E-EDD: INTEGRACIÓN EN ECLIPSE DEL DEPURADOR DECLARATIVO PARA ERLANG EDD

JOEL SÁNCHEZ PEDROZA

MÁSTER EN INGENIERÍA INFORMÁTICA. FACULTAD DE INFORMÁTICA.
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Máster en Ingeniería Informática

Convocatoria: Septiembre

Calificación obtenida: Sobresaliente (9)

07/09/2015

Directores:

Adrián Riesco Rodríguez / Salvador Tamarit Muñoz

Autorización de Difusión

JOEL SÁNCHEZ PEDROZA

07/09/2015

El abajo firmante, matriculado en el Máster en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: “E-EDD: Integración en Eclipse del Depurador Declarativo para Erlang EDD”, realizado durante el curso académico 2014-2015 bajo la dirección de Adrián Riesco Rodríguez y Salvador Tamarit Muñoz; en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

El presente trabajo tiene como finalidad el proporcionar una interfaz de depuración declarativa de fácil uso e interacción para programadores Erlang.

La idea viene promovida por el desarrollo inicial del proyecto *Erlang Declarative Debugging* (EDD) a cargo de Rafael Caballero, Enrique Martín-Martín, Adrián Riesco y Salvador Tamarit, quienes llevan tiempo trabajando en la investigación y la implementación de diversas técnicas para la depuración declarativa. Su campo de trabajo es un proyecto que incluye una serie de algoritmos que, en conjunto, ofrecen una herramienta de depuración declarativa para programas Erlang.

El presente trabajo tiene, por tanto, el objetivo de enriquecer dicha herramienta, además de intentar una mayor divulgación. Para ello, es necesario conseguir un mayor acercamiento a los desarrolladores y motivarles a hacer uso de la herramienta desde una perspectiva menos abstracta, permitiendo así que su uso pueda estar presente durante las fases de desarrollo.

De aquí nace la idea de Eclipse-EDD (E-EDD), un proyecto desarrollado en tecnología Eclipse, que es un IDE de gran aceptación, versatilidad y potencia ya que, mediante el desarrollo de plugins, permite que puedan integrarse extensiones a la herramienta con la finalidad de adaptarla a las necesidades especificas de los usuarios.

E-EDD permitirá dar una visión alternativa a los programadores Erlang (en lo que a la depuración se refiere) respecto a la forma de depuración tradicional muy presente en los lenguajes imperativos como el uso de *breakpoints,* ya que la inspección de variables no tiene sentido en la programación funcional.

Por tanto, lo que se ofrece con el presente proyecto es mejorar la usabilidad de EDD mediante una herramienta gráfica que mejore la experiencia del usuario en las arduas labores de depuración declarativa, además de buscar promover su uso y aceptación ofreciendo un entorno simple y amigable.

Palabras clave

Eclipse, Erlang, Depuración declarativa.

Abstract

This paper aims to provide a declarative debugging user interface of easy interaction and use for Erlang programmers.

The idea is promoted by the initial development of the Erlang Declarative Debugging (EDD) project by Rafael Caballero, Enrique Martin-Martin, Adrián Riesco, and Salvador Tamarit who have been working in the research and implementation of several techniques for early detection of errors in declarative programming. Their field of work includes a series of algorithms that offer together a declarative debugging tool for Erlang programmers.

The main idea of this paper is therefore to enrich this tool, and to increase its disclosure. To achieve these objectives, it is necessary to get closer to developers and encourage them to make use of the tool from a less abstract perspective, allowing its use may be present during the development stages.

The Eclipse-EDD (E-EDD) project that was born from these ideas is developed under Eclipse technology, a widely accepted IDE, versatile, and powerful, because developing plugins with their respective extension points which allows to customize the tool to the specific needs of users.

E-EDD gives an alternative vision to Erlang programmers (regarding debugging) over the traditional debugging way very present in imperative languages, such as using breakpoints, where inspecting variable makes no sense in functional programming.

Therefore, this project improves the usability of EDD with a graphical tool that enhances the user experience in the arduous task of declarative debugging, and seek to promote their use and acceptance by offering a simple and friendly environment.

Keywords

Eclipse, Erlang, Declarative debugging.

Índice de contenidos

Agradecimientos

En primer lugar me gustaría agradecer tanto a Salvador Tamarit como Adrián Riesco, mis tutores, el tiempo, la paciencia, el compromiso y el esfuerzo brindado durante el proceso y maduración del presente trabajo; sin ellos no hubiera sido posible el presente proyecto.

Gracias a sus labores de investigación y a sus conocimientos en el área de programación declarativa es que ha surgido la idea aquí presentada; para mí ha sido muy satisfactorio el poder aportar algo en su área de trabajo. Gracias por compartir sus líneas de investigación, sus aportaciones e ideas.

De igual forma quiero darle las gracias a María, mi compañera, ya que ha sido una pieza fundamental durante mi incursión en el máster; es por su apoyo incondicional que he podido seguir adelante. Muchas gracias por estar ahí todo este tiempo.

Finalmente, agradecer el esfuerzo constante, progresivo e ilimitado de mi familia; por ellos estoy donde estoy. Les estaré eternamente agradecido.

1. Introducción

El presente trabajo tiene como finalidad ofrecer una interfaz de depuración declarativa sobre Eclipse [1] para desarrolladores Erlang [2] basada en EDD, de tal forma que será posible depurar módulos Erlang aplicando depuración declarativa con todas las facilidades que ofrece el entorno de desarrollo Eclipse, una herramienta muy popular entre desarrolladores que goza además del privilegio de tener una amplia comunidad de usuarios que extienden constantemente las áreas de aplicación bajo dicha plataforma.

Una de las principales razones por las que hemos decidido tomar como referente el Entorno de Desarrollo Integrado (IDE) de Eclipse porque proporciona servicios integrales que facilitan el desarrollo de software a los programadores, además de permitir contribuciones y extensiones al mismo mediante el desarrollo de conectores o plugins. Con ello podremos establecer un entorno visual gráfico sobre el cual montaremos una interfaz de depuración acorde a las necesidades específicas del presente proyecto, que iremos desglosando a lo largo de este documento.

1.1. Motivación

La idea de realizar este proyecto surgió a raíz de conocer el trabajo realizado por los desarrolladores de EDD [3] durante una presentación en las aulas de la UCM. La charla surgió como refuerzo a la materia de Programación Declarativa Aplicada, en la cual se ejemplificaba, mediante un caso práctico, un área de trabajo que integra dicha tecnología.

Así pues, una vez conocido el proyecto y su campo de acción, surgió la idea de contribuir a mejorar dicha herramienta con una interfaz gráfica. Entre los motivos que nos impulsaron a ello está el hecho de ver un potencial oculto que podía ser mejorado para lograr mayor visibilidad y poder presentarlo de una forma menos abstracta, además de mejorar sustancialmente su forma de operar ya que el uso en modo consola no es precisamente fácil.

Si bien es cierto que es posible trabajar con entornos simples y limitados, muchos desarrolladores agradecen el uso de entornos amigables que les faciliten la programación. Hoy en día trabajar con un bloc de notas no es muy práctico cuando existen numerosas herramientas que facilitan el desarrollo pero es un uso que está generalizado en la comunidad de desarrolladores que usan lenguajes funcionales, quizás debido a la poca aceptación que se ha dado a este

paradigma de programación declarativa, lo cual ha impedido el desarrollo de entornos más amigables a lo largo de su historia: al gozar de menor popularidad se destinaban menos esfuerzos en mejoras y difusión, de forma que quienes lo usaban se limitaban a utilizarlo como se distribuía, bajo versiones más austeras en comparación a lo que se ofrecía con el modelo de programación imperativo.

Los tiempos han cambiado y parece que justo ahora empieza a cobrar fuerza la programación declarativa (se debe mencionar que Java 8 incluye ya parte de este paradigma), por lo que esperamos que mejoren también las herramientas para su desarrollo.

Por tanto, la contribución que se persigue es justamente la de enriquecer lo que hasta ahora ofrece EDD pero desde una interfaz a más alto nivel, dado que con ello se da una alternativa a la forma de operar que se realiza en modo consola. Para ello se ha preseleccionado una serie de tecnologías que están muy presentes en el entorno Eclipse (que se expondrán más adelante); de esta forma es posible hacerlas converger en una única arquitectura a base de capas en donde las mejoras y aportaciones que se hagan no impidan al desarrollo de otras capas (consultar la sección 3.3 y la sección 6). El usuario siempre podrá optar por operar tanto a bajo como alto nivel indistintamente; con ello se pretende ganar más adeptos a la herramienta.

Es por ello que un IDE de desarrollo como Eclipse reúne las características idóneas para operar con el depurador EDD; así pues, el proyecto que de ahora en adelante llamaremos E-EDD (Eclipse-EDD) es el resultado de la suma de varias vistas, en donde cada una de ellas aporta contenido a la depuración y facilita el seguimiento del usuario durante el proceso de depuración.

1.2. Plan de trabajo

En lo que a gestión se refiere, la forma en que se ha venido desarrollando el presente proyecto tiene como referente el desarrollo ágil bajo Scrumban [L1] (Scrum y Kanban), dado que es el modelo de desarrollo que consideramos que se adaptaba mejor a las necesidades del proyecto, separando las actividades a realizar en pequeños subgrupos con la finalidad de ir superando hitos dentro de un flujo de trabajo gestionado mediante el uso de un tablero cuyas columnas (en cola, activas, terminadas, bloqueadas, etc.) describen las acciones a realizar y cuya trayectoria, vista de izquierda a derecha, permite hacer un seguimiento visual del avance conseguido.

Existen numerosas herramientas en el mercado que dan soporte a dicha metodología, entre ellas Trello [4], que ha sido utilizada para gestionar el presente proyecto, además de ser el medio que facilitó la comunicación entre los miembros del equipo mediante el sistema de menciones utilizado para mantener informados a los otros de los cambios acontecidos en la tarea.

En cuanto a la parte funcional, el proyecto ha ido madurando según se iba avanzando en el desarrollo, lo mismo que la forma de operar, puesto que, en sus inicios, comenzó como un *wizard* asistido y terminó como la suma de varias vistas agrupadas bajo una perspectiva propia en Eclipse.

El desarrollo inicial de E-EDD comenzó con el intercambio de información a través de un archivo json [5], que era el que poseía toda la información estática usada en la interacción Erlang-Java; después se observó que se necesitaba una parte dinámica, por lo que se procedió a trabajar en una comunicación bajo TCP/UDP y, finalmente, la necesidad latente de usar el API de comunicación que ofrece Erlang para trabajar con Java, es decir, JInterface [6]. Gracias a esto conseguimos sortear todo obstáculo, ya que era la forma más clara de establecer un canal de comunicación transparente de cara a lo que es el paso de mensajes entre ambas tecnologías.

Se ha creado un modelo jerárquico de clases que representan la información utilizada durante la comunicación Java-Erlang de manera que, navegando a través de dicha estructura, podemos acceder a un dato concreto durante la depuración.

El aspecto visual ofrecido está perfectamente integrado en el entorno de desarrollo de Eclipse, porque se utilizan las mismas tecnologías usadas por éste. Tal es el caso de SWT [7] y JFace [8], dos herramientas que ofrecen un conjunto de componentes para la construcción de interfaces gráficas bajo Java.

Posteriormente, dado que EDD ya trabajaba con archivos dot [A1] para el renderizado de grafos en imágenes, y como existe un plugin de terceros que lo implementa en Eclipse (Eclipse GraphViz [9]), se decidió seguir por esa misma línea para la creación de los árboles de ejecución usados durante la fase de depuración. La limitación de su uso surgió porque, aun cuando se editasen los archivos dot y se regenerasen las imágenes del modelo actualizado, no dejaba de ser un contenido estático, es decir, que no se puede hacer selección de nodos en las mismas, para ello hacía falta un editor gráfico. Como resultado se obtuvieron los editores creados a partir de tecnología Eclipse; con ello se consiguió obtener grafos dinámicos que permiten la manipulación de nodos en todo momento.

Otra línea de investigación que permanece abierta es la del uso y creación de diagramas de secuencia de forma dinámica; para ello se estudiaron dos tecnologías, principalmente UMLGraph [10] y Papyrus [11]. Su uso viene motivado porque EDD tiene un módulo de depuración concurrente cuya implementación no ha sido posible integrar en este proyecto pero cuya funcionalidad está contemplada igualmente.

El resto de la memoria se estructura como sigue: la sección 3 presenta los preliminares necesarios para comprender y contextualizar el desarrollo del proyecto. La sección 4 introduce al proyecto EDD y su forma de operar. La sección 5 detalla la propuesta presentada a nivel operativo mientras que en la sección 6 se detalla a nivel arquitectónico. La sección 7 detalla la comunicación realizada entre Erlang y Java y finalmente la sección 8 define a nivel modelado el esquema usado para la representación jerárquica y gráfica de la información en Eclipse.

2. Introduction

This paper aims to provide a declarative debugging user interface on Eclipse [1] for Erlang [2] developers based on EDD, so it will be possible to apply declarative debugging [A8],[A9] to Erlang modules with all the facilities offered by the Eclipse development environment, a very popular tool among developers who also enjoys the privilege of having a wide community of users who constantly extend the application areas under that platform.

We decided to use the Eclipse Integrated Development Environment (IDE) because it provides comprehensive services that facilitate the development of software programmers and allows contributions and extensions because by developing plugins it will be possible to establish a visual environment which works as a debugging interface according to the specific needs of this project, as we will show throughout this document.

2.1. Motivation

The idea for this project stems from knowing the work done by the developers of EDD [3] during a presentation at the UCM; the conference emerged as a reinforcement for Applied Declarative Programming, where EDD was exemplified by a practical case: an area of work that integrates the technology.

Hence, the idea was to improve their tool with a user interface. Among the observations that led me to collaborate was the fact that I saw a hidden potential that could be improved to achieve greater visibility, and to present EDD in a less abstract way. Moreover, I thought that a graphical user interface would improve its usability substantially.

While it is possible to work with simple and limited environments, many developers appreciate to use friendly development environments to facilitate their programming; nowadays programming with the notepad is not very practical since there are numerous tools that provide a better user experience. However, it seems that this problem is widespread in the community of developers using functional languages. We guess that the problem is due to the lack of acceptance that developers have given to this declarative programming paradigm, which has prevented the development of more friendly environments throughout its history: with a low acceptance low popularity and any efforts to try improvement and gain more diffusion was hard.

People who used it were limited to use as was distributed under more austere versions compared to what is offered with the imperative programming model.

Times have changed and it seems that just now declarative programming begins to gain acceptance (it is important to mention that Java 8 now includes part of this paradigm), so we also hope to improve the tools for its development.

The current contribution is precisely to enrich EDD from a higher level interface, given that this would give an alternative to the modus operandi which is done in console mode. To this end, it has shortlisted a number of technologies that are very present in the Eclipse environment (which will be discussed below); in this way it is possible to make them converge into a single architecture based on layers where improvements and contributions do not impede the development of other layers (see section 3.3 and section 6). Hence it is possible to work in both low and high level modes, indistinctly, gaining in this way more adepts for the tool.

This is why a development IDE like Eclipse combines the best features of existing work with EDD, the project that henceforth call E-EDD (Eclipse-EDD) is the result of the sum of several views, where each view provides content to facilitate the debugging and tracking to the user.

2.2. Working plan

As far as management is concerned, the project has been developing using agile development techniques concerning to Scrumban [L1] (Scrum and Kanban), because this is the development model that we think is better suited to the needs of the project, separating the activities to be performed in smaller groups with the aim of moving ahead in a managed workflow using a board whose columns (queued, active, finished, locked, etc.) describe the actions to be take and whose path, seen from left to right, allows to track the progress achieved.

There are many tools in the market that support this methodology, including Trello [4], which has been used to manage this project. Trello, besides being a medium that facilitates the communication between the team members involved in the project, includes a notification system that is used to keep the rest of the team informed of the changes occurring in the task.

Regarding the functional part, meanwhile development progressed, the project matured as well. We decided to change the way of working because at the beginning it started as a guided wizard and finished as the sum of several views grouped under their own perspective.

The initial development of E-EDD began with the exchange of information through a json file [5], that stored all static information used between the Java-Erlang interaction; then it was observed that a dynamic part was needed, so we proceeded to work on a communication under TCP/UDP. Finally, it was needed to use JInterface [6], the Java API for the communication with Erlang. In this way it was easy to overcome every obstacle, as it was the clearest way to establish a clear channel of communication between the two technologies and to pass messages.

A hierarchical model of classes was created to represent the information used during Java-Erlang communication so, browsing through this structure, we can access an specific data during debugging.

The visual aspect offered is perfectly integrated to the Eclipse development environment, because we have used the same technologies. This is the case of SWT [7] and JFace [8], two tools that provide a set of components for building GUIs under Java.

Then, as EDD works with dot [A1] for rendering graphs in images files, and as there exists a third party plugins that implements it in Eclipse (Eclipse GraphViz [9]), we decided to follow the same line to create the execution trees used during the debugging phase. The limitation of their use arose because, even if the dot files were re-edited and could re-generate the images, it was still a static content, that is, the user could not select nodes in them; for it we need a graphical editor. As a result, a graphical editor was created using Eclipse technology; thus it was possible to obtain dynamic graphics that allow the manipulation of the nodes at all times.

We have left open another line of research that is the use and creation of sequence diagrams dynamically. For this, we studied two technologies, mainly UMLGraph [10] and Papyrus [11]; their use is motivated because EDD has a concurrent debugging module whose implementation has not been possible to integrate in this project but whose functionality is also provided.

The rest of the memory is organized as follows: Section 3 presents the necessary preliminaries to understand and contextualize the project, Section 4 introduces the EDD project and the way it operates. Section 5 details the proposal made at the operational level while in Section 6 we detail the architectural level. Section 7 describes the communication made between

Erlang and Java and, finally, Section 8 defines the level modelling scheme used for hierarchical and graphical representation of information in Eclipse.

3. Preliminares

Antes de proceder a detallar qué es el proyecto en sí mismo y en qué consiste, es necesario aclarar algunos conceptos que nos ayudarán a entender mejor lo que se pretende mostrar con el presente trabajo.

Así pues, empezaremos presentando algunas definiciones para luego proceder a explicar las herramientas y tecnologías usadas en la implementación del depurador declarativo bajo Eclipse.

3.1. Modelo imperativo vs Modelo Funcional [A2]

Existen muchos paradigmas de programación, cada uno de ellos con características propias, cuyas soluciones a los problemas clásicos del desarrollo de software son abordadas desde diferentes perspectivas y filosofías.

Entre los paradigmas más populares tenemos el imperativo y el funcional, que procederemos a describir brevemente**.** Ambos paradigmas son muy distintos entre sí, por lo que entender bien sus aportaciones y diferencias es fundamental para poder dimensionar el propósito que persigue el presente trabajo.

Las primeras computadoras digitales se construyeron siguiendo un esquema de arquitectura denominado de Von Neumann, que es básicamente una implementación de la máquina de Turing, un dispositivo que manipula símbolos (hay un número finito de símbolos y uno especial en blanco) sobre una tira de cinta infinita dividida en celdas que contienen un símbolo. Existe una tabla de reglas finita con instrucciones las cuales dada una entrada de la cinta modifica o no el estado de la máquina, el estado se guarda en un registro para el próximo computo.

El paradigma imperativo obedece al modelo presentado por Turing, que describe como debe resolverse el problema usando un algoritmo con una secuencia de instrucciones. En dicho modelo, tal y como muestra la figura 3.1, se describe la programación en términos del estado del programa y las sentencias que lo hacen cambiar.

La implementación hardware de la mayoría de los ordenadores es imperativa, está diseñado para ejecutar código de máquina, que es nativo al computador, escrito en una forma imperativa. Esto se debe a que implementa el paradigma de las Máquinas de Turing.

Desde esta perspectiva de bajo nivel, el estilo del programa está definido por los contenidos de la memoria, y las sentencias son instrucciones en el lenguaje de máquina nativo (por ejemplo el lenguaje ensamblador).

Los lenguajes imperativos de alto nivel usan variables y sentencias más complejas, pero aún siguen el mismo paradigma

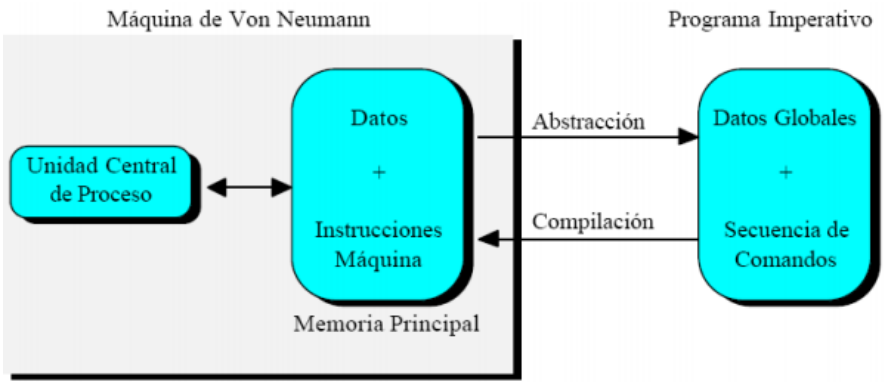

Figura 3.1 Modelo Imperativo

La programación funcional, como se observa en la figura 3.2, se aparta de esta concepción de máquina, y trata de ajustarse más a la forma de resolver el problema que a las construcciones del lenguaje necesarias para cumplir con la ejecución en esta máquina. Por ejemplo, un condicionamiento de la máquina de Von-Neumann es la memoria, por lo cual los programas procedimentales poseen variables. Sin embargo en la programación funcional pura, las variables no son necesarias, ya que no se considera a la memoria necesaria, pudiéndose entender un programa como una evaluación continua de funciones sobre funciones. Es decir, la programación funcional posee un estilo de computación que sigue la evaluación de funciones matemáticas y evita los estados intermedios y la modificación de datos durante la misma.

El paradigma declarativo obedece al modelo del cálculo lambda [L2], inventado por Alonzo Church en la década de 1930, donde una función toma como parámetros otras funciones y retorna funciones como resultado. Se dice que la función es el objeto de primera clase, en este sistema a la función se le denota con letra griega λ (lambda).

Muchos lenguajes de programación funcionales pueden ser vistos como elaboraciones del Lambda Cálculo, un sistema formal diseñado para investigar la definición de función, la aplicación de una función y la recursividad. Provee los mecanismos básicos para las abstracciones de procedimiento y abstracciones de aplicaciones (sub-programas).

La principal característica de lambda cálculo es su simplicidad ya que permite efectuar solo dos operaciones:

- **Abstracción funcional**: Definir funciones de un solo argumento y con un cuerpo específico, denotado por la siguiente terminología "λ" $\boldsymbol{x.B}$.
- **Aplicación de función**: Aplicar alguna de las funciones definidas sobre un argumento real (A). Es decir, ("λ" $\boldsymbol{x.B)A}$.

Ejemplos:

("λ" $\boldsymbol{x}$. $\boldsymbol{x}$+**2**)**3** →**5**

("λ" $\boldsymbol{f.f}$ **3**)("λ" $\boldsymbol{x.x}$+**2**) →"(λ" $\boldsymbol{x.x}$+**2**)**3**→**3**+**2** →**5**

En programación funcional no hay variables (o las variables son inmutables) dado que a las funciones solo les interesan hacer procesamiento sobre datos. En principio, no hay interés en guardar el estado de nada, cada función opera sobre sus datos y no depende de otra función, por lo cual no hay una secuencia de ejecución.

De esta forma, el modelo se basa en la especificación de un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen un problema para luego detallar su solución.

Por tanto, el resultado de evaluar una expresión compuesta depende únicamente del resultado de evaluar las sub-expresiones que la componen, es decir, no contempla la historia del programa en ejecución ni el orden de sus evaluaciones.

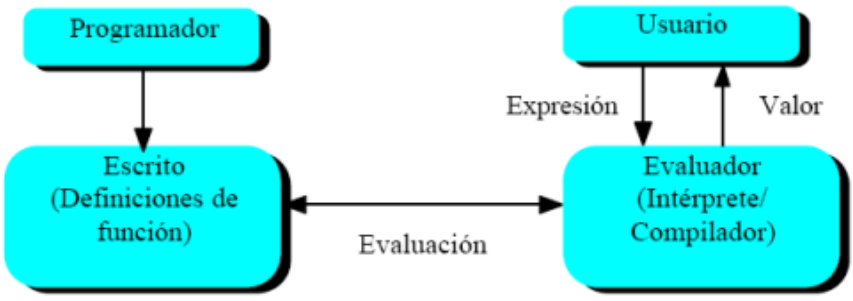

Figura 3.2 Modelo Funcional

La tabla 3.1 describe algunas de las diferencias generales entre los dos enfoques.

Característica	Enfoque imperativo	Enfoque funcional
Enfoque del programador	Cómo realizar tareas (algoritmos) y cómo realizar el seguimiento de cambios de estado.	Información deseada y transformaciones necesarias
Cambios de estado	Importante	Inexistente
Orden de ejecución	Importante	Baja importancia
Control del flujo primario	Bucles, elementos condicionales y llamadas a funciones (métodos).	Llamadas a funciones, incluyendo la recursividad.
Unidad de manipulación primaria	Instancias de estructuras o clases	Funciones como recopilaciones de datos y objetos de primera clase

Tabla 3.1 Comparación entre el modelo imperativo y funcional.

Una vez repasados ambos modelos, dejamos claro que el depurador EDD usa este último modelo no sólo en el ámbito de su desarrollo porque también bajo este mismo paradigma implementa la lógica de depuración, ya que se abstrae del orden de ejecución para centrarse en los resultados.

3.2. Erlang

Erlang [L3] es un lenguaje de programación funcional con evaluación estricta, asignación única y tipado dinámico.

- ✓ La evaluación estricta, también conocida como impaciente o *eager*, sucede cuando una función es evaluada en el momento de unirse a una operación; todo lo contrario a la no estricta, que es la evaluación perezosa o *lazy*, cuyo cálculo se retrasa hasta que sea requerido.
- ✓ La asignación única, implica que no existe la asignación destructiva ya que el propósito es hacer cumplir la transparencia referencial, es decir, se dice que una función tiene transparencia referencial si, para un valor de entrada, produce siempre la misma salida. En programación funcional esto es siempre así por definición.

- ✓ El tipado dinámico, cuando la comprobación de tipificación se realiza durante su ejecución en vez de durante la compilación. Implica que permite pasar funciones como parámetros.

Erlang está orientado a la concurrencia, su mayor fortaleza, y la distribución. El lenguaje posee un conjunto de primitivas para la creación y comunicación entre procesos que simplifican enormemente la programación; además, presume de tener la capacidad de crear una enorme cantidad de procesos sin que se llegue a degradar el rendimiento (se ha llegado a probar hasta 20 millones de procesos [12]).

Fue creado en el laboratorio de ciencias de la computación de la compañía Ericsson en Suecia a finales de 1980, con la finalidad de facilitar la creación de aplicaciones distribuidas, tolerantes a fallos, soft-real-time y de tiempo ininterrumpido; más tarde fue cedido como software de código abierto en 1998.

La distribución del lenguaje junto con otras herramientas y la base de datos en tiempo real (Mnesia [13]) se denominan Open Telecom Platform OTP.

Erlang se caracteriza por:

- **Concurrencia**: utiliza procesos ligeros cuyos requisitos de memoria pueden variar de forma dinámica. La comunicación se produce por medio de paso de mensajes asíncrono.
- **Distribución**: diseñado para entornos distribuidos, de forma que cada máquina virtual recibe el nombre de nodo, permitiendo así la creación de redes de nodos interconectados comunicándose entre sí e incluso en diversos sistemas operativos.
- **Robustez**: posee varias primitivas para la detección de errores con la idea de estructurar sistemas tolerantes a fallos. El uso de monitores permite registrar la actividad de otros procesos, de forma que se puede diseñar estructuras que permitan conmutar actividades en caso de fallo permitiendo así recuperar la funcionalidad sin mayor problema.
- **Soft-real-time**: son sistemas en tiempo real que requieren respuestas en el orden de milisegundos.
- **Actualización de código en caliente**: muy útil cuando un sistema en producción no puede ser detenido para mantenimiento. En la fase de transición coexisten

tanto el código antiguo como el nuevo, lo que permite la inserción de parches con suma facilidad sin verse alterado el comportamiento.

- **Código de carga incremental**: permite controlar con detalle cómo debe cargarse el código, por ejemplo durante el arranque (utilizado normalmente en sistemas empotrados) o bajo demanda de uso (común en entornos de desarrollo).
- **Interfaces externas**: utilizará el mismo mecanismo de paso de mensajes, como si de procesos Erlang se tratase, para comunicarse con herramientas externas lo que permite que la comunicación sea transparente. Este mecanismo se utiliza para la comunicación con el sistema operativo y la interacción con otros programas

3.3. Eclipse

Eclipse [L4] es un entorno de desarrollo integrado abierto y extensible que permite su personalización bajo necesidades específicas mediante la creación de plugins (unidad más pequeña e instalable de componentes de software en Eclipse, véase los tipos en la sección 3.3.2). Para ello se hace uso de los denominados "puntos de extensión", que no son otra cosa que el mecanismo mediante el cual un plugin puede añadir su contribución a la funcionalidad de otros plugins. Cada plugin puede definir sus propios puntos de extensión para que herramientas de terceros puedan contribuir y extender la implementación que los define.

3.3.1. Arquitectura Eclipse.

La figura 3.4 muestra una visión general de la arquitectura Eclipse [A3].

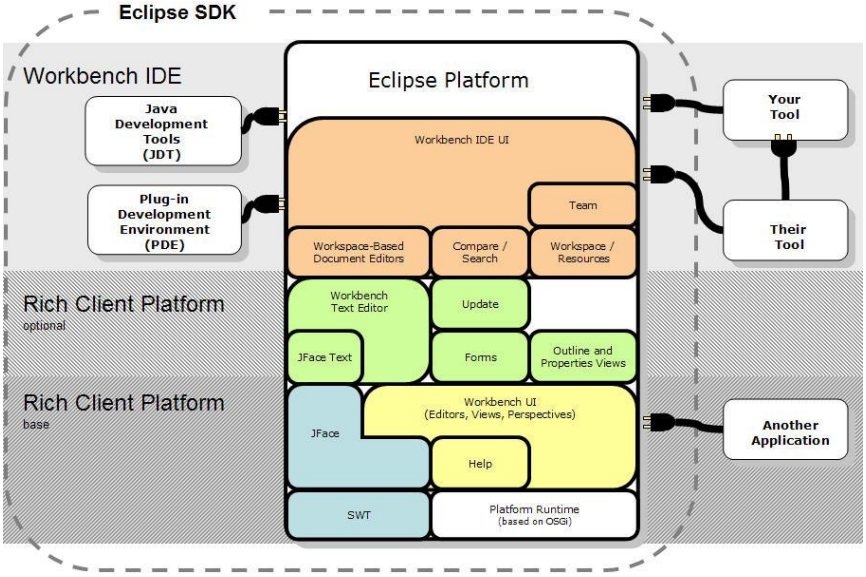

Figura 3.3 Arquitectura Eclipse

Tanto la capa base de la arquitectura como la capa opcional intermedia describen lo que podría definirse como “Plataforma de Cliente Rico”, un marco de desarrollo que permite la creación de aplicaciones de escritorio (además de manejar su ciclo de vida por completo).

Todos los componentes necesarios para el desarrollo están incluidos en la capa base, que ofrece mecanismos para ampliarse con el uso de otras aplicaciones; la aportación de la capa opcional es ofrecer un subconjunto de componentes avanzados que dan valor añadido al desarrollo, lo que permite mejorar aún más la estética de la aplicación.

Finalmente, la capa superior es la que provee las herramientas y mecanismos de extensión bajo Eclipse, define la estructura del espacio de trabajo llamado *Workspace* y gestiona los recursos contenidos en el mismo. Como se puede observar el IDE de Java llamado JDT [14] se entrega como parte de Eclipse.

A continuación se detallan algunos de los conceptos más relevantes de la arquitectura (ver figura 3.4 para ubicar la capa en que se encuentran):

- **Platform Runtime**: su principal función es descubrir qué plugins están disponibles. Estos no son cargados hasta que son requeridos, con la finalidad de minimizar el uso de recursos.
- **Workspace**: es el responsable de administrar los recursos de los usuarios, los cuales están organizados en uno o más proyectos. Además, mantiene a bajo nivel el historial de cambios de cada recurso.
- **Workbench**: es la interfaz gráfica de Eclipse. Está organizada en perspectivas que contienen vistas y editores, incluyendo los menús, barras de herramientas, etc. El Look & Feel está basando en SWT y JFace; la razón es que han sido portados a las plataformas más populares (Windows, Linux/Motif, Linux/GTK2, Solaris, QNX, AIX, HP-UX y Mac OS X).
- **Team Support**: facilita el uso de control de versiones y el trabajo en equipo, ya que Eclipse incluye un cliente CVS (Concurrent Versions System).
- **Help**: al igual que la plataforma de Eclipse, la ayuda es un sistema de documentación extensible. Se puede agregar documentación en formato HTML usando XML y definir una estructura de navegación, además de insertar temas en una estructura preexistente.

- **JDT** (Java Development Toolkit): proporciona las herramientas que implementan un IDE Java para apoyar al desarrollo de cualquier aplicación bajo este lenguaje; estas herramientas están incluidas en la característica que incluye la perspectiva Java que contiene todo lo necesario para el desarrollo en la misma.
- **PDE** [15] (Plugin Development Environment): provee las herramientas para crear, desarrollar, probar, depurar, construir y desplegar plugins de Eclipse. Además proporciona las herramientas OSGi [16], ideales para la programación de componentes, es decir, permiten la programación modular.
- **IDE** (Integrated Development Environment) [17]: es el uso más común dado a Eclipse, como un Entorno de Desarrollo Integrado (IDE) para el lenguaje de programación Java principalmente, aunque también es posible usarlo para otros lenguajes como C/C++, PHP, o como un IDE Web, mientras que para el lenguaje Erlang se ha desarrollado el proyecto Erlide.

Si se desea aumentar los conocimientos sobre la tecnología Eclipse se pueden consultar la referencia [18], en la cual se encuentra más información al respecto.

3.3.2. Clasificación de plugins

Técnicamente, cualquier herramienta que se ejecute dentro del entorno Eclipse se debe implementar como un plugin o conector. Las referencias [A4] y [A5] contienen más información a modo de introducción al método de desarrollo de plugins llamado PDE.

Dentro del entorno PDE encontraremos una variedad de tipos de proyectos que permitirán organizar el desarrollo según el propósito del contenido a desarrollar.

