

Developing an Eclipse Plug-in for CA Gen Studio

Contents

Introduction.....	2
Sample Plug-in	2
Prepare for developing the plug-in.....	4
Configuring Eclipse	4
Importing the Eclipse project.....	5
Development of the sample plug-in explained.....	7
General Structure of the plug-in.....	8
Declaring the Action Block Navigator view.....	12
Configuring the Action Block Navigator.....	14
Providing contents for Action Block Navigator.....	16
Displaying action block properties.....	21
Developing Text Editor for Action Block Description.....	24
Defining Perspective	28
Testing and deploying a new plug-in	31
Summary.....	31

Introduction

This document shows how to develop an uncomplicated Eclipse plug-in for CA Gen Studio. The Eclipse project with a fully functioning plug-in is provided along with this document. The plug-in shows how to integrate external tools with the existing functionality of Gen Studio, and how to access the local encyclopedia using the JMMI Application Program Interface (API) software for CA Gen.

The Gen Studio is built on top of the Eclipse Rich Client Platform (RCP). Eclipse RCP is designed to serve as an open tools platform and is architected so that its components can be used to build just about any client application. The minimal set of plug-ins needed to build a rich client application is collectively known as the Rich Client Platform. This platform includes the minimal set of plug-ins needed to build a platform application with a user interface. The smallest possible application needs two plug-ins, `org.eclipse.ui` and `org.eclipse.core.runtime`, and their respective prerequisites.

The Gen Studio is based on a dynamic plug-in model and the UI, and is built on top of RCP using the same toolkits and extension points as any another product build using the Eclipse framework. The layout and function of Gen Studio is under control of the plug-in's contributing functionality. You can find all plug-ins constituting the CA Gen Studio in the installation directory. This is a collection of plug-ins developed by CA, plug-ins from other Eclipse Projects, and some plug-ins developed by third-parties.

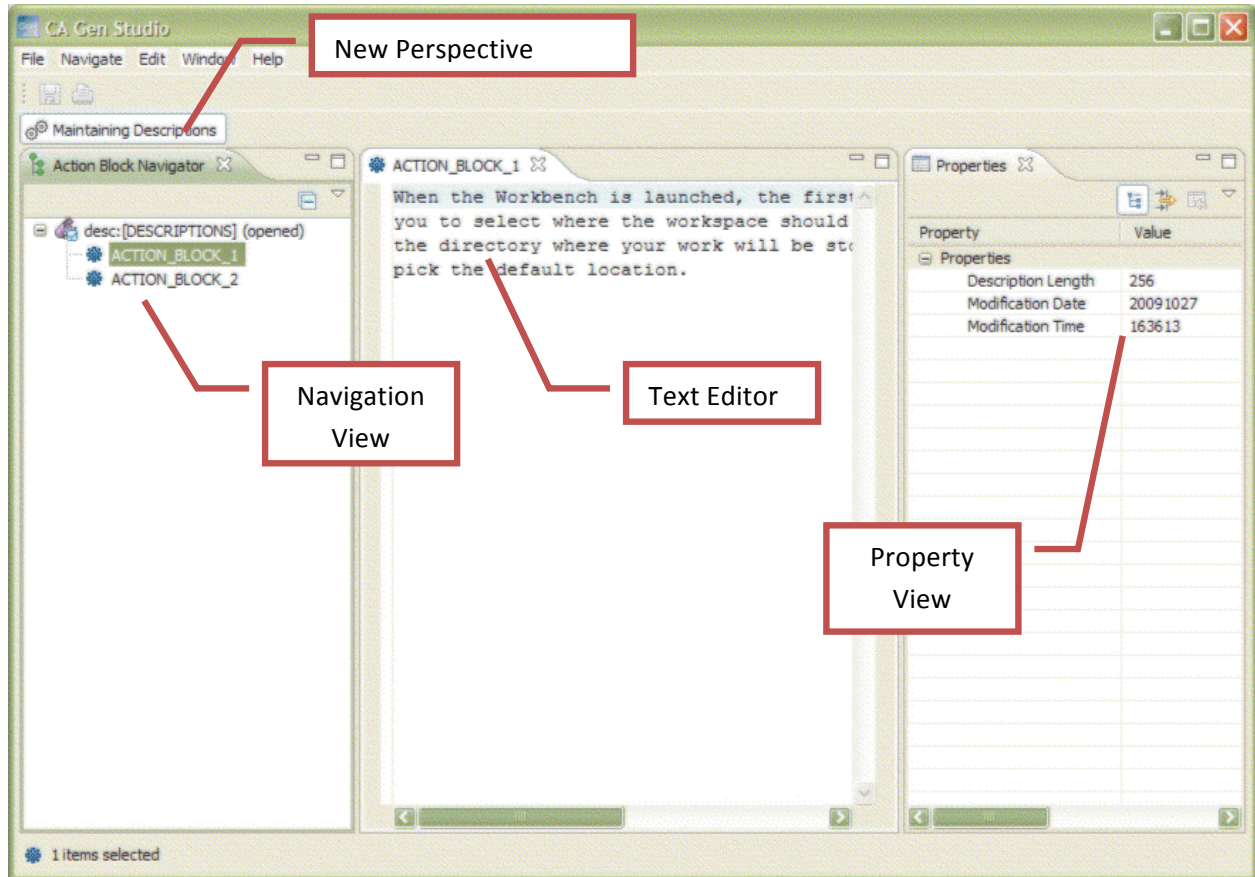
This paper provides basic information, allowing a developer familiar with Eclipse and plug-ins to create plug-ins that extend the functionality of Gen Studio. The JMMI Application Program Interface classes and the RCP framework allow CA Gen customers and Open Initiative partners to write software to retrieve and update encyclopedia information.

Sample Plug-in



As an example, imaginary Company is developing commercial applications using CA Gen. The Company would like to have additional functionality in their Gen Studio allowing action block descriptions to be reviewed by dedicated technical writers. These technical writers will use CA Gen Studio to open and change descriptions in the text editor, and save them back to the model.

After installing the sample plug-in, users are able to open Gen Studio, select dedicated perspective, select action blocks, and edit their descriptions. Model is opened, saved and closed using Gen Studio's standard functionality. Users are presented with a dedicated navigator view showing an expanded list of action blocks for any opened model. Editor is launched after selecting one or more action blocks and selecting Open Editor from the popup menu. One or more editor panes open in the central part of the workbench window. Users are prompted by the system to save contents of the description in the model each time editor is closed and description text has been updated. Additionally, information, such as *(XXXX – we want to tell the user what information to expect)* is displayed in the property view.

The following screen shows CA Gen Studio with sample plug-in installed. Plug-in opens a dedicated perspective consisting of two views and one editor, when activated.



The first **view** shown in the left pane is the *Action Block Navigator* and is used to show opened models, allow expanding, and show list of action blocks within the model. The second view is the *Description Text Editor* for the action block selected. Editors are opened in the editor area of the workspace in the central part of the workbench window. Each action block description can be opened in the text editor by selecting any number of action blocks in the *Action Block Navigator* and selecting action, *Edit Action Block Description* from the popup menu. The third view is *Property* view, which is a generic type of view made available by the Eclipse framework as a standard feature. Selecting items in the *Action Block Navigator* displays some basic properties of the selected items to appear in the *Property* view.

Users can open many instances of the editors for different action blocks. Contents of the editor can be saved at any time by clicking *Save*. The Editor can be closed at any time by clicking *Close*  on the right side of the editor tab. The plug-in displays a prompt asking if the user wants the contents of the changed description to be saved back in the model. All editor tabs have the  symbol in front of the action block name to show that contents of the editor has been changed since the last *Save* action, and its state is considered to be *dirty*. This is standard behavior in many applications developed using the Eclipse framework.

