Sofa 2.0

Repository Editor Eclipse plug-in.

Developer’s guide

Maksym Nesen

Introduction

The manual is aimed to describe the structure of the plug-in, interactions between the plug-in and the Eclipse IDE, and to provide detailed description of the contributions to Eclipse's workbench.

The manual describes OSGi basics, way of plugging into the workbench, main milestones for the plug-in development, plug-in structure, plug-in parts, Eclipse's tooling for the plug-in development, Eclipse libraries for the GUI development, interactions of the plug-in and the workbench, extension points.

The target audience for this manual is developers who wish to extend and/or modify plug-in's features and/or code structure.

Short workbench overview

The Eclipse platform is structured around the concept of plug-ins. Each subsystem in the platform is itself structured as a set of plug-ins that implements some key functions. Most of changes are visible in the Eclipse Workbench.

The Workbench contains one or more WorkbenchWindows, each of which contains zero or more WorkbenchPages. The WorkbenchWindow supplies the trim widgets, and the WorkbenchPage supplies the window contents. In theory, a WorkbenchWindow can contain any number of pages, but in practice there is never more than 1 page in a window.

Views and editors are owned by the page, through a ViewFactory and EditorManager respectively. EditorManager stores the list of editors and their shared resources, and ViewFactory stores a reference counted list of views. The workbench works in terms of EditorReferences and ViewReferences, and in this article the terms "editor" or "view" will refer to these classes specifically. In situations where the distinction between editors and views is not important, we will simply use the term "part". The implementation of the part (typically an IEditorPart or IViewPart) is created lazily when it is first needed. A part reference exists for every tab but the implementation is only created the first time it becomes visible.

The page owns a set of perspectives. Perspectives contain a layout and information about what action sets to enable. Although perspectives appear to contain views and the editor area, they only own a layout. The page itself maintains a reference count for how many perspectives are using each view, and has complete ownership of the parts and editor area.

Not shown in figure 1 are the classes PerspectiveHelper and EditorAreaHelper. These classes exist largely for historic purposes, and in this article we will treat the former as though it were part of the Perspective class and the latter as part of the WorkbenchPage class.

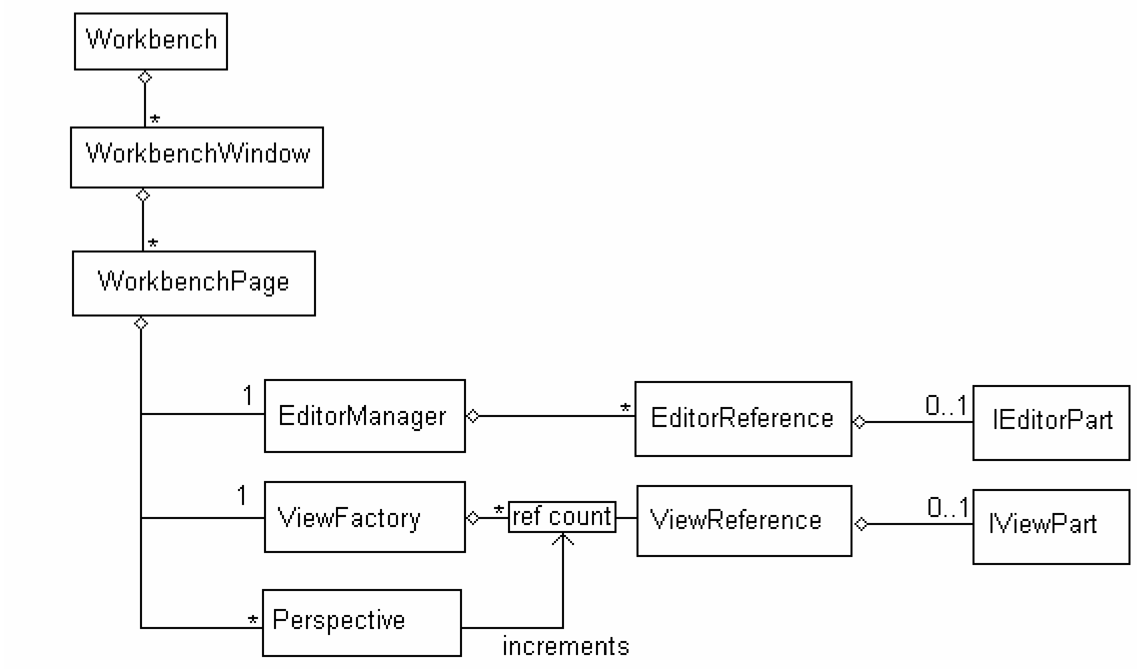


Figure 1: Ownership of views and editors

User interface integration

The developed plug-in contributes mainly to the UI of the Eclipse IDE. It closely interacts with Eclipse perspectives, views, editors, wizards, and actions. The UI integration allows the plug-in to participate with the rest of Eclipse UI as if it is designed as a single application.

