo.



El modelo tiene un elemento raiz llamado TestCases que contendra una serie caso de test(TestCase) sobre un componente. Un caso de test viene identificado por un ID, que identificara el caso de test. Un caso de test tiene como propiedades una versión, el tipo de testCase (TestCaseType), la severidad del test (Severity), así como información de algún dato de entrada, algún tipo de configuración previa y el resultado esperado del caso. Un caso de test puede depender de que se hagn primero otros test, esw decir que no se puede ejecutar si no se han realizado primero previamente otros, eso esta simbolizado en el modelo mediante la relación de dependencia. Además para realizar un test se requiere seguir unos pasos en detalle, para ello un testCase tendrá una serie de acciones a realizar en orden, estas acciones pueden conllevar a subacciones.

Ya tenemos el modelo final, ahora solo nos falta implementar la transformación, en este caso sera una transformación ATL, que dado un modelo FMF de entrada genere un modelo de salida TestCase. Antes de empezar a generar codigo en ATL debemos de tener claro las pruebas que se han de probar. Como hemos visto anteriormente el modelo FMF esta dividido en Paginas, secciones, composites entonces se ha de definir pruebas para cada uno de los elementos.

- Para cada página del editor:
  - Comprobar que el título especificado en el modelo FMF es aquel que aparece en la pestaña.
  - Para cada sección:

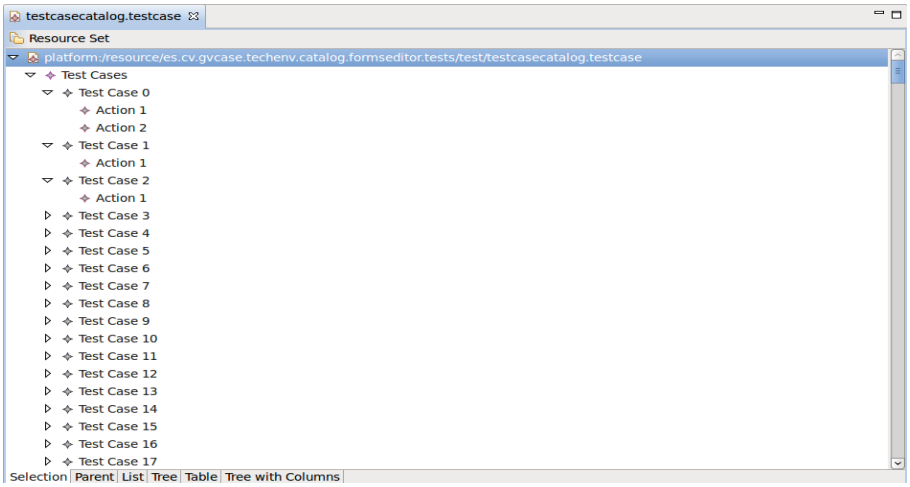
- Que existe en el editor
- Que sus propiedades visuales se muestran correctamente.
- Que su disposición en la página corresponde a la especificada en FMF.
- Para cada Composite:
  - Que existe en la sección
  - Que su disposición en la sección corresponde a lo especificado en FMF.
  - Que su etiqueta coincide con al especificada en FMF.
  - Que su valor de solo lectura coincide con al especificada en FMF.
  - Que el tipo de Composite especificado en FMF es realmente el que se representa en el editor.
  - Que los elementos (datos) que maneja el Composite en el editor son de la misma naturaleza que la feature del composite especificada en FMF.
  - Si la propiedad Master del Composite está especificada en FMF, probar que al cambiar el valor en el Composite “maestro”, también lo hace en el Composite “detalle”.

Una vez definidas las pruebas se van a empezar a construir las sentencias ATL que dado por ejemplo un elemento Page en el modelo de entrada genere un TestCase en el modelo de salida.

