EDITOR DE LEARNING DESIGN

PROYECTO DE SISTEMAS INFORMÁTICOS

CURSO 2005 – 2006

FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Profesores Directores:

Baltasar Fernández Manjón
Iván Martínez Ortíz

Autores:

Marco Antonio Hernández Gómez
Esther Picó Sanchis
Laura Rodríguez Sánchez

ÍNDICE

Resumen en castellano

Learning Design trata de modelar la interacción del profesor con los alumnos dentro de un entorno de aprendizaje. Este modelado se lleva a cabo mediante la creación de una Unidad de Aprendizaje, en la que se definen un conjunto de actividades que deben llevar a cabo los alumnos y profesores por separado y en conjunto.

El Graphical Editing Framework (GEF), consiste en un editor gráfico que utilizamos para modelar el diagrama de actividades. Una vez que tenemos el modelo, utilizamos UML2 junto con EMF para convertir este modelo en datos XML y lograr así su persistencia. A partir de éste, con JDOM, construimos un documento XML que se ajusta al patrón de Learning Design que utilizan otras herramientas como ReloadLD Editor.

Finalmente construimos una aplicación independiente de Eclipse con las herramientas que nos proporciona la Rich Clent Platform (RCP).

Abstract

Learning Design tries to model the interaction between the teacher and the students within a learning environment. This modelling is carried out by creating a Learning Unit, in which a set of activities are defined and carried out by the students and teachers separately and altogether.

The Graphical Editing Framework (GEF), consists of a graphical editor that we use to model activity diagrams. Once we have the model, we use UML2 with EMF to turn this model into XML data and to obtain its persistence. From the XML data, using JDOM, we construct a XML document that adjusts to the pattern of Learning Design which is used by other tools like ReloadLD Editor.

Finally, we construct an Eclipse independent application with the tools that Rich Clent Platform provides (RCP).

Especificación de Learning Design

Introducción

Para que un ordenador sea capaz de entender el lenguaje que describe el Learning Design, éste debe tener una sintaxis y una semántica concretas. Y eso nos lo ofrece su especificación.

Antes de ver la especificación aclaremos que significa la frase "**Learning Design**". "Learning Design" se traduce en nuestro idioma como "Diseño de Aprendizaje" y se deriva de otra expresión muy escuchada y conocida "**Instructional Design**" (Diseño Instruccional). Learning Design es el modelado de unidades de estudio.

El uso de la palabra *learning* ayuda a enfatizar la variedad de métodos y mecanismos que van más allá de "enseñar o impartir conocimiento" que está más asociado con el diseño instruccional.

Un **Diseño de Aprendizaje (Learning Design)** está definido en la especificación IMS como "*una descripción de un método que permite a los alumnos alcanzar ciertos objetivos por medio del desarrollo de unas actividades en un orden determinado, en el contexto de un ambiente concreto de aprendizaje*".

El lenguaje EML (*Educational Modelling Language*), desarrollado por la *Open University of the Netherlands' (OUNL*), permite especificar la estructura de una Unidad de Aprendizaje (*Unit of Learning* o UOL), mediante un documento XML. No olvidemos que XML nació precisamente como un lenguaje para intercambiar datos a través de aplicaciones en internet.

IMS, consciente de las limitaciones pedagógicas de las especificaciones que estaba desarrollando, empezó el proceso de creación de una especificación para la definición de aspectos pedagógicos, pero ya que EML existía y funcionaba decidieron adaptarlo en lugar de crear una totalmente nueva. El resultado es la especificación: IMS *Learning Design*.

El desarrollo de un marco que apoye la diversidad pedagógica e innovación, mientras que se promueve el intercambio e interoperabilidad de los materiales del e-learning, es uno de los principales desafíos en la industria del e-learning hoy por hoy. En la ausencia de una forma estandarizada de describir los procesos de aprendizaje, los diseñadores hoy utilizan HTML o lenguajes propietarios para escribir las secuencias de actividades a ser desarrolladas por los alumnos, para establecer la comunicación, para almacenar los resultados de las interacciones, etc. Sin mecanismos acordados y compatibles que describan las estrategias de enseñanza, los creadores de

materiales de enseñanza y la organización de éstos continúan experimentando dificultades innecesarias en:

- Documentar las estrategias de enseñanza utilizadas en estos materiales o con ellos.

- Apego a procedimientos preestablecidos para asegurar la consistencia de la documentación.

- Asegurar que la calidad de la enseñanza se cumple a través y entre organizaciones.

- Reutilización de elementos de materiales existentes de enseñanza.

IMS LD provee un nivel de abstracción en el proceso, ofreciendo piezas genéricas para diferentes enfoques pedagógicos. Utilizando el lenguaje, los diseñadores pueden conversar en términos más de pedagogía que de tecnología, haciendo, por lo tanto, explícitas las alternativas pedagógicas sujetas a revisión, inspección, crítica y comparación.

Propósito

El objetivo del *IMS Learning Design workgroup's* (LDWG) es trabajar hasta establecer especificaciones para describir los elementos y la estructura de cualquier unidad de aprendizaje. También se pretende permitir que se creen diversos tipos de diseños educacionales, y que éstos sean implementados de la misma forma en una gran diversidad de cursos o programas de aprendizaje.

Beneficios de la especificación de Learning Design

El LDWG tiene en cuenta las especificaciones existentes de IMS e intenta basarse en ellas cuando sea posible, produciendo muchas variantes si es necesario. Para facilitar el desarrollo de la especificación y su implementación correspondiente, el Learnig Design ha sido dividido en tres partes, las que se conocen como Nivel A (Level A), Nivel B (Level B) y Nivel C (Level C). Para cada nivel se ofrecen esquemas XML distintos, y los Niveles B y C se integran y extienden a nivel previo como explicamos más adelante.

A quién va dirigida

¿Para quién está creada ésta especificación? Como se puede imaginar, la especificación es un documento muy detallado enfocado para que los

desarrolladores de software puedan crear los sistemas y herramientas que implementa LD. Sin embargo, también está pensado para que puedan entenderlo quienes tienen que beneficiarse de él. El formato XML normalmente no debería ser visible, sin embargo en ocasiones es necesario trabajar directamente sobre él, sobre todo cuando es necesario mover contenidos entre las plataformas. Hay que tener en cuenta que, si bien, IMS es algo que ya está funcionando, aun carece de la solidez de un estándar y una facilidad para ser manipulada por usuarios finales. Precisamente mejorar eso es el objetivo de nuestro proyecto.

Pese a todo, el uso de ésta especificación intenta ofrecer una serie de beneficios para los usuarios de e-learning. Por nombrar algunos, podemos hablar de la liberación de los *Instructional and learning designers* a la hora de usar lenguajes para crear procesos de aprendizaje. Ya que podrán hablar en términos de pedagogía en lugar de tecnología, al disponer de un lenguaje específico para ello. También es muy importante destacar los beneficios que aporta el evitar sistemas cerrados ya que los cursos pueden ser exportados como unidades de aprendizaje de *Learning Design (LD Units of Learning)* de un sistema a otro. La búsqueda de la interoperabilidad, con herramientas comerciales y de código libre que permitan lograr los requerimientos del *e-learning* también es un beneficio importante.

El objetivo principal es facilitar la creación de muchos tipos de diseños educacionales, usando una notación consistente, los cuales puedan ser implementados uniformemente en múltiples cursos o programas de aprendizaje.

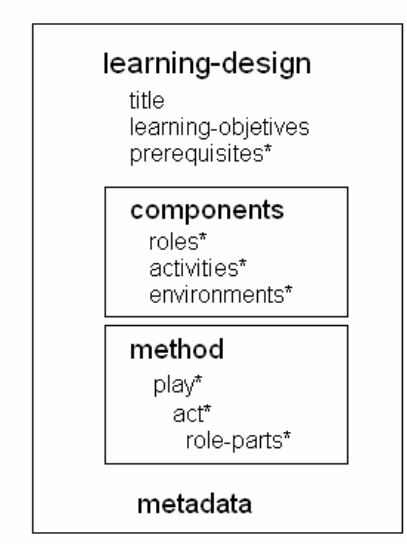
Funcionamiento de las especificaciones

Para comprender cómo funcionan las especificaciones veamos primero de qué se componen las Unidades de Aprendizaje. Ya sabemos que la especificación del LD nos ofrece una serie de elementos que pueden ser usados para describir formalmente el diseño de cualquier proceso de enseñanza-aprendizaje. Una Unidad de Aprendizaje (UOL) hace referencia a una completa unidad de educación o entrenamiento, como puede ser un curso, un módulo, una lección, etc. La creación de una UOL implica la creación de un diseño de aprendizaje, *learning design,* y también las ligaduras de todos sus recursos asociados, como archivos relacionados con la unidad, referencias web… Como resultado de todo esto, es necesario un mecanismo de empaquetamiento para agrupar el *learning design*, con sus archivos asociados en un contenedor simple. La especificación del LD recomienda el uso de la especificación de *Content Package* (CP) para llevarlo a cabo.

Un paquete de contenidos o *Content Package* consiste en una estructura de archivo que incluye un *manifest*, dónde aparecen unos meta-datos que describen el paquete, una estructura del contenido, que puede ser

una simple jerarquía en árbol o la propia especificación del LD y una lista de los archivos que están incluidos en el paquete. Aparte del *manifest,* el paquete incluye los archivos físicos, que pueden ser archivos multimedia, páginas HTML, descripciones de actividades y otros archivos.

Por su parte, el elemento *learning-design* tiene varios componentes, como podemos ver en el siguiente esquema de ejmplo.



(* = puede haber varios de ellos)

Aparte del título (la mayoría de los elementos importantes lo tienen), los objetivos, los prerrequisitos y los metadatos, aparecen los dos elementos más importantes del LD: Los componentes y el método.

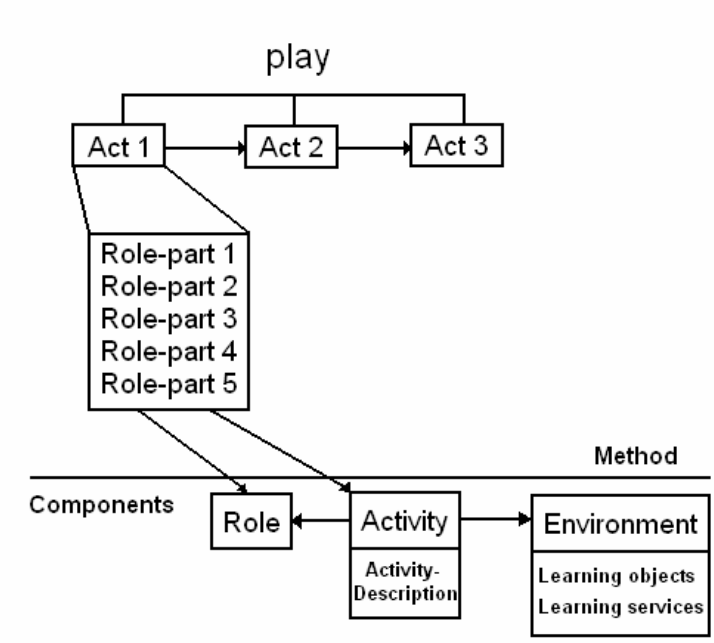
Los *Components* incluyen los tres componentes identificados como los elementos principales del lenguaje: Roles, actividades y entorno (que referencia a los recursos y servicios usados por las actividades).

Un método de learning design (Method) describe un proceso de enseñanza-aprendizaje, por ejemplo el proceso que se emprende por personas interactuando en un entorno de aprendizaje. *Method* soporta el flujo de trabajo o flujo de aprendizaje, *learning-flow*, para el LD, y contiene tres elementos anidados: *play*, que contiene uno o más *acts* y *act* que contiene uno o más *role-parts*.

Mientras que *Components* contiene los principales elementos estructurales, el *Method* dirige todo el proceso cuando un *learning design* se esta ejecutando.

Para entender mejor su funcionamiento vamos a pensar en la analogía con una obra de teatro. Una obra, o *play* en inglés consta de varios actos que se ejecutan en una secuencia determinada, uno detrás de otro. Como para que empiece uno, antes tiene que haber acabado el anterior, la transición entre actos puede ser usada como un punto de sincronización para los múltiples

participantes de un *learning design.* Por supuesto, si esto no es necesario podemos utilizar un solo acto simplemente. De nuevo, una obra de teatro incluye uno o más papeles que aparecen ejecutándose a la vez dentro de cada acto. Estos papeles o *role-parts*, son los que nos indican que es lo que tiene que hacer cada actor. En el contexto del LD, un *role-part* contiene dos links que hacen referencia a un rol y a una actividad, La actividad tiene a su vez un link al rol y otro al entorno, e incluye una descripción que nos dirá lo que tiene que hacer el rol con cada ítem incluido en el entorno. Por último el entorno incluirá *learning objects* (páginas web, *Content Packages*, test QTI, etc), y/o *learning services* que serán usados en la actividad.



Hay que tener en cuenta que si bien, cada acto ofrece un nuevo grupo de actividades y material para todos los participantes, no es necesario que aquellos que no han acabado una actividad se vean forzados a moverse, por defecto, todas las actividades y recursos continuaran estando disponibles. No obstante un diseñador puede crear una condición para que cierta parte del material no sea visible si por ejemplo, el siguiente acto consiste en un test en el que no quiere que se usen referencias. Algo muy importante de entender es la relación que existe entre participantes individuales y roles. A cada participante hay que asignarle un rol o roles, y cada rol puede tener varios participantes. Es lógico pensar en un ejemplo en el que una persona ejecutará el papel de profesor, mientras que tendremos varios estudiantes en el papel de alumnos.

En lo que a l*earning objects* y *learning services* se refiere, hay que señalar que lo que los diferencia es el hecho de que para los primeros, su localización (o URL) es conocida en tiempo de diseño, mientras que para los segundos no. Esto es así porque un *learning service* incluye un mapeado de los roles del LD y generalmente, como por ejemplo en el caso de una conferencia, cada participante tiene diferentes permisos, y en tiempo de diseño

no solo es imposible saber cuántos participantes habrá, sino que éstos cambiarán cada vez que se ejecute la UOL.

Creación de una unidad de aprendizaje

La creación de un Diseño de Aprendizaje generalmente comienza con ideas esbozadas en papel y termina con la producción de un documento XML. Según la Open University of The Netherlands se ha encontrado útil separar el proceso en las siguientes fases:

En la fase de análisis se estudia un problema educacional en concreto, generalmente conversando con los distintos actores. El análisis da como resultado un escenario didáctico que queda capturado en un texto narrativo, generalmente sobre la base de una lista de chequeo (checklist).

Posteriormente, este texto se representa con un diagrama de actividades UML para introducir mayor rigor al análisis. El diagrama UML forma la base del documento XML que seguirá la especificación IMS LD.

Este documento luego se transforma en la base para el desarrollo del contenido (recursos) en la fase de desarrollo.

El paquete de contenidos que incluye a los recursos y al diseño del aprendizaje luego es evaluado y puesto a pruebas.

Es importante destacar que la iteración es una característica natural del proceso.

Reglas de Learning Design

Podemos resumir la estructura de una regla de learning design de la siguiente forma:

Si situación de learning design S (y valor V)
Entonces usar el método M (con probabililidad P)

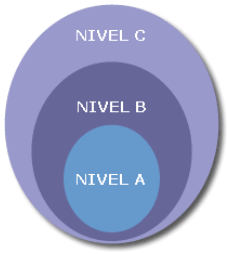
Los elementos entre paréntesis son difíciles de medir.

La pregunta es cómo crear reglas que funcionen, que ofrezcan una alta probabilidad de que los estudiantes realmente alcancen los objetivos de aprendizaje.

Hay tres categorías de buenas reglas:

1. Las derivadas de la teoría del diseño instruccional.
2. Las derivadas de las mejores prácticas (ejemplos).
3. Las derivadas de los patrones de los mejores ejemplos.

Niveles A, B y C de Learning Design



LD tiene tres niveles:

- El nivel A contiene el lenguaje núcleo del LD
- El nivel B añade propiedades y condiciones al nivel A.
- El nivel C añade notificaciones a los niveles A y B.

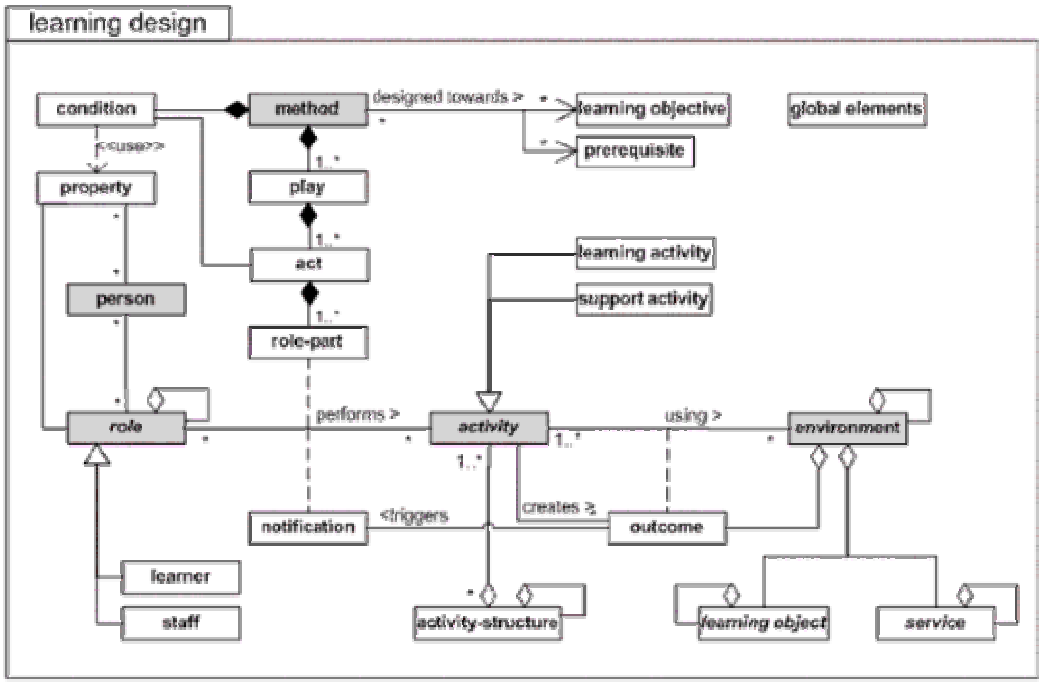
El Nivel B permite la personalización así como secuencias e interacciones más elaboradas basadas, por ejemplo, en los portafolios de los alumnos. Las propiedades pueden ser utilizadas para dirigir las actividades de aprendizaje así también como para registrar los resultados. El último nivel conocido como Nivel C añade *notificaciones (notifications)* al Nivel B. Una notificación se dispara u origina por el resultado de una actividad y puede producir que una nueva actividad quede disponible para el rol que se está ejecutando.

Vamos a ver con un poco más de detalle qué son las propiedades y las condiciones del nivel B. Hay dos tipos principales de Propiedades, Locales y Globales, que pueden ser *General*, *Person* o *Role properties*. Las locales únicamente existen durante la ejecución de una UOL. Las Generales persisten en múltiples ejecuciones de una UOL. Pueden ser utilizadas para diferentes propósitos, pero un uso común es utilizar las *Person Poperties* para obtener una información mas detallada de los alumnos a fin de adaptar un *learning design* a las preferencias y necesidades individuales. Otro uso es mantener la información de estado durante la ejecución de una UOL, y usar esta información para determinar dinámicamente cuándo una acción debe ser disparada. Las Condiciones ofrecen la posibilidad a los diseñadores de definir reglas como qué debería ocurrir cuando un determinado evento tiene lugar. El tipo de evento más simple que puede disparar una regla es la expiración de un tiempo que había sido asignado para realizar una acción específica. Las reglas también pueden dispararse cuando una Actividad, *Role-part*, Acto, Play o incluso toda la UOL ha sido completada.

En lo que se refiere al nivel C, una Notificación proporciona un mayor nivel de interactividad y control. Pueden ser usadas para hacer una nueva actividad visible o invisible a los participantes de un rol, o para establecer valor a una propiedad. Como las reglas pueden ser activadas por cambios en propiedades, establecer un valor a una propiedad que tiene una condición asociada a ella puede disparar otras acciones.

El nivel A ofrece el mínimo nivel de capacidad que necesitan los requerimientos de la especificación, y no presenta ninguna dependencia con el nivel B, ni éste con el nivel C. Sin embargo el nivel B depende y extiende al nivel A, y el C depende y extiende los elementos de los niveles A y B.

A modo de resumen, este esquema presenta las relaciones entre todos los elementos que acabamos de ver.



Expectativas de futuro

Los *Learning Services* nos ofrecen unas posibilidades grandísimas dentro del LD, sin embargo son un área relativamente poco desarrollada tanto en la especificación como en la práctica. Muchos servicios pueden ser añadidos, como chats, mensajería instantánea, clases virtuales y sistemas colaborativos mucho más sofisticados, como diseño virtual de entornos, simulación sofisticada e incluso sistemas de juego multiusuario.

El principal reto ahora es conseguir diseñar todo esto consiguiendo un alto grado de portabilidad entre los distintos compiladores de LD.

Arquitecturas para mantener la calidad y la gestión de contenidos con Learning Design

Flujos de trabajo para Learning Design

El primer paso para diseñar la arquitectura es ver qué tipos de tareas y herramientas necesitan los usuarios para trabajar con Learning Design.

Cualquier paso que implique mantenimiento de la claridad y gestión de contenidos de Learning Design debería incluir la mayoría de estas tareas:

- Restringir la variedad de Learning Designs: Ya que el Learning Design se puede aplicar a un gran rango de escenarios educacionales, al referirnos a un escenario concreto es preferible disminuir el número de herramientas y tareas que se pueden utilizar.

- Creación, edición y almacenamiento de plantillas de Learning Design: Otra forma de crear plantillas es trabajar con ejemplares de modelos concretos expresados en Learning Design. Estas plantillas son ricas y están bien diseñadas, permitiendo que los profesores las modifiquen sin ninguna ayuda. Trabajar con plantillas es un requisito de la arquitectura.

- Creación y edición de Learning Designs: Es necesario para las arquitecturas soportar la creación de nuevos Learning Designs, o bien desde cero, o a partir de uno ya existente o una plantilla. Para ello es suficiente tener en cuenta sus restricciones.

- Edición de la presentación de Learning Designs: Una posible forma es presentar directamente el diseño en XML, usando XSLT para transformar partes del Learning Design directamente a XHTML, Shockwave u otro formato de presentación.

- Descubrimiento y agregación de materiales al Learning Design: Es necesario añadir materiales sin diseño, como objetos de aprendizaje, HTML, imágenes, animaciones, etc. Estos materiales se suelen crear y gestionar fuera del Learning Design y luego necesitamos insertarlos en él.

- Unión de Learning Designs: Las unidades de aprendizaje (UOLs (Units Of Learning)), pueden ser referenciadas por otros Learning Designs. Una de las tareas del diseñador es unir UOLs de distintos Learning Designs en nuevas estructuras.

- Creación, edición y almacenamiento de materiales: La arquitectura que soporta el Learning Design necesita interactuar con la aplicación para el manejo de materiales ya que es necesario poder crearlos, modificarlos y almacenarlos.

- Prueba de Learning Designs: Es necesario probar los Learning Designs para comprobar su funcionamiento y probar las posibilidades del diseño. Dada la flexibilidad del Learning Design existe la posibilidad de encontrar muchos fallos debidos a las combinaciones de roles, actividades y propiedades definidas por el diseñador, por ello es necesario que soporte sofisticadas depuraciones para prevenir problemas a la hora de usar el Learning Design.

