Creating a pure::variants Model from a CSV File

Tutorial

Table of Contents

1. Overview

This tutorial shows the use of the **pure::variants Synchronization Framework** for creating and synchronizing pure::variants models from external data sources. The tutorial example is the import and update of feature models from CSV[1] files.

The synchronization framework is used by several pure::variants extensions like the Synchronizer for Doors and the Synchronizer for CaliberRM as well as the Connector for Source Code Management.

The presented implementation is an Eclipse plugin consisting of two parts, the import function and an update function. The importer consists of a wizard that is registered as a **pure::variants Importer** and appears on the menu point *Import->Variant Models or Projects->Simple CSV Import (Example)*. This wizard shows how a CSV file can be mapped to a pure::variants model. The feature model produced by the import can be compared with the original CSV file with the help of the update function. Changes in the CSV file can be visualized and merged into the imported model.

The tutorial is structured as follows. Chapter 2 describes how a new Eclipse plugin is created. Chapter 3 provides a short introduction to the synchronization framework and explains how to map the information from a CSV file to a pure::variants model. Chapter 4 shows how to create the pure::variants model from the mapped CSV information and it shows the steps needed to provide the import wizard. Chapter 5 explains the implementation and registration of a compare provider implementing the update function. Finally in chapter 6 it is shown how to use the new import wizard to import a CSV file. And it is shown how to use the model synchronization functionality of pure::variants to compare and update the imported model with the CSV file.

The reader must have basic knowledge of **pure::variants** and the **Java Plugin Develop-**

[1] CSV - Character Separated Values

ment under **Eclipse**. For more information about the Eclipse Plugin concept see chapter **Plattform Plug-in Developer Guide** in the **Eclipse Help**.

The plugin described in this tutorial is part of the pure::variants SDK. It can be installed by choosing *New->Example* from the Eclipse *File* menu, and then *Examples->Variant Management SDK->Extensibility Example Plugins->com.ps.pvesdk.examples.import.csv.plugin*.

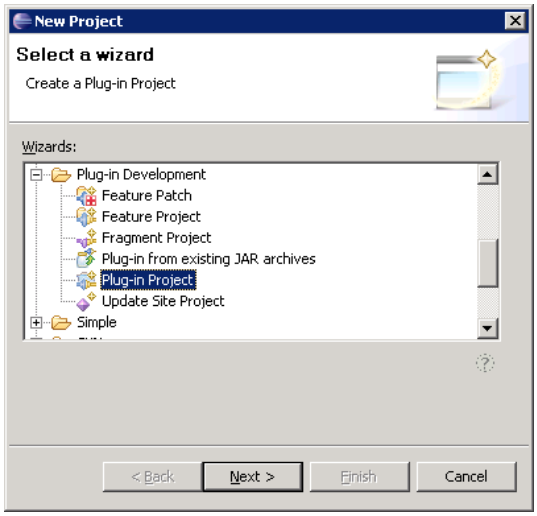
This tutorial is available as online help or in a printable PDF format here.

2. Setting up the Plugin Project

The first step to set up a new integration of an external data source, a CSV file in this case, is to create a new Eclipse plugin. This plugin contains the Java implementation of the importer and updater as well as the registration entries for the import wizard and the compare provider.

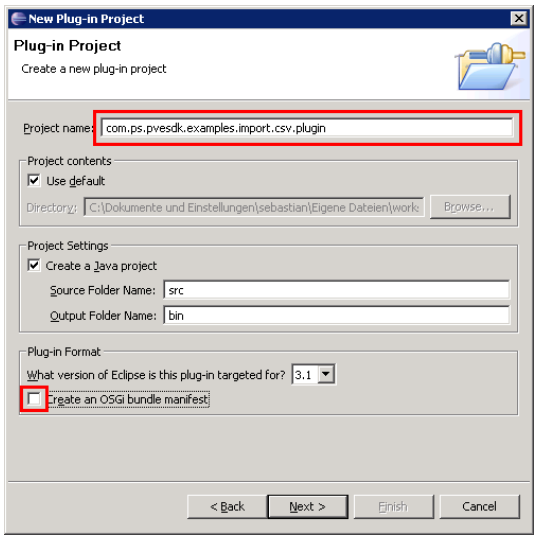
Choose item *File->New->Project* from the Eclipse menu and select "Plugin Project" in the list of available project wizards, see Figure 1, “Plug-in Project”.

