Introduction au développement par composants (d'extension) ou plugins

Intervenant : Chouki TIBERMACINE
Bureau : LIRMM (E.311)
Tél. : 04.67.14.97.24
Mél. : Chouki.Tibermacine@lirmm.fr
Web : https://github.com/ctiber/composants/

Composants s'ouvrant au monde extérieur

Du « *Programming in the Large* »

Composant = plusieurs classes (+ interfaces) = un JAR++

à l'« *Open Development* »

Composant = entité s'ouvrant sur le monde extérieur
en déclarant des points d'extension

Plan du cours

- Extensions, points d'extension et « Extension Registry »
- Environnement Eclipse

Plan du cours

- Extensions, points d'extension et « Extension Registry »
- Environnement Eclipse

Principe général

- Les bundles OSGi peuvent être étendus ou configurés :
  - en déclarant des points d'extension
  - Point d'extension = Contrat avec de potentielles extensions
- D'autres bundles apportent des contributions sous la forme d'extensions :
  - en fournissant des données ou en indiquant des classes à exécuter (à instancier et invoquer les méthodes des objets) qui respectent le contrat
- Ils s'enregistrent auprès d'un « Extension Registry »
- Plugin = Bundle++

Relation Extension-Point d'extension

- Dans chaque plugin Eclipse, nous disposons d'un descripteur (fichier de configuration) : plugin.xml
- Celui-ci comporte les déclarations des points d'extension et des extensions du plugin

- Exemple de point d'extension :

```
<extension-point id="afficheur.messages.a.afficher"
                 name="afficheur.messages.a.afficher"/>
```

Relation Extension-Point d'extension

- Exemple d'extension :

```
<extension point="afficheur.messages.a.afficher"
           id="messages">
  <message value="Hello World"></message>
</extension>
```

- Notez que dans Eclipse, il y a un environnement de développement de plugins (PDE) qui facilite l'édition de ces descripteurs de plugins

Recherche d'extensions d'un point d'extension

- C'est le plugin qui déclare le point d'extension qui contrôle les éventuelles extensions

- Obtenir une référence vers l'extension registry (DI possible) :

```
IExtensionRegistry registry =
                Platform.getExtensionRegistry();
```

- Rechercher les éléments de configuration (contenu de la balise extension) dans toutes les extensions fournies par les autres plugins pour un même point d'extension (id du point d'extension) :

```
IConfigurationElement[] elements =
 registry.getConfigurationElementsFor(
                          "afficheur.messages.a.afficher");
```

Recherche d'extensions d'un point d'extension -suite-

- Manipuler les extensions :

```
IConfigurationElement[] elements =
 registry.getConfigurationElementsFor(
                        "afficheur.messages.a.afficher");
for(IConfigurationElement e : elements) {
   String message = e.getAttribute("value");
   System.out.println(message);
}
```

Dépendances entre plugins

- Les plugins sont des bundles OSGi, leurs dépendances doivent donc être explicitement indiquées dans le Manifest
- Ces dépendances sont renseignées dans la partie : Require-Bundle
- Si votre plugin étend Eclipse (ajout de menus, vues, ...), il faudra ajouter les plugins étendus (liste des plugins fournie par PDE)
- Exemple :
  Require-Bundle: org.eclipse.ui,org.eclipse.core.runtime,...
- PDE fournit un éditeur de Manifest graphique

Contrat entre points d'extension et extensions

- Le contrat entre une extension et un point d'extension est décrit par un schéma XML défini dans le point d'extension :

```
<extension-point id="..." name="..."
                 schema="schema/mon_contrat.exsd" />
```

- Ce schéma XML décrit la structure que doivent respecter les éléments XML définis dans les extensions

- Exemple précédent : un élément message avec un attribut value

Contrat entre points d'extension et extensions

- Ces éléments XML décrivent soit :
  - **une configuration** : **des données** fournies au plugin déclarant le point d'extension
    Exemple précédent : des messages textuels à afficher Hello World -> éléments de configuration
  - **une extension avec du code** : un élément de configuration XML définissant un attribut ayant comme valeur le nom d'une classe