- Almacenamiento de Learning Designs en depósitos: Los diseñadores necesitan guardar sus Learning Designs, tanto los borradores para desarrollarlos como los diseños terminados para que puedan ser usados posteriormente.

- Búsqueda y recuperación de los Learning Designs de los depósitos: Es necesario para destruir borradores que ya no sirven, modificarlos o añadir otros nuevos.

Arquitectura de una herramienta flexible para la creación de Learning Design

El editor de LD tiene que representar dos tipos de funciones: proporcionar los medios para crear escenarios pedagógicos y definir el flujo de actividades sobre varias condiciones de derivación (bifurcaciones). Puede usar cualquiera de las dos como un solo diseño o plantilla. Existen requisitos concretos que permiten que un editor de LD pueda completar un diseño con recursos y servicios específicos.

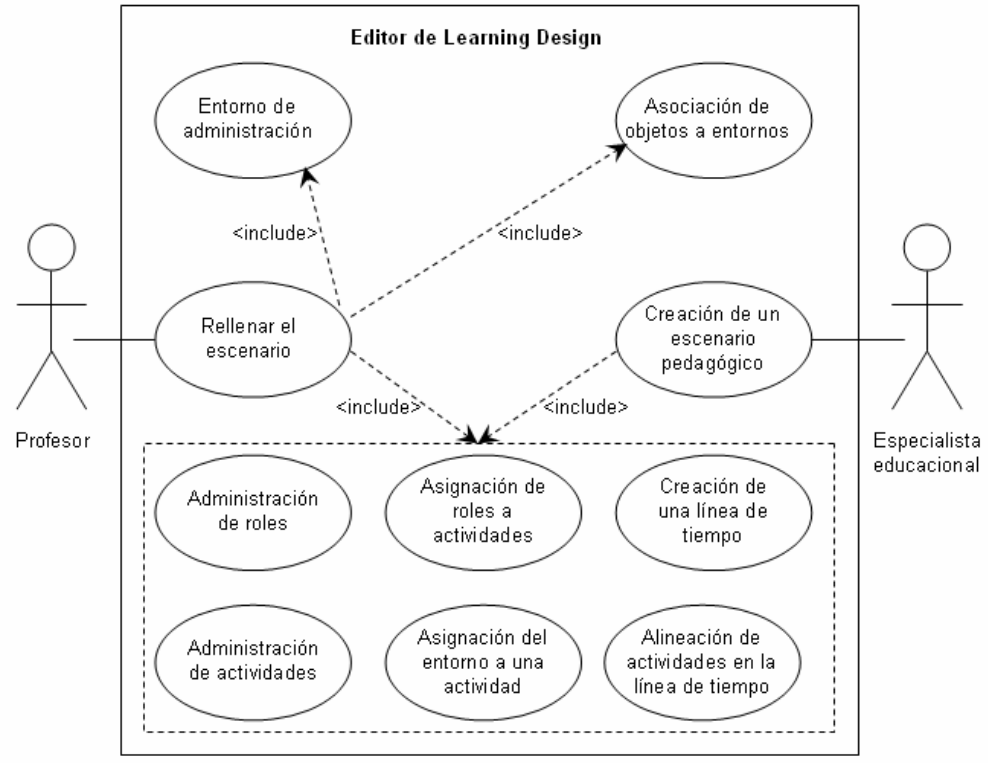
Estos conjuntos de requisitos se solapan, pero tienden a tener cierta especialización en la forma de actores que realizan tareas; Por ejemplo, en un primer caso, tenemos el rol de un especialista en educación que define un panorama pedagógico, mientras que es a menudo el profesor quien, en último caso, “rellena” el panorama con lo necesario para una sesión particular. La distinción principal entre estos agentes no tienen por que ser necesariamente las diversas funciones que utilizan, pero sí los requisitos de utilización de la interfaz de herramientas. Estos requisitos solapados se expresan mediante un diagrama de grano grueso de casos de uso en UML.

Una interpretación de estos requisitos es crear interfaces de usuario especializados que manejen las tareas de crear escenarios pedagógicos y rellenar escenarios y sus varias tareas secundarias, preferiblemente en un orden predefinido.

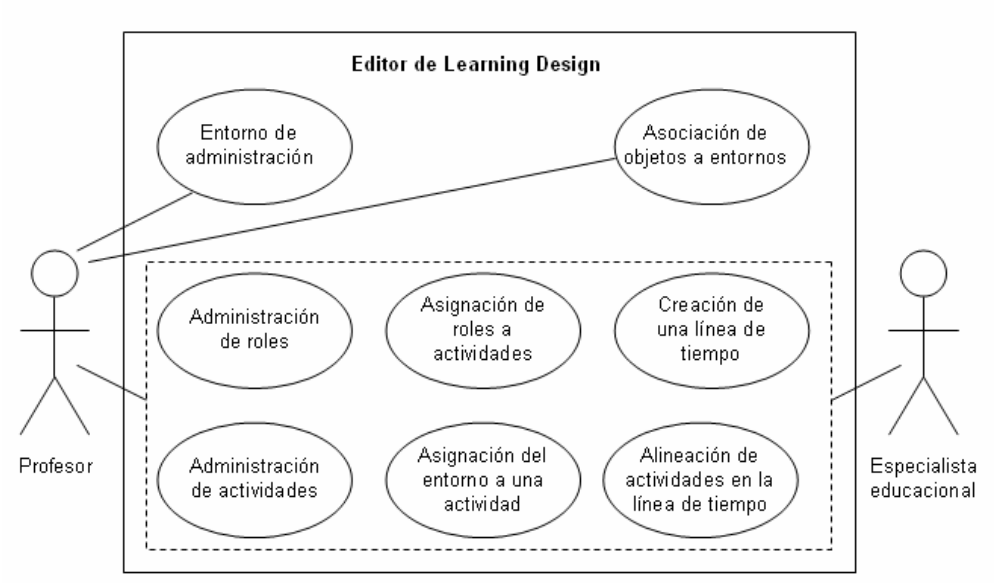
Alternativamente, los requisitos se podrían expresar como casos de uso de grano fino individuales, que se pueden realizar en cualquier orden o combinación, permitiendo más flexibilidad.

Éstos no son los dos únicos papeles posibles, o conjuntos de casos de uso, sino que son un conjunto de referencias útiles para definir lo que hace un editor de LD. Cuantas más herramientas aparezcan, más posibilidades hay de que se dé el caso en el que emergen diversos modelos para gestionar la creación de procesos.

En la siguiente figura podemos ver un diagrama UML de grano grueso de casos de uso. Nótese que “Manage” (gestionar) significa crear, leer editar y borrar algún tipo de objeto, pero “Manage Enviroment” (gestionar el entorno) significa crear, leer, editar y borrar elementos del entorno de LD.



En la siguiente figura se muestra un esquema equivalente al anterior pero utilizando un diagrama UML de casos de uso de grano fino.



Construcción de un editor de LD

El editor de LD es un extenso paquete y puede ser bastante complicado construirlo entero en un solo proyecto de desarrollo.

No obstante, es posible crear una estructura en la cual los distintos componentes del editor puedan "conectarse" para colaborar con el desarrollo del mismo. Para ello contemplamos dos tipos de organizaciones: una que controla la base del modelo de datos del LD y otra que maneja el tratamiento de la interfaz de usuario.

La capa del modelo de datos es también un punto en el cual es necesario hacer cumplir las restricciones dentro de la aplicación mediante la incorporación de un sistema de comprobación de esquemas XML, o a través de la delegación en una restricción externa para el manejo del servicio.

Los plug-ins, son una herramienta que proporciona vistas y controladores que encajan con la presentación y la organización básica, y acceso al modelo de datos a través del modelo de organización de LD.

Cada plug-in proporcionaría una clase particular para cada tipo de creación, como gestión de actividades, de papeles o del entorno. Las variaciones en la edición de la misma pueden ser a causa de los distintos niveles de usuarios. Para los usuarios expertos, el editor podía también tener un plug-in para el diseño del Learning design "crudo" que permitiera el acceso directo para editar la representación subyacente de XML.

La arquitectura de referencia en contexto

Desde que este marco fue creado en 2002, ha habido varios progresos importantes, tanto en e-learning como en el campo de la arquitectura del sistema.

Servicios web

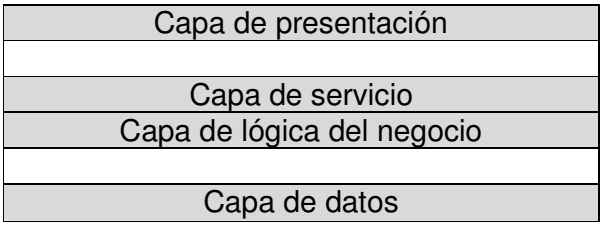
Los servicios web finalmente han emergido como una tecnología principal, con especificaciones maduras de W3C, de OASIS y de otros, y con herramientas disponibles para los entornos de programación principales de Microsoft y de la plataforma Java de Sun.

La madurez y la gran adopción SOAP y del lenguaje de descripción de servicios web han dado lugar al acercamiento al diseño del sistema conocido como arquitectura orientada al servicio.

Arquitectura orientada al servicio

La arquitectura orientada al servicio (SOA) es un acercamiento para la asociación de sistemas dentro de las empresas. Es relativamente un nuevo acercamiento, pero está ganando rápidamente renombre debido a los bajos costes de integración, junto con su flexibilidad y fácil configuración. SOA se basa en la experiencia del uso de servicios web para la integración.

En SOA, la lógica de la aplicación, contenida en varios sistemas de organización se expresa dividida en servicios, que se pueden utilizar por otras aplicaciones. Esta capa del servicio se interpone entre la de presentación y la lógica del negocio dentro de una arquitectura típica de tres grados.



Esta capa proporciona los medios para encapsular la lógica del negocio de un componente. En este sentido, SOA tiene mucho en campo común con COBRA, pero tiene un coste de implementación considerablemente más bajo.

La capa de servicio se convierte en el punto de la arquitectura donde se concreta la integración, mejor que en la capa de datos, la de presentación, o en la capa de lógica del negocio. El típico problema de integración en la capa de la lógica del negocio es que necesita un entorno de programación homogéneo.

Un acercamiento orientado al servicio no imposibilita también el uso de portales o almacenes de datos, y es totalmente independiente de la configuración del resto de la empresa. Es por ello por lo que consigue un buen acercamiento a la integración en ambientes heterogéneos.

Idealmente, sería óptimo crear definiciones estándar para cualquiera de estos servicios; por ejemplo, una definición estándar para un servicio de gestión de Learning Design permitiría a todos los editores de LD consumir fácilmente los servicios proporcionados por cualquiera de los depósitos del Learning Design.

La iniciativa del conocimiento abierto

La iniciativa del conocimiento abierto (Open Knowledge Initiative (OKI)) también ha estado desarrollando un sistema para arquitectura. Aunque el modelo OKI no define su arquitectura en términos de los servicios web, sino como un sistema de interfaces de programación de aplicaciones (APIs), hay mucha concordancia entre la iniciativa tomada por OKI y la arquitectura orientada al servicio.

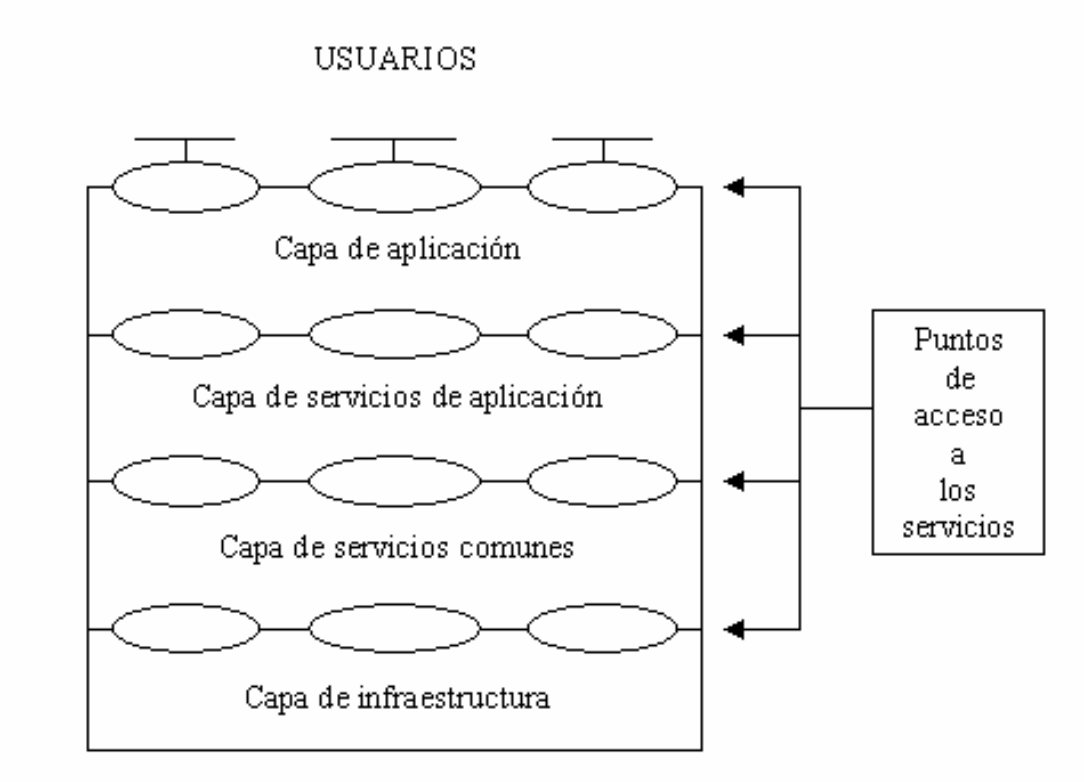
El modelo OKI define dos grandes agrupaciones de servicios: al primero nos referimos como servicios de aplicación, que se centran en el soporte de las necesidades en usos educativos, siempre desde la perspectiva del aprendizaje, administrativa o de la gestión de la información. Al segundo grupo lo llamamos servicios comunes, y es el conjunto de servicios asociados al acceso a las partes de la infraestructura técnica común, tales como la autentificación y gestión de datos.

Sobre estas dos capas de servicios se sitúan las aplicaciones del usuario, mientras que bajo ellas está la infraestructura real de la organización.

La organización abstracta IMS

La organización abstracta es un documento publicado por el IMS Global Consortium Inc., el cuerpo con la responsabilidad de la especificación del LD. En él, IMS define en un nivel abstracto los componentes de una arquitectura de e-learning basada en estándares. Esta organización no está pensada para dirigir el desarrollo o la implementación como tales, pero sí para proporcionar un modelo de referencia para las nuevas especificaciones que se proponen o desarrollan.

Estructuralmente, la organización abstracta del IMS tiene mucho en común con el modelo de arquitectura OKI, con las mismas cuatro capas, se muestra el esquema en la siguiente figura.



IMS proporciona gran cantidad de información sobre modelado de servicios y componentes, con una meta final de poder crear especificaciones que puedan ser usadas con SOA.

RCP (Rich Client Platform)

Introducción

Aunque Eclipse está diseñado para servir como una plataforma de herramientas, está estructurado de forma que sus componentes puedan ser usados para construir cualquier aplicación cliente. El mínimo conjunto de plug-ins que se necesita para elaborar una "rich client application" es lo que conocemos como "Rich Client Platform" (RCP).

Debido a que Eclipse está construido con código libre Java, se puede usar este código para crear nuestros propios programas de calidad. Antes de la versión 3.0, era mucho más difícil, esta versión introduce la RCP y la versión 3.1 la actualiza con nuevas capacidades, y lo que es más importante nuevas herramientas de ayuda para que crear aplicaciones sea más fácil que antes. Por todo esto vamos a usar la versión 3.1 para crear la base de nuestra aplicación.

Creación de una aplicación RCP

Cómo empezar

Las aplicaciones RCP están basadas en la arquitectura plug-in de Eclipse, luego lo primero que tenemos que hacer es crear un nuevo proyecto plug-in y marcar si en la casilla donde te pregunta si quieres crear una Rich Client Application, luego se puede crear el proyecto partiendo de una plantilla (como se recomienda en los primeros usos) o crear uno vacío. Una vez hecho esto se pulsa en finalizar y aparece el "Plug-in Manifest Editor" que es una fachada para tratar con los archivos de configuración que controlan la aplicación RCP, aquí, en la pestaña de overview, podemos pinchar en "Launch an Eclipse application" ejecutar la aplicación.

Creación de un "feature"

En eclipse los "features" son simplemente colecciones de plug-ins. Los *features* son opcionales pero recomendables porque los necesitaremos si más tarde queremos usar las capacidades del *Automatic Update Manager* de Eclipse, o si queremos exportar nuestra aplicación con JNLP. Para crear un *feature* tenemos que pinchar en **File > New > Project > Plug-in Development > Feature Project.** Por convención se usa el mismo nombre que el del proyecto plug-in poniendo "-feature" como terminación.

Conversión a producto

En términos de eclipse, un producto es todo lo que va junto con la aplicación, incluyendo todos los plug-ins de los que depende, un comando para ejecutar la aplicación, iconos… etc, que distinguen la aplicación. Aunque acabamos de ver como podemos ejecutar una aplicación RCP sin definir un producto, tener uno hace mucho más fácil ejecutar la aplicación fuera de Eclipse. Esta es una de las principales innovaciones que Eclipse 3.1 trae al desarrollo RCP.

Para crear un producto tenemos primero que añadir un archivo de configuración al proyecto. Damos botón derecho encima del proyecto plug-in y seleccionamos **New > Product Configuration.** Luego tenemos que rellenar los campos pedidos, nombre (acabado en .product), etc. Al pinchar en finalizar se abre el editor de la configuración del producto. Aquí rellenamos la ficha de overview con los parámetros deseados, sin olvidarnos de añadir los features que necesitamos (pinchando en el link *product configuration*). Después guardar el trabajo realizado.

Ejecución fuera de eclipse

El punto principal de todo esto es poder ser capaces de ejecutar aplicaciones sin que el usuario tenga que saber nada sobre el código de Java y Eclipse que se ha usado. Tenemos que crear una versión simplificada del directorio de instalación de Eclipse porque el cargador de plug-ins espera que las cosas tengan una cierta composición. Este directorio debe contener el programa ejecutable original, archivos de configuración y todos los *features* y plug-ins requeridos por el producto. Podemos darle al PDE (Plug-in Development Environment) suficiente información para que lo pueda hacer por nosotros.

En la sección *Exporting* del editor de configuración del producto pinchamos en el link para abrir el "Eclipse product export wizard", elegimos el directorio raíz y el directorio para desplegar el producto, si queremos que sea código libre también marcamos la opción de incluir el código y pinchamos en finalizar. En el directorio que hemos seleccionado tenemos el producto de forma independiente a eclipse.

Applications, Workbenches, and Workbench Windows

La *Application* es una clase que se crea para que actúe como la rutina principal del programa RCP que vamos a desarrollar, es como el controlador del programa, es corto y no cambia significativamente para diferentes

proyectos. Todo lo que hace es crear un *Workbench* y adjuntarle otra clase llamada *WorkbenchAdvisor*.

El Workbench se declara y se mantiene como parte del marco de trabajo del RCP. Solo hay un Workbench pero puede tener visible más de una Workbench Window. Por ejemplo, en el Eclipse IDE cuando se inicia Eclipse se ve una Workbench Window, pero si seleccionamos **Window > New Window** aparece una segunda ventana.

Plug-in manifest

El *plug-in manifest* une todo el código y los recursos. Cuando creamos un plug-in, Eclipse crea y abre el manifest automáticamente. Éste está dividido en dos archivos: MANIFEST.MF y plugin.xml. PDE provee un editor para modificar las opciones almacenadas en estos archivos (Figura 1) pero también nos permite editar el código directamente.

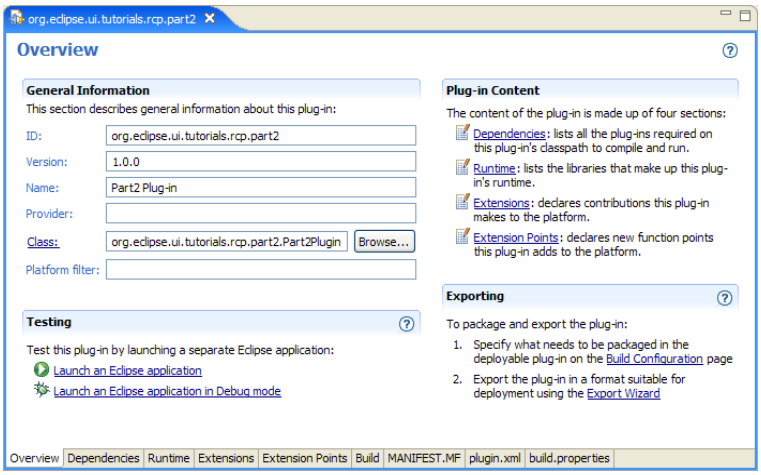


Figura 1.

MANIFEST.MF

El OSGi bundle manifest está almacenado en MANIFEST.MF. OSGi es el nombre de un estándar que Eclipse usa ara cargar plug-ins dinámicamente. Todo en este archivo pude ser editado por el editor Manifest, luego no hay necesidad de editarlo a mano. De todas formas si quisiéramos hacerlo, solo tenemos que dar doble clic en el Package Explorer para que parezca el Manifest editor y luego cliquear encima del MANIFEST.MF para ver y modificar el código.

Plugin.xml

La extensión del manifest se llama plugin.xml. Se usa para definir y usar las extensiones de Eclipse aunque no hace falta tratar directamente con él, ya se encarga el PDE.

El programa principal

La extensión org.eclipse.core.runtime.applications (plugin.xml)indica el nombre del programa principal. Esta es una clase que implementa IPlatformRunnable y un método run(). A continuación vemos un ejemplo:

```
import org.eclipse.core.runtime.IPlatformRunnable;
import org.eclipse.swt.widgets.Display;
import org.eclipse.ui.PlatformUI;
/** This class controls all aspects of the application's execution */
public class Application implements IPlatformRunnable {
  /* (non-Javadoc)
   * @see org.eclipse.core.runtime.IPlatformRunnable#run(java.lang.Object)
   */
  public Object run(Object args) throws Exception {
    Display display = PlatformUI.createDisplay();
    try {
      int returnCode = PlatformUI.createAndRunWorkbench(display, new
ApplicationWorkbenchAdvisor());
      if (returnCode == PlatformUI.RETURN_RESTART) {
        return IPlatformRunnable.EXIT_RESTART;
      }
      return IPlatformRunnable.EXIT_OK;
    } finally {
      display.dispose();
    }
  }
}
```

La perspectiva por defecto

Una Perspectiva es un conjunto de vistas, editores y menús incluyendo sus tamaños y posiciones. En un programa RCP se debe definir una perspectiva y ponerla como perspectiva por defecto. Las perspectivas se crean implementando IPerspectiveFactory, la parte importante de este interfaz es el método createInitialLayout() por el que se puede posicionar cualquiera de las vistas y/o editores con los que queremos que el usuario empiece.

Las clases Advisor

El programa principal hace referencia a una clase llamada ApplicationWorkbenchAdvisor. Es una de las tres clases Advisor que se usan para configurar todos los aspectos del Workbench, como el título, el menú, las barras... Estas son las clases más importantes para un desarrollador RCP. Hay que extender la versión base de la clase, por ejemplo WorkbenchAdvisor, en la aplicación RCP que se está desarrollando y sobrescribir uno o más métodos para establecer las opciones que se desean.