Figure 1. Plug-in Project



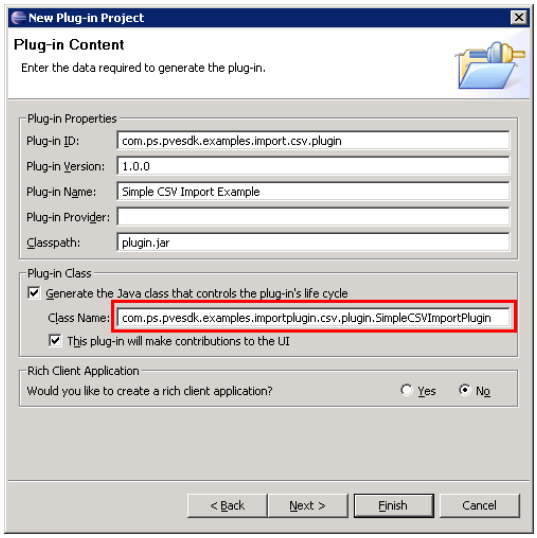
The name of the new project shall be "com.ps.pvesdk.examples.import.csv.plugin". All other settings should be made according to Figure 2, “Create new Plug-in Project”. Note that **Create an OSGI bundle manifest** remains unselected for this example plugin.

Figure 2. Create new Plug-in Project



Click on button Next to switch to the "Plug-in Content" page. Apply the settings as shown in Figure 3, “Plug-in Content”.

Figure 3. Plug-in Content



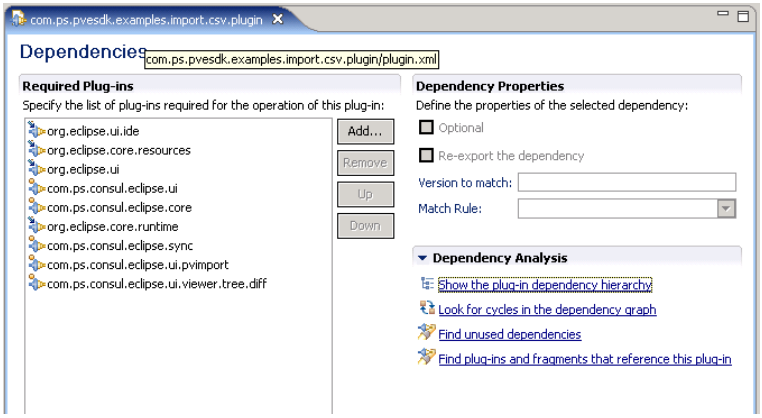
The new plugin project is created after clicking on Finish. It contains a source directory for the Java implementation of the importer, the plugin description file `plugin.xml`, and the file `SimpleCSVImportPlugin.java` defining the plug-in's life cycle class.

To be able to use the synchronization framework some additional plugin dependencies have to be specified. Open file `plugin.xml` with the **Plug-in Manifest Editor** by double-clicking on it. On the "Overview" page you can see the fundamental information about the project. Switch to the "Dependencies" page and click on the Add button to add the plugins listed below (see Figure 4, “Plug-in Dependencies”).

- org.eclipse.ui.ide

- org.eclipse.ui.views
- org.eclipse.jface.text
- org.eclipse.ui.workbench.texteditor
- org.eclipse.ui.editors
- org.eclipse.core.resources
- org.eclipse.ui
- com.ps.consul.eclipse.ui
- com.ps.consul.eclipse.core
- org.eclipse.core.runtime

Figure 4. Plug-in Dependencies



The plugin is now ready for the next step, i.e. using the synchronization framework to import a CSV file.

3. The Synchronization Framework

After preparing the plugin it is now shown how the CSV file entries are mapped to the elements of a pure::variants model using the synchronization framework.

The synchronization framework is implemented in the plugin com.ps.consul.eclipse.sync. It provides the functionality to import external data sources as pure::variants models and to do updates of the imported models from the external data sources. Every part of a pure::variants model is provided by an associated interface implementing the mapping from the external data source. These interfaces are marked by the prefix "External". The following table lists the mapping of the interfaces.

Table 1. Mapping model element to external element

`IConsulModel`	`IExternalModel`
`IElement`	`IExternalElement`
`IProperty`	`IExternalProperty`
`IPropertyConstant`	`IExternalContant`
...	

All interfaces used for mapping external data to pure::variants model parts are defined in the package modeling of the synchronization framework.

