

- [vogella.com](#)
- [Tutorials](#)

446

- [Contact us](#)

Free tutorial, donate to support

Donate



by Lars Vogel



BACK TO TOP

12 - Aug - 2013

Eclipse 4 RCP - Tutorial

second round use different color for all the commenting tools of adobe reader

4

Lars Vogel

Version 6.9

Copyright © 2009 , 2010 , 2011 , 2012 , 2013 Lars Vogel

04.07.2013

Revision History

| | | | |
|--------------------|-------------------------|------------|----------------------------|
| Revision 0.1 | 14.02.2009 | Lars Vogel | created |
| Revision 0.2 - 6.9 | 16.02.2009 - 04.07.2013 | Lars Vogel | bug fixes and enhancements |

Eclipse e4

This tutorial gives an overview about the Eclipse 4 application platform.

This tutorial describes the creation of Eclipse 4 based applications, e.g. Eclipse RCP applications. It describes the modeled application concept and the new programming model which is based on annotations and dependency injection.

Table of Contents

1. Eclipse 4

- 1.1. What is Eclipse 4?
- 1.2. Eclipse 4 vs. Eclipse 3.x
- 1.3. Terminology

2. The Architecture of Eclipse

2.1. Eclipse based applications

Training

Books

- 2.2. Core components of the Eclipse platform
- 2.3. Compatibility layer for Eclipse 3.x plug-ins
- 2.4. Eclipse RCP
- 2.5. Provisional API
- 2.6. Important configuration files

3. Tutorial: Install Eclipse IDE for RCP development

- 3.1. Prerequisites
- 3.2. Download and install Eclipse 4.3
- 3.3. Install the e4 tools
- 3.4. Official update site

4. Exercise: Wizard to create RCP application

- 4.1. Create project
- 4.2. Launch

5. The usage of run configurations

- 5.1. What are run configurations?
- 5.2. Launch configuration and Eclipse products
- 5.3. Launch arguments

6. Common launch problems

- 6.1. Launch problem number #1: missing plug-ins
- 6.2. Checklist for other common problems

7. Eclipse 4 application model

- 7.1. What is the application model?
- 7.2. ID and naming conventions
- 7.3. Limitations of the application model
- 7.4. How do you define the application model?
- 7.5. Using the e4 tools

8. User interface model elements

- 8.1. Windows
- 8.2. Views and editors - parts
- 8.3. Perspective
- 8.4. PartStacks and PartSashContainers
- 8.5. Using layout weight data for children elements

9. Connecting model elements to classes and resources

- 9.1. Connect model elements to classes
- 9.2. Connect model elements to resources
- 9.3. URI patterns
- 9.4. Model objects

9.5. Runtime application model

10. Model addons

- 10.1. Overview**
- 10.2. Framework Addons**
- 10.3. Additional SWT addons**
- 10.4. Relationship to other services**

11. Persisted model attributes

- 11.1. Tags**
- 11.2. Persisted State**
- 11.3. Transient data**

12. IDs and suggested naming conventions

- 12.1. Identifiers for model elements**
- 12.2. Best practices for naming conventions**

13. Features and Products

14. Tutorial: Create an Eclipse plug-in

- 14.1. Target**
- 14.2. Create a plug-in project**
- 14.3. Validate the result**

15. Exercise: From plug-in to Eclipse 4 application

- 15.1. Create product configuration file**
- 15.2. Create a feature project**
- 15.3. Enter feature dependencies in product**
- 15.4. Remove version dependency from features in product**
- 15.5. Create application model**
- 15.6. Add model elements to the application model**
- 15.7. Start application**

16. Enter bundle and package dependencies

- 16.1. Add plug-in dependencies**
- 16.2. Add package dependency**

17. Remove warnings for provisional API access

18. Configure the deletion of persisted model data

19. Exercise: Modeling a User Interface

- 19.1. Desired user interface**
- 19.2. Open the Application.e4xmi**
- 19.3. Add perspective**
- 19.4. Add PartSashContainer and PartStacks**
- 19.5. Create the Parts**
- 19.6. Create Java classes and connect to the**

model
19.7. Test

20. Tutorial: Using the SWT browser widget

20.1. Implementation
20.2. Solution

21. Introduction to dependency injection

22. Dependency injection and annotations

22.1. Define dependencies in Eclipse
22.2. On which objects does Eclipse perform dependency injection?
22.3. Re-injection

23. Objects available for dependency injection

23.1. Eclipse context (IEclipseContext)
23.2. Context relationship
23.3. How are objects selected for dependency injection
23.4. Default objects for dependency injection
23.5. Creation process of the Eclipse context
23.6. Tracking a child context with @Active

24. Behavior Annotations

24.1. API definition via inheritance
24.2. API definition via annotations

25. Tutorial: Using dependency injection

25.1. Getting a Composite
25.2. Validation

26. Exercise: Using @PostConstruct

26.1. Why using @PostConstruct?
26.2. Implement @PostConstruct
26.3. Implement @Focus and test your application
26.4. Validate

27. Menu and toolbar applicaton objects

27.1. Adding menu and toolbar entries
27.2. What are commands and handlers?
27.3. Mnemonics
27.4. Default commands
27.5. Naming schema for command and handler IDs

28. Dependency injection for handler classes

- 28.1. Behavior annotations for handler classes
- 28.2. Which context is used for a handler class?
- 28.3. Evaluation of @CanExecute
- 28.4. Scope of handlers

29. Tutorial: Defining and using Commands and Handlers

- 29.1. Overview
- 29.2. Defining Commands
- 29.3. Defining Handler classes
- 29.4. Defining Handlers in your model
- 29.5. Adding a Menu
- 29.6. Adding a Toolbar
- 29.7. Closing the application
- 29.8. Simulate save

30. View, popup and dynamic menus

- 30.1. View menus
- 30.2. Define popup menu (context menu)
- 30.3. Dynamic menu

31. Toolbars, ToolControls and drop-down tool items

- 31.1. Adding toolbars to parts
- 31.2. ToolControls
- 31.3. Drop-down tool items

32. More on commands and handlers

- 32.1. Passing parameters to commands
- 32.2. Usage of core expressions
- 32.3. Evaluate your own values in core expressions

33. Key bindings

- 33.1. Overview
- 33.2. BindingContext entries using by JFace
- 33.3. Define Shortcuts
- 33.4. Key Bindings for a Part
- 33.5. Activating Bindings
- 33.6. Issues with Keybinding

34. Application model modifications at runtime

- 34.1. Creating model elements
- 34.2. Modifying existing model elements

35. Example for changing the application model

- 35.1. Example: Dynamically create a new Window
- 35.2. Example: Dynamically create a new Part

36. Accessing and extending the Eclipse Context

- 36.1. Accessing the context**
- 36.2. OSGi services**
- 36.3. Objects and Context Variables**
- 36.4. Model Addons**
- 36.5. RunAndTrack**
- 36.6. Context Functions**

37. Using dependency injection for your own Java objects

- 37.1. Overview**
- 37.2. Using dependency injection to get your own objects**
- 37.3. Using dependency injection to create objects**

38. Relevant tags in the application model

39. Enable to start your product with right mouse click

40. Tips and tricks

- 40.1. Overview of all available annotations in Eclipse**
- 40.2. Meta-model of the application model**
- 40.3. Determine the command ID in a handler**
- 40.4. Live model editor**

41. Eclipse API best practice

- 41.1. Extending the Eclipse context**
- 41.2. Application communication**
- 41.3. Static vs. dynamic application model**
- 41.4. Component based development**
- 41.5. Usage of your own extension points**
- 41.6. API definition**
- 41.7. Packages vs. Plug-in dependencies**

42. Closing words

43. Thank you

44. Questions and Discussion

45. Links and Literature

- 45.1. Source Code**
- 45.2. Eclipse 4**

1. Eclipse 4

1.1. What is Eclipse 4?

The [Eclipse platform](#) and [IDE](#) is released every year. The last years the major version of Eclipse was 3, e.g. Eclipse 3.6, Eclipse 3.7. These releases and the corresponding API are referred to as Eclipse 3.x.

As of 2012 the main Eclipse release carried the major version number 4, e.g. Eclipse 4.2 in the year 2012 and Eclipse 4.3 in 2013. These releases and the [corresponding API](#) are referred to as Eclipse 4.

The [Eclipse 4 platform](#) is based on a [flexible](#) and [extendible programming](#) model. Eclipse 4 was an opportunity to rationalize the best parts of the Eclipse 3.x APIs and to fix pain points of Eclipse 3.x development.

1.2. Eclipse 4 vs. Eclipse 3.x

The major enhancements in Eclipse 4 compared to Eclipse 3.x are the following:

- the [structure on an Eclipse application](#) is described via a [logical model](#) called the application model
- the [application model can be modified at development and runtime](#)
- the application model can be [extended](#)
- the [programming model is based on dependency injection](#)
- the [styling of Eclipse widgets can be configured via external \(CSS like\) files](#)
- the [application model is decoupled from its presentation](#), this allows a flexible configuration of the user interface and to use different user interface [toolkits such as SWT](#) or [JavaFX](#)

1.3. Terminology

An Eclipse application consists of [several Eclipse components](#). A software component in Eclipse is called a [plug-in](#). A software component in OSGi is called a [bundle](#). Both terms can be used interchangeably.

This tutorial uses the terms *Eclipse based applications*, *Eclipse application*, *Eclipse 4 application* and *Eclipse RCP application* interchangeably for referring to an application which is based on the Eclipse 4 framework.

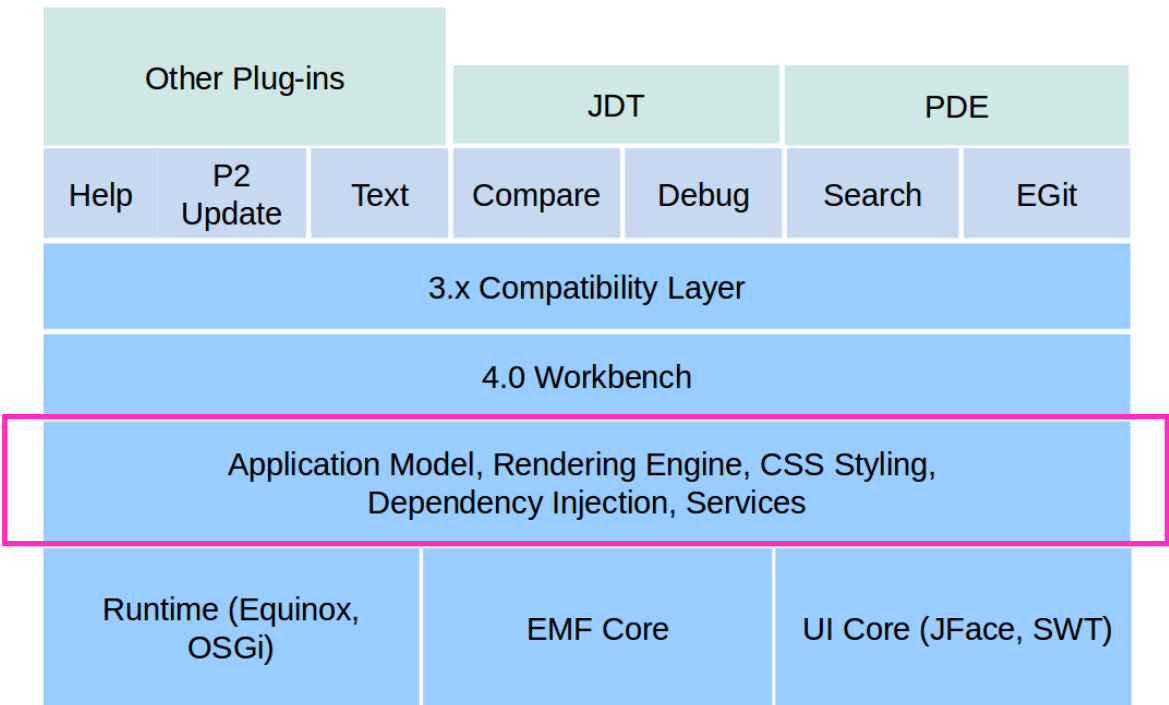
If a certain concept refers to Eclipse 3.x, then it is explicitly stated.

2. The Architecture of Eclipse

2.1. Eclipse based applications

An Eclipse application consists of individual software components. The Eclipse IDE can be viewed as a special Eclipse application with the focus on supporting software development.

The core components of the Eclipse IDE are depicted in the following graphic.



Note

The intension of the graphic is to demonstrate the general concept, the displayed relationships are not 100 % accurate.

The most important components of this graphic are described in the next section.

2.2. Core components of the Eclipse platform

OSGi is a **specification** which **describes a modular approach for Java application**. 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. The Equinox runtime provides the necessary 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**. The workbench is responsible for displaying all other UI components.

so, i need Workbench to build my desktop application.

On top of these base components, the Eclipse IDE adds components which are important for an IDE application, for example the Java Development Tools (JDT) or version control support (EGit).

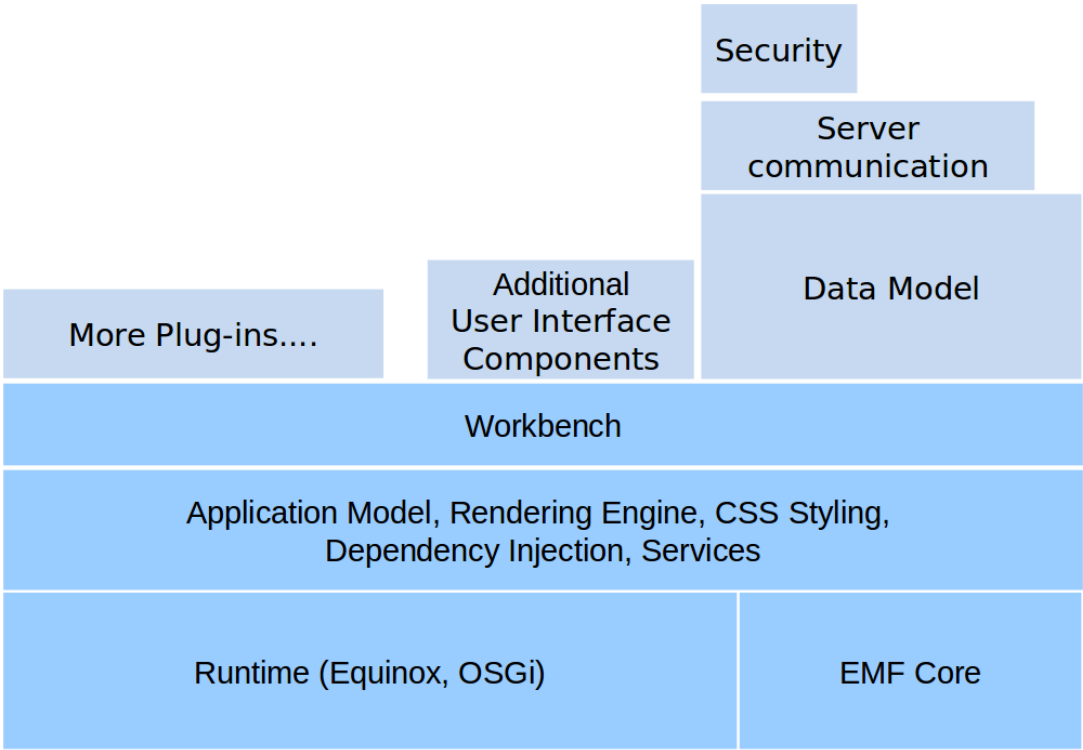
2.3. Compatibility layer for Eclipse 3.x plug-ins

Eclipse 4 provides a compatibility layer which allows that plug-ins using the Eclipse 3.x programming model can be used unmodified in an Eclipse based application.

Most plug-ins available for the Eclipse IDE are still based on the Eclipse 3.x programming model. These plug-ins use the compatibility layer to function in the Eclipse IDE.

2.4. Eclipse RCP

Eclipse based applications which are not primarily used as software development tools are called Eclipse RCP applications. An Eclipse 4 RCP application typically uses the base components of the Eclipse platform and adds additional application specific components.



Note

This tutorial focuses on the development of Eclipse RCP applications. The development of Eclipse plug-ins for the IDE is not covered even though some of the techniques are very similar.

2.5. Provisional API

Currently the Application Programming Interface (API) for Eclipse 4.3 is partially released.

Some API is still marked as provisional. This means that the API might be changed 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.

Note

If you use not released API you see a *Discouraged access* warning in the

Java editor.

2.6. Important configuration files

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 - Contains information about Eclipse specific extension mechanisms

An Eclipse plug-in defines its API and its dependencies via the *MANIFEST.MF* file, e.g. the Java packages which can be used by other plug-ins and its dependencies, e.g. the packages or plug-ins which are required by the plug-in.

The *plugin.xml* file provides the possibility to define additional Eclipse specific API. You can define 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.

Note

In Eclipse 4 the usage of extension points and extensions is less important than in Eclipse 3.x. Several extension points have been replaced with the usage of the application model.

3. Tutorial: Install Eclipse IDE for RCP development

3.1. Prerequisites

The following assumes that you have Java installed in at least version 1.6.

3.2. Download and install Eclipse 4.3

The description in this tutorial is based on the Eclipse 4.3 release. Download the latest available official *Eclipse SDK* build from the following URL.

<http://download.eclipse.org/eclipse/downloads/>

Eclipse Project 4.x Stream Downloads

Latest downloads from the Eclipse project



Support Eclipse! Become a friend.

Latest Downloads

On this page you can find the latest builds produced by the Eclipse Project. To get started run the program and go through the user and developer documentation provided in the online help system. If you have problems downloading the drops, contact the webmaster. If you have problems installing or getting the workbench to run, check out the Eclipse Project FAQ, or try posting a question to the newsgroup. All downloads are provided under the terms and conditions of the Eclipse Foundation Software User Agreement unless otherwise specified.

Eclipse 3.x downloads are available.

See the main Eclipse download site for other packages and projects.

Help out with Eclipse translations - check out the Babel project.

If you prefer, try downloading with the SDK Torrents

See also the build schedule, read information about different kinds of builds, access archived builds (including language packs), or see a list of p2 update sites.

| Build Type | Build Name | Build Status | Build Date |
|-----------------------|----------------|-----------------------|-----------------------------------|
| Latest Release | 4.3 | Ju (3 of 3 platforms) | Wed, 5 Jun 2013 -- 20:00 (-0400) |
| 4.4 Integration Build | I20130625-0800 | Ju (3 of 3 platforms) | Tue, 25 Jun 2013 -- 08:00 (-0400) |
| 4.4 Nightly Build | N20130629-1500 | Ju (3 of 3 platforms) | Sat, 29 Jun 2013 -- 15:00 (-0400) |

The download is a zip file, which is a compressed archive of multiple files. Most operating systems can extract zip files in their file browser. For example if you are using Windows7 as the operating system, right-click on the file in the explorer and select the *Extract all...* menu entry. If in doubt about how to unzip, search via Google for *How to unzip a file on ...*, replacing "... " with your operating system.

Warning

Do not extract Eclipse to a directory with a path which contains spaces as this might lead to problems in the usage of Eclipse. Also avoid path names longer than 255 characters as this seems to create problems under Microsoft Windows.

After you extracted the zip file, double-click the `eclipse.exe` (Windows) or the `eclipse` file (Linux) (or the launcher icon specific to your platform) to start Eclipse.

To avoid any collision with existing work select an empty directory as the workspace for this tutorial.

3.3. Install the e4 tools

The Eclipse SDK download does not include the **e4** tools, which makes creating Eclipse 4 applications easier. These tools provide wizards to create Eclipse 4 artifacts and the specialized model editor for the application model.

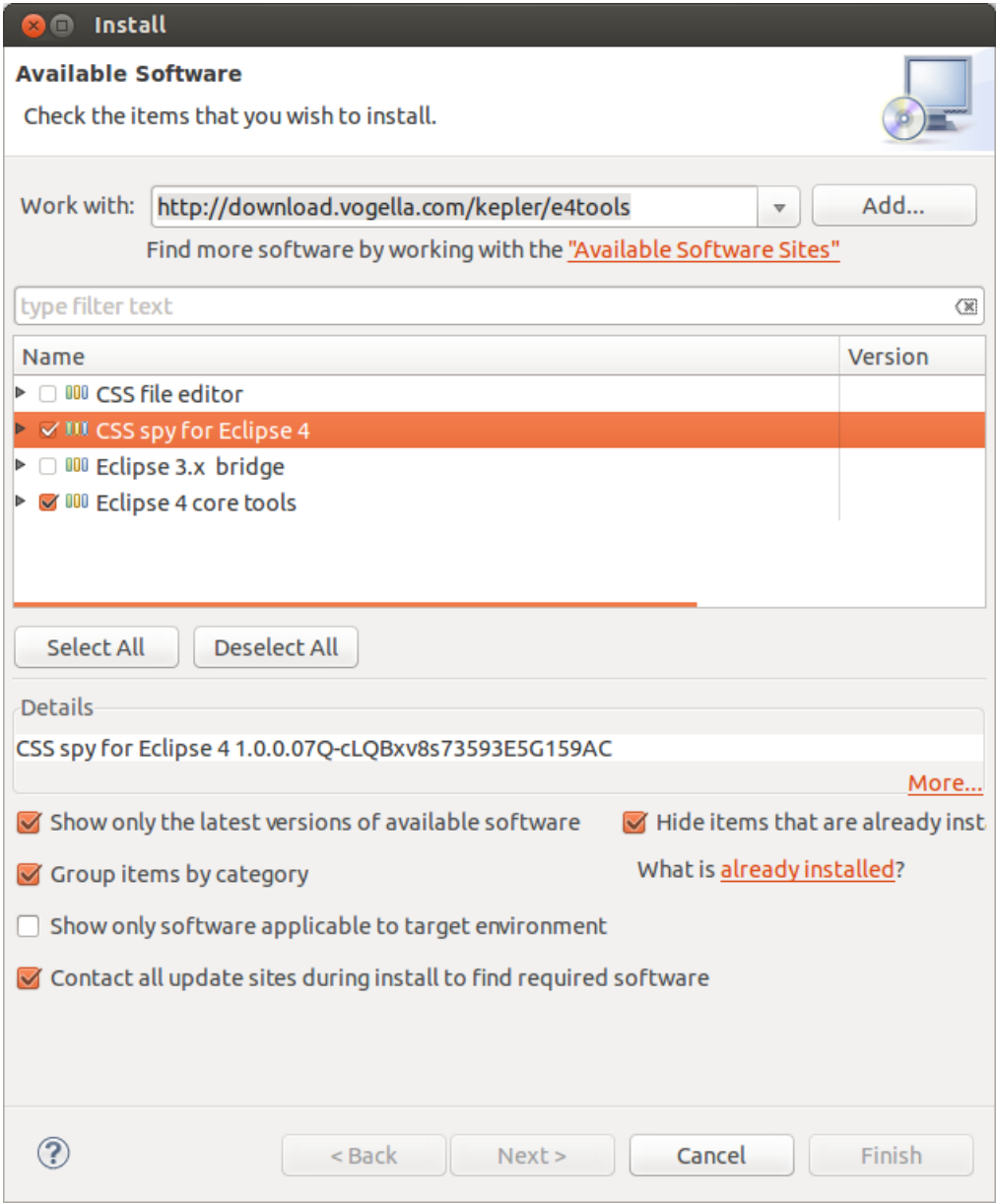
~~The author of this documentation provide a working and recent version of the e4 tools for the Eclipse 4.3 release under the following URL.~~

~~<http://download.vogella.com/kepler/e4tools>~~

~~You can install the tools via *Help* → *Install new software* and by entering the URL.~~

Note

From this update site, install only the E4 CSS Spy and the Eclipse e4 Tools. The other entries are not used in this tutorial and may cause different behavior.



Tip

If you want to use Eclipse 4.4 (which is as of the time of this writing under development) you can use the following update site:
`http://download.vogella.com/luna/e4tools`

3.4. Official update site

The above update site was created to provide a stable link for the reader of the tutorial. The same code basis is used by Eclipse.org to create an official update site for the *Eclipse e4 tooling*.

Unfortunately the link for this update site changes from time to time but it can be found on the following website: [Eclipse.org e4tools site](#).

If you click on a *Build Name* link you find also the URL for the update site. The following screenshots demonstrate this for a particular build of the e4 tools.

eclipse e4 project downloads

latest downloads from the eclipse e4 project

Latest Downloads

On this page you can find the latest e4 [builds](#). For more information on installing and using any of the builds that you find

All downloads are provided under the terms and conditions of the [Eclipse Foundation Software User Agreement](#) unless ot

Other [eclipse.org](#) project downloads are available [here](#).

| Build Type | Build Name |
|-------------------------------|--------------------------------|
| Release Build | 0.13 |
| 0.14 Stream Stable Build | |
| 0.14 Stream Integration Build | I20130407-2200 |
| 0.13 Stream Maintenance Build | |

Release Builds

| Build Name | Build Date |
|-----------------------|-----------------------------------|
| 0.13 | Thu, 28 Feb 2013 -- 22:00 (-0500) |
| 0.12 | Wed, 13 Jun 2012 -- 11:00 (-0400) |
| 0.112 | Thu, 23 Feb 2012 -- 09:00 (-0500) |
| 0.111 | Mon, 12 Sep 2011 -- 15:10 (-0400) |
| 0.11 | Mon, 20 Jun 2011 -- 16:31 (-0400) |
| 0.10 | Tue, 27 Jul 2010 -- 14:15 (-0400) |

Click here

Note

This website might change over time.

Eclipse e4

Eclipse SDK 4.2

Eclipse SDK 3.8

Release Build: 0.13

201302282200. Built against Eclipse 4.2 SDK These downloads are provided under the [Eclipse Foundation Software User Agreement](#).

The page provides access to the various sections of this build along with details relating to its results. Test results are provided below and performance results are posted once they are available. You may access the download page specific to each platform by selecting one of the tabs in the platform navigator above.


A list of pre-requisite components that you need to run e4. If you install e4 from the update site, the pre-requisites will be installed automatically:
E4 runs on the Eclipse 4.2 SDK or compatible.

Modeled UI and CSS

→ EMF runtime 2.7

Programming Language Support - JavaScript

→ WST SDK @wtpBuildId@ - JSDT
See [E4/JavaScript](#) for details on using the Rhino Debugging Support with Eclipse 3.6

**Download now: Eclipse e4**


To download a file via HTTP click on its corresponding http link below.

Related Links

- View the **compile logs** for the current build.
- View the **build logs** for the current build.
- View the **test results** for the current build.
- View the **map file entries** for the current build.

