



Eusko Jaurlaritzaren Informatika Elkarte  
Sociedad Informática del Gobierno Vasco

UDA – Utilidades de desarrollo de aplicaciones

## Plugin UDA. Guía de desarrollo

Fecha: 06/06/2011

Referencia:

EJIE S.A.  
Mediterráneo, 14  
Tel. 945 01 73 00  
Fax. 945 01 73 01  
01010 Vitoria-Gasteiz  
Posta-kutxatila / Apartado: 809  
01010 Vitoria-Gasteiz  
[www.ejje.es](http://www.ejje.es)



*UDA – Utilidades de desarrollo de aplicaciones* by [EJIE](http://www.ejje.es) is licensed under a [Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 Unported License](https://creativecommons.org/licenses/by-nc-sa/3.0/).

## Control de documentación

Título de documento: Plugin UDA - Guía de Desarrollo

### Histórico de versiones

Código:	Versión:	Fecha:	Resumen de cambios:
	1.0.0	06/06/2011	Primera versión.
	1.1.0	14/09/2011	Cambios en la estructura y contenido del documento.

### Cambios producidos desde la última versión

Cambios en la estructura y contenido del documento.

### Control de difusión

Responsable: Ander Martinez

Aprobado por:

Firma:

Fecha:

Distribución:

### Referencias de archivo

Autor:

Nombre archivo:

## Contenido

	Capítulo/sección	Página
1	Introducción	1
2	Desarrollo de un plugin en Eclipse	2
2.1	Arquitectura de Eclipse RCP	2
2.2	Generación de un plugin	2
2.3	Generación del instalador del plugin	8
3	Plugin de UDA	14
3.1	Implementar las preferencias del plugin	15
3.2	Generar un asistente en el plugin	17
3.3	Generar snippets de patrones de presentación de UDA	24
4	Bibliografía	27

## 1 Introducción

UDA (Utilidades de Desarrollo de Aplicaciones) es el conjunto de utilidades, herramientas, librerías, plugins, guías, y recomendaciones funcionales y técnicas que permiten acelerar el proceso de desarrollo de sistemas software con tecnología Java impulsado por EJIE/EJGV. Sus principales señas de identidad frente a sus predecesores son la **simplicidad y productividad** en el desarrollo y sus capacidades de interfaz gráfico para la construcción de aplicaciones ricas de Internet (Rich Internet Applications).

Una de las utilidades que proporciona UDA es un conjunto de asistentes que se instalan como plugin en el entorno de desarrollo Eclipse.

El objetivo de este documento es explicar la estructura del plugin UDA, su funcionamiento básico y citar un ejemplo de cómo crear un nuevo asistente dentro del mismo plugin. También se incluye un apartado para explicar cómo generar el instalador de un plugin.

El proyecto del pluginUDA se ha implementado con el objetivo principal de desarrollar varios asistentes que facilitarán la realización de diversas tareas al desarrollador. Se han creado los asistentes de:

- Crear nueva aplicación
- Generar código de negocio y control
- Generar mantenimiento
- Añadir proyecto WAR
- Añadir proyecto EJB
- Generar código para EJB Cliente
- Generar código para EJB Servidor

Para crear estos asistentes se ha utilizado la plataforma de Eclipse RCP que provee una estructura flexible y extensible a la hora de implementar diversas aplicaciones (plugins) que se ejecutarán dentro del propio Eclipse.

Para mayor comprensión, se comentarán las funcionalidades básicas de un proyecto RCP, pasando por su arquitectura y los ficheros de configuración que lo componen. A la par, se describirá la estructura del plugin de UDA con sus distintos paquetes.

Se proseguirá con la explicación de cómo crear un asistente y generar un proyecto en el workspace desde el mismo.

También habrá una demostración paso a paso de la generación de los ficheros de instalación del plugin mediante los proyectos de Feature y Update Site.

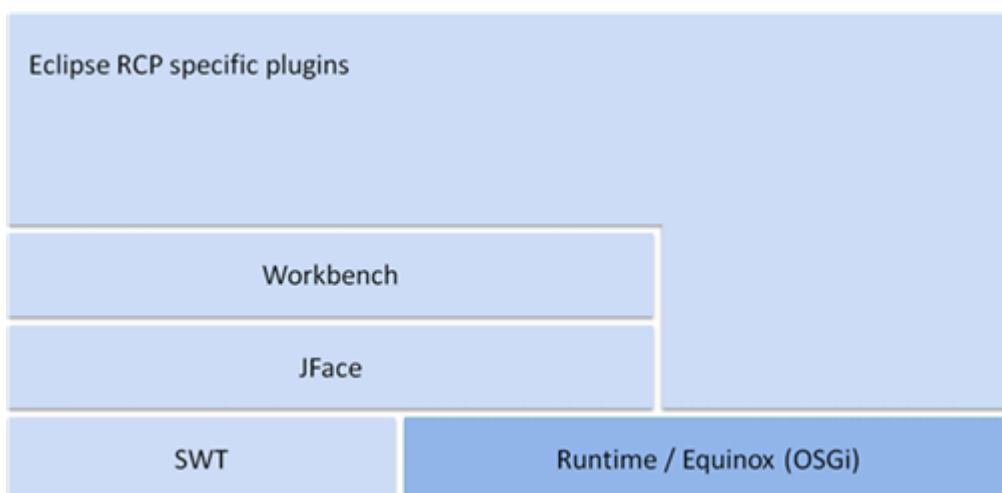
Se finalizará la guía con el método utilizado para la generación de los snippets de los patrones de presentación de UDA.

## 2 Desarrollo de un plugin en Eclipse

### 2.1 Arquitectura de Eclipse RCP

Un plugin de Eclipse es una pequeña aplicación que se puede instalar y ejecutar en el IDE (Entorno de Desarrollo Integrado) de Eclipse donde darán una nueva funcionalidad a la herramienta.

A nivel de arquitectura, una aplicación Eclipse RCP utiliza estos componentes del entorno Eclipse:



**Ilustración 1. Arquitectura básica de componentes de un plugin**

El Runtime OSGi provee un framework para ejecutar los plugins de manera modularizada. También define las especificaciones de los plugins desde las clases públicas que pueden ser utilizadas por otros plugins y sus dependencias que hacen que el plugin implementado funcione correctamente.

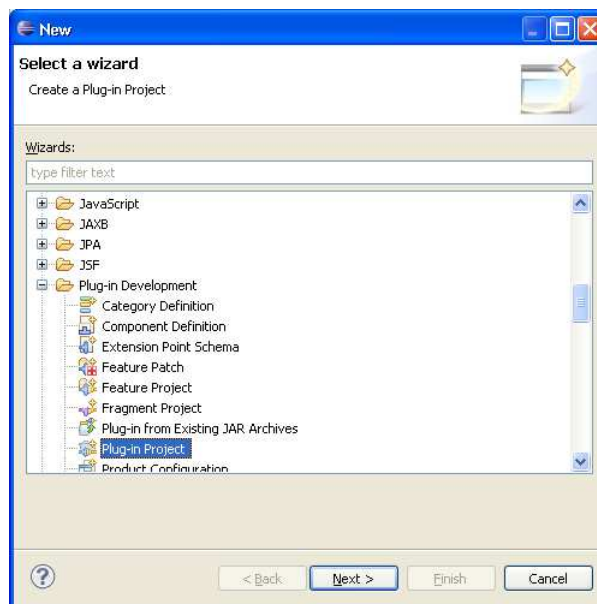
SWT es el estándar de interfaz de usuario de las bibliotecas de Eclipse y JFace ofrece una API como capa superior a SWT.

Otros elementos importantes que proporciona la arquitectura de Eclipse son las extensiones y los puntos de extensión. Estos elementos permiten aportar y recibir la funcionalidad de otros plugin sin cambiar el código existente, ya que el mecanismo de extensión es declarativo.