For creating a pure::variants model from an external data source the interface `IExternalModel` has to be implemented. `IExternalModel` provides all information needed to create a pure::variants model. The following methods have to be implemented:

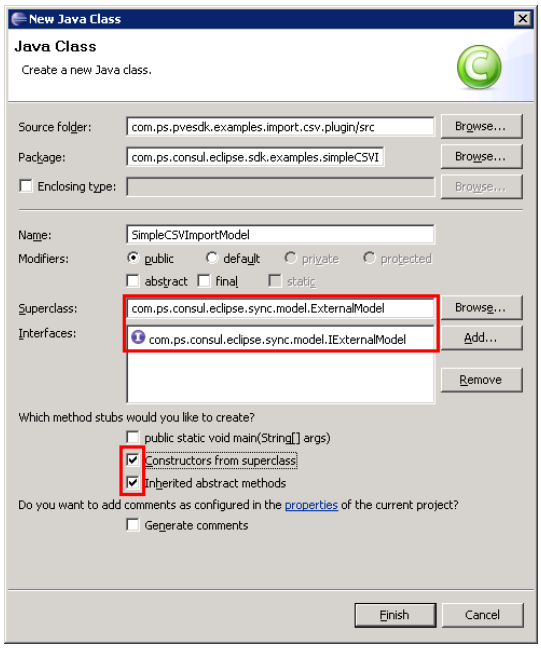
Table 2. Methods of IExternalModel

Method Name:Return Type	Description
`getID():ID`	Returns the ID of the model. The ID must be unique for all models.
`getName():String`	Returns the name of the model. The name must be an valid OCL identifier.
`getAuthor():String`	Returns the author of the model.
`getVersion():String`	Returns the version of the model.
`getDesc():String`	Returns the textual description of the model.
`getType():String`	Returns the model type. Possible types are "ps:fm" for the feature models and "ps:ccfm" for the family models.
`getLocation():IResource`	Returns the location of the imported model in the file system. This method returns `null` if there is no file system representation of the model.
`getFileName():String`	Returns the name of the file containing the imported model.
`getSize():int`	Returns the number of elements to import into the model. The value is used by the import progress monitor to show the progress of the import. If the number of elements is unknown, then -1 is returned.
`getRootElement():IExternalElement`	Returns the root element of the model. The root element must not be `null`.
`getRelations(): IExternalRelation[]`	Returns the relations between the elements of the model.

3.1. Implementing IExternalModel

For implementing `IExternalModel` a new Java class has to be created in the plugin project. Name the class **`SimpleCSVImportModel`** and add **`com.ps.consul.eclipse.sync.model.ExternalModel`** as the super-class (see Figure 5, “New Java Class Wizard”).

Figure 5. New Java Class Wizard



Now the methods of class **`SimpleCSVImportModel`** have to be implemented. Some of the methods defined in `IExternalModel` already are implemented by class `ExternalModel`. Thus, only the methods `getDesc()`, `getLocation()`, `getRootElement()`, `getRelations()` and `getSize()` need to be implemented.

A special implementation is required for method `getLocation()`. It returns the position of the model in the pure::variants project hierarchy, represented by an `IResource` handle. It is important that the handle does not point to an existing file when importing. Overwriting existing files is not permitted by the synchronization framework to prevent destroying existing models.The following example shows how an `IResource` handle can be created:

```
// Get the workspace root
IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();

// Create a project handle
IProject testProject = root.getProject("test");

if (testProject.exists() == true){
   // Create a resource handle for a pure::variants model
   IResource resourceHandle = testProject.getFile("modelFileName");
}
```

3.2. Creating the model from the CSV file

Now the information about the model elements have to be imported from the CSV file. The CSV file must fulfill some fundamental assumptions for this example. Each model element has a valid variation type (for example *ps:optional*), a unique ID, and a unique name (unique in the model). For each element the ID of the parent element is needed to build the hierarchy of the model. Thus, the CSV file needs the following 4 columns: *Unique ID*, *Unique Name*, *Type*, *Parent Unique ID*. All other columns are interpreted as element attributes in this example. Following steps are necessary to construct a model from a CSV file:

1. Open the CSV file and read the first line containing the column headers.
2. Read the other lines of the file containing the element definitions.
3. Create the elements using the information from the lines of the CSV file.
4. Create the model structure, i.e. the element hierarchy, using the parent element information.

