Eclipse RCP (Rich Client Platform) - Tutorial

Lars Vogel (c) 2009, 2016 vogella GmbH – Version 0.3, 06.07.2016

Table of Contents

This tutorial gives an overview about creating Eclipse RCP applications.

1. Eclipse RCP applications

1.1. What are Eclipse RCP applications?

An Eclipse RCP application is a stand-alone application based on Eclipse platform technologies. This tutorial uses the terms *Eclipse based applications*, *Eclipse application*, *Eclipse 4 application* and *Eclipse RCP application* interchangeably for referring to such applications.

The Eclipse IDE version 2.0 started as a modular IDE application. In 2004 Eclipse version 3.0 was released. Eclipse 3.0 supported reusing components of the Eclipse platform to build stand-alone applications based on the same technology as the Eclipse IDE.

At this point, the term *Eclipse RCP* was coined. Eclipse RCP is short for *Eclipse Rich Client Platform* and indicates that the Eclipse platform is used as a basis to create feature-rich stand-alone applications.

The release of Eclipse in version 4.x simplified and unified the Eclipse programming model which is now based on state-of-the-art technologies, like dependency injection and declarative styling via CSS files.

Eclipse RCP applications benefit from the existing user interface and the internal framework, and can reuse existing plug-ins and features.

1.2. History of Eclipse RCP

Eclipse was originally started as a modular IDE application.

In 2004 Eclipse version 3.0 was released. The Eclipse 3.0 release supported reusing of the Eclipse platform to build stand-alone applications based on the same technology as the Eclipse IDE.

At this point the term *Eclipse RCP* was coined. Eclipse RCP is short for *Eclipse Rich Client Platform* and indicates that the Eclipse platform is used as a basis to create feature-rich stand-alone applications.

The release of Eclipse in version 4.x simplified and unified the Eclipse programming model which is now based on state of the art technologies, like a logical model for the application date, dependency injection as primary programming model and declarative styling via CSS files. The Eclipse 4 release also supports that the user interface technology is replaced, for example you can exchange the SWT toolkit with JavaFX or GWT.

1.3. Why are customers selecting Eclipse RCP ?

Customers use Eclipse RCP because of:

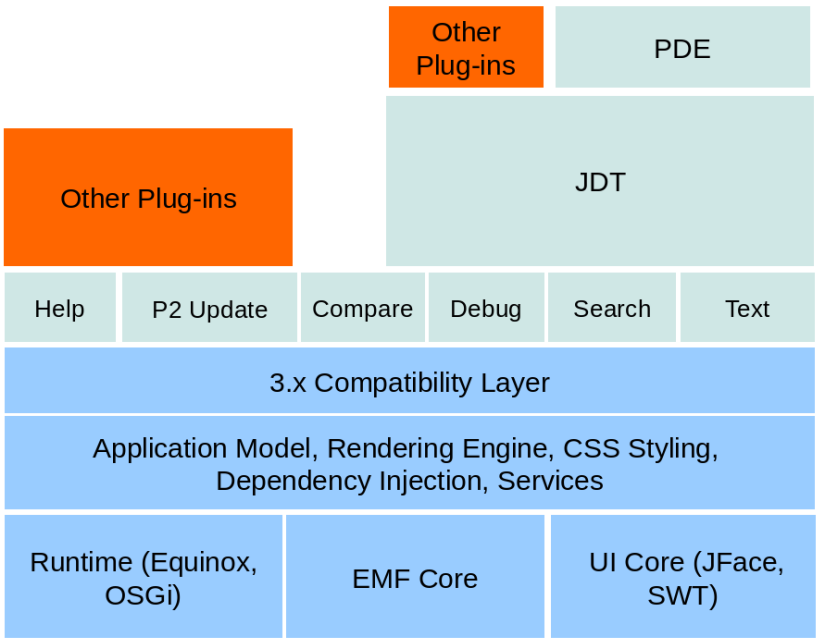
- The Eclipse platform forms the basis of the most successful Java IDE and therefore is very stable and broadly used.
- By default, it provides native user interface components which are fast and reliable.
- It has a strong modular approach it allows developers to design component based systems.
- Companies such as IBM, SAP and Google use the Eclipse framework as a basis for their products and therefore ensure that Eclipse is flexible, fast and continues to evolve.
- The Eclipse platform also fosters a large community of individuals which provide support, information and extensions to the Eclipse framework.

Tapping into this ecosystem allows you to find required resources and information.

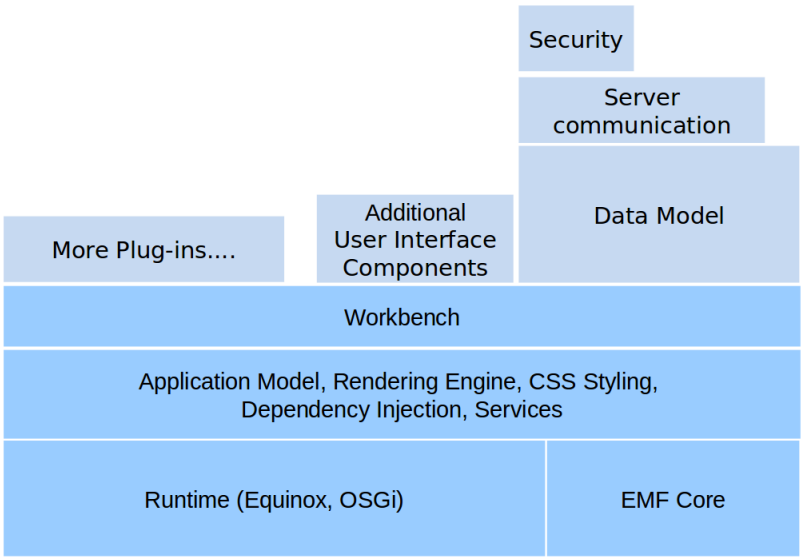
2. Architecture of Eclipse based applications

An Eclipse RCP application consists of individual software components. Eclipse applications are based on some basic components, other components use them and provided extended functionality on top of them.

The Eclipse IDE can be viewed as a special Eclipse application with the focus on supporting software development. For example, the Java development tools (JDT) provide the functionality to develop Java applications.



An Eclipse RCP application typically uses the same base components as the Eclipse IDE. On top of these, they add application specific components as depicted in the following graphic.



3. Core components of the Eclipse platform

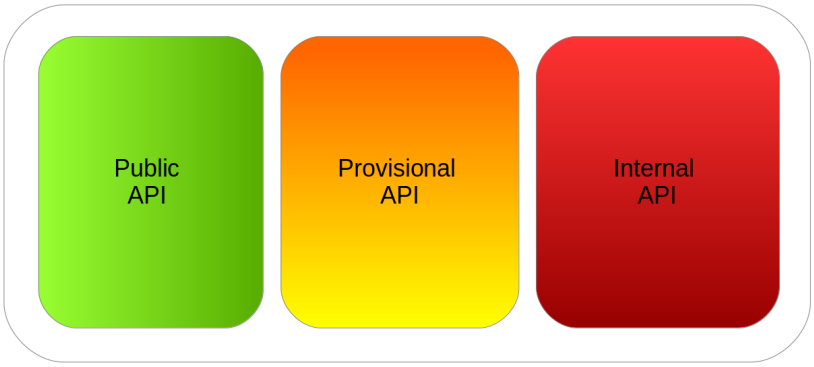
OSGi is a specification to describe a modular approach to develop component-based Java applications. The programming model of OSGi allows you to define dynamic software components, i.e., OSGi services. *Equinox* is one implementation of the OSGi specification and is used by the Eclipse platform as its runtime. This Equinox runtime provides the necessary API's and framework to run a modular Eclipse application.

SWT is the standard user interface component library used by Eclipse. *JFace* provides some convenient APIs on top of SWT. The *workbench* provides the framework for the application. It is responsible for displaying all other user interface components.

EMF is the Eclipse Modeling Framework which provides functionality to model a data model and to use this data model at runtime.

4. Eclipse API and internal API

An OSGi runtime allows the developer to mark Java packages as public, provisional or internal APIs. The internal API is private, therefore not visible. The provisional API are to test non-finalized APIs, therefore are visible but non-stable. The public API, or simply API, are the visible and stable API, that can be reused by other components.

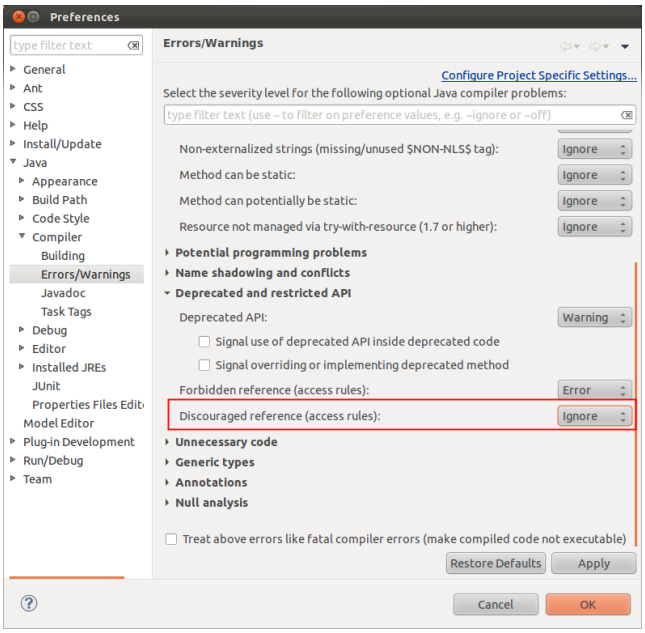


The Eclipse platform project marks packages either as public API or as provisional API, to make all Java classes accessible to Eclipse developers. If the Eclipse platform project releases an API, the platform project plans to keep this API stable for as long as possible.

If API is internal but accessible, i.e., marked as provisional, the platform team can change this API in the future. If you use such API, you must be prepared that you might have to make some adjustments to your application in a future Eclipse release.

If you use unreleased API, you see a *Discouraged access: The …is not API (restriction on required project …)* warning in the Java editor.

You can turn off these warnings for your workspace via Window ▸ Preferences ▸ Java ▸ Compiler ▸ Errors/Warnings and by setting the *Discouraged reference (access rules)* flag to *Ignore*.



Alternatively you can turn off these warnings on a per project basis, via right-click on the project Properties ▸ Java Compiler and afterwards use the same path as for accessing the global settings. You might have to activate the *Enable project specific settings* checkbox at the top of the Error/Warnings preference page.

5. Important configuration files for Eclipse plug-ins

An Eclipse plug-in has the following main configuration files. These files are defining the API, and the dependencies of the plug-in.

- *MANIFEST.MF* - contains the OSGi configuration information.
- *plugin.xml* - optional configuration file, contains information about Eclipse specific extension mechanisms

its exported API and its dependencies via the *MANIFEST.MF* file.

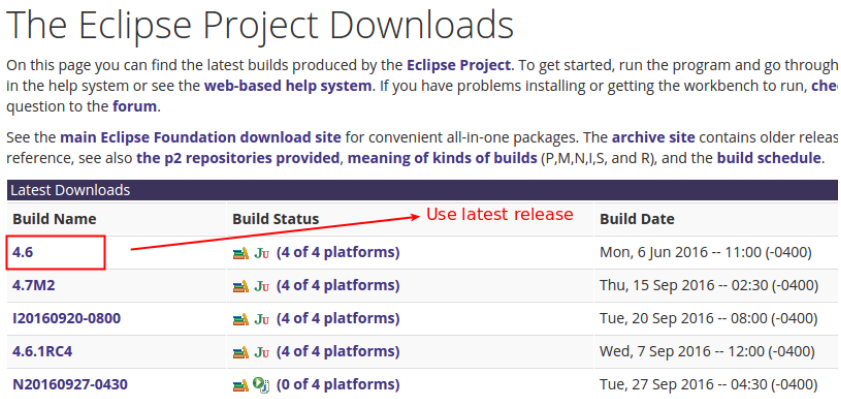
The *plugin.xml* file provides the possibility to create and contribute to Eclipse specific API. You can add *extension points* and *extensions* in this file. *Extension-points* define interfaces for other plug-ins to contribute functionality. *Extensions* contribute functionality to these interfaces. Functionality can be code and non-code based. For example, a plug-in might contain help content.

6. Download the Eclipse Software Development Kit (SDK)

The following description is based on the latest Eclipse 4.6 (Neon) release. Download the latest version of the *Eclipse SDK* build from the following URL:

```
http://download.eclipse.org/eclipse/downloads/
```

This website should look similar to the following screenshot. Click on the link of the latest released version (the release version with the highest number) to get to the download section.

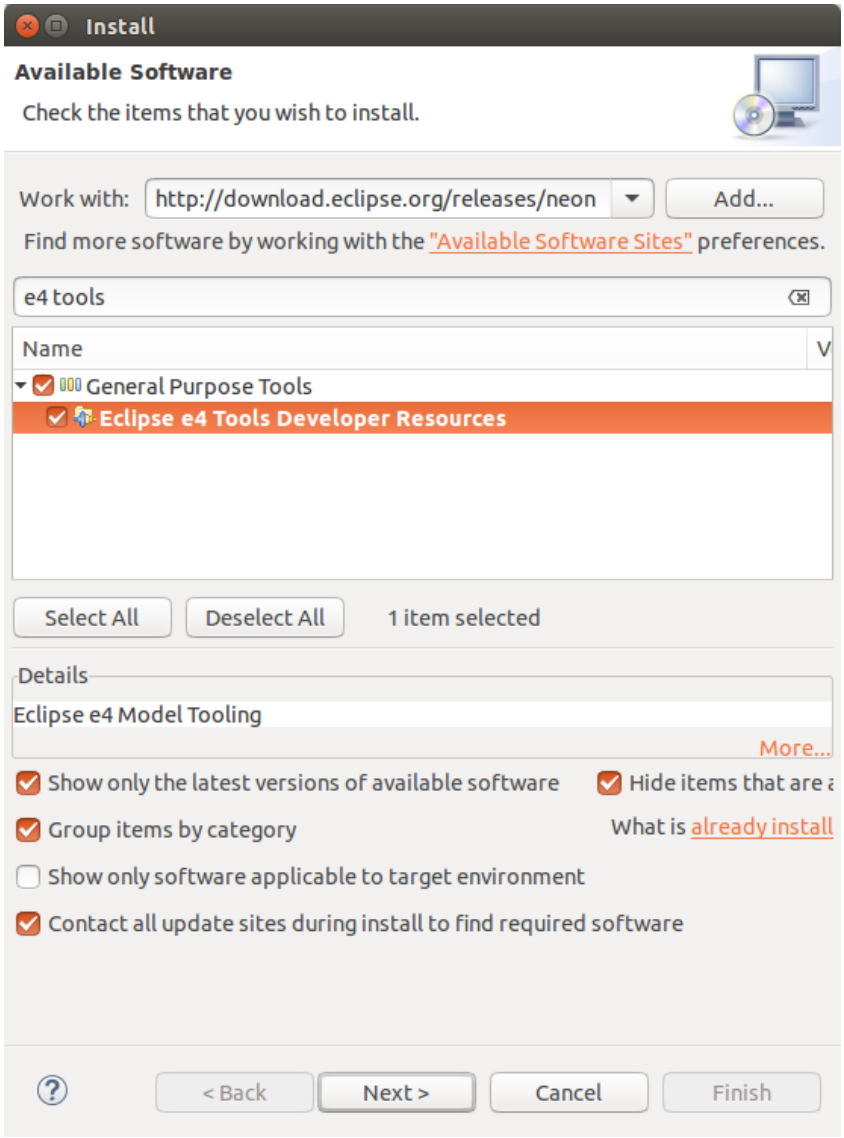


The download is a compressed archive of multiple files. This format depends on you platform, Windows the *zip* format, while Linux and Mac OS uses the *tar.gz* format.

7. Install the e4 tools

7.1. Using the update manager

The e4 tools provide the tools to develop Eclipse 4 RCP applications. These tools provide wizards to create Eclipse application artifacts and an application model editor. They can be installed into the Eclipse SDK via the Eclipse default update site.



Install the tools and restart your Eclipse IDE after the installation.

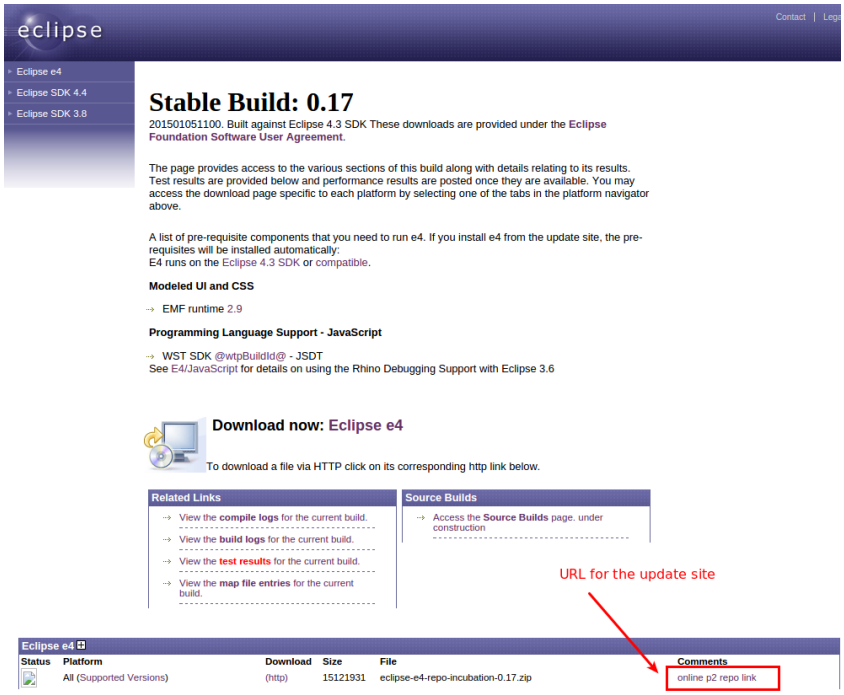
7.2. Install the e4 spies from Eclipse.org

To install the tools required for Eclipse RCP development open the Eclipse update manager via Open Help ▸ Install new software…. You can use the following URL to install the latest tools:

http://download.eclipse.org/e4/snapshots/org.eclipse.e4.tools/latest/

Other versions of the tools can can be found on the Eclipse.org

following screenshots demonstrate this for a particular build of the e4 tools. Note, that this website might change over time.

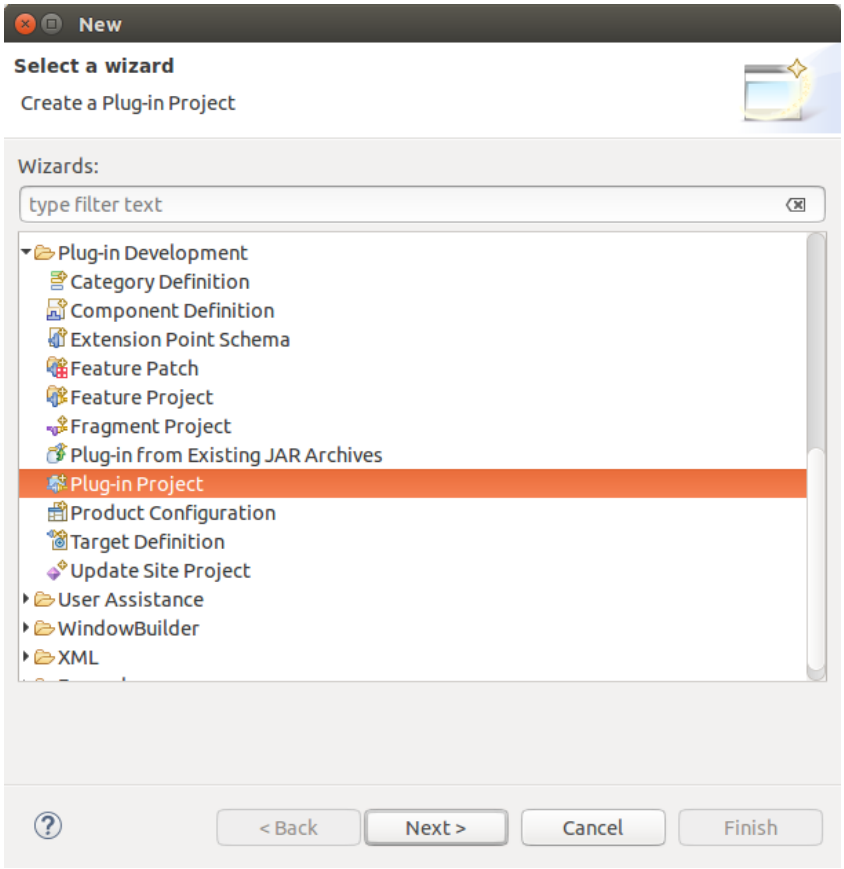


8. Exercise: Create an RCP application with the wizard

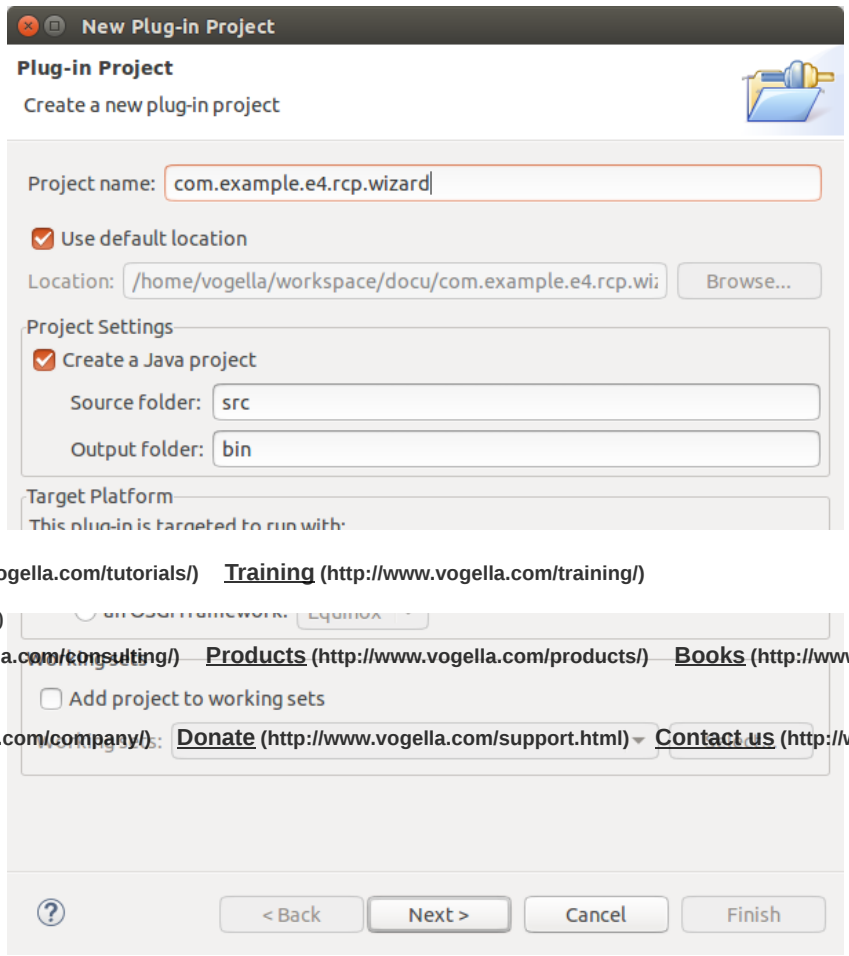
The following exercise demonstrates how to create an Eclipse RCP application based on a template. It also shows how to start the application via the Eclipse IDE. You learn all the details of what happened here in later chapters.