- **Plugin**: es un proyecto de plugin normal, el tipo más común, que contiene el archivo básico de configuración (plugin.xml) y la estructura de carpetas generadas donde se alojarán las clases de negocio de la contribución del proyecto a Eclipse.
- **Fragmento**: permite agregar contenido a un plugin ya existente (para idiomas, targets, etc).
- **Característica**: unidad de instalación que incluye uno o más plugins.
- **Sitio de actualizaciones**: sitio web para instalaciones automáticas de características.

- **Parche de característica**: como su nombre indica, es usado cuando no se desea liberar una nueva versión del producto integral pero es necesario realizar algún parche en la misma.
- **Plugin de archivadores JAR existentes**: crea un nuevo proyecto de plugin de OSGi a partir de un archivo JAR existente.
- **Plugin de producto**: permite crear un ejecutable de nuestra aplicación basada en Eclipse sin que se integre en el mismo. Para ello se indican los plugins o características que serán exportadas y utilizadas en dicho ejecutable.

3.3.3. Erlide

El proyecto Erlide [19] es una herramienta de desarrollo Erlang basada en Eclipse, que destaca por ofrecer las siguientes características:

- Un editor, para la edición de módulos Erlang, con resaltado de sintaxis.
- Un constructor de proyectos (*builder*) para la compilación y creación de binarios.
- Un depurador, bajo un entorno de depuración tradicional basado en *breakpoints*.
- Un nodo Erlang subyacente.
- Vista para evaluación de expresiones en vivo.
- Autocompletado de código.
- Vista de documentación para bibliotecas estándar.
- Está en constante actualización.

Erlide forma una parte importante en el actual desarrollo de E-EDD ya que se han reaprovechado parte de algunas de sus funcionalidades (principalmente la creación de proyectos y el editor) con el fin de poder convergir y buscar una integración entre los proyectos E-EDD y Erlide.

3.4. Introducción a la depuración

Según la definición que ofrece el Diccionario de la Real Academia Española **depurar** [20] significa limpiar o purificar, sin embargo si se utiliza como término aplicado al ámbito de la programación su significado implica los actos de revisar y analizar la sintaxis de un programa con la finalidad de determinar si es correcto o genera errores, tanto durante la redacción como en el momento de su ejecución.

Hoy en día, en lo que a la tecnología se refiere, predominan los vocablos ingleses que son quienes dan nombre a muchas de las acciones y a la acción de depurar se le ha dado el nombre de *debugging,* esto es, eliminar *bugs* (bichos) o errores. En otras palabras se trata de la acción que realizan los programadores durante la busca y captura de errores en sus programas.

Mediante el proceso de depuración lo que se suele hacer es una ejecución paso a paso en la cual se comprueba que todo vaya según lo esperado revisando las sentencias una a una, por ejemplo, mediante el uso de puntos de interrupción que detengan la ejecución del programa para su inspección de forma que obtengamos información sobre el estado de la aplicación en ese momento, o mediante la técnica de ir dejando trazas encargadas de ofrecer pistas para saber lo que está sucediendo. Si, por alguna razón, existiera una sentencia errónea entonces se tendría que generar un informe de error para esa sentencia en particular, avisando de que es incorrecta o incomprensible para el ordenador (para más información véase la sección 3.5.2).

La depuración o *debugging* es una herramienta imprescindible para cualquier desarrollador, independientemente del lenguaje de programación utilizado. La no utilización de dicha depuración dificultaría mucho la comprensión de lo que está pasando durante la ejecución de los programas, y cualquier fallo o anomalía sería muy difícil de detectar. Todo error en el código fuente implica un funcionamiento incorrecto e inesperado, lo que obliga a su localización y corrección de forma imperativa, y esto se lleva a cabo con las distintas técnicas de depuración existentes, así como también con el uso de procedimientos automatizados de errores, conocidos como depuradores.

Depurar íntegramente un programa es una tarea tan ardua como utópica porque lo cierto es que, a día de hoy, no existe herramienta capaz de asegurar que todos los errores han sido detectados y corregidos.

3.4.1. Errores de programación

Ahora bien, es necesario entender qué es un error de programación para poder detectarlo. Conocer cómo depurar una aplicación y encontrar estos errores es una parte importantísima de la programación; es por ello que conviene conocer los tipos de errores que se deben buscar y corregir.

Todo error de programación se podría clasificar dentro de las siguientes categorías [21]:

- **Errores de compilación**: impiden la ejecución del programa; normalmente están relacionados con la sintaxis del código fuente.
- **Errores en tiempo de ejecución**: aparecen cuando se ejecuta el programa y se realiza alguna operación imposible de ejecutar como, por ejemplo, una división por cero.
- **Errores lógicos**: son debidos a una lógica mal empleada que impide que se lleve a cabo lo previsto, obteniendo resultados no esperados. Esta clase de errores son los más difíciles de detectar y corregir.

3.4.2. Tipos de depuración

Existen numerosas técnicas [22] y tipos de depuración [23], los cuales a su vez operan en diversas fases y etapas durante el proceso de desarrollo, además de trabajar también de muy diversas maneras. Es por ello que el presente documento no pretende ser un análisis exhaustivo de las mismas, sino que solo pretende dar a conocer algunas de ellas para poder entender el propósito del trabajo y poder contextualizarlo.

3.4.2.1. Depuradores imperativos

- **Depuración a base de trazas (*Tracing*)** [24]

Consiste en ir dejando trazas que permitan a los desarrolladores poder identificar problemas según la forma en que se esté ejecutando el programa, contrastando los resultados esperados con dichas trazas.

- **Depuración paso a paso** [25]

Quizá es la más extendida y consiste en una depuración por instrucciones, es decir, ejecuta la instrucción actual y, a continuación, se detiene en la siguiente instrucción para su análisis (y así sucesivamente). Se suele hacer uso de puntos de interrupción con la finalidad de facilitar la depuración además de permitir la inspección de variables.

- **Depuración remota** [26]

Se trata de un tipo de depuración en el cual el programa está siendo ejecutado en un equipo distinto respecto a donde se realiza la depuración. Obviamente, es indispensable la comunicación entre ambos mediante una red, útil cuando se trabaja para entornos distribuidos.

- **Depuración *post-mortem*** [A6]

Su función es identificar los errores una vez el programa haya fallado. El proceso se basa en la monitorización del estado de la aplicación para así generar un informe de lo que ha ocurrido.

- **Depuración delta** [27]

Automatiza el proceso de depuración y analiza sistemáticamente la localización de errores aislándolos, permitiendo comprobar cómo sería el funcionamiento real de la aplicación cuando han sido retirados del entorno.

- **Saff Squeeze** [A7]

Es un proceso creado por David Saff, en el cual un problema se aísla mediante la ejecución de dos procesos. El primero es un proceso de alto nivel que se utiliza para identificar los grandes problemas en la aplicación. A continuación, las pruebas de unidad específicas se ejecutan para aislar el problema exacto o el error. Esto acelera el proceso de depuración al tiempo que identifica los fallos.

3.4.2.2. Tipos y sistemas de depuración funcionales

- **Depuración declarativa** [T1] y [T2]

La depuración declarativa**,** al igual que la programación declarativa, abstrae el orden de ejecución para centrarse en los resultados; dicha técnica es posible encontrarla también en lenguajes imperativos como Java.

Como ya se ha explicado en el apartado 3.1, en los lenguajes declarativos las sentencias que se utilizan describen el problema que se quiere solucionar, y para ello se utilizan mecanismos internos de inferencia a partir de la descripción realizada.

Se suele distinguir entre dos tipos de depuración declarativa: uno a base de respuestas incorrectas (que se aplica cuando se obtiene un resultado incorrecto desde un valor inicial) y el otro en base a la falta de respuestas o respuestas perdidas (que se aplica cuando un resultado está incompleto. Se trata de una técnica menos abordada dada su complejidad).

La depuración declarativa es un proceso que se realiza en dos fases:

1. Lo primero es calcular un árbol de ejecución, donde cada nodo representa un paso en el cálculo y cada resultado es una posible bifurcación en la jerarquía.

2. En la segunda fase, se realiza un recorrido siguiendo una estrategia de navegación para ello, donde es necesario consultar a un oráculo externo que nos confirme los resultados de los nodos. Dicho oráculo es el usuario, que es quien conoce los resultados esperados.

Así pues, la idea es ir navegando en el árbol hasta dar con la causa del problema, un nodo cuya descendencia de nodos sea correcta salvo él mismo, dejando de manifiesto que él es el problema.

- **Depuración declarativa con zoom** [A8]

Definida como extensión de la anterior, cuyo propósito está justificado puesto que, aunque se conozca una función errónea, la ubicación del fallo dentro de la misma puede resultar aún complicada de detectar.

Existen muchos caminos por los cuales se puede navegar en el interior del cuerpo de una función debido al uso de expresiones anidadas o a los posibles valores que puedan tomar las variables que a su vez dependen de variable previas, lo que complica la situación. Mediante el uso de esta técnica se hace una depuración inspeccionando la función errónea con la finalidad de dar un resultado más preciso.

- **ART (Advanced Redex Trails, que después se convirtió en Hat)** [T3]

Depuradores basados en la observación de un grafo de cómputo. Tanto la dificultad para realizar dicha depuración como la de interpretar los resultados hicieron que esta primera aproximación se desechara.

- **Freja y Buddha** [T3][T4]

Uno de los primeros depuradores que aparecieron en programación funcional perezosa fueron los depuradores declarativos. En estos no se veían los pasos de evaluación que se realizaban en el cómputo, sino las reducciones semánticas que se habían producido; es decir, los resultados de la aplicación de una función a sus argumentos.

Ambos son depuradores declarativos para el lenguaje funcional Haskell. Están basados en preguntas del tipo sí/no, donde las respuestas del usuario hacen que se acote el fragmento de código a analizar, de tal forma que mediante estas respuestas el programa devuelve la función que está incorrectamente implementada.

- **Hood** [T3][T4]

Hood surgió con la idea de poder observar el cómputo perezoso sin modificar el comportamiento perezoso de los programas. Se creó como una librería independiente de Haskell

y lo único necesario para utilizarlo era cargar la librería y anotar las expresiones a observar. Se puede considerar como una instrucción de escritura perezosa, que observa el cómputo de las estructuras marcadas

Hood genera ficheros pequeños concentrados sólo en la parte de traza que desea ver el usuario.

- **Hat** [T3][T4]

Hat no sólo modificó su nombre sino que fue uno de los depuradores que más ha cambiado y se ha mantenido a lo largo de los años. Por un lado, incorporó las ideas de otros depuradores y aportó nuevas herramientas para la depuración. La idea principal de esta herramienta consistía en analizar el grafo generado por el cómputo tras la evaluación del programa.

Actualmente Hat se puede considerar como un conjunto de herramientas. Para ejecutar este depurador, al igual que en Freja y Buddha, es necesario compilar el programa de forma especial para que cuando se ejecute genere una traza en un fichero; su principal desventaja es que el archivo que genera es demasiado grande. Por tanto, dicha traza puede ser analizada con las herramientas para ver el motivo de error o la situación en la que se encontraba el cómputo.

4. Erlang Declarative Debugger

EDD [A9] es un depurador declarativo para programas Erlang, escrito en el mismo lenguaje (Erlang OTP deberá estar instalado en el equipo), que basa su funcionamiento en una serie de algoritmos en los cuales, a base de un proceso de preguntas que se realizan al usuario, se comprueban los resultados esperados de algunas de las evaluaciones (funciones, llamadas a funciones, evaluaciones en grupos de expresiones, etc.) con la finalidad de señalar el fragmento de código que está causando el error o problema.

Construye un árbol de ejecución que será recorrido mediante la interacción del usuario y las preguntas realizadas por el depurador. El propósito será buscar un nodo "malo" con hijos incorrectos, llamado *buggy node* con la finalidad de señalar el fragmento de código que está causando el error o problema en el código que ha sido depurado.

Para más información se puede acceder al repositorio del código fuente [3] en donde se podrá además consultar información extra sobre el proyecto EDD. Allí se explica también en qué consiste la herramienta y los algoritmos usados

Esta herramienta centra su atención en la depuración declarativa, la cual basa sus fundamentos en demostraciones con una cierta lógica asociada; es por ello que EDD parte de una llamada a una función que devuelve un resultado incorrecto con la finalidad de conducirnos al sitio donde se está produciendo el fallo.

4.1. Ejemplo de utilización mediante un caso práctico

Para entender correctamente el funcionamiento del depurador vamos a proceder a explicarlo mediante un ejemplo práctico, lo que permitirá tener una visión objetiva del mismo en base a la experiencia de usuario.

El ejemplo en cuestión es uno de los muchos que se adjuntan con la propia herramienta en el directorio "examples"; allí tenemos una serie de casos que han sido probados usando el depurador.

Para demostrar el funcionamiento vamos a partir con una implementación incorrecta de mergesort (figura 4.1) [28], un caso muy conocido en el ámbito de la programación cuya función es la de ordenar elementos bajo el algoritmo del mismo nombre, también llamado "método de

ordenación por mezcla". Se trata de un algoritmo de la familia "divide y vencerás", que se caracteriza por seguir tres fases:

- **Dividir**: como su nombre indica lo que se hace en esta primera fase es dividir el problema en partes más pequeñas.
- **Conquistar**: se realiza por medio de la resolución recursiva de los problemas pequeños.
- **Combinar**: valiéndonos de los resultados de los problemas pequeños estos serán combinados con la finalidad de resolver el problema en su totalidad.

```
-module(merge).
-export([mergesort/2, comp/2]).

% Calls to the declarative debugger:
% > edd:dd("merge:mergesort([b,a], fun merge:comp/2)").
% > edd:dd("merge:mergesort([o,h,i,o], fun merge:comp/2)").

mergesort([], _Comp) -> [];
mergesort([X], _Comp) -> [X];
mergesort(L, Comp) ->
    Half = length(L) div 2,
    L1 = take(Half, L),
    L2 = last(length(L) - Half, L),
    LOrd1 = mergesort(L1, Comp),
    LOrd2 = mergesort(L2, Comp),
    merge(LOrd1, LOrd2, Comp).

merge([], [], _Comp) ->
    [];
merge([], S2, _Comp) ->
    S2;
merge(S1, [], _Comp) ->
    S1;
merge([H1 | T1], [H2 | T2], Comp)  ->
        case Comp(H1,H2) of
%            false -> [H2 | merge([H1 | T1], T2, Comp)];
            false -> [H1 | merge([H2 | T1], T2, Comp)];
            true ->  [H1 | merge(T1, [H2 | T2], Comp)]
        end.


comp(X,Y) when is_atom(X) and is_atom(Y) -> X < Y.


take(0,_) -> [];
take(1,[H|_])->[H];
take(_,[])->[];
% take(N,[H|T])->[H | take(N-1, T)]. % Correct
take(N,[_|T])->[N | take(N-1, T)].   % Incorrect

last(N, List) ->
    lists:reverse(take(N, lists:reverse(List))).
```

Figura 4.1 implementación incorrecta de mergesort (merge.erl)

Antes de proceder con el ejemplo, veremos cómo iniciar el depurador EDD que nos ayudará a resolver el error de programación que causa la mala implementación del algoritmo anteriormente descrito.

Lo primero será compilar y cargar el modulo necesario para hacer funcionar EDD, el cual es "`edd_comp.erl`", encargado de cargar todos los módulos que tienen la lógica del depurador.

Si trabajamos desde un entorno Linux se puede ejecutar el comando `make` junto al archivo Makefile [29] que ya viene proporcionado para facilitar la compilación y carga de los módulos (hay que tener en cuenta que el presente documento describe su uso bajo el entorno Windows, por lo tanto las compilaciones han debido hacerse de forma manual).

Para proceder ejecutamos una consola Erlang con el comando `erl` (es importante posicionarse en la ubicación raíz del proyecto donde esta EDD); una vez dentro compilamos y verificamos que funciona utilizando el comando "`edd_comp:compile().`", tal y como muestra la figura 4.2.

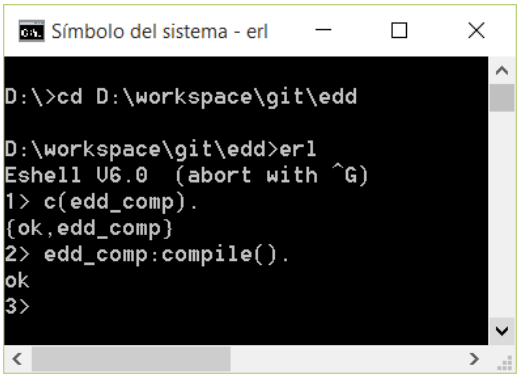

Figura 4.2 Compilación y validación del módulo edd_comp.erl

Una vez verificado procedemos saliendo de la sesión Erlang creada previamente para volver a iniciar una nueva donde esta vez se carga el módulo recién compilado encargado de cargar los módulos necesarios para trabajar con EDD, para ello ejecutamos el comando "`erl -run edd_comp load`" como se muestra en la figura 4.3.

Una vez iniciada la sesión ya podremos empezar a trabajar con el depurador declarativo según el ejemplo propuesto:

- Nos posicionamos en la ubicación del archivo fuente Erlang a depurar y lo compilamos (1).
- Ejecutamos y verificamos los resultados que devuelve la función que consideramos errónea (2); analizamos el resultado y vemos que los elementos no están ordenados correctamente.
- Iniciamos la depuración de dicha función que hemos detectado como errónea (3).
- El depurador computará un árbol de ejecución para ir lanzando interrogantes (4, 5 y 6) a las que deberá responder el usuario; para ello se ofrece la lista de respuestas posibles (sí, no, confiable, no lo sabe, inadmisible, etc.).

- La dinámica pregunta - respuesta no terminará hasta que se haya detectado la causa del problema (*buggy node*), en donde se dice al usuario que revise la cláusula que contiene el error.

La figura 4.3 resume lo anterior llevado a consola bajo el ejemplo propuesto.

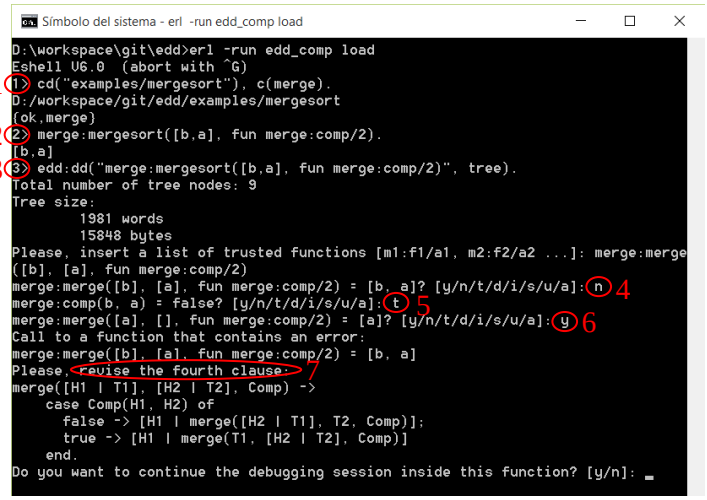

Figura 4.3 Ejemplo de depuración declarativa estándar bajo EDD

Ahora sabemos dónde se encuentra el fallo: la cuarta cláusula del código (7, obtenida a partir de la 1ª fase, llamada depuración declarativa estándar). A partir de este momento se ofrece también la posibilidad de continuar depurando dicha cláusula errónea entrando a un nivel de detalle más específico (2ª fase, denominada depuración con zoom).

La depuración con zoom tiene sentido cuando el error ha sido encontrado en una cláusula muy grande o compleja al estar compuesta por más cláusulas, lo que dificulta detectar el fallo aun cuando de antemano se haya reducido a la zona donde ocurre. Si el usuario considera que la información ofrecida no es suficiente entonces deberá continuar con la depuración con zoom:

- Para ello debe responder que sí cuando se le pregunte si desea continuar.
- Una vez entrando a la depuración con zoom, se desplegará una serie de cláusulas que deberemos revisar detenidamente para poder responder a la próxima pregunta planteada.
- Al igual que el caso anterior la ronda de pregunta-respuesta no terminará hasta que se haya encontrado el problema en detalle.

- Siguiendo con nuestro ejemplo concreto, vemos que la causa del problema viene definida por la cláusula número 4 (8), listada en las opciones dadas depurador, cuya afirmación es incorrecta.
- Finalmente vemos que el problema estaba dado en la línea 27 (9, véase figura 4.1 donde se muestra el código fuente en depuración), tal y como se observa en la figura 4.4, que es la raíz del problema.

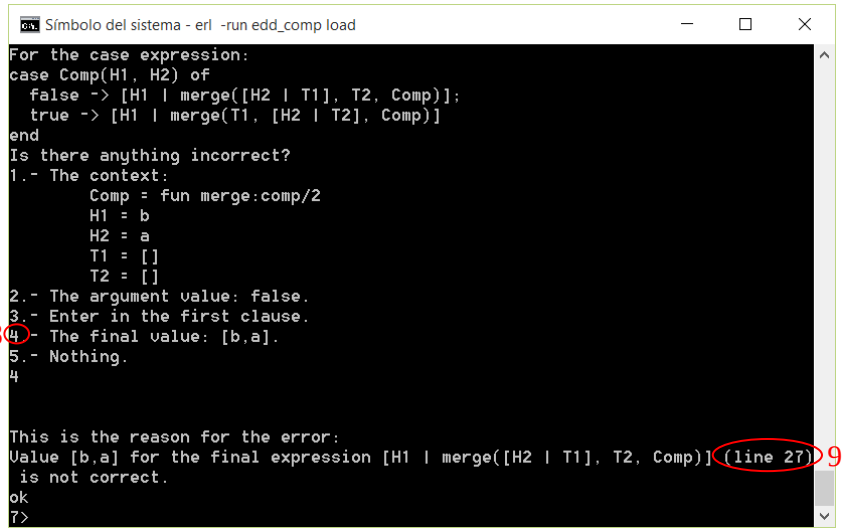

Figura 4.4 Ejemplo de depuración con zoom bajo EDD

Una vez llegado hasta aquí, podremos hacernos una idea general de lo que es el depurador, en qué consiste y cómo se utiliza.

4.2. Conclusiones sobre EDD

Una vez que el usuario está familiarizado con la herramienta de depuración declarativa, podrá obtener su primera impresión al respecto. La idea es que el usuario pueda valorar la aportación que podría darle la herramienta en su uso habitual y saber si podría obtener una utilidad práctica en sus desarrollos bajo Erlang.

Sin embargo, como se puede observar, la falta de usabilidad está de manifiesto durante la ejecución: la propuesta presenta una serie de inconvenientes; el simple hecho de manipular todo por consola ocasiona que la información dada al usuario esté recargada e, incluso, que en ocasiones sea imposible leer su contenido en su totalidad al haberse superado el buffer de líneas permitido por consola (probado en un entorno Windows, en la depuración con zoom fue imposible leer las cláusulas en su totalidad).

Entre otros inconvenientes está también el hecho de realizar comprobaciones respecto al código fuente. Continuamente se deben estar haciendo cambios de ventana para contrastar la información (dado que deberá leerse desde otra aplicación). Todo ello lleva a una pérdida de tiempo y esfuerzo durante la fase de depuración, lo que ocasiona que sea una herramienta que no consigue mantener el centro de atención de los usuarios en la misma, algo muy importante cuando se trabaja en modo consola, ya que se debería poder acceder a toda la información sin que ello le reste usabilidad, aun cuando se trabaje a bajo nivel.

Por tanto, la finalidad de enriquecer y dar usabilidad es lo que fundamenta el presente desarrollo; el propósito es dar a conocer el potencial que alberga el depurador declarativo, un proyecto que presenta ya una madurez algorítmica digna de ser mostrada y explotada. Ahora toca dotarlo de una interfaz sencilla y práctica que permita una manipulación amigable; con ello será más fácil acercarla a los desarrolladores facilitando su promoción y uso.

Teniendo en cuenta todo lo anterior no es difícil entender que la única forma de ganar terreno y aceptación con la depuración declarativa es ofreciendo herramientas que cubran unas necesidades cuya aplicación es imprescindible para poder ofrecer alternativas prácticas y eficaces.

5. Eclipse - Erlang Declarative Debugger

E-EDD [30] es la versión del depurador declarativo Erlang bajo el entorno Eclipse, cuyo objetivo es proporcionar una herramienta de depuración gráfica basada en SWT/JFace [L5].

Ha sido desarrollada usando plugins de Eclipse [L6] y hace uso de los proyectos Eclipse Modeling Framework (EMF) [L7] y Graphical Modeling Frameork (GMF) [L8] para la construcción de un editor gráfico propio con la finalidad de construir y generar los árboles de ejecución utilizados en la depuración.

Aprovecha parte de la funcionalidad ofrecida por el proyecto Erlide: la creación de proyectos y el editor de módulos Erlang.

Utiliza el API de JInterface para la comunicación Java-Erlang.

Ante la escasa difusión de herramientas auxiliares en entornos de programación funcional, no es difícil entender que el trabajo en ellas tuviese lugar bajo entornos realmente pobres, donde la falta de interfaces gráficas era habitual dado que los esfuerzos estaban más centrados en lo operativo que en lo práctico, es decir, lo normal era trabajar en modo consola (de la misma forma que en el apartado anterior), siempre a menor nivel respecto a lo que suele hacerse con las herramientas de programación imperativa. Estas últimas tienen más que ofrecer dado que su ámbito de aceptación ha sido mucho mayor y diverso durante mucho más tiempo.

Es por ello que, observando el gran trabajo que vienen haciendo desde hace tiempo los creadores de EDD, surgió la idea de intentar introducir una herramienta gráfica que simplificara la forma de operar, ofreciendo lo mismo pero con una capa de presentación que facilite el uso de la misma en un entorno más amigable.

El proyecto E-EDD es una primera aproximación a la integración de la depuración declarativa en un entorno de desarrollo integrado ampliamente utilizado en el mundo de desarrolladores como es Eclipse. La elección viene fundamentada por el hecho de que Eclipse, además de ser un IDE, es a su vez una comunidad de desarrolladores, lo que permite que sea una herramienta de fácil extensión: mediante el desarrollo de plugins para Eclipse podremos adaptarlo, configurarlo y personalizarlo a necesidades específicas.

Esta primera aproximación pretende ser un referente de aplicación para una serie de algoritmos de depuración que pueden ser ejecutados en el marco de referencia propuesto bajo EDD y ser llevados a E-EDD, donde todo pueda convergir de forma simple y sencilla.

La propuesta presenta una serie de vistas en donde se integra perfectamente el mecanismo pregunta - respuesta empleado por el motor de depuración declarativo de EDD, de tal manera que futuros algoritmos de depuración declarativa podrían ser utilizados bajo la misma forma de operación expuesta en el presente documento.

Hemos de mencionar que, durante el desarrollo de E-EDD, ha habido una serie de decisiones que han ido cambiando la forma de operar propuesta inicialmente hasta obtener lo que aquí se presenta con el objetivo de aprovechar la herramienta de depuración al máximo al fusionarla de la mejor manera posible en el entorno Eclipse.

Con esta primera versión se consigue un primer paso a los objetivos planteados inicialmente: ofrecer un entorno de depuración más sofisticado, usable y que pueda ser ampliado, permitiendo integrar con facilidad posteriores algoritmos de depuración declarativa.

5.1. Creación de proyectos Erlang

Antes de empezar a trabajar con E-EDD, es necesario haber instalado los plugins de Erlide (consultar anexos de instalación en el apéndice A), necesario para poder crear un proyecto Erlang. Para ello debemos seleccionar desde el menú de opciones: File/New/Erlang Project descrito en la figura 5.1.

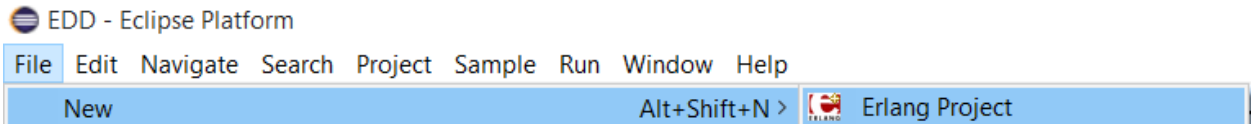

Figura 5.1 Creación de un proyecto Erlang

Aparecerá el diálogo de la figura 5.2 donde deberemos especificar el nombre del proyecto a crear.

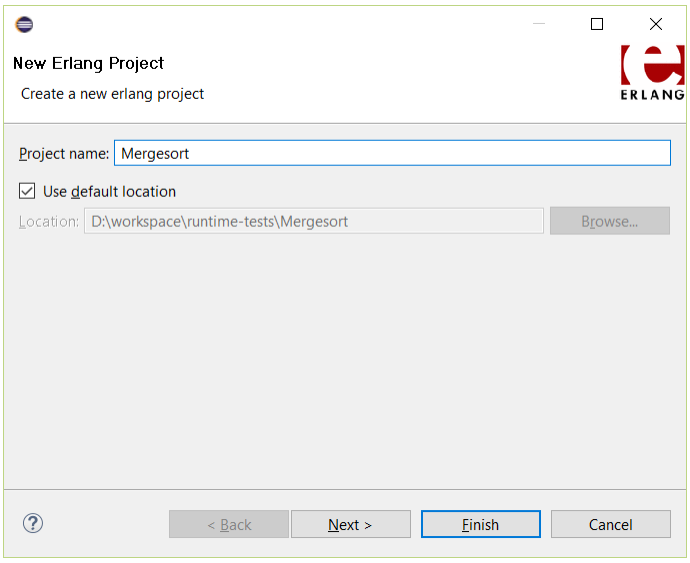

Figura 5.2 Nuevo proyecto Erlang

Avanzamos a la siguiente página, tal y como muestra la figura 5.3, para proceder a configurar la versión de compilador Erlang a utilizar en el proyecto.

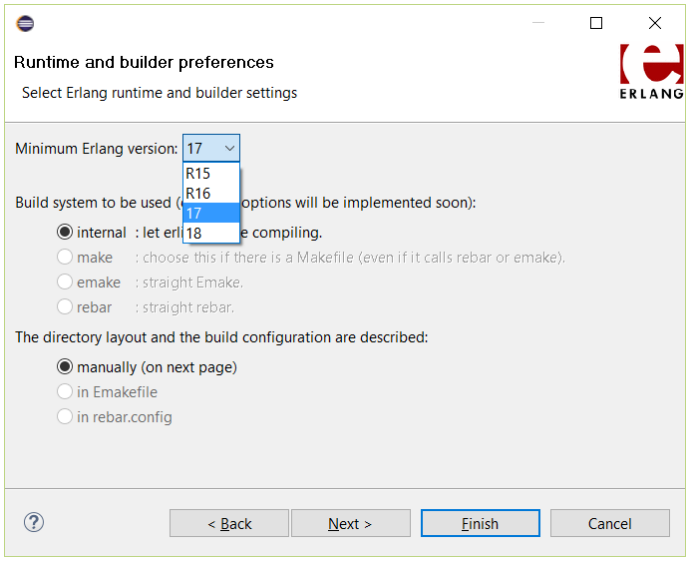

Figura 5.3 Preferencias del proyecto

En la página siguiente debemos indicar las rutas donde desea alojar los archivos fuentes y binarios. Se recomienda modificar el directorio de salida de *ebin* (valor por defecto) a *src* tal y como se ejemplifica en la figura 5.4, dado que la ubicación de los binarios puede dar lugar a errores de ejecución.

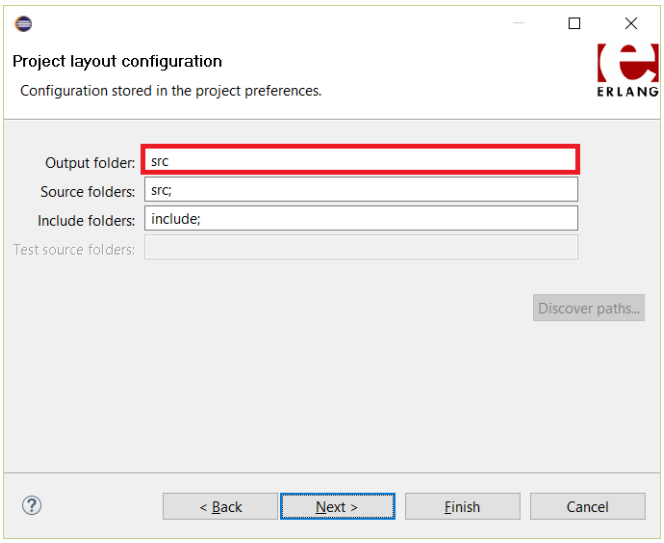

Figura 5.4 Configuración del proyecto

Es interesante mencionar que existe una última página en el asistente donde se deben indicar las referencias a otros proyectos; es un paso que no hace falta explicar dado que depende de la distribución de código que quiera dar el usuario a sus archivos.

En la ventana Erlang Navigator podremos consultar el proyecto recién creado, que tendrá un aspecto similar como al que se muestra en la figura 5.5, en cuyo ejemplo detallamos también la existencia de archivos fuentes *erl* y binarios *beam* bajo la carpeta *src*, tal y como habíamos definido durante su creación.

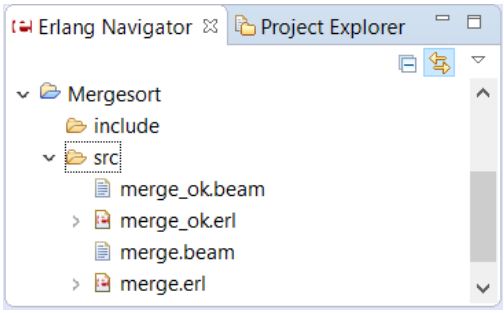

Figura 5.5 Estructura del proyecto

5.2. Introducción a la perspectiva EDD

Antes de empezar a utilizar la herramienta debemos definir lo que es una perspectiva Eclipse, que no es otra cosa que una agrupación de vistas y editores configurados de manera que den apoyo a una actividad completa e integral dentro del entorno de desarrollo bajo Eclipse.

Toda perspectiva parte de un diseño o distribución inicial, la cual siempre será posible personalizar a gusto del usuario, permitiendo que pueda añadir y quitar vistas a demanda y reconfigurar las posiciones de las mismas con la finalidad de agruparlas de la forma que le resulten más cómodas en el momento de trabajar.

La configuración por defecto de la perspectiva EDD, que es con la cual se deberá trabajar para proceder a la depuración declarativa de programas Erlang de ahora en adelante, presenta la siguiente distribución inicial tal y como se muestra en la figura 5.6.