Extensions apportant des données pour une configuration

- Exemple : Extension apportant des messages à afficher

```
<extension point="afficheur.messages" id="plugin.msgs">
   <message value="Bonjour à tous"></message>
   <message value="Hello World"></message>
</extension>
```

Attributs de type String

- Celui qui affiche les messages est le plugin déclarant le point d'extension

- Le schéma XML représentant le contrat indique qu'il doit y avoir 1 ou plusieurs éléments nommés « message » ayant un attribut nommé « value » de type String (les noms de l'élément et de son attribut sont libres)

Extensions apportant des programmes (classes)

- Exemple : Supposons que notre plugin afficheur déclare un autre point d'extension `afficheur.visualiseur`
- Il attend que des plugins contribuent avec des visualiseurs (graphiques, sur console, …)
- Ce point est étendu par un plugin affichant des messages de façon graphique

Extensions apportant des programmes (classes)

- L'extension dans plugin.xml :

```
<extension point="afficheur.visualiseur"
    id="visualiseur.graphique">
   <visualiseur class="visualiseur.GUI_MessageViewer">
   </visualiseur>
</extension>
```

- Le schéma XML représentant le contrat indique qu'il doit y avoir un élément nommé « visualiseur » ayant un attribut nommé « class » de type Java (les noms de l'élément et de son attribut sont libres)

Extensions apportant des programmes (classes) -suite-

- Dans le contrat du point d'extension, on indique le nom d'une interface ou d'une classe pour lesquelles les extensions doivent fournir une implémentation ou une spécialisation, resp.

- Exemple : indiquer le nom d'une interface IViewer (interface requise du composant qui déclare le point d'extension)

```
public interface IViewer {
   public void view(String msg);
}
```

- La classe indiquée dans l'extension implémente cette interface :

```
<extension ...>
   <visualiseur class="visualiseur.GUI_MessageViewer">
...
```

Lancer l'exécution des « extensions avec du code »

- A partir du code qui s'exécute dans le plugin qui déclare le point d'extension (par exemple, dans sa classe Activator, ou en réponse à une action effectuée sur une extension graphique fournie par ce plugin (voir plus loin)), rechercher dans le registry les extensions qui contribuent au point d'extension et les lancer :

```
IConfigurationElement[] elements =
  registry.getConfigurationElementsFor("afficheur.visualiseur");
...
Object o = e.createExecutableExtension("class");
if (o instanceof IViewer)
   ((IViewer)o).view(msg);
```

Si exceptions levées par cette extension, possibilité de plantage des autres extensions (dysfonctionnement d'Eclipse)

Structure d'un plugin

- Un plugin peut être fourni dans un JAR (cas le plus fréquent) :
  - Classes Java respectant la structure répertoire/package
  - Répertoires icons/ ou images/
  - META-INF/MANIFEST.MF
  - plugin.xml
- Il peut être fourni également sous la forme d'un répertoire
  - Classes dans un JAR
  - Le nom du jar indiqué dans le Manifest : Bundle-ClassPath

- Avant, l'installation d'un nouveau plugin consistait à télécharger puis copier le répertoire/Jar, et le placer dans le répertoire plugins Maintenant, on a des solutions plus simples via l'Eclipse Marketplace ou les sites Web de mise à jour

Plan du cours

- Extensions, points d'extension et « Extension Registry »
- Environnement Eclipse

Structure générale

- Eclipse n'est pas un programme monolithique
- Il est constitué d'un noyau comportant des services de base et un nombre important de plugins (des centaines voire des milliers, selon l'installation Eclipse)
- Le noyau est une implémentation du framework OSGi (Equinox)

Catégories de plugins trouvés dans Eclipse

- Core : des plugins de bas niveau fournissant les services de base de traitement des extensions, ...
- SWT (Standard Widget Toolkit) : bibliothèque générale de widgets pour construire des interfaces graphiques
- JFace : une bibliothèque graphique construite sur SWT (comporte des widgets plus riches)
- Workbench Core et Workbench UI : plugins utilisés par Eclipse fournissant la gestion des projets, des éditeurs, des vues, ...
- JDT (Java Development Tooling) : plugins utilisés par Eclipse pour la programmation en Java
- PDE (Plugin Development Environment)
- ...