### 2.2 Generación de un plugin

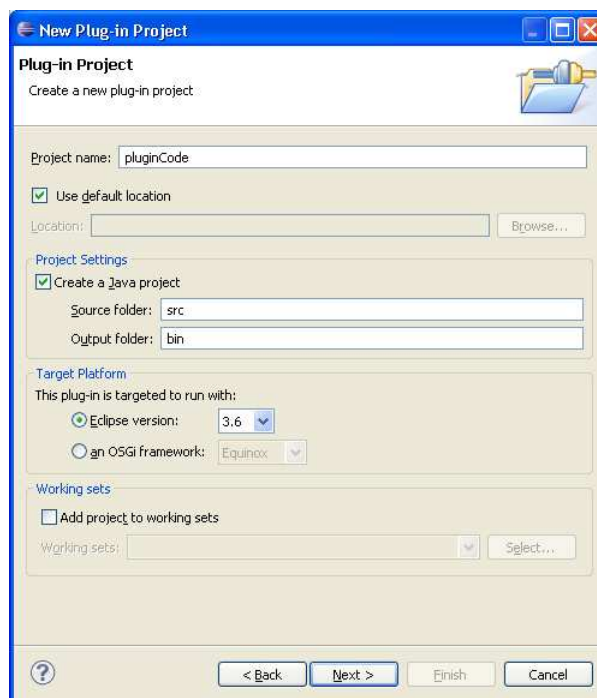
A continuación se detallará la manera de generar un plugin de Eclipse basándose en los pasos seguidos en la creación del plugin de UDA.

1. Se crea un un proyecto que contenga el código con las funcionalidades de tipo Plug-In Project en **"File > New > Other... > Plug-in Development > Plug-In Project"**



**Ilustración 2. Proyecto “Plug-In”**

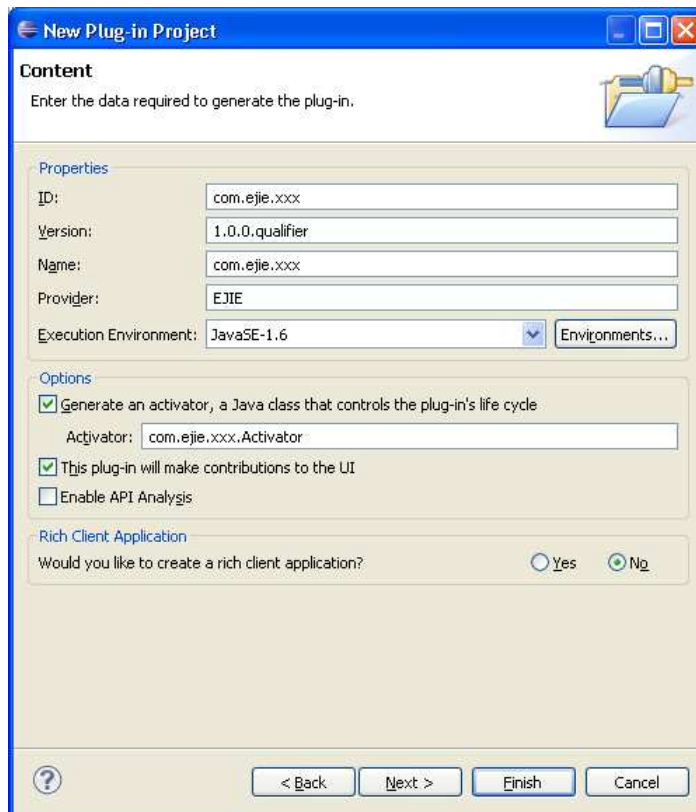
2. Se define el nombre del proyecto, es decir, la carpeta que lo contendrá en el espacio de trabajo (en nuestro caso el proyecto se llamará pluginCode).



**Ilustración 3. Datos del proyecto “Plug-In”**

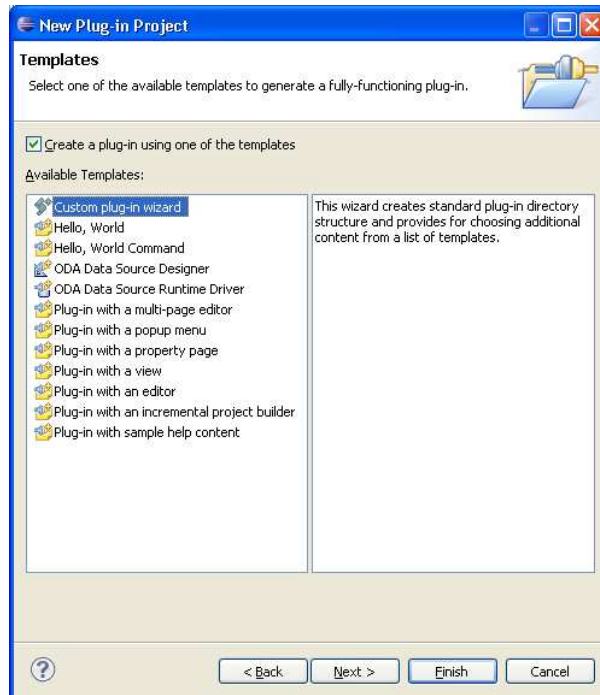
3. Se indica el ID, la versión (por defecto 1.0.0.qualifier que se explicará más adelante), el nombre y el proveedor (EJIE). También se debe definir la clase que inicia el plugin (Activator) con la paquetería requerida (com.ejje.xxx).

NOTA: Se ha utilizado XXX como nombre de plugin/código de aplicación que se va a generar.

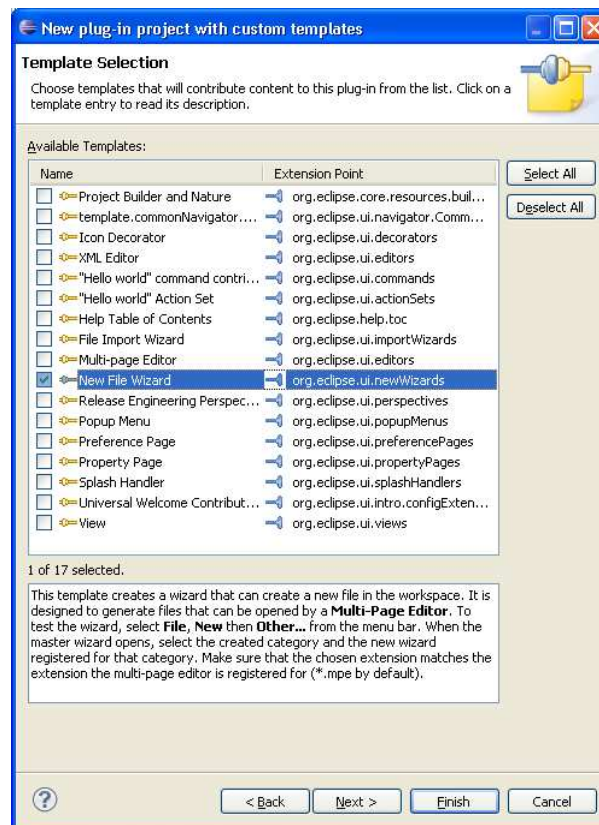


**Ilustración 4. Configuración del plugin**

4. Se selecciona la plantilla “Custom plug-in wizard” y contendrá la plantilla “New File Wizard”



**Ilustración 5. Plantilla del plugin**

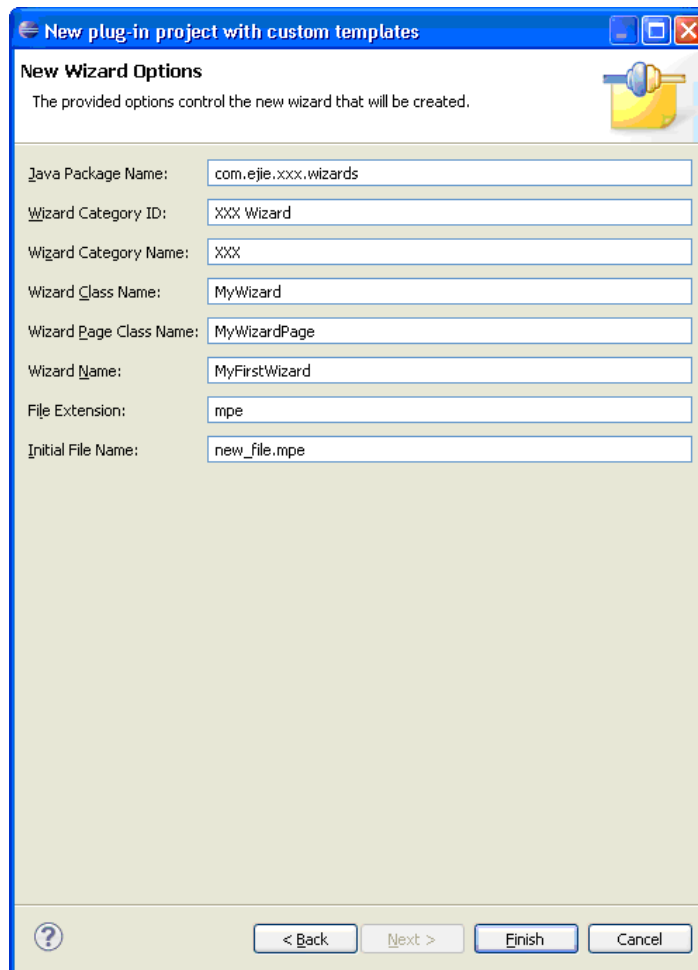


**Ilustración 6. Elementos de la plantilla del plugin**

5. Se rellenan los datos del asistente de ejemplo:
  - a. Java Package Name → Nombre del paquete donde se alojarán las clases
  - b. Wizard Category ID → Identificador de la categoría
  - c. Wizard Category Name → Nombre en el asistente de proyectos de Eclipse (carpeta)
  - d. Wizard Class Name → Clase que implementa el asistente
  - e. Wizard Page Class Name → Clase que implementa la ventana del asistente
  - f. Wizard Name → Nombre en el asistente de proyectos de Eclipse (elemento seleccionable)

