

vogella series



Eclipse Rich Client Platform

Third Edition: Revised edition based on Eclipse 4.4

Written by the Eclipse Platform UI and e4 project co-lead



Lars Vogel

Foreword by Mike Milinkovich, Executive Director, Eclipse Foundation

Eclipse Rich Client Platform

Table of Contents

[Foreword](#)

[Preface](#)

[1. Welcome](#)

[2. About the author - Lars Vogel](#)

[3. About the vogella company](#)

[4. Screenshots](#)

[5. Shortcuts on a Mac](#)

[6. How this book is organized](#)

[7. Prerequisites](#)

[8. Errata](#)

[9. Exercises, optional exercises and examples](#)

[10. Long lines](#)

[11. Example code](#)

[12. Acknowledgment](#)

[I. The Eclipse open source project](#)

[1. Introduction to the Eclipse project](#)

[1.1. The Eclipse community and projects](#)

[1.2. A short Eclipse history](#)

[1.3. Eclipse releases](#)

[2. Eclipse project structure](#)

[2.1. Eclipse foundation](#)

[2.2. Staff of the Eclipse foundation](#)

[3. The Eclipse Public License](#)

[3.1. Source code and the Eclipse Public License](#)

[3.2. Intellectual property cleansing of Eclipse code](#)

[II. The concepts behind Eclipse applications](#)

[4. Eclipse plug-ins and applications](#)

[4.1. What is an Eclipse RCP application?](#)

- [4.2. Eclipse software components - Plug-ins](#)
- [4.3. Advantages of developing Eclipse plug-ins](#)
- [4.4. What is the Eclipse Rich Client Platform \(Eclipse RCP\)?](#)

[5. Important Eclipse projects for RCP](#)

- [5.1. The Eclipse Platform project](#)
- [5.2. The Eclipse e4 project](#)
- [5.3. The Eclipse Plug-in Development Environment \(PDE\)](#)
- [5.4. The e4 tools project](#)

[6. Architecture of Eclipse](#)

- [6.1. Architecture of Eclipse based applications](#)
- [6.2. Core components of the Eclipse platform](#)
- [6.3. Compatibility layer for Eclipse 3.x plug-ins](#)
- [6.4. Eclipse API and internal API](#)
- [6.5. Important configuration files for Eclipse plug-ins](#)

[III. Setting up an Eclipse RCP development environment](#)

- [7. Install Java for Eclipse RCP development](#)
 - [7.1. Java requirements of the Eclipse IDE](#)
 - [7.2. Check installation](#)
 - [7.3. Install Java on Ubuntu](#)
 - [7.4. Install Java on MS Windows](#)
 - [7.5. Installation problems and other operating systems](#)
 - [7.6. Validate installation](#)
 - [7.7. How can you tell you are using a 32 bit or 64 bit version of Java?](#)

[8. Install Eclipse IDE for RCP development](#)

- [8.1. Download the Eclipse Software Development Kit \(SDK\)](#)
- [8.2. Install the Eclipse IDE](#)
- [8.3. Starting the Eclipse IDE](#)
- [8.4. Appearance](#)

[9. Install the e4 tools](#)

- [9.1. Requirements](#)
- [9.2. Install the e4 tools from the vogella GmbH update site](#)

[9.3. Install the e4 tools from the Eclipse.org update site](#)

[10. Start Eclipse and use a new workspace](#)

[10.1. About the Eclipse IDE](#)

[10.2. Workspace](#)

[10.3. Review the Eclipse 4 model editor preferences](#)

[11. Enable Java access to all plug-ins](#)

[11.1. Filtering by the Java tools](#)

[11.2. Include all plug-ins in Java search](#)

[IV. Eclipse applications and product configuration files](#)

[12. Product configuration file](#)

[12.1. The Product configuration file and the application](#)

[12.2. Creating a new product configuration file](#)

[12.3. Using the product editor](#)

[12.4. Define the plug-ins which are included in this product](#)

[13. Branding](#)

[13.1. Splash screen](#)

[13.2. Icons, launcher name and program arguments](#)

[13.3. Product configuration limitations](#)

[14. The usage of run configurations](#)

[14.1. What are run configurations?](#)

[14.2. Reviewing run configurations](#)

[14.3. Run arguments](#)

[14.4. Launch configuration and Eclipse products](#)

[15. Common launch problems](#)

[15.1. Checklist for common launch problems](#)

[15.2. Finding missing plug-in dependencies during a product launch](#)

[16. Exercise: Create and run an RCP application](#)

[16.1. Target](#)

[16.2. Create project](#)

[16.3. Launch your Eclipse application via the product file](#)

[16.4. Validating](#)

[17. Features and feature projects](#)

[17.1. What are feature projects and features?](#)

[17.2. Creating a feature](#)

[17.3. Feature or plug-in based products](#)

[17.4. Advantages of using features](#)

[18. Exercise: Use a feature based product](#)

[18.1. Create a feature project](#)

[18.2. Include plug-in into feature project](#)

[18.3. Change product configuration file to use features](#)

[18.4. Add features as dependency to the product](#)

[18.5. Start the application via the product](#)

[V. Deploying Eclipse applications](#)

[19. Deployment of your RCP application](#)

[19.1. Creating a stand-alone version of the application](#)

[19.2. Exporting via the product file](#)

[19.3. Defining which artifacts are included in the export](#)

[19.4. Mandatory plug-in artifacts in build.properties](#)

[19.5. Export for multiple platforms via the delta pack](#)

[19.6. More about the target platform](#)

[19.7. Including the required JRE into the export](#)

[19.8. Headless build](#)

[20. Common product export problems](#)

[20.1. Problems with the export and log files](#)

[20.2. Export problem number #1: export folder is not empty](#)

[20.3. Checklist for common export problems](#)

[21. Exercise: Export your product](#)

[21.1. Export your product](#)

[21.2. Validate that the exported application starts](#)

22. Exercise: Splash screen and launcher name

22.1. Using a static splash screen

22.2. Include a splash screen into the exported application

22.3. Change launcher name

VI. Application model

23. Eclipse application model

23.1. What is the application model?

23.2. Scope of the application model

23.3. How do you define the application model?

24. Important user interface model elements

24.1. Window

24.2. Parts

24.3. Part container

24.4. Perspective

25. Connecting model elements to classes and resources

25.1. URI patterns or classes and resources

25.2. Connect model elements to classes

25.3. Connect model elements to resources like icons

26. Model objects and the runtime application model

26.1. Model objects

26.2. Runtime application model

27. Using the model spy

27.1. Analyzing the application model with the model spy

27.2. Installation requirements

27.3. Model spy and the Eclipse IDE

28. Exercise: Create an Eclipse plug-in

28.1. Target

28.2. Creating a plug-in project

28.3. Validate the result

29. Exercise: From plug-in to Eclipse RCP

29.1. Target

29.2. Create a project to host the product configuration file

29.3. Create a product configuration file

29.4. Configure the start levels

29.5. Create a feature project

29.6. Enter the feature dependencies in product

29.7. Remove the version dependency from the features in product

29.8. Create an application model

29.9. Add a window to the application model

29.10. Start the application

30. Exercise: Configure the deletion of persisted model data

30.1. Delete the persisted user changes at startup

30.2. Why is this setting necessary?

31. Exercise: Modeling a user interface

31.1. Desired user interface

31.2. Open the Application.e4xmi file

31.3. Add a perspective

31.4. Add part sash and part stack containers

31.5. Create the parts

31.6. Validate the user interface

32. Exercise: Connect Java classes with the parts

32.1. Create a new package and some Java classes

32.2. Connect the Java classes with your parts

32.3. Validating

33. Exercise: Enter the dependencies

33.1. Add the plug-in dependencies

33.2. Validating

33.3. More on dependencies

34. Optional Exercise: Using the model spy

- [34.1. Target](#)
- [34.2. Adding the model spy to your runtime configuration](#)
- [34.3. Use the model spy](#)
- [34.4. Cleanup - remove model spy from your run configuration](#)

VII. Annotations and dependency injection

- [35. The concept of dependency injection](#)
- [35.1. What is dependency injection?](#)
- [35.2. Using annotations to describe class dependencies](#)
- [35.3. Where can objects be injected into a class?](#)
- [35.4. Order in which dependency injection is performed on a class](#)

36. Dependency injection in Eclipse

- [36.1. Define class dependencies in Eclipse](#)
- [36.2. Annotations to define class dependencies in Eclipse](#)
- [36.3. On which objects does Eclipse perform dependency injection?](#)
- [36.4. Dynamic dependency injection based on key / value changes](#)

VIII. Dependency injection and the Eclipse context

- [37. The Eclipse context](#)
- [37.1. What is the Eclipse context?](#)
- [37.2. Relationship definition in the Eclipse context](#)
- [37.3. Which model elements have a local context?](#)
- [37.4. Life cycle of the Eclipse context](#)

38. Object selection during dependency injection

- [38.1. How are objects selected for dependency injection](#)
- [38.2. How to access the model objects?](#)
- [38.3. Default entries in the Eclipse context](#)
- [38.4. Qualifiers for accessing the active part or shell](#)
- [38.5. Tracking a child context with @Active](#)

IX. Behavior annotations

- [39. Using annotations to define behavior](#)
- [39.1. API definition](#)
- [39.2. API definition via inheritance](#)

[39.3. API definition via annotations](#)

[39.4. Behavior annotations imply method dependency injection](#)

[39.5. Use the @PostConstruct method to build the user interface](#)

[39.6. Why is the @PostConstruct method not called?](#)

[40. Exercise: Using @PostConstruct](#)

[40.1. Implement an @PostConstruct method](#)

[40.2. Validating](#)

[X. Eclipse modularity based on OSGi](#)

[41. Software modularity with OSGi](#)

[41.1. What is software modularity?](#)

[41.2. What is OSGi?](#)

[41.3. OSGi implementations](#)

[41.4. Plug-in or bundles as software component](#)

[41.5. Naming convention: simple plug-in](#)

[41.6. Find the plug-in for a certain class](#)

[42. OSGi metadata](#)

[42.1. The manifest file \(MANIFEST.MF\)](#)

[42.2. Bundle-SymbolicName and Version](#)

[42.3. Semantic Versioning with OSGi](#)

[43. Defining the dependencies of a plug-in](#)

[43.1. Specifying plug-in dependencies via the manifest file](#)

[43.2. Life cycle of plug-ins in OSGi](#)

[43.3. Dynamic imports of packages](#)

[44. Defining an API](#)

[44.1. Specifying the API of a plug-in](#)

[44.2. Provisional API](#)

[44.3. Provisional API with exceptions via x-friends](#)

[45. Using the OSGi console](#)

[45.1. The OSGi console](#)

[45.2. Required bundles](#)

[45.3. Telnet](#)

[45.4. Access to the Eclipse OSGi console](#)

[46. Exercise: Data model plug-in](#)

[46.1. Target of the exercise](#)

[46.2. Create the plug-in for the data model](#)

[46.3. Create the base class](#)

[46.4. Generate constructors](#)

[46.5. Generate getter and setter methods](#)

[46.6. Generate `toString\(\)`, `hashCode\(\)` and `equals\(\)` methods](#)

[46.7. Write a `copy\(\)` method](#)

[46.8. Create the interface for the todo service](#)

[46.9. Define the API of the model plug-in](#)

[47. Exercise: Service plug-in](#)

[47.1. Target of the exercise](#)

[47.2. Create a data model provider plug-in \(service plug-in\)](#)

[47.3. Define the dependencies in the service plug-in](#)

[47.4. Provide an implementation of the `ITodoService` interface](#)

[47.5. Create a factory](#)

[47.6. Export the package in the service plug-in](#)

[48. Exercise: Use the new plug-ins](#)

[48.1. Update the plug-in dependencies](#)

[48.2. Update the product configuration \(via your feature\)](#)

[48.3. Use the data model provider in your parts](#)

[48.4. Validate your implementation](#)

[48.5. Review your implementation](#)

[49. OSGi fragments and fragment projects](#)

[49.1. What are fragments in OSGi?](#)

[49.2. Typical use cases for fragments](#)

[XI. Using the service layer of OSGi](#)

[50. OSGi service introduction](#)

[50.1. What are OSGi services?](#)

[50.2. Life cycle status for providing services](#)

[50.3. Best practices for defining services](#)

[50.4. Service properties](#)

[50.5. Service priorities](#)

[51. Defining OSGi services](#)

[51.1. Defining declarative services](#)

[51.2. Required bundles](#)

[51.3. Activating the declarative service plug-in](#)

[52. Steps to declare an OSGi service](#)

[52.1. Defining the service interface](#)

[52.2. Providing a service implementation](#)

[52.3. Service declaration with a component definition file](#)

[52.4. Reference to the service in the MANIFEST.MF file](#)

[52.5. Low-level OSGi service API](#)

[53. Eclipse RCP and OSGi services](#)

[53.1. Using declarative services in RCP applications](#)

[53.2. OSGi services and Eclipse dependency injection](#)

[54. Exercise: Define and use an OSGi service](#)

[54.1. Target of this exercise](#)

[54.2. Define the component definition file](#)

[54.3. Set the lazy activation flag for the service plug-in](#)

[54.4. Get the ITodoService injected](#)

[54.5. Possible issues: ITodoService cannot get injected](#)

[54.6. Clean-up the service implementation](#)

[54.7. Review the service implementation](#)

[55. Optional exercise: Create an image loader service](#)

[55.1. Target of this exercise](#)

[55.2. Creating a new plug-in](#)

[55.3. Creating a service implementation](#)

[55.4. Defining a new service](#)

[55.5. Adding the new plug-in to your feature project](#)

[55.6. Exporting the API](#)

[55.7. Add a MANIFEST.MF dependency to your new plug-in](#)

[55.8. Using the new service](#)

[55.9. Reviewing the implementation](#)

XII. User interface development with SWT

[56. Standard Widget Toolkit](#)

[56.1. What is SWT?](#)

[56.2. Eclipse applications and SWT](#)

[56.3. Display and Shell](#)

[56.4. Event loop](#)

[56.5. Relationship to JFace](#)

[56.6. Using SWT in a plug-in project](#)

57. Using SWT widgets

[57.1. Available widgets in the SWT library](#)

[57.2. Memory management](#)

[57.3. Constructing widgets](#)

[57.4. Basic containers](#)

[57.5. Event Listener](#)

58. Using layout managers in SWT

[58.1. The role of a layout manager](#)

[58.2. Layout Data](#)

[58.3. FillLayout](#)

[58.4. RowLayout](#)

[58.5. GridLayout](#)

[58.6. Using GridDataFactory](#)

[58.7. Tab order of elements](#)

[58.8. Example: Using layout manager](#)

59. SWT widget examples and controls

[59.1. SWT snippets and examples](#)

[59.2. The Nebula and Opal widgets](#)

60. SWT Designer

[60.1. SWT Designer \(WindowBuilder\)](#)

[60.2. Install SWT Designer](#)

[60.3. Using SWT Designer](#)

[61. Exercise: Getting started with SWT Designer](#)

[61.1. Installation](#)

[61.2. Building an user interface](#)

[61.3. Creating an event handler](#)

[61.4. Review the generated code](#)

[62. Exercise: Build a first SWT UI](#)

[62.1. TodoOverviewPart](#)

[62.2. Example code for TodoOverviewPart](#)

[63. Exercise: Implement a UI for TodoDetailsPart](#)

[63.1. TodoDetailsPart](#)

[63.2. Implement focus setting for one of your widgets](#)

[63.3. Example code for TodoDetailsPart](#)

[64. Exercise: Prepare TodoDetailsPart for data](#)

[64.1. Preparing for data](#)

[64.2. Add dependency](#)

[64.3. Prepare for dependency injection](#)

[64.4. Usage of this method](#)

[65. Exercise: Using the SWT Browser widget](#)

[65.1. Implementation](#)

[65.2. Solution](#)

[XIII. User interface development with JFace](#)

[66. JFace](#)

[66.1. What is Eclipse JFace?](#)

[66.2. JFace resource manager for Colors, Fonts and Images](#)

[66.3. ControlDecoration](#)

[66.4. User input help with field assistance](#)

[67. The JFace viewer framework](#)

[67.1. Purpose of the JFace viewer framework](#)

[67.2. Standard JFace viewer](#)

[67.3. Standard content and label provider](#)

[67.4. JFace ComboViewer](#)

[68. Using tables](#)

[68.1. Using the JFace TableViewer](#)

[68.2. Content provider for JFace tables](#)

[68.3. Columns and label provider](#)

[68.4. Reflect data changes in the viewer](#)

[68.5. Selection change listener](#)

[68.6. Column editing support](#)

[68.7. Filtering data](#)

[68.8. Sorting data with ViewerComparator](#)

[68.9. TableColumnLayout](#)

[68.10. StyledCellLabelProvider and OwnerDrawLabelProvider](#)

[68.11. Table column menu and hiding columns](#)

[68.12. Tooltips for viewers](#)

[68.13. Virtual tables with LazyContentProvider](#)

[68.14. Alternative table implementations](#)

[69. Exercise: Using TableViewer](#)

[69.1. Create a JFace TableViewer for the Todo items](#)

[69.2. Add dependencies](#)

[69.3. Create implementation](#)

[70. Exercise: Using more viewer functionality](#)

[70.1. Add a filter to the table](#)

[70.2. Add a ControlDecoration to the TodoDetailsPart](#)

[71. Optional exercise: Using ComboViewer](#)

[71.1. Target](#)

[71.2. Implementation](#)

[71.3. Validation](#)

72. Using trees

72.1. Using viewers to display a tree

72.2. Selection and double-click listener

73. Optional exercises: Using TreeViewer

73.1. Create a new application

73.2. Add an image file

73.3. Create a part

73.4. Validating

XIV. Defining menus and toolbars

74. Menu and toolbar application objects

74.1. Adding menu and toolbar entries

74.2. What are commands and handlers?

74.3. Mnemonics

74.4. Standard commands

74.5. Naming schema for command and handler IDs

75. Dependency injection for handler classes

75.1. Handler classes and their behavior annotations

75.2. Which context is used for a handler class?

75.3. Scope of handlers

75.4. Evaluation of @CanExecute

75.5. Learn about the event system

76. Exercise: Adding a menu and menu entries

76.1. Target of this exercise

76.2. Create command model elements

76.3. Creating the handler classes

76.4. Creating handler model elements

76.5. Adding a menu

76.6. Implement a handler class for exit

76.7. Validating

76.8. Possible issue: Exit menu entry on a Mac OS

77. Exercise: Adding a toolbar

[77.1. Target of this exercise](#)

[77.2. Adding a toolbar](#)

[77.3. Validating](#)

[78. View, popup and dynamic menus](#)

[78.1. View menus](#)

[78.2. Popup menu \(context menu\)](#)

[78.3. Dynamic menu and toolbar entries](#)

[78.4. Creating dynamic menu entries](#)

[79. Exercise: Add a context menu to a table](#)

[79.1. Target](#)

[79.2. Dependencies](#)

[79.3. Implementation](#)

[80. Toolbars, ToolControls and drop-down tool items](#)

[80.1. Adding toolbars to parts](#)

[80.2. ToolControls](#)

[80.3. Drop-down tool items](#)

[81. More on commands and handlers](#)

[81.1. Passing parameters to commands](#)

[81.2. Usage of core expressions](#)

[81.3. Evaluate your own values in core expressions](#)

[81.4. Example for dynamic model contributions](#)

[XV. Using key bindings](#)

[82. Key bindings](#)

[82.1. Using key bindings in your application](#)

[82.2. JFace default values for binding contexts](#)

[82.3. Define Shortcuts](#)

[82.4. Activate bindings](#)

[82.5. Key bindings for a part](#)

[83. Exercise: Define key bindings](#)

[83.1. Create binding context entries](#)

[83.2. Create key bindings for a BindingContext](#)

[XVI. Dialogs and wizards](#)

[84. Dialogs](#)

[84.1. Dialogs in Eclipse](#)

[84.2. SWT dialogs](#)

[84.3. JFace Dialogs](#)

[85. Exercise: Dialogs](#)

[85.1. Confirmation dialog at exit](#)

[85.2. Create a password dialog](#)

[86. Wizards](#)

[86.1. What is a wizards](#)

[86.2. Wizards and WizardPages](#)

[86.3. Starting the Wizard](#)

[86.4. Changing the page order](#)

[86.5. Working with data in the wizard](#)

[86.6. Updating the Wizard buttons from a WizardPage](#)

[87. Exercise: Create a wizard](#)

[87.1. Create classes for the wizard](#)

[87.2. Adjust part](#)

[87.3. Adjust handler implementation](#)

[87.4. Validating](#)

[XVII. Data Binding with JFace](#)

[88. Data Binding with JFace](#)

[88.1. What are Data Binding frameworks?](#)

[88.2. JFace Data Binding](#)

[88.3. JFace Data Binding Plug-ins](#)

[89. Listening to changes](#)

[89.1. Ability to listen to changes in UI components](#)

[89.2. Ability to listen to changes in the domain model](#)

[89.3. Property change support](#)

[89.4. Data Binding and Java objects without change notification](#)

[90. Create bindings](#)

[90.1. Observing properties with the IObservableValue interface](#)

[90.2. Creating instances of the IObservableValue](#)

[90.3. Connecting properties with the DataBindingContext](#)

[90.4. Example: how to observe properties](#)

[90.5. Observing nested properties](#)

[91. Updates, convertors and validators](#)

[91.1. UpdateValueStrategy](#)

[91.2. Converter](#)

[91.3. Validator](#)

[92. More on bindings](#)

[92.1. ControlDecorators](#)

[92.2. Placeholder binding with WritableValue](#)

[92.3. Listening to all changes in the binding](#)

[92.4. More information on Data Binding](#)

[93. Exercise: Data Binding for SWT widgets](#)

[93.1. Add the plug-in dependencies](#)

[93.2. Implement the property change support](#)

[93.3. Remove the modification listeners in your code](#)

[93.4. Add a field for the data binding context to TodoDetailsPart](#)

[93.5. Implement data binding in TodoDetailsPart](#)

[93.6. Validating](#)

[94. Data Binding for JFace viewer](#)

[94.1. Binding Viewers](#)

[94.2. Observing list details](#)

[94.3. ViewerSupport](#)

[94.4. Master Detail binding](#)

[94.5. Chaining properties](#)

[95. Exercise: Data Binding for viewers](#)

- [95.1. Implement Data Binding for the viewer](#)
- [95.2. Clean up old code](#)
- [95.3. Validating](#)

[XVIII. Using Eclipse services](#)

- [96. Eclipse platform services](#)
- [96.1. What are Eclipse platform services?](#)
- [96.2. Overview of the available platform services](#)

[97. Implementation](#)

- [97.1. How are Eclipse platform services implemented?](#)
- [97.2. References](#)

[XIX. Selection Service](#)

- [98. Selection service](#)
- [98.1. Usage of the selection service](#)
- [98.2. Changing the current selection](#)
- [98.3. Getting the selection](#)

[99. Exercise: Selection service](#)

- [99.1. Target of this exercise](#)
- [99.2. Retrieving the selection service](#)
- [99.3. Setting the selection in TodoOverviewPart](#)
- [99.4. Review TodoDetailsPart](#)
- [99.5. Validate selection propagation](#)

[100. Exercise: Selection service for deleting data](#)

- [100.1. Implement the RemoveTodoHandler handler](#)
- [100.2. Validate that the deletion works](#)
- [100.3. Test the context menu for deletion](#)

[XX. Model service and model modifications at runtime](#)

- [101. Model service](#)
- [101.1. What is the model service?](#)
- [101.2. How to access the model service](#)
- [101.3. Cloning elements or snippets](#)

[101.4. Searching model elements](#)

[102. Modifying the application model at runtime](#)

[102.1. Creating model elements](#)

[102.2. Modifying existing model elements](#)

[103. Example for application model modifications](#)

[103.1. Example: Search for a perspective and change its attributes](#)

[103.2. Example: Dynamically create a new window](#)

[103.3. Example: Dynamically create a new part](#)

[104. Exercise: Creating dynamic menu entries](#)

[104.1. Target of this exercise](#)

[104.2. Create handler class](#)

[104.3. Create class for DynamicMenuContribution](#)

[104.4. Add DynamicMenuContribution model element](#)

[104.5. Validating](#)

[XXI. Part service and implementing editors](#)

[105. Using the part service](#)

[105.1. What is the part service?](#)

[105.2. How to access the part service](#)

[105.3. Example: Showing and hiding parts](#)

[105.4. Example: Switching perspectives](#)

[105.5. Using part descriptors](#)

[105.6. Example: Part descriptors and creating parts dynamically](#)

[106. Implementing editors](#)

[106.1. Parts which behave similar to editors](#)

[106.2. MDirtyable and @Persist](#)

[106.3. Use part service to trigger save in editors](#)

[106.4. MPart and multiple editors](#)

[106.5. MInputPart](#)

[106.6. Code examples for editor implementations](#)

[107. Exercise: Implement an editor](#)

[107.1. Add the plug-in dependencies](#)

[107.2. Convert TodoDetailsPart to an editor](#)

[107.3. Implement the save handler](#)

[107.4. Validating](#)

[107.5. Confirmation dialog for modified data](#)

[108. Exercise: Enable handlers and avoiding data loss](#)

[108.1. Enable the save handler only if necessary](#)

[108.2. Enable the deletion handler only if necessary](#)

[108.3. Avoid data loss in your ExitHandler](#)

[109. Exercise: Using multiple perspectives](#)

[109.1. Target of this exercise](#)

[109.2. Create a new perspective](#)

[109.3. Create new menu entries](#)

[109.4. Using command parameters to define the perspective ID](#)

[110. Exercise: Sharing elements between perspectives](#)

[110.1. Target](#)

[110.2. Using shared parts between perspectives](#)

[111. Optional exercise: Dynamic creation of parts based on a part descriptor](#)

[111.1. Create a class for the part descriptor](#)

[111.2. Create the part descriptor model element](#)

[111.3. Create a handler for creating new parts](#)

[112. Exercise: Implement multiple editors](#)

[112.1. Prerequisites](#)

[112.2. New menu entries](#)

[112.3. Validate ID of the PartStack](#)

[112.4. Add a handler and a part implementation](#)

[112.5. Validate multiple editor implementation](#)

[XXII. Handler and command services](#)

[113. Command and handler service](#)

[113.1. Purpose of the command and handler service](#)

[113.2. Access to command and handler service](#)
[113.3. Example for executing a command](#)
[113.4. Example for assigning a handler to a command](#)

[114. Optional exercise: Using handler service](#)

[114.1. Delete Todos only via the handler](#)
[114.2. Implementation](#)

[XXIII. Asynchronous processing](#)

[115. Threading in Eclipse](#)
[115.1. Concurrency](#)
[115.2. Main thread](#)
[115.3. Using dependency injection and UISynchronize](#)
[115.4. Eclipse Jobs API](#)
[115.5. Priorities of Jobs](#)
[115.6. Blocking the UI and providing feedback](#)

[116. Progress reporting](#)

[116.1. IProgressMonitor](#)
[116.2. Reporting progress in Eclipse RCP applications](#)

[117. Exercise: Using asynchronous processing](#)

[117.1. Simulate delayed access](#)
[117.2. Use asynchronous processing](#)
[117.3. Validating](#)
[117.4. Remove delay](#)

[XXIV. Event service for message communication](#)

[118. Eclipse event notifications](#)
[118.1. Event based communication](#)
[118.2. The event bus of Eclipse](#)
[118.3. Event service](#)
[118.4. Required plug-ins to use the event service](#)
[118.5. Sending and receiving events](#)
[118.6. Usage of the event system](#)
[118.7. Asynchronous processing and the event bus](#)

119. Exercise: Event notifications

[119.1. Creating a plug-in for event constants](#)

[119.2. Add the new plug-in to your product](#)

[119.3. Enter the plug-in dependencies](#)

[119.4. Send out notifications](#)

[119.5. Receive updates in your parts](#)

[119.6. Validating](#)

[119.7. Review the implementation](#)

XXV. Extending and modifying the Eclipse context

[120. Accessing and extending the Eclipse context](#)

[120.1. Accessing the context](#)

[120.2. Objects and context variables](#)

[120.3. Replacing existing objects in the IEclipseContext](#)

[120.4. Accessing the IEclipseContext hierarchy from OSGi services](#)

[120.5. Model add-ons](#)

[120.6. RunAndTrack](#)

[121. Using dependency injection for your Java objects](#)

[121.1. Creating and injecting custom objects](#)

[121.2. Using dependency injection to create objects](#)

[121.3. Create your custom objects automatically with @Creatable](#)

[121.4. Create automatically objects in the application context with @Singleton](#)

[122. Exercise: Dependency injection for your objects](#)

[122.1. Target of this exercise](#)

[122.2. Prepare the wizard classes for dependency injection](#)

[122.3. Create the wizard via dependency injection](#)

XXVI. Eclipse context functions

[123. Context functions](#)

[123.1. What are context functions?](#)

[123.2. Creation of a context function](#)

[123.3. Examples for context function registrations](#)

[123.4. When to use context functions?](#)

[123.5. Publishing to the OSGi service registry from a context function](#)

124. Exercise: Create a context function

124.1. Target

124.2. Add dependencies to the service plug-in

124.3. Create a class for the context function

124.4. Register the context function

124.5. Specify the responsibility of the context function

124.6. Error analysis

124.7. Deactivate your ITodoService OSGi service

124.8. Notifications from the ITodoService

124.9. Clean-up your user interface code

124.10. Validating

124.11. Review implementation

XXVII. Model add-ons

125. Using model add-ons

125.1. What are model add-ons?

125.2. Add-ons from the Eclipse framework

125.3. Additional SWT add-ons

125.4. Relationship to other services

126. Exercise: Model add-on to change the close behavior

126.1. Target

126.2. Creating a plug-in

126.3. Creating the model add-on

XXVIII. Supplementary application model data

127. Supplementary model data

127.1. Adding additional information on the model elements

127.2. Tags

127.3. Variables

127.4. Persisted state

127.5. Transient data

127.6. Relevant tags and persisted state keys in the application model

128. Exercise: Using model tags and persisted data

128.1. Target

[128.2. Use tags](#)

[128.3. Optional: Use persisted state](#)

[XXIX. Application model modularity](#)

[129. Contributing to the application model](#)

[129.1. Modularity support in Eclipse RCP](#)

[129.2. Contributing to the application model](#)

[129.3. Constructing the runtime application model](#)

[129.4. Fragment extension elements](#)

[130. Exercise: Contributing via model fragments](#)

[130.1. Target](#)

[130.2. Create a new plug-in](#)

[130.3. Add the dependencies](#)

[130.4. Create a handler class](#)

[130.5. Create a model fragment](#)

[130.6. Validate that the fragment is registered as extension](#)

[130.7. Adding model elements](#)

[130.8. Update the product configuration \(via the feature\)](#)

[130.9. Validating](#)

[130.10. Exercise: Contributing a part](#)

[131. Exercise: Implementing a model processor](#)

[131.1. Target](#)

[131.2. Enter the dependencies](#)

[131.3. Create the Java classes](#)

[131.4. Register processor via extension](#)

[131.5. Validating](#)

[XXX. Eclipse application life cycle](#)

[132. Registering for the application life cycle](#)

[132.1. Connecting to the Eclipse application life cycle](#)

[132.2. Accessing application startup parameters](#)

[132.3. Close static splash screen](#)

[132.4. How to implement a life cycle class](#)

[132.5. Example life cycle implementation](#)

133. Exercise: Life cycle hook and a login screen

133.1. Target

133.2. Create a new class

133.3. Register life cycle hook

133.4. Validating

XXXI. Handling preferences

134. Eclipse preference basics

134.1. Preferences and scopes

134.2. Storage of the preferences

134.3. Eclipse preference API

134.4. Setting preferences via plugin_customization.ini

135. Preferences and dependency injection

135.1. Preferences and dependency injection

135.2. Persistence of part state

136. Exercise: Using preferences in the life cycle class

136.1. Target

136.2. Dependency

136.3. Create interface for the preference constants

136.4. Using preferences in the life cycle class

136.5. Validate life cycle handling

137. Exercise: Using preferences in a handler and in a part

137.1. Target

137.2. Using preferences in your handler

137.3. Validate menu entry

XXXII. Internationalization

138. Internationalization and localization in Eclipse

138.1. Translation of a Java applications

138.2. Property files

138.3. Encoding of property files in Java

138.4. Relevant files for translation in Eclipse applications

[138.5. Where to store the translations?](#)

[138.6. Setting the language in the launch configuration](#)

[138.7. Translation service](#)

[139. Translating the application model and plugin.xml](#)

[139.1. OSGi resource bundles](#)

[139.2. Translating the application model](#)

[139.3. Translating plugin.xml](#)

[140. Source code translation with the Eclipse translation service](#)

[140.1. Translation with POJOs](#)

[140.2. Search process for translation files](#)

[140.3. Dynamic language switch](#)

[141. Source code translation with NLS support](#)

[141.1. NLS compared to the Eclipse translation service](#)

[141.2. Translating your custom code](#)

[141.3. Translating SWT and JFace code](#)

[142. Exporting and common problems with translations](#)

[142.1. Exporting Plug-ins and Products](#)

[142.2. Common problems with i18n](#)

[143. Optional Exercise: Internationalization for the application model](#)

[143.1. Target](#)

[143.2. Create the translations for the application model](#)

[143.3. Translate the application model](#)

[143.4. Test the translation of the application model](#)

[143.5. Application model and translations](#)

[144. Optional exercise: Internationalization for the source code](#)

[144.1. Target](#)

[144.2. Creating a plug-in to host the translations](#)

[144.3. Create a Message class for the source code translations](#)

[144.4. Create the translations for the source code](#)

[144.5. Export the translations as API](#)

- [144.6. Define dependencies to the translation plug-in](#)
- [144.7. Update the product \(via the feature\)](#)
- [144.8. Using @Translation to get the messages injected](#)
- [144.9. Test your translation](#)