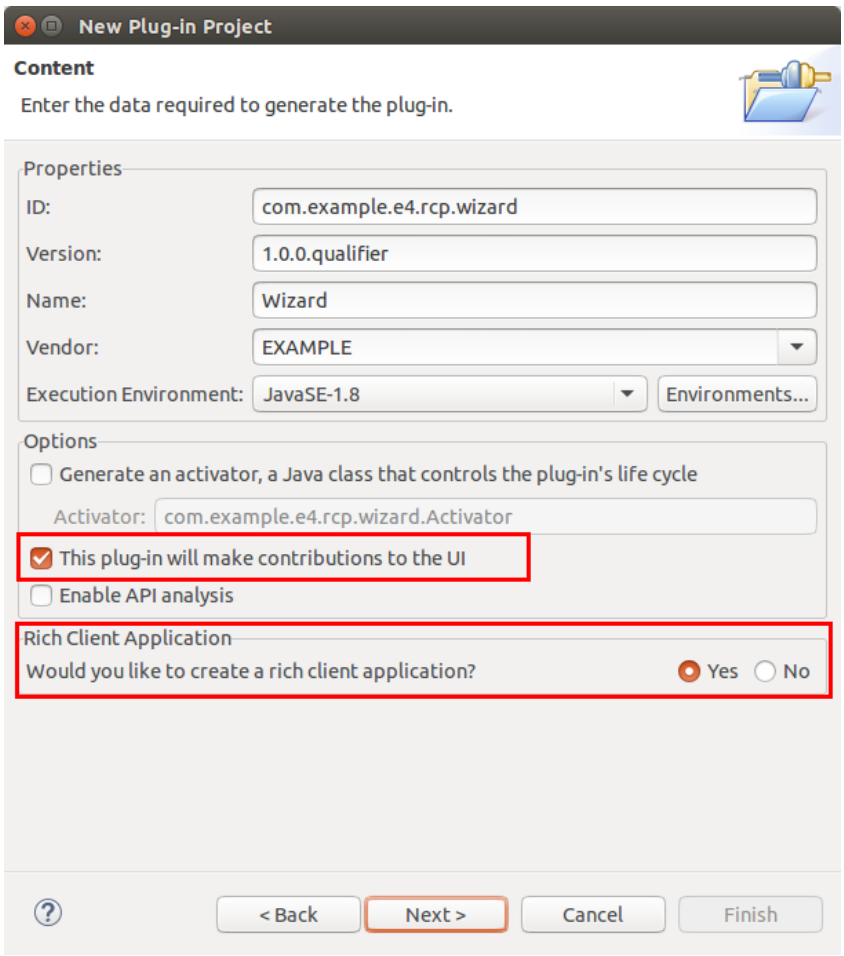
8.1. Create project

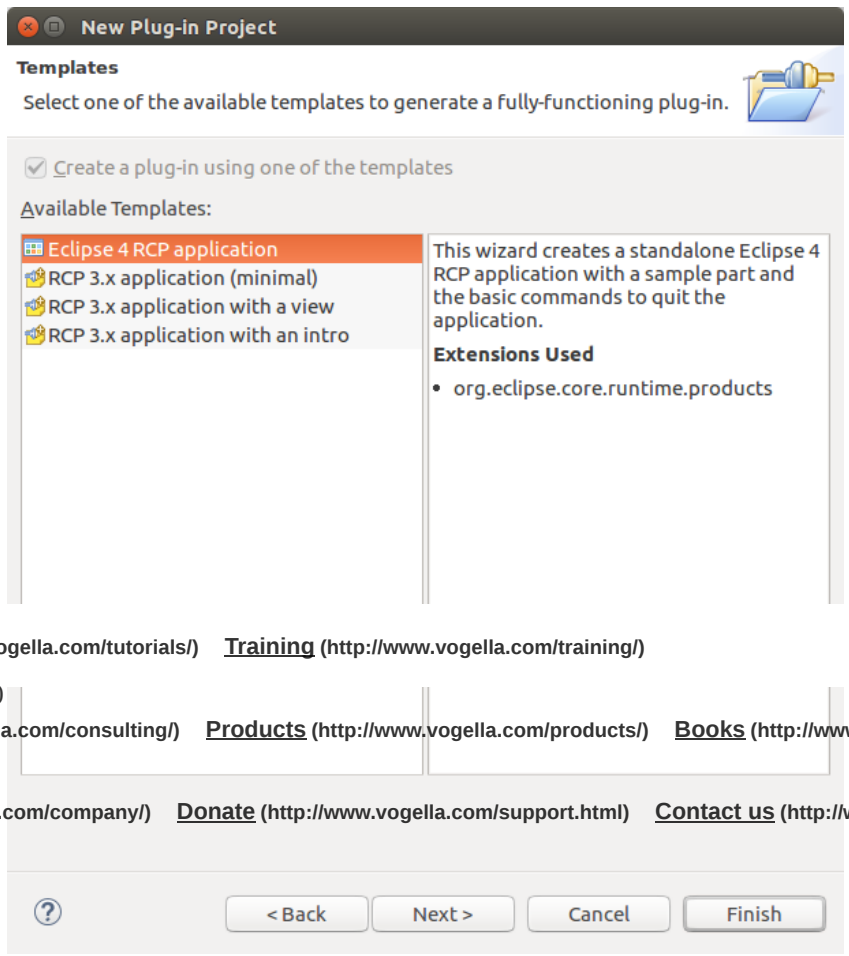
Select File ▸ New ▸ Other… ▸ Plug-in Development ▸ Plug-in Project from the menu of your Eclipse IDE.



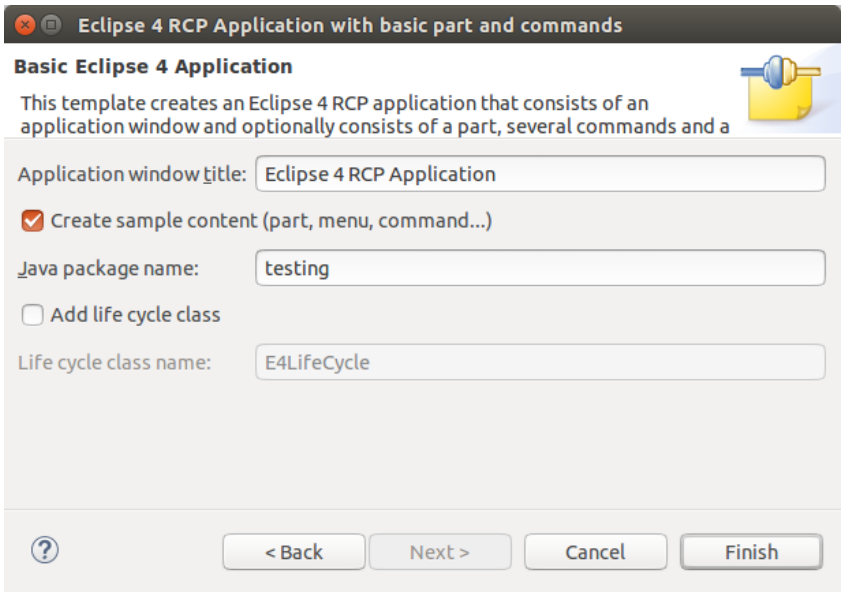
Create a project called `com.example.e4.rcp.wizard`. Use the settings similar to the following screenshots, most of them are the default settings.







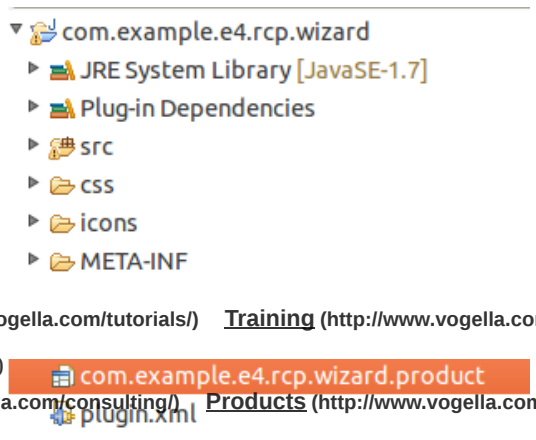
On the last wizard page, select the *Create sample content (parts, menu etc.)* flag. Via this flag you configure that the generated application has example content, e.g., a view and some menu and toolbar entries.



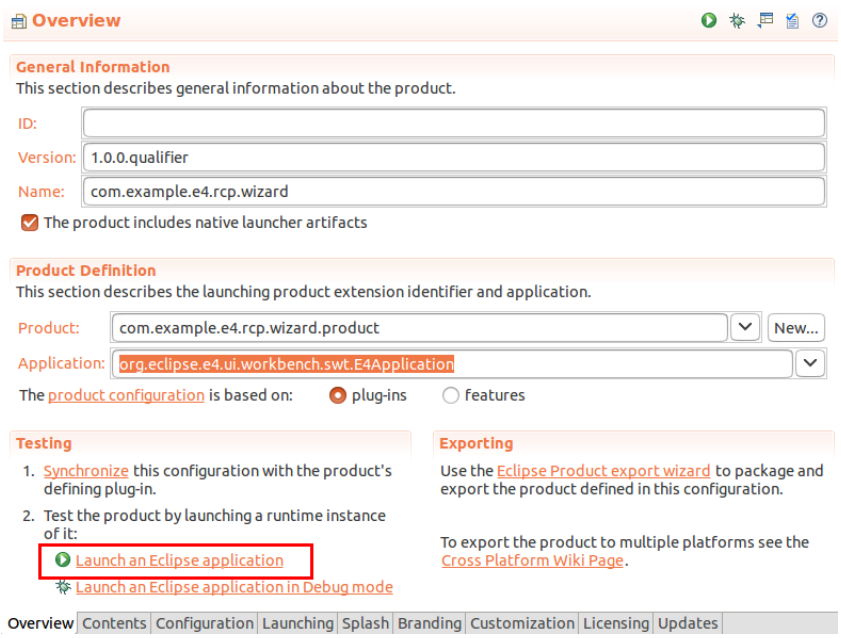
The wizard adds also the *clearPersistedState* flag to the product configuration file. This ensures that changes during development in your application model are always visible. See Ensure to delete the persisted user interface state at startup for more information.

8.2. Launch your Eclipse application via the product file

Open the generated product file by double-clicking on the file.

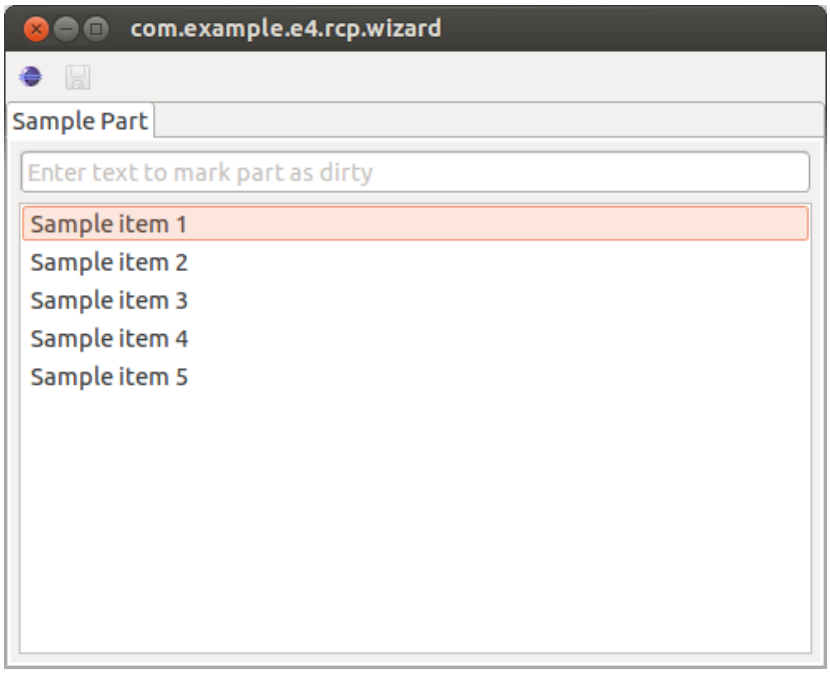


Switch to the *Overview* tab in the editor and launch your Eclipse application by pressing the *Launch an Eclipse application* hyperlink. This selection is highlighted in the following screenshot.



8.3. Validating

As a result your Eclipse application should start. The application should look similar to the following screenshot.



9. Eclipse 4 application model

9.1. What is the application model?

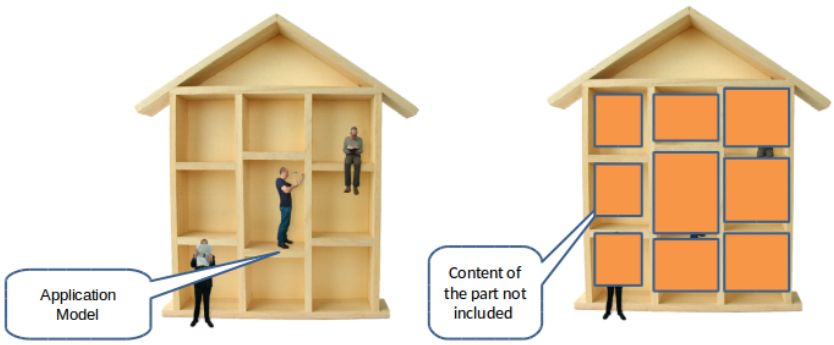
The Eclipse platform uses an abstract description, called the *application model*, to describe the structure of an application. This application model contains the visual elements as well as some non-visual elements of the application.

The visual parts are, for example, windows, parts (views and editors), menus, toolbars, etc. Examples for non-visual components are handlers, commands and key bindings.

Each model element has attributes which describe its current state, e.g. the size and the position of a window. The application model also expresses the relationship of the model elements via a hierarchy.

The individual user interface widgets, which are displayed in a part, are not defined via the application model, e.g., the content of the part is still defined by your source code.

If the application model was a house, it would describe the available rooms (parts) and their arrangement (perspectives, part stacks, part sash containers), but not the furniture of the rooms. This is illustrated by the following image.



The base of the application model is typically defined as a static file. By default, it is called `Application.e4xmi` and located in the main directory of the plug-in which defines the product extension. This file is read at application startup and is used to construct the initial application model. Changes made by the user are persisted and re-applied at startup.

The application model is extensible, e.g., other plug-ins can contribute to it via *model processors* and *model fragments*.

9.2. Connecting model elements to classes and

Model elements can point to a class or to a static resource via a Uniform Resource Identifier (URI). For this purpose Eclipse defines two URI patterns. Eclipse instantiates the referred objects or resources in most cases lazily. For example, the classes for a part are instantiated when the part gets visible.

The following table describes the support URI patterns. The example assumes that the bundle is called *test* to have a short name.

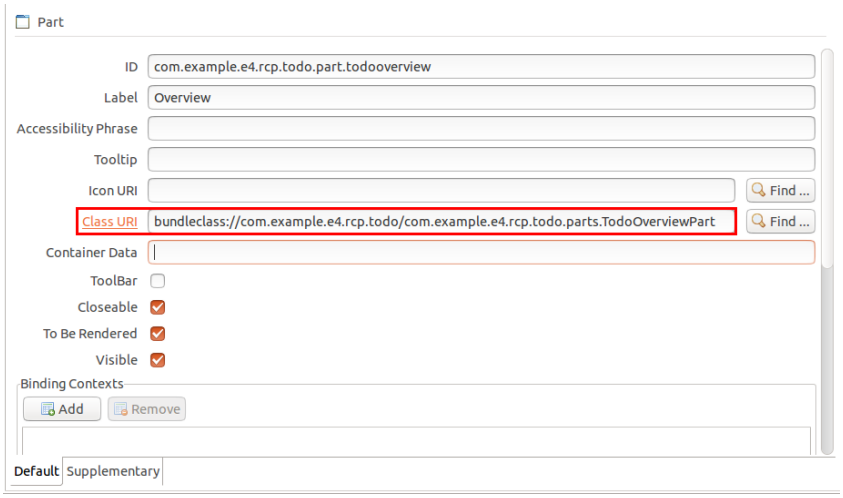
Table 1. URI pattern for class and static file references

Pattern	Description

`bundleclass://Bundle-SymbolicName/package.classname`
 	
Example:
 	
`bundleclass://test/test.parts.MySavePart`
	Identifier for a Java class. It consists of the following parts: `bundleclass://` is a fixed prefix. Bundle-SymbolicName is defined in *MANIFEST.MF* file. The Bundle-SymbolicName is followed by a '/' and the fully qualified classname.

`platform:/plugin/Bundle-SymbolicName/path/filename.extension`
 	
Example:
 	
`platform:/plugin/com.example.plugin/icons/save_edit.gif`
	Used to identify resources. Identifier for a resource in a plug-in. `platform:/plugin/` is a fixed prefix, followed by the Bundle-SymbolicName, followed by the path to the file and the filename.

For example a part, have a `Class URI` attribute which points to a Java class via the `bundleclass://` URI. This class provides the behavior of the part. The corresponding object is created by the Eclipse framework. Using the house/rooms metaphor from earlier, the class is responsible for defining the furnishings, the layout of the room and how the interactive objects behave.



An example for a static resource reference is the the `Icon URI` attribute of a part. This attribute can point to an icon that is used for the part.

9.3. Runtime application model

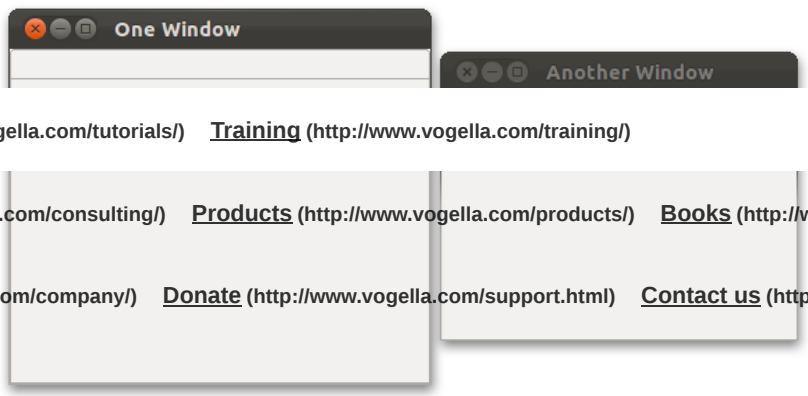
At runtime of the application the created set of model objects is called the *runtime application model*. This runtime application model is dynamic, i.e., you can change the model objects and its attributes and these changes are reflected in your application. The Eclipse platform has change listeners registered on the model objects and updates the user interface whenever you change relevant attributes.

10. User interface model elements

The following model elements represents the basic elements which you use to create the user interface of your application.

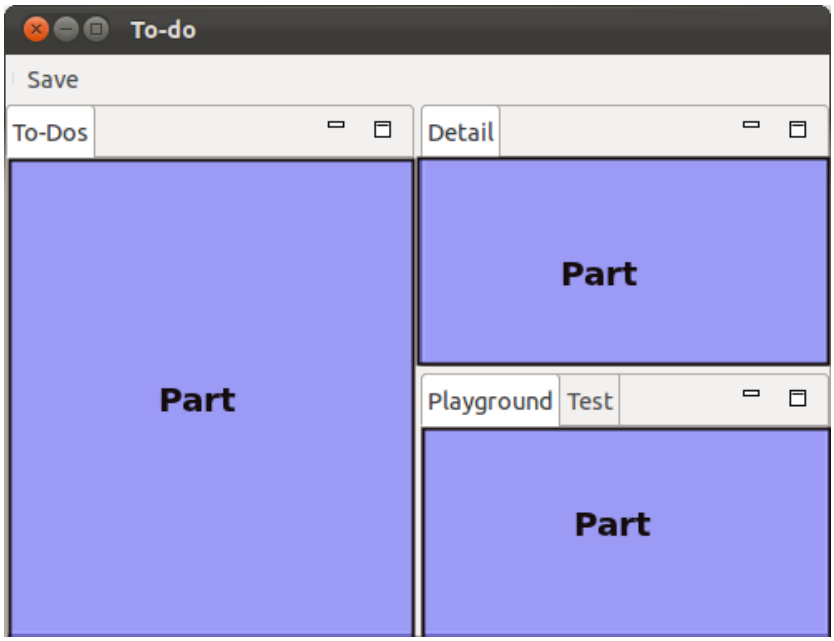
10.1. Window

Eclipse applications consist of one or more windows. Typically, an application has only one window, but you are not limited to that, e.g., if you want to support multiple connected monitors.