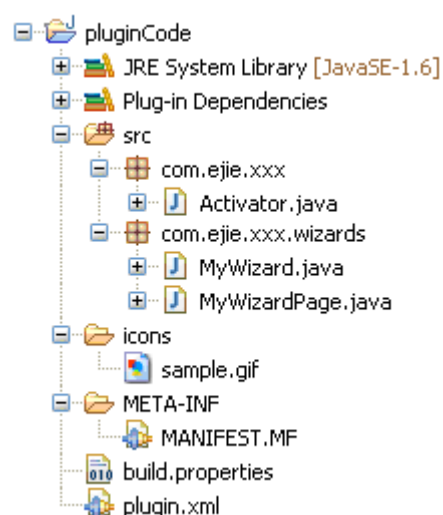
Los campos "File Extensión" e "Inicial File Name" no se comentan por ser propios del plugin seleccionado como ejemplo.





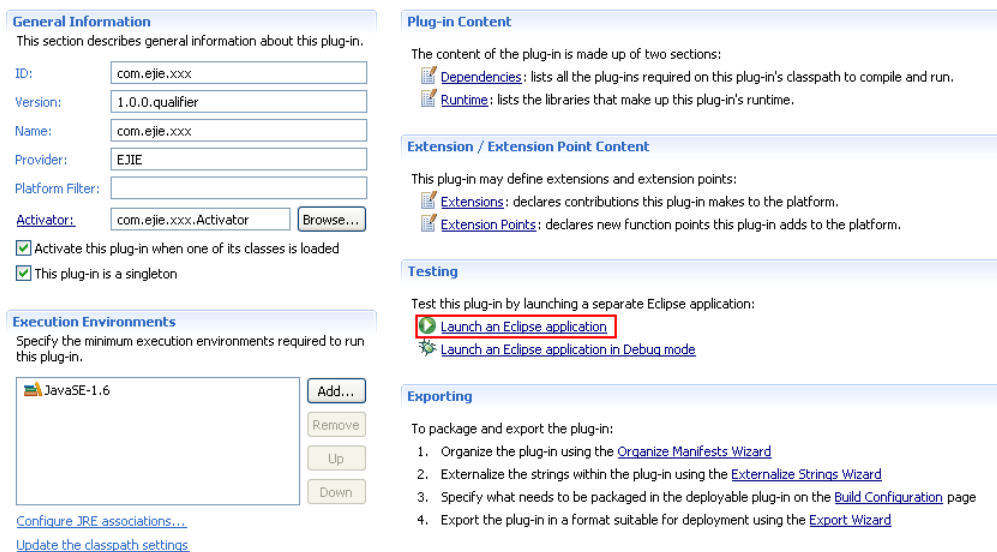
**Ilustración 7. Platilla del plugin**

6. La estructura resultante del plugin será la siguiente:



**Ilustración 8. Estructura del proyecto del plugin**

Si se desea lanzar el plugin para ver que todo funciona correctamente bastará con abrir el fichero “plugin.xml” y pinchar sobre “Launch an Eclipse application” (pinchando en el enlace inferior se arrancará en modo depuración y se podría tracear la ejecución del wizard)



**General Information**  
This section describes general information about this plug-in.

ID:   
Version:   
Name:   
Provider:   
Platform Filter:   
Activator:    
☒ Activate this plug-in when one of its classes is loaded  
☒ This plug-in is a singleton

**Execution Environments**  
Specify the minimum execution environments required to run this plug-in.

[Configure JRE associations...](#)  
[Update the classpath settings](#)

**Plug-in Content**  
The content of the plug-in is made up of two sections:  
[Dependencies](#): lists all the plug-ins required on this plug-in's classpath to compile and run.  
[Runtime](#): lists the libraries that make up this plug-in's runtime.

**Extension / Extension Point Content**  
This plug-in may define extensions and extension points:  
[Extensions](#): declares contributions this plug-in makes to the platform.  
[Extension Points](#): declares new function points this plug-in adds to the platform.

**Testing**  
Test this plug-in by launching a separate Eclipse application:  
[Launch an Eclipse application](#)  
[Launch an Eclipse application in Debug mode](#)

**Exporting**  
To package and export the plug-in:  
1. Organize the plug-in using the [Organize Manifests Wizard](#)  
2. Externalize the strings within the plug-in using the [Externalize Strings Wizard](#)  
3. Specify what needs to be packaged in the deployable plug-in on the [Build Configuration](#) page  
4. Export the plug-in in a format suitable for deployment using the [Export Wizard](#)

Ilustración 9. Fichero plugin.xml

Se abrirá un nuevo proceso de Eclipse cuyo entorno de trabajo (workspace) se encontrará a la altura (mismo nivel en el disco duro) del espacio de trabajo del Eclipse en el que tenemos abierto el proyecto del plugin dentro de la carpeta “runtime-EclipseApplication”. A través del menú “**File > New > Other...**” podremos ver el asistente generado.

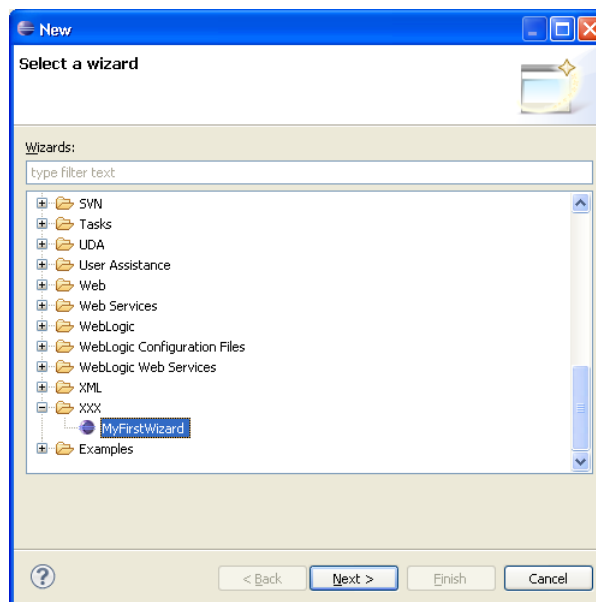


Ilustración 10. Asistente generado

## 2.3 Generación del instalador del plugin

Una vez se ha desarrollado el plugin se deberá generar el instalador del mismo para que los usuarios actualicen su Eclipse y puedan contar con las funcionalidades desarrolladas. Para ello se deberá crear un proyecto *Feature* que recogerá el código del plugin y un proyecto *Update Site* que permitirá su instalación.

Primeramente debemos crear un proyecto **Feature** de un plugin.