Source Builds

- Access the **Source Builds** page. under construction

| Eclipse e4 | | | | |
|---|--------------------------|----------|----------|-------------------------------------|
| Status | Platform | Download | Size | File |
|  | All (Supported Versions) | (http) | 11314327 | eclipse-e4-repo-incubation-0.13.zip |

Comments
[online p2 repo link](#)

URL for the update site

4. Exercise: Wizard to create RCP application

4.1. Create project

n:

Select *File* → *New* → *Other...* → *Eclipse 4* → *Eclipse 4 Application Project* from the Eclipse menu.

Create a project called `com.example.e4.rcp.wizard`. Leave the default settings on the first two wizard pages. These settings are similar to the following screenshots.

file:///G:/0.%20Tech%20Books/1.%20Java%20and%20JEE%20Programming/11.%20Java%20Desktop%20Programming/1.%20SWT%20-%20RCP%20-%20Eclipse/Eclipse%204%20RCP%20-%20Tutorial.html[8/12/2013 7:33:22 PM]

New Plug-in Project

Plug-in Project

Create a new plug-in project

Project name:

com.example.e4.rcp.wizard

☒ Use default location

Location:

/home/vogella/workspace/stuttgart/com.example.e4.rcp.w

Browse...

Project Settings

☒ Create a Java project

Source folder:

src

Output folder:

bin

Target Platform

This plug-in is targeted to run with:

☐ Eclipse version:

3.5 or greater

☒ an OSGi framework:

Equinox

Working sets

☐ Add project to working sets

Working sets:

Select...

?

< Back

Next >

Cancel

Finish

file:///G:/0.%20Tech%20Books/1.%20Java%20and%20JEE%20Programming/11.%20Java%20Desktop%20Programming/1.%20SWT%20-%20RCP%20-%20Eclipse/Eclipse%204%20RCP%20-%20Tutorial.html[8/12/2013 7:33:22 PM]

New Plug-in Project

Content

Enter the data required to generate the plug-in.

Properties

ID:

com.example.e4.rcp.wizard

Version:

1.0.0.qualifier

Name:

Wizard

Vendor:

EXAMPLE

Execution Environment:

JavaSE-1.7

Environments...

Options

☐ Generate an activator, a Java class that controls the plug-in's life cycle

Activator: com.example.e4.rcp.wizard.Activator

☐ This plug-in will make contributions to the UI

☐ Enable API analysis

?

< Back

Next >

Cancel

Finish

On the third wizard page, enable the *Enable development mode for application model* and *Create sample content (parts, menu etc.)* flags.

New Plug-in Project

Eclipse 4 Application

Configure application with special values.

Product

Name:* com.example.e4.rcp.wizard

Properties

CSS Style:

css/default.css

Preference Customization:

☒ Enable development mode for application model

Template option

☒ Create sample content (parts, menu etc.)

?

< Back

Next >

Cancel

Finish

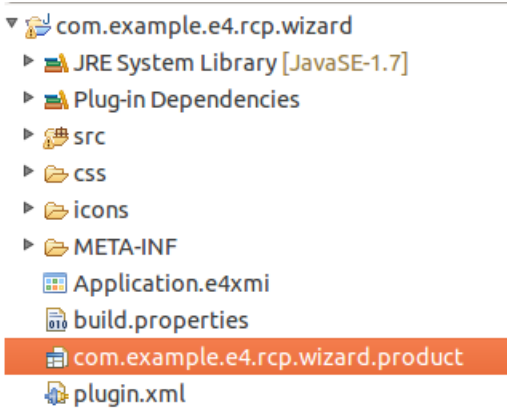
Note

The *development mode* flag adds the `clearPersistedState` flag to the product configuration file. This ensures that changes during development in your application model are always visible. See **Section 18, “Configure the deletion of persisted model data”** for more information. Via the *sample content* flag you define if a minimal Eclipse RCP application is generated or if the generated application contains example content, e.g. a *view* and some menu and toolbar entries.

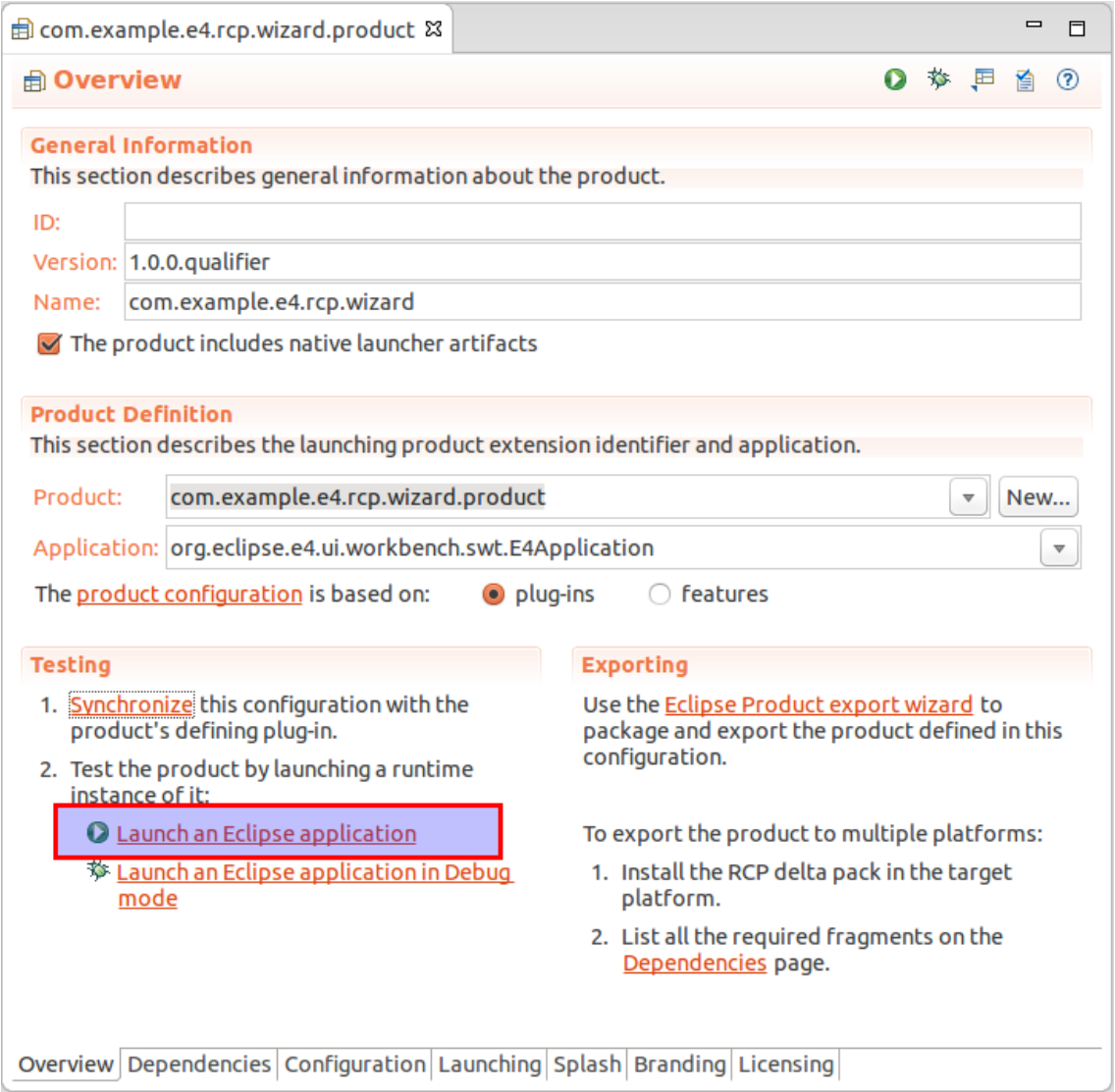
This wizard creates all the necessary files to start your application. The central file for starting your application is the [.product](#) file, created in your project folder.

4.2. Launch

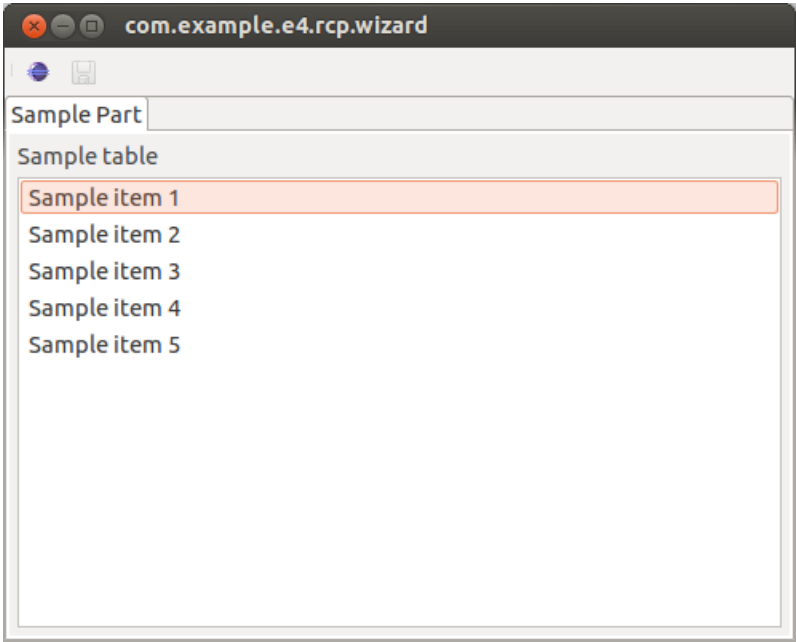
Open the editor for your `com.example.e4.rcp.wizard.product` product configuration file by double-clicking on the file in the *Package Explorer* view.



Switch to the *Overview* tab in the editor and launch your Eclipse application by pressing the *Launch an Eclipse application* hyperlink.



This starts your Eclipse application which should look similar to the following screenshot.



Note

You learn all the details of what happened here in later chapters.

5. The usage of run configurations

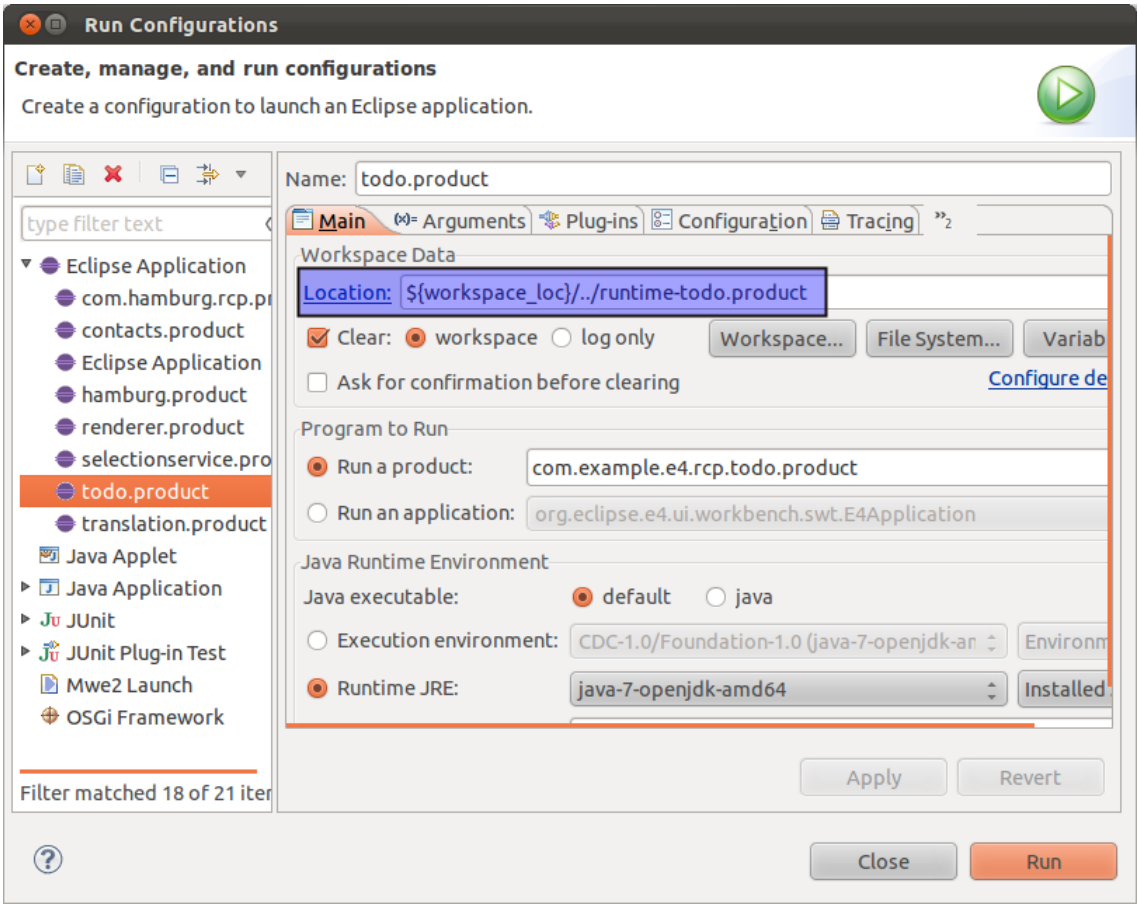
5.1. What are run configurations?

A *run configuration* defines the environment where a generic launch will be executed from within. For example it defines arguments to the Java virtual machine (VM), plug-in (classpath) dependencies etc. Sometimes a *run configuration* is called *launch configuration*.

If you start your Eclipse application, using the link in the product [file, the corresponding run configuration is automatically created or updated.](#)

To review and edit your run configurations select *Run* → *Run Configurations...* from the Eclipse menu.

On the *Main* tab in the field *location* you specify where the Eclipse IDE will create the files necessary to start your Eclipse based application.



5.2. Launch configuration and Eclipse products

The launch configuration persists the settings from the product configuration file. The launch configuration is created or updated every time you start your application via the product.

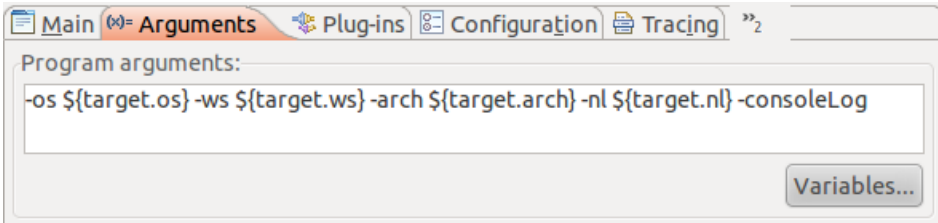
You can use the created run configuration directly for starting the application again. In this case changes in the product configuration file are not considered.

Warning

Using an existing run configuration is a common source of frustrating and time consuming error analysis. To ensure that you use the latest configuration from your product, start via the product configuration file.

5.3. Launch arguments

The run configuration allows you to add additional start arguments for your application on the *Arguments* tab. By default Eclipse includes already some arguments, e.g. parameters for -os, ws and -arch to specify the architecture on which the application is running.



The following table lists useful launch arguments.

Table 1. Launch parameters

| Parameter | Description |
|----------------------------|---|
| <i>consoleLog</i> | Error messages of the running Eclipse application are written to the Eclipse IDE <i>Console view</i> started by this application. |
| <i>nl</i> | Specifies the runtime language for your application. For example <code>-nl en</code> starts your application using the English language. This is useful for testing translations. |
| <i>console</i> | Provides access to an OSGi console where you can check the status of your application. |
| <i>noExit</i> | Keeps the OSGi console open even if the application <u>crashes. This allows to analyze the application dependencies even if the application crashes during startup.</u> |
| <i>clearPersistedState</i> | Deletes <u>runtime changes of the Eclipse 4 application model.</u> |

6. Common launch problems

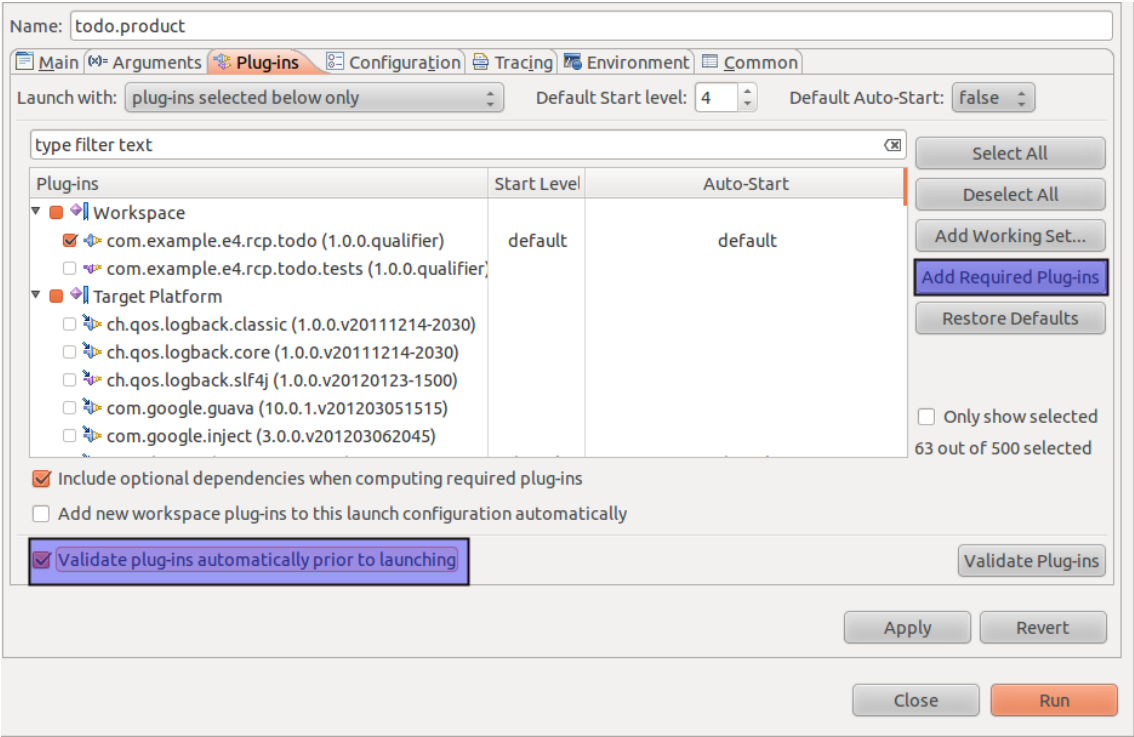
Error in the run configurations of Eclipse RCP application are frequently the source of problems. This section describes common problems related to the start of RCP applications. It can be used as a reference in case you face issues during the startup of your application.

6.1. Launch problem number #1: missing plug-ins

The most common problem is that some required plug-ins are missing in your product. If you are using a feature based product configuration you need to ensure that all plug-ins which are referred to in the MANIFEST.MF file are also included in your features.

Eclipse can check for missing dependencies automatically for you before every start. On the *Plug-ins* Tab select the *Validate plug-ins automatically prior to launching* option. This will check if you have all the required plug-ins in your run configuration.

If this check reports that some plug-ins are missing, try clicking the *Add Required Plug-ins* button.



After identifying the missing plug-ins ensure that you add them to your product (if the product is **plug-in base**) or to your features (if the product **is feature based**).

Warning

Never fix problems with plug-in only in the run configuration. The run configuration is created and updated based on the product configuration file. Ensure that the product file is correctly configured.

6.2. Checklist for other common problems

The following table lists potential problems and solutions.

Table 2. Run Configuration problems

| Problem | Investigate |
|---|--|
| During start you get error messages such as "One or more bundles are not resolved because the following root constraints are not resolved" or "java.lang.RuntimeException: No application id has been found." | <p>Check that all required plug-ins are included in your run configuration. Make sure that your product defines dependencies to all required plug-ins or features.</p> <p>Bundles may also require a certain version of the Java virtual machine, e.g. a bundle may require Java 1.6 and will therefore not load in a Java 1.5 VM. Check the <i>MANIFEST.MF</i> file on the <i>Overview</i> tab in the <i>Execution Environments</i> section which Java version is required.</p> |
| Strange behavior but no error message. | Check if your run configuration includes the <i>-consoleLog</i> parameter. This option allows you to see errors from Eclipse based application in the console view of the Eclipse IDE. |
| Runtime configuration is frequently missing required plug-ins | Make sure that your product or your feature(s) includes all required dependencies. |
| A change in the product dependencies, e.g. a new plug-in is added but is not included in the run configuration. | A product updates <u>an existing run configuration if you start the product directly from the product definition file. If you select the run configuration directly it will not be updated.</u> |
| Application model changes are not reflected in the Eclipse 4 application. | Eclipse 4 persists user changes in the application model. During development this might lead to situations where model changes are not correctly applied to the runtime model, e.g. you define a new menu entry and this entry is not displayed in your application. Either set the <i>Clear</i> flag on the <i>Main</i> tab in your Run configuration or add the - |

| | |
|--|--|
| | <i>clearPersistedState</i> parameter for your product configuration file or Run configuration. |
| Services, e.g. key bindings or the selection service, are not working in an Eclipse 4 application. | Ensure that every <i>part</i> correctly implements <code>@Focus</code> . Eclipse 4.2 requires that one control get the focus assigned. Eclipse 4.3 has solved this. |
| Menu entries are disabled in an Eclipse 4.3 build. | Eclipse 4.3 introduces a new model add-on which you need to register with your application model. If you are using Eclipse 4.3 ensure that your application model has an entry pointing to in the <code>HandlerProcessingAddon</code> class in the package <code>org.eclipse.e4.ui.internal.workbench.addons</code> . The bundle symbolic name is <code>org.eclipse.e4.ui.workbench</code> . |
| Application "org.eclipse.ant.core.antRunner" could not be found in the registry or Application could not be found in the registry. | Ensure that you pressed the <i>New</i> button in the product configuration file and selected the <code>E4Application</code> as application to start. You can check the current setting in your <i>plugin.xml</i> file on the <i>Extensions</i> tab and in the details of the <i>org.eclipse.core.runtime.products</i> extension. |

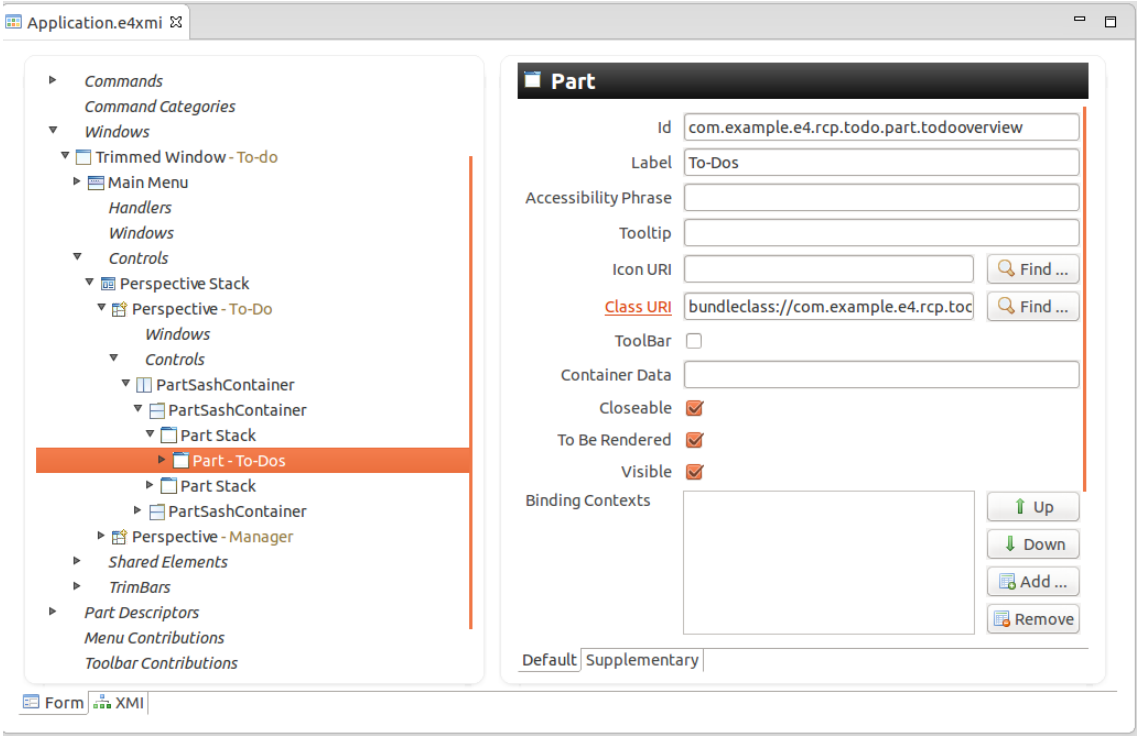
7. Eclipse 4 application model

7.1. What is the application model?

Eclipse 4 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 Eclipse 4 application**.

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

The following screenshot shows an example application model opened in **the *e4 tools editor***.



Each model element has attributes which describe its current state, e.g. the size and the position for a *Window*. Model elements might be in a hierarchical order, for example *parts* might be grouped below a *perspective*.

7.2. ID and naming conventions

Each model element should have a unique ID assigned.

A good convention is to start IDs with the *top level package name* of your project and to use only lower case. After the top level package name typically a group descriptor is also used. For example *com.example.e4.rcp.todo.part.todooverview*, where *com.example.e4.rcp.todo* is the top level package, *part* is group descriptor for all visible parts (Views and Editors) in your application and *todooverview* gives an idea about the purpose of this part.

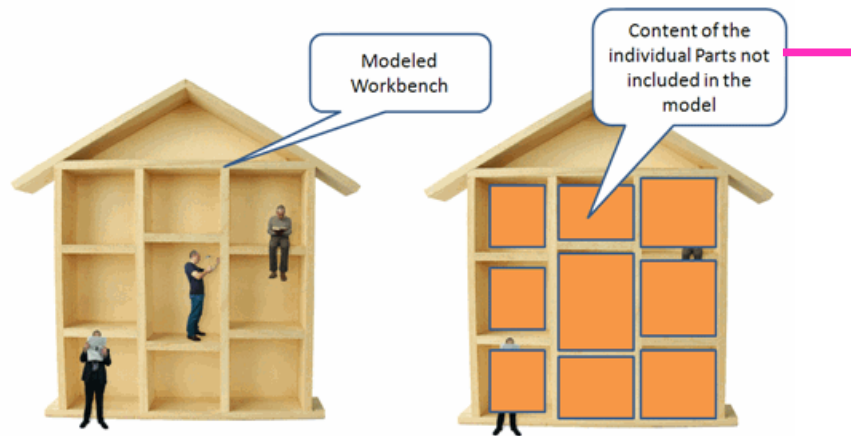
7.3. Limitations of the application model

The application model defines the structure of the application; it does not describe the content of the individual user interface components.