Vamos a examinar estas clases ahora con más detalle.

Workbench Advisor

Los métodos de esta clase se llaman desde la plataforma para notificar en cada punto el ciclo de vida del Workbench. También proporcionan una forma de capturar excepciones en el bucle de eventos e importantes parámetros para el Workbench como la perspectiva por defecto. Con la excepción de dos de ellos que son: createWorkbenchWindowAdvisor() y getInitialWindowPerspectiveId(), los métodos del WorkbenchAdvisor no tienen que ser sobrescritos. Éste es el aspecto de esta clase:

```
import org.eclipse.ui.application.IWorkbenchWindowConfigurer;
import org.eclipse.ui.application.WorkbenchAdvisor;
import org.eclipse.ui.application.WorkbenchWindowAdvisor;

public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {

  private static final String PERSPECTIVE_ID =
"org.eclipse.ui.ejemplo.perspective";
```

```
        public WorkbenchWindowAdvisor
createWorkbenchWindowAdvisor(IWorkbenchWindowConfigurer configurer) {
      return new ApplicationWorkbenchWindowAdvisor(configurer);
    }
  public String getInitialWindowPerspectiveId() {
    return PERSPECTIVE_ID;
  }
}
```

El porqué de que esta clase sea una clase abstracta y no un interfaz es simplemente por elección de diseño.

Ciclo de vida del Workbench

La clase Workbench Advisor tiene muchos métodos que se llaman en puntos determinados del ciclo de vida del Workbench (Tabla 1). Sobrescribiendo estos métodos se puede ejecutar cualquier código que se desee en estos puntos.

Método	Descripción	Parámetro(s)
initialize	Se llama el principio para llevar a cabo cualquier setup como parsear la línea de comandos, declarar imágenes, etc.	IWorkbenchConfigurer
preStartup	Se llama después de la inicialización pero antes de que se abra la primera ventana. Se usa para establecer las opciones de inicialización para los editores y las vistas.	
postStartup	Se llama después de que se han abierto o restaurado todas las ventanas, pero antes de que empiece el bucle de evento. Puede ser usado para empezar procesos automáticos y abrir otras ventanas.	
preShutdown	Se llama después de que haya acabado el bucle de evento pero antes de que ninguna ventana haya sido cerrada.	
postShutdown	Se llama después de que todas las ventanas se hayan cerrado durante el cierre del Workbench . Se puede usar para guardar el estado de la aplicación actual y limpiar cualquier cosa creada por initialize.	

Tabla 1

Bucle de evento

El bucle de evento es el código que se está ejecutando la mayor parte del tiempo durante la vida del Workbench. Maneja todas las entradas del usuario y las envía a los oyentes apropiados. Vemos ahora los métodos en la tabla 2:

Método	Descripción	Parámetro(s)
eventLoopException	Se llama si hay una excepción sin capturar en el bucle de evento. La implementación por defecto registraría el error.	Throwable
eventLoopIdle	Se llama cuando el bucle de evento no tiene nada que hacer.	Display

Tabla 2

Obtención de la información

Ahora vemos en la tabla 3 unos pocos métodos que se pueden implementar para que la plataforma obtenga información sobre la aplicación que desarrollamos. El más importante de éstos (y el único que no es opcional) es getInitialWindowPerspectiveId.

Método	Descripción	Parámetro(s)
getDefaultPageInput	Devuelve la entrada por defecto para nuevas páginas del workbench. Por defecto a null.	
getInitialWindowPerspectiveId	Devuelve la perspectiva inicial usada para las nuevas ventanas del workbench. No tiene valor por defecto.	
getMainPreferencePageId	Devuelve la página que debe ser visualizada primero. Por defecto a null, significando que las páginas arrancan por orden alfabético.	

Tabla 3

Workbench Window Advisor

Esta clase se usa para controlar la línea de estado, la barra de herramientas, el título, el tamaño de la ventana y otras cosas que casi siempre se quieren personalizar. Este es un ejemplo de implementación:

```
import org.eclipse.swt.graphics.Point;
import org.eclipse.ui.application.ActionBarAdvisor;
import org.eclipse.ui.application.IActionBarConfigurer;
import org.eclipse.ui.application.IWorkbenchWindowConfigurer;
import org.eclipse.ui.application.WorkbenchWindowAdvisor;

public class ApplicationWorkbenchWindowAdvisor extends
WorkbenchWindowAdvisor {

    public ApplicationWorkbenchWindowAdvisor(IWorkbenchWindowConfigurer
configurer) {
        super(configurer);
    }

    public ActionBarAdvisor createActionBarAdvisor(IActionBarConfigurer
configurer) {
        return new ApplicationActionBarAdvisor(configurer);
    }

    public void preWindowOpen() {
        IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
        configurer.setInitialSize(new Point(400, 300));
        configurer.setShowCoolBar(false);
        configurer.setShowStatusLine(false);
        configurer.setTitle("Hello RCP");
    }
}
```

Los métodos del WorkbenchWindowAdvisor (Tabla 4) necesitarán acceso a la interfaz Configurer (en este caso, IWorkbenchWindowConfigurer) .

Método	Descripción	Parámetro(s)
preWindowOpen	Se llaman en el construcor del Workbench Window. Se usa para establecer opciones como si la ventana tendrá o no un menú.	
postWindowRestore	Se llama en los casos en que una ventana se restaura desde un estado guardado pero antes de que sea abierta.	
PostWindowCreate	Se llama después de que una ventana haya sido creada pero antes de que se abra.	
openIntro	Se llama inmediatamente después de que se abra una ventana para crear el componente Intro (si hay).	
postWindowOpen	Se llama justo después de que se abre la ventana del Workbench. Se puede usar para por ejemplo poner un título o cambiar el tamaño de la ventana..	
preWindowShellClose	Se llama después de que se cierra la ventana del Workbench. Es un buen sito para un diálogo de “¿Seguro que desea cerrar la aplicación?”	
postWindowClose	Se llama después de que se cierra la ventana del Workbench. Se usa para limpiar todo lo creado por preWindowOpen.	

Tabla 4

Hay más métodos pero no entramos en ellos por ser de configuración avanzada.

ActionBar Advisor

En la jerga de Eclipse , "action bar" encapsula los términos menú, barra de herramientas, y barra de estado. El ActionBar Advisor maneja la creación de Actions (acciones) en estas situaciones. Un plug-in también puede intervenir acciones dinámicamente con su archivo plugin.xml.

ApplicationActionBarAdvisor.java

```
import org.eclipse.jface.action.IMenuManager;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.application.ActionBarAdvisor;
import org.eclipse.ui.application.IActionBarConfigurer;
```

```
public class ApplicationActionBarAdvisor extends ActionBarAdvisor {
    public ApplicationActionBarAdvisor(IActionBarConfigurer configurer) {
        super(configurer);
    }
    protected void makeActions(IWorkbenchWindow window) {
    }
    protected void fillMenuBar(IMenuManager menuBar) {
    }
}
```

Método	Descripción	Parámetro(s)
makeActions	Se llama para crear las acciones que se usan en los métodos.	IWorkbenchWindow
fillMenuBar	Se llama para rellenar el menú con los principales menús para la ventana.	IMenuManager
fillCoolBar	Se llama para rellenar la barra movible con las principales barras de herramientas para la ventana.	ICoolBarManager
fillStatusLine	Se llama para rellenar la línea de estado.	IStatusLineManager
isApplicationMenu	Devuelve cierto si el menú es de la aplicación que se está desarrollando.	String

La clase plug-in

La clase plug-in es una clase opcional que se usa para almacenar la información global para el plug-in. Vemos un ejemplo de esta clase:

```
import org.eclipse.ui.plugin.*;
import org.eclipse.jface.resource.ImageDescriptor;
import org.osgi.framework.BundleContext;

/** The main plugin class to be used in the desktop. */
public class Part2Plugin extends AbstractUIPlugin {
  //The shared instance.
```

```
  private static Part2Plugin plugin;
   /** The constructor.  */
  public Part2Plugin() {
   plugin = this;
  }
  /** This method is called upon plug-in activation */
  public void start(BundleContext context) throws Exception {
   super.start(context);
  }
  /** This method is called when the plug-in is stopped */
  public void stop(BundleContext context) throws Exception {
   super.stop(context);
   plugin = null;
  }
  /** Returns the shared instance. */
  public static Part2Plugin getDefault() {
   return plugin;
  }
  /** Returns an image descriptor for the image file at the given
   * plug-in relative path.
   * @param path the path
   * @return the image descriptor
   */
  public static ImageDescriptor getImageDescriptor(String path) {
   return AbstractUIPlugin.imageDescriptorFromPlugin("org.eclipse.ui.ejemplo",
path);
  }
}
```

Las propiedades "build"

El archivo de propiedades build se necesitará cuado se exporte la aplicación. En particular si la aplicación necesita cualquier recurso como iconos que se puedan catalogar en la sección bin.includes. El editor del Plug-in Manifest proporciona una interfaz para modificar este archivo para que sea más difícil cometer un error que modificándolo a mano.

```
source.. = src/
output.. = bin/
bin.includes = plugin.xml,\
               META-INF/,\
               .
```

Aplicando RCP al editor de Learning Design

Vista toda la teoría sobre RCP ahora vamos a ver cómo la aplicamos a nuestro proyecto. Para adaptarlo a nuestra aplicación sólo hemos necesitado unas pocas clases:

- *Application.*
- *ApplicationActionBarAdvisor.*
- *ApplicationWorkbenchAdvisor.*
- *ApplicationWorkbenchWindowAdvisor.*
- *Perspective*.

La más interesante es *ApplicationActionBarAdvisor* ya que es donde se configuran las acciones que se van a llevar a cabo mediante el menú.
Para configurar la opción de crear un nuevo editor hemos necesitado añadir una nueva clase *NewAction* que se encarga de invocarle de la siguiente forma:

```
String path = openFileDialog();
If (path != null) {
        IEditorInput input = new SchemaDiagramEditorInput ( new Path (path));
        try {
                workbenchWindow.getActivePage().openEditor (
                        input,
```

```
                    "ElEditor",
                    true);
        } catch (PartInitException e) {
                e.printStackTrace();
        }
}
```

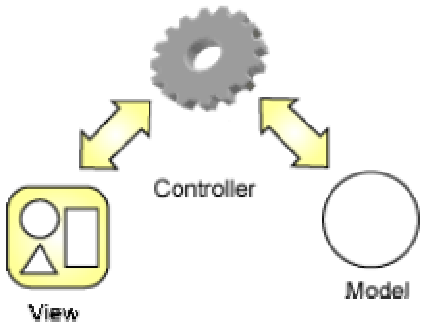
Para que GEF soporte esto hemos tenido que crear la clase IEditorInput, ya que si usamos la que se suele usar con GEF, IFileEditor, RCP no la soporta porque pertenece a. plugin **org.eclipse.ui.ide**

GEF (Graphical Editing Framework)

GEF permite a los desarrolladores coger un modelo de aplicación existente y rápidamente crear un editor gráfico muy rico.

Descripción de GEF

GEF consiste en dos plug-ins: org.eclipse.gef y org.eclipse.draw2d, éste último es el que proporciona una herramienta de composición e interpretación para desarrollo gráfico. El desarrollador puede aprovecharse de las múltiples operaciones simples que proporciona GEF y extenderlas hacia un dominio específico. GEF utiliza como patrón de diseño para la arquitectura MVC (modelo-vista-controlador), que permite cambios simples aplicables al modelo desde la vista. Es esquema del MVC es el siguiente:



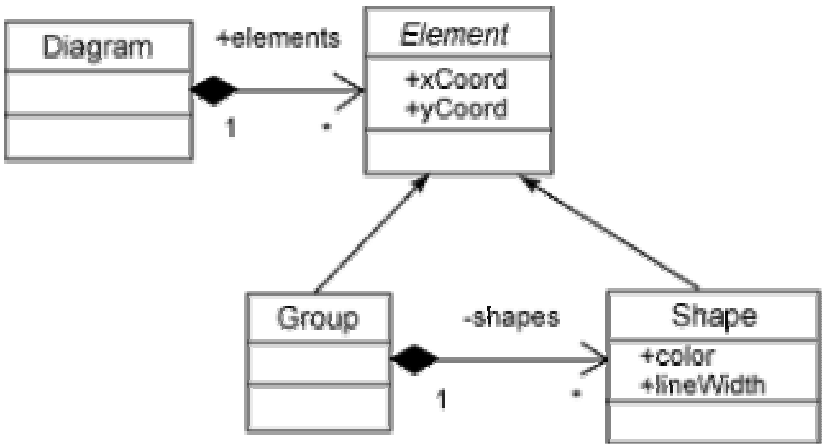
GEF es una aplicación neutral y proporciona la base de trabajo para crear casi cualquier tipo de aplicación, incluyendo, pero no limitándose a ello: diagramas de actividades, de estados y de clases, constructores de GUI (interfaces gráficas), editores de diagramas de clases, máquinas de estados, e incluso editores de texto WYSIWYG.

Creación de un modelo

GEF no sabe nada sobre el modelo, en el modelo está todo, es la única cosa persistente y modificable, la aplicación debe manejar todos los datos importantes desde el modelo.

Cuando el usuario trabaja con el controlador o controladores, el modelo no se manipula directamente desde éste, en su lugar se crea una función especial que registra el cambio. Estas funciones se usan para verificar las acciones de los usuarios, pero sólo son parte del modelo conceptualmente, no son parte del modelo en sí, pero sí la forma en la que se puede editar éste.

Otra de las aplicaciones comunes del GEF es un editor de UML, como un editor de diagramas de clases. Una información importante en el diagrama es la posición (x, y) donde aparece la clase. Suponemos que el modelo es una clase con las propiedades x e y. La mayoría de desarrolladores evitan "ensuciar" el modelo con atributos que no le aportan significado. En este tipo de aplicaciones el término "business model" es usado para hacer referencia al modelo básico en el que se aplican los detalles semánticos importantes. Mientras que la información específica del diagrama se implementa en la vista, esto significa una vista de algo que hay en el "business model", un objeto puede aparecer más de una vez en el diagrama. A continuación vemos el esquema de un modelo simple:



Si el modelo se divide en dos partes, o incluso en muchas, no es problema para el GEF. El término modelo se usa para referirse al modelo de la aplicación entero, un objeto en la pantalla puede asociarse a muchos objetos del modelo.

Las actualizaciones de la vista casi siempre son el resultado de una notificación del modelo. El modelo puede tener varios mecanismos de notificación, salvo excepciones.

Las estrategias de notificación pueden ser bien distribuidas entre los objetos, o bien centralizadas por el dominio. Un dominio de notificación debe conocer cualquier cambio de cualquier objeto del modelo y propagar estos cambios a los oyentes del dominio. Si una aplicación utiliza este tipo de notificaciones lo más normal es añadir un oyente de dominio a cada vista. Cuando un oyente recibe la notificación de un cambio, lo transmite al controlador afectado para que lo trate correctamente. Si la aplicación utiliza notificación distribuida, cada controlador añadirá sus propios oyentes a cualquier modelo de objetos que lo afecte.

Definición de la vista

El siguiente paso es decidir como representar el modelo utilizando dibujos del plug-in Draw2D. Algunos dibujos se pueden utilizar para representar directamente un modelo de objetos. Utilizar Draw2D puede facilitar el manejo y reutilización del proyecto, siguiendo estos pasos:

- No reinventar la rueda: puedes combinar las herramientas de dibujo proporcionadas para dibujar casi cualquier objeto. Sólo deberás crear tu propio "layout manager" como último recurso. Vamos a ver como ejemplo la paleta que proporciona el GEF. La paleta está dibujada usando muchas de las figuras simples del Draw2D.

- Mantener una separación clara entre el controlador y la parte gráfica: si tu controlador usa una estructura compuesta por varios dibujos mantén la mayor parte oculta al controlador. Es posible, pero no una buena idea, que el controlador cree todo él mismo, ya que no se ve una clara separación entre la vista y el controlador. En cuenta de esto se debe escribir una subclase de la vista que esconda los detalles de su estructura.

- No referenciar el modelo o el controlador desde la vista: Por lo general el controlador debe añadirse como oyente a la vista, pero la vista solo tiene acceso al controlador mediante el oyente, no mediante el controlador en sí.

- Uso de "contents pane": A veces se tiene un elemento gráfico que contiene otros elementos gráficos. Esto se puede hacer gracias a la composición de múltiples elementos gráficos. A la hora de implementar el controlador, es trivial indicar que el "content pane" debe ser usado como padre de todos los elementos que contiene.

Los controladores

Vamos a ver la forma de unir la vista y el modelo con el controlador. Las clases proporcionadas por el GEF son abstractas, por lo que se debe escribir el código. La mejor forma de unir el modelo con la vista es mediante subclases.

Hay tres implementaciones básicas para las subclases:

- AbstractTreeEditPart
- AbstractGraphicalEditPart
- AbstractConnectionEditPart

Veamos el ciclo de vida del controlador. Antes de implementar el controlador hay que fijarse de donde viene y a donde va cuando ya no se va a

usar más. Cada vista se implementa como una factoría de creación de controladores. Cuando se establece el contenido de la vista, se hace proporcionando el modelo que contiene la información a representar por la vista. La vista utiliza la factoría para crear un controlador para cada objeto de entrada. Cada controlador, en la vista puede manejar su controlador hijo, delegando en la factoría del controlador cuando sea necesario crear nuevos controladores. Según van apareciendo nuevos modelos de objetos, los controladores afectados van creando los nuevos controladores necesarios. La creación de la vista y del controlador se hace paralelamente, por lo que cada vez que se crea un controlador y se asocia con su padre, pasa lo mismo con la vista.

Cada vez que se borra un modelo de objetos, ocurre lo mismo con su controlador. Si se recupera un modelo de objetos borrado, se crea un nuevo controlador, por esto los controladores no contienen información que deba ser persistente ni son referenciados.

Un controlador, al ser creado se mantiene a la espera como oyente ante cualquier notificación de cambio por parte del modelo. Como GEF es un modelo neutral, todas las aplicaciones tienen que añadir sus oyentes y manejar los resultados de las notificaciones. Cuando se recibe una notificación, el manejador tiene que llamar a los métodos necesarios para hacer las actualizaciones. Por ejemplo si se borra un objeto, se tiene que borrar su controlador y la vista asociada.

Añadir oyentes y olvidar borrarlos cuando es necesario es un error frecuente que produce pérdida de memoria.

Otras utilidades

El GEF también proporciona otros elementos importantes para el diseño gráfico de las aplicaciones:

- Paleta: La paleta tiene un conjunto de herramientas de creación de objetos. El usuario puede activar las herramientas o arrastrar los iconos desde la paleta.

- Barras de acciones: Tanto los controladores como las vistas pueden actuar sobre las barras de herramientas, menús, etc. GEF proporciona muchas implementaciones reutilizables.

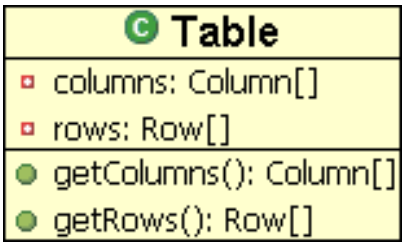
- Hoja de propiedades: Se puede utilizar para mostrar las propiedades de los iconos seleccionados. GEF permite que la hoja de propiedades sea soportada tanto por el modelo como por el controlador.

- "Outline" (vista previa): Permite tener una representación estructural del diagrama, aunque puede utilizarse en general para cualquier cosa. Normalmente se utiliza la "TreeViewer" de GEF.

Draw2D

Draw2d proporciona una ligera interpretación y posibilidades de composición en un SWT Canvas. Figuras, "Layout Managers" y bordes pueden ser combinados y anidados para crear composiciones más complejas y adaptarse a casi cualquier tipo de aplicación. Elegir la combinación correcta de figuras y "layouts" para conseguir el efecto deseado puede ser una tarea muy delicada.

A continuación vamos a ver un ejemplo de composición de una figura, queremos dibujar el siguiente diagrama de clases UML:



El primer paso en la creación de cualquier figura es tener claro los componentes por los que está formada y como se van a ordenar éstos.

La figura del ejemplo está compuesta por tres figuras "hijo", esta composición es "UMLClassFigure", su primer hijo es un "Label" (una etiqueta, donde podemos escribir por el nombre de la clase, Table en este ejemplo).

Los otros dos hijos son contenedores a su vez para los métodos y los atributos de la clase, por que vamos a crear una figura llamada "CompartmentFigure" para cada uno. Tanto la clase como los dos últimos contenedores que hemos creado utilizarán un "ToolbarLayout" para añadir a los hijos, ya que los dos últimos compartimentos también van a tener "labels".

El resultado se muestra en la figura1.

Una vez que tenemos el diagrama de una clase, podemos hacer lo mismo para las demás clases.

Draw2D ofrece un diagrama especial de conexión llamado "connection" para conectar otros dos diagramas, solo se necesita señalar los dos puntos a conectar, origen y final, normalmente la conexión se representará con una línea, pero se ofrece la posibilidad de modificar el tipo de conexión, por una flecha, ect.

Otra de las posibilidades es añadir objetos a las conexiones, por ejemplo añadir un “label”, nos podría ser de gran ayuda a la hora de representar relaciones entre clases, como las uno a uno, uno a varios, etc.

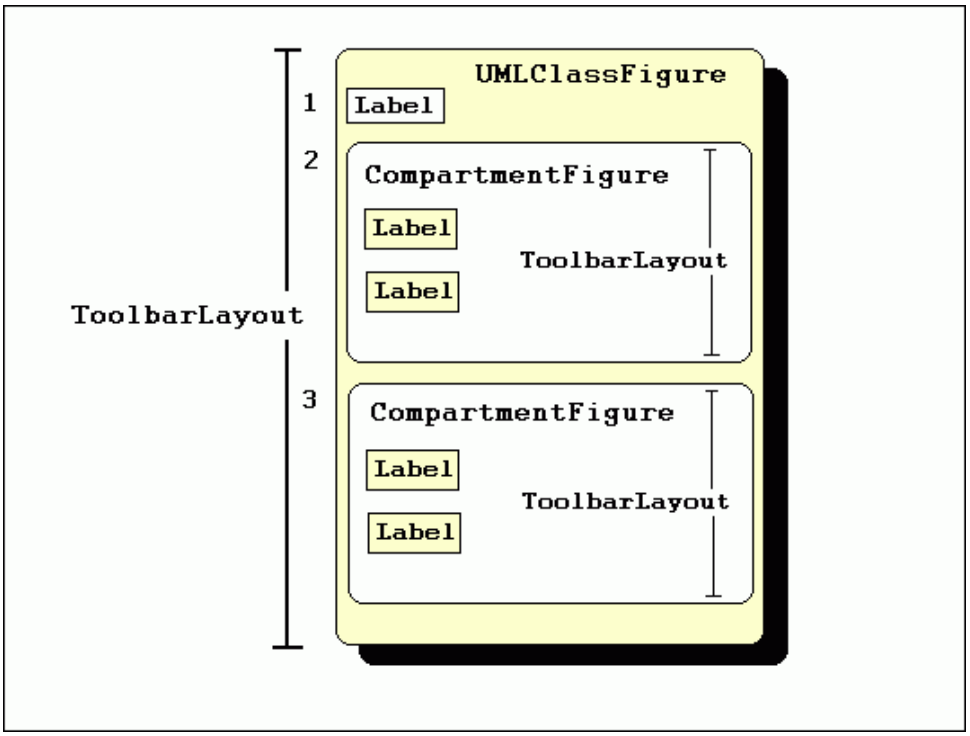


Figura 1.

