Developing a pure::variants Model Validation Check

Tutorial

Table of Contents

1. Overview

The reader must have basic knowledge of **pure::variants** and the **Java Plugin Development** under **Eclipse**. For more information about the Eclipse Plugin concept see chapter **Plattform Plug-in Developer Guide** in the **Eclipse Help**.

This tutorial explains how to develop a new check and corresponding quick fix for the pure::variants **Model Validation Framework**. Model Validation checks are applied in order to examine the correctness of a pure::variants model. If a check detects problems in a model, the provided quick fix can be used to solve this problem automatically.

A check is a Java class that is registered as Model Validation Framework extension in the Eclipse plugin containing the check. The quick fix also is a Java class that does not need to be registered. In the following it is shown how to setup a new Eclipse plugin, implement and register the check, and provide a quick fix for the check. The presented example check examines all unique names of the elements of a feature model. The names must begin with the string *feature*, otherwise a problem is announced.

The tutorial is structured as follows. Chapter 2 describes how a new Eclipse plugin is created. Chapter 3 shows the implementation and registration of the check class. Chapter 4 shows how the new check is activated and applied to a model. Chapter 5 explains how the quick fix for the check is implemented and connected to the check. Chapter 6 shows how the quick fix for the check is used. The last chapter provides information about how to install the new plugin in an Eclipse installation.

Before reading this tutorial it is recommended to read section 6.4.2 (Model Check Framework) from the **pure::variants User's Guide**.

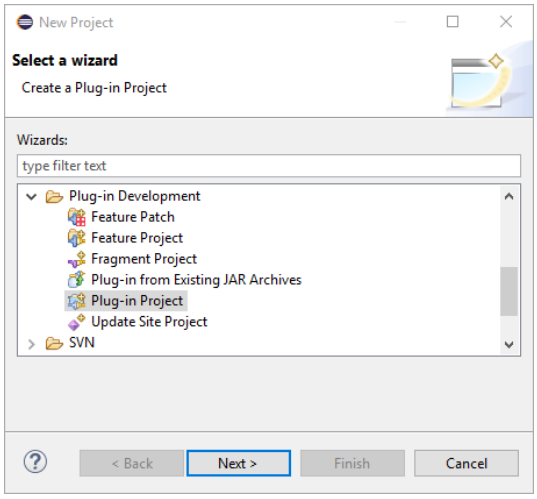
The plugin described in this tutorial is part of the pure::variants SDK. It can be installed by choosing *New -> Example* from the Eclipse *File* menu, and then *Examples -> Variant Management SDK -> Extensibility Example Plugins -> com.ps.pvesdk.examples.modelvalidation.plugin*.

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

2. Setting up the Plugin Project

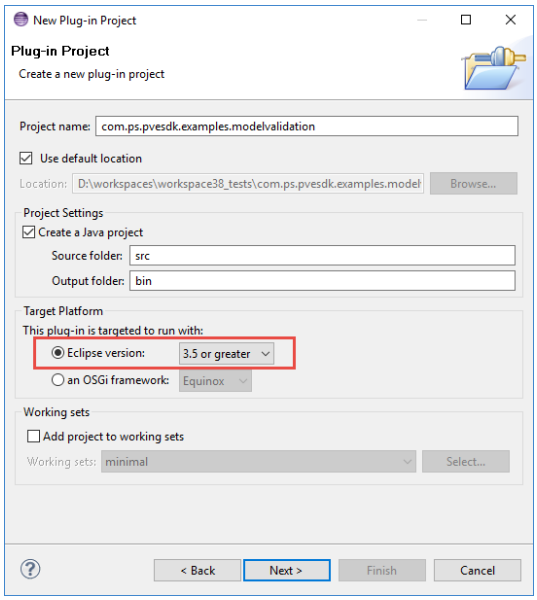
As first a new Eclipse plugin project has to be created. Right-click in the Eclipse Projects View and choose *New -> Project -> Plug-in Project* from the context menu (see Figure 1, “New Plug-in Project Wizard”).