1. Se crea un proyecto Feature en “**File > New > Other... > Plug-in Development > Feature Project**”

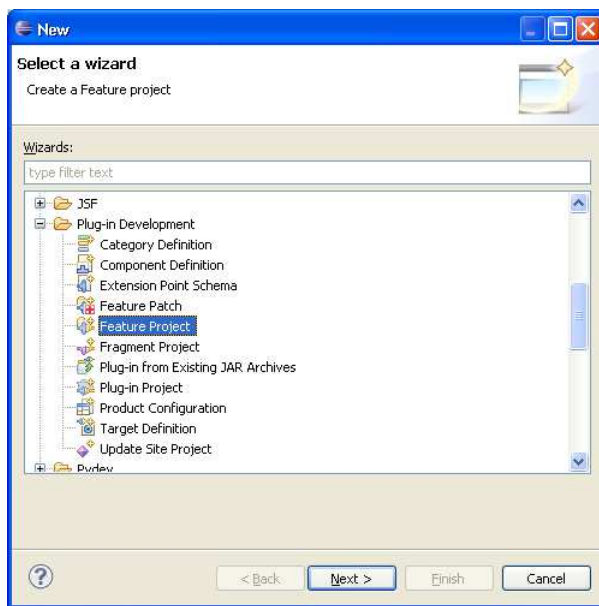
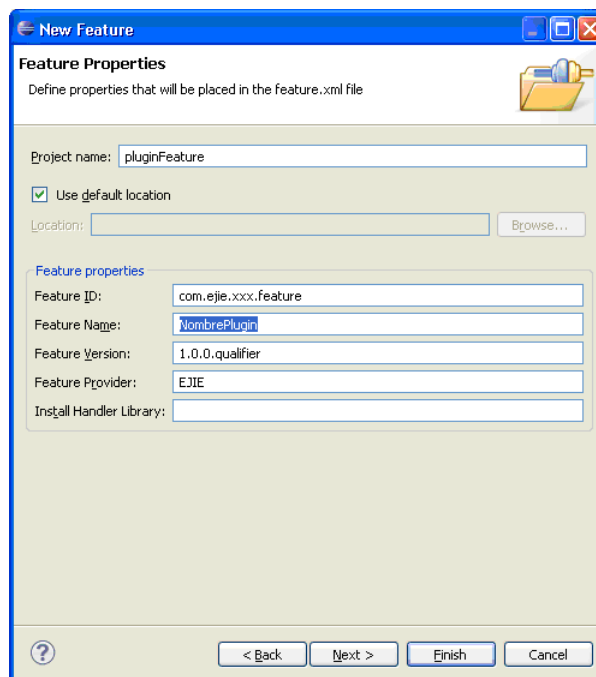


Ilustración 11. Proyecto “Feature”

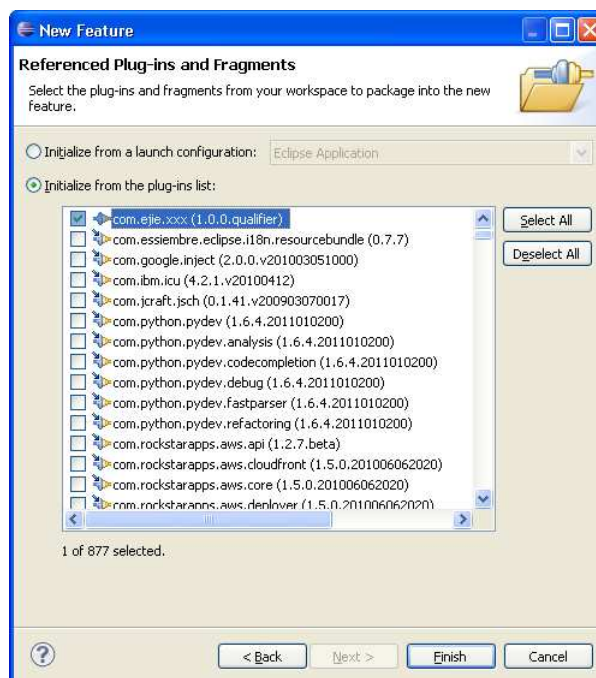
2. Se define el nombre del proyecto, es decir, la carpeta que lo contendrá en el espacio de trabajo (en nuestro caso el proyecto se llamará pluginFeature). Además se indica el ID, la versión (por defecto 1.0.0.qualifier que se explicará más adelante), el nombre y el proveedor (EJIE).

NOTA: Se ha utilizado XXX como nombre de plugin/código de aplicación que se va a generar.



**Ilustración 12. Configuración del Proyecto “Feature”**

3. En la siguiente pantalla enlazamos el proyecto del código del plugin con el proyecto feature. Se deberá buscar el nombre definido en el plugin.



**Ilustración 13. Relación Plug-In con Feature**

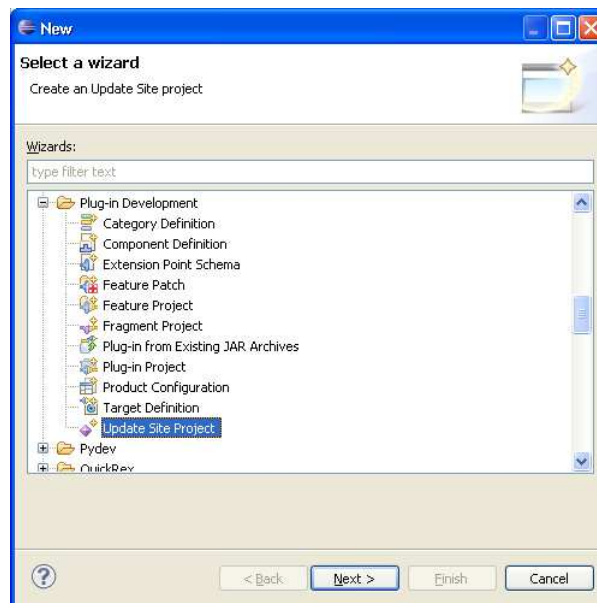
7. La estructura resultante del proyecto será la siguiente:



**Ilustración 14. Estructura del proyecto Feature**

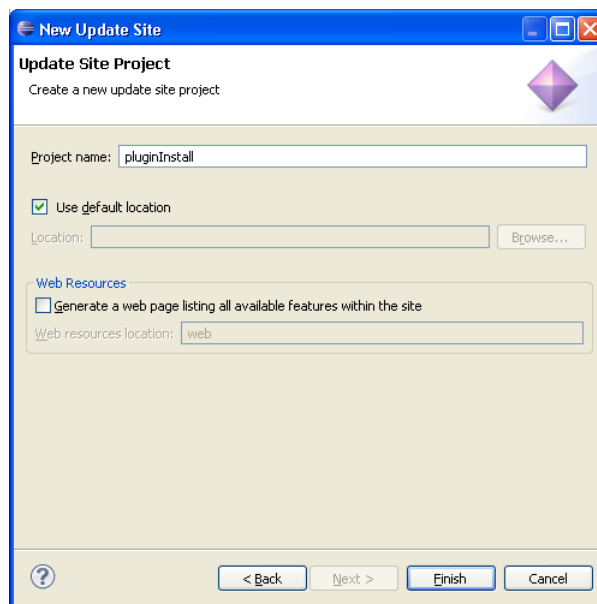
Una vez generado el proyecto Feature, se creará un proyecto **Update Site** encargado de la generación y publicación del instalador.

1. Se crea un proyecto Udate Site en “**File > New > Other... > Plug-in Development > Update Site Project**”



**Ilustración 15. Proyecto “Update Site”**

2. Se define el nombre del proyecto, es decir, la carpeta que lo contendrá en el espacio de trabajo (en nuestro caso el proyecto se llamará pluginInstall).



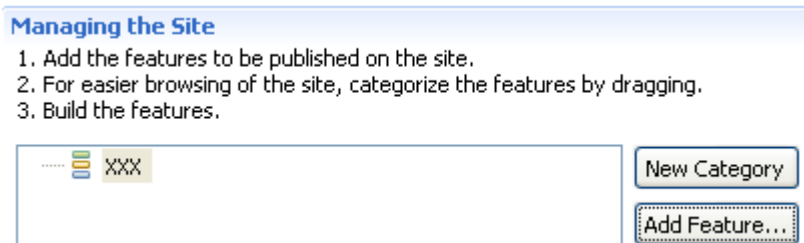
**Ilustración 16. Configuración del Proyecto “Update Site”**

3. La estructura resultante del proyecto será la siguiente:



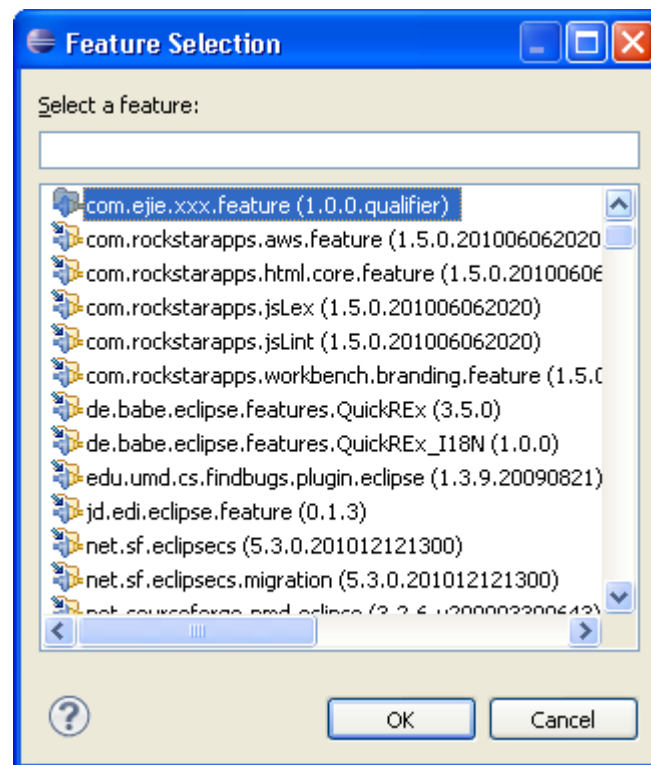
**Ilustración 17. Estructura del proyecto Update Site**

4. Se deberá abrir el fichero site.xml para añadir una categoría (“New Category”) y asignarle el nombre que deseemos por ejemplo XXX.