For example the application model describes which *parts* are available. It also describes the *parts*

properties, e.g. if a *part* is closable, its label, ID, etc. But it does not describe the content of the *part*, e.g. the labels, text fields and buttons it consists of. 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*, *PastStacks*, *PartSashContainer*) but not the furniture of the rooms. This is illustrated by the following image.



7.4. How do you define the application model?

The basis of the application model is typically defined as a static file. The default name for this file is `Application.e4xmi` and the default location is the main directory of your application plug-in.

Tip

You can change the default name and location via the `org.eclipse.core.runtime.products` extension point. Via the `applicationXMI` parameter you define the URI to the application model file. See ??? for a detailed description of this procedure.

This XMI file is read at application startup and the initial application model is constructed from this file.

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

7.5. Using the e4 tools

The e4 tools provide a wizard to create a new project which includes an application model file and also an editor to work on the application model. The usage of the e4 tools is described in the various exercises in this tutorial.

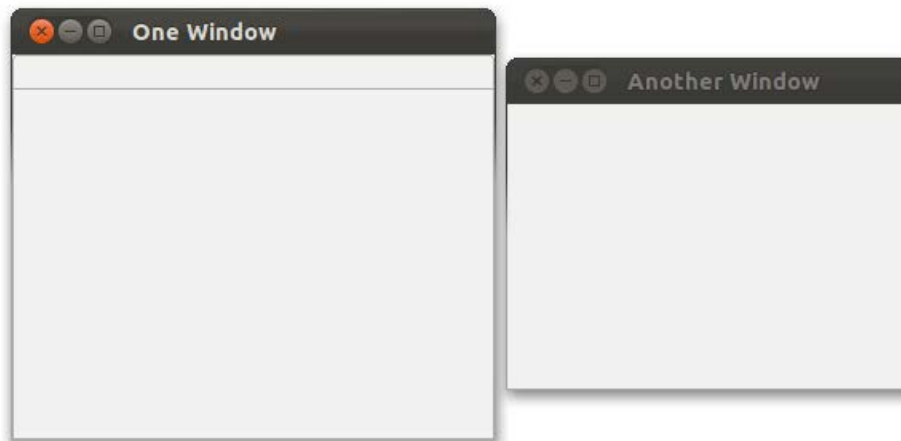
You installed these tools in Section 3.3, “Install the e4 tools”.

8. User interface model elements

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

8.1. Windows

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 displays for two connected monitors.



8.2. Views and editors - parts

Parts are user interface components which allow you to navigate and modify data. A *part* can have a menu and a toolbar. *Parts* are typically divided into views and editors.



The distinction into *views* and *editors* is not based on technical differences, but on a different concept of using and arranging these *parts*.

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 us to open an *editor* for a selected set of data.

An example for a *view* is the *Package Explorer*, which allows you to browse the files of Eclipse projects. If you change data in the *Package Explorer*, e.g. renaming a file, the file name is directly changed on the file system.

Editors are typically used to modify a single data element, e.g. 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.

Editors and views can be freely positioned in the user interface.

For example the *Java editor* is used to modify Java source files. Changes to the source file are applied once the user selects the *Save* command. A dirty editor is marked with an asterisk.



8.3. Perspective

A *perspective* is a visual container for a set of *parts*. You can stack *perspectives* in a perspective stack.

Switching *perspectives* can be done via the *EPartService* service. This service is covered later.

8.4. PartStacks and PartSashContainers

Parts can be directly assigned to a *Window* or a *Perspective* but typically you want to group and arrange them. in my adagu application parts added to window itself

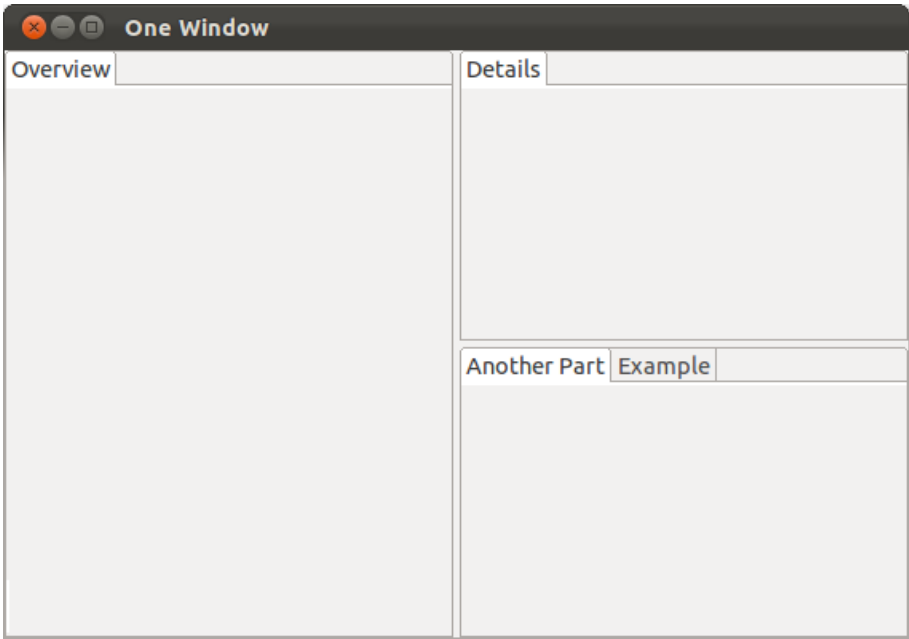
For this you can use the *PartStacks* and *PartSashContainers* model elements.

PartStacks contain a stack of *parts*. A *PartStack* shows the headers for all *parts* it contains. One *part* is active and the user can switch to another *part* by selecting the corresponding tab.

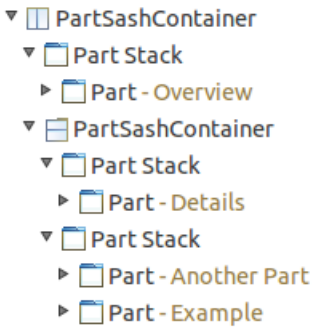
A *PartSashContainer* displays all its children at the same time either horizontally or vertically.

The following shows a simple Eclipse application layout using two *PartSashContainers* and a few

PartStacks.



On the top of this layout there is a horizontal *PartSashContainer* which contains another *PartSashContainer* and some *PartStacks*. The hierarchy is depicted in the following graphic.

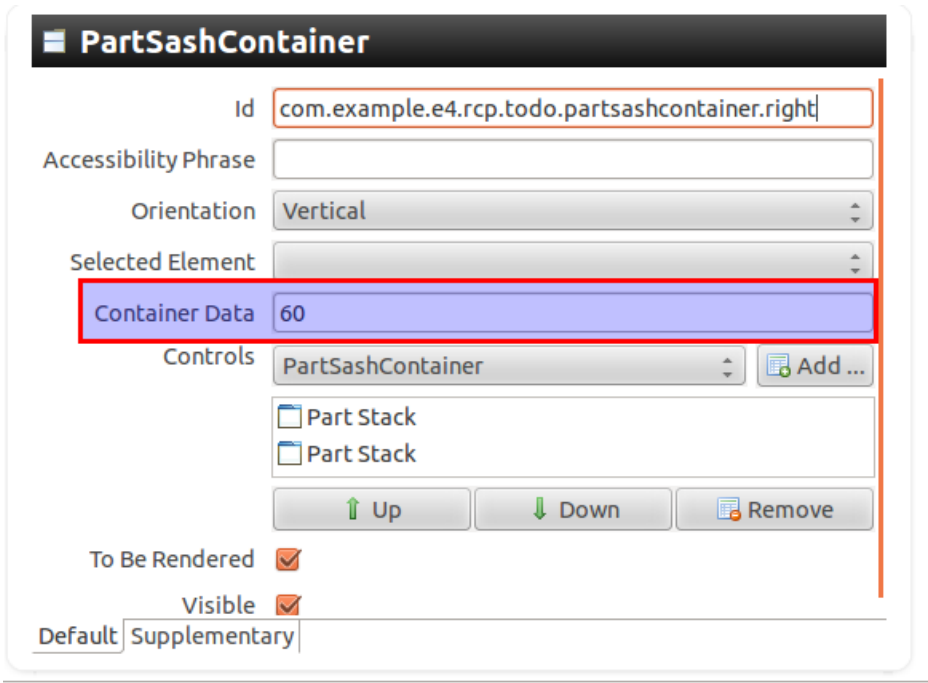


8.5. Using layout weight data for children elements

You can use the *Container Data* attribute on a children of a *PartSashContainer* to assign a layout weight.

This layout weight is interpreted as the relative space the corresponding child element should get assigned in the *PartSashContainer*.

The setting is depicted in the following screenshot.



Warning

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

9. Connecting model elements to classes and resources

9.1. Connect model elements to classes

Certain model elements in the application model can contain references to Java classes via an Uniform Resource Identifier (URI).

The URI describes the location of the Java class. The first part of this URI is the plug-in, the second one the package and the last one the class.

It contains a *Class URI* which points to a Java class for this element. This class provides the behavior of the *part* — using the house/rooms metaphor from earlier, the class is responsible for defining the furnishings and the layout of the room, and how the interactive objects behave.

Part

Id

Label

Sample Part

Accessibility Phrase

Tooltip

Icon URI

Find ...

Class URI

bundleclass://com.example.e4.rcp.wizard/com.example.e4.rcp.wizard.parts.SamplePart

Find ...

ToolBar

☐

Container Data

Closeable

☐

To Be Rendered

☒

Visible

☒

Binding Contexts

Up

Down

Add ...

Remove

Persisted State

| Key | Value |
|-----|-------|
|-----|-------|

Add ...

Reset

Default

Supplementary

Eclipse instantiates the referred classes lazily, if the model elements get activated this class will get instantiated. For example the objects for parts are only created once the part becomes visible.

9.2. Connect model elements to resources

Model elements can also point to static resources. For example the *part* model element contains the attribute *icon URI* which can point to an icon which is used by the Eclipse platform once it shows the

part.

9.3. URI patterns

URIs follow one of two patterns, one for identifying *classes* and another one for identifying *resources*. The following table describes these two patterns. The example assumes that the bundle is called *test*.

Table 3. URI pattern

| Pattern | Description |
|---|---|
| <div>bundleclass://Bundle-SymbolicName/ package.classname</div> <div>Example:</div> <div>bundleclass://test/test.parts.MySavePart</div> | Used to identify Java classes. It consists of the following parts: "bundleclass://" is a fixed schema, <u>Bundle-SymbolicName as defined in the <i>MANIFEST.MF</i> file</u> , and the fully qualified classname. |
| <div>platform:/plugin/Bundle-SymbolicName/ path/filename.extension</div> <div>Example:</div> <div>platform:/plugin/test/icons/save_edit.gif</div> | Identifier for a resource in the plug-in. "platform:/plugin/" is a fixed schema, followed by the Bundle-SymbolicName of the <i>MANIFEST.MF</i> file, followed by the path to the file and the filename. |

9.4. Model objects

The attributes of the application model elements are stored in Java objects at runtime. These objects are called model objects in this tutorial.

You can use these model objects to change its attributes or children. The Eclipse platform has change listeners registered on the model objects and updates the user interface whenever you change relevant attributes.

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

Table 4. Eclipse model elements

| Model element | Description |
|---------------|--|
| MApplication | Describes the application object. Can be used for example to add new windows to your application |
| | |

| | |
|-----------------|--|
| MAddon | <u>Non visual component. It typically registers for events in the application lifecycle and handles these events.</u> |
| MWindow | Represents a Window in your application. |
| MTrimmedWindow | Represents a Window in your application. The underlying SWT shell has been created with the SWT.SHELL_TRIM attribute which means, it has a tile, a minimize, maximize and resize button. |
| MPerspective | Object for the perspective model element. |
| MPart | Represents the model element part, e.g. a View or a 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 trigger a save. |
| MPartDescriptor | MPartDescriptor is a template for new Parts. You define in your application model a PartDescriptor. A new Part based on this PartDescriptor can be created via the EPartService and shown its showPart () method. |
| Snippets | Snippets can be used to <u>pre-configure model parts</u> which you want to create via your program. You can use <code>EcoreUtil.copy</code> to copy a Snippet and assign it to another model element, e.g. on a <code>MSash</code> you can add a newly copied <code>MStack</code> called <code>copy</code> , via <code>getChildren().add(copy)</code> . Also the model service has some methods to create and clone a snippet. |

Tip

Access to these model objects is done via dependency injection which is covered in **Section 22, “Dependency injection and annotations”**.

9.5. Runtime application model

During startup the Eclipse platform creates the model objects based on the `Application.e4xmi` and instantiates the referred classes in the model if required.

The created set of model objects is typically referred too as *runtime application model*.

The life cycle of every model object and the objects created based on the `class URI` attributes are therefore controlled by the Eclipse platform.

10. Model addons

10.1. Overview

The application model allows you to globally register *addons* model objects. These *addons* can enhance the application with additional functionality.

Addons point to Java classes via their *Class URI* attribute using the `bundleclass:// URI` convention.

Addons add flexibility to the application model. For example the default Drag and Drop support of *parts* in Eclipse is implemented via an *addon*. Other examples are the keybinding or the command processing. *Addons* allow you to extend or change the default Eclipse behavior without having to modify existing Eclipse code, you just have to replace the related *addons*.

10.2. Framework Addons

Currently the following standard *addons* are useful for Eclipse applications. Their class names give an indication of their provided functionality. Check their Javadoc to get a short description of their purpose.

- CommandServiceAddon
- ContextServiceAddon
- BindingServiceAddon
- HandlerProcessingAddon
- CommandProcessingAddon
- ContextProcessingAddon
- BindingProcessingAddon

10.3. Additional SWT addons

Additional *addons* are available, e.g. to support drag-and-drop of *parts* in your application.

To support drag-and-drop for *parts* you need to add the `org.eclipse.e4.ui.workbench.addons.swt` plug-in to your product configuration file. Then you can use the `DnDAddon` and the `CleanupAddon` from this bundle as *addons* in your application model. This plug-in contains also the `MinMax` add-on which adds the minimize and maximize functionality to

your application.

The `org.eclipse.e4.ui.workbench.addons.swt` plug-in contributes these plug-ins to your application model via a processors, e.g. a Java class which changes the application model. If you remove the plug-in from your product then these *addons* are not available.

10.4. Relationship to other services

Addons are created before the *rendering engine* renders the model and after the `EventAdmin` service has been created.

This allows *addons* to alter the user interface that is produced by the rendering engine. For example, the min/max *addon* changes the tab folders created for `MPartStacks` to have min/max buttons in the corner.

The Eclipse platform uses the `EventAdmin` service to communicate events for changes in the Eclipse application. For example if a *part* is activated, the Eclipse platform send out an event for this. *Addons* can subscribe to these events from the Eclipse platform and react to them.

11. Persisted model attributes

11.1. Tags

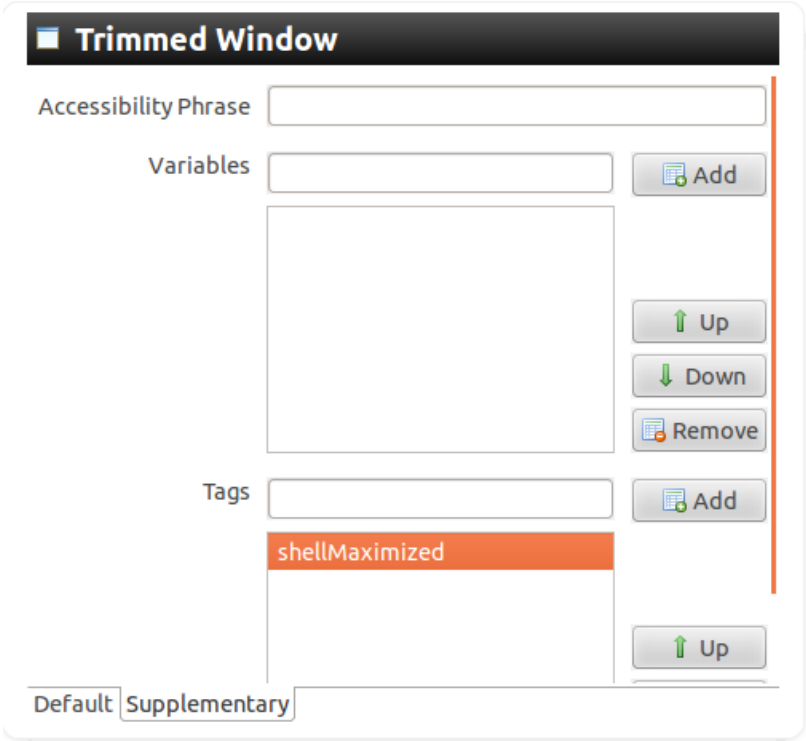
The *Supplementary* tab in the model editor allows you to enter additional information about a model element.

All model elements can have *tags* assigned to them. These *tags* can be used by the Eclipse platform or by customer coding to trigger functionality.

Tags are automatically persisted by the Eclipse runtime between application restarts and are represented as a `Collection` of `Strings`.

By default Eclipse uses some predefined *tags* to determine the state of certain model elements. For example the *shellMaximized* and *shellMinimized* tag on a *Window* is used by Eclipse to determine if the *Window* should be maximized or minimized.

The following screenshot shows how to define the maximization of a *Window* model element.



You find more information about the default tags in **Section 38, “Relevant tags in the application model”**.

You can also define *Variables* in the *Supplementary* tab which can be used as *context variables*. *Context variables* allow you to modify values which can be used for dependency injection. Dependency injection is covered in **Section 22, “Dependency injection and annotations”** and you learn more about context variables in **???**.

11.2. Persisted State

Model elements allow also to have persisted state. If you retrieve the model element you can get and set this persisted state.

```
// modelObject is the model object
// retrieved via dependency injection (e.g. an MPart)

// Get the state by the "yourKey" key
String state = modelObject.getPersistedState().get(yourKey);

// Store the state
modelObject.getPersistedState().put(yourKey, state)
```

Persisted data for model elements is automatically restored by the Eclipse application between

application restarts and allow to store key/values pairs based on Strings.

11.3. Transient data

Each model element supports also that transient data is attached to it. This transient data is based on a `Map<String , Object>` data structure and can be accessed on the model object via the `getTransientData()` method.

12. IDs and suggested naming conventions

12.1. Identifiers for model elements

Every model element allows you to define an ID. This ID is used by the Eclipse framework to identify this model element. Make sure you always maintain an ID for every model elements and ensure that these IDs are unique.

Tip

Ensure that all model elements have a unique ID assigned to them to avoid strange behavior in your Eclipse application.

12.2. Best practices for naming conventions

The following suggest best practice for naming conventions, which are also used in this tutorial.

Table 5. Naming conventions

| Object | Description |
|--------------------------------|--|
| Project Names | The plug-in project name is the same as the top-level package name. |
| Packages | For plug-in containing lots of user interface components use sub-packages based on the primary purpose of the components. For example the <code>com.example</code> package may have the <code>com.example.parts</code> and <code>com.example.handler</code> sub-package. |
| Class names for model elements | Use the primary purpose of the model element as a suffix in the <u>class name</u> . For example a class which represents a Part which displays <code>Todo</code> objects, might be called <code>TodoOverviewPart</code> . |

| | |
|-----|--|
| IDs | <p>IDs in your application model and other configuration files</p> <p>IDs should always start with the top-level package. If appropriate use the sub-package of the implementing class also. The remainder of the ID should be descriptive for the purpose of the component. For example: "com.example.parts.todoist".</p> <p>ID <u>should be only lower cases</u> (some Eclipse projects also use camelCase for the last part of the ID).</p> |
|-----|--|

13. Features and Products

The following description uses feature projects and products, please see **Feature projects** and **Eclipse Products and Deployment** for a description of these topics. n :

14. Tutorial: Create an Eclipse plug-in

14.1. Target

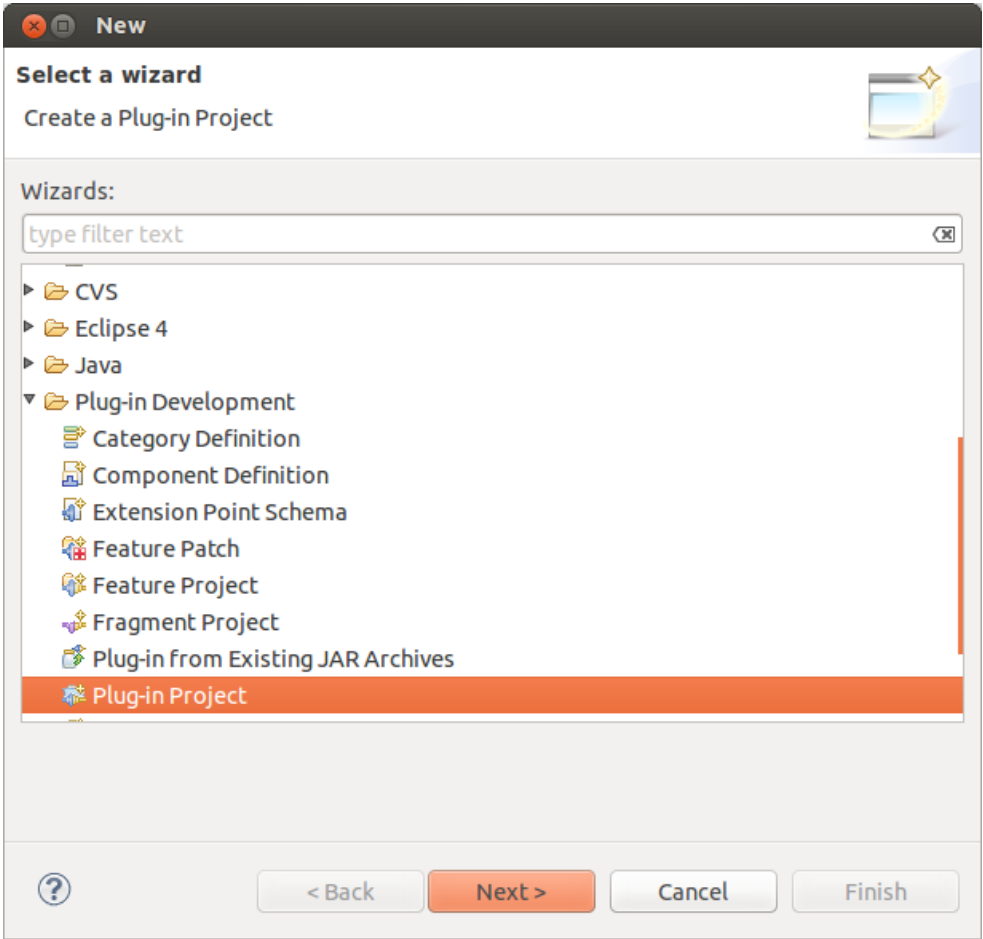
In the following tutorial you create a standard Eclipse plug-in called *com.example.e4.rcp.todo*. This plug-in is later converted into an Eclipse RCP application.

Note

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

14.2. Create a plug-in project

In Eclipse select *File* → *New Project* → *Plug-in Development* → *Plug-in Project*.



Give your plug-in the name *com.example.e4.rcp.todo*.

New Plug-in Project

Plug-in Project

Create a new plug-in project

Project name:

com.example.e4.rcp.todo

☒ Use default location

Location: /home/vogella/workspace/runtime-EclipseApplication/com

Browse...

Project Settings

☒ Create a Java project

Source folder:

src

Output folder:

bin

Target Platform

This plug-in is targeted to run with:

☒ Eclipse version:

3.5 or greater

☐ an OSGi framework:

Equinox

Working sets

☐ Add project to working sets

Working sets:

Select...

?

< Back

Next >

Cancel

Finish

Press *Next* and make the following settings. Select No at the question *Would you like to create a rich client application* and uncheck *This plug-in will make contributions to the UI* . Uncheck the *Generate an activator, a Java class that controls the plug-ins life-cycle* option.

New Plug-in Project

Content

Enter the data required to generate the plug-in.

Properties

ID:

com.example.e4.rcp.todo

Version:

1.0.0.qualifier

Name:

Todo

Vendor:

EXAMPLE

Execution Environment:

JavaSE-1.6

Environments...

Options

☐ Generate an activator, a Java class that controls the plug-in's life cycle

Activator: com.example.e4.rcp.todo.Activator

☐ This plug-in will make contributions to the UI

☐ Enable API analysis

Rich Client Application

Would you like to create a rich client application?

☐ Yes☒ No

?

< Back

Next >

Cancel

Finish

Warning

The *Would you like to create a rich client application* and the *This plug-in will make contributions to the UI* options relate to an Eclipse 3.x API compliant application. NEVER use this option if you want to use Eclipse 4 API.

Press the *Finish* button; the usage of the template is not required.

14.3. Validate the result

Open the project and check if any Java classes were created. You should have no classes in the source folder.

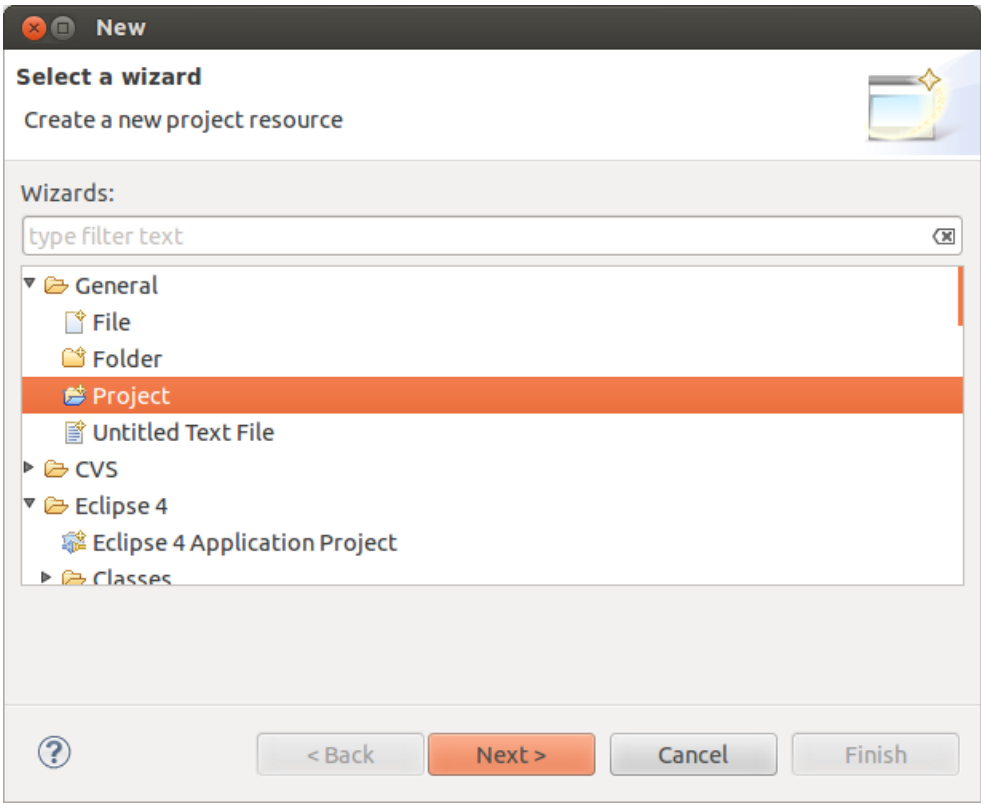
Open the *MANIFEST.MF* file and switch to the *Extensions* tab. Validate that the list of Extensions is currently empty.