Figure 1. New Plug-in Project Wizard



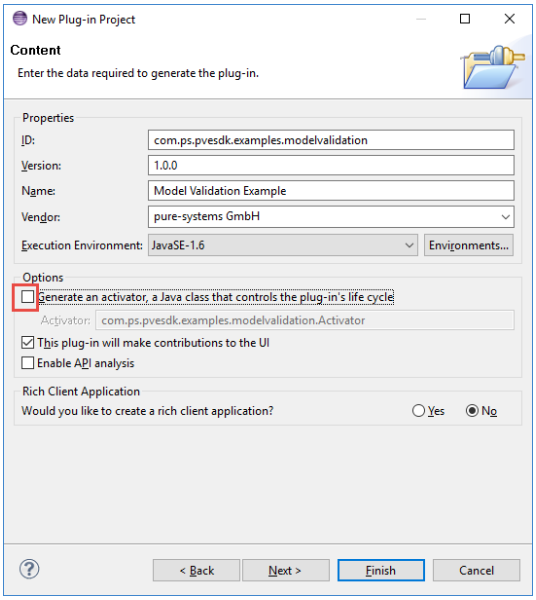
The name of the new project shall be **com.ps.pvesdk.examples.modelvalidation**. Figure 2, “Plug-in Project Settings” shows further settings required for the plugin. Please note that the plugin has to work with an eclipse version and must not work with OSGI framework.

Figure 2. Plug-in Project Settings



After pressing *Next* the **Plug-in Content** page is opened. Please apply the settings as shown in Figure 3, “Plug-in Content Settings”.

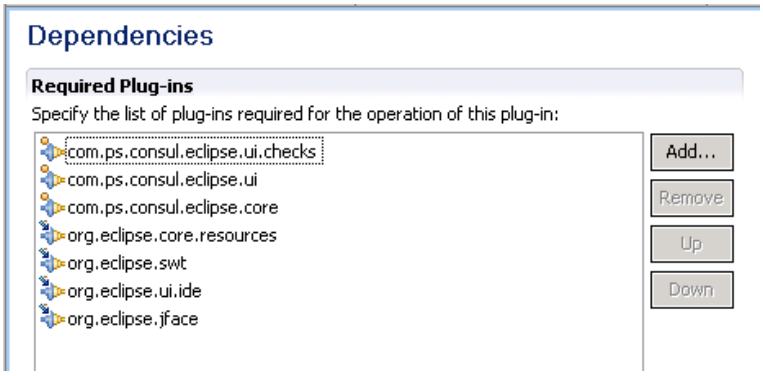
Figure 3. Plug-in Content Settings



After pressing the *Finish* button a new plugin project is created. The new project contains an empty **src**-directory for the Java code and a **plugin.xml** file.

Next the dependencies should be added for the plug-in. For this double-click on the *plugin.xml* file to open it in the **Plug-in Manifest Editor**. Switch to the **Dependencies** page and press the *Add* button to add the plugin-ins listed in Figure 4, “Dependencies”.

Figure 4. Dependencies



This is all to setup the project. The next step is to write the check class.

3. Writing the Check Implementation

This chapter shows how to implement the Java class for the new Model Validation check.

First the new Java package **com.ps.pvesdk.examples.modelvalidation.plugin** has to be created in the *src*-directory of the plug-in. Then create a new Java class within the package and name it **CheckElementUniqueNameExample.java**. This class has to be derived from class **Check** and has to implement the **IElementCheck** interface.

Each check class implements the **check()** methods from the interfaces it implements (**IElementCheck** in this case). These methods are called by the Model Validation Framework for each model item to check (model elements in this case), and implement the check functionality. The return value of a check() method is an object of type **ICheckResult**. This object contains the problems found by this check (of type **Problem**).

In the presented example a problem is provided for all features with a unique name that does not start with the string *feature*. The problem object contains:

1) The problem-**class**. The problem-class has to match the name of the check as given at the extension point of the plugin.

2) The problem-**type**, here *ELEMENTCHECK_TYPE*. The type corresponds to the interfaces implemented by the check.

3) The problem-**code** to identify the problem. This code has no special format but shall be unique.

4) The problem-**severity**, here *ERROR_SEVERITY*. A problem can also have the severities warning and info.