UI integration is accomplished by building the user interface for the tool using the Eclipse UI frameworks, and integrating with the Workbench through its extension points. Most application development tools (in fact, most tools in general) operate on hierarchically structured data. The Eclipse platform exploits this commonality by providing a number of reusable viewers that can be easily customized through content providers to provide a user interface for a tool's object model. There are many other facilities in the Eclipse platform that support tool UI development including SWT, the Source Editing Framework, contribution frameworks, wizard frameworks, etc. The Workbench provides extension points for adding new views, editors, wizards, preference pages, builders, project natures, perspectives, splash screens, etc. Tools can also extend other tool's menus as well as add extensions to their extension points.

Chapter 1 Structure of the plug-in

The core of the plug-in is located in the project named plugin.core. This project includes all main functionality of the plug-in and all configuration details for the plug-in deploying. The plug-in folder structure consists of the following:

```
Root folder/
                /bin/
                /icons/
                /lib/
                /META-INF/
                /plugin.xml
```

This structure is used when the plug-in is being deployed and installed by the Eclipse IDE. The icons/ folder contains all icons, which are used in the project. The lib folder contains all jar libraries which are needed for the proper plug-in work. The META_INF/ folder contains just the manifest.mf file, which is the main file for the plug-in initialization. The plugin.xml file defines all extension points for the Eclipse UI contribution.

Class packages structure

The java class files are located in the bin/ folder. The package structure of the class files is the following:

org.objectweb.dsrg.sofa.repository.plugin – the root package of the plug-in. It contains the Activator.class file which is invoked when plug-in is initialized.

org.objectweb.dsrg.sofa.repository.plugin.actions – action commands for the repository related actions.

org.objectweb.dsrg.sofa.repository.plugin.cushion – the package which contains all cushion related implementations. This includes implementation of cushion graphic elements, cushion data handlers, cushion structure handlers, cushion listeners, container classes etc.

org.objectweb.dsrg.sofa.repository.plugin.cushion.data – the package contains file to save changes made using the properties editor. The package is used for each data saving event.

org.objectweb.dsrg.sofa.repository.plugin.cushion.elements – cushion logic containers. They are used to keep all data, received from the adl files.

org.objectweb.dsrg.sofa.repository.plugin.handlers – structure handlers for the cushion data

org.objectweb.dsrg.sofa.repository.plugin.cushion.listeners – cushion related listeners, like rename listener for cushion graphic elements, moving listener, etc.

org.objectweb.dsrg.sofa.repository.plugin.cushion.queues – cushion queue of graphic elements.

org.objectweb.dsrg.sofa.repository.plugin.readers – cushion readers from the adl files.

org.objectweb.dsrg.sofa.repository.plugin.cushion.stubs – graphic elements, which represents cushion objects.

org.objectweb.dsrg.sofa.repository.plugin.cushion.utils – miscellaneous utility classes used to support cushion related part of the plug-in.

org.objectweb.dsrg.sofa.repository.plugin.editors – editors, which contributed to Eclipse's workbench.

org.objectweb.dsrg.sofa.repository.plugin.editors.java – java source code editor with syntax highlight.

org.objectweb.dsrg.sofa.repository.plugin.editors.xml – xml source code editor with syntax highlight.

org.objectweb.dsrg.sofa.repository.plugin.perspectives – perspectives used in the plug-in. Actually it is one SOFA 2.0 perspective.

org.objectweb.dsrg.sofa.repository.plugin.popup.actions – actions, which are invoked when Project Explorer context menu is called.
org.objectweb.dsrg.sofa.repository.plugin.views – contribution to the Eclipse's views.
org.objectweb.dsrg.sofa.repository.plugin.widgets – graphical elements which are used when it is needed to work directly with repository server.
org.objectweb.dsrg.sofa.repository.plugin.widgets.handlers – data handlers for logical part of repository widget representation.
org.objectweb.dsrg.sofa.repository.plugin.widgets.listeners – listeners which are listening to changes on the properties view or drawing area.
org.objectweb.dsrg.sofa.repository.plugin.widget.queues – queues with widget graphical elements and links.
org.objectweb.dsrg.sofa.repository.plugin.widget.utils – miscellaneous utility classes which are used to support the part of the plug-in which is related to the direct work with repository.
org.objectweb.dsrg.sofa.repository.plugin.wizards – wizards used in project (independently from purpose).