15. Exercise: From plug-in to Eclipse 4 application

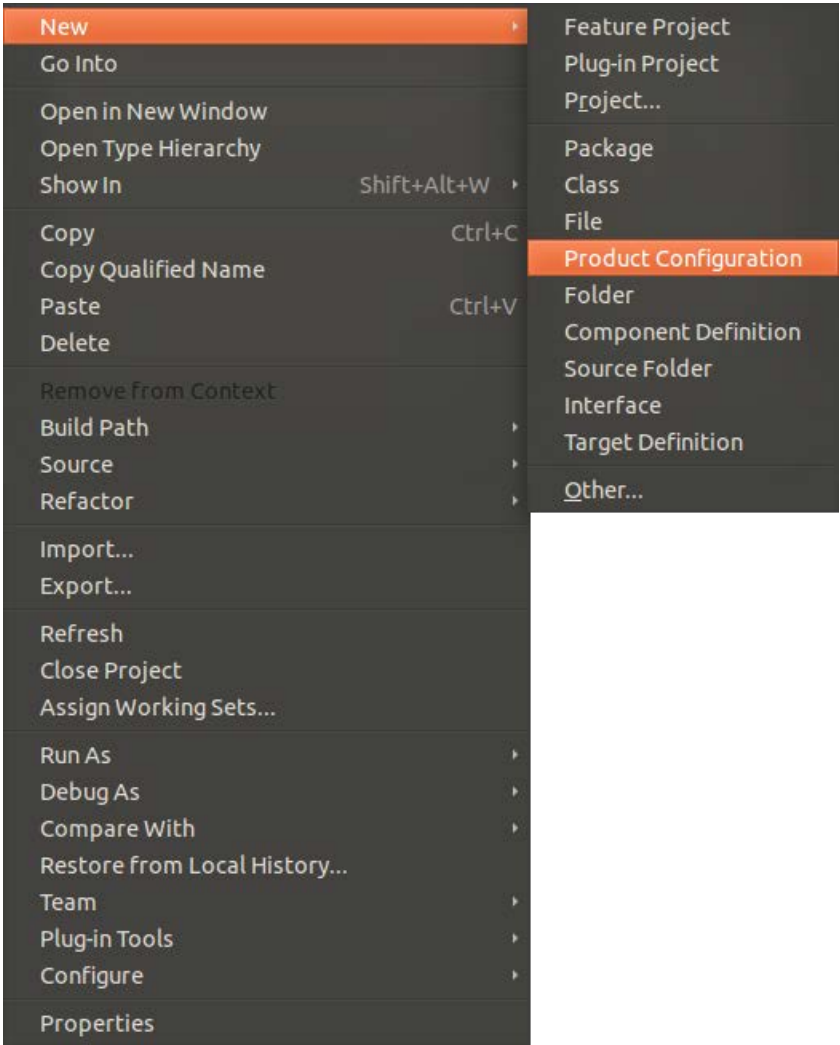
In this chapter we convert the Eclipse plug-in into an Eclipse 4 application.

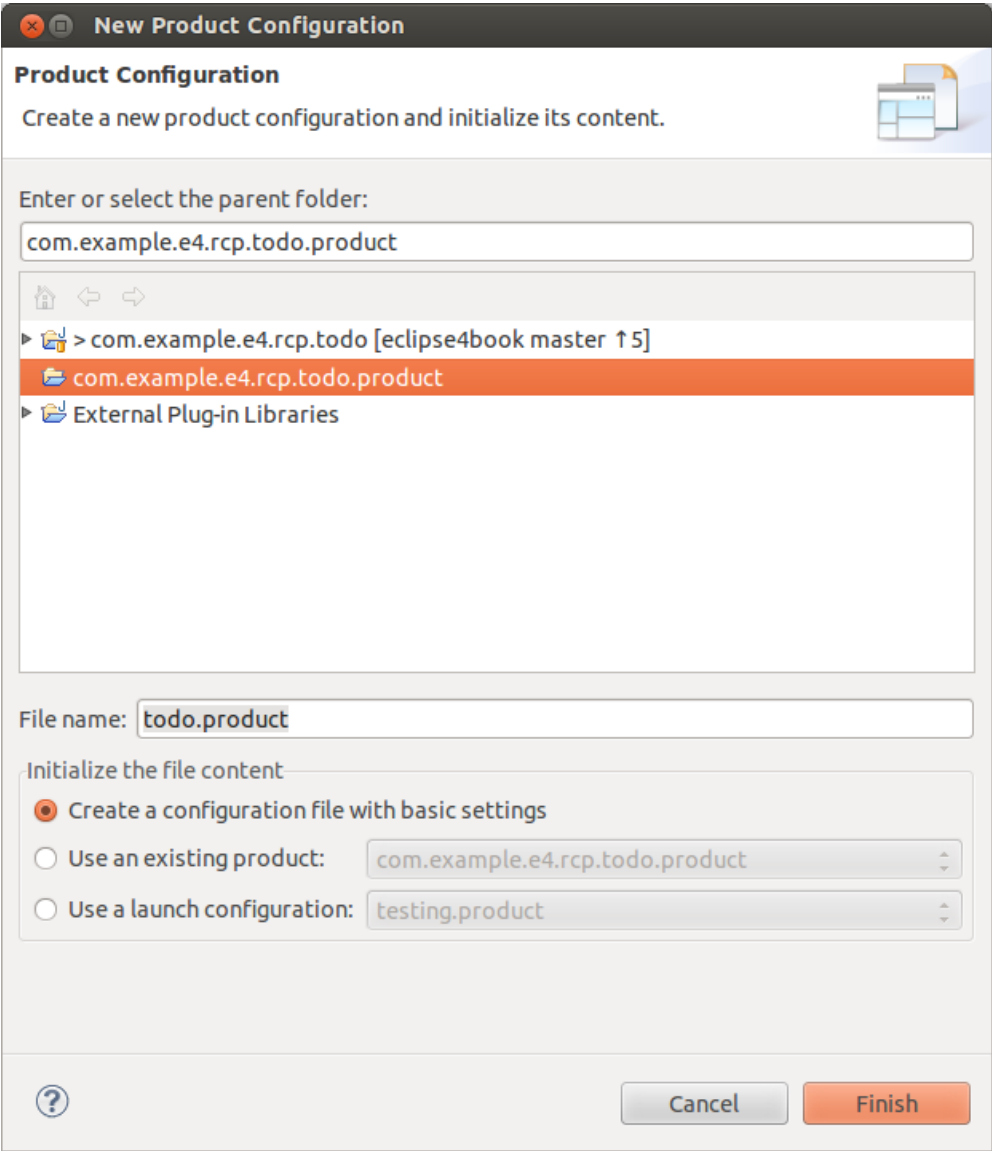
15.1. Create product configuration file

Create a new project called *com.example.e4.rcp.todo.product* of type *General* → *Project*.

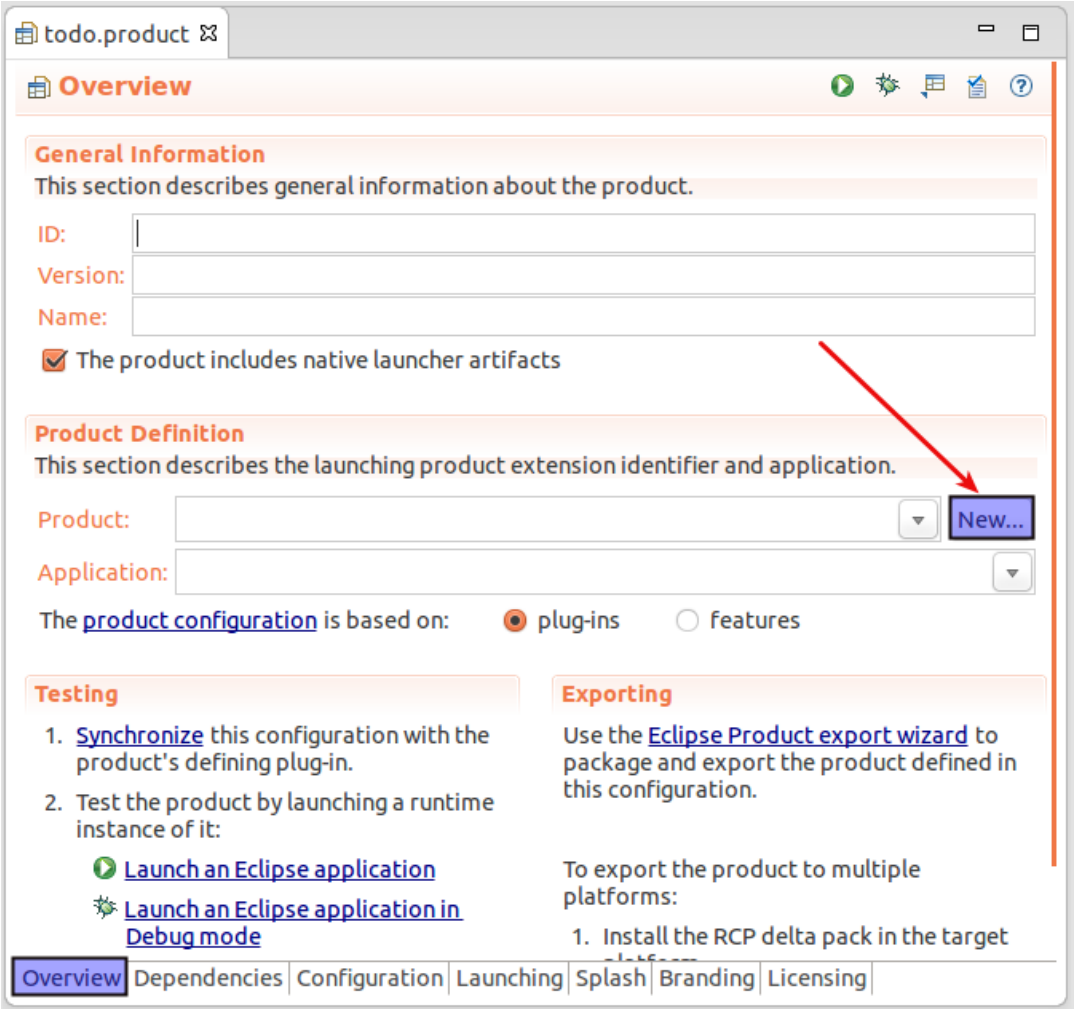


Right-click on this project and select *New* → *Product Configuration*. Create a *todo.product* product configuration file.

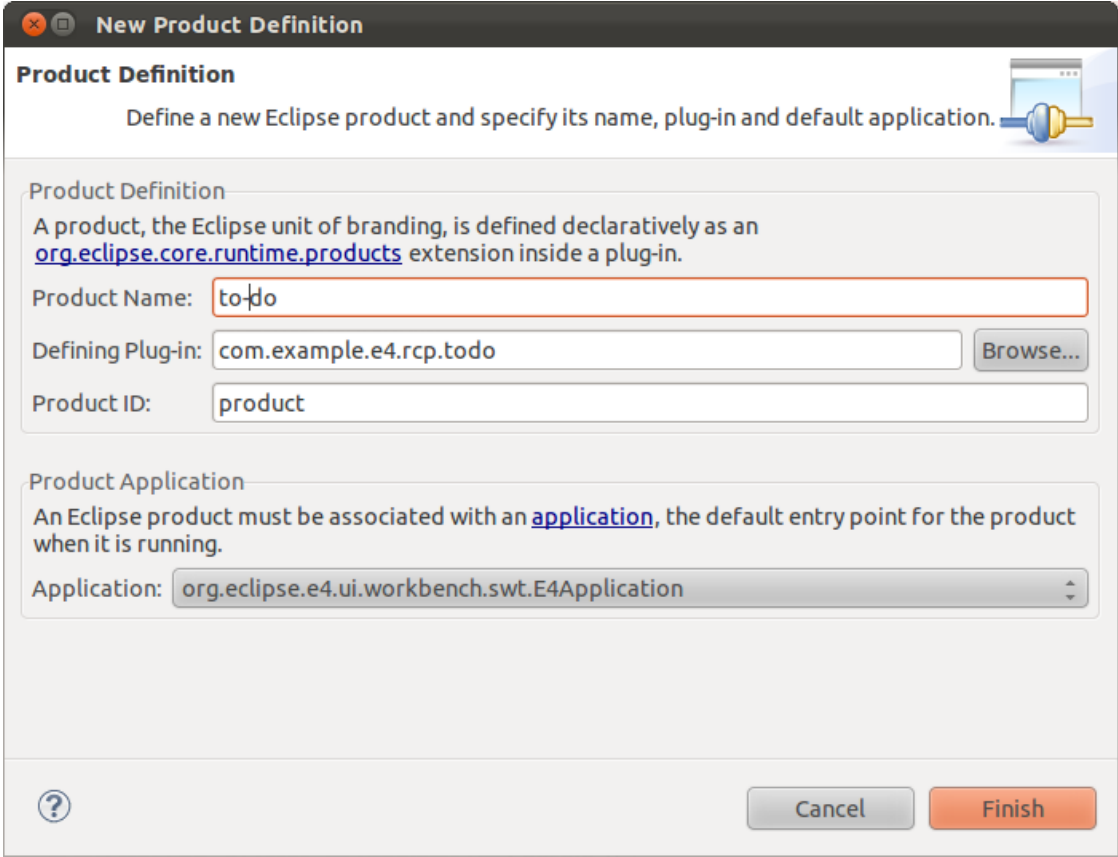




Press the *New* button on the *Overview* tab of the product editor.

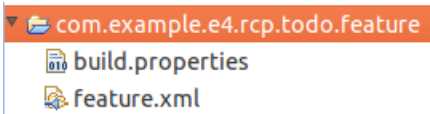


Enter *to-do* as the name, the defining plug-in is your plug-in and use the name *product* as the ID. Select as *Application* the `E4Application` application class.

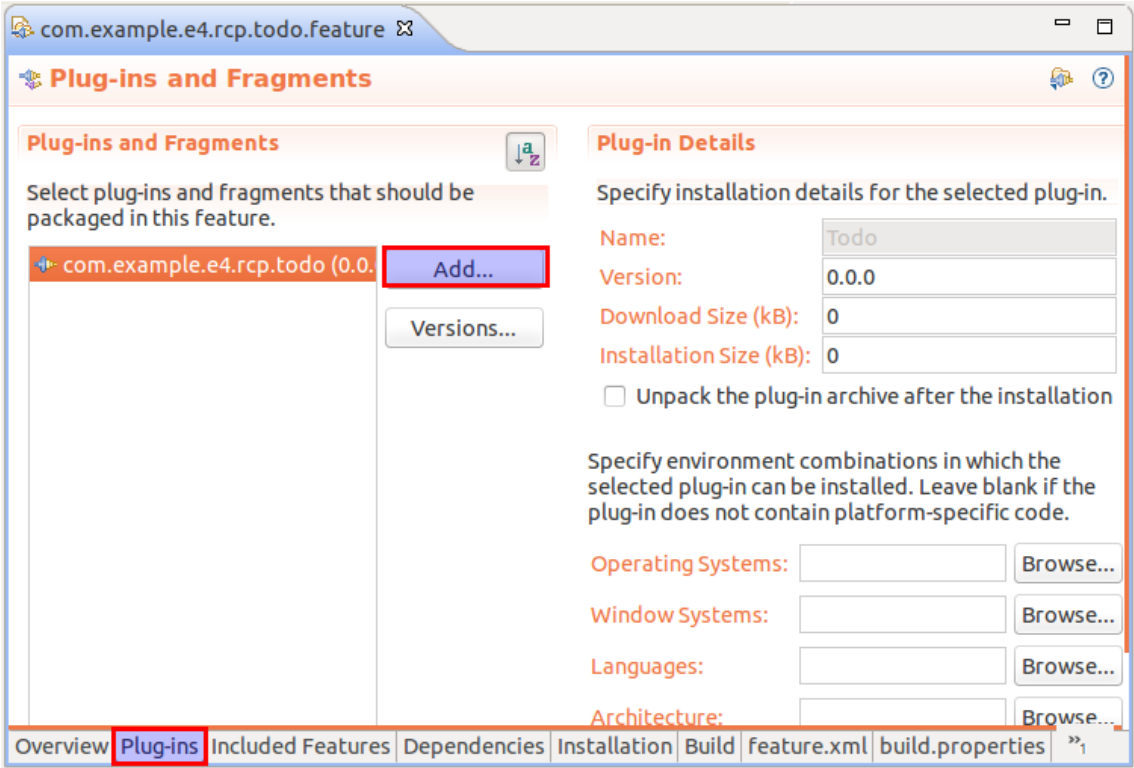


15.2. Create a feature project

Create a new feature project called *com.example.e4.rcp.todo.feature*.

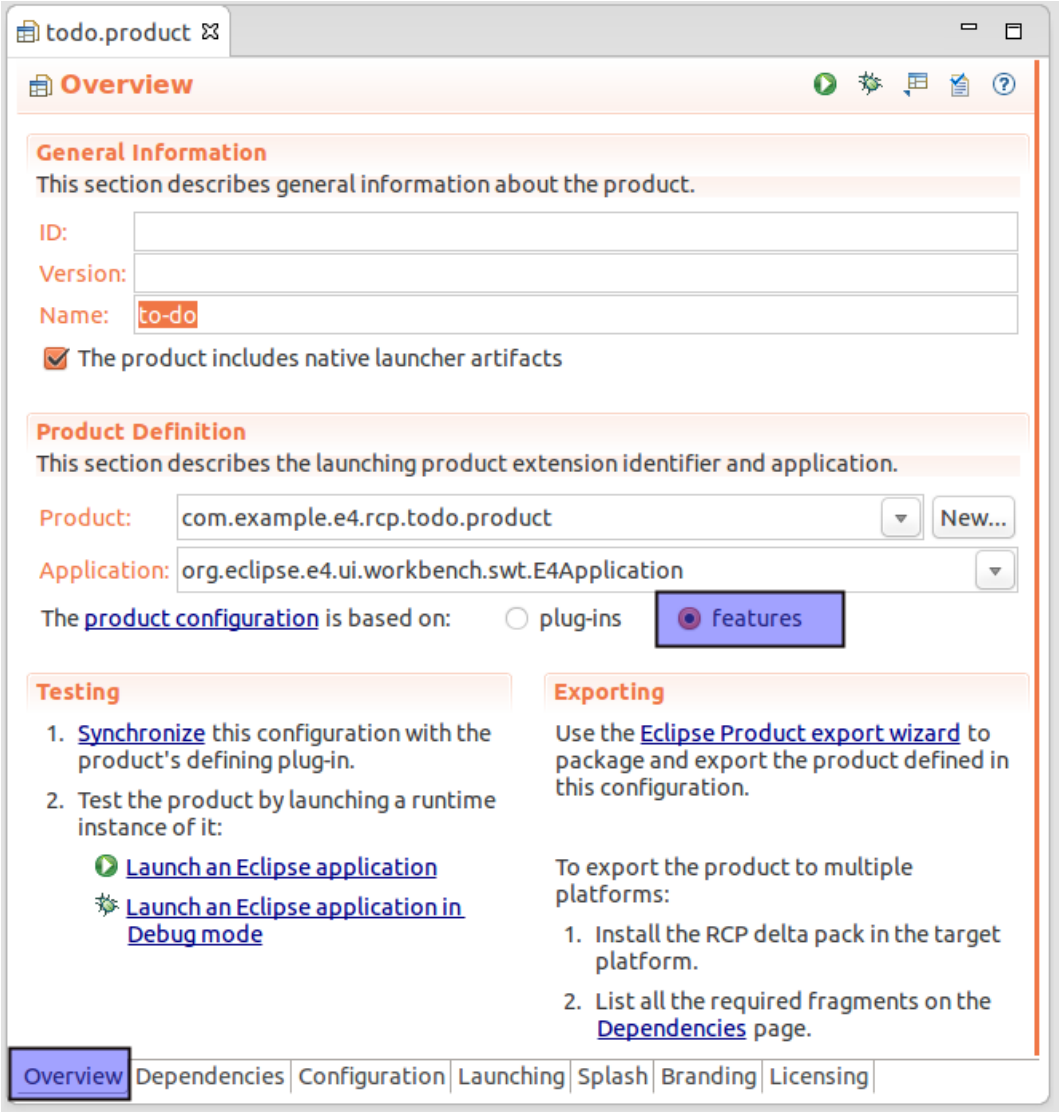


Include the `com.example.e4.rcp.todo` plug-in into this feature via the `feature.xml` file.

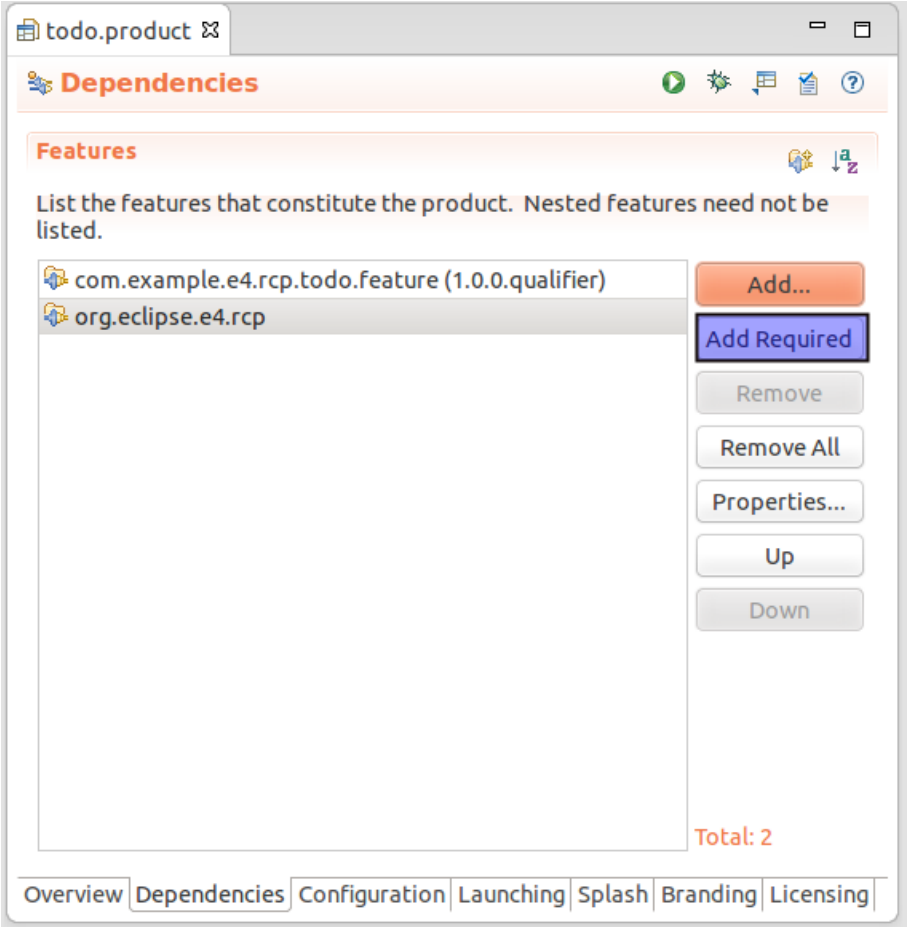


15.3. Enter feature dependencies in product

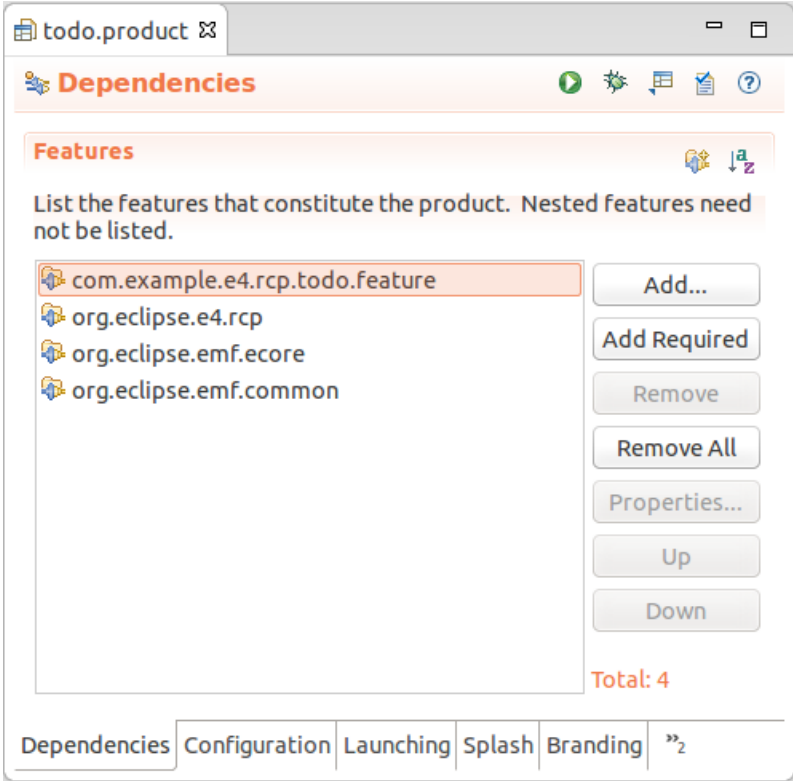
Change your product configuration file to use features. To do this open your product configuration file and select the *Feature* option on the *Overview* tab of the product editor.



Select the *Dependencies* tab and add the `org.eclipse.e4.rcp` and the `com.example.e4.rcp.todo.feature` features as dependencies via the *Add* button.



