# eclipse en el soporte al desarrollo de software

Abel Gómez Llana [agomez@dsic.upv.es].

15-19 de Septiembre 2008.

Universidad Politécnica de Valencia. Facultad de Informática.




# Desarrollo avanzado de plug-ins

18 de Septiembre de 2008



eclipse

# Contenido

- Creación de una nueva consola de eclipse.
- Creación de un editor sencillo con coloreado de sintaxis.
- Creación de una hoja de preferencias.
- Creación de un *plug-in* de ayuda.
- Creación de un punto de extensión.
- Creación de una perspectiva.

# Creación de una consola de eclipse





# Creación de una consola

- Cuando eclipse se ejecuta en modo normal, las salidas estándar y de error no se muestran.
- Por tanto, ¿cuál es la interfaz estándar en Eclipse para mostrar mensajes textuales (información, depuración, etc.) al usuario? → Las vistas de consola.

# Creación de una consola

- La API para el uso de una consola en Eclipse se encuentra definida en el *plug-in* «*org.eclipse.ui.console*».
- En primer lugar, hemos de crear una nueva, y registrarla en el gestor de consolas:

```
MessageConsole console = new MessageConsole(consoleName, null);
ConsolePlugin.getDefault().getConsoleManager().
     addConsoles(new IConsole[]{console});
```

- A continuación, debemos de crear un *stream* donde poder escribir los mensajes que se mostrarán por consola. Una consola puede tener múltiples *streams* donde escribir.

```
MessageConsoleStream redStream = console.newMessageStream();
```

- Un *stream* puede tener asociado un determinado color, para poder diferencias distintos tipos de mensajes:

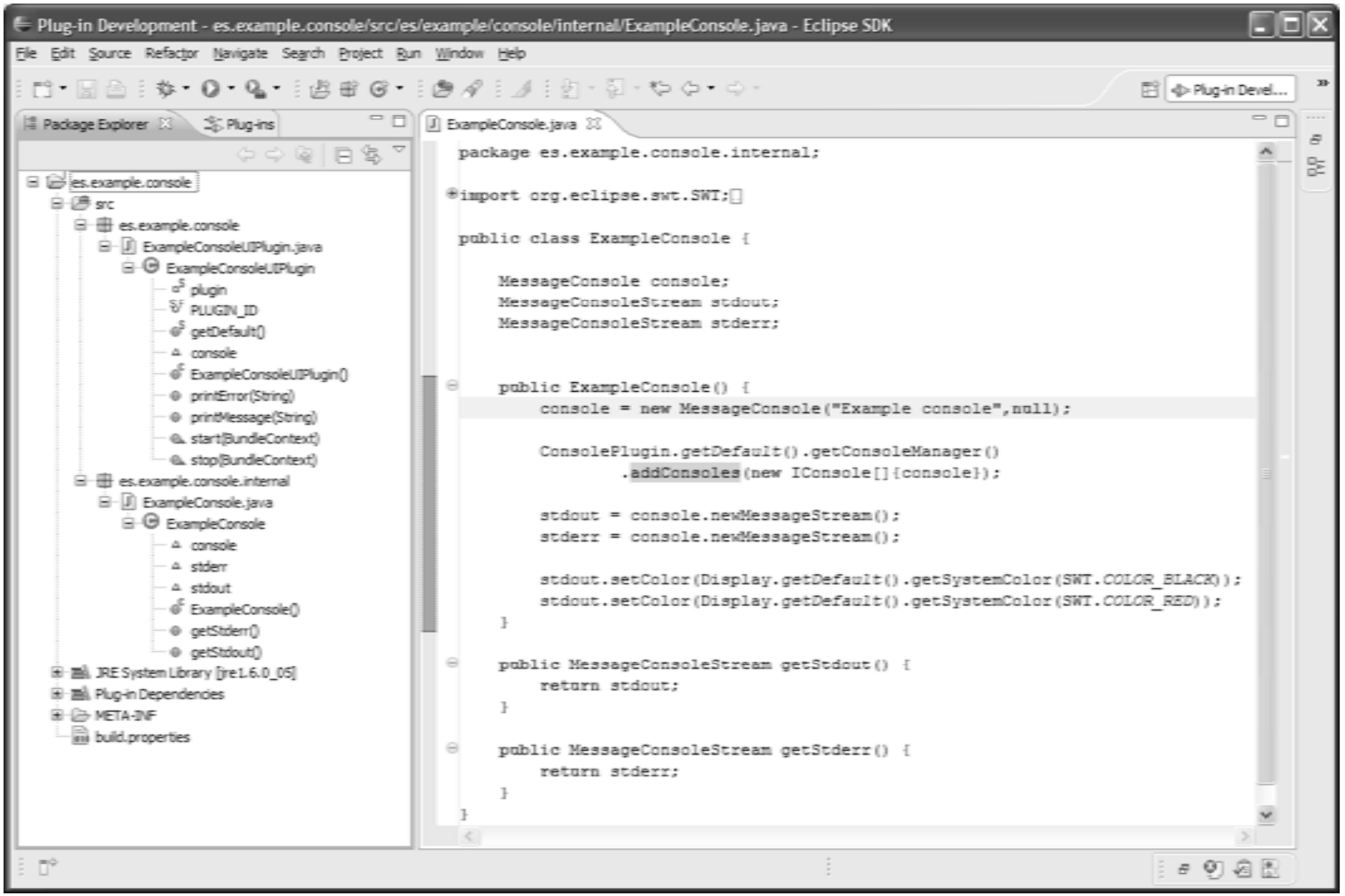
```
redStream.setColor(Display.
          getDefault().getSystemColor(SWT.COLOR_RED));
```

# Creación de una consola

- Si estamos construyendo un conjunto de *plug-ins* que serán distribuidos de forma conjunta, puede resultar interesante crear un *plug-in* para que albergue la consola de nuestra aplicación.
- En este caso, crearemos un *plug-in* al que se acceda de la siguiente manera para registrar los mensajes en la consola:

```
ExampleConsoleUIPlugin.getDefault().printMessage(“mensaje”);
ExampleConsoleUIPlugin.getDefault().printError(“mensaje”);
```

# Creación de una consola



# Creación de un editor de texto



# Creación de un editor de texto

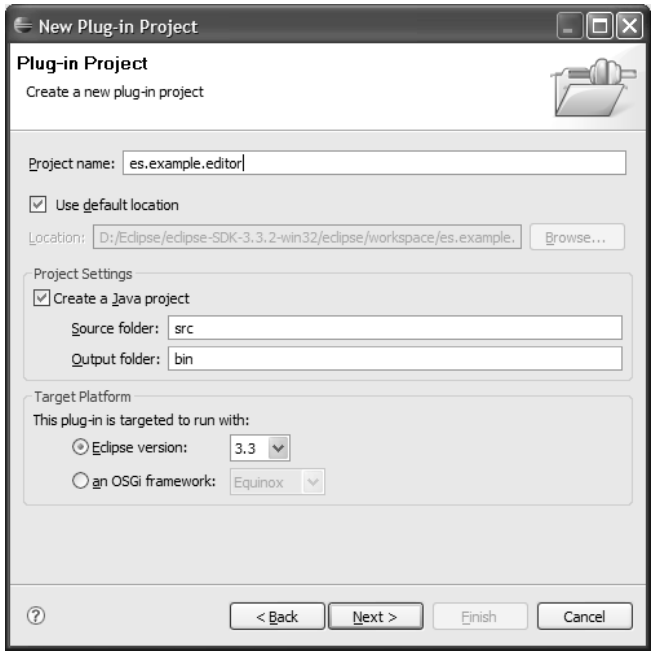
- Eclipse proporciona una rica API (aunque compleja) para la creación de editores textuales: *JFace Text*.
- Esta API proporciona funciones para poder crear editores con coloreado de sintaxis, marcado de errores, autocompletado de código, etc.
- Eclipse proporciona un asistente que permite crear un nuevo *plug-in* que defina un nuevo editor de texto.
- Por defecto, este asistente crea un editor de texto para archivos XML.
- El ejemplo de creación de un editor de texto sencillo que mostraremos a continuación no hará uso de este asistente.

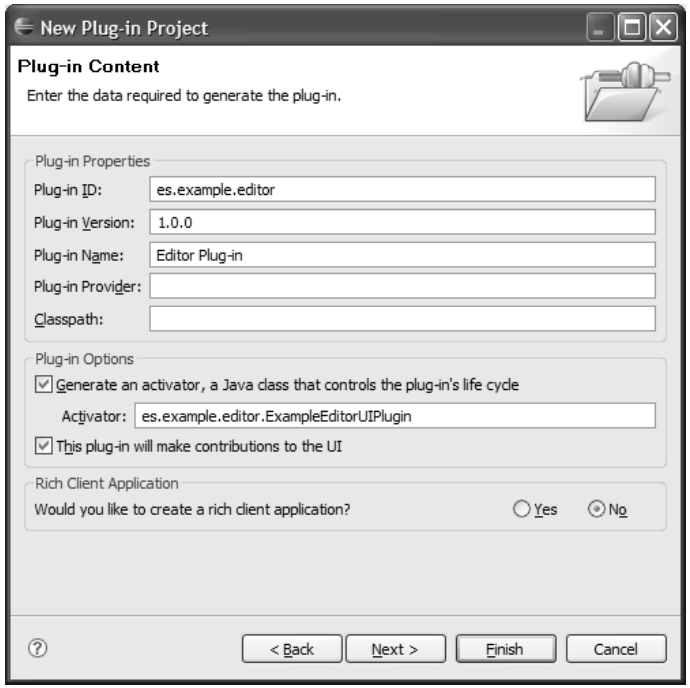
# Uso del punto de extensión de un nuevo editor



# Creación de un editor de texto

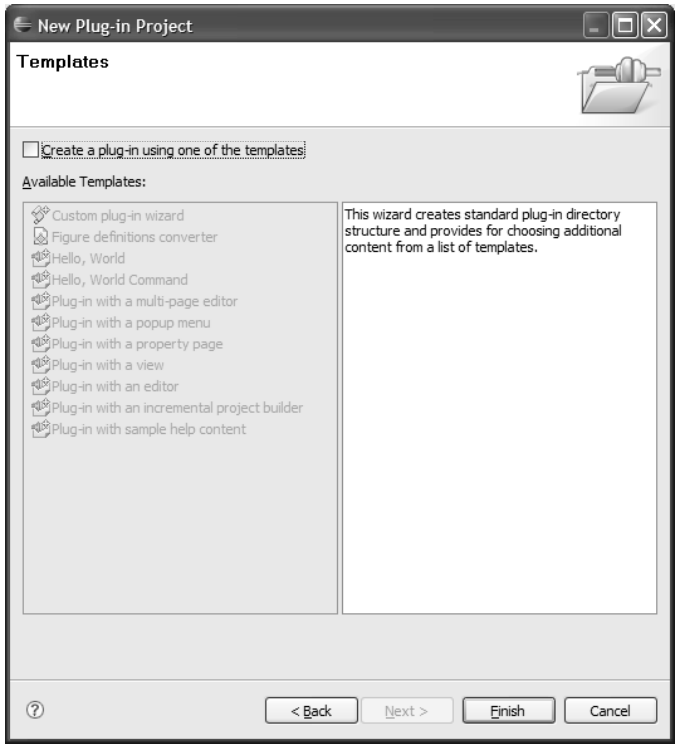
- En primer lugar, crearemos un nuevo proyecto de *plug-in* vacío.

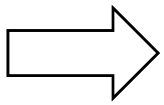


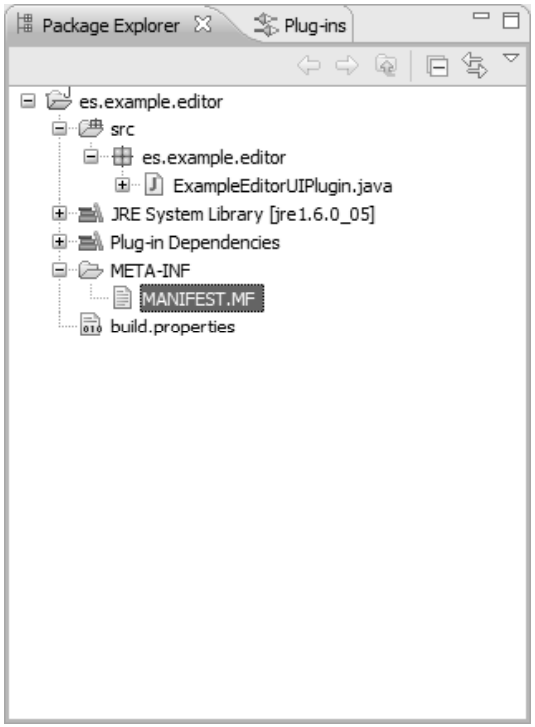



# Creación de un editor de texto

- En primer lugar, crearemos un nuevo proyecto de *plug-in* vacío.

