Eclipse application model modularity with fragments and processors - Tutorial

Lars Vogel

Version 5.9

08.07.2013

Eclipse Application Model contributions

This tutorial explains how to contribute to the Eclipse application model via other plug-ins.

Table of Contents

1. Prerequisites

The following assumes that you have already basic Eclipse RCP development experience. See the **Eclipse RCP Tutorial** for details. .

2. Modularity support in Eclipse RCP

Eclipse RCP applications are based on OSGi and therefore support the modularity concept of OSGi. To contribute to the application model, the Eclipse platform implements support for static and dynamic contributions.

The initial structure of an RCP application is described via the application model in the *Application.e4xmi* file.

Other plug-ins can extend this base application model with contributions. Model contributions can be statically defined in files. These extensions are called fragments or *model fragments*. Model contributions can also extend the model dynamically via code. These extensions are called *processors* or *model processors*.

These model contributions are registered with the Eclipse framework via an extension point. To register your contributions you provide extensions to the `org.eclipse.e4.workbench.model` extension point.

This extension point is defined in the `org.eclipse.e4.ui.workbench` plug-in.

The model contributions are read during startup and the contained information is used to build the runtime application model.

3. Contributing to the application model

3.1. Model fragments

A model *fragment* is a file which typically ends with the *.e4xmi* extension. It statically specifies model elements and the location in the application model to which it should be contributed.

For example a fragment can define that it extends a certain menu with additional menu entries.

The e4 tools project provides a wizard and an editor for model fragments.

> **Tip:** The application model editor also allows you to extract a subtree into a new or existing fragment. Select a model element, right click on it and select *Extract into a fragment* from the context menu.

3.2. Model processors

A *processor* allows you to contribute to the model via program code. This enables the dynamic creation of model elements during the start of the application.

3.3. Position of new model elements

Fragments define the desired position of new model elements via the *Position in List* attribute. The following values are allowed:

Table 1. Position in list

Value	Description
first	Positions the element on the beginning of the list.
index: *theIndex* Example index:0	Places the new model elements at position *theIndex*.
before: *theOtherElementsId*	Places the new model elements before the model element with the ID *theOtherElementsId*.
after: *theotherelementsid*	Places the new model elements after the model element with the ID *theotherelementsid*.

fragments of independent plug-ins are processed in arbitrary order by the Eclipse runtime, therefore *first* or *index* might not always result in the desired outcome.

3.4. Usage of IDs

If you want to contribute to an element of the application model you must specify the ID of the element to which you are contributing.

> **Tip:** In general it is good practice to always specify unique IDs in your application model. If not you may experience strange application behavior.

3.5. Comparison with Eclipse 3.x

The programming model of Eclipse 3.x primarily uses extension points to define contributions to the application. These extensions define new parts, new menus, etc. This approach is no longer used in Eclipse 4 RCP applications. All contributions are made via fragments or processors.

4. Constructing the runtime application model

4.1. User Changes

Changes during runtime, are written back to the model. An example for such a change is that the user moves a part to a new container via drag and drop.

If the RCP application is closed, theses changes are recorded and saved independently in a *workbench.xmi* file in the *.metadata/.plugins/org.eclipse.e4.workbench* folder.

> **Tip:** User changes can be deleted at start of your application via the *clearPersistedState* parameter as a launch parameter. In most cases which is undesired for an exported application and only used during development.

4.2. Runtime application model

At runtime the application model of an Eclipse application consists of different components:

- Application model - By default defined via the Application.e4xmi file
- Model contributions - Based on fragments and processors
- User changes - Changes the user did to the user interface during his last usage

These different components of the runtime application model need to be combined.

The Eclipse platform creates the runtime application model based on the initial application model (Application.e4xmi) and applies the model contributions to it. User deltas are applied afterwards. If these deltas do not apply anymore, e.g. because the base model has changed, they will be skipped.

The deltas are applied to the model based on the IDs of the user interface component.

> **Note:** This behavior can be surprising during development. The developer adds a new part and this part is not visible after startup of the application because Eclipse assumes that the user closed it in an earlier session. Use the *clearPersistedState* parameter to avoid the processing of user changes at startup.

5. Fragment extension elements

In fragments you contribute to an existing model element which is defined via its ID. You also have to specify the *Featurename* to which you want to contribute. A *Featurename* is a direct link to the structure of the application model.

The following table lists some *Featurename* values and their purposes.

Table 2. Contribution, Featurename and Element id

You want to contribute to a	Featurename	Element Id
Command to the application	commands	ID of your application
Handler to the application	handlers	ID of your application
New MenuItem / HandledMenuItem to existing menu	children	ID of the menu

You want to contribute to a	Featurename	Element Id
New menu to the main menu of the window	children	ID of your main menu
New Part to existing PartStack	children	ID of your PartStack

6. Exercise: Contributing via model fragments

6.1. Target

In this exercise you create a model fragment to contribute a menu entry, a command and a handler to your application model.

6.2. Create new plug-in

Create a simple plug-in project called *com.example.e4.rcp.todo.contribute*. The following description abbreviates the plug-in name to the `contribute` plug-in.

6.3. Add dependencies

In the `MANIFEST.MF` file, add the following plug-ins as dependencies to your `contribute` plug-in.

- org.eclipse.swt
- org.eclipse.jface
- org.eclipse.e4.core.di
- org.eclipse.e4.ui.workbench
- javax.inject
- org.eclipse.e4.ui.di

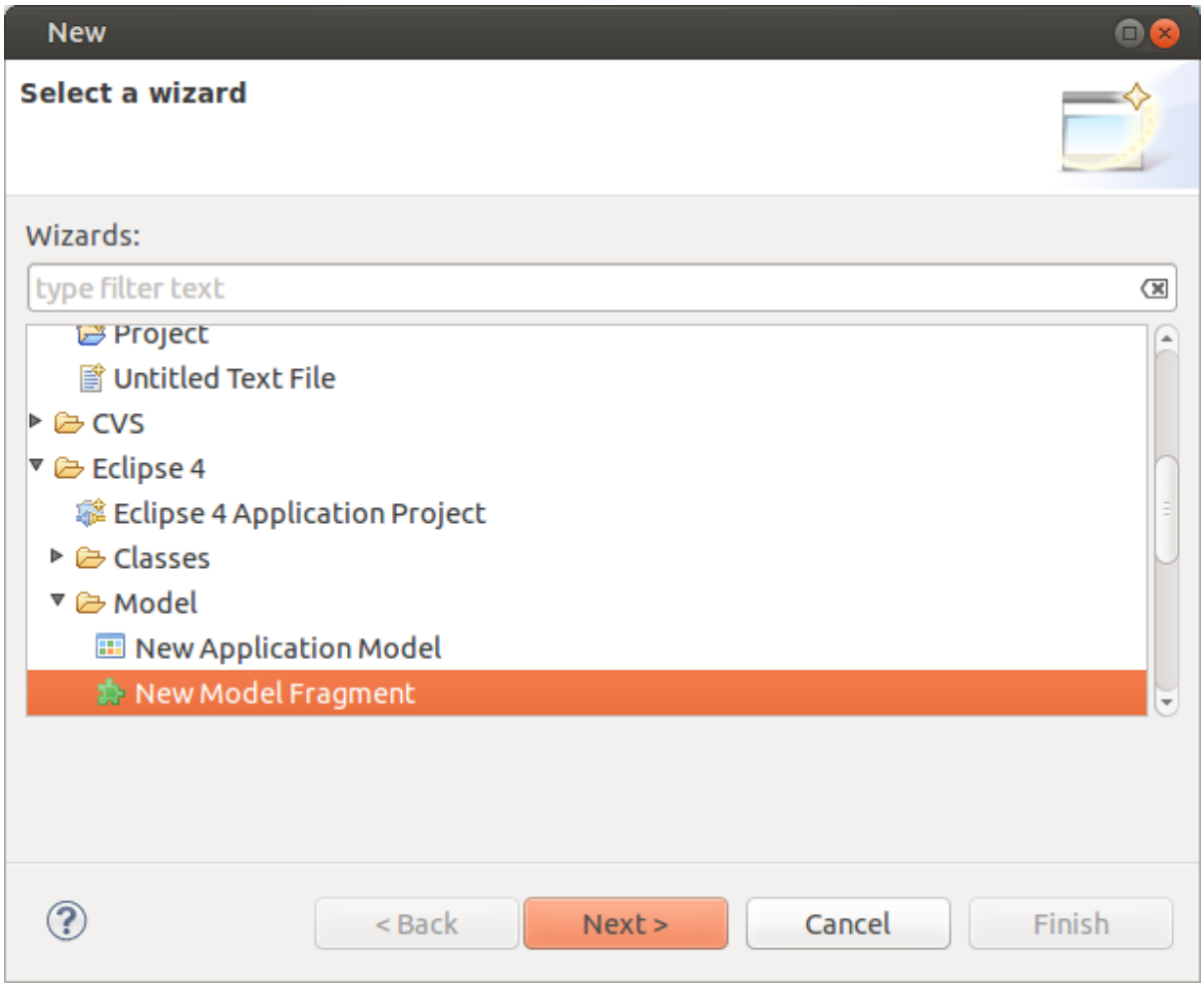
6.4. Create a handler class

Create the `com.example.e4.rcp.todo.contribute.handlers` package and the following class.