**Ilustración 18. Añadir una categoría**

5. Situados sobre la nueva categoría añadiremos la feature (“Add Feature”) creada anteriormente



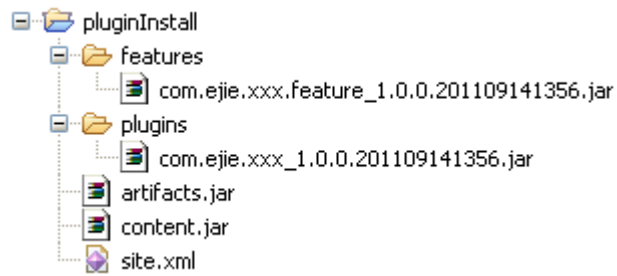
**Ilustración 19. Seleccionar la feature a añadir**

6. Una vez añadida bastará con pulsar sobre “Build All” para que se generen todos los ficheros.



**Ilustración 20. Construir todo**

7. La estructura resultante del proyecto será la siguiente:



**Ilustración 21. Estructura del proyecto Update Site**

Una vez generados los ficheros del instalador vemos como el número de versión que antes era 1.0.0.qualifier se ha modificado pasando a ser tener la fecha en lugar del literal “qualifier”. El formato en concreto es el siguiente (sin separadores): año-mes-día-hora-minuto.

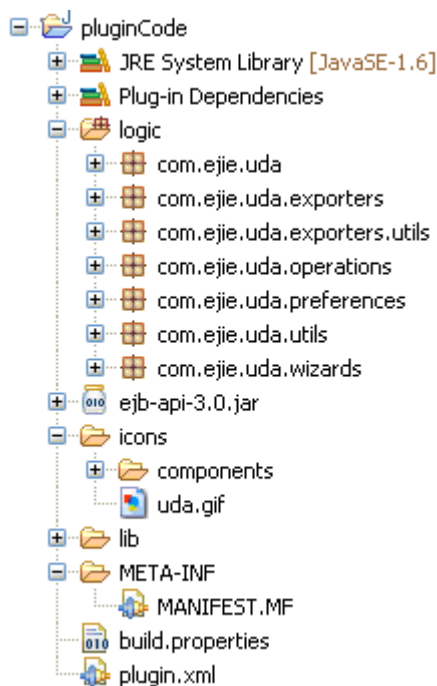
En caso de necesitar generar versiones posteriores, bastaría con modificar el número de versión (tanto de plugin como de feature) y regenerar los ficheros. El proceso sería el siguiente:

1. **Eliminar todos los ficheros .jar del proyecto “pluginInstall”.** A pesar de que los ficheros se generan con la fecha en el nombre del fichero, en los jars de artifacts y content se van acumulando las diferentes versiones generadas lo que puede provocar errores.
2. **Eliminar la feature en el fichero site.xml.** Se deberá añadir la nueva feature generada.
3. **Pulsar “Build All”.** Se crearan los ficheros necesarios para la instalación.



### 3 Plugin de UDA

Un plugin está compuesto de una colección de ficheros. En la siguiente ilustración se muestra la estructura del plugin de UDA.



**Ilustración 22. Estructura de proyecto pluginCode de UDA**

Las partes más relevantes de esta estructura son:

- **Nombre del proyecto:** pluginCode
- **JRE System Library:** Indica que utilizamos Java 1.6
- **Plug-in Dependencies:** Incluye las dependencias del plugin.
- **Estructura de clases:** Todas las clases parten de **logic**.
  - **com.ejie.uda:** se sitúa la clase Activator.java que está encargada de controlar el ciclo de vida del plugin.
  - **com.ejie.uda.exporters:** contiene las clases necesarias para la generación de código de backend (controller, service, dao y model) a partir del modelo de la base de datos. También encontramos un fichero con la ruta de las plantillas utilizadas y el Reveng.java (contiene las condiciones usadas en la ingeniería inversa).
  - **com.ejie.uda.exporters.utils:** contiene las clases de ayuda para la generación de código de backend (controller, service, dao y model) que utilizan las plantillas.
  - **com.ejie.uda.operations:** aquí están las clases estáticas encargadas de ejecutar tareas ANT, copiar ficheros y estructuras de carpetas, realizar copias mediante plantillas utilizando Freemarker, ejecutar tareas ANT, entre otras funcionalidades.
  - **com.ejie.uda.preferences:** están las clases encargadas de inicializar y recuperar las preferencias del plugin UDA dentro de Eclipse.

- **com.ejje.uda.utils**: en este paquete van las clases de utilidades, de conexión, la clase de constantes y otras clases auxiliares para la programación.
- **com.ejje.uda.wizard**: las clases contenidas en este paquete son encargadas de las pantallas de los asistentes, son clases que se extienden de la clase *WizardPage* (pantalla de los asistentes) y de la clase *Wizard* (encargada de recuperar los datos de las pantallas y ejecutar las acciones necesarias).
- **icons**: Contiene todas las imágenes utilizadas en el plugin.
- **lib**: Contiene librerías externas auxiliares necesarias en la implementación del plugin.
- **META-INF**: Contiene el fichero MANIFEST.MF que describirá el plugin y sus dependencias.
- **plugin.xml**: Este fichero se rellena con la información de las extensiones y puntos de extensión que sirven para definir las funcionalidades utilizaremos de otros plugins. Se utilizará para configurar las preferencias del plugin, las referencias y los diversos asistentes que vamos a implementar.

### 3.1 Implementar las preferencias del plugin

Las preferencias de Eclipse son un conjunto de clave y valores, donde la clave tendrá un nombre y los valores pueden ser de varios tipos como: una cadena, un valor lógico, tipo primitivo. Además sirven para guardar y recuperar configuraciones que podrán ser tratadas a través de métodos desde el plugin UDA.

Primeramente, para crear la página de preferencias del plugin, se añade un punto de extensión que hará referencia a **org.eclipse.ui.preferencePages** en el fichero *plugin.xml*. En el punto de extensión se indica la clase que se encargará de guardar los valores de las preferencias.

A continuación se ve cómo están referenciadas las preferencias en el fichero *plugin.xml*.

```
<extension
    point="org.eclipse.ui.preferencePages">
    <page
        id="com.ejje.uda.preferences.preferencesPage"
        name="UDA"
        class="com.ejje.uda.preferences.PreferencesMainPage">
    </page>
</extension>
```

- **extension point**: Indica el punto de extensión con determinada característica que se va a utilizar. El punto de extensión *org.eclipse.ui.preferencePages* es el encargado de generar la nueva entrada en las preferencias generales de Eclipse y referenciar a la clase que lo implementará.
- **id**: Identificador único de la página de preferencia.
- **name**: Nombre de la entrada en las preferencias.
- **class**: Clase que implementará las preferencias.

Si es necesario inicializar valores a las preferencias, lo haremos a través de este punto de extensión.

```
<extension
    point="org.eclipse.core.runtime.preferences">
    <initializer class="com.ejje.uda.preferences.Initializer"/>
</extension>
```

- **extension point:** Indica el punto de extensión con determinada característica que se va a utilizar. La extensión `org.eclipse.core.runtime.preferences` es la encargada de inicializar los valores por defecto de la pantalla de preferencias.
- **class:** Clase encargada de inicializar los valores por defecto de la pantalla de preferencias.

Una vez añadido los puntos de extensión necesarios, pasaremos a implementar la clase indicada anteriormente con los campos que guardarán la configuración del plugin.