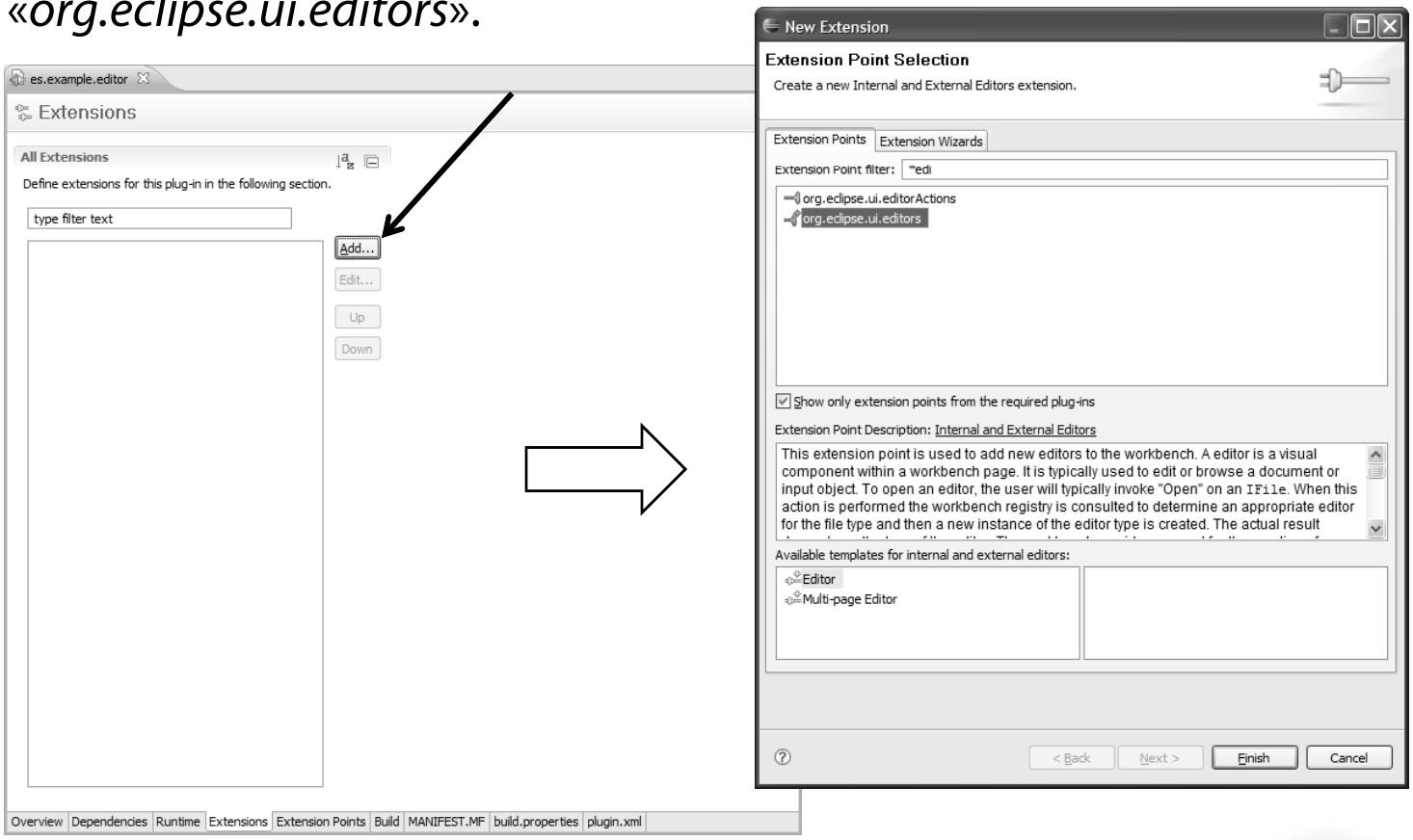




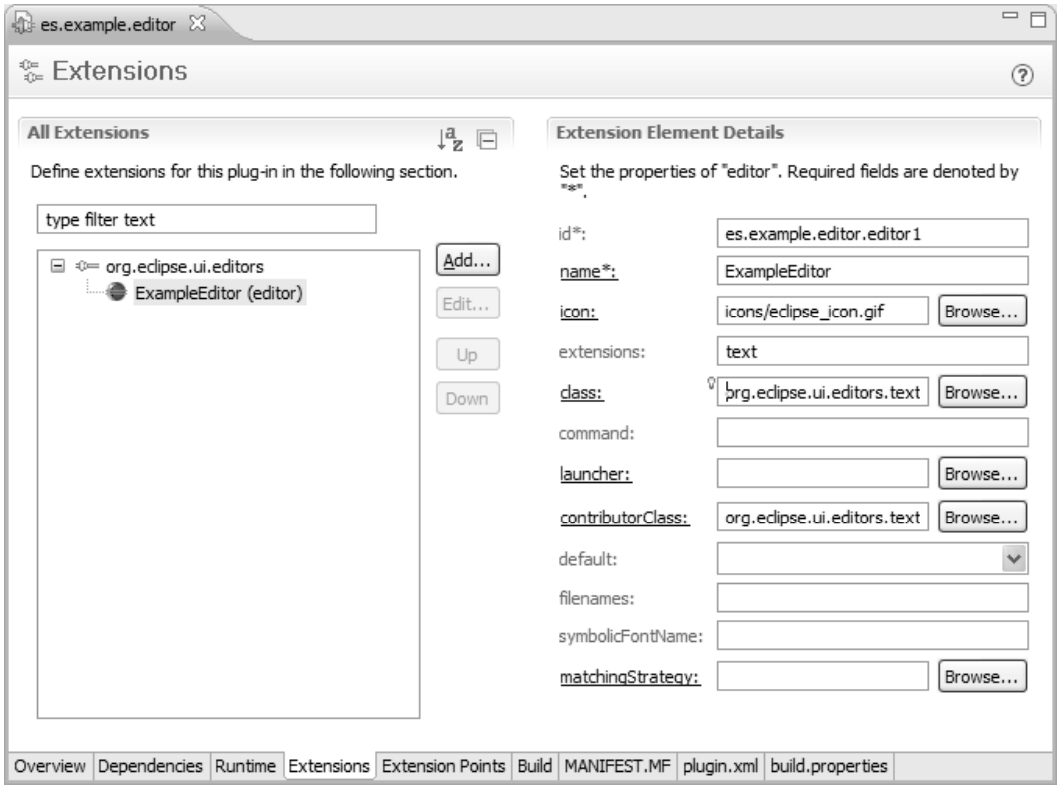
# Creación de un editor de texto

- A continuación, registraremos nuestro editor de texto (aún sin funcionalidad) invocando el punto de extensión «*org.eclipse.ui.editors*».



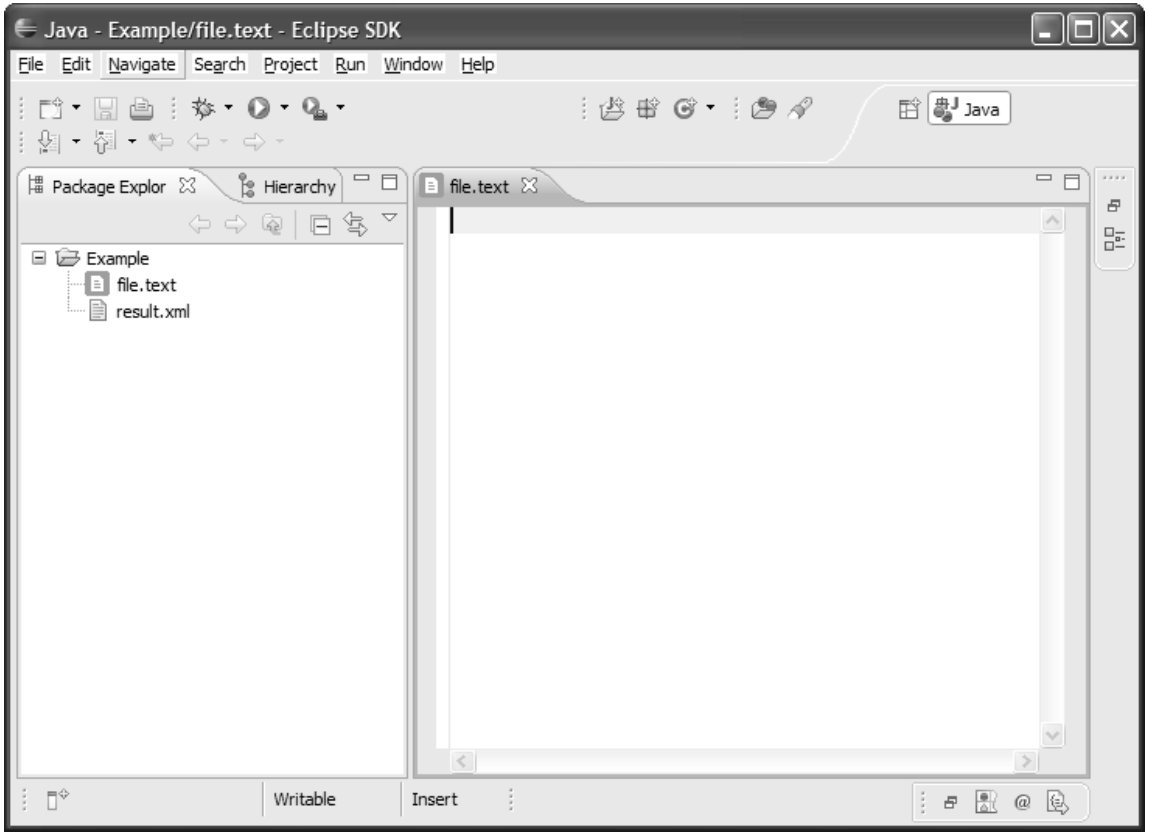
# Creación de un editor de texto

```
<extension
   point="org.eclipse.ui.editors">
    <editor
      class="org.eclipse.ui.editors.text.TextEditor"
      contributorClass="org.eclipse.ui.editors.text.TextEditorActionContributor"
      extensions="text"
      icon="icons/icon.gif"
      id="es.example.editor.editor1"
      name="ExampleEditor">
   </editor>
</extension>
```



# Creación de un editor de texto

- El *plug-in* que hemos creado asocia el editor de texto estándar de eclipse (*org.eclipse.ui.editors.text.TextEditor*) y el icono seleccionado a los archivos con extensión «**.text*».



# Creación de la implementación personalizada

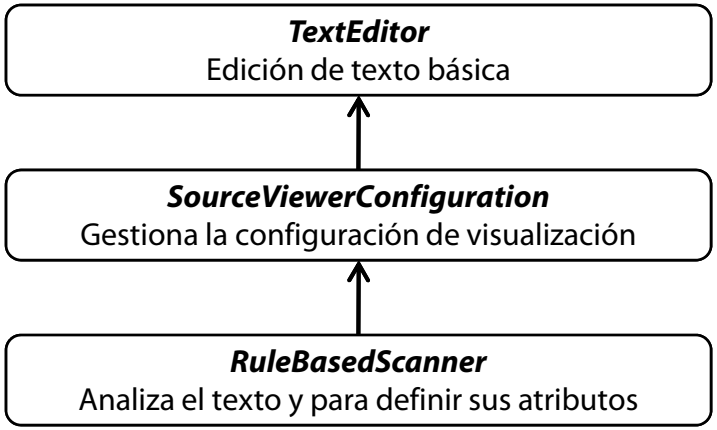


# Introducción a la API de JFace Text

- Eclipse emplea el modelo *daño, reparación, reconciliación (damage, repair, reconcile)*.
- Cada vez que el usuario cambia el documento, el *reconciliador* determina qué región ha sido invalidada, y cómo debe repararse:
  - *Daño* es el texto que debe redibujarse.
  - *Reparación* es el método empleado para redibujar el área dañada.
  - *Reconciliación* es el proceso de mantener la representación visual de un documento cada vez que se efectúa un cambio.
- Para personalizar el modelo de daño/reparación de nuestro editor, extenderemos la clase por defecto *TextEditor*, y la modificaremos para que emplee nuestra propia clase *SourceViewerConfiguration*.

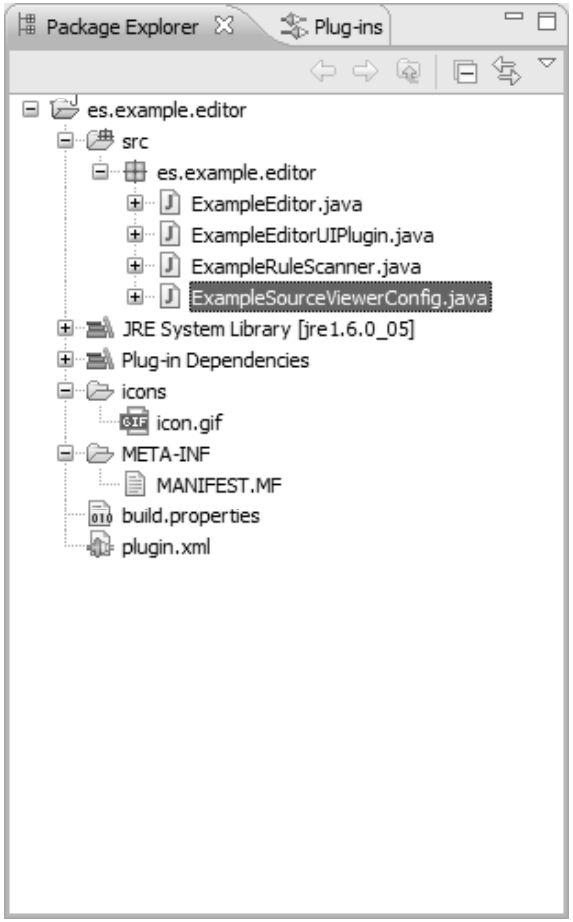
# Introducción a la API de JFace Text

- La clase *SourceViewerConfiguration* el clase central para configurar el editor con funcionalidad adicional (p.e., coloreado de sintaxis o autocompletado).
- Por defecto, esta clase no soporta coloreado de sintaxis, por ello, crearemos nuestra propia clase *SourceViewer* que herede de *SourceViewerConfiguration,* pudiendo sobreescribir los métodos que nos interesen.
- Nuestro *SourceViewer* hará uso de sus propias reglas para determinar las distintas partes del documento, y cómo deberán representarse.
- Podremos definir nuestras propias reglas de forma sencilla extendiendo la funcionalidad ofrecida por la clase *RuleBasedScanner*.



# Introducción a la API de JFace Text

- De esta manera, nuestro *plug-in* requerirá de tres clases adicionales:
  - *ExampleEditor* — Extiende la clase por defecto *TextEditor*. Cambiaremos el *SourceViewer* por defecto.
  - *ExampleRuleScanner* — Hereda de *RuleBasedScanner* y define las reglas para nuestro propio editor que se emplearán en la clase de *daño/reparación*.
  - *ExampleSourceViewerConfig* — Hereda de *SourceViewerConfiguration*, instanciará la regla de nuestro editor, y definirá la acción de reconciliación.



# Introducción a la API de JFace Text

- En primer lugar, debemos configurar nuestro editor para que emplee nuestro *SourceViewer:*

# Introducción a la API de JFace Text

- A continuación, debemos establecer el modelo *damage/repair/reconcile* sobreescribiendo el método *SourceViewerConfiguration.getPresentationReconciler():*

es.example.editor | ExampleEditor.java | ExampleRuleScanner.java | ExampleSourceViewerConfig

```
14
15 public class ExampleSourceViewerConfig extends SourceViewerConfiguration {
16
17     private static final Color DEFAULT_COLOR =
18         new Color(Display.getCurrent(), new RGB(0,0,0));
19
20     ExampleRuleScanner scanner;
21
22     protected ExampleRuleScanner getScanner() {
23         if (scanner == null) {
24             scanner = new ExampleRuleScanner();
25             scanner.setDefaultReturnToken(
26                 new Token(
27                         new TextAttribute(DEFAULT_COLOR)));
28         }
29         return scanner;
30     }
31
32     @Override
33     public IPresentationReconciler getPresentationReconciler(
34             ISourceViewer sourceViewer) {
35         PresentationReconciler reconciler = new PresentationReconciler();
36         DefaultDamagerRepairer dr = new DefaultDamagerRepairer(getScanner());
37         reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
38         reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);
39
40         return reconciler;
41     }
42
43 }
```

# Introducción a la API de JFace Text

- Por último, definimos las reglas que marcarán como se coloreará el texto de nuestro editor. En este caso, definiremos un color azul para algunas palabras clave de SQL, y color verde para comentarios:

es.example.editor | ExampleEditor.java | ExampleRuleScanner.java | ExampleSourceViewerConfig.java

```
14
15 public class ExampleRuleScanner extends RuleBasedScanner {
16
17     public static String[] keyWords = {
18         "CREATE", "DROP", "SCHEMA", "TABLE", "PRIMARY", "FOREIGN", "KEY",
19         "REFERENCES", "VARCHAR", "INTEGER", "DATE", "TIME", "NOT", "NULL"};
20
21
22     private static Color KEYWORDS_COLOR =
23         new Color(Display.getCurrent(), new RGB(0, 0, 200));
24
25     private static Color COMMENTS_COLOR =
26         new Color(Display.getCurrent(), new RGB(0, 200, 0));
27
28     public ExampleRuleScanner() {
29         IToken keywordToken = new Token(new TextAttribute(KEYWORDS_COLOR));
30         IToken commentToken = new Token(new TextAttribute(COMMENTS_COLOR));
31
32         IWordDetector wordDet = new IWordDetector() {
33             public boolean isWordStart(char c) {
34                 return Character.isJavaIdentifierStart(c);
35             }
36
37             public boolean isWordPart(char c) {
38                 return Character.isJavaIdentifierPart(c);
39             }
40         };
41
42         WordRule keyWordsRule = new WordRule(wordDet);
43
44         // add tokens for each reserved word
45         for (int i = 0; i < keyWords.length; i++) {
46             keyWordsRule.addWord(keyWords[i], keywordToken);
47         }
48
49         setRules(new IRule[] {
50                 keyWordsRule,
51                 new EndOfLineRule("--", commentToken)
52         });
53     }
54 }
55
```