EMF (Eclipse Modeling Framework)

EMF es un marco de trabajo y una utilidad para la generación de código utilizado para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurado. Partiendo de la especificación de un modelo descrita en XMI, EMF proporciona herramientas y soporte de ejecución para producir un conjunto de clases Java para ese modelo, las clases necesarias que permiten la vista y edición mediante comandos del mismo, así como un editor básico. Los modelos pueden ser especificados mediante Java documentado, documentos XML, o herramientas de modelado como Racional Rose que después se importarán a EMF. Y lo más importante de todo, EMF proporciona la base de interoperabilidad con otras aplicaciones y herramientas basadas en este marco.

Un modelo en EMF es menos general y de no tan alto nivel como lo que entendemos comúnmente como modelo. No requiere de complejas y

sofisticadas herramientas de modelado, para empezar a utilizarlo, lo único que necesitaremos es *Eclipse Java Development Tools*. EMF transforma los conceptos directamente a sus implementaciones, de esta manera ofrece a Eclipse (y a los desarrolladores de Java en general) los beneficios del modelado a un coste muy bajo.

Resumiendo, ¿por qué es necesario construir una clase java, crear un diagrama UML para representar el modelo, y escribir un documento XML para garantizar su persistencia si los tres representan una misma cosa: el modelo de datos? EMF nos permite definir éste modelo en cualquiera de las tres formas anteriores y a partir de ésta generar las otras, así como su correspondiente implementación mediante clases. Un modelo EMF es por tanto la representación común de todas las demás.

La principal diferencia entre EMF y otras herramientas que realizan esta conversión a partir del modelado es que está pensado e integrado para una programación eficiente. De esta manera no es necesario que nos hagamos la pregunta "*¿Programar o modelar?"* ya que gracias a EMF son una misma cosa.

XMI y el meta-modelo

Como sabemos, un modelo se describe usando conceptos que van mas allá de clases y métodos. Para definirlo correctamente necesitamos una terminología común para todos los elementos. Además para implementar las herramientas y el generador de EMF será también necesario un modelo para la información, un modelo para describir los modelos EMF. El meta-modelo.

Este meta-modelo se llama Ecore, y utiliza unas clases básicas que utilizaremos para representar nuestro modelo:

EClass: Utilizada para representar una clase.
EAtributte: Usada para representar un atributo.
EReference: Que representa un final de una asociación entre clases
EDataType: Usado para representar el tipo de un atributo…

Las instancias de esas clases definidas en Ecore serán las que usaremos para describir la estructura de clases de los modelos de nuestras aplicaciones.

Para definir el modelo en sí, podemos utilizar un documento UML, por ejemplo uno creado con el Racional Rose, o utilizar una herramienta como el editor gráfico Omondo para Eclipse que genera lo que se conoce como *direct Ecore editing*. Una vez que tenemos el modelo, hemos visto que disponemos de al menos tres formas de representarlo físicamente: Código Java, UML y esquema XML. Sin embargo utilizaremos una cuarta forma de representación persistente conocida como XMI (XML *Metadata Interchange*). La razón de que usemos esta forma es que es un estándar para serializar meta-datos, que es precisamente lo que es Ecore. Dado que excepto el código Java, el resto de

formas era opcional, y que evaluar todos los archivos Java cada vez que queramos reproducir el modelo no es buena idea, parece razonable utilizar XMI para representar la forma canónica de Ecore. La ventaja frente al UML es que mientras que éste tiene su propio formato de modelo de persistencia, un archivo XMI de Ecore es una serialización XML estándar de los metadatos exactos que usa EMF.

Componentes de EMF

EMF consta de tres partes fundamentales:

- EMF. El núcleo que incluye el meta-modelo Ecore. Describe los modelos y proporciona soporte para su ejecución y persistencia mediante serialización a archivos XMI. Dispone de una API muy eficiente para manejar los objetos EMF de forma genérica.

- EMF.Edit. Éste marco de trabajo incluye clases genéricas reutilizables para construir editores de modelos EMF.

- EMF.Codegen. Esta utilidad puede generar todo lo que necesitamos para construer un editor complete para un modelo EMF. Incluye una GUI donde son especificadas las opciones e invocados los generadores.

UML2

El proyecto UML2 (un sub-proyecto del *Eclipse Tools*) es una implementación basada en EMF del metamodelo UML™ 2.0 para la plataforma Eclipse. Su objetivo es ofrecer una implementación del metamodelo que soporte el desarrollo de herramientas de modelado. Provee un esquema común de XMI que facilite el intercambio de modelos semánticos. El pug-in de UML2 permite por lo tanto la especificación de un sistema usando UML y convertir este modelo en código java (como hemos visto que se podía hacer gracias al EMF)

Cómo crear un modelo con UML2

A continuación veremos los pasos a seguir para la construcción de un modelo. Todos estos pasos pueden llevarse a cabo usando el editor de UML2 o directamente escribiendo el código java equivalente. En el documento *"Getting started with UML2"*(ver bibliografía) aparecen todos estos pasos detallados así como el código fuente necesario.

-Creación de un nuevo proyecto

Lo primero que necesitaremos, como es obvio, es crear un proyecto en el workspace. Este proyecto servirá como contenedor para el modelo que crearemos usando el editor de UML2.

-Creación del modelo

En la raíz de todo modelo UML2 está el elemento de modelado, que contiene un conjunto (jerárquico) de elementos que juntos, describen el sistema físico que está siendo modelado.

-Creación de paquetes

Un paquete contiene ciertos elementos que son sus miembros y puede también contener otros paquetes. Puede importar los miembros de otros paquetes. Si el modelo que estamos creando es simple, no será necesario crear paquetes ya que éste no los tendrá…excepto el paquete raíz (el modelo), que es en sí mismo un tipo de paquete.

-Creación de tipos primitivos

Un tipo primitivo define un tipo de datos ya predefinido, sin ninguna subestructura relevante. Los tipos primitivos que se usan en UML incluyen Boolean, Integer, UnlimitedNatural y String.

-Creación de enumeraciones

Tipo de datos, cuyas instancias pueden ser un número de literales definidos por el usuario.

-Creación de literales

Un literal es un valor definido por el usuario para una enumeración.

-Creación de clases

Las clases son un tipo de clasificadores cuyas características son atributos (algunos de los cuales pueden representar extremos navegables de asociaciones) y operaciones.

-Creación de generalizaciones

Una generalización es una relación taxonómica entre un clasificador específico y otro más general, donde cada instancia del clasificador específico es también una instancia indirecta del general y tiene las características inherentes de éste

-Creación de atributos

Cuando un clasificador tiene una propiedad, ésta representa un atributo; así, es relacionada una instancia del clasificador con un valor o conjunto de valores del tipo del atributo. Los tipos que pueden tener los atributos en UML2 incluyen Artifact, Datatype,Interface, Signal y StructuredClassifier (y sus subtipos)

-Creación de asociaciones

Una asociación especifica una relación semántica entre dos o más instancias. Sus extremos se representan mediante propiedades, cada una de ellas conectada al tipo del extremo. Cuando una asociación tiene una propiedad, ésta representa un extremo no navegable de dicha asociación, en cuyo caso el tipo de la propiedad es el tipo de del extremo de la asociación (Esto no tiene por que ser así según la última especificación aún no terminada de UML 2.0).

-Guardar el modelo

Cuando creamos nuestro modelo usando el *UML2 model wizard* (es decir usando el editor de UML2), se crea un recurso UML2 con lo cual lo único que tenemos que hacer es serializar los contenidos de nuestro modelo como un archivo XMI.

Utilización del modelo

Ahora que hemos creado un modelo podemos usarlo para crear los elementos necesarios para nuestro diseño en UML. Vamos a ver por ejemplo cómo se llevaría a cabo la creación de casos de uso, de actores y de una relación *"include".*

Pasos para la creación de Casos de Uso:

1. Seleccionar el Modelo en el editor de UML2.
2. Pulsar el botón derecho del ratón y seleccionar la **opción New Child -> Owned Member Use Case** del menú contextual.
3. Introducir un valor para la propiedad "**Name**" en la vista "**Properties**"

Pasos para la creación de Actores:

1. Seleccionar el Modelo en el editor de UML2.
2. Pulsar el botón derecho del ratón y seleccionar la **opción New Child -> Owned Member Actor** del menú contextual.
3. Introducir un valor para la propiedad "**Name**" en la vista "**Properties**"
4. Seleccionar un valor o valores para la propiedad "**User Case**" en la vista "**Properties**"

Pasos para especificar una relación "include":

1. Seleccionar el Caso de Uso en el editor de UML2.
2. Pulsar el botón derecho del ratón y seleccionar la **opción New Child -> Include**.
3. Introducir valores para las propiedades "**Name**" y "**Addition**"en la vista "**Properties**".

Por supuesto estos pasos están descritos siguiendo el editor. Para crear elementos usando código java tenemos que crear métodos que accedan al modelo y que le añadan los elementos que queremos introducir. Veamos cómo sería la creación de algunos de los elementos que aparecen en un diagrama de Actividades.

Creación del Modelo:

```
public static Model createModel(String name) {


   Model model = UML2Factory.eINSTANCE.createModel();
   model.setName(name);
   out("Model '" + model.getQualifiedName() + "' created.");
   return model;
        }
```

Creación de un SwimLane:

```
    public static org.eclipse.uml2.Package createSwimLane(
      org.eclipse.uml2.Package modelo, String name) {

    org.eclipse.uml2.Package SwimLanePackage =
(org.eclipse.uml2.Package)
    modelo.createOwnedMember(UML2Package.eINSTANCE.getPackage());
    SwimLanePackage.setName(name);
    System.out.println("SwimLane " + name + " creado");
    return SwimLanePackage;
        }
```

Creación de una Actividad:

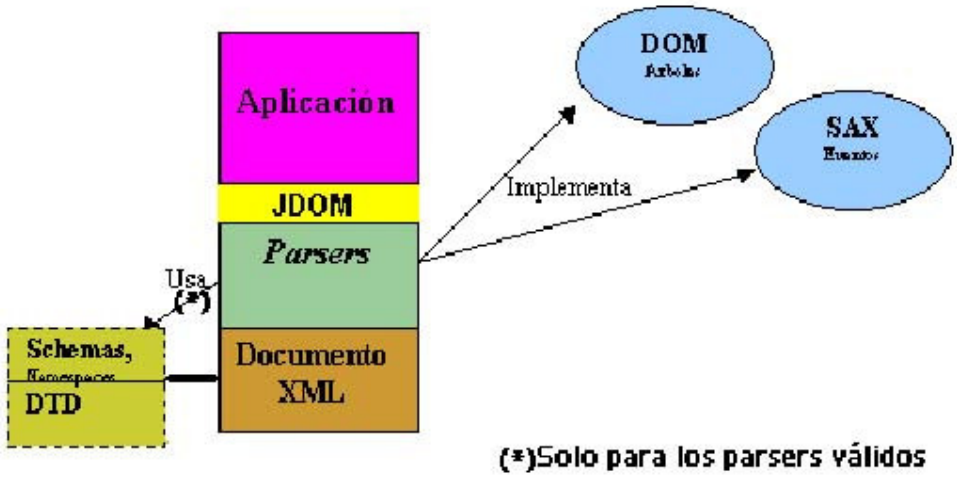
```
public static org.eclipse.uml2.Activity createActivity(
  org.eclipse.uml2.Package SwimLanePackage, String name) {

  Activity activity = (Activity) SwimLanePackage
        .createOwnedMember(UML2Package.eINSTANCE.getActivity());
  activity.setName(name);
  System.out.println("Actividad " + name + " creada dentro de "
        + SwimLanePackage.getName());
  return activity;
        }
```

JDOM

JDOM son las siglas en inglés de **J**ava **D**ocument **O**bject **M**odel (Documento de Modelado de Objetos en Java) es una librería de código fuente para manipulaciones de datos XML optimizados para Java. A pesar de su similitud con DOM del consorcio World Wide Web (W3C), es una alternativa como documento para modelado de objetos que no está incluido en DOM. La principal diferencia es que mientras que DOM fue creado para ser un lenguaje neutral e inicialmente usado para manipulación de páginas HTML con JavaScript, JDOM se creó específicamente para usarse con Java y por lo tanto beneficiarse de las características de Java, incluyendo sobrecarga de métodos, colecciones, etc. Para los programadores de Java, JDOM es una extensión más natural y correcta.

A continuación podemos ver un pequeño esquema para hacernos una primera idea sobre JDOM.



DOM y SAX son dos especificaciones que como tal no podemos trabajar con ellas, pero sí con las implementaciones de dichas especificaciones, es decir, los parsers: Xerces, XML4j, Crimson, Oracle's parsers,...

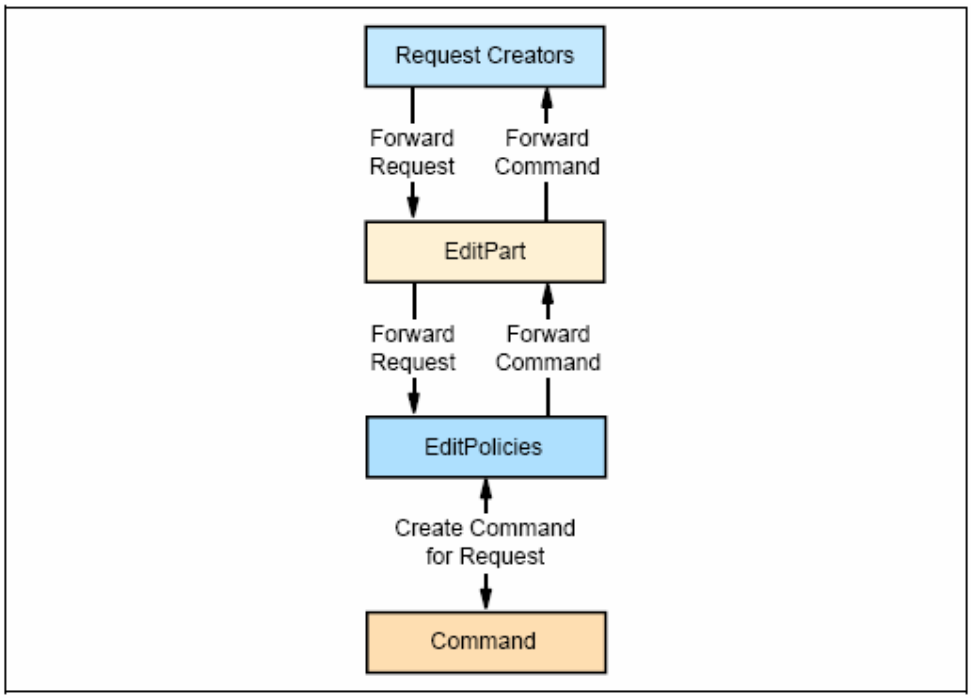
La API JDOM no es un parser, de hecho, usa un parser para su trabajo, JDOM "solo" nos aporta una capa de abstracción en el tratado de documentos XML facilitándonos bastante la tarea.

Nosotros hemos utilizado el Plug-in de JDOM para Eclipse.

Documentación del código

Antes de estudiar el código en si, vamos a recordar cómo funciona el editor gráfico para poder entender posteriormente la función de los distintos paquetes en los que está dividido el código.

El siguiente diagrama nos muestra los elementos principales de los que consta GEF (excluyendo claro el Modelo) y cómo interactúan entre ellos.



Los *Requests* son los objetos encargados de la comunicación. Contienen información que podrá ser necesaria para ejecutar la petición. Son creados por acciones, por manejadores de drag & drop, por herramientas, etc. En nuestro caso un request se producirá, por ejemplo, cuando pulsemos en el boton “New Activity” de nuestro menú, o cuando pulsemos al botón Supr para borrar un elemento del Diagrama.

Las *Edit Parts* son los elementos principales en las aplicaciones GEF. Son los controladores que especifican cómo los elementos del modelo son mapeados a las figuras visuales, y cómo estas figuras se comportarán en las diferentes situaciones.

Tenemos una Clase Edit Part para cada Clase de los elementos del Modelo:

SchemaDiagramPart, que se corresponde con el Diagrama de Actividades en su conjunto.
SwimLanePart, que se corresponde con los SwimLanes.
TablePart, que se corresponde con las Tables, que serán las que representen las Actividades.
RelationshipPart, las Dependencias entre Actividades.
ColumnPart, que se corresponde con las Column, que utilizamos para añadir descripciones a las Actividades.

Las *EditPolicies* son las encargadas de dar funcionalidad a las EditParts. Una EditPolicy define qué podemos hacer con una EditPart. De esta manera, una EditPart sin EditPolicies no hará nada. Por ejemplo:

SwimLaneContainerEditPolicy es una clase que permite a los EditParts de los SwimLanes tener la funcionalidad de contener Actividades. Actuará cuando le llegue un CreateRequest que le indique que una Actividad ha sido creada en su interior y será el encargado de crear un TableAddCommand.

Los *Commands* son los objetos que modifican nuestro modelo. Así, como hemos visto, cuando se crea, por ejemplo, un TableAddCommand, éste será el encargado de actualizar el modelo, añadiendo una nueva Actividad.

Aparte de todo esto, tenemos paquetes de utilidades para tratar los layouts, las acciones, la edición directa de nombres, etc. Así como los paquetes necesarios para utilizar RCP y UML2.

Por último y antes de entrar a ver las clases en detalle, solo nos queda comentar que todo lo relacionado con RCP se manejará con las clases del paquete editor, del paquete rcp y con la clase del propio plugin, y que UML2 lo usaremos con las clases del paquete UML. Accederemos a estas clases desde los Commands, que como hemos visto son los únicos elementos en GEF que tienen acceso directo a nuestro Modelo, y en ocasiones, serán los mismos elementos del modelo los que hagan usos de ellas, como en el caso de que sea necesario renombrar un SwimLane o una Actividad.

Javadoc

Paquetes principales

Model

- Schema.java: Representa el esquema del Modelo, es decir el diagrama de Actividad en si.

  Atributos principales:
  private String name. El nombre de nuestro Diagrama
  private ArrayList tables. Donde se almacenan las Actividades que contiene.
  private ArrayList swimLanes. Donde se almacenan los SwimLanes.

  Métodos principales:
  public void addTable(Table table). Añade la Actividad al Diagrama.
  public void removeTable(Table table). Elimina la Actividad del Diagrama
  public void addSwimLane(SwimLane s). Añade un SwimLane al Diagrama
  public void removeSwimLane(SwimLane s). Elimina el SwimLane del Diagrama.

- Table.java: Es el objeto del modelo que representa a una Actividad. Incluye un ArrayList para almacenar Columns, que contendrán las descripciones de la Actividad. Tiene también dos ArrayList para indicar las Dependencias. Uno para almacenar aquellas en las que la Actividad es prerrequisito de otra, y el otro para almacenar las dependencias en las que otra actividad es prerrequisito de ésta. El atributo activity almacenará el objeto UML2 que represente esta actividad.
- SwimLane.java: El objeto del modelo que representa un SwimLane. Tiene un ArrayList con las Actividades que contiene.
- Column.java: Objeto para incluir descripciones de las Actividaes.
- Join.java: Es una extensión de Table. Lo usaremos como un tipo especial de Actividad que sirve para sincronizar actividades.
- PropertyAwareObject.java: Proporciona la clase base para los objetos del modelo que participan en el framework de tratamiento de eventos. El resto de clases del modelo son subclases suyas.

Part

La estructura de todas las Parts es similar. Todas estas clases necesitan los metodos :

public void activate() y public **void deactivate().** Que activan o desactivan el Edit part en función de si tienen algún objeto del modelo al que mapear.

protected void createEditPolicies() que carga las políticas que va a usar cada EditPart concreto. Por ejemplo en el caso de los SwimLanes, éstos han de actuar como contenedores y van a poder ser editados, por lo tanto la clase SwimLanePart.java ha de cargar en éste método los roles: CONTAINER_ROLE, LAYOUT_ROLE, DIRECT_EDIT_ROLE y COMPONENT_ROLE.

protected IFigure createFigure() que crea una figura que representara a cada elemento.

protected void refreshVisuals() encargado de volver a dibujar la figura cada vez que sea necesario (cuando se pinta un nuevo elemento, cuando cambiamos el tamaño o la posición, etc)

public IFigure getContentPane() este método es el encargado de devolver el Content Pane para añadir o eliminar hijos. Nosotros definimos este método en la TablePart, cuyos hijos son las ColumnPart (las descripciones de las Actividades) y en SwimLanePart, cuyos hijos son las Actividades.

Aparte de los métodos comunes, cada clase tiene los métodos que necesita para definir su comportamiento. Los necesarios para la edición directa (poder cambiar el nombre de un elemento haciendo clic con el ratón), para actuar como oyentes de los cambios en las bounds, etc, los relacionados con el layout (aparte de los tres que acabamos de ver…)

- SchemaDiagramPart: EditPart para el objeto Schema, usa la SchemaFigure como contenedor de todos los objetos gráficos.
- TablePart.java: Representa una tabla editable que puede contener columnas, ser renombrada, etc. Es el controlador de las Actividades.
- JoinPart.java: Representa una Join, a la cual pueden estar conectadas varias actividades.
- SwimLanePart.java: Representa un SwimLane editable al que se pueden añadir Actividades, borrar, cambiar de nombre, etc.
- ColumnPart.java: Representa un objeto editable Columna del modelo, usado para almacenar las descripciones de las Actividades.
- RelationshipPart.java: Representa la relación que indica dependencia entre dos Actividades fuente y destino.

- PropertyAwarePart.java: Implementación Abstracta de una EditPart que responde a PropertyChangeEvents disparados por el modelo. Es la superclase de todas las demas EditParts, excepto de RelationshipPart.
- PropertyAwareConnectionPart.java: Una clase base de ConnectionEditPart, capaz de tratar eventos. Las ConnectionEditPart deben ser sublases suya. En nuestro caso, es la superclase de RelathionshipPart.

La razón por la cual hacemos esta distinción es que GEF es un framework orientado al trabajo con objetos gráficos y conexiones entre estos, por este motivo ofrece las clases AbstractConnectionEditPart y AbstractGraphicalEditPart que son las superclase de nuestras clases base, pensadas para trabajar con mayor facilidad.

Estas clases abstractas deben implementar la interfaz **PropertyChangeListener** y el método **public void propertyChange(PropertyChangeEvent evt)** para poder actuar correctamente como oyentes de los eventos que puedan suceder.

Policy

Todas las políticas que definimos extienden a una superclase que nos ofrece GEF. Dependiendo de su funcionalidad nos encontraremos con policies que extiendan a **XYLayoutEditPolicy (**las encargadas del layout de los objetos**), ContainerEditPolicy (**las que de encargan de manejar la creación de nuevos componentes dentro del EditPart al que se refieren**), ComponentEditPolicy** (políticas encargadas de manejar el borrado de componentes) y **DirectEditPolicies (**las que manejan la edición directa de los nombres y campos de texto**)**

Sólo vamos a comentar algunas de las más importantes porque su estructura es muy parecida. Lo único que hacen es crear instancias de objetos Commands pasándoles los datos que puedan necesitar.

- SchemaContainerEditPolicy.java: Maneja la creación de nuevas Actividades y SwimLanes. Evidentemente extiende a la clase ContainerEditpolicy y define el método **protected Command getCreateCommand(CreateRequest request)** que será el encargado de crear el command que corresponda en función de si estamos creando una nueva Actividad o SwimLane, pasándole la información de quién es su padre y de la localización de la nueva figura.