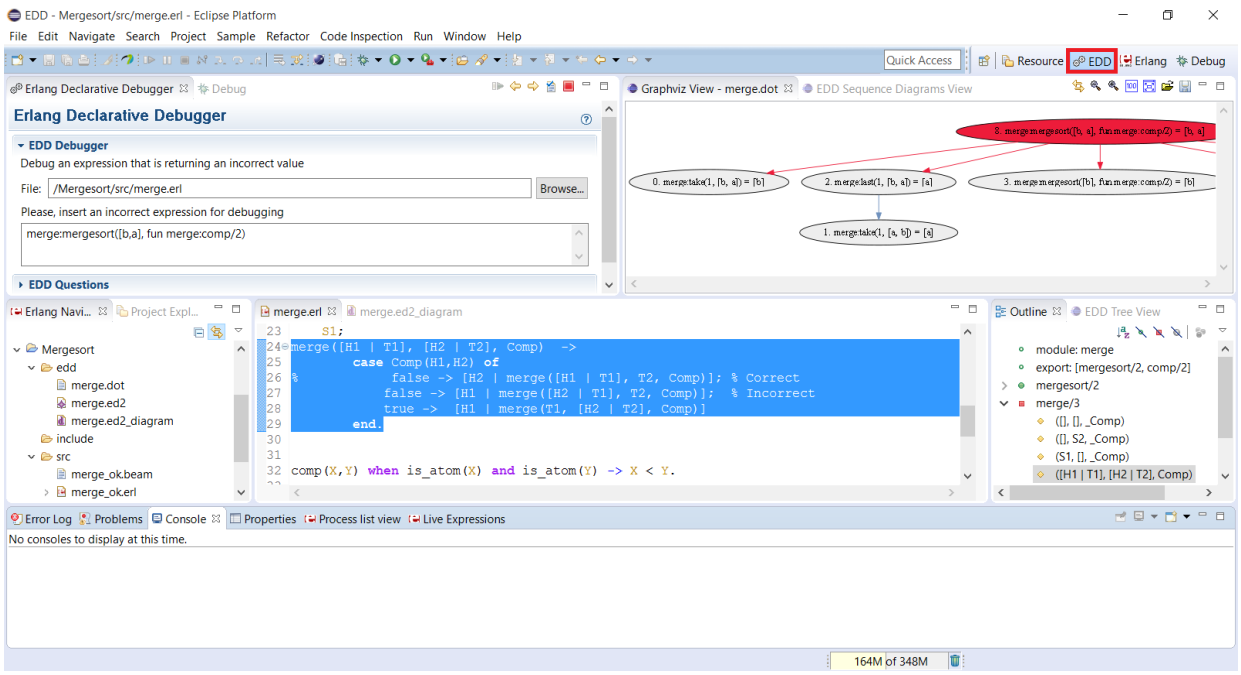

Figura 5.6 Perspectiva EDD

Como se puede observar en la figura 5.6, en la parte superior hay dos grupos de ventanas: en la izquierda la ventana de depuración EDD, y en la derecha las ventanas que nos irán mostrando los árboles de ejecución empleados durante la sesión de depuración.

En la parte central tendremos en la izquierda el navegador de proyectos, en el centro los editores usados para la manipulación de archivos relacionados con la depuración y en la derecha, una vista jerárquica o esquemática de los editores, lo que nos permitirá tener una visión más simplificada de cómo está estructurada la información.

Finalmente, en la parte inferior hay una serie de ventanas con propósitos específicos que servirán como auxiliares para distintas actividades realizadas durante la interacción del usuario con la herramienta. La sugerencia es que este último grupo sea minimizado con la finalidad de tener un campo de visión más amplio; cuando sea necesario su uso bastará con restaurarlo.

5.3. Primera fase, depuración declarativa

Para proceder a la depuración declarativa de programas Erlang en su modalidad más estándar bastará con posicionarse en la vista Erlang Declarative Debugger que tiene el aspecto de la figura 5.7.

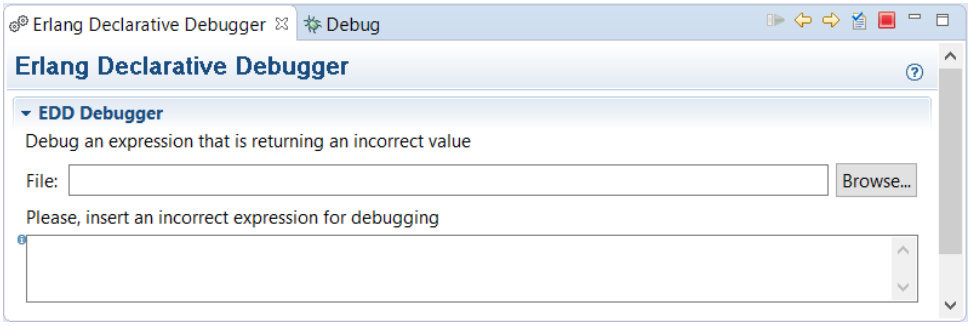

Figura 5.7 Vista de depuración EDD sobre Eclipse

Dicha ventana es la que gobierna la interacción con la depuración, así que, antes de comenzar propiamente con la depuración, se deberá proporcionar la ubicación del archivo que se desea depurar y la llamada de la función a evaluar, que creemos está dando resultados no deseados.

Es importante introducir dichos datos para poder iniciar la sesión de depuración, de no ser así, nos saldrá un dialogo donde se nos indique que dichos datos deben ser introducidos previamente.

Una vez rellenados, podremos entonces proceder al arranque del depurador presionando el botón de inicio o *play*, localizado en la barra de herramientas de dicha ventana, para iniciar con la serie de preguntas que nos irá lanzando el depurador con la finalidad de detectar la raíz del problema.

Una vez arrancado el servidor, se abrirá el editor del archivo fuente en depuración (si éste no estuviese abierto ya), con la finalidad de ir resaltando la línea que se está depurando según haya coincidencia entre el tipo de sentencia y la pregunta realizada por el depurador.

Así pues, al igual que hicimos con la depuración sobre EDD, vamos a proceder de la misma forma pero ahora utilizando la herramienta E-EDD:

- Seleccionamos el archivo Erlang a depurar e introducimos la cláusula errónea; a continuación arrancamos la sesión de depuración, ver figura 5.8.

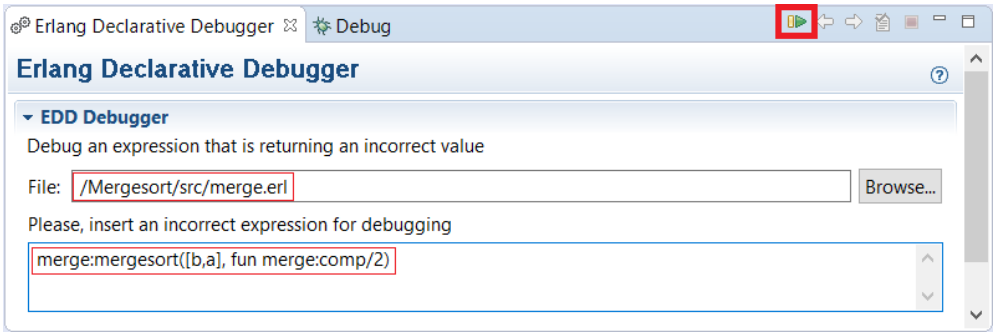

Figura 5.8 Ejemplo de inicio de depuración

- La sesión de depuración nos dará la primera pregunta a verificar; con ella se abrirá el editor Erlang, el cual contiene el código fuente del archivo en depuración. Allí se verá resaltado el extracto de código en el cual debemos centrar nuestra atención (ver figura 5.9).

  Como parte opcional se puede verificar la vista esquemática, para hacernos una idea del árbol de ejecución por el cual navegará el depurador.
- Para el caso propuesto en la figura 5.9. responderemos "No", dado que el resultado esperado en esa instrucción es incorrecto: no corresponde con lo esperado ya que la función `merge` devuelve una lista no ordenada.
- De momento, la información obtenida es insuficiente para detectar la causa del problema, así que se vuelve a formular una cuestión más (figura 5.10).
- En este caso la expresión será respondida como "Confiable" dado que se trata de una función que consideramos suficientemente sencilla y confiamos que está bien definida. Con ello quedará registrada como tal y no volverá a ser cuestionada.
- Una vez más obtenemos una nueva interrogante (figura 5.11).

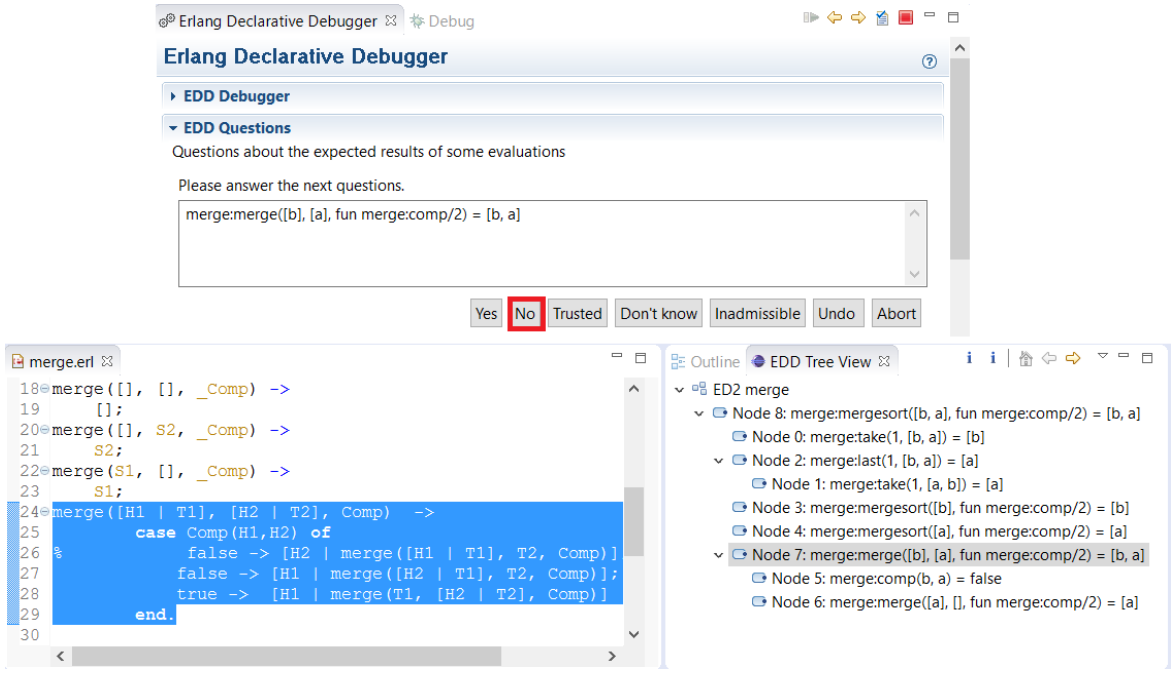

Figura 5.9 Primera pregunta, selección del editor y vista de árbol del caso de prueba

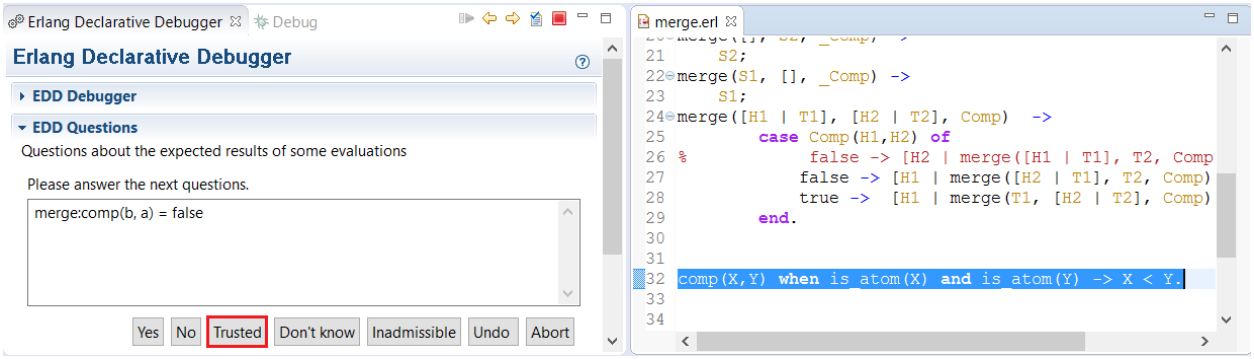

Figura 5.10 Relación entre la segunda pregunta y la sentencia en el editor

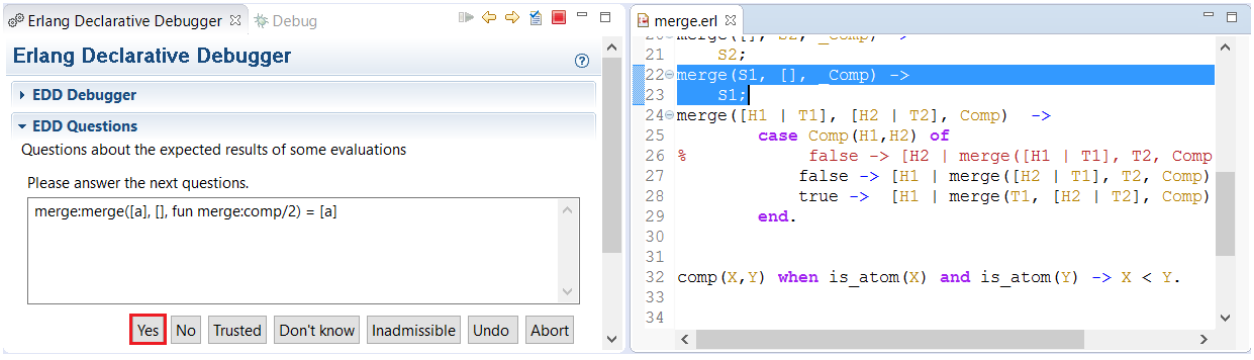

Figura 5.11 Relación entre la tercera pregunta y la sentencia en el editor

- En esta ocasión se trata de una expresión a la cual responderemos con un “Sí”, dado que el resultado de la evaluación se corresponde con lo esperado: la función `merge` para la lista [a] combinada con la lista vacía es la lista ordenada [a].
- Finalmente, el depurador ha sido capaz de identificar la causa del problema; para ello nos avisará mediante un diálogo de cuál es la sentencia que está causando el problema, quedando resaltada además en el editor (figura 5.12).

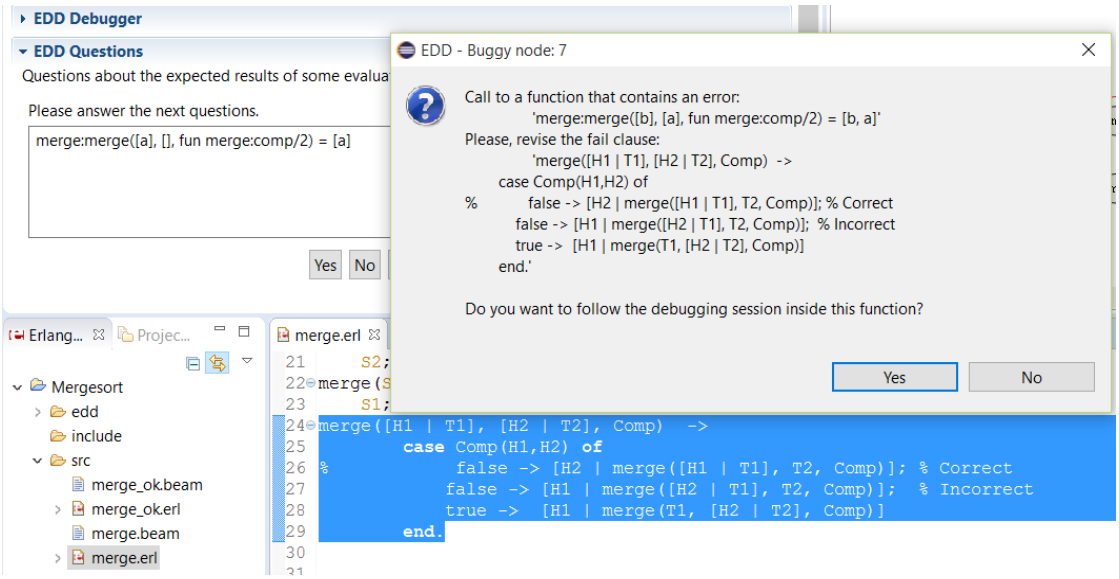

Figura 5.12 Fin de la depuración declarativa estándar y detección de la sentencia errónea

- Con la detección del fallo se preguntará también al usuario si desea parar ahí o continuar con la depuración en profundidad usando esta última sentencia errónea exclusivamente.
- En este caso vamos a decir que “No” dado que en el apartado siguiente se explicará con más detalle en qué consiste este tipo de depuración.

Llegando a este punto el programador deberá revisar ese extracto de código erróneo y realizar los cambios que considere pertinentes para corregir el fallo.

Veremos también que se ha generado un directorio llamado “edd” en el proyecto donde estaba contenido el archivo Erlang que fue usado durante la depuración; allí están contenidos una serie de archivos que han sido utilizados durante la depuración (figura 5.13).

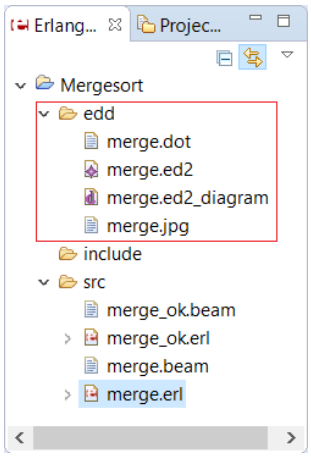

Figura 5.13 Archivos generados durante la depuración declarativa estándar

Entre ellos está un archivo con sintaxis dot que contiene el grafo que representa el árbol de ejecución, el cual se va actualizado según el usuario vaya contestando; se utiliza durante la sesión de depuración (figura 5.14). Veremos que algunos nodos están coloreados (nodos correctos en verde, incorrectos en rojo y en amarillo aquellos donde el usuario no estaba seguro): representan el camino seguido hasta dar con el problema.

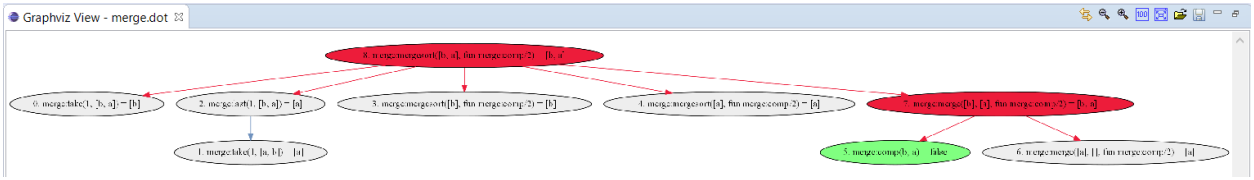

Figura 5.14 Árbol de ejecución bajo Graphviz (*.dot)

Los archivos con extensión *.ed2 y *.ed2_diagram representan el mismo árbol de ejecución pero con tecnología EMF [A10] y GMF [31]. La principal diferencia es que en el caso anterior se renderiza en una imagen mientras que, con estos últimos, podríamos interactuar con el árbol dado que los nodos son seleccionables (figura 5.15).

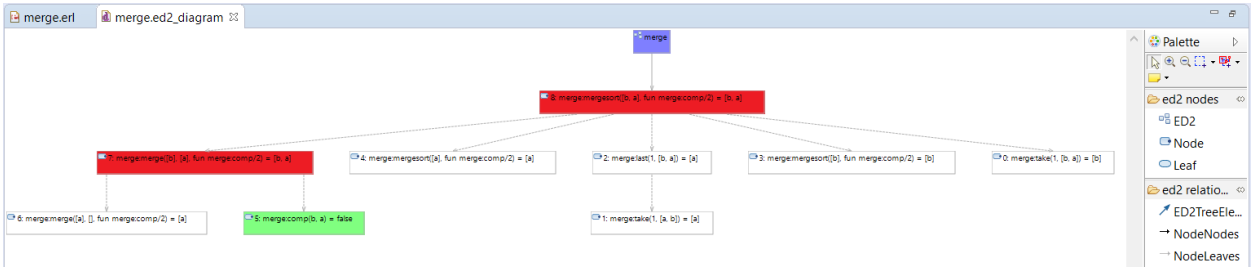

Figura 5.15 Árbol de ejecución bajo EMF y GMF (*.ed2 y *.ed2_diagram)

5.4. Segunda fase, depuración con zoom

La depuración con zoom está basada en la depuración declarativa estándar descrita anteriormente, con la salvedad de que en este caso las respuestas serán dinámicas puesto que la construcción de las preguntas varía también. No es posible definir un árbol íntegro de posibles soluciones pero sí podemos partir de un subconjunto finito. Entraremos en más detalles respecto a la sentencia que se esté depurando y para ello es necesario ampliar el repertorio de posibles respuestas, con la finalidad de obtener datos más precisos para formular la pregunta que nos ayude en la labor de la detección del fallo según por donde se vaya encaminando la depuración.

Recordemos que, en el apartado anterior, en el diálogo de *buggy node* se nos preguntaba si deseábamos continuar con la depuración; pues bien, si retomamos la depuración hasta ese punto y decimos que "Sí" en esta ocasión, procederemos a entrar en la depuración con zoom:

- Se muestra una nueva pregunta al usuario (figura 5.16), esta vez mucho más completa, que deberá ser analizada y respondida con las posibles opciones disponibles para ello.

  Dentro de las posibles opciones vemos que la opción 4 es la que mejor responde a los resultados que esperamos.

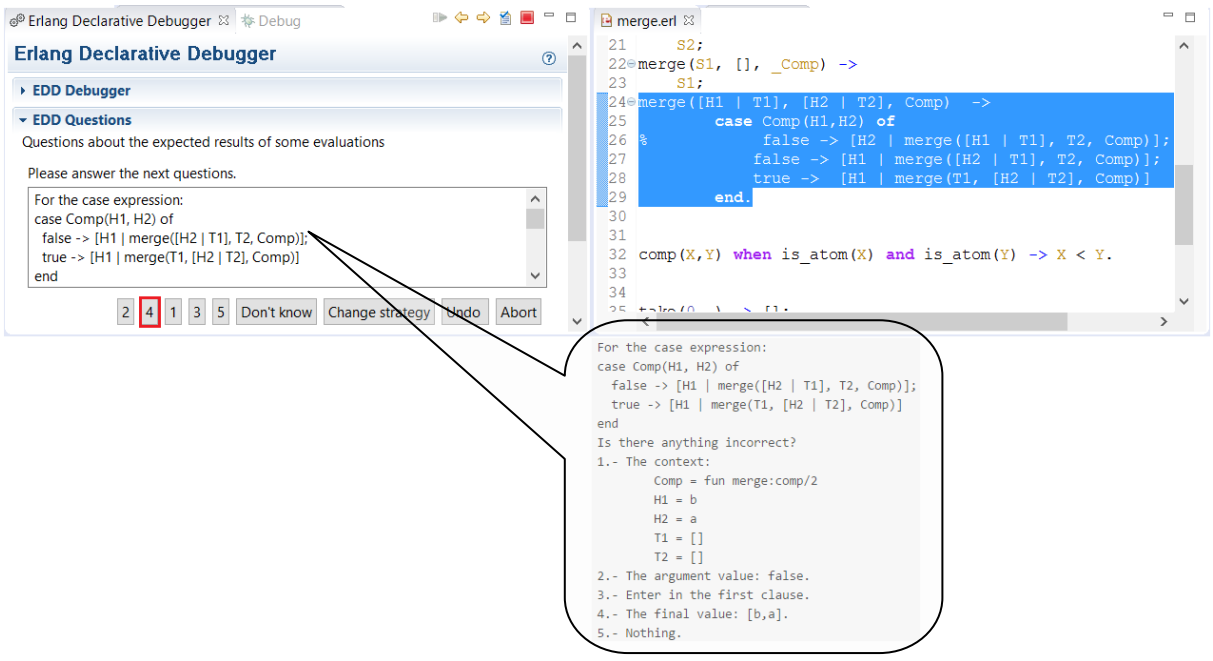

Figura 5.16 Pregunta con la depuración con zoom

- En este caso en particular, con la respuesta anterior concluye el análisis del problema, dado que con esa respuesta el depurador ha sido capaz de obtener la línea exacta donde se está produciendo el fallo (figura 5.17).

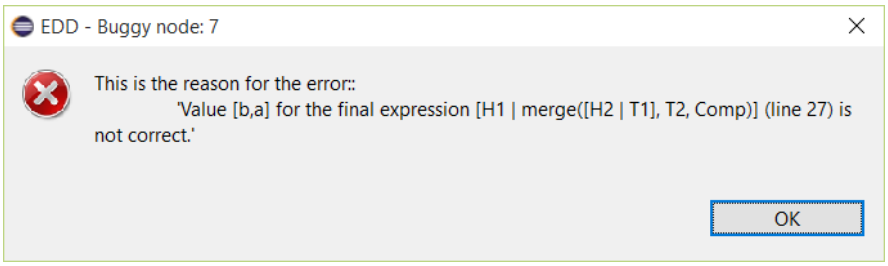

Figura 5.17 Fin de la depuración con zoom y línea errónea

Al igual que en el caso anterior, hemos obtenido una serie de archivos más que representan los árboles de ejecución utilizados para la depuración con zoom (figura 5.18).

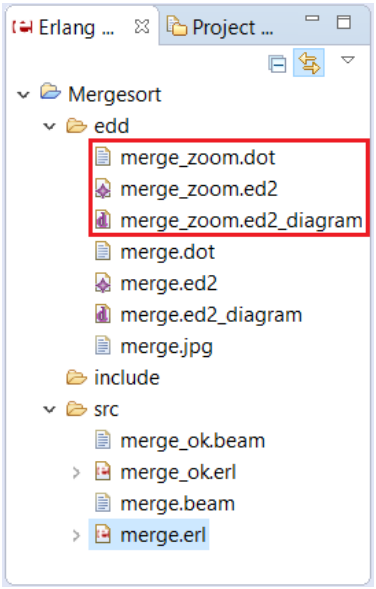

Figura 5.18 Archivos generados durante la depuración con zoom

El árbol de ejecución utilizado para el ejemplo que venimos desarrollando se corresponde con la figura 5.19 donde, al igual que se ha explicado anteriormente, los colores indican el camino seguido hasta encontrar la causa más específica del problema.

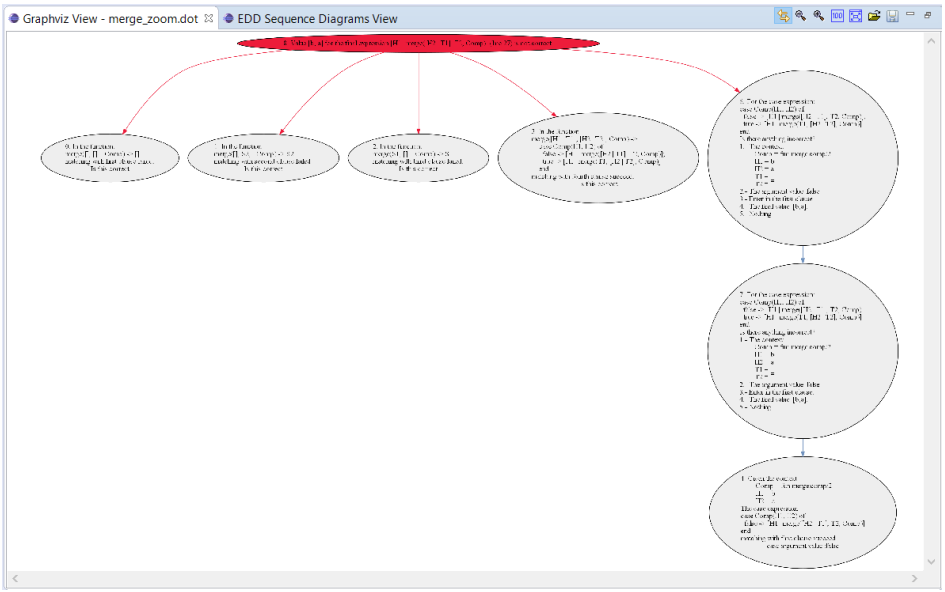

Figura 5.19 Árbol de ejecución con zoom bajo Graphviz

De la misma forma obtenemos los ficheros en su versión EMF y GMF (figura 5.20).

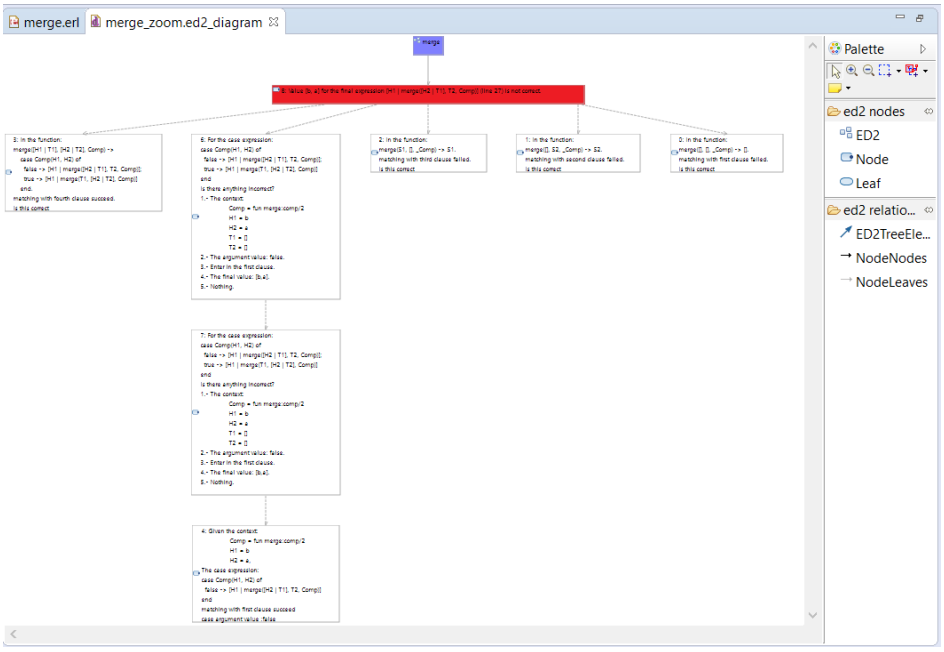

Figura 5.20 Árbol de ejecución con zoom bajo EMF y GMF

En lo que respecta a éstos árboles de ejecución es importante mencionar que existe un subconjunto de preguntas dinámicas que serán calculadas y formuladas sobre la marcha. No es posible representarlas en el grafo dado el enorme universo de posibilidades.

Las preguntas tendrán el mismo aspecto que las anteriores, tal y como muestra la figura 5.21, la única diferencia es que el propio motor de depuración es el encargado de calcularlas dependiendo de la estrategia utilizada en ese momento.

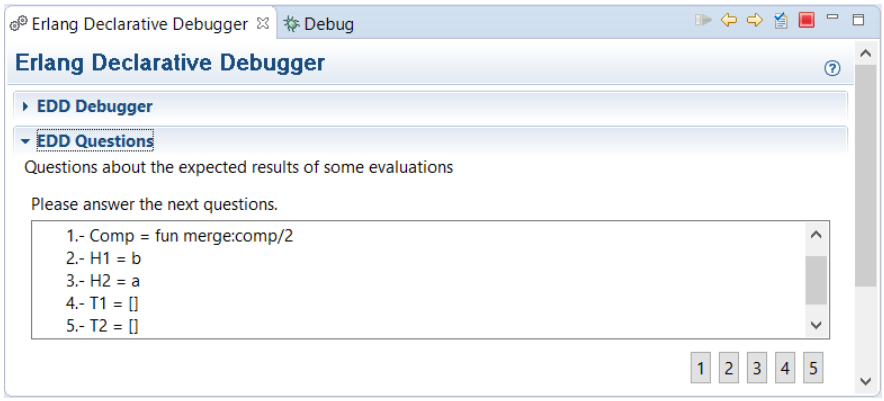

Figura 5.21 Ejemplo de pregunta calculada dinámicamente

Después del inciso anterior y siguiendo con los árboles de ejecución, vemos que otra alternativa de representar la información es utilizar un modelo jerárquico descrito en la figura 5.22, el cual muestra en una estructura de árbol las distintas alternativas posibles existentes en el modelo relacionado al programa en depuración. Bajo dicho esquema será posible mostrar un decorador que represente el estado de la respuesta dada, similar a como se hacía en el grafo al colorear, lo que nos permitirá tener una visión más amplia del camino realizado por el depurador.

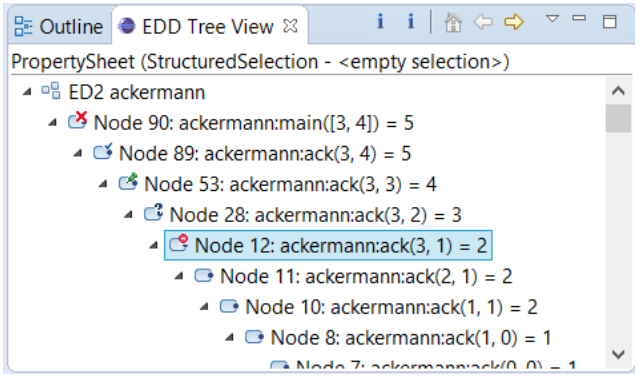

Figura 5.22 Vista jerárquica del árbol de ejecución con decoradores

Una vez llegado hasta aquí, es posible dimensionar perfectamente la forma de operar del depurador declarativo.

5.5. Depuración concurrente

Modelo de depuración concurrente, en el cual se analiza el paso de mensajes entre los procesos concurrentes o nodos Erlang con el fin de detectar anomalías y errores, siguiendo un modelo de depuración similar al propuesto anteriormente.

Se contribuye con parte de la investigación para su implementación y uso posterior.

Sin integración, dado que la versión de EDD aún no lo tiene concluida y en E-EDD ha quedado pendiente su implementación a falta de tiempo.

Al trabajar en un entorno concurrente, debemos recordar que se hará a través de mensajes asíncronos, los cuales podrán ser representados utilizando diagramas de secuencia en donde cada nodo representará una línea de vida. Dichos diagramas sustituyen a los árboles de ejecución vistos anteriormente. Al utilizar los diagramas de secuencia podemos observar cómo se está produciendo la comunicación entre todos los nodos que participen en la ejecución de forma que es posible hacer un seguimiento de la ejecución del programa.

Es posible utilizar editores gráficos, ya existentes, para la creación de diagramas de secuencia en Eclipse, como Papyrus, que ofrecen una serie de herramientas de modelado bajo UML. El único inconveniente aquí es el hecho de que nos interesaría la creación programática de los mismos, cosa que es posible pero requiere conocimientos avanzados en la herramienta y una investigación previa para familiarizarse con la misma.

En la depuración concurrente es importante estar generando diagramas constantemente, dado que la idea es ir reflejando cada paso de mensaje durante la depuración; es por ello que, para facilitar el proceso de creación de dichos diagramas de una forma simple y cómoda, se ha decidido optar por la herramienta pic2plot [A11] de la familia UMLGraph. La razón principal es que, mediante el uso de pequeños scripts [A12] *.pic como el de la figura 5.23, podremos generar imágenes de diagramas de secuencia como el de la figura 5.24 muy fácilmente, de forma similar a como sucede con el uso de archivos *.dot, los cuales se renderizaban en imágenes que representan grafos.