Press the *Add Required* button. This adds the `org.eclipse.emf.common` and `org.eclipse.emf.ecore` features to the dependencies.

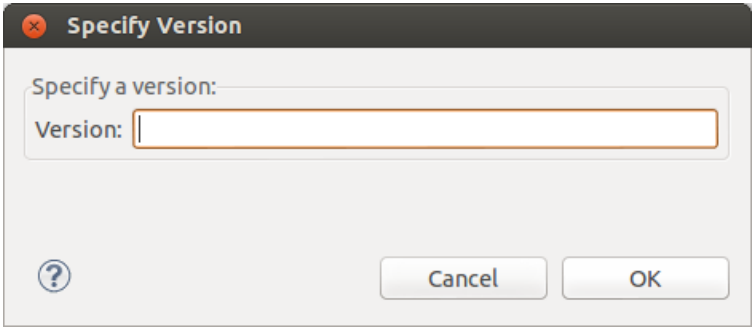
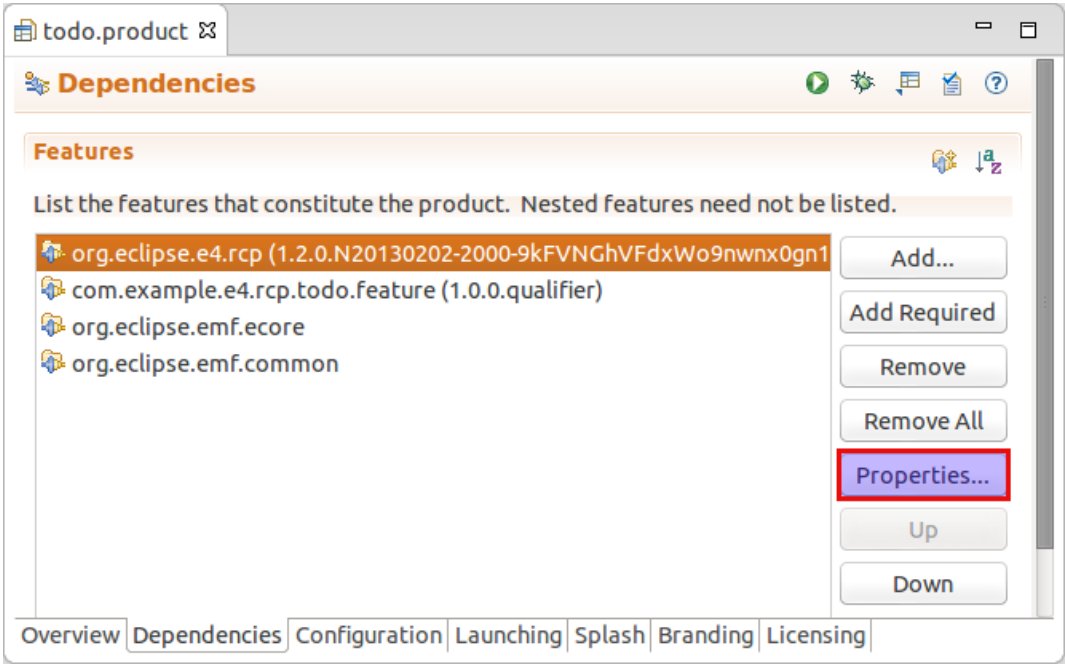


Note

Ensure that after this step you have a total of four features in your product configuration file. If you cannot add feature to your product check that you have changed your product to be based on features.

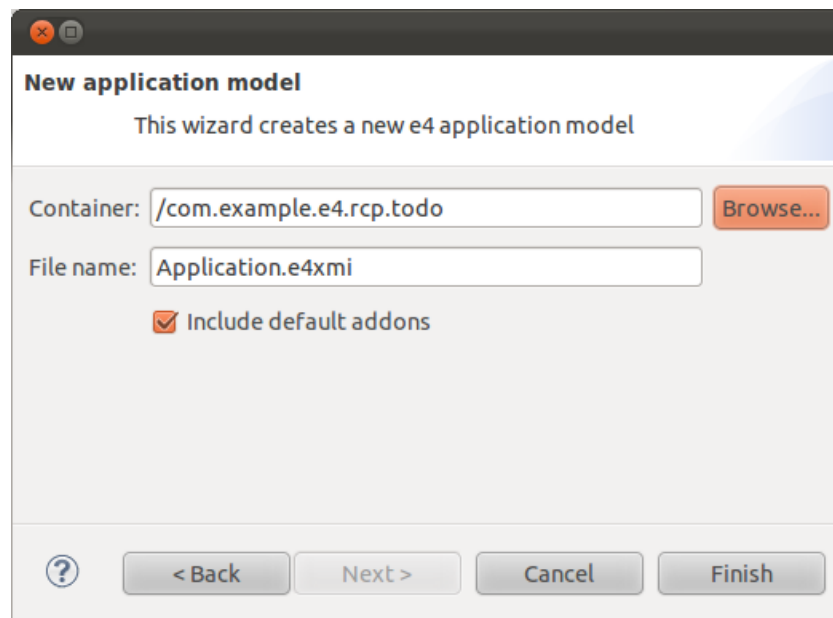
15.4. Remove version dependency from features in product

To avoid dependency problems with different versions of the `org.eclipse.e4.rcp` plug-in, delete the version number from your features. You can do this via the *Properties* button on the *Dependencies* tab of the product configuration file editor.



15.5. Create application model

Select *File* → *New* → *Other...* → *Eclipse 4* → *Model* → *New Application Model* to open a wizard. Enter your todo application plug-in as the container and the filename suggested by the wizard.

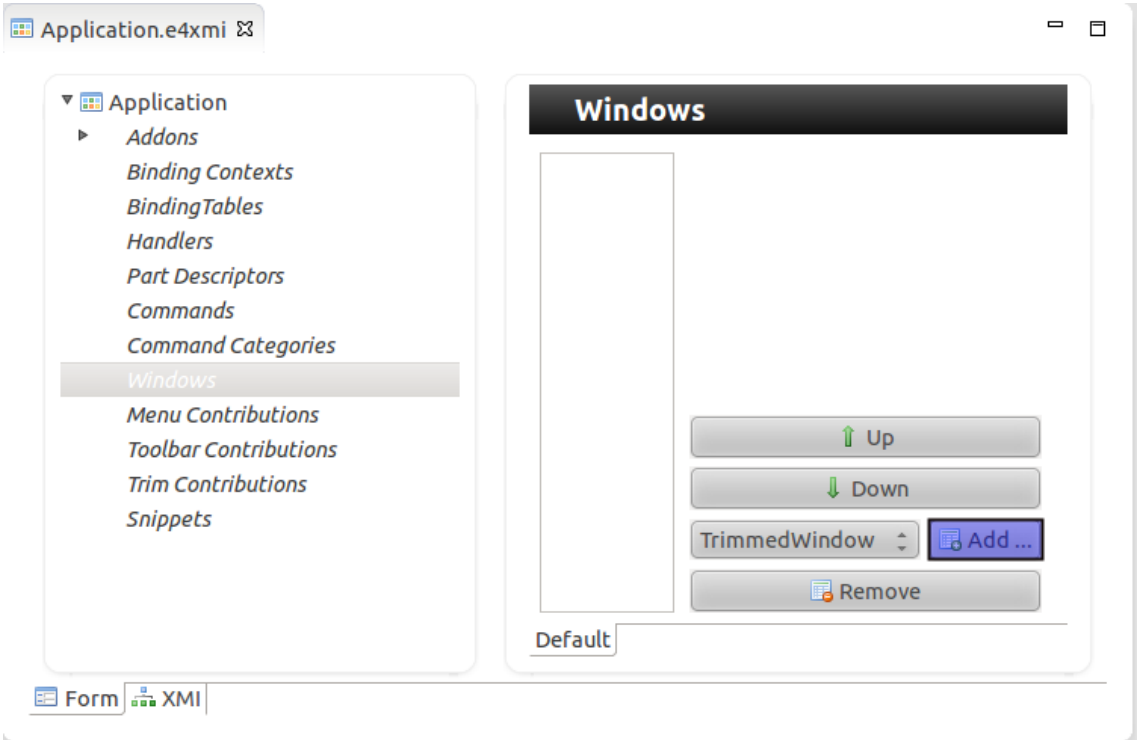


This will create the `Application.e4xmi` file and open this file with the [application model editor](#).

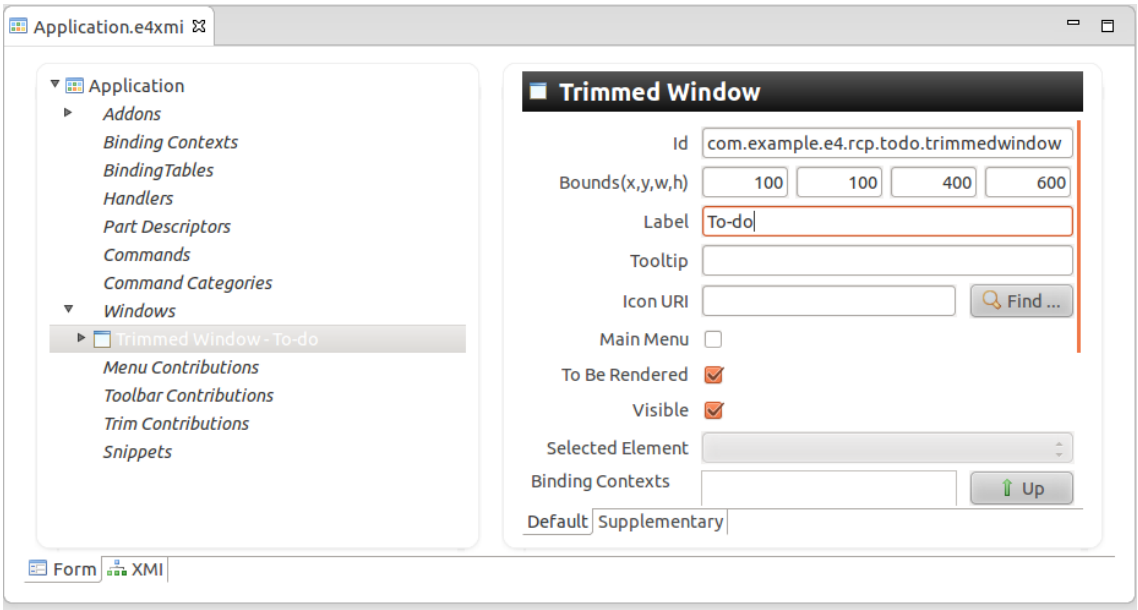
15.6. Add model elements to the application model

Add one `Window` to your application model so you have a visual component.

Select the `Windows` node and press the `Add` Button for a `TrimmedWindow`.



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





Tip

If you started your application before having created the Window model element, you need to do the setting described in **Section 18, “Configure the deletion of persisted model data”** to be able to start your application.

15.7. Start application

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

Testing

- 1. [Synchronize](#) this configuration with the product's defining plug-in.
- 2. Test the product by launching a runtime instance of it:
 -  [Launch an Eclipse application](#)
 -  [Launch an Eclipse application in Debug mode](#)

| | | | | |
|----------|--------------|---------------|-----------|--------|
| Overview | Dependencies | Configuration | Launching | Splash |
|----------|--------------|---------------|-----------|--------|

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

16. Enter bundle and package dependencies

16.1. Add plug-in dependencies

In the upcoming exercises you will use the functionality from other Eclipse plug-ins. This requires that you define a dependency to these plug-ins in your application. The exact details of applying this modular approach will be covered in a later chapter.

Note

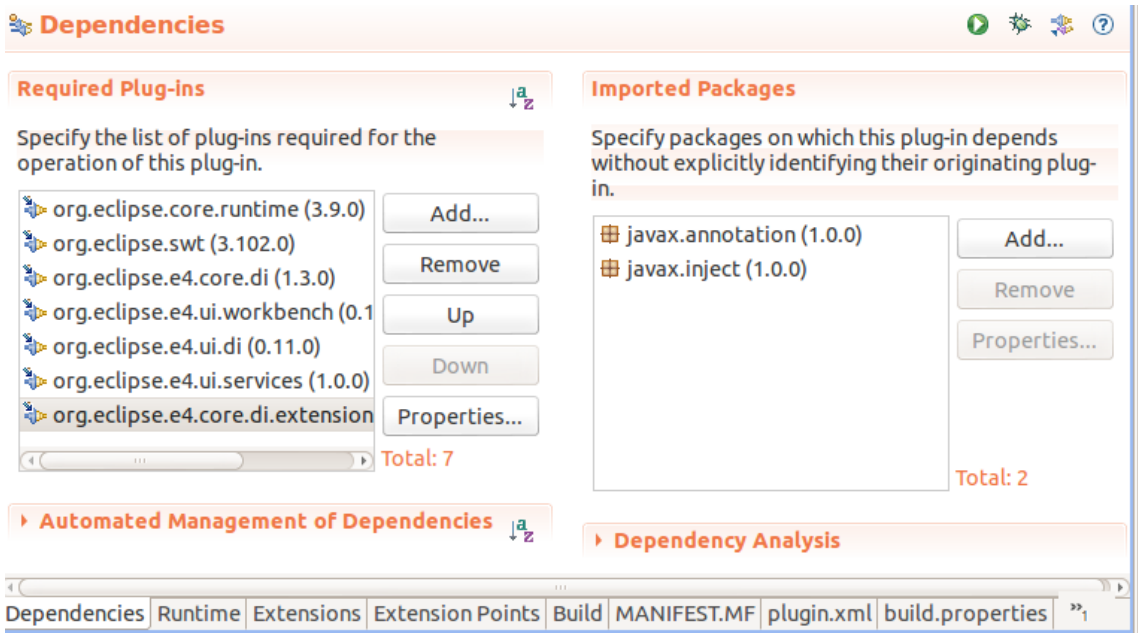
Remember that the *application plug-in* in this description is a short form for the `com.example.e4.rcp.todo` plug-in.

Open your `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`

16.2. Add package dependency

Also add `javax.annotation` and `javax.inject` as package dependencies.



Warning

Ensure that generally `javax.annotation` as well as `javax.inject` are both added with minimum version `1.0.0` as package dependencies. Otherwise your application will not work correctly in later exercises.

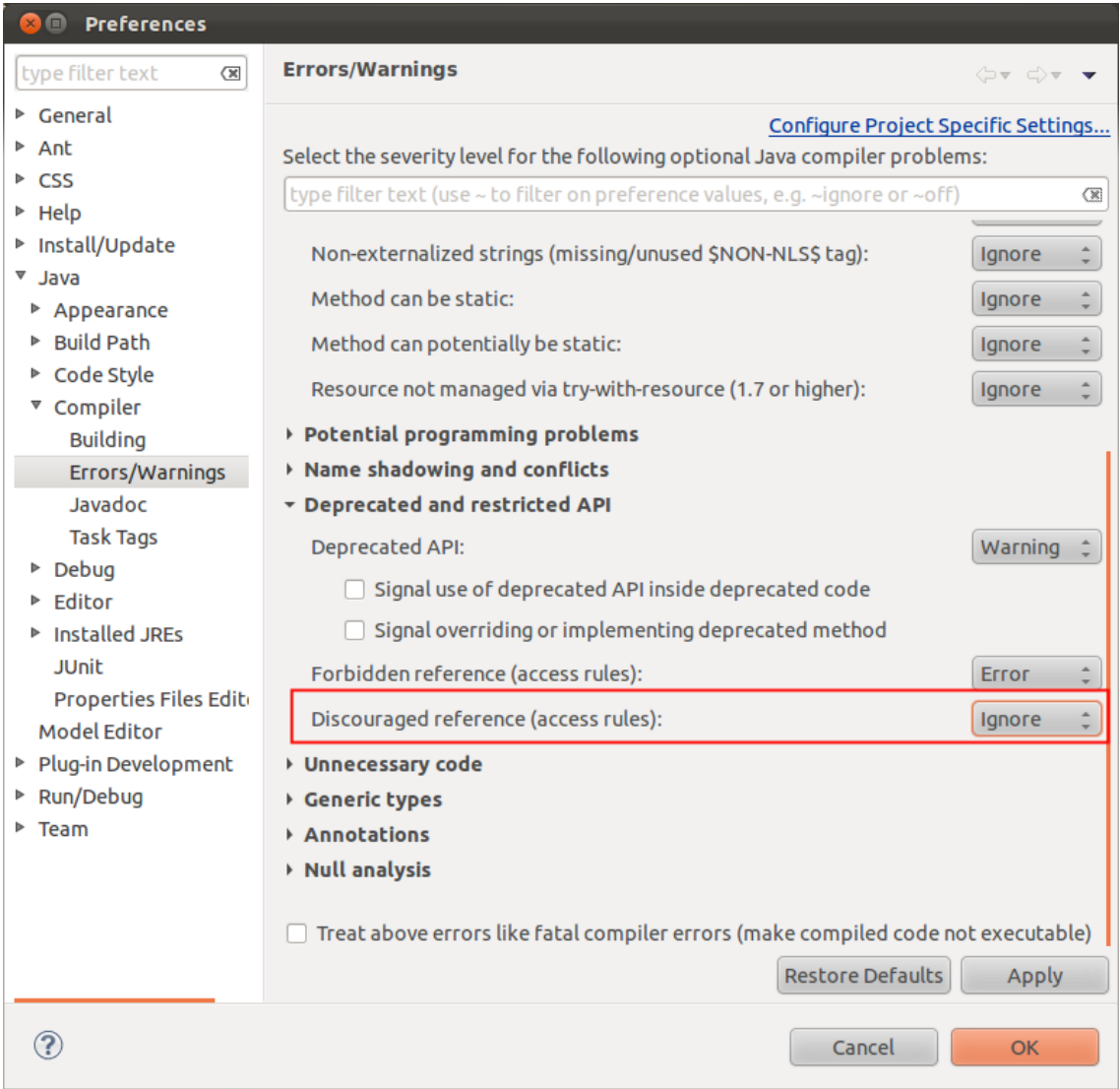
17. Remove warnings for provisional API access

Note

This exercise is optional.

The Eclipse 4 API has not yet been completely released. By default the Eclipse IDE creates warnings in your coding, if you use a provisional API.

You can turn off these warnings for your workspace via *Window* → *Preferences* → *Java* → *Compiler* → *Errors/Warnings* and by setting the *Discouraged Access* flag to *Ignore*.



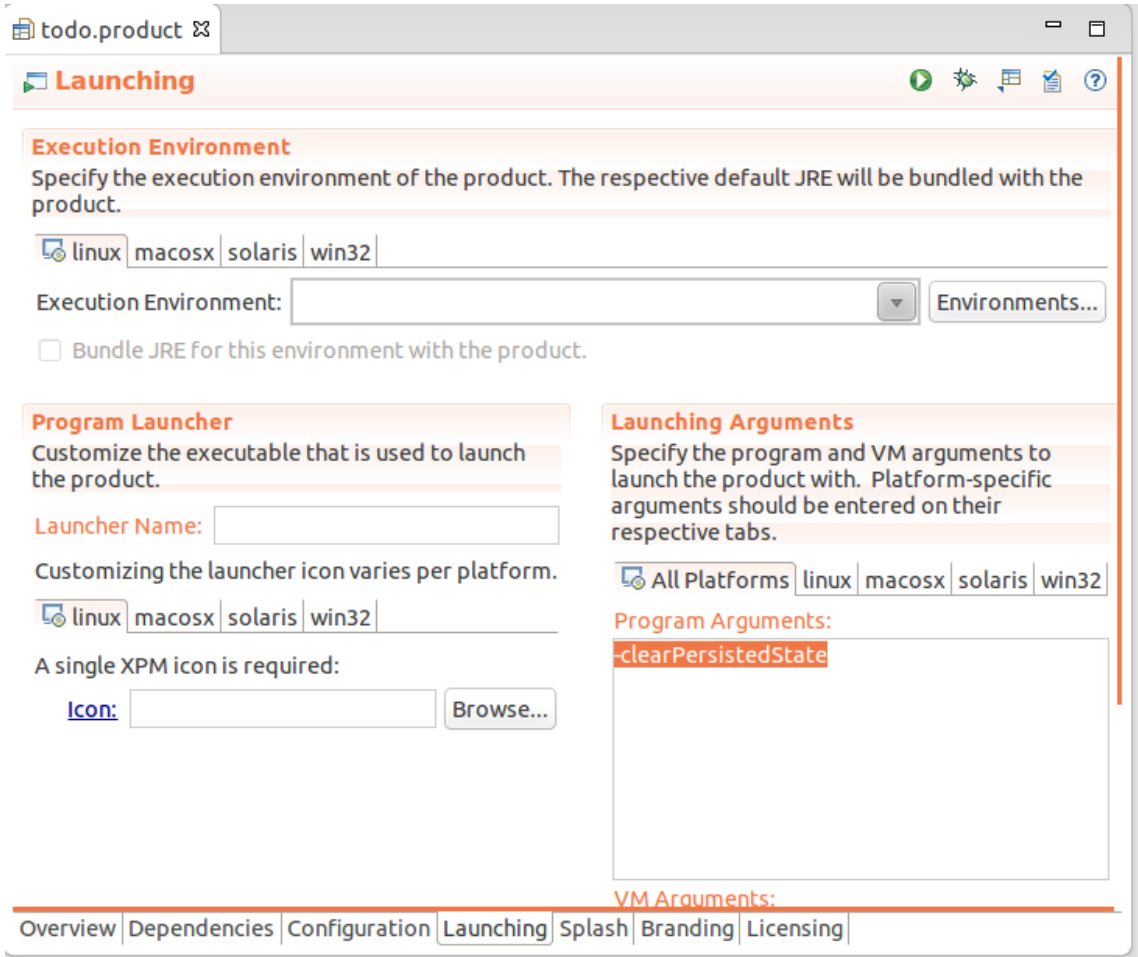
Alternatively you can turn off these warnings on a per project basis, via right-click on the project *Properties* → *Java Compiler* and afterwards use the same path as for accessing the global settings. You might have to activate the *Enable project specific settings* checkbox.

18. Configure the deletion of persisted model data

Eclipse 4 persists certain user changes in your application. During development this might lead to situations where changes are not correctly applied and displayed, e.g. you define a new menu entry and this entry is not displayed in your application.

Either set the *Clear* flag on the *Main* tab in your Run configuration or add the *clearPersistedState* parameter for your product configuration file or Run configuration.

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



It is recommended that you set this during your development phase to avoid unexpected behavior. Please note that parameters must be specified via the - sign, e.g. *-clearPersistedState*.

Warning

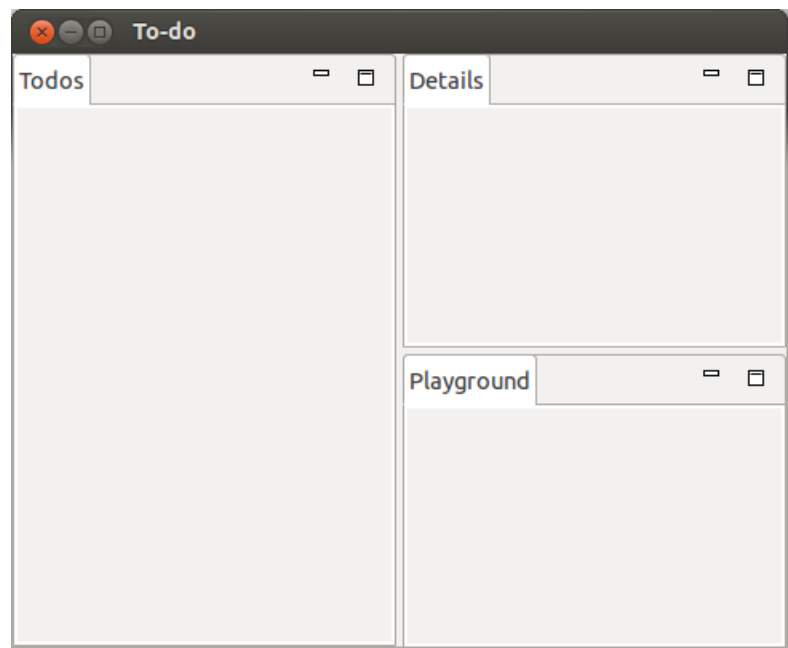
If you don't do this step, changes in the application model after a restart of your application are not visible because Eclipse restores the last state of

your application.

19. Exercise: Modeling a User Interface

19.1. Desired user interface

In this exercise you create the basis of your user interface for your RCP application. At the end of this exercise your user interface should look similar to the following screenshot.



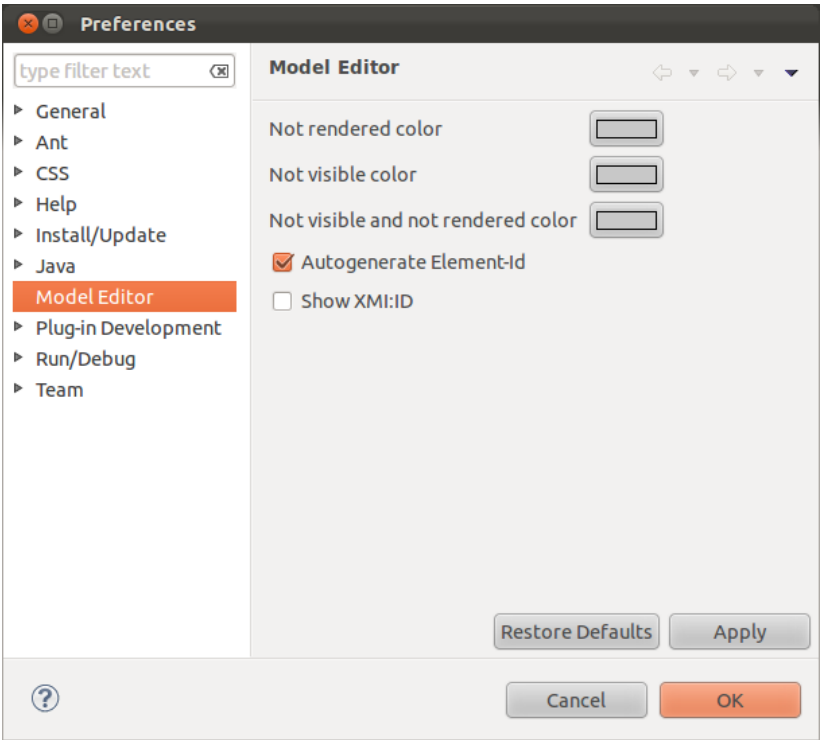
19.2. Open the Application.e4xmi

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*.