Prepare for developing the plug-in

Developing the plug-in using the Eclipse SDK is a relatively simple process assuming the user has an understanding of the Eclipse plug-in architecture and practical skills in using development tools included with the Eclipse SDK.

CA Gen Studio is built on top of Eclipse 3.4 and all development should be done using the same version of the Eclipse SDK used to build Gen Studio. The sample project was developed using Eclipse Classic 3.4. You can download required software from the following address:

<http://www.eclipse.org/downloads/download.php?file=/eclipse/downloads/drops/R-3.4-200806172000/eclipse-SDK-3.4-win32.zip>

It is recommended that you always check the *CA Gen Technical Requirements* for the correct version of the Eclipse used to build the current version of CA Gen Studio.

The classic Eclipse download has the Eclipse Platform, Java Development Tools, and Plug-in development environment. This download includes sources and both user and programmer documentation. You will need to have CA Gen 8.0 or higher installed on the same workstation. This is all you need to develop the sample plug-in.

Configuring Eclipse

Installation of Eclipse is a simple process

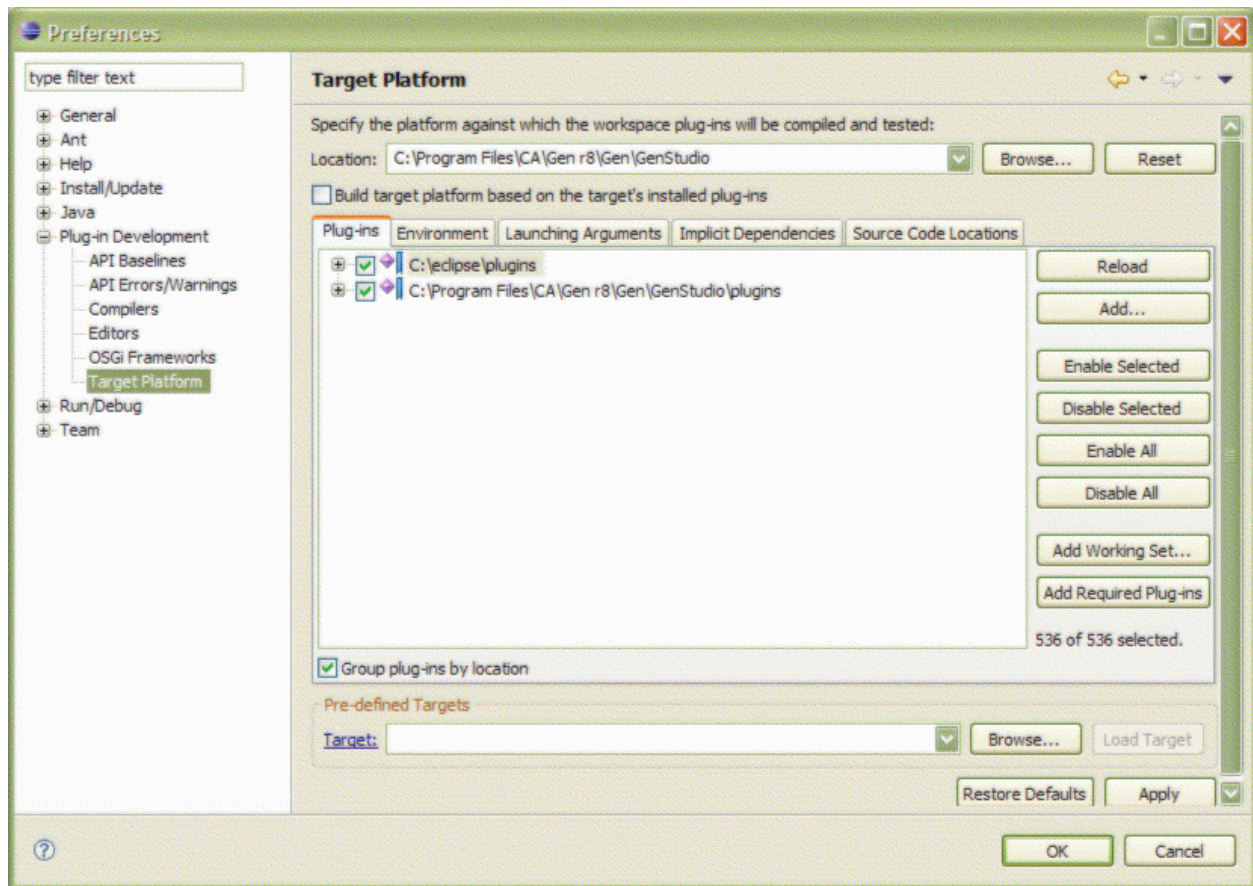
1. Unzip the downloaded file.
2. Create a shortcut for the *eclipse.exe* file.

Eclipse displays the Workspace Launcher asking you to choose the location of your workspace. Initially, the workspace should be any empty folder. Eclipse creates all the required infrastructure of the workspace when opening it for the first time. You will notice that a newly opened workspace does not have any projects.

You need to configure your workspace before you start developing a plug-in to be integrated with CA Gen Studio. Development and all tests should be done in the framework of CA Gen Studio. CA Gen must be installed on the workstation you are using for this project as it needs to be visible to the Eclipse SDK.

1. Open the Preferences dialog box to configure the workspace.
2. Select *Plug-in Development*, *Target Platform* to open the required preference settings.

The Preferences dialog shows how to set preferences.



The Target Platform page allows you to set the target platform plug-in content for the plug-ins you are developing and testing. You need to use a mix of plug-ins from Eclipse Classic and several plug-ins installed on your workstation by CA Gen. Both collections of plug-ins should be referring to exactly the same versions of the plug-ins. If the versions are not the same, the target environment will not function correctly.

The *Plug-ins* tab shows all the plug-ins found at the specified target location. Only the plug-ins checked are in the target content. Unchecked plug-ins are ignored by the Plug-in Development Environment. By default, all plug-ins are checked.

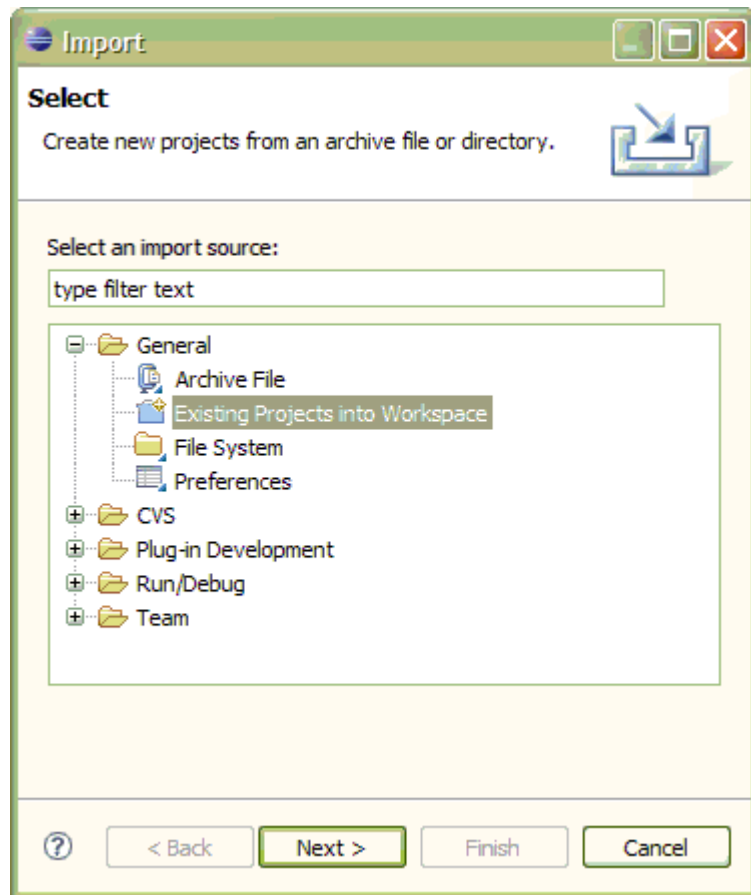
Importing the Eclipse project

A fully functioning Eclipse project for the sample plug-in comes with this document. It is recommended that you import the project to your workspace and review its contents after reading this document.