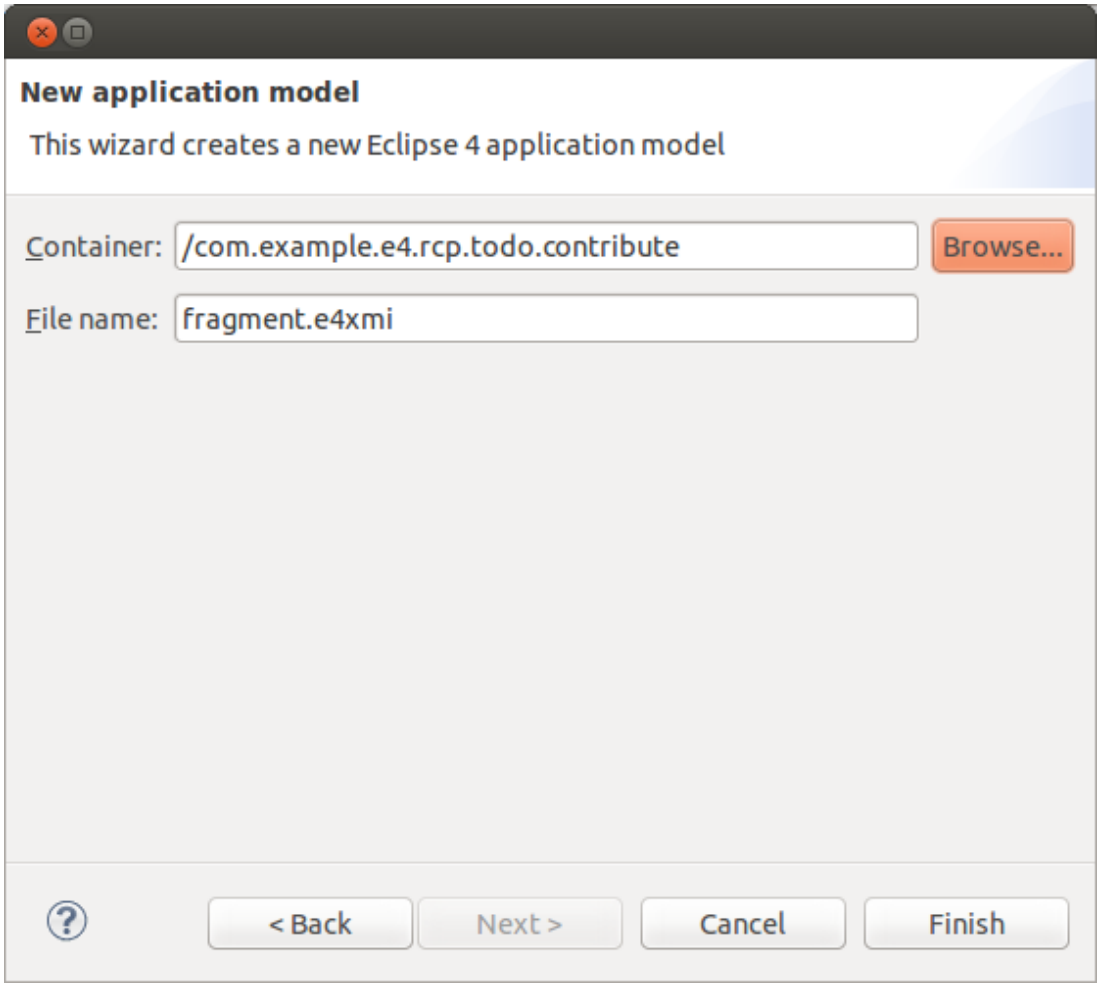
```
package com.example.e4.rcp.todo.contribute.handlers;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Shell;

public class OpenMapHandler {

  @Execute
  public void execute(Shell shell) {
    MessageDialog.openInformation(shell, "Test", "Just testing");
  }
}
```

6.5. Create a model fragment

Use the fragment wizard from the e4 tools project to create a new model fragment via the following menu: *File → New → Other... → Eclipse 4 → Model → New Model Fragment*.



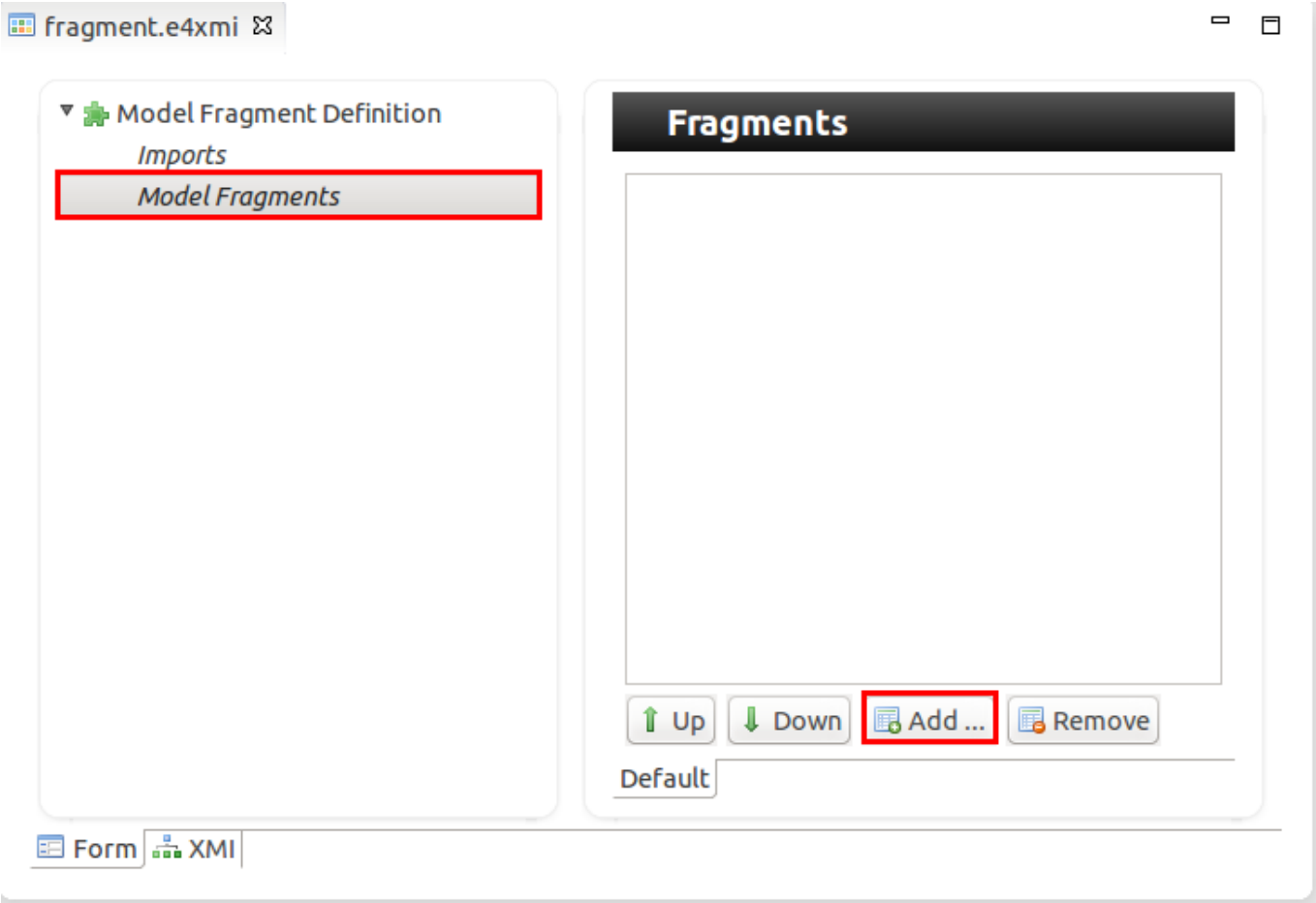
Select the `contribute` plug-in as the container and use *fragment.e4xmi* as the name for the file.



Press the *Finish* button.