[XXXIII. Custom annotations and ExtendedObjectSupplier](#)

- [145. Usage of custom annotations](#)
- [145.1. Custom annotations](#)
- [145.2. Restrictions](#)
- [145.3. Example usage](#)

[146. Exercise: Defining custom annotations](#)

- [146.1. Target](#)
- [146.2. Creating a new plug-in](#)
- [146.3. Define and export annotations](#)
- [146.4. Create class](#)
- [146.5. Register the annotation processor](#)
- [146.6. Add the plug-in as dependency](#)
- [146.7. Update the product configuration \(via the features\)](#)
- [146.8. Update the build.properties](#)
- [146.9. Validate: Use your custom annotation](#)

[XXXIV. Using Java libraries in Eclipse applications](#)

- [147. Defining and using libraries in Java](#)
- [147.1. What is a JAR file?](#)
- [147.2. Using Java libraries](#)

[148. Using JAR files in Eclipse applications](#)

- [148.1. JAR files without OSGi meta-data](#)
- [148.2. Integrating external jars / third party libraries](#)

[XXXV. Testing of Eclipse plug-ins and applications](#)

- [149. Introduction to JUnit](#)
- [149.1. The JUnit framework](#)
- [149.2. How to define a test in JUnit?](#)
- [149.3. Example JUnit test](#)

[149.4. JUnit naming conventions](#)

[149.5. JUnit test suites](#)

[149.6. Run your test from the command line](#)

[150. Basic JUnit code constructs](#)

[150.1. Available JUnit annotations](#)

[150.2. Assert statements](#)

[150.3. Test execution order](#)

[151. Eclipse IDE support for JUnit tests](#)

[151.1. Creating JUnit tests](#)

[151.2. Running JUnit tests](#)

[151.3. JUnit static imports](#)

[151.4. Wizard for creating test suites](#)

[151.5. Testing exception](#)

[151.6. JUnit Plug-in Test](#)

[151.7. Setting Eclipse up for using JUnits static imports](#)

[152. Testing Eclipse 4 applications](#)

[152.1. General testing](#)

[152.2. Fragment projects](#)

[152.3. Testing user interface components](#)

[152.4. Testing dependency injection](#)

[153. UI testing with SWTBot](#)

[153.1. User interface testing with SWTBot](#)

[153.2. Installation](#)

[153.3. SWTBot API](#)

[XXXVI. Eclipse application updates](#)

[154. Implementing updates in your application](#)

[154.1. Eclipse application updates](#)

[154.2. Creating p2 update sites](#)

[154.3. p2 composite repositories](#)

[155. Using the p2 update API](#)

[155.1. Required plug-ins for updates](#)

[155.2. Updating Eclipse RCP applications](#)

[156. Exercise: Performing an application update](#)

[156.1. Preparation: Ensure the exported product works](#)

[156.2. Select an update location](#)

[156.3. Add dependencies](#)

[156.4. Add the p2 feature to the product](#)

[156.5. Create a user interface](#)

[156.6. Enter a version in your product configuration file](#)

[156.7. Create the initial product export](#)

[156.8. Start the exported application and check for updates](#)

[156.9. Make a change and export the product again](#)

[156.10. Update the application](#)

[XXXVII. Using target platforms](#)

[157. Target Platform](#)

[157.1. Defining available plug-ins for development](#)

[157.2. Target platform definition](#)

[157.3. Using an explicit target definition file](#)

[157.4. Defining a target platform](#)

[158. Exercise: Defining a target platform](#)

[158.1. Creating a target definition file](#)

[158.2. Activate your target platform for development](#)

[158.3. Validate that target platform is active](#)

[158.4. Solving potential issues for development](#)

[XXXVIII. Using custom extension points](#)

[159. Creating and evaluating extension points](#)

[159.1. Extensions and extension points](#)

[159.2. Creating an extension point](#)

[159.3. Adding extensions to extension points](#)

[159.4. Accessing extensions](#)

[159.5. Extension Factories](#)

[160. Exercise: Create and evaluate extension point](#)

- [160.1. Target for this exercise](#)
- [160.2. Creating a plug-in for the extension point definition](#)
- [160.3. Create an extension point](#)
- [160.4. Export the package](#)
- [160.5. Add dependencies](#)
- [160.6. Evaluating the registered extensions](#)
- [160.7. Create a menu entry and add it to your product](#)
- [160.8. Providing an extension](#)
- [160.9. Add the plug-in to your product](#)
- [160.10. Validating](#)

[XXXIX. Eclipse styling with CSS](#)

- [161. Introduction to CSS styling](#)
- [161.1. Cascading Style Sheets \(CSS\)](#)
- [161.2. Styling Eclipse applications](#)
- [161.3. Limitations](#)

[162. How to style in Eclipse](#)

- [162.1. Fixed styling in Eclipse](#)
- [162.2. Dynamic styling using themes](#)

[163. More details on Eclipse styling](#)

- [163.1. CSS attributes and selectors](#)
- [163.2. Styling based on identifiers and classes](#)
- [163.3. Colors and gradients](#)
- [163.4. CSS imports](#)
- [163.5. CSS Tools](#)

[164. Exercise: Styling with CSS files](#)

- [164.1. Target](#)
- [164.2. Create a CSS file](#)
- [164.3. Define the applicationCSS property](#)
- [164.4. Validating](#)
- [164.5. Adjust the build.properties file](#)

[165. Exercise: Using the theme service](#)

[165.1. Target](#)

[165.2. Add dependencies](#)

[165.3. Create a CSS file](#)

[165.4. Remove the applicationCSS property](#)

[165.5. Create the theme extensions](#)

[165.6. Validating](#)

[165.7. Implement a new menu entry](#)

[165.8. Validate theme switching](#)

[165.9. Optional: Reusing the dark theme of Eclipse](#)

[165.10. Adjust the build.properties file](#)

[166. Exercise: Styling the widgets created by the life cycle class](#)

[166.1. Target](#)

[166.2. Implement styling](#)

[166.3. Validating](#)

[XL. The renderer framework](#)

[167. The usage of renderer](#)

[167.1. Renderer](#)

[167.2. Renderer factory and renderer objects](#)

[167.3. Context creation of model objects](#)

[167.4. Using a custom renderer](#)

[167.5. Using a custom renderer for one model element](#)

[168. Existing alternative renderer implementations](#)

[168.1. Alternatives to SWT](#)

[168.2. Vaadin renderer - Vaaeclipse](#)

[168.3. JavaFX renderer - e\(fx\)clipse](#)

[168.4. Eclipse RAP](#)

[168.5. Additional UI toolkits](#)

[169. Exercise: Defining a renderer](#)

[169.1. Target](#)

[169.2. Creating a plug-in](#)

[169.3. Enter the dependencies](#)

[169.4. Create the renderer implementation](#)

[169.5. Register the renderer](#)

[169.6. Validating](#)

[169.7. Exercise: A custom part renderer](#)

[XLI. Application model persistence and model extensions](#)

[170. Custom persistence for the application model](#)

[170.1. Specifying the location of the application model file](#)

[170.2. Custom application model persistence handler](#)

[171. Saving and restoring parts of the application model](#)

[171.1. Store certain model elements](#)

[171.2. Load stored model elements](#)

[172. Extend the Eclipse application model](#)[XLII. Good development practices](#)

[173. Eclipse development good practices](#)

[173.1. Project, package and class names](#)

[173.2. Naming conventions for model identifiers \(IDs\)](#)

[173.3. Create isolated components](#)

[173.4. Usage of your custom extension points](#)

[173.5. Avoid releasing unnecessary API](#)

[173.6. Packages vs. plug-in dependencies](#)

[174. Application communication and context usage](#)

[174.1. Application communication](#)

[174.2. Example: Using events together with the IEclipseContext](#)

[174.3. Which dependency injection approach to use for your implementation](#)

[XLIII. Migrating Eclipse 3.x components and RCP applications](#)

[175. Why migrating an Eclipse 3.x RCP application?](#)

[175.1. Using the Eclipse 3.x API on top of an 4.x runtime](#)

[175.2. Technical reasons for migrating to the 4.x API](#)

[176. Running Eclipse 3.x plug-ins on top of Eclipse 4](#)

[176.1. Using the compatibility mode](#)

[176.2. The e4 API and e4 runtime terminology](#)

[176.3. Running 3.x RCP applications on top of an e4 runtime](#)

[176.4. Benefit of adjusting the runtime to Eclipse 4.x](#)

[177. Excursion: Extending the Eclipse IDE](#)

[177.1. Extending the Eclipse IDE](#)

[177.2. Starting the Eclipse IDE from Eclipse](#)

[177.3. Starting a new Eclipse instance](#)

[177.4. Debugging the Eclipse instance](#)

[178. Partial migrating to the Eclipse 4.x API](#)

[178.1. Using e4 API in 3.x applications](#)

[178.2. Adding e4 commands, menus and toolbars to 3.x based applications](#)

[178.3. Adding Eclipse 4.x parts and perspectives to 3.x based applications](#)

[178.4. Accessing the IEclipseContext from 3.x API](#)

[179. Exercise: Adding an e4 menu and toolbar to a 3.x based application](#)

[179.1. Target of this exercise](#)

[179.2. Creating a plug-in project](#)

[179.3. Starting an Eclipse IDE with your plug-in](#)

[179.4. Adding the plug-in dependencies for the e4 API](#)

[179.5. Creating the handler class](#)

[179.6. Creating a model contribution](#)

[179.7. Adding a toolbar contribution](#)

[179.8. Validating the presence of the menu and toolbar contribution](#)

[180. Exercise: Using POJOs to contribute views to a 3.x based application](#)

[180.1. Target](#)

[180.2. Using e4part and the org.eclipse.ui.views extension point](#)

[180.3. Add the view to a perspective extension](#)

[180.4. Validating](#)

[181. Exercise: Adding e4 part descriptors to 3.x based applications](#)

[181.1. Target](#)

[181.2. Adding a part descriptor](#)

[181.3. Validating](#)

[182. Optional Exercise: Model add-on to change the close behavior of the IDE](#)

[182.1. Target](#)

[182.2. Register model add-on via model fragment with the Eclipse IDE](#)

[182.3. Verifying](#)

[183. Migrating to an e4 API application without 3.x API usage](#)

[183.1. Migrating an Eclipse 3.x application completely to the e4 API](#)

[183.2. Using 3.x components in e4 API based applications](#)

[183.3. Reusing platform components](#)

[183.4. Existing replacements for 3.x components for e4 API based applications](#)

[183.5. Example for using 3.x components in e4 API based applications](#)

[XLIV. Questions, feedback and closing words](#)

[184. Questions, feedback](#)

[184.1. Reporting Eclipse bugs and feature requests](#)

[184.2. Using the Eclipse bugzilla system](#)

[184.3. Eclipse bug priorities](#)

[184.4. Asking \(and answering\) questions](#)

[184.5. Eclipse 4 feedback](#)

[185. Closing words](#)

[XLV. Appendix](#)

[A. Eclipse annotations, extension points](#)

[A.1. Standard annotations in Eclipse](#)

[A.2. Relevant extension points for Eclipse 4](#)

[B. Solutions for the exercises](#)

[B.1. Getting the example implementation](#)

[B.2. More information about Git and Eclipse](#)

[C. Recipes](#)

[C.1. Eclipse update manager](#)

[C.2. Performing an update and install new features](#)

[C.3. See the installed components](#)

[C.4. Uninstalling components](#)

[C.5. Restarting Eclipse](#)

[C.6. Reading resources from plug-ins](#)

[C.7. Loading images from a plug-in](#)

[C.8. Getting the command line arguments](#)

D. Architectural background of the application model

D.1. Main areas of the model

D.2. Advantages of using mix-ins

E. OSGi low level service API

E.1. Using the service API

E.2. BundleContext

E.3. Registering services via API

E.4. Accessing a service via API

E.5. Low-level API vs OSGi declarative services

F. Links and web resources

F.1. Eclipse RCP resources

F.2. Links and Literature

F.3. Eclipse product and export resources

F.4. OSGi Resources

F.5. OSGi Resources

F.6. Eclipse SWT resources

F.7. JFace resources

F.8. Eclipse Data Binding resources

F.9. Eclipse Jobs resources

F.10. Eclipse i18n resources

F.11. CSS styling resources

F.12. Eclipse p2 updater resources

F.13. Logging

Index

Eclipse Rich Client Platform

The complete guide to Eclipse application development

Lars Vogel

Third edition

Copyright © 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015 Lars Vogel

ALL RIGHTS RESERVED. This book contains material protected under International and Federal Copyright Laws and Treaties. Any unauthorized reprint or use of this material is prohibited. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system without express written permission from the author.

The vogella name and logo and related trade dress are trademarks of vogella GmbH.

Eclipse is a trademark of the Eclipse Foundation, Inc. Oracle and Java are registered trademarks of Oracle and/or its affiliates.

All other trademarks referenced in this book are trademarks of their respective firms.

The publisher and author(s) assume no responsibility for errors or omissions or for damages resulting from the use of the information contained herein.

01.05.2015

Dedication

For Jennifer, Kiran, Liam

Foreword

The Eclipse platform is one of the world's most successful open source software projects. Millions of developers use Eclipse every day as their tooling platform. Millions of users every day use applications built on top of the Eclipse Rich Client Platform. The primary motivation of our community's newest major release - Eclipse 4 - is to make Eclipse an even better platform, particularly a better application platform. The Eclipse team has been working on this major revision for over six years, and Eclipse 4 has been the underlying platform for the Eclipse simultaneous releases since the Juno release in June of 2012. We are all excited to see this major new innovation launching into the mainstream of enterprise development.

One of the keys to Eclipse's success is the commitment of the development team to platform stability, and release-to-release compatibility. This requires remarkable effort and discipline on their part, but has made Eclipse a stable foundation on which to build. Eclipse 4 will make it even easier for Java developers to build and deploy great-looking and portable desktop applications, by bringing a simpler, yet more powerful, set of APIs for application developers to use.

Eclipse projects are driven by great developers, and the Eclipse 4 project has had many important contributors. I would like to recognize the contributions of just a few: the present Eclipse platform leader Mike Wilson, and project leaders Dani Megert, John Arthorne, all from IBM and Boris Bokowski, originally with IBM but now with Google. I would also like to recognize the special contributions to Eclipse 4 of Brian de Alwis, Oleg Besedin, Danail Branekov, Eric Moffatt, Bogdan Gheorghe, Paul Webster, Thomas Schindl, Remy Suen, Kai Tödter, and, of course, your author Lars Vogel. Finally, it bears mentioning that the Eclipse 4 platform would not exist without the continuing investment and resource commitments of IBM, BestSolution.at, and SAP.

Lars Vogel has been contributing to the Eclipse 4 project almost since its inception. For many years, he has also been one of the most visible evangelists for Eclipse in general, and for Eclipse 4 in particular. In short, he is one of the leading experts in using Eclipse 4 for building applications. This book provides developers with the information they need to be successful with this exciting new Eclipse platform version. I am sure that you will find it an excellent addition to your Eclipse library.

Mike Milinkovich
Executive Director, Eclipse Foundation

Preface

1. Welcome

Thank you for your interest in learning Eclipse RCP programming. Eclipse is an exciting and established programming platform and I hope you will enjoy it.

If you come to this book after having read other books from the vogella series, thanks for sticking with the series! If you come to this book after having learned about Eclipse RCP and plug-in development on the vogella.com website, then I like to thank you for getting the extended off-line version.

I tried very hard to put together consistent and comprehensive material so that you can perform your work as a developer as efficient as possible. I sincerely hope that this book will augment your knowledge of Eclipse RCP and help you to get the most out of the Eclipse platform.

2. About the author - Lars Vogel



Lars Vogel is the founder and CEO of the *vogella GmbH* company. As co-lead of the Eclipse Platform UI and the e4 project, he is actively developing and designing the Eclipse platform code. Lars loves to share his knowledge by writing books and tutorials or by presenting at international software conferences.

He also likes to share knowledge with the customers of the *vogella GmbH*. Here he delivers development, consulting, coaching and training in the areas of Eclipse, Android and Git. These customers include Fortune 100 corporations as well as individual developers.

Lars is a nominated *Java Champion* since 2012. In 2010 he received the *Eclipse Top Contributor Award* and in 2012 the *Eclipse Top Newcomer Evangelist Award*.

3. About the vogella company



The *vogella GmbH* company offers expert development, consulting, coaching and training support in the areas of Eclipse, Android and Git. See [training offerings](#) and [implementation support](#) for details.

With more than one million visitors per month vogella.com is one of the central sources for Eclipse, Java and Android programming information.

4. Screenshots

The screenshots used in this book are based on Ubuntu 14.04 operating system and the Eclipse 4.4 (Luna) release.

5. Shortcuts on a Mac

The shortcuts listed in this book should be valid for Window and Linux users. Mac users must replace the Ctrl key with the Cmd key for most shortcuts.

6. How this book is organized

This book gives a detailed introduction to the Eclipse platform and covers all relevant aspects of Eclipse RCP development.

Every topic in this book has a content section in which the topic is explained and afterwards you have several exercises to practice your learning.

You will be guided through all relevant aspects of Eclipse 4 development using a comprehensive example which you continue to extend in the exercises.

You will learn about the new programming concepts of Eclipse 4, e.g., the application model, dependency injection, CSS styling, the renderer framework, the event system and much more.

Proven Eclipse technologies like SWT, JFace viewers, OSGi modularity and services, data binding, etc. are also covered in detail.

7. Prerequisites

This book requires a working knowledge of Java and assumes that you are familiar in using the Eclipse IDE for standard Java development. It assumes no previous experience of Eclipse plug-in and Eclipse RCP development.

In case you picked this book to learn about Java and development with Eclipse, you should start with a different reference first. The author of this book has also published an introduction into the Eclipse IDE (ISBN-10: 3943747042) which you can use to learn about Java development with Eclipse.

8. Errata

Every book has errors to a certain degree. You can find a list of the known bugs on the errata page of the vogella.com webpage. The URL to this page is <http://www.vogella.com/book/eclipse4book/errata.html>

In case you find errors which have not yet been reported, please let me know by sending an email to: erratabooks@vogella.com.

Errors might be one of the following:

- Typographical errors
- Examples that do not work as described in the book

9. Exercises, optional exercises and examples

In this book you find lots of example code and exercises. Most exercises are built upon each other. Optional exercises are explicitly marked and they can be left out without affecting the exercises later in this book.

Note

To learn the concepts described in this book, it is highly recommended following at least the mandatory exercises.

10. Long lines

Sometimes the code example did not fit into one line. In this case I use *[CONTINUE...]* to break the lines. If you see such a line, the next line must continue at the position of *[CONTINUE...]*.

11. Example code

The source code for the exercises of this book is available via a GitHub repository. All source code of this book is licensed under the *Eclipse Public License*.

The URL for this repository is <https://github.com/vogella/eclipse4book>. To clone this repository you can use the following URL:
`git://github.com/vogella/eclipse4book.git`

See [Appendix B, *Solutions for the exercises*](#) for a description how to import the exercise solutions into your Eclipse workspace.

12. Acknowledgment

The creation of this book did not follow the typical book creation process. It is based on tutorials available on the vogella.com web site and the material in this book has been used in countless RCP training session conducted by the vogella GmbH.

I got many suggestions and corrections from countless readers of my website and I would like to express my deepest gratitude for their contributions.

I would like to thank Paul Webster, John Arthorne, Tom Schindl and Eric Moffatt from the e4 development team for their shared efforts and their support in answering my questions over the last years. I'm very much impressed by their friendliness and knowledge.

For this third edition I want thank René Brandstetter for dissecting the second edition and most of the new chapters. I think there is not a single page in this book which René did not help to improve. Also, a big thanks to Dirk Fauth who gave feedback to several new chapters, especially the Internationalization chapter, which has been reworked to cover the translation service of Eclipse 4.4.

Also my thanks goes to the reviewers of past editions: Dirk Fauth, René Brandstetter, Aurélien Pupier, Eric Moffatt, Rabea Gransberger, Matthias Mailänder, Simon Scholz, Paul Webster, Remy Suen, Sopot Çela, Kai Tödter, Brian de Alwis, Matthew Hall, Cole Markham, Phill Perryman, Marcel Bruch, Boris Bokowski, Daniel Stainhauser, Christoph Keimel and Holger Voormann. They did a terrific job, all remaining errors are my fault.

Part I. The Eclipse open source project

Chapter 1. Introduction to the Eclipse project

1.1. The Eclipse community and projects

Eclipse is an open source community. The Eclipse open source community consists of more than 150 projects covering different aspects of software development.

Eclipse projects cover lots of different areas, e.g., as a development environment for Java or Android applications. But the Eclipse project also provides very successful runtime components, like the popular *Jetty* webserver and the *Hudson* build server.

1.2. A short Eclipse history

The roots of Eclipse go back to 2001. The initial code base was provided by IBM. In November 2001, a consortium was formed to support the development of Eclipse as open source software. This consortium was controlled by IBM.

In 2004 it became the *Eclipse Foundation*, which is a vendor neutral foundation where no single company has control of the direction.

The *Eclipse* name at this time was viewed by many as declaration of war against *Sun Microsystems*, the company responsible for developing the Java programming language. IBM stated that the name was targeting at "Eclipsing" Microsoft. See [Eclipse: Behind the Name](#) for details.

With the purchase of *Sun Microsystems* by Oracle this conflict finally went away. Oracle is currently among the 5 largest contributor companies of the Eclipse project.

1.3. Eclipse releases

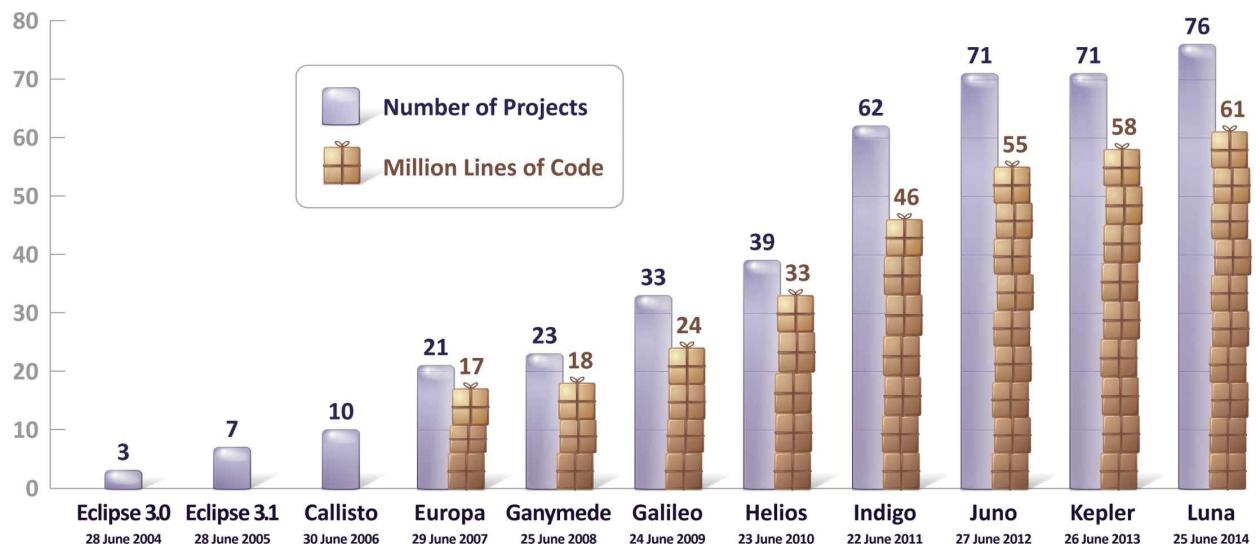
The Eclipse open source project has a simultaneous release every year. Eclipse projects can choose to join this release but must fulfill the requirements described in the [Release requirements wiki](#).

This simultaneous release includes new versions of the Eclipse IDE distributions. Before 2012 Eclipse was released in version 3.x, 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, Eclipse 4.3 in 2013 and Eclipse 4.4. in 2014.

The following graphic depicts the number of projects and lines of code (measured in millions) joining this release over the years.

eclipse Simultaneous Releases



Chapter 2. Eclipse project structure

2.1. Eclipse foundation

The Eclipse projects are governed by the *Eclipse Foundation*. The Eclipse Foundation is a non-profit, member supported corporation that hosts the Eclipse Open Source projects and helps to cultivate both its open source community and its ecosystem of complementary products and services.

The Eclipse Foundation does not work on the Eclipse code base, i.e., it does not have employee developers working on Eclipse. The mission of the Eclipse Foundation is to *enable* the development by providing the infrastructure (Git, Gerrit, Hudson build server, the download sites, etc.) and a structured process.

There is also an Eclipse Foundation Europe based in Germany. See [Eclipse Foundation Europe FAQ](#) for details.

2.2. Staff of the Eclipse foundation

In 2015 the Eclipse Foundation had approximately 20 employees. See [Eclipse Foundation Staff](#) for details.

Chapter 3. The Eclipse Public License

3.1. Source code and the Eclipse Public License

The *Eclipse Public License* (EPL) is an open source software license used by the Eclipse Foundation for its software. The EPL is designed to be business-friendly. EPL licensed programs can be used, modified, copied and distributed free of charge. The consumer of EPL-licensed software can choose to use this software in closed source programs.

Only modifications in the original EPL code must also be released as EPL code. This can for example be done by filling a bug report at the public Eclipse bug tracker and by uploading a Gerrit change.

3.2. Intellectual property cleansing of Eclipse code

The Eclipse Foundation validates that source code contributed to Eclipse projects is free of intellectual property (IP) issues. This process is known as IP cleansing. Contributions with more than 1000 lines of code require the creation of a Contribution Questionnaire, and a review and approval by the IP team.

The permissive EPL and the IP cleansing effort of the Eclipse Foundation makes reusing the source code of Eclipse projects attractive to companies.

Part II. The concepts behind Eclipse applications

Chapter 4. Eclipse plug-ins and applications

4.1. What is an Eclipse RCP application?

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

4.2. Eclipse software components - Plug-ins

An Eclipse application consists of several Eclipse components. A software component in Eclipse is called a *plug-in*. The Eclipse platform allows the developer to extend Eclipse applications like the Eclipse IDE with additional functionalities via plug-ins.

Eclipse applications use a runtime based on a specification called *OSGi*. A software component in OSGi is called a *bundle* but a bundle is also always a plug-in. Both terms can be used interchangeably. A software component in Eclipse is called a *plug-in*.

For example a new plug-in can create new menu entries or toolbar entries.

4.3. Advantages of developing Eclipse plug-ins

The Eclipse platform forms the basis of the most successful Java IDE and therefore is very stable and broadly used. It uses native user interface components which are fast and reliable. It has a strong modular approach based on the industry standard module system for Java (OSGi) that allows developers to design component based systems.

Companies such as IBM, SAP and Google use the Eclipse framework as a basis for their products and therefore need to ensure that Eclipse is flexible, fast and continues to evolve.

The Eclipse platform also fosters a large community of individuals which provide support, documentation and extensions to the Eclipse framework. Tapping into this ecosystem allows you to find required resources and information.

4.4. What is the Eclipse Rich Client Platform (Eclipse RCP)?

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

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

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

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

Chapter 5. Important Eclipse projects for RCP

5.1. The Eclipse Platform project

The *Eclipse Platform* project provides the core frameworks and services upon which all Eclipse based applications are created. It also provides the runtime in which Eclipse components are loaded, integrated, and executed. The primary purpose of the *Eclipse Platform* project is to enable other developers to easily build and deliver integrated tools and applications.

5.2. The Eclipse e4 project

Eclipse e4 is the name used for the *Eclipse Platform* incubator project. This incubator is used for exploratory projects relating to improving the Eclipse Platform.

The e4 project includes several technology evaluations. Some of these evaluations have been ported back to the core Eclipse framework. All functionalities described in this document are part of the official Eclipse release, except the *Eclipse e4 tooling* project.

5.3. The Eclipse Plug-in Development Environment (PDE)

The Eclipse Plug-in Development Environment (PDE) project provides wizards, editors and other tools to develop Eclipse plug-ins.

5.4. The e4 tools project

The project *Eclipse e4 tooling* provides tools for developing Eclipse 4 applications. They are very useful, but have not yet been added to the Eclipse platform project.

Projects such as XWT, TM or OpenSocial Gadgets, which are also part of the Eclipse e4 project, are not included in the standard Eclipse 4 core platform and are not described in this document.

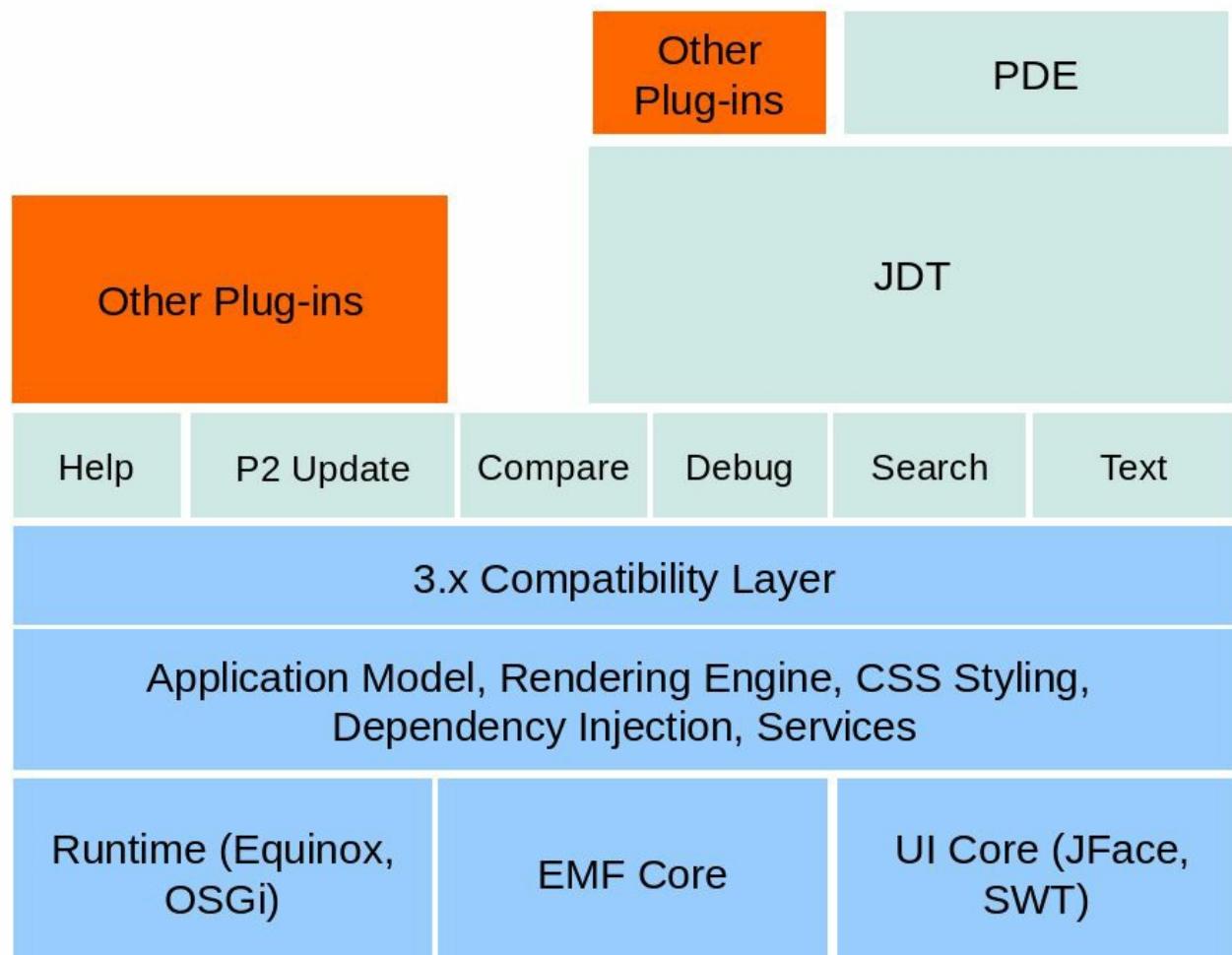
Chapter 6. Architecture of Eclipse

The following chapter gives an overview of the Eclipse architecture, explains the term *plug-ins* and what extensions and extension points are.