Example 1. Simplified code from method `initFileContent()` in file `SimpleCSVImportModel.java`

```
// Open the CSV file
BufferedReader reader = new BufferedReader(new FileReader(file));
// Read the first line
String tableHeader = reader.readLine();
// Parse the first line and identify the columns
String[] columns = m_Parse.parse(tableHeader);
// Read all other lines
while(reader.ready()){
   // Read the next line
   String line = reader.readLine();
}
```

Creating an ExternalElement

With knowledge of the columns in the CSV file, an `ExternalElement` can be created for each line. First the columns of a line have to be identified. Then, if the values for *Unique ID*, *Unique Name* and *Type* are known, a new empty external element can be created and the values can be set.

Example 2. Simplified code from method `createElement()` in file `SimpleCSVImportModel.java`

```
// Create an empty external element
ExternalElement newElement = new ExternalElement();
// Parse the current line
String[] values = m_Parse.parse(line);
// Find the element properties
String ID         = getID(values)
String type       = getType(values)
String uniqueName = getUniqueName(values)

// Set the unique element ID
newElement.setID(ID)
// Set the variation type
newElement.setRelType(type)
// Set the unique name of the element
newElement.setName(uniqueName)
```

Additionally the value for *Parent Unique ID* must be stored for later use. Later the unique IDs of the parent and the current element are used to create the element hierarchy. If no parent ID is given for an element, then this element is taken as the root element of the model. There must be exactly one root element in a pure::variants feature or family model.

Creating the attributes for an ExternalElement

All columns other than the columns described above are interpreted as element attributes. The element attributes are described by objects of type `ExternalProperty`. The name of the attribute is gathered from the column header. The values are defined in the lines. Only constant attributes are supported by this example.

```
// Get the column name
String columnName = ...;

// Create an external property
ExternalProperty prop = new ExternalProperty();

// Set the property name
prop.setName(columnName);

// Set the property type
prop.setType(ModelConstants.ATTRIBUTES_STRING_TYPE);

// Create a string constant with the value given in the current line
ExternalConstant constant = new
ExternalConstant("lineValue",true,ModelConstants.ATTRIBUTES_STRING_TYPE);

// Add the constant value to the property
prop.addConstant(constant);

// At last the property must be added to the element
```

Creating the model structure

After all elements are created, the hierarchy of the elements has to be created. The starting point for this process is the root element of the model, i.e. the element that has no parent ID. The default implementation of `IExternalElement` has a method

```
addChildren(child:IExternalElement)
```

With this method the child elements of an element are specified. Beginning with the root element all elements are added to the element hierarchy step-by-step.

4. Creating a pure::variants model from IExternalModel

After implementing class `SimpleCSVImportModel` containing all the information about the model to import, the next step is to build the pure::variants model from the collected information. For this purpose the class `ModelGenerator` is used. It takes the collected model information and builds the pure::variants model automatically.

```
// Create a new model generator using the
IExternalModel model = new SimpleCSVImportModel(..);
ModelGenerator gen = new ModelGenerator(new IExternalModel[]{model});
```

The build process of the pure::variants model is started by calling `createModels()` on the model generator.

```
// Create the model and import it into the Eclipse project hierarchy
gen.createModels(new NullProgressMonitor());
```

First a pure::variants model is created from the `IExternalModel` object. This model is then send to the pure::variants server where it is saved. If the model shouldn't be save in the file system, for example to create a reference model for the model comparison, this can be switched off by calling

```
gen.setImportModels(false);
```

The created model can be queried by the following call

```
gen.getModel(modelID:ID)
```

Finally the produced model needs a Nature ID that is used during the model compare to identify the kind of model.

```
IConsulModel newModel =
ConsulCorePlugin.getDefault().getModelManager().openModel(
  model.getLocation().getLocation().toFile().toURL());
// Add the model nature
NatureModeler.addNature(newModel,
  "com.ps.consul.eclipse.sdk.examples.wizards.CSVImportNature");
// Save and close the model
ConsulCorePlugin.getDefault().getModelManager().saveModel(newModel);
ConsulCorePlugin.getDefault().getModelManager().closeModel(newModel);
```