- SchemaXYLayoutPolicy.java: Maneja la edición manual del Diagrama, Extiende a XYLayoutPolicy y define:
  protected Command createChangeConstraintCommand(EditPart child, Object constraint), método encargado de crear los commands que mueven las Actividades y los SwimLanes (y también hacen posible el redimensionamiento de estos últimos)
- TableDirectEditPolicy.java: política para la edición directa de los nombres de las Actividades. Contiene el método que crea el command encargado de actualizar el modelo y otros auxiliares para asegurarnos que el nombre original no se pierde si no finalizamos correctamente la operación de cambio de nombre.
- TableNodeEditPolicy.java: Maneja la manipulación de Dependencias entre actividades. Esta clase extiende a **GraphicalNodeEditPolicy.** Como ya hemos comentado GEF está orientado al tratamiento de Objetos y conexiones entre estos, así que también ofrece facilidades, como esta clase base, para el tratamiento de las políticas relacionadas con la creación de nexos entre objetos, que nosotros usaremos para crear las Dependencias.

El resto de clases son únicamente variaciones de las que hemos visto, pero referidas al resto de elementos del modelo. Por ejemplo, también tenderemos políticas para manejar el layout de los swimlanes y su edición directa, etc.

Command

Todas las clases de este paquete extienden a la clase abstracta de GEF **Command,** y deben implementar (para que la clase tenga sentido) los métodos **public void execute(),public boolean canExecute()** y de manera opcional el método **public void undo().**

Hace falta poca descripción de estos métodos. CanExecute() nos impedirá que el command se ejecute cuando alguno de los datos que necesite sea incorrecto. Por ejemplo en el caso de **ChangeSwimLaneNameCommand** no podremos ejecutar el cambio si no le hemos pasado un nuevo nombre al swimLane. Execute() es el método que accede al modelo para modificarlo y undo() sirve por si queremos deshacer el último cambio que le hayamos hecho al modelo.

Igual que ocurre con las políticas, la estructura de los commands es muy similar así que solo comentaremos los más importantes.

- SwimLaneAddCommand.java: Comando que crea un nuevo SwimLane.

Atributos:

private Schema schema. Todos los commands han de tener una referencia al objeto padre del elemento al que se refieren si necesitan modificarlo.
private SwimLane swimLane. Igualmente todos han de tener una referencia al objeto del Modelo que están modificando.
private Rectangle _bounds. En este caso, también es necesario un atributo para guardar la posición y tamaño del objeto y saber dónde y cómo ha de pintarse.

Métodos:

public boolean canExecute(). Será cierto si ninguno de los atributos tiene valor nulo.
public void execute(). Pone el nombre y el padre al SwimLane y le da sus bounds. ***Y es el encargado de crear un objeto SwimLane UML2 y añadirlo al modelo (haciendo una llamada al método correspondiente de Factoria.java del paquete UML)***.
public void undo(). Elimina el swimLane del diagrama.

Además define los setters correspondientes.

- SwimLaneMoveCommand.java. Comando para mover un swimLane dentro del Diagrama.

Atributos:

private SwimLane swimLane;
private Rectangle oldBounds;
private Rectangle newBounds;

Métodos:

public SwimLaneMoveCommand(SwimLane swimLane, Rectangle oldBounds, Rectangle newBounds). Actualiza los valores de los atributos con los nuevos que se le pasan (recordemos que estos valores se los envía la policy correspondiente. Aunque igualmente pueden definirse en el cuerpo del método)
public void execute(). Hace una llamada al método que modifica las bounds del modelo pasando por parámetro el atributo newBounds.
public void undo().Hace una llamada al método que modifica las bounds del modelo pasando por parámetro el atributo oldBounds.

- DeleteSwimLaneCommand.java. Command para borrar swimLanes del Diagrama. La única particularidad que tienen los comandos relacionados con el borrado es que hemos incluido un método redo(), que se limita a llamar a execute().

- ColumnTransferCommand.java. Comando que permite mover una columna de una Tabla a otra, es decir que permite mover la descripción de una Actividad a otra.
- RelationshipCreateCommand.java. Comando que crea una nueva relación de dependencia entre Actividades. Tiene como atributos las dos tablas a las que hace referencia la conexión, y un objeto de modelo del tipo Relationship. El método execute() es el encargado como ya hemos visto de modificar el modelo ***y crear el objeto UML2 correspondiente.***

Como vemos si miramos el código el resto de commands son similares a estos, pero referidos a otros elementos del modelo.

Figures

Lo que estamos haciendo es un editor gráfico, por lo tanto los objetos que representamos han de tener una figura asociada. Ya hemos visto como esta asociación se hace a través de las EditPart. En este paquete creamos las clases correspondientes a las figuras que queremos que cada objeto utilice para su representación. Todas ellas extienden clases que nos proporciona draw2d.

- SchemaFigure.java. Figura que representa todo el Diagrama. Extiende a FreeformLayer.
- SwimLaneFigure. Figura usada para representar un SwimLane en el esquema. Extiende a Figure.
  Atributos:
  public static Color swimLaneColor = new Color(null, 255, 255, 255);
  private EditableLabel nameLabel;
  private TablesFigure tablesFigure = new TablesFigure();

Este último es necesario ya que la figura del SwimLane contendrá en su interior figuras de Actividades. En GEF es una buena práctica definir una figura "contenedor" que será la que agrupe al resto de figuras hijas que un elemento pueda tener. En nuestro caso un swimlane puede tener varias actividades dentro, y todas ellas estarán dentro de TablesFigure. Esto lo hemos visto al estudiar la clase SwimLanePart, donde hablamos del método getContentPane(), que precisamente lo que devuelve es un elemento del tipo TablesFigure.

- TableFigure.java. Figura usada para representar una Actividad (no confundir con TablesFigure.java). Extiende a Figure. Define el layout de las Tablas, los colores, el tipo de borde cuando seleccionamos ese objeto, etc. Igual que SwimLaneFigure.

- TablesFigure.java. Figura que contiene a las TableFigure, es decir a los hijos de un SwimLane.

- ColumnsFigure.java. Figura usada para contener las descripciones de las Actividades.

- EditableLabel.java. Representa una etiqueta para escribir texto normal. Extiende a la clase Label. Esta clase se utiliza en el ejemplo “flow” de GEF y la hemos reutilizado sin necesidad de cambiarla en absoluto.

UML

Este paquete contiene las tres clases necesarias para utilizar UML2.

- UML2Article. Es la clase básica para desarrollar herramientas UML2. Todo el código de esta clase está sacado del manual de UML2 de Eclipse. Define, entre otros, métodos para dar al usuario información del status del programa, y el método **public static void save(org.eclipse.uml2.Package package_, URI uri)** encargado de la persistencia en formato XML.
- UmlImplementacion.java. Esta clase contiene los métodos y todo el código necesario para la creación de modelos UML2. Nosotros únicamente utilizaremos los relacionados con la creación de Actividades, SwimLanes y Dependencias (y por supuesto el que crea el modelo en si). Como se puede ver todo se reduce a la creación de instancias de objetos EClass. Recordemos que UML2 actúa utilizando EMF y es por eso por lo que podemos crear diagramas UML mediante java y obtener persistencia en XML.
- Factoria.java. Esta es la clase que utilizaremos para crear las instancias de los objetos UML desde nuestro editor. Es la encargada de relacionar GEF con UML2. Solo accedemos a ella desde los Commands, y sus métodos hacen llamadas a métodos de la clase UmlImplementacion.java. Todos sus métodos y atributos son estáticos, los que entre otras cosas nos garantiza que únicamente tendremos una instancia del Modelo.

RCP

Este paquete contiene todas las clases que tienen que ver con esta tecnología dentro del proyecto, excepto el editor que se encuentra en el paquete Editor. Estas clases son las siguientes:

- Application.java. Esta clase es la controla la ejecución de la aplicación. El código es el mismo que se crea por defecto en eclipse al crear una nueva aplicación RCP.

- ApplicationActionBarAdvisor.java. Ésta es la responsable de crear, añadir y tratar con las acciones que ofrece el menú de la aplicación. En el apartado de RCP se explica su funcionamiento, simplemente contiene los atributos que representan a cada acción del menú y los métodos protected void makeActions(final IWorkbenchWindow window) y protected void fillMenuBar(IMenuManager menuBar), que son los que tratan con éstas acciones y las añaden al menú.
- ApplicationWorkbenchAdvisor.java. En esta clase se trata con los parámetros por defecto. En nuestro caso, sólo consta del atributo que contiene el id de la perspectiva por defecto y el método para devolver éste identificador.
- ApplicationWorkbenchWindowAdvisor.java. Ésta clase está formada principalmente por el método public void preWindowOpen() que es llamado desde el constructor de la ventana principal. Éste método se usa para configurar opciones como por ejemplo si la ventana va a tener una barra de menú o para establecer el título de la aplicación.
- NewAction.java. En esta clase se implementa la acción del menú que abre un nuevo editor.
- Perspective.java. Es la clase que configura la perspectiva por defecto para la aplicación.

DOM

Este paquete contiene a la clase encargada de parsear el documento XMI que genera UML2 y generar un documento que se ajuste al modelo de Learning Design que nos ofrece la herramienta Reload LD. Debido a su sencillez y claridad hemos utilizado **Jdom** para realizar el parseo del documento XMI.

- Dom.java. Esta clase toma como entrada el documento XML cuyo nombre coincida con el nombre del proyecto, y genera a partir de éste un documento que sigue la estructura de la especificación de LD: Components y Method. Los métodos **imprimirRoles()** e **imprimirActividades(),** junto con métodos auxiliares son los encargados de la información del Components, **imprimirRoleParts()** y sus auxiliares, generan la parte del Method.

Dado que estamos siguiendo el patrón de Reload, y éste no tiene en cuenta información sobre dependencias entre actividades ni sincronizaciones, no hemos incluido esta información en el documento que generamos. No obstante, como nuestro producto si las puede implementar, hemos desarrollado los métodos:
imprimirDependencias() e **imprimirSincronismo()**, que nos muestran por consola la información referida a estos campos.

Paquetes de utilidades

Factory

- SchemaEditPartFactory.java. Factoria para la creación de EditParts. Esta clase es la encargada de crear un EditPart en función del objeto que reciba. Es necesaria en todos los proyectos de GEF.

Action

- SchemaActionBarContributor.java. Implementa las acciones que se asocian al pulsar botón derecho del ratón sobre el editor. Estas acciones son principalmente las de hacer y deshacer.
- SchemaContextMenuProvider.java. Incorpora las acciones implementadas en la clase interior al menú.

DirectEdit

- LabelCellEditorLocator.java. Clase que trata con las etiquetas de las actividades.
- ValidationMessageHandler.java. Interfaz para controlar los mensajes de validación.
- StatusLineValidationMessageHandler.java. Clase que implementa el interfaz anterior.
- ColumnNameTypeCellEditorValidator.java. Clase que controla que el nombre introducido por el usuario para renombrar las actividades sea válido.
- ExtendedDirectEditManager.java. Esta clase añade y elimina las etiquetas de las actividades.
- TableNameCellEditorValidator.java. Esta clase controla los nombres de las etiquetas a través del interfaz ValidationMessageHandler.java.

Dnd

- DataElementFactory.java. Factoría usada para crear los objetos que se obtienen al pinchar sobre la paleta. Implementa el interfaz de GEF CreationFactory.

Editor

En este paquete están todas las clases encargadas de trabajar con el editor.

- GraphicalViewerCreator.java. Clase para configurar el Graphical Viewer.
- PlaletteFlyoutPreferences.java. Contiene las preferencias de la paleta.
- PaletteViewerCreator.java. Crea la paleta.
- SchemaDiagramEditor.java. Implementación del editor, extiende el interfaz GraphicalEditorWithFlyoutPalette. Sus principales métodos son:

    - public void init(IEditorSite site, IEditorInput input) throws PartInitException: inicializa el editor.
    - public void selectionChanged(IWorkbenchPart part, ISelection selection): actualiza las acciones cuando cambia la selección.
    - public void commandStackChanged(EventObject event): Oyente para los cambios en la pila de comandos.
    - public Object getAdapter(Class adapter): implementación adaptable del ditor.
    - public void doSave(IProgressMonitor monitor): método que guarda el modelo.
    - public boolean isDirty(): indica si el editor tiene cambios sin salvar.
    - public CommandStack getCommandStack(): devuelve la pila de comandos.
    - public Schema getSchema(): devuelve el modelo del esquema asociado con el editor.
    - protected void setInput(IEditorInput input): establece la entrada al editor.
    - protected PaletteViewerProvider createPaletteViewerProvider(): crea la paleta dentro del editor.
    - protected void createGraphicalViewer(Composite parent): crea un nuevo “GraphicalViewer”, lo configura y lo inicializa.
    - protected void createActions(): crea las acciones cuando se pulsa botón derecho sobre el editor y las registra.

- SchemaDiagramEditorInput.java. Establece la entrada al editor, guardando la ruta del fichero.
- SchemaDiagramPlugin.java. Clase de implementación del plug-in.
- SchemaPaletteViewerProvider.java. Subclase de PaletteViewerProvider para capturar los eventos de arrastrar y soltar con el ratón.

Connector

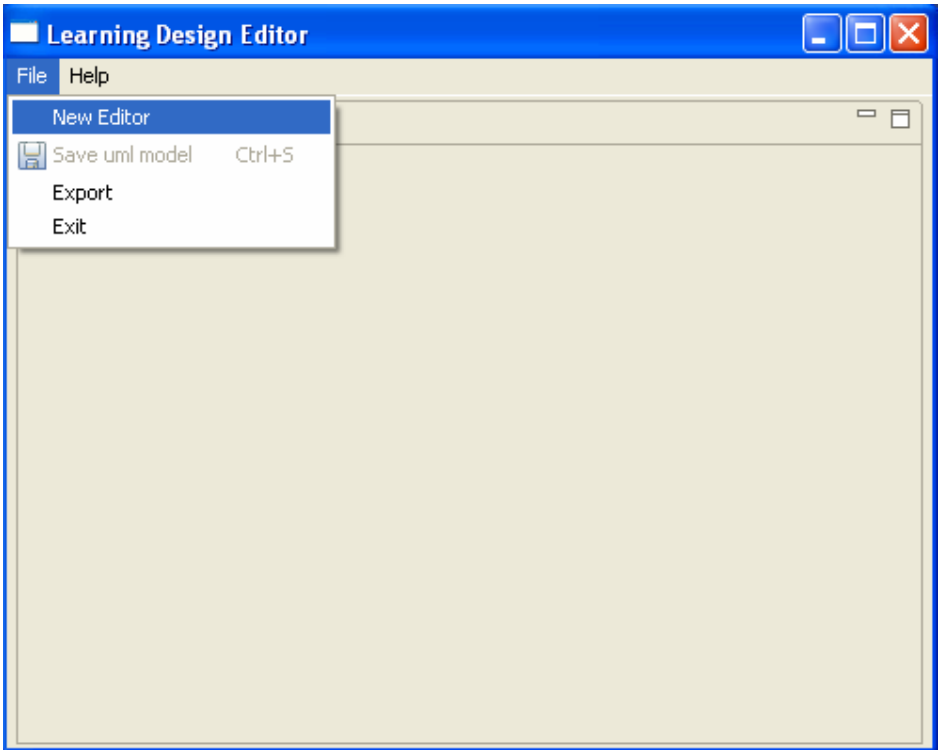
- TopAnchor.java.
- BottomAnchor.java.

Estas dos clases definen, respectivamente, el anclaje superior e inferior de las conexiones entre figuras.

A parte de estos paquetes, también utilizamos los siguientes: Jdom, Adapter, Filter, Input y Output que son necesarios para poder utilizar RCP junto con JDom.

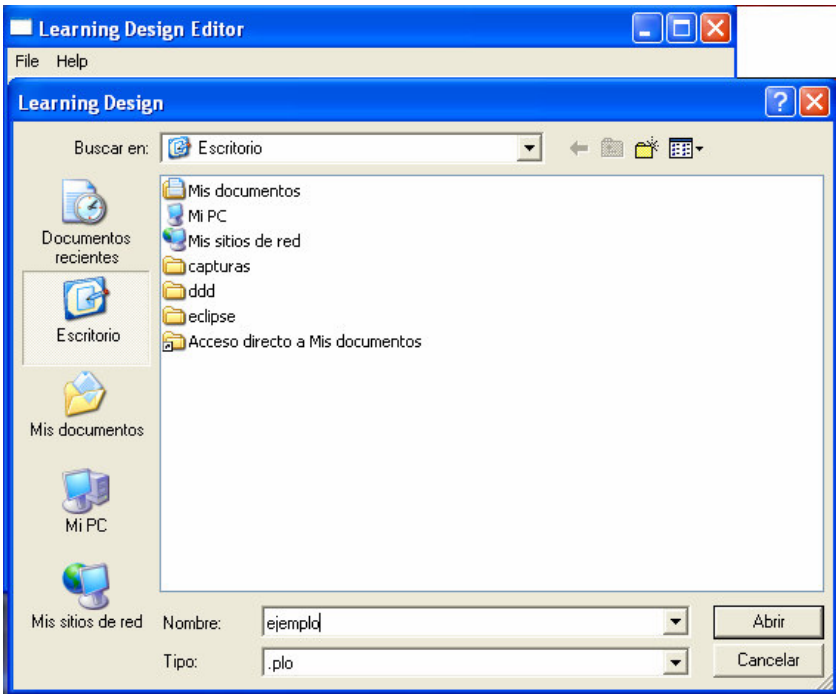
Manual de usuario

Nuestro proyecto se presenta en formato .zip, para ejecutarlo se debe descomprimir el archivo y ejecutar el icono “Eclipse” o “Startup” en su defecto. Una vez llevado esto a cabo se nos abrirá una pantalla como la que se muestra a continuación.

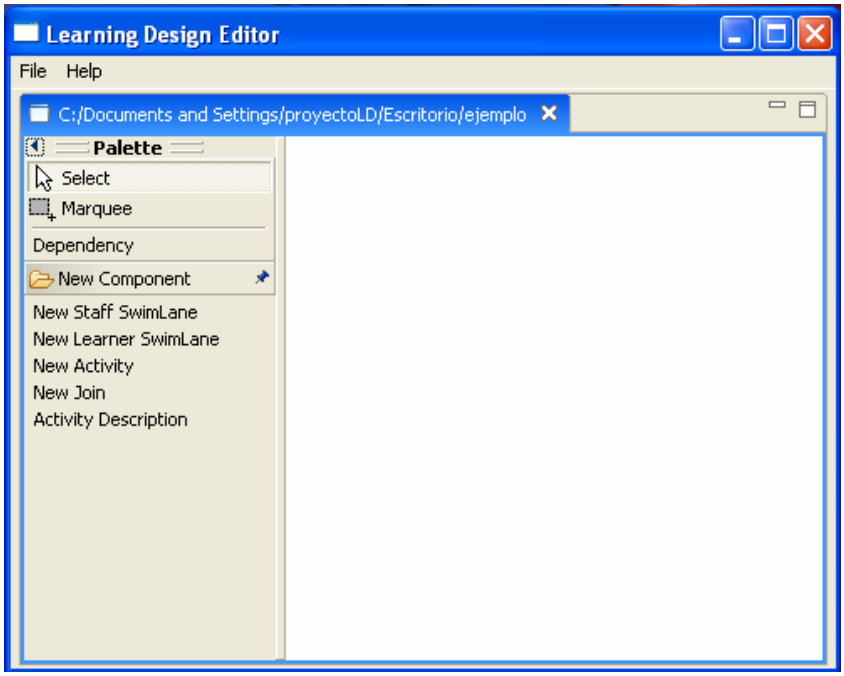


Como se puede ver en la imagen, en esta pantalla podemos crear un nuevo editor pinchando sobre el menú File -> New Editor. Esto solo se puede hacer si todavía no hay un editor abierto.

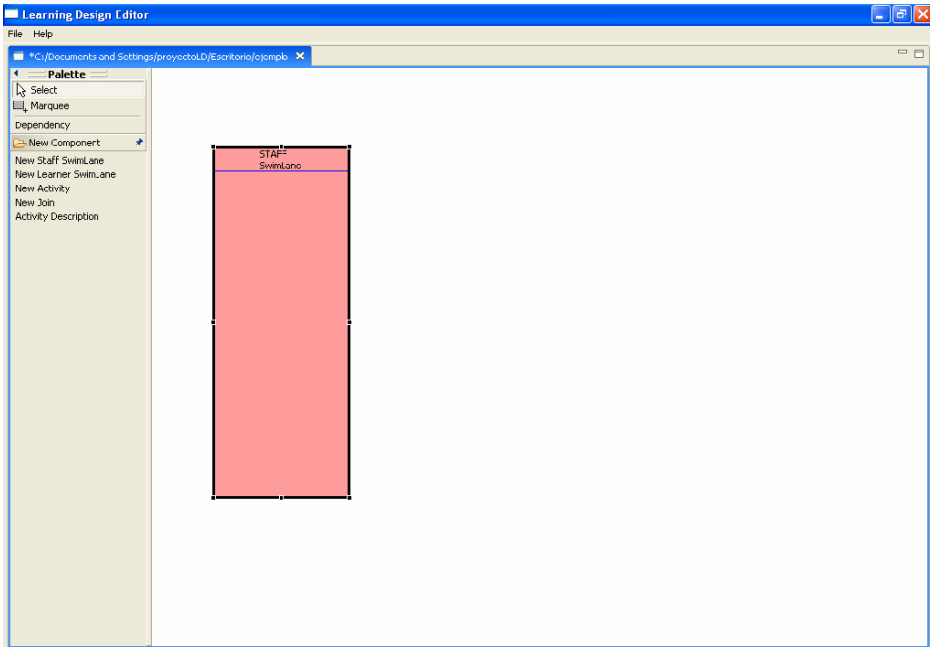
Una vez hemos creado un nuevo editor debemos seleccionar su ubicación y darle un nombre con extensión .plo.



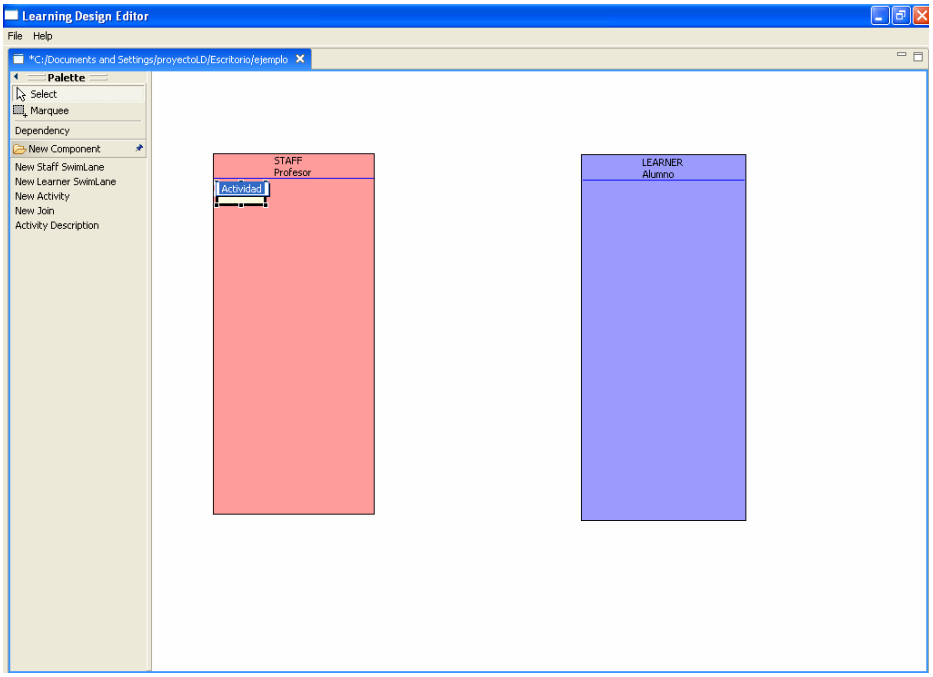
Ya tenemos una ventana con nuestro editor y su aspecto es el que se ve en la siguiente imagen. La paleta seleccionable se puede ocultar.