10.2. Parts

Parts are user interface components which allow you to navigate and modify data. Parts can be stacked or positioned next to each other depending on the container into which they are dropped. A part can have a drop-down menu, context menus and a toolbar.



Parts can be classified as *views* and *editors*.

A view is typically used to work on a set of data, which might be a hierarchical structure. If data is changed via the view, this change is typically directly applied to the underlying data structure. A view sometimes allows the user to open an editor for the selected set of data. For example, the *Package Explorer* view allows you to browse the files of Eclipse projects. If you for example rename a file via the *Package Explorer* view, the file name is directly changed on the file system.

Editors are typically used to modify a single data element, e.g., the content of a file or a data object. To apply the changes made in an editor to the data structure, the user has to explicitly save the editor content.

For example, the *Java* editor is used to modify Java source files. Changes to the source file are applied once the user selects the

10.3. Available part containers

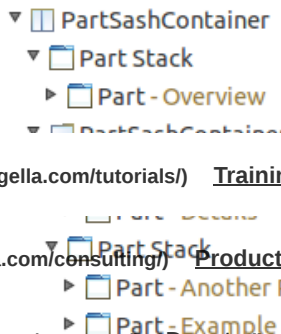
Parts can be directly assigned to a window or a perspective. They can also be grouped and arranged via additional model elements, i.e., via part stack (*Part Stack*) or via part sash container (*Part Sash Container*) elements.

A part stack contains a stack of parts showing the content of one part while displaying only the headers of the other parts. One part is active and the user can switch to another part by selecting the corresponding tab.

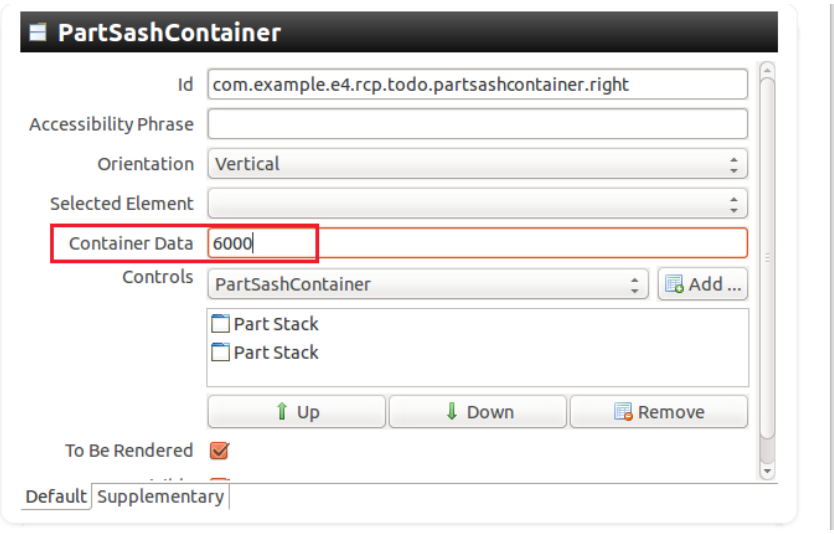
A part sash container displays all its children at the same time either horizontally or vertically aligned.

The following screenshot shows a simple Eclipse application layout using two part sash container and three part stacks.

On the top of this layout there is a horizontal part sash container which contains another part sash container and one part stacks. The part sash container on the next level contains two part stacks. The hierarchy is depicted in the following graphic.



You can use the *Container Data* attribute on a child of a part sash container to assign a layout weight. This layout weight is interpreted as the relative space the corresponding child element should get assigned in the part sash container. The setting is depicted in the following screenshot.



If you set the *Container Data* for one element, you must define it for all the other elements, too. Otherwise the missing values are interpreted as very high and these elements take up all available space.

TIP:The initial total of all the container data values is maintained when elements in the sash are moved. In order to allow fine grained/smooth dragging this total must be similar to the screen resolution. A too low value (i.e., 50 / 50) causes the part to be moved multiple pixels per sash unit, which the user will realize as a jerky movement. Therefore, use a sufficient high value, e.g., 10000.

10.4. Perspective

A perspective is an optional container for a set of parts. Perspectives can be used to store different arrangements of parts. For example, the Eclipse IDE uses them to layout the views appropriate to the task (development, debugging, review, ...) the developer wants to perform.

You can place perspectives in a perspective stack of the application model. Switching perspectives can be done via the part service provided by the Eclipse platform.

11. Overview of available model objects

persisted user changes and model contributions). It stores this information in Java objects during runtime. These objects are called *model objects* and at runtime they represent the attributes from the model elements.

The following table lists the types of the important model objects.

Table 2. Eclipse model elements

Model element	Description

MApplication	Describes the application object. All other model elements are contained in this object.
MAddon	A self-contained component typically without user interface. It can register for events in the application life cycle and handle these events.
MWindow	Represents a window in your application.
MTrimmedWindow	Similar to MWindow but it allows containing toolbars for the windows (via the TrimBars model elements).
MPerspective	Represents a different layout of parts to be shown inside the window. Should be contained in a MPerspectiveStack.
MPart	Represents the model element part, e.g., a view or an editor.
MDirtyable	Property of MPart which can be injected. If set to true, this property informs the Eclipse platform that this Part contains unsaved data (is dirty). In a handler you can query this property to provide a save possibility.
MPartDescriptor	MPartDescriptor is a template for new parts. A new part based on this part descriptor can be created and shown via the Eclipse framework.
Snippets	Snippets can be used to pre-configure model parts which you want to create via your program. You can use the Eclipse framework to clone such a snippet and use the result object to attach it to the application model at runtime.

12. More to learn about Features and Products

The following description uses feature projects and products, please see Feature projects (http://www.vogella.com/tutorials/EclipseFeatureProject/article.html) and Eclipse Products and Deployment (http://www.vogella.com/tutorials/EclipseProductDeployment/article.html) for a description of these topics.

13. Exercise: Creating an Eclipse RCP application

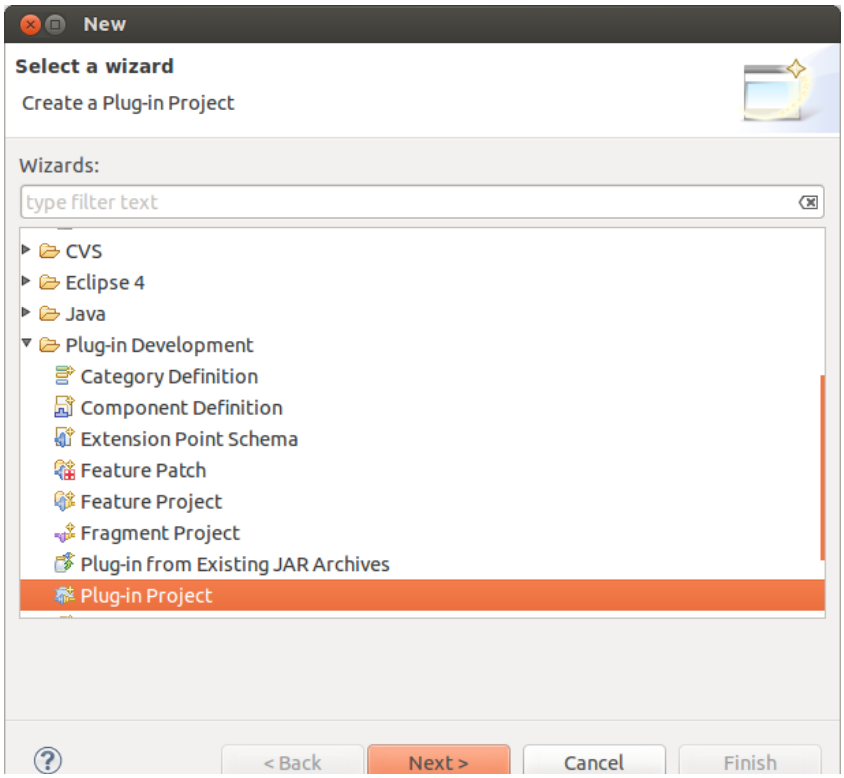
13.1. Create an Eclipse plug-in

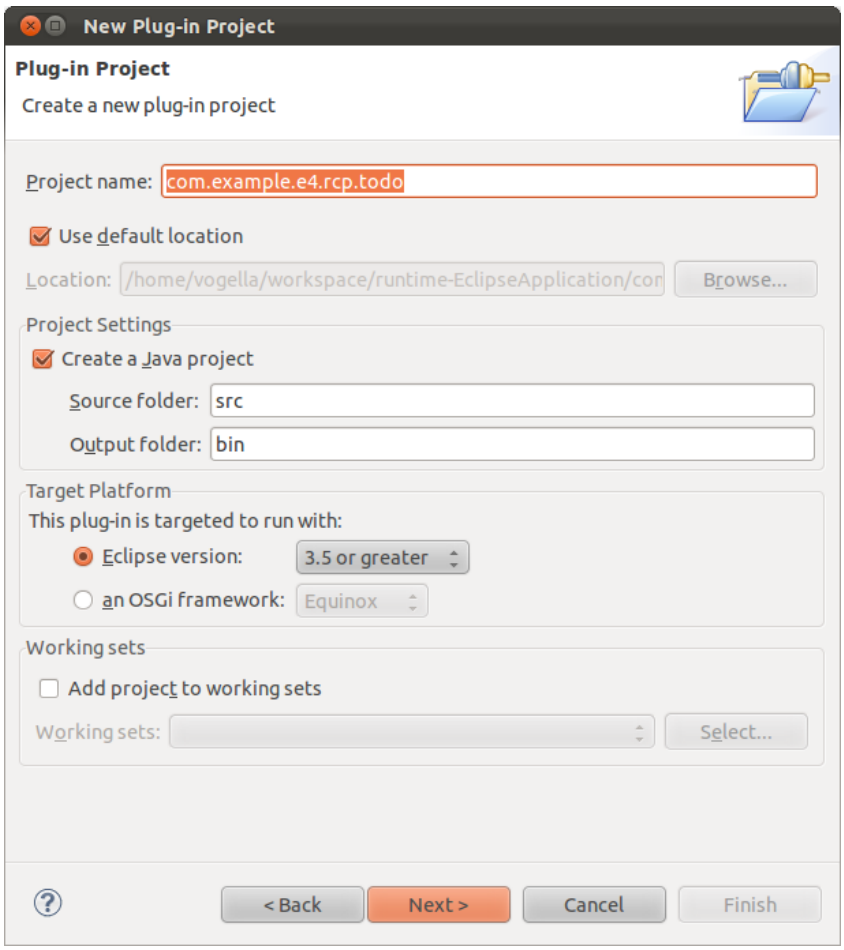
In the following exercise you create a standard Eclipse plug-in. This plug-in is later converted into an Eclipse RCP application.

The following description calls this plug-in the *application plug-in* as this plug-in will contain the main application logic.

13.1.1. Creating a plug-in project

In Eclipse select File ▸ New ▸ Other… ▸ Plug-in Development ▸ Plug-in Project.





On the next wizard page make the following settings. Select *No* at the *Would you like to create a rich client application?* option and uncheck *This plug-in will make contributions to the UI*. Uncheck the *Generate an activator, a Java class that controls the plug-in's life cycle* option.

Press the `Finish` button. If you click the `Next` button instead of `Finish`, the wizard shows you a template selection page which you can bypass without a selection.

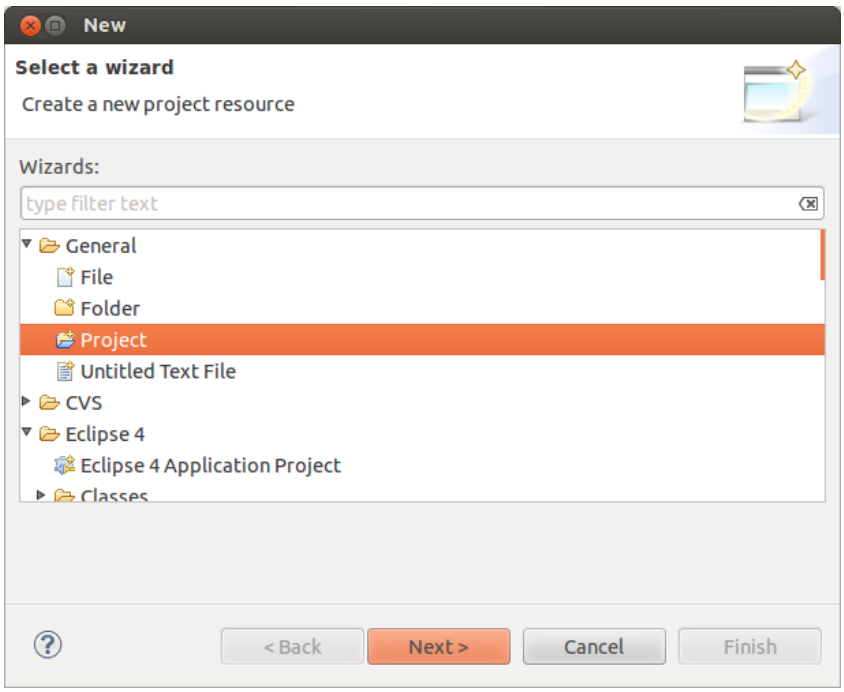
13.1.2. Validate the result

Open the project and ensure that no Java classes were created in the *src* folder.

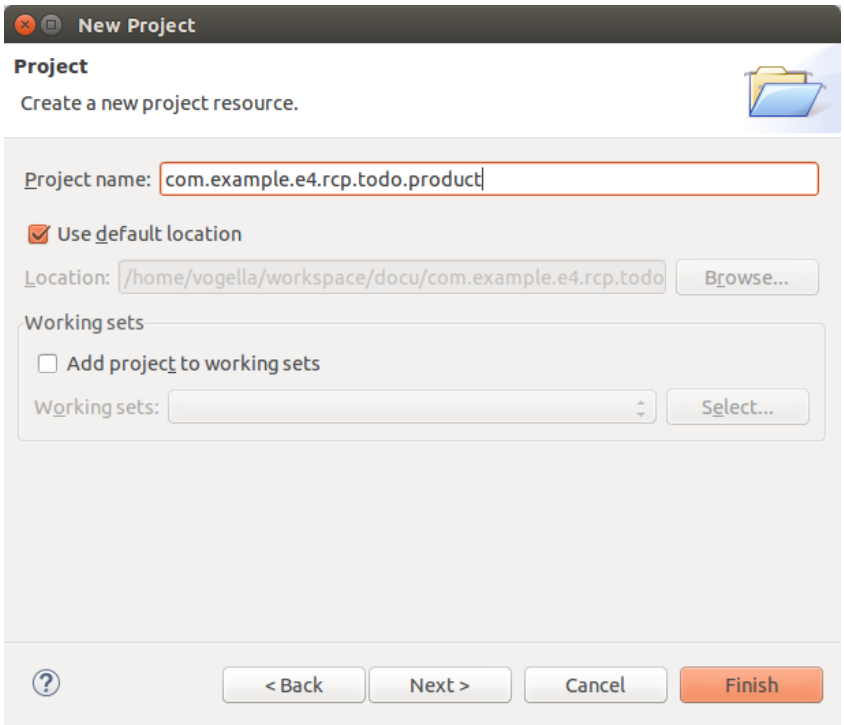
In the manifest editor switch to the *Dependencies* tab and ensure that there are no entries.

13.2. From plug-in to Eclipse 4 application

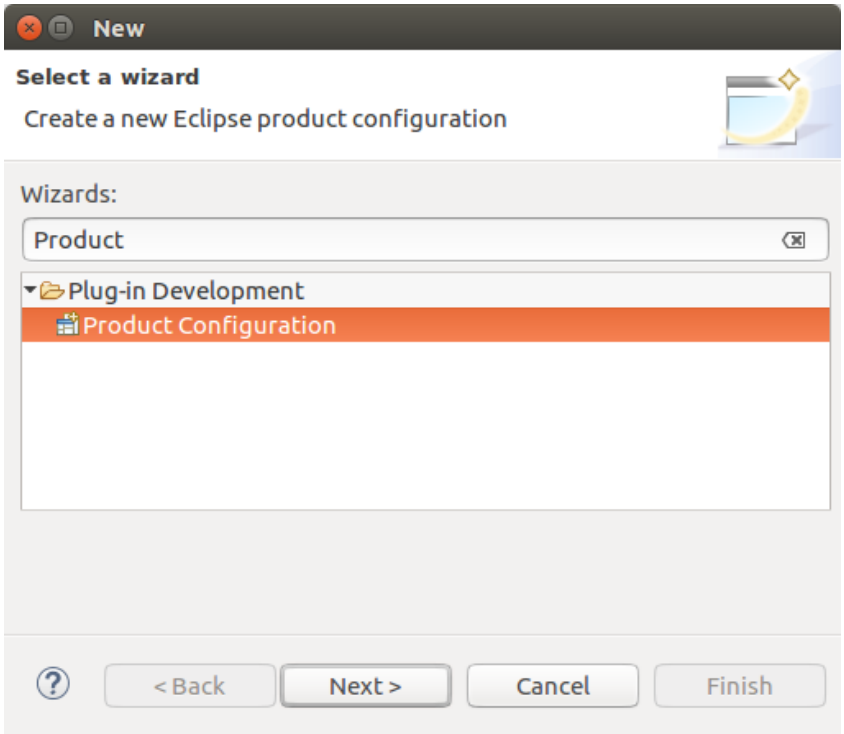
In this chapter you convert the generated plug-in into an Eclipse RCP application.



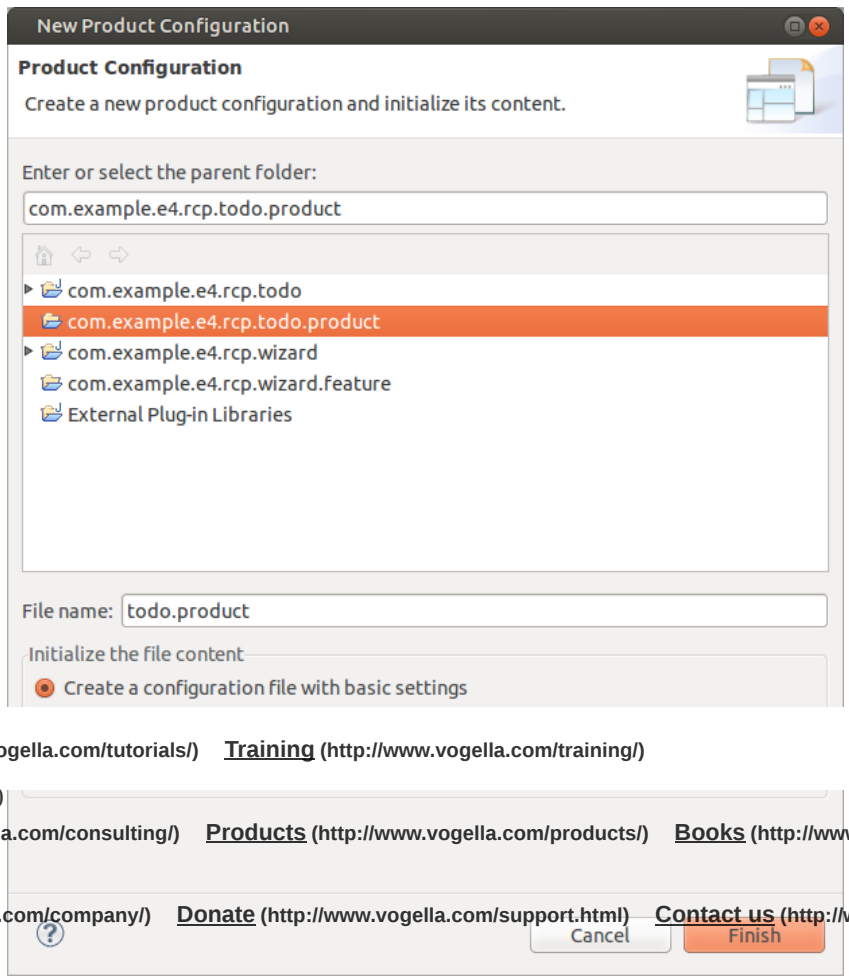
Enter the project name and press `Finish`.



13.2.2. Create a product configuration file

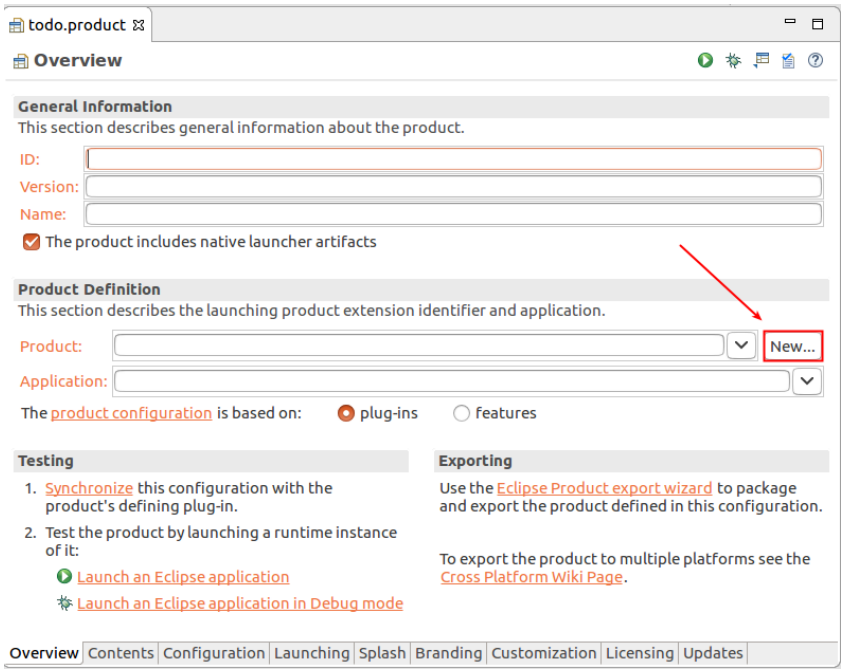


Create a product configuration file called *todo.product* inside the *com.example.e4.rcp.todo.product* folder.