6.6. Adding model elements

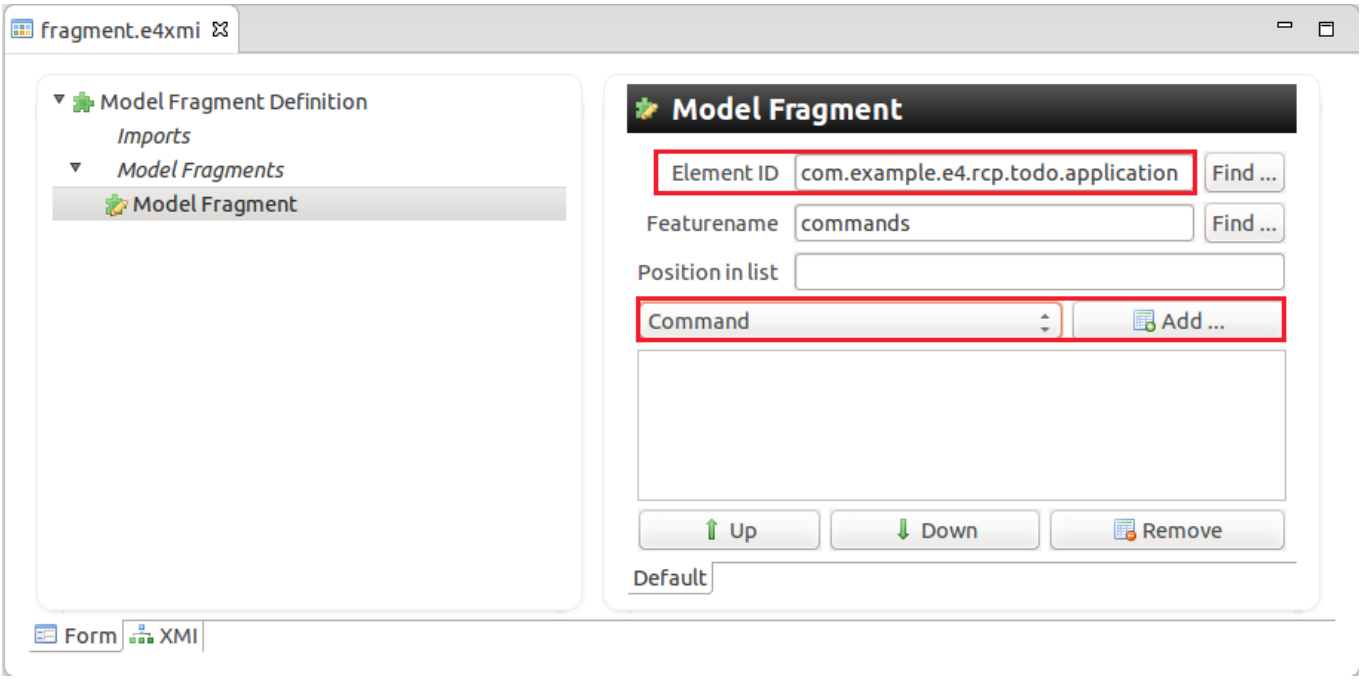
Afterwards the new file is opened in the model fragment editor. Select the *Model Fragments* node and press the *Add...* button.



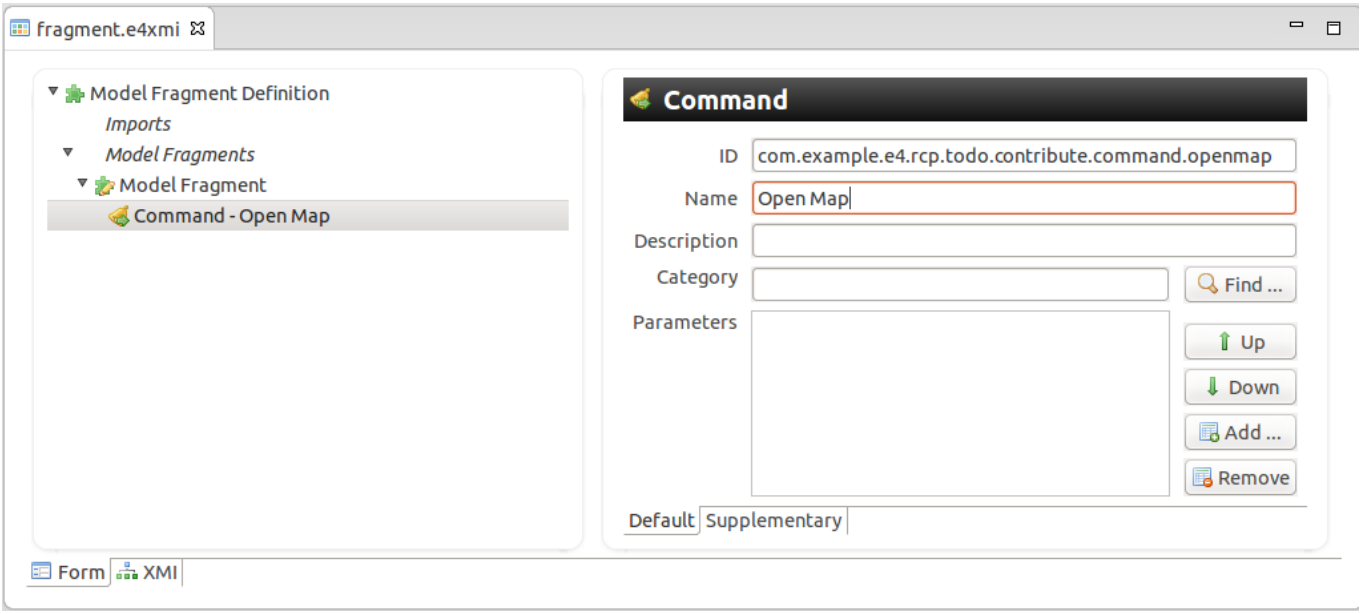
Use `com.example.e4.rcp.todo.application` as the *Element ID*. This is the ID of the *Application* model element in your *Application.e4xmi* file.

> **Warning:** Ensure that `com.example.e4.rcp.todo.application` is the ID you are using for the top node in the *Application.e4xmi* file. Otherwise the contribution does not work. This is because the Eclipse runtime does not find the correct model element to contribute to.

You also need to define to which feature you will be adding to. For *Featurename*, specify the value *commands*. Make sure you have the *Model Fragment* selected and use the *Add...* button to add a *Command* to your model fragment.

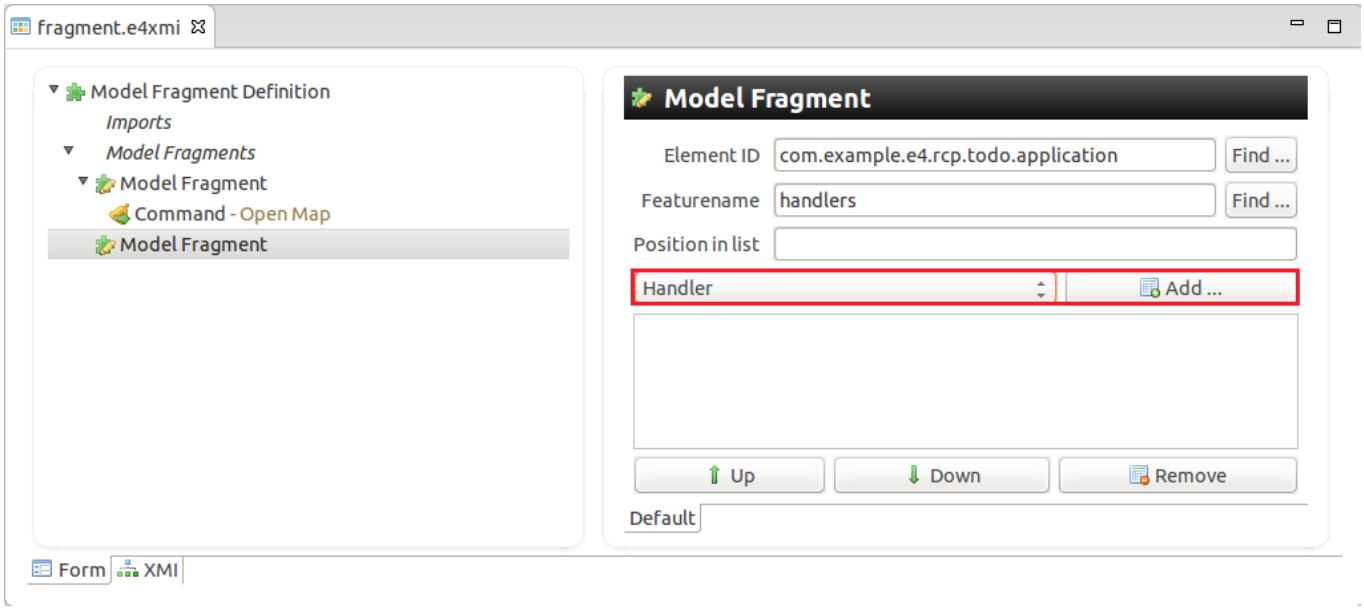


Use `com.example.e4.rcp.todo.contribute.command.openmap` for the *ID* field and `Open Map` for the *Name* field.