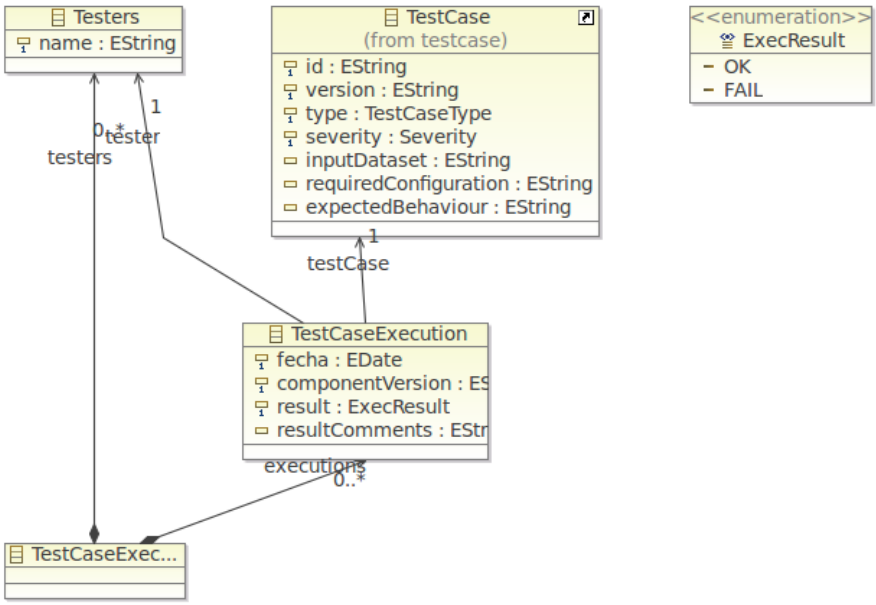
```
-- Porción de coligo de la transformación en ATL (RULE Page2TestCase).
rule Page2testcase{
          from
           i:       FMF!Page (
                              i.oclIsTypeOf(FMF!Page)
                              )
          to
                    o1 : TESTCASE!TestCase (
                              id <- thisModule.cont.toString(),
                              version <- '1.0.0',
                              type <- #userinterface,
                              severity <- #blocker,
                              expectedBehaviour <- 'Comprobar que existe la pagina con el nombre ' + i.name+ ' y esta
situada en la posicion ' +  thisModule.contpage.toString() ,
                              component <- i.getComponent(),
                              depends <- Sequence{thisModule.resolveTemp(i.eContainer(), 'op')},
                              actions <- Sequence{ac0}
                    ),
                    ac0 : TESTCASE!Action(
                              order <- 1,
                              description <- 'Doble click sobre el modelo ' + i.getEditor().name +' creado con el caso de
prueba ' + i.getInitialTestCase().id
                    )
          do {
                    thisModule.cont <- thisModule.cont + 1;
                    thisModule.contpage <- thisModule.contpage + 1;
                    thisModule.root.testCases <- thisModule.root.testCases.append(o1);
          }
}
```

En el codigo anterior se especifica que dado en el modelo de entrada un elemento FMF!Page nos genere TESTCASE!TestCase que consista en verificar en el editor que la pagina existe con un nombre y que esta en la posición correcta.

Lanzada la transformación con el modelo de FMF de nuestro editor obtenemos el siguiente modelo:

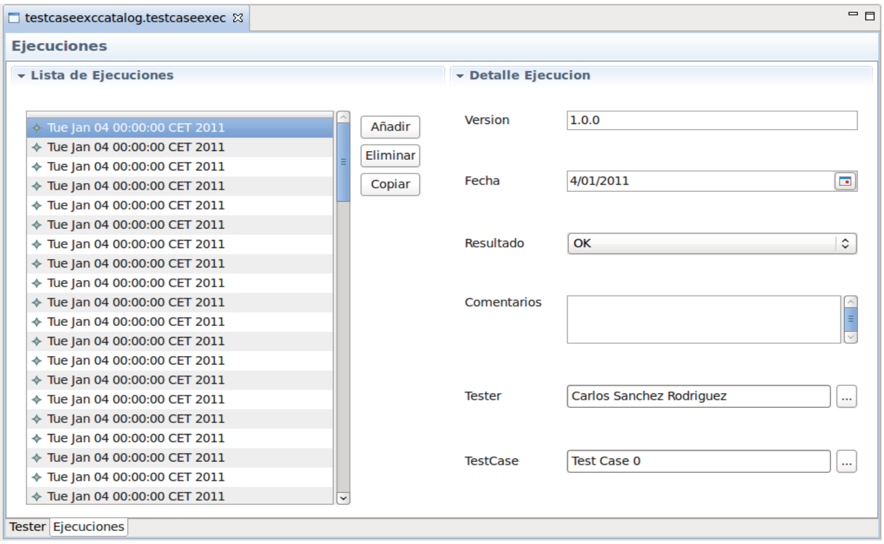


Se ha generado un modelo TestCases con bastantes Casos de test y sus respectivas acciones, ahora tendría un tester probar una a una cada una de los distintos casos de test, para facilitar el almacenado de los resultados del test así como la realización de las pruebas se crea un editor capaz de almacenar dicha información. El modelo del editor para que almacene los datos sera el siguiente.



El elemento raíz del modelo es TestCaseExec que contendrá una lista de ejecuciones de casos de test y una serie de testers. Una ejecución de test tiene como propiedades la fecha en la que se realiza el test, la versión del componente, el resultado de la ejecución así como un espacio para los comentarios de la ejecución. Toda ejecución de test tendrá asignado un tester además de un Caso de Test.