SWT

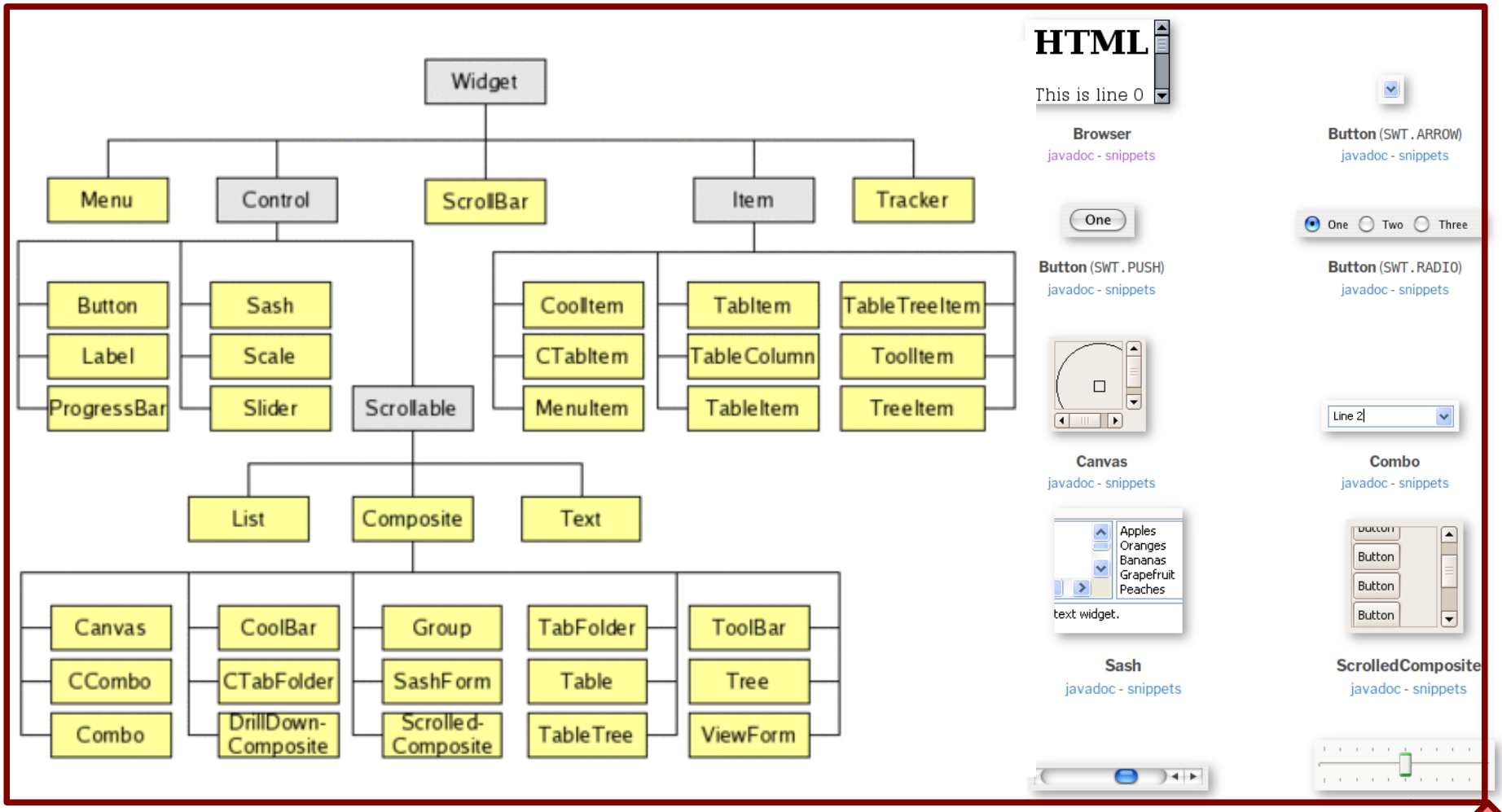
- Bibliothèque développée par OTI (Object Technology International), entreprise ayant développé Smalltalk, achetée par IBM

- Concurrent de Swing, qui a déçu à ses débuts (mauvaises performances). De +, les widgets AWT étaient jugés trop simples