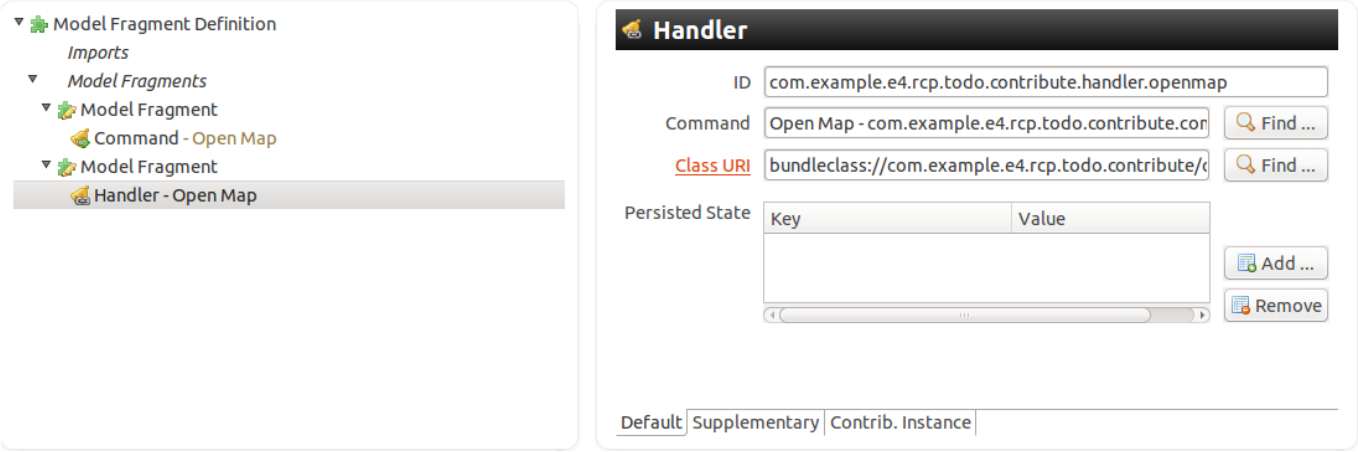


Create a new model fragment for the handler. The *Element ID* is again your application ID, the *Featurename* is `handlers`.

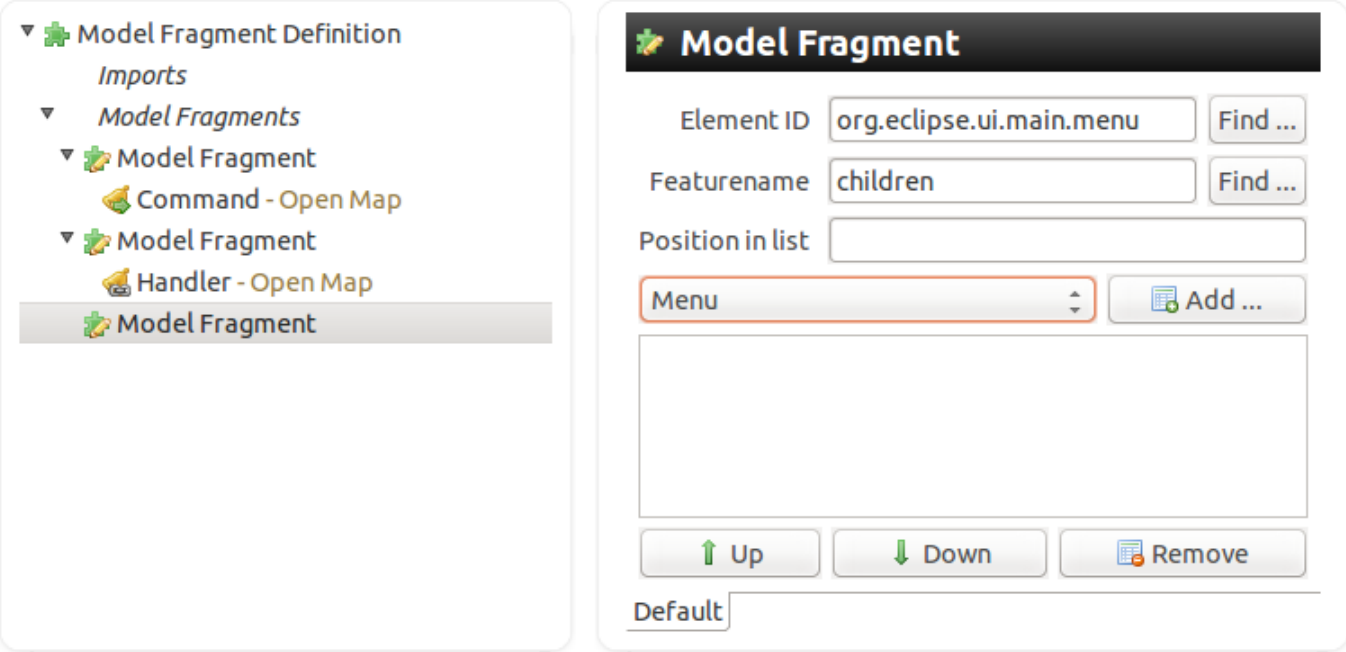
Add a *Handler* to this model fragment.



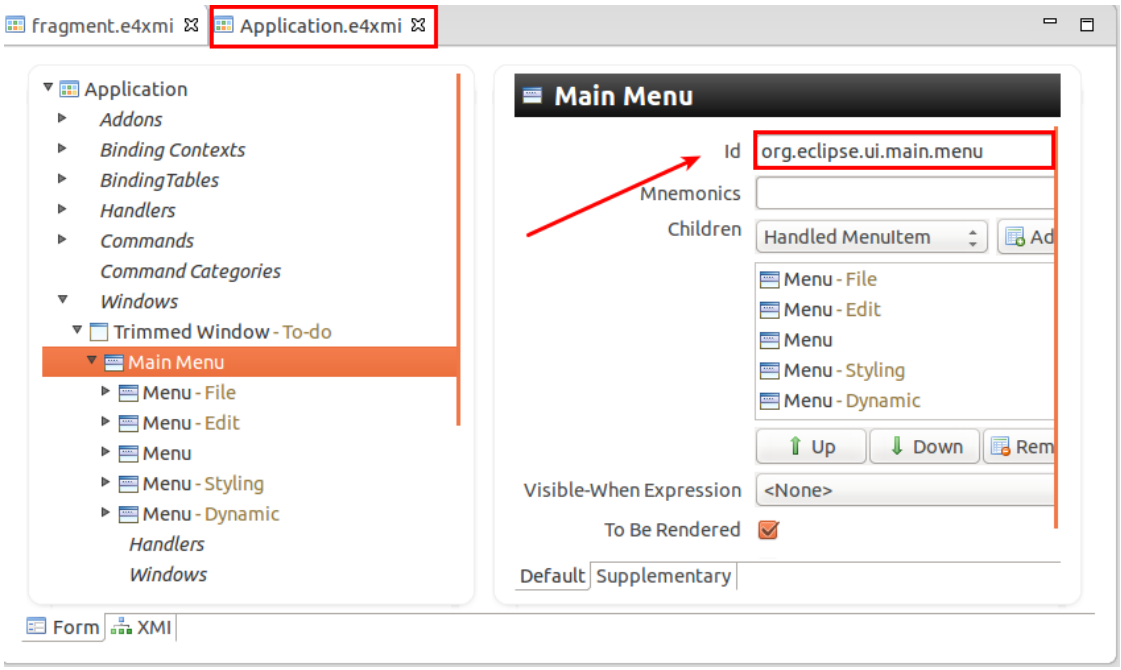
Use `com.example.e4.rcp.todo.contribute.handler.openmap` as ID for the handler. Point to the *Open Map* command and the `OpenMapHandler` class.



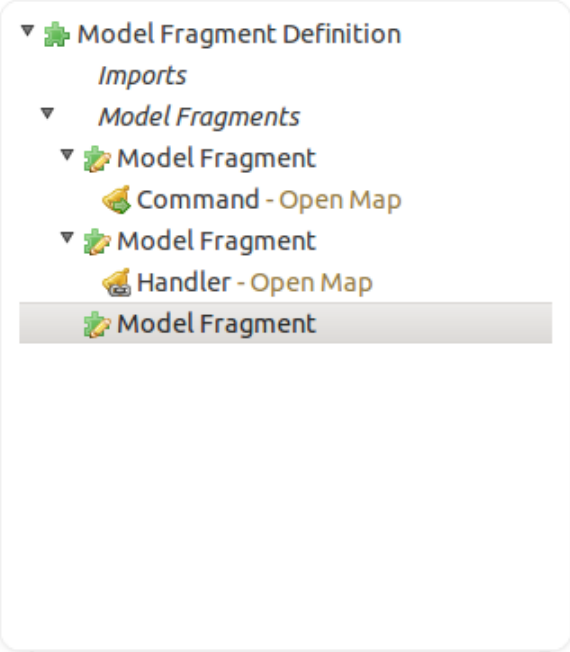
Add another *Model Fragment* to contribute a new menu to your application model. Contribute to the main menu of your *Application.e4xmi*. If you followed the earlier exericses correctly this should be the `org.eclipse.ui.main.menu` ID. The *Featurename* is `children`.