4.1. Adding the Wizard

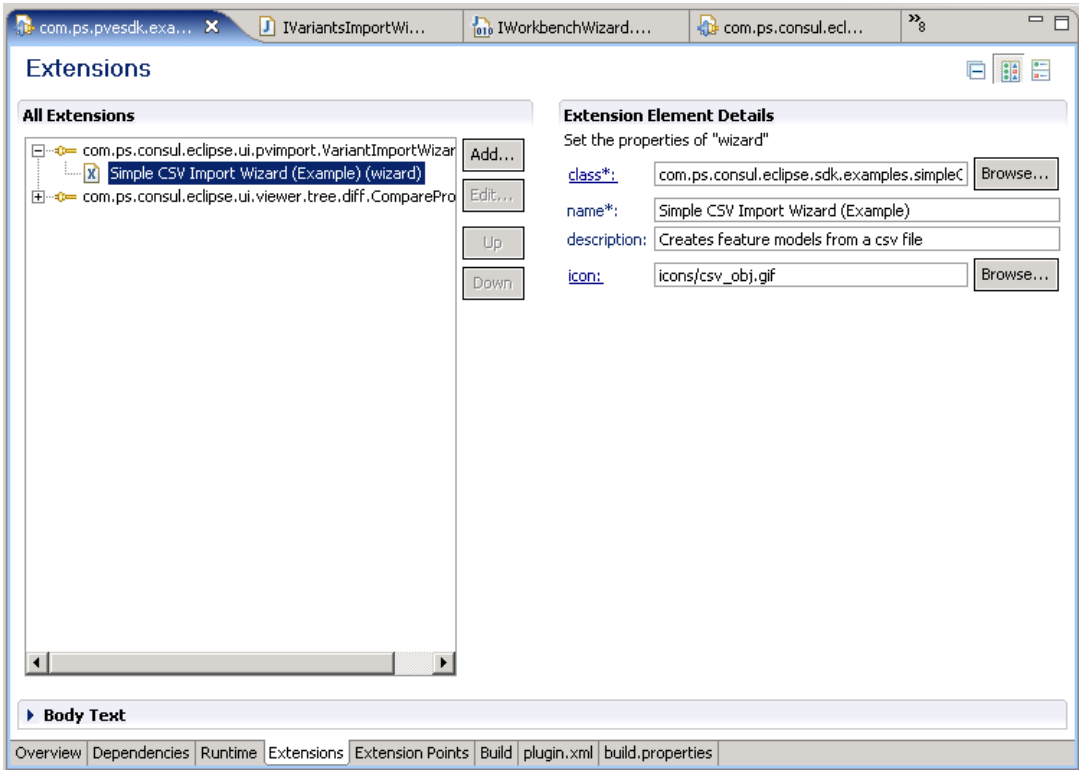
The last step for realizing the import is providing the import wizard. The import wizard is used to select the CSV source file, the target model name and location, and to start the import process. Therfor a new wizard has to be added to the list of pure::variants import wizards. How to write an import extension is described in detail in the section *Writing an import extension* in the *pure::variants Extensibility Guide -> Tasks -> Client Extension.*

Following steps have to be performed:

1. Add the extension point *com.ps.consul.eclipse.ui.pvImport.VariantImportWizards* to extensions list of the plugin
2. Create a new class named `SimpleCSVImportWizard` and register it as a "wizard" extension at this extension point
3. Implement a wizard page named `TargetSelectionPage` for the `SimpleCSVImportWizard`
4. Implement the `performFinish()` method of the `SimpleCSVImportWizard`

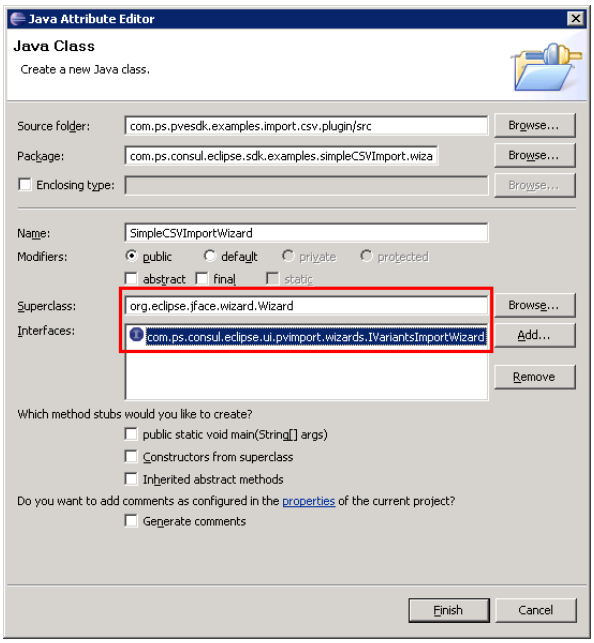
To register a wizard the file `plugin.xml` has to be opened in the **Plug-in Manifest Editor**. The "Extensions" page shows all extensions the plugin provides to the eclipse architecture.