# Introducción a la API de JFace Text

- En este punto, podremos ejecutar nuestro editor (debemos recordar actualizar el archivo de manifiesto, para apuntar a la nueva clase del editor):

ejemplo.text

```
-- ================================================================================
-- Nombre del esquema: Model
-- Creado en: 08/01/2008 13:16
-- ================================================================================

-- Creacion de tablas

DROP TABLE "Model"."Solicitud";
CREATE TABLE "Model"."Solicitud"
(
    "fk_Identificador_Acta"    VARCHAR NOT NULL,
    "fk_Identificador_Alumno"    VARCHAR NOT NULL,
    "fk_Identificador_Convocatoria"    VARCHAR NOT NULL,
    "fk_Identificador_Examen"    VARCHAR NOT NULL,
    "Identificador_Solicitud"    VARCHAR NOT NULL,
    "Están_Fotos_Disponibles"    VARCHAR(1) NOT NULL,
    "Está_Fotocopia_DNI_Disponible"    VARCHAR(1) NOT NULL,
    "Está_Certificado_Médico_Disponible"    VARCHAR(1) NOT NULL,
    "fk_Identificador_Resolución"    VARCHAR NOT NULL,
    PRIMARY KEY( "fk_Identificador_Convocatoria", "fk_Identificador_Examen", "Identificador_Soli
);

DROP TABLE "Model"."Resolución";
CREATE TABLE "Model"."Resolución"
(
    "fk_Identificador_Acta"    VARCHAR NOT NULL,
    "fk_Identificador_Convocatoria"    VARCHAR NOT NULL,
    "Identificador_Resolución"    VARCHAR NOT NULL,
    "Nota"    VARCHAR NOT NULL,
    "Es_Convalidación"    VARCHAR(1) NOT NULL,
    "Es_Nota"    VARCHAR(1) NOT NULL,
    PRIMARY KEY( "fk_Identificador_Acta", "fk_Identificador_Convocatoria", "Identificador_Resolu
);

DROP TABLE "Model"."Examen";
CREATE TABLE "Model"."Examen"
(
    "fk_Identificador_Asignatura_para_Titulación"    VARCHAR NOT NULL,
    "fk_Identificador_Convocatoria"    VARCHAR NOT NULL,
```

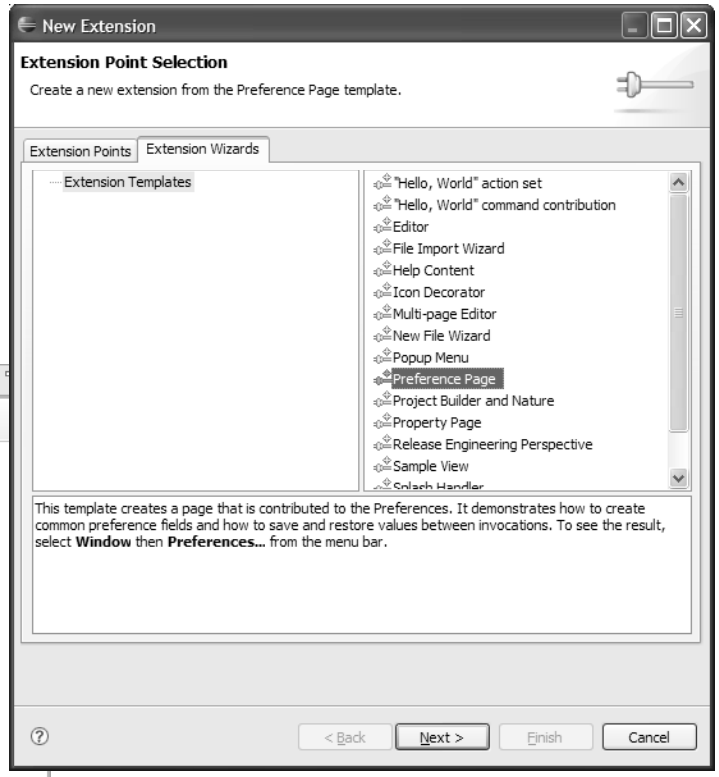
# Creación de una hoja de preferencias

# Creación de una hoja de preferencias

- Cada *plug-in*, dispone de un elemento *IPreferenceStore*, al que se accede mediante el método *AbstractUIPlugin.getPreferenceStore()*.
- *IPreferenceStore* es una interfaz que representa una tabla que mapea un valor (*boolean, double, float, int, long, String*) a una determinada clave (String).
- Una hoja de preferencias es la interfaz por defecto que Eclipse proporciona para que el desarrollador establezca los mecanismos de modificación de los valores del elemento *IPreferenceStore*, y que el usuario pueda establecer sus preferencias
- Como para los principales tipos de *plug-ins* Eclipse proporciona un asistente para crear una hoja de preferencias.
- En este caso, vamos a crear una hoja de preferencias que nos permita configurar la consola creada anteriormente.
- De esta manera, el usuario podrá determinar qué tipos de mensajes desea que se muestren por la consola (la consola muestra mensajes normales, de *debug* y de error).

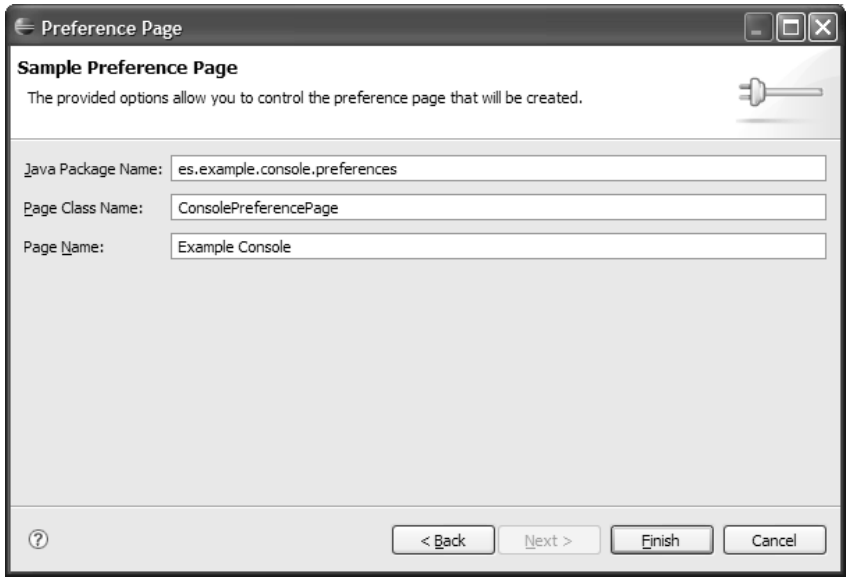
# Creación de una hoja de preferencias

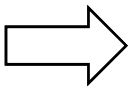
- En este caso, puesto que las preferencias son de la propia consola, definiremos las clases para la hoja de preferencias dentro del mismo *plug-in* de la consola.
- Para invocar un asistente para añadir funcionalidad a un *plug-in* existente, deberemos seleccionar el botón «*Add…*» en la pestaña «*Extensions*» en el editor del archivo de manifiesto de un *plug-in*.

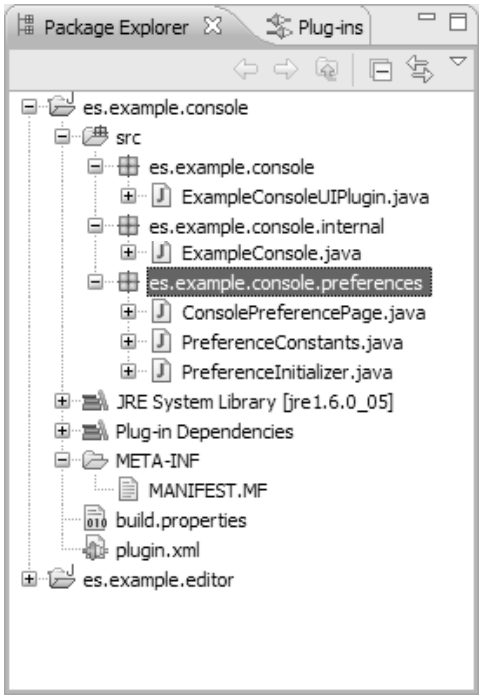



# Creación de una hoja de preferencias

- Para la creación de una hoja de preferencias, basta con indicar el nombre de la clase para la nueva hoja de preferencias, el nombre del paquete donde se colocará, y el nombre de la hoja de preferencias.



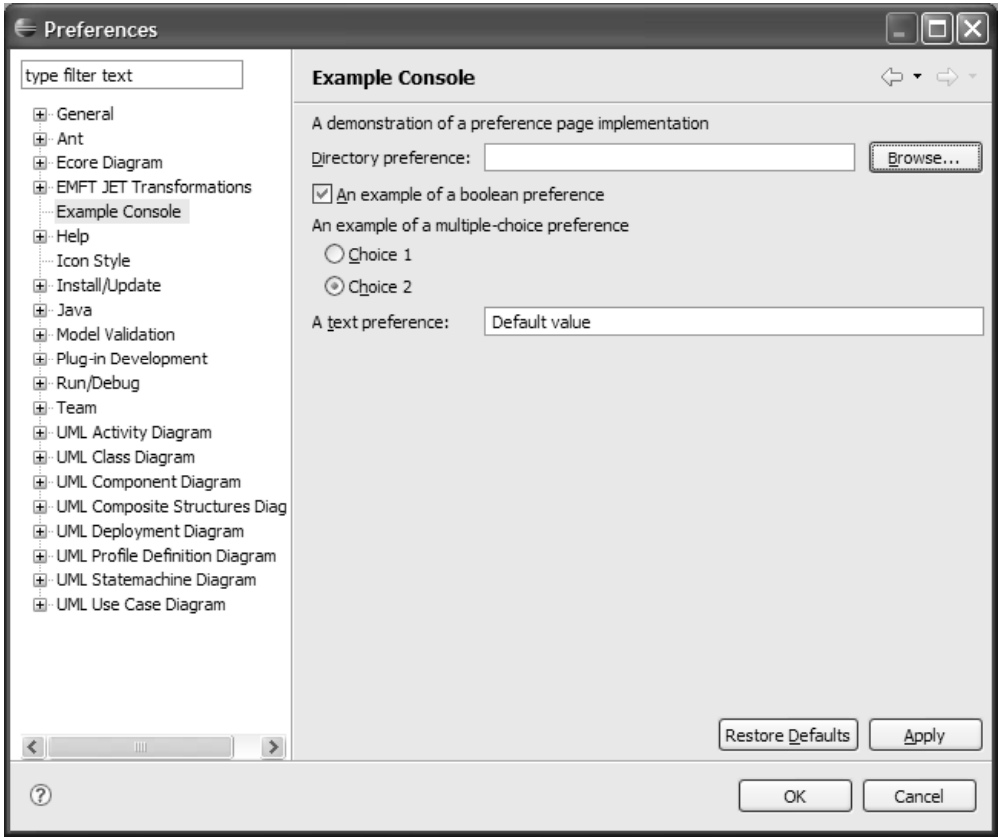


# Creación de una hoja de preferencias

- Por defecto, el asistente crea tres clases:
  - *ConsolePreferencePage* — Es la clase que implementa la hoja de propiedades. La clase creada por defecto hereda de *FieldEditorPreferencePage*, que es un tipo especial de hoja de propiedades que:
    - Proporciona constructores sencillos para crear los controles SWT más comúnmente usados en hojas de propiedades.
    - Proporciona una implementación para guardar/recuperar los valores de estos controles al presionar los botones de *aplicar* y *aceptar*.
  - *PreferenceConstants*— En esta clase se establecen las constantes (*Strings*), que determinan las claves empleadas en el elemento *IPreferenceStore*.
  - *PreferenceInitializer* — Es la clase que determina los valores por defecto de *IPreferenceStore*. Estos se establecen la primera vez que se ejecuta el *plug-in*, o cuando el usuario pulsa el botón «*Restore Defaults*».

# Creación de una hoja de preferencias

- Si ejecutamos el *plug-in* con la implementación por defecto, tendremos la siguiente hoja de ejemplo:



# Creación de una hoja de preferencias

- Para personalizar los controles que queremos emplear en nuestra hoja de ejemplo, en primer lugar deberemos determinar cuáles serán las preferencias que vamos a guardar, y actualizar la clase *PreferenceConstants* con los valores apropiados.

ConsolePreferencePage.java | PreferenceConstants.java

```
package es.example.console.preferences;

/**
 * Constant definitions for plug-in preferences
 */
public class PreferenceConstants {

    public static final String P_ENABLE_STDOUT = "enableStdout";

    public static final String P_ENABLE_STDERR = "enableStderr";

    public static final String P_ENABLE_STDDBG = "enableStddbg";

}
```

# Creación de una hoja de preferencias

- Después, deberemos modificar el método *ConsolePreferencePage.createFieldEditors()*, creando los controles adecuados dependiendo del tipo de valor que deseemos guardar, y asociándolo a una clave definida entre las constantes:

ConsolePreferencePage.java | PreferenceConstants.java

```
/**
 * Creates the field editors. Field editors are abstractions of
 * the common GUI blocks needed to manipulate various types
 * of preferences. Each field editor knows how to save and
 * restore itself.
 */
public void createFieldEditors() {
    addField(
            new BooleanFieldEditor(
                    PreferenceConstants.P_ENABLE_STDOUT,
                    "Enable std&out",
                    getFieldEditorParent()));
    addField(
            new BooleanFieldEditor(
                    PreferenceConstants.P_ENABLE_STDERR,
                    "Enable std&err",
                    getFieldEditorParent()));
    addField(
            new BooleanFieldEditor(
                    PreferenceConstants.P_ENABLE_STDDBG,
                    "Enable std&dbg",
                    getFieldEditorParent()));
}
```

# Creación de una hoja de preferencias

- También, se debe ajustar la clase *PreferenceInitializer* para que establezca los valores iniciales de forma correcta.

# Creación de una hoja de preferencias

- Por último, se deben modificar los métodos que escriben en la consola para que consulten si se debe escribir o no según las preferencias.