Use the imported project to verify that your target definition and development environment is correct, and that the plug-in can be deployed and run successfully.

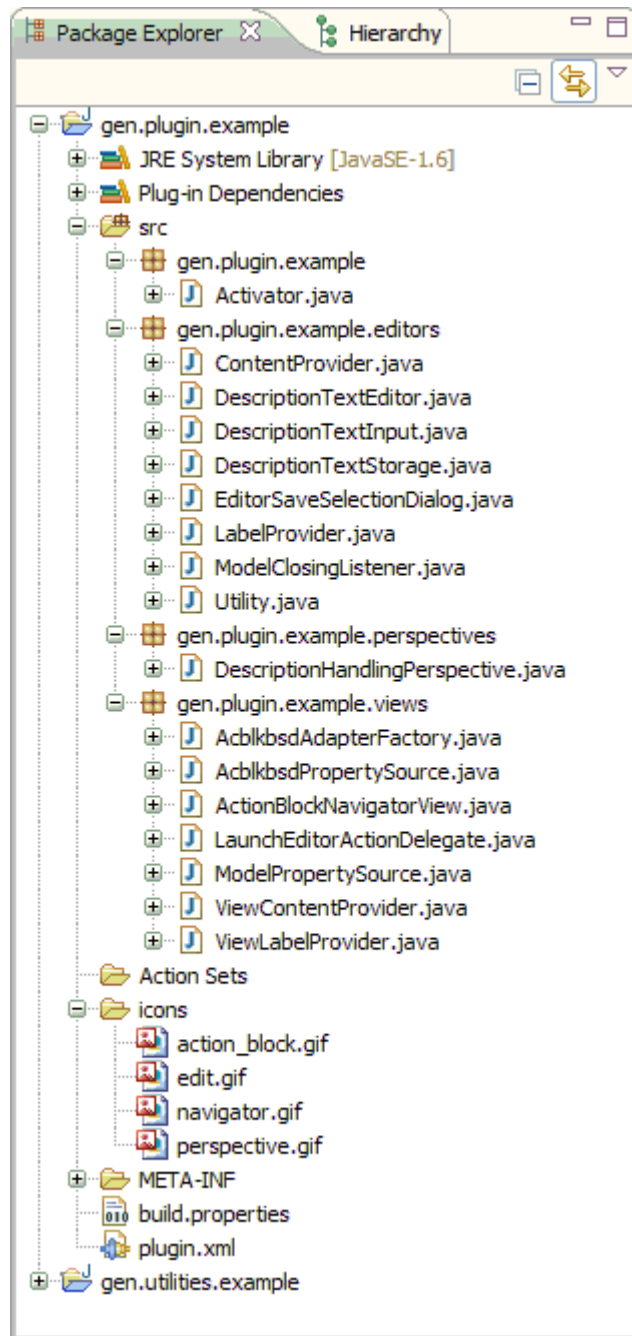
Where do they access the Import Existing Projects into Workspace wizard? We should tell them where to access it.

1. Open the *Import Existing Projects into Workspace* wizard.



2. Click *Next*.
3. Select the location where the project is stored.
4. Select the Copy Projects into Workspace check box so a content is copied into your workspace, keeping the original untouched as a backup.

A new project is created in your workspace and Package Explorer displays the new project.:



Development of the sample plug-in explained

The simplest way to develop and maintain a plug-in is to use the special wizards and editors provided with the Plug-in Development Environment. Wizards are capable of generating a basic plug-in and basic code implementing plug-in functionality. The plug-in manifest editor is used to edit the XML description contained in the plugin.xml file, which is a plug-in manifest file.

There are a number of XML definitions and classes to develop when implementing desired functionality. No one single class is particularly complex to develop. Most classes extend some other classes, reusing functionality that already exists in Eclipse. Complexity is in mutual inter-dependencies between different

runtime components and it is necessary to have good understanding of the functionality of the Eclipse framework.

We can group all the tasks as follows:

- Developing the action block navigator view. Common Navigator Framework (CNF) is used to closely integrate with Gen Studio. This framework was used to create Gen Common Navigator, which is used in the project.
- Showing properties of selected nodes in the navigator. The existing generic Property view is used, which may require some interconnection of components within a plug-in.
- Developing an editor allowing editing text of the action block description. The action block description is a block of text of limited size and Eclipse provides extensive support to build the editor.
- Developing a dedicated Eclipse perspective accommodating our two views and single editor. The perspective is fully integrated with Gen Studio.

The rest of this document will focus on explaining more important parts of the sample plug-in code..

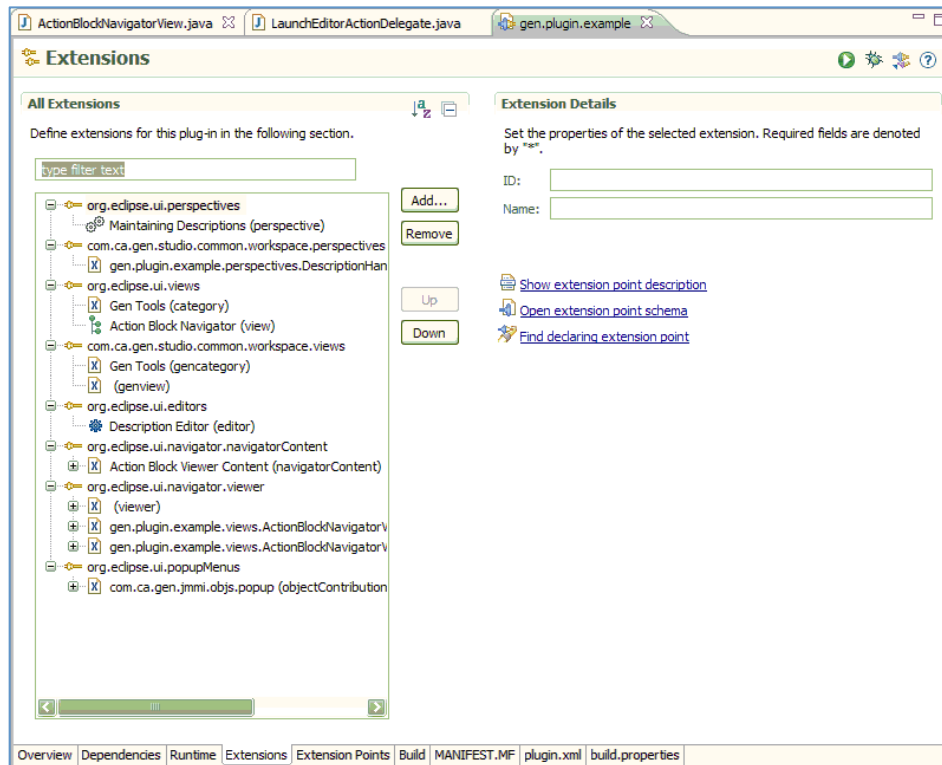
General Structure of the plug-in

The process of developing a plug-in includes adding number elements to the plug-in manifest file `plugin.xml`, and development of a number of Java classes implementing plug-in functionality.

The plug-in, in this example, describes how the plug-in extends the platform and how it implements its functionality. The manifest file is written in XML and is parsed by the platform when the plug-in is loaded into the platform. All the information needed to display the plug-in in the UI, such as icons, menu items, and so on, is contained in the manifest file. The implementation code is loaded only when the plug-in needs to be run; this concept is referred to as *lazy* loading.