6.1. Architecture of 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.

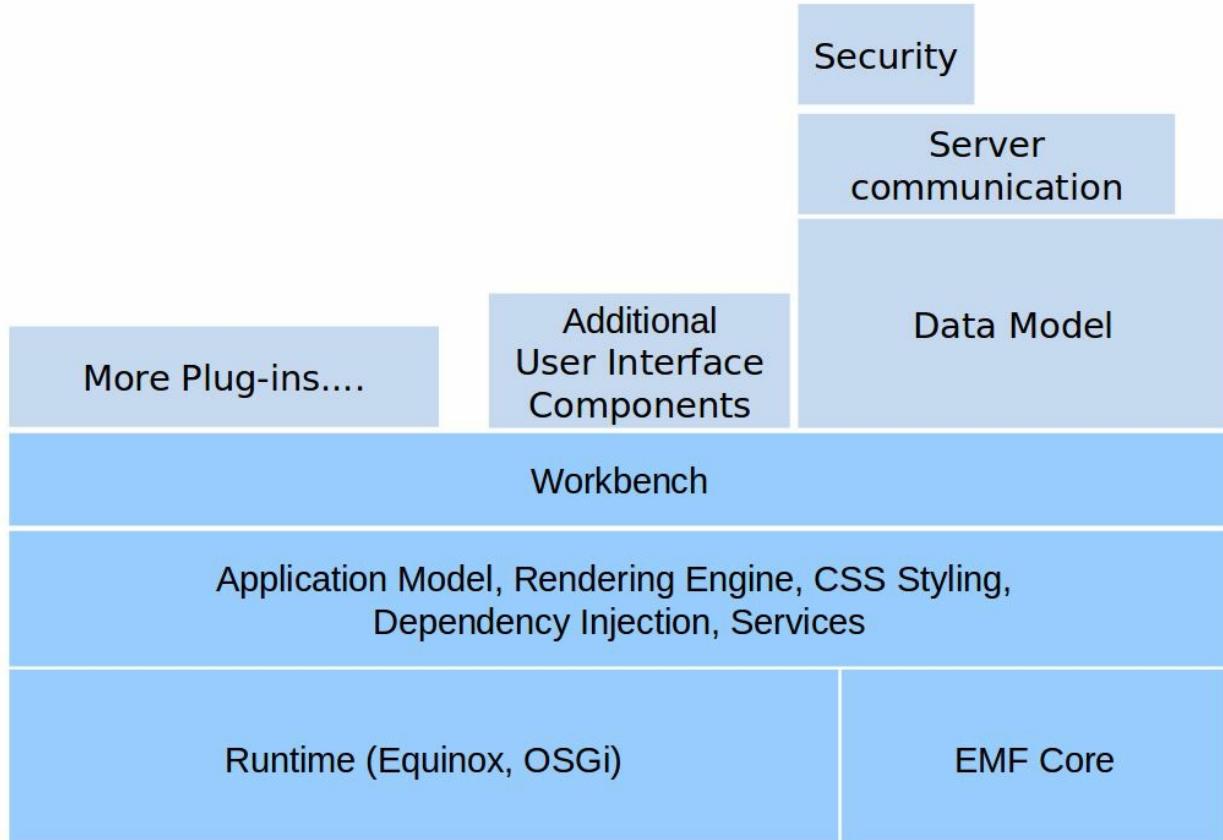
The core components of the Eclipse IDE are depicted in the following graphic. The intention of the graphic is to demonstrate the general concept, the displayed relationships are not 100 % accurate.



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

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

An Eclipse RCP application typically uses the same base components of the Eclipse platform and adds additional application specific components as depicted in the following graphic.



6.2. 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. It is responsible for displaying all other user interface components.

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

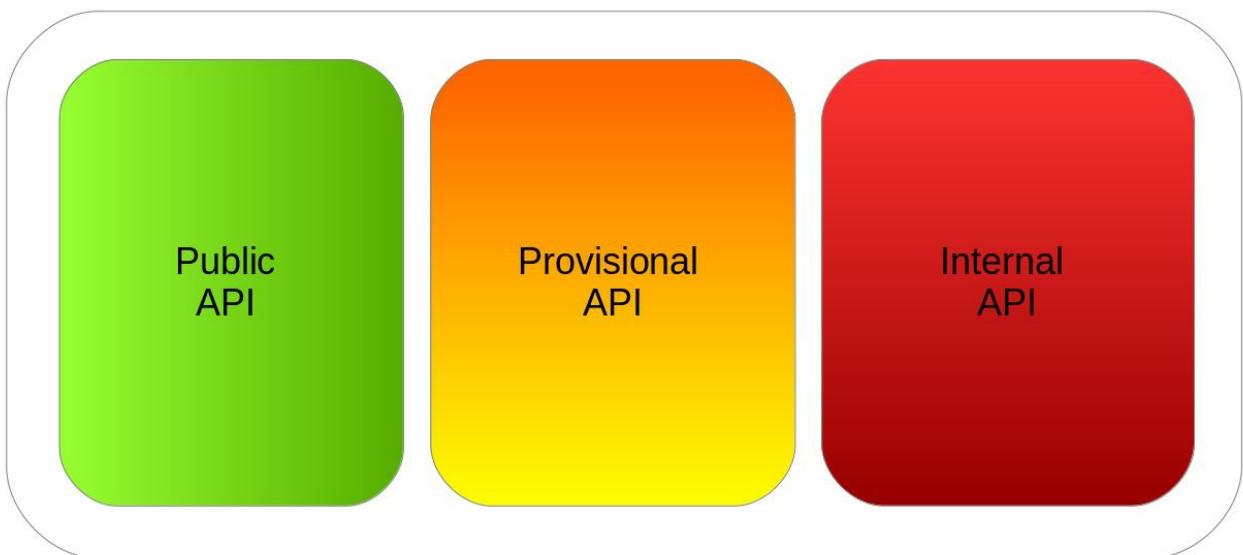
6.3. Compatibility layer for Eclipse 3.x plug-ins

The Eclipse platform in version 4.x uses a different API than in version 3.x. Most existing plug-ins available for the Eclipse IDE are still based on the Eclipse 3.x programming model.

The Eclipse platform provides a compatibility layer supporting that plug-ins using the Eclipse 3.x API can run unmodified on top of an Eclipse 4 based application.

6.4. Eclipse API and internal API

An OSGi runtime allows developers to mark Java packages as API and to mark packages as internal API. It is also possible to mark a Java package as provisional API. This allows developers to test this API but indicates that the API is not finalized.



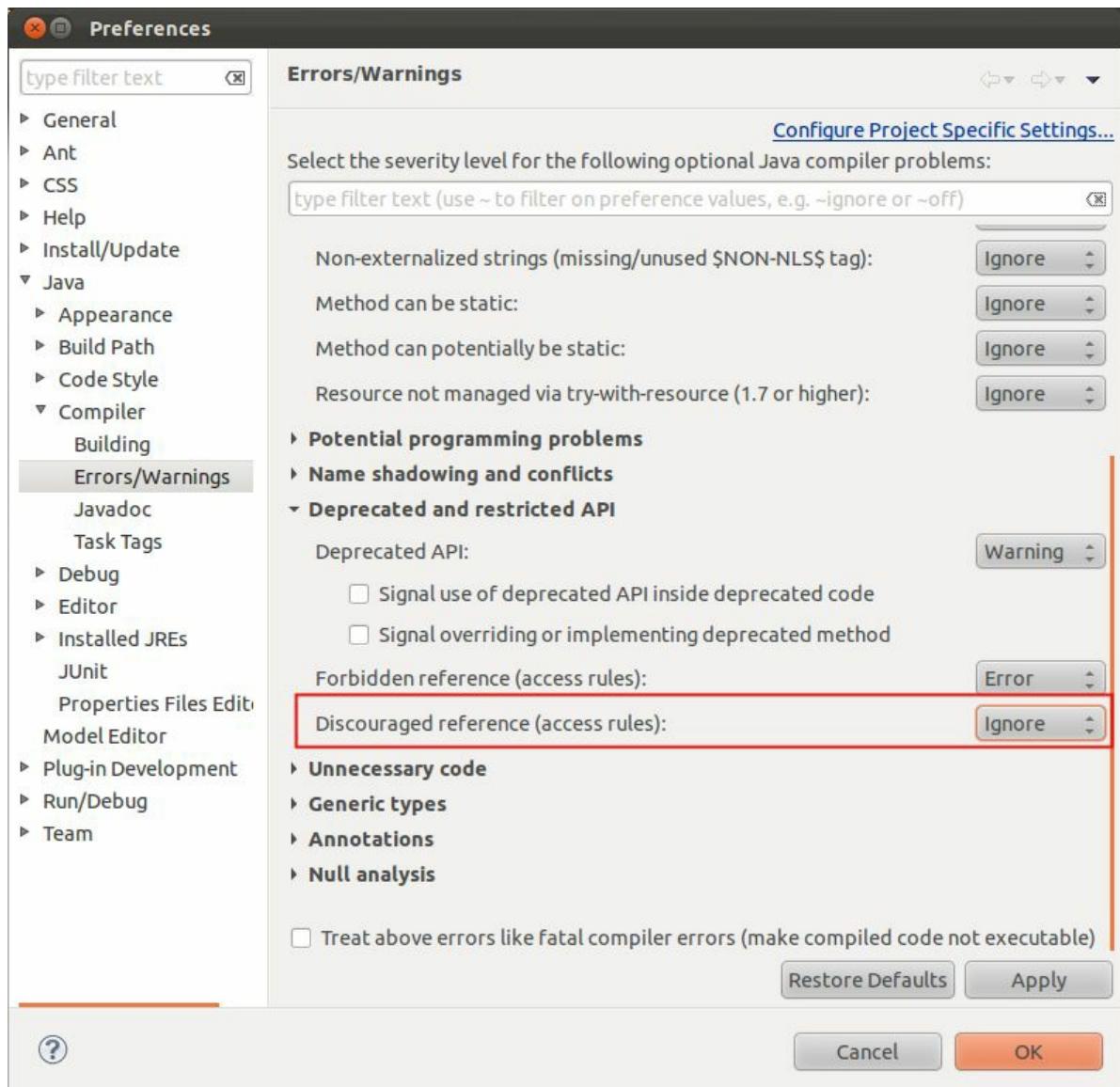
The Eclipse platform project marks packages either as public API or as provisional API, to make all Java classes accessible to Eclipse developers. If the Eclipse platform project releases an API, the platform project plans to keep this API stable for as long as possible.

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

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

Tip

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



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

6.5. Important configuration files for Eclipse plug-ins

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

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

An Eclipse plug-in defines its meta data, like its unique identifier, its exported API and its dependencies via the *MANIFEST.MF* file.

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

Part III. Setting up an Eclipse RCP development environment

Chapter 7. Install Java for Eclipse RCP development

7.1. Java requirements of the Eclipse IDE

To run the Eclipse IDE a Java 7 JRE/JDK is required for most of the Luna package downloads based on Eclipse 4.4.

The Eclipse IDE contains its custom Java compiler hence a JRE is sufficient for most tasks with Eclipse. The *JDK* version of Java is only required if you compile Java source code on the command line and for advanced development scenarios, for example, if you use automatic builds or if you develop Java web applications.

Please note that you need a 64 JVM to run a 64 bit Eclipse and a 32 bit JVM to run a 32 bit Eclipse.

7.2. Check installation

To run Java programs on your computer you must at least have the Java runtime environment (JRE) installed. This might already be the case on your machine. You can test if the JRE is installed and in your current path by opening a console (if you are using Windows: Win+R, enter *cmd* and press Enter) and by typing in the following command:

```
java -version
```

If the JRE is installed and within your path, this command prints information about your Java installation. In this case you can skip the Java installation description. You may want to note down if you have a 32 bit or 64 bit version of Java, see [Section 7.7, “How can you tell you are using a 32 bit or 64 bit version of Java? ”](#).

If the command line returns the information that the program could not be found, you have to install Java.

7.3. Install Java on Ubuntu

On Ubuntu you can install Java 7 via the following command on the command line.

```
sudo apt-get install openjdk-7-jdk
```

7.4. Install Java on MS Windows

For Microsofts Windows, Oracle provides a native installer which can be found on the Oracle website. The central website for installing Java is located under the following URL and also contains instructions how to install Java for other platforms.

java.com

7.5. Installation problems and other operating systems

If you have problems installing Java on your system, search via Google for *How to install JDK on YOUR_OS*. This should result in helpful links. Replace *YOUR_OS* with your operating system, e.g., Windows, Ubuntu, Mac OS X, etc.

7.6. Validate installation

Switch again to the command line and run the following command.

```
java -version
```

The output should be similar to the following output.

```
java version "1.7.0_25"
OpenJDK Runtime Environment (IcedTea 2.3.10) (7u25-2.3.10-1ubuntu0.13.04.2)
OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode)
```

7.7. How can you tell you are using a 32 bit or 64 bit version of Java?

You can run a 32 bit or a 64 bit version of Java on a 64 bit system. If you use `java -version` and the output contains the "64-Bit" string you are using the 64 bit version of Java otherwise your are using the 32 bit version. The following is the output of a 64-bit version.

```
java version "1.7.0_25"
OpenJDK Runtime Environment (IcedTea 2.3.10) (7u25-2.3.10-1ubuntu0.13.04.2)
OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode)
```

Chapter 8. Install Eclipse IDE for RCP development

The following description explains how to install the Eclipse IDE and the required tools to develop Eclipse RCP applications.

8.1. Download the Eclipse Software Development Kit (SDK)

The following description is based on the latest Eclipse 4.4 release. Download the latest version of the *Eclipse SDK* build from the following URL:

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

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

Latest Release	Build Name	Build Status	Build Date
	4.4.1	Ju (3 of 3 platforms)	Thu, 25 Sep 2014 -- 04:00 (-0400)
	4.4	Ju (3 of 3 platforms)	Fri, 6 Jun 2014 -- 12:15 (-0400)
	4.3.2	Ju (3 of 3 platforms)	Fri, 21 Feb 2014 -- 17:00 (-0500)

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

8.2. Install the Eclipse IDE

After you downloaded the file with the Eclipse distribution, unpack it to a local directory. Most operating systems can extract zip or tar.gz files in their file browser (e.g., *Windows 7*) with a right-click on the file and selecting "Extract all...".

Note

As a developer person you probably know how to extract a compressed file but if in doubt, search with Google for "How to extract a zip (or tar.gz on Linux and Mac OS) file on ...", replacing "..." with your operating system.

Warning

Extract Eclipse to a directory without spaces in its path and do not use a mapped network drive (Windows). Also avoid path names longer than 255 characters under Microsoft Windows. Installations of the Eclipse IDE sometimes have problems with such a setup.

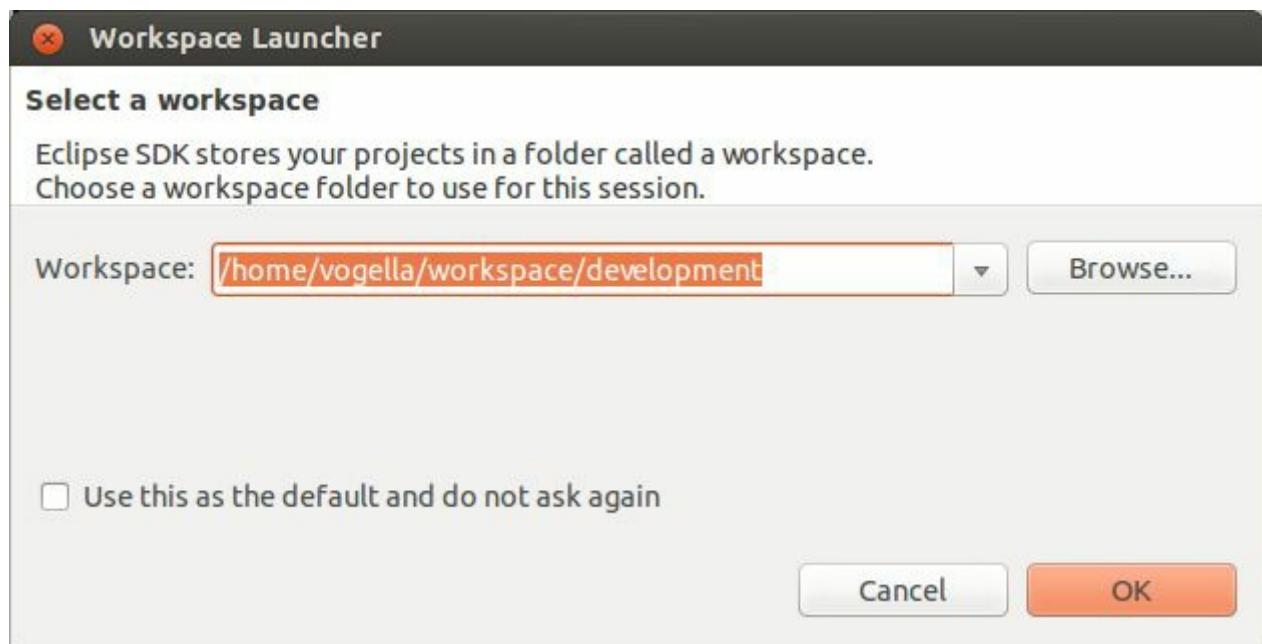
After you extracted the compressed file you can start Eclipse, no additional installation procedure is required.

8.3. Starting the Eclipse IDE

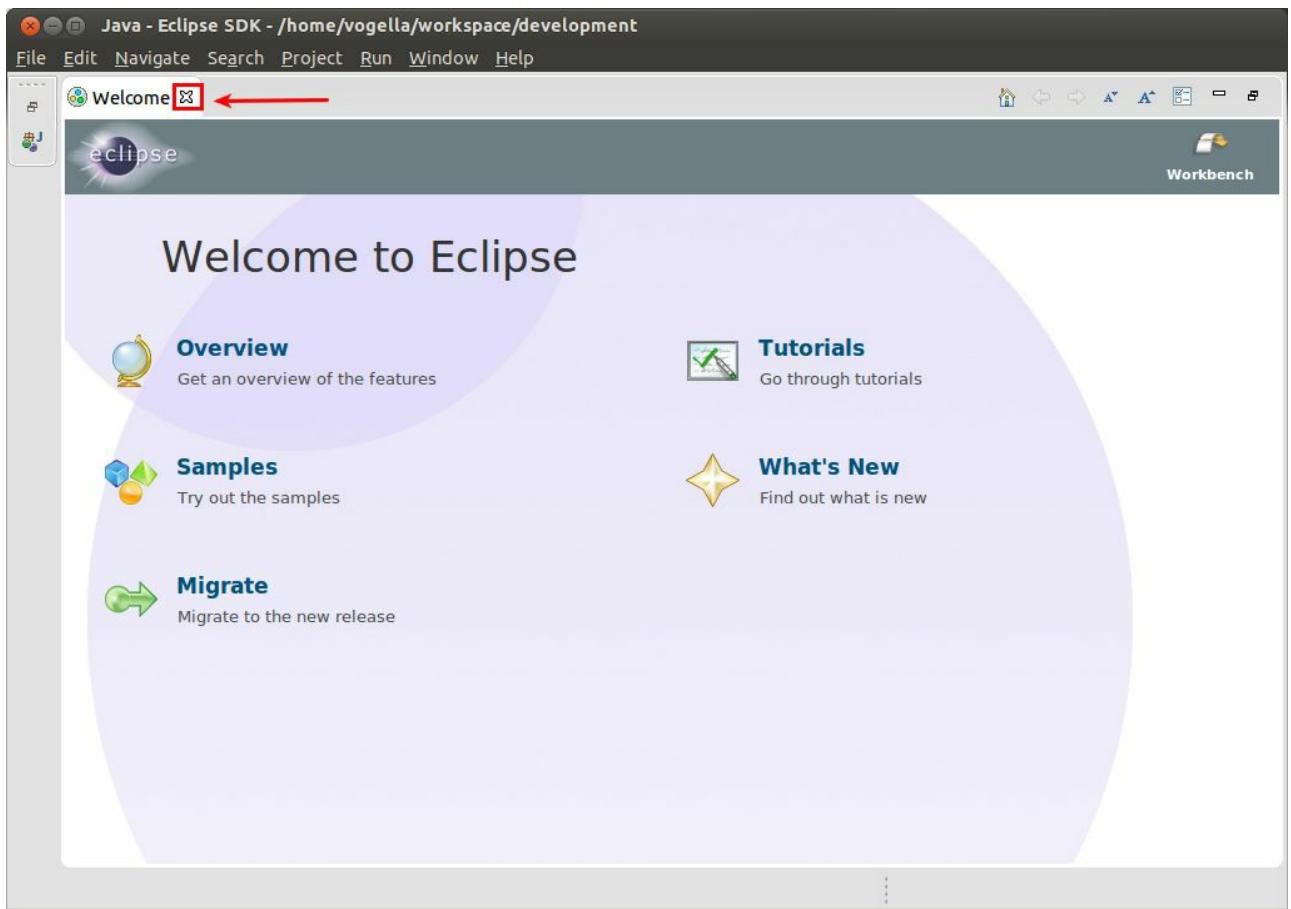
To start Eclipse, double-click the `eclipse.exe` (Microsoft Windows) or `eclipse` (Linux / Mac) file in the directory where you unpacked Eclipse.

The Eclipse system prompts you for a *workspace*. The *workspace* is the location in your file system where Eclipse stores its preferences and other resources. For example your projects can be stored in the workspace.

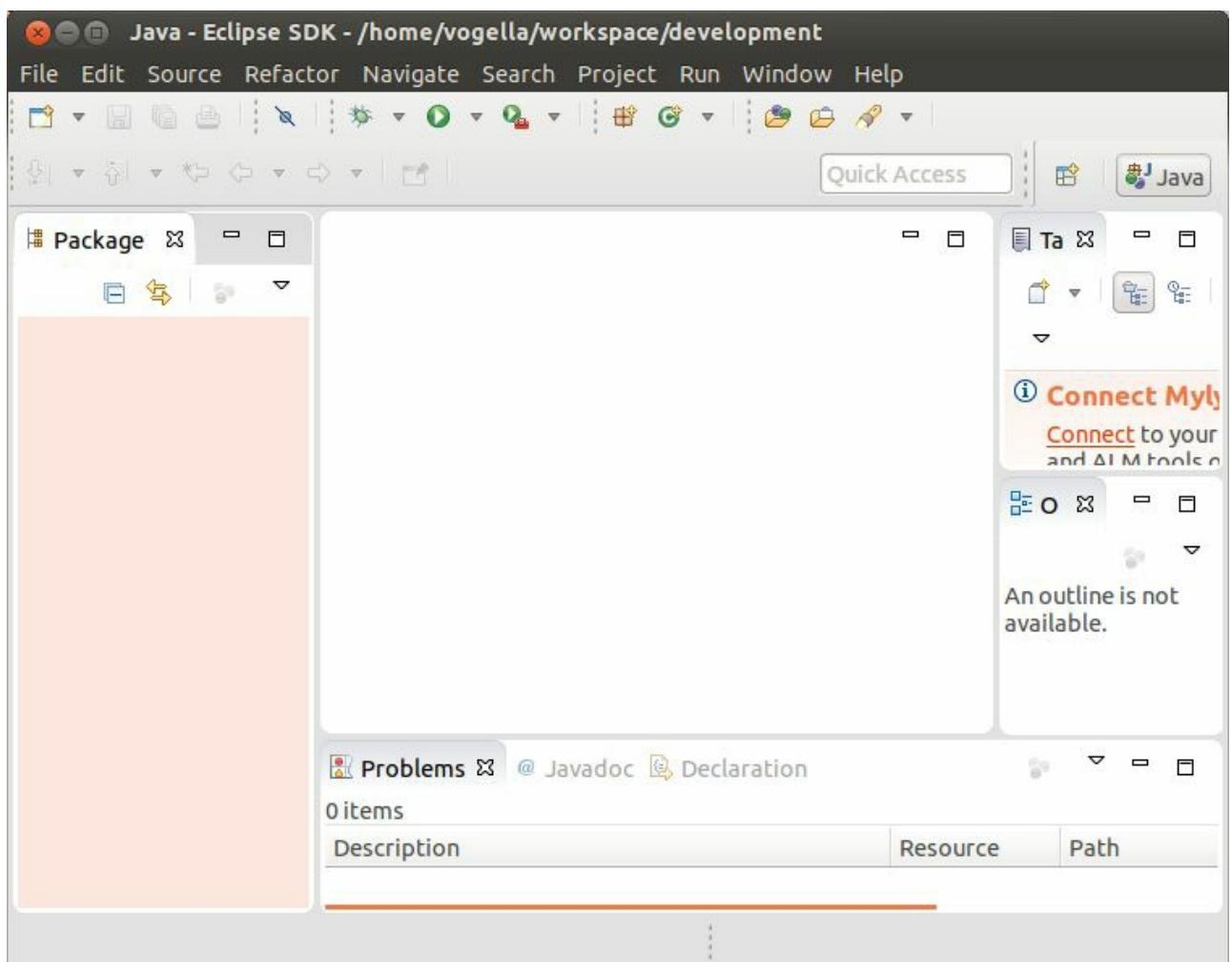
Select an empty directory and click the *OK* button.



Eclipse starts and shows the *Welcome* page. Close this page by clicking the *x* beside *Welcome*.



After closing the welcome screen, the application should look similar to the following screenshot.

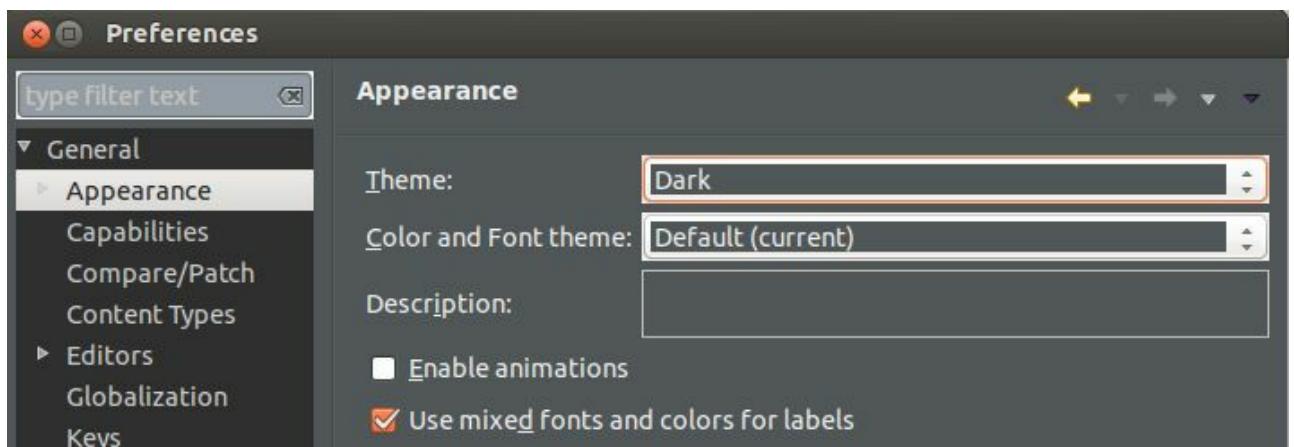


8.4. Appearance

The appearance of Eclipse can be configured. By default, Eclipse ships with a few themes but you can also extend Eclipse with new themes.

To change the appearance, select from the menu Window → Preferences → General → Appearance.

The *Theme* selection allows you to change the appearance of your Eclipse IDE. For example you can switch to the *Dark* theme of Eclipse.



Chapter 9. Install the e4 tools

9.1. Requirements

To install the tools required for Eclipse RCP development you need to know how to use the Eclipse update manager. In case you are unfamiliar with the usage of the update manager please see [Section C.1, “Eclipse update manager”.](#)

9.2. Install the e4 tools from the vogella GmbH update site

The Eclipse SDK download does not include the e4 tools, which provide the tools to create and analyze Eclipse 4 RCP applications. These tools provide wizards to create Eclipse application artifacts, an application model editor and several tools to analyze an Eclipse application.

The vogella GmbH company provides a recent version of the e4 tools for the Eclipse 4.4 release under the following URL:

<http://dl.bintray.com/vogellacompany/e4tools-luna>

You can install the e4 tools via Help → Install New Software... by entering the URL. The e4 tools installation from the vogella update site is not signed. For a signed e4 tools installation use [Section 9.3, “Install the e4 tools from the Eclipse.org update site”](#).

Available Software

Check the items that you wish to install.

The screenshot shows the 'Available Software' dialog in Eclipse. At the top, it says 'Work with:' followed by a text input field containing the URL <http://dl.bintray.com/vogellacompany/e4tools-luna/>. Below this is a 'type filter text' input field. The main area is titled 'Name' and contains a list of items, each preceded by a checkbox and an icon. All items in the list have their checkboxes checked. The items are:

- ▶ Eclipse 4 - All Spies
- ▶ Eclipse 4 - Context Spy
- ▶ Eclipse 4 - Core Tools
- ▶ Eclipse 4 - CSS Spy
- ▶ Eclipse 4 - Event Spy
- ▶ Eclipse 4 - Model Spy

Only the *Eclipse 4 - Core Tools* are required to create an Eclipse 4 RCP application. In case one of the other features cannot be installed, you can ignore this and still develop Eclipse 4 RCP applications. Restart your Eclipse IDE after the installation.

Tip

This update site or its content might change. The [e4 tools info page](#) contains the latest information about the e4 tools update site. Especially for Eclipse 4.5 you should check this side, as the process of building the tools is planned to be changed in Eclipse 4.5.

9.3. Install the e4 tools from the Eclipse.org update site

The vogella GmbH update site was created to provide a stable link for users of the e4 tools. The same code base 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 also find the URL for the update site. The following screenshots demonstrate this for a particular build of the e4 tools.



Note

This website might change over time.

eclipse

Contact | Legal

» Eclipse e4
» Eclipse SDK 4.4
» Eclipse SDK 3.8

Stable Build: 0.17

201501051100. Built against Eclipse 4.3 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 4.3 SDK or compatible](#).

Modeled UI and CSS

» EMF runtime 2.9

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	Source Builds
<ul style="list-style-type: none"> » 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. 	<ul style="list-style-type: none"> » Access the Source Builds page, under construction

URL for the update site



Eclipse e4				
Status	Platform	Download	Size	File
	All (Supported Versions)	(http)	15121931	eclipse-e4-repo-incubation-0.17.zip

Comments
online p2 repo link

Chapter 10. Start Eclipse and use a new workspace

10.1. About the Eclipse IDE

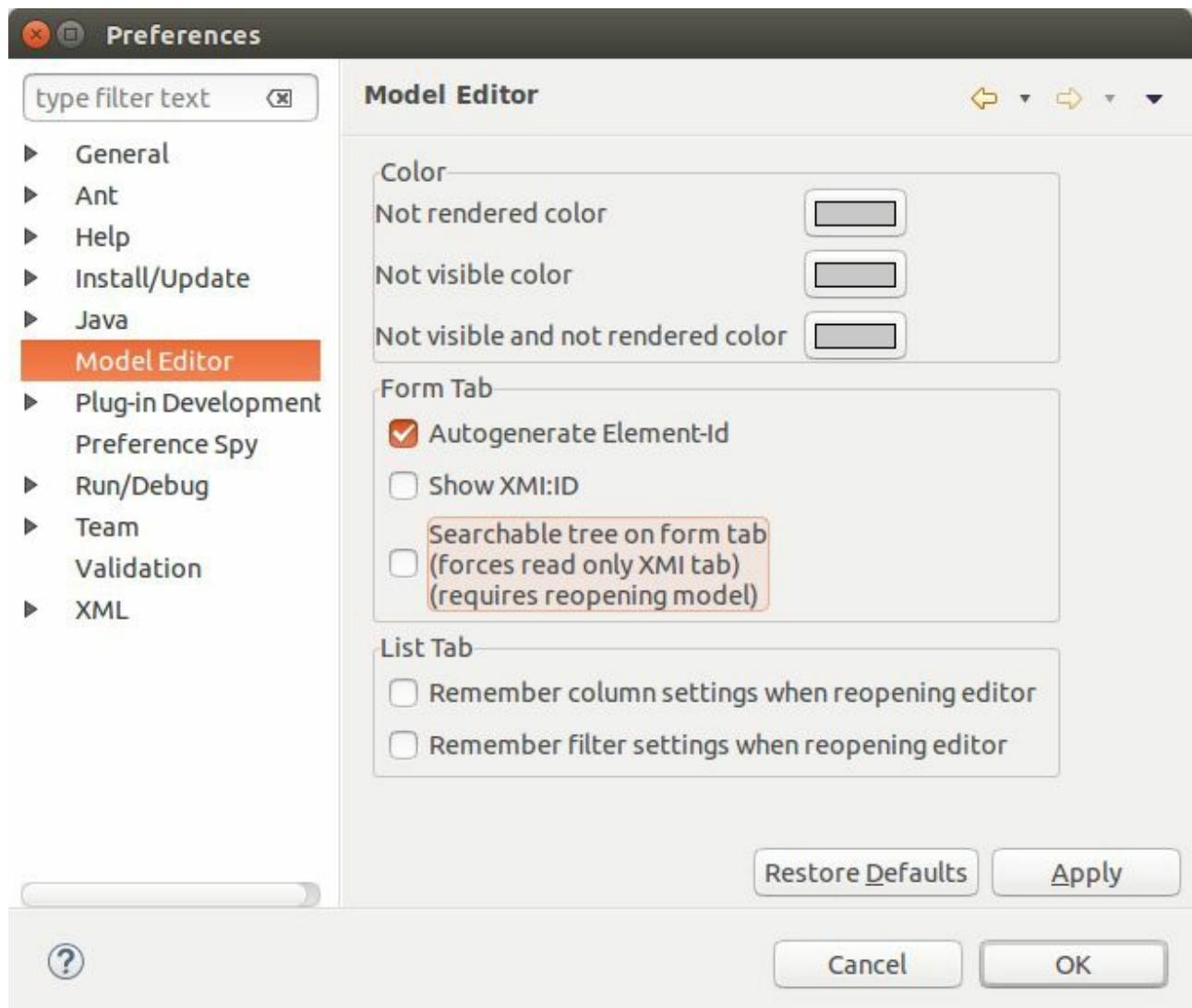
If you don't have experience with using the Eclipse IDE for Java development, you may want to read the [Eclipse IDE book](#) or at least the [Eclipse IDE online tutorial](#).

10.2. Workspace

It is suggested to use a new workspace for the exercises of this book to avoid any side effects of your developments with existing projects.

10.3. Review the Eclipse 4 model editor preferences

In Eclipse RCP you work with an application model. To work efficiently with the underlying data format, you use the *Eclipse 4 model editor*. Its preference settings can be reached via Window → Preferences → Model Editor. The following screenshot shows the preference page.



