Eclipse RCP Part VII

Automotive Financial Services Insurance Life Science & Healthcare Public Sector
Telecommunications & Media Travel & Logistics Utilities Automotive Financial
Services Insurance Life Science & Healthcare Public Sector Telecommunications
& Media Travel & Logistics Utilities Automotive Financial Services Insurance
Life Science & Healthcare Public Sector Telecommunications & Media Travel
& Logistics Utilities Automotive Financial Services Insurance Life Science
Healthcare Public Sector Telecommunications & Media Travel & Logistics
Utilities Automotive Financial Services Life Science & Healthcare Public
Sector Telecommunications & Media Travel & Logistics Utilities Automotive
Financial Services Insurance Life Science & Healthcare Public Sector
Telecommunications & Media Travel & Logistics Utilities Automotive
Financial Services Insurance Life Science & Healthcare Telecommunications
& Media Travel & Logistics Utilities Automotive Financial Services Insurance
Life Science & Healthcare Public Sector Telecommunications & Media Travel &
Logistics Utilities Automotive Financial Services Insurance Life Science &
Healthcare Public Sector Telecommunications & Media Travel & Logistics Utilities
Automotive Financial Services Insurance Life Science & Healthcare Public Sector

.consulting .solutions .partnership

.msg systems

Objective

- Building a JFace View

JFace

- JFace is a UI toolkit that provides helper classes for developing common UI related features:
  - structured viewers (Tree, Table, List, Combo)
  - Image and Font registries
  - dialogs and wizards
  - data binding (since 3.3)
  - text editing support
- see org.eclipse.jface.* packages

JFace

- Lists, trees, and tables share many common capabilities from a user's point of view, such as:
  - population with objects
  - object update
  - selection
  - sorting
  - filtering
  - SWT can not handle that kind of abstractions

JFace Viewers

- JFace structured viewers is an MVC-like extension of low level SWT widgets
  - JFace provides viewers for the following widgets in SWT:
  - List → org.eclipse.jface.viewers.ListViewer
  - Tree → org.eclipse.jface.viewers.TreeViewer
  - Table → org.eclipse.jface.viewers.TableViewer
  - Combo → org.eclipse.jface.viewers.ComboViewer

Providers

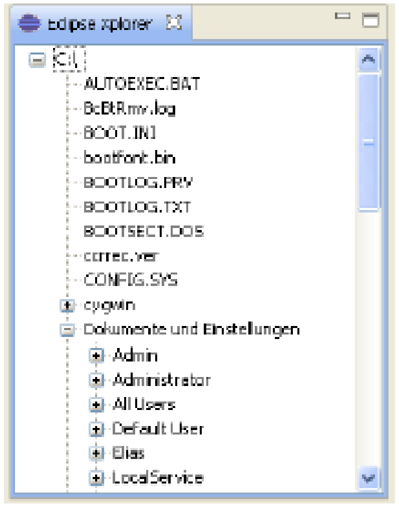
- A viewer keeps an input object and displays it in the underlying SWT widget
- The list of domain objects (elements) to be displayed is obtained using an instance of **IContentProvider** which has to be set on the viewer
- A viewer obtains the label or an image from an **ILabelProvider** which has to be set on the viewer

TreeViewer

- The TreeViewer class is based on the SWT widget Tree
- To use a TreeViewer:
  - Create a TreeViewer passing the underlying SWT Tree to it's constructor (alternatively – an instance of Composite)
  - Define a content provider (ITreeContentProvider) and set it with the method setContentProvider(IContentProvider)
  - Define a label provider (ILabelProvider) and set it with the method setLabelProvider(IBaseLabelProvider)
  - Pass the input object to the TreeViewer using the method setInput(Object)

Sample

```
public void createPartControl(Composite parent) {
    // ...
    TreeViewer viewer  = new TreeViewer(parent);
    viewer.setContentProvider(new FileContentProvider());
    viewer.setLabelProvider(new FileLabelProvider());
    viewer.setInput(new Object[] { new File( "C:/" ) });
}
```



```
Eclipse Xplorer
C:\
    AUTOEXEC.BAT
    BcBtRmv.log
    BOOT.INI
    bootfont.bin
    BOOTLOG.PRV
    BOOTLOG.TXT
    BOOTSECT.DOS
    correc.ver
    CONFIG.SYS
    cygwin
    Dokumente und Einstellungen
        Admin
        Administrator
        All Users
        Default User
        Elias
        LocalService
```

LabelProvider

```
class FileLabelProvider extends LabelProvider {

    public String getText(Object element) {
        File file = (File) element;
        String label = (file.getParent() == null)
            ? file.toString() : file.getName();
        return label;
    }

    public Image getImage(Object element) {
        return null;
    }
}
```

ContentProvider

```
class FileContentProvider implements ITreeContentProvider {

    public Object[] getChildren(Object parentElement) {
        File file = ( File ) parentElement;
        return file.listFiles();
    }

    public boolean hasChildren(Object element) {
        File file = (File) element;
        return file.isDirectory();
    }

    public Object[] getElements(Object inputElement) {
        return (Object[]) inputElement;
    }
    // ...

}
```

```java
class ViewContentProvider implements IStructuredContentProvider {
    public void inputChanged(Viewer v, Object oldInput, Object newInput) {
    }
    public void dispose() {
    }
    public Object[] getElements(Object parent) {
        return new String[] { "One", "Two", "Three" };
    }
}
class ViewLabelProvider extends LabelProvider implements ITableLabelProvider {
    public String getColumnText(Object obj, int index) {
        return getText(obj);
    }
    public Image getColumnImage(Object obj, int index) {
        return getImage(obj);
    }
    public Image getImage(Object obj) {
        return PlatformUI.getWorkbench().
                getSharedImages().getImage(ISharedImages.IMG_OBJ_ELEMENT);
    }
```

Lab

- Import the plug-in net.eclipsetraining.pm.core into your workspace using File > Import > Existing Projects into workspace
- This will provide your application with a very simple data storage for Person objects. You can access this with:
- CoreActivator.getDefault().getPersonManager()
- Create a "Person List" view to your plug-in and add it to the default perspective
- Use a JFace TableViewer to display a list of persons as shown in the objective screenshot. For accessing the data you can call enumeratePersons() on the PersonManager.
- The array contains two persons by default, so you can try it out immediately.

Vielen Dank für Ihre Aufmerksamkeit

.consulting .solutions .partnership