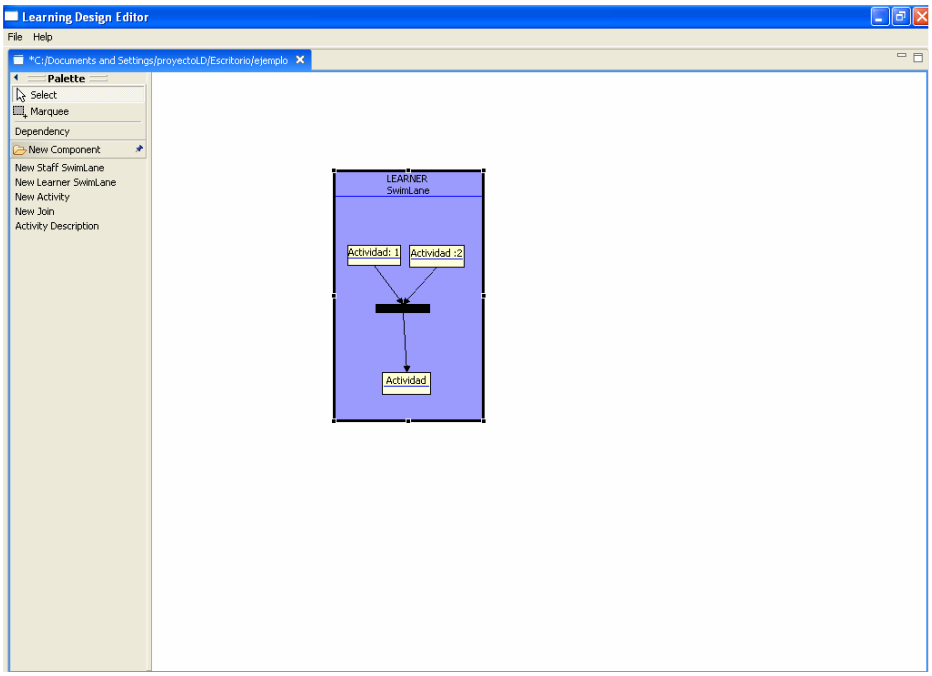
En la siguiente imagen podemos observar el resultado de crear un swimlane tipo staff, otros elementos que se pueden crear son: swimlane tipo learner, actividades, joins y dependencias, que se crean de la misma forma.



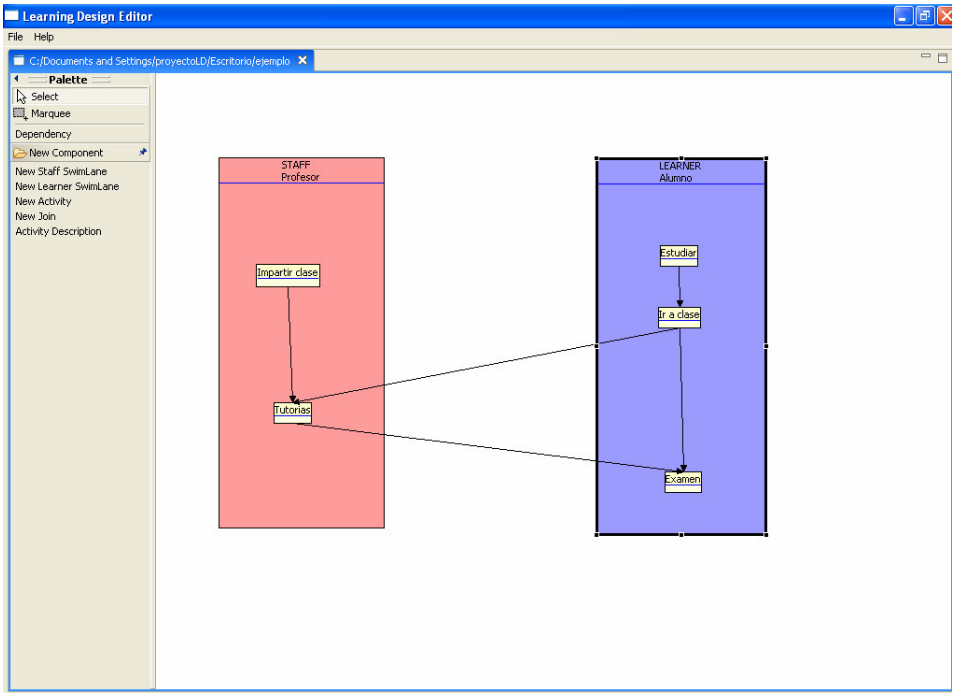
Podemos ver que los swimlanes por defecto se llaman “Swimlane”, las actividades “Actividad”, etc, pero se les puede poner el nombre deseado haciendo clic encima y escribiendo el nuevo.



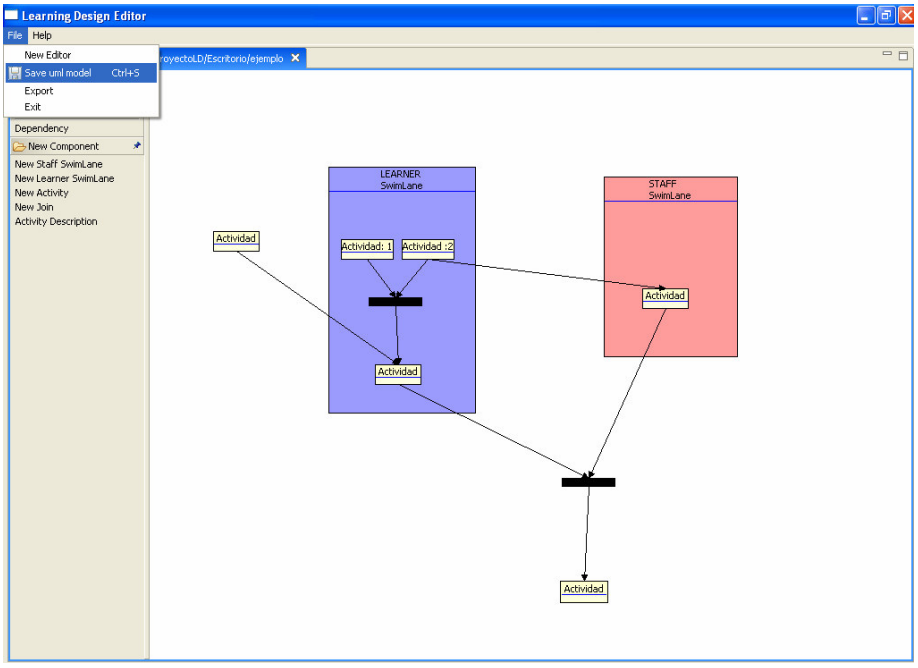
A continuación podemos ver un pequeño diseño en el que hemos utilizado un “join”, que sirve para sincronizar actividades.



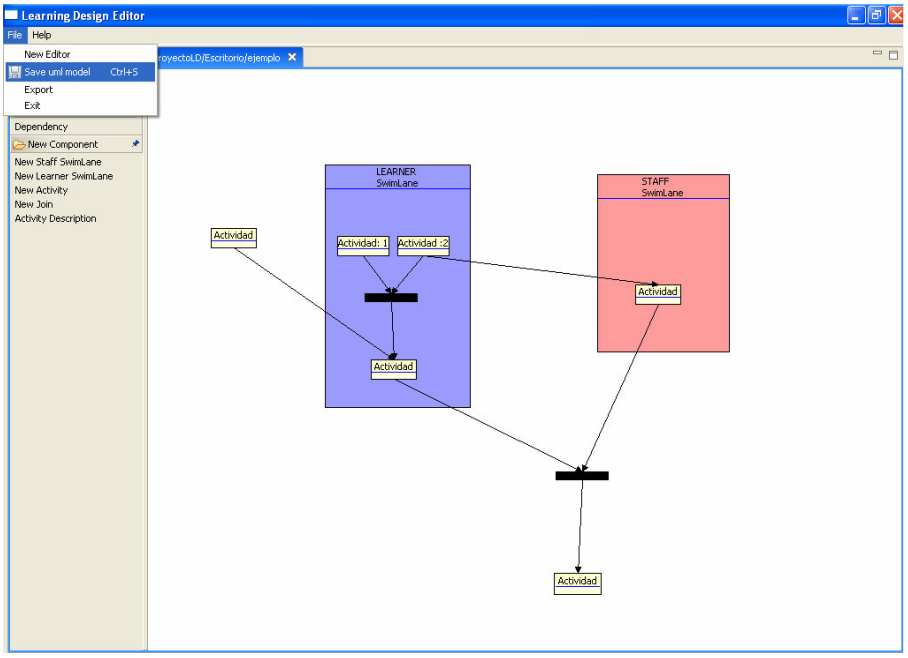
Aquí vemos como las dependencias se pueden crear entre actividades de distintos swimlanes.



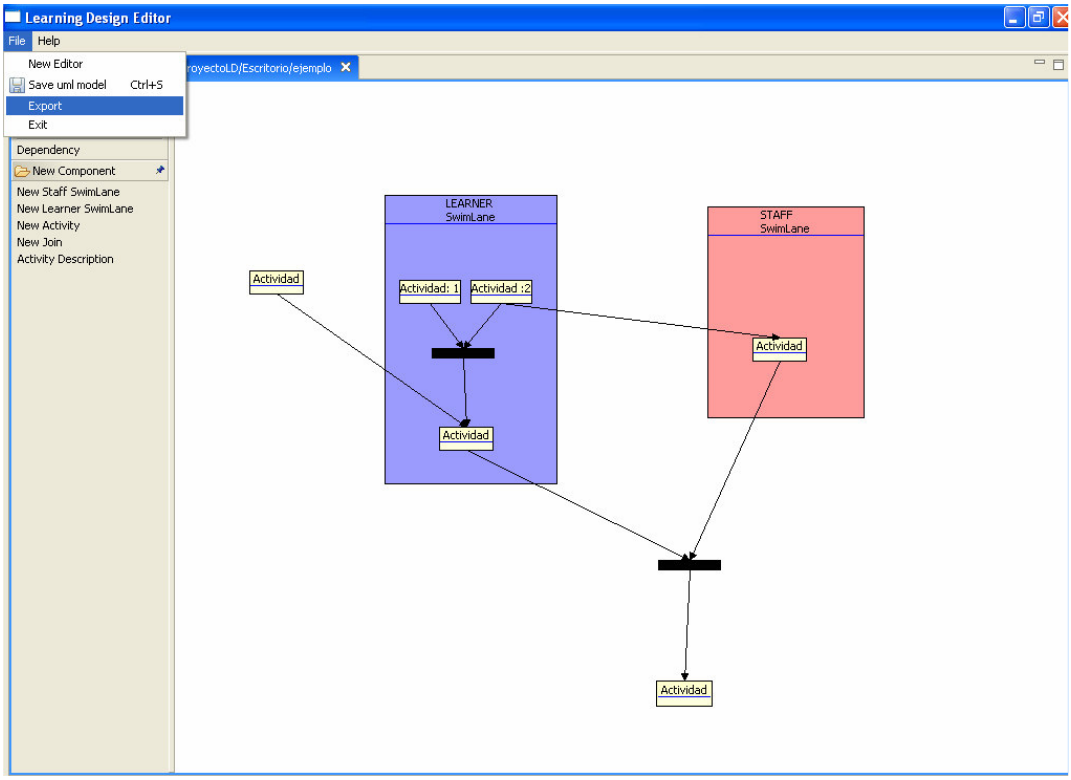
Este es un esquema que demuestra que se pueden crear tanto dependencias en actividades dentro de un mismo swimlane, como entre varios o desde fuera de ellos.



Una vez creado el diseño podemos pasar a guardarlo, para ello basta cliquear File -> Save UML model y se guardará un archivo .uml2 en la ruta que habíamos especificado al crear el nuevo editor y con el nombre que le habíamos puesto.



Por último podemos exportar nuestro diagrama a un documento .xml utilizando la acción del menú File -> Export. Para poder exportar el diagrama debe estar guardado previamente en el documento .uml2, ya que se cogen de ahí los datos, el nuevo archivo se llama como el .uml con el sufijo JDom y tiene extensión .xml.



Palabras clave

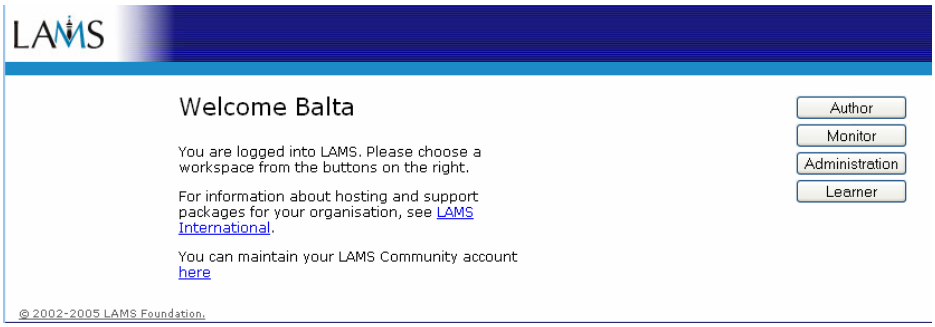
- Learning Design
- Eclipse
- GEF (Graphical Editing Framework)
- Draw2D
- RCP (Rich Client Platform)
- EMF (Eclipse Modelling Framework)
- UML2 (Unified Modeling Language)
- JDOM (Java Document Object Model)
- XML (Extensible Markup Language)

Apéndices

A continuación vamos a ver unos ejemplos de otras aplicaciones basadas en Learning Design que hemos utilizado para ver cómo encaminar nuestro proyecto, una de ellas es el LAMS y la otra es el RELOAD LD Editor que es en la que nos hemos basado para generar nuestro código XML.

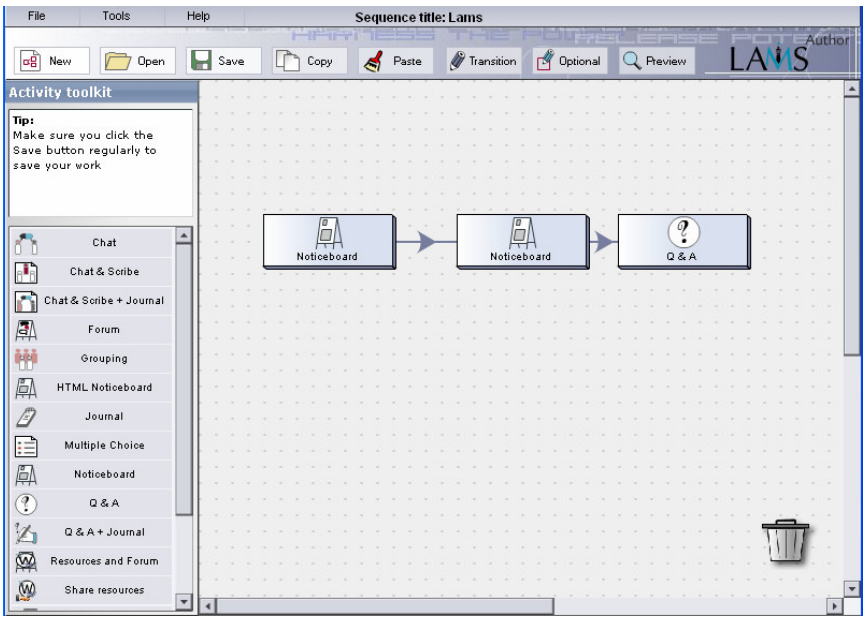
LAMS

Al empezar a trabajar con Lams la primera pantalla que nos sale es esta:



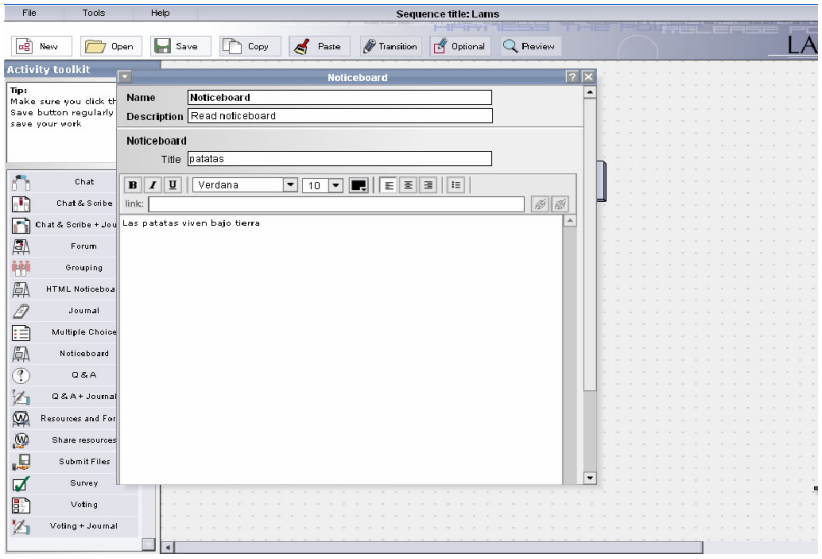
Aquí es donde elegimos el modo en el que trabajar. Para crear una nueva secuencia pinchamos en Author.

Creamos una secuencia de actividades: para ello, seleccionamos las actividades y hacemos una secuencia uniéndolas con las flechas.



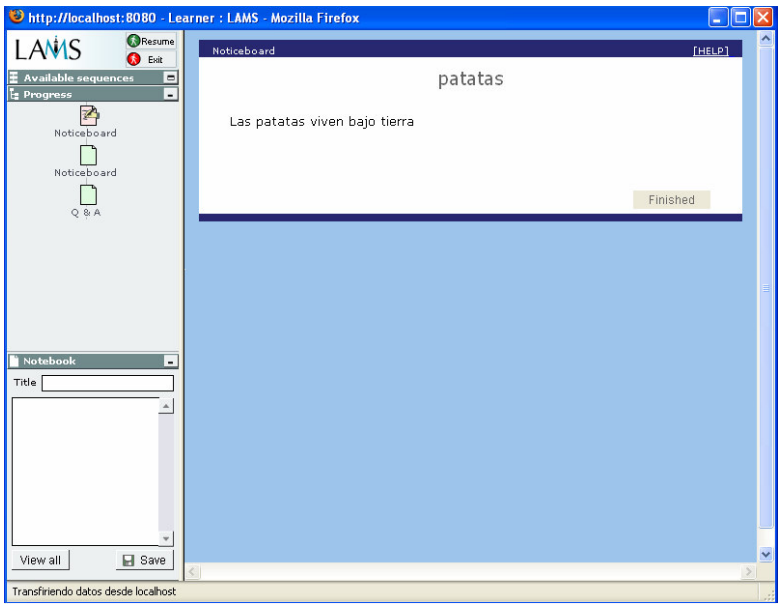
Una vez hecho esto se guarda el diseño.

Para definir el contenido de las actividades hacemos doble clic en su cuadrado

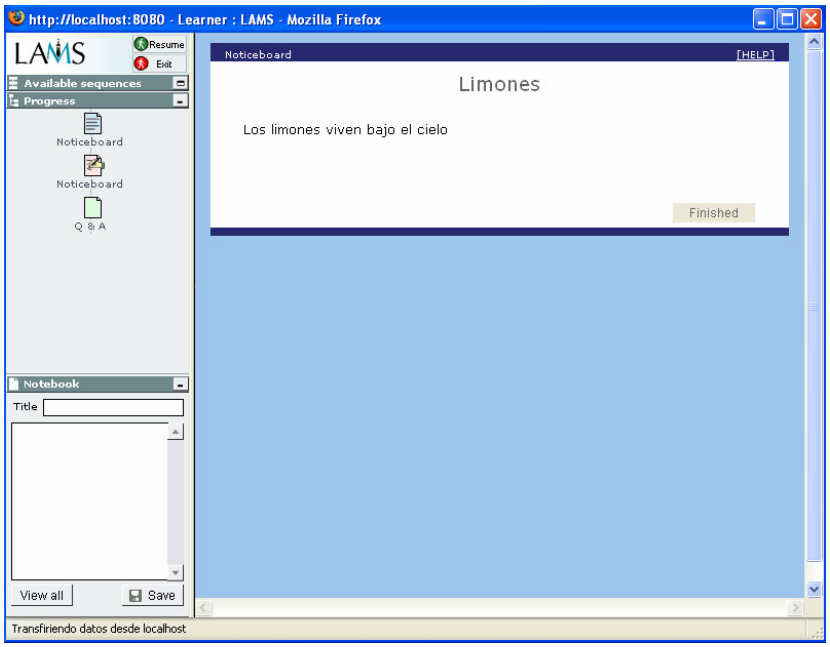


Aquí definimos un titulo y el contenido de lo que queremos mostrar cuando estamos en esta actividad. En el dibujo la actividad se corresponde con un Noticeboard, luego el contenido únicamente es un mensaje de texto.

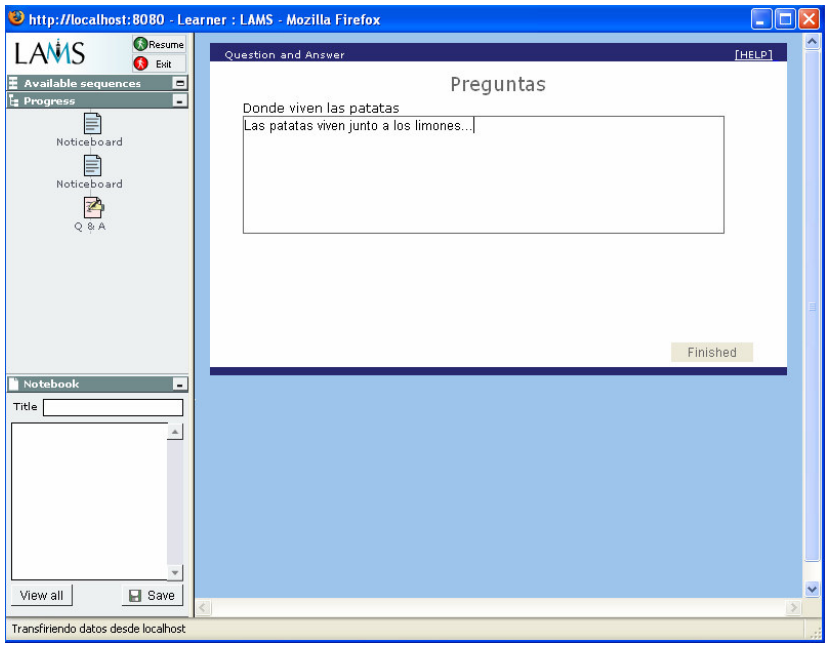
Cuando hemos definido todas las actividades y las transiciones entre ellas, pasamos a probarlas entrando en la vista de Learner. Lo que se nos muestra es lo siguiente:



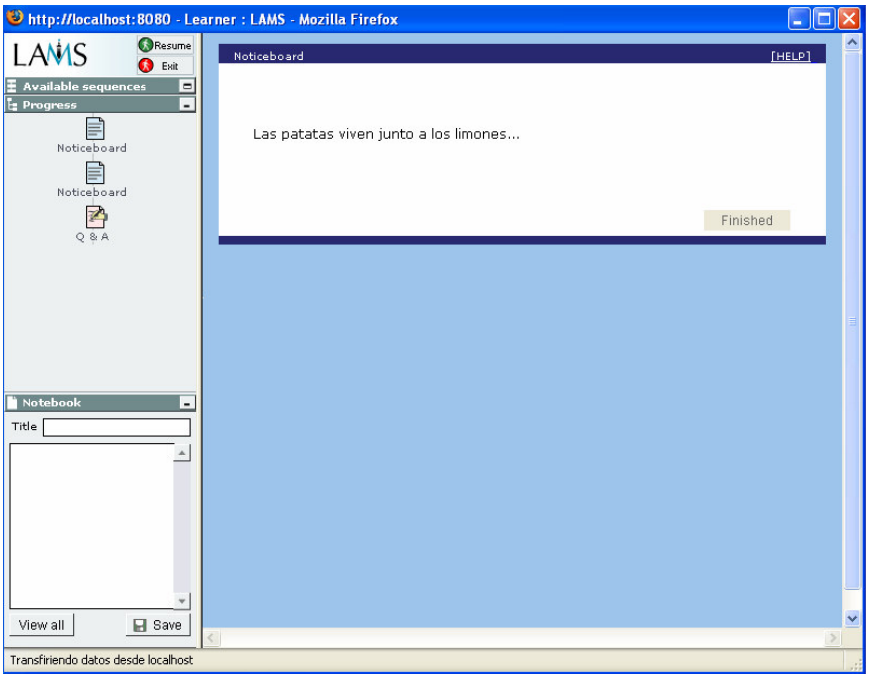
En la parte izquierda nos aparece la secuencia de actividades, resaltando en la que nos encontramos actualmente. Y en el centro la imagen que se corresponde con esa actividad.