Chapter 2 Plug-in configuration

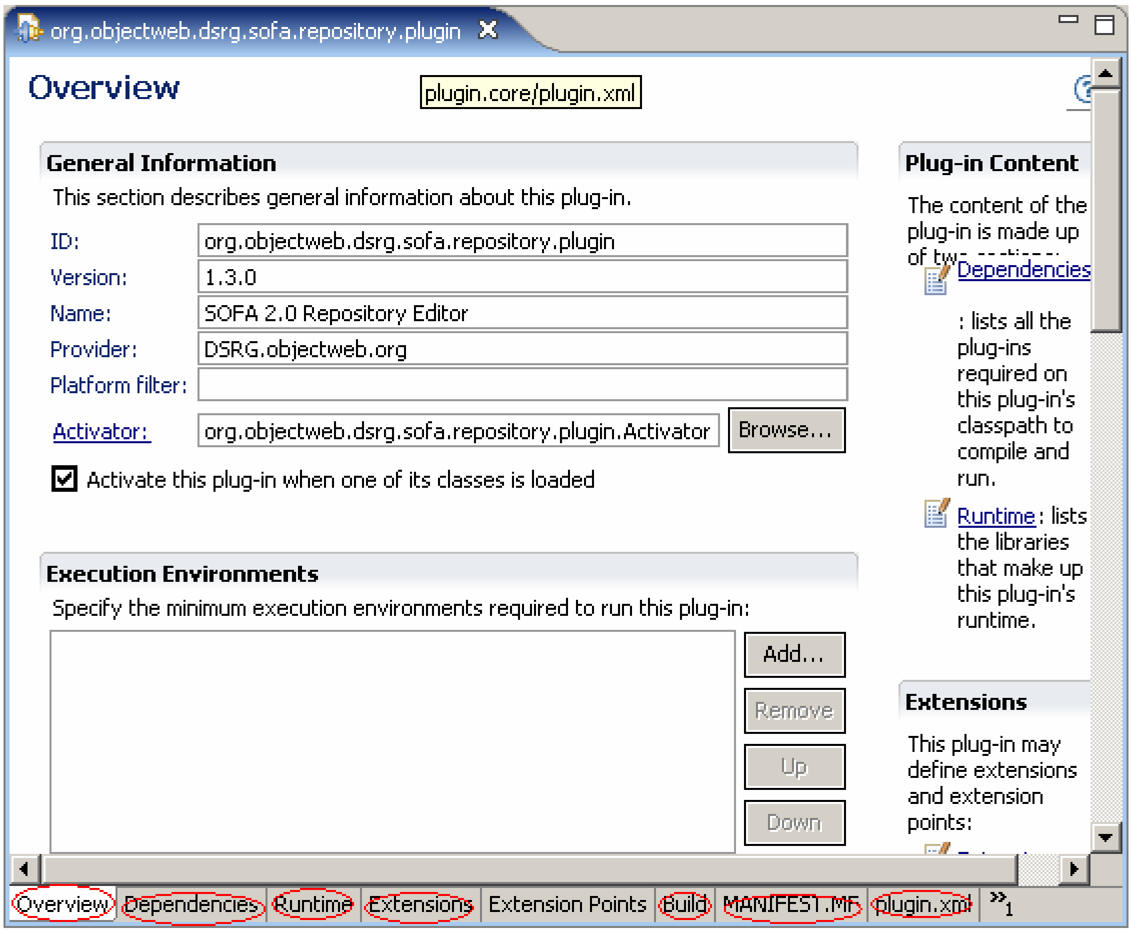
Bundle configuration

The plug-in main configuration is described in the META-INF/manifest.mf file. This wile is required in order to compile valid working plug-in bundle. The manifest file contains following fields:
Manifest-Version – the version of the manifest file syntax. Usually it is *1.0*
Bundle-ManifestVersion – the version of the manifest bundle.
Bundle-Name – the name of the bundle, which will be shown in the configuration manager. It will be also used in the bundle description, osgi debugging console and other technical and debugging parts of the Eclipse IDE. In our case the bundle is called *SOFA 2.0 Repository Editor*.
Bundle-SymbolicName – this is in fact the path to the root package of the plug-in. Also it indicates if the plug-in is singleton or not. If plug-in is defined as a singleton, then it can be initialized only once in the workbench.
In our case the value is: *org.objectweb.dsrg.sofa.repository.plugin;singleton:=true*
Bundle-Version – the version of the bundle. It is changed each time the new version is released. Now we have version *1.3.0*.
Bundle-Activator – the name of the plug-in's activator. In fact it can be any file within the bundled package. So we haveto specify explicitly, which file is a bundle activator.
In our case this is *org.objectweb.dsrg.sofa.repository.plugin.Activator*.
Bundle-Vendor – the vendor of the bundle. For us this is DSRG.objectweb.org.
Bundle-Localization – it indicates if plugin is developed as a plug-in for Eclipse IDE, or if it is RCP, or some other kinds of possible plug-ins. For this project the value is *plugin*.
Require-Bundle – list of required bundles. The developed plug-in requires only those bundles, which are originally supplied with Eclipse IDE. Nothing additional should be installed.
Eclipse-LazyStart – indicates, whether the plug-in should be activated only when it is needed, or just at once the Eclipse is initialized. It is recommended to allow lazy start to free system resources. We allow it. The value is set to *true*.
Eclipse-RegisterBuddy – the root package: *org.objectweb.dsrg.sofa.repository*.
Export-Package – list of packages which are exported along with given plug-in. This means all java packages which are included in the project's implementation. The list should start not from the root package, but from the . to allow class path reference.
Bundle-ClassPath – list of jar libraries which should be included in plug-in's class path. In our case this is the whole lib/ folder. But all jars should be written one by one here. No wildcards are allowed.