ConsolePreferencePage.java | PreferenceConstants.java | PreferenceInitializer.java | ExampleConsoleUIPlugin.java

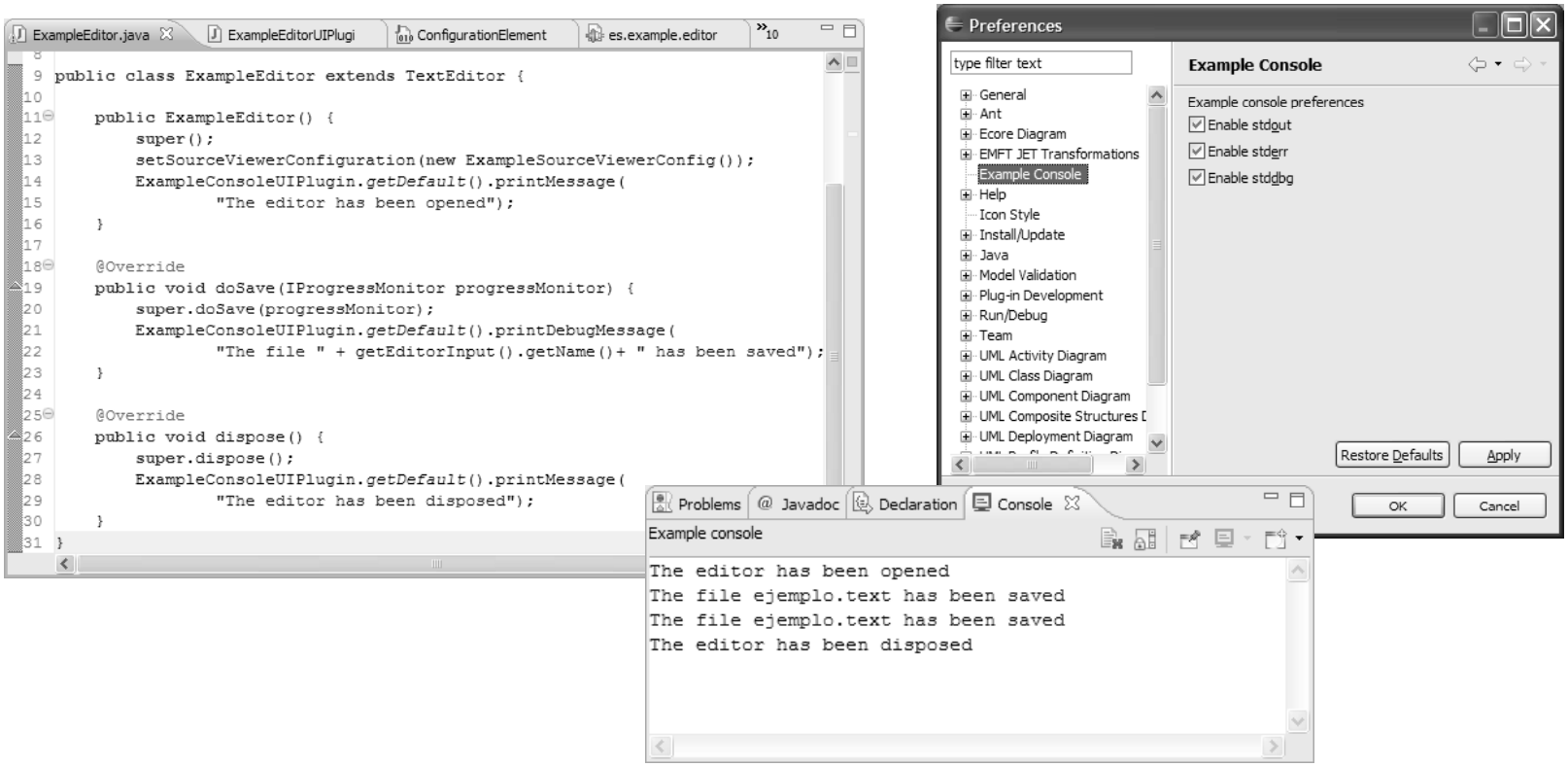
```
public void printError(String message) {
    if (getPreferenceStore().getBoolean(PreferenceConstants.P_ENABLE_STDERR))
        console.getStderr().println(message);
}

public void printMessage(String message) {
    if (getPreferenceStore().getBoolean(PreferenceConstants.P_ENABLE_STDOUT))
        console.getStdout().println(message);
}

public void printDebugMessage(String message) {
    if (getPreferenceStore().getBoolean(PreferenceConstants.P_ENABLE_STDDBG))
        console.getStddebg().println(message);
}
```

# Creación de una hoja de preferencias

- Podemos probar la funcionalidad de las preferencias de la consola, si por ejemplo modificamos la clase del editor para que nos avise cuando el editor se abre, se cierra, y cuando un archivo es salvado:



# Creación de una hoja de preferencias con controles avanzados

# Creación de una hoja de preferencias

- En el ejemplo visto anteriormente, una hoja de preferencias extiende a la clase *FieldEditorPreferencePage* porque esta proporciona constructores para crear los controles más comunes (casillas de texto, botones de radio, *checkboxes*, selectores de directorio, etc.). ¿Pero qué ocurre si necesitamos crear una hoja con controles avanzados (tablas, *layouts* en columnas, etc.)
- En ese caso, la clase de nuestra hoja de preferencias debe implementar la interfaz IWorkbenchPreferencePage. Para simplificar la implementación, nuestra clase puede heredar directamente de la clase *PreferencePage* (o sobreescribir los métodos necesarios de *FieldEditorPreferencePage* y heredar de ésta).
- Si se hereda de *PreferencePage* perderemos toda esa funcionalidad que añade *FieldEditorPreferencePage*.

# Creación de una hoja de preferencias

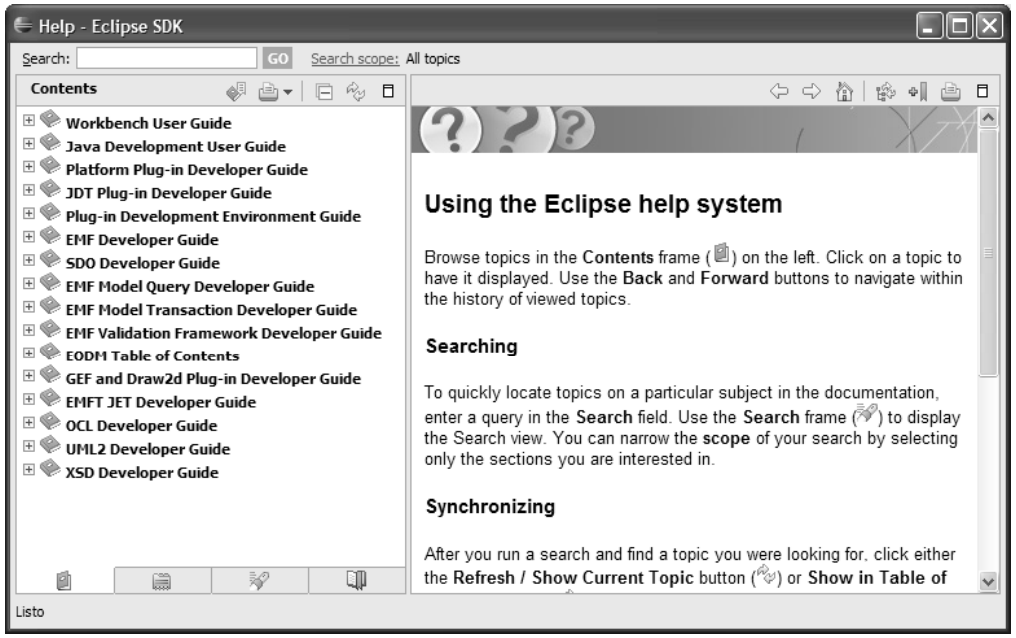
- Cuando se crea una hoja de propiedades que extiende *PreferencePages*, la clase hija:
  - Debe implementar el método abstracto *createControl()*.
  - Se recomienda implementar el método *doComputeSize()*.
  - Puede (y debería para que la página funcione correctamente, ya que en estos métodos es donde se carga/resetea/salva el estado de las preferencias de/hacia los controles) sobreescribir los métodos *performOk()*, *performApply()*, *performDefaults()*, *performCancel()*, y *performHelp()*.
  - Las subclases también pueden emplear el método *noDefaultAndApplyButton()* permite eliminar los botones «*Default*» y «*Apply*».
- En caso de que la modificación de un valor deba reflejarse inmediatamente en el estado de otro control (puede ser la propia hoja de preferencia, u otro control completamente diferente, como una vista, o un editor) se puede implementar un listener de la siguiente manera:
  - El segundo control debe tener una clase que implemente la interfaz *IPropertyChangeListener* y el método *propertyChange(…)*. Este será el método que será invocado al detectar un cambio en las preferencias.
  - El segundo control debe registrar el *listener* en el *IPreferenceStore* correspondiente:

*PluginClass.getDefault().getPreferenceStore.addPropertyChangeListener(miListener).*

# Creación de un *plug-in* de ayuda

# Ayuda en Eclipse

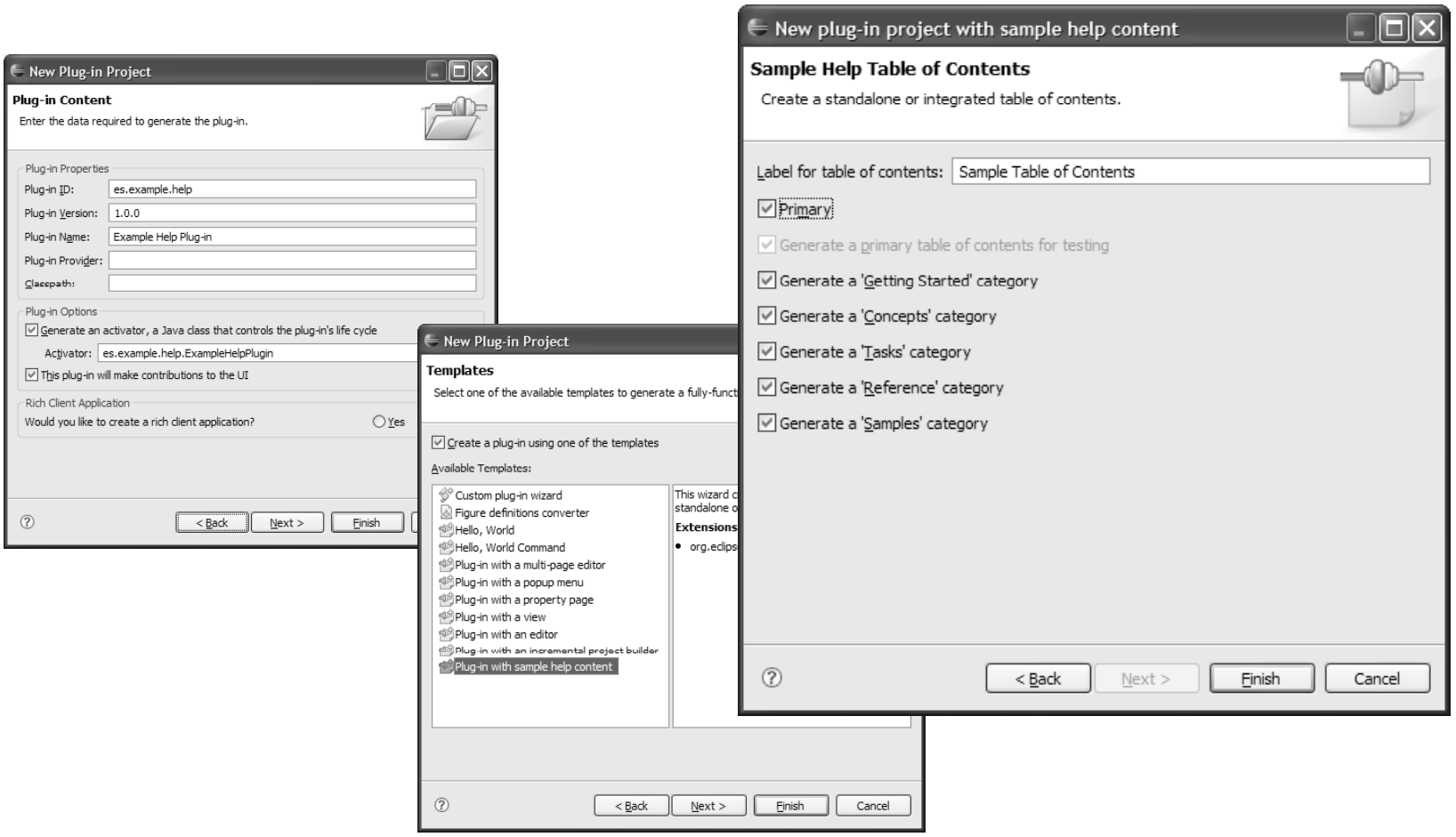
- Eclipse proporciona también un sistema centralizado y extensible de ayuda por medio de puntos de extensión.
- Cada *plug-in* puede contribuir al centro de ayuda de Eclipse con su propio contenido.
- El centro de ayuda se ejecuta en eclipse dentro del navegador web del sistema, realizando las peticiones sobre un servidor del propio Eclipse (un servidor *Apache Tomcat* incrustado en Eclipse).



# Ayuda en Eclipse

- Crear un nuevo *plug-in* de ayuda para Eclipse es extremadamente sencillo gracias a los habituales asistentes de creación de nuevo *plug-in*, aunque el proceso de preparación puede resultar tedioso.
- Los documentos de ayuda son documentos HTML ordinarios, siendo un sistema simple y fácilmente entendible.
- Además, al tratarse de HTML estándar, se puede emplear cualquier otro editor externo a gusto del desarrollador.
- Con referencia a este tipo de *plug-ins* se recomienda que estén aislados del resto de *plug-ins* que conformen nuestra herramienta, no incluyendo mas que la funcionalidad necesaria.
- Para la creación de un nuevo proyecto de ayuda, se procede de la forma habitual (*New → Project; Plug-in Project*).

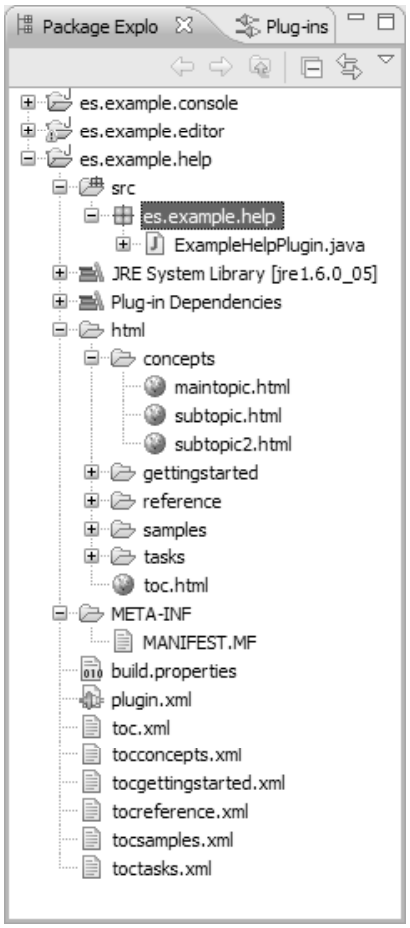
# Ayuda en Eclipse



# Ayuda en Eclipse

Por defecto, al crear un plug-in de ayuda de ejemplo, se crean los siguientes elementos:

- Archivos de manifiesto (*plugin.xml*, *MANIFEST.MF*)
- Carpeta «*src/*», con la clase activadora por defecto.
- Carpeta «*html/*», donde se albergan los documentos HTML de cada una de las páginas de nuestro *plug-in* de ayuda.
- Archivos «*toc*.xml*», archivos XML con la tabla de contenido del *plug-in*.