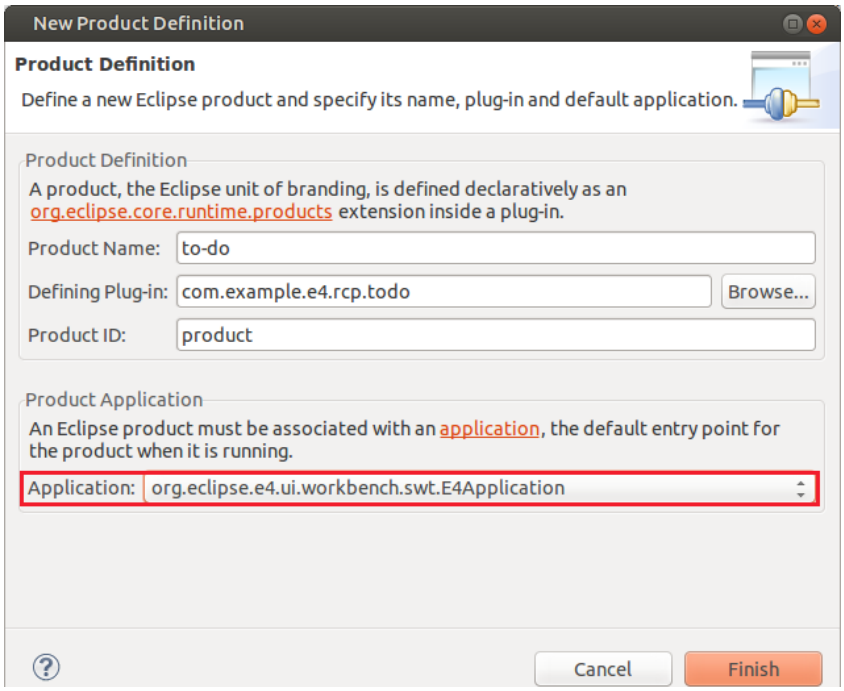


Press the `Finish` button. The file is created and opened in an editor.

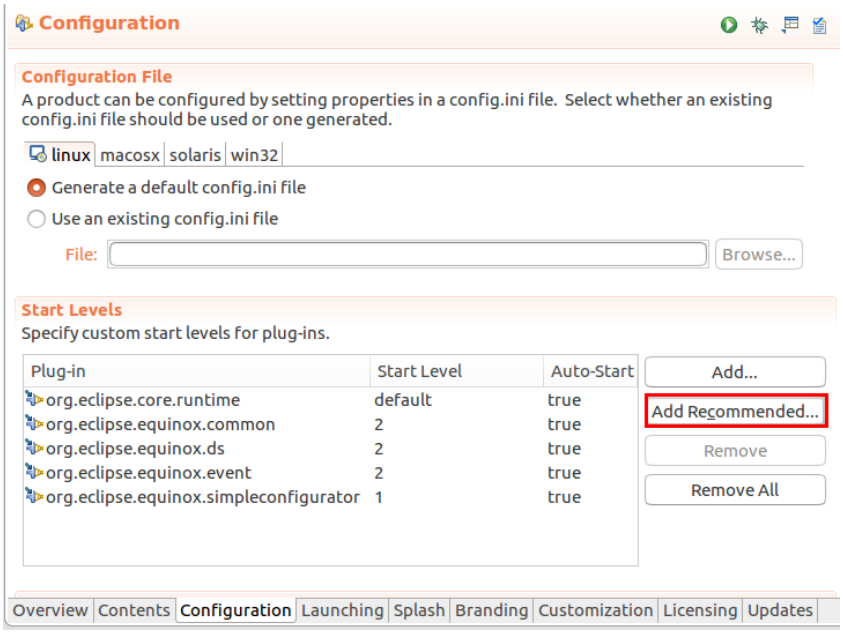
Press the `New...` button on the *Overview* tab of the product editor.



Enter *to-do* as the *Product Name*, your plug-in as the *Defining Plug-in*, *product* as the *Product ID* and select `org.eclipse.e4.ui.workbench.swt.E4Application` in the *Application* combo box.



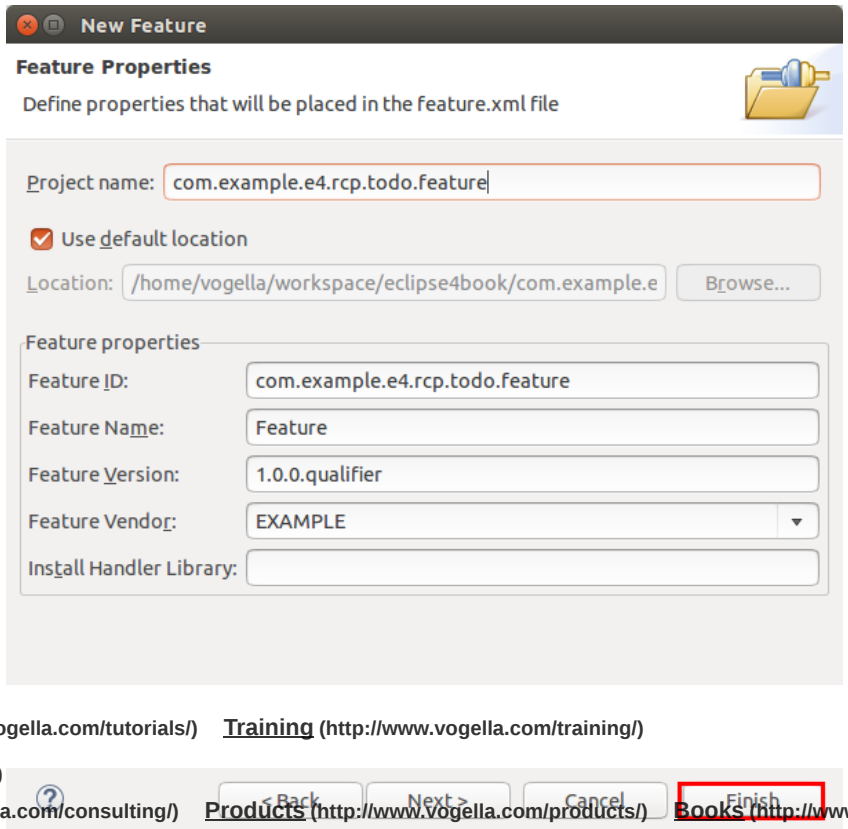
Switch to the *Configuration* tab in the product editor and press the Add Recommended... button. These settings are for example used by the Maven Tycho build system. It is good practice to configure them.



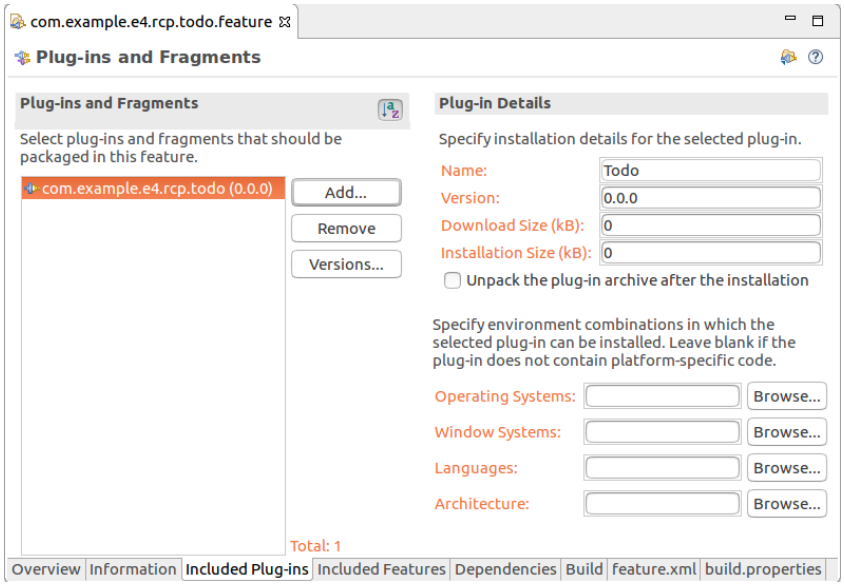
13.2.4. Create a feature project

Create a new feature project called *com.example.e4.rcp.todo.feature* via File ▸ New ▸ Other... ▸ Plug-in Development ▸ Feature Project.

You can press the `Finish` button on the first wizard page.



Afterwards select the *Included Plug-ins* tab in the editor of the *feature.xml* file. Press the `Add...` button and include the `com.example.e4.rcp.todo` plug-in into this feature.

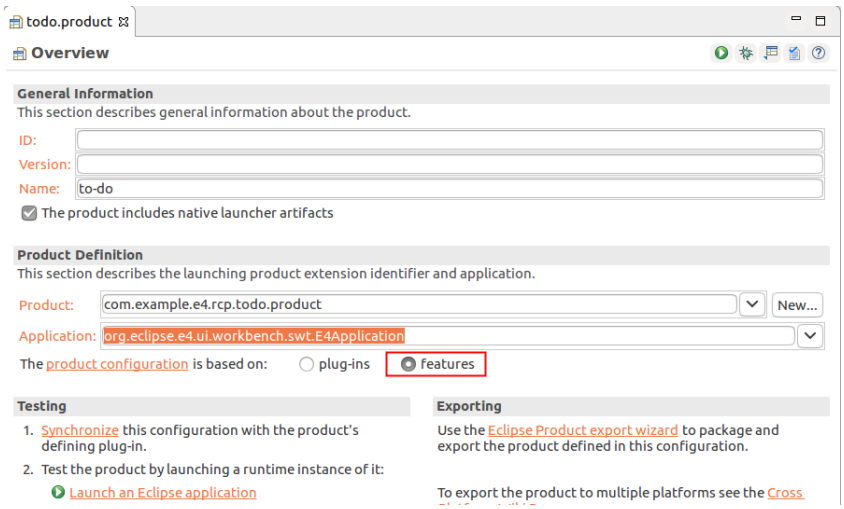


Ensure you have added the plug-in on the *Included Plug-ins* tab to include it into your feature. Using the *Dependencies* tab is wrong for this exercise.

13.2.5. Enter the feature as content into the product

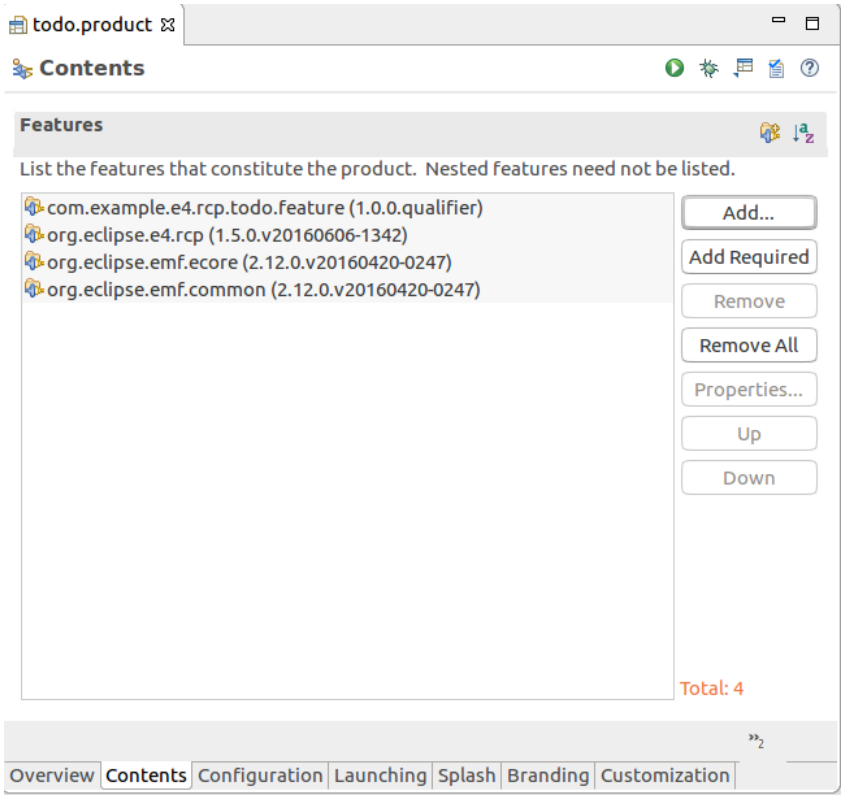
Open your *todo.product* product file and change your product configuration file to use features.

For this select the *features* option on the *Overview* tab of the product editor.



Select the *Contents* tab and add the following features via the Add... button.

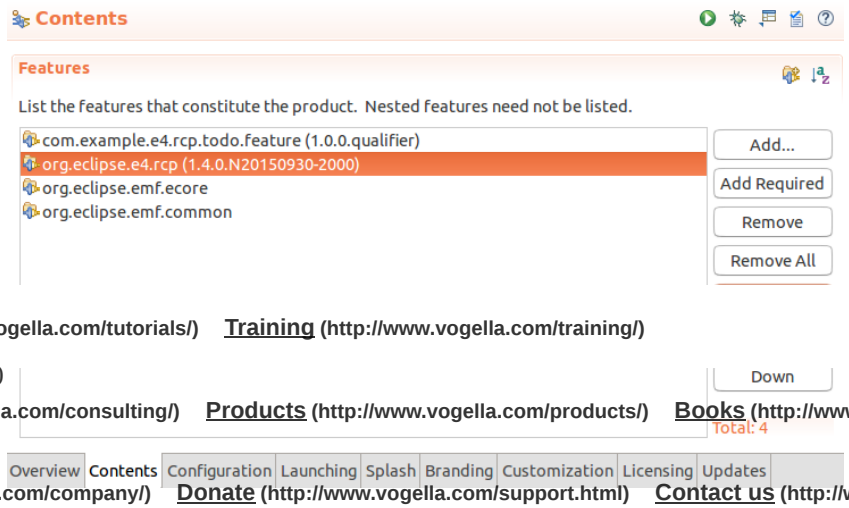
- `com.example.e4.rcp.todo.feature`
- `org.eclipse.e4.rcp`
- `org.eclipse.emf.ecore`
- `org.eclipse.emf.common`

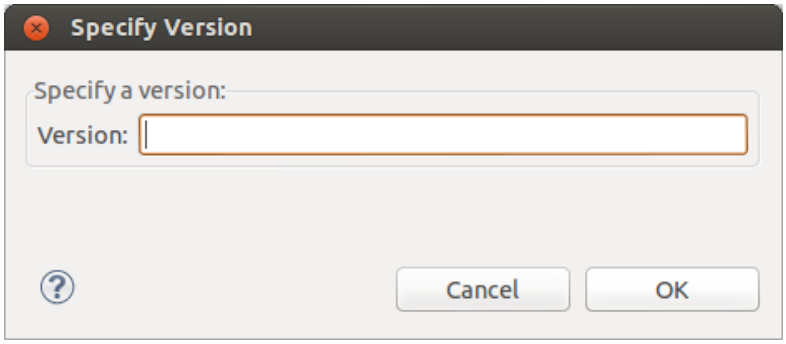


If you cannot add one of the listed features to your product, ensure that you have changed your product to be based on features.

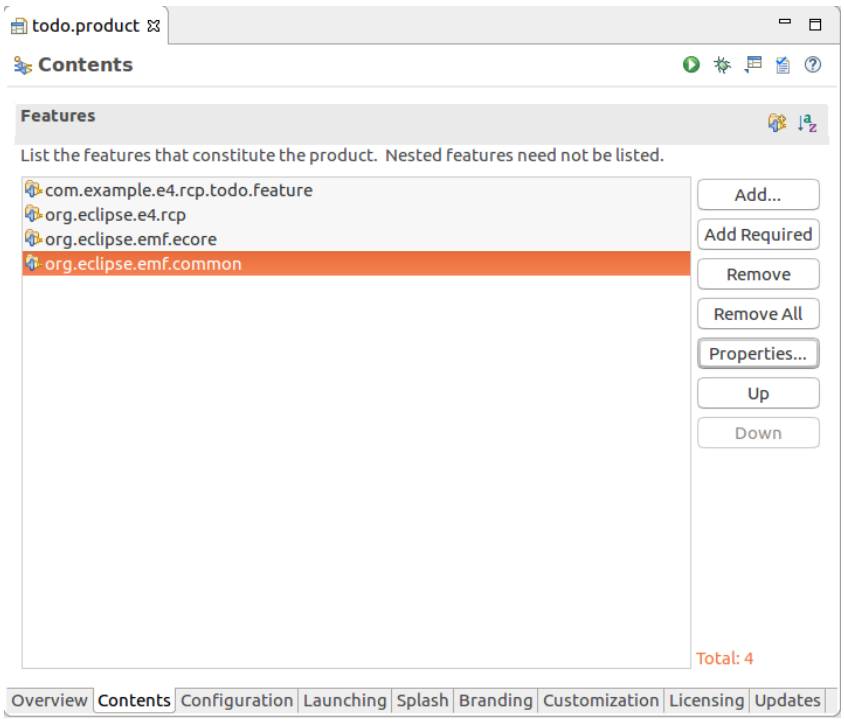
13.2.6. Remove the version dependency from the features in the product

To avoid problems with different versions of your features, delete the version number from your product. You can do this via the `Properties...` button on the *Contents* tab of the product configuration editor.



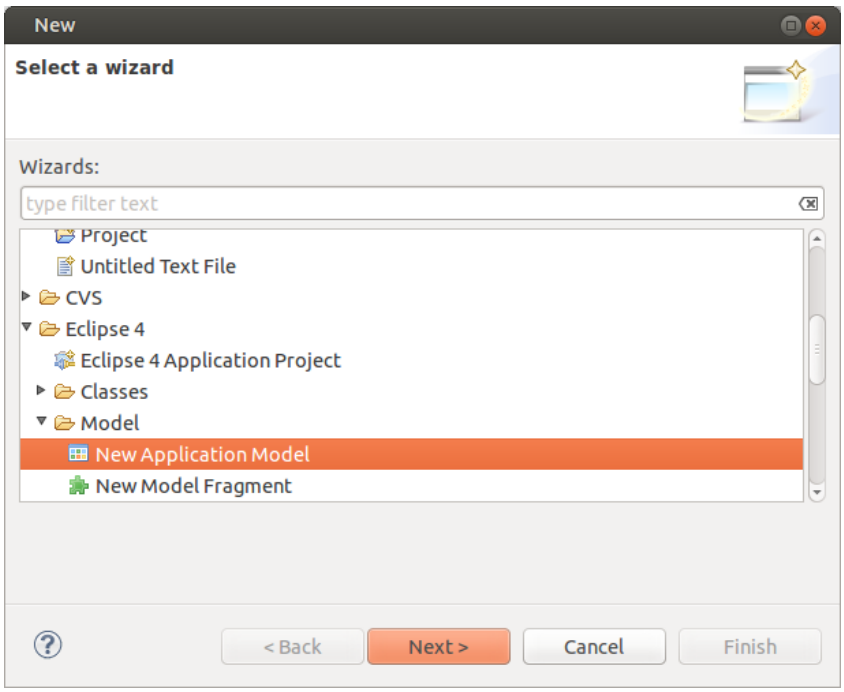


Remove the version dependency from all features. The result should look similar to the following screenshot.

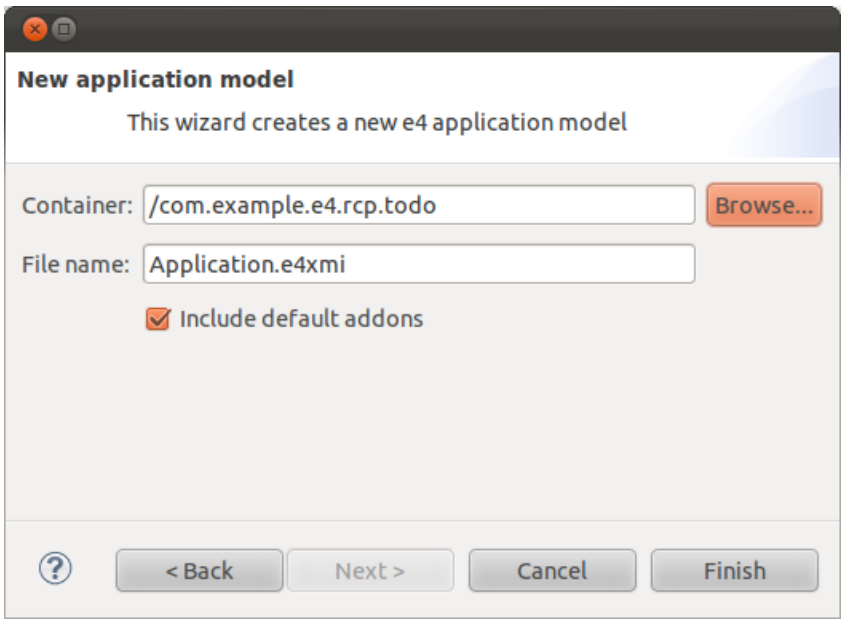


13.2.7. Create an application model

Now create an application model file in your `com.example.e4.rcp.todo` plug-in. Select File ▸ New ▸ Other... ▸ Eclipse 4 ▸ Model ▸ New Application Model to open a wizard.