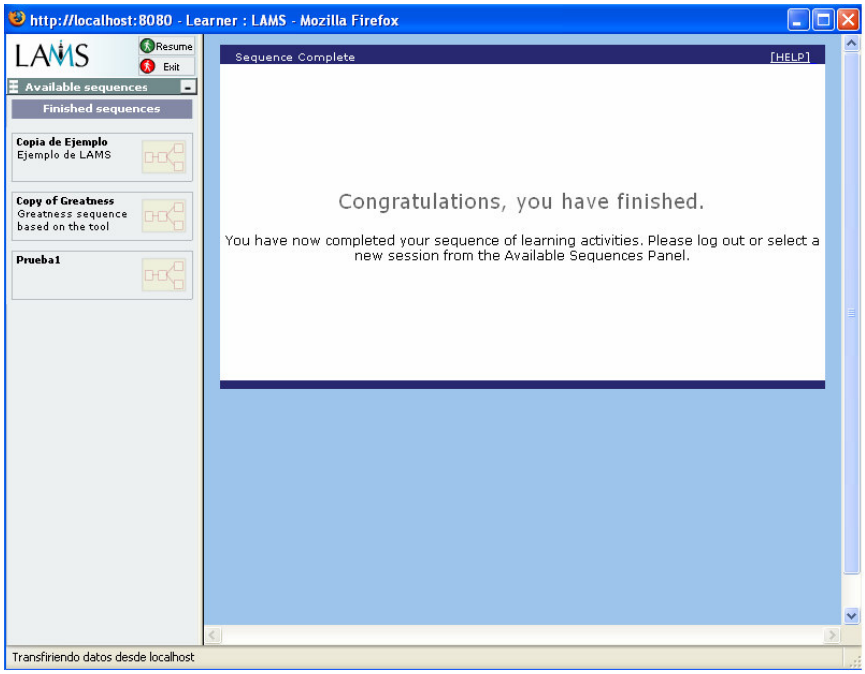
Al llegar a la ultima actividad (después de pasar por las anteriores, no podemos saltarnos ninguna) Vemos que el aspecto cambia. Ya que no es un Noticeboard, sino una Q&A (una pregunta), a la cual tendrá que contestar el usuario.



Después de responder a una pregunta, la respuesta aparecerá por pantalla.



Cuando hemos terminado la secuencia de actividades nos saldrá un mensaje que nos informará de que hemos terminado con esa secuencia. Nos ofrece la opción de abandonar la sesión o de cargar una nueva secuencia.



Cuando estamos en el modo de Autor, podemos guardar las secuencias para acceder en futuras sesiones, o importarlas al disco en un archivo .las, que puede convertirse a xml.

El código xml es el siguiente:


```
- <wddxPacket version="1.0">
  <header />
- <data>
- <struct>
- <var name="objectType">
  <string>LearningDesign</string>
    </var>
- <var name="LDId">
  <number>-1.0</number>
    </var>
- <var name="activitiesTransitions">
- <array length="5">
- <struct>
- <var name="objectType">
  <string>Activity</string>
    </var>
- <var name="id">
  <number>101.0</number>
    </var>
- <var name="sid">
  <number>1216.0</number>
    </var>
- <var name="nextTransition">
  <number>-1.0</number>
    </var>
- <var name="tasksTransitions">
- <array length="3">
- <struct>
```

```
- <var name="inputContent">
  <number>108.0</number>
    </var>
- <var name="toolType">
  <string>noticeboard</string>
    </var>
- <var name="objectType">
  <string>task</string>
    </var>
- <var name="id">
  <number>105.0</number>
    </var>
- <var name="sid">
  <number>1217.0</number>
    </var>
- <var name="grouping">
  <string>c</string>
    </var>
- <var name="nextTransition">
  <number>-1.0</number>
    </var>
- <var name="completion">
  <string>learner</string>
    </var>
- <var name="outputContent">
  <number>-1.0</number>
    </var>
- <var name="title">
  <string>Noticeboard</string>
    </var>
- <var name="description">
  <string>nbhtml</string>
```

```
    </var>
    </struct>
- <struct>
- <var name="inputContent">
  <number>107.0</number>
    </var>
- <var name="toolType">
  <string>qa</string>
    </var>
- <var name="objectType">
  <string>task</string>
    </var>
- <var name="id">
  <number>103.0</number>
    </var>
- <var name="sid">
  <number>1218.0</number>
    </var>
- <var name="grouping">
  <string>c</string>
    </var>
- <var name="nextTransition">
  <number>104.0</number>
    </var>
- <var name="completion">
  <string>learner</string>
    </var>
- <var name="outputContent">
  <number>108.0</number>
    </var>
- <var name="title">
  <string>Question</string>
```

```
    </var>
- <var name="description">
  <string>Question</string>
    </var>
    </struct>
- <struct>
- <var name="fromTasks">
  <number>103.0</number>
    </var>
- <var name="objectType">
  <string>transition</string>
    </var>
- <var name="sid">
  <number>1219.0</number>
    </var>
- <var name="toTasks">
  <number>105.0</number>
    </var>
- <var name="completionType">
  <string />
    </var>
- <var name="id">
  <number>104.0</number>
    </var>
    </struct>
    </array>
    </var>
- <var name="y">
  <number>151.0</number>
    </var>
- <var name="x">
  <number>451.0</number>
```

```
    </var>
- <var name="title">
  <string>Q & A</string>
    </var>
- <var name="libId">
  <string>questionanswer</string>
    </var>
- <var name="description">
  <string>Q & A</string>
    </var>
- <var name="firstTask">
  <number>103.0</number>
    </var>
    </struct>
- <struct>
- <var name="objectType">
  <string>Activity</string>
    </var>
- <var name="id">
  <number>31.0</number>
    </var>
- <var name="sid">
  <number>1212.0</number>
    </var>
- <var name="nextTransition">
  <number>109.0</number>
    </var>
- <var name="tasksTransitions">
- <array length="1">
- <struct>
- <var name="inputContent">
  <number>65.0</number>
```

```
    </var>
- <var name="toolType">
  <string>noticeboard</string>
    </var>
- <var name="objectType">
  <string>task</string>
    </var>
- <var name="id">
  <number>64.0</number>
    </var>
- <var name="sid">
  <number>1213.0</number>
    </var>
- <var name="grouping">
  <string>c</string>
    </var>
- <var name="nextTransition">
  <number>-1.0</number>
    </var>
- <var name="completion">
  <string>learner</string>
    </var>
- <var name="outputContent">
  <number>-1.0</number>
    </var>
- <var name="title">
  <string>Noticeboard</string>
    </var>
- <var name="description">
  <string>Display text</string>
    </var>
    </struct>
```

```
    </array>
    </var>
- <var name="y">
  <number>151.0</number>
    </var>
- <var name="x">
  <number>121.0</number>
    </var>
- <var name="title">
  <string>Noticeboard</string>
    </var>
- <var name="libId">
  <string>noticeboard</string>
    </var>
- <var name="description">
  <string>Read noticeboard</string>
    </var>
- <var name="firstTask">
  <number>64.0</number>
    </var>
    </struct>
- <struct>
- <var name="objectType">
  <string>Activity</string>
    </var>
- <var name="id">
  <number>66.0</number>
    </var>
- <var name="sid">
  <number>1214.0</number>
    </var>
- <var name="nextTransition">
```

```
<number>110.0</number>
</var>
- <var name="tasksTransitions">
- <array length="1">
- <struct>
- <var name="inputContent">
<number>100.0</number>
</var>
- <var name="toolType">
<string>noticeboard</string>
</var>
- <var name="objectType">
<string>task</string>
</var>
- <var name="id">
<number>99.0</number>
</var>
- <var name="sid">
<number>1215.0</number>
</var>
- <var name="grouping">
<string>c</string>
</var>
- <var name="nextTransition">
<number>-1.0</number>
</var>
- <var name="completion">
<string>learner</string>
</var>
- <var name="outputContent">
<number>-1.0</number>
</var>
```

```
- <var name="title">
  <string>Noticeboard</string>
    </var>
- <var name="description">
  <string>Display text</string>
    </var>
    </struct>
    </array>
    </var>
- <var name="y">
  <number>151.0</number>
    </var>
- <var name="x">
  <number>301.0</number>
    </var>
- <var name="title">
  <string>Noticeboard</string>
    </var>
- <var name="libId">
  <string>noticeboard</string>
    </var>
- <var name="description">
  <string>Read noticeboard</string>
    </var>
- <var name="firstTask">
  <number>99.0</number>
    </var>
    </struct>
- <struct>
- <var name="fromTasks">
  <number>66.0</number>
    </var>
```

```
- <var name="objectType">
  <string>transition</string>
    </var>
- <var name="sid">
  <number>1220.0</number>
    </var>
- <var name="toTasks">
  <number>101.0</number>
    </var>
- <var name="completionType">
  <string />
    </var>
- <var name="id">
  <number>110.0</number>
    </var>
    </struct>
- <struct>
- <var name="fromTasks">
  <number>31.0</number>
    </var>
- <var name="objectType">
  <string>transition</string>
    </var>
- <var name="sid">
  <number>1221.0</number>
    </var>
- <var name="toTasks">
  <number>66.0</number>
    </var>
- <var name="completionType">
  <string />
    </var>
```

```
- <var name="id">
  <number>109.0</number>
    </var>
    </struct>
    </array>
    </var>
- <var name="writeAccessDesc">
  <string>Private</string>
    </var>
- <var name="readAccessDesc">
  <string>Private</string>
    </var>
- <var name="lamsVersion">
  <string>1.0.2</string>
    </var>
- <var name="writeAccessStatus">
  <boolean value="true" />
    </var>
- <var name="readAccessStatus">
  <boolean value="true" />
    </var>
- <var name="writeAccess">
  <number>-1.0</number>
    </var>
- <var name="maxId">
  <number>110.0</number>
    </var>
- <var name="content">
- <array length="4">
- <struct>
- <var name="objectType">
  <string>content</string>
```

```
    </var>
- <var name="id">
  <number>108.0</number>
    </var>
- <var name="contentDefineLater">
  <boolean value="false" />
    </var>
- <var name="body">
  <string />
    </var>
- <var name="contentShowUser">
  <boolean value="false" />
    </var>
- <var name="isHTML">
  <boolean value="true" />
    </var>
- <var name="title">
  <string>Answers to Q & A</string>
    </var>
- <var name="description">
  <string />
    </var>
- <var name="contentType">
  <string>standard</string>
    </var>
- <var name="isReusable">
  <boolean value="false" />
    </var>
    </struct>
- <struct>
- <var name="objectType">
  <string>content</string>
```

```
    </var>
- <var name="id">
  <number>100.0</number>
    </var>
- <var name="contentDefineLater">
  <boolean value="false" />
    </var>
- <var name="body">
  <string><p><font face="Verdana" size="2">Los limones viven bajo el
      cielo</font></p></string>
    </var>
- <var name="contentShowUser">
  <boolean value="false" />
    </var>
- <var name="isHTML">
  <boolean value="true" />
    </var>
- <var name="title">
  <string>Limones</string>
    </var>
- <var name="description">
  <string />
    </var>
- <var name="contentType">
  <string>standard</string>
    </var>
- <var name="isReusable">
  <boolean value="false" />
    </var>
    </struct>
- <struct>
- <var name="objectType">
```

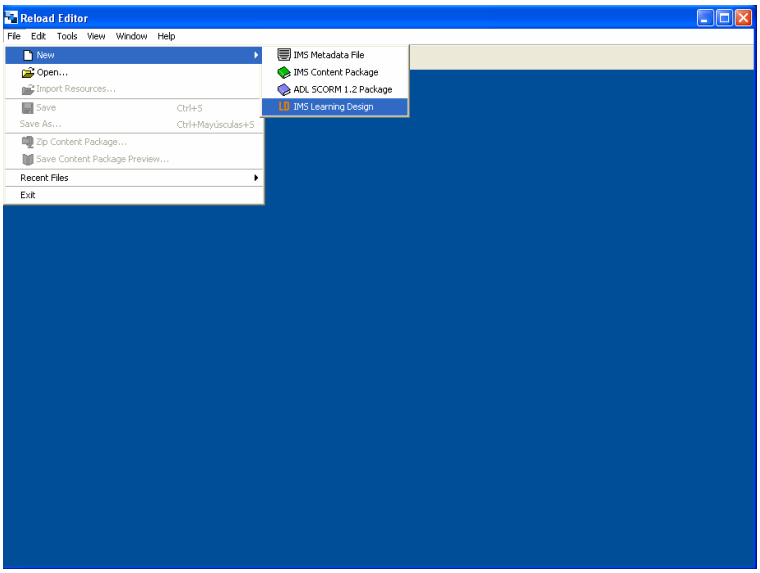
<string>content</string>
</var>
- <var name="id">
<number>107.0</number>
</var>
- <var name="contentDefineLater">
<boolean value="false" />
</var>
- <var name="body">
<string>Donde viven las patatas</string>
</var>
- <var name="contentShowUser">
<boolean value="false" />
</var>
- <var name="isHTML">
<boolean value="false" />
</var>
- <var name="title">
<string>Preguntas</string>
</var>
- <var name="description">
<string />
</var>
- <var name="contentType">
<string>standard</string>
</var>
- <var name="isReusable">
<boolean value="false" />
</var>
</struct>
- <struct>
- <var name="objectType">

<string>content</string>
</var>
- <var name="id">
<number>65.0</number>
</var>
- <var name="contentDefineLater">
<boolean value="false" />
</var>
- <var name="body">
<string> Las patatas viven bajo
tierra </string>
</var>
- <var name="contentShowUser">
<boolean value="false" />
</var>
- <var name="isHTML">
<boolean value="true" />
</var>
- <var name="title">
<string>patatas</string>
</var>
- <var name="description">
<string />
</var>
- <var name="contentType">
<string>standard</string>
</var>
- <var name="isReusable">
<boolean value="false" />
</var>
</struct>
</array>

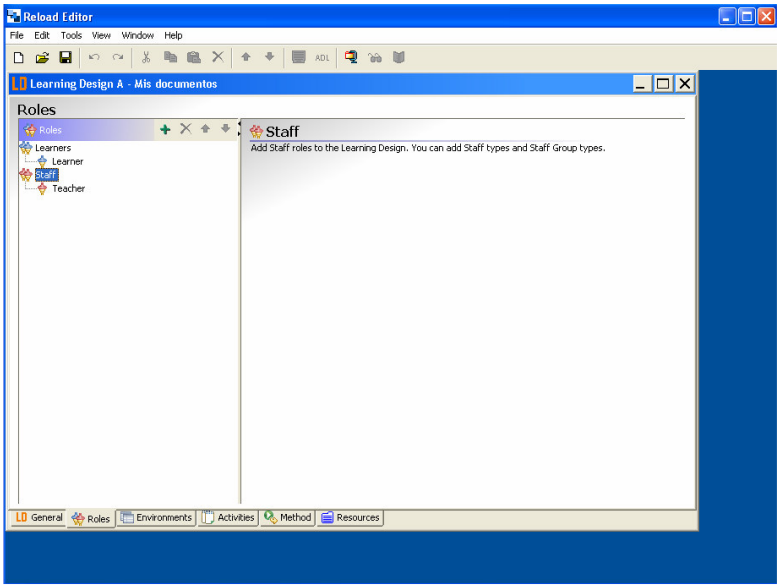
```
    </var>
- <var name="firstActivity">
  <number>31.0</number>
    </var>
- <var name="title">
  <string>Lams</string>
    </var>
- <var name="readAccess">
  <number>-1.0</number>
    </var>
- <var name="readOnly">
  <boolean value="false" />
    </var>
- <var name="description">
  <string />
    </var>
    </struct>
    </data>
    </wddxPacket>
```

RELOAD LD Editor

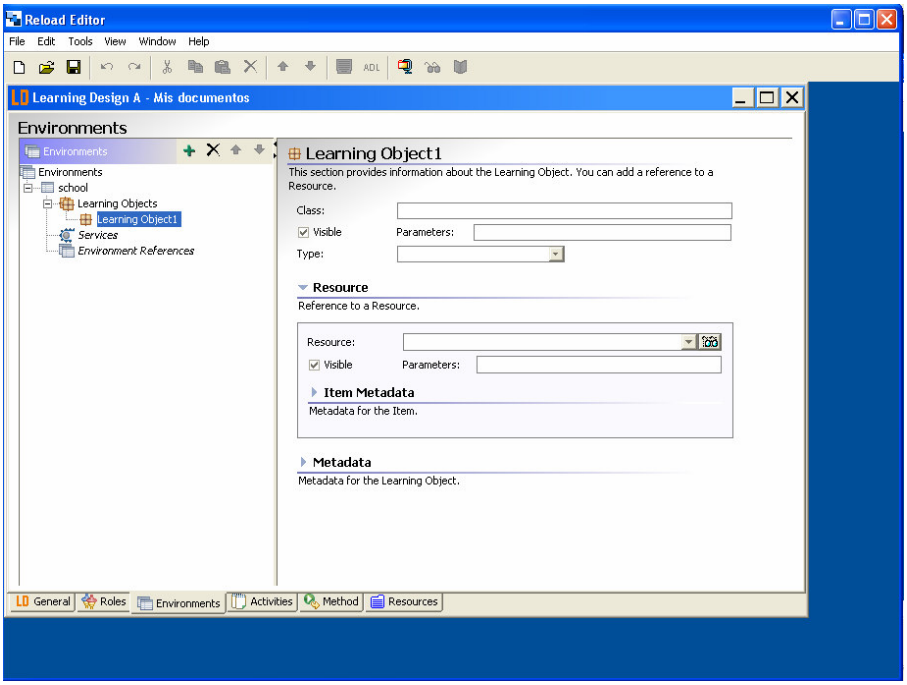
Para comenzar a usar Reload tenemos que seleccionar un nuevo proyecto de IMS Learning Design.



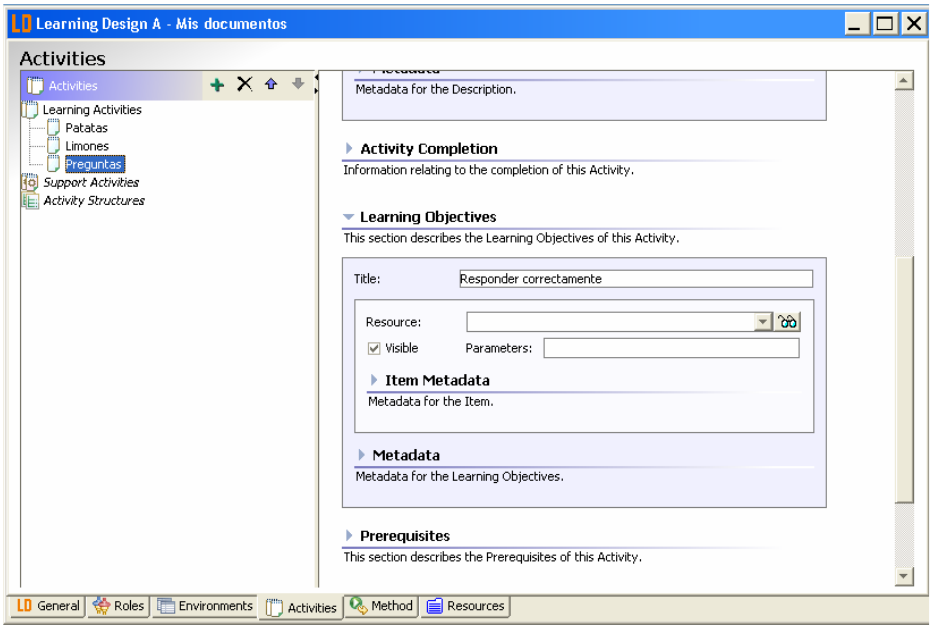
A continuación aparecen unas pestañas que se corresponden con los elementos del LD: Roles, Environments, Activities, etc. Donde iremos definiendo lo que nos haga falta. En el ejemplo de la imagen, se nos muestran los Roles y nos da la opción de crear nuevos.



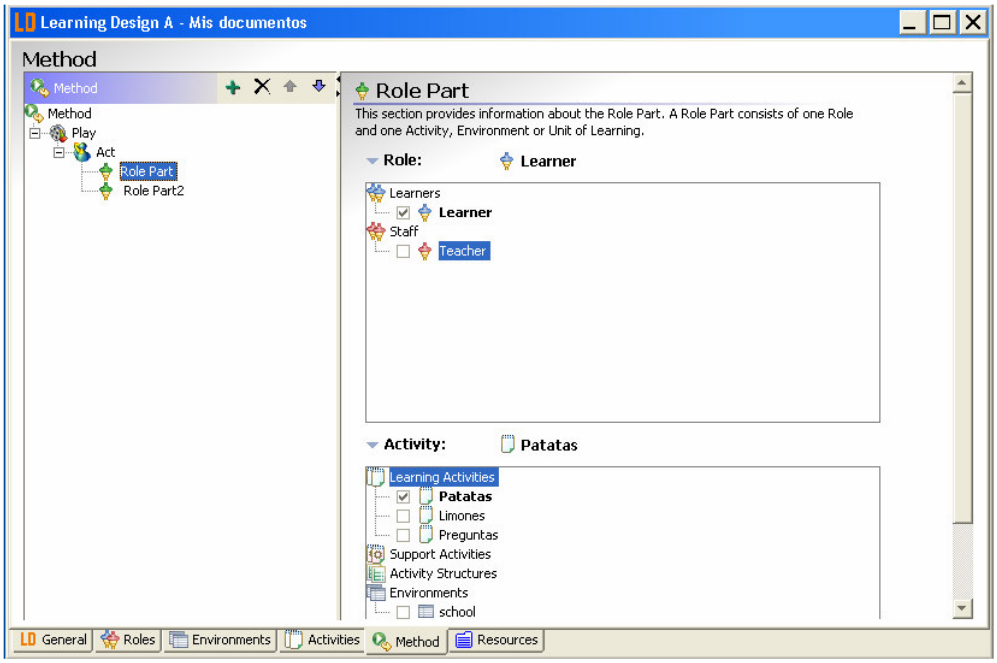
En la pestaña de Environments, cuando creamos uno, nos aparecen las diferentes posibilidades que tenemos, como definir un Learning Object asociado a un recurso.