All those fields are generated when the plugin.xml file is edited via plugin.xml editor in the Eclipse IDE. But some times it is required to edit it manually when generation is out of synch with the actual file. For example, when big refactoring is done and many packages are moved, renamed, or deleted. Then manifest.mf can be not updated automatically by the refactoring manager.

Extension points

The other very important configuration file is the plugin.xml file, which contains references to all extension and contribution points which are used in the project. The plugin.xml file is located in the project's root. Eclipse provides very user friendly plugin.xml editor. It allows making changes in all fields of the plugin.xml and automatically generates build.xml file for ant build and generates the manifest.mf file.


org.objectweb.dsrg.sofa.repository.plugin

Overview

plugin.core/plugin.xml

General Information

This section describes general information about this plug-in.

ID:	org.objectweb.dsrg.sofa.repository.plugin
Version:	1.3.0
Name:	SOFA 2.0 Repository Editor
Provider:	DSRG.objectweb.org
Platform filter:	
Activator:	org.objectweb.dsrg.sofa.repository.plugin.Activator

☑ Activate this plug-in when one of its classes is loaded

Execution Environments

Specify the minimum execution environments required to run this plug-in:

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.

Extensions

This plug-in may define extensions and extension points:

Figure 2: plugin.xml editor

The editor is divided into pages. Each page provides separate group of properties to edit. The first – overview tab – is for general properties, like name, version, provider, activator class etc. This data is mainly used to generate the manifest file. The Dependencies tab gives possibility to edit plug-in dependencies list – the list of plug-ins which must be loaded into workbench in order to support correct functioning of the current plug-in This tab is used to make changes to the manifest file. The Runtime tab gives possibility to edit list of packages which should be exported with the current plug-in and define jar libraries which should be loaded in order for the plug-in to work. The jar libraries are not plug-ins, just some 3d parties libraries. This tab is used in the manifest file. The Extension tab gives possibility to add/edit/remove extensions to the workbench. This tab is used in the plugin.xml file. Using this tab it is possible to declare contributions to almost any part of the plug-in. Eclipse allows contributions to any place of the workbench – editors, views, actions, context menus, pop-up menus, tool bars, main menu, properties list, and many others. The Build tab gives possibility to define which files and folders will be included in the source and binary builds. Source build (which is obvious) is intended to include the folder with source java files. And on the other hand the binary build does not include the source folder. This is the main difference. The manifest.mf and plugin.xml tabs provides possibility of the raw editing of appropriate files.

Chapter 3 Plugging into the workbench

Activator

The Activator file is indicated in the manifest under the *Bundle-Activator* property. This file should extend *org.eclipse.ui.plugin.AbstractUIPlugin*. This class must define *public static final String PLUGIN_ID* which is the unique plug-in identifier in the current workbench context. The Activator should also implement *start()* and *stop()* methods which is necessary to make plug-in accessible from the Eclipse platform and there should be two static methods – *getDefault()* which should return the reference to the current plug-in activator and *getImageDescriptor(String path)* which should return descriptor for an image by the given path. All other methos can be called from the super class.

The Activator class can be used within the plug-in to receive plug-in related path to resources or image descriptor. It also can stop the plug-in using the *stop()* method. But this can only happened if user will use Configuration Manager to disable the plug-in.

Editor contribution

plugin.xml reference:

```
<editor
class="org.objectweb.dsrg.sofa.repository.plugin.editors.MultiPageEditor"

contributorClass="org.objectweb.dsrg.sofa.repository.plugin.editors.MultiPage
EditorContributor"
            default="true"
            extensions="xml"
            icon="icons/small.gif"

id="org.objectweb.dsrg.sofa.repository.plugin.editors.MultiPageEditor"
            name="Repository Project">

      </editor>
```

The editor must implement IEditorPart. We actively use only one editor in the plug-in. This is the MultiPageEditor. In our case the editor extends MultiPageEditorPart implements IResourceChangeListener which gives possibility of multiple pages for one document. The extension is declared so, that the editor opens all files which have the .xml extension. The editor is set to be default editor for xml files. The unique identifier of the editor is the same as its package path. The multi page editor must implement the init() method, which is called when the editor is initialized, the doSave() method, doSaveAs() method, isDirty() method which is called to check if there is something to save. The main method to be implemented is createPages(). This method is used when some file is open for editing. It should create all pages which are defined to be created for such a file type. Pages can be created of a very different kind. This can be properties pages, draw/graphic editing pages, text editor pages and others.

It is possible to contribute to the new multi page editor. For example, add some buttons as a toolbar to the top of the editor. For this reason the MiultiPageEditorContributor is used. It is located in the same package as the MultiPageEditor class. We do not use much contributions to the editor, it just implements all editor related actions like undo, redo, copy, paste etc. The contributor class extends the `*org.eclipse.ui.part.MultiPageEditorActionBarContributor*` class. But there are other possible contributions in the *org.eclipse.ui.part* package.