The usual method to define extensions is to use the plug-in editor. However, this document shows native xml to explain the definition required to run this plug-in. This is a more consistent way of presenting and allows direct reference to the extension point reference manual.

The Extensions dialog shows all extensions created to allow the sample plug-in to work as designed.



There is a number of classes that need to be developed. The following table contains brief descriptions of all implementation classes used by the plug-in.

Class Name	Function
package gen.plugin.example	
Activator	Our plug-in specifies in the manifest file a concrete subclass of <code>AbstractUIPlugin</code> . This class consists mostly of convenience methods for accessing various platform utilities, and it may also implement <code>start()</code> and <code>stop()</code> methods that define the lifecycle of the plug-in within the platform.
package gen.plugin.example.editors	
ContentProvider	This class implements the <code>IStructuredContentProvider</code> interface and is used to find all opened text editors having a modified action block description. These editors are marked dirty and a special dialog box will use the list of action block descriptions that should be saved in the model.
DescriptionTextEditor	This class extends a standard <code>TextEditor</code> class which implements a basic functionality allowing editing text. Method <code>doSave()</code> is overridden, providing a customized sequence of saving the modified action block description back in the model rather than in the text file.
DescriptionTextInput	This class extends <code>PlatformObject</code> class and implements the <code>IStorageEditorInput</code> interface. A unique class is needed since the implementation of <code>IStorage</code> input to an editor is an object in the Gen model. All editor inputs must implement the <code>IAdaptable</code> interface so extensions can be managed by the platform's adapter manager. Class <code>PlatformObject</code> provides default implementation of the <code>IAdaptable</code> interface.
DescriptionTextStorage	This class extends <code>PlatformObject</code> and implements the <code>IStorage</code> interface. A unique non-default implementation of the interface is needed because the action block description is stored in the Gen model object rather than in the text file. Storage objects have to implement the <code>IAdaptable</code> interface so extensions can be managed by the platform's adapter manager. Class <code>PlatformObject</code> provides default implementation of <code>IAdaptable</code> interface.
EditorSaveSelectionDialog	This class extends <code>ListSelectionDialog</code> , which is a standard dialog soliciting a list of selections from the user. This class is configured with an arbitrary data model represented by content and label provider objects. The <code>getResult()</code> method returns the selected elements. This class is used to display a list of <i>dirty</i> editors and asks

	the user to decide if saving the modified action block contents in the model is required.
LabelProvider	This class implements the <code>ILabelProvider</code> interface and provides icons and action block names to be displayed on the list of opened action block description editors, which are marked as dirty, and need its contents saved.
ModelClosingListener	This class implements <code>IModelListener</code> . Its main function is to provide save contents sequence for all opened action block description editors marked as dirty. Saving contents sequence is activated whenever any of the opened models in the Gen Studio is in the <i>closing</i> state. This class is invoked during start up of the plug-in and it adds itself to the list of <code>ModelManager</code> listeners. This listener unregisters itself from the <code>ModelManager</code> when the plug-in stops.
Utility	This is the utility class, which has a number of static methods producing a list of open and dirty editors in the workspace.
package gen.plugin.example.perspectives	
DescriptionHandlingPerspective	This class implements the <code>IPerspectiveFactory</code> interface and generates the initial page layout and visible action set for a page.
package gen.plugin.example.views	
AcblkbsdAdapterFactory	This class implements the <code>IAdapterFactory</code> interface. The <code>Acblkbsd</code> object can be selected from the <i>Action Block Navigator</i> , but does not implement the <code>IPropertySource</code> interface. Therefore, Property view cannot correctly display properties of the action block. This class allows substitution of <code>Acblkbsd</code> with the other object that can be used by the <i>Property</i> view.
AcblkbsdPropertySource	This class implements the <code>IPropertySource</code> interface and can be used by Property view.
ActionBlockNavigatorView	This class extends the standard <code>GenCommonNavigator</code> class.
LaunchEditorActionDelegate	This class implements the <code>IObjectActionDelegate</code> interface. Implementing this interface for an object action selected from the popup menu for the Action Block Navigator view launches the description text editor for the selected action blocks.
ViewContentProvider	<p>This class implements three interfaces:</p> <ul style="list-style-type: none"> • <code>IStructuredContentProvider</code>, • <code>ITreeContentProvider</code> • <code>IModelSourceListener</code> <p>The main purpose of this class is to provide a list of action</p>

	blocks for the selected data source and display them as children nodes in the tree view of the Action Block Navigator. This class also listens to what model sources are currently opened so the tree view can be refreshed accordingly.
ViewLabelProvider	This class extends the <code>LabelProvider</code> class and is responsible for providing icons and names for action blocks displayed in the tree view of the Action Block Navigator.

Declaring the Action Block Navigator view

Declaring the Action Block Navigator view is the first step. Most perspectives in Eclipse applications have a Navigator view displaying a tree of data. Gen Studio is no different as each perspective displays its own navigator. Some aspects of the navigator content is common between all the various navigators. In particular, the displaying of models with their behaviors is shared between all navigators. The Action Block Navigator is no exception. The Common Navigator Framework (CNF) will be used to develop the Navigator.

We expect the navigator to display a list of models as seen by Gen Studio. Expanding the node of the opened model source displays a list of all action blocks defined in the model. This list allows selection of one or more action blocks and asks the editor to open, displaying the current action block description for editing. The first part of behavior, displaying model sources, should be inherited from the Gen Common Navigator. Data representing action blocks in the model are provided by the purpose written contents provider. The Common Navigator Framework (CNF) provides the extension mechanism.

The First step is always to define a new extension in the `plugin.xml` file.

```
<extension
  point="org.eclipse.ui.views">
  <category
    name="Gen Tools"
    id="gen.plugin.example">
  </category>
  <view
    name="Action Block Navigator"
    icon="icons/navigator.gif"
    category="gen.plugin.example"
    class="gen.plugin.example.views.ActionBlockNavigatorView"
    id="gen.plugin.example.views.ActionBlockNavigatorView">
  </view>
</extension>
```

The extension point `org.eclipse.ui.views` is a standard way to define additional views for the workbench. This extension point, in this case, creates an implementation class for viewing. The `ActionBlockNavigatorView` is the chosen view. Implementation of the view extends the `GenCommonNavigator` class and inherits all behavior. Below is the implementation code.

```

package gen.plugin.example.views;

import com.ca.gen.studio.navigator.GenCommonNavigator;

public class ActionBlockNavigatorView extends GenCommonNavigator {

    public static final String ID = "gen.plugin.example.views.ActionBlockNavigatorView";

    public ActionBlockNavigatorView() {
        super();
    }
}

```

A new view needs to be registered with Gen Studio by adding an extra element to the plugin.xml manifest file. The Gen Studio provides a special extension point `com.ca.gen.studio.common.workspace.views` to register the regular view with Gen Studio.