Ahora visualizaremos el código de la clase **PreferencesMainPage** situada en `com.ejje.uda.preferences`.

```
/**
 * Crea en las preferencias del eclipse una entrada referente a este plugin
 * para indicar la ruta física de las plantillas
 */

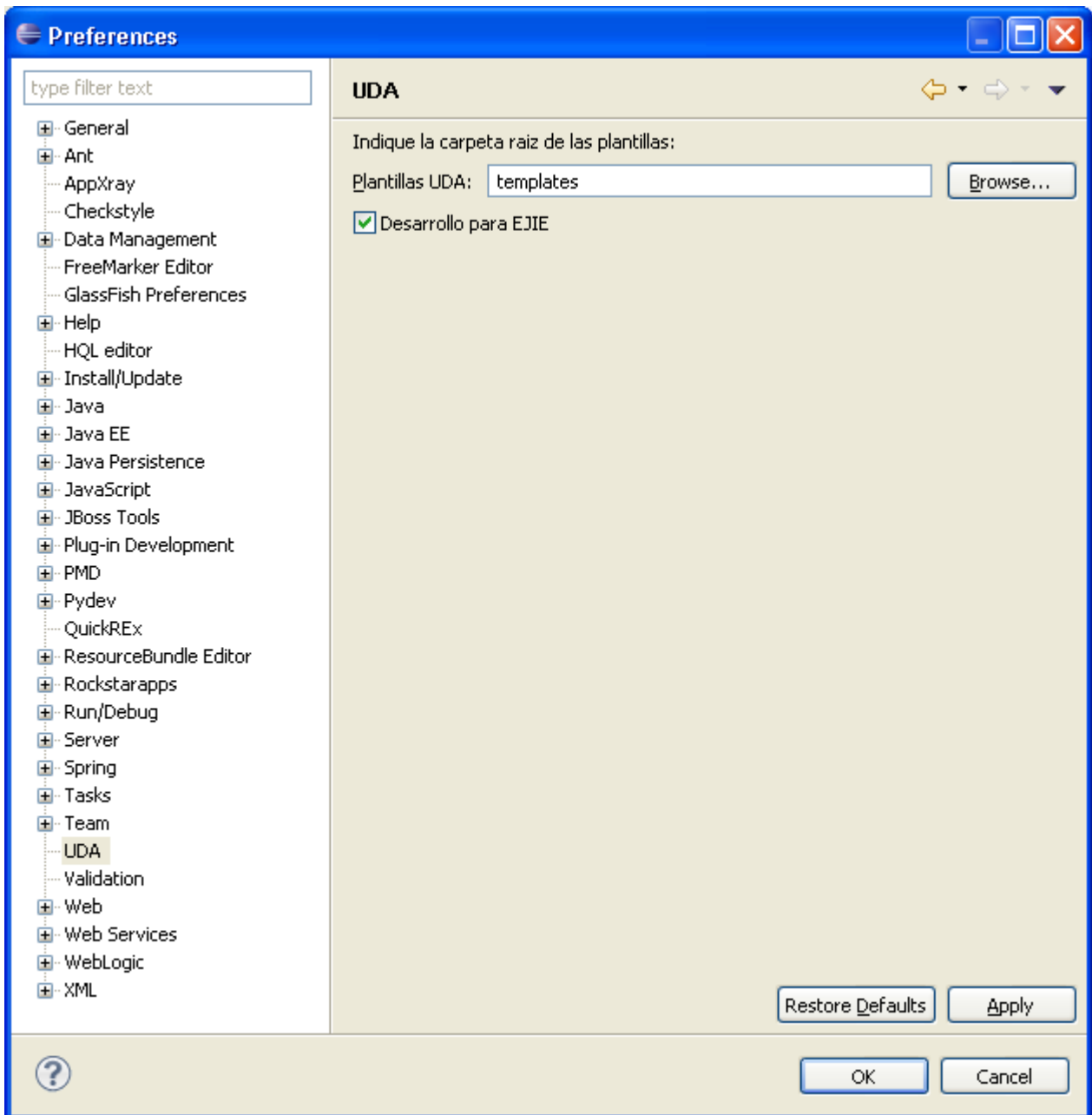
public class PreferecesMainPage extends FieldEditorPreferencePage implements
    IWorkbenchPreferencePage {

    /**
     * Crea un campo para la introducción de la ruta de la plantilla
     */
    @Override
    protected void createFieldEditors() {
        // Campo de texto acompañado de un botón para seleccionar una ruta
        DirectoryFieldEditor templatesUDAFIELD = new DirectoryFieldEditor(
            Constants.PREF_TEMPLATES_UDA_LOCALPATH, "&Plantillas UDA:",
            getFieldEditorParent());
        // Añade el campo
        addField(templatesUDAFIELD);
        // Check que guardará el si estamos o no en un entorno EJIE
        BooleanFieldEditor checkEjje = new BooleanFieldEditor(
            Constants.PREF_EJIE, "Desarrollo para EJIE ",
            getFieldEditorParent());
        checkEjje.loadDefault();
        // Añade el campo
        addField(checkEjje);
    }

    /**
     * Inicializa la preferencia de eclipse con la descripción indicada
     * @param arg0 - workbench
     */
    @Override
    public void init(IWorkbench arg0) {

        setDescription("Indique la carpeta raíz de las plantillas:");
        setPreferenceStore(Activator.getDefault().getPreferenceStore());
        Activator.getDefault().
            getPreferenceStore().
                getBoolean("Constants.PREF_EJIE");
    }
}
```

El resultado final de la configuración de los puntos de extensión y la implementación de la clase de preferencias de Eclipse es la siguiente ilustración.



**Ilustración 23. Preferencias del plugin de UDA.**

### 3.2 Generar un asistente en el plugin

Un asistente (*wizard*) sirve para facilitar al usuario la realización de una determinada tarea, paso a paso, mediante una o varias pantallas en secuencia.

Eclipse interpreta un asistente desde la clase “*org.eclipse.jface.wizard.WizardDialog*”, esta clase controla el proceso de navegación y proporciona la interfaz de usuario como diálogos de mensajes, barra de progreso, etcétera.

El contenido de implementación del asistente va en una clase tipo “*org.eclipse.jface.wizard.Wizard*” y las páginas se proporcionan mediante clases de tipo “*org.eclipse.jface.wizard.WizardPage*”.

Para explicar de manera sencilla la generación de un asistente citaremos a modo de ejemplo las partes más importantes del asistente de **Crear nueva aplicación** del proyecto pluginCode.

Primeramente definiremos en el fichero plugin.xml un punto de extensión de “*org.eclipse.ui.newWizards*” que nos facilitará a referenciar un nuevo asistente en el plugin. En el código abajo indica cómo hacerlo:

```
<extension
    point="org.eclipse.ui.newWizards">
    <category
        id="UDA wizard"
        name="UDA">
    </category>
    <wizard
        icon="icons/uda.gif"
        name="Crear nueva aplicación"
        category="UDA wizard"
        class="com.ejje.uda.wizard.NewApplicationWizard"
        id="com.ejje.uda.wizard.NewApplicationWizard">
    </wizard>
</extension>
```

- **extension point:** Indica el punto de extensión con determinada característica que se va a utilizar. La extensión *org.eclipse.ui.newWizards* es la encargada de generar una nueva entrada en los asistentes de Eclipse.
- **category:** Nombre de la categoría del asistente.
  - **id:** Identificador único de la categoría del asistente.
  - **name:** Nombre que se visualizará al enseñar la categoría.
- **wizard:** Es la definición del asistente.
  - **icon:** fichero del icono del asistente.
  - **name:** Nombre del asistente que se mostrará en pantalla.
  - **category:** Categoría a la que va a pertenecer el asistente.
  - **class:** Clase que implementará el asistente.
  - **id:** Identificador único del asistente.