Furthermore the problem object contains information about the model item that was checked, i.e. the element id in this case, and a textual problem description.

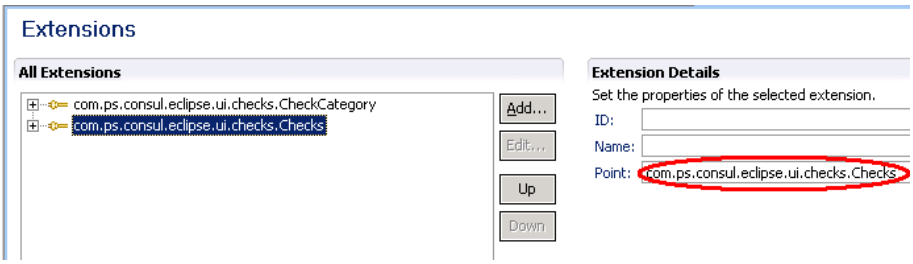
This is the implementation of the check() method of the example check.

```
/**
   * This method implements the check. It is called by the model validation
   * framework for every element of the checked model. It gets the element to
   * check and an abort listener that is used to find out whether the user has
   * aborted the current model validation run. In this case the check also
   * should be aborted. This is only useful for long running checks.
   *
   * The result of the check is an object of type CheckResult (or any other type
   * implementing the ICheckResult interface). This result object contains the
   * problems that were found during the check, i.e. that the unique name of the
   * checked element does not start with 'feature'.
   *
   * @param element
   *          The element to check.
   * @param listener
   *          The abort listener.
   * @return ICheckResult with a vector of problems if the check fails.
   */
  public ICheckResult check(IPVElement element, ICheckAbortListener listener) {
    /*
     * Create an empty CheckResult object. If no problem is added to the result
     * object, then this is interpreted as success by the model validation
     * framework, i.e. the element's unique name starts with 'feature' as
     * claimed by the check.
     */
    CheckResult result = new CheckResult();
    /*
     * Ensure that the element is valid and has a non-empty unique name. Family
     * model elements do not need to have a unique name. Since this check is
     * also applicable for family models, simply ignore elements that have no
     * unique name.
     */
    if (element != null && element.getName().length() > 0) {
      /*
       * Get the unique name of the element. This is name that is to be checked
       * in the next step.
       */
      String name = element.getName();
      if (name.startsWith("feature") == false) { //$NON-NLS-1$
        /*
         * The unique name of the element does not start with 'feature'. This
         * means that the check is failed. To let the user know that the check
         * failed and what exactly is wrong, a problem description is created
         * represented by a CheckProblem object.
         *
         * The first argument of the constructor of class CheckProblem is the
         * name of the check, followed by the check type (an element check),
         * followed by a unique problem code for this problem, followed by the
         * severity of the problem.
         */
        CheckProblem problem = new CheckProblem(element.getModelContainer(), getName(),
            CheckConstants.ELEMENTCHECK_TYPE,
            Messages.CheckElementUniqueNameExample_2, CheckConstants.ERROR_SEVERITY);
        /*
         * To let the model validation framework know on which element to place
         * the problem marker for this problem, the unique ID of the checked
         * element has to be set in the problem description.
         */
```

```
        problem.setElementID(element.getID());
        /*
         * This message is shown to the user for instance as the label of a
         * corresponding problem marker in the Problems View.
         */
        problem.setMessage(MessageFormat.format(Messages.CheckElementUniqueNameExample_3, name));
        result.addProblem(problem);
      }
    }
    return result;
  }
  ...
```

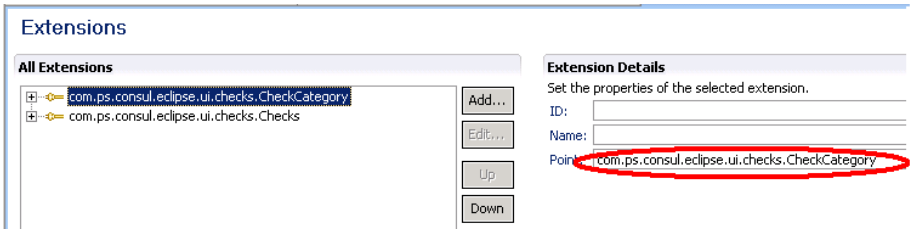
In the next step the new check must be registered as an **Extension** for the Model Validation Framework. For this purpose open the file *plugin.xml* with the **Plug-in Manifest Editor** again and switch to the Extensions page. Click the *Add* button to select **com.ps.consul.eclipse.ui.checks.Checks** extension. After press *Finish* button the new check extension is added to the extensions list. The **Check-Extension** is shown in Figure 5, “Check-Extension”.