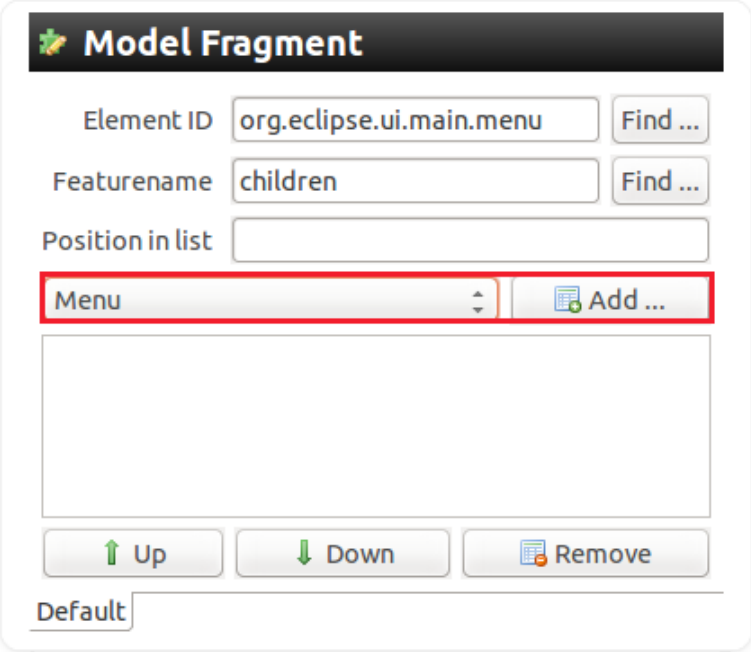


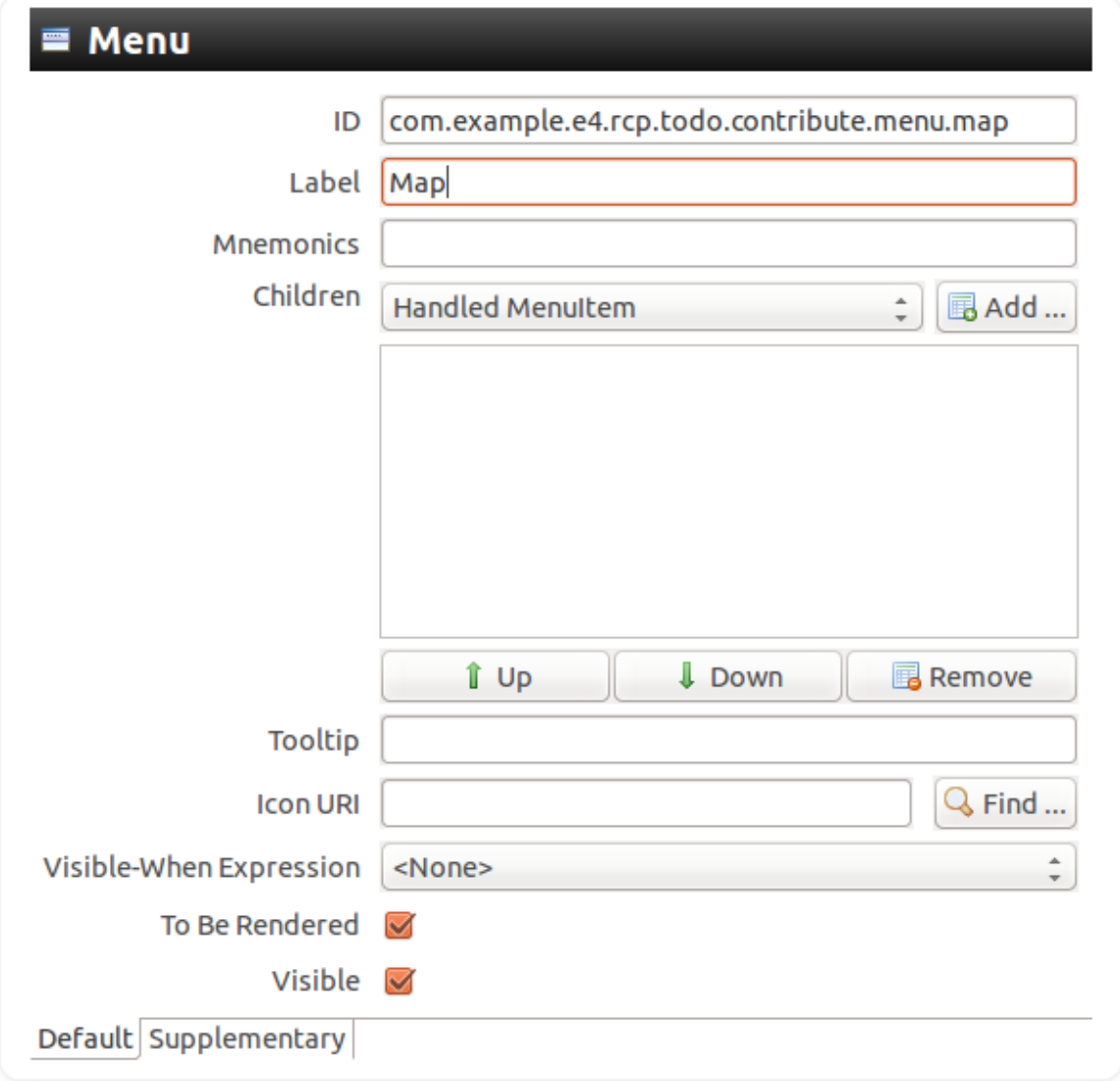
Warning: Ensure in your *Application.e4xmi* file that you are using the same ID for your menu and your application. The following screenshot highlights this entry.



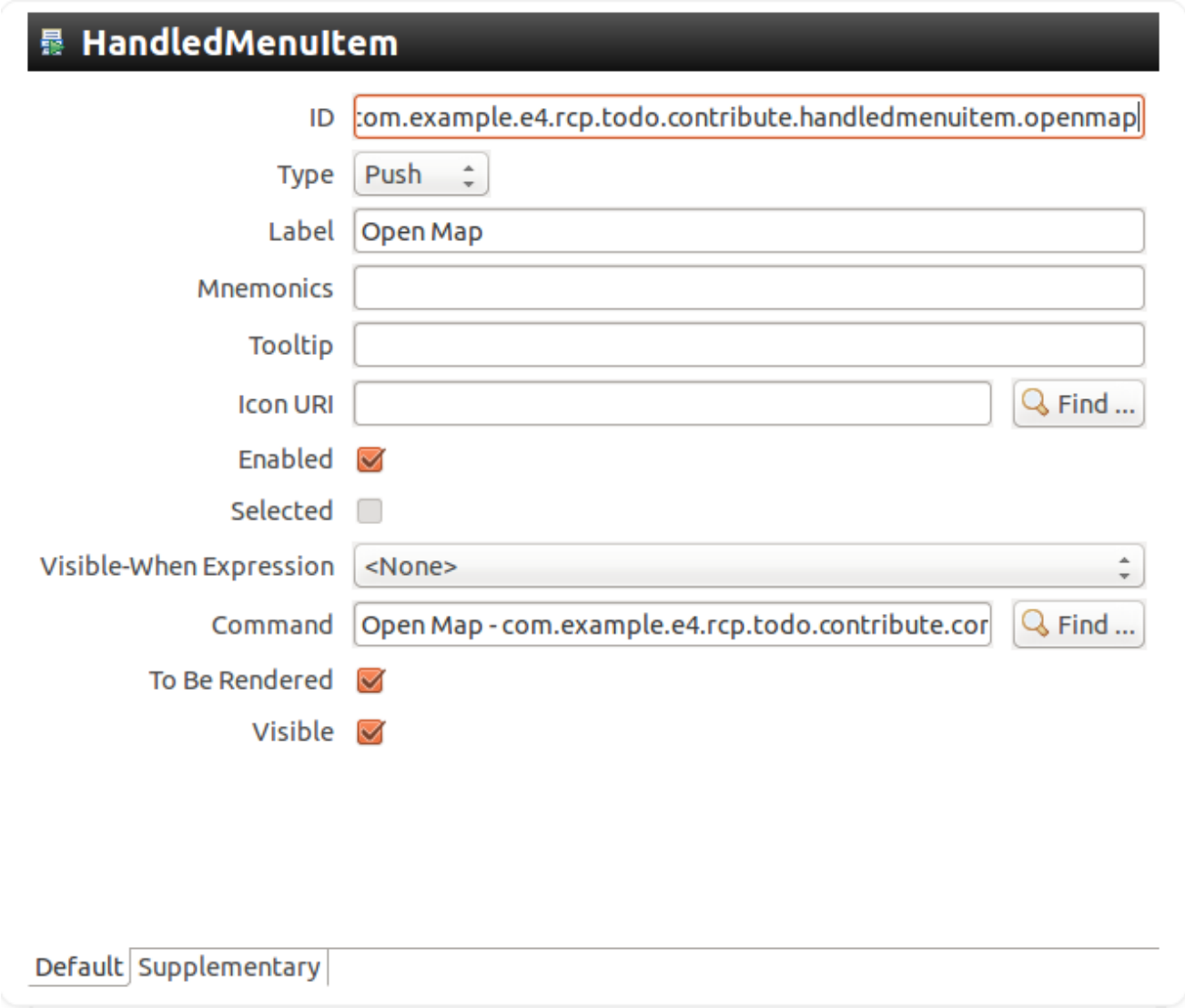
In your *fragment.e4xmi* file add a *Menu* with the *com.example.e4.rcp.todo.contribute.menu.map* ID and the *Map* label.







Add a *HandledMenuItem* which points to your new command. The process of defining these entries is the same as defining menus in the *Application.e4xmi* file. See **???** for further information. The created entry should be similar to the following screenshot.