Tip

The model editor has several preference settings which can be reached via *Window* → *Preferences* → *Model Editor*. The following screenshot shows the

preference page.



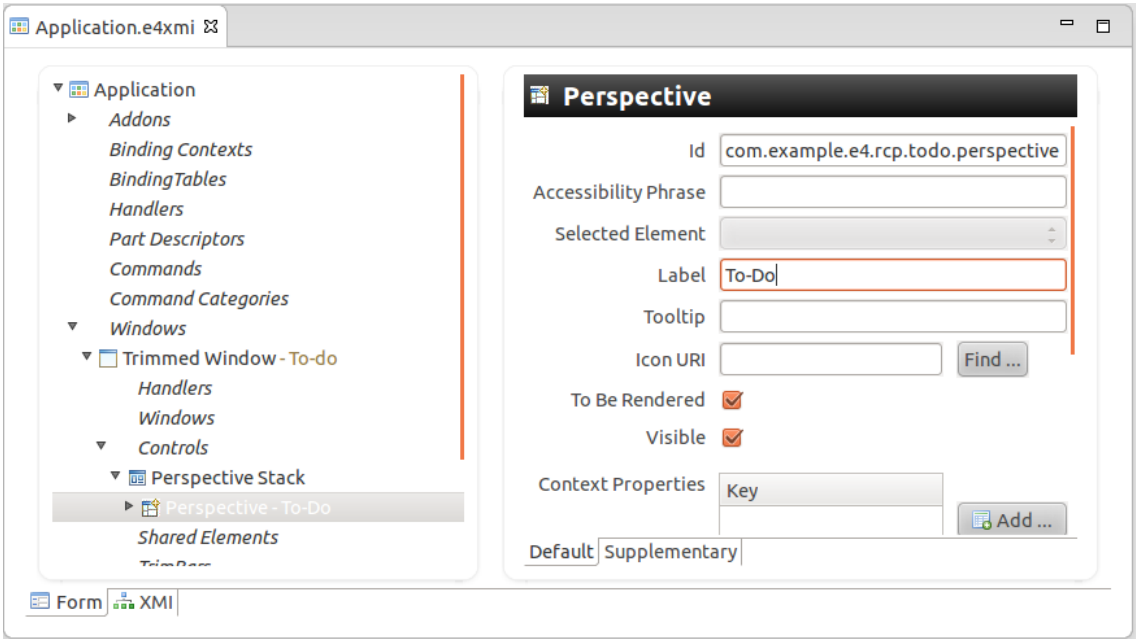
19.3. Add perspective

Note

Perspectives are optional model elements. You add a *Perspective* in this exercise so that you can later easily add more of them.

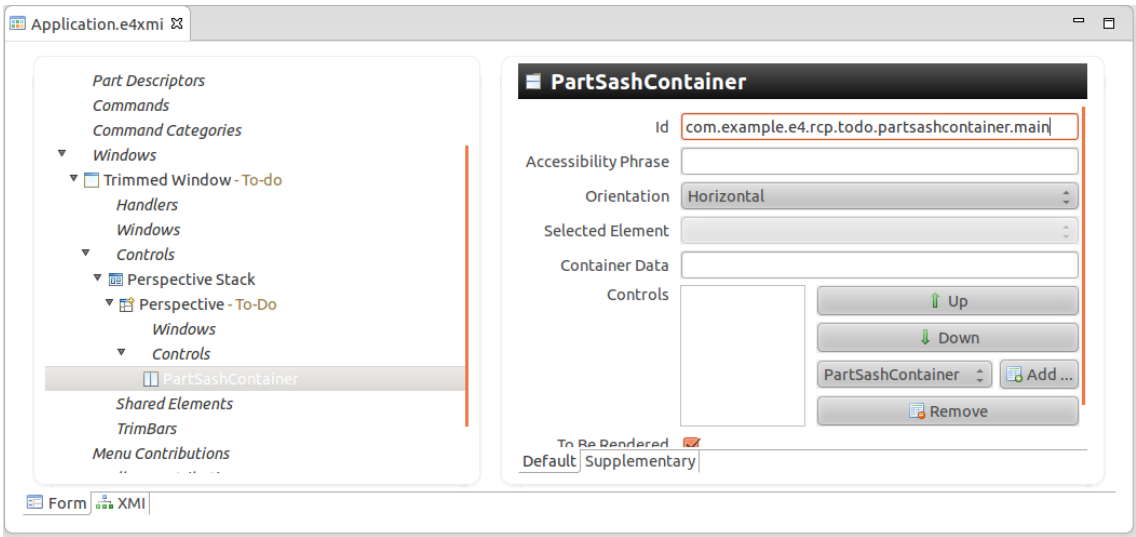
Open the `Application.e4xmi` file. Go to your *To-do* Window and select the *Controls* node. Add a *PerspectiveStack*. Press the *Add* button to create a *Perspective* entry.

Enter the value *To-Do* in the *Label* field and the value `com.example.e4.rcp.todo.perspective` in the *Id* field.



19.4. Add PartSashContainer and PartStacks

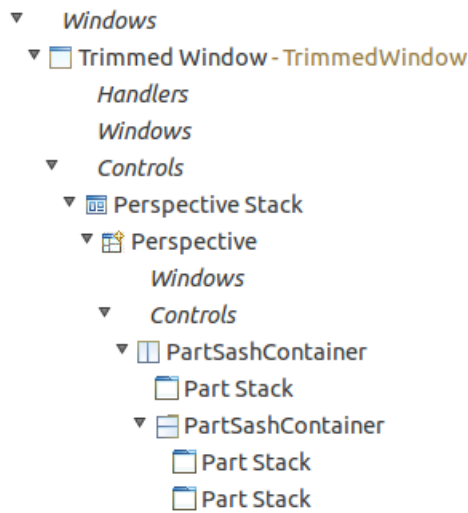
Select *Controls* below the newly created *Perspective* and add a *PartSashContainer*. Change its *Orientation* attribute to *Horizontal*.



In the drop-down list of the *PartSashContainer* select *PartStack* and press the *Add* button.

Re-select the parent *PartSashContainer* and add another *PartSashContainer*. Now add two *PartStacks* to the second *PartSashContainer*.

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



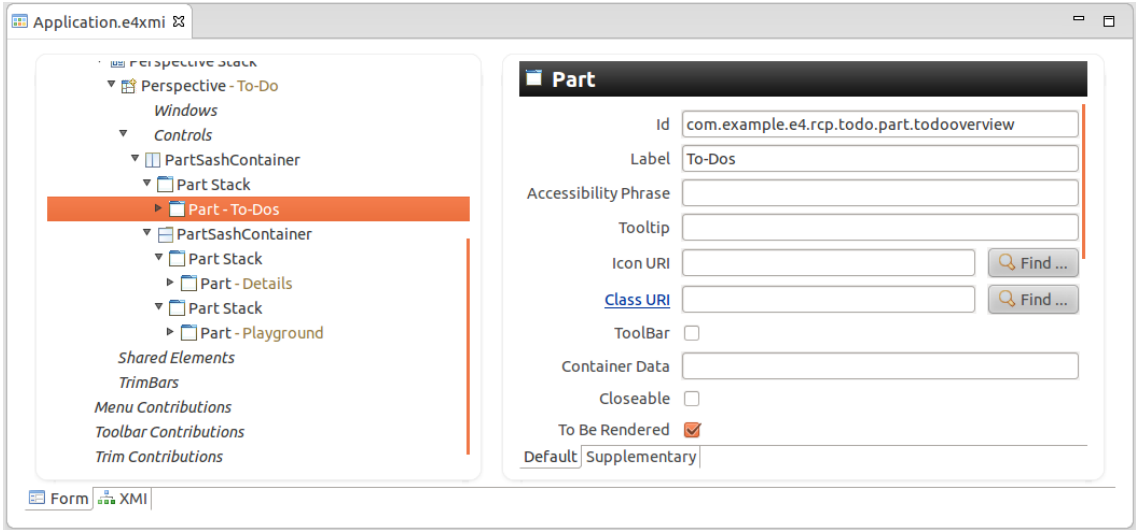
19.5. Create the Parts

Add a *Part* to each *PartStack*. As ID for the *Parts* use the prefix `com.example.e4.rcp.todo.part` and the suffix from the following table.

Table 6. Label and ID from the Parts

| ID Suffix | Label |
|----------------|------------|
| *.todooverview | To-Dos |
| *.tododetails | Details |
| *.playground | Playground |

The following screenshot shows the data for one *part*.



Start your product and validate that the user interface looks as planned. Reassign your *parts* to other *PartStacks*, if required. The model editor supports drag-and drop for reassignment.

Please note that you have not yet created a Java class for your application.

19.6. Create Java classes and connect to the model

You will now create Java objects and connect them to the application model.

Create the `com.example.e4.rcp.todo.parts` package.

Create three Java classes called *TodoOverviewPart*, *TodoDetailsPart* and *PlaygroundPart* in this package. These classes do not extend another class, nor do they implement any interface.

Tip

You can create the classes by clicking on the *Class URI* hyperlink for the part data in the application model.

The following code shows the `TodoDetailsPart` class.

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

public class TodoDetailsPart {

}
```

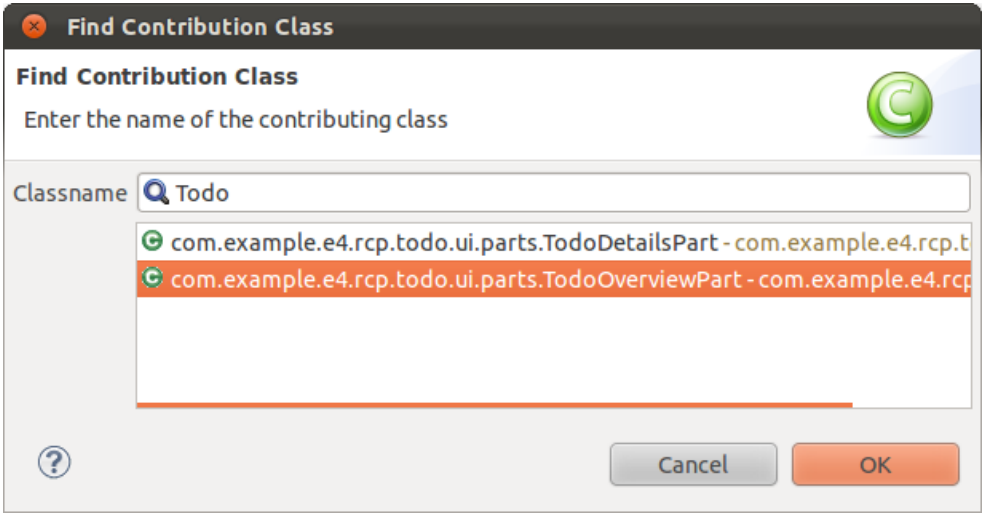
Open the `Application.e4xmi` file and connect the class with the correct model object. You can do this via the *Class URI* property of the *part* model element.

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

Table 7. Connecting Java classes with Model Element

| Class | Part ID suffix |
|------------------|----------------|
| TodoOverviewPart | *.todooverview |
| TodoDetailsPart | *.tododetail |
| PlaygroundPart | *.playground |

The Eclipse 4 model editor allows you to search for an existing class via the *Find...* button. The initial list is empty, start typing the class name to see the results.



19.7. Test

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

To validate that the model objects are created by the Eclipse runtime, create a no-argument

constructor, e.g. with no parameters, for one of the classes and add a `System.out.println()` statement. Afterwards verify that the constructor is called, once you start your application.

20. Tutorial: Using the SWT browser widget

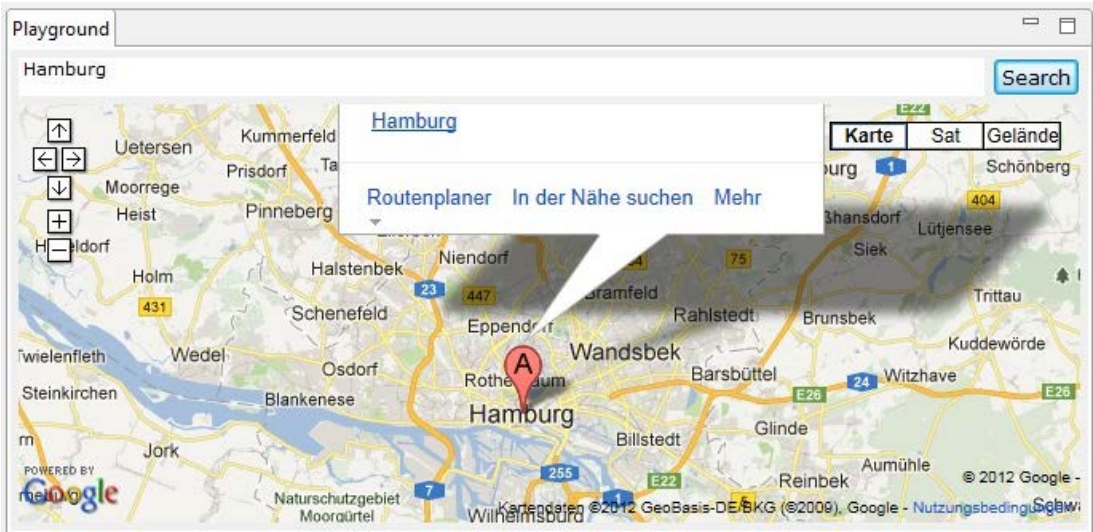
20.1. Implementation

In this tutorial you will display Google Maps in a SWT `Browser` widget.

Note

This exercise does not always work on a Linux system because on certain Linux version the SWT `Browser` widget does not work.

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



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

20.2. Solution

As a result your `Playground` class should look similar to the following code.

```

package com.example.e4.rcp.todo;

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.NONE);
        button.setText("Search");
        button.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {
                String city = text.getText();
                if (city == null || city.length() == 0) {
                    return;
                }
                try {
                    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();
    }
}

```

21. Introduction to dependency injection

See [Dependency injection in Java](#) for an introduction into the concept of dependency injection,

22. Dependency injection and annotations

22.1. Define dependencies in Eclipse

The programming model in Eclipse supports constructor, method and field injection according to the Java Specification Request 330 (JSR330). It uses the standard `@Inject` and `@Named` annotations.

Note

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

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

In addition to these annotations, Eclipse also declares and supports additional annotations as for example the `@Optional` annotation.

The following table gives an overview of the usage of these annotations for the purpose of dependency injection.

Table 8. Basic annotations for dependency injection

| Annotation | Description |
|----------------------|---|
| @javax.inject.Inject | Marks a field, a constructor or a method. The Eclipse framework tries to inject the corresponding objects into the field or the parameters of the constructor or method. |
| @javax.inject.Named | Defines the name of the key for the value which should be injected. By default the fully qualified class name is used as key. Several default values are defined as constants in the <code>IServiceConstants</code> interface. |
| @Optional | <p>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.</p> <p>The specific behavior depends where the <code>@Optional</code> is placed.:</p> <ul style="list-style-type: none">for parameters: a <code>null</code> value will be injected;for methods: <u>the method calls will be skipped</u>for fields: the values will not be injected. |

Note

The Eclipse platform supports by default additional annotations for special purposes, e.g. for the propagating events or working with preferences. For a summary of all standard annotations defined in the Eclipse platform see **Section 40.1, “Overview of all available annotations in Eclipse”**.

22.2. On which objects does Eclipse perform dependency injection?

The Eclipse runtime creates objects for the Java classes referred to 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.

Note

For objects **which are not referred to by the application model, Eclipse is not responsible for creating them and therefore does not perform dependency injection on them.**

22.3. Re-injection

The Eclipse framework tracks the injected values and if they change, it can re-inject the new values. This means applications can be freed from having to install (and remove) listeners.

Note

This is a very useful feature. Eclipse has, to the knowledge of the author of this tutorial, the only dependency injection framework which supports re-injection.

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

23. Objects available for dependency injection

23.1. Eclipse context (IEclipseContext)

During startup the Eclipse runtime creates a *context* based on the `IEclipseContext` interface. This context contains objects which can be injected via dependency injection.

The context is similar to a transient map, in which objects can be placed under a certain key. The key can be the Class name or a String.

If the key used to store an object in the context is a String, this key/value pair is called a *Context variable*.

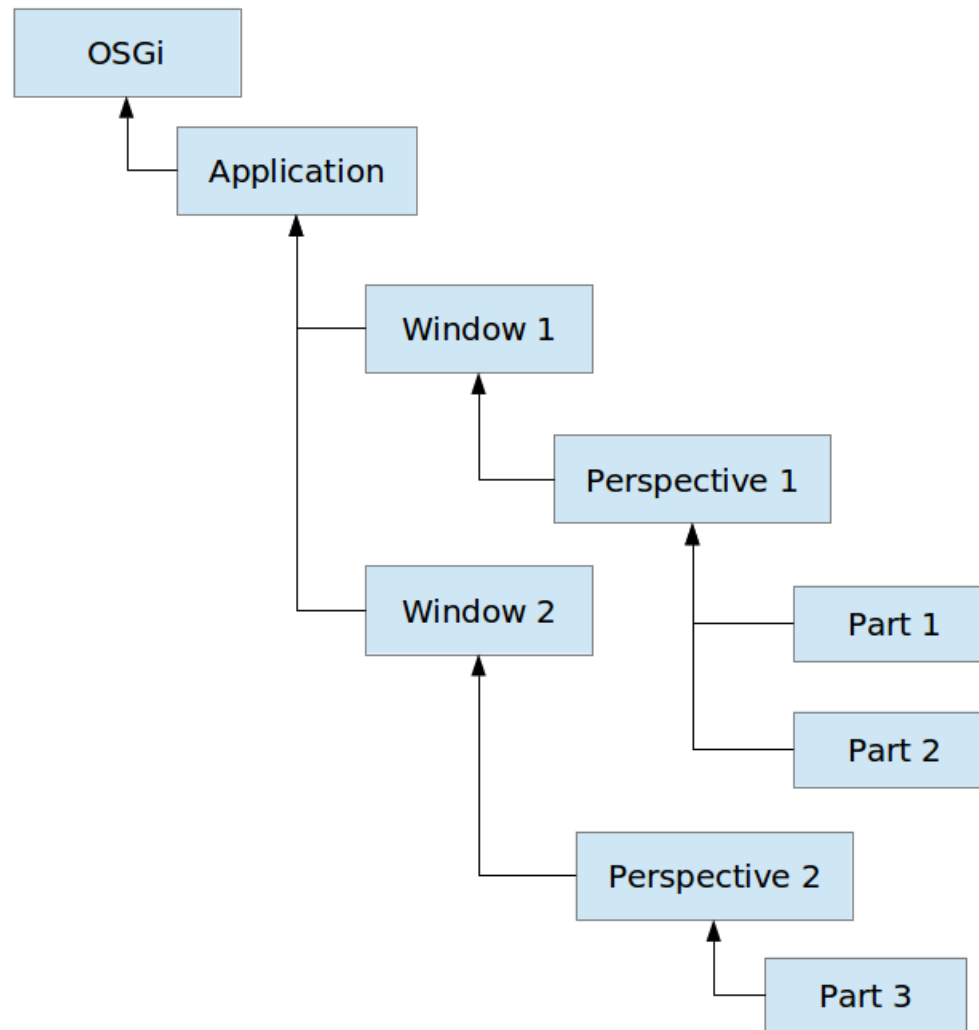
The Eclipse platform places objects into the context by default, for example services to trigger Eclipse framework functionality.

23.2. Context relationship

As said before the context is similar to a `Map` data structure. The difference is that it is not a flat structure. It is hierarchical and can also dynamically compute values for requested keys.

Several contexts can be linked together to form a tree structure.

For every model object in your application of type `MContext`, an own local context is created. These contexts are connected based on the structure of your application model. An example context hierarchy is depicted in the following picture.



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

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

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

- `MApplication`

- MPerspective
- MPart
- MWindow
- MPopupMenu

23.3. How are objects selected for dependency injection

By default a context created by the Eclipse platform is associated to a model object.

The default context hierarchy is created by the Eclipse framework and all context objects of the model elements are hierarchically connected to the main context object.

If Eclipse performs dependency injection on a Java object it first searches for an fitting object based on the specified key. The search starts in the local context associated with the 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. At this point the framework would check for fitting OSGi services in the OSGi registry.

The search happens transparently for the caller of the injection.

23.4. Default objects for dependency injection

By default the following objects can be injected via dependency injection:

- model objects - contain the data of the application model
- all Preferences - key/value pairs which are persisted between application restarts and are typically used to store configuration of the application
- Eclipse and OSGi services - software components which are defined by the Eclipse platform or via the OSGi service registry
- all other objects which have explicitly been added to the context

Note

Preferences are actually not stored in the context. They are accessed via a special OSGi service which registers itself for the `ExtendedObjectSupplier` interface.

This *context* can be modified, e.g. the application and the framework can add elements to the context. It is also possible to define your own `ExtendedObjectSupplier`, which allows you and the framework to inject objects which are not in the context.

23.5. Creation process of the Eclipse context

During startup the Eclipse framework creates the context hierarchy and register services. For each model element it also determines which objects should be available in the local context of the model object and the Java object created based on the model object `class URI` property.

To some degree this creation of the context is handled by the so-called renderer framework. This framework allows you to define classes responsible for setting up the Java implementation of the model objects. For example the `ContributedPartRenderer` class is by default responsible for creating the Java objects referred to by the *part* model objects.

`ContributedPartRenderer` creates a `Composite` for every *part* and injects this `Composite` into the local context of the *part*.

23.6. Tracking a child context with `@Active`

The `@Active` annotation allows you 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 value 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 following code snippet.

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

Note

`@Active` is mainly useful for the Eclipse framework itself, the author of this tutorial has not yet managed to find a good use case for Eclipse RCP

applications.

24. Behavior Annotations

24.1. API definition via inheritance

In general, every framework defines an Application Programming Interface (API). If you use a framework you need to have a convention for which methods are called at which point of the execution of your program. For example if a Java class is responsible for handling a toolbar button click, the framework needs to know which method of this class it should call.

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

The framework defines for example an abstract class which defines methods which must be implemented. In this example the method might be called `execute()` and the framework knows that this method must be called once the toolbar 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.

24.2. API definition via annotations

The Eclipse 4 platform API is not based on inheritance. To identify which methods should be called at a certain point in time, the Eclipse platform uses annotations.

These annotations are called *behavior annotations*.

The following tables list the available behavior annotations for *parts*.

Table 9. Eclipse lifecycle annotations for Parts

| Annotation | Description |
|----------------|--|
| @PostConstruct | Is called after the class is constructed and the field and method injection has been performed. |
| @PreDestroy | Is called before the class is destroyed. Can be used to clean up resources. |
| @Focus | Indicates that this method should be called, once the Part gets the focus. It is required to set the focus on one user interface control otherwise certain workbench functionality does not work. |

| | |
|---------------|--|
| @Persist | Is called if a save request on the Part is triggered. Used by the EPartService to identify the method to call if a save is triggered via this service. |
| @PersistState | Is called before the model object is disposed, so that the Part can save its state. |

Warning

All these annotations also implies that the framework need to provide the specified parameters to the method, i.e. the framework also performs method dependency injection. Adding the `@Inject` annotation would trigger twice the method call, first during the dependency injection phase and later for the behavior annotation. This is typically undesired and therefore an error.

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

Eclipse defines also more behavior annotations. The following table lists a few more but you also have annotations for commands which are covered in the command chapter.

Table 10. Other Eclipse Behavior Annotations

| Annotation | Description |
|-------------------------------------|---|
| @GroupUpdates | Indicates that updates for this <code>@Inject</code> should be batched. If you change such objects in the <code>IEclipseContext</code> the update will be triggered by the <code>processWaiting()</code> method on <code>IEclipseContext</code> . |
| @EventTopic and @UIEventTopic | Allows you to subscribe to events send by the EventAdmin service. |

25. Tutorial: Using dependency injection

25.1. Getting a Composite

In the following tutorial we extend our classes to use dependency injection.

Change the `TodoOverviewPart` class to the following:

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

import javax.inject.Inject;

import org.eclipse.swt.widgets.Composite;

public class TodoOverviewPart {

    @Inject
    public TodoOverviewPart(Composite parent) {

        // assuming that dependency injection works
        // parent will never be null
        System.out.println("Woh! Got a Composite via DI.");

        // does it have a layout manager?
        System.out.println("Layout: " + parent.getLayout().getClass());
    }
}
```

25.2. Validation

Run your application and check in the *Console view* of your Eclipse IDE to see if the `Composite` parameter was injected. Note down the layout class which the `Composite` has assigned to, if it is not `null`.

26. Exercise: Using @PostConstruct

26.1. Why using @PostConstruct?

It is possible to create the user interface of a *part* in the constructor but injection for fields and method has not been done at this point.

Therefore it is recommended that the user interface is created in a `@PostConstruct` method.

Realizing the user interface in a method annotated with `@PostConstruct` requires that `@Inject` methods be aware that the user interface might not yet be created.

26.2. Implement @PostConstruct

Add the following method to your `TodoOverviewPart`, `TodoDetailsPart` and `PlaygroundPart` classes.

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

Remove all constructors from your classes.

26.3. Implement @Focus and test your application

Implement a @Focus method in one of your *parts*.

```
@Focus  
private void setFocus() {  
    System.out.println("@Focus method called");  
}
```

Tip

Before Eclipse 4.3 it was mandatory to implement at least one user interface control in each part and put focus on it via a method annotated with the @Focus annotation. Since the Eclipse 4.3 release this is optional but still good practice. Once you implement the user interface in the parts you should put focus on one of them.

Start your application and ensure that the corresponding method is called whenever the user reselects this part after having clicked on another part.

26.4. Validate

Run your application and validate that the @PostConstruct method is called. Use either debugging or a `System.out.println()` statement.

If you are familiar with SWT, add a few more controls to your user interface.

Note

If the @PostConstruct method is not called, ensure that you have entered a package dependency to the `javax.annotation` package and set the version to 1.0.0. See <http://wiki.eclipse.org/Eclipse4/RCP/FAQ> for details on this issue.

27. Menu and toolbar application objects

27.1. Adding menu and toolbar entries

You can add menus and toolbars to your RCP application via the application model. These entries can be positioned on various places, for example a window can have a menu but also each part can have a menu (which is displayed as a drop-down menu).

The application model provides several options to contribute menu and toolbar entries. For simple cases you can use the *Direct MenuItem* or a *Direct ToolItem* model elements , which allows you to define a reference to a class which is executed if the corresponding item is selected. The following description calls these elements: *direct items*.

If you use the *MenuItem* and *ToolItem* model elements you refer to *commands* to define the corresponding logic. *Commands* are different model elements which are described later in this section.