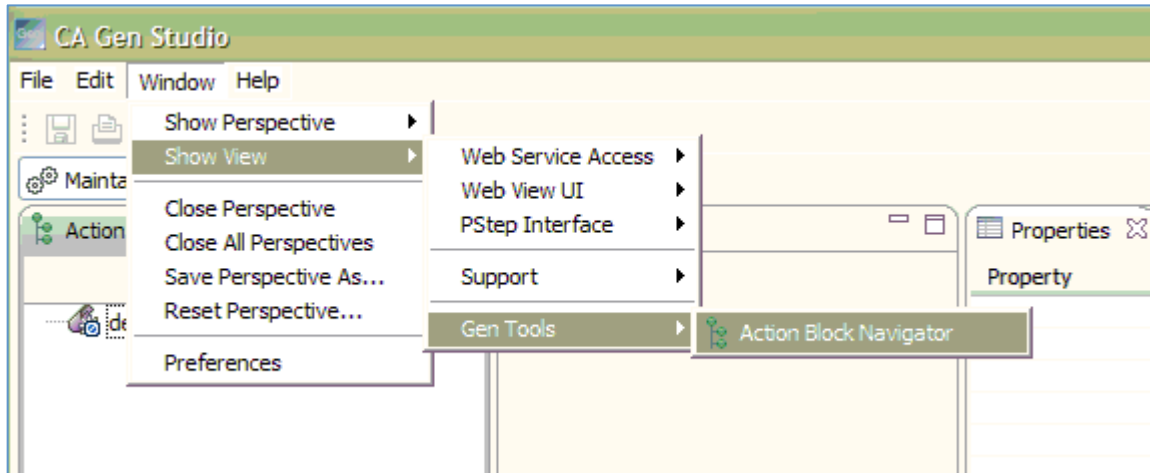
```

<extension
    point="com.ca.gen.studio.common.workspace.views">
    <gencategory
        id="gen.plugin.example.utilities"
        label="Gen Tools"
        mnemonic="T"
        priority="600">
    </gencategory>
    <genview
        gencategoryid="gen.plugin.example.utilities"
        id="gen.plugin.example.views.ActionBlockNavigatorView"
        mnemonic="E"
        priority="100">
    </genview>
</extension>

```

Note that regular views are hidden with Gen Studio unless the view is further extended by a `genview` extension. Once a `genview` extension is defined, the view will display in the Window, Show View menu.

Gen Studio also supports categories for the views so views can be grouped into categories. However, regular categories are hidden within Gen Studio. To group views into categories and sub-categories, reference the newly added `gencategory` from a `genview`. It is possible to add `genview` to any `gencategory` already defined within the product if the `gencategory` contributing plug-in is referenced appropriately. See CA Gen Studion dialog below.



The CA Gen Studio dialog shows that the collection of Gen Studio views has a new category, Gen Tools, and that there is an Action Block Navigator view defined as part of the category.

Configuring the Action Block Navigator

Configuring Action Block Navigator view is the second step. This section defines the contents of the view. The Action Block Navigator uses the tree view to display data. The top level is a list of model sources and is assembled by Gen Studio. The list is updated to reflect the action taken by the Gen Studio users. The list is updated each time a new model opened, closed, was saved or used. The Action Block Navigator is refreshed to show changes to the updated list. Each model source with an opened model allows expanding tree nodes and shows the action blocks defined in the associated model. The list of action blocks for each opened model source is created by the purpose written content provider class.

Some additional entries to the plug-in manifest file are now needed.

```
<extension
  point="org.eclipse.ui.navigator.navigatorContent">
  <navigatorContent
    activeByDefault="true"
    contentProvider="gen.plugin.example.views.ViewContentProvider"
    id="gen.plugin.example.views.ActionBlockNavigator.content"
    labelProvider="gen.plugin.example.views.ViewLabelProvider"
    name="Action Block Viewer Content"
    priority="highest"
    providesSaveables="false">
    <triggerPoints>
      <or>
        <instanceof
          value="com.ca.gen.studio.model.core.source.IModelSource">
        </instanceof>
      </or>
    </triggerPoints>
    <possibleChildren>
      <or>
        <instanceof
          value="com.ca.gen.jmmi.objs.Acblkbsd">
        </instanceof>
      </or>
    </possibleChildren>
  </navigatorContent>
</extension>
</extension>
```

Eclipse framework has a special extension point

`org.eclipse.ui.navigator.navigatorContent` to declare how contents of the navigator view is assembled. The above declaration indicates that we need to develop two special classes: first to provide array of the action blocks for any specific model source and second to provide labels for objects to be displayed in the tree view. Class `ViewContentProvider` is for data objects and class `ViewLabelProvider` is for labels. Both classes are used when object instantiating class `IModelSource` is selected in the navigator view.

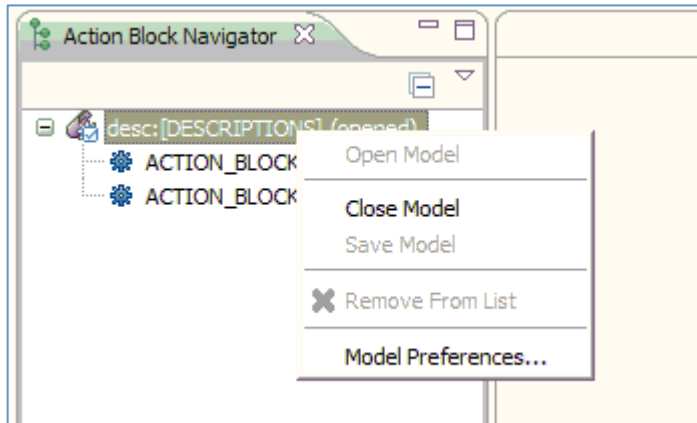
```
<extension
  point="org.eclipse.ui.navigator.viewer">
  <viewer
    viewerId="gen.plugin.example.views.ActionBlockNavigatorView"
    popupMenuId="gen.plugin.example.menu">
    <options>
      <property
        name="org.eclipse.ui.navigator.enforceHasChildren"
        value="true">
      </property>
      <property
        name="org.eclipse.ui.navigator.hideLinkWithEditorAction"
        value="true">
      </property>
    </options>
  </viewer>
  <viewerContentBinding
    viewerId="gen.plugin.example.views.ActionBlockNavigatorView">
    <includes>
      <contentExtension
        isRoot="true"
        pattern="com.ca.gen.studio.model.ui.navigator.modelContent">
      </contentExtension>
      <contentExtension
        isRoot="false"
        pattern="gen.plugin.example.views.ActionBlockNavigator.content">
      </contentExtension>
    </includes>
  </viewerContentBinding>
  <viewerActionBinding
    viewerId="gen.plugin.example.views.ActionBlockNavigatorView">
    <includes>
    </includes>
  </viewerActionBinding>
</extension>
```

The `org.eclipse.ui.navigator.viewer` element defines more configurations for a common viewer. For example, the extension may provide a custom popup menu id and set some other properties. In addition, nested configuration elements give full control over the structure and behavior of the popup context menu. The `viewerContentBinding` binds defined content extensions (through the `navigatorContent` extension point) to viewers (defined through the `org.eclipse.ui.views` extension point).

Above declaration indicates that Gen Studio will provide root level objects and action block navigator contents will provide children nodes.

Above declaration indicates also that our navigator will have two types of pop-menus depending on what node has been selected.

Popup for selected model source will show action applicable to the model like Close Model.

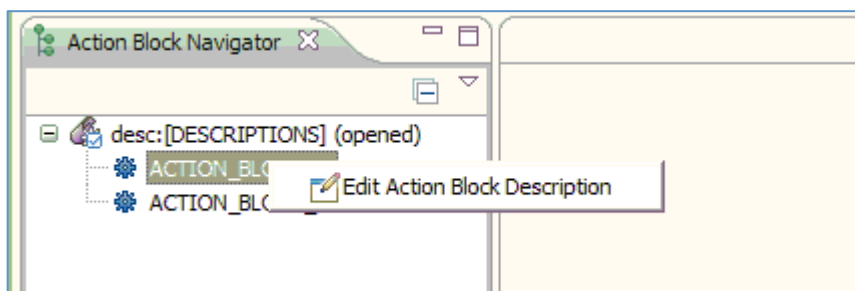


The extension point `popupMenu` defines action whenever action block objects is being selected. Opening text editor to edit action block description is the only action defined. See below.

```
<extension
  point="org.eclipse.ui.popupMenus">
  <objectContribution
    adaptable="false"
    id="com.ca.gen.jmmi.objs.popup"
    objectClass="com.ca.gen.jmmi.objs.Achblkbsd">
    <action
      class="gen.plugin.example.views.LaunchEditorActionDelegate"
      enablesFor="*"
      icon="icons/edit.gif"
      id="gen.plugin.example.menu"
      label="Edit Action Block Description"
      state="false"
      tooltip="Open action block description in the text editor."/>
    </objectContribution>
  </extension>
```

Providing implementation of `LaunchEditorActionDelegate` will allow display the following popup menu when one or more action blocks are selected in the Action Block Navigator.