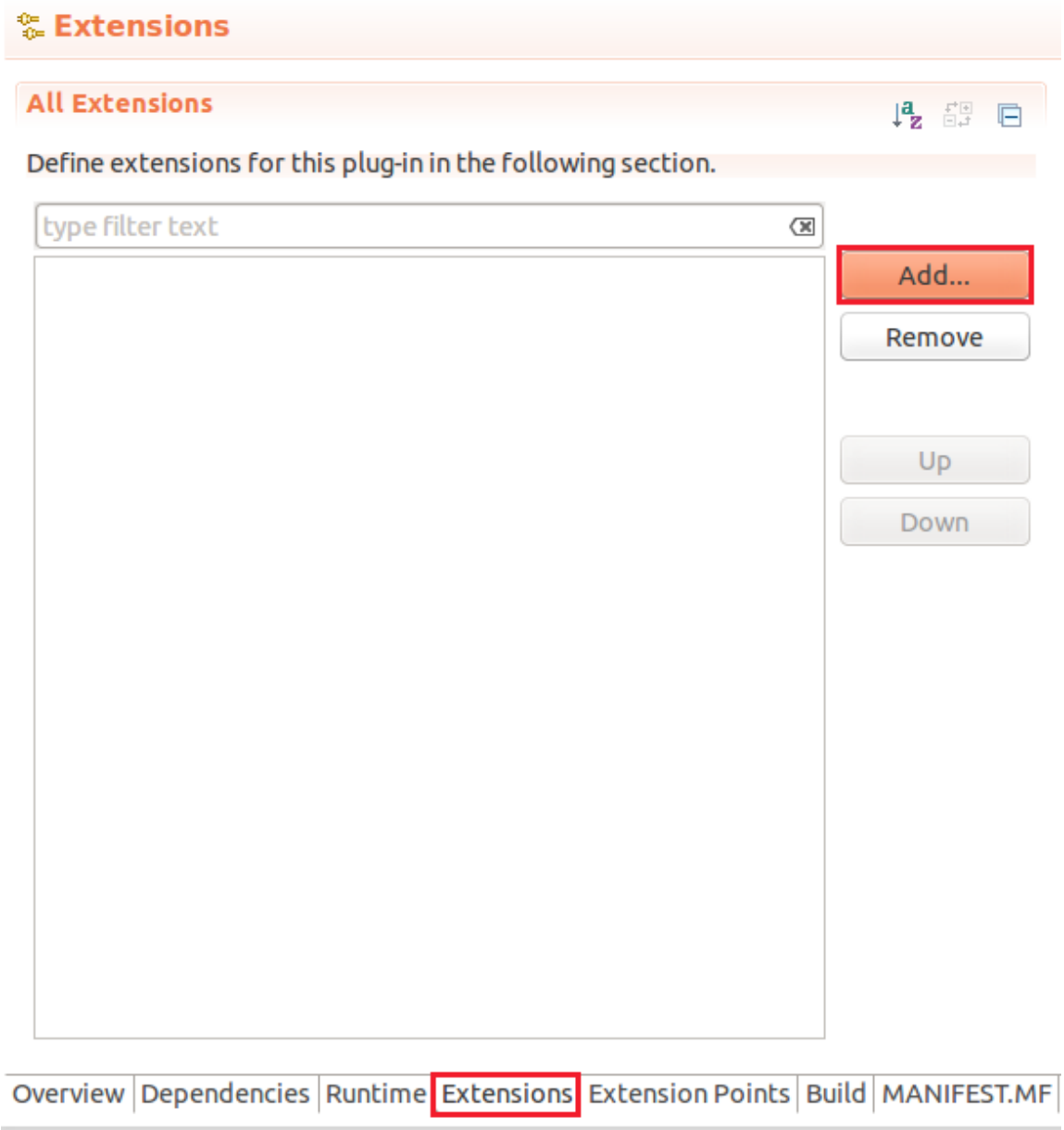


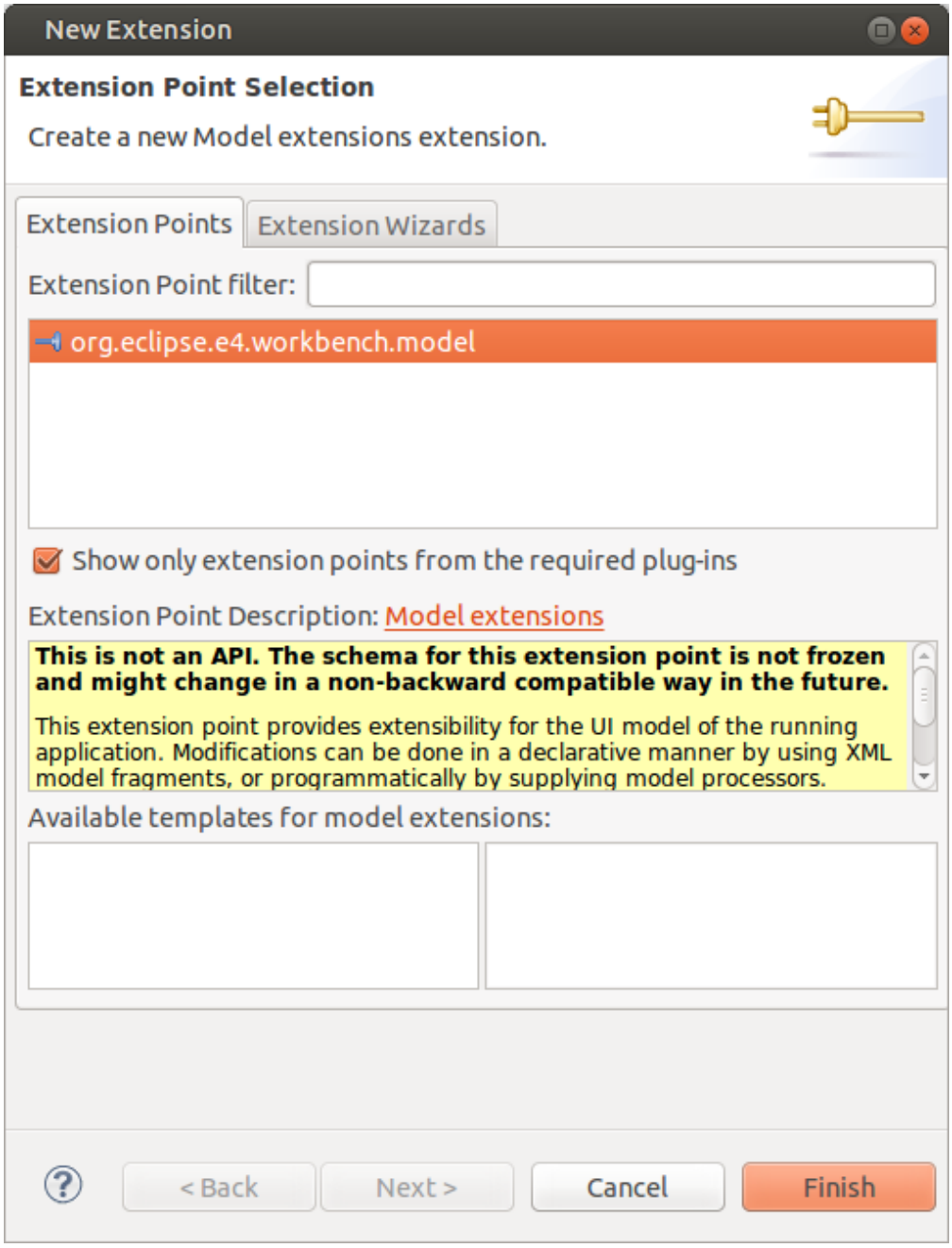
6.7. Register the fragment via extension

Add the `org.eclipse.e4.workbench.model` extension to your `contribute` plug-in. For this open the *plugin.xml* file.

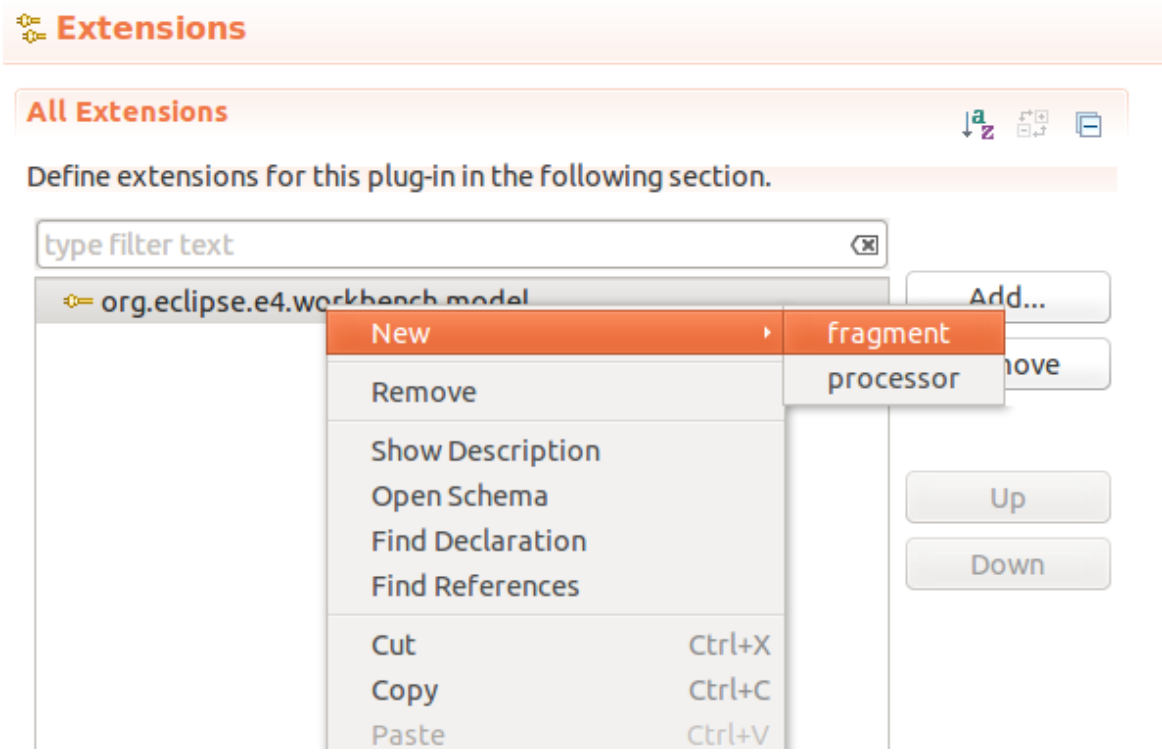
> **Tip:** If the *plugin.xml* file is missing, open your *MANIFEST.MF* file, select the *Overview* tab and click on the *Extensions* link. This shows the *Extensions* tab in the editor and once you add an extension in this tab the *plugin.xml* file is generated.