El resultado de esta configuración será lo visualizado en esta ventana, que se obtiene al pinchar en el menú **File > New > Other...**

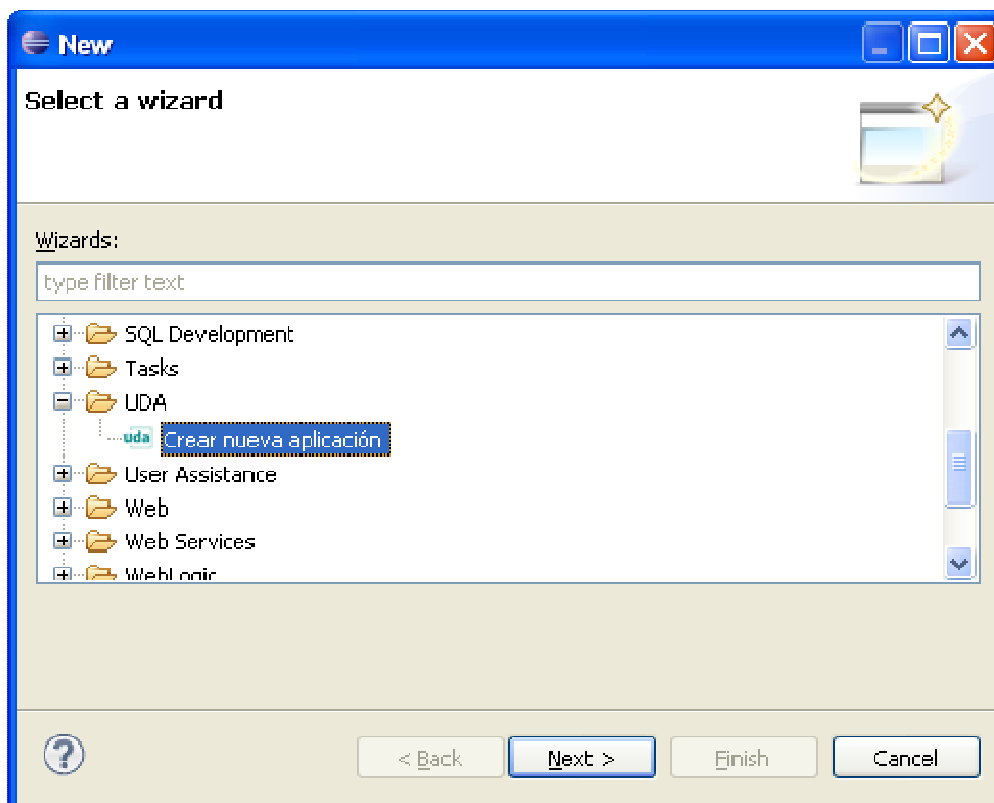


Ilustración 4. Referencia al nuevo asistente.

Nos queda implementar la clase que implementará el asistente y las ventanas del mismo.

La clase que implementará el asistente será **NewApplicationWizard.java**, situada en el paquete *org.ejje.uda.wizard*.

Empezamos enseñando el constructor de la clase, donde se indica que se habilita la barra de progreso durante la ejecución del asistente.

```
/**
 * Constructor
 */
public NewApplicationWizard() {
    super();
    setNeedsProgressMonitor(true);
}
```

Otro método importante de esta clase es **addPages()**, aquí se referencia y se vincula(n) la(s) página(s) del asistente, en las que se situarán los controles que se visualizarán al ejecutar el asistente.

Indicamos que la página de controles es una instancia de la clase *NewApplicationWizardPage* que extiende de *WizardPage*.

```
/**
 * Crea una instancia de nueva ventana del wizard
 */
public void addPages() {
    page = new NewApplicationWizardPage(selection);
    addPage(page);
}
```

Siguiendo el flujo de ejecución, se inicia desde la clase `NewApplicationWizard` y pasa el control a `NewApplicationWizardPage` que pinta controles en la pantalla.

En el constructor de `NewApplicationWizardPage` se indica el título de la ventana y la descripción que se va a mostrar al usuario.

```
public NewApplicationWizardPage(ISelection selection) {  
    super("wizardPage");  
  
    setTitle("Crear nueva aplicación");  
    setDescription("Este Wizard genera la estructura necesaria  
        para desarrollar una aplicación estándar");  
}
```

Se llama al método **createControl()** para que vayamos pintando de manera ordenada los controles. De todos los controles implementados en la pantalla, se ha elegido uno de tipo `Text` cuyo funcionamiento explicaremos de manera detallada mediante comentarios en el código. Los demás controles funcionan, en líneas generales, de forma muy similar.

```
// Controles utilizados en la pantalla  
private Text appCodeText;  
  
...  
public void createControl(Composite parent) {  
  
    Composite container = new Composite(parent, SWT.NULL);  
    GridLayout layout = new GridLayout();  
    container.setLayout(layout);  
    layout.numColumns = 5;  
  
    GridData gd = new GridData(GridData.FILL_HORIZONTAL);  
    GridData gd2 = new GridData(GridData.FILL_HORIZONTAL);  
    gd2.horizontalSpan = 2;  
    GridData gd3 = new GridData(GridData.FILL_HORIZONTAL);  
    gd3.horizontalSpan = 3;  
    GridData gd4 = new GridData(GridData.FILL_HORIZONTAL);  
    gd4.horizontalSpan = 4;  
    GridData gd5 = new GridData(GridData.FILL_HORIZONTAL);  
    gd5.horizontalSpan = 5;  
  
    // Campo texto de Código de aplicación  
    Label labelCodapp = new Label(container, SWT.NULL);  
    labelCodapp.setText("Código de aplicación:");  
    appCodeText = new Text(container, SWT.BORDER | SWT.SINGLE);  
    appCodeText.setTextLimit(10);  
    appCodeText.addListener(SWT.KeyUp, new Listener() {  
  
        ...  
  
        setControl(container);  
    }  
}
```

Para complementar, en esta clase implementamos los métodos privados necesarios y también los métodos públicos (getters) encargados de recuperar los valores de los campos.

```
/**
 * Recupera el código de la aplicación
 *
 * @return código de la aplicación
 */
public String getAppCode() {
    if (appCodeText != null) {
        return appCodeText.getText();
    } else {
        return "";
    }
}
```

En la pantalla siguiente se ve el resultado de lo que hemos desarrollado y una vez cumplimentados los campos obligatorios pinchamos en el botón *Finish*.



**Ilustración 6. Ventana del asistente de Crear nueva aplicación.**

Al presionar el botón *Finish*, el flujo de ejecución vuelve a la clase `NewApplicationWizard`, más precisamente al método **performFinish()** donde recuperaremos los valores introducidos en la ventana. Si es necesario, también haremos más validaciones para garantizarnos que todos los datos obtenidos son correctos.



```
/**
 * Recupera los datos de la ventana e inicia el tratamiento del plugin
 *
 * @return true si la ejecución es correcta, false ecc.
 */
public boolean performFinish() {

    // Recupera la información de la ventana
    final String appCode = page.getAppCode();

    ...

}
```

A partir de este momento, empezamos a procesar toda la información adquirida. Se hace a través del método **doFinish([parámetros])** donde implementaremos la verdadera lógica del asistente, en este caso, generar una aplicación UDA.

El asistente de **Generar nueva aplicación** internamente hace varias tareas como generar cinco proyectos en el workspace, descargar las librerías necesarias, generar y copiar ficheros, etcétera.

Para simplificar nos centraremos en cómo generar el proyecto contenedor [EAR] ([nombre de aplicación]EAR) de la aplicación UDA en el workspace.

```
private void doFinish(String appCode, ...) throws Exception {

    // Recupera la consola
    consola = ConsoleLogger.getDefault();
    consola.println("UDA - INI", Constants.MSG_INFORMATION);

    // Contexto del plugin
    Map<String, Object> context = new HashMap<String, Object>();
    context.put(Constants.CODAPP_PATTERN, appCode.toLowerCase());
    ...
    context.put(Constants.EARCLASSES_NAME_PATTERN, appCode +
        Constants.EARCLASSES_NAME);

    ...

    // Recupera el Workspace para crear los proyectos
    IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();

    ...

    monitor.setTaskName("Creando proyecto EAR...");
    // Crea el proyecto xxxEARClasses
    IProject projectEARClasses = createProjectEARs(root, locationCheck,
        locationText, context, monitor);
    monitor.worked(1);
    ...

    ProjectWorker.refresh(projectEARClasses);
    monitor.worked(1);
    consola.println("UDA - END", Constants.MSG_INFORMATION);

    //Generar resumen si ha generado correctamente
    this.summary = createSummary(context, ejbCheck);

}
```

Dentro de la función **createEAR (...)** que se encarga de implementar varias funcionalidades:

- Crear el proyecto (IProject) de tipo Java en el workspace y organiza el ClassPath
- Añadir el runtime de WebLogic
- Copia y configura ficheros de la carpeta *Templates* través de plantillas FreeMarker,
- Copia el jar necesario para la ejecución de las dependencias de librerías (Maven)
- etc..