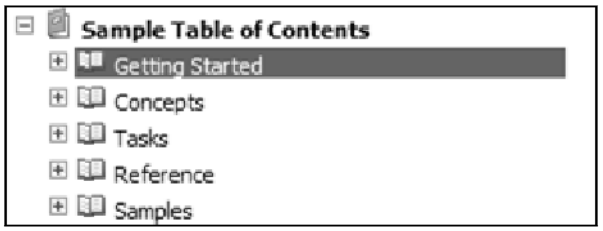
# Ayuda en Eclipse

- Como se observa en el archivo de manifiesto, de los archivos «*toc*.xml*», el fichero «*toc.xml*» es el denominado «*primario*».
- Este es el que se mostrará en primer nivel de la ventana de ayuda de Eclipse.

es.example.help

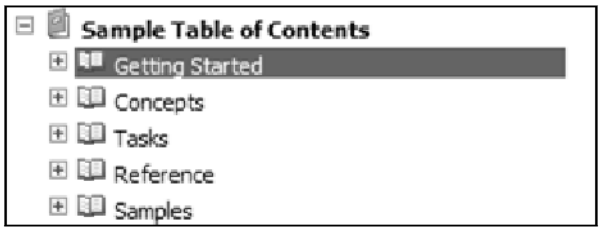
```
1<?xml version="1.0" encoding="UTF-8"?>
2<?eclipse version="3.2"?>
3<plugin>
4
5   <extension
6         point="org.eclipse.help.toc">
7      <toc
8            file="toc.xml"
9            primary="true">
10     </toc>
11     <toc
12           file="tocconcepts.xml">
13     </toc>
14     <toc
15           file="tocgettingstarted.xml">
16     </toc>
17     <toc
18           file="tocreference.xml">
19     </toc>
20     <toc
21           file="tocsamples.xml">
22     </toc>
23     <toc
24           file="toctasks.xml">
25     </toc>
26  </extension>
27
28</plugin>
29
```

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | plugin.xml



# Ayuda en Eclipse

- La etiqueta <*toc*> representa el elemento raíz de una tabla de contenido.
- El atributo *topic* es un enlace a un archivo HTML que describe el contenido del elemento TOC.
- El atributo *label* es un título breve para el elemento TOC, que sea legible para los humanos.

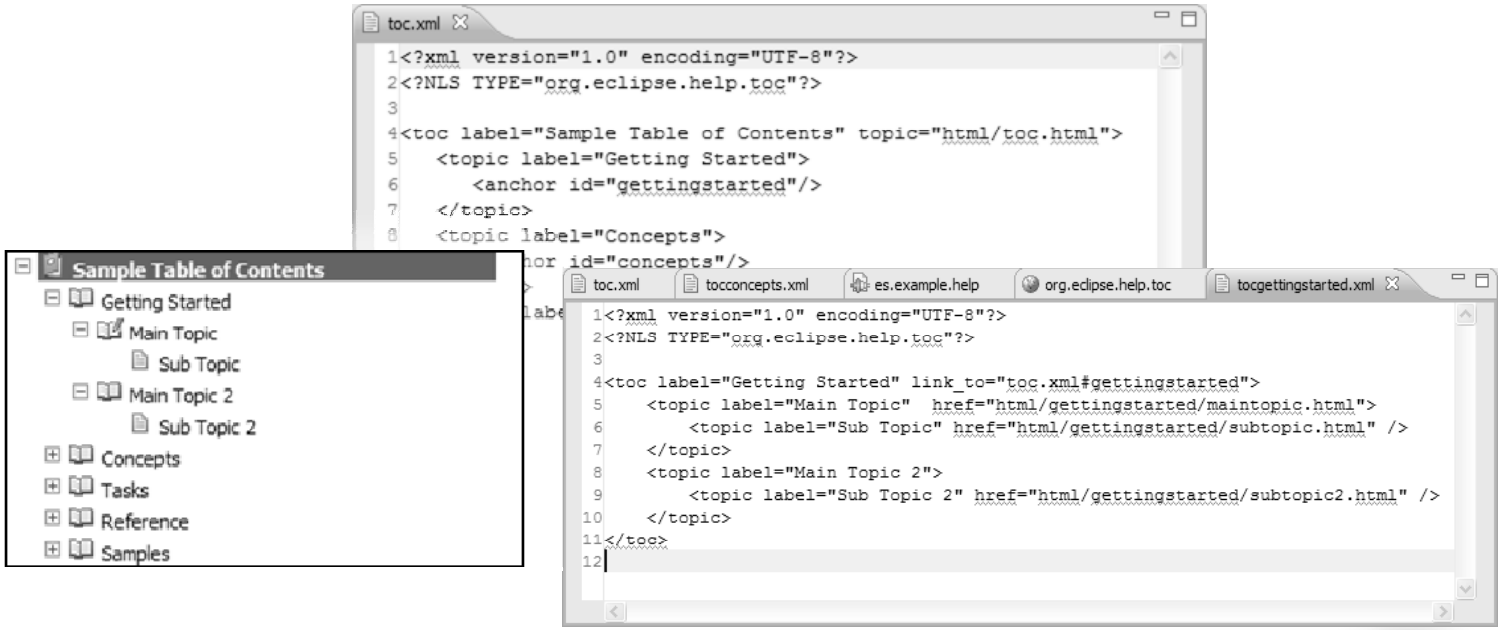



toc.xml

```
1<?xml version="1.0" encoding="UTF-8"?>
2<?NLS TYPE="org.eclipse.help.toc"?>
3
4<toc label="Sample Table of Contents" topic="html/toc.html">
5   <topic label="Getting Started">
6      <anchor id="gettingstarted"/>
7   </topic>
8   <topic label="Concepts">
9      <anchor id="concepts"/>
10  </topic>
11  <topic label="Tasks">
12     <anchor id="tasks"/>
13  </topic>
14  <topic label="Reference">
15     <anchor id="reference"/>
16  </topic>
17  <topic label="Samples">
18     <anchor id="samples"/>
19  </topic>
20</toc>
21
```

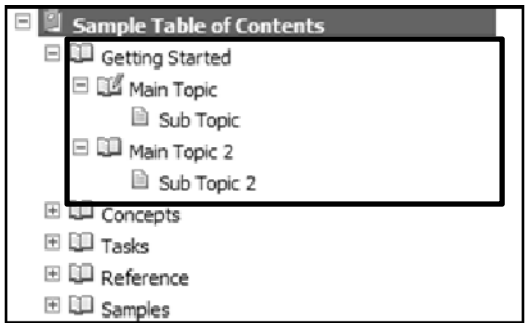
# Ayuda en Eclipse

- La etiqueta <*topic*> marca un determinado elemento del *toc*.
- Al igual que el elemento <*toc*>, dispone de una etiqueta.
- El elemento <*anchor*> define un identificador que será enlazado desde otro elemento <*toc*> definido en un archivo separado. Esto permite jerarquizar y anidar distintas tablas de contenido en diferentes ficheros.



# Ayuda en Eclipse

- Como ejemplo, el archivo «*tocgettingstarted.xml*» es enlazado por la tabla de contenido «*toc.xml*».
- Este archivo define una estructura similar al archivo anterior.
- A resaltar encontramos los atributos *href*, que enlazan un determinado *topic* con un archivo HTML existente, y el atributo *link_to*, que enlaza con el *anchor* definido en la tabla de contenido de primer nivel. De esta manera se establece que esta tabla se anida dentro de la de primer nivel, tal y como se observa en el menú de ayuda.




toc.xml | tocconcepts.xml | es.example.help | org.eclipse.help.toc | tocgettingstarted.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>

<toc label="Getting Started" link_to="toc.xml#gettingstarted">
    <topic label="Main Topic"  href="html/gettingstarted/maintopic.html">
        <topic label="Sub Topic" href="html/gettingstarted/subtopic.html" />
    </topic>
    <topic label="Main Topic 2">
        <topic label="Sub Topic 2" href="html/gettingstarted/subtopic2.html" />
    </topic>
</toc>
```

# Ejecución de un *Infocenter*



# Ejecución de un *Infocenter*

- El sistema de ayuda de Eclipse puede accederse desde una intranet o incluso Internet.
- Para ello, puede configurarse Eclipse únicamente como servidor web, sin que sea necesario lanzar el entorno por completo.
- Este servidor web se denomina *Infocenter* en la documentación de eclipse.
- En su invocación, permite especificar numerosos argumentos, como el puerto donde se desean escuchar las peticiones. La lista completa de opciones se encuentra en la documentación de eclipse http://help.eclipse.org/help33/topic/org.eclipse.platform.doc.isv/guide/ua_help_setup.htm).

# Ejecución de un *Infocenter*

- La creación de un *Infocenter* es sencilla:
  - Se ha de descargar la plataforma básica de eclipse (*Eclipse Platform Runtime Binary*).
  - Se debe descomprimir la plataforma básica de eclipse. Esto proporcionará todos los binarios mínimos necesarios, incluido el sistema de ayuda. (El conjunto mínimo de *plug-ins* requeridos es: *org.apache.lucene, org.eclipse.core.runtime, org.eclipse.help, org.eclipse.help.appserver, org.eclipse.help.base, org.eclipse.help.webapp, org.eclipse.osgi, org.eclipse.tomcat, org.eclipse.update.configurator*)
  - Se deben de incluir los *plug-ins* propios de ayuda en la carpeta de plug-ins.
- El *Infocenter* se ejecuta mediante el comando

```
java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base_[version].jar
org.eclipse.help.standalone.Infocenter -command start -eclipsehome d:\myApp\eclipse
                                        -port 8081
```

- Para detener el *Infocenter* se ejecutará:

```
java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base_[version].jar
     org.eclipse.help.standalone.Infocenter -command shutdown -eclipsehome
                                    d:\myApp\eclipse
```

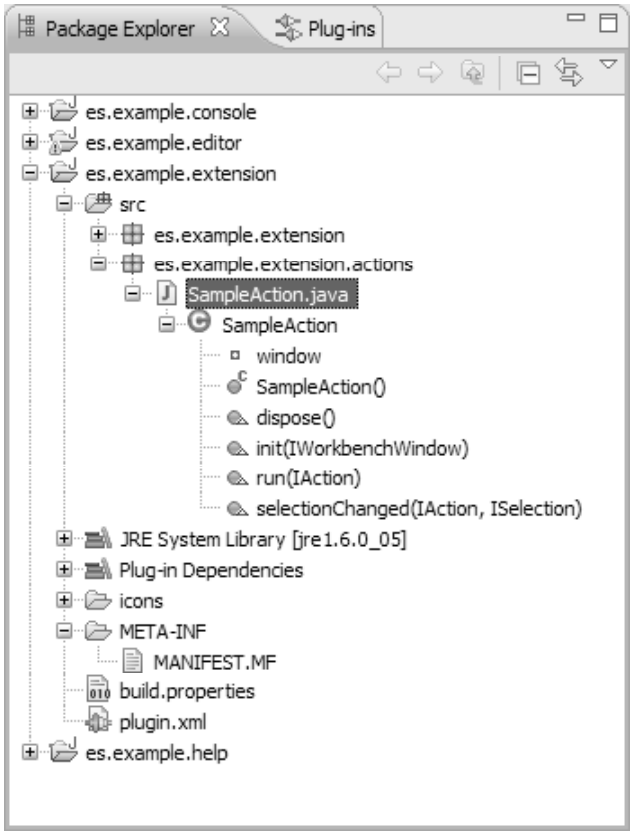
# Creación de un punto de extensión

# Puntos de extensión

- Como se ha visto anteriormente, un *punto de extensión* es una forma de comunicación entre dos *plug-ins*.
- Fundamentalmente, permite que un plugin *A*, haga pública una interfaz a la que se conectará cualquier otro *plug-in (B)*.
- A diferencia de la mera dependencia entre *plug-ins*, donde el *plug-in* B hace uso del código de A, sin que este último tenga control sobre éste, los puntos de extensión permiten al *plug-in* A conocer a B, y ejecutar código de este (puesto que A determina la interfaz que éste debe implementar).
- A continuación veremos dos ejemplos de puntos de extensión.

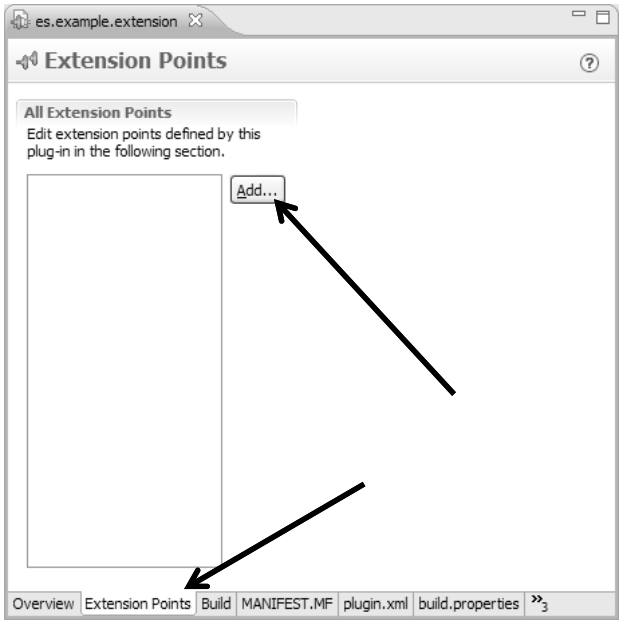
# Puntos de extensión

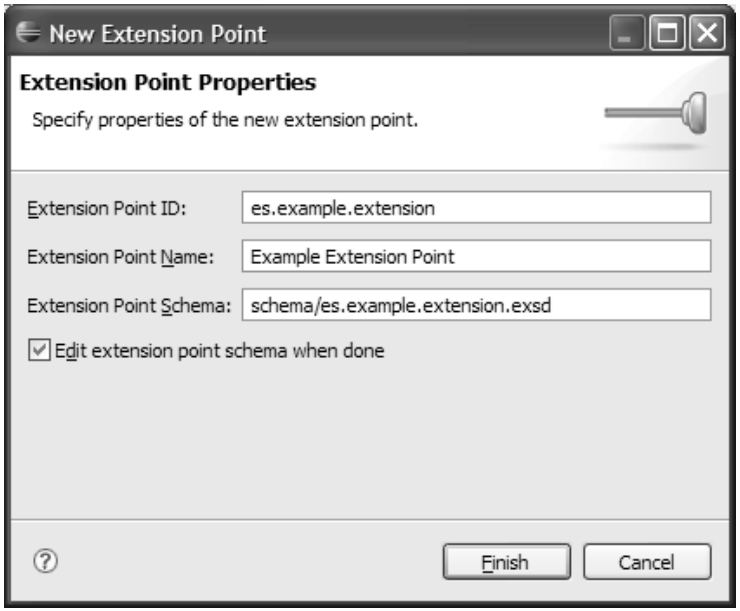
- En primer lugar, vamos a crear un sencillo *plug-in* de ejemplo empleando el asistente de creación de un proyecto «Hola mundo!».
- Este tipo de *plug-ins* simplemente contribuyen un botón en la barra de iconos de eclipse, que al ser pulsado, ejecuta un método *run()*. Esto nos permitirá invocar la funcionalidad del *plug-in* de forma sencilla.



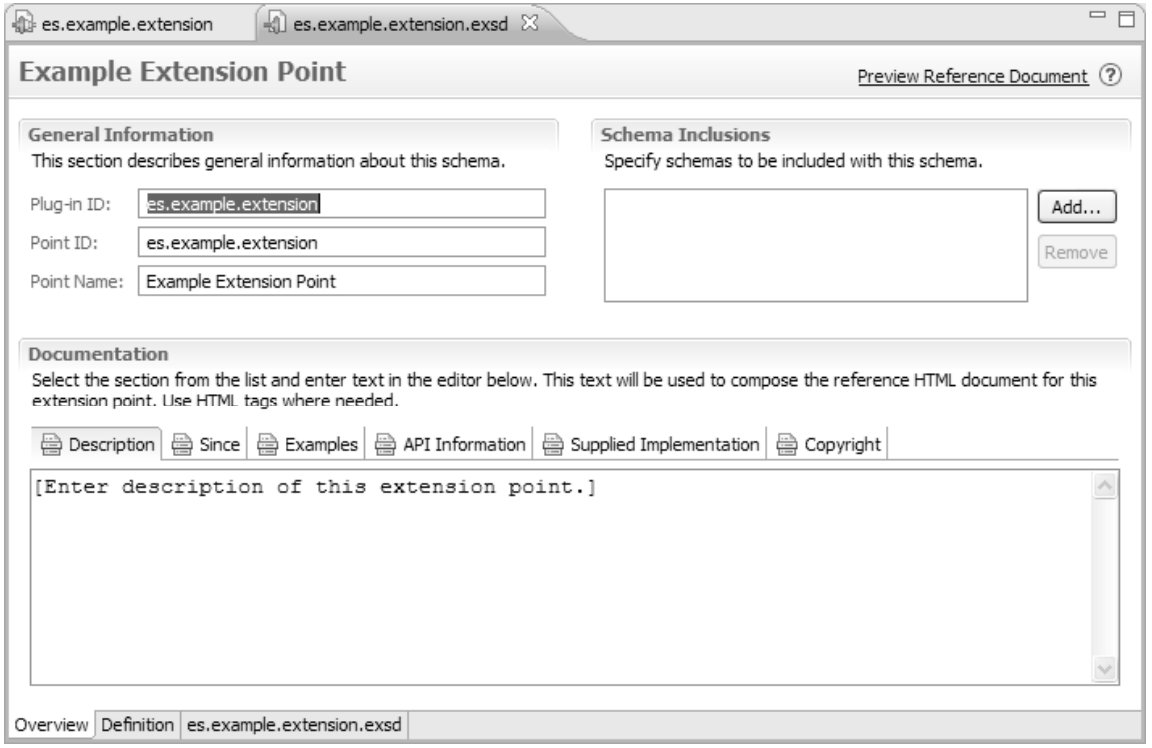
# Puntos de extensión

- Para añadir un punto de extensión, empleamos el botón «*Add…*» de la pestaña «*Extension Points*» en el editor de archivos de manifiesto.



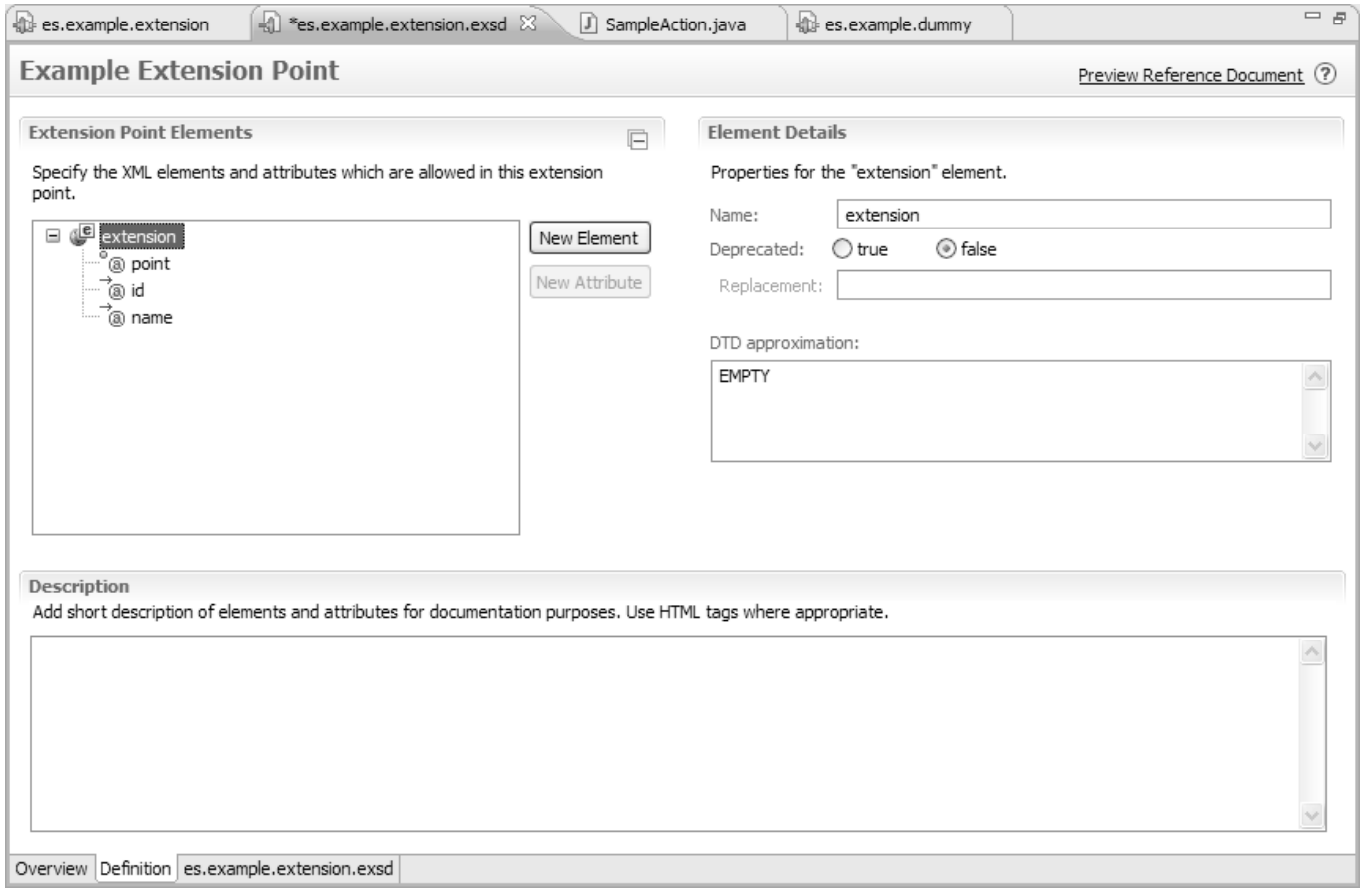



# Puntos de extensión



# Puntos de extensión

- Esquema del *punto de extensión* por defecto:

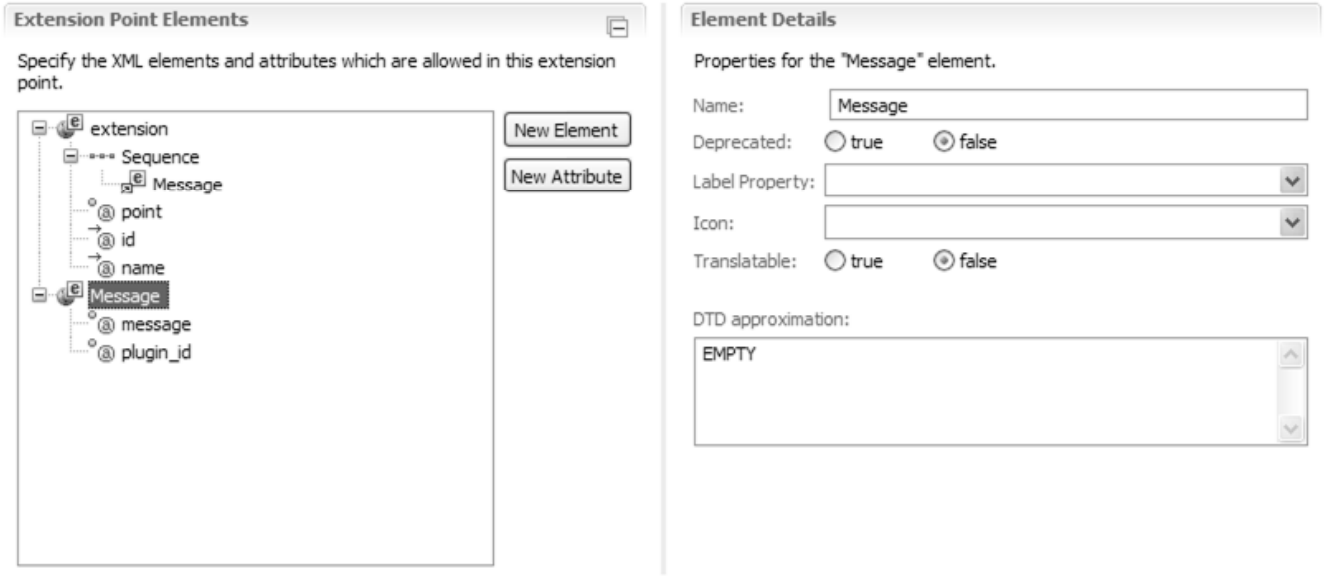


# Puntos de extensión

- Al esquema por defecto, vamos a añadir inicialmente un nuevo elemento, *Message*, que tendrá dos atributos, *plugin_id* y *message*.
- Cada *plug-in* que se conecte al *extension point*, notificará un identificador y un mensaje que quiera mandar al mundo.
- Nuestro *plug-in* del punto de extensión, será capaz de consultar la lista de *plug-ins* que están conectados, y cuando pulsemos el botón de «Hola mundo!», nos dirá todos los mensajes que hayan indicado los *plug-ins* conectados.

# Puntos de extensión

- En primer lugar, el nuevo esquema del punto de extensión queda como el siguiente:



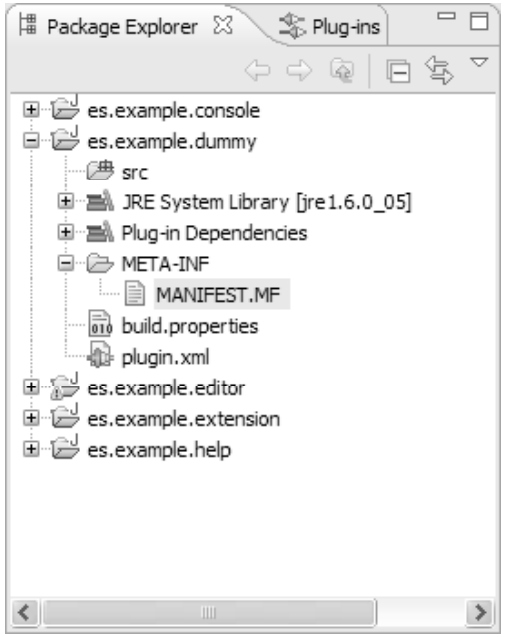
# Puntos de extensión

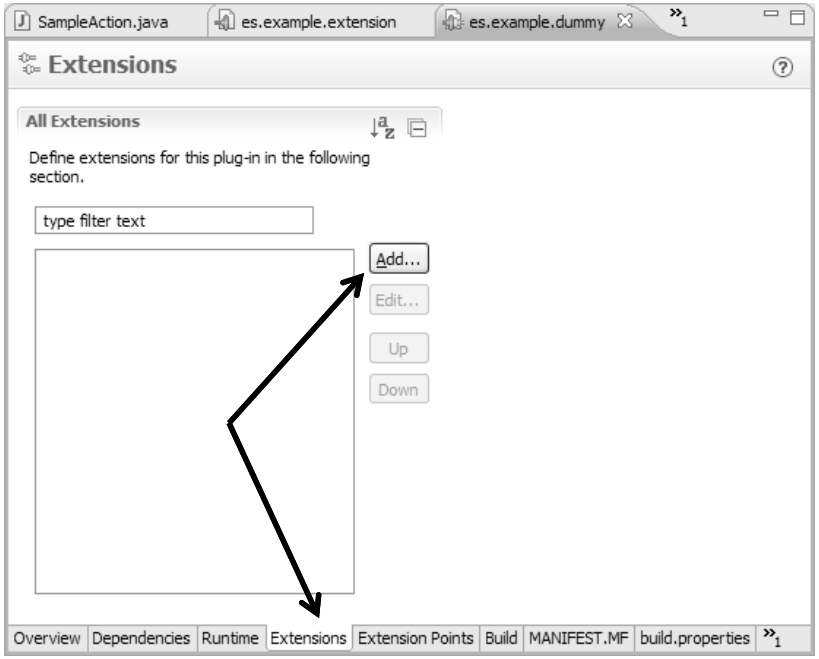
- Y por su parte, el método *run()* de nuestro botón, de la siguiente manera:

```
public void run(IAction action) {
        StringBuffer buffer = new StringBuffer();
        IExtensionRegistry reg = Platform.getExtensionRegistry();
        IConfigurationElement[] extensions = reg
                                .getConfigurationElementsFor("es.example.extension");
        for (int i = 0; i < extensions.length; i++) {
                IConfigurationElement element = extensions[i];
                buffer.append(element.getAttribute("plugin_id"));
                buffer.append(": \"");
                buffer.append(element.getAttribute("message"));
                buffer.append("\"\n");
        }
        MessageDialog.openInformation(
                window.getShell(),
                "Plug-ins using the extension point",
                buffer.toString());
}
```

# Puntos de extensión

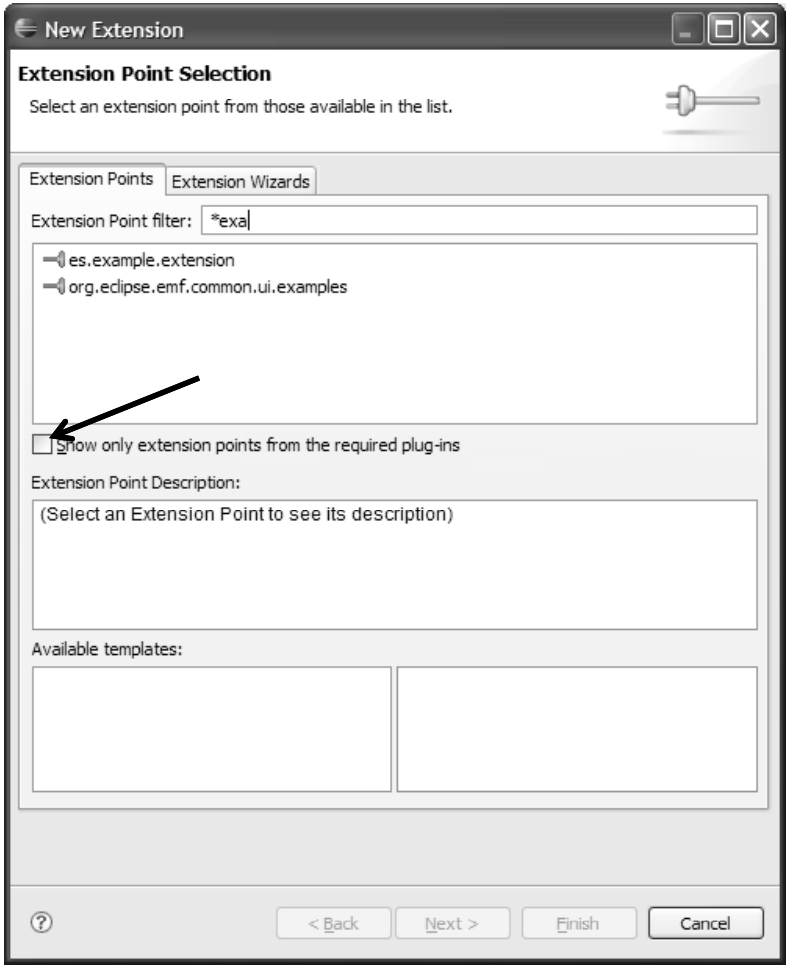
- En este punto, está todo listo para que nuestros *plug-ins* se conecten al punto de extensión.
- Para ello podemos crear un proyecto de *plug-in* vacío, y editaremos el archivo de manifiesto:



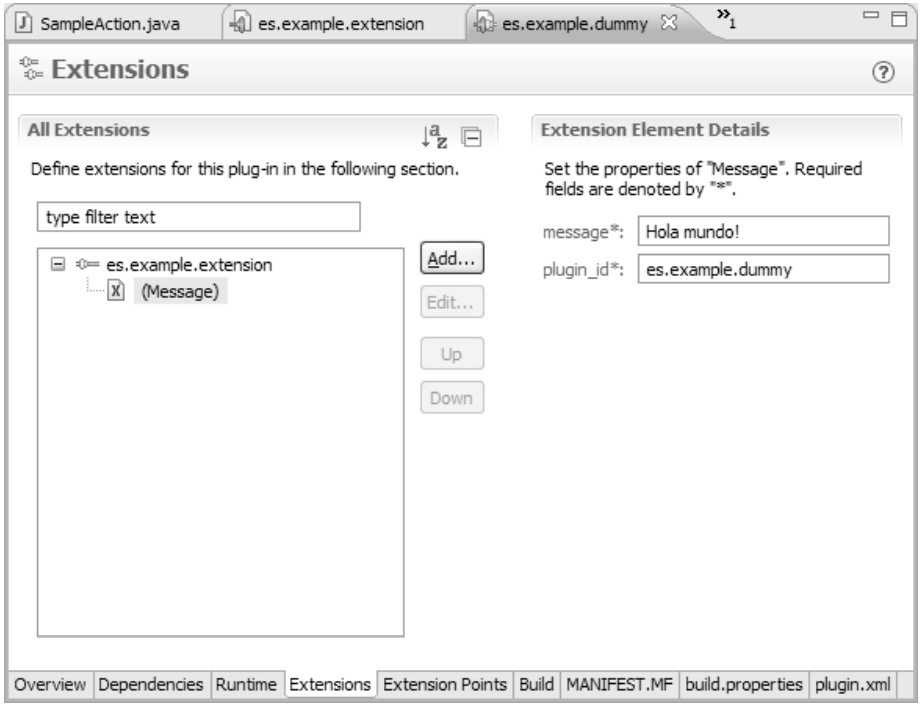
# Puntos de extensión

- Y seleccionaremos el punto de extensión recién creado. Si nuestro *plug-in* con el punto de extensión de ejemplo no está entre las dependencias, deberemos desmarcar el *checkbox*.



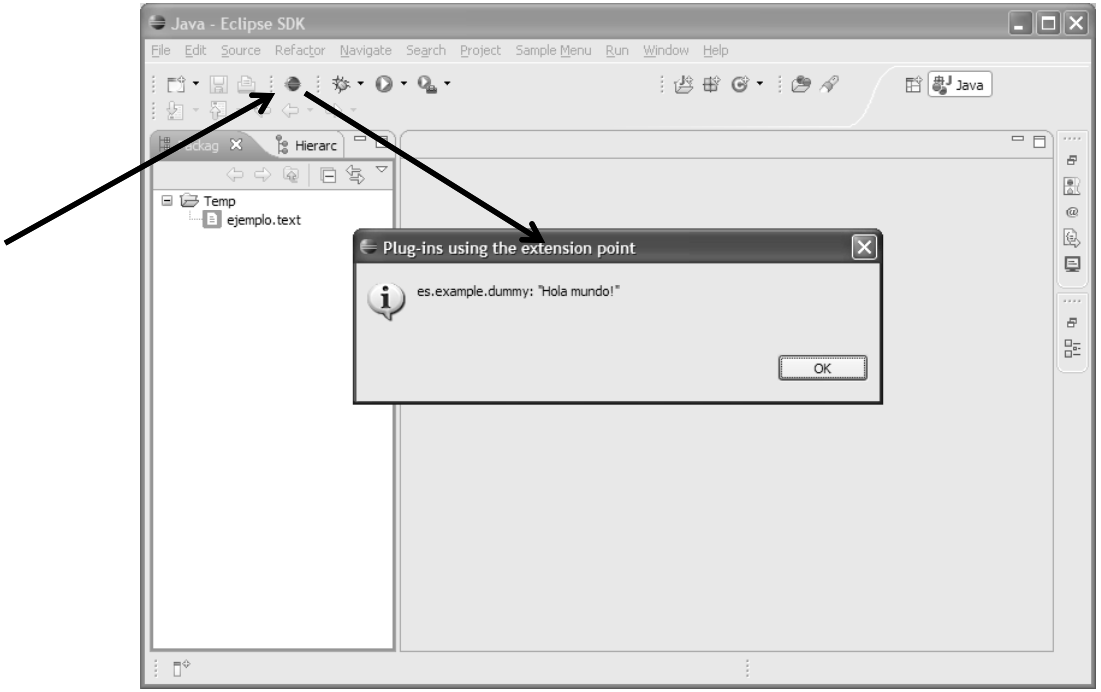
# Puntos de extensión

- Una vez se ha incluido el punto de extensión, aparecen en el editor los dos campos que hemos indicado que se pueden establecer.
- Basta rellenar su contenido, y podemos lanzar el conjunto de *plug-ins* a ejecución.



# Puntos de extensión

- Una vez se ha incluido el punto de extensión, aparecen en el editor los dos campos que hemos indicado que se pueden establecer.
- Basta rellenar su contenido, y podemos lanzar el conjunto de *plug-ins* a ejecución.



# Puntos de extensión

- Hemos visto un ejemplo de cómo el *plug-in* que ofrece el punto de extensión es capaz de consultar los atributos rellenados por aquellos *plug-ins* que importan el punto de extensión.
- Pero, ¿cómo hacer para ejecutar código de un *plug-in* que importa el punto de extensión?
- Tomemos el siguiente caso, y supongamos que nuestro *plug-in* de «Hola mundo!» quiere que cada *plug-in* salude por sí mismo cuando pulsemos el botón, en lugar de solicitar el mensaje a cada *plug-in* que emplea el punto de extensión…

# Puntos de extensión

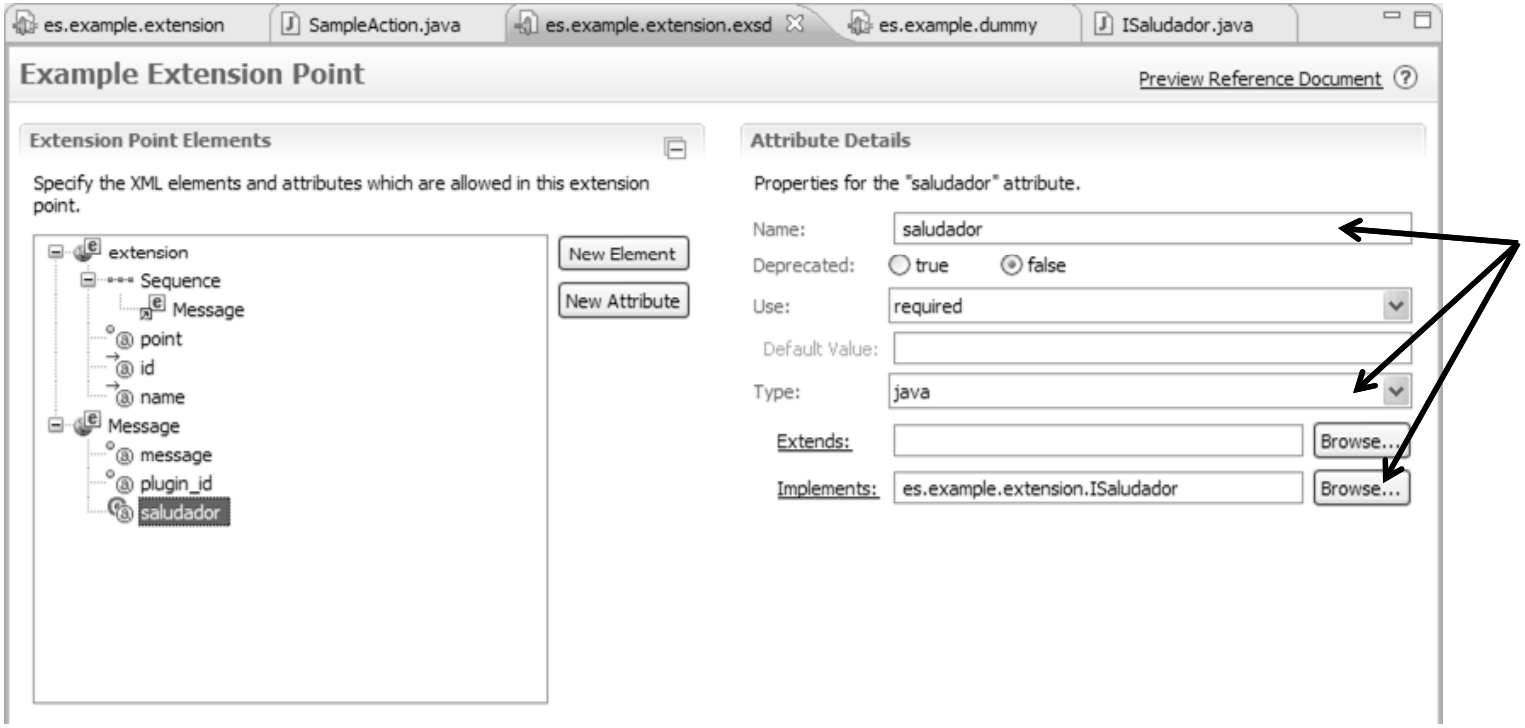
- En primer lugar, hemos de definir la interfaz que debe implementar cada *plug-in* para que se pueda conectar al punto de extensión.
- Llamaremos a esta interfaz *ISaludador:*

```
package es.example.extension;

public interface ISaludador {
  public void saludar();
}
```

# Puntos de extensión

- Igualmente, deberemos modificar la definición del punto de extensión, indicando que necesitaremos un nuevo atributo de tipo *Java,* y que implementará la interfaz *ISaludador*. Debemos asegurarnos de que nuestro exporta el paquete donde se define la interfaz *ISaludador.*



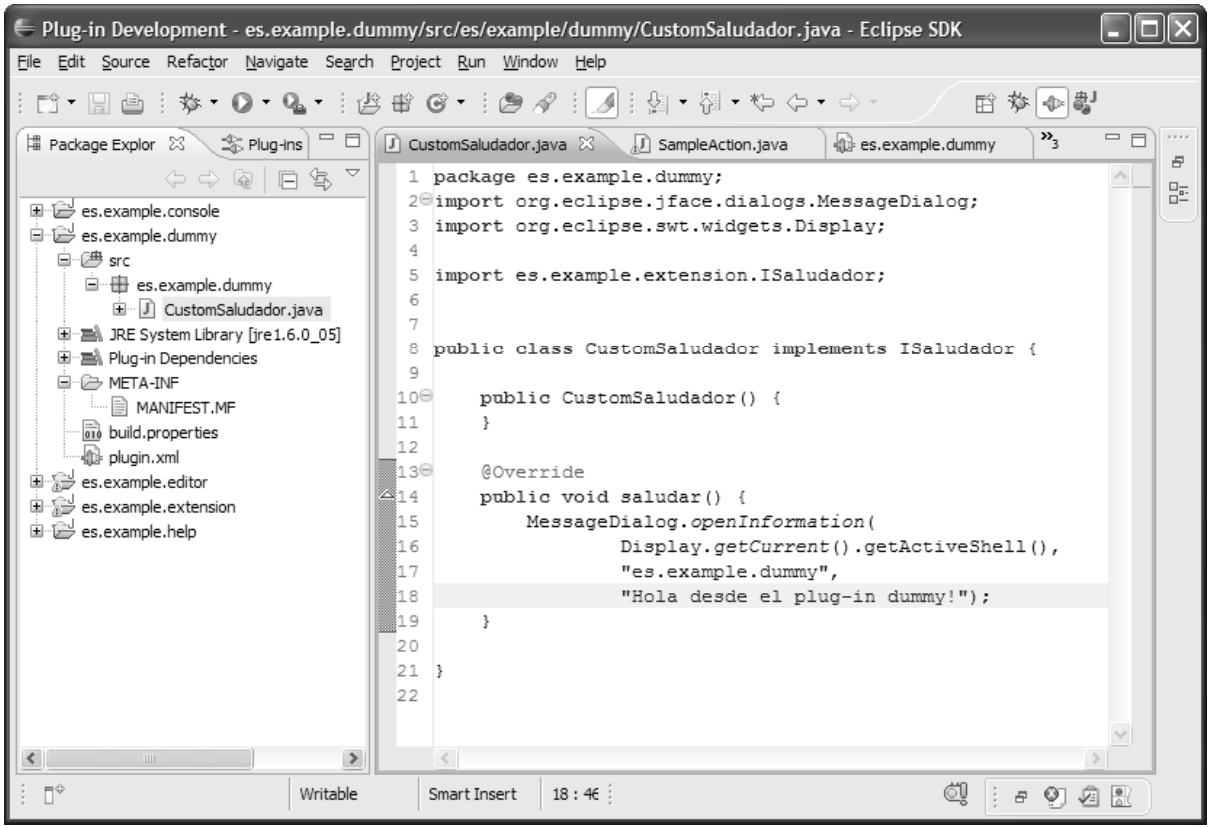
# Puntos de extensión

- Y debemos modificar nuestro método *run()*, para que consulte el nuevo atributo «*saludador*», que será un nombre de clase, y cree una instancia de ésta.