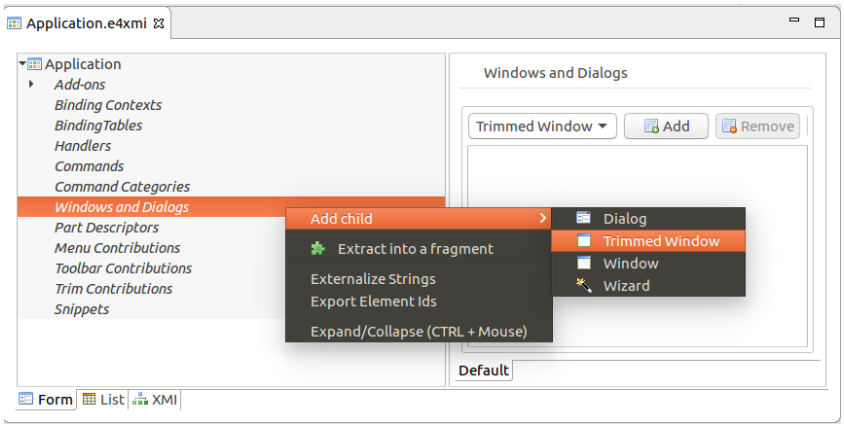
Enter your `com.example.e4.rcp.todo` application plug-in as the container and use the file name suggested by the wizard.



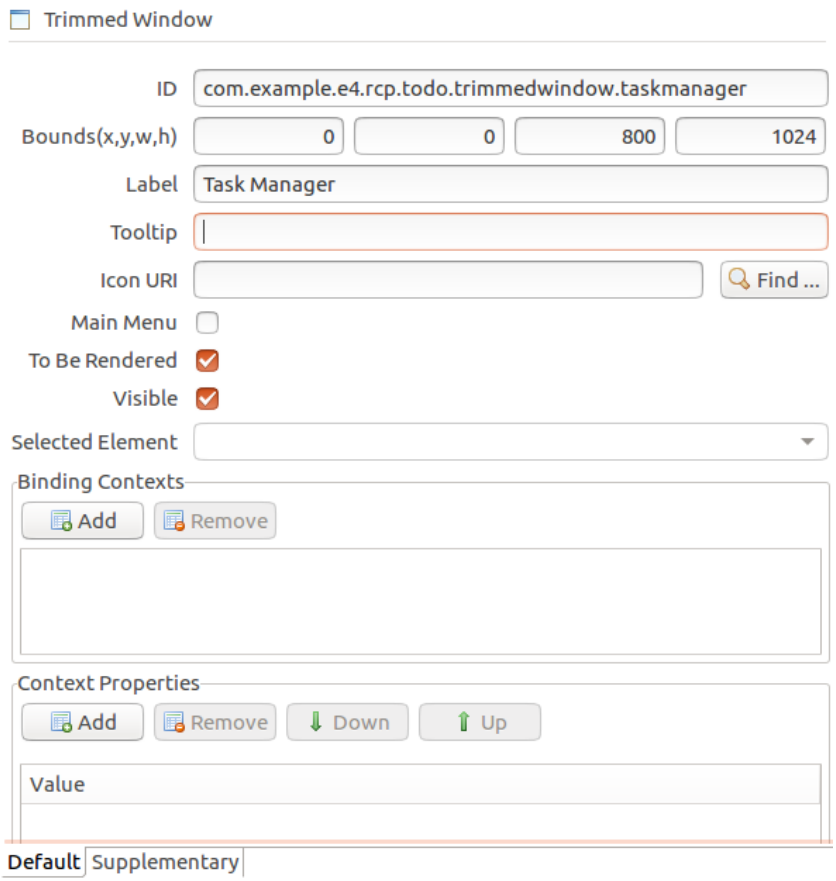
Press the `Finish` button. This triggers the creation of the *Application.e4xmi* file inside the `com.example.e4.rcp.todo` plug-in and opens it.

Add one window to your application model so your application has a visual component.

Right-click on the *Windows and Dialogs* node, and select `Trimmed Window` as depicted in the following screenshot.



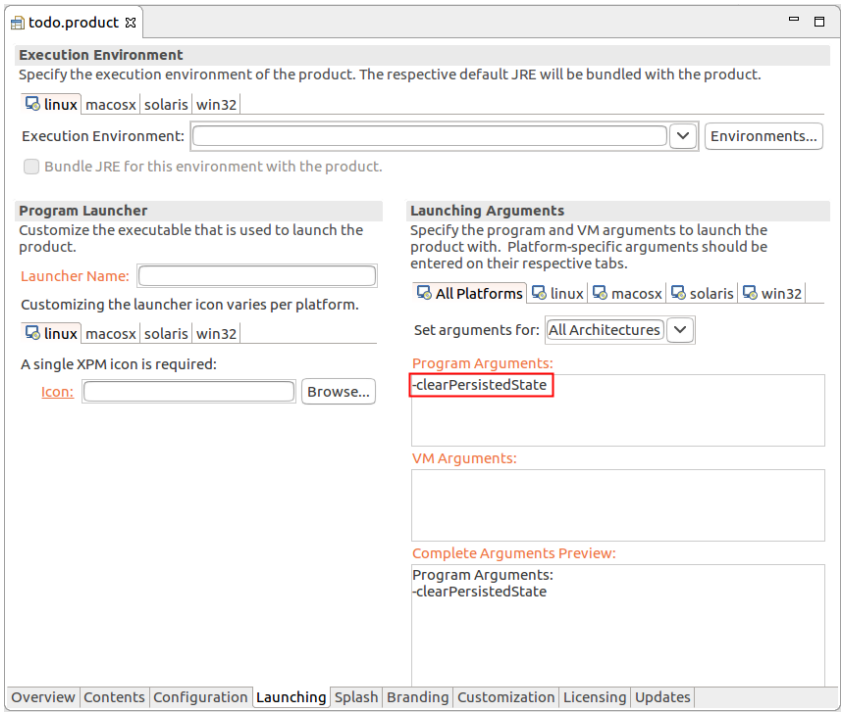
Enter an ID with the `taskmanager` suffix, the position and size of the window and a label as shown in the screenshot below.



13.2.9. Ensure to delete the persisted user interface state at startup

time you start this application. This is undesired during development, as the latest state from the application model file should be used. To ensure that always the latest version of your application model is used, add the *-clearPersistedState* parameter to your product configuration file.

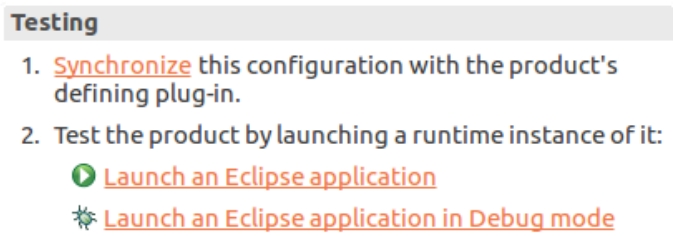
The following screenshot shows this setting in the product configuration file.



Ensure that you used the - sign and that the parameter is typed correctly. Is must be: *-clearPersistedState*

13.2.10. Start the application

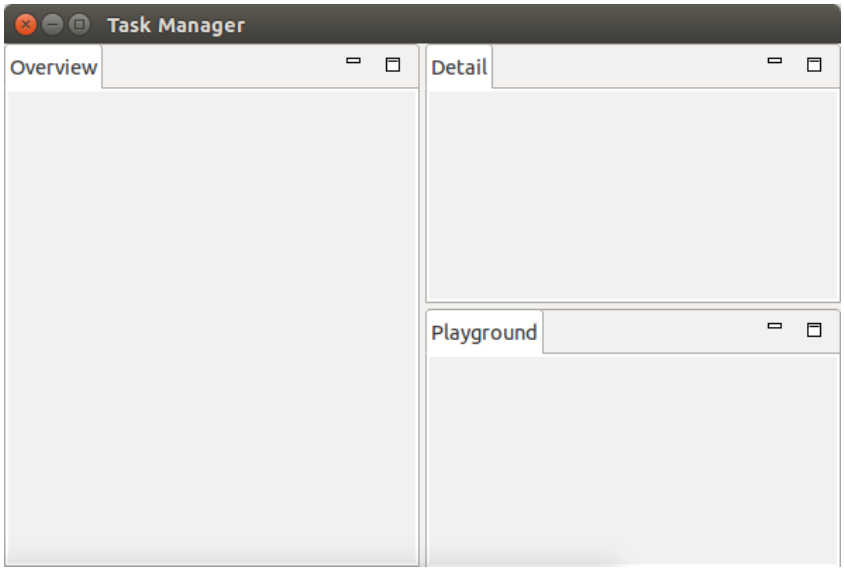
Open the product file and select the *Overview* tab. Press the *Launch an Eclipse application* hyperlink in the *Testing* section.



Validate that your application starts. You should see an empty window, which can be moved, resized, minimized, maximized and closed.

13.3. Modeling a User Interface

We will now model the application user interface. At the end of this exercise, your user interface should look similar to the following screenshot.

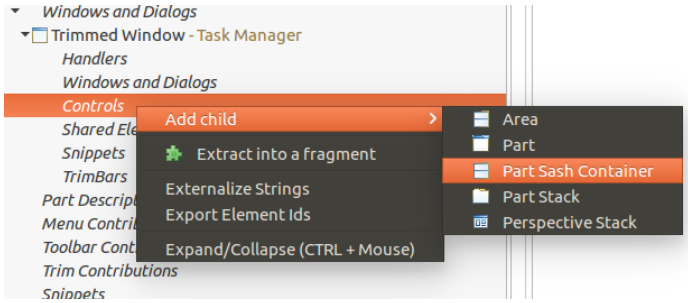


13.3.1. Open the Application.e4xmi file

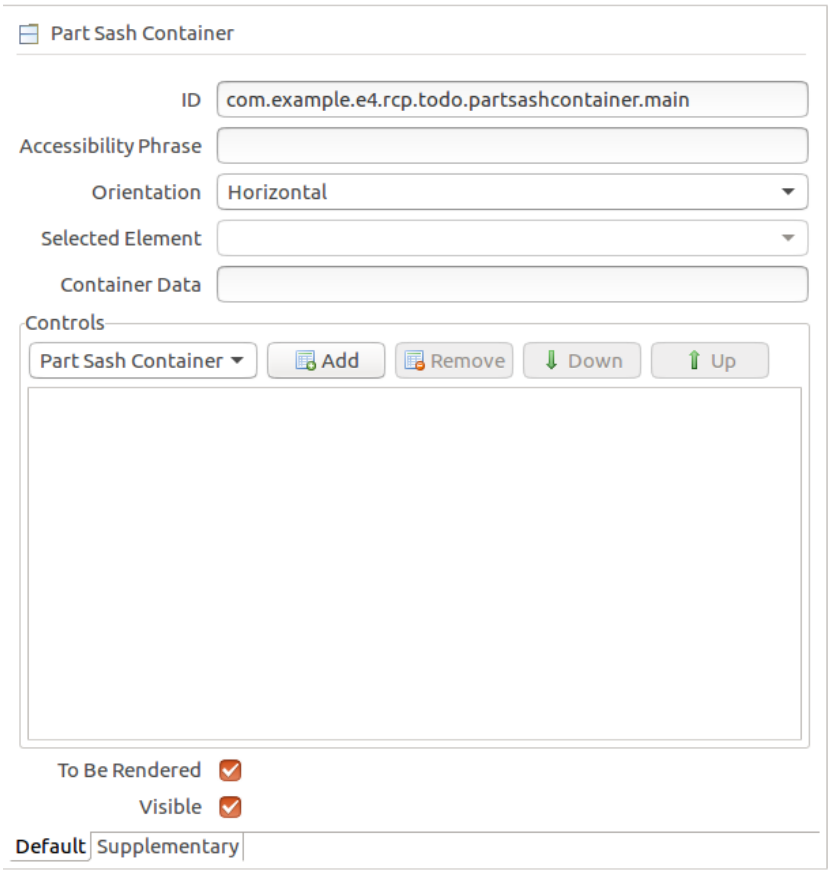
Open the *Application.e4xmi* file in the *Eclipse 4 model editor* via a double-click or right-click on it and select Open With ▸ Eclipse 4 model editor.

13.3.2. Add part sash and part stack containers

Select *Controls* below the window and add a part sash container element.



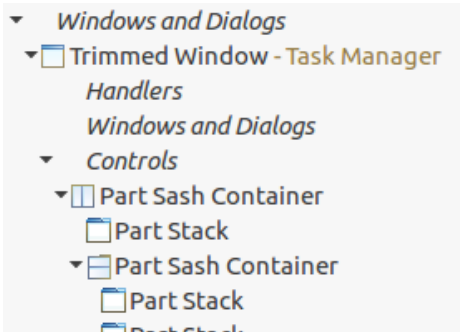
Change its *Orientation* attribute to *Horizontal* and enter into the ID field the *com.example.e4.rcp.todo.partsashcontainer.main* value.



Add a part stack as the first child to your part sash container element.

Re-select the parent part sash container and add another **part sash container** element. Add to stacks to this new element.

After these changes your application model should look similar to the following screenshot.



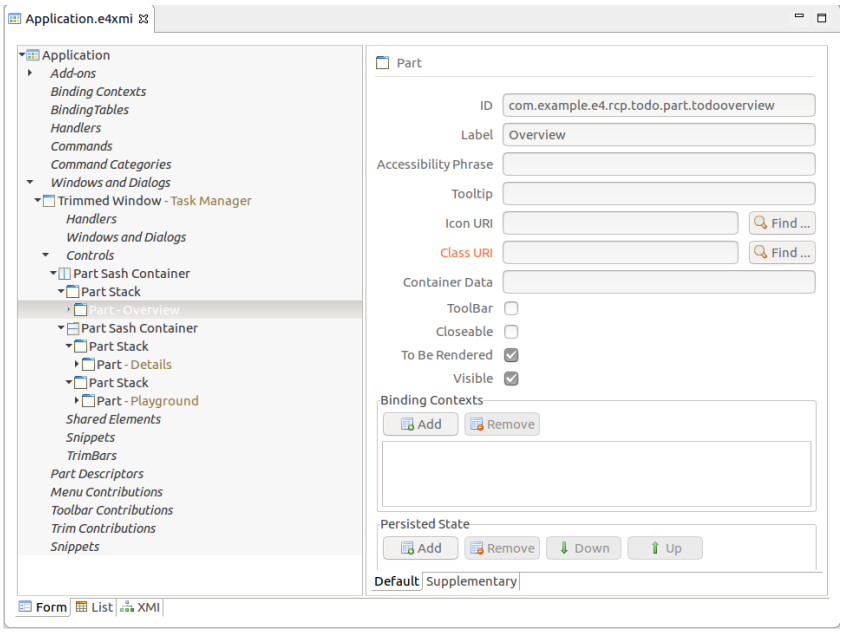
Add one part to each stack. Use the ID's and labels from the following table.

Enter the name first, the ID should be adjusted based on the name, if the label was empty.

Table 3. Label and ID for the parts

ID Suffix	Label
com.example.e4.rcp.todo.part.todooverview	Overview
com.example.e4.rcp.todo.part.tododetails	Details
com.example.e4.rcp.todo.part.playground	Playground

The final structure of your application model should be similar to the following screenshot. The screenshot also shows the detailed data of the overview part inside the detail pane of the application model editor.



13.3.4. Validate the user interface

Start your product and validate that the user interface looks as planned. Reassign your model elements if required. The model editor supports drag-and drop for reassignment.

Also note that you can already see the structure, even though you have not created any Java classes so far.

13.4. Enter the dependencies

In this part of the exercise you prepare your API dependencies.

13.4.1. Add the plug-in dependencies

In the source code of you new plug-in, you will later use the API from other Eclipse plug-ins. The Eclipse programming model requires that you define a dependency to such plug-ins or packages in your new plug-in.

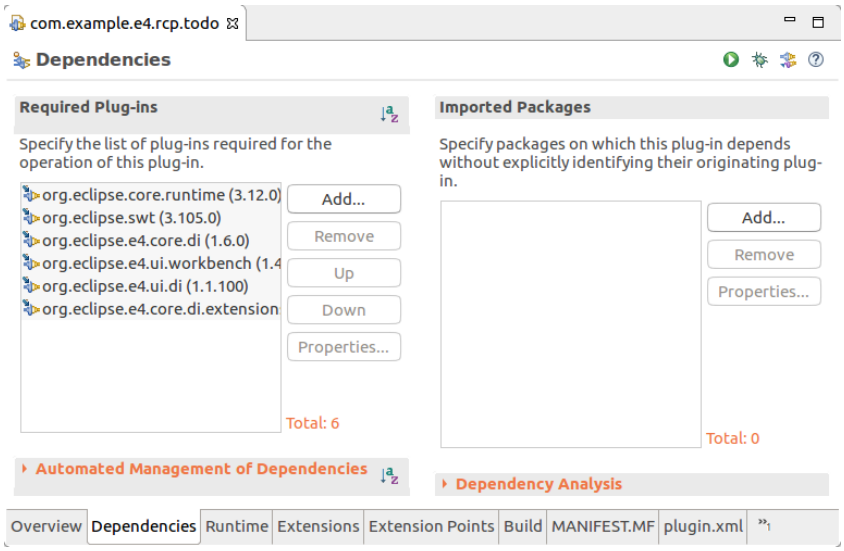
Remember that *application plug-in* is the short form for the `com.example.e4.rcp.todo` plug-in.

Open the *META-INF/MANIFEST.MF* file in your *application plug-in* and select the *Dependencies* tab. Use the Add... button in the *Required Plug-ins* section to add the following plug-ins as dependency.

- org.eclipse.core.runtime
- org.eclipse.swt
- org.eclipse.e4.core.di
- org.eclipse.e4.ui.workbench
- org.eclipse.e4.ui.di
- org.eclipse.e4.core.di.extensions

13.4.2. Validating

The result should be similar to the following screenshot.



13.5. Connect Java classes with the parts

13.5.1. Create a new package and some Java classes

Create the `com.example.e4.rcp.todo.parts` package in the application plug-in.

Create three Java classes called *TodoOverviewPart*, *TodoDetailsPart* and *PlaygroundPart* in this package. All classes should not extend

You can create the classes by clicking on the *Class URI* hyperlink in the detail pane of the model editor for the part. This also connects the created class to the model object. If you do this, you can skip Connect the Java classes with your parts.

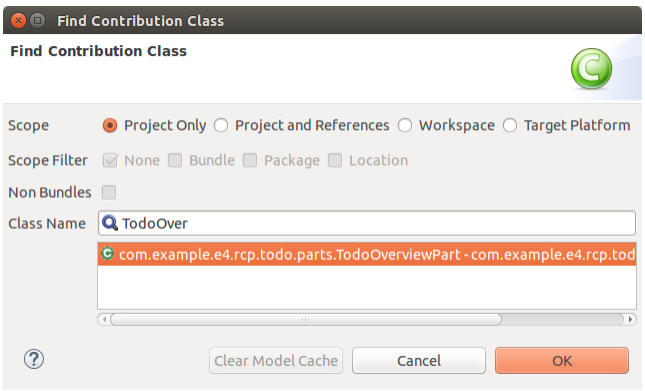
The following code shows the `TodoDetailsPart` class.

```java
package com.example.e4.rcp.todo.parts;

public class TodoDetailsPart {

}
```

13.5.2. Connect the Java classes with your parts

Open the *Application.e4xmi* file and connect the class with the corresponding part model element. You can do this via the *Class URI* property of the part model element.

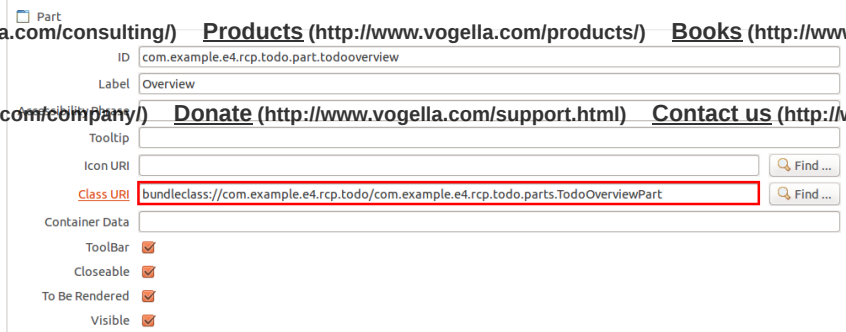
The Eclipse 4 model editor allows you to search for an existing class via the `Find...` button. The initial list of *Contribution Classes* is empty, start typing in the *Class Name* field to see the results.



The following table gives an overview of which elements should be connected.

Table 4. Mapping Java classes with part model element

Class	Part ID suffix
TodoOverviewPart	*.todooverview
TodoDetailsPart	*.tododetails
PlaygroundPart	*.playground



13.5.3. Validating

Start your application. It should start, but you should see no difference in the user interface.

To validate that the model objects are created by the Eclipse runtime create a no-argument constructor for one of the classes and add a `System.out.println()` statement. Verify that the constructor is called, once you start the application.

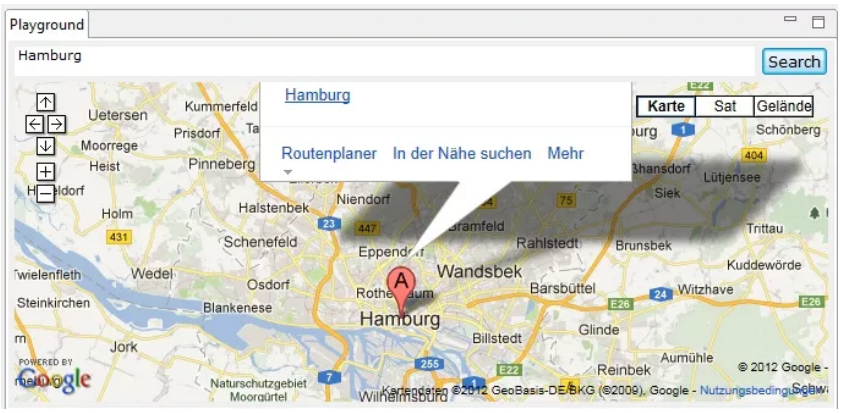
14. Exercise: Using the SWT browser widget

14.1. Implementation

In this exercise you use several SWT widgets to allow to search and display cities on Google maps.