- Exemple avec SWT :

```
Display display = new Display(); // Instance SWT<->OS
Shell shell = new Shell(display); // Fenêtre
shell.setText("Hello World");
shell.setBounds(100,100,200,50);
shell.setLayout(new FillLayout());
Label label = new Label(shell, SWT.CENTER);
label.setText("Hello World");
shell.open(); ... (traiter les événements)
display.dispose();
```

Widgets SWT


Widget
Menu
Control
ScrollBar
Item
Tracker
Button
Label
ProgressBar
Sash
Scale
Slider
Scrollable
CoolItem
CTabItem
MenuItem
TabItem
TableColumn
TableItem
TableTreeItem
ToolItem
TreeItem
List
Composite
Text
Canvas
CCombo
Combo
CoolBar
CTabFolder
DrillDown-Composite
Group
SashForm
Scrolled-Composite
TabFolder
Table
TableTree
ToolBar
Tree
ViewForm
HTML
This is line 0
Browser
javadoc - snippets
Button (SWT.ARROW)
javadoc - snippets
One
Button (SWT.PUSH)
javadoc - snippets
One Two Three
Button (SWT.RADIO)
javadoc - snippets
Canvas
javadoc - snippets
Line 2
Combo
javadoc - snippets
Apples
Oranges
Bananas
Grapefruit
Peaches
text widget.
Sash
javadoc - snippets
Button
Button
Button
ScrolledComposite
javadoc - snippets
Slider
javadoc - snippets
Scale
javadoc - snip

Évènements SWT

- Même mécanisme que dans AWT : objets représentant des **événements**, objets **écouteurs** d'événements et objets **sources** d'événements

- API pour gérer les événements :
  Méthodes : add<EventName>Listener(...)
  (ex : addMouseListener(...))
  Interface Listener, Classes Adapter, ...

- Quelques types d'événements :
  - Control, Dispose, Focus, Help, Key, Menu, Mouse, MouseMove, Selection, ...

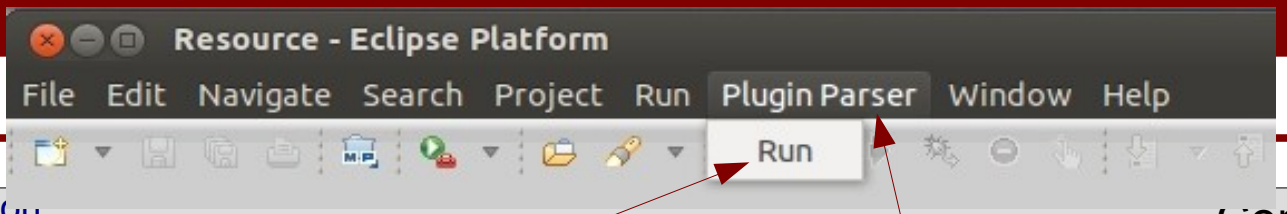
JFace Viewers

- SWT : librairie pour représenter graphiquement des données simples : chaînes de caractères, nombres, ...
- Pour représenter des données complexes, JFace propose des widgets plus évolués : tables, arbres, ...
- Exemples : ListViewer, TableViewer, TreeViewer, TextViewer, ...
  Des interfaces à implémenter pour une mise en correspondance entre les données métier et les éléments qui composent ces viewers
  Ensuite, instancier ces viewers et les ajouter comme widgets

Commands & Actions

- Commands : une API pour ajouter une contribution sous la forme d'un item de menu ou un bouton d'une barre d'outils
- A déclarer : 1) une commande -> 2) un menu -> 3) un handler
- Contribuer par des extensions à trois points d'extension :
  org.eclipse.ui.commands
  org.eclipse.ui.menus
  org.eclipse.ui.handlers
- Actions : une API dépréciée, probablement supprimée à l'avenir, pour faire la même chose mais d'une manière moins modulaire (pas de séparation entre déclarations et implémentation)