Como se puede observar, es realmente muy sencilla la generación de los diagramas, por lo que la idea en este apartado es la de estar generando el script programáticamente cada vez que se reciba un paso de mensaje. De esta forma se puede tener una foto de la comunicación en cada paso de la depuración concurrente.

```
#/usr/bin/pic2plot -Tps
#
# Run as pic filename | groff | ps2eps
#
# DNS query collaboration diagram
#
#

.PS

copy "sequence.pic";

boxwid = 1.3;

# Define the objects
object(B,":Web Browser");
object(W,":Workstation Kernel");
object(S,":Server Kernel");
object(D,":DNS Server");
step();

# Message sequences
active(B);
active(D);
active(W);
active(S);
message(D,S,"select");
inactive(D);
message(B,W,"socket");
message(B,W,"connect");
message(B,W,"sendto");
message(W,W,"send packet");
message(W,S,"DNS A query");
message(B,W,"recvfrom");
inactive(B);
message(S,S,"receive packet");
rmessage(S,D,"select returns");
active(D);
message(D,S,"recvfrom");
message(D,S,"sendto");
message(S,S,"send packet");
message(S,W,"DNS A reply");
message(W,W,"receive packet");
rmessage(W,B,"recvfrom returns");
active(B);
message(B,W,"close");

complete(B);
complete(W);
complete(S);
complete(D);

.PE
```

Figura 5.23 Ejemplo de un archivo *.pic (script UMLGraph)

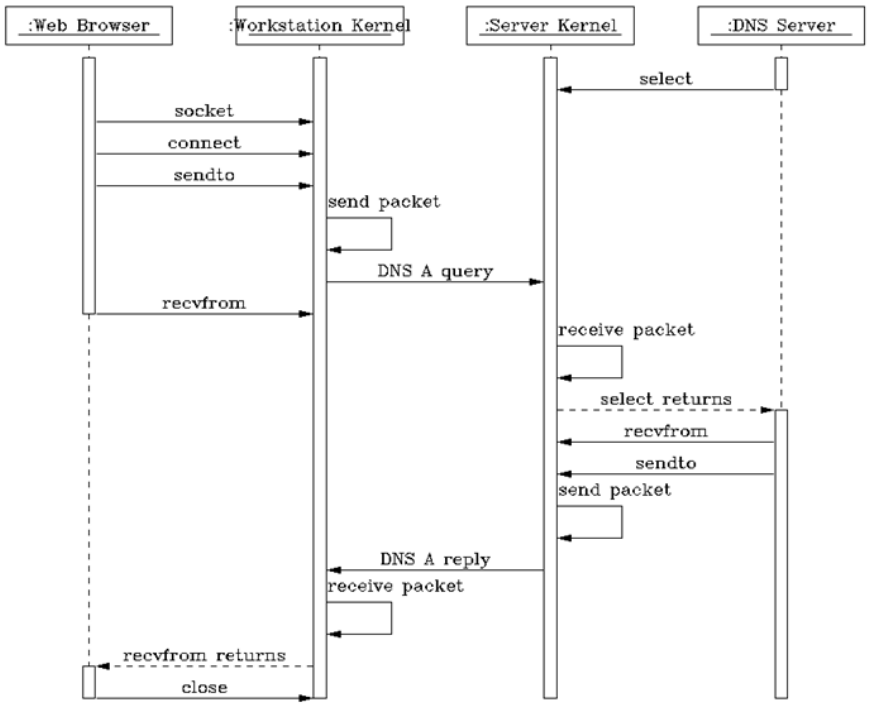

Figura 5.24 Ejemplo de diagrama de secuencia UMLGraph

6. Arquitectura E-EDD y entorno de desarrollo

Una vez explicada la parte operativa de la herramienta, toca el turno de los detalles técnicos que veremos resumidos en los siguientes apartados.

La herramienta de depuración E-EDD está implementada en forma de plugins para Eclipse, tal y como se ha comentado anteriormente, y a su vez tiene una serie de dependencias a plugins de terceros para poder operar correctamente. Para entender mejor cómo ha sido implementada veamos antes una descripción breve de lo que es la arquitectura propuesta del proyecto.

6.1. Visión general E-EDD

Como hemos explicado en la sección 5, E-EDD está basado en Eclipse y en plugins de terceros. La figura 6.1 muestra cómo están interconectados los distintos componentes en los que se basa para dar como resultado final el proyecto E-EDD.

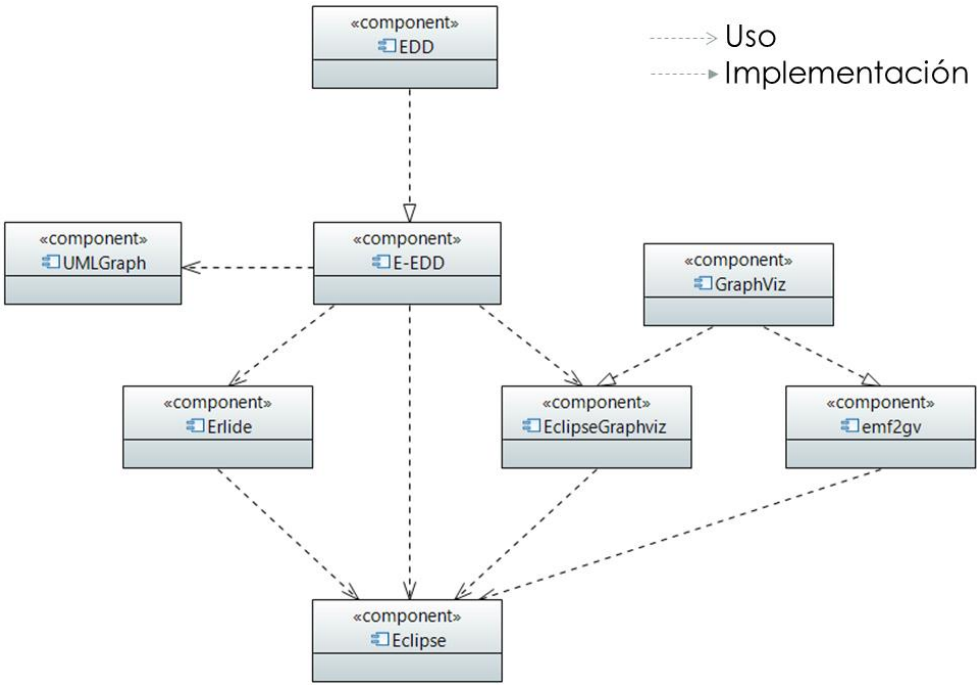

Figura 6.1 Visión general E-EDD

E-EDD es la implementación gráfica del proyecto EDD. Asimismo utiliza a su vez el proyecto Erlide y el proyecto EclipseGraphviz, cuyas contribuciones a Eclipse son necesarias para la funcionalidad total de la herramienta en cuestión.

UMLGraph es, a su vez, una herramienta que permite la especificación declarativa para crear diagramas de clases y diagramas de secuencia UML.

Graphviz [32] es un software de visualización gráfica, que no es otra cosa que la representación gráfica de información estructurada. Dicha herramienta está presente en herramientas como EclipseGraphviz, que encapsula la versión bajo el IDE de Eclipse, y Emf2gv [33], que la utiliza para transformar modelos EMF (Eclipse Modeling Framework) en representaciones bajo Graphviz (véase el apéndice D).

6.2. Visión estructural de los plugins E-EDD

La figura 6.2 muestra cómo están interconectados los distintos plugins que componen el proyecto E-EDD.

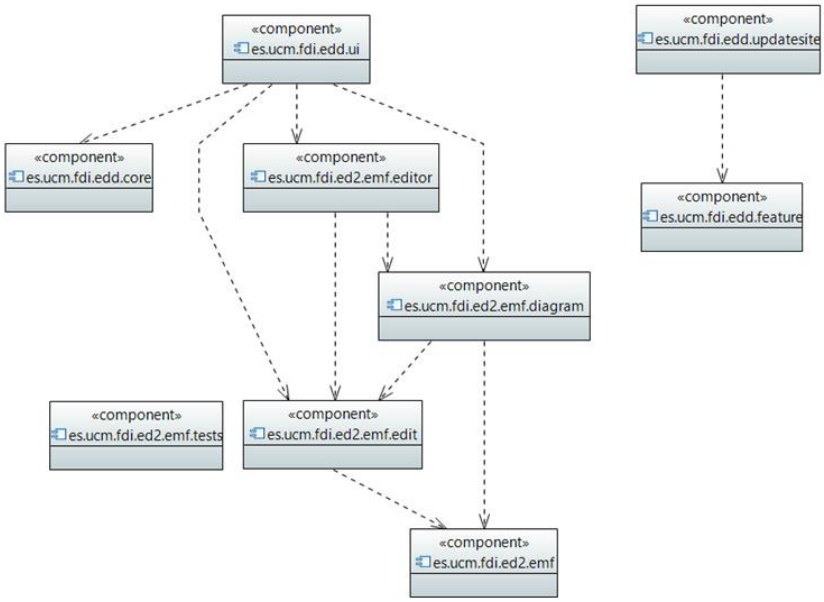

Figura 6.2 Lista de plugins que componen el proyecto E-EDD

El plugin core es el que tiene la lógica de negocio de la aplicación, además de bibliotecas de utilidades varias. Dicho plugin contiene el módulo de comunicación Java-Erlang descrito en la sección 7, además de clases de utilidad para la manipulación de los archivos dot, json y pic (véanse los apartados 5.3, 5.4 y 5.5).

Todo lo referente al aspecto visual (perspectiva, vistas, acciones, comandos, editores, menús, etc.) está comprendido en el plugin UI, que es donde se realizan las extensiones necesarias para implementar las contribuciones en Eclipse.

Todo proyecto de plugin que contribuye a la interfaz de usuario contiene un archivo denominado `plugin.xml` cuyo editor se muestra en la figura 6.3; allí se definen todos los puntos de extensión utilizados y sus implementaciones.

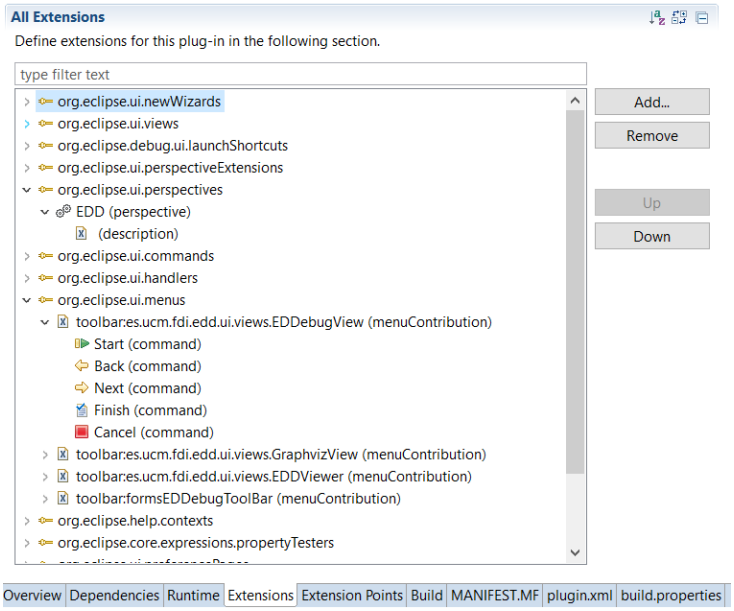

Figura 6.3 Puntos de extensión utilizados para la UI

Como introducción al desarrollo de plugins se ha creado el apéndice B que muestra mediante un pequeño ejemplo la forma en que se deben hacer las implementaciones de los puntos de extensión en Eclipse, dado que no es el propósito detallarlo en este documento pero sí relevante mencionarlo. La figura 6.3 describe parte de los puntos de extensión usados por E-EDD.

Finalmente, el resto de proyectos, están relacionados con EMF (Eclipse Modeling Framework) y GMF (Graphical Modeling Framework), utilizados para la creación de las herramientas tanto del modelo jerárquico (véase sección 8) como el editor gráfico respectivamente (véase el apéndice C), usados principalmente para las representaciones de los árboles de ejecución generados durante la depuración.

El proyecto de sitio de actualizaciones, que utiliza a su vez el proyecto de características, es el que contiene los plugins que han de ser instalados para trabajar con E-EDD (véase el apéndice B).

7. Comunicación Java-Erlang

Como ya se ha mencionado anteriormente, E-EDD es una implementación gráfica de EDD; para conseguirlo es necesario establecer un mecanismo de comunicación entre la tecnología Java y Erlang. Para ello fue necesario utilizar JInterface, un API que ofrece una serie de herramientas para la comunicación con los procesos Erlang. En el plugin core reside gran parte de la lógica necesaria para realizar dicha comunicación y para poder integrarla a su vez con Eclipse.

El diagrama mostrado en la figura 7.1 representa el modelo estático de clases que participan en la comunicación: `Erlang2Java` es la clase principal que orquesta la comunicación, encargada de iniciar el nodo servidor y un nodo cliente (que son representados en las clases `ErlangServer` y `ErlangClient`, respectivamente) para establecer el envío/recepción de mensajes entre Erlang y Java; el resto de clases son utilizadas como auxiliares por la clase principal para el procesado de entradas, salidas y errores que ocurran durante la comunicación entre procesos. Además, existen otra serie de clases que servirán para listar los nodos Erlang activos que haya arrancado la herramienta para proceder a su finalización.

Es importante mencionar que se hace uso de la línea de comandos para ejecutar el servidor Erlang (`edd_jserver.erl`) en segundo plano; se trata de una pieza importante para la comunicación cliente - servidor.

Finalmente, hay un conjunto de clases cuyo común denominador es la clase cliente; dichas clases representan un modelo de abstracción de toda la información procesada por cada sesión del depurador. En ella se almacenarán todos los datos que sea necesario recuperar durante la depuración; por ejemplo, el número de pregunta activa, información extra de los nodos, etc.

Dicho modelo es el encargado de ir ofreciendo información a la interfaz de usuario cuando algún dato sea solicitado. Es por ello que, en caso de verse incrementadas las funcionalidades de E-EDD, será importante considerar las clases aquí mencionadas.

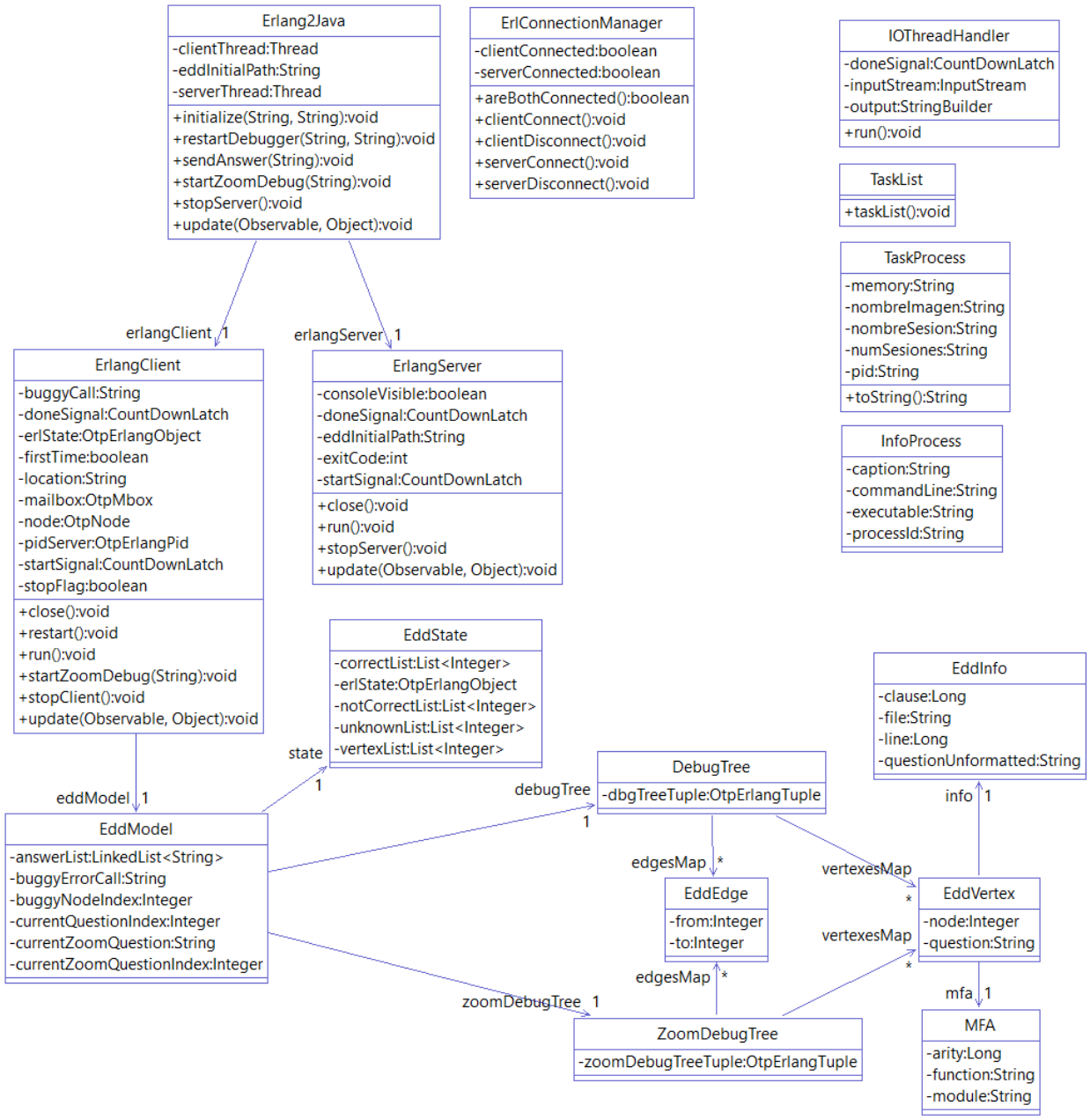

Figura 7.1 Diagramas de clases del módulo de comunicación Java-Erlang

Para simplificar el procedimiento y dar una visión más exacta del procedimiento de comunicación, procedemos a explicar la visión dinámica de la comunicación detallada en figura 7.2, donde podemos observar los nodos Erlang que serán creados desde Java y que son los encargados del procesamiento de cada paso de mensajes que se produzca en una u otra dirección:

1. El primer paso es notificar al cliente cuál va a ser el nodo servidor que será el encargado de procesar todas las peticiones en la sesión de depuración. Para ello, una vez que se inician tanto el servidor como el cliente, se envía un mensaje *ready* con su *pid* asociado (identificador de proceso, desde el servidor al cliente) notificándole que está listo.
2. El cliente responde con un mensaje *buggy_call* donde anuncia la ubicación y la expresión a depurar.
3. El servidor envía un mensaje *dbg_tree* con la información completa para proceder en la depuración, creando un árbol de ejecución para la toma de decisiones.
4. A modo de bucle, mientras no sea encontrado el *buggy_node* habrá un intercambio pregunta - respuesta entre cliente y servidor
    a. El servidor, mediante el algoritmo aplicado, hará una pregunta con la finalidad de calcular el siguiente paso a dar.
    b. El cliente deberá retroalimentar la pregunta lanzada por el servidor contestando según lo que esperaría como resultado de la misma.
5. Cuando el servidor, siguiendo el algoritmo propuesto, haya descifrado el fallo devolverá el *buggy_node* con la instrucción incorrecta.

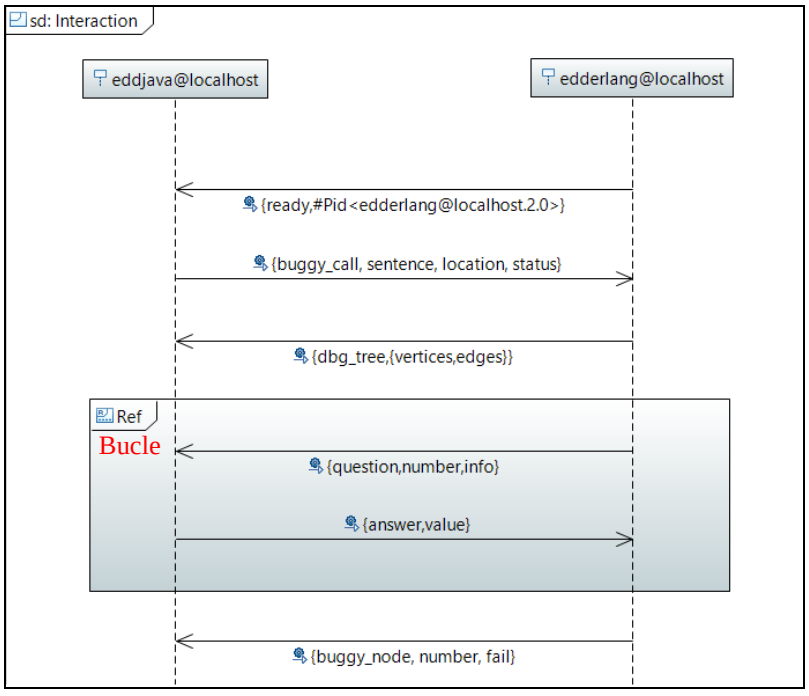

Figura 7.2 Comunicación Java-Erlang

Esta primera fase es una depuración estándar, que termina cuando ha sido descubierto el *buggy_node*; sin embargo, se ofrece la posibilidad de continuar la depuración haciendo uso de una depuración más exhaustiva haciendo una introspección en la sentencia errónea, exclusivamente con la finalidad de obtener más precisión en la misma.

En la figura 7.3, que extiende a la figura 7.2, podemos observar un paso de mensajes similar al expuesto con anterioridad, con la diferencia de que, en este caso, tiene información más precisa respecto a la cláusula errónea.

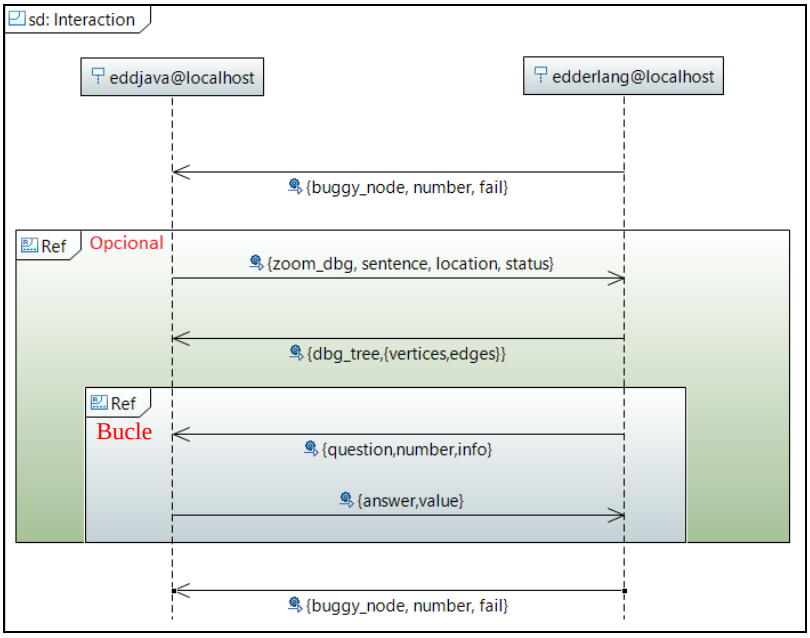

Figura 7.3 Comunicación Java-Erlang con zoom

Debemos mencionar que la depuración con zoom tiene un universo muy amplio de posibilidades, que dependerá de la complejidad de la sentencia analizada y la estrategia que esté empleando el depurador.

Aquí es habitual encontrar respuestas dinámicas y preguntas procesadas a partir de la estrategia usada, las cuales se irán calculando según el camino que se vaya siguiendo en la depuración. Es importante prestar atención a las cláusulas mostradas y examinarlas con cuidado con la finalidad de aportar información verídica que pueda facilitar el procesamiento de detección de errores bajo los algoritmos empleados.

Es importante mencionar que cada paso de mensajes, tanto de envío como de recepción, debe ser procesado como si de nodos Erlang se tratase; para ello debemos conocer la lista de objetos que ofrece el API de JInterface para trabajarlos desde Java.

La figura 7.4 lista algunos de los más representativos a modo de introducción, en los que, además, se puede observar fácilmente la equivalencia que tendría con objetos Java si suprimimos el prefijo `OtpErlang`.

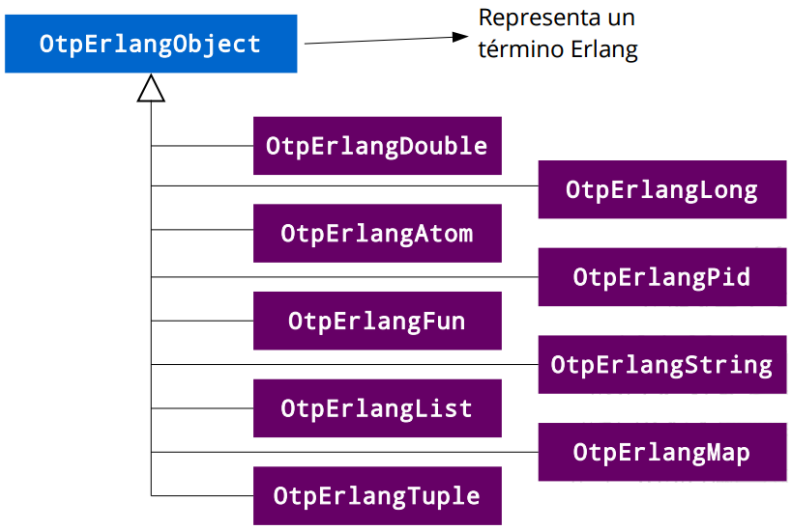

Figura 7.4 Jerarquía de algunos objetos OtpErlangObject

Veamos por ejemplo, cómo se debería codificar una llamada "`buggy_call`" representada en el diagrama de secuencia de la figura 7.2, que hace un paso de mensaje desde el nodo `edd_java@localhost` al nodo `edderlang@localhost`.

Si lo hiciésemos desde una consola Erlang debería tener el formato:

`{buggy_call, <C>,<L>,<S>}` donde:

C - es la instrucción errónea,

L - la ubicación del archivo fuente a depurar y

S - el estado del servidor en ese momento (el átomo `none` cuando se inicia).

Un mensaje bien formado sería, por ejemplo:

```
{buggy_call,"merge:mergesort([b,a], fun merge:comp/2)","D:/examples",none}
```

Una vez definido el tipo de mensaje y su formato lo vamos a codificar haciendo uso de la jerarquía de objetos OtpErlangObject. La figura 7.5 muestra la forma de implementar la función *buggy call* en Java compuesta de un átomo, dos strings y un átomo, respectivamente. Además se muestra la forma de convertir el mensaje en una tupla mediante el método `sendMessagge()`.

```java
/**
 * Send reply with buggy call and its location
 */
private void sendBuggyCall() {
    OtpErlangObject[] reply = new OtpErlangObject[4];
    reply[0] = new OtpErlangAtom("buggy_call");
    reply[1] = new OtpErlangString(buggyCall);
    reply[2] = new OtpErlangString(location);
    if (firstTime) {
        erlState = new OtpErlangAtom("none");
        firstTime = false;
    }
    reply[3] = erlState;

    sendMessage(pidServer, reply);
}

/**
 * Sends a message given the source node.
 *
 * @param pidSender
 *          The source node.
 * @param message
 *          The message.
 */
private void sendMessage(OtpErlangPid pidSender, OtpErlangObject[] message) {
    OtpErlangTuple tuple = new OtpErlangTuple(message);
    System.out.println("--> SEND message [" + pidSender + "]: " + tuple.toString());
    mailbox.send(pidSender, tuple);
}
```

Figura 7.5 Codificación del mensaje buggy_call

8. Metamodelo ED2

Para poder representar en Eclipse de forma gráfica y jerárquica el modelo de objetos usado por E-EDD es necesario partir de un metamodelo similar al de la figura 8.1 que represente los objetos que consideramos formarán parte de las representaciones visuales en Eclipse.

Para nuestro propósito, dado que el esquema es una representación en forma de árbol, el metamodelo es muy sencillo: contamos con el nodo raíz (`Model`) que está compuesto por un nodo (`ED2`) que representa el modelo de estudio que, a su vez, contiene una lista de elementos que serán representados como nodos (`Node`) u hojas (`Leaf`) en nuestro modelo; estos últimos heredan de la clase abstracta `TreeElement`.

La enumeración `TreeElementType` es utilizada para marcar y decorar los nodos que hayan sido respondidos con alguna de las respuestas dadas.

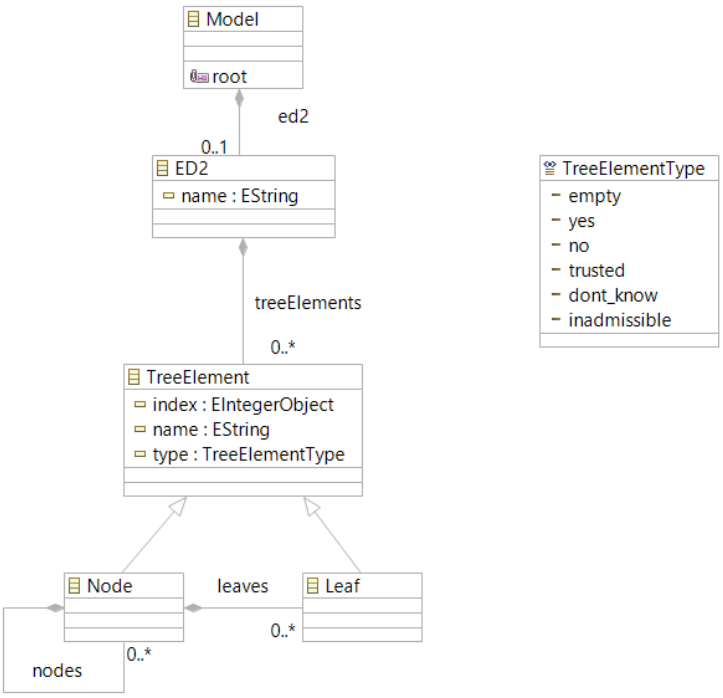

Figura 8.1 Metamodelo ed2

Dicho metamodelo será el encargado de orquestar la creación de nodos que utilizaremos en la creación de los árboles de ejecución durante la depuración.

Para entender cómo se construye es necesario seguir el apéndice C que explica con detalle como es el proceso de creación.

Debemos mencionar también que EMF es un framework y una herramienta de generación de código para la construcción de herramientas basadas en modelos estructurados, donde los elementos deberán ser creados utilizando una factoría (`Ed2Factory`) para crear los distintos objetos de la jerarquía como se detalla en la figura 8.2, (véase las variables model, ed2 y node y su correspondencia con la figura 8.1).

```
private EObject createInitialModel() {
    Model model = Ed2Factory.eINSTANCE.createModel();
    ED2 ed2 = Ed2Factory.eINSTANCE.createED2();
    ed2.setName(modelName);
    model.setEd2(ed2);

    Map<Integer, Node> nodesMap = new HashMap<Integer, Node>();
    Map<Integer, EddVertex> vertices = eddModel.getDebugTree().getVertexesMap();
    for (Entry<Integer, EddVertex> entry : vertices.entrySet()) {
        EddVertex vertice = entry.getValue();
        Node node = Ed2Factory.eINSTANCE.createNode();
        int index = vertice.getNode();
        node.setIndex(index);
        node.setName(index + ": " +vertice.getQuestion());

        nodesMap.put(index, node);
    }

    List<EddEdge> edges = eddModel.getDebugTree().getEdgesMap();
    for (EddEdge edge : edges) {
        int from = edge.getFrom();
        int to = edge.getTo();

        Node source = nodesMap.get(from);
        Node target = nodesMap.get(to);
        source.getNodes().add(target);
    }

    int root = eddModel.getDebugTree().getEdgesMap().size();
    EList<TreeElement> elements = ed2.getTreeElements();
    elements.add(nodesMap.get(root));

    return model;
```

Figura 8.2 Creación de objetos usando la factoría Ed2Factory

Con esto conseguiremos la representación estructurada EMF similar al de la figura 8.3, en la izquierda en formato XMI como fuente y en la derecha en estructura de árbol, que será de uso múltiple y que permite la interacción con otras ventanas en el entorno Eclipse, además de permitir la navegación entre objetos de forma rápida y simple.

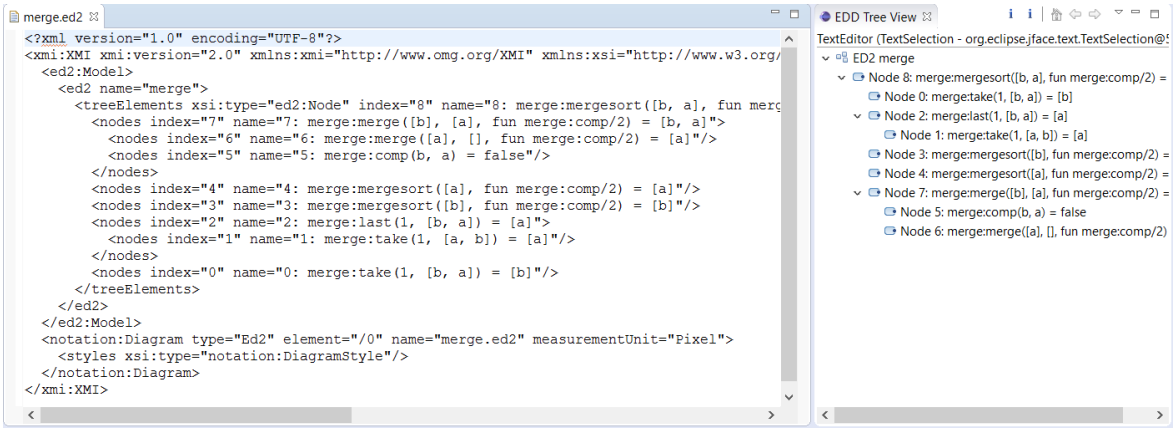

Figura 8.3 Estructura y jerarquía de un modelo ed2 bajo EMF

Además, dicho modelo permitirá trabajar con el entorno GMF para la creación gráfica del contenido (figura 8.4), tomando como referencia la representación en EMF, cuyo editor se obtiene cumplimentando todos los pasos descritos en el Apéndice C.