Chapter 11. Enable Java access to all plug-ins

11.1. Filtering by the Java tools

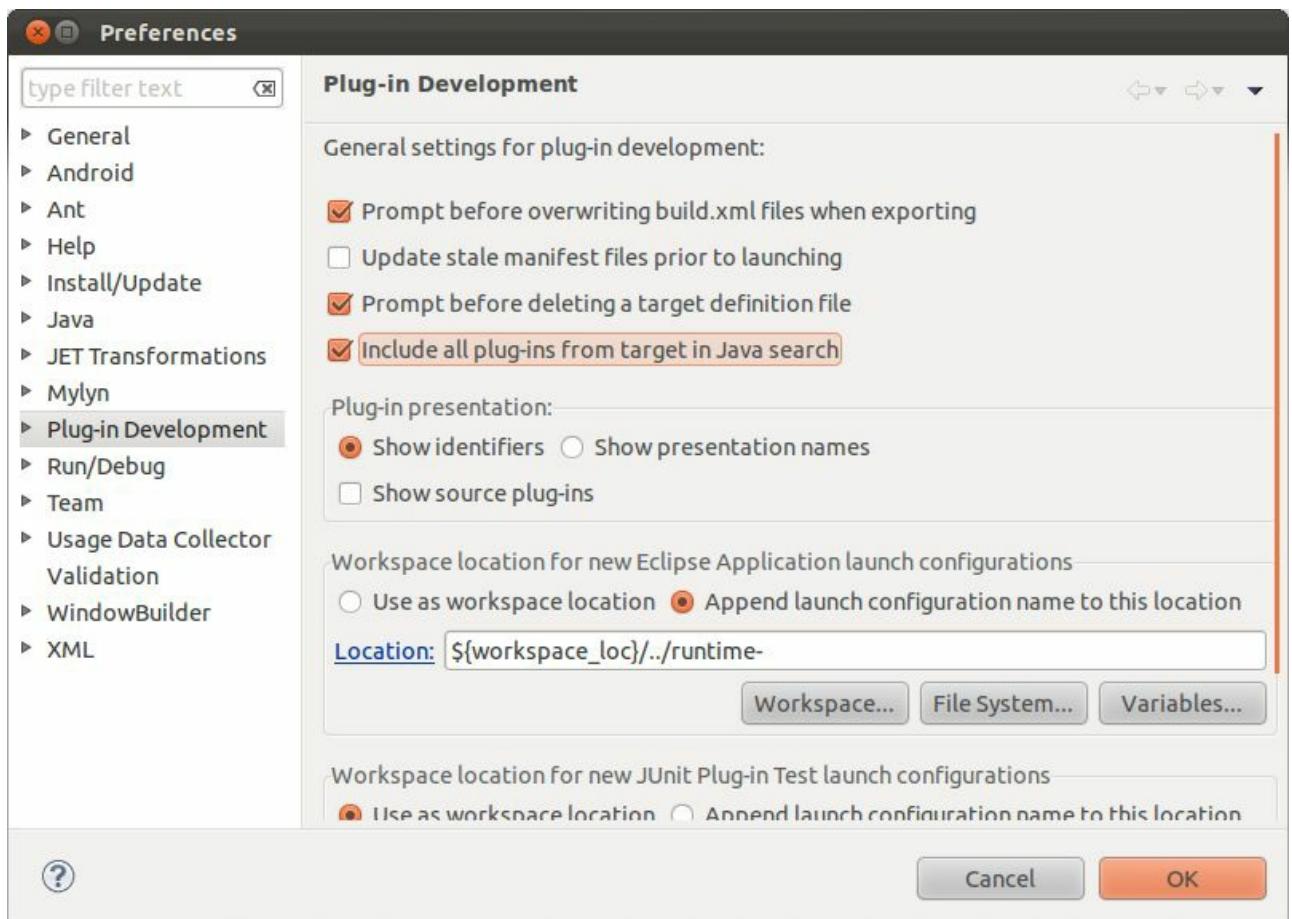
The Java Development Tools (JDT) of the Eclipse IDE limit the scope of search related activities. By default, JDT considers elements from opened projects including their dependencies as well as elements from the standard Java library.

For example, the **Ctrl+Shift+T** (Open Type) shortcut will not find the `ISources` interface if it is not referred to in the current workspace. As plug-in developer you want to have access to all classes in your current *target platform*.

The *target platform* is the set of plug-ins against you develop. By default, the plug-ins from the Eclipse IDE installation are used as target platform.

11.2. Include all plug-ins in Java search

You can include all classes from the current target platform to be relevant for the Eclipse Java tools via the following setting: Window → Preferences → Plug-in Development → Include all plug-ins from target in Java search.



Part IV. Eclipse applications and product configuration files

Chapter 12. Product configuration file

12.1. The Product configuration file and the application

A *product configuration file* (in a shorter form this is called: *product*) defines the configuration of an Eclipse application. This includes icons, splash screen and the plug-ins (directly or via the usage of features) which are included in your application.

A product always points to one application class. The default application for Eclipse RCP applications is the `org.eclipse.e4.ui.workbench.swt.E4Application` class which should be sufficient for most scenarios. In case customers want to adjust the default Eclipse initialization logic they could also provide their own implementation, such implementation must implement the `IApplication` interface.

A product is a development artifact and is not required at runtime.

12.2. Creating a new product configuration file

A product is typically created in a separate project. In the *Plug-in Development* perspective you can create a new product configuration file via a right-click on a project and by selecting New → Product Configuration.

12.3. Using the product editor

You can edit the product file via a special editor. The product extension and the containing plug-in can be defined in the *Product Definition* section.

The screenshot shows the 'Overview' tab of the Eclipse Product Editor. At the top, there's a toolbar with icons for Overview, Synchronize, Configuration, Launching, Splash, Branding, Customization, Licensing, and Updates. Below the toolbar, the 'General Information' section contains fields for ID (empty), Version (empty), and Name ('to-do'). A checked checkbox indicates the product includes native launcher artifacts. The 'Product Definition' section is highlighted with a red border; it shows 'Product: com.example.e4.rcp.todo.product' and 'Application: org.eclipse.e4.ui.workbench.swt.E4Application'. Below these, a note says the configuration is based on plug-ins (radio button selected). The 'Testing' section lists steps to synchronize with the product's defining plug-in and to launch the application. The 'Exporting' section describes using the Eclipse Product export wizard to package and export the product. At the bottom, a navigation bar includes links for Overview, Dependencies, Configuration, Launching, Splash, Branding, Customization, Licensing, and Updates.

On the *Overview* tab of this editor you can start the product. Pressing the *Synchronize* link writes the relevant product configuration information into the `plugin.xml` file. Both settings are highlighted in the following screenshot.

Overview



General Information

This section describes general information about the product.

ID:

Version:

Name: **to-do**

The product includes native launcher artifacts

Product Definition

This section describes the launching product extension identifier and application.

Product: **com.example.e4.rcp.todo.product**

Application: **org.eclipse.e4.ui.workbench.swt.E4Application**

The [product configuration](#) is based on: plug-ins features

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](#)

Exporting

Use the [Eclipse Product export wizard](#) to package and export the product defined in this configuration.

To export the product to multiple platforms:

1. Install the RCP delta pack in the target platform.
2. List all the required fragments on the [Dependencies](#) page.

[Overview](#) [Dependencies](#) [Configuration](#) [Launching](#) [Splash](#) [Branding](#) [Customization](#) [Licensing](#) [Updates](#)

It is possible to enter an ID for the product. Avoid using the same ID for the product as for a plug-in as this might create problems during a product export. Convention is to add the .product extension to the ID.

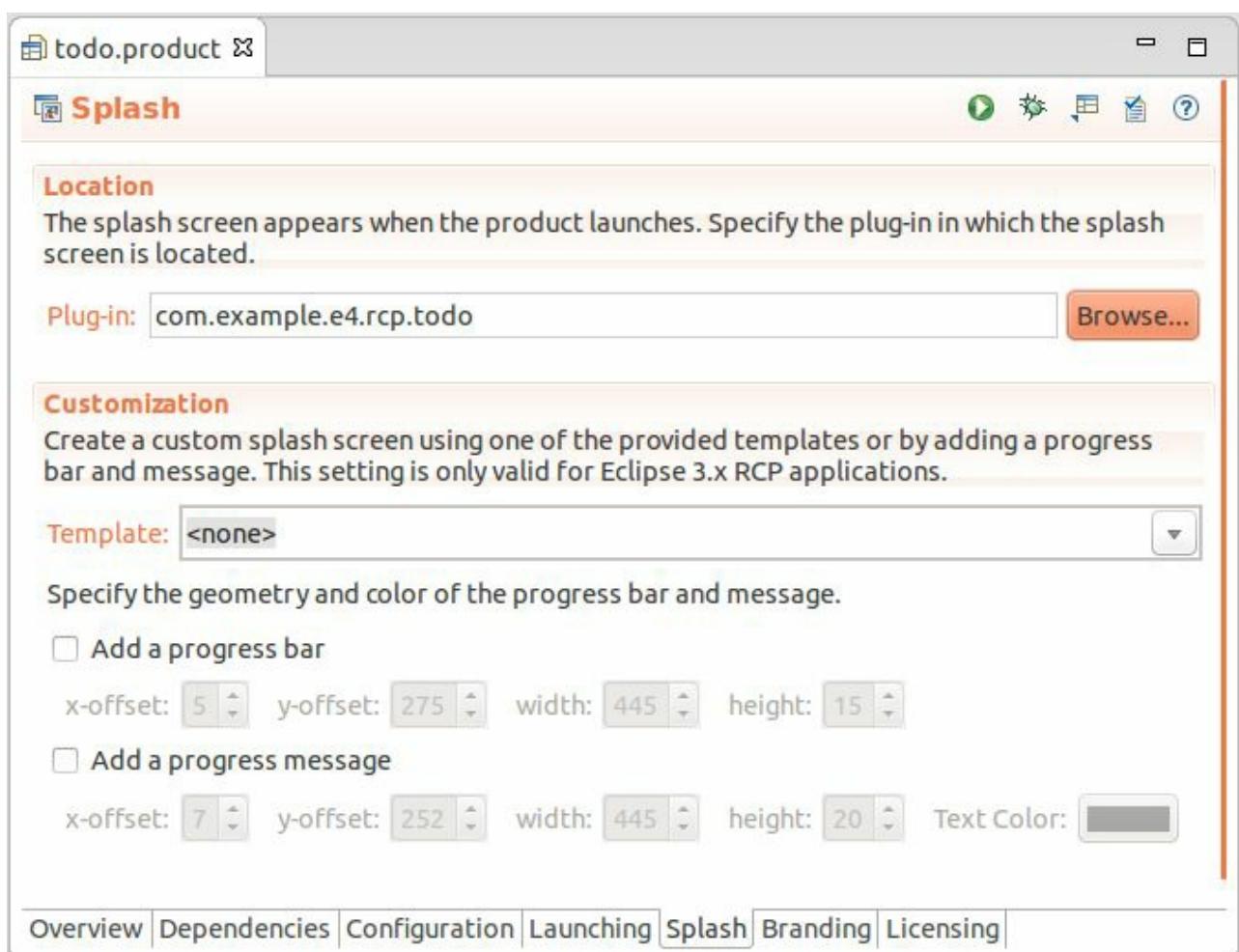
12.4. Define the plug-ins which are included in this product

On the *Dependencies* tab you can define the set of plug-ins which are included in this product, e.g., the plug-ins that are deployed with your product. Depending on the *Overview* selection these plug-ins are defined directly or via features. The usage of this tab is demonstrated in later exercises.

Chapter 13. Branding

13.1. Splash screen

The *Splash* tab allows you to specify the plug-in which contains the splash screen. The name is predefined as *splash.bmp* and it must be located in the root of the plug-in directory. Therefore, you need to put a file called *splash.bmp* file into the project main directory. Currently Eclipse supports only Bitmap (*.bmp*) files.



The predefined name and location of the splash screen image can be changed via the *osgi.splashPath* parameter, the file name can be changed with the *-showsplash path_to_file*. See [Section 14.4, “Launch configuration and Eclipse products”](#) to learn how to set launch parameters.

13.2. Icons, launcher name and program arguments

You can configure the launcher name and icon for your product. The launcher is the executable program that is created during the deployment of the product. A launcher is platform specific. For example the default launcher is called `eclipse.exe` on the MS Windows platform. This launcher has also an icon associated with it. To change the name and the icon, select the *Launching* tab of your product configuration.

Here you can specify the file name of the launcher and the icon which should be used. Make sure the format of the icons is correct, otherwise Eclipse will not use them.

Tip

The icon configuration depends on the platform you are using. Eclipse allows you to export your application for multiple platforms and uses the correct ones based on your product configuration.

In the *Launching Arguments* section you can specify parameters for your Eclipse application and arguments for the Java runtime environment. *Program Arguments* are parameters passed to the Eclipse application.

The relevant sections are highlighted in the following screenshot.

Overview



General Information

This section describes general information about the product.

ID:

Version:

Name: **to-do**

The product includes native launcher artifacts

Product Definition

This section describes the launching product extension identifier and application.

Product: **com.example.e4.rcp.todo.product**

Application: **org.eclipse.e4.ui.workbench.swt.E4Application**

The [product configuration](#) is based on: plug-ins features

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](#)

Exporting

Use the [Eclipse Product export wizard](#) to package and export the product defined in this configuration.

To export the product to multiple platforms:

1. Install the RCP delta pack in the target platform.
2. List all the required fragments on the [Dependencies](#) page.

13.3. Product configuration limitations

Currently the splash handlers, which can be registered via the *Customization* part of the *Splash* tab are not supported by the Eclipse 4 standard. Also, the configuration in the *About Dialog* and the *Welcome Page* section on the *Branding* tab is not directly supported in Eclipse 4 RCP applications.

Chapter 14. The usage of run configurations

This chapter describes the usage of the *run configuration* and highlights typical problems with them.

14.1. What are run configurations?

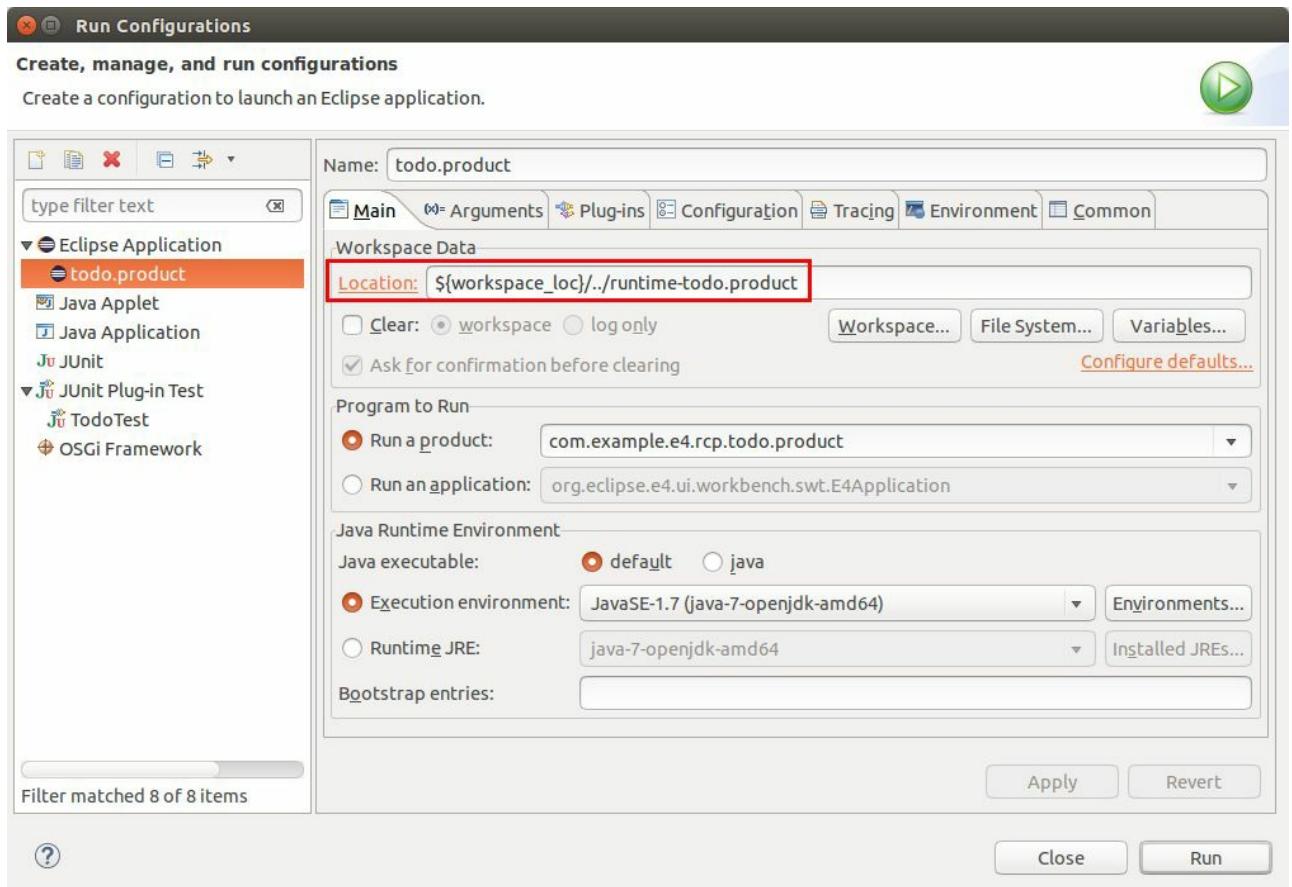
A *run configuration* defines the environment which will be used to execute a generic launch. For example, it defines arguments to the Java virtual machine (VM), plug-in (classpath) dependencies, etc.

If you start an Eclipse application the corresponding run configuration is automatically created or updated.

14.2. Reviewing run configurations

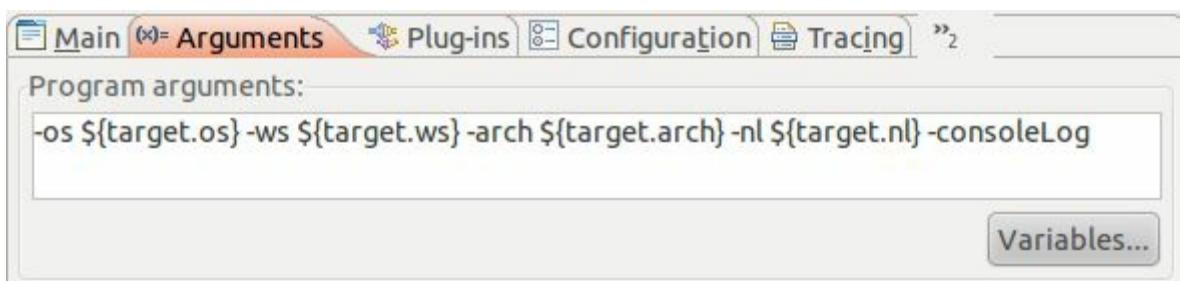
To review and edit your run configurations select Run → Run Configurations... from the Eclipse menu.

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.



14.3. Run arguments

The run configuration allows you to add additional start arguments for your application on the *Arguments* tab. By default Eclipse includes already several parameters, 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 14.1. Launch parameters

Parameter	Description
<code>consoleLog</code>	Error messages of the running Eclipse application are written to standard-out (<code>System.out</code>) which can be viewed in the Eclipse IDE <i>Console</i> view that started the RCP application.
<code>nl</code>	Specifies the locale used for your application. The locale defines the language specific settings, i.e., which translation is used and the number, date and currency formatting.
<code>console</code>	For example <code>-nl en</code> starts your application using the English language. This is useful for testing translations. Provides access to an OSGi console where you can check the status of your application.
<code>noExit</code>	Keeps the OSGi console open even if the application crashes. This allows to analyze the application dependencies even if the application crashes during startup.
<code>clearPersistedState</code>	Deletes cached runtime changes of the Eclipse 4 application model.

14.4. Launch configuration and Eclipse products

The launch configuration stores 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 frustration and time consuming error analysis. To ensure that you use the latest configuration from your product, start your application via the product file.

Chapter 15. Common launch problems

15.1. Checklist for common launch problems

Errors in the run configurations of Eclipse RCP application are frequently the source of problems. This chapter 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. The following table lists potential problems and solutions.

Table 15.1. Run configuration problems

Problem	Investigate
"Could not resolve module" message during start up.	Check that all required plug-ins are included in product configuration. Make sure that your product defines dependencies to all required plug-in features. See Section 15.2, “Finding missing plug-in dependencies during a product launch”
""java.lang.RuntimeException: No application id has been found." message during start up.	Bundles may also require a certain version of the 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.
Strange behavior but no error message.	See "Could not resolve module" message during start up error. In most cases also triggered by a missing plug-in dependency.
Runtime configuration is frequently missing required plug-ins	Check if your run configuration includes the <i>-consoleLog</i> parameter. This option allows you to view errors from Eclipse based applications in the <i>Console</i> view of the Eclipse IDE.
A change in the product <i>Dependencies</i> tab is not reflected in the run	Make sure that your product or your feature(s) includes all required dependencies.
A product updates an existing run configuration	start the product directly from the product definition.

configuration (e.g., a new plug-in is added but is not included in the run configuration)

file. If you select the run configuration directly, it will not be updated.

Application model changes are not reflected in the Eclipse 4 application.

Eclipse 4 persists user changes in the application delta file which is restored at startup. In development this might lead to situations where changes are not correctly applied to the application model, e.g., you define a new menu entry and the entry is not displayed in your application.

Services, e.g., key bindings or the selection service, are not working in an Eclipse 4 application.

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

In Eclipse releases before 4.3 every part needed to implement a *@Focus* method which places the focus on an SWT control. This error does not occur anymore with Eclipse 4.3 or a higher release.

Menu entries are disabled in the Eclipse application.

Ensure that the `HandlerProcessingAddon` class in the `org.eclipse.e4.ui.internal.workbench.addons` package is registered as model add-on. The bundle symbolic name is `org.eclipse.e4.ui.workbench`.

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 have pressed the *New...* button in your product configuration file and selected the *E4Application* as application to start. You can change the current setting in your `plugin.xml` file on the *Extensions* tab and in the details of the `org.eclipse.core.runtime.products` extension.

15.2. Finding missing plug-in dependencies during a product launch

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. This error is reported in the *Console* view, typically it is one of the first error messages and you need to scroll up to see it.

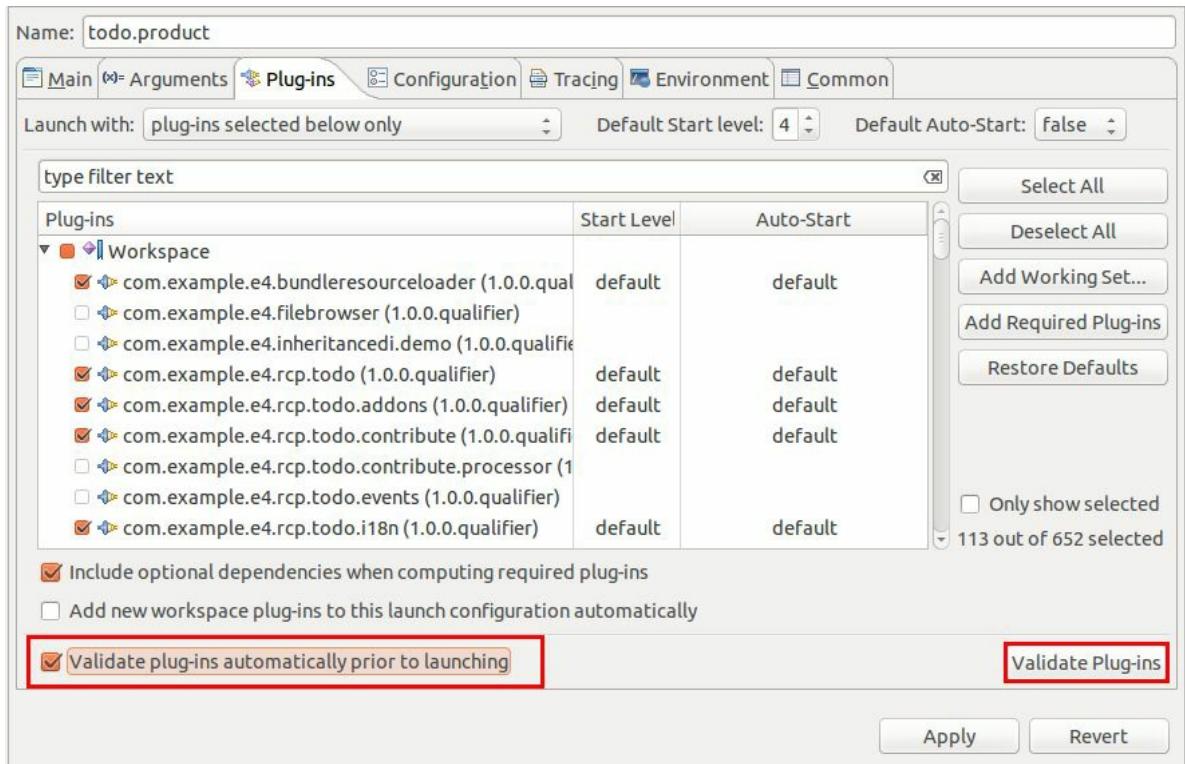
The following listing shows how this message typically looks like (formatting changed to make the text fit better).

```
org.osgi.framework.BundleException:  
    Could not resolve module: com.example.e4.rcp.todo.services [9]  
        Unresolved requirement:  
            Require-Bundle: com.example.e4.rcp.todo.events;  
                bundle-version="1.0.0"
```

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

Tip

Eclipse can check for missing dependencies automatically before you run the Launch configuration. On the *Plug-ins* Tab press the *Validate Plug-ins* button or 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.



Avoid fixing problems with dependencies in the run configuration because the run configuration is created and updated based on the product configuration file. So always ensure that the product file is correctly configured instead of changing the derived information. The product configuration is used for the export of your product, hence an error in the product dependencies results in an exported application which cannot be started.

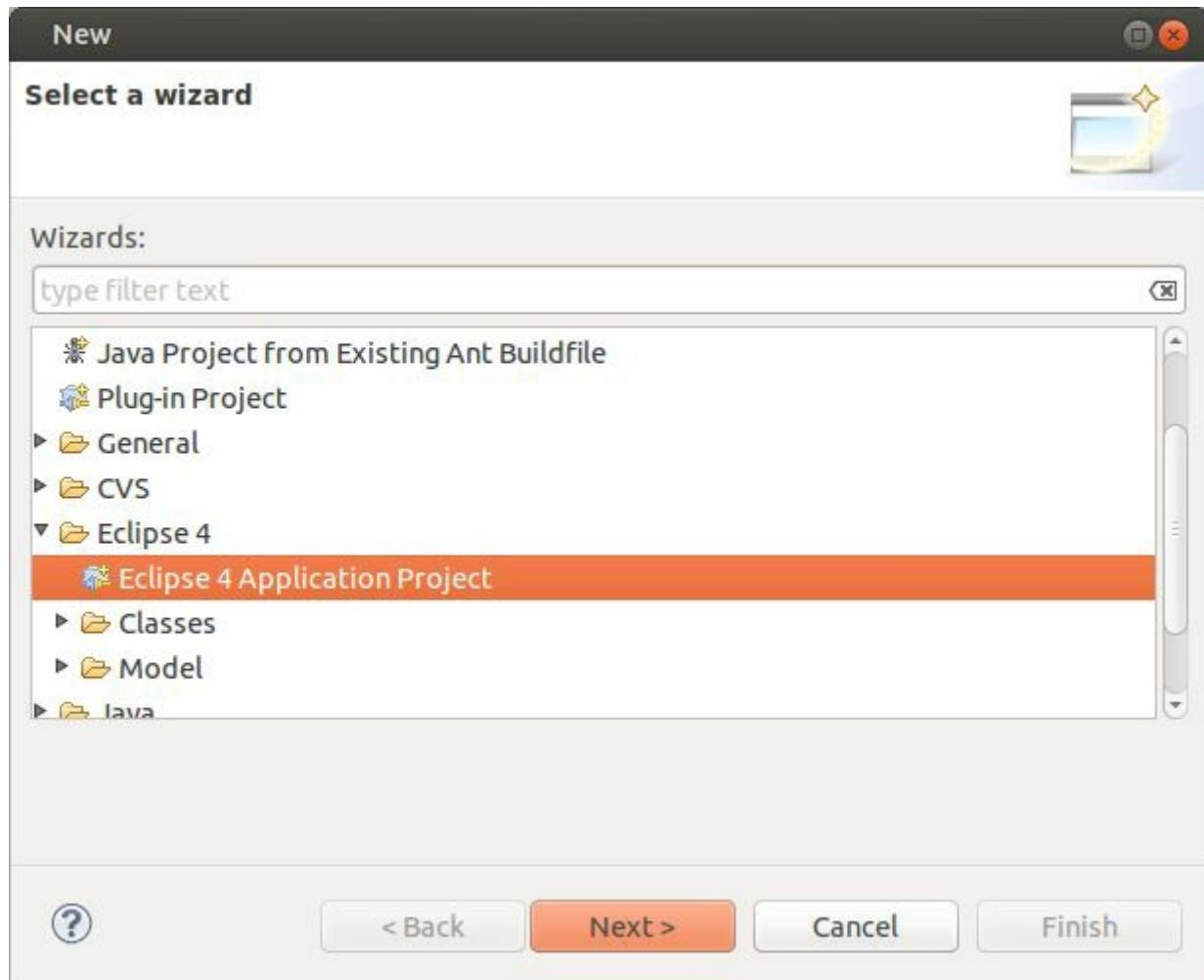
Chapter 16. Exercise: Create and run an RCP application

16.1. Target

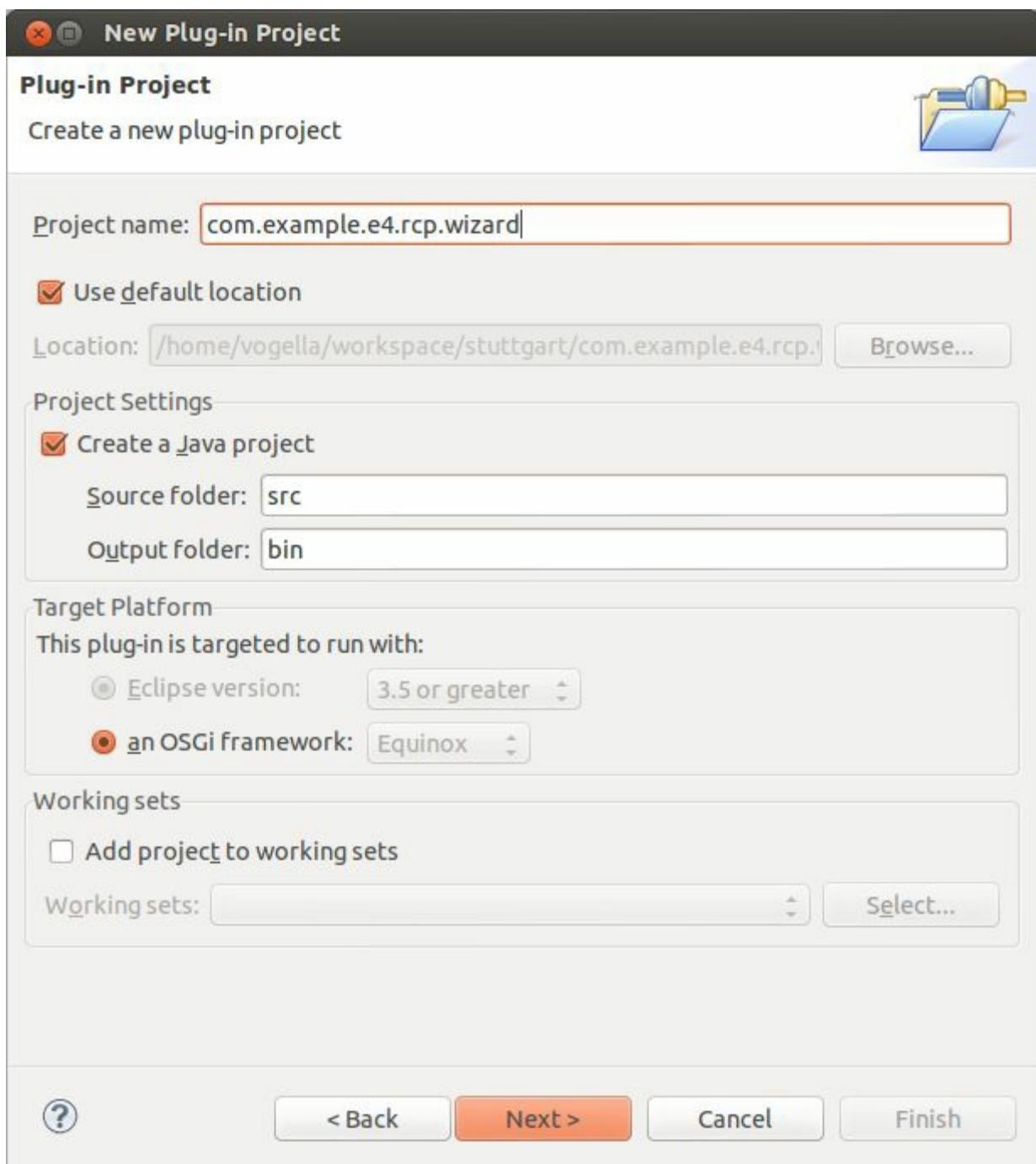
The following exercise demonstrates how to use the project creation wizard provided by the *Eclipse e4 tooling* project to create an Eclipse RCP application. It also shows how to start the application via the Eclipse IDE.