Exemple de commande : les extensions




```
<plugin>
   <extension
         point="org.eclipse.ui.commands"
         id="fr.lirmm.marel.pluginparser.command"
         name="Command related to running the plugin parser">
      <command name="Run"
         id="fr.lirmm.marel.pluginparser.run"/>
   </extension>
   <extension
         point="org.eclipse.ui.menus">
         <menuContribution locationURI="menu:org.eclipse.ui.main.menu">
               <menu id="fr.lirmm.marel.pluginparser.menu" label="Plugin Parser" mnemonic="l">
                     <command commandId="fr.lirmm.marel.pluginparser.run"
                           id="fr.lirmm.marel.pluginparser.parse" mnemonic="r"/>
               </menu>
         </menuContribution>
   </extension>
   <extension
         point="org.eclipse.ui.handlers">
         <handler commandId="fr.lirmm.marel.pluginparser.run"
                  class="fr.lirmm.marel.pluginparser.ParserHandler"/>
   </extension>
</plugin>
```

Commande

Menu

Handler

Exemple de commande : le handler

```
package fr.lirmm.marel.pluginparser;

import org.eclipse.jface.dialogs.MessageDialog;

import org.eclipse.core.commands.AbstractHandler;
import org.eclipse.core.commands.ExecutionEvent;
import org.eclipse.core.commands.ExecutionException;

public class ParserHandler extends AbstractHandler {

    @Override
    public Object execute(ExecutionEvent arg0)
                    throws ExecutionException {
        MessageDialog.openInformation(null, "Info", "Hello World!");
        return null;
    }

}
```

Commandes présentées sous d'autres formes (valeur de locationURI)

- Dans un menu existant :
  - menu:file?after=open.ext
  - menu:window?before=newEditor
- Dans la barre d'outils :
  - toolbar:org.eclipse.ui.menu.toolbar?after=additions
- Un menu contextuel :
  - popup:org.eclipse.ui.popup.any

Parties d'un workbench Eclipse (*Workbench Parts*)

- Les **éditeurs** : permettent de visualiser et modifier une ressource (fichier de code source Java, par exemple). Ils fonctionnent avec un mode : ouvrir-enregistrer-fermer
  Ils sont affichés dans un endroit unique

- Les **vues** : constituent les autres parties autour de l'éditeur. Elles peuvent être associées à une ressource (ou plusieurs), ou à aucune ressource. Elles sont censées impacter les modifications automatiquement sur les ressources (ex : l'explorateur de projets, modification automatique du nom d'un fichier .java)

- Les parties (***parts***) sont des vues ou des éditeurs

- Une **perspective** est un ensemble de vues et de commandes

Vues

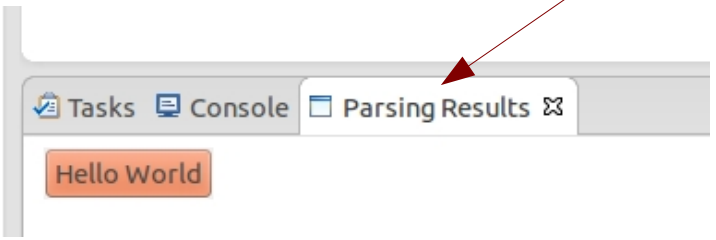
- Les vues sont affichées autour de l'éditeur
- Exemples de vues : Console, Problems, Package Explorer, … (voir menu Window → Show View)
- Des centaines de vues existent dans un workbench : organisées en catégories
- Déclarer une nouvelle vue :
  - Définir la catégorie de la vue dans la config du plugin
  - Déclarer la vue dans la config du plugin
  - Écrire le code (classes) qui composent la vue : view part