Popup menu for action block is showing action Edit Action Block Description.



Providing contents for Action Block Navigator

Our next step is to explain purpose and how are working two classes providing contents for the tree view used by the Action Block Navigator to list of action blocks for each opened model.

Model Source and Model

There are two classes representing model in Gen Studio:

- `com.ca.gen.jmmi.Model`
- `com.ca.gen.studio.model.core.source.IModelSource`

The `Model` represents an actual opened model. It is the gateway to the actual model data. It contains listener support for when actual model data changes. The `IModelSource` represents a handle to an actual JMMI Model. The actual model may or may not be open. The `IModelSource` can be thought of as a persistent aspect of a model within the tool, whereas the JMMI Model is the actual model itself.

Two plug-ins makeup the JMMI model-access:

- `com.ca.gen.studio.model.core`
- `com.ca.gen.studio.model.ui`

These plug-ins will need to be added as dependencies for any plug-in needing to access model data.

Models are opened, saved, closed, etc via two manager classes:

- `IModelSourceManager`
- `IModelSourceUIManager`

Ideally, a contributing plug-in would not need to add code to open, save, and close a model. Instead the Gen Studio plug-ins would perform those operations. However, a plug-in should listen and react to these model state changes. For instance when a model is closed, a view or editor displaying model data should refresh or close as needed. The listener must implement the `IModelSourceListener` interface and register the listener with the `IModelSourceManager`.

The main listeners involved with model access are:

- `IModelSourceListener`
- `IModelListener`
- `IDataChangedListener`
- `IUOWListener`

Implementing contents provider

Top level contents (root nodes) are provided by the Gen Studio and we do not have to worry about it. However, we need to develop two classes providing children whenever model source node is selected and expanded.

First class is `ViewContentProvider`. It implements three interfaces:

`IStructuredContentProvider`, `ITreeContentProvider` and `IModelSourceListener`.

Therefore, our `ViewContentProvider` class has to implement a number of methods which are invoked when navigating thru the tree view.

Retrieving information about action blocks

There are two important methods of the `ViewContentProvider` class responsible providing contents when user starts expanding nodes of the tree view. They are `hasChildren()` and `getChildren()`.

Method `hasChildren()` is invoked to check if there any children for the specified parent. The following fragment of code is responsible to do just that.

```
@Override
public boolean hasChildren(Object object) {
    if (object instanceof IModelSource) {
        IModelSource modelSource = (IModelSource) object;
        if (modelSource.isOpened()) {
            Model model = modelSource.getModel();
            List<ObjId> list;
            try {
                list = model.getObjIds(ObjTypeCode.ACBLKBSD);
                return list.size() > 0;
            } catch (EncyUnsupportedOperationException e) {
                e.printStackTrace();
            }
        }
    }
    return false;
}
```

We need to check if object is instance of model source and model is opened. Next we need to retrieve list of object Ids for type ACBLKBSD and check if list is empty or not.

Method `getChildren()` returns array of objects representing children nodes for the provided parent node.

```

@Override
public Object[] getChildren(Object object) {
    if (object instanceof IModelSource) {
        IModelSource modelSource = (IModelSource) object;
        if (modelSource.isOpened()) {
            Model model = modelSource.getModel();
            List<ObjId> list;
            try {
                list = model.getObjIds(ObjTypeCode.ACBLKBSD);
                Acblkbsd[] array = new Acblkbsd[list.size()];
                int i = 0;
                for (ObjId objId : list) {
                    array[i] = (Acblkbsd) MObj.getInstance(model, objId);
                    i++;
                }
                return array;
            } catch (EncyUnsupportedOperationException e) {
                e.printStackTrace();
            }
        }
    }
    return new Object[] {};
}

```

Again, object implementing `IModelSource` interface is used to get model. Again, please notice that only opened models can be searched. All other model sources are ignored and empty object array is returned by the method. Instance of `Model` class is returned by the `getModel()` method. We are looking for list of all object Ids for object type **ACBLKBSD**. Array of objects `Acblkbsd` matching selected list is created and populated by getting the instances of the `Acblkbsd` objects.

Listening to the changes in data sources

A number of methods have to be implemented because our class implements `IModelSourceListener` interface. They are as follows.

- `modelSourceAdded()`
- `modelSourceChanged()`
- `modelSourceClosed()`
- `modelSourceDirtied()`
- `modelSourceOpened()`

Those methods are executed whenever some important action in Gen Studio is taken like opening or closing model sources. Each above action can cause that Action Block Navigator has to be refreshed to reflect current state of the model sources.

These methods are executed only when this class adds itself to list of listeners provided by Gen Studio.

The following snapshot of code shows how to register and unregister contents provider class as listener.

```

public ViewContentProvider() {
    ModelCorePlugin.getDefault().getModelSourceManager()
        .addModelSourceListener(this);
}

@Override
public void dispose() {
    ModelCorePlugin.getDefault().getModelSourceManager()
        .removeModelSourceListener(this);
}

```

The second snapshot shows how to invoke refresh method of the tree viewer from the contents provider.

```

private void refreshViewer() {
    viewer.getControl().getDisplay().asyncExec(new Runnable() {
        @Override
        public void run() {
            viewer.refresh();
        }
    });
}

```

Implementing label provider

Second class we need to provide is ViewLabelProvider. This class extends existing LabelProvider class and we need to override two methods. Method getText () is responsible to provide name of action block and method getImage () suitable icon visualizing action block in the tree viewer.

```

public String getText(Object object) {
    if (object instanceof Acblkbsd) {
        Acblkbsd acblkbsd = (Acblkbsd) object;
        return acblkbsd.getName();
    }
    return "";
}

public Image getImage(Object object) {
    String imageKey = null;
    if (object instanceof Acblkbsd) {
        return Activator.getImageDescriptor("icons/action_block.gif").createImage();
    }
    return PlatformUI.getWorkbench().getSharedImages().getImage(imageKey);
}

```

As you can see in the above snapshot we expect the only object that need to have text and icon returned is object of type Acblkbsd. Text and icon for model source object is assigned by Gen Studio's label provider.

Displaying action block properties

Property view is made as part of our perspective and used to display properties of the selected nodes in the Action Block Navigator tree view. Selecting root model source node will display basic information about model. This information is provided by Gen Studio. However, we would like to display some information about selected action block as well. This is how it is done.

Property View is generic type of the view. The Properties view displays properties for whatever is selected in the workbench. This view listens to all possible selections done anywhere in Gen Studio and checks if it can display any properties associated with the selected objects. It is able to display properties of only those objects which implement `IPropertySource` interface. We would like to display some basic information about action block represented by the `Acblkbsd` object. Unfortunately all object classes like: `Acblkbsd` and `MObj` do not implement `IPropertySource` interface by the definition. We need to use Eclipse Adapter technology to overcome this problem.

The Properties view goes through a few steps to sort out how it's going to display properties. First, it determines whether or not the selected object implements the `IPropertySource` interface. If it does, it uses the selected object directly after casting it to `IPropertySource`. If that check fails, the Property view then determines whether or not the selected object implements the `IAdaptable` interface. If the selected object is adaptable, it is asked—via the `getAdapter()` method—for an adapter with the `IPropertySource` type. The `getAdapter()` method either returns an object of the appropriate type or null if it cannot be adapted to the requested type. If the method returns an adapter, it is used by the Property view to gather properties (if it is null the Property view shows nothing).