This exercise does not always work on a Linux system because on certain Linux versions the `Browser` widget does not work. See the Eclipse SWT FAQ answered at How do I use the WebKit renderer on Linux-GTK for details (http://www.eclipse.org/swt/faq.php#browserwebkitgtk).

Change the `PlaygroundPart` class so the part looks like the following screenshot.



This example might not work in case Google

If you enter a text in the text field and press the button, the map should center based on the input in the text field. This input should be interpreted by

14.2. Solution

Your `PlaygroundPart` class should look similar to the following code.

JAVA

```
package com.example.e4.rcp.todo.parts;

import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;

import javax.annotation.PostConstruct;

import org.eclipse.e4.ui.di.Focus;
import org.eclipse.swt.SWT;
import org.eclipse.swt.browser.Browser;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Text;

public class PlaygroundPart {
        private Text text;
        private Browser browser;

        @PostConstruct
        public void createControls(Composite parent) {
                parent.setLayout(new GridLayout(2, false));

                text = new Text(parent, SWT.BORDER);
                text.setMessage("Enter City");
                text.setLayoutData(new GridData(SWT.FILL,
SWT.CENTER, true, false, 1, 1));

                Button button = new Button(parent, SWT.PUSH);
                button.setText("Search");
                button.addSelectionListener(new
SelectionAdapter() {
                        @Override
                        public void
widgetSelected(SelectionEvent e) {
                                String city = text.getText();
                                if (city.isEmpty()) {
                                        return;
                                }
                                try {
                                        // not supported at
the moment by Google
                                        //
browser.setUrl("http://maps.google.com/maps?q="
                                        // +
URLEncoder.encode(city, "UTF-8")
                                        // +
"&output=embed");
                                } catch
(UnsupportedEncodingException e1) {
                                        e1.printStackTrace();
                                }
                        }
                });

                browser = new Browser(parent, SWT.NONE);
                browser.setLayoutData(new GridData(SWT.FILL,
SWT.FILL, true, true, 2, 1));
```

```
        }

        @Focus
        public void onFocus() {
                text.setFocus();
        }
}
```

14.3. Introduction to dependency injection

See Dependency injection in Java (http://www.vogella.com/tutorials/DependencyInjection/article.html) for an introduction into the concept of dependency injection,

15. Dependency injection and Eclipse

15.1. Define class dependencies in Eclipse

The programming model in Eclipse supports constructor, method and field injection according to the Java Specification Request 330 (JSR330). Eclipse also defines additional annotations for the purpose of dependency injection. The most important annotations are covered in Annotations to define class dependencies in Eclipse, other more special annotations are covered in there corresponding chapters.

The Eclipse dependency framework ensures that the key and the type of the injected object is correct. For example, if you specify that you want to have an object of type `Todo` for the "xyz" key, as shown in the following field declaration, the framework will only inject an object if it finds one with an assignable type.

JAVA

```
@Inject @Named("xyz") Todo todo;
```

15.2. Annotations to define class dependencies in Eclipse

The following table gives an overview of dependency injection related annotations based on JSR330 and the Eclipse specific ones.

Table 5. Basic annotations for dependency injection

@javax.inject.Inject	Defined by JSR330, can be added to a field, a constructor or a method. The Eclipse framework tries to inject the corresponding objects into the fields or the parameters of the instance.

Annotation	Description
@javax.inject.Named	Defined by JSR330, defines the key for the value which should be injected. By default, the fully qualified class name is used as the key. Several keys for default values are defined as constants in the `IServiceConstants` interface.
@Optional	Eclipse specific annotation, marks an injected value to be optional. If no valid object can be determined for the given key (and type), the framework does not throw an exception. The specific behavior depends on where the `@Optional` is placed. The following description is based on the key. If the key cannot be resolved the following happens: * for parameters: a `null` value will be injected; * for methods: the method calls will be skipped * for fields: the values will not be injected. Note that null is an acceptable value to be set in the context, and it is different from a key being removed from the context. For example, if the following is called `context.set(SOMEKEY, null)`, anybody listening for SOMEKEY will be injected with null.
@GroupUpdates	Eclipse specific annotation, indicates that updates for this @Inject should be batched. If you change such objects in the Eclipse context, the update is triggered by the `processWaiting()` used by the platform for performance optimization and should rarely be necessary in RCP applications.

The Eclipse platform supports additional annotations for special purposes, e.g., for receiving events (sent by the event service) or working with preferences. For a summary of all standard annotations defined in the Eclipse platform see [eclipse4_annotations]

15.3. On which objects does Eclipse perform dependency injection?

The Eclipse runtime creates objects for the Java classes referred by the application model. During this instantiation the Eclipse runtime scans the class definition for annotations. Based on these annotations the Eclipse framework performs the injection.

Eclipse does not automatically perform dependency injection on objects which are created in your code with the `new` operator.

15.4. Dynamic dependency injection based on key / value changes

The Eclipse framework tracks which object expressed a dependency to which key and type. If the value to which a key points changes, the Eclipse framework re-injects the new value in the object which expressed a dependency to the corresponding type. This means applications can be freed from having to install (and remove) listeners.

For example, you can define via `@Inject` that you want to get the current selection injected. If the selection changes, the Eclipse framework will inject the new value.

The re-injection only works on methods and fields which are marked with `@Inject` . It will not work on parameters injected into constructors and methods which are marked with `@PostConstruct` , as these methods are only executed once.

This does not mean that Eclipse tracks the fields of the value to which the key points. For example if re-injection of the value to all objects which have a relevant class dependency. But if a field inside the existing `Todo` object changes, it does not trigger a re-injection.

16. The Eclipse context

16.1. What is the Eclipse context?

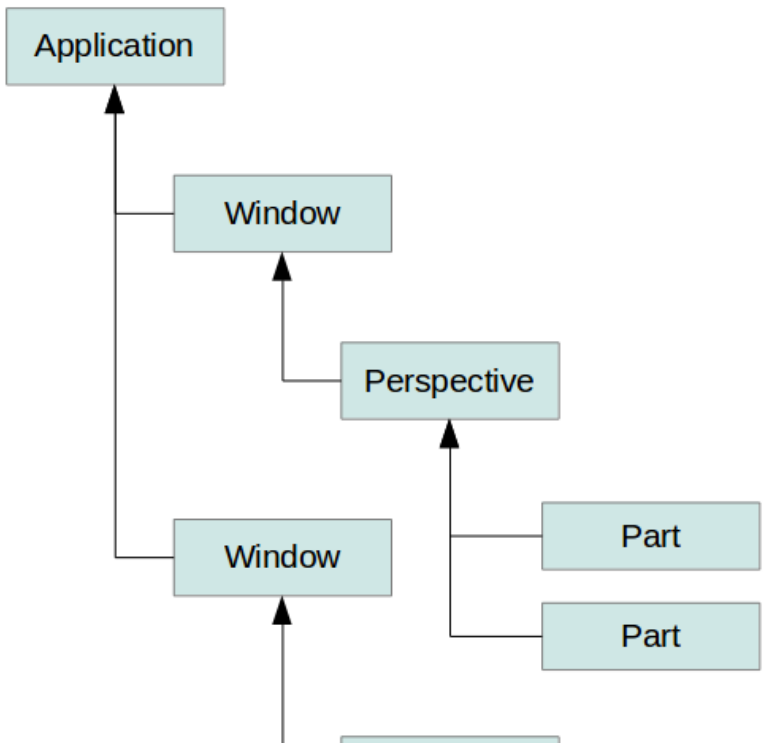
During startup of an Eclipse application the Eclipse runtime creates an object based on the `IEclipseContext` interface. This object is called the *context* or the *Eclipse context*.

The context is similar to a `Map` data structure, in which objects can be placed under a certain key. The key is a String and in several cases the fully qualified class name is used as key. The value (to which the key points) can be injected into other objects. But unlike a map, the Eclipse context is hierarchical and can also dynamically compute values for requested keys.

For certain model objects (see Connecting model elements to classes and resources) a local context is created. Such a context is associated with an application model object.

The different context objects are connected to form a hierarchical tree structure based on the structure of your application model. The highest level in this hierarchy is the application context.

A sample context hierarchy is depicted in the following picture.



Objects can be placed at different levels in the context hierarchy. This allows that the same key points to different objects in the hierarchy.

For example, a part can express a dependency to a `Composite` object via a field declaration similar to: `@Inject Composite parent;` Since parts have different local contexts they can receive different objects of the type `Composite`.

16.2. Which model elements have a local context?

Currently the following model elements implement the `MContext` interface and therefore have their own context:

- MApplication
- MWindow
- MPerspective
- MPart
- MPopupMenu

16.3. Life cycle of the Eclipse context

The Eclipse framework creates the context hierarchy based on the application model during the start process. By default, it places certain objects under predefined keys into the context, e.g., services to control the Eclipse framework functionality.

The model objects and the created objects based on the *class URI* attributes are created by the Eclipse platform. For each model element with a custom context the Eclipse framework determines which objects should be available in the local context of the model object. If required, it also creates the required Java objects referred by the *Class URI* property of the model elements. This is for example the case if a part is visible to the user.

The renderer framework is responsible for creating the local context of the UI related model elements. This framework allows you to define classes which are responsible for setting up the UI implementation of the model objects. A class responsible for a model element is called the *renderer* for this model element.

For example, the `ContributedPartRenderer` class is the default renderer for part model objects. This part.

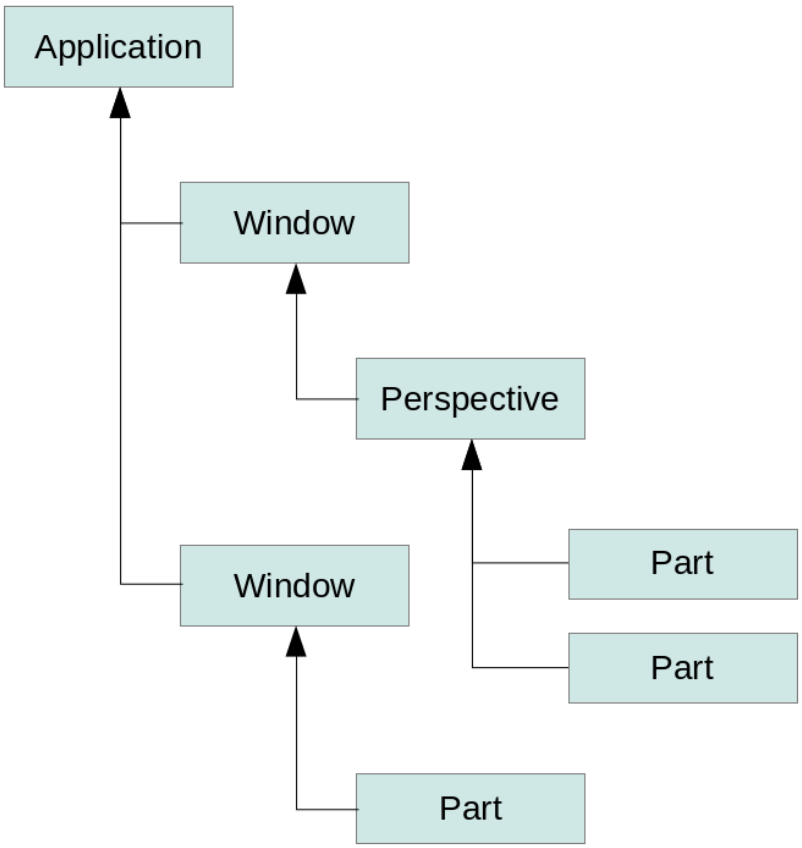
After the initial creation of the Eclipse context hierarchy, the framework or the application code can change the key-value pairs stored in the context. In this case objects which were created with the related Eclipse functionality (for example by the Eclipse dependency injection framework) are updated with the new values.

Objects in the context are persisted in memory (transient), i.e., once the application is stopped the context gets destroyed.

16.4. How are objects selected for dependency injection

As described in On which objects does Eclipse perform dependency injection? an object which is created by Eclipse can use annotations to describe its class dependencies.

During dependency injection for an object created by Eclipse, the Eclipse framework searches for a fitting object based on the specified key. The search starts in the local context associated with the application model object. If this key is not available, Eclipse continues to search in the parent context. This process continues until the main context has been reached.



custom objects. The search happens (mostly) transparently for the caller of the injection.

16.5. How to access the model objects?

For the class references in the application model, the Eclipse framework creates the corresponding objects when needed. Such an object has access to its corresponding model object via dependency injection.

For example, in the implementation of a part you can access the model information of a part via: `@Inject MPart part;`

16.6. Default entries in the Eclipse context

The Eclipse framework creates several objects in the context. These are:

- model objects - contain the data of the application model
- services - software components which are defined by the Eclipse platform or via the OSGi service registry
- several other objects which have explicitly been added to the context

The context can be modified by the application code and the framework. As the Eclipse framework automatically tracks the dependencies of the objects it creates, it can update them as described in Dynamic dependency injection based on key / value changes.

16.7. Qualifiers for accessing the active part or shell

The Eclipse platform places the part which is currently selected and the active shell into the `IEclipseContext` of the application object. The related keys are defined in the `IServiceConstants` interface.

For example, the following method would allow you to track the current active part in another part.

JAVA

```
// tracks the active part
@Inject
@Optional
public void receiveActivePart(
        @Named(IServiceConstants.ACTIVE_PART) MPart
activePart) {
        if (activePart != null) {
                System.out.println("Active part changed "
                                + activePart.getLabel());
        }
}
```


JAVA

```
// tracks the active shell
@Inject
@Optional
public void receiveActiveShell(
        @Named(IServiceConstants.ACTIVE_SHELL) Shell shell) {
        if (shell != null) {
                System.out.println("Active shell (Window)
changed");
        }
}
```

Eclipse uses handlers to define actions which can be triggered via menu or toolbar entries. For a handler implementation class it is not necessary to use these qualifiers, as a handler is executed in the active context of the application.

16.8. Tracking a child context with @Active

The `@Active` annotation allows you to track values in a child context. The Eclipse framework keeps track of the current active branch in the hierarchy of the `IEclipseContext` . For example, if the user selects a part, the path in the `IEclipseContext` hierarchy from the root to the `IEclipseContext` of the part is the current active branch.

With the `@Active` annotation you can track values in the current active branch of a child element. Whenever the active branch changes and the value of the referred key changes this value is re-injected into the object which uses the `@Active` annotation.

The usage of this annotation is demonstrated by the followingcode snippet.

JAVA

```
public class MyOwnClass {
        @Inject
        void setChildValue(
                        @Optional
@Named("key_of_child_value") @Active String value) {
                this.childValue = value;
        }
}
```

The `@Active` annotation is currently not used within the Eclipse framework itself and the author of this tutorial has not yet managed to find a good use case for this annotation.

17. Using annotations to define behavior

17.1. API definition in a framework

convention for how your application interacts with the framework. For example, if a Java object is responsible for handling a toolbar button click, the framework needs to know which method of this object needs to be called.

For this purpose every framework defines an Application Programming Interface (API). This API defines how you can interact with the framework from your code. The API also defines

the interaction of application objects created or controlled by the framework. Typically, a framework uses inheritance or annotations for this purpose.

17.2. API definition via inheritance

The "traditional" way of defining an API is via inheritance. This approach requires that your classes extend or implement framework classes and interfaces. The Eclipse 3.x platform API used this approach.

The framework defines, for example, an abstract class which defines methods to be implemented.

In the example of the toolbar button the method might be called `execute()` and the framework knows that this method must be called once the button is clicked.

API definition via inheritance is a simple way to define an API, but it also couples the classes tightly to the framework. For example, testing the class without the framework is difficult. It also makes extending or updating the framework difficult as such an update may affect clients. This is why the Eclipse 4.x does not use this approach anymore.

17.3. API definition via annotations

The Eclipse 4.x platform API is based on annotations, e.g., annotations are used to identify which methods should be called at a certain point in time. These annotations are called *behavior annotations*.

The following table lists the available behavior annotations for parts.

Table 6. Eclipse life cycle annotations for parts

Annotation	Description
@PostConstruct	Is called after the class is constructed and the field and method injection has been
@PreDestroy	Is called before the class is destroyed. Can be used to clean up resources.
@Focus	Is called whenever the part gets the focus.
@Persist	Is called if a save request on the part is triggered by the Eclipse framework.

Annotation	Description
@PersistState	Is called before the model object is disposed, so that the part is able to save its instance state. This method is called before the @PreDestroy method.

The `@PostConstruct`, `@PreDestroy` annotations are included in the `javax.annotation` package. `@Persist`, `@PersistState` and `@Focus` are part of the `org.eclipse.e4.ui.di` package.

Eclipse defines additional behavior annotations for commands and for the application life cycle which are covered in the respective chapters.

Behavior annotations imply that the framework needs to provide the specified parameters to the method, i.e., the framework also performs method dependency injection. If you also add the `@Inject` annotation, the method is called twice, first during the dependency injection phase and later for the behavior annotation. This is typically undesired and therefore an error.

17.4. Use the @PostConstruct method to build the user interface

It is recommended to construct the user interface of a part in a method annotated with the `@PostConstruct` annotation. It would also be possible to create the user interface in the constructor, but this is not recommended as field and method injection have not been done at this point.

Creating the user interface in an `@PostConstruct` method requires that `@Inject` methods are aware that the user interface might not have been created yet.

Why is the @PostConstruct method not called?

The following description is only valid for Eclipse versions before the Eclipse 4.6 (Eclipse Neon) release. As of Eclipse 4.6 the framework uses the Java version of `@PostConstruct` and the problem described here, cannot happen anymore.

Before Eclipse 4.6, both Java 7 and the Eclipse platform exposed the `@PostConstruct` annotation. In your Eclipse application you need to tell the framework that the annotation from the Eclipse platform should be used. `org.eclipse.core.runtime` exports `javax.annotation` in the correct version. If, for some reasons, you want to avoid a dependency to `org.eclipse.core.runtime`, you could define a package dependency to the `javax.annotation` package and set the version to 1.0.0. See Eclipse 4 RCP FAQ (http://wiki.eclipse.org/Eclipse4/RCP/FAQ) for details on this issue.

18. Exercise: Using @PostConstruct

18.1. Implement an @PostConstruct method

Add the following method to your `TodoOverviewPart`, `TodoDetailsPart` and `PlaygroundPart` classes. In case you created constructors for these classes you can remove them.

JAVA

```
import javax.annotation.PostConstruct;
import org.eclipse.swt.widgets.Composite;

// more code

@PostConstruct
public void createControls(Composite parent) {
        System.out.println(this.getClass().getSimpleName()
        + " @PostConstruct method called.");
}
```

18.2. Validating

method is called.

19. Menu and toolbar application objects

19.1. Adding menu and toolbar entries

You can add menus and toolbars to your RCP application via the application model. These entries can be positioned at various places. You can, for example, add a menu to a window or a part. These elements define, directly or indirectly, a link to a class. An

instance of this class is created by the framework and responsible for the behavior if the menu or toolbar entry is selected. Such a class is called *handler class*.

19.2. The usage of commands and handlers

The Eclipse application model allows you to specify *commands* and *handlers*.

The usage of the commands and handlers model element is optional. You can use the *Direct MenuItem* or a *Direct ToolItem* model elements. These entries define a reference to a class (handler class). An instance of this handler class is created by the framework and its annotated methods are called by the framework if necessary. Menus and toolbars support separators.

A command is a declarative description of an abstract action which can be performed, for example, *save*, *edit* or *copy*.

A command is independent from its implementation details. The Eclipse framework does not provide standard commands, e.g., you have to create all required commands in your application model.

The behavior of a command is defined via a handler. A handler model element points to a class (handler class) via the `contributionURI` property of the handler. This attribute is displayed as *Class URI* in the model editor.

Commands are used by the *Handled MenuItem* and *Handled ToolItem* model elements.

Prefer the usage of commands over the usage of direct (menu or tool) items. Using commands together with handlers allows you to define different handlers for different scopes (applications or part) and you can define key bindings for the handler's associated commands.

19.3. Behavior annotations and dependency injection for handler classes

In a handler class exactly one method must be annotated with the

than one method with the same annotation, the framework calls only one of them. The Eclipse runtime uses dependency injection to provide the parameters of the method. The purpose of these annotations are described in the following table.

Table 7. Behavior annotations for handler classes

Annotation	Description

@Execute	Marks the method which is responsible for the action of the handler class. The framework executes this method once the related user interface element, e.g., the menu entry, is selected.
@CanExecute	Marks a method to be visited by the Eclipse framework to check if the handler class can be executed. If a handler class returns `false` in this method, Eclipse disables the corresponding user interface element. For example, the save button is active if the handler class returns true in the `@CanExecute` method. The default for this method is true, which means, if the handler class can always be executed, it does not need to implement a `@CanExecute` method.

The following example demonstrates the implementation of a handler class.