Figure 6. Adding the Import Wizard extension



The right side of Figure 6, “Adding the Import Wizard extension” shows the attributes describing the new wizard. Provide as name "Simple CSV Import Wizard (Example)" and as description "Creates feature models from a csv file". The icon field is optional and does not need to be filled in. Once Eclipse is notified about the wizard the wizard class has to be implemented. Click on the "class" field name and the "New Class" wizard appears.

Figure 7. Creating an Import Wizard



The wizard class should be named `SimpleCSVImportWizard`. The class has to extend class `org.eclipse.jface.wizard.Wizard` and implement the interface `com.ps.consul.eclipse.ui.pvimport.IVariantImportWizard`. Click on Finish to create the new class.

Once the wizard class is created, a wizard page has to be implemented that is displayed in the wizard. The page should allow to select a target file for the imported model in the file system. This is the location returned by the method `getLocation()` of class `SimpleCSVImportModel`. Also the page should provide a file selection dialog that allows the user to select the CSV source file. The wizard itself is finished by clicking on the Finish button. If the button is clicked the wizard's `performFinish()` method is called from by Eclipse environment.

The `performFinish()` method collects the values entered in the target selection page. Then a new `SimpleCSVImportModel` is created and initialized with the collected values. Finally the import process is started.

Example 3. Simplified code of method `performFinish()` in file `SimpleCSVImportWizard.java`

```
// The wizard page that provides the information about target location,
// model name, file name, and the CSV file
private TargetSelectionPage m_TargetPage = new TargetSelectionPage(..);

public boolean performFinish() {
  // The CSV file to import
  File sourceFile = m_TargetPage.getSourceFile();
  // The target container, i.e. the target folder for the created model
  IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
  IPath container = root.getLocation().append(m_TargetPage.getSelection());
  IContainer res = root.getContainerForLocation(container);
  // The file and name of the model to create
  String fileName = m_TargetPage.getFileName();
  String modelName = m_TargetPage.getModelName();
  String author = System.getProperty("user.name");

  try {
    // Create a unique id for the model.
    ID modelID = new ID();
    // Construct this specific implementation of IExternalModel which
    // provides the model information for the model generator that
    // finally does the work and parses the CSV file.
    SimpleCSVImportModel model = new SimpleCSVImportModel(modelID,
      modelName, fileName, author, "1.0", ModelConstants.FM_TYPE,
      sourceFile, res, new CSVParser());
    // Construct the model generator and feed it with the external model.
    // The generator in general allows to create several models at once.
    ModelGenerator gen = new ModelGenerator(new IExternalModel[]{model});
    // Create the models. If desired a progress monitor can be set showing
    // the user the progress of the model generation.
    gen.createModels(new NullProgressMonitor());
  } catch(Exception e) {
    // signal the wizard not to close
    return false;
  } finally {
    // Refresh the created models container independent on any exception.
    refreshProject(res);
  }
  // signal the wizard to close
  return true;
}
```

5. Updating the imported model

After implementing the import functionality it is now explained how to provide the update functionality. The model imported from the CSV file can be compared with the original CSV file. The changes between the model and the CSV file are shown in the Compare Editor and can also be merged. For comparing two models it is substantial that both models have the same model ID. All model elements are compared on the basis of its IDs. Two elements are identified as pair and can only be compared if both have the same ID. Otherwise an "Element Removed" and an "Element Added" action indicates the change. The same applies to element properties, relations, constants, etc.

Following steps have to be performed to add the compare functionality:

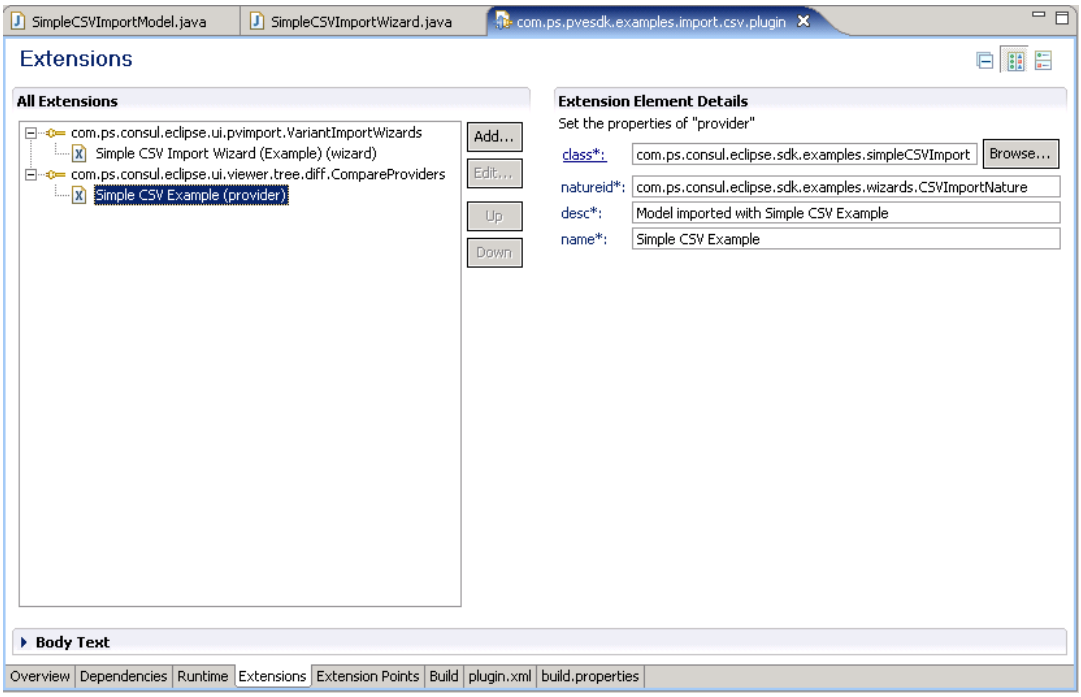
- Add the extension point *com.ps.consul.eclipse.ui.viewer.tree.diff.CompareProvider* to the plugin's extension list
- Create a new class implementing the interface `IConsulModelCompareProvider`

- Register the new class as a "provider" extension at the Compare Provider extension point

5.1. Register a Compare Provider

In pure::variants a Compare Provider has to be registered to compare a model. For this purpose open the file `plugin.xml` again. Then change to page "Extensions".

Figure 8. Extensions - Compare Provider



Click on button *Add* to add a new extension for the extension point *com.ps.consul.eclipse.ui.viewer.tree.diff.CompareProviders*. Select the extension point and choose New->provider from the context menu. On the right side of the "Extensions" page the required settings must be made. The attribute "class" specifies the path to the class implementing the `IConsulModelCompareProvider`. For the attribute "natureid" insert the Nature ID as the one used for creating the imported models, i.e. "com.ps.consul.eclipse.sdk.examples.wizards.CSVImportNature". The description and name are freely selectable.

5.2. Implementing the CompareProvider

Each comparison between two models must implement the `IConsulModelCompareProvider` interface. For this add a new class to the plugin. This class implements the interface and has to extend class `com.ps.consul.eclipse.ui.viewer.tree.diff.editor.ConsulModelCompareProvider`. The models to compare should be opened with the method `initialize(input:IEditorInput)`. The left model, i.e. the imported model, can be got with the following code

```
private File getFileForInput(IEditorInput input) {
  File result = null;
  if (input instanceof FileEditorInput) {
```

```
    result = ((FileEditorInput) input).getFile().getLocation().toFile();
  }
  return result;
}
```

Then the file can be opened with the pure::variants model manager:

```
ConsulCorePlugin.getDefault().openModel(url:URL):IConsulModel
```

The model manager opens the model file and creates a corresponding `IConsulModel` object. The right model must be temporarly produced from the CSV file. For this purpose, the same steps can be performed as for importing a model from the CSV file. The only difference is, that it is not necessary to write the model into the file system because it is only used for the comparison.

```
// Create a model generator
SimpleCSVImportModel model = new SimpleCSVImportModel(...);
ModelGenerator gen = new ModelGenerator(new IExternalModel[]{model});

// Turn off saving the model into the file system
gen.setImportModels(false);

// Create the pure::variants model and import it into the Eclipse project
hierarchy
gen.createModels(new NullProgressMonitor());

// Get the imported model
IConsulModel pvmodel = gen.getModel(model.getID());
```

The next two method calls pass the models to the compare editor:

```
setLeftModel(left:IConsulModel)

setRightModel(right:IConsulModel)
```

Finally the text identifying the models in the compare editor has to be defined. The initialize method must return true to let the compare editor evaluate the models and show the changes.