View contribution

plugin.xml reference:

```
<view
            allowMultiple="false"
            category="org.objectweb.dsrg.sofa.repository.plugin"

class="org.objectweb.dsrg.sofa.repository.plugin.views.PropertiesView"
            icon="icons/pview.gif"

id="org.objectweb.dsrg.sofa.repository.plugin.views.PropertiesView"
            name="Properties View">
      </view>
```

Each view in the Eclipse plug-in must extend org.eclipse.ui.part.ViewPart. In the plug-n we use 5 views. All of them are in the *org.objectweb.dsrg.sofa.repository.plugin.views* package. The main method in the view is *createPageControl(Composite parent)*. This method is used to create all UI layouts which will be visible on the view. It is possible to create any GUI on the view part. It is even possible to create edit-like pane on it. We use this trick several times in order to use views instead of editors. Eclipse provides only one shared editor placeholder for all projects. And we required flexible parts layout in the workbench. That is why we should use views instead of editors. According to the Eclipse's specification it is possible to create the editor part only in center of the workbench, and all views should surround the editor. And views can be created in any part of the workbench. So, it is much more comfortable.

The syntax of the xml contribution says, that it is forbidden to open more than one instance of a view (this is valid for all 5 views used), it also assigns icon to a view (icons differ for each view), the id of each view is the same as its full package name (but it can be different, but it must be unique within the plug-in). And the name of a view is assigned in the xml contribution, not from the view itself. The view can be opened from the Window->Open view->other->SOFA 2.0. And then the name of the view to be open is listed there. That is why it is useful to name views outside of according classes.

Wizard contribution

plugin.xml reference:

```
<wizard
            canFinishEarly="true"
            category="org.objectweb.dsrg.sofa.repository.plugin"

class="org.objectweb.dsrg.sofa.repository.plugin.wizards.NCushionInterfaceTyp
eWizard"
            hasPages="true"
            icon="icons/newi.gif"
            id="plugin.core.wizard3"
            name="New ITF"/>
```

Any wizard manager in the project should extend at least *org.eclipse.jface.wizard.Wizard* class in order to be identified as wizard manager. There are more complicated wizards to be extended (like new project wizard, new file wizard, etc) but we use primitive wizard class because we need totally different functionality. There are 8 wizard managers in the project. They are located in the *org.objectweb.dsrg.sofa.repository.plugin.wizards* package. They are used to create new frame, new architecture, new subcomponent, new frame from the architecture centric view, new cushion frame, new cushion architecture, new cushion interface type, and the new cushion project. All of them extends the Wizard class. There are 3 methods here to be implemented. These are *addPages(), canFinish(), performFinish()*. Those methods should be sub classed in order for the wizard manager to work properly. But the *canFinish()* is called only once – when wizard is created. To check if finish can be really performed, it is required to implement another method, directly in the class which represents wizard page. Please note, that after the finish is performed, the wizard manager itself is not disposed and all gathered data can be taken from it. So, there is no need to implement actions which should be done according to the wizard's results directly in the wizard manager class (until there is a real need for that).

The syntax of the xml contribution says, that it is allowed for the wizard to be finished before all steps are passed thru (this means, not only the Next> button is available in the middle of the wizard, but also the Finish button is active if all needed data is already gathered). Also it is indicated, that the wizard has pages (this is obvious, each wizard should have pages). The id of the wizard is assigned, now it differs from the qualified class name in the package, but it is still unique. The name is assigned outside of the wizard class, in the contribution details. This is mainly done because it is possible to open wizard using the New->Other->SOFA projects option. And the wizard class is inactive then.

Wizard pages

Each wizard should have pages. It is possible to add many pages to one wizard and it is possible to decide which page to display next dependably on the gathered data. Each wizard page should extend the *org.eclipse.jface.wizard.WizardPage* class. This class provides the *createControl()* method which is intended to be overridden. This method allows creating GUI of the page. It is possible to create any valid GUI. But it is not recommended to create editors on wizard pages.

Also it is useful to implement change listener to check if all proper data is gathered. The *canFinish()* method from the wizard manager is performed only once per wizard lifecycle. So, it is important to track all changes via custom listener.

In the project all wizard pages are kept in the same packages as wizard managers.