Utilizando el modelo anterior y la tecnología FMF en la generación de código de editores de formulario, creamos un editor de formulario para que el usuario pueda almacenar de manera simple las distintas ejecuciones de los casos de test anteriormente generados.


testcaseexccatalog.testcaseexec
Ejecuciones
Lista de Ejecuciones
Detalle Ejecucion
Tue Jan 04 00:00:00 CET 2011
Añadir
Eliminar
Copiar
Version
1.0.0
Fecha
4/01/2011
Resultado
OK
Comentarios
Tester
Carlos Sanchez Rodriguez
TestCase
Test Case 0
Tester
Ejecuciones

9. Mirando al futuro

Este trabajo fue presentado a miembros de la Conselleria de Infraestructuras y Transporte evaluándolo en el estado actual.

Las conclusiones extraídas fueron que les parecía muy buena idea la integración de CDO al proyecto MOSKitt, pero que actualmente tenia sus limitaciones debido principalmente a la actual versión sobre la que esta desarrollada MOSKitt, Eclipse Galileo 3.5.0

Eclipse Galileo 3.5.0 soporta perfectamente la versión 2.0 de CDO, actualmente esta versión da soporte a la autenticación de cara al servidor CDO y no por recursos. Como comprenderéis eso en una coselleria no es factible. Todo el mundo que tenga acceso al servidor CDO podría recuperar todos los recursos y modificarlos sin consentimiento.

Una mejora que se propone desde conselleria para que el trabajo sea factible es introducir la adaptación de la autenticación sobre los recursos, no únicamente sobre el acceso al servidor como esta actualmente.

Para poder realizar esta adaptación CDO3.0 y superiores poseen de mecanismos de creación de roles, lo cuales pueden adquirir una serie de privilegios para la alteración o no de los recursos. De esta manera se podía especificar que grupos de usuarios son los que pueden alterar el recurso o que grupos de usuarios solo tendrán acceso en modo lectura.

Trabajando sobre esta adaptación hemos llegado a instalar sin problemas sobre la ultima versión de MOSKitt 1.3.5 la herramienta de CDO3.0, todo parecía que iba a resultar bastante facil hasta que me tope con el siguiente bug de CDO.

Bug 303868 - [DataBinding] Caching of old value makes Eclipse-Databinding unusable for CDO. https://bugs.eclipse.org/bugs/show_bug.cgi?id=303868

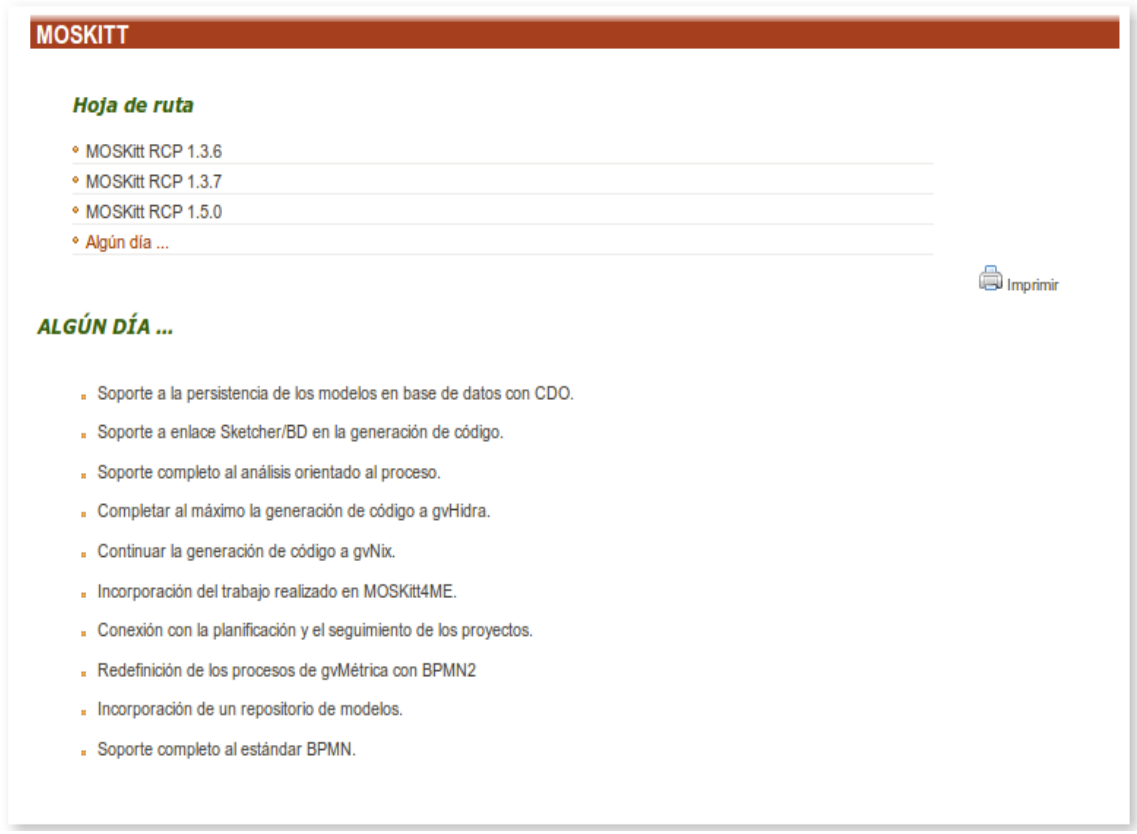
Por lo que hace imposible usar CDO en la versión actual sobre la versión de Eclipse sobre la que esta desarrollado MOSKitt actualmente.