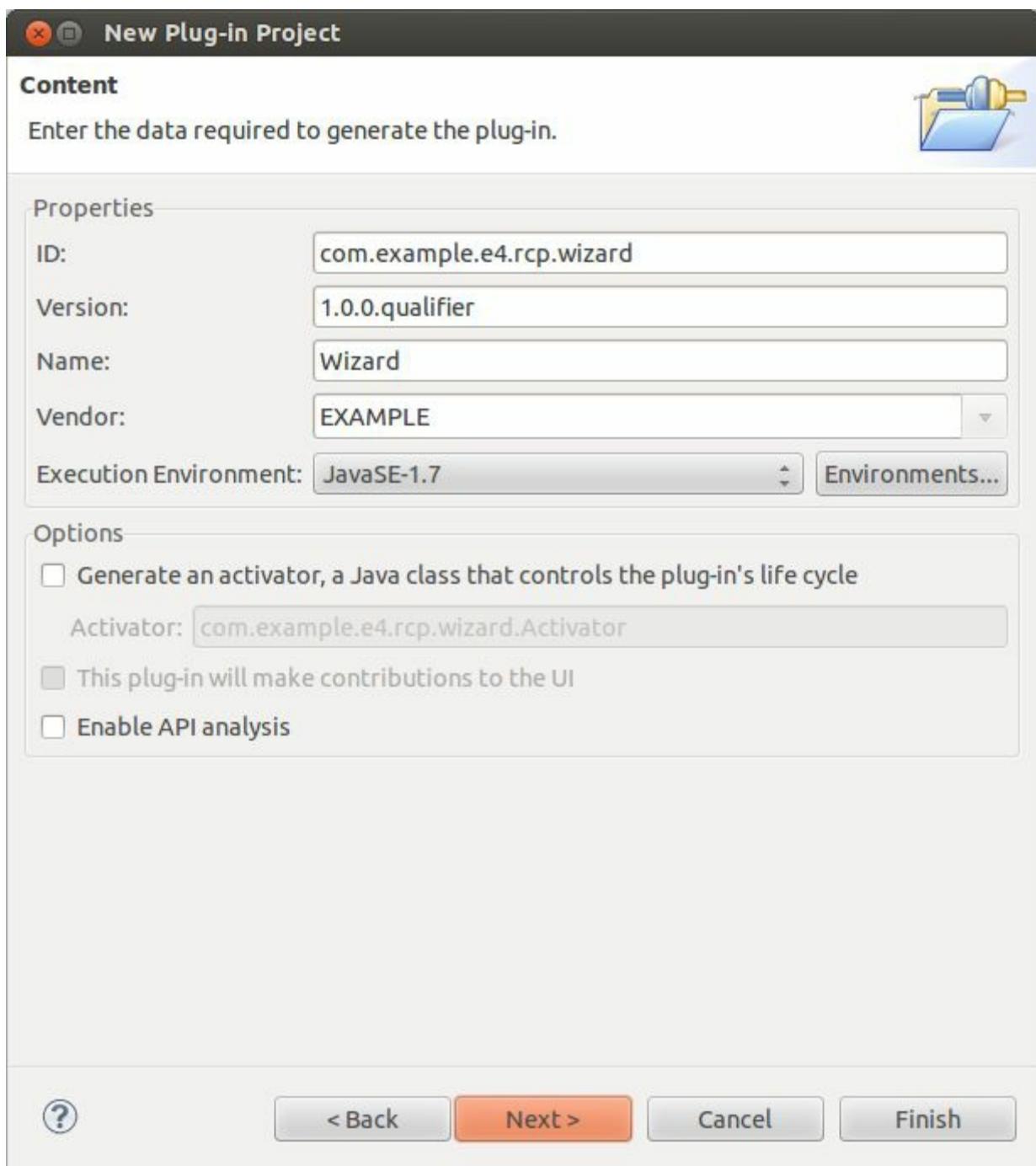
16.2. Create project

Select File → New → Other... → Eclipse 4 → Eclipse 4 Application Project from the menu of your Eclipse IDE.

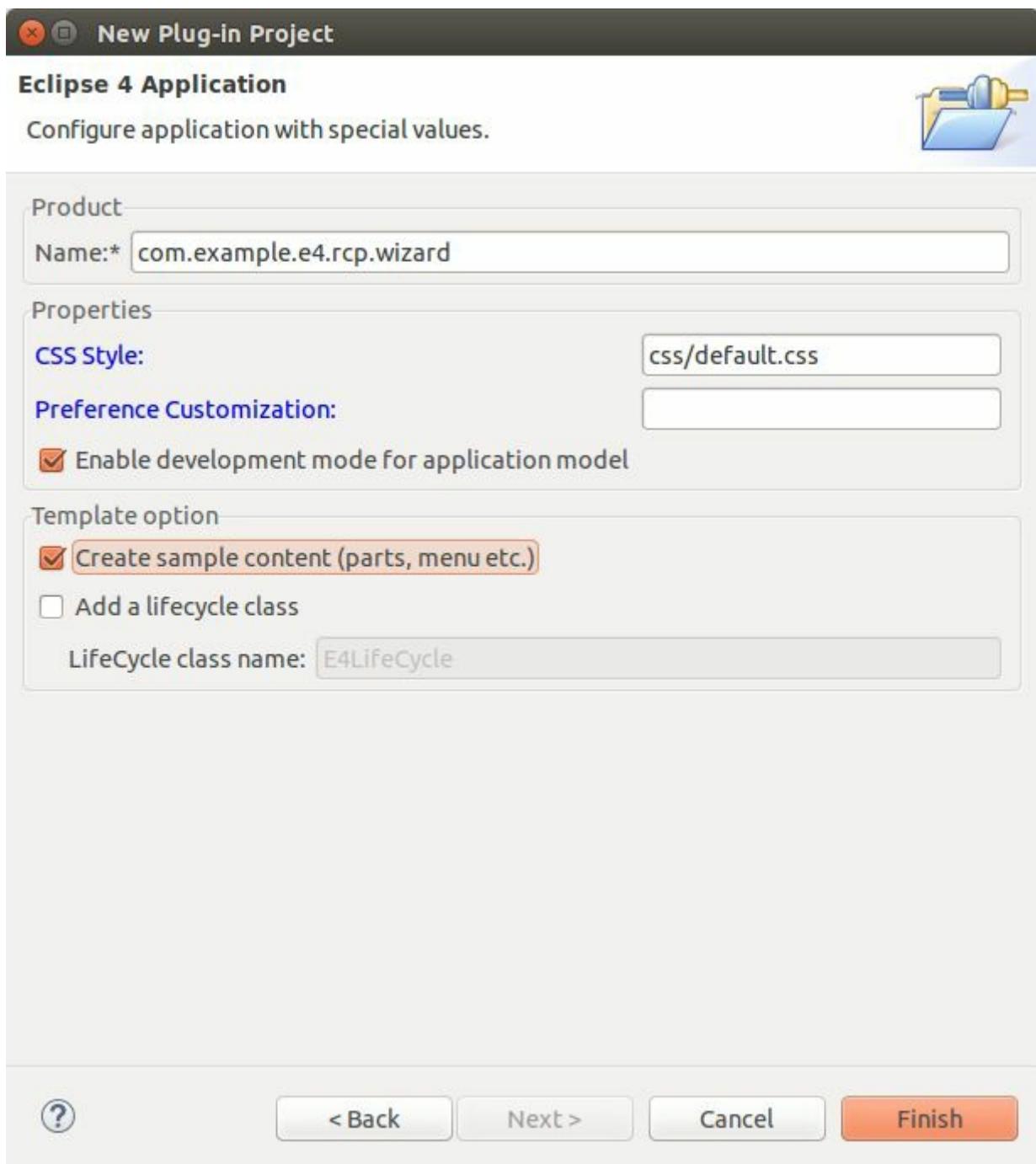


Create a project called `com.example.e4.rcp.wizard`. Leave the default settings on the first two wizard pages. These settings are similar to the following screenshots.





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



Note

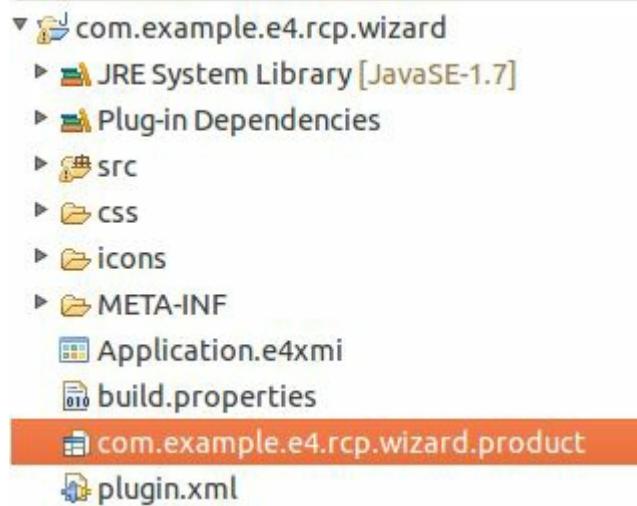
The *Enable development mode for application model* flag adds the `clearPersistedState` flag to the product configuration file. This ensures that changes during development in your application model are always visible. See [Section 30.1, “Delete the persisted user changes at startup”](#) for more information. Via the *Create sample content (parts, menu etc.)* flag you configure that the generated application should contain example content, e.g., a view and some

menu and toolbar entries.

This wizard creates all the necessary files to start your application.

16.3. Launch your Eclipse application via the product file

Open the generated product file by double-clicking on it 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 selection is highlighted in the following screenshot.

com.example.e4.rcp.wizard.product

Overview

General Information
This section describes general information about the product.

ID:

Version: 1.0.0.qualifier

Name: com.example.e4.rcp.wizard

The product includes native launcher artifacts

Product Definition
This section describes the launching product extension identifier and application.

Product: com.example.e4.rcp.wizard.product

Application: org.eclipse.e4.ui.workbench.swt.E4Application

The product configuration is based on: plug-ins features

Testing

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

Exporting
Use the [Eclipse Product export wizard](#) to package and export the product defined in this configuration.

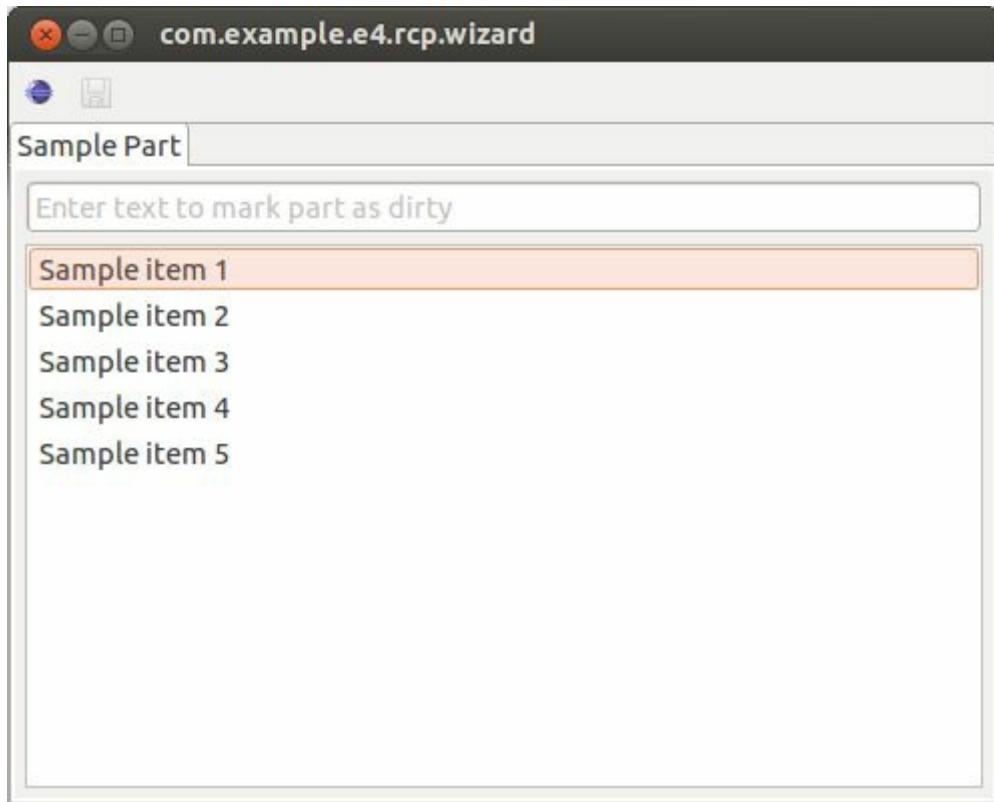
To export the product to multiple platforms:

- Install the RCP delta pack in the target platform.
- List all the required fragments on the [Dependencies](#) page.

Overview | Dependencies | Configuration | Launching | Splash | Branding | Licensing

16.4. Validating

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



Note

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

Chapter 17. Features and feature projects

17.1. What are feature projects and features?

Similar to plug-in projects you can also create feature projects via the Eclipse wizards. An Eclipse feature project contains *features*.

A feature describes a list of plug-ins and other features which can be understood as a logical unit. It also has a name, version number and license information assigned to it.

A feature is described via a *feature.xml* file. The following listing shows an example of such a file.

S

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
    id="com.vogella.eclipse.featuretest.feature"
    label="Feature"
    version="1.0.0.qualifier"
    provider-name="vogella GmbH">

    <description url="http://www.example.com/description">
        [Enter Feature Description here.]
    </description>

    <copyright url="http://www.example.com/copyright">
        [Enter Copyright Description here.]
    </copyright>

    <license url="http://www.example.com/license">
        [Enter License Description here.]
    </license>

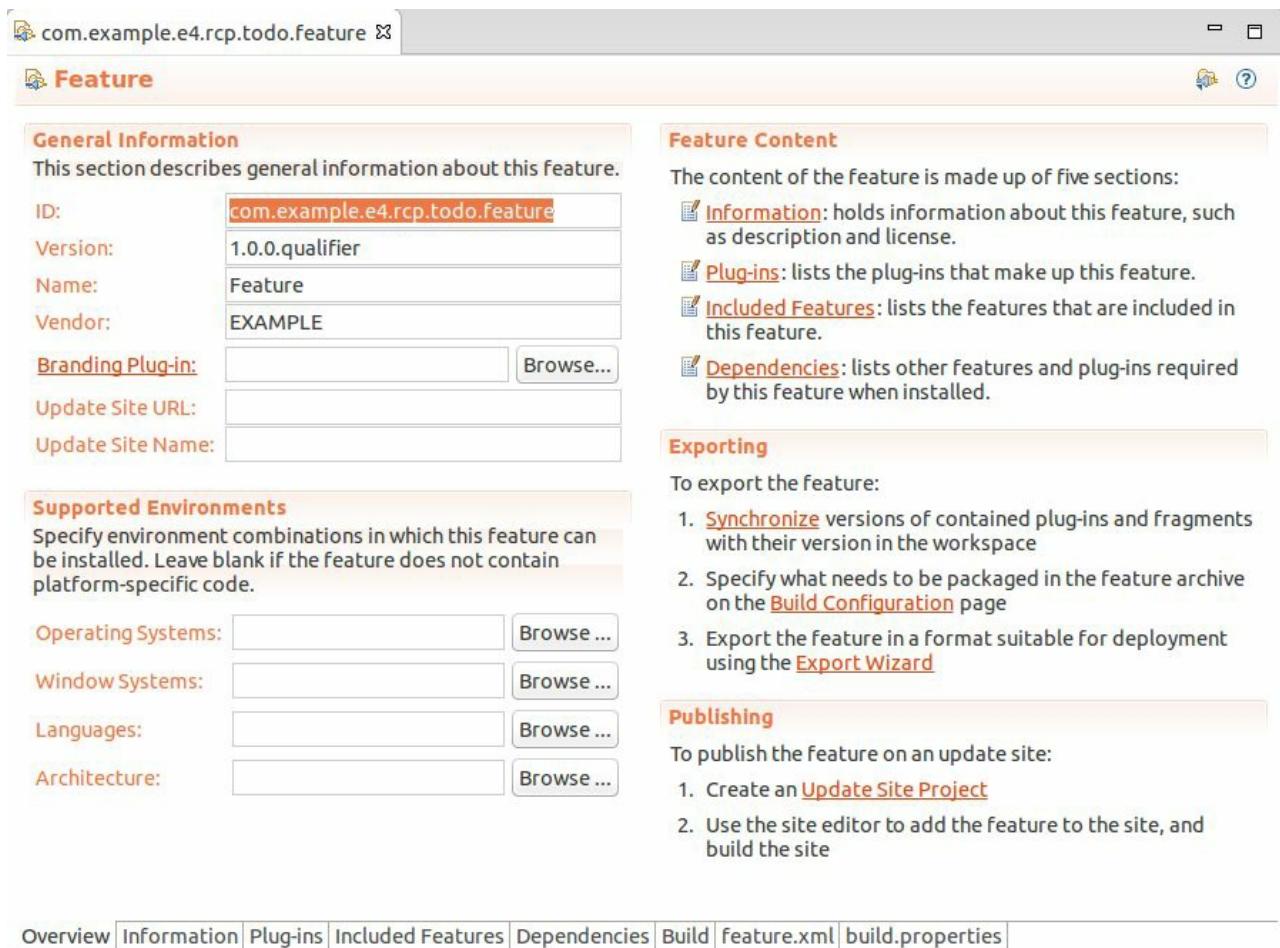
    <plugin
        id="com.vogella.eclipse.featuretest.feature"
        download-size="0"
        install-size="0"
        version="0.0.0"
        unpack="false"/>