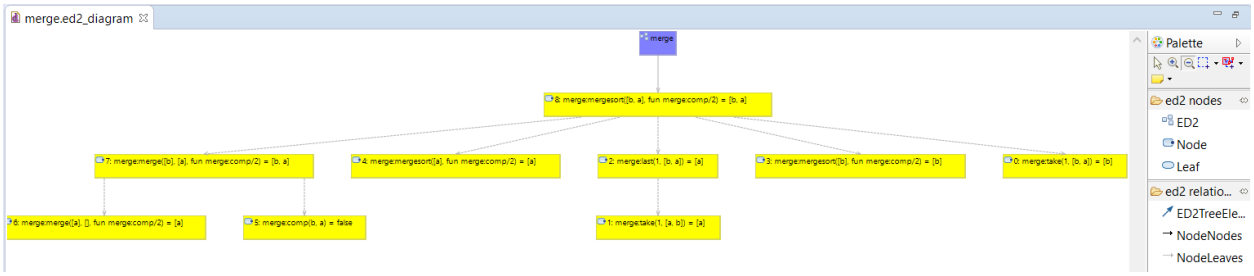

Figura 8.4 Editor de modelos ed2_diagram bajo GMF

Al igual que sucede con los puntos de extensión de los plugins para Eclipse usados en la sección 6, el uso de los metamodelos tanto para EMF como para GMF es muy extenso de detallar por lo cual la presente memoria solo pretende dejar de manifiesto las nociones mínimas necesarias para entender y conceptualizar el trabajo realizado. Para más información consulte las referencias [L7] y [L8].

9. Conclusiones y trabajo futuro

Como se ha podido observar a lo largo de la presente memoria, el sistema desarrollado ofrece una aportación significativa que contribuye a la mejora de la depuración declarativa que se había estado trabajando hasta ahora en el proyecto EDD.

Además de mejorar la usabilidad, permite hacer entender de una forma menos abstracta lo que está sucediendo durante la depuración. Llevarla a un entorno de desarrollo como Eclipse ha sido acertado, ya que con ello se puede ganar usuarios y, además, futuros colaboradores interesados en aportar mejoras a lo propuesta, dado que es y pretende seguir siendo una infraestructura extensible.

El proyecto presentado es una buena primera aproximación a los objetivos planteados inicialmente: no solo mejora la usabilidad sino que deja un marco de referencia abierto en donde puedan ser ejecutadas más técnicas de depuración declarativa buscando con ello una diversidad algorítmica de mayor alcance. Dichas técnicas podrían ser acogidas e implementadas bajo este mismo entorno con el propósito de enriquecer y dar continuidad a este proyecto.

El presente trabajo podría mejorarse significativamente con el uso de editores gráficos más potentes, los cuales, aparte de mejorar visualmente la interfaz, podrían ofrecer también una parte dinámica e interactiva a la cual pudiera sacarse más partido; por ejemplo, al seleccionar un nodo se podrían realizar evaluaciones puntuales, tratándolo como un objeto más, similar a como sería una vista de evaluación de expresiones, solo que en este caso sería un evaluador de funciones.

Una de las actividades que ha quedado pendiente es la de mejorar la integración de E-EDD con Erlide, con la finalidad de dar más usabilidad al usuario. Entre ellas estaba también la opción de sugerir y evaluar las instrucciones Erlang al momento de la redacción en la vista de depuración, que permita sugerencias y autocompletado para la introducción de sentencias válidas sin necesidad de esperar por un mensaje de error al iniciar la depuración.

Respecto a la ubicación de código fuente y binarios y a la variedad de opciones disponibles en la configuración de proyectos bajo Eclipse, se ha visto la necesidad de realizar mejoras tanto en EDD como en E-EDD para que sean capaces de parametrizarse de acuerdo a las configuraciones que estime oportuno el usuario (múltiples carpetas fuente, niveles de anidación

de directorios, elección del compilador Erlang, referencias a otros proyectos bajo el mismo classpath, etc.).

Esperamos que el presente trabajo refleje lo ambicioso que puede llegar a ser la interacción con el depurador declarativo y la utilidad que puede dar a los usuarios. Nuestra intención durante todo este tiempo siempre ha sido la de hacer llegar la herramienta a más desarrolladores con la finalidad de dar mayor visibilidad y de paso intentar conseguir voluntarios que puedan seguir contribuyendo a su desarrollo. Esperamos que así sea.

10. Conclusions and future work

The system presented in this work provides a significant contribution which improves the way declarative debugging has been used so far in the EDD project.

Besides improving its usability, it eases the understanding of what is happening during the debugging process in a less abstract way. The integration of EDD into a development environment such as Eclipse has been successful, so we expect it can gain users and also developers interested in contributing to future improvements to the proposal, because it intends to remain an extensible infrastructure.

The project presented is a good approximation to the initial objectives: it does not just meet the usability but also leaves a framework of open reference where we can execute more techniques of declarative debugging thereby seeking a broader algorithmic diversity. These techniques could be accepted and implemented under the same environment with the purpose of enriching and giving continuity to this project.

This work could be significantly improved with the use of powerful graphics editors, which, in addition to a visually improvement of the appearance, could also offer a dynamic and interactive part which would provide more utility; for example, when selecting a node it could send specific assessments, treating it as an instance; it would be similar to a view of evaluation of expressions although in this case it would be a function evaluator.

Otra actividad que permanece abierto es el de mejorar la integración de E-EDD con Erlide, con el fin de dar mayor facilidad de uso para el usuario. Entre las opciones también existe la posibilidad de sugerir y evaluar instrucciones Erlang en el momento de la escritura en la vista de depuración declarativa, permitiendo sugerencias de validación y de auto-completar para la introducción de la validación de frases sin esperar un mensaje de error al iniciar la depuración.

Regarding the location of the source code and the binary files and the variety of options available in the configuration of projects under Eclipse, it is important to improve both EDD and E-EDD projects to parameterize them according to the user settings (multiple source folders, nested directories levels, choice of Erlang compiler, references to another projects under the same classpath, etc.).

We hope that this work reflects how ambitious can become an interaction with the declarative debugger and the utility that it can offer. Our intention all along has always been in

order to get more developers to gain visibility and incidentally tries to get volunteers to continue to contribute to its development. Let us hope that it will.

Referencias y bibliografía

Referencias

[1] Eclipse Página oficial

http://www.eclipse.org/

[2] Erlang Página oficial

http://www.erlang.org/

[3] EDD (Github)

https://github.com/tamarit/edd

[4] Trello Página oficial

https://trello.com/

[5] Introducing JSON

http://json.org/

[6] Ericsson AB. The Jinterface Package (2015) [en línea]

http://www.erlang.org/doc/apps/jinterface/jinterface_users_guide.html

[7] SWT

https://www.eclipse.org/swt/

[8] JFace

https://wiki.eclipse.org/JFace

[9] EclipseGraphViz (GitHub)

https://github.com/abstratt/eclipsegraphviz

[10] UMLGraph Página oficial

http://www.umlgraph.org/

[11] Papyrus Página oficial

https://eclipse.org/papyrus/

[12] Notas sobre los 20 millones de procesos ejecutados en Erlang

https://groups.google.com/forum/message/raw?msg=comp.lang.functional/5kldn1QJ73c/T3py-yqmtzMJ

[13] Erlang Mnesia

http://www.erlang.org/doc/man/mnesia.html

[14] JDT

http://www.eclipse.org/jdt/

[15] PDE

http://www.eclipse.org/pde/

[16] The OSGi Architecture

http://www.osgi.org/Technology/WhatIsOSGi

[17] Eclipse IDEs

http://www.eclipse.org/ide/

[18] Eclipse Corner, Artículos varios de la comunidad Eclipse
http://www.eclipse.org/articles
[19] Erlide Página oficial
http://erlide.org/
[20] RAE. Definición del término depurar
http://lema.rae.es/drae/?val=depurar
[21] Tipos de errores de programación
http://www.frlp.utn.edu.ar/materias/algoritmos/errores2010.pdf
[22] Microsoft. Standard Debugging Techniques
https://msdn.microsoft.com/en-us/library/windows/hardware/hh439390(v=vs.85).aspx
[23] Los tipos de depuración
http://ordenador.wingwit.com/criticar/pc-troubleshooting/192464.html#.Vdd1tLLtmko
[24] Tracing (software)
https://en.wikipedia.org/wiki/Tracing_(software)
[25] Using the Eclipse Debugger for Beginning Programmers
http://agile.csc.ncsu.edu/SEMaterials/tutorials/eclipse-debugger /
[26] Remote Debugging
http://wiki.freepascal.org/Remote_Debugging
[27] Delta Debugging, From automated testing to automated debugging
https://www.st.cs.uni-saarland.de/dd/
[28] EDD (GitHub). Ejemplo mergesort
https://github.com/tamarit/edd/tree/master/examples/mergesort
[29] EDD (GitHub). Makefile
https://github.com/tamarit/edd/blob/master/Makefile
[30] E-EDD (GitHub)
https://github.com/jsanchezp/e-edd
[31] Graphical Modeling Project (GMP)
http://www.eclipse.org/modeling/gmp/
[32] Graphviz Página oficial
http://www.graphviz.org/
[33] EMF To GraphViz Manual (emf2gv v1.1.0)
http://emftools.tuxfamily.org/help/emf2gv/1.1.0/
[34] Eclipse Modeling Tools
http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/marsr

Artículos

[A1] GANSNER Emden, KOUTSOFIOS Eleftherios and NORTH Stephen. (2006) Drawing graphs with dot. [en línea] dot User's Manual
<http://www.graphviz.org/Documentation/dotguide.pdf>

[A2] RAMIRO Rivadera, Gustavo. Artículo sobre La Programación Funcional: Un Poderoso Paradigma. [en línea]. Cuadernos de la Facultad n. 3, 2008.
<http://www.ucasal.edu.ar/htm/ingenieria/cuadernos/archivos/3-p63-Rivadera.pdf>

[A3] Eclipse Platform Technical Overview. Copyright © 2006 International Business Machines Corp. [en línea].
<https://eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf > [consulta: septiembre 2015]

[A4] BOLOUR Azad (2003) Notes on the Eclipse Plug-in Architecture [en línea] Bolour Computing.
<http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html>

[A5] MELHEM Wassim and GLOZIC Dejan (2003) PDE Does Plug-ins [en línea] IBM Canada Ltd.
<http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html>

[A6] WÖRTHMÜLLER Stefan (2006) Postmortem Debugging
<http://www.drdobbs.com/tools/postmortem-debugging/185300443>

[A7] BRIA, Mike. Forget Your Debugger, Use The "Saff Squeeze" (2008) [en línea]
<http://www.infoq.com/news/2008/11/beck-saff-squeeze>

[A8] CABALLERO Rafael, MARTIN-MARTIN Enrique, RIESCO Adrian and TAMARIT Salvador. A Zoom-Declarative Debugger for Sequential Erlang Programs (extended version). (2014) [en línea] Informe técnico SIC02/14. Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid.
<http://maude.sip.ucm.es/~adrian/files/zoomTR.pdf>

[A9] CABALLERO Rafael, MARTIN-MARTIN Enrique, RIESCO Adrian and TAMARIT Salvador. EDD:
A Declarative Debugger for Sequential Erlang Programs (2014) [en línea] Informe técnico SIC02/14. Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid.
<http://maude.sip.ucm.es/~adrian/files/tacas14.pdf>

[A10] Eclipse Documentation. The Eclipse Modeling Framework (EMF) Overview (2005) [en línea]
<http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.emf.doc%2Freferences%2Foverview%2FEMF.html>

[A11] S. Maier, Robert & B. Tufillaro, Nicholas. The GNU Plotting Utilities: Programs and functions for vector graphics and data plotting v2.4.1. [en línea].
<http://math.arizona.edu/~rsm/software/info/plotutils.pdf > [consulta: septiembre 2015]

[A12] Spinellis, Diomidis. Drawing UML Diagrams with UMLGraph (2014) [en línea] Sequence Diagrams
<http://www.umlgraph.org/doc/seq-intro.html>

[A13] VOGEL, Lars. Eclipse Target Platform - Tutorial v7.4. (2014) [en línea] vogella GmbH
<http://www.vogella.com/tutorials/EclipseTargetPlatform/article.html>

Libros

[L1] Henrik Kniberg & Mattias Skarin (2010). Kanban y Scrum – obteniendo lo mejor de ambos (1st Edition). – C4Media, editores de InfoQ.com.
<http://www.proyectalis.com/documentos/KanbanVsScrum_Castellano_FINAL-printed.pdf>

[L2] Greg Michaelson (2011). An Introduction to Functional Programming Through Lambda Calculus (1st Edition) – USA: Dover Publications.

[L3] Martin Logan, Eric Merritt & Richard Carlsson (2010). Erlang and OTP in Action (1st Edition). – USA: Manning.

[L4] Jeff McAffer, Jean-Michel Lemieus & Chris Aniszczyk (2010). Eclipse Rich Client Platform (2nd Edition). – USA: Addison-Wesley.

[L5] Robert Harris & Robert Warner (2004). The Definitive Guide to SWT and JFace (1st Edition). – USA: Apress.

[L6] Eric Clayberg & Dan Rubel (2008). Eclipse plug-ins (3rd Edition) – USA: Addison Wesley.

[L7] Dave Steinberg, Frank Budinsky, Marcelo Paternostro & Ed Merks (2003). EMF Eclipse Modeling Framework (2nd Edition). – USA: Addison-Wesley.

[L8] Dan Rubel, Jaime Wren & Eric Clayberg (2011). The Eclipse Graphical Editing Framework (GEF) (1st Edition). – USA: Addison-Wesley.

Tesis

[T1] RIESCO Rodríguez, Adrián. Depuración declarativa y verificación heterogénea en Maude. Tesis (Doctor en Informática). Madrid, España.
Universidad Complutense de Madrid, Facultad de Informática, Depto. de Sistemas Informáticos y Computación, 2011. <http://maude.sip.ucm.es/~adrian/files/thesis.pdf>

[T2] CABALLERO Roldán, Rafael. Técnicas de diagnóstico y depuración declarativa para lenguajes lógico-funcionales. Tesis (Doctor en C.C. Matemáticas). Madrid, España.
Universidad Complutense de Madrid, Facultad de Informática, Depto. de Sistemas Informáticos y Programación, 2004. <http://gpd.sip.ucm.es/rafa/tesis/tesis.pdf>

[T3] DE LA ENCINA Vara, Alberto. Formalizando el proceso de depuración en programación funcional paralela y perezosa. Tesis (Doctor en C.C. Matemáticas). Madrid, España.
Universidad Complutense de Madrid, Facultad de Ciencias Matemáticas, Depto. de Sistemas Informáticos y Computación. <http://www.mathematik.uni-marburg.de/~eden/paper/tesisAlbertoe.pdf>

[T4] URQUIZA Fuentes, Jaime. Generación Semiautomática de Animaciones de Programas Funcionales con Fines Educativos. Tesis (Doctor en C.C. Matemáticas). Madrid, España.
Universidad Rey Juan Carlos, Escuela Superior de Ingeniería Informática, Depto. Departamento de Lenguajes y Sistemas Informáticos. 2007
<https://eciencia.urjc.es/bitstream/handle/10115/3317/Tesis-JaimeUrquizaFuentes.pdf?sequence=1>

Apéndice A - Instalación

Antes de proceder a la instalación E-EDD sobre Eclipse será necesario realizar la instalación de las herramientas de terceros usadas en el proyecto.

Instalación de Erlide

La forma más rápida de instalar Erlide es accediendo a su página web oficial y utilizar el botón *Install* (figura A.1), para proceder a la instalación mediante una acción *drag and drop*, es decir, seleccionar el botón y arrastrarlo hasta la instancia de Eclipse previamente abierta donde lo soltaremos.

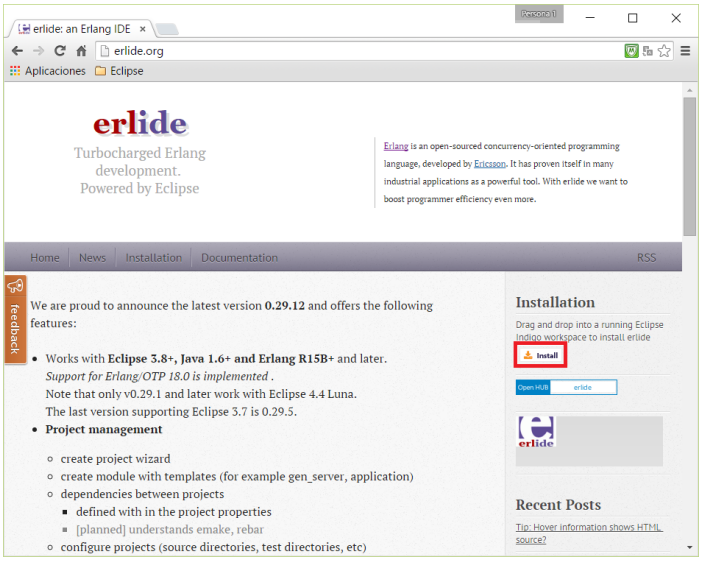

Figura A.1 Botón de instalación Erlide en modo drag and drop.

Dicha acción abrirá el diálogo de instalación de Eclipse (figura A.2), donde seleccionaremos las características a instalar. Confirmamos la instalación y continuamos (ver final del apéndice B).

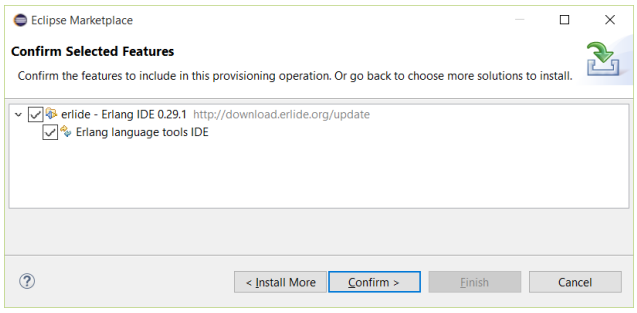

Figura A.2 Sitio de actualizaciones Erlide

Instalación de Graphviz

Graphviz es una herramienta de visualización gráfica *open source*, que será usada para representar información estructurada como diagramas.

Para proceder a su instalación bastará con descargar el software de su página oficial (está disponible para varias plataformas) y descomprimirlo en la ubicación que mejor se considere. En este caso se explicará la forma de hacerlo en Windows utilizando el archivo archivo *.zip (se puede optar por descargar el archivo instalable *.msi si se desea). Una vez descargado sólo restará descomprimirlo y añadir la ruta a las variables de entorno del sistema como se muestra en la figura A.3.

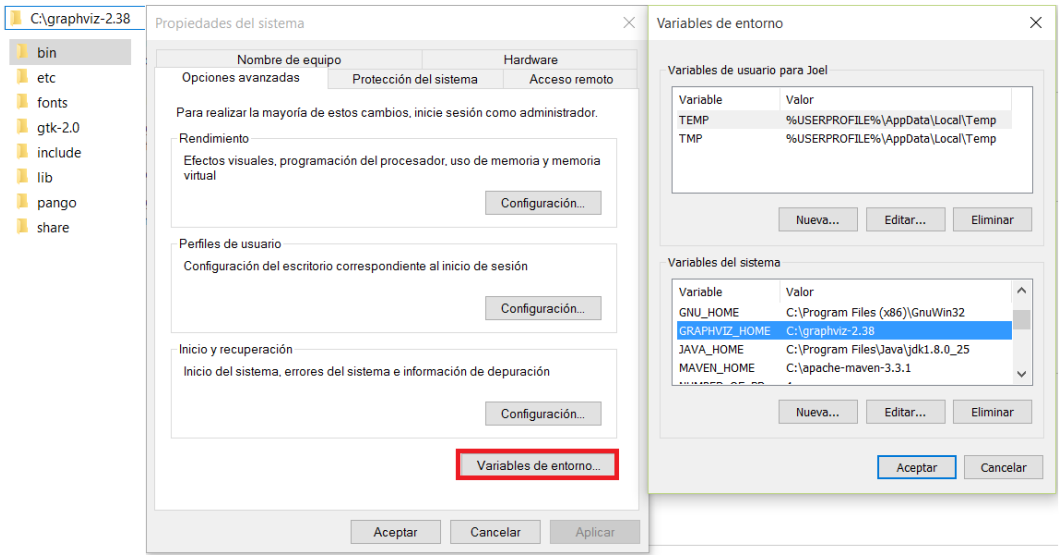

Figura A.3 Configuración de la variable de entorno Graphviz

Se deberá editar también la variable "path" para incluir la variable recién creada anteriormente, tal y como se muestra en la figura A.4.

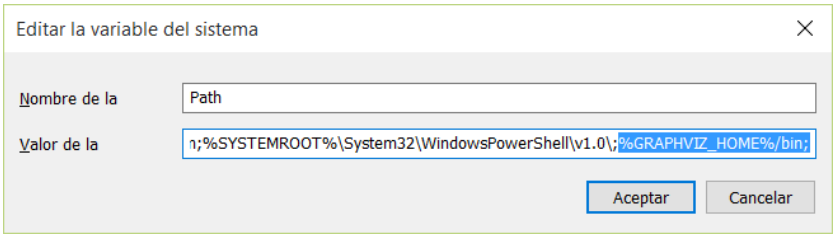

Figura A.4 Utilización de la variable de entorno en la variable de sistema

Para verificar que está funcionado desde línea de comandos ejecutaremos la instrucción "`dot -v`" para verificar la versión utilizada en dicho comando (figura A.5).

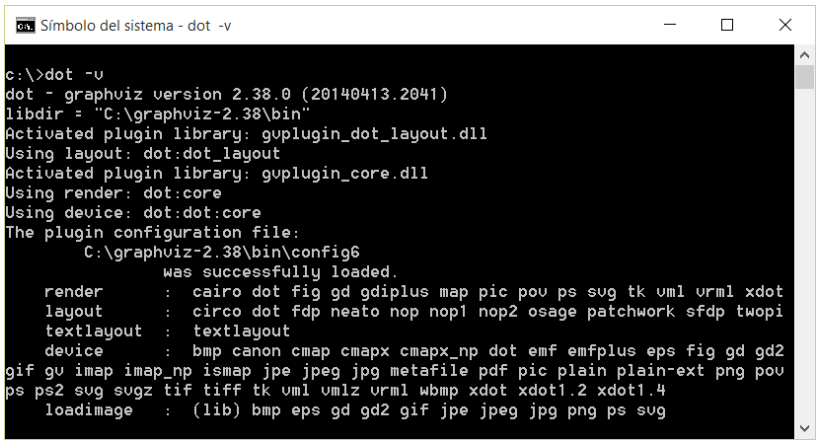

Figura A.5 Validación de la instalación Graphviz

Instalación de UMLGraph

UMLGraph es una herramienta que, mediante una especificación declarativa, permite construir diagramas de clase y diagramas de secuencia bajo el estándar UML. Para su instalación, al igual que el caso anterior, bastará con descargar el *.zip desde la página oficial del proyecto y proceder a la declaración de la variable de entorno (figura A.6).

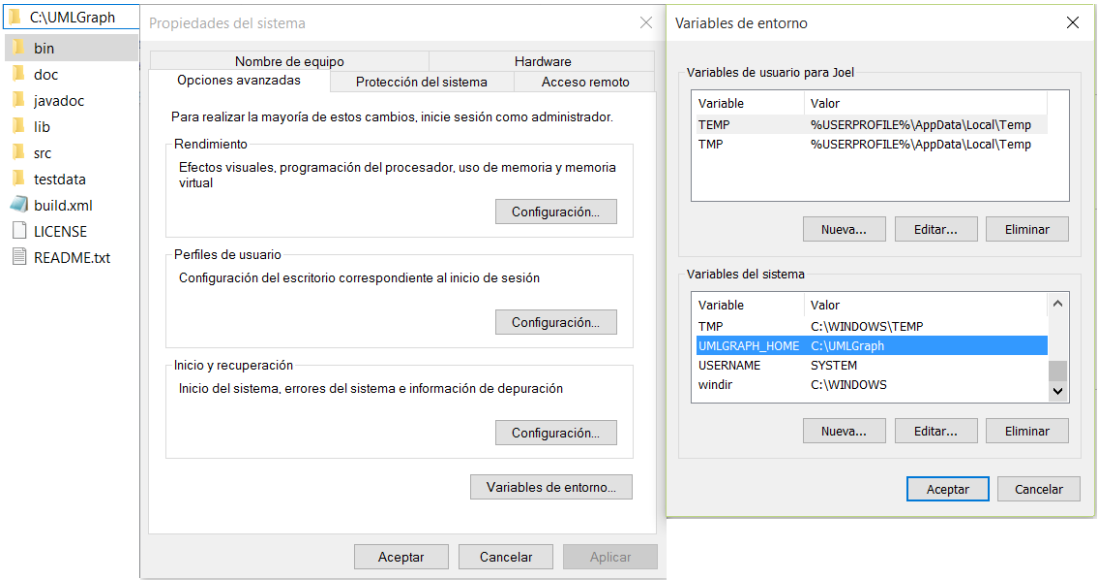

Figura A.6 Creación de la variable de entorno UMLGraph

Nuevamente habrá que editar la variable “path” para referenciarlo (figura A.7).

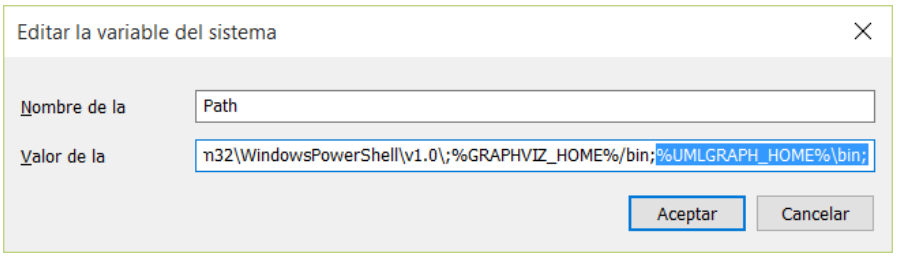

Figura A.7 Configuración de la variable de entorno

Para verficar que está correctamente instalado se ejecuta en línea de comandos la instrucción “`pic2plot --v`”, para verificar la versión de la herramienta (figura A.8).

Figura A.8 Validación de la instalación UMLGraph

Instalación de los plugins de E-EDD sobre Eclipse

Lo primero será descargar el IDE de Eclipse de su página oficial; se recomienda la versión Eclipse Modeling Tools [33] bajo la distribución Luna (ver figura A.9), dado que la propia versión ya cuenta con los plugins para trabajar bajo EMF y GMF. En caso contrario, deberán ser instalados por separado.

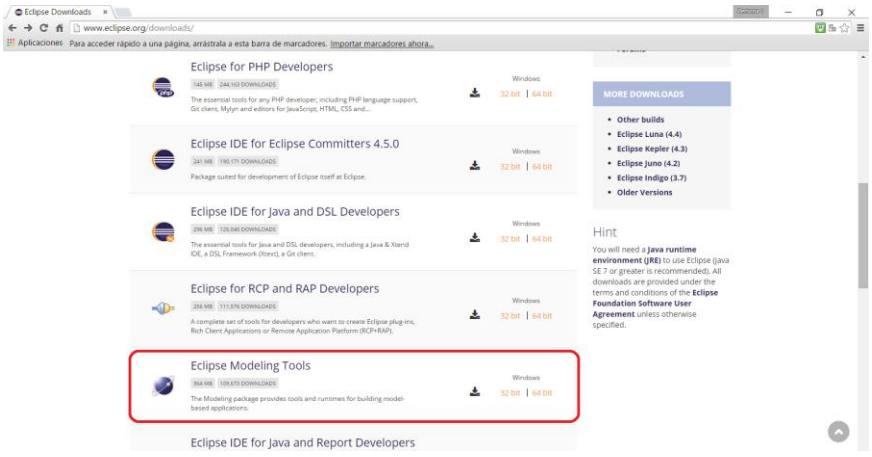

Figura A.9 Descarga de Eclipse Modeling Tools

Una vez descargado Eclipse, se deberá ejecutar para proceder a la instalación de los plugins, los cuales se instalarán a través del comando Help/Install New Software… de la barra de herramientas de Eclipse, donde se añadirá el repositorio del sitio de actualizaciones de E-EDD (figura A.10).

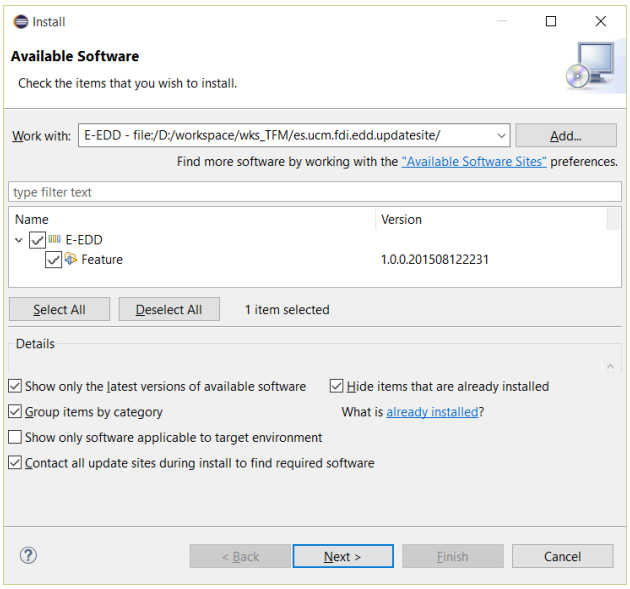

Figura A.10 Sitio de actualizaciones E-EDD

Apéndice B - Introducción al desarrollo de plugins

El presente apéndice es una introducción al desarrollo de plugins para Eclipse. Como ya se mencionó en la sección 5.2, el objetivo de la memoria no es solo explicar cada una de las contribuciones realizadas al proyecto, además se pretende dar una visión general de cómo hacerlo. Para ello, el presente anexo, a modo de *"hola mundo PDE"*, explica los pasos necesarios para crear, configurar e implementar un punto de extensión; el ejemplo en cuestión consiste en contribuir con un menú contextual basado en la selección múltiple.

Lo primero será crear un proyecto de tal naturaleza y para ello se va a utilizar el asistente para la creación de un proyecto de plugin, siguiendo los pasos que se describen a continuación:

1. Creación del proyecto: *File/New/Project/Plug-in Project* (figura B.1).

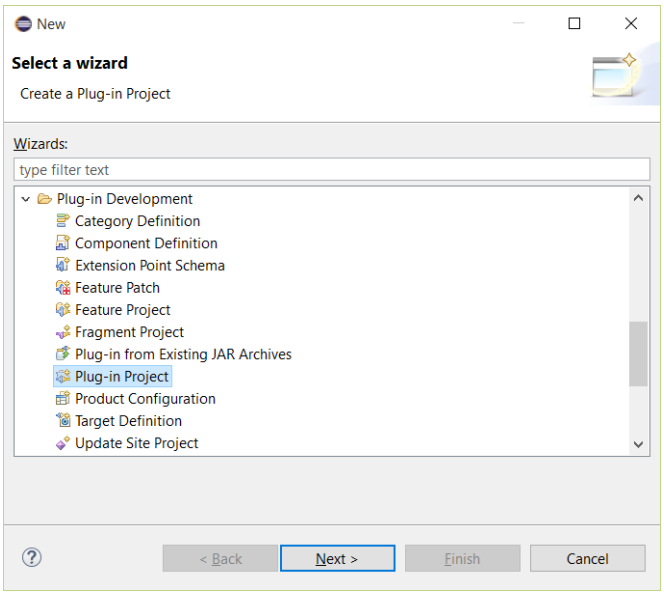

Figura B.1 Creación de un proyecto de plugin

2. Configuración: en la ventana que aparece a continuación en la figura B.2 se pondrá el nombre del proyecto, la ubicación, los valores del proyecto Java y la plataforma de salida.

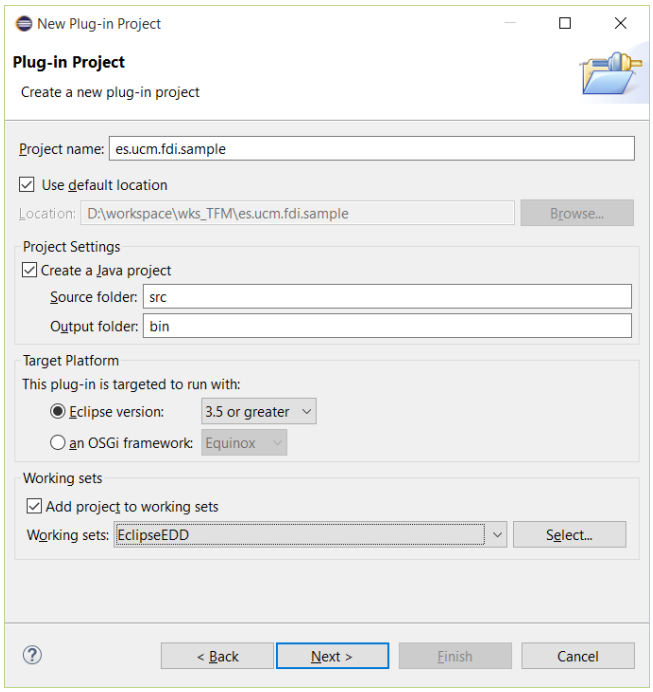

Figura B.2 Configuración del proyecto de plugin

3. Propiedades: esta ventana incluye las propiedades del plugin (figura B.3).

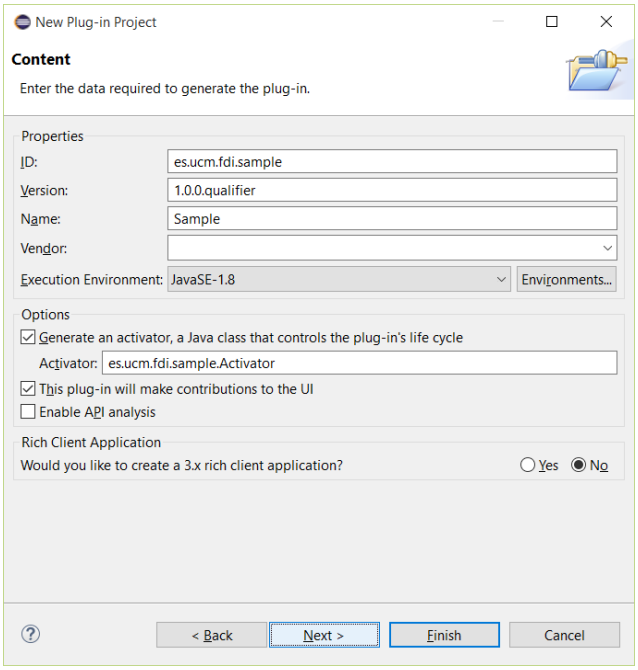

Figura B.3 Propiedades del proyecto de plugin

4. Plantillas: Eclipse ofrece plantillas de ejemplo predefinidas que nos servirán de guía durante la etapa de aprendizaje para el desarrollo de plugins. Los usuarios avanzados puede omitir dicho paso e incluir las extensiones más adelante, pero en esta documentación se utilizará el ejemplo el uso de un menú emergente como primera toma de contacto al desarrollo PDE (figura B.4).