En este método manejaremos el proyecto que se va a generar de acuerdo con nuestras necesidades. A continuación se enseña las partes más relevantes del código de generación del proyecto EAR. Los comentarios en el código explican de manera más detallada las líneas del código.

```
private IProject createProjectEAR(IWorkspaceRoot root,
    boolean locationCheck, String locationText,
    Map<String, Object> context, IProgressMonitor monitor)
    throws Exception {

    ...

    // Crea el proyecto de xxxEAR
    IProject projectEAR = root.getProject((String) context
        .get(Constants.EAR_NAME_PATTERN));
    IFacetedProject fpEAR = ProjectFacetsManager.create(
        projectEAR.getProject(), true, null);
    ...

    // Añade el runTime de Oracle
    Set<IRuntime> runtimes = RuntimeManager.getRuntimees();
    for (Iterator<IRuntime> iterator = runtimes.iterator(); iterator
        .hasNext();) {
        IRuntime runtime = (IRuntime) iterator.next();
        if (runtime.getName().contains("WebLogic")
            && runtime.supports(ProjectFacetsManager
                .getProjectFacet("jst.ear"))) {
            fpEAR.addTargetedRuntime(runtime, new SubProgressMonitor(
                monitor, 1));
        }
    }

    ...

    // Crea carpetas del proyecto
    ProjectWorker.createGetFolderPath(projectEAR, "EarContent");
    ProjectWorker.createGetFolderPath(projectEAR, "EarContent/APP-INF");
    ProjectWorker.createGetFolderPath(projectEAR,
        "EarContent/APP-INF/classes");
    ProjectWorker.createGetFolderPath(projectEAR, "EarContent/APP-
INF/lib");

    // Genera los ficheros de configuración del proyecto
    path = projectEAR.getLocation().toString();
    ProjectWorker.createFileTemplate(pathEar, path, "pom.xml", context);
    ProjectWorker.createFileTemplate(pathEar, path, "build.xml",
context);
    path = ProjectWorker.createGetFolderPath(projectEAR,
```

```
        "EarContent/META-INF" );

    // Ruta para las plantillas que se deben procesar (.ftl)
    String pathFileTemplate = projectEAR.getLocation().toString();
    ProjectWorker.createFileTemplate(pathEar, pathFileTemplate,
        "EarContent/META-INF/application.xml", context);
    ProjectWorker.createFileTemplate(pathEar, pathFileTemplate,
        "EarContent/META-INF/weblogic-application.xml", context);

    // Libreria de Ant task para Maven
    path = AntCorePlugin.getPlugin().getPreferences().getAntHome() +
"/lib";

    String pathTemplates = Activator.getDefault().getPreferenceStore()
        .getString(Constants.PREF_TEMPLATES_UDA_LOCALPATH);
    ProjectWorker.copyFile(pathTemplates, path,
        "maven-ant-tasks-2.1.1.jar", context);

    return projectEAR;
}
```

Con todos los pasos realizados tendremos implementado el asistente que facilitará al usuario final la generación de la estructura de una aplicación UDA.

### 3.3 Generar snippets de patrones de presentación de UDA

Los snippets son componentes que permiten que se genere un fragmento de código, en este caso, el código de configuración de un patrón de UDA. Se utilizan de manera sencilla, simplemente se debe seleccionar el snippet con el ratón y arrastrarlo a un editor de código fuente.

En el Eclipse, dentro de la categoría UDA de la vista de Snippets, se muestra un snippet por cada componente de la capa de vista.

Para el desarrollo de los snippets se ha utilizado el punto de extensión de la clase *org.eclipse.wst.common.snippets.SnippetContributions*, que ofrece la capacidad de generar un listado de snippets agrupados en una categoría. Además posibilita pasar valores a las partes variables de estos fragmentos de código mediante una pantalla de entrada de datos.

A continuación se muestra un ejemplo con la generación del snippet del Mensaje.

Primeramente, debemos editar el fichero *plugin.xml* del proyecto *PluginUDA* y añadir el siguiente punto de extensión:

```
<extension point="org.eclipse.wst.common.snippets.SnippetContributions">
    <category
        label="UDA"
        description="Patrones de UDA"
        id="org.ejia.uda.components.category"
        initial_state="1"
        smallicon="icons/uda.gif"
        largeicon="icons/uda.gif">
        <item
            label="Mensajes"
            id="org.ejia.uda.components.item.messages"
            smallicon="icons/components/messages.png"
            largeicon="icons/components/messages.png"
            class="org.eclipse.wst.common.snippets.ui.DefaultSnippetInsertion">
            <description><![CDATA[Patrón Mensajes]]></description>
```

```

        <content>
            <![CDATA[
                $.rup_messages("${typeMessage}", {
                    title: ${propertyTitle},
                    message: ${propertyMessage}
                });
            ]]>
        </content>
        <variable description="Tipo de mensaje (msgError, msgConfirm,
            msgOK, msgAlert)" id="typeMessage"
            default="[typeMessage]"></variable>
        <variable description="Propiedad title" id="propertyTitle"
            default="[propertyTitle]"></variable>
        <variable description="Propiedad message" id="propertyMessage"
            default="[propertyMessage]"></variable>
    </item>
</category>
</extension>

```

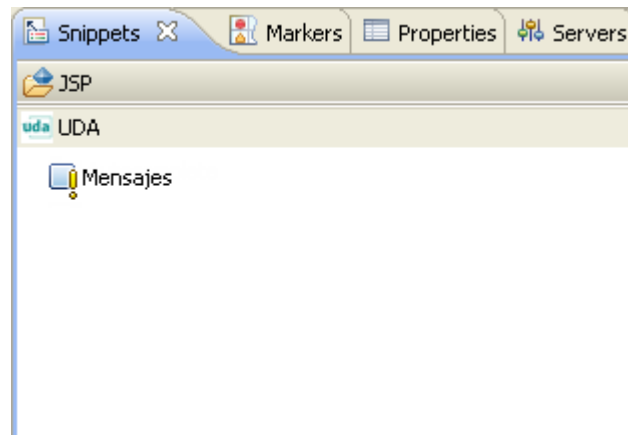
- **extension point:** Indica el punto de extensión de creación de snippets desde **SnippetContributions**.
- **category:** Define la categoría que englobará los varios snippets generados.
  - **name:** Nombre que se visualizará en la vista de Snippets de Eclipse.
  - **description:** Texto que describe la categoría (funciona como un tooltip).
  - **id:** Identificador único de la categoría.
  - **initial\_state:** Si la categoría se mostrará plegada o desplegada en la vista de snippets.
  - **smallicon:** Ruta de la imagen de la categoría cuando se presente de forma reducida en pantalla.
  - **largeicon:** Ruta de la imagen de la categoría cuando se presente de forma ampliada en pantalla.
- **item:** Snippet a generar en la categoría.
  - **label:** Nombre del snippet.
  - **id:** Identificador único del snippet.
  - **smallicon:** Ruta de la imagen del snippet cuando se presente de forma reducida en pantalla.
  - **largeicon:** Ruta de la imagen del snippet cuando se presente de forma ampliada en pantalla.
  - **class:** Se usa la clase *org.eclipse.wst.common.snippets.ui.DefaultSnippetInsertion* para generar el snippet.
  - **description:** Texto que describe el snippet (funciona como un tooltip).
  - **content:** Donde se inserta el fragmento de código que servirá de plantilla a la hora de generar el snippet.
  - **variable:** Si el código tiene alguna parte dinámica, se podrá referenciar mediante esta etiqueta. Se mostrará una ventana que pedirá el dato que se va a cambiar antes de pintar el código del snippet.

En la etiqueta content, una variable se referencia de la siguiente manera: *\${nombre de la variable}*.

- **description:** Comentario sobre la variable que aparecerá en la pantalla de introducción de datos antes de pintar el snippet arrastrado.

- **id:** Identificador único de esta variable.
- **default:** Valor por defecto que tendrá dicha variable.

El resultado final del snippet generado en el entorno Eclipse:



**Ilustración 19. Resultado del snippet generado.**

## 4 Bibliografía

- Documento de Ayuda de Eclipse (situada en el propio IDE).
- Eclipse: <http://www.eclipse.org/documentation/>
- Eclipse RCP: <http://www.vogella.de/articles/EclipseRCP/article.html>
- Plug-in en Eclipse: <http://www.vogella.de/articles/EclipsePlugIn/article.html>
- Desarrollando plug-ins en Eclipse: <http://www.ibm.com/developerworks/library/os-ecplug/>
- Puntos de extensión en Plugins: <http://www.vogella.de/articles/EclipseExtensionPoint/article.html>
- Preferencias de Eclipse: <http://www.vogella.de/articles/EclipsePreferences/article.html>
- Asistentes en Eclipse: <http://www.vogella.de/articles/EclipseWizards/article.html>
- Proyecto Feature: <http://www.vogella.de/articles/EclipseFeatureProject/article.html>
- Proyecto de Update Site:  
[http://wiki.eclipse.org/FAQ\\_How\\_do\\_I\\_create\\_an\\_update\\_site\\_%28site.xml%29%3F](http://wiki.eclipse.org/FAQ_How_do_I_create_an_update_site_%28site.xml%29%3F)