</feature>
```

17.2. Creating a feature

You can create a new feature project via the following menu path: File → New → Other... → Plug-in Development → Feature Project.

If you open the `feature.xml` file you can change the feature properties via a special editor.



The *Information* tab allows you to enter a description and copyright related information for this feature.

The *Plug-ins* tab allows you to change the included plug-ins in the feature.

The *Included Features* tab allows you to include other features into this feature. Via the *Dependencies* tab you can define other features which must be present to use this feature.

Warning

If you want to add a plug-in to a feature, use the *Plug-ins* tab. A frequent error of new Eclipse developers is to add it to the *Dependencies* tab.

The *Build* tab is used for the build process and should include the `feature.xml` file. The last two tabs give access to the configuration files in text format.

17.3. Feature or plug-in based products

A product can either be based on plug-ins or on features. This setting is done on the *Overview* tab of the product configuration file.

Product Definition
This section describes the launching product extension identifier and application.

Product:	com.example.e4.rcp.todo.product	<input type="button" value="▼"/>	<input type="button" value="New..."/>
Application:	org.eclipse.e4.ui.workbench.swt.E4Application	<input type="button" value="▼"/>	
The product configuration is based on: <input type="radio"/> plug-ins <input checked="" type="radio"/> features			

On the *Dependency* tab you enter the plug-ins or features your products consists of.

Note

A product does not perform automatic dependency resolution. If you add a feature to your product and want to add its dependencies, press the *Add Required* button.

17.4. Advantages of using features

The grouping of plug-ins into logical units makes it easier to handle a set of plug-ins. Instead of adding many individual plug-ins to your product configuration file you can group them using features. That increases the visibility of your application structure.

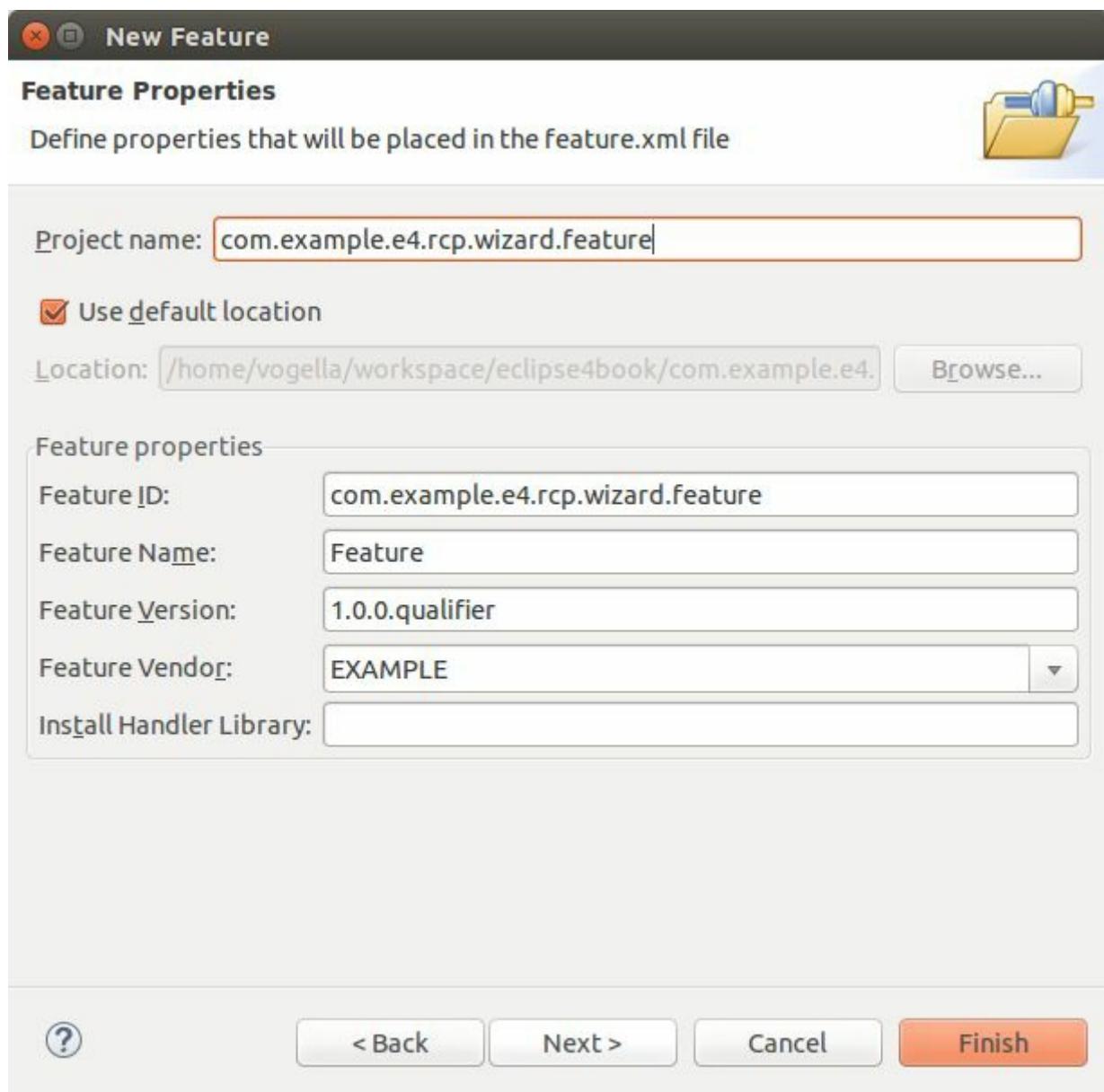
Features can be used by the Eclipse update manager, the build process and optionally for the definition of Eclipse products. Features can also be used as the basis for a launch configuration.

Eclipse provides several predefined features, e.g. the `org.eclipse.e4.rcp` for Eclipse based RCP applications.

Chapter 18. Exercise: Use a feature based product

18.1. Create a feature project

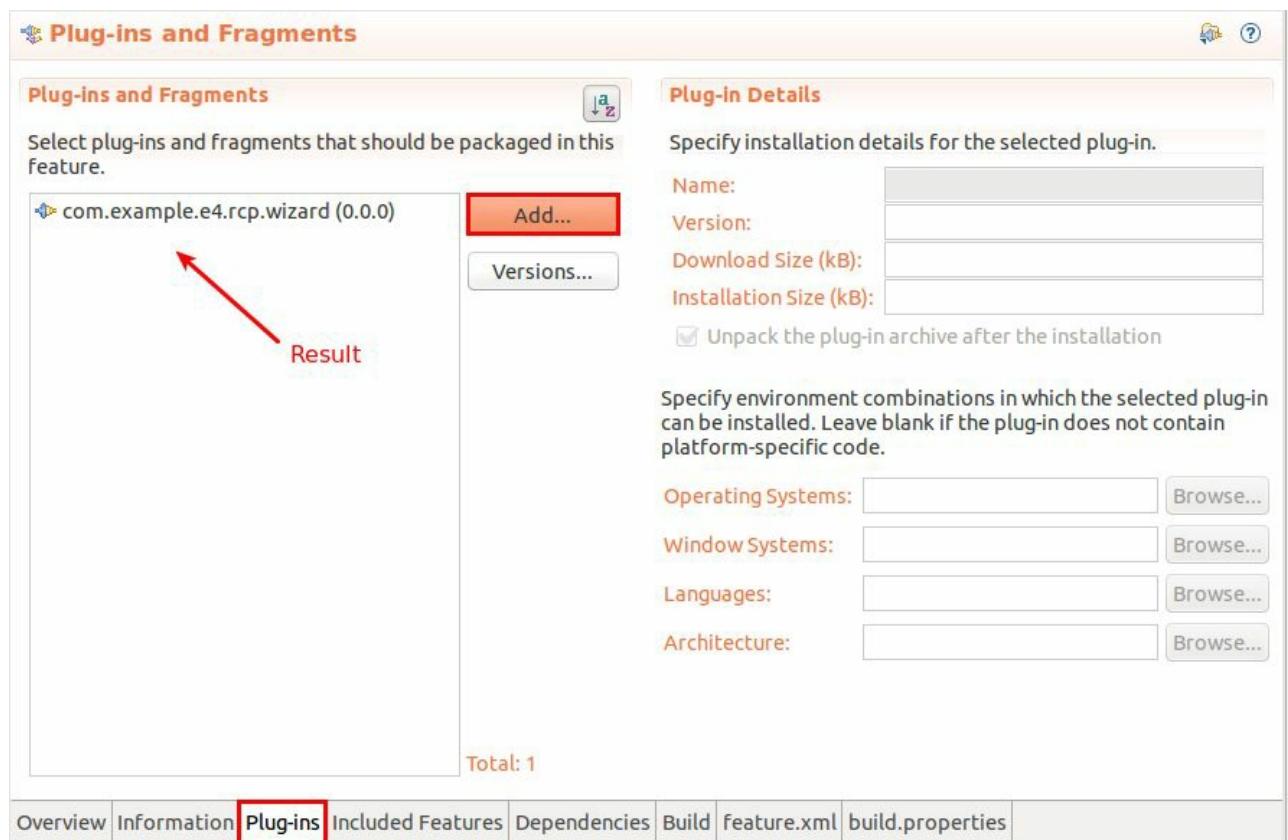
Create a new feature project called *com.example.e4.rcp.wizard.feature* via File → New → Other... → Plug-in Development → Feature Project. The following screenshot shows the first wizard page.



Press *Finish* on this first wizard page. This leaves the plug-in selection empty, you fill that in the next part of this exercise.

18.2. Include plug-in into feature project

Include the `com.example.e4.rcp.wizard` plug-in into the new feature. For this open the `feature.xml` file, select the *Plug-ins* tab and press the *Add...* button.



18.3. Change product configuration file to use features

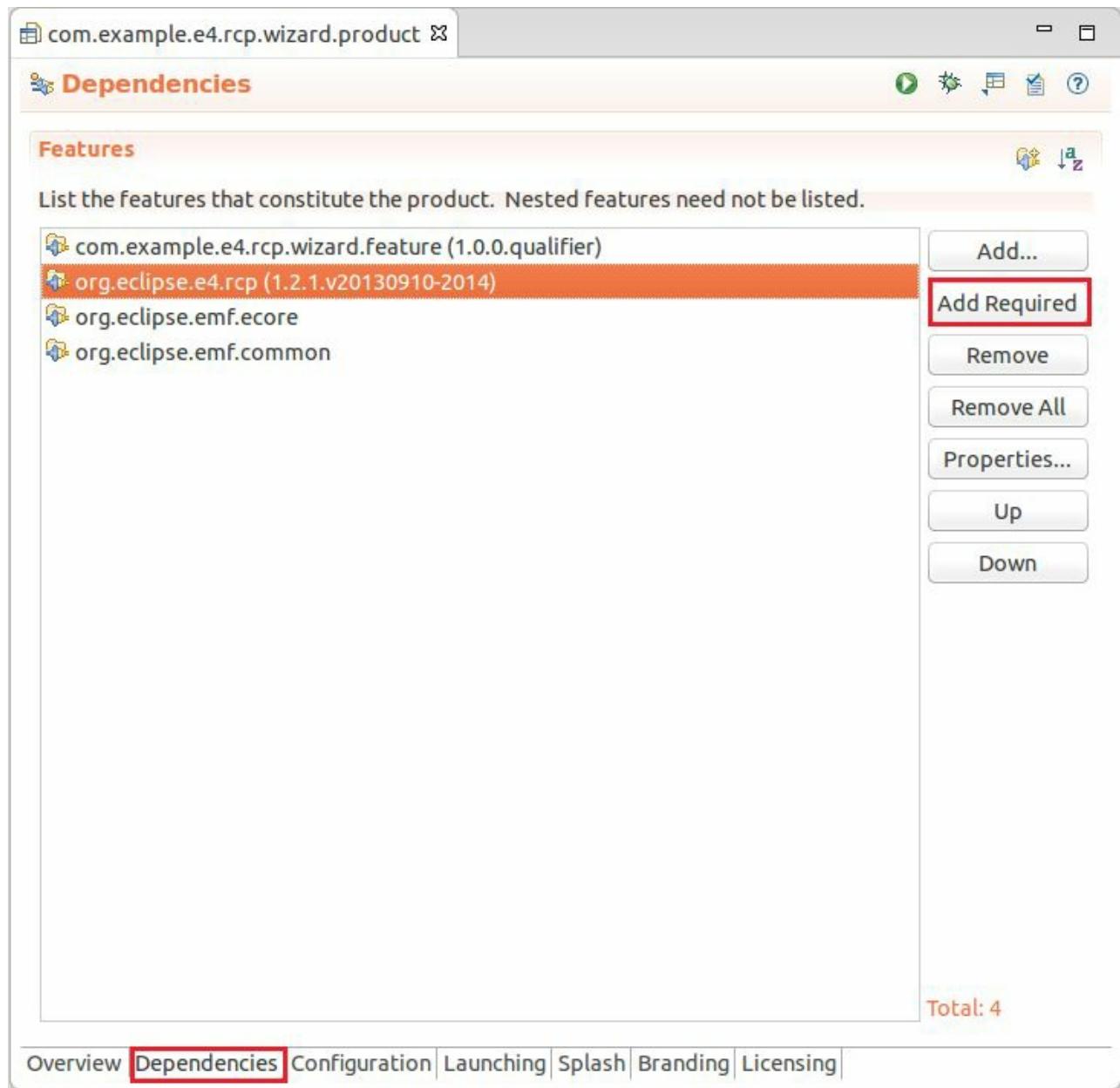
Open your product configuration file and select the *Overview* tab. Select the option that your product configuration file is based on features. This selection is highlighted in the following screenshot.

The screenshot shows the Eclipse Product Configuration interface with the 'Overview' tab selected. The 'Product Definition' section contains fields for 'Product' (com.example.e4.rcp.wizard.product) and 'Application' (org.eclipse.e4.ui.workbench.swt.E4Application). Below these, under 'The product configuration is based on:' is a radio button group where 'features' is selected, indicated by a red box. The 'Testing' section lists steps for synchronization and launching. The 'Exporting' section provides instructions for exporting the product. At the bottom, a navigation bar includes 'Overview' (which is redboxed), 'Dependencies', 'Configuration', 'Launching', 'Splash', 'Branding', and 'Licensing'.

18.4. Add features as dependency to the product

Switch to the *Dependencies* tab in your product editor. Add the `com.example.e4.rcp.wizard.feature` feature to your product with the *Add...* button.

Afterwards add the `org.eclipse.e4.rcp` feature and press the *Add Required* button to add the dependencies of the `org.eclipse.e4.rcp` feature to the product.



Ensure that you have the four features from the screenshot added to your product.

18.5. Start the application via the product

Start your application via the product configuration file and ensure that the application starts correctly.

Tip

If the application does not work, ensure that: All four features are required to start your application and you MUST start via the product and not with an existing launch configuration to ensure that your new product configuration is used.

Part V. Deploying Eclipse applications

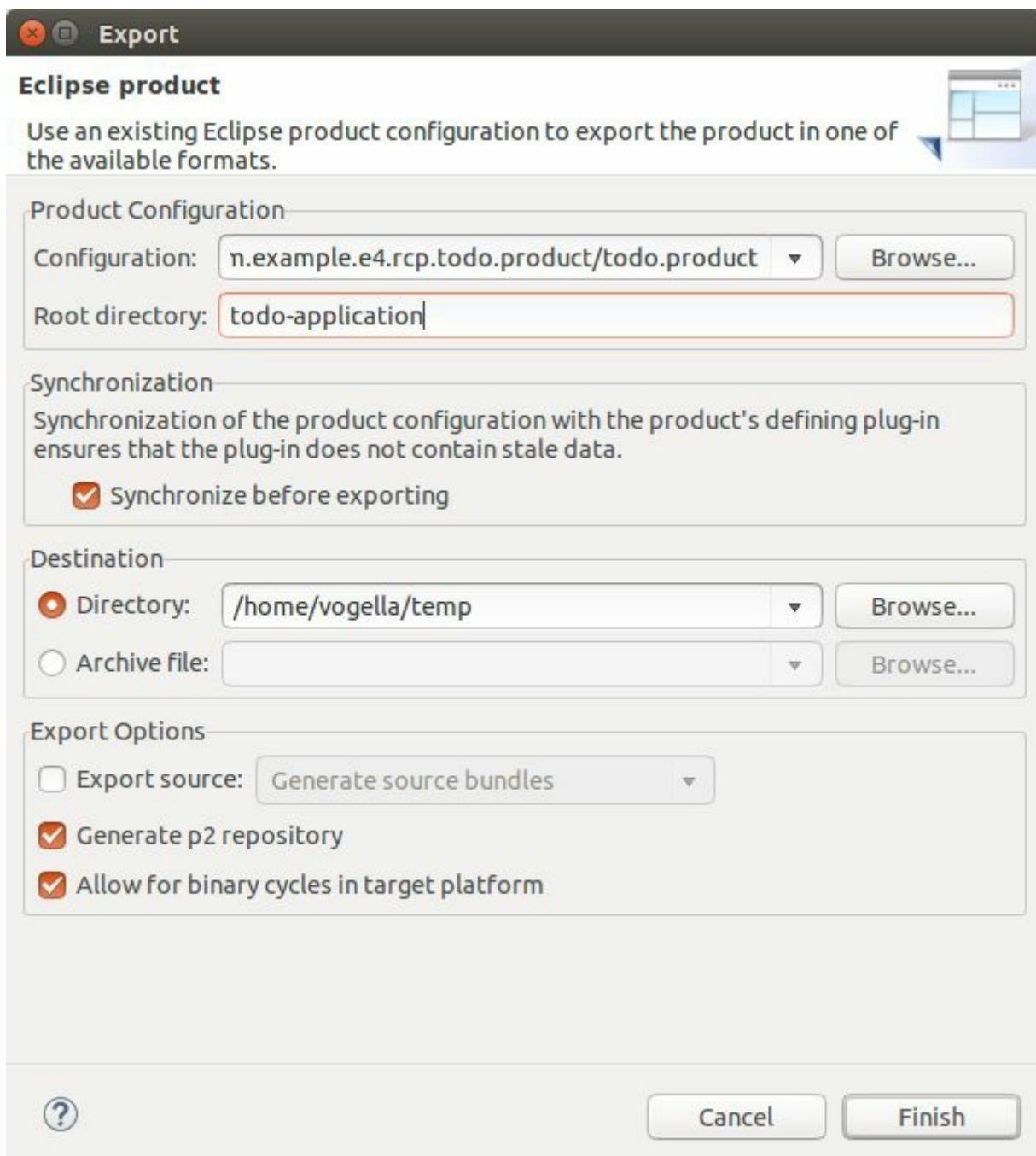
Chapter 19. Deployment of your RCP application

19.1. Creating a stand-alone version of the application

An Eclipse application needs to be exported (also called: deployed) to run outside of Eclipse. Your product configuration file can be used for that purpose. Exporting your product creates a folder with all required artifacts to run your application. This includes a native launcher specific to the platforms, e.g., Windows or Linux, you have exported it for.

19.2. Exporting via the product file

To export the product switch to your product configuration file, select the *Overview* tab and click on the *Eclipse Product export wizard* link. In the wizard you can specify the location of the exported product via the *Directory* property in the *Destination* group/section. The export wizard is depicted in the following screenshot.



The *Root directory* property can be used to specify a sub-folder in the

destination which contains the complete exported application. This is useful if you export the product as an archive file.

The resulting directory can be, for example, compressed (zipped) and shared with others. The export dialog allows you to create an archive file directly, which makes it easier to transfer all files to another machine.

If the *Generate p2 repository* option is selected, an (p2) update site is generated in a folder called *repository*. This folder can be used to update the Eclipse RCP application.

If you transfer the content of this directory to another machine (with the same architecture, e.g., Linux 64 bit), your application can start on this machine. Of course the correct Java version must be installed there.

19.3. Defining which artifacts are included in the export

The artifacts which are included in an export are defined by the *build.properties* file of the plug-in. Eclipse provides an graphical editor for this file.

Eclipse adds the compiled Java classes by default. You have to add other files manually, e.g., icons or splash screen images.

An Eclipse application started from the Eclipse IDE has access to all resources available in the IDE. But to make them available in the exported application you need to select them in the *build.properties* file.

It is good practice to include new required resources immediately in the *build.properties* file. This avoids errors after the export of your application.

19.4. Mandatory plug-in artifacts in build.properties

Make sure the following items (if available) are included in each plug-in of the exported application::

- META-INF/MANIFEST.MF
- plugin.xml
- other static files, e.g., icons, splash.bmp, etc.
- Application.e4xmi
- CSS files
- OSGi service definition files
- model fragments
- translation files

The screenshot below shows *build.properties* file for a plug-in with most of these components.

build.properties

Build Configuration

Custom Build

Runtime Information
Define the libraries, specify the order in which they should be built, and list the source folders that should be compiled into each selected library.

Add Library... **src/** Add Folder...

Up Down

Binary Build
Select the folders and files to include in the binary build.

- .classpath
- .project
- .settings**
 - Application.e4xmi
- META-INF**
- OSGI-INF**
- bin**
- build.properties
- css**
- icons**
- images**
 - plugin.xml
 - pom.xml
 - splash.bmp
- src**
- target**

Source Build
Select the folders and files to include in the source build.

- .classpath
- .project
- .settings**
 - Application.e4xmi
- META-INF**
- OSGI-INF**
- bin**
- build.properties
- css**
- icons**
- images**
 - plugin.xml
 - pom.xml
 - splash.bmp
- src**
- target**

Extra Classpath Entries

Build **build.properties**

19.5. Export for multiple platforms via the delta pack

The *delta pack* contains the platform specific features and plug-ins which are required to build and export Eclipse applications for multiple platforms. It also includes binary launchers for all platforms in the `org.eclipse.equinox.executable` feature.

See [Wiki entry for Cross-platform builds](#) for the usage. It basically requires that you add the delta pack update site to your target platform via the Window → Preferences → Plug-in Development → Target Platform menu entry.

19.6. More about the target platform

See [Part XXXVII, “Using target platforms”](#) for more information about the target platform configuration.

19.7. Including the required JRE into the export

You can also deploy your own RCP application bundled with a JRE to make sure that a certain JRE is used. An Eclipse application first searches in the installation directory for a folder called `jre` and for a contained Java-VM. If it finds one, then this JRE is used to start the Eclipse application.

To include the JRE from your running environment, select the *Bundle JRE for this environment with the product* flag on the *Launching* tab of your product configuration file.

19.8. Headless build

A *headless build* is an automatic build without user interaction and without a graphical user interface. It can be triggered from the command line. Typically, the build is automatically done via an additional software component called the *build server* which does so in a clean (and remote) environment.

Different solutions exist to do a headless build. Currently the most popular approach for building Eclipse RCP applications is based on Maven Tycho. Describing a headless build is beyond the scope of this description but you can see the online [Maven Tycho tutorial](#) for an introduction into headless builds for Eclipse RCP applications.

An example for a build server would be the *Jenkins* continuous integration (system). See the online [Jenkins tutorial](#) for an introduction into the setup, configuration and usage of Jenkins.

Chapter 20. Common product export problems