On the *Extensions* tab, click the *Add...* button to add a new extension for the `org.eclipse.e4.workbench.model` extension point.

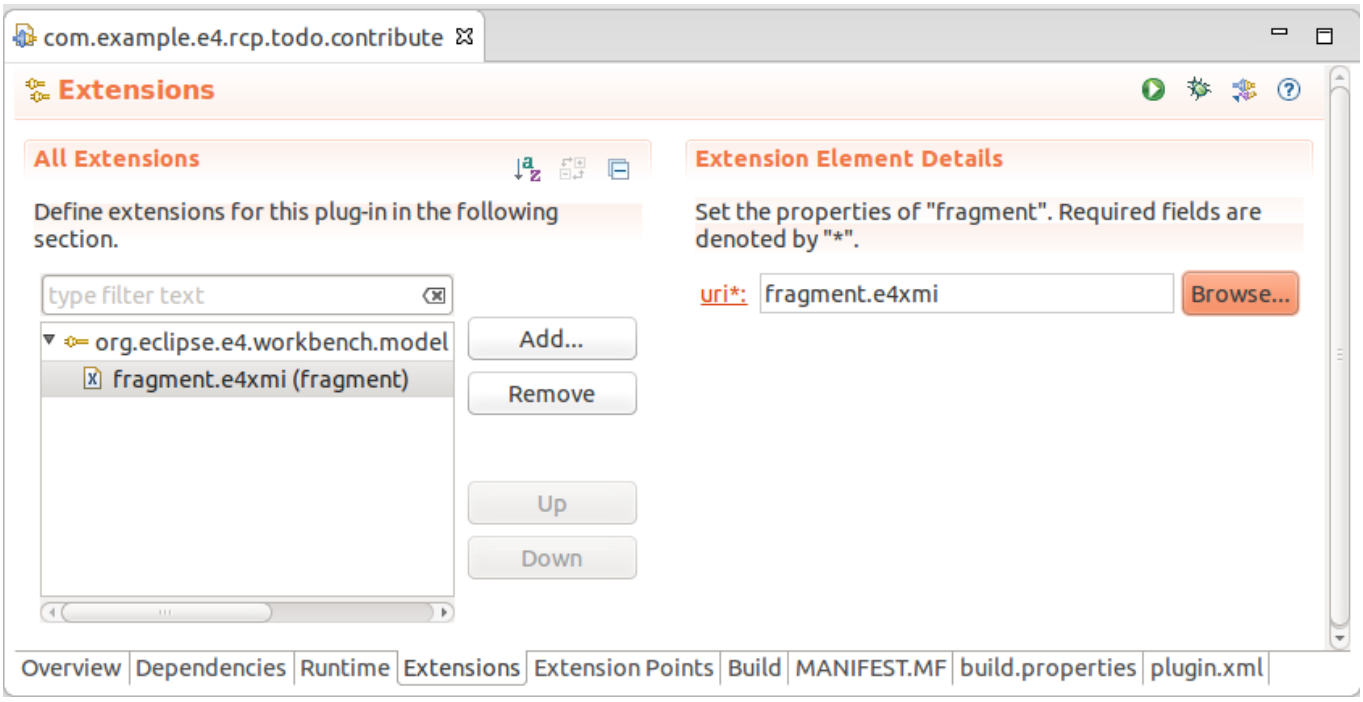

Extensions
All Extensions
Define extensions for this plug-in in the following section.
type filter text
Add...
Remove
Up
Down
Overview
Dependencies
Runtime
Extensions
Extension Points
Build
MANIFEST.MF



Right-click on the extension and select *New* → *fragment*.



Use the *Browse...* button to point to your model fragment file.



The resulting plugin.xml file should look similar to the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
   <extension
         id="modelContribution"
          point="org.eclipse.e4.workbench.model">
      <fragment
            uri="fragment.e4xmi">
      </fragment>
   </extension>

</plugin>
```

6.8. Update product via feature

Add the `contribute` plug-in to your `com.example.e4.rcp.todo.feature` feature.

6.9. Validate

Start your application.

> **Warning:** Remember to start via the product to update the launch configuration.

You should see the new *Map* entry in the application menu. If you select this entry a message dialog opens.

If the menu entry is not displayed, ensure that your IDs are correctly entered and that you either use the *clearPersistedState* flag or clear the workspace data in your *Launch configuration*.

6.10. Exercise: Contributing a part

> **Note:** This exercise is optional.

Define a new model fragment which contributes a part to an existing *PartStack*. Use the ID of an existing *PartStack* and use `children` as *FeatureName*.

7. Exercise: Implementing model processors

7.1. Target

In this exercise you replace an existing menu entry with another menu entry.

7.2. Enter dependencies

Continue to use the `com.example.e4.rcp.todo.contribute` plug-in for this exercise.

In the *MANIFEST.MF*, add the following plug-ins as dependencies to your *contribute* plug-in.

- `org.eclipse.e4.ui.services`
- `org.eclipse.e4.core.contexts`
- `org.eclipse.e4.ui.model.workbench`

7.3. Create Java classes

Create the following dialog and handler classes.

```
package com.example.e4.rcp.todo.contribute.dialogs;

import javax.inject.Inject;
import javax.inject.Named;

import org.eclipse.e4.ui.services.IServiceConstants;
import org.eclipse.jface.dialogs.Dialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;

public class ExitDialog extends Dialog {
  @Inject
  public ExitDialog(@Named(IServiceConstants.
      ACTIVE_SHELL) Shell shell) {
    super(shell);
  }

  @Override
  protected Control createDialogArea(Composite parent) {
    Label label = new Label(parent, SWT.NONE);
    label.setText("Closing this application may result in data loss. "
        + "Are you sure you want that?");
    return parent;
  }

}
```

```java
package com.example.e4.rcp.todo.contribute.handlers;

import org.eclipse.e4.core.contexts.ContextInjectionFactory;
import org.eclipse.e4.core.contexts.IEclipseContext;
import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.workbench.IWorkbench;
import org.eclipse.jface.window.Window;

import com.example.e4.rcp.todo.contribute.dialogs.ExitDialog;