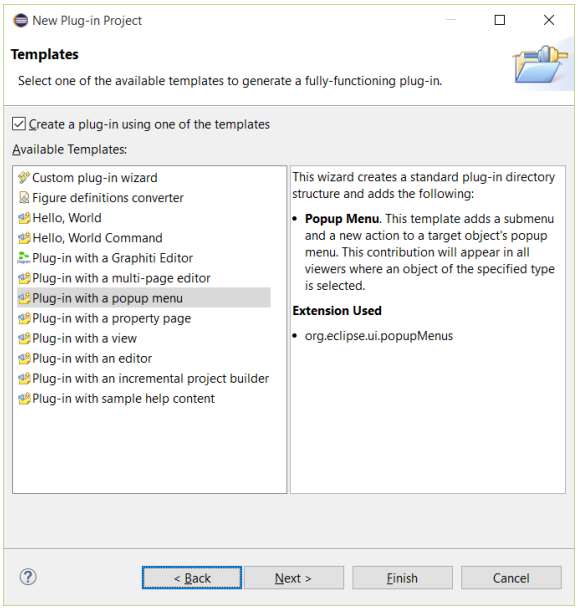

Figura B.4 Plantillas existentes para la implementación de puntos de extensión

5. Configuración del punto de extensión: se deben rellenar los valores referentes a cómo deseamos realizar nuestra contribución (figura B.5):

- **Clase del objeto de destino**: se indica el tipo de clase sobre la cual se va a detonar la acción, es decir, el objeto donde se creará el menú contextual. En nuestro caso será aplicable para cualquier objeto del tipo IFile (archivos existentes en el espacio de trabajo).
- **Nombre del submenú**: nombre de nuestro submenú contextual.
- **Etiqueta de acción**: es el nombre que aparecerá en el comando.
- **Nombre del paquete Java**: nombre del paquete donde se va a localizar la clase.
- **Clase de acción**: es el nombre de la clase que implementará la acción del menú contextual cuando se haga clic.

Es necesario recordar que una acción se puede aplicar a una selección múltiple, es decir, para varios elementos a la vez, o de forma simple procesando solo el primer elemento seleccionado.

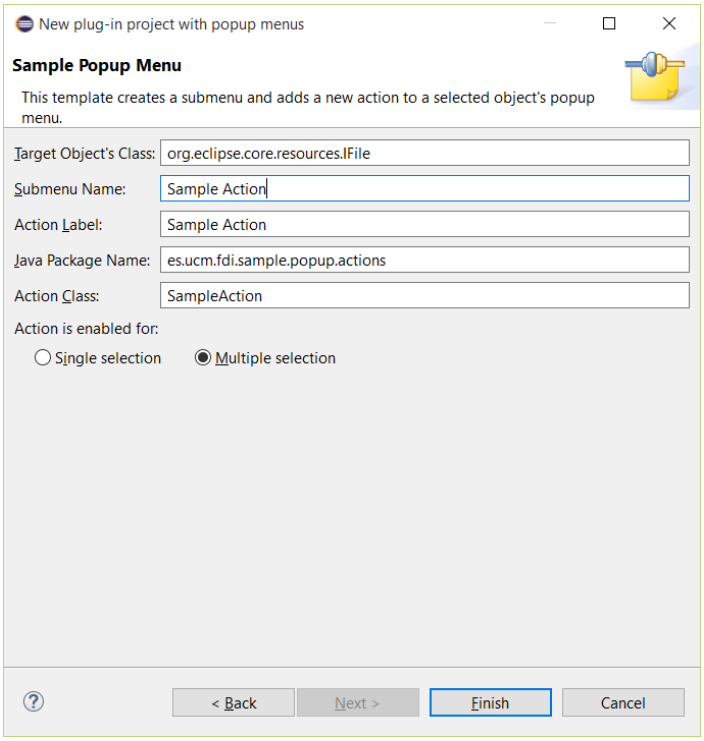

Figura B.5 Configuración de las propiedades para el proyecto de ejemplo

6. Finalizar: una vez que se ha hecho clic en finalizar se abre la perspectiva de "Desarrollo de plugins PDE" y se genera la estructura del proyecto como se muestra en la figura B.6.

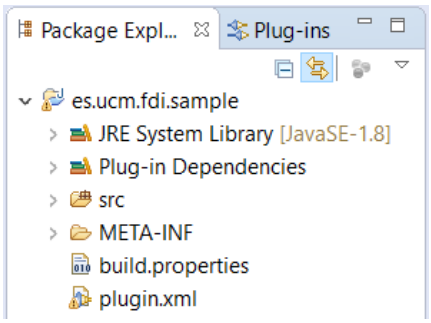

Figura B.6 Estructura del proyecto de plugin

Para modificar el contenido del proyecto bastará con abrir el archivo *`plugin.xml`*, que es el que orquesta la definición del plugin; toda aportación deberá estar declarada y definida en este archivo. El editor del archivo `plugin.xml` contiene una serie de pestañas de propósitos específicos para configurar el proyecto y sus contribuciones. A continuación se explica de forma rápida cada una de ellas como primera toma de contacto e introducción al desarrollo PDE.

- **Visión general**

Esta pestaña, como su nombre indica, representa la visión generalizada de la estructura de un proyecto de plugin. Desde aquí se pueden cambiar las propiedades del mismo, acceder a los distintos apartados por medio de los enlaces definidos, configurar el entorno JDK utilizado, etc. (figura B.7).

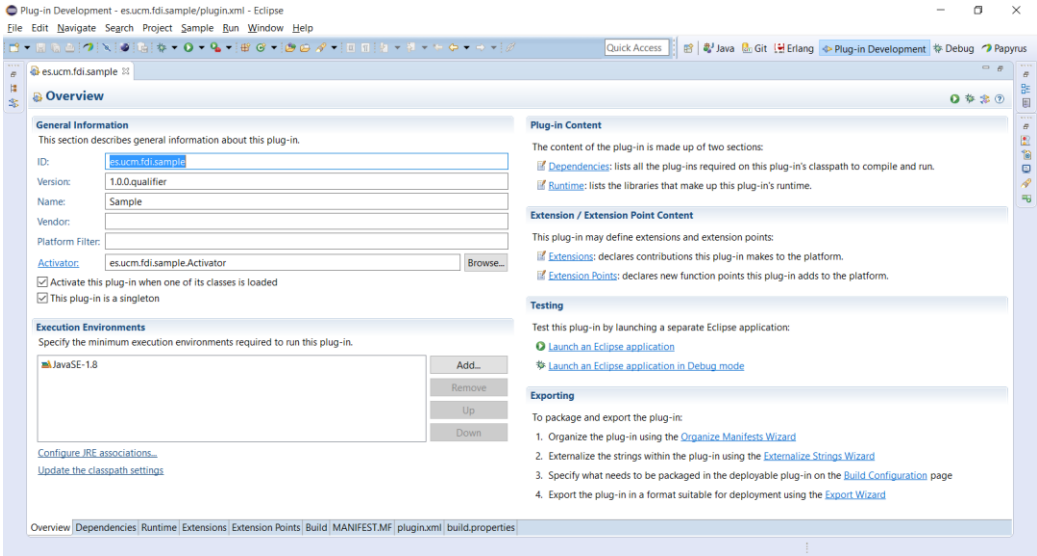

Figura B.7 Pestaña visión general del plugin

- **Dependencias**

En esta pestaña se configuran las dependencias entre plugins, las cuales pueden ser tanto a nivel plugin como a nivel paquete. También se ofrecen herramientas útiles para la gestión de las mismas, por ejemplo el análisis de dependencias (figura B.8).

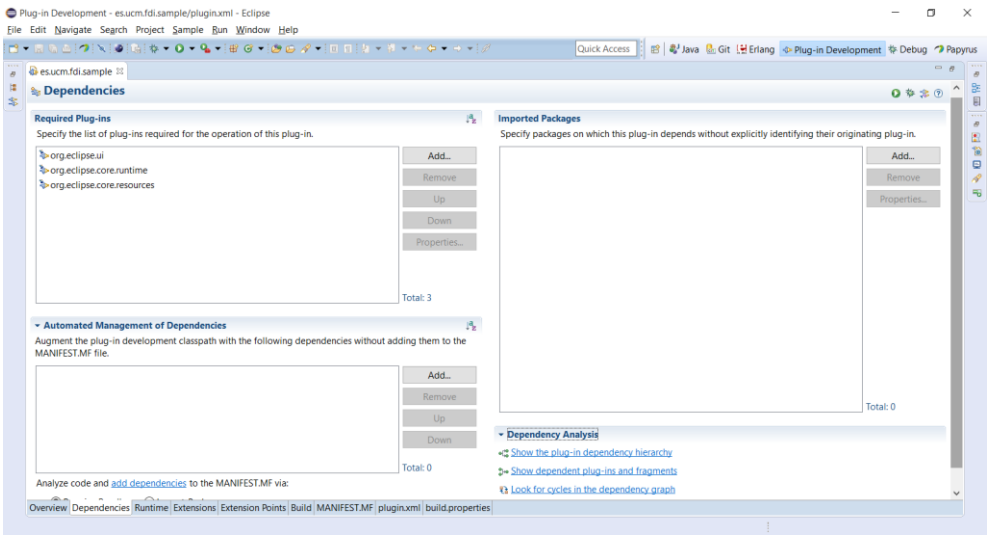

Figura B.8 Pestaña de dependencias

- **Tiempo de ejecución**

Esta pestaña enumera los paquetes que serán visibles por otros plugins, es decir, da visibilidad a aquellas clases que expondremos para que otros plugins las puedan utilizar. Dicha visibilidad puede parametrizarse editando sus opciones. De la misma forma, en el apartado *classpath* se pueden añadir bibliotecas (archivos jar) cuando el plugin las necesite (figura B.9).

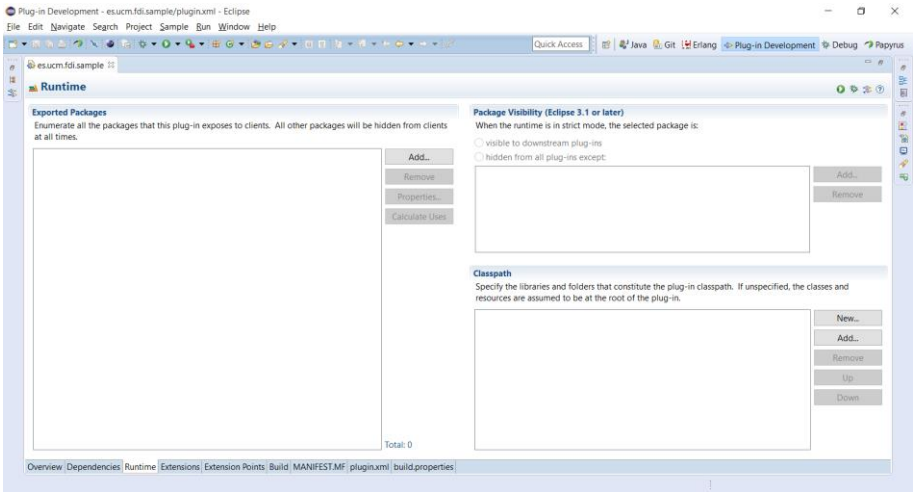

Figura B.9 Pestaña de tiempo de ejecución

- **Extensiones**

En esta pestaña es donde radica el mecanismo extensible de Eclipse; es aquí donde se añaden las funcionalidades que serán incluidas en el IDE, de forma que permite enriquecer el entorno y adaptarlo a necesidades específicas. En nuestro caso vamos a contribuir a Eclipse añadiendo un menú emergente adicional a los que ya tiene predefinidos (figura B.10).

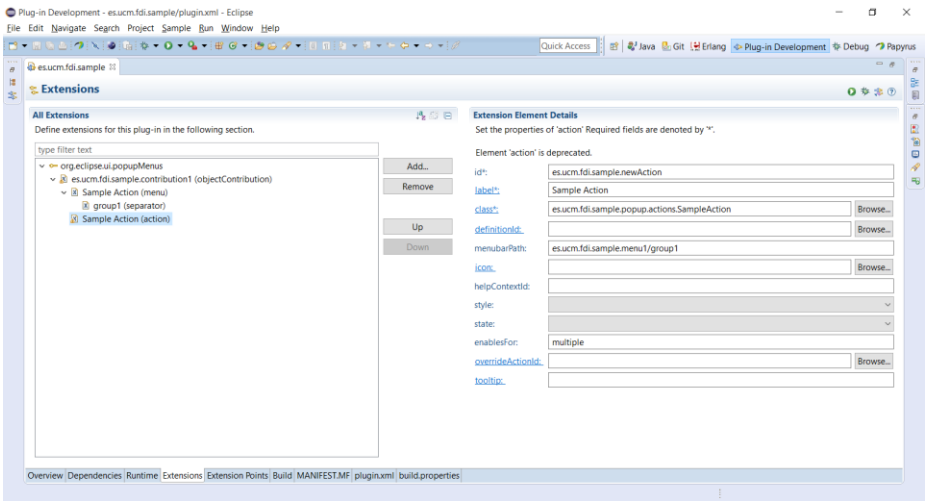

Figura B.10 Pestaña de extensiones

- **Puntos de extensión**

En el caso anterior éramos nosotros quienes extendíamos funcionalidades existentes, por tanto la contraparte es permitir que otro extienda nuestras contribuciones. Es necesario declarar los accesos desde donde permitiremos dicha ampliación; mediante la definición de puntos de extensión consentimos a otros para que aumenten la funcionalidad ya implementada. En este ejemplo no se define ninguno pero es una posibilidad interesante para otras aplicaciones (figura B.11).

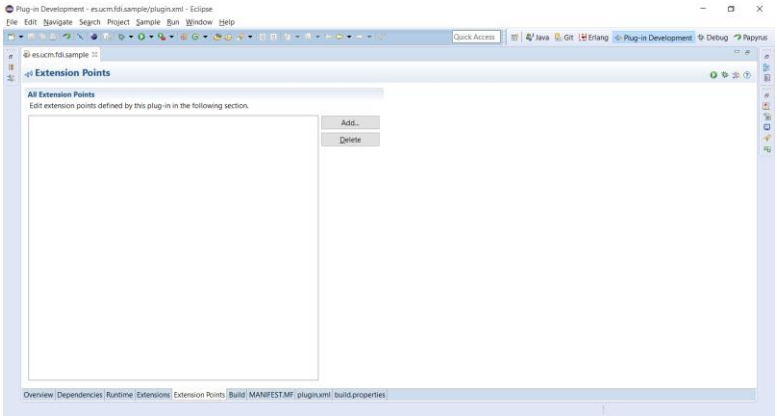

Figura B.11 Pestaña de puntos de extensión

- **Construcción**

En esta pestaña se definen los archivos que serán empaquetados durante la etapa de construcción. Lo normal es distribuir los binarios pero se podría incluir el código fuente si se desea. Es importante seleccionar el uso de imágenes y archivos de configuración puesto que, si el plugin los utiliza, deberán estar contemplados (figura B.12).

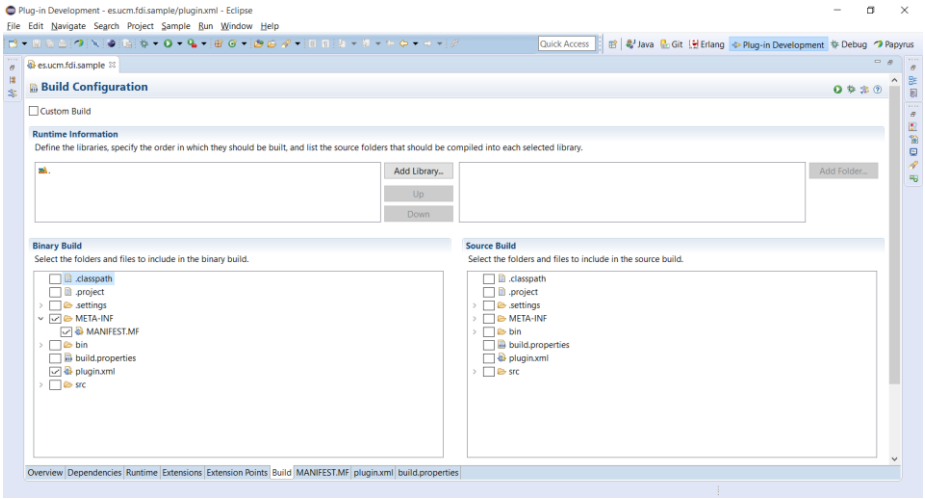

Figura B.12 Pestaña de construcción

- **MANIFEST.MF**

En la figura B.13 se puede observar el archivo de manifiesto resultante, necesario para que exista comunicación con otros plugins. Este archivo se modifica desde las pestañas previas pero no impide su edición manual, si se desea.

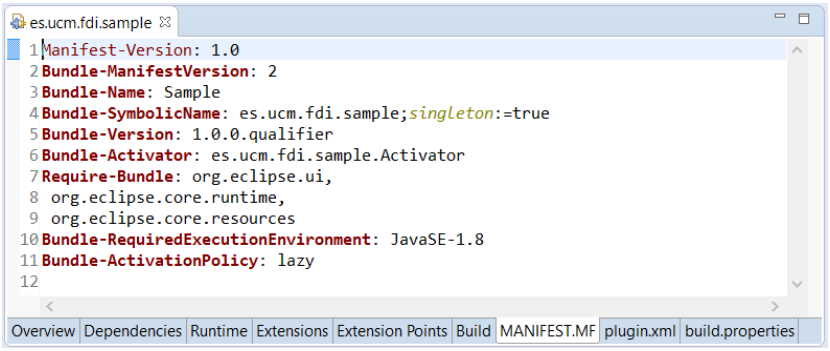

Figura B.13 Editor del archivo de manifiesto

- **Plugin.xml**

La figura B.14 representa el documento fuente de la definición de un plugin.xml, desde aquí puede también modificar manualmente su contenido (obtenido de la edición de la pestaña de extensiones).

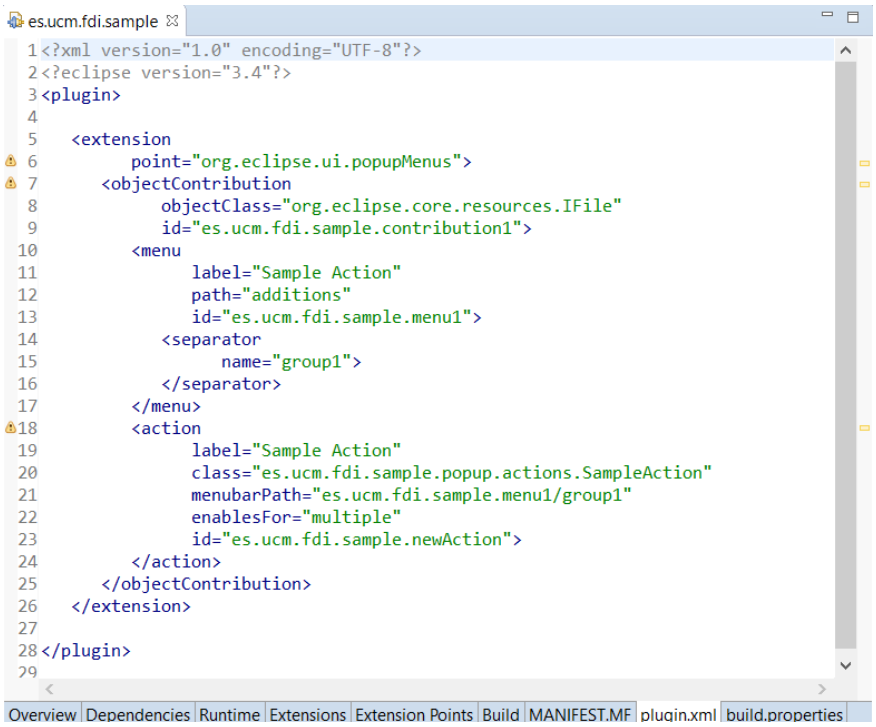

Figura B.14 Editor del archivo plugin.xml

- **Build.properties**

Contiene la definición usada de los archivos que serán empaquetados durante la etapa de construcción. Es la representación del archivo fuente de la pestaña *build* (figura B.15).

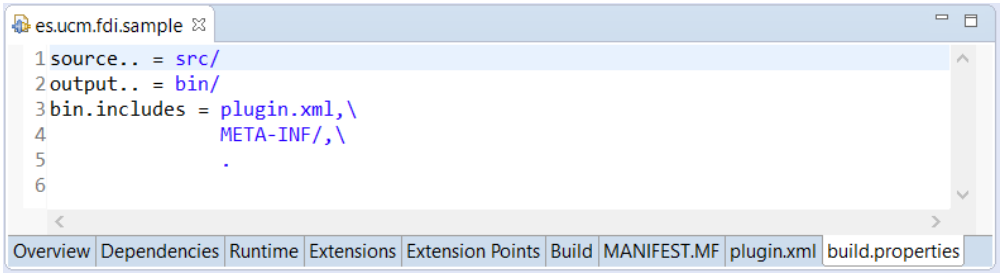

Figura B.15 Editor del archivo de propiedades de construcción

Ejecutar el plugin

Para probar la contribución de los plugins que estamos desarrollando se deben usar las opciones habilitadas para ello mostradas en la figura B.16, es decir, siendo ejecutado el desarrollo como una "Aplicación Eclipse"; de esta forma se abrirá otra instancia de Eclipse que funcionará como una entidad huésped de la instancia principal, permitiéndonos ver las contribuciones realizadas durante el desarrollo.

Para ello iremos a la vista previa para activar la ejecución de depuración según se desee.

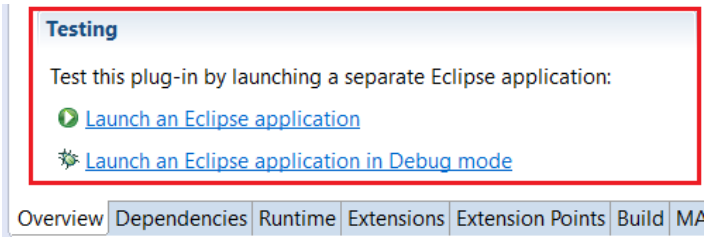

Figura B.16 Ejecución de la instancia Eclipse para pruebas con la aplicación

Si es la primera ocasión que se ejecuta puede tardar un par de segundos más en iniciarse, dado que deberá crear la estructura de directorios para manejar dicha instancia huésped.

Como forma alternativa, se puede optar por parametrizar la ejecución de la instancia, para ello habrá que ir a Run/Run configurations… y establecer los valores que mejor se adapten a las necesidades del usuario (figura B.17).

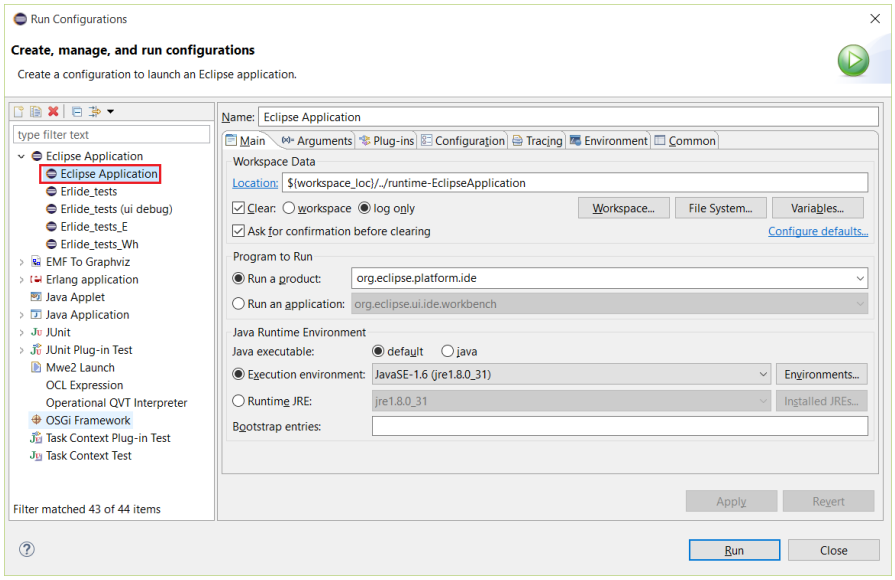

Figura B.17 Configuración del archivo de lanzamiento para la aplicación Eclipse

Finalmente, resta comprobar la contribución realizada sobre la instancia huésped, ejecutando la nueva acción. La figura B.18 detalla el uso del menú recién creado.

Recordemos que creamos un submenú contextual llamado “Sample Action”, que tiene un comando también llamado “Sample Action” de forma que, al hacer una selección múltiple sobre los recursos, deberá aparecer habilitado.

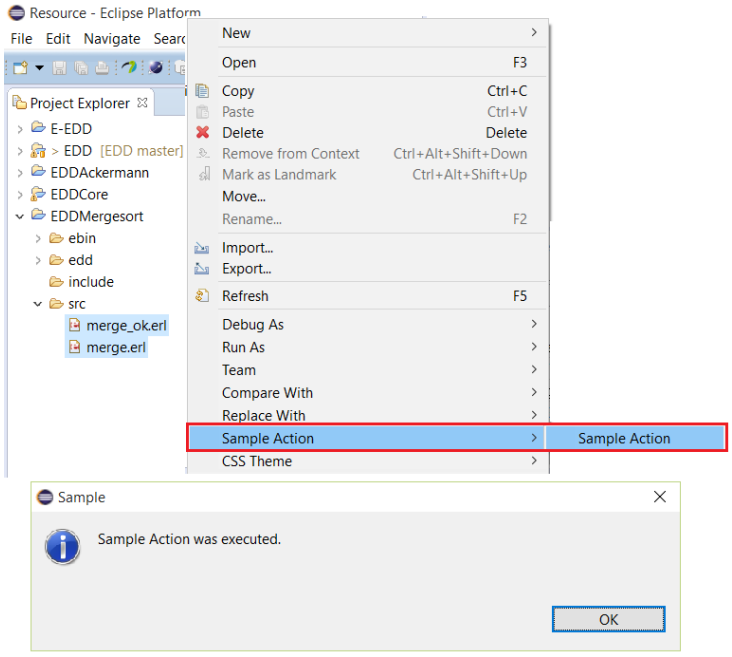

Figura B.18 Ejecución del comando creado en el tutorial

Empaquetado del plugin

Para empaquetar y distribuir plugins hay que crear dos nuevos proyectos: un proyecto de característica y un proyecto de sitio de actualizaciones. El primero describe los plugins que lo conforman y el segundo, el acceso para proceder a la instalación que depende del primero para listar lo que se quiere instalar.

- **Proyecto de característica**

1. Para crear un proyecto de característica se selecciona File/New/Project… y se elige "Feature Project" en la categoría de desarrollo de plugins (figura B.19).

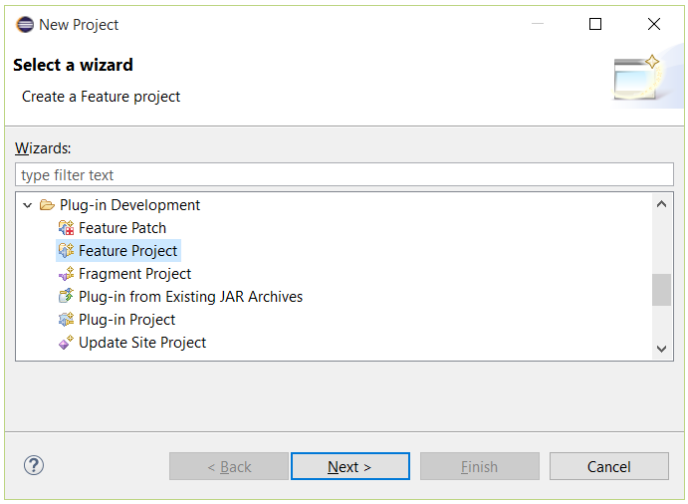

Figura B.19 Creación de un proyecto de características

2. Propiedades: se debe completar el diálogo Propiedades de característica con el nombre de proyecto (figura B.20).

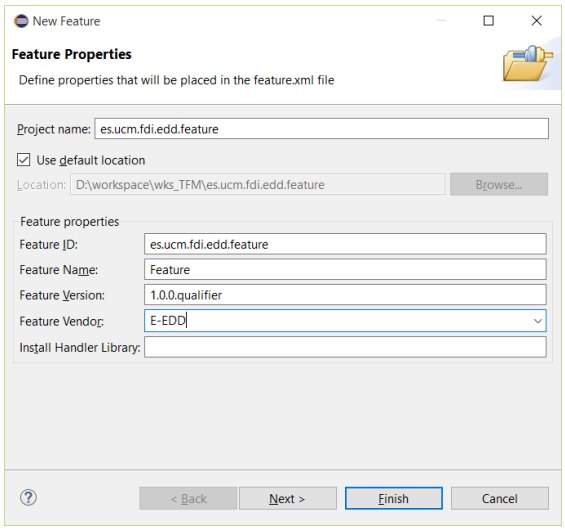

Figura B.20 Propiedades del proyecto de características

3. Plugins y fragmentos referenciados: representan a todos aquellos plugins que serán administrados por la característica en cuestión (figura B.21).

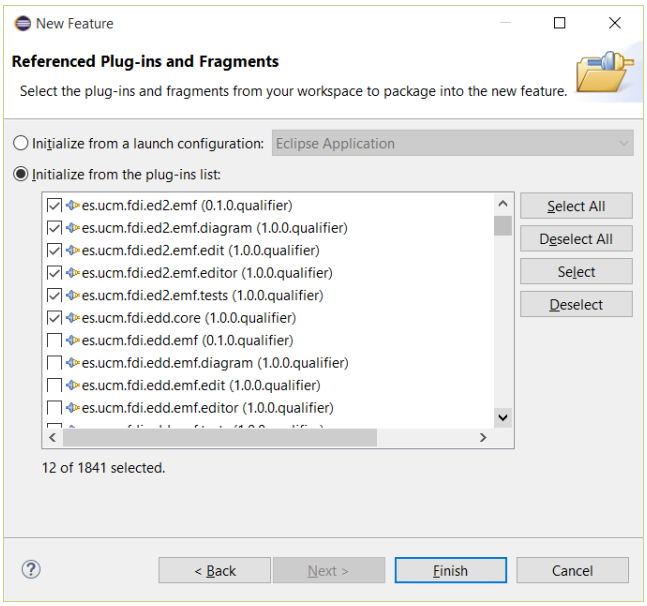

Figura B.21 Lista de proyectos referenciados a empaquetar por la característica

4. Finalizar: para terminar y crear la estructura del proyecto.

De forma similar a lo anterior respecto al proyecto de plugin, veremos las pestañas que corresponden en este caso al proyecto de característica; el editor de características abre por defecto la página Visión General.

Añadir descripción y acuerdo de licencia a la característica

Para añadir la descripción de característica y el acuerdo de licencia, se selecciona la pestaña Información tal y como se muestra en la figura B.22; aquí se deberán cumplimentar las sub-pestañas Descripción de la característica, Aviso de copyright, Acuerdo de licencia y Sitios para visitar según los términos y condiciones a los que se sujete el desarrollo y se deseen aplicar a la característica en cuestión.

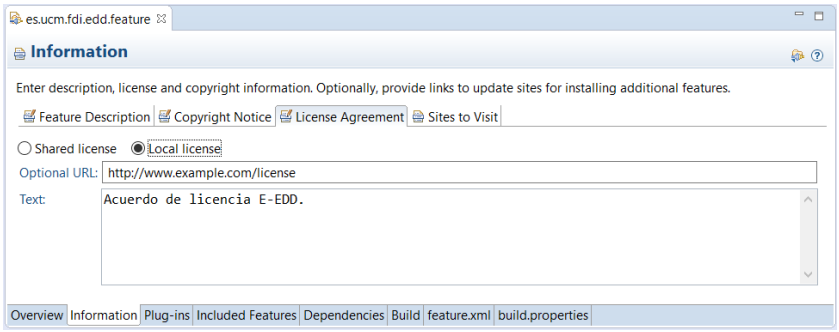

Figura B.22 Editor del archivo de característica para su configuración

- **Proyecto de sitio de actualizaciones**

Para crear el sitio de actualizaciones seleccionamos File/New/Project… y elegimos Update site project en la categoría desarrollo de plugins (figura B.23).

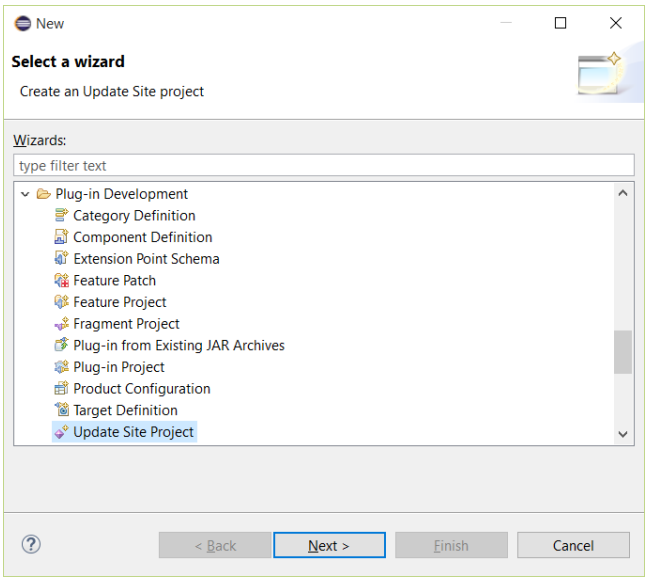

Figura B.23 Creación del sitio de actualizaciones

Propiedades: se debe introducir el nombre del sitio en el recuadro de texto nombre de proyecto y seleccionar la ubicación donde se alojará (figura B.24).

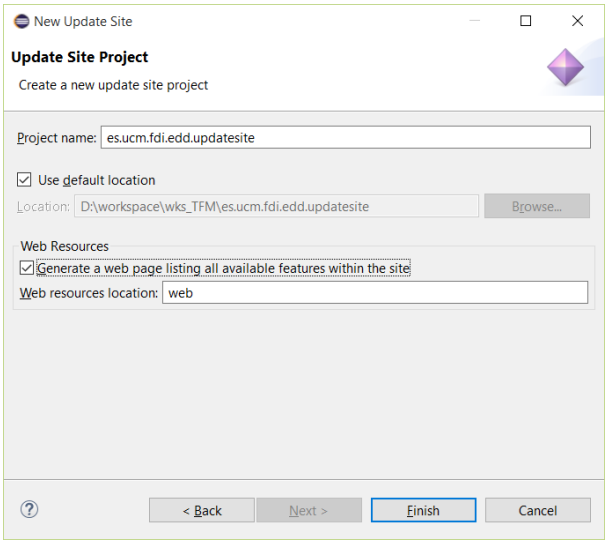

Figura B.24 Propiedades del sitio de actualizaciones

Crear la descripción del sitio de actualizaciones

El editor de sitios abrirá por defecto la página Mapa del Sitio de Actualizaciones descrito en la figura B.25; aquí podremos categorizar los plugins en subconjuntos para darle más sentido al proyecto permitiendo que el usuario pueda tener una idea de lo que va a instalar.

Para añadir una característica a una categoría, seleccionamos la categoría en cuestión y pulsamos el botón añadir característica…, donde deberemos escoger la característica deseada.