20.1. Problems with the export and log files

During the export or the start of the exported application you may encounter problems. If your application got successfully exported but cannot be started you find a log file ending with the `.log` extension in the `configuration` folder of your application. This file contains the error you encountered.

Alternatively the export process may fail. In both cases you can use this chapter as reference to try to find and fix the reason for the failure.

20.2. Export problem number #1: export folder is not empty

The most common problem is that the folder to which you export is not empty. If you export to a certain folder, ensure that the folder is empty. Exporting twice to the same folder may create file locks or results in error messages reporting version conflicts.

20.3. Checklist for common export problems

If the export encounters a problem please have a look into the following table for a solution:

Table 20.1. Problems with the product export

Problem	Possible cause
Export fails	Try using an empty target directory, sometimes the export cannot delete the existing files and therefore fails.
No executable file after the export	Check the flag "The product includes native launcher artifacts" in your .product file on the <i>Overview</i> tab.
Product could not be found	Validate that all dependencies are included in the product. Delete an existing launch configuration and restart the product from the IDE to see if everything is configured correctly.
Splash screen or other icons are missing	Check the <i>build.properties</i> file to see if all required images and icons are included in the export.
Splash screen is missing	Ensure that you have entered the defining plug-in in the "Splash" tab on the product configuration file. If this is not set, the splash screen is not displayed after the export. Unfortunately, it is displayed if you start the plug-in from the Eclipse IDE.
Issues during start of the application	Check the log file in the workspace folder of your exported application to see the error messages during the start process. Alternatively add the "-consoleLog" parameter to the ".ini" file in folder of the exported application.
applicationXMI argument is missing	Check the <i>build.properties</i> file to see if the <i>Application.e4xmi</i> and the <i>plugin.xml</i> files are included in the export.
Service could not be found or injected	Make sure that the bundle which provides the service has the <i>Activate this plug-in when one of its classes is loaded</i> flag set. Also make sure that the <i>org.eclipse.equinox.ds</i> bundle is started automatically with a <i>Start Level</i> less than 4.
Application ID could not be found	Define a start level of 1 and set auto-start to true for the <i>org.eclipse.core.runtime</i> plug-in.

Translations not Ensure via the *build.properties* file of the relevant plug-in available in the that the files containing the translations are included in the exported export.
product

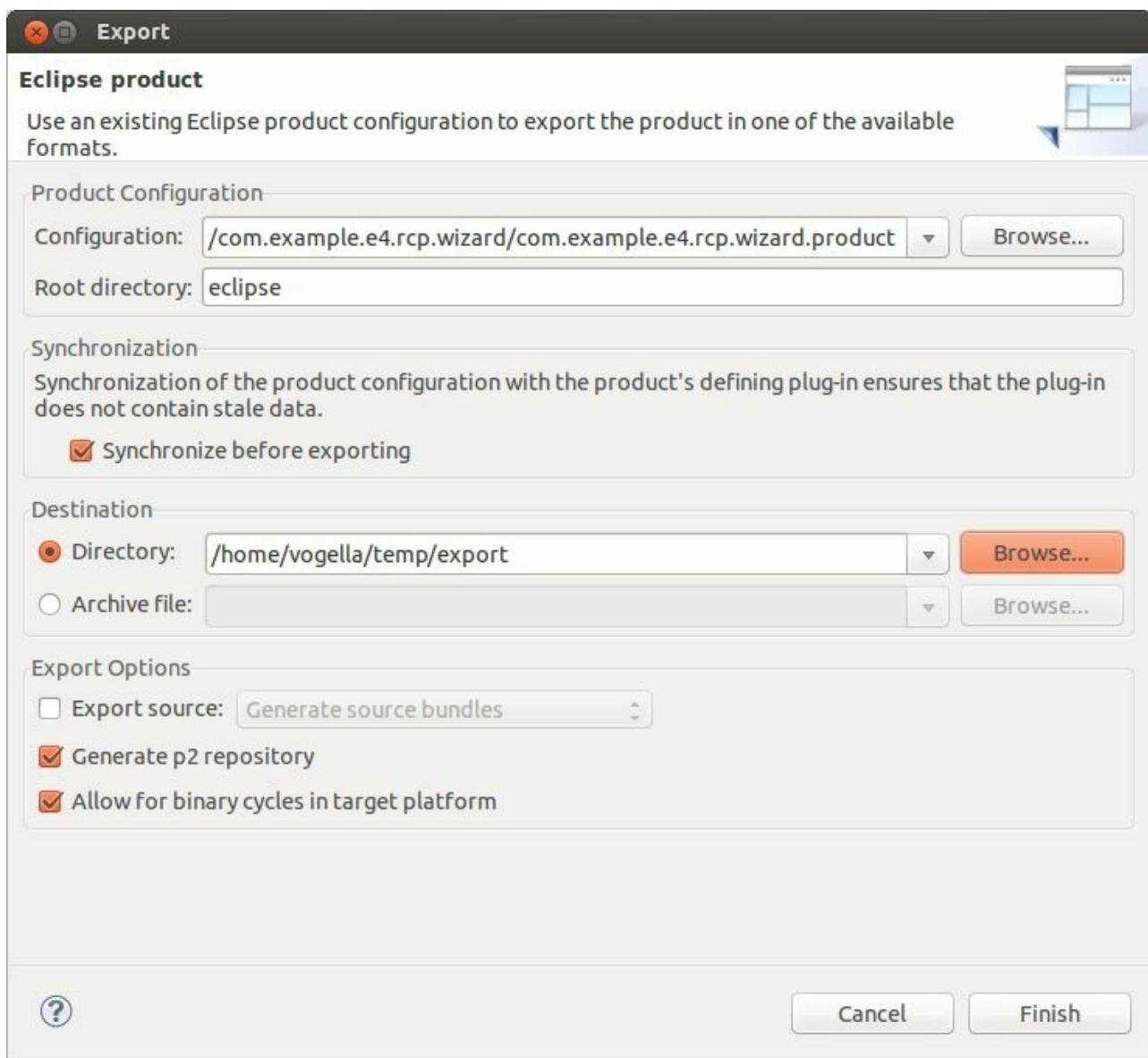
Chapter 21. Exercise: Export your product

21.1. Export your product

Use the product file of your `com.example.e4.rcp.wizard` plug-in to export your Eclipse application.

The screenshot shows the 'Overview' tab of the Eclipse Product Configuration interface. At the top right, there is a toolbar with several icons. One icon, which is a blue square with a white grid, is highlighted with a red box and has a red arrow pointing towards it from the bottom left. Below the toolbar, the 'General Information' section is visible, containing fields for 'ID', 'Version', and 'Name'. A checkbox labeled 'The product includes native launcher artifacts' is checked. To the right of this checkbox, the text 'Press to start the export' is displayed. The 'Product Definition' section follows, showing the selected 'Product' as 'com.example.e4.rcp.wizard.product' and 'Application' as 'org.eclipse.e4.ui.workbench.swt.E4Application'. Below this, a note states that the configuration is based on 'plug-ins'. The 'Testing' section contains two numbered steps: 'Synchronize' the configuration with the product's defining plug-in and 'Test the product by launching a runtime instance of it'. Under 'Testing', there are two links: 'Launch an Eclipse application' and 'Launch an Eclipse application in Debug mode'. The 'Exporting' section provides instructions on how to use the 'Eclipse Product export wizard' to package and export the product. It also mentions that to export to multiple platforms, the RCP delta pack must be installed in the target platform and dependencies listed. At the bottom of the screen, a navigation bar includes tabs for 'Overview', 'Dependencies', 'Configuration', 'Launching', 'Splash', 'Branding', 'Licensing', and 'Updates'. The 'Overview' tab is currently selected.

Enter a new directory in the *Directory* field. If in doubt use the *Browse* button to find a valid directory. The following screenshot shows an example under Ubuntu.



Press the *Finish* button to start the export.

Warning

The exported directory should be empty. If you export twice to the same directory, delete its content, otherwise the export fails with a dependency conflict.

21.2. Validate that the exported application starts

After the export finishes check the *Root directory* folder in the *Directory* folder.

A double-click on the native launcher starts your application. Ensure that you can start the application from your exported directory.

If you face issues during the product export, check the list of common export problems from [Chapter 20, Common product export problems](#) and try to solve the problem.

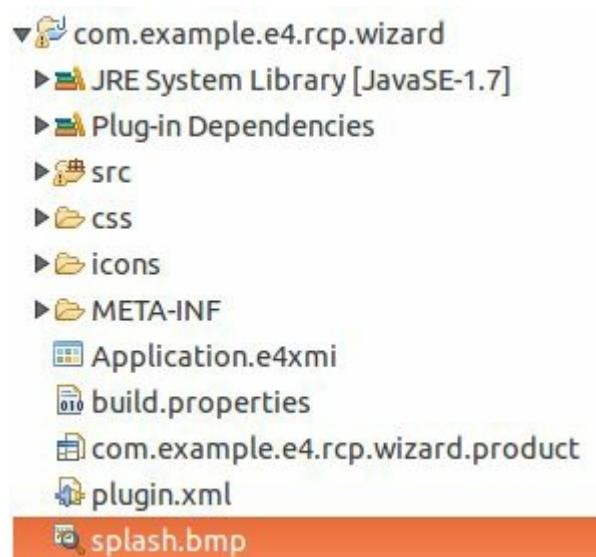
Chapter 22. Exercise: Splash screen and launcher name

22.1. Using a static splash screen

Add a splash screen to your application. For this create or download a *splash.bmp* bitmap file. You find an example under the following URL: [Example splash screen](#).

Add the *splash.bmp* file to the main directory of your application. You can copy and paste it into the *Package Explorer* view. If you add files outside of the Eclipse IDE, you need to *Refresh* (via F5) your project in the *Package Explorer* view, to see the file in Eclipse.

Afterwards, your project should look like the following screenshot.



Warning

The file name and the location of the file must be correct, otherwise Eclipse will not use your splash screen.

On the *Splash* tab of your product configuration file, define that your application plug-in contains the splash screen.

com.example.e4.rcp.wizard.product

Splash

Location
The splash screen appears when the product launches. Specify the plug-in in which the splash screen is located.

Plug-in: com.example.e4.rcp.wizard

Customization
Create a custom splash screen using one of the provided templates or by adding a progress bar and message. This setting is only valid for Eclipse 3.x RCP applications.

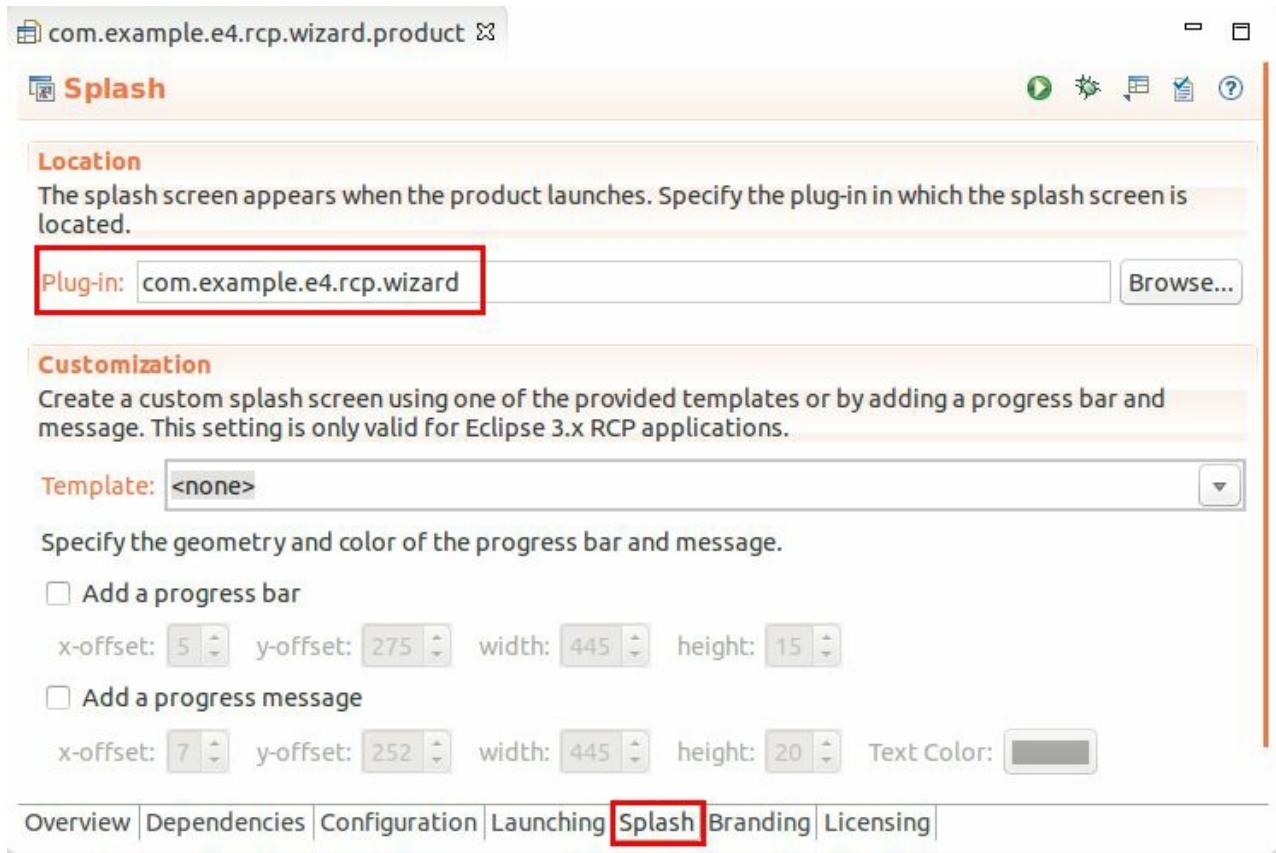
Template: <none>

Specify the geometry and color of the progress bar and message.

Add a progress bar
x-offset: 5 y-offset: 275 width: 445 height: 15

Add a progress message
x-offset: 7 y-offset: 252 width: 445 height: 20 Text Color:

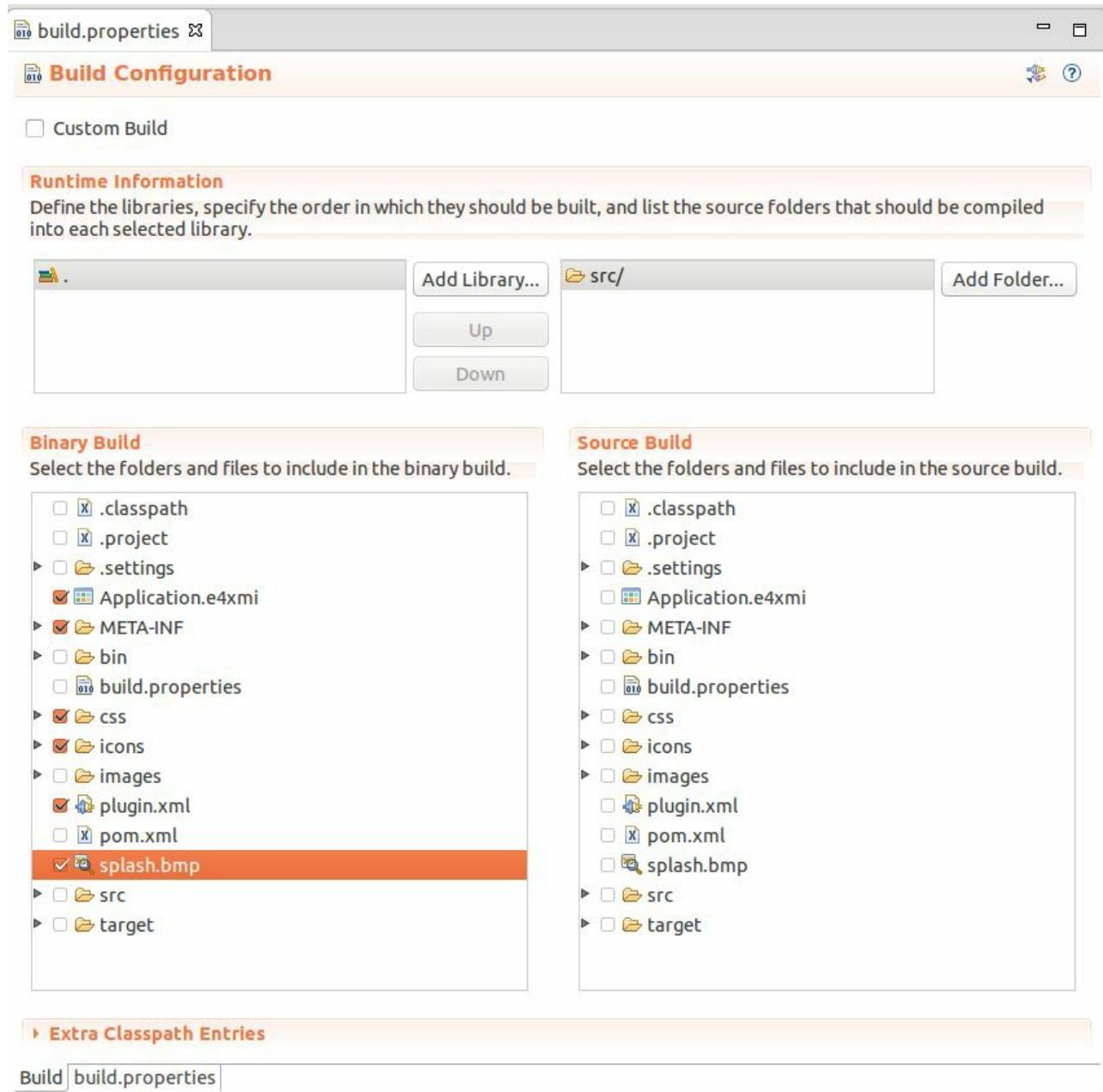
Overview | Dependencies | Configuration | Launching | **Splash** | Branding | Licensing



Start your application from the Eclipse IDE and verify that the splash is displayed.

22.2. Include a splash screen into the exported application

Configure that the `splash.bmp` file is included into the exported application by adding it to the `build.properties` file of your application plug-in.



Export your product again and ensure that the splash screen is also shown if you start the exported application.

Warning

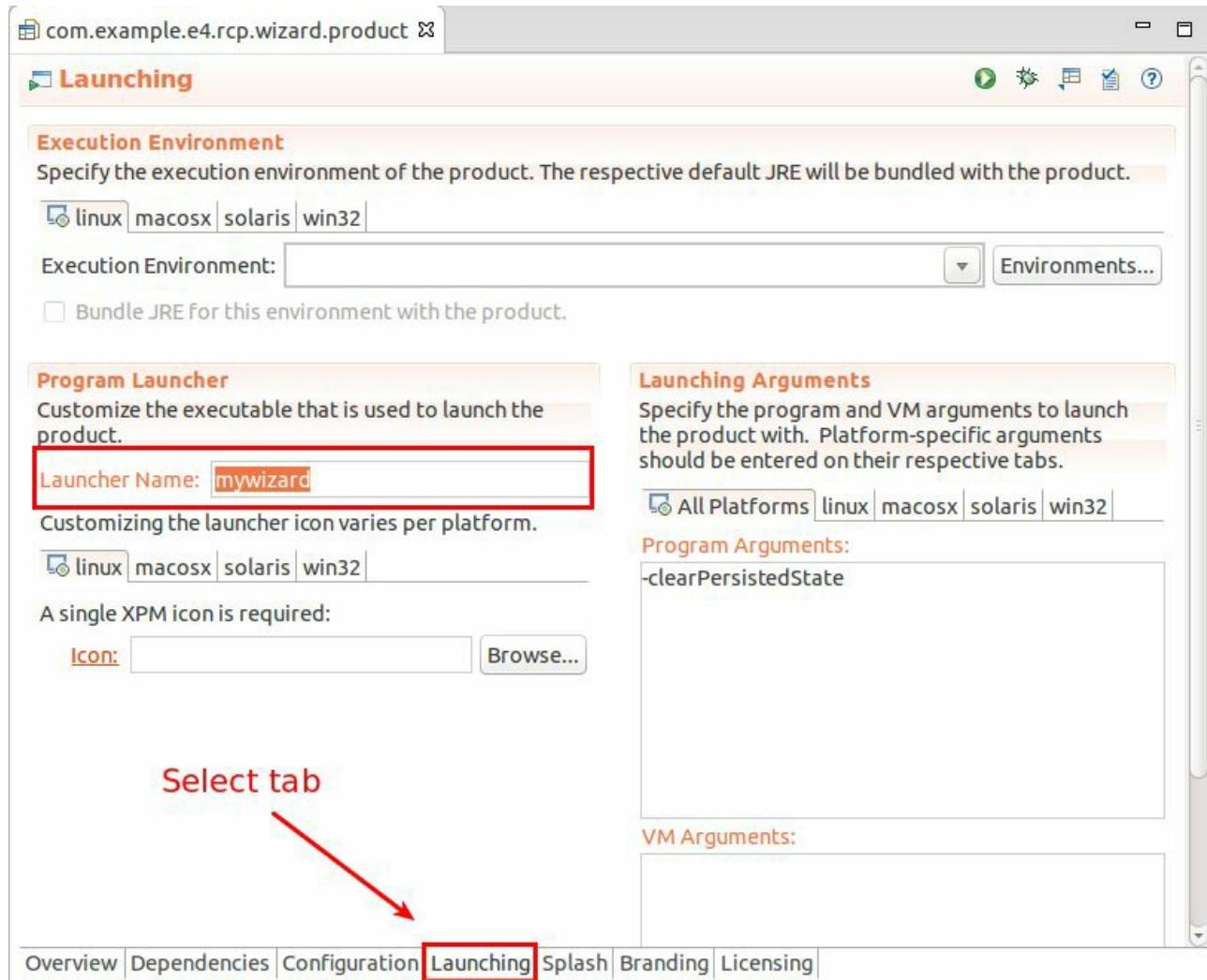
Make sure that the target directory of the export is empty before you

start the export. See [Chapter 20, *Common product export problems*](#) for details.

22.3. Change launcher name

The Eclipse export wizards uses *eclipse* as default for your application launcher name, e.g., on MS Windows this results in an *eclipse.exe* launcher file.

Change this launcher name to *mywizard* on the *Launching* tab of your product configuration file.



Export your application via the product file again and validate that the launcher name has changed.

Part VI. Application model

Chapter 23. Eclipse application model

23.1. What is the application model?

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

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

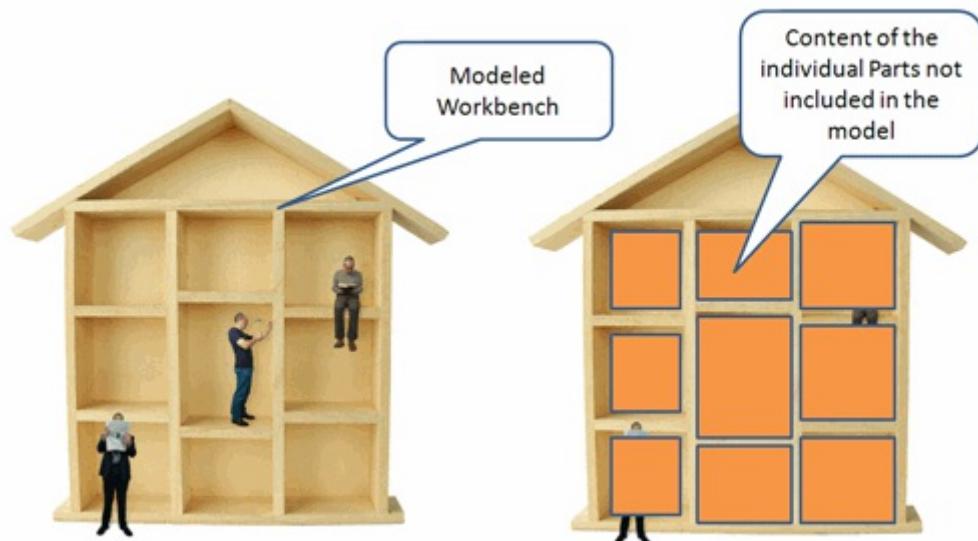
Each model element has attributes which describe its current state, e.g. the size and the position of a window. The application model also expresses the relationship of the model elements via a hierarchy.

23.2. Scope of the application model

The application model defines the structure of the application. For example, it describes which parts are available. It also describes the properties of the parts, e.g., if a part can be closed, its label, ID, etc.

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

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



23.3. How do you define the application model?

The base of the application model is typically defined as a static file. By default, it is called `Application.e4xmi` and located in the main directory of the plug-in which defines the product extension.

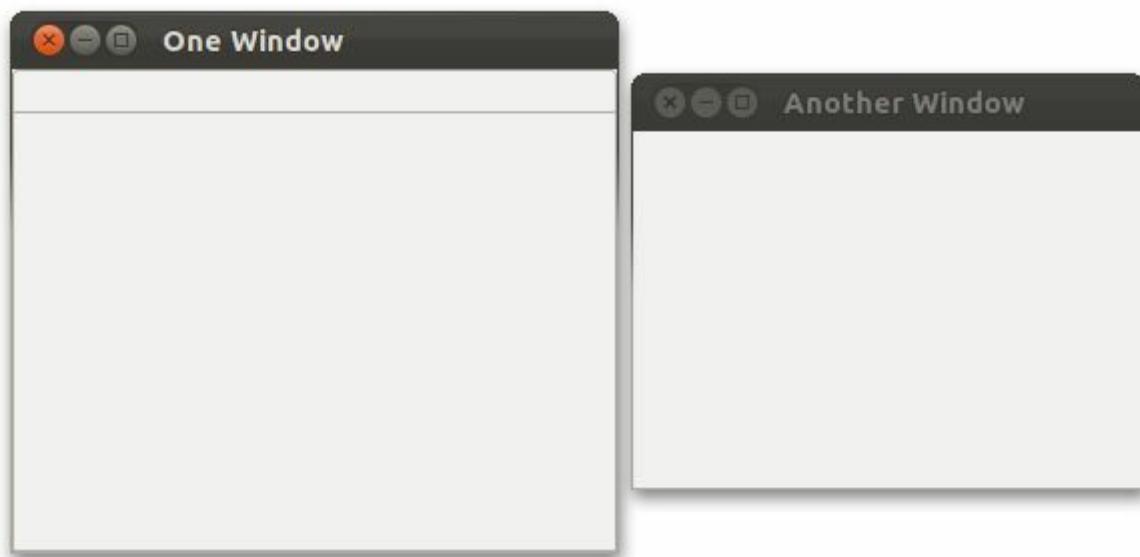
This file is read at application startup and is used to construct the initial application model. Changes made by the user are persisted and reapplied at startup.

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

Chapter 24. Important user interface model elements

24.1. Window

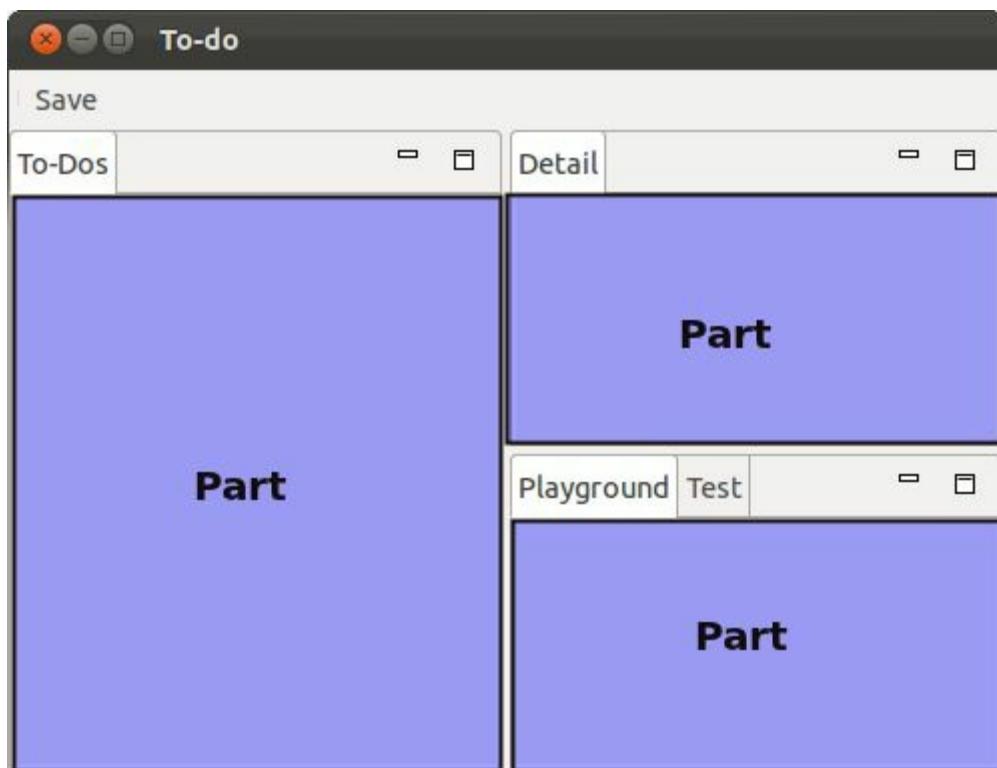
Eclipse applications consist of one or more windows. Typically, an application has only one window, but you are not limited to that, e.g., if you want to support multiple connected monitors.



24.2. Parts

What are parts?

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



Parts behaving as views or as editors

Parts are typically classified into *views* and *editors*. The distinction between views and editors is not based on technical differences, but on a different concept of using 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 the user to open an editor for the selected set of data.

An example for a view in the Eclipse IDE is the *Package Explorer*, which allows you to browse the files of Eclipse projects. If you change data in the

Package Explorer, e.g., if you rename a file, the file name is directly changed on the file system.

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

For example, the *Java* editor is used to modify Java source files. Changes to the source file are applied once the user selects the *Save* button. A dirty editor tab is marked with an asterisk left to the name of the modified file.