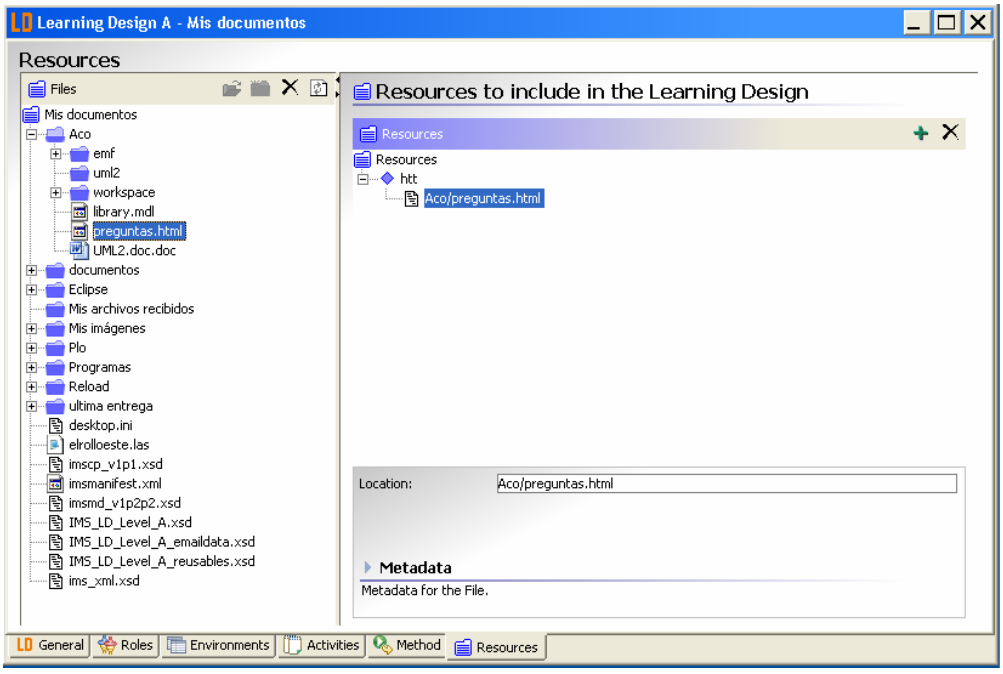
En Activities crearemos las actividades y pondremos su descripción, objetivos, etc



En la pestaña del Método, definiremos cuántos Acts tiene el Play, y crearemos los Role Part que asociaran cada rol con la actividad o actividades que le correspondan.



Finalmente la pestaña de Resources nos permite importar documentos, imágenes y otro tipo de recursos desde nuestro disco. Un recurso también puede ser una referencia a una URL externa.



Código generado por el Reload LD Editor:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <!-- This is a Reload version 2.0.2 Learning Design document  -->
<!-- Spawned from the Reload Learning Design
Generator - http://www.reload.ac.uk  -->
- <manifest
        xmlns=http://www.imsglobal.org/xsd/imscp_v1p1
        xmlns:imsmd="http://www.imsglobal.org/xsd/imsmd_v1p2"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:imsld="http://www.imsglobal.org/xsd/imsld_v1p0"
        identifier="MANIFEST-1B483663-3660-3B44-3611-4E8B1DAC8B43"
        xsi:schemaLocation="http://www.imsglobal.org/xsd/imscp_v1p1
        imscp_v1p1.xsd http://www.imsglobal.org/xsd/imsmd_v1p2
        imsmd_v1p2p2.xsd http://www.imsglobal.org/xsd/imsld_v1p0
        IMS_LD_Level_A.xsd">
- <organizations>
- <imsld:learning-design identifier="LD-27FB1837-78F1-E54B-F62C-
A008C3C85763" level="A" uri="">
- <imsld:components>
- <imsld:roles identifier="LD-44F690C8-56BF-BBDC-07C8-F29D3389806F">
- <imsld:learner identifier="LD-48CDC567-2A18-5176-0BDB-0F20DD74FE7F">
  <imsld:title>Learner</imsld:title>
  </imsld:learner>
- <imsld:staff identifier="LD-44C95DF8-F8FE-B648-6592-6921ED838972">
  <imsld:title>Teacher</imsld:title>
  </imsld:staff>
  </imsld:roles>
- <imsld:activities>
- <imsld:learning-activity identifier="LD-E40451FD-4666-D877-DDF9-
2B15E47124A6">
  <imsld:title>Patatas</imsld:title>
```

- <imsld:learning-objectives>
<imsld:title>Aprender mas acerca de las patatas</imsld:title>
<imsld:item />
</imsld:learning-objectives>
- <imsld:activity-description>
<imsld:title>La vida de las patatas</imsld:title>
<imsld:item />
</imsld:activity-description>
</imsld:learning-activity>
- <imsld:learning-activity identifier="LD-55E9E412-5FB4-8900-318A-905BE22E062A">
<imsld:title>Limones</imsld:title>
- <imsld:learning-objectives>
<imsld:title>Aprender mas sobre la vida de los limones</imsld:title>
<imsld:item />
</imsld:learning-objectives>
- <imsld:activity-description>
<imsld:title>La vida de los limones</imsld:title>
<imsld:item />
</imsld:activity-description>
</imsld:learning-activity>
- <imsld:learning-activity identifier="LD-F6D5AAF2-1796-4220-519E-FA9611C26F6B">
<imsld:title>Preguntas</imsld:title>
- <imsld:learning-objectives>
<imsld:title>Responder correctamente</imsld:title>
<imsld:item />
</imsld:learning-objectives>
- <imsld:activity-description>
<imsld:title>Pregunta sobre las patatas</imsld:title>
<imsld:item />
</imsld:activity-description>

```
  </imsld:learning-activity>
  </imsld:activities>
- <imsld:environments>
- <imsld:environment identifier="LD-CB36F7DC-7B70-C747-7CFD-
E91DDCBB8EC9">
  <imsld:title>school</imsld:title>
- <imsld:learning-object identifier="LD-B3CD1E1C-A32F-9884-8AB7-
73AF35B63830">
  <imsld:title>Learning Object1</imsld:title>
  <imsld:item />
  </imsld:learning-object>
  </imsld:environment>
  </imsld:environments>
  </imsld:components>
- <imsld:method>
- <imsld:play identifier="LD-93B98635-438E-463A-8CD7-D3F90C10A5B5">
  <imsld:title>Play</imsld:title>
- <imsld:act identifier="LD-B87408CA-CD9F-9AD3-7456-0E72B1D7CE48">
  <imsld:title>Act</imsld:title>
- <imsld:role-part identifier="LD-60D08B43-7B4B-8A20-21AD-7C6E30043552">
  <imsld:title>Role Part</imsld:title>
  <imsld:role-ref ref="LD-48CDC567-2A18-5176-0BDB-0F20DD74FE7F" />
  <imsld:learning-activity-ref ref="LD-E40451FD-4666-D877-DDF9-
2B15E47124A6" />
  </imsld:role-part>
- <imsld:role-part identifier="LD-24D22EE3-852A-58CD-91F9-
FA0CFE10BE43">
  <imsld:title>Role Part2</imsld:title>
  <imsld:role-ref ref="LD-44C95DF8-F8FE-B648-6592-6921ED838972" />
  </imsld:role-part>
  </imsld:act>
  </imsld:play>
```

```
  </imsld:method>
  </imsld:learning-design>
  </organizations>
- <resources>
- <resource identifier="RES-59F87950-2814-4381-9993-813478795B43"
type="webcontent" href="Aco/preguntas.html">
  <file href="Aco/preguntas.html" />
  </resource>
  </resources>
  </manifest>
```

Comparativa entre nuestro proyecto y ReloadLD Editor

Hemos desarrollado un ejemplo sencillo tanto en Reload como en nuestro editor:

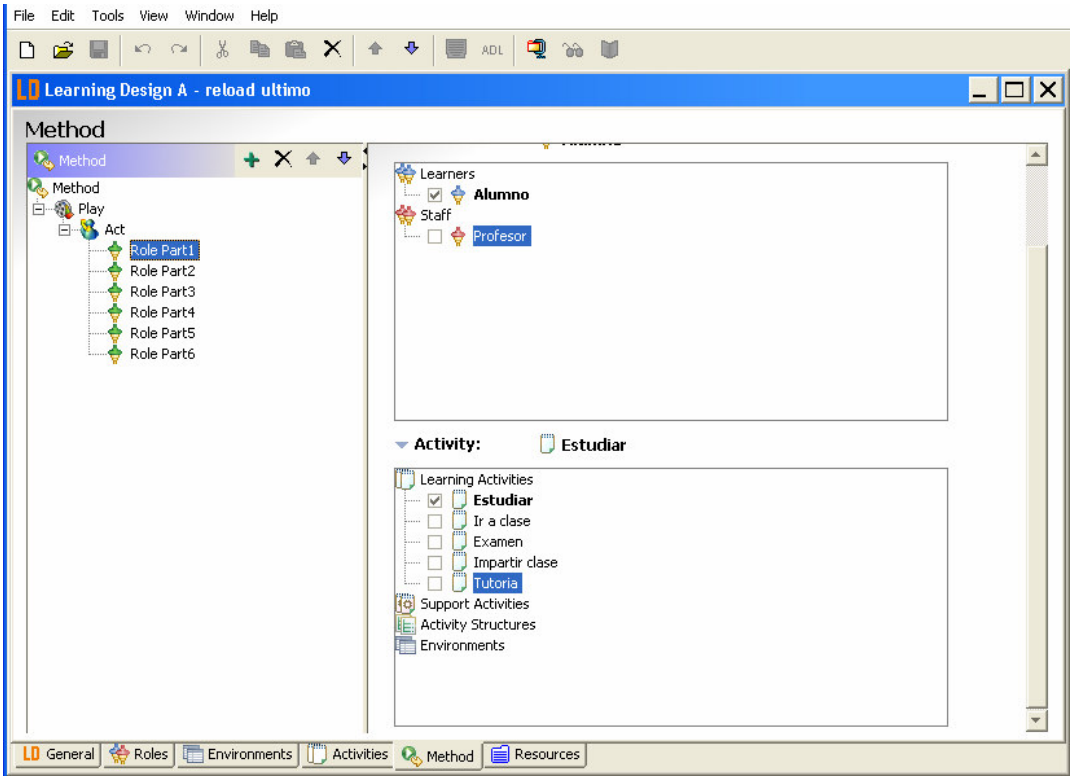


Figura 1: Reload

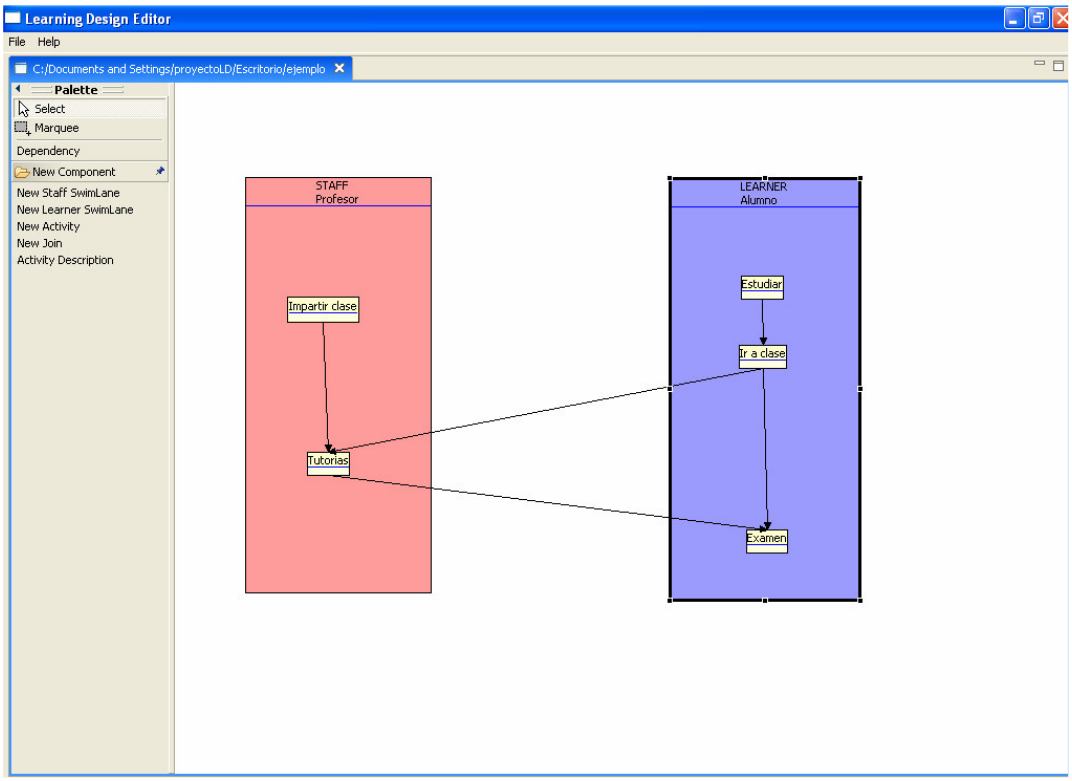


Figura 2: Nuestro Editor

Como podemos observar nuestro editor ofrece una interfaz más amigable al permitir la edición gráfica de los elementos de LD.

No obstante el producto que ambos programas exportan en formato XML es muy similar.

Documento Reload:

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

- <!--

This is a Reload version 2.0.2 Learning Design document

-->

- <!--

Spawned from the Reload Learning Design Generator - http://www.reload.ac.uk

-->

- <manifest xmlns="http://www.imsglobal.org/xsd/imscp_v1p1"

xmlns:imsmd="http://www.imsglobal.org/xsd/imsmd_v1p2"

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:imsld="http://www.imsglobal.org/xsd/imsld_v1p0" identifier="MANIFEST-
    560C75A6-5E22-92F9-667A-377FA08997D2"
    xsi:schemaLocation="http://www.imsglobal.org/xsd/imscp_v1p1 imscp_v1p1.xsd
    http://www.imsglobal.org/xsd/imsmd_v1p2 imsmd_v1p2p2.xsd
    http://www.imsglobal.org/xsd/imsld_v1p0 IMS_LD_Level_A.xsd">
- <organizations>
- <imsld:learning-design identifier="LD-42916C24-DCD3-CC36-91C9-
    7A4CDACDB26F" level="A" uri="">
- <imsld:components>
- <imsld:roles identifier="LD-11FF8F2B-D11C-CB75-5C8A-4FEC294FD765">
- <imsld:learner identifier="LD-2090F1A3-1A6B-9B2A-8012-0931FECB9801">
  <imsld:title>Alumno</imsld:title>
    </imsld:learner>
- <imsld:staff identifier="LD-DEF59483-CD5D-376C-42CE-CAE7E3562058">
  <imsld:title>Profesor</imsld:title>
    </imsld:staff>
    </imsld:roles>
- <imsld:activities>
- <imsld:learning-activity identifier="LD-866CC6F9-9B3D-FA42-920A-
    AD011F5670AF">
  <imsld:title>Estudiar</imsld:title>
- <imsld:activity-description>
  <imsld:title>Repasar en casa</imsld:title>
  <imsld:item />
    </imsld:activity-description>
    </imsld:learning-activity>
- <imsld:learning-activity identifier="LD-286D4DEB-3DC3-9B60-E4A5-
    ABAD958D22DE">
  <imsld:title>Ir a clase</imsld:title>
- <imsld:activity-description>
  <imsld:title>Asistir a clase</imsld:title>
```

<imsld:item />
</imsld:activity-description>
</imsld:learning-activity>
- <imsld:learning-activity identifier="LD-54C197E7-19DD-7F05-B989-2E6A0DF8AC97">
<imsld:title>Examen</imsld:title>
- <imsld:activity-description>
<imsld:item />
</imsld:activity-description>
</imsld:learning-activity>
- <imsld:learning-activity identifier="LD-84BAB53F-99F6-00EE-31DD-45298AC41ACE">
<imsld:title>Impartir clase</imsld:title>
- <imsld:activity-description>
<imsld:item />
</imsld:activity-description>
</imsld:learning-activity>
- <imsld:learning-activity identifier="LD-F73BD94A-74CE-D58B-5EC1-807959706487">
<imsld:title>Tutoria</imsld:title>
- <imsld:activity-description>
<imsld:item />
</imsld:activity-description>
</imsld:learning-activity>
</imsld:activities>
</imsld:components>
- <imsld:method>
- <imsld:play identifier="LD-191937E3-CC7A-3E07-661C-904D8E67A39C">
<imsld:title>Play</imsld:title>
- <imsld:act identifier="LD-4D03A691-A6AB-F9FE-75B9-26B94FF60381">
<imsld:title>Act</imsld:title>
- <imsld:role-part identifier="LD-81EDDD8E-0C13-4168-31D6-3BA2A531E0B8">

<imsld:title>Role Part1</imsld:title>
<imsld:role-ref ref="LD-2090F1A3-1A6B-9B2A-8012-0931FECB9801" />
<imsld:learning-activity-ref ref="LD-866CC6F9-9B3D-FA42-920A-AD011F5670AF" />
</imsld:role-part>
- <imsld:role-part identifier="LD-FA741865-A39B-5717-5E1F-5CD2E9ECE32C">
<imsld:title>Role Part2</imsld:title>
<imsld:role-ref ref="LD-2090F1A3-1A6B-9B2A-8012-0931FECB9801" />
<imsld:learning-activity-ref ref="LD-286D4DEB-3DC3-9B60-E4A5-ABAD958D22DE" />
</imsld:role-part>
- <imsld:role-part identifier="LD-E5DB2491-2B98-BD92-5307-A2306E1A2597">
<imsld:title>Role Part3</imsld:title>
<imsld:role-ref ref="LD-2090F1A3-1A6B-9B2A-8012-0931FECB9801" />
<imsld:learning-activity-ref ref="LD-54C197E7-19DD-7F05-B989-2E6A0DF8AC97" />
</imsld:role-part>
- <imsld:role-part identifier="LD-7F6349B2-1971-3E6E-1F07-567984E35780">
<imsld:title>Role Part4</imsld:title>
<imsld:role-ref ref="LD-DEF59483-CD5D-376C-42CE-CAE7E3562058" />
<imsld:learning-activity-ref ref="LD-84BAB53F-99F6-00EE-31DD-45298AC41ACE" />
</imsld:role-part>
- <imsld:role-part identifier="LD-408A78D9-A8DB-A967-960C-CD7FD11DD6FB">
<imsld:title>Role Part5</imsld:title>
<imsld:role-ref ref="LD-DEF59483-CD5D-376C-42CE-CAE7E3562058" />
<imsld:learning-activity-ref ref="LD-F73BD94A-74CE-D58B-5EC1-807959706487" />
</imsld:role-part>
- <imsld:role-part identifier="LD-9EBE7140-78B4-FE96-2E0A-F1329DDAE4F3">
<imsld:title>Role Part6</imsld:title>
<imsld:role-ref ref="LD-2090F1A3-1A6B-9B2A-8012-0931FECB9801" />

```
<imsld:learning-activity-ref ref="LD-F73BD94A-74CE-D58B-5EC1-807959706487"
    />
   </imsld:role-part>
   </imsld:act>
   </imsld:play>
   </imsld:method>
   </imsld:learning-design>
   </organizations>
<resources />
   </manifest>
```

Nuestro documento:

```
 <?xml version="1.0" encoding="UTF-8" ?>
- <learning-design>
- <components>
- <roles>
- <staff id="__Q85AAgvEduECdwQqoc-Tg">
 <title>STAFF Profesor</title>
   </staff>
- <learner id="__Q85BQgvEduECdwQqoc-Tg">
 <title>LEARNER Alumno</title>
   </learner>
   </roles>
- <activities>
- <ld_activity id="__Q85AQgvEduECdwQqoc-Tg">
 <title>Impartir clase</title>
   </ld_activity>
- <ld_activity id="__Q85AggvEduECdwQqoc-Tg">
 <title>Tutorias</title>
   </ld_activity>
- <ld_activity id="__Q85BggvEduECdwQqoc-Tg">
```

```
  <title>Estudiar</title>
    </ld_activity>
- <ld_activity id="__Q85BwgvEduECdwQqoc-Tg">
  <title>Ir a clase</title>
    </ld_activity>
- <ld_activity id="__Q85CAgvEduECdwQqoc-Tg">
  <title>Examen</title>
    </ld_activity>
    </activities>
    </components>
- <method>
- <play id="1">
  <title>Play</title>
- <act id="1">
  <title>Act</title>
- <role-part id="1">
  <title>RolePart</title>
  <role-ref>__Q85AAgvEduECdwQqoc-Tg</role-ref>
  <ld_activity-ref>__Q85AQgvEduECdwQqoc-Tg</ld_activity-ref>
    </role-part>
- <role-part id="2">
  <title>RolePart</title>
  <role-ref>__Q85AAgvEduECdwQqoc-Tg</role-ref>
  <ld_activity-ref>__Q85AggvEduECdwQqoc-Tg</ld_activity-ref>
    </role-part>
- <role-part id="2">
  <title>RolePart</title>
  <role-ref>__Q85BQgvEduECdwQqoc-Tg</role-ref>
  <ld_activity-ref>__Q85BggvEduECdwQqoc-Tg</ld_activity-ref>
    </role-part>
- <role-part id="3">
  <title>RolePart</title>
```

<role-ref>__Q85BQgvEduECdwQqoc-Tg</role-ref>
<ld_activity-ref>__Q85BwgvEduECdwQqoc-Tg</ld_activity-ref>
</role-part>
- <role-part id="4">
<title>RolePart</title>
<role-ref>__Q85BQgvEduECdwQqoc-Tg</role-ref>
<ld_activity-ref>__Q85CAgvEduECdwQqoc-Tg</ld_activity-ref>
</role-part>
</act>
</play>
</method>
</learning-design>

Bibliografía

- www.imsglobal.org/learningdesign/index.html
- www.eclipse.org/gef
- www.128.ibm.com/developerworks/opensource/library/os-gef/
- http://www.eclipse.org/articles/Article-GEF-EMF/gef-emf.html
- Eclipse Development using gef & emf, IBM Redbooks.
- www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html
- www.eclipse.org/rcp
- www.eclipse.org/emf
- www.eclipse.org/uml2
- www.sankhya.com/articles/eclipse-uml.html
- Addison.Wesley.Eclipse.Rich.Client.Platform.Designing.Coding.and.chm
- Addison.Wesley.-.Contributing.To.Eclipse.Principles,.Patterns,.A.chm
- Rich Client Tutorial, by Ed Burnette, SAS.
- Eclipse Rich Client Platform FAQ.
- www.benjaminbooth.com/tableorbooth/2005/05/painfree_rcp_in.html
- http://www.w3schools.com/dom/dom_intro.asp
- http://www.jdom.org/downloads/docs.html
- http://javahispano.org/articles.article.action?id=50
- http://es.wikipedia.org/wiki/JDOM
- http://www.javahispano.org/articles.article.action?id=50

Los alumnos Marco Antonio Hernández Gómez, Esther Picó Sanchis y Laura Rodríguez Sánchez, autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales, tanto la memoria como el código, la documentación y el prototipo desarrollado.

Fdo. Marco Antonio Hernández Gómez

Fdo. Esther Picó Sanchis

Fdo. Laura Rodríguez Sánchez