Figure 5. Check-Extension



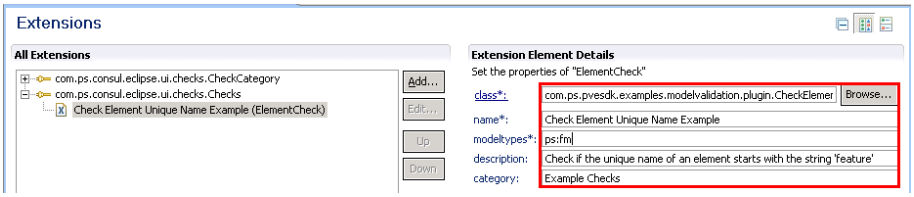
Since each check has a specific category (like whole model or single element check) the example check also needs an extension for the category of the check. Select **com.ps.consul.eclipse.ui.checks.CheckCategory** extension from the extensions list. The **CheckCategory-Extension** is shown in Figure 6, “CheckCategory-Extension”.

Figure 6. CheckCategory-Extension



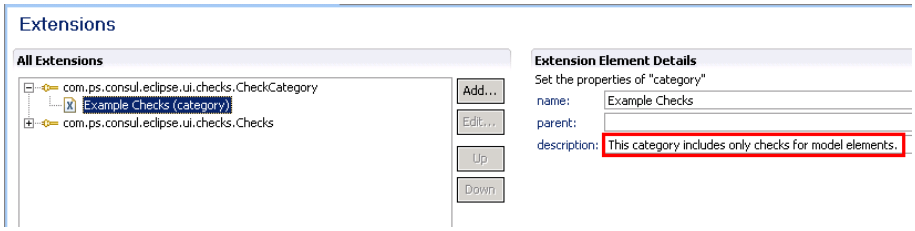
Now a new *Category* and a new *Check* can be added to the Extensions. Right-click on the check extension and choose *New -> ElementCheck* from the context menu. In the description field add a description for the new check. Fill in the other fields as shown in Figure 7, “New ElementCheck”. **class** is the path the check class, **modeltypes** is used to specify for which model types the check is applicable, and **category** specifies the check category the check belongs to.

Figure 7. New ElementCheck



Right-click on the category extension and choose *New -> Category* from the context menu. Add a description and fill in the other fields as shown in Figure 8, “New Category”.

Figure 8. New Category

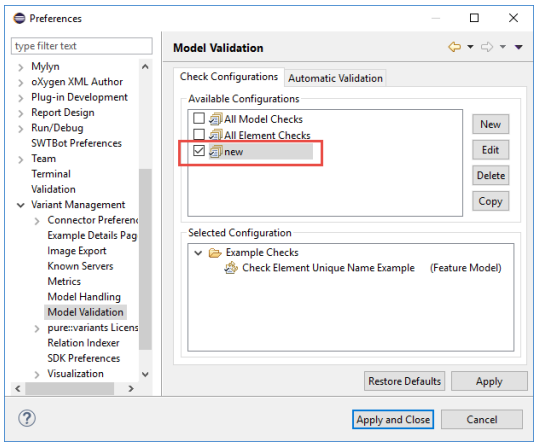


4. Testing the new Check

For testing the new check *CheckElementUniqueNameExample* the plugin has to be installed. Therefor 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*.