In our case adapter framework actually go get the `AdapterManager` trying to find suitable object to represent our object and pass it to the Property view. This is object of type `AcblkbsdPropertySource`.

The following snapshot shows `AcblkbsdAdaptorFactory` implementing `IAdapterFactory`. This class `getAdapter()` method is invoked to substitute instance of `Acblkbsd` class by the instance of `AcblkbsdPropertySource` class.

```

public class AcblkbsdAdapterFactory implements IAdapterFactory {

    public AcblkbsdAdapterFactory() {
        Platform.getAdapterManager().registerAdapters(this, AcblkbsdPropertySource.class);
    }

    public void dispose() {
        Platform.getAdapterManager().unregisterAdapters(this, AcblkbsdPropertySource.class);
    }

    @SuppressWarnings("unchecked")
    private static Class[] SUPPORTED_TYPES = new Class[] { IPropertySource.class };

    @SuppressWarnings("unchecked")
    public Class[] getAdapterList() {
        return SUPPORTED_TYPES;
    }

    @SuppressWarnings("unchecked")
    @Override
    public Object getAdapter(Object object, Class adapterType) {
        if (adapterType.isInstance(object))
        {
            return object;
        }
        if (IPropertySource.class.equals(adapterType)) {
            return new AcblkbsdPropertySource(
                (Acblkbsd) object);
        }
        return null;
    }
}

```

We need to register adaptor factory with the `AdapterManager`. It can be done as a first thing after plug-in is activated. Method `registerAdaptor()` takes two parameters. First one is factory itself and second is type of object to be adapted. Please notice that adapter is unregistered when plug-in is stopped.

The following snapshot of code shows two methods responsible for registration and un-registration of our adaptor factory. They are `start()` and `stop()` methods of the `Activator` class.


```

public void start(BundleContext context) throws Exception {
    super.start(context);
    modelClosingListener = new ModelClosingListener();
    log = getLog();
    plugin = this;
    factory = new AcblkbsdAdapterFactory();
    IAdapterManager manger = Platform.getAdapterManager();
    manger.registerAdapters(factory, Acblkbsd.class);
}

public void stop(BundleContext context) throws Exception {
    plugin = null;
    super.stop(context);
    IAdapterManager manger = Platform.getAdapterManager();
    manger.unregisterAdapters(factory, Acblkbsd.class);
    factory = null;
    modelClosingListener.dispose();
}

```

Object substituting instance of Acblkbsd class is an instance of the AcblkbsdPropertySource class. It implements proper IPropertySource interface and can be processed successfully by the Property view.

Two methods `getPropertyDescriptors()` and `getPropertyValue()` are responsible to provide all required contents for the Property view.

The below sample code defines what property label should be used and how to obtain property value for the each action block.

```

@Override
public IPropertyDescriptor[] getPropertyDescriptors() {
    if (propertyDescriptors == null) {
        propertyDescriptors = new Vector<PropertyDescriptor>();
    }
    PropertyDescriptor propertyDescriptor = new PropertyDescriptor(
        new Integer(S_LENGTH), "Description Length");
    propertyDescriptor.setCategory(PROPERTIES);
    propertyDescriptors.add(propertyDescriptor);
    propertyDescriptor = new PropertyDescriptor(
        new Integer(S_MODDATE), "Modification Date");
    propertyDescriptor.setCategory(PROPERTIES);
    propertyDescriptors.add(propertyDescriptor);
    propertyDescriptor = new PropertyDescriptor(
        new Integer(S_MODTIME), "Modification Time");
    propertyDescriptor.setCategory(PROPERTIES);
    propertyDescriptors.add(propertyDescriptor);
    return (IPropertyDescriptor[]) propertyDescriptors
        .toArray(new IPropertyDescriptor[propertyDescriptors.size()]);
}

@Override
public Object getPropertyValue(Object object) {
    int key = ((Integer) object).intValue();
    switch (key) {
        case S_LENGTH:
            return new Integer(acblkbsd.getDesc().length());
        case S_MODDATE:
            return new Integer(acblkbsd.getModdate());
        case S_MODTIME:
            return new Integer(acblkbsd.getModtime());
        default:
            break;
    }
    return null;
}

```

Developing Text Editor for Action Block Description

At this point we are able to expand opened model sources showing all action blocks defined in the model. We are able also to show some properties of the action block and size of the text constituting its description. We need to create text editor now which we can use for editing text contents and properly integrate editor with Gen Studio.

Eclipse provides support for creating editors that operate on a text. The framework has been designed in several layers of increasing coupling to the Eclipse Platform. We are going to use parts that can be used only within a running Eclipse Platform application such as Gen Studio. We do not need to create very powerful editor. We need to develop something simple with little work.

Defining editor

We need add additional element to the plug-in manifest file extending `org.eclipse.ui.editors`. Class `DescriptionTextEditor` is our editor implementation class.

```
<extension
  point="org.eclipse.ui.editors">
  <editor
    name="Description Editor"
    icon="icons/action_block.gif"
    contributorClass="org.eclipse.ui.texteditor.BasicTextEditorActionContributor"
    class="gen.plugin.example.editors.DescriptionTextEditor"
    id="gen.plugin.example.editors.DescriptionTextEditor">
  </editor>
</extension>
```

Preparing input for the editor

The `DescriptionTextInput` class extends `PlatformObject` class and implements `IStorageEditorInput` interface. We need to have our own class since our implementation of `IStorage` input to an editor is object in the Gen model. All editor inputs must implement the `IAdaptable` interface so extensions can be managed by the platform's adapter manager. Class `PlatformObject` provides default implementation of `IAdaptable` interface.

The `DescriptionTextStorage` class extends `PlatformObject` and implements `IStorage` interface. We need our own non default implementation of the interface because action block description is stored in the Gen model object rather than in the text file. Storage objects have to implement the `IAdaptable` interface so extensions can be managed by the platform's adapter manager. Again class `PlatformObject` provides default implementation of `IAdaptable` interface.

The `DescriptionTextStorage` has two methods. Method `getContents()` is responsible for getting input stream to load description text into the text editor. Method `updateGenObject()` takes modified text from the editor of the action block description and saves as property of the action block.

The following snapshot of code shows how it is done.

```

@Override
public InputStream getContents() throws CoreException {
    return new ByteArrayInputStream(description.getBytes());
}

public void updateGenObject(String description) {
    Model model = mmObj.getModel();
    if (model.isReadOnly()) {
        return;
    }
    model.beginUnitOfWork();
    mmObj.setTextProperty(PrpTypeCode.DISC, description);
    mmObj.setIntProperty(PrpTypeCode.MODDATE, getCurrentDate());
    mmObj.setIntProperty(PrpTypeCode.MODTIME, getCurrentTime());
    model.commitUnitOfWork();
}

private int getCurrentDate() {
    Calendar now = Calendar.getInstance();
    int date = now.get(Calendar.YEAR) * 10000
        + (now.get(Calendar.MONTH) + 1) * 100
        + now.get(Calendar.DAY_OF_MONTH);
    return date;
}

private int getCurrentTime() {
    Calendar now = Calendar.getInstance();
    int time = now.get(Calendar.HOUR_OF_DAY) * 10000
        + now.get(Calendar.MINUTE) * 100 + now.get(Calendar.SECOND);
    return time;
}

```

Launching editor

As it was already mentioned user can right click on the Action Block Navigator and if some action blocks are selected than user is presented with possibility to launch text editor. Class `LaunchEditorActionDelegate` is responsible for the action.

```

<extension
    point="org.eclipse.ui.popupMenus">
    <objectContribution
        adaptable="false"
        id="com.ca.gen.jmmi.objs.popup"
        objectClass="com.ca.gen.jmmi.objs.Acblkbsd">
        <action
            class="gen.plugin.example.views.LaunchEditorActionDelegate"
            enablesFor="+"
            icon="icons/edit.gif"
            id="gen.plugin.example.menu"
            label="Edit Action Block Description"
            state="false"
            tooltip="Open action block description in the text editor."/>
        </objectContribution>
    </extension>

```

The `LaunchEditorActionDelegate` class implements `IObjectActionDelegate` interface. Implementing this interface for an object action selected from popup menu for the Action Block Navigator view. It is used to launch description text editor for the selected action blocks.