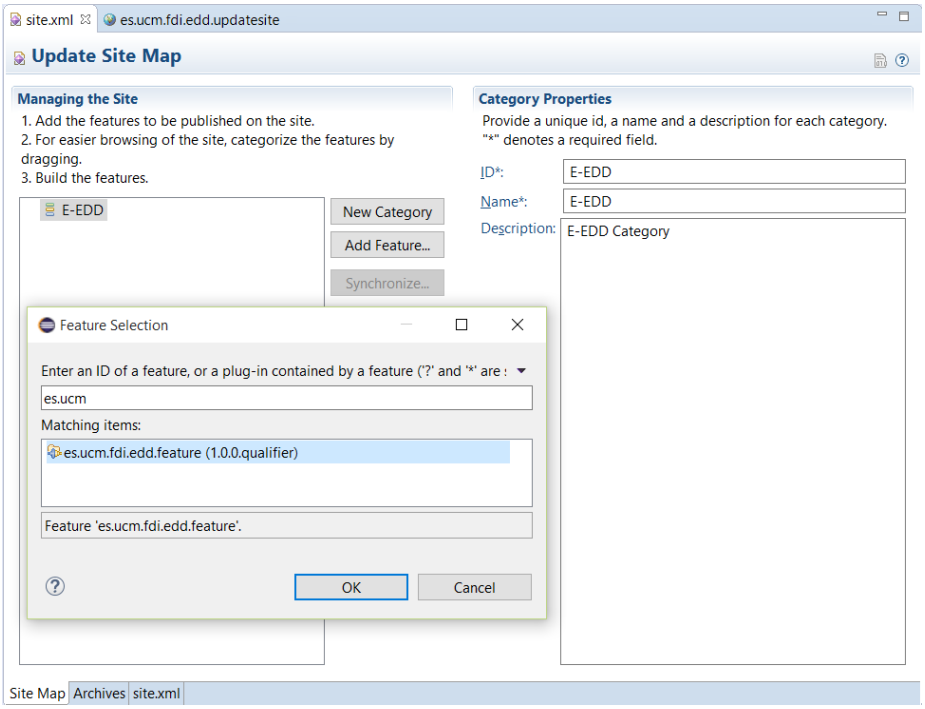

Figura B.25 Categoría del sitio de actualizaciones

El siguiente paso es definir los plugins que componen la característica y seleccionar la pestaña Archivadores, donde introduciremos la URL y la descripción del sitio de actualizaciones (figura B.26).

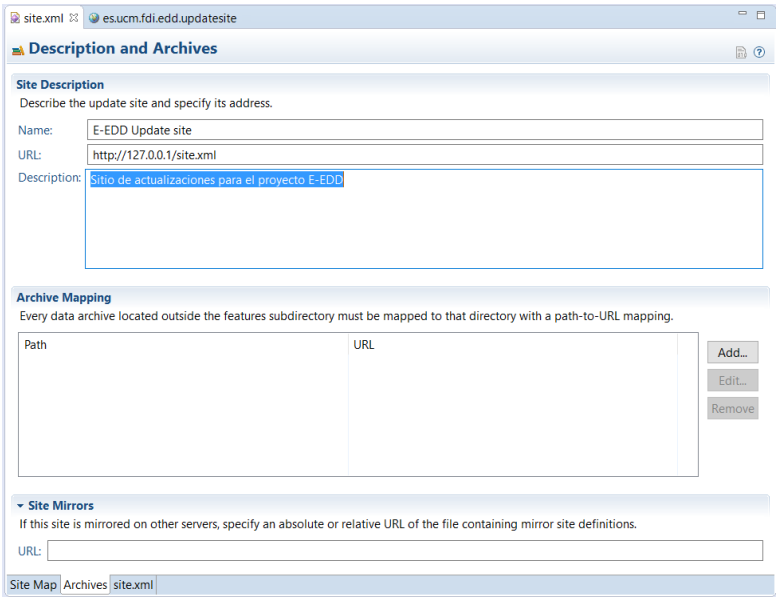

Figura B.26 Editor del sitio de actualizaciones

Construir y exportar características del sitio de actualizaciones

Para construir la jerarquía del sitio de actualizaciones para su exportación se hará presionando el botón Construir o, en su defecto, Construir todo, en caso de tener más de una categoría.

Finalmente, sólo se debe verificar que los archivos *.jar se hayan creado en las carpetas de plugins y feautures, respectivamente, en el sitio de actualizaciones (figura B.27). Dicha estructura es estándar para realizar instalaciones en Eclipse, por tanto bastará consultar dicho directorio comprobando que existen los plugins que conforman la característica.

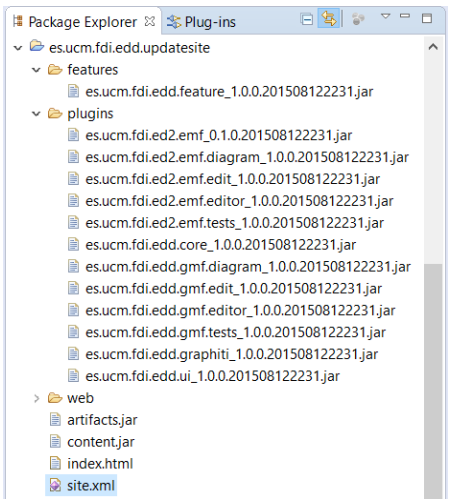

Figura B.27 Estructura del sitio de actualizaciones

- **Instalar y desinstalar una característica**

Para instalar un proyecto en Eclipse desde un sitio de actualizaciones, se debe ir a Help / Install New Software… para lanzar el asistente de instalación.

Mediante el botón Add podremos añadir la ubicación desde donde queremos proceder con la instalación, de forma similar a como se muestra en la figura B.28.

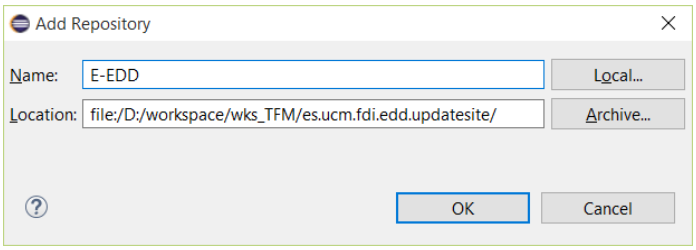

Figura B.28 Definición de un nuevo repositorio

Una vez añadido el sitio podremos seleccionar las características a instalar (figura B.29). Recordemos que el proyecto es pequeño y tiene sólo una característica, pero podría darse el caso de tener componentes adicionales en los que el usuario puede determinar si desea instalarlos o no.

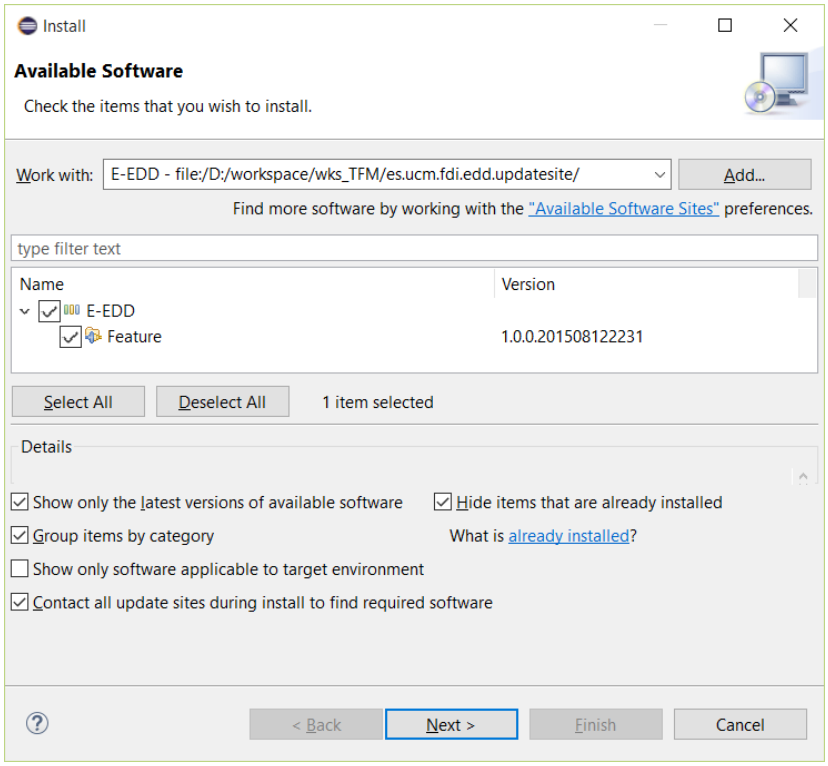

Figura B.29 Instalación desde el sitio de actualizaciones recién creado

Una vez seleccionado el conjunto de plugins a instalar se realizará un cálculo de dependencias donde se verificará que se cumplen todos los requisitos para proceder a la instalación. De no ser el caso, el resultado dará errores, solicitando instalar aquellas dependencias que resulten necesarias para poder continuar. En caso contrario, si la instalación ha ido bien, solo pedirá aceptar las licencias definidas en las características para continuar con la instalación.

Por último se pedirá autorización para reiniciar Eclipse para que pueda ser actualizado con el software recientemente instalado. Es importante dejar que reinicie siempre ya que, de no ser así, se advierte que podría tener problemas con el registro de los nuevos plugins, lo que podría manifestarse como resultados no esperados en el próximo arranque de Eclipse.

- **Verificar la instalación de la característica**

Para verificar la instalación de alguna característica recientemente instalada bastará con ejecutar el comando Help/Instalation details… desde el menú de herramientas de Eclipse.

Se abrirá el diálogo mostrado en la figura B.30 con la información de la instalación y allí habrá que buscar la característica que hemos instalado recientemente; por ejemplo, verificar que esté instalado el proyecto Erlide.

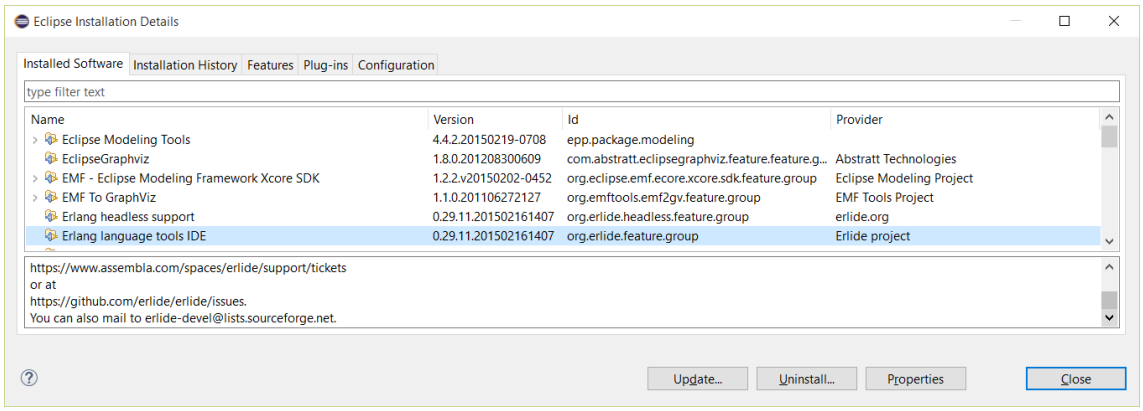

Figura B.30 Detalles de la instalación

- **Desinstalar característica**

Para desinstalar la característica ejecutamos nuevamente el comando Help->Installation Details; aquí seleccionaremos la característica a desinstalar presionaremos el botón Uninstall…, tal y como se muestra en la figura B.31.

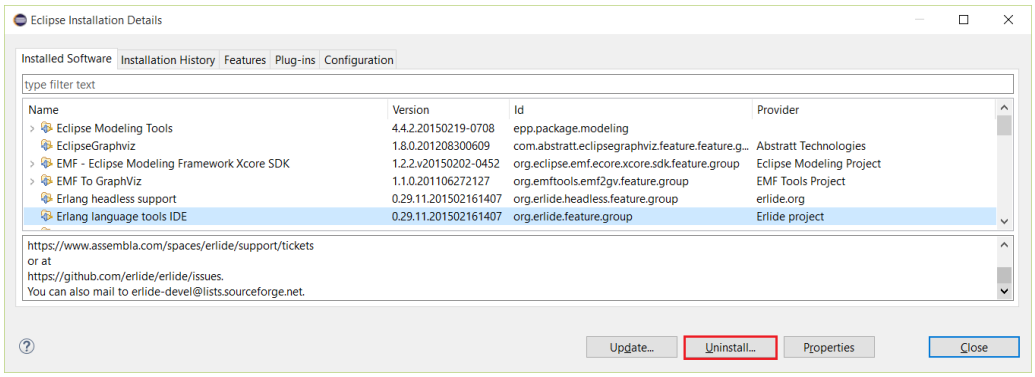

Figura B.31 Desinstalación de una característica previamente instalada

Se abrirá un diálogo donde se describen las características que se pretenden eliminar. El usuario debe confirmar si desea continuar con la desinstalación o no; en caso afirmativo, bastará con presionar el botón finalizar y esperar a que termine y reinicie Eclipse, similar al proceso de instalación (figura B.32).

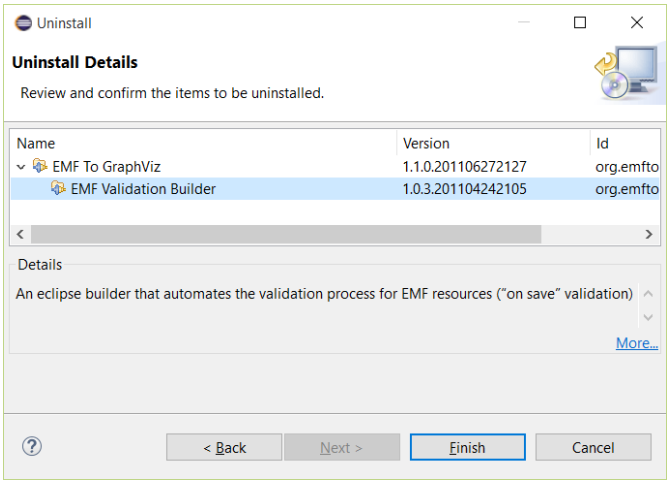

Figura B.32 Confirmación de la desinstalación

Existen aún más conceptos fundamentales en el desarrollo de plugins, pero lo que aquí se describe se limita a las nociones mínimas que se deberían tener para empezar en el mundo del desarrollo de plugins para Eclipse.

Terminamos este apartado mencionando que el uso de *targets* para Eclipse [A13] es otra pieza fundamental que resulta idónea en el desarrollo PDE, ya que permite definir en un archivo las versiones exactas de los plugins usados, facilitando así partir de la misma configuración inicial en el fin de evitar problemas de dependencias debido al uso de instalaciones distintas entre desarrolladores.

Apéndice C - Creación del modelo EMF y GMF

El presente apéndice tiene como propósito describir las nociones básicas para crear las herramientas de modelo específicas de dominio utilizadas por E-EDD, las cuáles han sido creadas utilizando Eclipse Modeling Framework (EMF) y Graphical Modeling Framework (GMF), ambas proporcionadas para la plataforma Eclipse.

Lo que aquí se expone es la forma de proceder para la creación de editores gráficos, tal y como se indicó en la sección 4 y 7, usados para representar los árboles de ejecución, tanto de forma jerárquica como gráfica. En la sección 5 se hizo una pequeña introducción de cómo están distribuidos los proyectos que conforman los editores gráficos. En este apéndice, por tanto, se da la visión técnica del procedimiento para su construcción, un procedimiento que consta de 6 pasos bien definidos en la metodología (1 y 2 para EMF y 4, 5 y 6 para GMF). Los procedimientos que lo componen son:

1. Definición del metamodelo (ecore).
2. El modelo de generación (genmodel).
3. El modelo de definición gráfica (gmfgraph).
4. El modelo de definición de las herramientas gráficas (gmftool).
5. El mapeo entre la definición gráfica y las herramientas gráficas (gmfmap).
6. El modelo de generación del editor gráfico (gmfgen).

La figura C.1 muestra la vista GMF Dashboard, que es en la que nos basaremos para la creación de los modelos listados anteriormente para construir las herramientas de modelado.

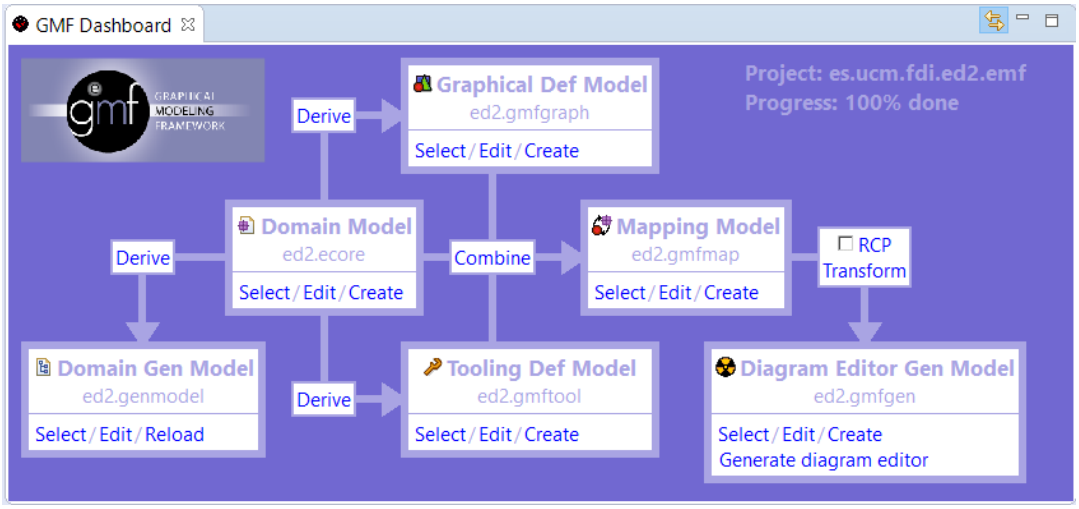

Figura C.1 Vista GMF Dashboard

A continuación se explica brevemente como proceder en cada fase para la construcción integral del editor gráfico.

1. Definición del modelo de dominio

El modelo de dominio (*.ecore) es la definición de un metamodelo, es decir, la definición de un modelo de modelos que podría ser expresado como una plantilla que nos servirá para la creación de modelos basados en ese ámbito descrito.

Para tal propósito se hace uso de EMF, que no es otra cosa que un *framework* de modelado estructurado, especialmente útil para construir herramientas y aplicaciones que se basan en modelos de datos estructurados. Soporta y maneja documentos XMI con un esquema XML Ecore que es el que define la especificación del dominio.

Antes de nada, habremos de crear el proyecto que será el que contenga todos los modelos. Para ello se dispone de dos opciones:

- Creando un proyecto EMF vacío, File->New->Other (figura C.2).
- Creando un proyecto GMF vacío, File->New->Other (figura C.3).

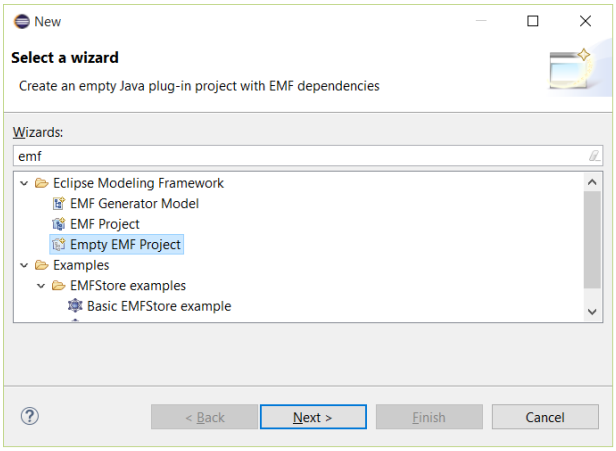

Figura C.2 Creación de un proyecto EMF

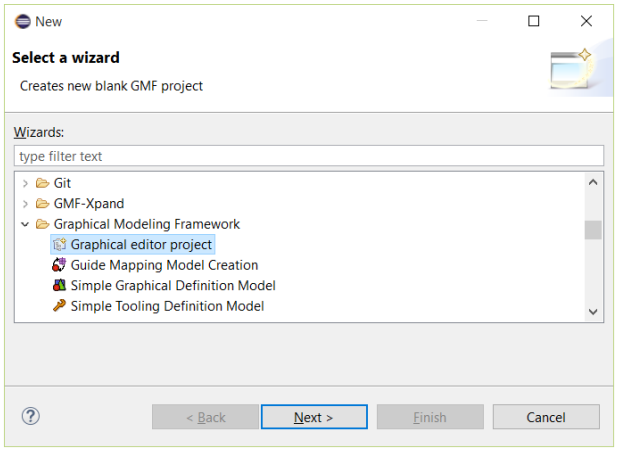

Figura C.3 Creación de un proyecto GMF

La decisión de escoger uno u otro depende del propósito del proyecto, ya que se puede optar por trabajar exclusivamente con EMF, o trabajar con ambos EMF + GMF; aun así, los dos tipos de proyecto sirven para los mismos propósitos en términos prácticos.

Una vez creado el proyecto se debe abrir la vista **GMF Dashboard** en caso de no estar ya visible; para ello se ha de ir a *Window->Show view->Other…* (figura C.4).

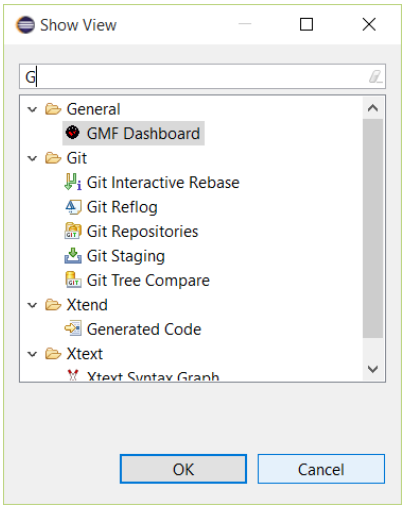

Figura C.4 Búsqueda de la vista GMF Dashboard

Una vez abierta, seleccionamos el enlace *create* del recuadro *Domain Model*, como se muestra en la figura C.5, para proceder a la creación del modelo de dominio (ecore):

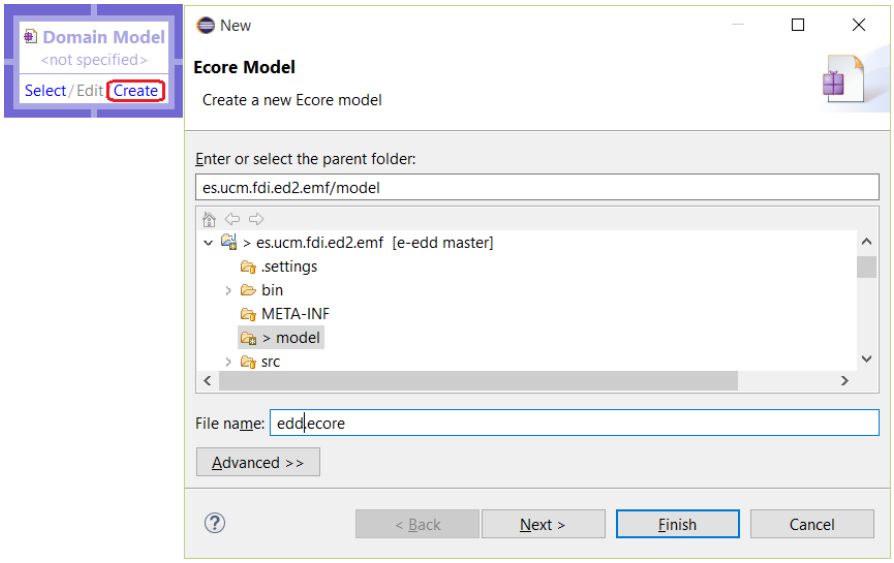

Figura C.5 Creación del modelo de dominio (ecore)

Este archivo deberá plasmar la representación del metamodelo a diseñar; puede tratarse como XMI editando sus propiedades pero, para facilitar su edición, podemos tratarlo también de forma gráfica, que será más sencillo. Para ello solo es necesario seleccionar el archivo *.ecore y con el menú contextual seleccionar *Initialize Ecore Diagram…* (figura C.6).

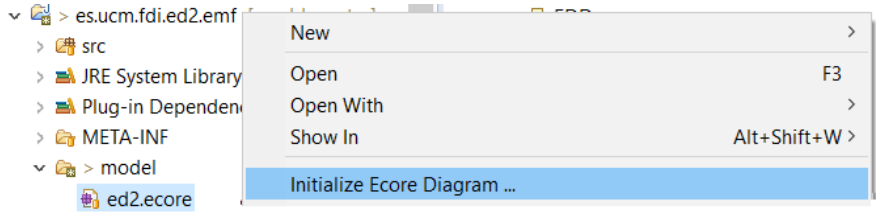

Figura C.6 Inicialización del editor de diagramas a partir del ecore

En el editor de diagramas debemos usar las reglas de asociación, composición, herencia, etc., de UML con el fin de definir lo que será nuestro metamodelo (figura C.7 - véase sección 7).

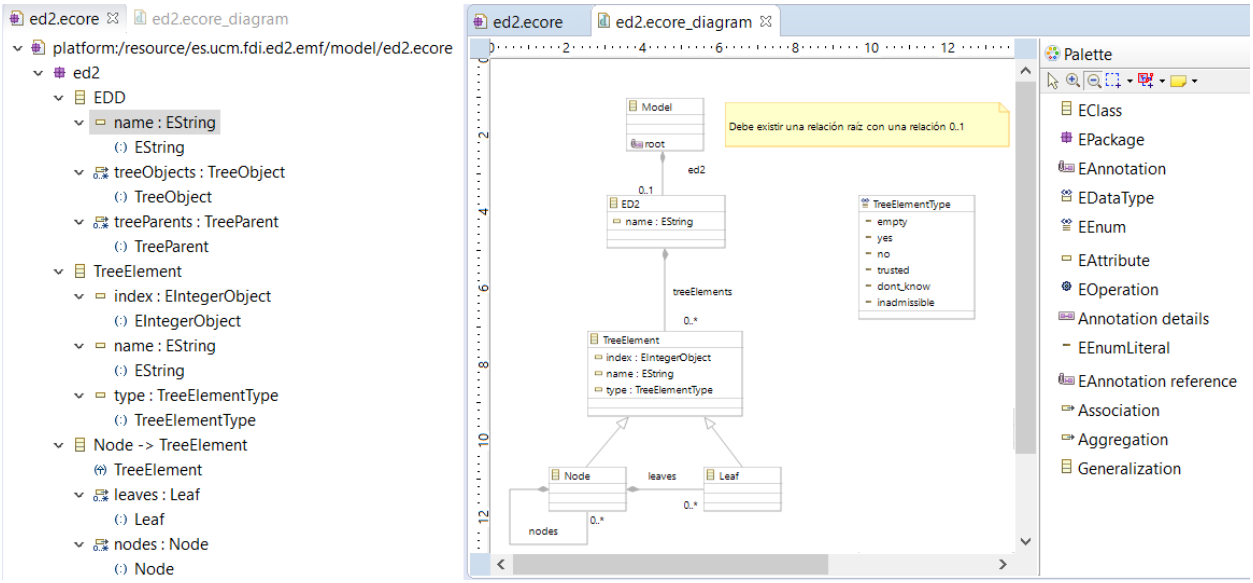

Figura C.7 Definición del metamodelo en el editor gráfico

Finalmente, solo restará dar un identificador al paquete, el cual permitirá registrarlo de forma unívoca. Para ello, nos posicionaremos en el archivo ecore donde deberemos seleccionar el nodo paquete y editar sus propiedades (figura C.8).

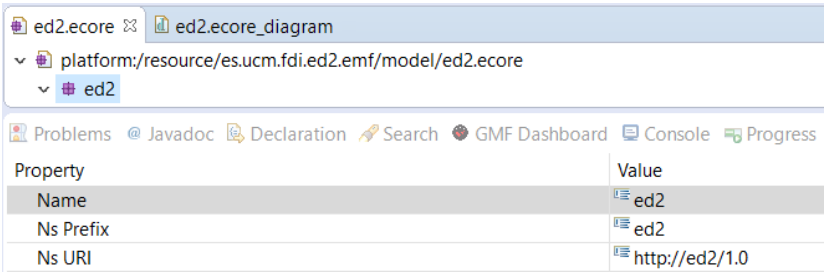

Figura C.8 Propiedades del metamodelo

Finalmente habrá que guardar todos los cambios y con esto quedará definido el modelo EMF.

2. Definición del modelo de dominio de generación

El siguiente paso es crear el modelo de dominio de generación de código EMF (genmodel); seleccionamos el enlace *Derive* que nos abrirá un wizard para crearlo a partir del ecore definido anteriormente (figura C.9).

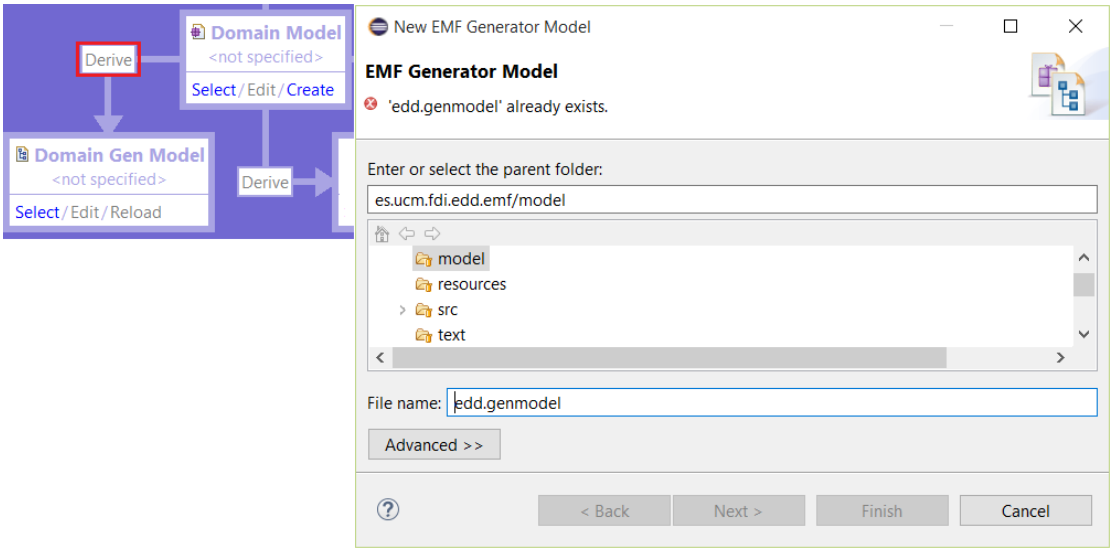

Figura C.9 Creación del generador del modelo de dominio

El fichero genmodel es el encargado de transformar el modelo definido en el ecore a código fuente, pero antes de su creación se debe definir el paquete base a partir del cual estarán contenidas las clases. Abrimos el fichero genmodel y seleccionamos el nodo paquete para editar sus propiedades (figura C.10).

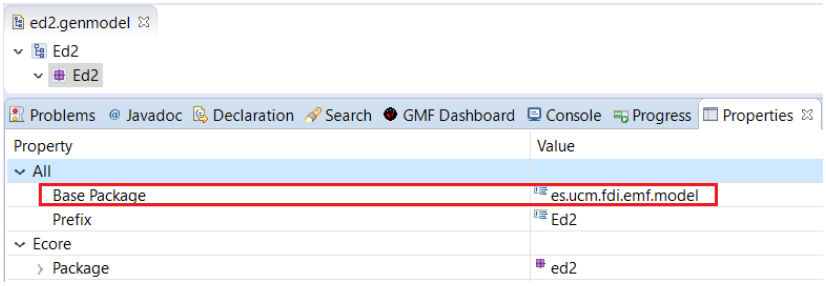

Figura C.10 Propiedades del generador del modelo de dominio

Finalmente, para generar el código fuente bastará con seleccionar el comando "*Generate All*" del menú contextual a partir del archivo genmodel (figura C.11).

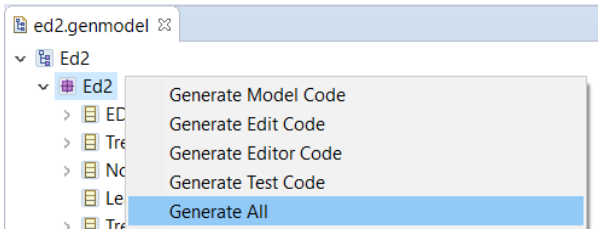

Figura C.11 Generación de proyectos utilizados para el nuevo modelo EMF

Tanto el paso 1 como el 2 son de uso exclusivo para EMF, es decir, podremos tratar estructuras de árboles y usar la factoría Ed2 para la navegación entre objetos, tal y como se describe en la sección 7.

Los pasos siguientes son para la construcción exclusiva del editor gráfico pero es necesario haber cumplimentado los pasos previos.

3. Definición gráfica del modelo

El siguiente paso consiste en la creación del modelo de definición gráfica, es decir, la definición de los ítems que serán usados para dibujar nodos o conexiones. Debemos seleccionar el enlace "*Derive*" para crear el archivo gmfgraph, que define el modelo del aspecto visual que tendrán las instancias en el editor gráfico (figura C.12).

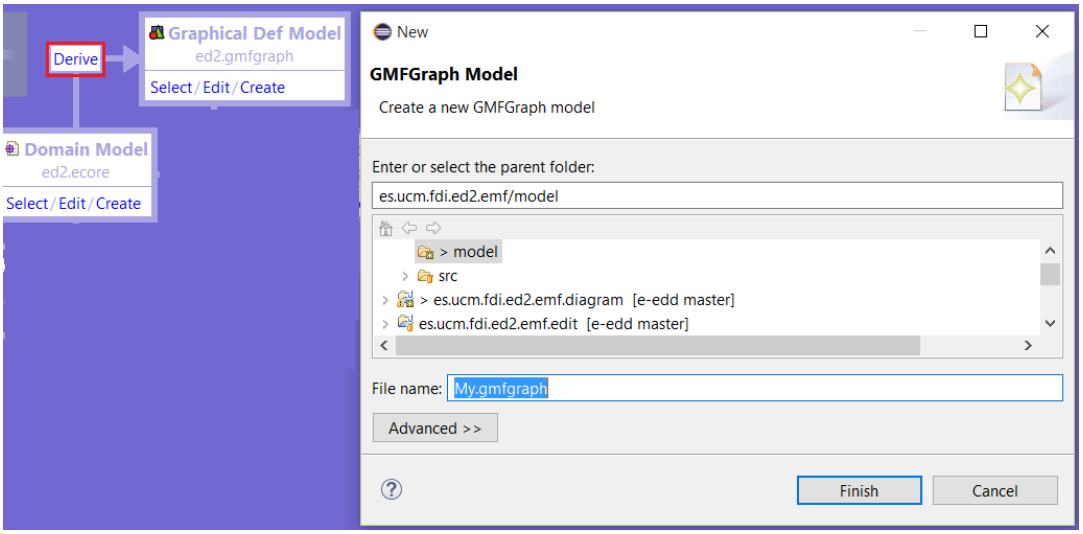

Figura C.12 Creación del modelo de definición gráfica

Se abrirá un diálogo en donde debe seleccionar el nodo raíz de su metamodelo que represente al elemento del diagrama, es decir, la clase Model (figura C.13).