JAVA

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

// import statements cut out
// ..

public class ExitHandler {
        @Execute
        public void execute(IWorkbench workbench) {
                workbench.close();
        }

        // NOT REQUIRED IN THIS EXAMPLE
        // just to demonstrates the usage of
        // the annotation
        @CanExecute
        public boolean canExecute() {
                return true;
        }

}
```

A handler instance does not have its own Eclipse context (`IEclipseContext`). It is executed with the Eclipse context of the

All required parameters should be injected into the method annotated with `@Execute` , as you want the handler class to retrieve its runtime information during execution.

To ensure that you get the expected values from the active context ALWAYS get the required values injected as parameters into your methods annotated with @Execute or @CanExecute.

19.4. Determining the relevant handler for a command

If a command is selected, the runtime determines the relevant handler for the command. The application model allows you to create a handler for the application, a window and a part.

Each command can have only one valid handler for a given scope. The Eclipse framework selects the handler most specific to the model element.

For example, if you have two handlers for the "Copy" command, one for the window and another one for the part then the runtime selects the handlers closest to model element which is currently selected by the user.

19.5. Evaluation of @CanExecute

A method annotated with `@CanExecute` is called by the framework, if a change in the Eclipse context happens. For example, if you select a new part. If the method returns `false`, the framework disables any menu and tool items that point to that command.

You can request the re-evaluation of the `@CanExecute` methods by sending out an event via the event broker.

JAVA

```
// evaluate all @CanExecute methods
eventBroker.send(UIEvents.REQUEST_ENABLEMENT_UPDATE_TOPIC,
UIEvents.ALL_ELEMENT_ID);

// evaluate a context via a selector
Selector s = (a selector that an MApplicationElement or an
ID);
eventBroker.send(UIEvents.REQUEST_ENABLEMENT_UPDATE_TOPIC,
s);
```


19.6. Mnemonics

The application model allows you to define *mnemonics*. A mnemonic appears as an underlined letter in the menu when the user presses and holds the `ALT` key and allows the user to quickly access menu entries by keyboard.

You specify mnemonics by prefixing the letter intended to be the mnemonic with an ampersand (&) in the label definition. For example, if you use the the label *&Save*, the S will be underlined if the `Alt` key is pressed.

19.7. Naming schema for command and handler IDs

A good convention is to start IDs with the *top level package name* of your project and to use only lower case letters.

The IDs of commands and handlers should reflect their relationship. For example, if you implement a command with the `com.example.contacts.commands.show` ID, you should use `com.example.contacts.handler.show` as the ID for the handler. If you have more than one handler for one command, add another suffix to it, describing its purpose, e.g., `com.example.contacts.handler.show.details` .

In case you implement commonly used functions, e.g., save, copy, you should use the existing platform IDs, as some Eclipse contributions expect these IDs to better integrate with the OS (e.g., on Mac OS, preferences are normally placed under the first menu). A more complete list of command IDs is available in `org.eclipse.ui.IWorkbenchCommandConstants` .

Table 8. Default IDs for commonly used commands

Command	ID
Save	org.eclipse.ui.file.save
Save All	org.eclipse.ui.file.saveAll
Undo	org.eclipse.ui.edit.undo
Redo	org.eclipse.ui.edit.redo
Cut	org.eclipse.ui.edit.cut
Copy	org.eclipse.ui.edit.copy
Delete	org.eclipse.ui.edit.delete
Import	org.eclipse.ui.file.import
Export	org.eclipse.ui.file.export
Select All	org.eclipse.ui.edit.selectAll
About	org.eclipse.ui.help.aboutAction

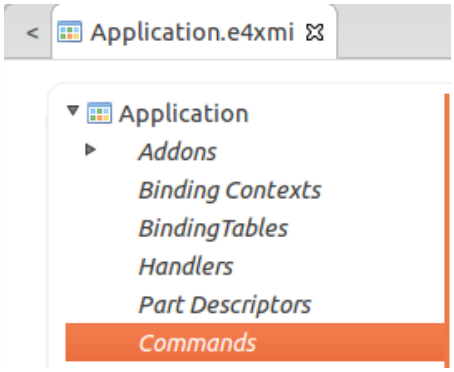
Command	ID
Preferences	org.eclipse.ui.window.preferences
Exit	org.eclipse.ui.file.exit

20. Exercise: Adding menus

In this exercise you create commands and handlers for your application. Afterwards you will create menu entries using these commands.

20.1. Create command model elements

Open the *Application.e4xmi* file of your `com.example.e4.rcp.todo` plug-in and select the *Commands* entry. This selection is highlighted in the following screenshot.



Via the Add... button you can create new commands. The name and the ID are the important fields. Create the following commands.

Table 9. Commands

ID	Name
org.eclipse.ui.file.saveAll	Save
com.example.e4.rcp.todo.command.new	New Todo
com.example.e4.rcp.todo.command.remove	Remove Todo
com.example.e4.rcp.todo.command.test	For testing

20.2. Creating the handler classes

Create the `com.example.e4.rcp.todo.handlers` package for your handler classes.

All handler classes implement an `execute()` method annotated with `@Execute`.

JAVA

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

import org.eclipse.e4.core.di.annotations.Execute;

public class SaveAllHandler {
        @Execute
        public void execute() {
          System.out.println((this.getClass().getSimpleName()
+ " called"));
        }
}
```

Using this template for all classes, implement the following classes.

- SaveAllHandler
- ExitHandler
- NewTodoHandler
- RemoveTodoHandler
- TestHandler

20.3. Creating handler model elements

Select the application-scoped *Handlers* entry in your application model and create the handlers from the following table for your commands. For the definition of handlers the ID, command and class are the relevant information.

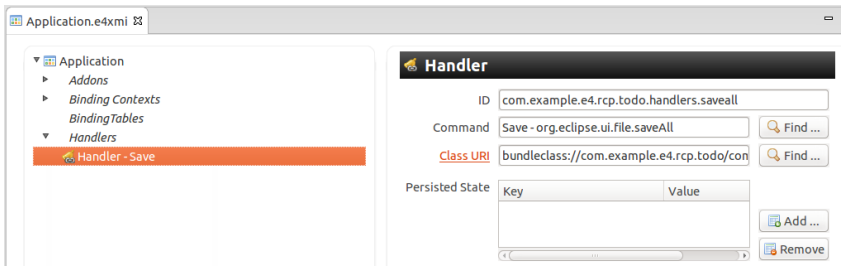
Use the `com.example.e4.rcp.todo.handler` prefix for all IDs of the handlers.

Table 10. Handlers

Handler ID	Command	Class
saveall	Save	SaveAllHandler
.exit	Exit	ExitHandler
.new	New Todo	NewTodoHandler
.test	For testing	TestHandler

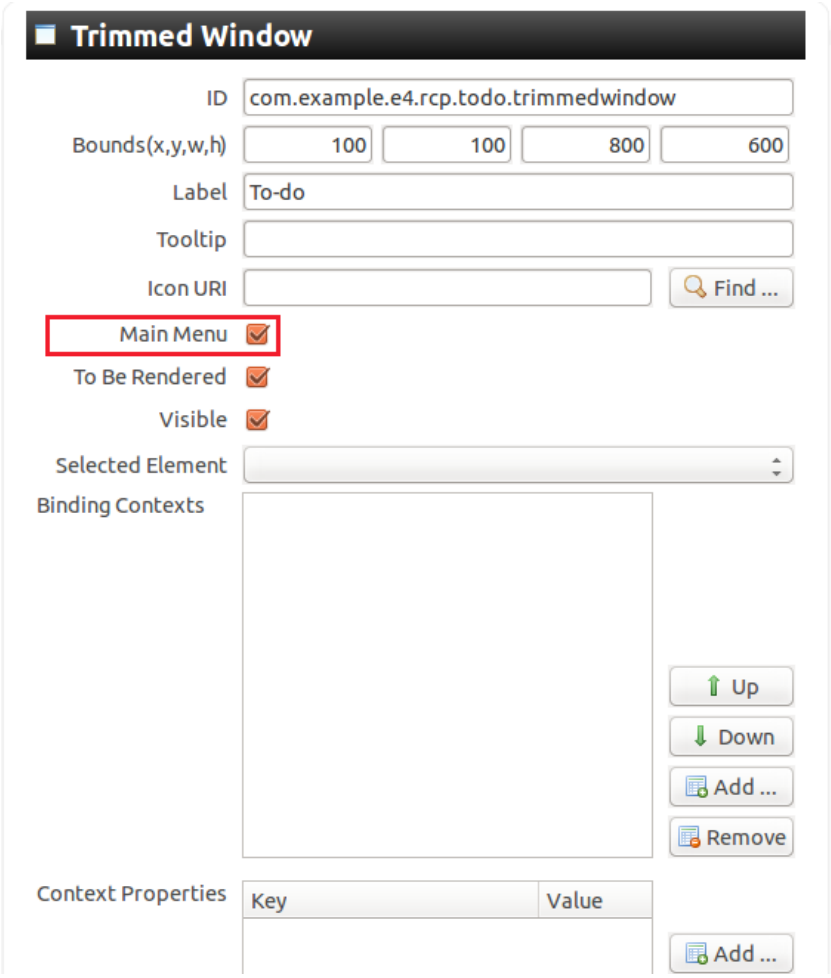
The application model editor shows both the name and the ID of the command. The class URI follows the `bundleclass://` schema, the table only defines the class name to make the table more readable. For example, for the save handler this looks like the following:

```
bundleclass://com.example.e4.rcp.todo/com.example.e4.rcp.todo.handlers.SaveAllHandler
```
JAVA



20.4. Adding a menu

In your *Application.e4xmi* file select your *TrimmedWindow* entry in the model and flag the *Main Menu* attribute.

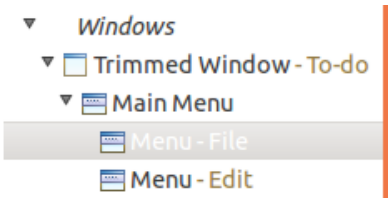


Assign the `org.eclipse.ui.main.menu` ID to your main menu.

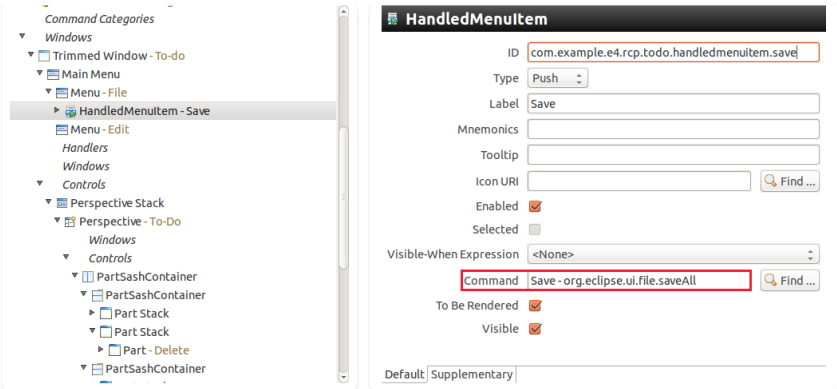
Ensure that this ID of the main menu is correct. You use it later to contribute another menu entry via another plug-in.

Add two menus, one with the name "File" and the other one with the name "Edit" in the *Label* attribute.

Also set the *org.eclipse.ui.file.menu* ID for the File menu. Use *com.example.e4.rcp.todo.menu.edit* as ID for the Edit menu.



Add a *Handled MenuItem* model element to the File menu. This item should point to the *Save* command via the *Command* attribute.



Add a *Separator* after the Save menu item and after that add an entry for the Exit command.

Add all other commands to the Edit menu.

20.5. Implement a handler class for exit

To test if your handler is working, change your `ExitHandler` class, so that it closes your application, once selected.

JAVA

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

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.workbench.IWorkbench;

public class ExitHandler {
        @Execute
        }
}
```


20.6. Validating

Validate that your save handler is called if you select Save from the menu.

Also check that you can exit the application via the *Exit* menu entry.

20.7. Possible issue: Exit menu entry on a MacOS

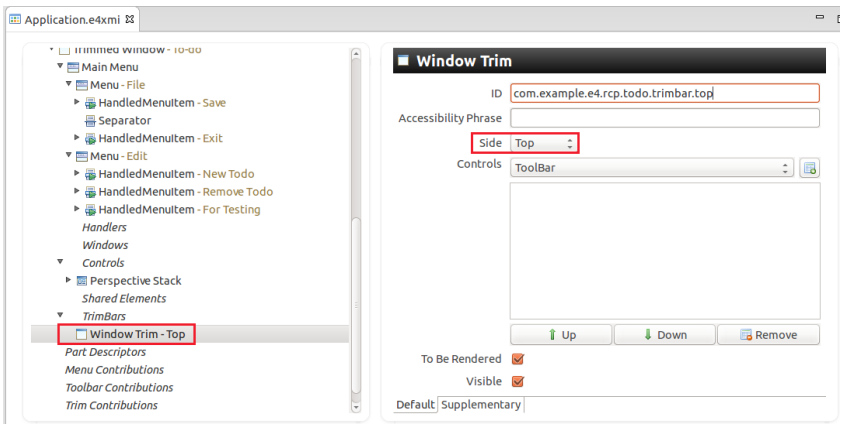
If you use the "org.eclipse.ui.file.exit" ID for your exit command, the Eclipse framework tries to map the exit command to its default menu location on the MacOS. If you don't see your exit menu, in its defined position, check this location.

21. Exercise: Adding a toolbar

In this exercise you add a toolbar to your application.

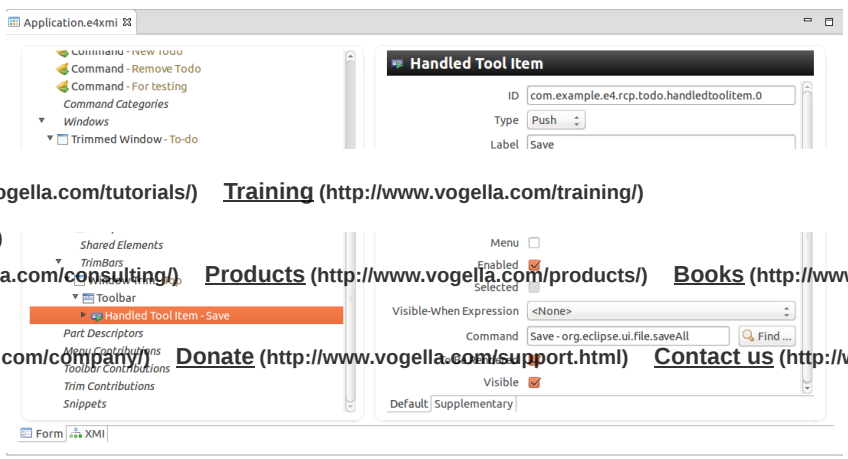
21.1. Adding a toolbar

Select the *TrimBars* node under your *TrimmedWindow* entry and press the Add... button. The *Side* attribute should be set to *Top*, so that all toolbars assigned to that trimbar appear on the top of the application.



Add a *ToolBar* model element to your TrimBar. Add a *Handled ToolItem* to this toolbar which points to the `org.eclipse.ui.file.saveAll` command.

Set the label for this entry to *Save*.



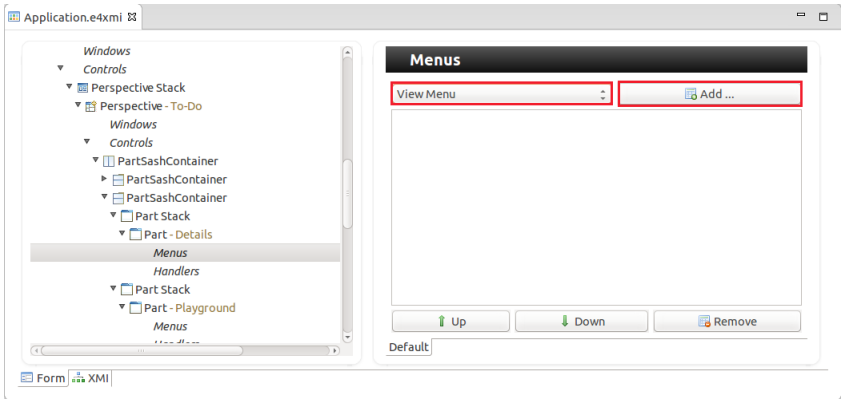
21.2. Validating

Validate that your save handler is called if you select Save from the menu or the toolbar.

22. View, popup and dynamic menus

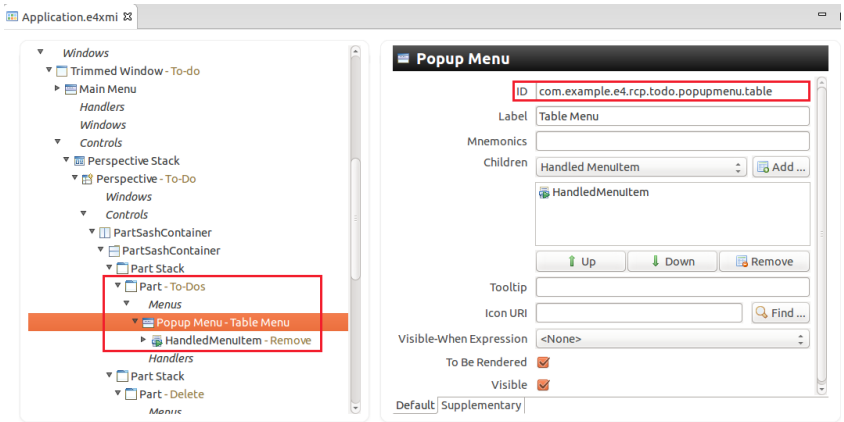
22.1. View menus

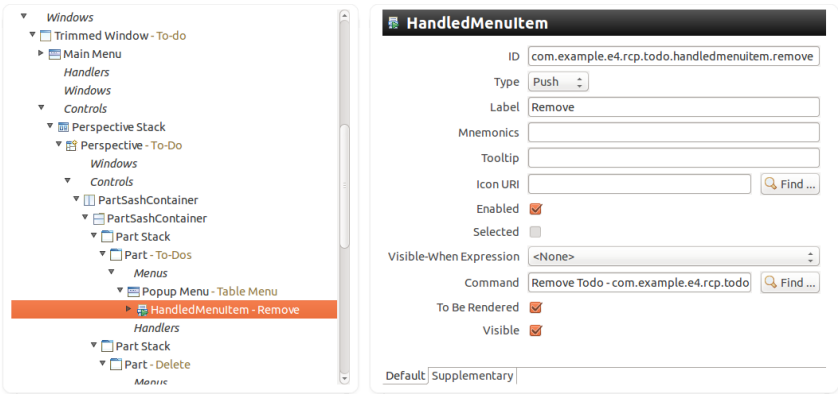
You can define one menu in a part as a *view menu*. Please note that the application model allows to define more than one menu but the default Eclipse implementation supports only one menu for a part. To add a view menu entry, select the *Menus* entry under the part and append a *ViewMenu* model entry to it.



22.2. Popup menu (context menu)

You can also define a popup menu for SWT controls via the application model. To achieve this create a *Popup Menu* for the part which contains the SWT control.





After this the pop menu can be assigned to an SWT control with the `EMenuService` service which can be accessed via dependency injection. This class provides the `registerContextMenu(control, id)` method for this purpose. The *id* parameter of the `registerContextMenu` method must be the ID attribute of your *Popup Menu* model element.

The following pseudo code shows an example for the registration. It uses a JFace viewer, as the popup menu needs to be registered on the SWT control, the example code demonstrates how to access this control.

JAVA

```
package com.example.e4.rcp.todo.parts;

import javax.annotation.PostConstruct;

import org.eclipse.e4.ui.services.EMenuService;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Text;

public class TodoOverviewPart {

        @PostConstruct
        public void createControls(Composite parent,
EMenuService menuService) {
                // more code...
                TableViewer viewer = new TableViewer(parent,
SWT.FULLSELECTION | SWT.MULTI);

                // more code

                // register context menu on the table
                menuService.registerContextMenu(
                }
}
```

If you want to implement this example, your plug-in must have dependencies defined for the `org.eclipse.e4.ui.workbench.swt`, the `org.eclipse.e4.ui.services` and the `org.eclipse.e4.ui.model.workbench` plug-ins in its *MANIFEST.MF* file.

22.3. Dynamic menu and toolbar entries

You can also create menu and toolbar entries at runtime with the *DynamicMenuContribution* model element.

This model element points to a class in which you annotate a method with the `@AboutToShow` annotation. The annotated method is called if the user selects the user interface element. The `@AboutToHide` annotation can be used to annotate a method which is called before the menu is hidden.

In these methods you can dynamically create menu entries.

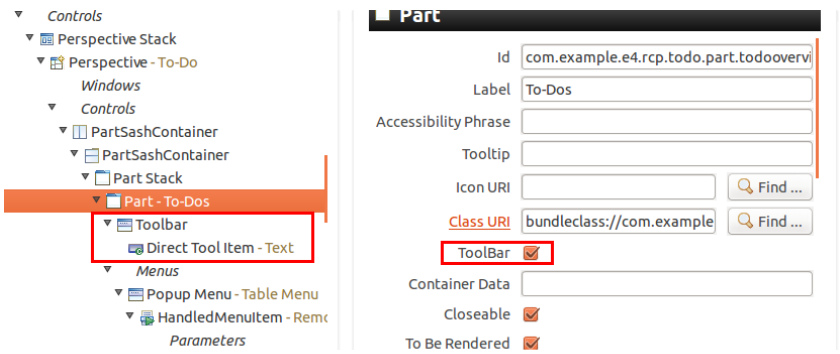
23. Toolbars, ToolControls and drop-down tool items

23.1. Adding toolbars to parts

Toolbars in the application model are defined within the *Trimbars* model element. A trimbar can be defined for *TrimmedWindow* model elements. Via its *Side* attribute you define if the trimbar should be placed on the top, left, right or bottom corner of the resulting window.

To add a toolbar to a view, set the *Toolbar* flag on the model element for the part and create the entries in the application model.

Such an example setup is displayed in the following screenshot.



23.2. ToolControls

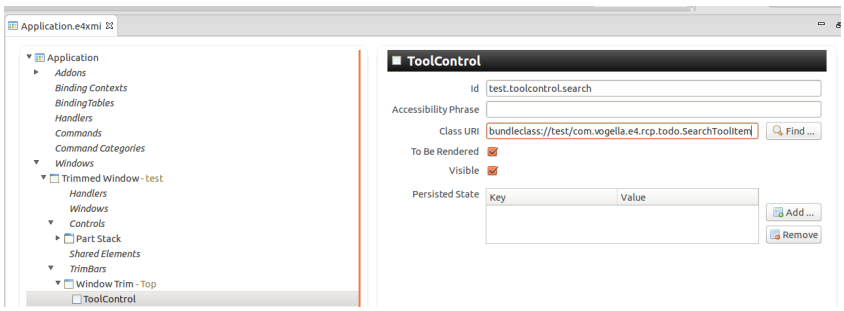
ToolControl model element points to a Java class which can create controls that are displayed in the toolbar.

For example, the following code creates a `Text` field in the toolbar which looks like a search field.

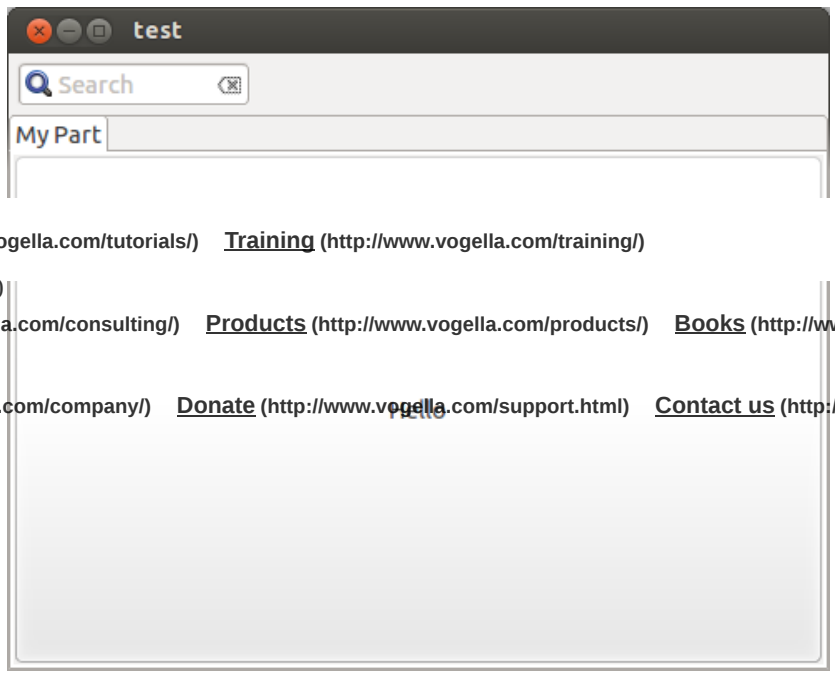
JAVA

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

import javax.annotation.PostConstruct;

import org.eclipse.jface.layout.GridDataFactory;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Text;

public class SearchToolItem {
        @PostConstruct
        public void createControls(Composite parent) {
                final Composite comp = new Composite(parent,
SWT.NONE);
                comp.setLayout(new GridLayout());
                Text text = new Text(comp, SWT.SEARCH |
SWT.ICON_SEARCH | SWT.CANCEL
                                | SWT.BORDER);
                text.setMessage("Search");
                GridDataFactory.fillDefaults().hint(130,
SWT.DEFAULT).applyTo(text);

        }
}
```

You can add such a `ToolControl` , for example, to your windows trimbar as depicted in the following screenshot.