```
public void run(IAction action) {
   StringBuffer buffer = new StringBuffer();
   IExtensionRegistry reg = Platform.getExtensionRegistry();
   IConfigurationElement[] extensions = reg
         .getConfigurationElementsFor("es.example.extension");
   for (int i = 0; i < extensions.length; i++) {
      IConfigurationElement element = extensions[i];
      ISaludador saludador = null;
      try {
         saludador = (ISaludador)
              element.createExecutableExtension("saludador");
      } catch (CoreException e) { e.printStackTrace(); }
      saludador.saludar();
   }
}
```

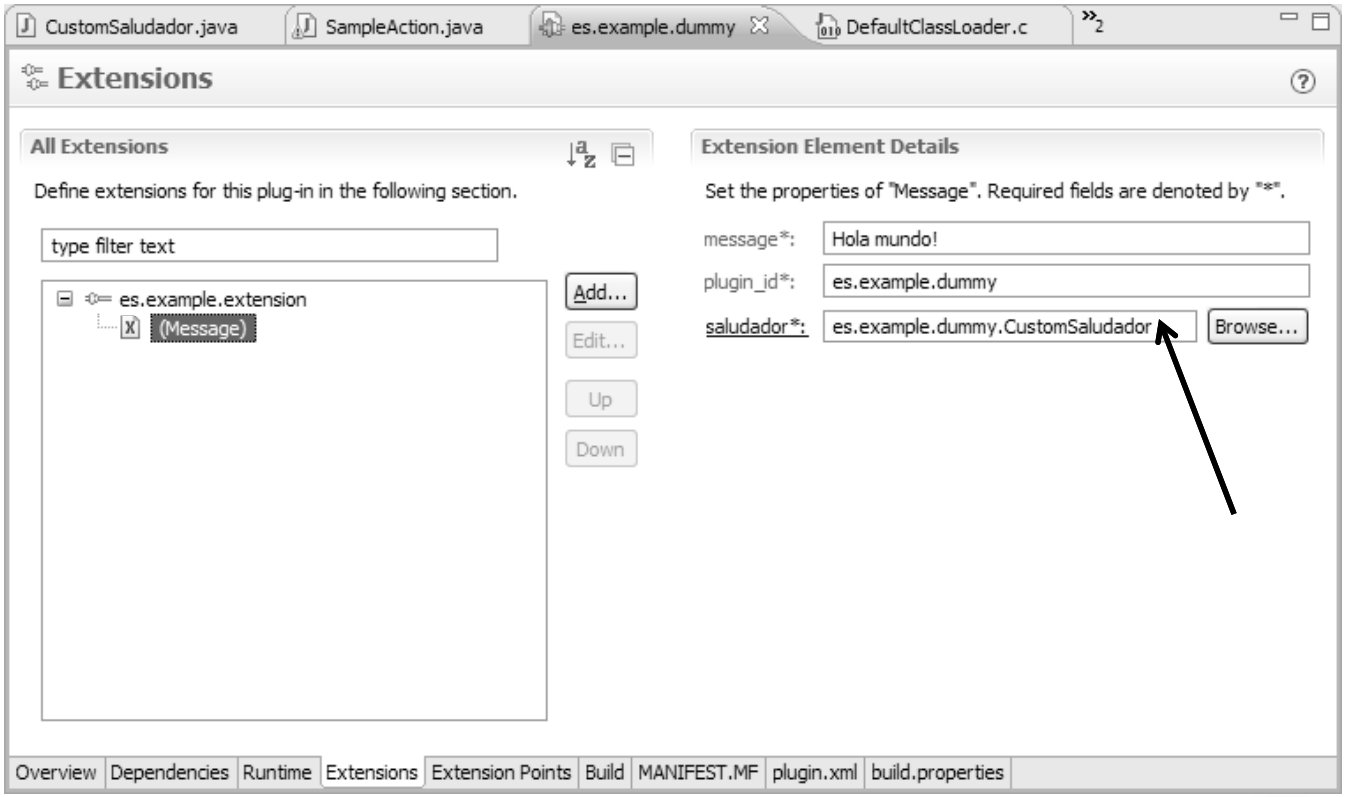
# Puntos de extensión

- Por último, modificamos nuestro *plug-in dummy*:
  - Creamos la nueva clase *CustomSaludador*, que implementa la interfaz *ISaludador*.



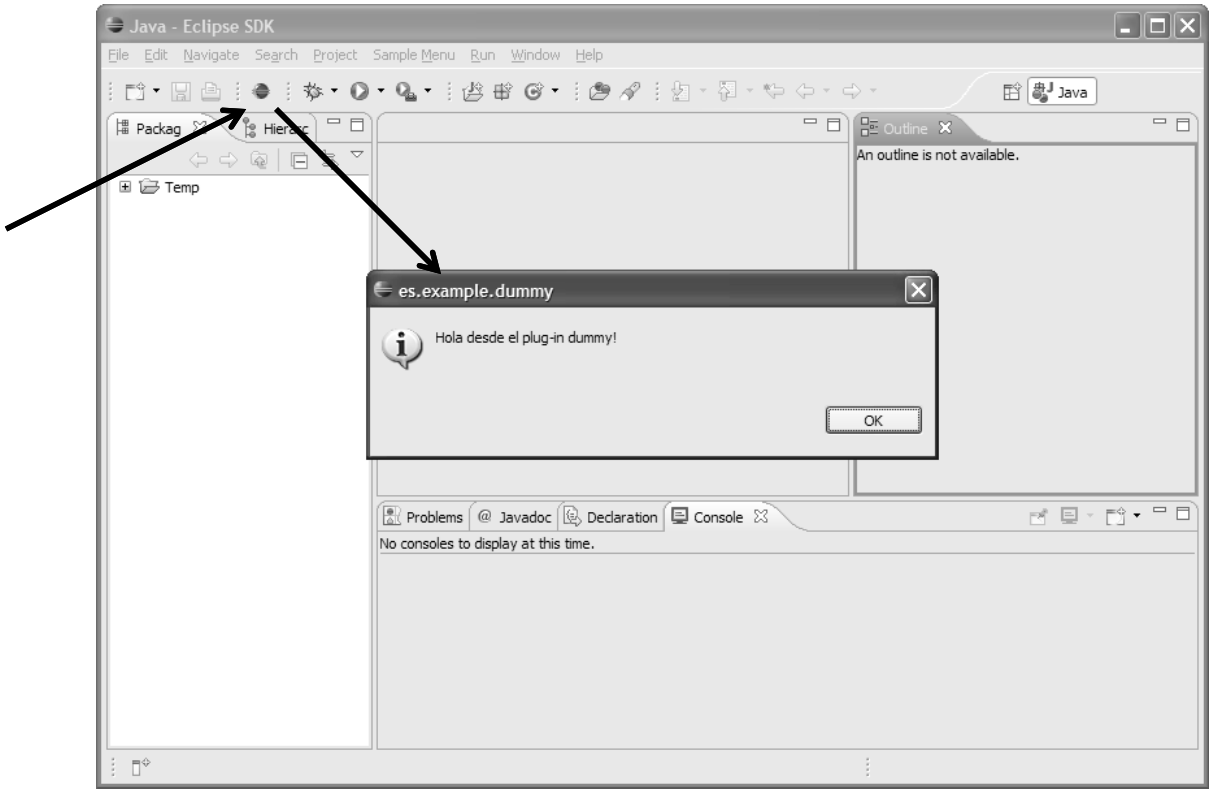
# Puntos de extensión

- Por último, modificamos nuestro *plug-in dummy*:
  - Actualizamos el archivo de manifiesto, para indicar la clase que implementa la interfaz *ISaludador*.



# Puntos de extensión

- En este punto, ya podemos ejecutar nuestros *plug-ins* en una nueva instancia de eclipse:



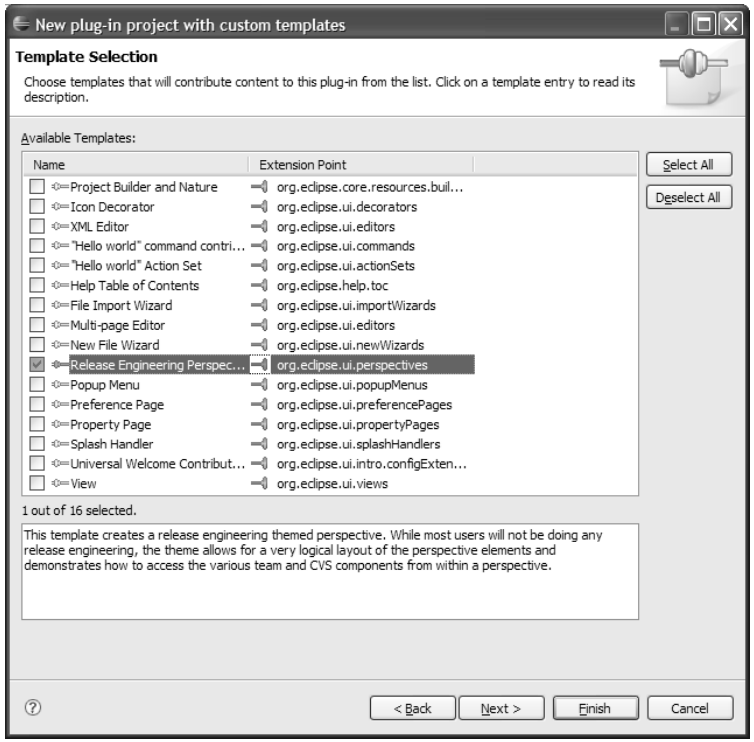
# Creación de una nueva perspectiva

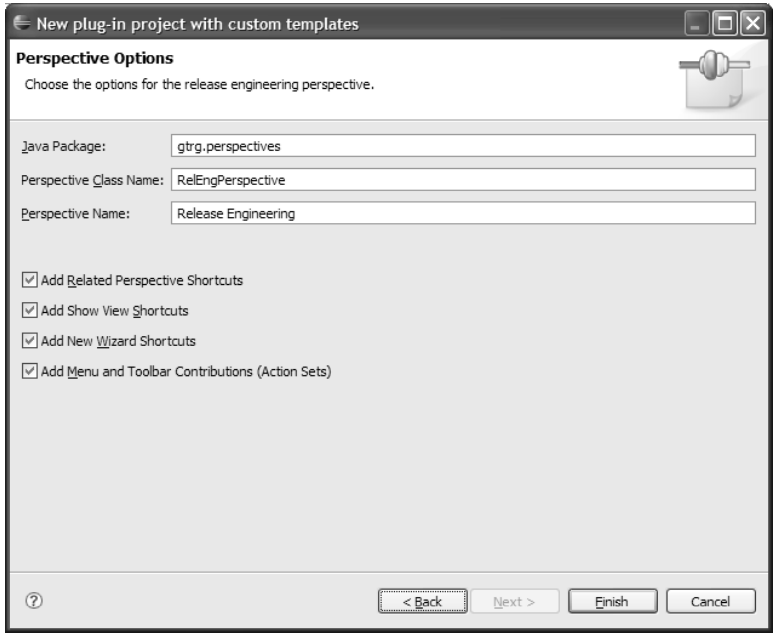


eclipse

# Creación de una perspectiva

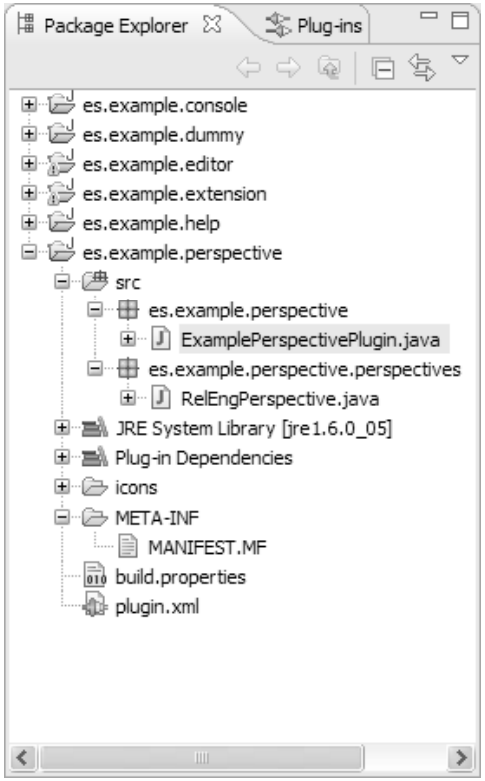
- Eclipse proporciona un asistente de ejemplo para crear una nueva perspectiva.
- Éste se lanza desde la localización habitual, en el asistente de creación de un nuevo *plug-in*.

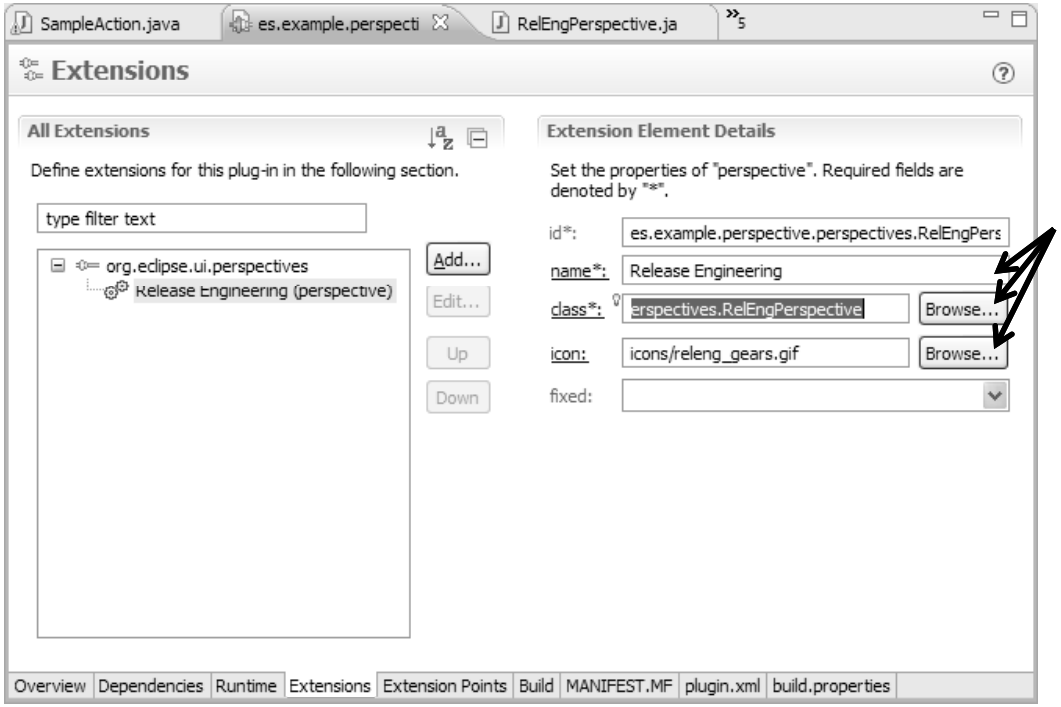


# Creación de una perspectiva

- El asistente crea un proyecto típico.
- La configuración de la perspectiva se encuentra toda en una única clase, en este caso, *RelEngPerspective*.



# Creación de una perspectiva

- Todo elemento del *workbench* tiene un identificador (id) asociado.
- Este ide es el que se emplea para poder determinar cuáles serán los elementos visibles por defecto en la perspectiva.
- La clase *RelEngPerspective* define los siguientes métodos, cada uno de ellos mostrando un grupo distinto de elementos del *workbench*:
  - createInitialLayout(IPageLayout) — Este método invoca a todos los demás. Inicializa la perspectiva.
  - addViews() — Define las vistas activas del workbench, así como su posición en él.
  - addActionSets() — Añade los grupos de iconos que estarán disponibles en la barra de herramientas.
  - addPerspectiveShortcuts() — Determina para qué perspectivas se monstrarán los iconos en la zona de atajos.
  - addNewWizardShortcuts() — Indica qué Asistentes se mostrarán en el menú contextual de crear un nuevo elemento.
  - addViewShortcuts() — Determina cuáles serán los atajos para activar una vista que se mostrarán.

# Ejercicio de evaluación



eclipse

# Ejercicio de evaluación

- Modificar el *plug-in* de consola para que ofrezca un punto de extensión que solicitará dos parámetros:
  - plugin_id — el id del plugin que importe el punto de extensión.
  - enabled — si la consola se encuentra activa por defecto para ese plug-in.
- Modificar la hoja de propiedades de la consola para que consulte todos los *plug-ins* que importan el punto de extensión.
  - Debe definir un control que permita, para cada uno de los *plug-ins*, definir si la consola está activa o no.
- Modificar los métodos *printError*, *printMessage*, y *printDebugMessage* para que reciban dos argumentos: el identificador del *plug-in* que lo invoca, y el mensaje a mostrar.
  - El *plug-in* de consola sólo mostrará el mensaje si el *plug-in* ha definido la consola como activa en las preferencias.