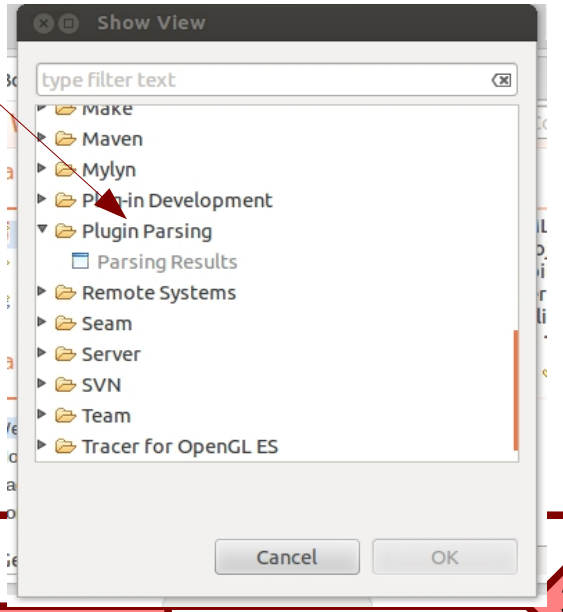
Exemple de vue : déclarations

- Déclarations de la vue et de sa catégorie :

```
<extension point="org.eclipse.ui.views">
        <category id="fr.lirmm.marel.pluginparser.viewcategory"
                  name="Plugin Parsing"/>

        <view category="fr.lirmm.marel.pluginparser.viewcategory"
              class="fr.lirmm.marel.pluginparser.PluginParserView"
               id="fr.lirmm.marel.pluginparser.view" inject="true"
               name="Parsing Results"/>
</extension>
```





Exemple de vue : code de la vue

- Classe représentant la vue :

```
package fr.lirmm.marel.pluginparser;
import org.eclipse.swt.widgets.*;
import org.eclipse.ui.part.*;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.*;
public class PluginParserView extends org.eclipse.ui.part.ViewPart {
     Button b;
     @Override
     public void createPartControl(Composite parent) {
          parent.setLayout(new org.eclipse.swt.layout.GridLayout());
          b = new Button(parent, SWT.CENTER);
          b.setText("Hello World");
          b.addSelectionListener(new SelectionAdapter() {
               @Override
               public void widgetSelected(SelectionEvent e) {
                    System.out.println("Hello World");
               }
          });
     }
     @Override
     public void setFocus() {b.setFocus();}
}
```

Éditeurs

- Éditeurs déjà fournis par Eclipse : éditeur de texte, de code source Java, de fichiers XML, de config. de plugins Eclipse, …
- Une classe qui représente un éditeur doit étendre org.eclipse.ui.part.EditorPart
- Créer un nouvel éditeur :
  - Déclarer l'éditeur comme extension

```
<extension point="org.eclipse.ui.editors">
  <editor
    id="fr.lirmm.marel.jsoneditor"
    name="JSON Editor"
    extensions="json"
    icon="icons/json.gif"
   contributorClass="org.eclipse.ui.texteditor.BasicTextEditorActionContributor"
    class="fr.lirmm.marel.json.Editor"/>
</extension>
```

  - Écrire la classe de l'éditeur `fr.lirmm.marel.json.Editor`

Perspectives

- Un moyen de grouper des commandes et des vues
- Perspectives fournies dans Eclipse : Java, Plug-in Development, Web, Java EE, …
- Elles sont fournies par des contributions majeures à Eclipse, sinon, étendre les perspectives existantes
- Créer une nouvelle perspective :
  - Déclarer une extension au point : org.eclipse.ui.perspectives
  - Écrire une classe Factory qui implémente IPerspectiveFactory dans laquelle il faudra utiliser un objet de type IPageLayout pour placer l'éditeur, les différentes vues et les commandes ou actions

Features et *Products*

- Une *feature* = Un ensemble de plugins représentant une fonctionnalité métier bien définie (permettant son installation et sa mise à jour depuis des serveurs dédiés)
  - Pas de code dans une feature
  - Décrite par une configuration (manifest) : liste de plugins, URL de mise à jour, ...
- Exemples : platform, JDT, EMF, Mylyn, ...
- Un product = Un ensemble de features et de plugins packagés ensemble au sein d'une même entité destinée à être rendue disponible pour téléchargement (update, ...)

Quelques références

- **OSGi and Equinoxe: Creating Highly Modular Java Systems.** Jeff McAffer, Paul VanderLei et Simon Archer. The Eclipse Series. Dans les éditions de Jeff McAffer, Erich Gamma et John Weigand. Addison Wesley, 2010.

- **Eclipse Plug-ins, 3rd edition.** Eric Clayberg et Dan Rubel. The Eclipse Series. Dans les éditions de Erich Gamma, Lee Nackman et John Wiegand. Addison Wesley, 2009.

- **Tutoriel de Lars Vogel** : http://www.vogella.com/

Questions