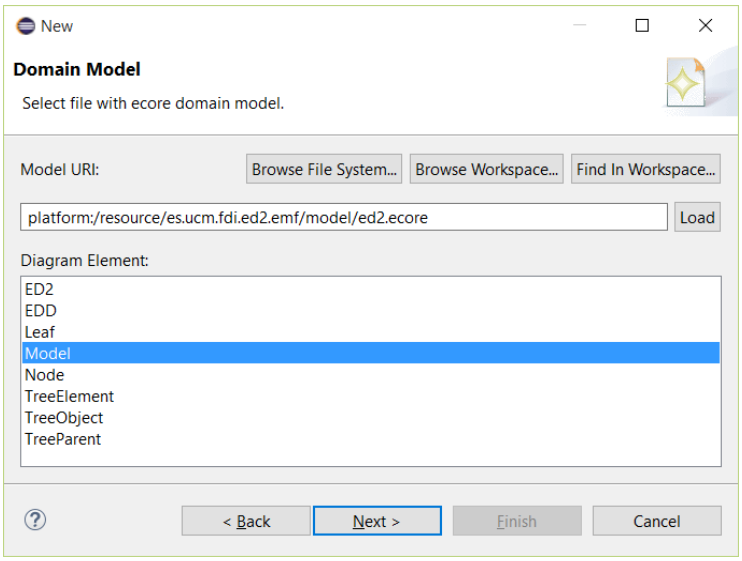

Figura C.13 Selección del nodo raíz del metamodelo

A continuación, se listan todos las clases y relaciones que conforman el metamodelo en donde deberá seleccionar Nodos, Conexiones y Etiquetas, respectivamente (figura C.14 - véase orden de las columnas).

Los nodos que seleccionamos son los objetos que pueden dibujarse en el lienzo, mientras que las conexiones son las líneas que los unirán y las etiquetas son los literales que aparecen en cada tipo de objeto, según se desee.

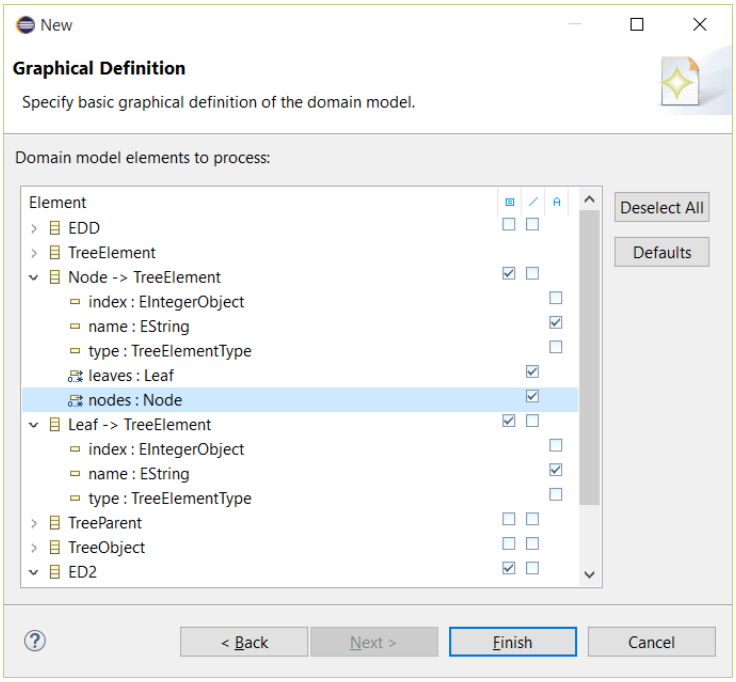

Figura C.14 Selección de nodos, conexiones y etiquetas

Una vez finalizada la elección, se generará el archivo que lo define. En dicho archivo es posible personalizar el aspecto de las instancias de metamodelo, color, tipo de línea, etc; para hacerlo se debe ir modificando las propiedades de cada tipo para adaptarlo a las necesidades específicas (figura C.15).

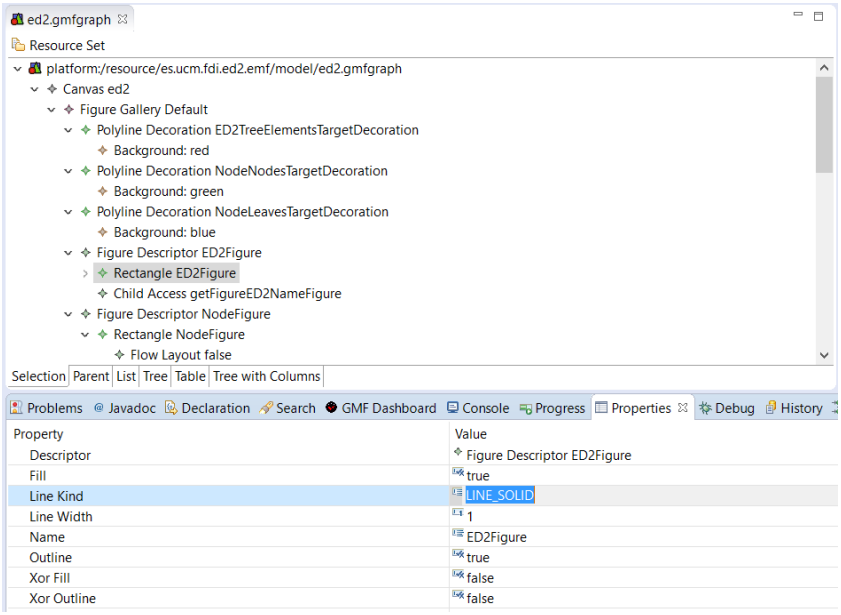

Figura C.15 Editor del modelo de definición gráfica

4. Definición de la paleta de herramientas de modelo

Este apartado está destinado para la construcción de la paleta que se usará durante la edición en el editor gráfico. Siguiendo con la misma dinámica empleada hasta ahora seleccionamos el enlace "*Derive*" para abrir el wizard de creación de este modelo (figura C.16).

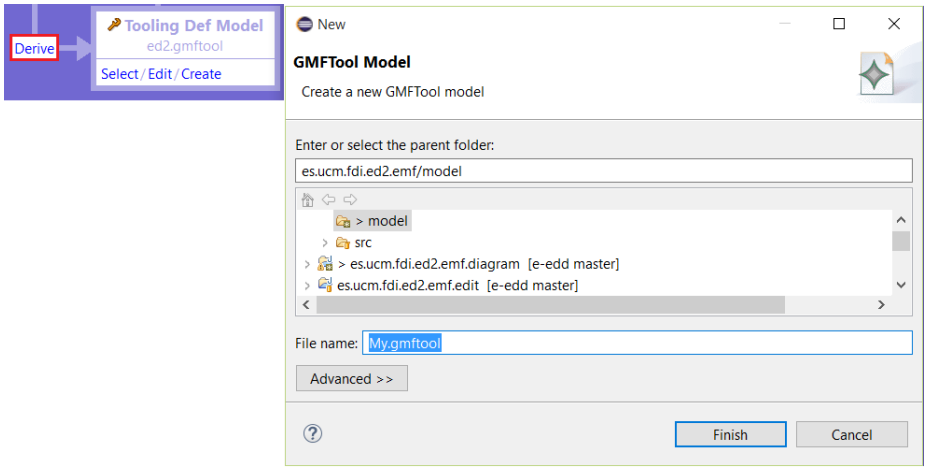

Figura C.16 Creación de la paleta del modelo de definición gráfica

De forma similar al caso anterior, lo primero será seleccionar el nodo raíz del metamodelo, es decir, la clase Model (figura C.17).

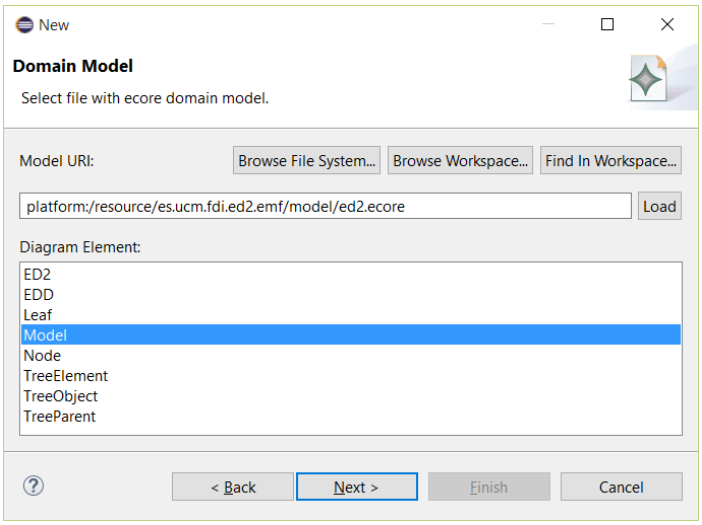

Figura C.17 Selección del elemento raíz del metamodelo

En la siguiente página del asistente se deben seleccionar Nodos y Conexiones, respectivamente. Con esta selección indicará qué instancias se desea que aparezcan en la paleta del editor gráfico, usadas para dibujar en el lienzo (figura C.18).

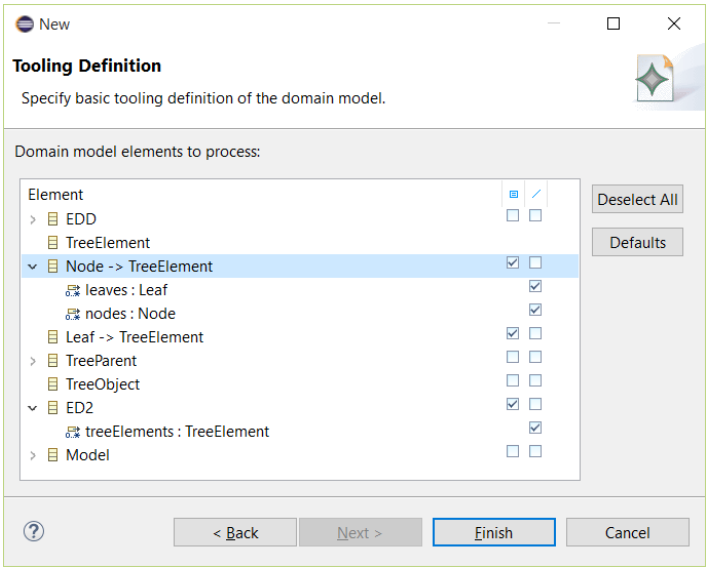

Figura C.18 Selección de nodos y conexiones para la paleta

Obtendremos un archivo con la definición de la configuración realizada (figura C.19). Dicho archivo podrá editarse para personalizar el comportamiento de la paleta (si se permite colapsar o no, imágenes, estilo, etc.)

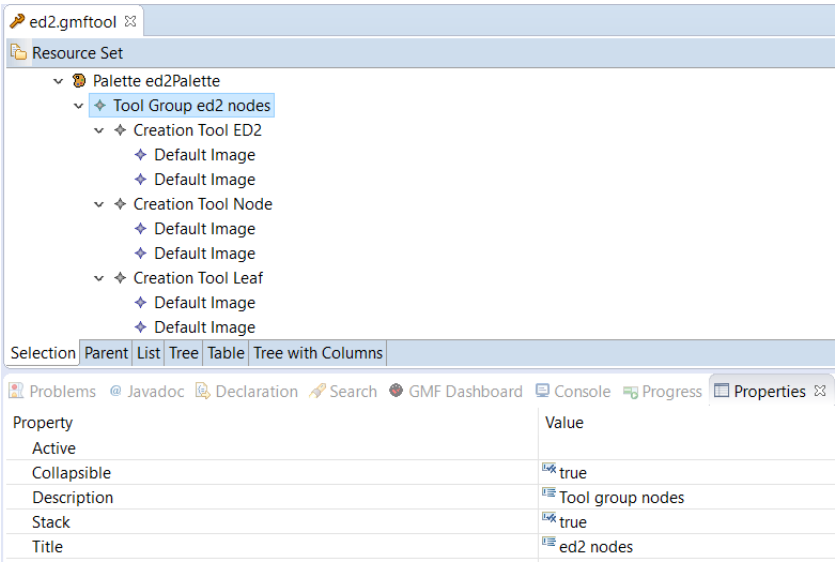

Figura C.19 Editor de la paleta del modelo de definición gráfica

Modificamos el archivo para que dé como resultado el aspecto plasmado en la figura C.20.

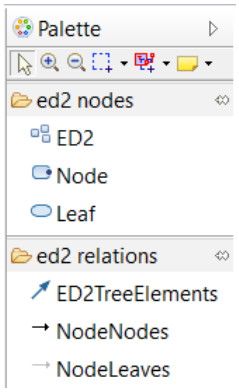

Figura C.20 Selección de nodos y conexiones para la paleta

5. Definición de mapeo de la paleta de herramientas y la definición gráfica

El proceso de *mapping* o correspondencia es el encargado de dar sentido a cada una de las partes que hemos creado hasta ahora. Por un lado, tenemos una definición de elementos implicados en el pintado y, por otro lado, una paleta de herramientas; sin embargo, de momento no existe un vínculo que los una, que es justamente en lo que consiste el presente proceso. Se debe crear el archivo gmfmap tal y como se muestra en la figura C.21, utilizando el enlace *Combinar*.

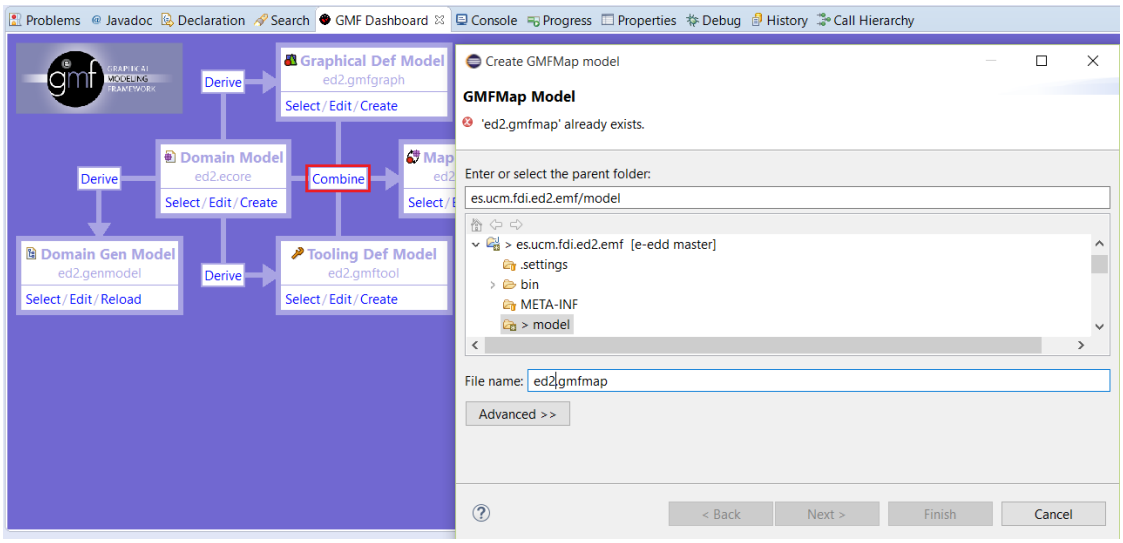

Figura C.21 Creación del modelo de mapeo entre la paleta y la definición gráfica

Al igual que en casos anteriores se debe especificar el elemento raíz del metamodelo (figura C.22).

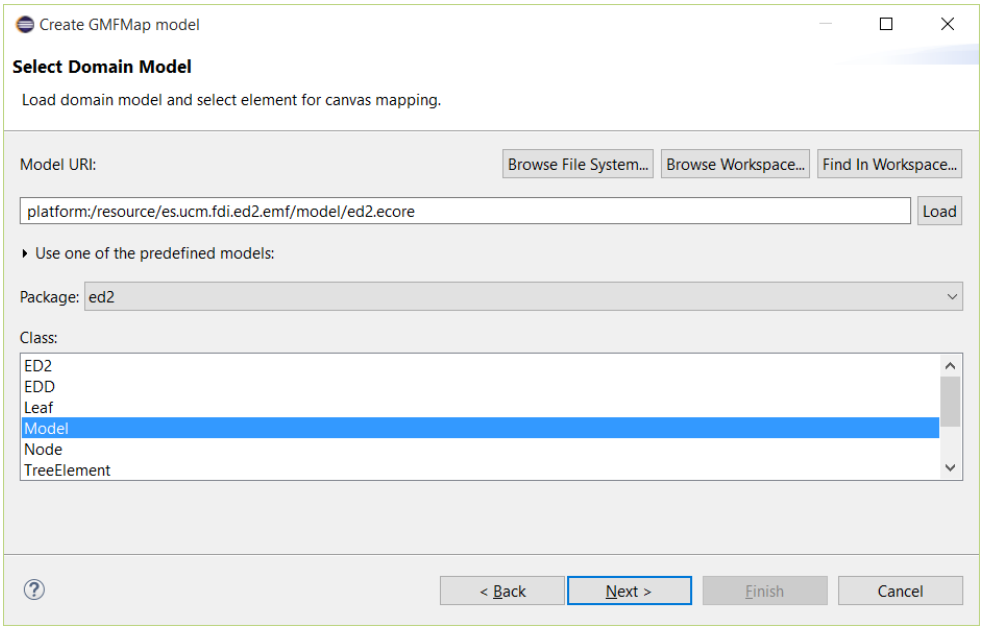

Figura C.22 Selección del objeto raíz del metamodelo.

La siguiente página del asistente solicitará la herramienta de la paleta a utilizar, defina previamente, por lo que habrá que indicarla (figura C.23).

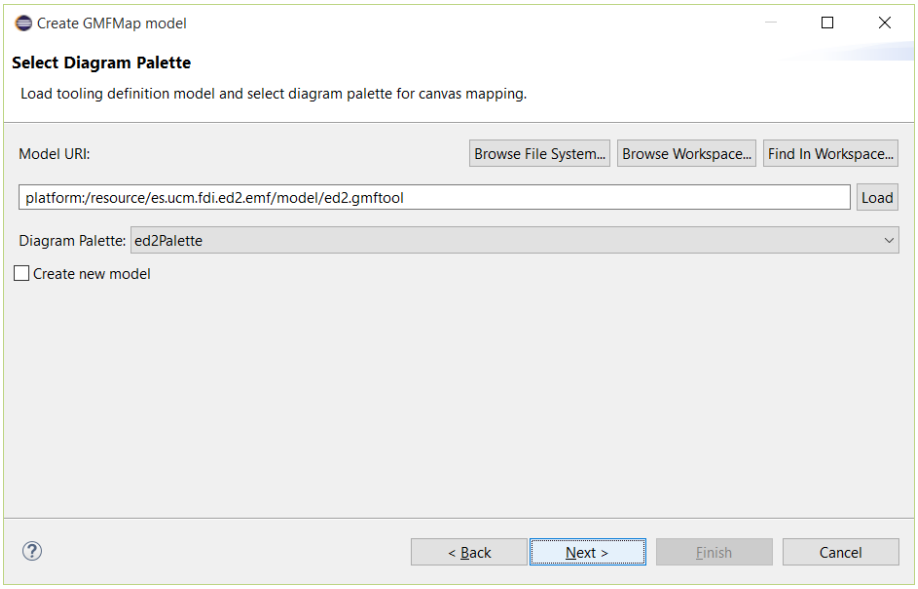

Figura C.23 Selección de la paleta gráfica.

En la última página del diálogo, se debe indicar lo que serán nodos y lo que serán relaciones; disponemos de varios botones para moverlos y configurarlos (figura C.24).

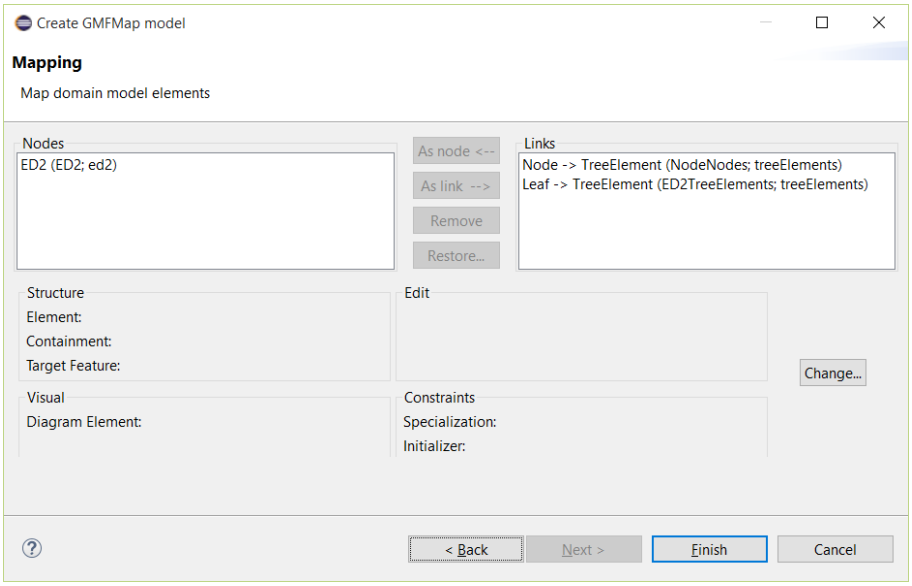

Figura C.24 Selección de la paleta gráfica.

Una vez finalizada la configuración, se obtendrá el correspondiente archivo de definición (figura C.25). Dicho archivo debe ser cuidadosamente verificado y editado para que cada parte definida en los modelos previos haya sido mapeada correctamente, de lo contrario se pueden obtener anomalías durante la interacción con el editor gráfico.

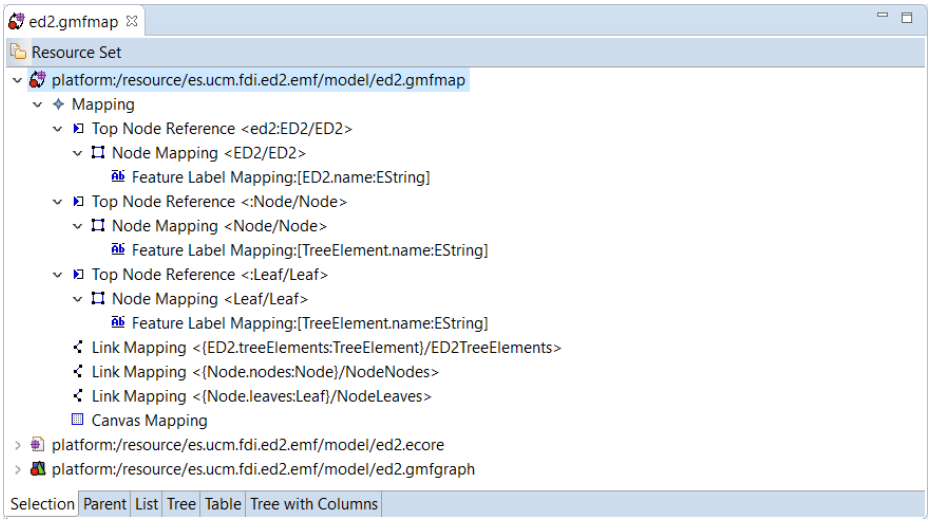

Figura C.25 Modelo de mapeos entre la definición gráfica y la paleta.

6. Definición del modelo de generación del editor de diagrama

El último de los pasos consiste en la creación del modelo de generación para el editor gráfico. Para ello procedemos como se indica en la figura C.26.

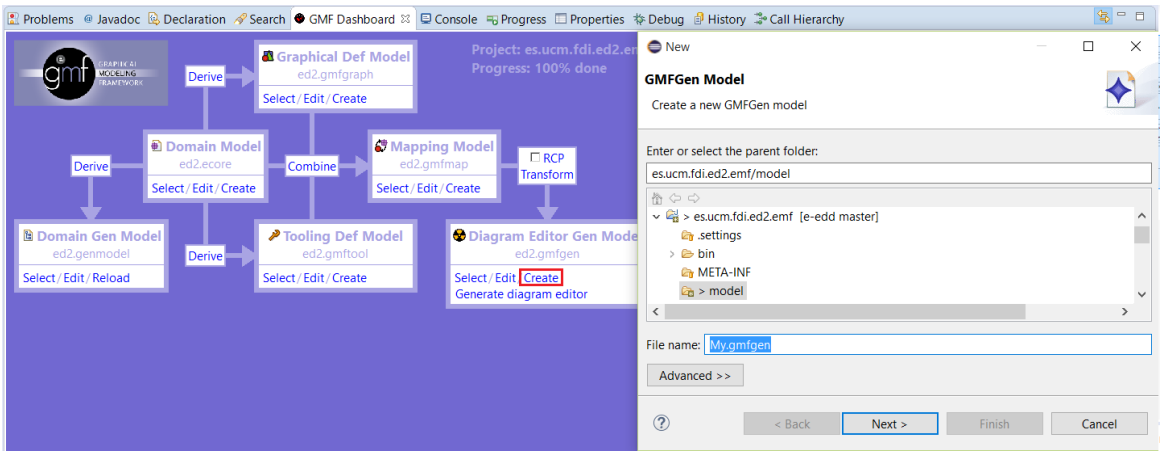

Figura C.26 Creación del modelo generador del editor de gráfico

Una vez finalizada la creación del modelo, esta completado el proceso de definición de la herramienta. Los 6 tipos de modelo más el diagrama deberán estar presentes dentro del directorio model del proyecto GMF (figura C.27).

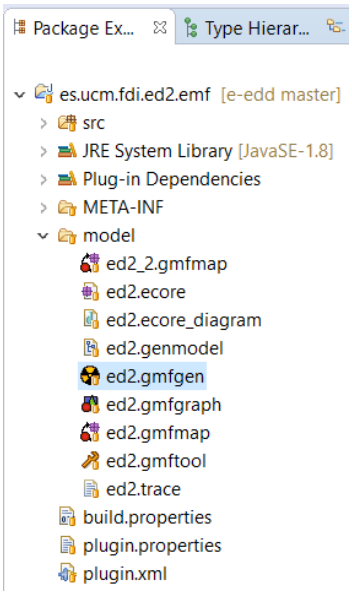

Figura C.27 Listado de los modelos definidos por GMF

El último de los pasos consiste en la generación del código fuente del editor gráfico, para ello debemos posicionarnos sobre el archivo *ed2.gmfgen* y, mediante el menú contextual, seleccionar *Generate Diagram Code*. Si todo ha ido bien aparecerá un mensaje de confirmación, en caso contrario se mostrarán errores que deberán repararse para poder generar el código; de ser así debemos repasar los modelos previos y verificarlos.

Un último paso, no menos importante, es asignar el nombre de la extensión que será usada para manejar el tipo de editor que hemos creado. Para ello, desde el modelo gmfgen, seleccionamos el nodo GenEditor y editamos sus propiedades como se muestra en la figura C.28.

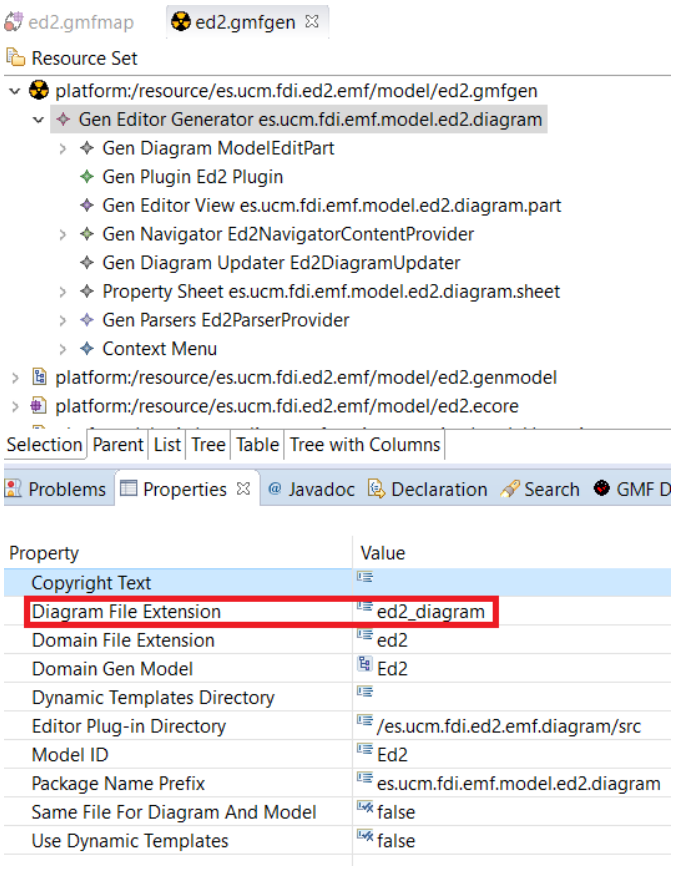

Figura C.28 Listado de los modelos definidos por GMF

Con esto queda concluida la creación del editor gráfico. Para probarlo ejecutamos una instancia de una aplicación de Eclipse o lo instalamos (véase el apéndice B). La figura C.29 muestra un ejemplo del aspecto que debe presentar.

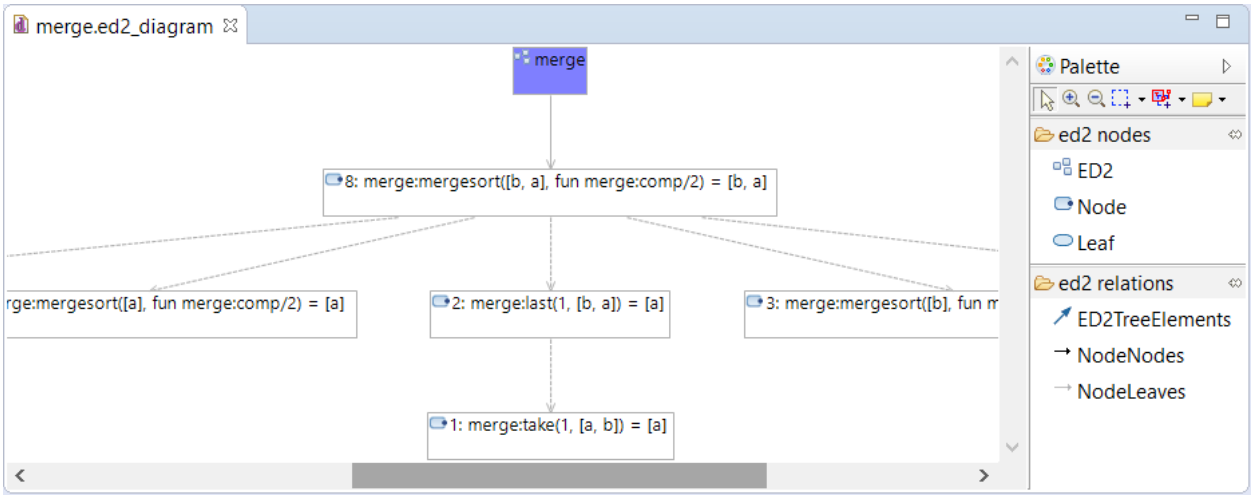

Figura C.29 Ejemplo del editor gráfico recién creado (ed2_digram)

Apéndice D - EMF a Graphviz

La idea original, en la que se basa la herramienta emf2gv, es la de proporcionar una solución ligera como una alternativa a GMF (véase el apéndice C), ya que permite una representación de los objetos EMF (véase sección 7 y el apéndice C, pasos 1 y 2) como si de un diagrama de clases se tratara, utilizando la utilidad Graphviz.

Para proceder a la representación visual a partir del modelo EMF, que es quién tiene la representación jerárquica, bastará con posicionarse y seleccionar el archivo *.ed2 sobre el cual deseamos obtener su representación gráfica y, mediante el menú contextual, seleccionar "*Run As / Emf To Graphviz Transformation*" (figura D.1).

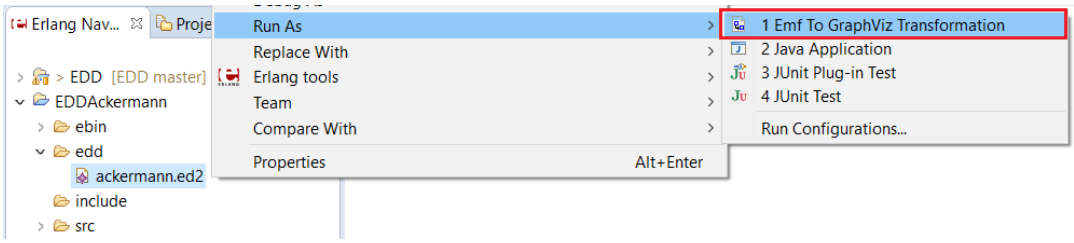

Figura D.1 Transformación del modelo EMF a Graphviz

Una vez ejecutado, se generará un archivo con el mismo nombre del archivo seleccionado pero con extensión jpg, es decir, la imagen que representa el contenido del modelo (figura D.2).

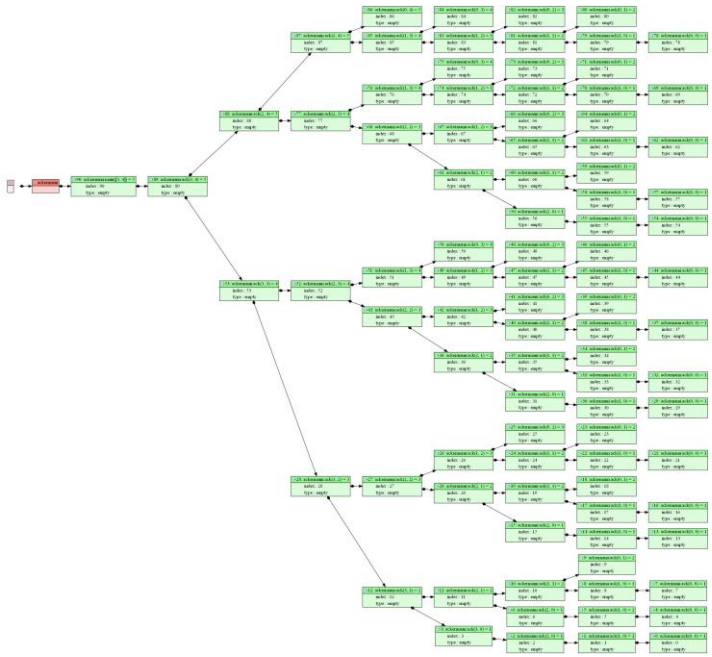

Figura D.2 Ejemplo de transformación de un modelo EMF a Graphviz