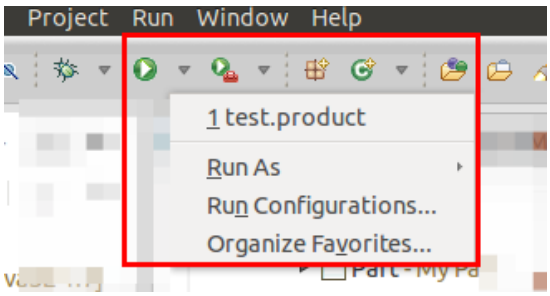


The following screenshot shows this `ToolControl` used in an example RCP application.



23.3. Drop-down tool items

Set the *Menu* attribute on an toolitem to be able to define a menu similar to the Run As... button in the Eclipse IDE. This button is depicted in the following screenshot.

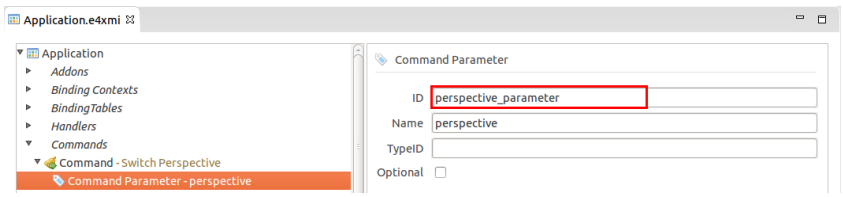


24. More on commands and handlers

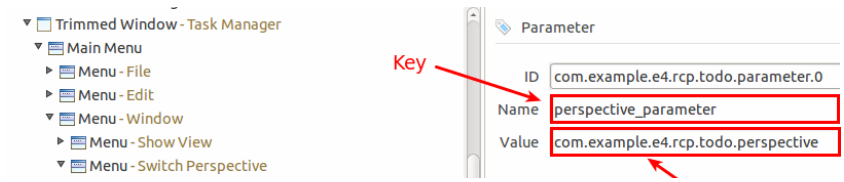
24.1. Passing parameters to commands

You can also pass parameters to commands. For this select a command and press the Add button in the *Parameters* section.

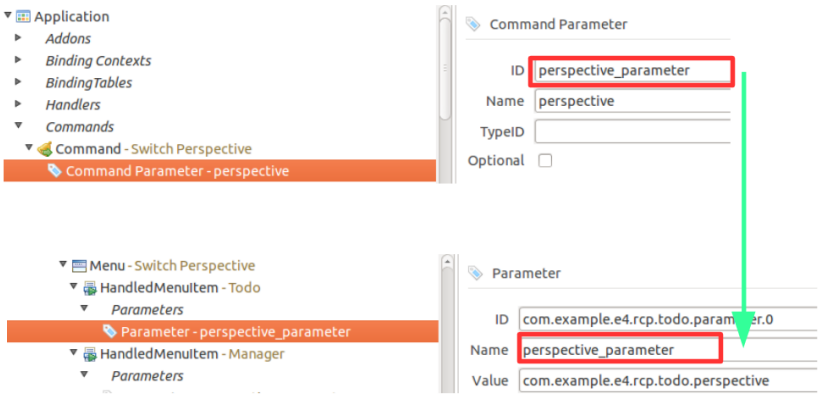
The ID is the identifier which you can use to get the parameter value injected via the `@Named` annotation.



In your HandledMenuItem or HandledToolItem add a parameter and put the ID from the command parameter definition into the *Name* field. The entry from the *Value* field is passed to the handler of the command.



The ID of the parameter is the important one. This ID must be injected via the `@Named` annotation and used as *Name* (second field) during the definition of the menu or toolbar. This is highlighted in the following picture.



To get the parameter injected into your handler class you specify the ID of the parameter via the `@Named` annotation. This is demonstrated in the following code example.

JAVA

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

import java.util.List;

import javax.inject.Named;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.model.application.MApplication;
import
org.eclipse.e4.ui.model.application.ui.advanced.MPerspective;
import org.eclipse.e4.ui.model.application.ui.basic.MWindow;
import org.eclipse.e4.ui.workbench.modeling.EModelService;
import org.eclipse.e4.ui.workbench.modeling.EPartService;

// switcher perspectives based on a command parameter
public class PerspectiveSwitchHandler {

        @Execute
        public void switchPerspective(
                        MWindow window,
                        EPartService partService,
                        EModelService modelService,
                        @Named("perspective_parameter")
String perspectiveId) {
                // use parameter to find perspectives
                List<MPerspective> perspectives =
modelService.findElements(window,
                                        perspectiveId,
MPerspective.class, null);

                // switch to perspective with the ID if found
                if (!perspectives.isEmpty()) {
                        ,
                }
        }
}
```

Alternatively to injecting each parameter, you can also inject the `ParameterizedCommand` command and access the parameters via API.

JAVA

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

import javax.inject.Named;

import
org.eclipse.e4.core.di.annotations.Execute;

public class TestHandlerWithCommandInjected {
        private String parametername =
"com.example.e4.rcp.todo" +

".commandparameter.input";
        @Execute
        public void
execute(ParameterizedCommand command) {
          Object queryId =
command.getParameterMap().get(parametername);
          // more...
        }


}
```

24.2. Usage of core expressions

The visibility of menus, toolbars and their entries can be restricted via *core expressions*. You add the corresponding attribute in the application model to the ID defined by the `org.eclipse.core.expressions.definitions` extension point in the *plugin.xml* file.

To add this extension point to your application, open the *plugin.xml* file and select the *Dependencies* tab in the editor. Add the `org.eclipse.core.expressions` plug-in in the *Required Plug-ins* section.

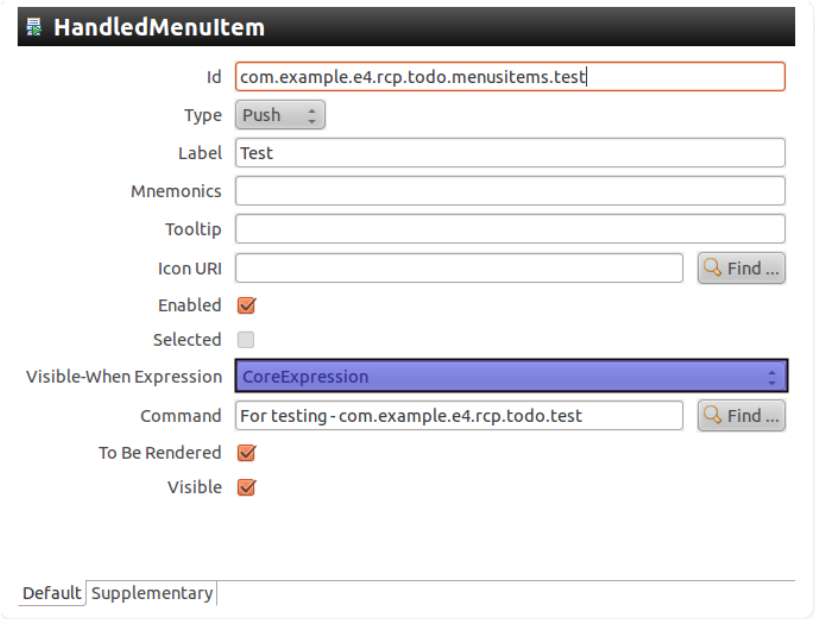
Afterwards select the *Extensions* tab, press the Add button and add the `org.eclipse.core.expressions.definitions` extension. You define an ID under which the core expression can be referred

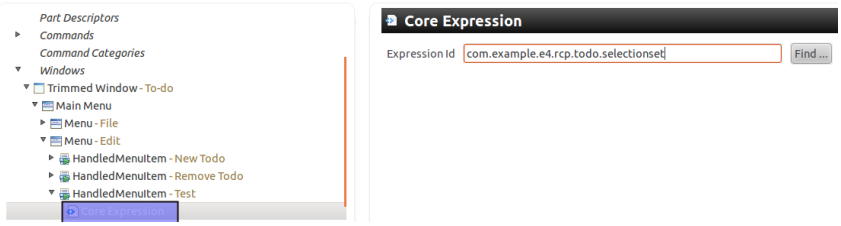
Via right-click on the extension you can start building your expression.

The following example can be used to restrict the visibility of a menu entry based on the type of the current selection. You will later learn how to set the current selection. Please note that the variable for the selection is currently called `org.eclipse.ui.selection` . In Eclipse 3.x this variable is called `selection` .

```xml
<extension
      point="org.eclipse.core.expressions.definitions">
        <definition
            id="com.example.e4.rcp.todo.selectionset">
      <with variable="org.eclipse.ui.selection">
                <iterate ifEmpty="false" operator="or">
                  <instanceof
value="com.example.e4.rcp.todo.model.Todo">
                  </instanceof>
                </iterate>
      </with>
        </definition>
</extension>
```

You can assign this core expression to your menu entry in the application model. It can be used to restrict the visibility of model elements.





This approach is similar to the definition of core expressions in

The values available for Eclipse 3.x are contained in the `ISources` interface and documented in the Eclipse core expressions wiki (http://wiki.eclipse.org/Command_Core_Expressions). Eclipse 4 does not always support the same variables, but the wiki documentation might still be helpful.

24.3. Evaluate your own values in core expressions

You can also place values in the `IEclipseContext` of your application and use these for your visible-when evaluation.

The following code demonstrates an example handler class which places a value for the *myactivePartId* key in the context (you will learn more about modifying the `IEclipseContext` later).

JAVA

```
@Execute
public void execute(IEclipseContext context) {
  // put an example value in the context
  context.set("myactivePartId",
  "com.example.e4.rcp.todo.part.todooverview");
}
```

The following shows an example core expression which evaluates to `true` if an *myactivePartId* key with the value `com.example.e4.rcp.ui.parts.todooverview` is found in the context.

XML

```
<extension
         point="org.eclipse.core.expressions.definitions">
      <definition

id="com.example.e4.rcp.todo.todooverviewselected">
         <with
               variable="myactivePartId">
            <equals

value="com.example.e4.rcp.todo.part.todooverview">
            </equals>
         </with>
      </definition>
</extension>
```

This core expression can get assigned to a menu entry and control the visibility.

25. Key bindings

It is also possible to define key bindings (short cuts) for your Eclipse application. This requires two steps, first you need to enter values for the *BindingContext* node of your application model.

Afterwards you need to enter the key bindings for the relevant binding context in the *BindingTable* node of your application model. A binding table is always assigned to a specific binding context. A binding context can have several binding tables assigned to it.

Binding contexts are defined hierarchically, so that key bindings in a child override the matching key binding in the parent.

Even though they sound similar a binding context is used for keybindings while the Eclipse context (`IEclipseContext`) is used as source for dependency injection.

25.2. JFace default values for binding contexts

The binding context is identified via its ID. They can get assigned to a window or a part in the application model. This defines which keyboard shortcuts are valid for the window and which are valid for the part.

Eclipse JFace uses predefined IDs to identify binding contexts. These IDs are based on the `org.eclipse.jface.contexts.IContextIds` class. JFace distinguishes between shortcuts for dialogs, windows or both.

The following table gives an overview of the supported IDs and their validity.

Table 11. Default BindingContext values

Context ID	Description
org.eclipse.ui.contexts.dialogAndWindow	Key bindings valid for dialogs and windows
org.eclipse.ui.contexts.dialog	Key bindings valid for dialogs
org.eclipse.ui.contexts.window	Key bindings valid for windows

As an example, `Ctrl` + `C` (Copy) would be defined in *dialogAndWindows* as it is valid everywhere, but `F5` (Refresh) might only be defined for a Window and not for a Dialog.

The *BindingTable* node in the application model allows you to create key bindings based on a binding context. For this you create a new *BindingTable* model element and define a reference to the binding context via its ID.

In your key binding entry you specify the key sequence and the command associated with this shortcut.

The control keys are different for each platform, e.g., on the Mac vs. a Linux system. You can use Ctrl, but this would be hardcoded. It is better to use the M1 - M4 meta keys.

Table 12. Key mapping

Control Key	Mapping for Windows and Linux	Mapping for Mac
M1	Ctrl	Command
M2	Shift	Shift
M3	Alt	Alt
M4	Undefined	Ctrl

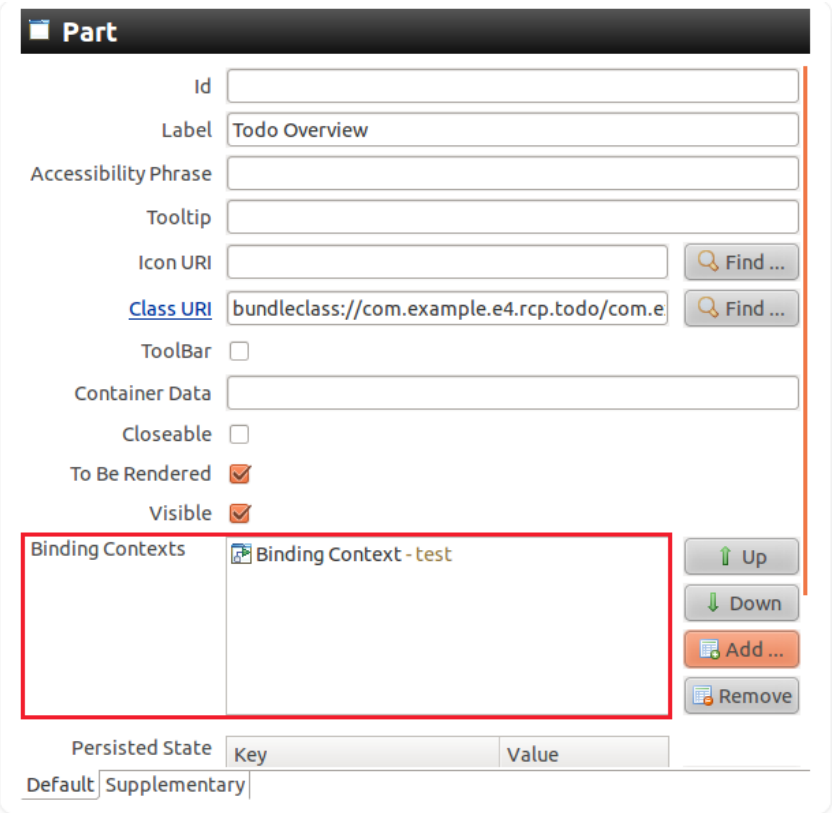
These values are defined in the `SWTKeyLookup` class.

25.4. Activate bindings

If there are several valid key bindings defined, the `ContextSet` class is responsible for activating one of them by default. `ContextSet` uses the binding context hierarchy to determine the lookup order. A binding context is more specific depending on how many ancestors are between it and a root binding context (the number of levels it has). The most specific binding context is considered first, the root one is considered last.

You can also use the `EContextService` service which allows you to explicitly activate and deactivate a binding context via the `activateContext()` and `deactivateContext()` methods.

25.5. Key bindings for a part

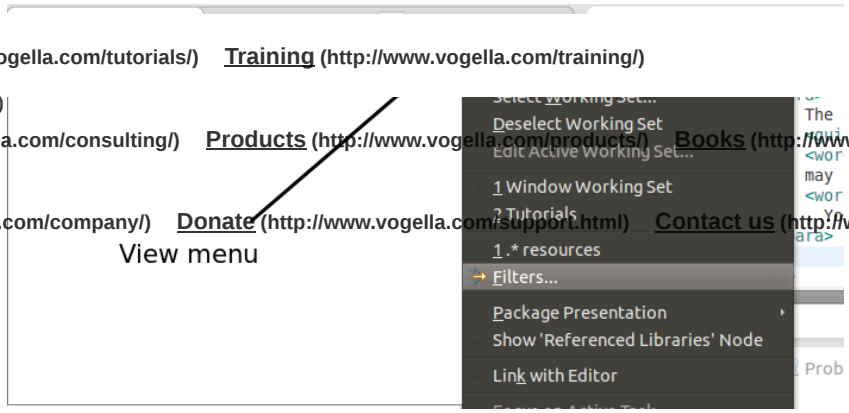


Key bindings assigned to a part are valid in addition to the key bindings provided by the currently active binding context, i.e. your global key bindings are still active in addition with the key bindings of the part.

26. Enable to start your product with right mouse click

You can also add the *pde nature* to your project in which you placed the product configuration file, if you want to be able to start your product via a right-click on the product and by selecting Run-as ▸ Eclipse Application.

The *Package Explorer* view may have a filter set for *.*resources*. You can modify this filter via the view menu as depicted in the following screenshot.



For this purpose remove the filter in the *Package Explorer* view for files starting with . (dot) and modify the *.project* file to the following.

```
XML
<projectDescription>
        <name>com.example.e4.rcp.todo.product</name>
        <comment></comment>
        <projects>
        </projects>
        <buildSpec>
                <buildCommand>

<name>org.eclipse.pde.ManifestBuilder</name>
                        <arguments>
                        </arguments>
                </buildCommand>
                <buildCommand>

<name>org.eclipse.pde.SchemaBuilder</name>
                        <arguments>
                        </arguments>
                </buildCommand>
        </buildSpec>
        <natures>
                <nature>org.eclipse.pde.PluginNature</nature>
        </natures>
</projectDescription>
```

27. 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 (http://www.vogella.com/books/eclipsercp.html).

28. About this website

29. Eclipse RCP resources

29.1. Eclipse Bug Tracker and Eclipse forum

Eclipse Forum for asking questions and providing feedback (http://eclipse.org/forums)

Eclipse Bug Tracker for reporting errors or feature requests (https://bugs.eclipse.org/bugs)

29.2. Eclipse RCP development resources

Eclipse 4 RCP Wiki (http://wiki.eclipse.org/Eclipse4/RCP)

Eclipse 4 RCP FAQ (http://wiki.eclipse.org/Eclipse4/RCP/FAQ)

vogella Eclipse Tutorials (http://www.vogella.com/eclipse.html)

Eclipse wiki with Eclipse 4 tutorials (http://wiki.eclipse.org/Eclipse4/Tutorials)

Eclipse 4 dependency injection wiki (http://wiki.eclipse.org/Eclipse4/RCP/Dependency_Injection)

Eclipse 4 RCP Wiki for tags for the application model (http://wiki.eclipse.org/Eclipse4/RCP/Modeled_UI/Tags)

Eclipse 4 Build Schedule (http://www.eclipse.org/eclipse/platform-releng/buildSchedule.html)

29.3. vogella GmbH training and consulting support

TRAINING (http://www.vogella.com/training/)	**SERVICE & SUPPORT** (http://www.vogella.com/consulting/)
The vogella company provides comprehensive training and education services (http://www.vogella.com/training/) from experts in the areas of Eclipse RCP, Android, Git, Java, Gradle and Spring. We offer both public and inhouse training. Whichever course you decide to take, you are guaranteed to	The vogella company offers expert consulting (http://www.vogella.com/consulting/) services, development support and coaching. Our customers range from Fortune 100 corporations to individual developers.

Appendix A: Copyright and License

Version 0.3
Last updated 2016-09-27 17:55:10 +02:00