The following snapshot shows how implement `run()` method. We are checking first if text editor is opened for the selected action block. We do not want open another instance of the editor for the same action block.

```
@Override
public void run(IAction action) {
    if (selection != null) {
        List<?> list = ((IStructuredSelection) selection).toList();
        for (Object object : list) {
            if (object instanceof Acblkbsd) {
                openEditor((Acblkbsd) object);
            }
        }
    }
}

private void openEditor(Acblkbsd acblkbsd) {
    if (Utility.isEditorAlreadyOpened(acblkbsd)) {
        return;
    }
    IStorageEditorInput input = new DescriptionTextInput(new DescriptionTextStorage(acblkbsd));
    IWorkbenchPage page = site.getWorkbenchWindow().getActivePage();
    if (page != null) {
        try {
            page.openEditor(input,
                "gen.plugin.example.editors.DescriptionTextEditor");
        } catch (PartInitException e) {
            Activator.log.log(new MultiStatus(Activator.PLUGIN_ID,
                IStatus.ERROR,
                "A workbench part cannot be initialized correctly", e));
        }
    }
}
```

Saving and Closing Editor

`DescriptionTextEditor` overrides two methods of the `TextEditor`. They are `doSave()` and `doSaveAs()`.

The following snapshot of the code shows how it done. Please notice that method `doSaveAs()` does not have any statements. We want modified description to be saved only in the action block object from it comes from.

```

@Override
public void doSave(IProgressMonitor progressMonitor) {
    try {
        DescriptionTextStorage storage = (DescriptionTextStorage) ((DescriptionTextInput) getEditorInput())
            .getStorage();
        storage
            .updateGenObject(getSourceViewer().getTextWidget()
                .getText());
        super.doSave(progressMonitor);
        setPartName(storage.getMmObj().getTextProperty(PrpTypeCode.NAME));
    } catch (CoreException e) {
        Activator.log
            .log(new MultiStatus(
                Activator.PLUGIN_ID,
                IStatus.ERROR,
                "A status object describing the cause of the exception",
                e));
    }
}

@Override
public void doSaveAs() {
}

@Override
public void propertyChanged(Object source, int propId) {
    if (propId == IEditorPart.PROP_DIRTY && firstTime) {
        firstTime = false;
        return;
    } else if (propId == IEditorPart.PROP_DIRTY) {
        setPartName("*" + getPartName());
    }
}

```

Defining Perspective

Last action is to define on a dedicated perspective using elements defined earlier. The definition of a new perspective is a two steps process. Firstly, we need to add a perspective extension to the plug-in manifest file. Secondly, we define a perspective class for the extension within the plug-in.

Gen Studio is architected to have a perspective designed for each aspect of the product. To contribute our perspective to Gen Studio we must add

`com.ca.gen.studio.common.workspace.perspectives.genperspective` extension for the previously added perspective. The following snapshot shows two extensions added to the `plugin.xml` file.

```

<extension
    point="org.eclipse.ui.perspectives">
    <perspective
        name="Maintaining Descriptions"
        icon="icons/perspective.gif"
        class="gen.plugin.example.perspectives.DescriptionHandlingPerspective"
        id="gen.plugin.example.perspectives.DescriptionHandlingPerspective">
    </perspective>
</extension>
<extension
    point="com.ca.gen.studio.common.workspace.perspectives">
    <genperspective
        id="gen.plugin.example.perspectives.DescriptionHandlingPerspective"
        mnemonic="M"
        priority="999">
    </genperspective>
</extension>

```

We need to develop `DescriptionHandlingPerspective` class which implements `IPerspectiveFactory` interface. Eclipse executes method `createInitialLayout()` when perspective is created first time. The below implementation code adds two views to the perspective. *Action Block View* is added on the left of editor area and *Property View* is designated on right side of the editor area. Editor area occupies central part of the window and is used to open text editor allowing edit action block descriptions.


```

public class DescriptionHandlingPerspective implements IPerspectiveFactory {

    private IPageLayout factory;

    public DescriptionHandlingPerspective() {
        super();
    }

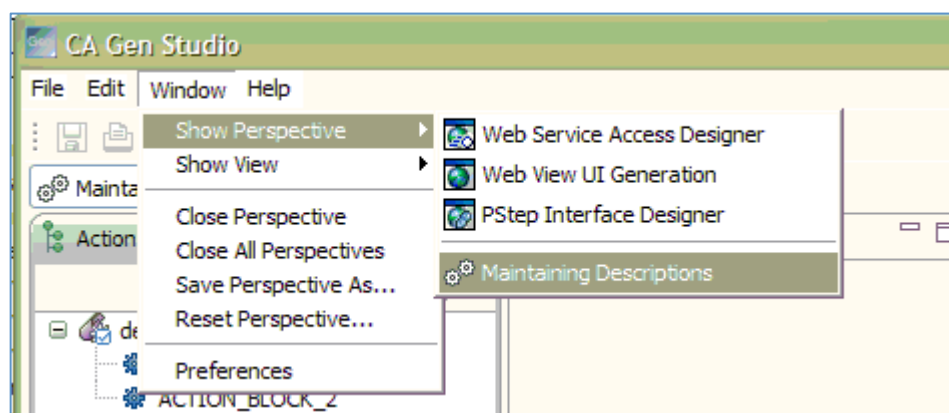
    public void createInitialLayout(IPageLayout factory) {
        this.factory = factory;
        addViews();
        addViewShortcuts();
    }

    private void addViews() {
        factory.setEditorAreaVisible(true);
        IFolderLayout topLeft = factory.createFolder("topLeft",
            IPageLayout.LEFT, 0.20f, factory.getEditorArea());
        topLeft.addView(ActionBlockNavigatorView.ID);
        IFolderLayout topRight = factory.createFolder("topRight",
            IPageLayout.RIGHT, 0.25f, factory.getEditorArea());
        topRight.addView(IPageLayout.ID_PROP_SHEET);
    }

    private void addViewShortcuts() {
        factory.addShowViewShortcut(ActionBlockNavigatorView.ID);
        factory.addShowViewShortcut(IPageLayout.ID_PROP_SHEET);
    }
}

```

Regular perspectives are hidden within Gen Studio, but those with the additional `genperspective` extension are displayed in the *Window->Show Perspective* menu and on the *Welcome View* page. Our perspective has name *Maintaining Descriptions* is added to the bottom of the submenu.



Testing and deploying a new plug-in

The Eclipse SDK provides a powerful environment for testing plug-ins. It is beyond the scope of this document to explain in detail how to test and debug plug-in application. There are many published articles and books explaining in great detail how to do it. You find also in those publications how to package and deploy ready plug-in.

Summary

These Gen Studio classes allow the CA Gen customers and Open Initiative partners to write software to retrieve and update encyclopedia information and integrate its functionality with Gen Studio. Sample plug-in used in this knowledge document do not show full potential of this new interface, but it should convince you that writing such plug-ins should not be too difficult.

This knowledge document does not show every detail of how plug-in functioning. Reader really needs to look into the complete source code to find out how certain things work. The Eclipse SDK allows run plug-in in the Gen Studio context in the debug mode. It would be very helpful to follow logic to better understand interdependences between classes constituting sample plug-in.