Action contributions

plugin.xml reference:

[For actions which implements IObjectActionDelegate interface]

```
<action

class="org.objectweb.dsrg.sofa.repository.plugin.popup.actions.CommitAction"
               icon="icons/start.gif"
               id="org.objectweb.dsrg.sofa.repository.plugin.newAction"
               label="commit"

menubarPath="org.objectweb.dsrg.sofa.repository.plugin.menu1/group1"/>
```

[For actions which implements IWorkbenchWindowActionDelegate]

```
<action
               label="&amp;Save all changes to repository server"
               icon="icons/saveall.gif"

class="org.objectweb.dsrg.sofa.repository.plugin.actions.RepositoryAction"
               tooltip="Save all changes to repository server"
               toolbarPath="sampleGroup"

id="org.objectweb.dsrg.sofa.repository.plugin.actions.RepositoryAction">
```

```
        </action>
```

There are two possibilities for actions to be declared – implement IObjectActionDelegate, this is interface for an object action that is contributed into a popup menu for a view or editor. It extends IActionDelegate and adds an initialization method for connecting the delegate to the part it should work with. The other way is to implement IWorkbenchWindowActionDelegate. This is interface for an action that is contributed into the workbench window menu or tool bar. It extends IActionDelegate and adds an initialization method for connecting the delegate to the workbench window it should work with.

In the project we use both types of actions. Instances of the first type are kept in the *org.objectweb.dsrg.sofa.repository.plugin.popup.actions* package and the other one is in the *org.objectweb.dsrg.sofa.repository.plugin.actions* package. The first type of actions is used for contribution to the context menus and the second type is used for contributions to the main application menu and the main tool bar.

For the first XML snippet (actions which implements IObjectActionDelegate interface) the syntax of the XML contribution says, that the action is triggered when a pop-up menu invoked. The action is assigned to the menu1/group1 action group. Also the action has assigned icon, and label. Label here acts as a name. It is visible to user when s/he is going to click a pup-up menu point. The full syntax (not introduced here, please refer to the original **plugin.xml** file) says, that the title of the group is SOFA repository and it is shown between other menu choices when the popup menu is triggered over the Package Explorer.

For the second XML snipped (actions which implements IWorkbenchWindowActionDelegate interface) the syntax of the XML contribution says, that the action is triggered when a button on the main toolbar is clicked, or an option from the main application menu is selected. We do not contribute to the main application menu in the plug-in, but we put the button on the main toolbar to save all changes on the repository part. The contribution defines tool-tip text and icon for the button. These are main changes here.

Perspective contribution

plugin.xml reference:

```
<perspective

class="org.objectweb.dsrg.sofa.repository.plugin.perspectives.PerspectivesFac
tory1"
            fixed="true"
            icon="icons/perspective.gif"
            id="Sperspective1"
            name="SOFA 2.0">
         <description/>
      </perspective>
```

A perspective is a visual container for a set of views and editors (parts). These parts exist wholly within the perspective and are not shared. A perspective is like a page within a book. It exists within a window along with any number of other perspectives and, like a page within a book, only one perspective is visible at any time.
In the project we use one perspective, which is stored in the *org.objectweb.dsrg.sofa.repository.plugin.perspectives* package. This perspectives defines placeholders for views used in the plug-in. It allocates placeholders for 2 views and ties other 5 views to the perspective view. So, when the perspective is opened there are 5 visible views on it

and it is also possible to add other 2 views on predefined placeholders. There is also predefined place for an instance of the multi page editor which is used in the cushion projects. When the editor is invoked, its placed on the predefined place.

To be able to layout views and editors properly a perspective class realization should implement IPerspectiveFactory interface.

This layout is performed in the perspective factory, a Java class associated with the perspective type. When the perspective is initialized, it consists of an editor area with no additional views. The perspective factory may add new views, using the editor area as the initial point of reference.

The size and position of each view is controlled by the perspective factory. These attributes should be defined in a reasonable manner, such that the user can resize or move a view if they desire it.
The syntax of the XML contribution says that perspective has fixed layout (fixed=”true”), and there is assigned icon and name for it. The id is differs from the qualified package name, but still it is unique. Also it is possible to add perspective description to the contribution, but now the appropriate tag (<description/>) is empty.

Chapter 4 SWT and jFace for GUI development

Standard Widget Toolkit (SWT) provides a platform-independent API tightly integrated with the operating system's native windowing environment. SWT's approach provides Java developers with a cross-platform API to implement solutions that "feel" like native desktop applications. This toolkit overcomes many of the design and implementation trade-offs that developers face when using the Java Abstract Window Toolkit (AWT) or Java Foundation Classes (JFC).

The JFace toolkit is a platform-independent user interface API that extends and interoperates with the SWT. This library provides a set of components and helper utilities that simplify many of the common tasks in developing SWT user interfaces. This toolkit includes many utility classes that extend SWT to provide data viewers, wizard and dialog components, text manipulation, and image and font components.

It were used several widgets and UI factories in the project. The very important here are UI elements like scrolled forms, sections, form toolkit factory, table and table layout.

Form Toolkit

The toolkit is responsible for creating SWT controls adapted to work in Eclipse forms. In addition to changing their presentation properties (fonts, colors etc.), various listeners are attached to make them behave correctly in the form context.

In addition to being the control factory, the toolkit is also responsible for painting flat borders for select controls, managing hyperlink groups and control colors.

The toolkit creates some of the most common controls used to populate Eclipse forms. Controls that must be created using their constructors, adapt() method is available to change its properties in the same way as with the supported toolkit controls.