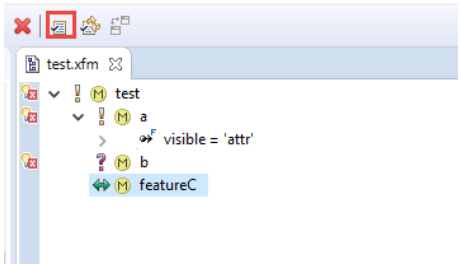
After the Runtime is started or the Deployable Plugin is installed open the **Preferences** by choosing *Window->Preferences* from the Eclipse menu. Change to page *VariantManagement -> Model Validation* where the registered checks can be configured and activated. Create a new check configuration by clicking on button *New*. Choose the example check *Check Element Unique Name Example* as the only check in this configuration (see Figure 9, “New Check Configuration”).

Figure 9. New Check Configuration



After closing the preferences open for instance a feature model. Then click on button *Validate Model* in the Eclipse toolbar. Figure 10, “Validated Model” shows a sample feature model where the check has found three problems. On the left side of the model editor markers are shown for each problem in the model, placed on the corresponding elements. The whole list of problems also is shown in the **Problems View**.

Figure 10. Validated Model



After the check is tested, the next step is to write a quick fix for the problems found by the check (if possible and/or needed).

5. Writing the Quick Fix Implementation

After a model is validated and problems were found in the model, the user can apply automatic quick fixes for these problems if available. This chapter explains how a **quick fix** can be provided for problems found by a check, and how the quick fix can be connected to the check class.

In the package **com.ps.pvesdk.examples.modelvalidation.plugin** create a new Java class with name **CheckElementUniqueNameExampleQuickFix.java**. This class has to be derived from the class **CheckQuickFix**. Each quick fix class has a **m_Label**-Variable, **m_Image**-Variable and **m_Description**-Variable, where the functionality of the check is explained. The **get..()**-methods return the values of this variables. See the following code.

```
public class CheckElementUniqueNameExampleQuickFix extends CheckQuickFix {
  /** Textual description of the quick fix. */
  private String m_Label = "";

  /** Return the detailed description of the quick fix. */
  public String getDescription() {
    return m_Label;
  }
  /** Get the short description of the quick fix. */
  public String getLabel() {
    return m_Label;
  }
  /** Get the image for the quick fix. */
  public Image getImage() {
    ComposeImageManager im = UiPlugin.getDefault().getImageManager();
    return im.getImage(ICheckImages.CHANGE_IMG);
  }

  /**
   * This method is called by the check to initialize the quick fix object. From
   * the given problem marker the quick fix gets the information about the
   * current model and the model element that was checked. With this information
   * it can create a meaningful label and description for the quick fix.
   *
   * @param marker
   *          The problem marker.
   * @param model
   *          The model.
   */
  @Override
  public void initialize(IMarker marker, IPVModel model) {
    /*
     * Get the element that has the problem and that needs to be fixed. For this
     * purpose the class VariantMarkerResolver is used that provides various
     * useful methods to get information from a problem marker. In this case
     * getRelatedElement is used to get the checked element.
     */
    IPVElement element = VariantMarkerResolver.getRelatedElement(marker, model);
    if (element != null) {
      /*
       * Using the element a meaningful label for the quick fix can be created.
       */
      m_Label = MessageFormat.format(Messages.CheckElementUniqueNameExampleQuickFix_1,
       new Object[] { element.getName(), "feature" + element.getName() });
    }
  }
 ...
```

The **initialize()**-method initializes the quick fix by evaluating the given problem marker containing the description of the problem to fix.

Each quick fix has a **run()**-method that is called when the quick fix is applied. It implements the quick fix functionality, i.e. adding the string *feature* to the unique name of the element for which the given problem marker is delivered. The following code shows how the quick fix is implemented.

```
  /**
   * This method is called by Eclipse to perform the quick fix if the user has
   * chosen it from the list of the available quick fixes for a problem. The
   * problem is described in the given problem marker.
```

```
   *
   * This quick fix prepends 'feature' to the unique name of a model element.
   * For this purpose it has to get the checked element, has to calculate the
   * new unique name, and has to set the new unique name to the element. If the
   * quick fix succeeds it has to remove the problem marker to show the user
   * that the problem is fixed.
   *
   * @param marker
   *          The problem marker.
   * @param op
   *          The {@link ModelOperation}.
   * @throws CoreException
   */
  @Override
  public void run(IMarker marker, ModelOperation op) throws CoreException {
    /*
     * Get the checked element to change its unique name. As explained above,
     * first the model has to be opened and then the element can be got from the
     * marker.
     */
    IPVElement element = VariantMarkerResolver.getRelatedElement(marker, op.getModel());

    /*
     * Calculate the new unique name by prepending 'feature' to the original
     * unique name of the element.
     */
    String newname = "feature" + element.getName(); //$NON-NLS-1$
    /*
     * Changes on the element cannot be performed directly. Instead the element
     * has to be put into changing mode.
     */
    Element changed = op.changeElement(element);
    /*
     * Now the name can be set using the Model API.
     */
    changed.setName(newname);
    /*
     * The changes are executed by calling the perform() method of the
     * ModelOperation. After that call, the model changes are committed.
     */
    op.perform();
  }
```