Note that all models opened with the pure::variants model manager must also be closed with this model manager:

```
ConsulCorePlugin.getDefault().getModelManager().closeModel(model:IConsulModel)
```

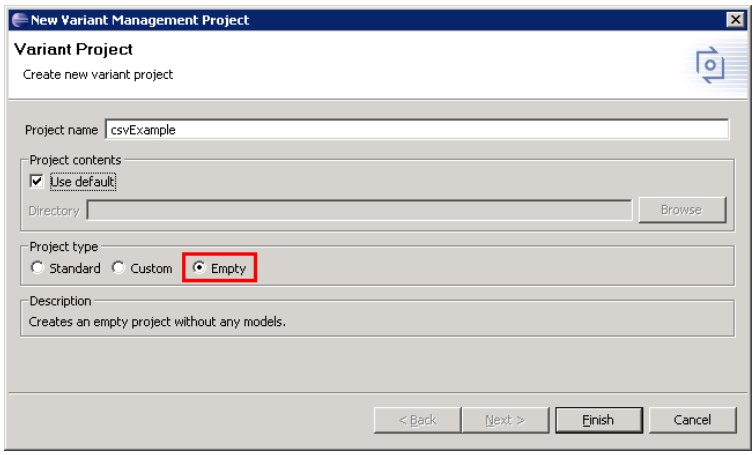
6. Using the CSV Example Plugin

Now it is time to test the importer and update functionality. For testing the plugin two different possibilities exist. Either the plugin is exported as Deployable Plugin and installed into pure::variants. Or an Eclipse Runtime is started using the CSV Example plugin. This approach is described in the Eclipse help in chapter *PDE Guide -> Getting Started -> Basic Plug-in Tutorial -> Running a plug-in*. How to export and install the plugin as a Deployable Plugin is described in the *PDE Guide -> Getting Started -> Basic Plug-in Tutorial -> Exporting a Plugin* and in *pure::variants Extensibility Guide -> Concepts -> Plugin Creation and Deployment -> Tutorial:Simple Plugin*

6.1. Using the CSV Import

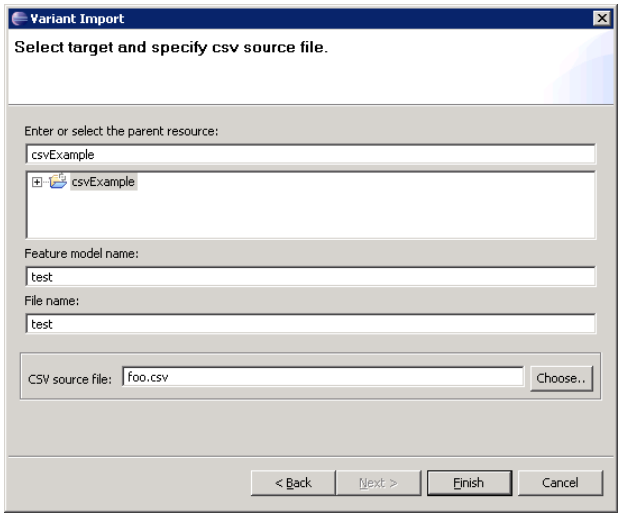
To demonstrate the use of the CSV importer, a new pure::variants project should be created. In the context menu of the Variant Projects view choose *New->Variant Project*.

Figure 9. New pure::variants Project



The models created from the CSV files can be imported into the new project. Select the new project in the Variant Project View. Then choose *Import->Variant Models or Projects->Simple CSV Import (Example)* from the context menu. This opens the CSV Import wizard.

Figure 10. CSV Import Wizard



In the upper part of Figure 10, “CSV Import Wizard” the target project or directory for the imported model has to be selected. In the lower part the model and the file names have tp be specified. At last the CSV input file has to be selected using the button *Choose*. After clicking on *Finish* the `performFinish()` method of class `SimpleCSVImportWizard` is called and the import starts. This method executes the algorithm described in section 4. It uses the `SimpleCSVImportModel` and the `ModelGenerator` to create a pure::variants model from the CSV file and saves it into the target project or directory.

6.2. Synchronizing an imported model

The pure::variants synchronization functionality is invoked by opening the imported model and clicking on the Synchronize Model button in the Eclipse tool bar. Then choose the

original CSV file. The compare editor opens showing the changes, if there are any.

Figure 11. Compare Editor - Comparing a Model with a CSV file