The application model supports also the dynamic creation of menus via the `DynamicMenuContribution` model elements.

Toolbars in the application are encapsulated in the application model via the *trimbar* model element. A trimbar can be defined for Windows. Via its attribute you define if the trimbar should be placed on the top, left, right or bottom corner of the Window.

Menus and toolbars support separators. Menus can have submenus.

27.2. What are commands and handlers?

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

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 behavior of a command is defined via a *handler*. A handler model elements points to a class via the `contributionURI` attribute of the handler. This attribute is displayed as *Class URI* in the model editor. Such a class is called *handler class* in this tutorial.

Commands can be used to define user interface components to perform actions in your application, for example menus or toolbar items.

Tip

Prefer the usage of commands vs. the usage of direct menu and direct toolitems. Using commands together with handlers allows you to define different handlers for different scopes (Applications or Views) and you can define key bindings for the handler's associated commands.

27.3. 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 **ALT** key and allows the user to quickly access menu entries by keyboard.

You define a mnemonics by prefixing the letter intended to be the mnemonic with an ampersand in the label definition. For example the label &Save shows up as Save with the S underlined when the **Alt** key is pressed.

27.4. Default commands

If you know Eclipse 3.x you are probably searching for the predefined commands which you can re-use. The Eclipse 4 platform tries to be as lean as possible.

Eclipse 4 does not include standard commands anymore. You have to define all your commands.

27.5. 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.

The IDs of commands and handler 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 defined, 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. A more complete list of command IDs is available in `org.eclipse.ui.IWorkbenchCommandConstants`.

Table 11. 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 |
| Paste | org.eclipse.ui.edit.paste |
| 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 |
| Preferences | org.eclipse.ui.window.preferences |
| Exit | org.eclipse.ui.file.exit |

28. Dependency injection for handler classes

28.1. Behavior annotations for handler classes

The class referred to by the *direct items* or by the handlers uses *behavior annotations* to define the methods which are called by the framework in case the user selects a related user interface item. For brevity the following description use the *handler classes* term for such classes.

The behavior annotations for handler classes are described in the following table.

Table 12. Other Eclipse Behavior Annotations

| Annotation | Description |
|------------|-------------|
| | |

| | |
|-------------|--|
| @Execute | <p>Marks the method which is responsible for the action of the handler class. The framework executes the method once the related user interface element, e.g. the menu entry, is selected.</p> <div><p>Warning</p><p>Only one method is allowed to be annotated with @Execute.</p></div> |
| @CanExecute | <p>Marks a method to be visited by the Eclipse framework to check if the class is active. For example a save method might return false in this method if there is nothing to save and Eclipse disables in this case the corresponding Save menu entry.</p> <p>The default for this method is true, i.e. if the handler class can always be executed, it does not need to implement a @CanExecute method.</p> |

Note

The Eclipse runtime tries to inject all parameters which are specified by these methods.

The following example demonstrates the implementation of a handler class.

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

// Imports 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
    @CanExecute
    // the annotation
    public boolean canExecute() {
        return true;
    }
}
```

28.2. Which context is used for a handler class?

The handler class is executed with the IEclipseContext in which the handler is called. In most common cases this is in the context of the active part. The handler class is instantiated during startup

of your application in another context, i.e. in the application or the windows context.

You want the handler class to retrieve its runtime information during execution hence all required parameters should be injected into the method annotated with `@Execute`.

Warning

To ensure this NEVER use field or constructor injection in a handler class.

28.3. Evaluation of `@CanExecute`

`@CanExecute` is called by the framework if the `SWT.SHOW` event happens. This event is for example triggered if a new Part is displayed. Also if you add items to the toolbar, a timer is automatically registered by the Eclipse framework which, as of the time of this writing, executes every 400 Milliseconds. This timer will check the `@CanExecute` to enable or disable the related toolbar entry.

28.4. Scope of handlers

If a command is selected, the runtime will determine the relevant handlers for the command. The handler can be global to the application, or scoped to a window or a part. This means that you can define for example a copy handler globally in your application but define special copy handlers for certain parts. Eclipse uses the most detailed one.

Each command can have only one valid handler for a given scope. The application model allows you to define a handler for the application, for a *Window* and for a *part*.

If more than one handler is defined for a command, Eclipse will select the handler most specific to the model element.

For example if you define a handler for the "Copy" command for your Window and if you define another "Copy" handler for your Part, the runtime will select the handlers closest to model element.

Once the handler is selected, `@CanExecute` is called so the handler can determine if it is able to execute in the given context. If it returns false it will disable any menu and tool items that point to that command.

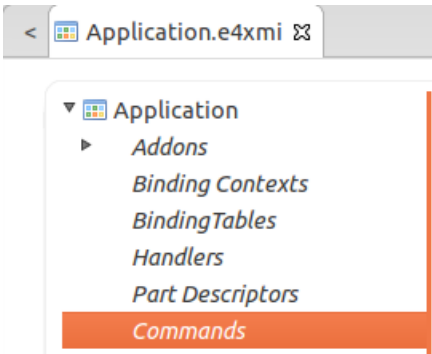
29. Tutorial: Defining and using Commands and Handlers

29.1. Overview

You will now define commands and handlers for your application. We will define our handlers for the whole application.

29.2. Defining Commands

Open the *Application.e4xmi* file and select *Commands*.



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

Table 13. Commands

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

29.3. Defining Handler classes

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

All handler classes will implement the `execute()` method.

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

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

public class SaveAllHandler {
    @Execute
    public void execute() {
        System.out.println("Called");
    }
}
```

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

- SaveAllHandler
- ExitHandler
- NewTodoHandler
- RemoveTodoHandler
- TestHandler

29.4. Defining Handlers in your model

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

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

Table 14. Handlers

| Handler ID | Command - Class |
|------------|-----------------------|
| *.save | Save - SaveAllHandler |
| *.exit | Exit - ExitHandler |

| | |
|----------|---------------------------------|
| *.new | New Todo - NewTodoHandler |
| *.remove | Remove Todo - RemoveTodoHandler |
| *.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/[CONTINUE...]  
com.example.e4.rcp.todo.handlers.SaveAllHandler
```



29.5. Adding a Menu

You will now add a Menu to your application model.

Select the `Application.e4xmi` file. To add a menu to a *Window* select your *TrimmedWindow* entry in the model and flag the *Main Menu* attribute.

Trimmed Window

Id

com.example.e4.rcp.todo.trimmedwindow

Bounds(x,y,w,h)

100

100

400

600

Label

To-do

Tooltip

Icon URI

Find ...

Main Menu

☒

To Be Rendered

☒

Visible

☒

Selected Element

Binding Contexts

Up

Down

Add ...

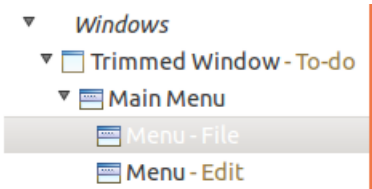
Default

Supplementary

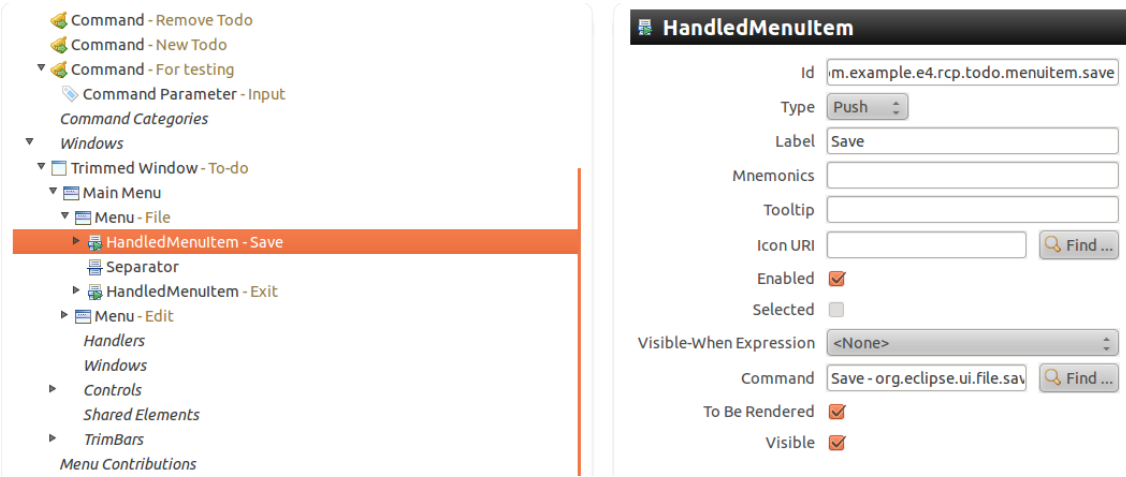
Assign the `org.eclipse.ui.main.menu` ID to your main menu.

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 `HandledMenuItem` 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 add after that an entry for the exit command.

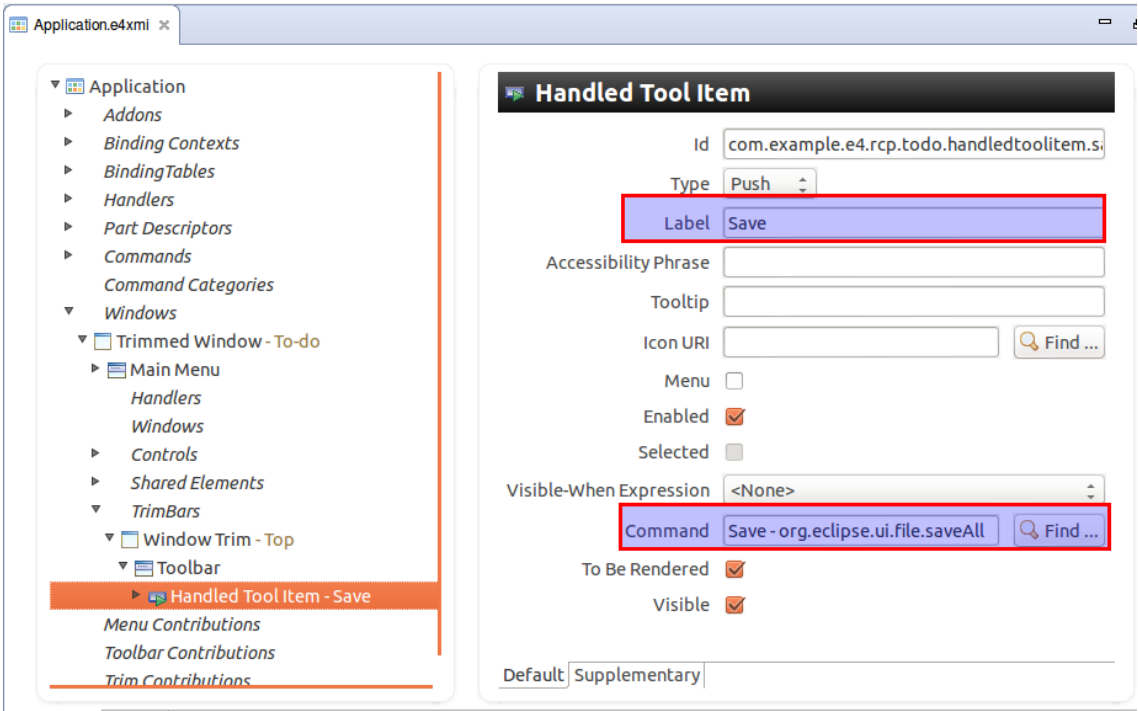
Add all other commands to the *Edit* menu.

29.6. Adding a Toolbar

Select the *TrimBars* node under your *Window* 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* 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*.



29.7. Closing the application

To test if your handler is working, change your `ExitHandler` class, so that it will close your application.

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

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

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

29.8. Simulate save

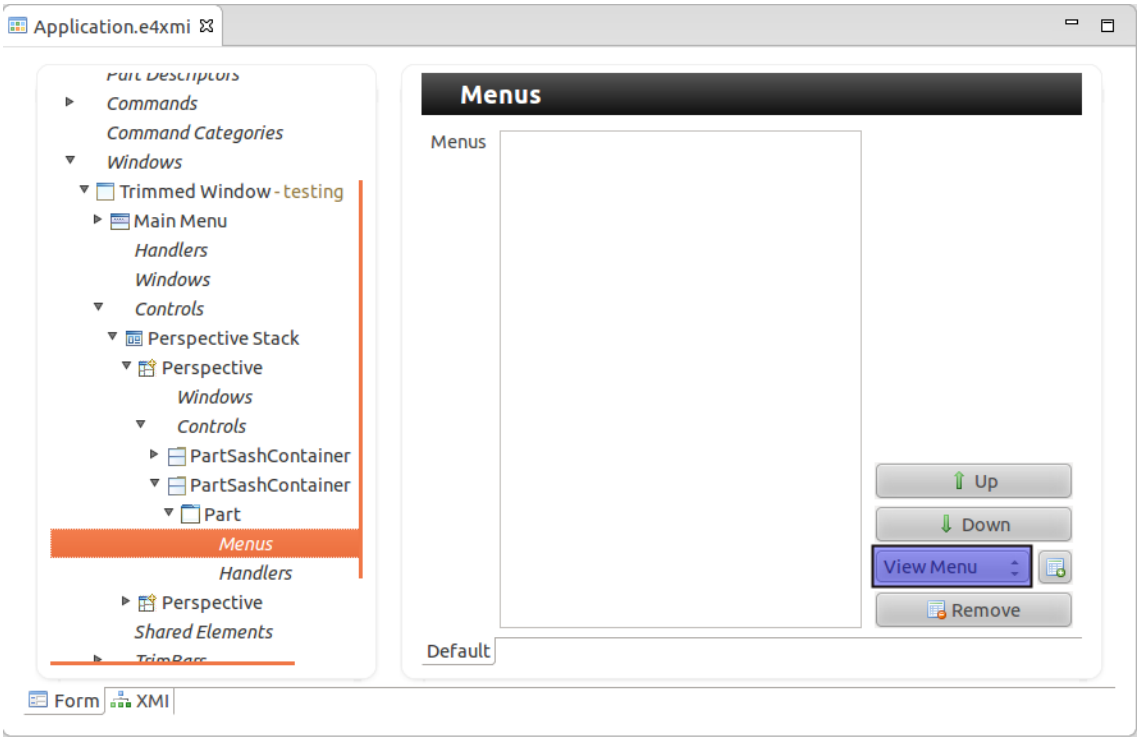
Change your `SaveAllHandler` class, so that `System.out.println` writes the following message to the console: `SaveAllHandler called..`

30. View, popup and dynamic menus

30.1. View menus

One menu in a *part* can be defined as a *view menu*. Please note that you can define a maximum of one menu as view menu.

To add such a *view menu* entry, select the *Menus* entry under our Part, select *ViewMenu* and press the *Add* button.



30.2. Define popup menu (context menu)

You can also define a popup menu for *SWT* controls. For this you define a *Popup Menu* for your *part* in the application model.

Popup Menu

Id

com.example.e4.rcp.todo.popupmenu.table

Label

Table Menu

Mnemonics

Children

Handled MenuItem

Add ...

HandledMenuItem

Up

Down

Remove

Tooltip

Icon URI

Find ...

Visible-When Expression

<None>

To Be Rendered

☒

Visible

☒

Default

Supplementary

You can then assign it via the `EMenuService` class with the `registerContextMenu(control, id)` to a `SWT` control. The `id` parameter of the `registerContextMenu` method must be the `elementId` of your *Popup Menu* model element.

The following code shows an example for the registration.

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

import javax.annotation.PostConstruct;

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

public class TodoDetailsPart {

    @PostConstruct
    public void createUi(Composite parent, EMenuService service) {
```

```

    final Text text = new Text(parent, SWT.BORDER);
    text.setText("Hello");
    // Make use to use the correct ID
    // from the application model
    service.registerContextMenu(text,
        "com.example.e4.rcp.todo.popupmenu.table");
}
}

```

If you using a JFace viewer you have to use the SWT control.

```

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

import javax.annotation.PostConstruct;

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

public class TodoDetailsPart {

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

        // more code

        /register context menu on the table
        menuService.registerContextMenu(viewer.getTable(),
            "com.example.e4.rcp.todo.popupmenu.table");
    }
}

```

If you want to implement this example you also need to have a dependency to the `org.eclipse.e4.ui.workbench.swt` plug-in in your application.

30.3. Dynamic menu

You can use the `DynamicMenuContribution` model element to point to a class which can create menu entries at runtime. Annotated for this purpose a method in this class with the `@AboutToShow` annotation. This method is called if the user selects the menu to displays its entries.

The `@AboutToHide` annotation can be used to annotate a method which is called shortly before the menu is hidden.

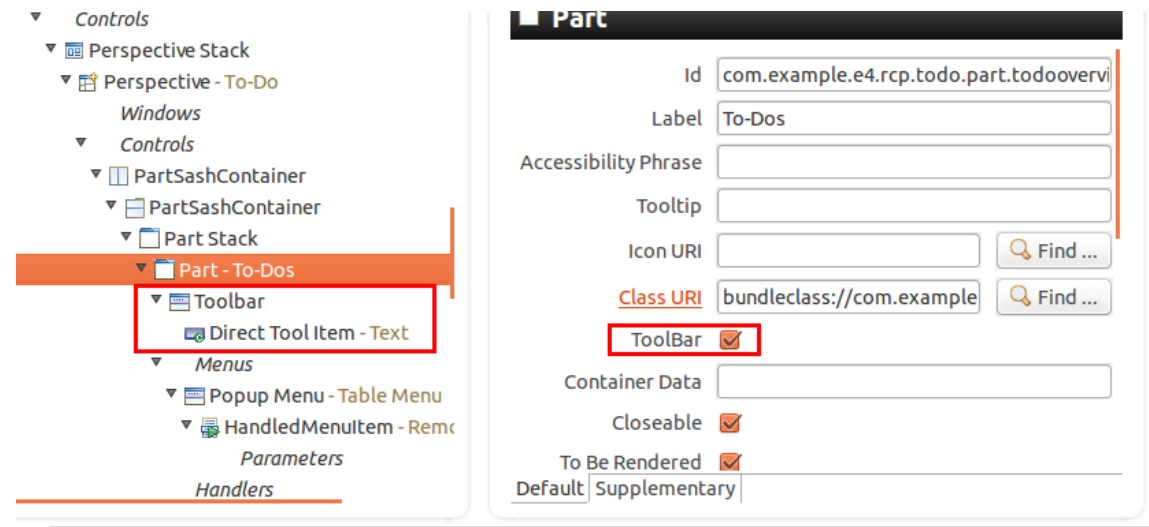
In the methods you can dynamically adjust the application model.

31. Toolbars, ToolControls and drop-down tool items

31.1. Adding toolbars to parts

To add a toolbar to a *view* set the *ToolBar* flag on the part and create the entries in the application model.

Such an example setup is displayed in the following screenshot.



31.2. ToolControls

ToolControls allows you to add items to your toolbar which point to a Java class. This Java class can create controls which are displayed in the toolbar.

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

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

import javax.annotation.PostConstruct;

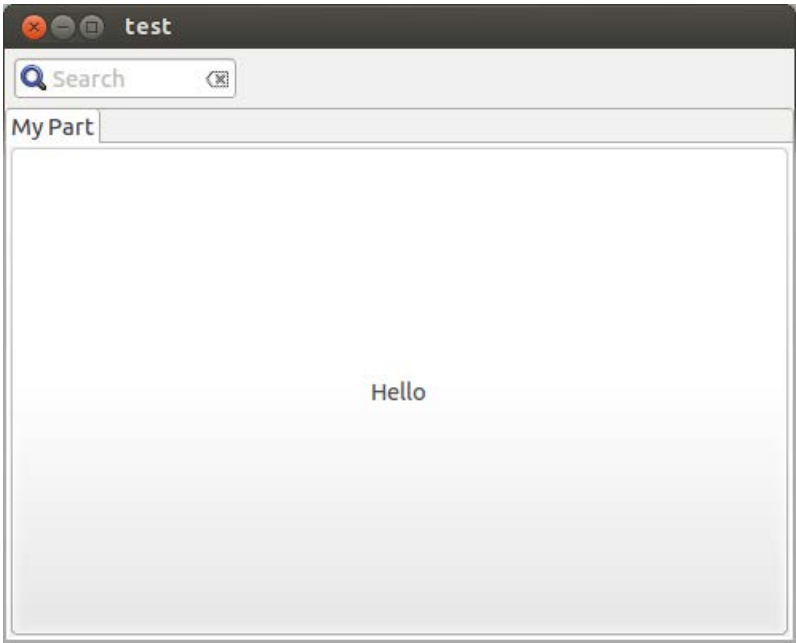
import org.eclipse.jface.layout.GridDataFactory;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Text;

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 `ToolItem` for example to your Windows Trim as depicted in the following screenshot.

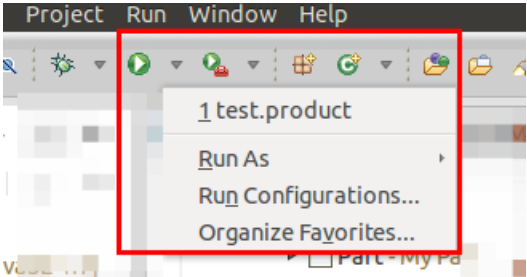


The following screenshot shows this `ToolItem` used in an example RCP application.



31.3. Drop-down tool items

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




32. More on commands and handlers

32.1. Passing parameters to commands

You can also pass parameters to commands.

To define that a command accepts a parameter, select your command and press the *Add* button in the *Parameter* section.

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

 **Command**

Id

com.example.e4.rcp.todo.test


Name

For testing

Description

Category

Parameters

 Command Parameter - Input

Find ...

Up

Down

Add ...

Remove

Default

Supplementary

BindingTables

▶ Handlers

Part Descriptors

▼ Commands

Command - Save

Command - Exit

Command - Remove Todo

Command - New Todo

▼ Command - For testing

Command Parameter - Input

Command Parameter

Id

com.example.e4.rcp.todo.commandparameter.input

Name

Input

TypeId

Optional

☒

Getting the parameter via @Named annotation in your handler is demonstrated in the following code example.

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

import javax.inject.Named;

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

public class TestHandler {
    @Execute
    public void execute(@Named("com.example.e4.rcp.todo" +
        ".commandparameter.input") String param) {
        System.out.println(param);
    }
}
```

In your menu or toolbar entries the *Name* field must be equal to the ID entry in the parameter you defined in your command. The entry in the *Value* field is passed to the command.

Trimmed Window - To-do

▼ Main Menu

Menu - File

▼ Menu - Edit

HandledMenuItem - New Todo

HandledMenuItem - Remove Todo

▼ HandledMenuItem - Test

Parameters

Parameter - com.example.e4.rcp.todo.parameter.0

Parameter

Id

menuinputparameter

Name

com.example.e4.rcp.todo.commandparameter.input

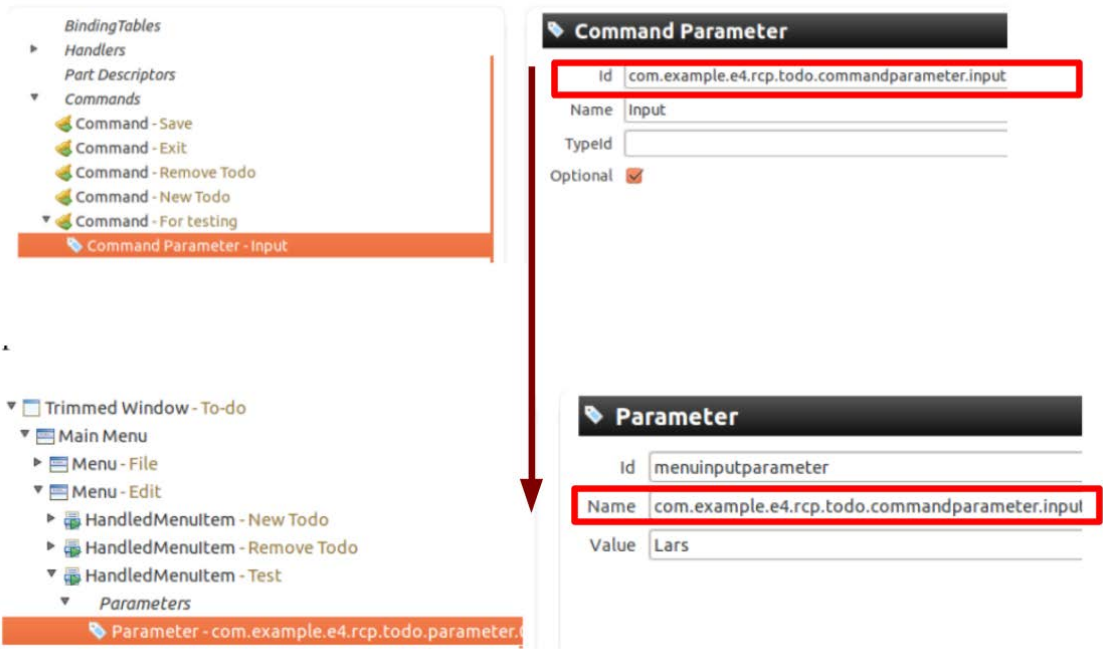
Value

Lars

Warning

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

following picture.



32.2. Usage of core expressions

The visibility of menus, toolbars and their entries can be restricted via the *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 to in the application model.

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

You can assign this core expression to your menu entry in the application model. The following

screenshot shows this for the *Test* menu entry.

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`.

```
<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>
```

This expression can be used to restrict the visibility of model elements.

HandledMenuItem

Id

com.example.e4.rcp.todo.menusitems.test

Type

Push

Label

Test

Mnemonics

Tooltip

Icon URI

Find ...

Enabled

☒

Selected

☐

Visible-When Expression

CoreExpression

Command

For testing - com.example.e4.rcp.todo.test

Find ...

To Be Rendered

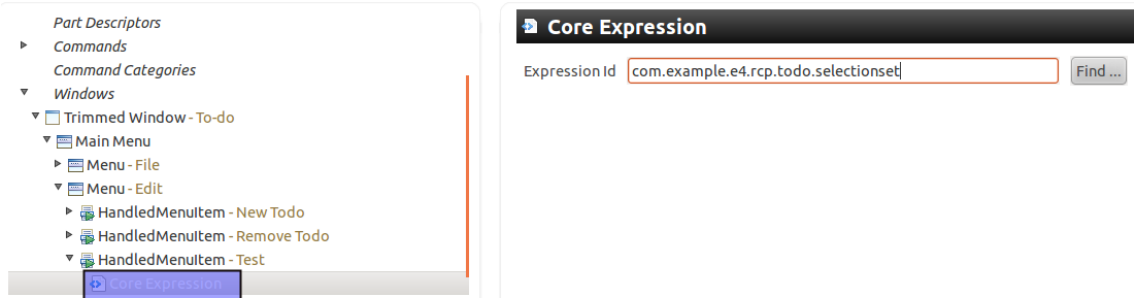
☒

Visible

☒

Default

Supplementary



This approach is similar to the definition of core expressions in Eclipse 3.x.

The values available for Eclipse 3.x are documented in the Eclipse Wiki under the following link: [Core Expressions](#) and contained in the `ISources` interface. Eclipse 4 does not always support the same variables, but the wiki might still be helpful.

32.3. 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.

Later you will learn more about modifying the `IEclipseContext` but the following code is an example `Handler` which places a value for the `myactivePartId` key in the context.