For connecting the quick fix with the example check, two methods have to be added to the check class. The first method, **hasResolutions()**, has to return true if there are quick fixes for problems reported by the check. The second method, **getResolutions()**, returns the available quick fixes and is only called when **hasResolutions()** has returned true.

```
  /**
   * This method is called by the model framework to find out if
   * there are any quick fixes available for the given problem marker.
   */
  public boolean hasResolutions(IMarker marker){
    return true;
  }

  /**
   * This method is called by the model validation framework
   * if hasResolutions returned true for the given marker.
   */
  public IMarkerResolution[] getResolutions(IMarker marker){
    // This vector is used to collected the quick fixes
    // for the problems reported by this check.
    Vector resolutions = new Vector();

    // Create and initialize a quick fix object.
    CheckElementUniqueNameExampleQuickFix fix =
      new CheckElementUniqueNameExampleQuickFix();
    fix.initialize(marker);

    // Add the quick fix to the vector.
    resolutions.add(fix);
    ...
```

If a model contains several problems of the same type, then a **MultiQuickFix** can be added for fixing these problems at once. For this a new *MultiQuickFix()*-object has to be created and added to the *resolutions* vector.

```
   ...
   MultiQuickFix multifix = new MultiQuickFix(fix.getID());
```

```
  // Set an image and a label for the multi quick fix.
  ComposeImageManager im = UiPlugin.getDefault().getImageManager();
  multifix.setImage(im.getImage(ICheckImages.CHANGE_IMG));
  multifix.setLabel("Apply the same fix to all elements
                    with the same problem");

  // Initialize and add the quick fix.
  multifix.initialize(marker);
  resolutions.add(multifix);
  return (IMarkerResolution[]) resolutions.toArray(
    new IMarkerResolution[resolutions.size()]);
}
```

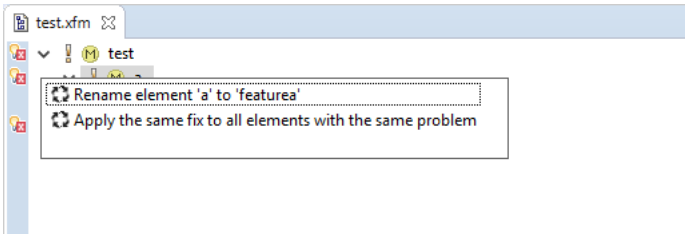
6. Testing the new Quick Fix

This chapter shows how the problem reported by the example check can be fixed automatically using the provided quick fix.

Install the plugin and change the preferences as described in chapter 4. Then open a model and click on the *Validate Model* button in the toolbar of Eclipse. If the check found elements in the model with unique names not starting with 'feature', then markers are shown on the left side of the editor and in the **Problems View**.

If quick fixes are available for a problem, then a yellow lamp is shown at the corresponding problem marker. Left-clicking on such a marker opens a window with the list of available quick fixes for the problem (see Figure 11, “Resolve Problem()”). For the example check two quick fixes are shown. The first renames the corresponding element by predending 'feature' to its unique name. The second is the multi quick fix that applies all the quick fixes for problems found by the example check.

Figure 11. Resolve Problem()



7. Deploying the new Check

To be able to install the new plugin in an Eclipse installation, the plugin has to be exported as "Deployable Plugin". 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*.