public class ExitHandlerWithCheck {
  @Execute
  public void execute(IEclipseContext context, IWorkbench workbench) {
    ExitDialog dialog = ContextInjectionFactory.
        make(ExitDialog.class, context);
    dialog.create();
    if (dialog.open() == Window.OK) {
      workbench.close();
    }
  }
}
```

Create the model processor class. This class removes all menu entries which have "exit" in their ID in the menu with the `org.eclipse.ui.file.menu` ID. It also adds a new entry.

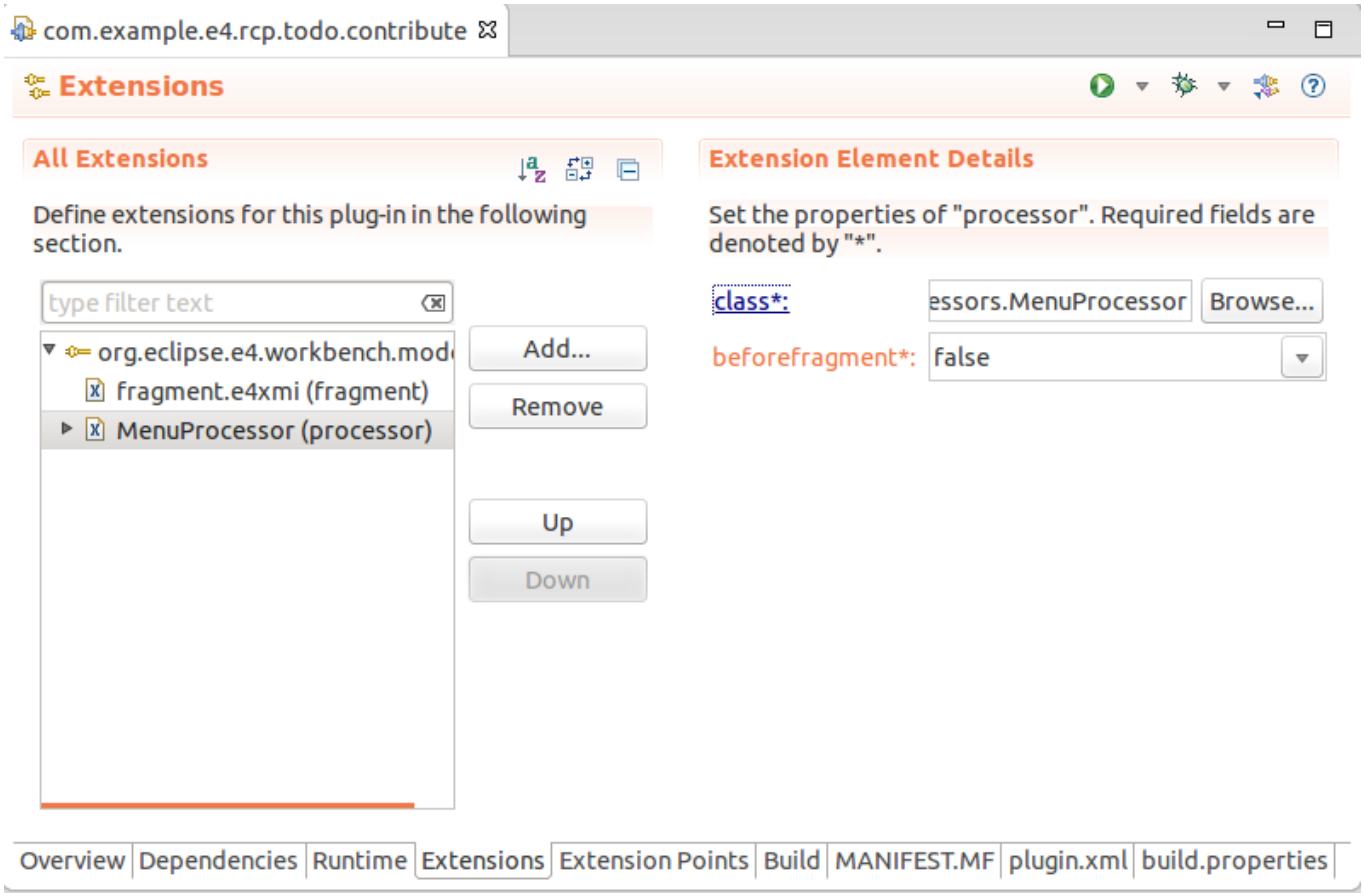
```java
package com.example.e4.rcp.todo.contribute.processors;

import java.util.ArrayList;
import java.util.List;

import javax.inject.Inject;
import javax.inject.Named;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.model.application.ui.menu.MDirectMenuItem;
import org.eclipse.e4.ui.model.application.ui.menu.MMenu;
import org.eclipse.e4.ui.model.application.ui.menu.MMenuElement;
import org.eclipse.e4.ui.model.application.ui.menu.MMenuFactory;

import com.example.e4.rcp.todo.contribute.handlers.ExitHandlerWithCheck;

public class MenuProcessor {

  // the menu is injected based on the parameter
  // defined in the extension point
  @Inject
  @Named("org.eclipse.ui.file.menu")
  private MMenu menu;

  @Execute
  public void execute() {
    // starting processor
    // remove the old exit menu entry
    if (menu != null && menu.getChildren() != null) {
      List<MMenuElement> list = new ArrayList<MMenuElement>();
      for (MMenuElement element : menu.getChildren()) {
        System.out.println(element);

        // Use ID instead of label as label is later translated
        if (element.getElementId() != null) {
          if (element.getElementId().contains("exit")) {
            list.add(element);
          }
        }
      }
      menu.getChildren().removeAll(list);
    }

    // now add a new menu entry
    MDirectMenuItem menuItem = MMenuFactory.INSTANCE.createDirectMenuItem();
    menuItem.setLabel("Another Exit");
    menuItem.setContributionURI("bundleclass://"
        + "com.example.e4.rcp.todo.contribute/"
        + ExitHandlerWithCheck.class.getName());
    menu.getChildren().add(menuItem);
  }
}
```

Warning: Ensure that your menu entry labeled with "Exit" in the *Application.e4xmi* file, contains

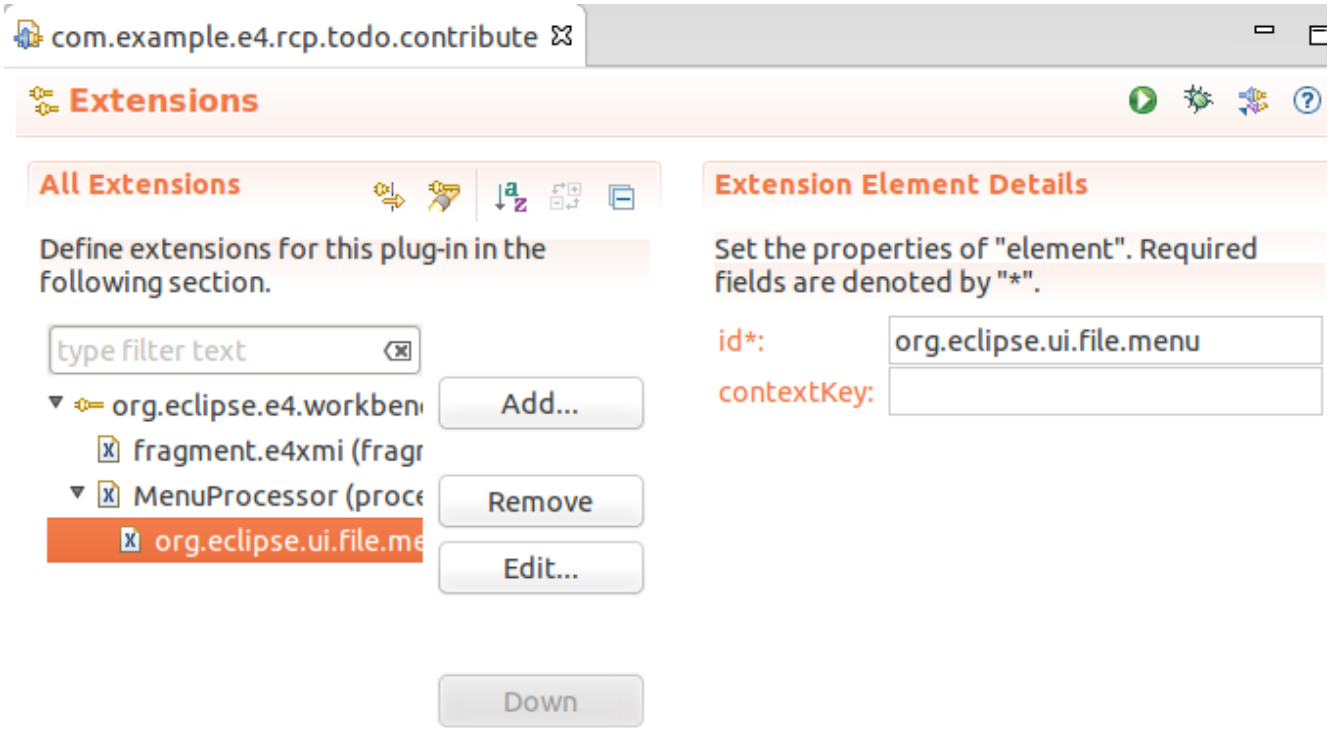
"exit" in its ID.

7.4. Register processor via extension

In your `contribute` plug-in register your `processor` via the `org.eclipse.e4.workbench.model` extension.



Right-click on the processor and select *New → element*. The parameter with the ID is the model element which is injected into your processor class. Use *org.eclipse.ui.file.menu* as *id** parameter.



> **Warning:** This assumes that you used *org.eclipse.ui.file.menu* as ID for your *File* menu in the main application model.
>
> **Warning:** The ID of the element defined in the extension point must match the `@Named` value in the processor, otherwise your menu is not injected into the processor.

7.5. Validate

Start your application. In the model fragment exercises, the contribute plug-in was already added to your product.

Ensure that the existing "Exit" menu entry is removed and your new menu entry with the "Another Exit" label is added to the file menu.

8. Learn more about Eclipse 4 RCP development

I hope you enjoyed this tutorial. You find this tutorial and much more information also in the **Eclipse 4 RCP book** from this author.

9. About this website

9.1. Donate to support free tutorials

Please consider a contribution  if this article helped you. It will help to maintain our content and our Open Source activities.

9.2. Questions and discussion

Writing and updating these tutorials is a lot of work. If this free community service was helpful, you can support the cause by giving a tip as well as reporting typos and factual errors.

If you find errors in this tutorial, please notify me (see the **top of the page**). Please note that due to the high volume of feedback I receive, I cannot answer questions to your implementation. Ensure you have read the **vogella FAQ** as I don't respond to questions already answered there.

9.3. License for this tutorial and its code

10. Links and Literature

10.1. Source Code

Source Code of Examples

10.2. Links and Literature

http://wiki.eclipse.org/E4 Eclipse E4 - Wiki

Eclipse RCP

Eclipse EMF

Dependency Injection