Typically, one toolkit object is created per workbench part (for example, an editor or a form wizard). The toolkit is disposed when the part is disposed. To conserve resources, it is possible to create one color object for the entire plug-in and share it between several toolkits. The plug-in is responsible for disposing the colors (disposing the toolkit that uses shared color object will not dispose the colors).

FormToolkit is normally instantiated, but can also be sub classed if some of the methods need to be modified. In those cases, super must be called to preserve normal behavior.

We use the toolkit in the project almost everywhere in views. This is good way to create flat web-like look of forms.

Sections

One of the most versatile custom controls in Eclipse Forms (and seen in all PDE editors) is Section. It extends the expandable composite and introduces the following concepts:

- Separator - a separator control can be created below the title
- Description - an optional description can be added below the title (and below the separator, if present)
- Title bar - a title bar that encloses the section can be painted behind the title (note that separator and title bar should not be used simultaneously)

The sections are widely used within the project. It is important to remember some notes about sections. Sections are not containers. It is required to create some composites over a section to place widgets on it. It is required to set the composite as a client for the section when the composite is prepared and all widgets are correctly placed on it. It is required to use layout with sections. Otherwise you will see nothing. Layout can not be omitted.

Scrolled form

ScrolledForm is a control that is capable of scrolling an instance of the Form class. It should be created in a parent that will allow it to use all the available area (for example, a shell, a view or an editor).

Children of the form should typically be created using FormToolkit to match the appearance and behavior. When creating children, use a form body as a parent by calling 'getBody()' on the form instance. Example:

```
FormToolkit toolkit = new FormToolkit(parent.getDisplay());
ScrolledForm form = toolkit.createScrolledForm(parent);
form.setText("Sample form");
form.getBody().setLayout(new GridLayout());
toolkit.createButton(form.getBody(), "Checkbox", SWT.CHECK);
```

No layout manager has been set on the body. Clients are required to set the desired layout manager explicitly.

The main advantage of the scrolled composite is that it supports automatic reflow on content changed. The scrolled composite does not support this. It is required to implement appropriate listeners to listen when content is bigger than a scrolled area. That is why all scrolled composites in the project where changed to scrolled forms.

Chapter 5 Feature and site projects

In order to make possible plug-in installation via the updates manager it is required to develop two related projects for the main plug-in project. The Eclipse IDE requires the feature and site projects to be developed and exported to the installation state in order to make possible plug-in installation via the update manager. Some times it is enough to have just site project to install plug-in properly. But for the current project we need both site and feature project to be developed.

Feature project

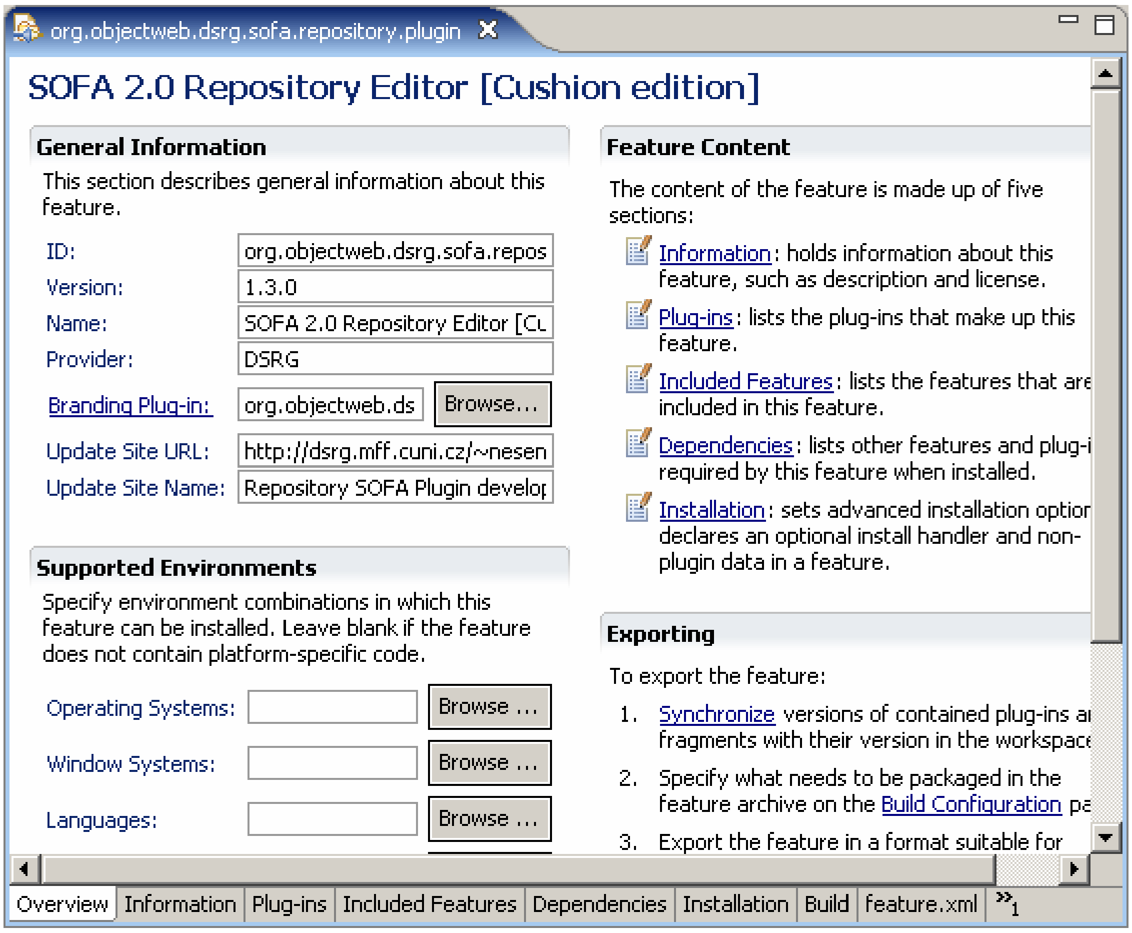
Features are important because they are the unit of Eclipse configuration management, they support product branding, and they are part of how products build customized solutions on top of the Eclipse Platform. Using features you can:

- Identify, based on feature branding, which provided the different functions that are available when you are working with an Eclipse-based product
- Build on product branding capabilities to further customize Eclipse
- Automate tasks in the plug-in development environment
- Dynamically change the configuration for a given workspace by managing the enabled/disabled state of root features, or features included by another but defined as optional, using the Install/Update perspective

The current project uses only simple feature to make installation process user friendly. The feature project includes only two files – build.properties and the feature.xml. The feature.xml file is the most important because it is exported to the feature/ folder as a jar bundle and shows to the installer where and what plug-in to install.

The feature.xml file contains such information as short description of the plug-in usage, copyright information, license information, url for updates, list of required plug-ins for the plug-in in order to be installed, and information about installed plugin – its name, version, download-size, install-size, and id.

The eclipse IDE provides user friendly feature editor to put all this parameters in file. The editor is multi paged and all pages are grouped by appropriate categories. For example, all licensing, copyright and update information is on one page. The general information about the feature is on the firs (overview) page. There is also export wizard link there. When all editing is done, just click on this link to call export wizard and to export feature to the appropriate folder.


org.objectweb.dsrg.sofa.repository.plugin

SOFA 2.0 Repository Editor [Cushion edition]

General Information

This section describes general information about this feature.

ID: org.objectweb.dsrg.sofa.repos
Version: 1.3.0
Name: SOFA 2.0 Repository Editor [Cu
Provider: DSRG
Branding Plug-in: org.objectweb.ds
Update Site URL: http://dsrg.mff.cuni.cz/~nesen
Update Site Name: Repository SOFA Plugin develop

Supported Environments

Specify environment combinations in which this feature can be installed. Leave blank if the feature does not contain platform-specific code.

Operating Systems:
Window Systems:
Languages:

Feature Content

The content of the feature is made up of five sections:

Information: holds information about this feature, such as description and license.

Plug-ins: lists the plug-ins that make up this feature.

Included Features: lists the features that are included in this feature.

Dependencies: lists other features and plug-i required by this feature when installed.

Installation: sets advanced installation optior declares an optional install handler and non-plugin data in a feature.

Exporting

To export the feature:

1. Synchronize versions of contained plug-ins ar fragments with their version in the workspace
2. Specify what needs to be packaged in the feature archive on the Build Configuration pa
3. Export the feature in a format suitable for

Browse...
Overview | Information | Plug-ins | Included Features | Dependencies | Installation | Build | feature.xml

Figure 4: feature editor

Site project

The site project is created just for one purpose – to create the site.xml file. This site should not be packed, it should be just created and edited according to the feature and the project settings. The site.xml file should be placed in the root of the update folder in order for the update manager to update plug-in correctly. The install ready folder structure should look like this:

```
/
        /features/feature_version.jar
        /plugins/plugin_version.jar
        site.xml
```

Very often plugin and feature jars have the same name and version, they are just placed in different folders.

The site.xml is very simple and looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<site>
   <feature
url="features/org.objectweb.dsrg.sofa.repository.plugin_1.3.0.jar"
id="org.objectweb.dsrg.sofa.repository.plugin" version="1.3.0" patch="true">
      <category name="RSP"/>
   </feature>
   <category-def name="RSP" label="RSP"/>
</site>
```

It can be easily created manually without any project. The file contains reference to the feature location and the id and version of the plug-in to be installed. Special attention should be paid to the *patch* property here. If the first version of a plug-in is released, then the patch can be set to false (*patch=”false”*) in all other cases this value should be true.

Chapter 6 Support and troubleshooting

The plug-in is currently maintained by Maksym Nesen. All support requests, bug report, issues, proposals etc should be forwarded to him directly. The contact e-mail is nesen@dsrg.mff.cuni.cz or mavines@oksima.biz. All requests will be carefully solved and all questions will be answered.