Por lo tanto actualmente el proyecto a quedado paralizado en expectativas de retomarlo para cuando MOSKitt haga su migración a la plataforma Indigo de Eclipse que tiene estimada su salida el día 22 de Junio de 2011.

Lo que si queda claro es que se retomara la adaptación de CDO tal y como dice el

roadmap de MOSKitt.

http://www.moskitt.org/cas/off/road_map-algun-dia/



10. Conclusiones

La conclusión general del proyecto es que gracias a este trabajo se podrán mantener recursos persistidos en bases de datos, lo que facilitara la compartición de estos recursos así como su edición. Este proyecto no solo ha englobado la persistencia de los recursos si no también la creación rápida de editores de formulario basándome en las herramientas actuales que incorpora MOSKitt.

A nivel personal has sido una suerte formar parte del proyecto MOSKitt ya que me han dado los conocimientos y la experiencia necesaria para formar parte de cualquier proyecto. He podido presenciar la manera de trabajar en un proyecto externo a la universidad lo que me ha conllevado a una buena experiencia personal que me servirá en futuros trabajos.

Con respecto a los conocimientos adquiridos en función de proyectos en general, he podido trabajar con herramientas de control de versionado como subversión y herramientas de gestión colaborativa de proyectos como gforge.

He podido comprobar que un proyecto de gran escala como en este caso MOSKitt no puede ser elaborado por una única persona sino que es el conjunto de muchas personas y colaboraciones, donde cada una de las personas son como un eslabón de una cadena donde si se rompe una parte o se queda atrás en algo el resto también se quedan detrás.

Este proyecto también me ha ayudado a ser paciente y perseverante, a no tirar la toalla a la primera de cambio y a saber buscar y leer documentación técnica.

Aunque este proyecto es un paso pequeño hacia lo que me voy a enfrentar fuera de la universidad se que aun me queda muchos pasos por recorrer y mucha experiencia que ganar.

11 Referencias

[1] Clayberg. E , Rubel. D. Eclipse: Building Commercial-Quality Plug-ins. 2nd Edition. Addison-Wesley Professional, 2006. 809p.

[2] Generalitat Valenciana. Conselleria de Infraestructuras y Transporte. MOSKitt [en linea]. [Fecha de consulta: 31 Mayo 2011]. Disponible en : http://www.moskitt.org

[3] Eike Stepper. CDO – Eclipsepedia [en linea]. Actualizada: 12 Mayo 2011.[Fecha de consulta: 31 Mayo 2011]. Disponible en : http://wiki.eclipse.org/CDO

[4] Eclipse Foundation, Inc. Eclipse - The Eclipse Foundation open source community website. Fecha de consulta: 31 Mayo 2011]. Disponible en : http://eclipse.org/

[5] Martin Taal. Teneo – Eclipsepedia [en linea]. Actualizada: 1 Marzo 2011.[Fecha de consulta: 31 Mayo 2011]. Disponible en : http://wiki.eclipse.org/Teneo

[5] Stephan Zehrer. Eclipse Modeling Framework – Eclipsepedia [en linea]. Actualizada: 5 Enero 2011.[Fecha de consulta: 31 Mayo 2011]. Disponible en : http://wiki.eclipse.org/EMF

[6] The Eclipse Foundation. M2M/Atlas Transformation Language (ATL) - Eclipsepedia [en linea]. Actualizada: 5 Enero 2011.[Fecha de consulta: 31 Mayo 2011]. Disponible en : http://wiki.eclipse.org/ATL