```
@Execute
public void execute(IEclipseContext context) {
    // put an example value in the context
    context.set("myactivePartId",
        "com.example.e4.rcp.ui.parts.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.

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

```
<extension
    point="org.eclipse.core.expressions.definitions">
    <definition
        id="com.example.e4.rcp.todo.todooverviewselected">
        <with
            variable="myactivePartId">
            <equals
                value="com.example.e4.rcp.ui.parts.todooverview">
            </equals>
        </with>
    </definition>
</extension>
```



33. Key bindings

33.1. Overview

It is also possible to define key bindings (shortcuts) 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 keybindings for the relevant *BindingContext* in the *BindingTable* node of your application model. A *BindingTable* is always assigned to a specific *BindingContext*. A *BindingContext* can have several *BindingTables* assigned to it.

BindingContexts are defined in a hierarchical fashion, so that keybindings in child *BindingContexts* override the matching keybinding in the parent *BindingContext*.

33.2. BindingContext entries using by JFace

The *BindingContext* is defined via its ID. A *BindingContext* can get assigned to a *Window* or a *part* in the application model. This would define which keyboard shortcuts are valid for the *Window* or the *part*.

Eclipse JFace uses predefined *BindingContext* identifier which are based on the `org.eclipse.jface.contexts.IContextIds` class. JFace distinguishes between shortcuts for dialogs, windows or both.

The following gives an overview of the supported ID and the validity of keybindings defined with reference to this Context ID.

Table 15. 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 in Dialogs |
| org.eclipse.ui.contexts.window | Key bindings valid for Windows |

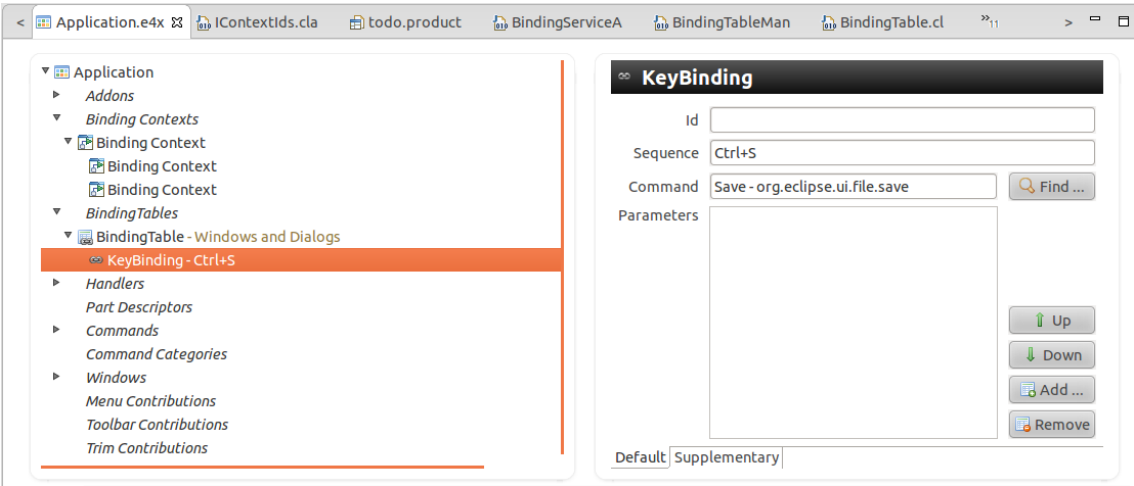
As an example, **Ctrl+C** (Copy) would be defined in *dialogAndWindows* as it is valid everywhere, but **F5** (Refresh) might only be defined for a Window and not for a Dialog.

33.3. Define Shortcuts

The *BindingTable* node in the application model allows you to define shortcuts for a specific *BindingContext*.

To define a shortcut you create a new node for *BindingTable* and define a reference for the Context ID.

In your keybinding you define the key *Sequence* and the command associated with this shortcut.



The control keys are different for the different platforms, e.g. on the Mac vs. a Linux system. For example you can use Ctrl but this would be hardcoded. It is better to use the M1 - M4 metakeys .

Table 16. 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

33.4. Key Bindings for a Part

You can assign a specific *BindingContext* to be active while a Part is active.

Part

Id

Label

Todo Overview

Accessibility Phrase

Tooltip

Icon URI

Find ...

Class URI

bundleclass://com.example.e4.rcp.todo/com.e

Find ...

ToolBar

☐

Container Data

Closeable

☐


To Be Rendered

☒

Visible

☒

Binding Contexts

 Binding Context - test

Up

Down

Add ...

Remove

Persisted State

Key

Value

Default

Supplementary

33.5. Activating Bindings

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

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

33.6. Issues with Keybinding

The keybinding implementation requires that all *parts* correctly implement `@Focus`. Eclipse requires that one control get the focus assigned.

34. Application model modifications at runtime

34.1. Creating model elements

As the model is interactive, you can change it at runtime, for example you can change the size of the current window, add Parts to your application or remove menu entries.

To add your new model elements to the application you can use the `modelService` or get existing elements injected.

34.2. Modifying existing model elements

You can also get the model elements injected and change their attribute.

35. Example for changing the application model

35.1. Example: Dynamically create a new Window

To create new model objects you can use the `MBasicFactory.INSTANCE` class. This is a factory used to create new model objects via typed `create*()` methods. For example you can create a new Window at runtime as shown in the following snippet.

```
// Create a new window and set its size
MWindow window = MBasicFactory.INSTANCE.createTrimmedWindow();
window.setWidth(200);
window.setHeight(300);

// Add new Window to the application
application.getChildren().add(window);
```

35.2. Example: Dynamically create a new Part

For example the following adds a new part to the currently active window.

```
package testing.handlers;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.model.[CONTINUE...]
    , application.descriptor.basic.MPartDescriptor;
import org.eclipse.e4.ui.model.application.ui.basic.MBasicFactory;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.e4.ui.model.application.ui.basic.MWindow;

public class AddPartHandler {
```

```

@Execute
public void execute(MWindow window) {
    MPart part = MBasicFactory.INSTANCE.createPart();
    part.setElementId("mynewid");
    part.setLabel("A new Part");
    part.setContributionURI("bundleclass://com.example." +
        "e4.rcp.todo/com.example.e4.rcp.todo.parts.TODOOverviewPart");
    window.getChildren().add(part);
}
}

```

36. Accessing and extending the Eclipse Context

36.1. Accessing the context

To access an existing context you can use dependency injection, if the relevant object is managed by the Eclipse runtime, i.e. if you are using a model object.

```

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

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

public class ShowMapHandler {
    @Execute
    public void execute(IEclipseContext context) {
        // Add objects to this local context of this handler
        // ...
    }
}

```

Alternatively if your model object extends `MContext` you can use DI to get the model object injected and use the `getContext()` method to access its context. For example `MPart`, `MWindow`, `MApplication` and `MPerspective` extend `MContext`.

```

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

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import org.eclipse.e4.core.contexts.IEclipseContext;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.swt.widgets.Composite;

// Getting the application context
// via the MApplication object

public class TodoDetailsPart {

    @PostConstruct
    public void createControls(Composite parent,
        MApplication application) {
        IEclipseContext context = application.getContext();
        // Add or access objects to and from the application context
        // ...
    }
}

```

Another example is the modification of the context of the *MWindow* model element. This context is

originally created by the `WBWRenderer` class. By default it puts an instance of the `IWindowCloseHandler` and the `ISaveHandler` interface into the local context of the *Window*. The first is responsible for the behavior of a *Window* during close, the other one for saving. For example the default `IWindowCloseHandler` would prompt you if you want to save *parts* which indicate that they have saveable content via the `MDirtyable` model attribute. You can change this default `IWindowCloseHandler` implementation via the `MWindow` model object. The following example shows an `@Execute` method in a handler implementation which overrides this class at runtime.

```
@Execute
public void execute(final Shell shell, EModelService service,
    MWindow window) {
    IWindowCloseHandler handler = new IWindowCloseHandler() {
        @Override
        public boolean close(MWindow window) {
            return MessageDialog.openConfirm(shell,
                "Close",
                "You will loose data. Really close?");
        }
    };
    window.getContext().set(IWindowCloseHandler.class, handler);
}
```

If you are outside of a model object, you still can access the OSGi context via the following:

```
public Object start() {
    // Get Bundle Information
    Bundle bundle = FrameworkUtil.getBundle(getClass());
    BundleContext bundleContext = bundle.getBundleContext();
    IEclipseContext eclipseCtx =
        EclipseContextFactory.getServiceContext(bundleContext);

    // Fill Context with information using set(String, Object)
    // ....

    // Create instance of class
    ContextInjectionFactory.make(MyPart.class, eclipseCtx);
}
```

36.2. OSGi services

You can add objects to the context via OSGi services. OSGi services are automatically available in the context. But OSGi services are global to the framework runtime — they can not access the workbench model or its context.

36.3. Objects and Context Variables

You can add key / value pairs directly to the `IEclipseContext`, for example a `todo` object under the *active* key. Adding objects to a Context can be done via the `set()` method on `IEclipseContext`. The following example creates a new context via the `EclipseContextFactory.create()` factory method call and adds some objects to it.

```
@Inject
```

```

public void addingContext(IEclipseContext context) {
    // We want to add objects to context

    // Create instance of class
    IEclipseContext myContext = EclipseContextFactory.create();

    // Putting in some values
    myContext.set("mykey1", "Hello1");
    myContext.set("mykey2", "Hello2");

    // Adding a parent relationship
    myContext.setParent(context);

    // Alternatively you can also
    // establish a parent/child
    // relationship via the
    // context.createChild() method call
}

```

A *Context variable* can be declared as *modifiable* via the `declareModifiable(key)` method call.

```

@Inject
public void addingContext(IEclipseContext context) {
    // Putting in some values
    context.set("mykey1", "Hello1");
    context.set("mykey2", "Hello2");

    // Declares the named value as modifiable
    // by descendants of this context. If the value does not
    // exist in this context, a null value is added for the name.
    context.declareModifiable("mykey1");
}

```

Modifiable *Context variables* are added to particular levels of the `IEclipseContext` hierarchy and can also be modified using the `modify()` method rather than `set()` method of the `IEclipseContext`.

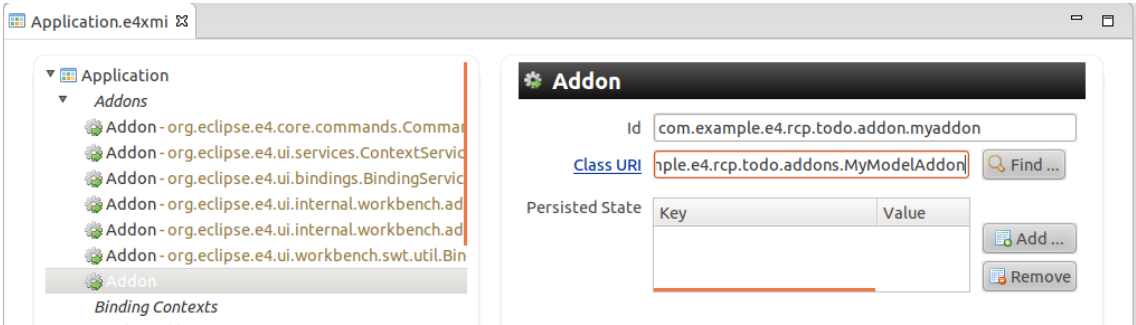
The `modify()` methods searches up the chain to find the `Context` defining the variable. If none of the `Context` on the parent chain have a value set for the name, the value will be set in this `Context`. If the key already exists in the context, then `modify()` requires that the key has been set to modifiable with the `declareModifiable()` method, if not, the method throws an exception.

You can add key/values pairs and *Context variables* at different levels of the context hierarchy so supply your application with different objects.

36.4. Model Addons

You can add *Addons* to the application model. These *Addons* contain a reference to Java classes, which can extend the context or interact with other Eclipse platform services, e.g. the `EventAdmin`.

For example you could register a model *Addon* to put some values into the context.



The following code shows an example implementation for the *Addon*.

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

import javax.annotation.PostConstruct;

import org.eclipse.e4.core.contexts.IEclipseContext;

public class MyModelAddon {
    @PostConstruct
    public void init(IEclipseContext context) {
        context.set("test1", "Hello");
    }
}
```

36.5. RunAndTrack

The `IEclipseContext` allows you via the `runAndTrack()` method to register a Java object of type `RunAndTrack`.

A `RunAndTrack` is basically a `Runnable` which has access to the context. If the runnable accesses any values in this context during its execution, the runnable will be executed again after any of those values change.

The `runAndTrack()` method allows a client to keep some external state synchronized with one or more values in this context.

The runnable does not need to be explicitly unregistered from this context when it is no longer interested in tracking changes. If this object is injected and it returns `false` from its `RunAndTrack.changed()` method it is automatically unregistered from change tracking on this context.

36.6. Context Functions

If the Eclipse dependency injection framework does not find an object in the current context it queries the defined *context functions* to see if a class has been registered to create it.

A *context functions* implement the `IContextFunction` interface or extends the abstract `ContextFuntion` class and allow you to lazily create an object.

In Eclipse 4 you would extend the `ContextFunction` class to receive an `IEclipseContext` in the `compute()` method as input.

Context Functions are contributed as OSGi Services. They implement the `IContextFunction` interface from the `org.eclipse.e4.core.contexts` package. The Eclipse 4 runtime adds *context functions* by default to the application context.

Via the `service.context.key` property, they define their key under which they are added to the context.

37. Using dependency injection for your own Java objects

37.1. Overview

Using dependency injection for your own Java objects has two flavors. First you want the Eclipse dependency container to create your own objects and then get them injected into your model objects. Secondly you want to create objects which declare their dependencies with `@Inject` and want to create it via dependency injection.

Both approaches are described here.

37.2. Using dependency injection to get your own objects

If you want the Eclipse framework to create your objects for you, annotate them with `@Creatable`. This way you are telling the Eclipse DI container that it should create a new instance of this object if it does not find an instance in the context.

The Eclipse DI container will use the constructor with the highest number of parameters of this class for which the Eclipse DI container can find values in the Eclipse context. You can also use `@Inject` on the constructor to indicate that Eclipse should try to run dependency injection also for this constructor.

For example, assume that you have the following domain model.

```
@Creatable
class Todo {
    @Inject
    public Todo(Dependent depend, YourOSGiService service) {
        // placeholder
    }
}
```

```
@Creatable
class Dependent {
public Dependent() {
// placeholder
}
}
```

If no fitting constructor is found, the Eclipse framework throws an exception.

Assuming that you have defined the `YourOSGiService` service in your application, you can get an instance of your `Todo` data model injected in a `Part`.

```
// Field Injection
@Inject Todo todo
```

37.3. Using dependency injection to create objects

Using dependency injection is not limited to the objects created by the Eclipse runtime. You can use `@Inject` in a Java class and use the dependency injection framework to create your class.

```
// Create instance of class
ContextInjectionFactory.make(MyJavaObject.class, context);
```

The `ContextInjectionFactory.make()` method creates the object. You can also put it into the context. If you want to retrieve the object from the `Context`, you can use the key you specified.

For this you can either use an existing context as described in the last section or a new context. Using a new context is preferred to avoid collision of keys and to isolate your changes in a local context.

```
IEclipseContext context = EclipseContextFactory.create();

// Add your Java objects to the context
context.set(MyDataObject.class.getName(), data);
context.set(MoreStuff.class, moreData);
```

38. Relevant tags in the application model

The following table lists the most important tags for model elements of Eclipse 4 applications.

Additional tags are defined in the `IPresentationEngine` class. It is up to the renderer implementation and model AddOns to interpret these tags. Renderer might also define additional tags. You also find more information about the available tags in the Eclipse 4 Wiki (see resources for link).

Table 17. Relevant tags of application model elements

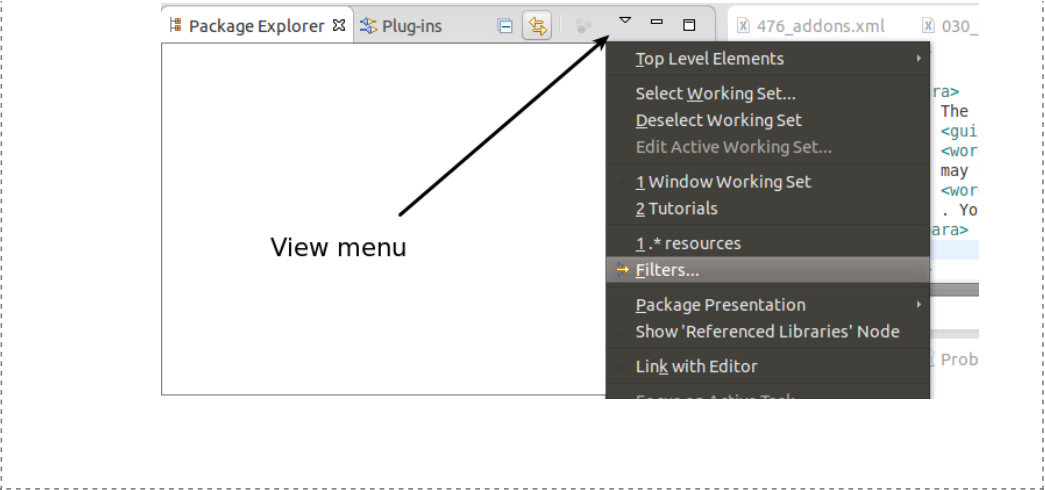
| Tag | Model element | Description |
|----------------|--------------------------|---|
| shellMaximized | Window or Trimmed Window | Window is maximized at start of the application. |
| shellMinimized | Window or Trimmed Window | Window is minimized at start of the application. |
| NoAutoCollapse | PartStack | Can be added to a <i>PartStack</i> container. With this flag the <i>PartStack</i> is not collapse by the MinMax addon even if you remove all parts from it. |
| FORCE_TEXT | ToolItem | Enforces that text and icon is shown for a toolbar item. |
| NoMove | Part | Prevents the user from moving the part (based on the DndAddON). |

39. 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*.

Note

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 version="1.0" encoding="UTF-8"?>
<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>
```

40. Tips and tricks

40.1. Overview of all available annotations in Eclipse

The following table gives an overview of the usage of these annotations for the purpose of dependency injection.

Table 18. Annotations for Dependency Injection

| Annotation | Description |
|------------|-------------|
|------------|-------------|

| | |
|----------------------|---|
| @javax.inject.Inject | Marks a field, a constructor or a method. The Eclipse framework tries to inject the parameter. |
| @javax.inject.Named | Defines the name of the key for the value which should be injected. By default the fully qualified class name is used as key. Several default values are defined as constants in the IServiceConstants interface. |
| @Optional | <p>Marks an injected value to be optional. If it can not be resolved, <code>null</code> is injected. Without <code>@Optional</code> the framework would throw an Exception.</p> <p>The specific behavior depends where <code>@Optional</code> is used:</p> <ul style="list-style-type: none">• for parameters: a <code>null</code> value will be injected;• for methods: the method calls will be skipped• for fields: the values will not be injected. |
| @Active | <p>Marks an injected value to be optional. If it can not be resolved, <code>null</code> is injected. Without <code>@Optional</code> the framework would throw an Exception.</p> <p>The specific behavior depends where <code>@Optional</code> is used:</p> <ul style="list-style-type: none">• for parameters: a <code>null</code> value will be injected;• for methods: the method calls will be skipped• for fields: the values will not be injected. |

40.2. Meta-model of the application model

The possible structure of the application model is defined by a meta-model created with the Eclipse Modeling Framework (EMF). A meta-model describes the structure of a data model, e.g. it defines which properties a Part has.

EMF is a popular general purpose modeling framework and is the basis for lots of Eclipse based projects. EMF allows to generate Java classes from a meta-model.

Eclipse EMF uses an `.ecore` file to define the meta-model.

The meta-model of the Eclipse 4 applications is stored in the `org.eclipse.e4.ui.model.workbench` plug-in inside the `model` folder. The base model definition can be found in the `UIElements.ecore` file. The Eclipse 4 model classes have been generated based on this model.

If you want to investigate this model, you could install the EMF tooling via the Eclipse update manager and import the defining plug-in into your workspace. To import a plug-in from your current Target Platform (default is the Eclipse IDE) into your workspace, use the *Plug-ins view*, right-click on a plug-in and select *Import As* → *Source Project*.

The *Application.e4xmi* file, which describes the Eclipse application model, is a persisted version of an EMF model.

40.3. Determine the command ID in a handler

In the handler class you can determine the command ID if the command was triggered via the user interface. Determining the ID is not possible, if it was triggered via the command service. The following code snippet shows how to get the command ID.

```
@Execute
public void execute(MHandledItem item) {
    MCommand command = item.getCommand();
    // Prints out the command ID
    System.out.println(command.getElementId());
}
```

40.4. Live model editor

The application model is also available at runtime. The application can access the model and change it via a defined API.

The *e4 tools project* provide a test tool for demonstrating that the application model can be modified as runtime which can be integrated into your application or used within the Eclipse IDE. Add the `org.eclipse.e4.tools.emf.liveeditor` plug-in and its dependencies to your launch configuration to make the model editor available in your application.

Afterwards you can open the model editor for your running application via the **Alt+Shift+F9** shortcut. This also works for the Eclipse 4 IDE itself.

Opening the live editor in your Eclipse 4 application requires that your application has keybindings configured.

You can change your application model directly at runtime by using the model editor. Most changes are directly applied, e.g. if you change the orientation of a `PartSashContainer` your user interface will update itself automatically. If you modifying the Eclipse IDE model you should be careful as this might put the running Eclipse IDE into a bad state.

In the live model editor, you can select the *part* element, right click on it and select *Show Control* to get the *part* highlighted.

41. Eclipse API best practice

Best practices tend to be subjective. If you disagree with certain suggestions, feel free to use your own approach.

41.1. Extending the Eclipse context

The application can use the Eclipse context for providing functionality, communication and for changing Eclipse behavior. You have several options to contribute to the context: OSGi services, context functions, context elements, context variables and ModelAddons.

The programming model of Eclipse 4 makes it relatively easy to use OSGi services, compared to Eclipse 3.x. OSGi services are effectively Singletons and do not have access to the Eclipse application context. OSGi services are a good approach for infrastructure services which are Singletons.

Context functions are OSGi services which are typically stored on the application level and have access to the Eclipse application context. If this is required, they can be used instead of pure OSGi services.

Context elements are useful, if values should be stored on certain places in the Eclipse context hierarchy. Also replacing Eclipse platform implementations is a valid use case to use context elements, e.g. to change the default window close handler.

Context functions and *context variables* are useful in situations where the variables need to change, and are different for different levels in the Eclipse context hierarchy.

Model AddOns allow you to contribute Java objects to the application context. The advantages of *Model AddOns* are that they are part of the application model, hence they are very visible components. A *Model AddOn* can register itself to events or contribute again to the Eclipse context.

41.2. Application communication

For user interface scoped communication it is recommended to use *Context variables* or the `EventAdmin` service for communicating the state.

The `EventAdmin` service is a good choice, if there is no scope involved in the communication. The strengths of `EventAdmin` is that arbitrary listeners can listen to events and that the publish / subscribe mechanism is relatively simple.

41.3. Static vs. dynamic application model

If your application model is primarily static, you should define it statically, as this static model provides a good visibility of your application at development time.

If required, add dynamic behavior. These dynamic parts can be evaluated at runtime via the live model editor.

41.4. Component based development

Eclipse and OSGi support component based development.

User interface related and core functionalities should be separated into different plug-ins.

The data model of the application should be kept in its own plug-in. Almost all plug-ins will depend on this plug-in, therefore keep it as small as possible.

41.5. Usage of your own extension points

The programming model of Eclipse 4 has reduced the need for using *Extension points* but Extension points still have valid use cases.

If you have multiple plug-ins which should contribute to a defined API, you can still define and use your own extension points.

41.6. API definition

Eclipse plug-ins explicitly declare their API via their exported packages. Publish only the packages which other plug-ins should use. This way you can later on change your internal API without affecting other plug-ins. Avoid exporting packages just for testing.

41.7. Packages vs. Plug-in dependencies

OSGi allows you to define dependencies via plug-ins or via packages.

Dependencies based on packages express an API dependency as they allow you to exchange the implementing plug-in. Dependencies based on plug-ins imply a dependency on an implementation.

Use package dependencies whenever you intent to exchange the implementing plug-in.

Package dependencies add complexity to the setup as you usually have more packages than plug-ins.

Therefore use plug-in dependencies, if there is only one implementing plug-in and an exchange of this plug-in in the near future is unlikely.

42. Closing words

I hope you enjoyed this introduction into the Eclipse 4 framework. Of course there is much more, check out the "Eclipse 4 development" section under my lists of [Eclipse Plug-in and Eclipse RCP Tutorials](#) section.

43. Thank you

Please help me to support this article:

 [Flickr this!](#)




44. Questions and Discussion

Before posting questions, please see the [vogella FAQ](#). If you have questions or find an error in this article please use the [www.vogella.com Google Group](#). I have created a short list [how to create good questions](#) which might also help you.

45. Links and Literature

45.1. Source Code

[Source Code of Examples](#)

45.2. Eclipse 4

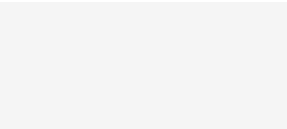
[Eclipse 4 RCP Wiki](#)

[Eclipse 4 RCP FAQ](#)

[vogella Eclipse Tutorials](#)

[Eclipse wiki with Eclipse 4 tutorials](#)

[Eclipse 4 Dependency Injection Wiki](#)



Eclipse 4 RCP Wiki for tags for the Application model

Eclipse 4 Build Schedule