24.3. Part container

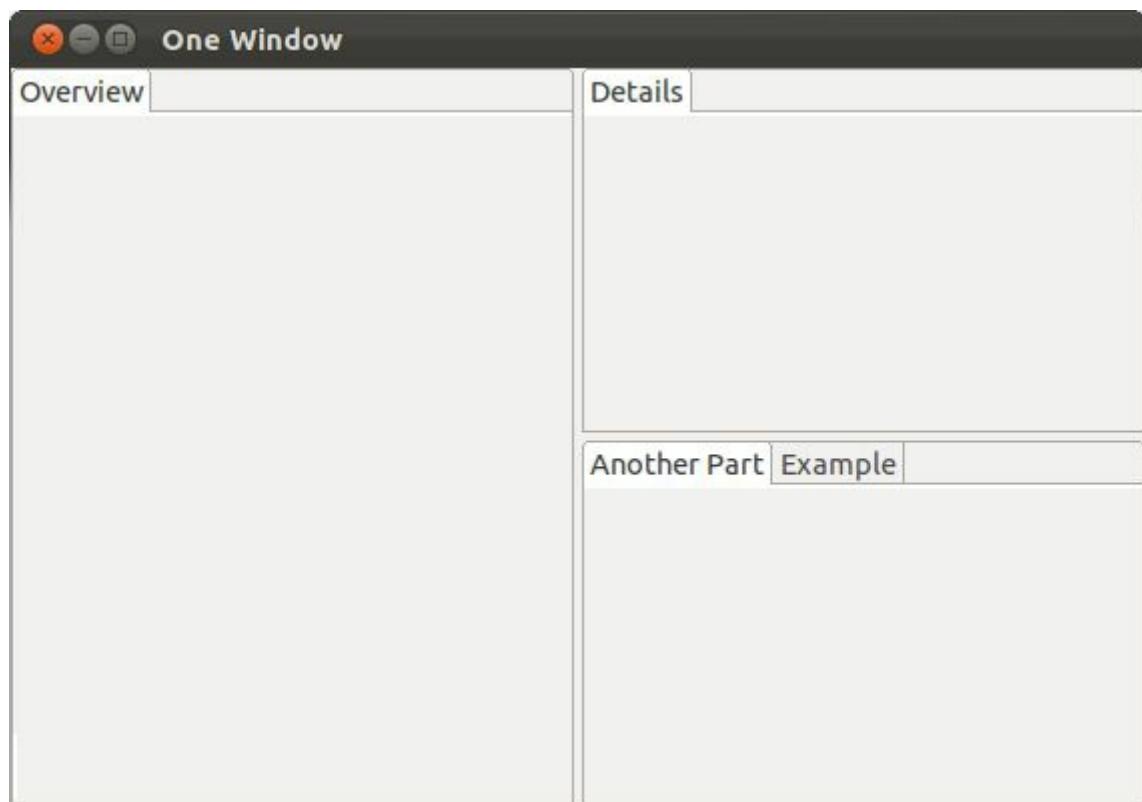
Part stack and part sash container

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

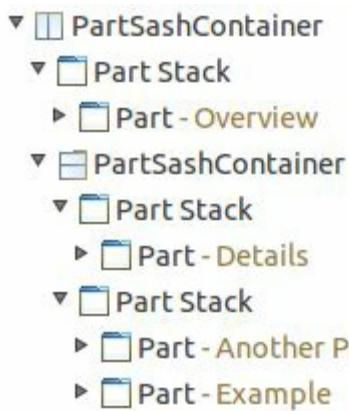
A part stack contains a stack of parts showing the content of one part while displaying only the headers of the other parts. One part is active and the user can switch to another part by selecting the corresponding tab.

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

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



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

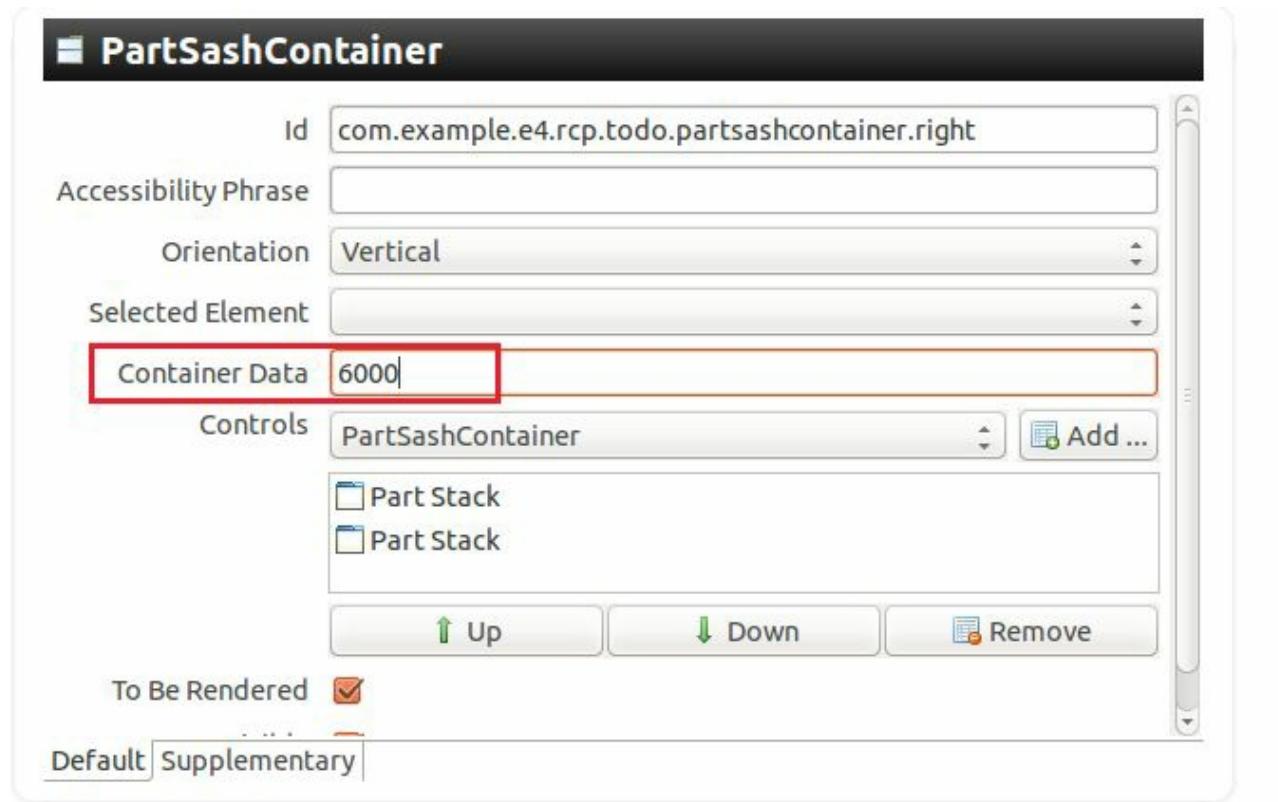


Using layout weight data for children elements

You can use the *Container Data* attribute on a child of a part sash container to assign a layout weight.

This layout weight is interpreted as the relative space the corresponding child element should get assigned in the part sash container.

The setting is depicted in the following screenshot.



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

and these elements take up all available space.

Tip

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

24.4. Perspective

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

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

Chapter 25. Connecting model elements to classes and resources

25.1. URI patterns or classes and resources

Model elements can point to a class or to a static resource via a Uniform Resource Identifier (URI). For this purpose Eclipse defines two URI patterns. The following table describes these patterns. The example assumes that the bundle is called *test* to have a short name.

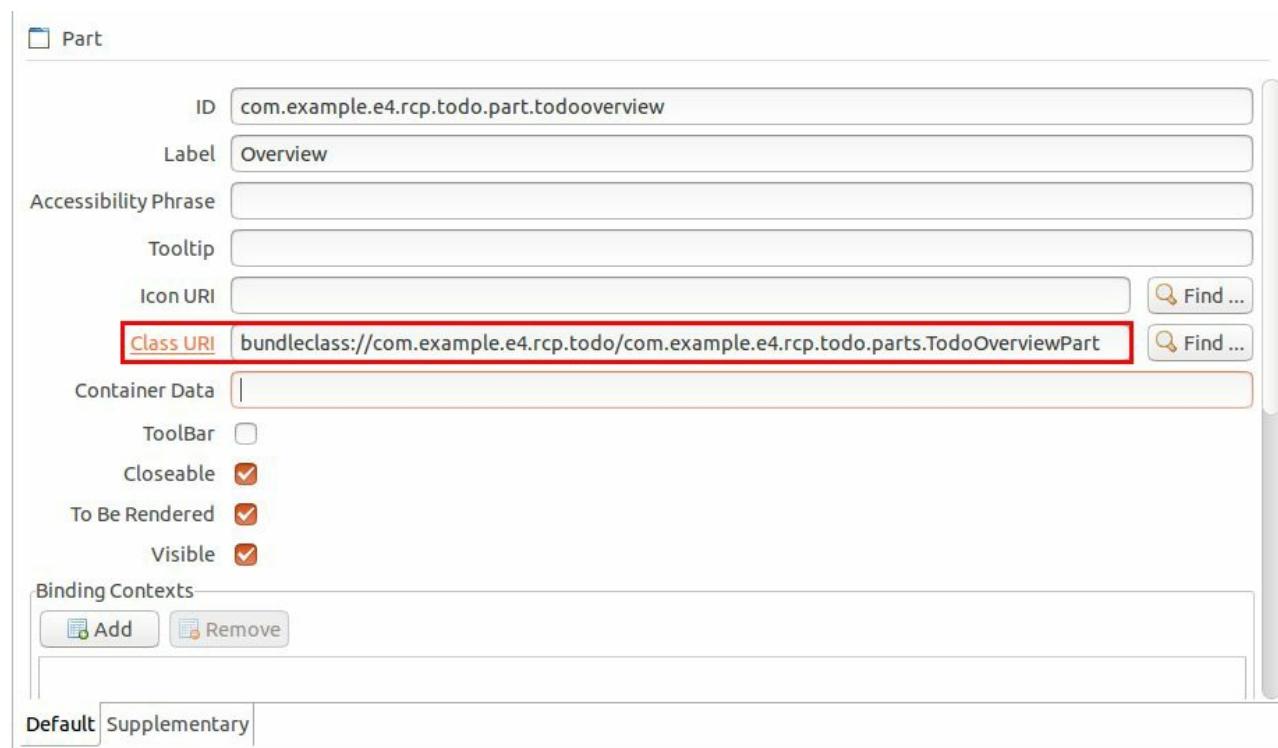
Table 25.1. URI pattern

Pattern	Description
<i>bundleclass:// BSN/package.classname</i>	Identifier for a Java class.
Example: <i>bundleclass://test/test.parts.MySavePart</i>	It consists of the following parts: <i>bundleclass://</i> is a fixed prefix, BSN stands for Bundle-SymbolicName and is defined in a configuration file called <i>MANIFEST.MF</i> file. The Bundle-SymbolicName is followed by a '/' and the fully qualified classname.
<i>platform:/plugin/ BSN/path/filename.extension</i>	Used to identify resources.
Example: <i>platform:/plugin/test/icons/save_edit.gif</i>	Identifier for a resource in a plug-in. <i>platform:/plugin/</i> is a fixed prefix, followed by the BSN stands for Bundle-SymbolicName as defined in the <i>MANIFEST.MF</i> file, followed by the path to the file and the filename.

25.2. Connect model elements to classes

Several application model elements, e.g., a part, have a *Class URI* attribute which points to a Java class via the `bundleclass://` URI. This class provides the behavior of the part. The corresponding object is created by the Eclipse framework.

Using the house/rooms metaphor from earlier, the class is responsible for defining the furnishings, the layout of the room and how the interactive objects behave.



Eclipse instantiates the objects of the referred classes lazily (for most of the model elements). This means for example that the classes for a part are instantiated when the part gets visible.

25.3. Connect model elements to resources like icons

Several model elements can point to static resources using the `platform:/plugin/` URI. For example, the part model element contains the attribute *Icon URI* which can point to an icon that is used for this part.

Chapter 26. Model objects and the runtime application model

26.1. Model objects

During startup the Eclipse framework parses the available information about the application model (`Application.e4xmi`, persisted user changes and model contributions) and stores this information in Java objects. These objects are called *model objects* and at runtime they represent the attributes from the model elements.

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

Table 26.1. Eclipse model elements

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

26.2. Runtime application model

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

Chapter 27. Using the model spy

27.1. Analyzing the application model with the model spy

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

To analyzing this application model and for testing model changes, you can use a test tool from the e4 tools project which allows modifying the application model interactively. This tool is called *Model Spy* (used to be called: Live model editor) and can be integrated into your RCP application. Most changes are directly applied, e.g., if you change the orientation of a part sash container, your user interface is updated automatically.

In the model spy, you can select a part in the application model, right click on it and select *Show Control* to get the part highlighted.

The usage of the model spy requires the installation of the Eclipse 4 - Core Tools as described in [Section 9.2, “Install the e4 tools from the vogella GmbH update site”](#).

27.2. Installation requirements

27.3. Model spy and the Eclipse IDE

If installed you can also use the model spy to see the application model of the running Eclipse IDE itself. You can open it via the **Alt+Shift+F9** shortcut.

Warning

If you modify the Eclipse IDE model, you should be careful as this might put the running Eclipse IDE into a bad state. To fix such issues, start the Eclipse IDE from the command line with the `-clearPersistedState` parameter.

Chapter 28. Exercise: Create an Eclipse plug-in

28.1. Target

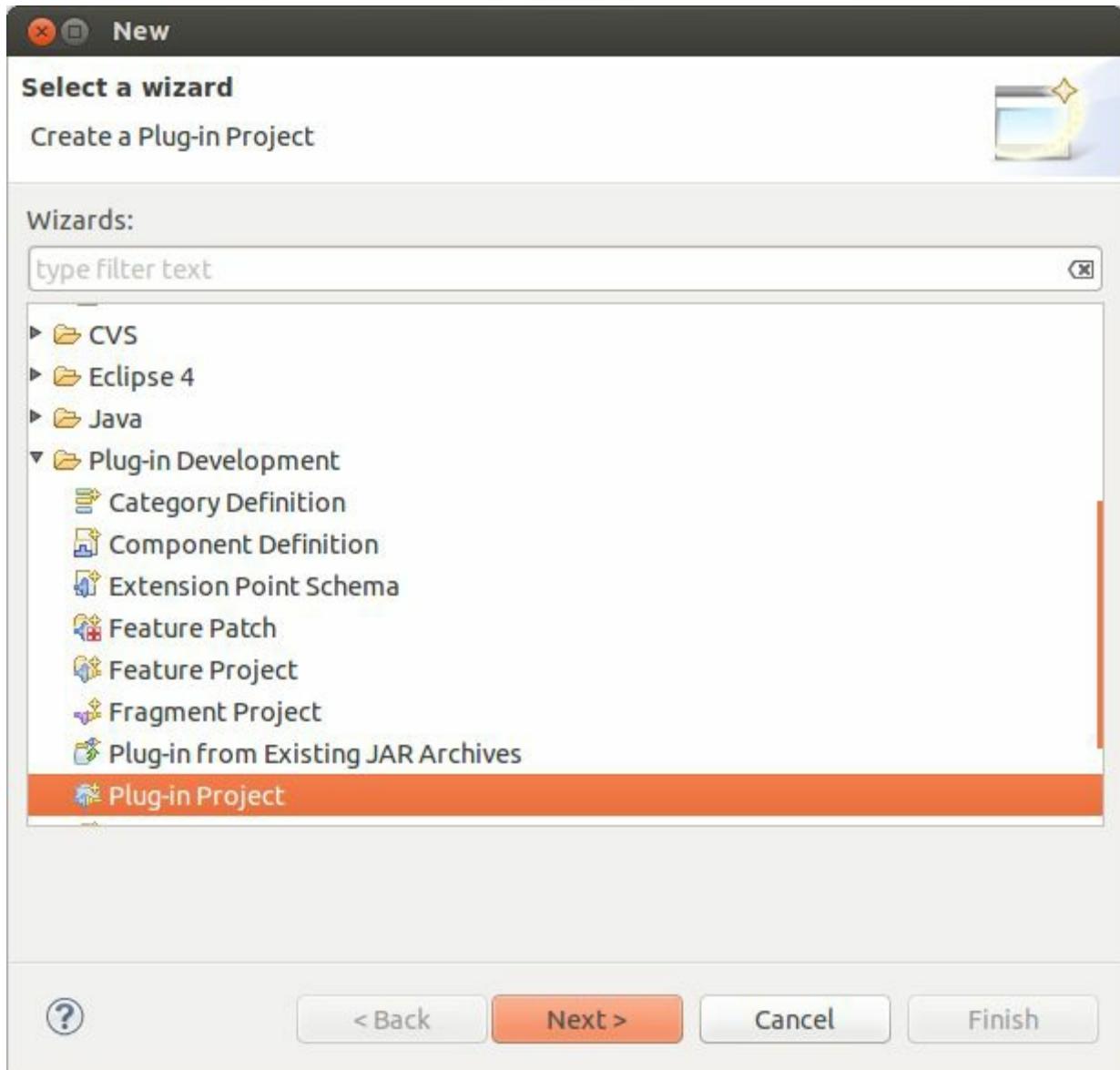
In the following exercise you create a standard Eclipse plug-in. This plug-in is later converted into an Eclipse RCP application.

Note

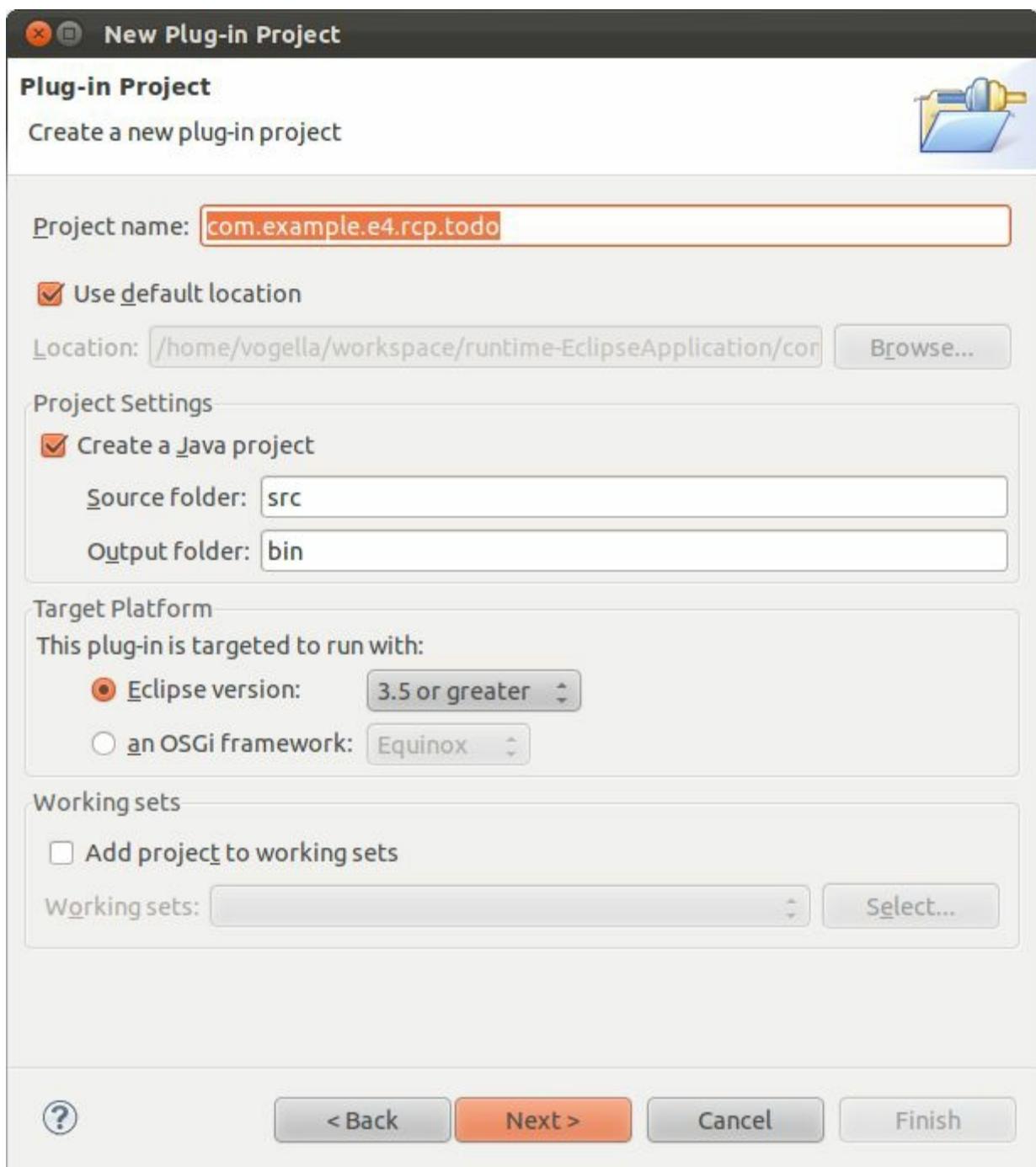
The following description calls this plug-in the *application plug-in* as this plug-in will contain the main application logic.

28.2. Creating a plug-in project

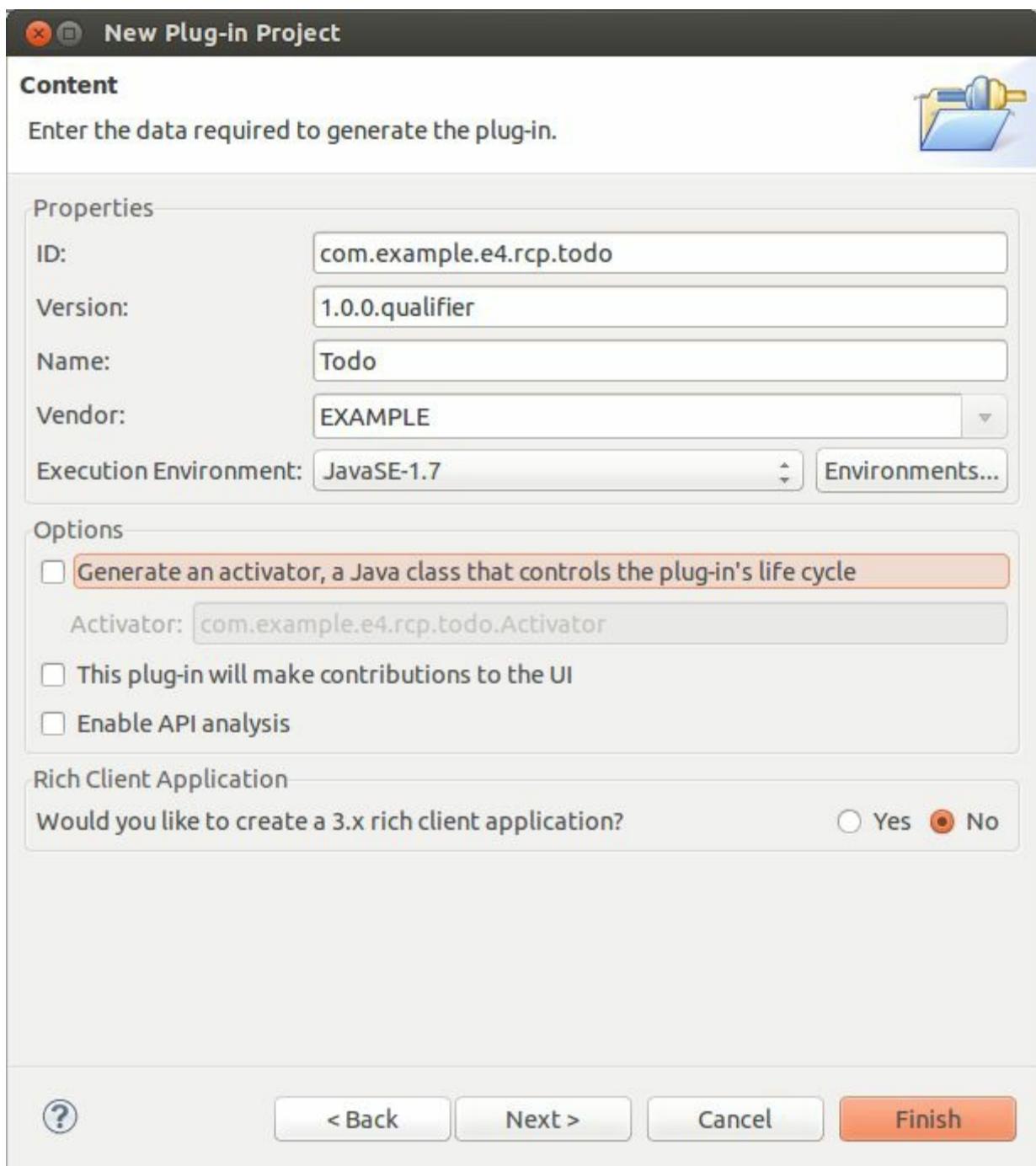
In Eclipse select File → New → Other... → Plug-in Development → Plug-in Project.



Give your plug-in the name `com.example.e4.rcp.todo` and press the *Next* button.



On the next wizard page make the following settings. Select *No* at the *Would you like to create a 3.x rich client application?* option and uncheck *This plug-in will make contributions to the UI*. Uncheck the *Generate an activator, a Java class that controls the plug-in's life cycle* option.



Warning

The *Would you like to create a 3.x 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 these options if you want to use the Eclipse 4 API.

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

28.3. Validate the result

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

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

In manifest editor switch to the *Dependencies* tab and ensure that here the list is also empty.

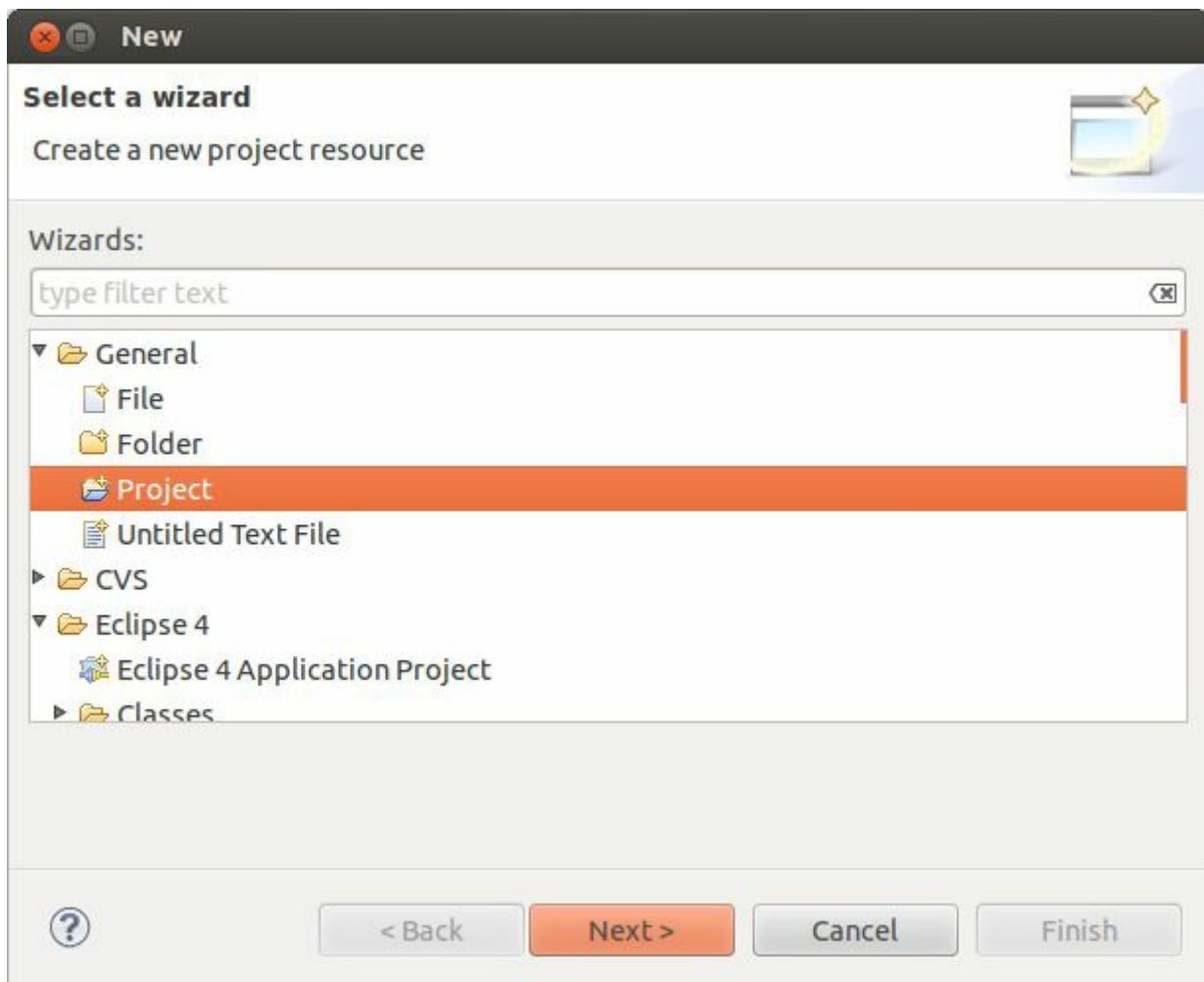
Chapter 29. Exercise: From plug-in to Eclipse RCP

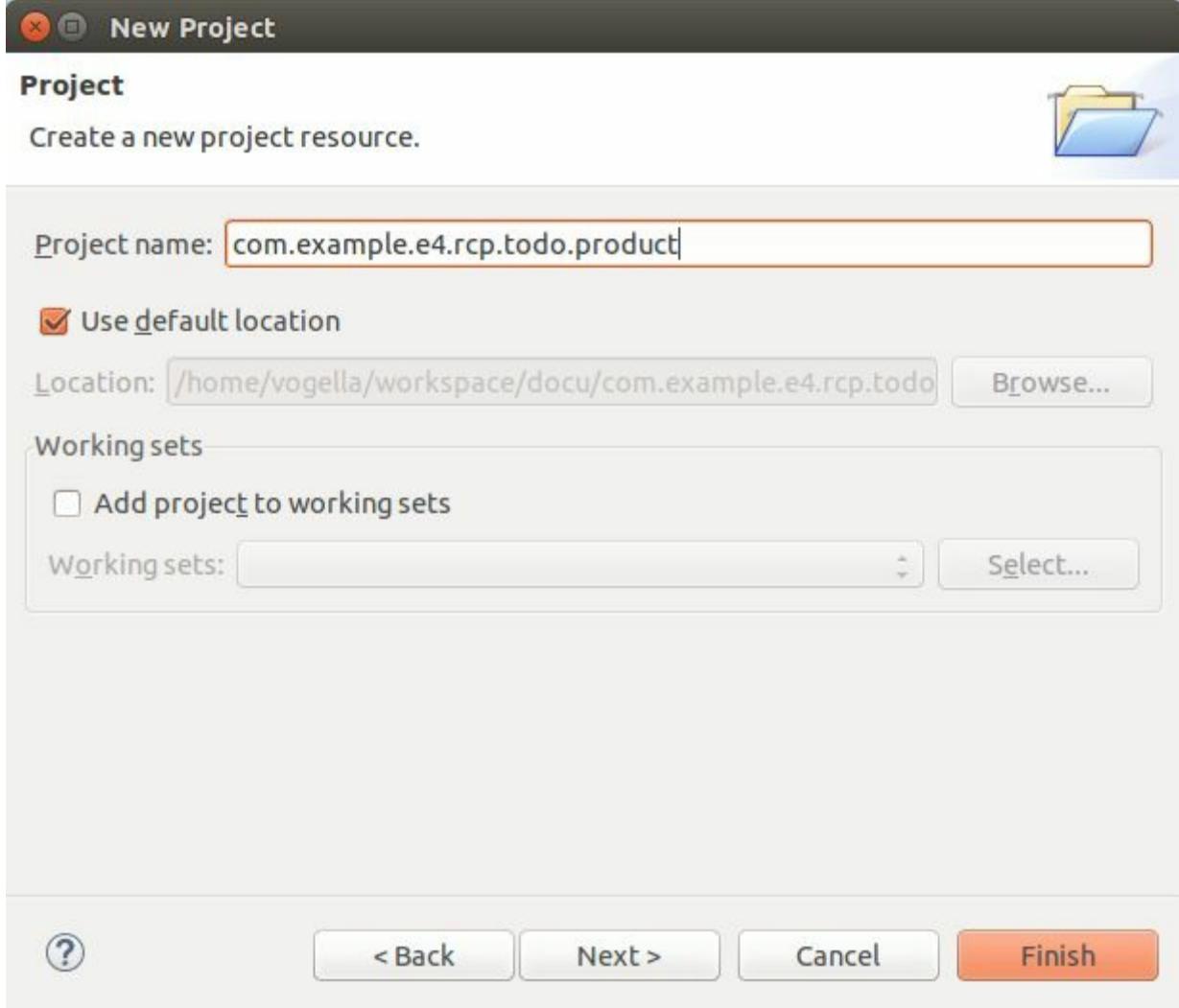
29.1. Target

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

29.2. Create a project to host the product configuration file

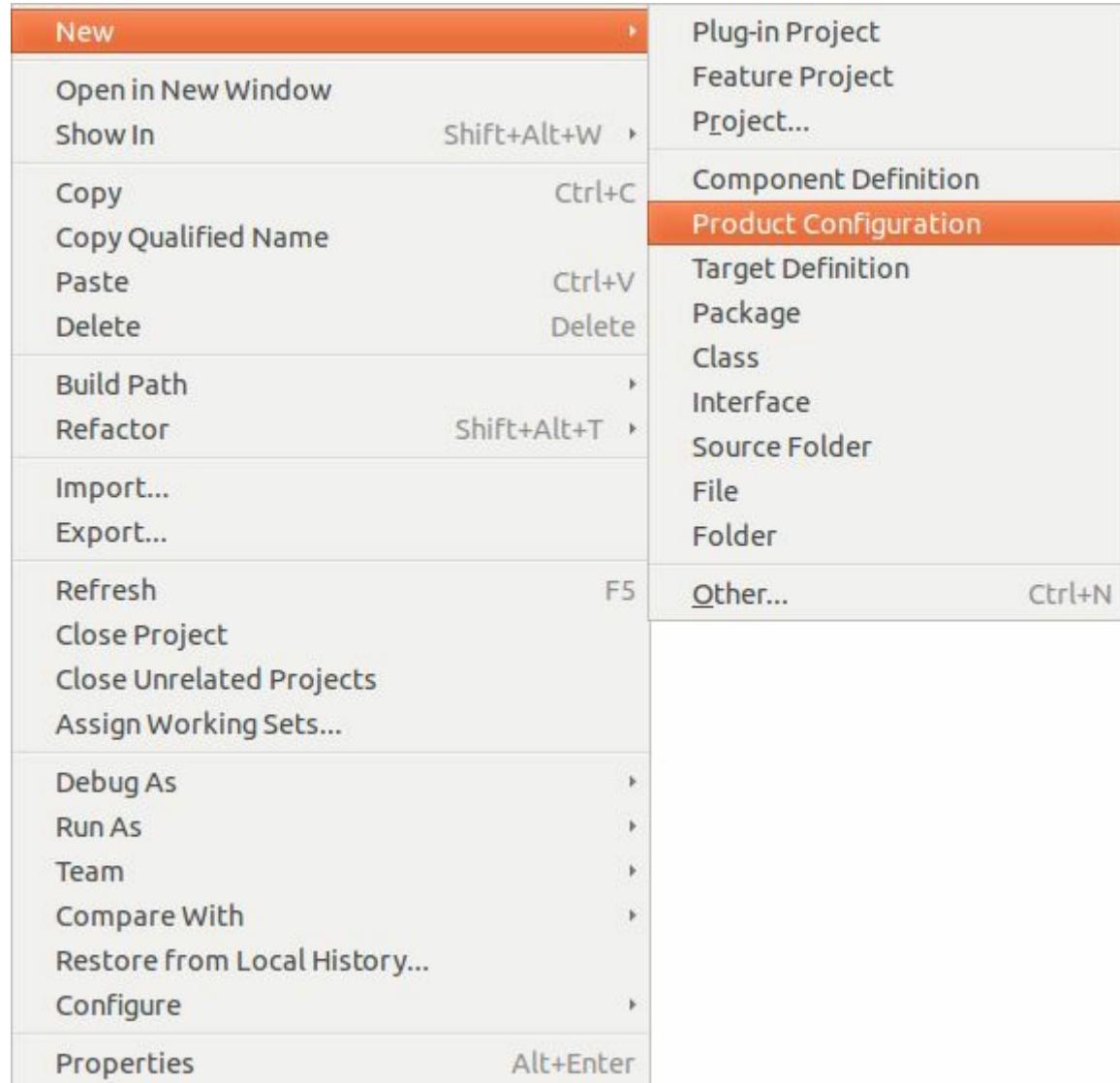
Create a new project called *com.example.e4.rcp.todo.product* via File → New → Others... → General → Project.



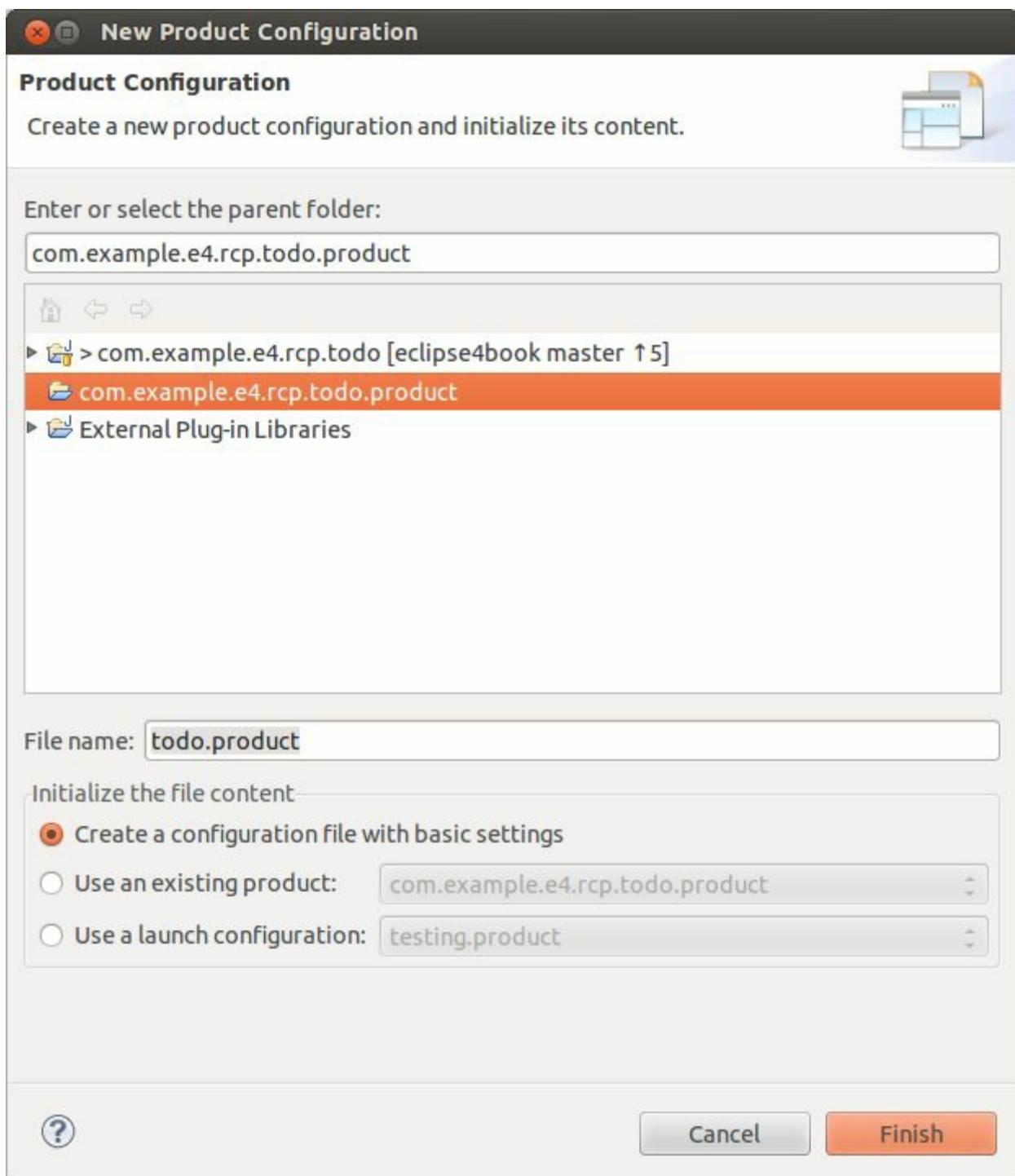


29.3. Create a product configuration file

Right-click on this project and select New → Product Configuration.



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



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

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

General Information
This section describes general information about the product.

ID:

Version:

Name:

The product includes native launcher artifacts

Product Definition
This section describes the launching product extension identifier and application.

Product:

Application:

The [product configuration](#) is based on: plug-ins features

Testing

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

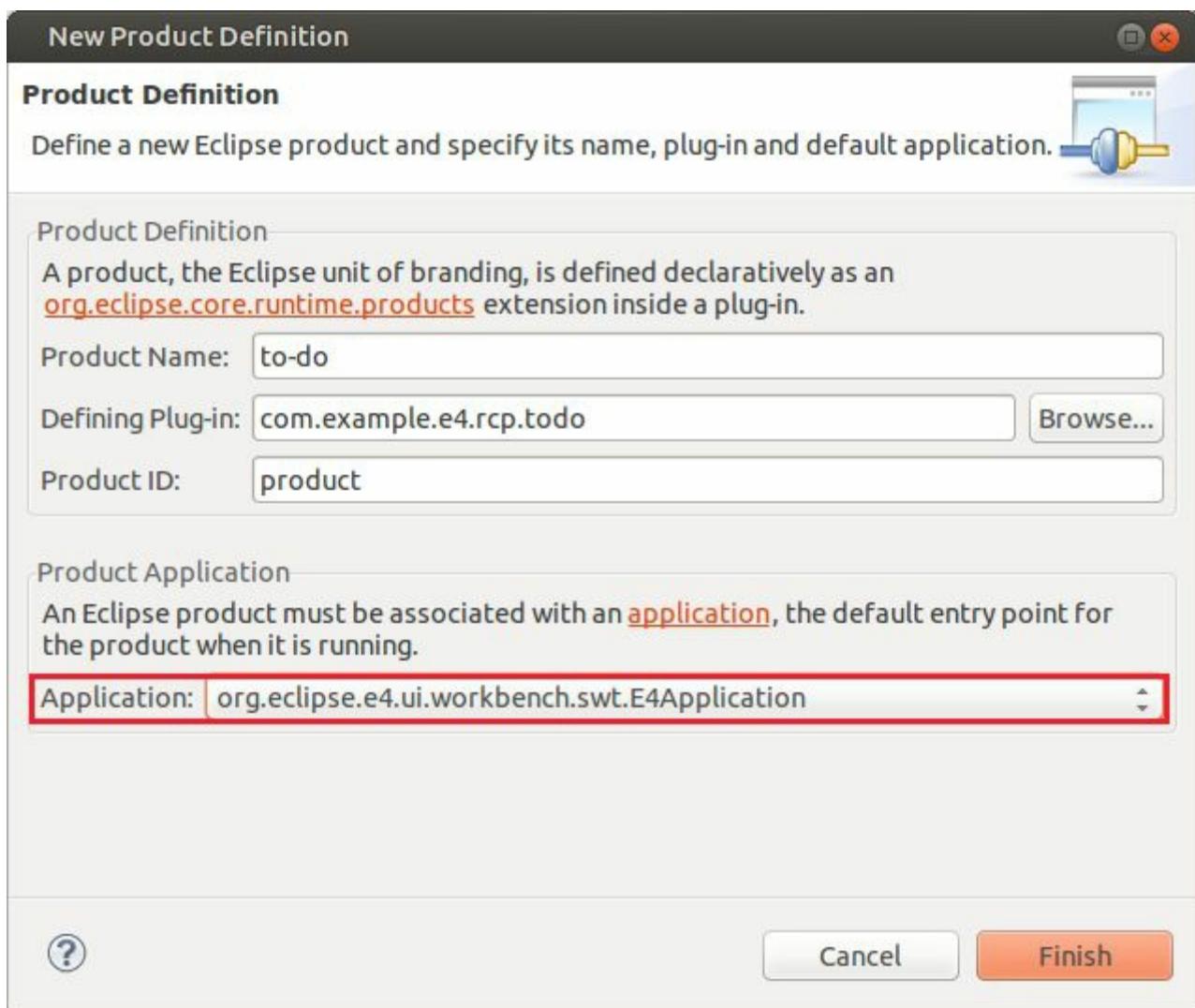
[Launch an Eclipse application](#)

Exporting

Use the [Eclipse Product export wizard](#) to package and export the product defined in this configuration.

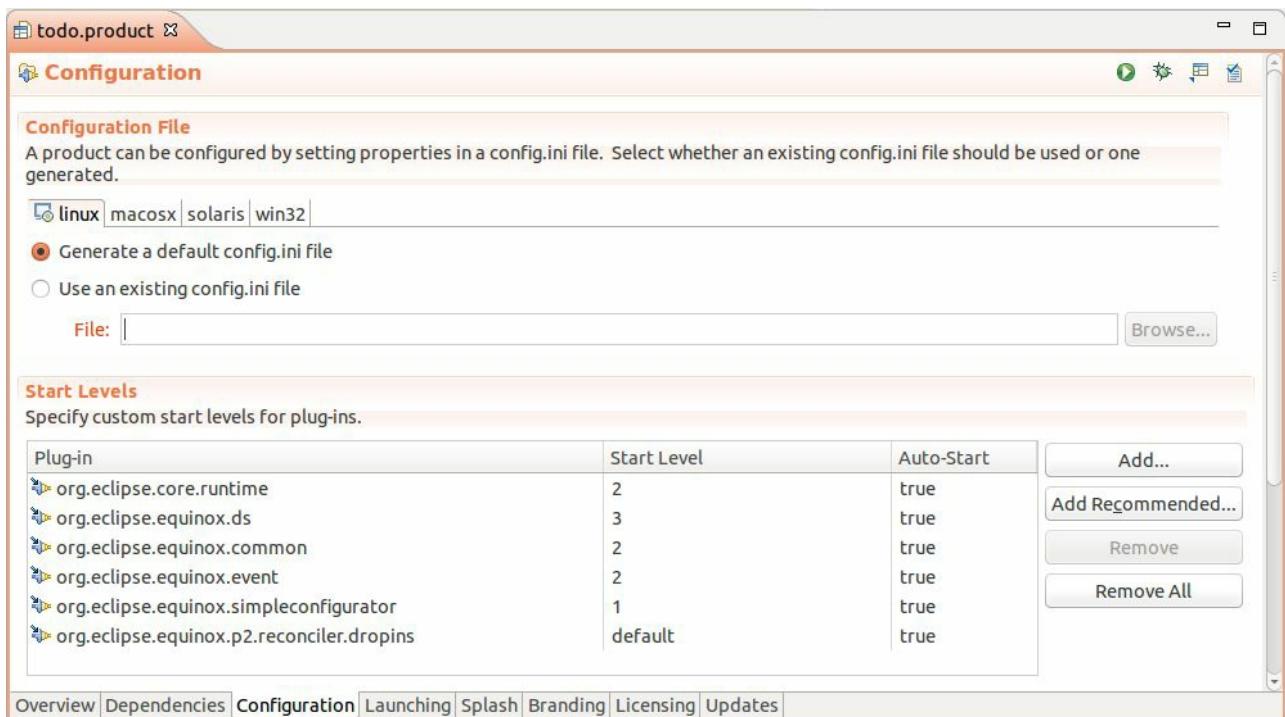
To export the product to multiple platforms: [Overview](#) | [Dependencies](#) | [Configuration](#) | [Launching](#) | [Splash](#) | [Branding](#) | [Licensing](#)

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



29.4. Configure the start levels

Switch to the *Configuration* tab in the product editor and press the *Add Recommended...* button.

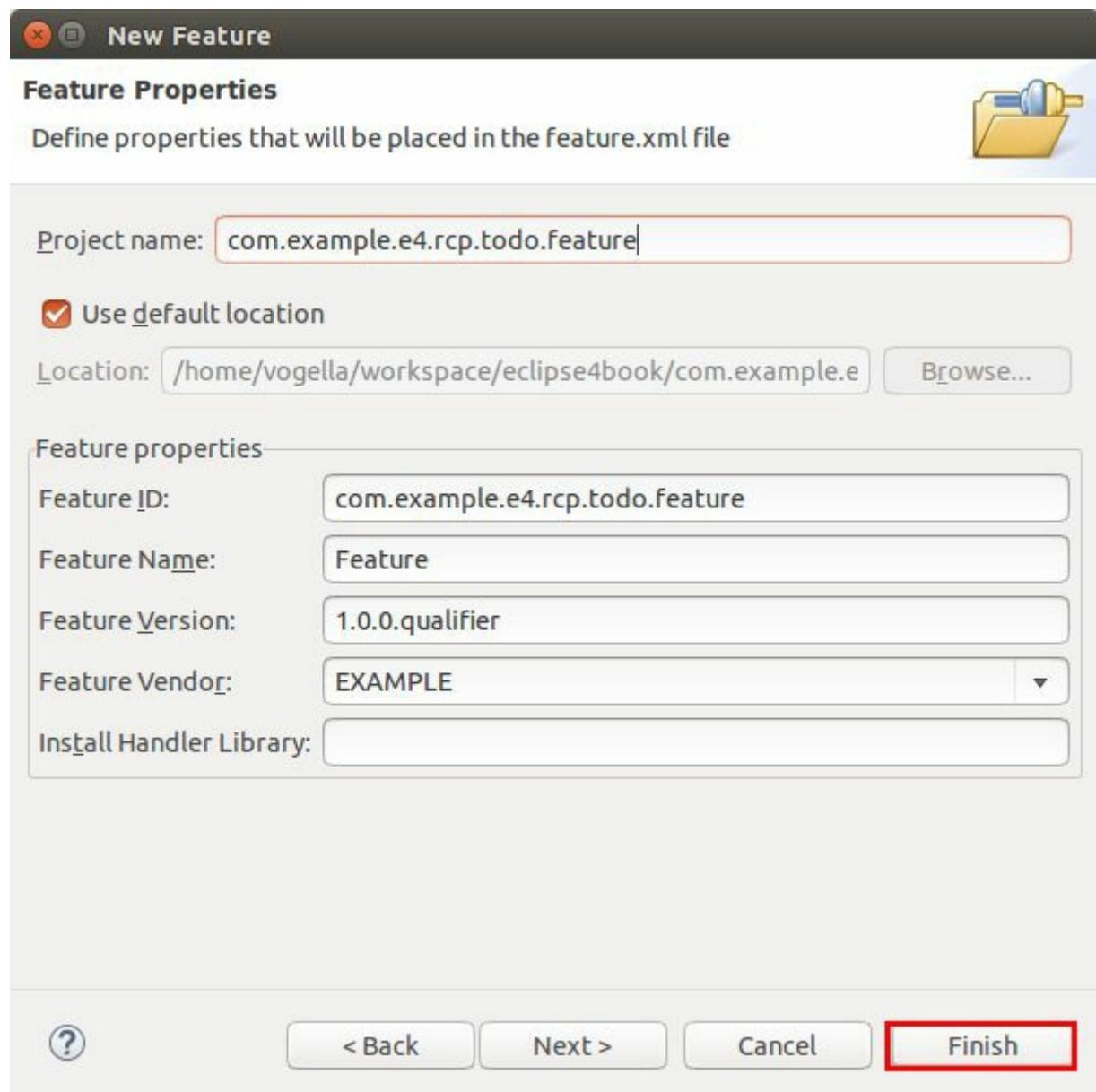


This step is not necessary if you develop, start and export your product via the Eclipse IDE. But command line build systems like Maven/Tycho requires that you set the start levels explicitly hence it is good practice to configure them.

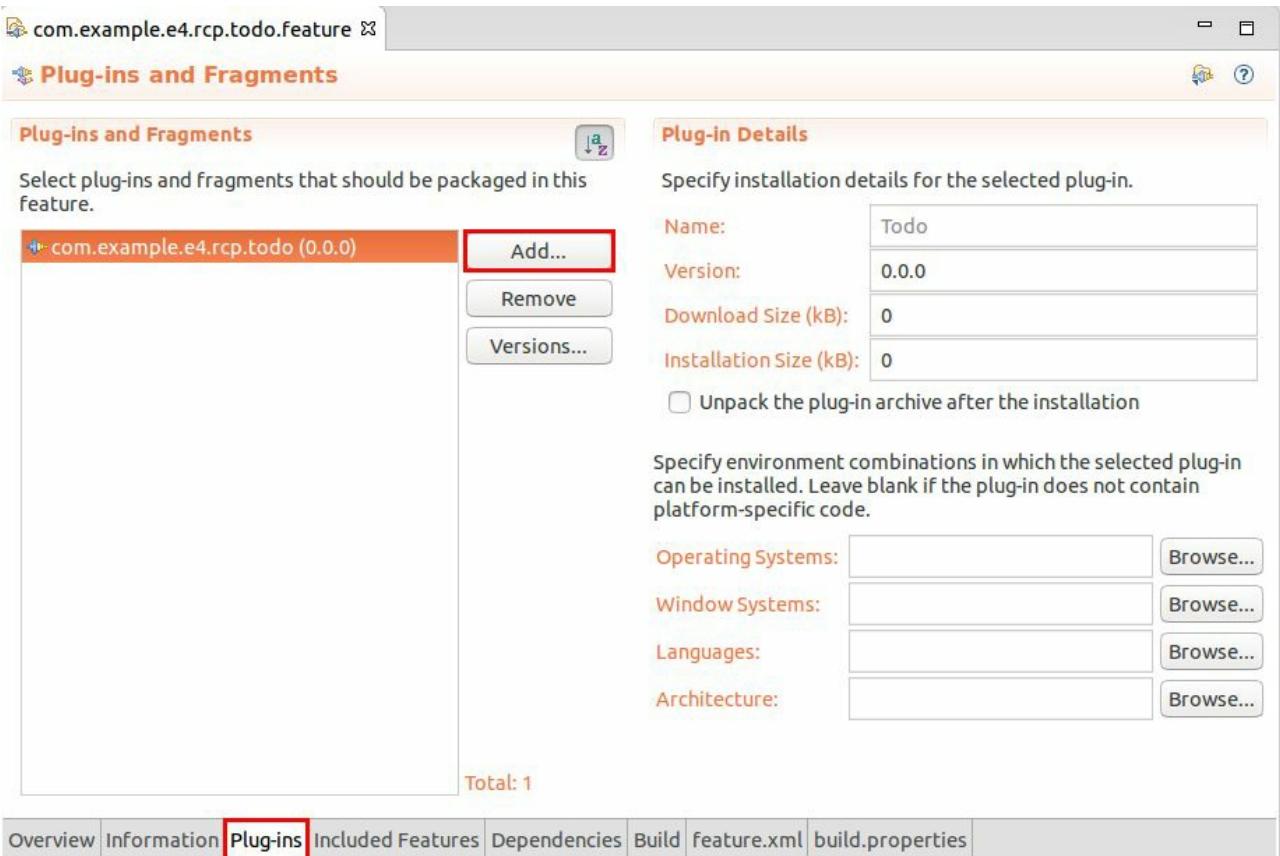
29.5. Create a feature project

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

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



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



Warning

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

29.6. Enter the feature dependencies in product

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

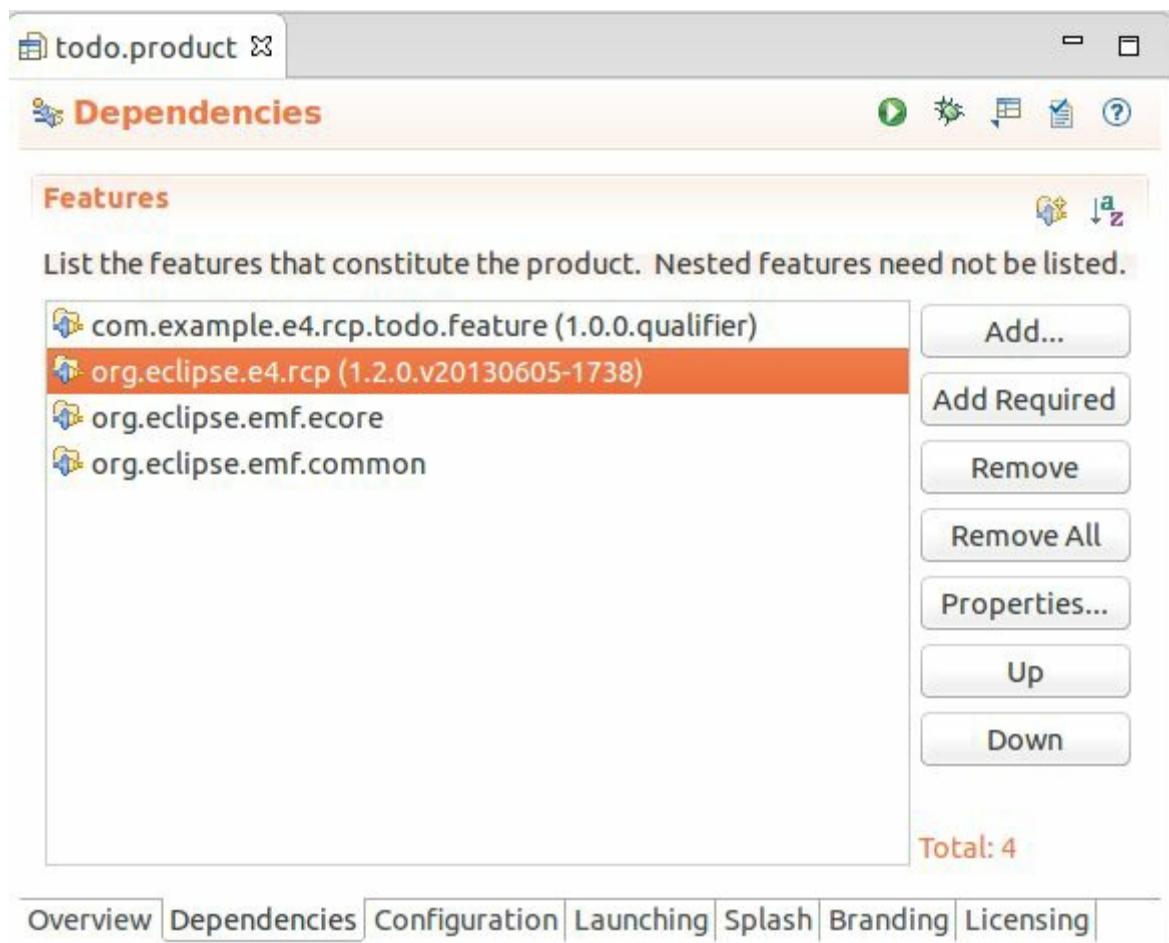
For this select the *features* option on the *Overview* tab of the product editor.

The screenshot shows the Eclipse Product Editor interface with the title bar "todo.product". The "Overview" tab is selected. The "Product Definition" section contains fields for "Product" (set to "com.example.e4.rcp.todo.product") and "Application" (set to "org.eclipse.e4.ui.workbench.swt.E4Application"). Below these, a radio button group indicates that the product configuration is based on "Features" (which is selected) rather than "Plug-ins". The "Testing" section lists two steps: "Synchronize" the configuration with the product's defining plug-in and "Test the product by launching a runtime instance of it", followed by links to "Launch an Eclipse application" and "Launch an Eclipse application in Debug mode". The "Exporting" section provides instructions for using the Eclipse Product export wizard to package and export the product. At the bottom, a navigation bar includes tabs for Overview, Dependencies, Configuration, Launching, Splash, Branding, and Licensing, with the "Overview" tab highlighted.

Select the *Dependencies* tab and add the following dependencies via the *Add...* button.

- com.example.e4.rcp.todo.feature
- org.eclipse.e4.rcp

- org.eclipse.emf.ecore
- org.eclipse.emf.common

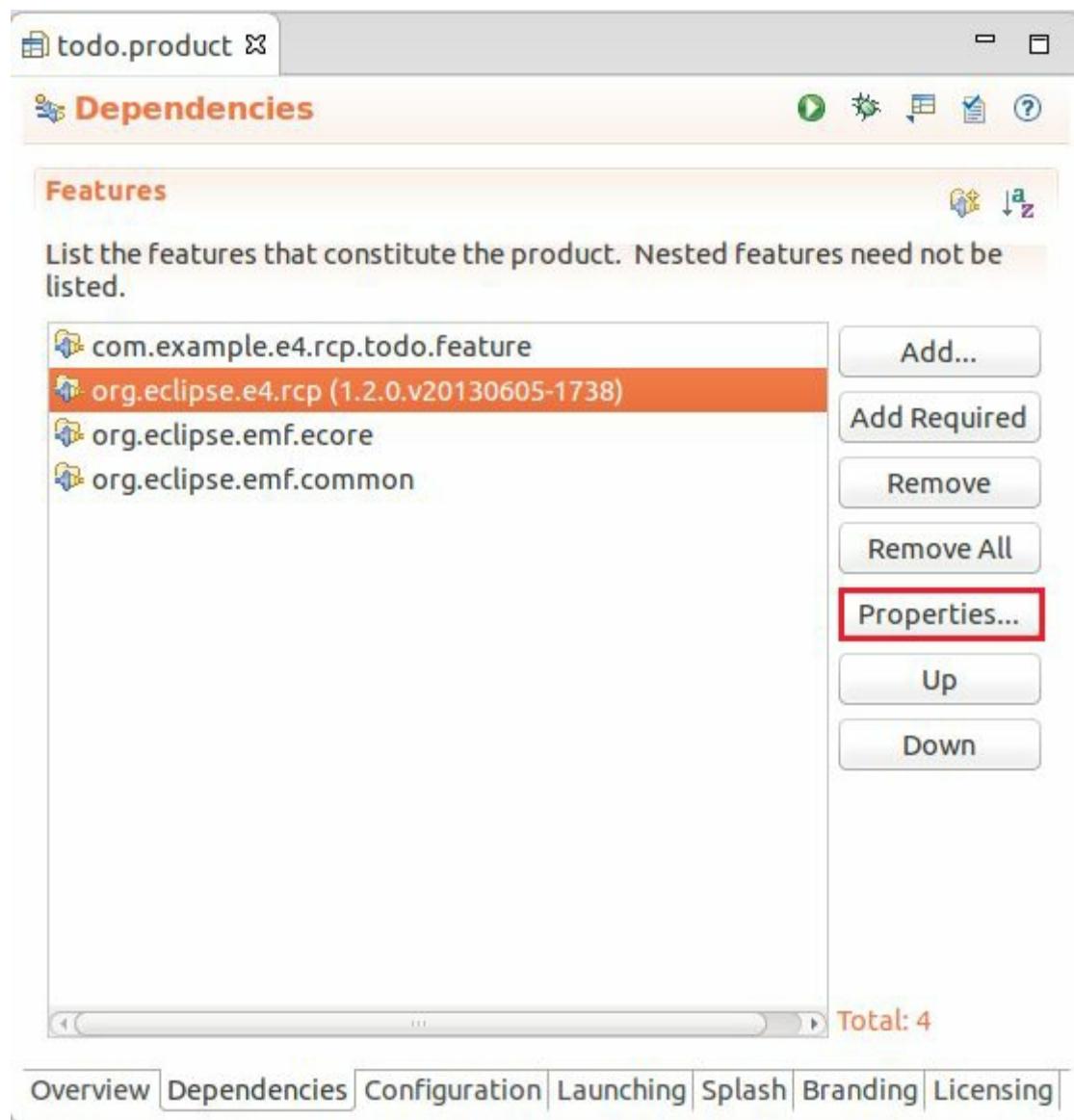


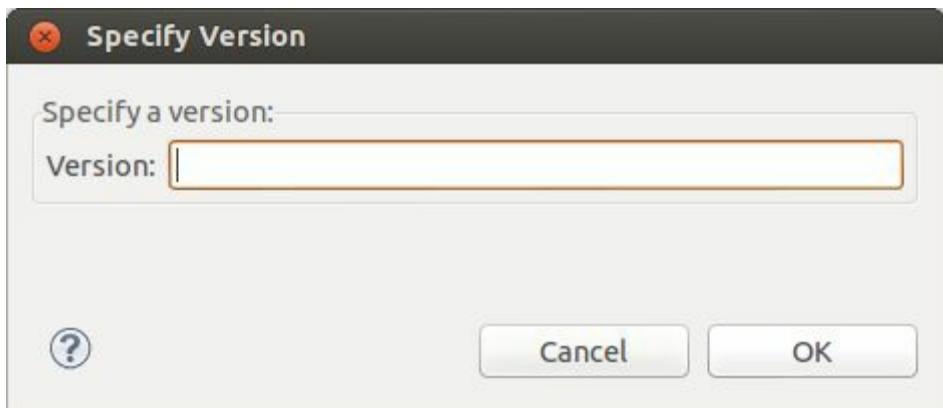
Note

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

29.7. Remove the version dependency from the features in product

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



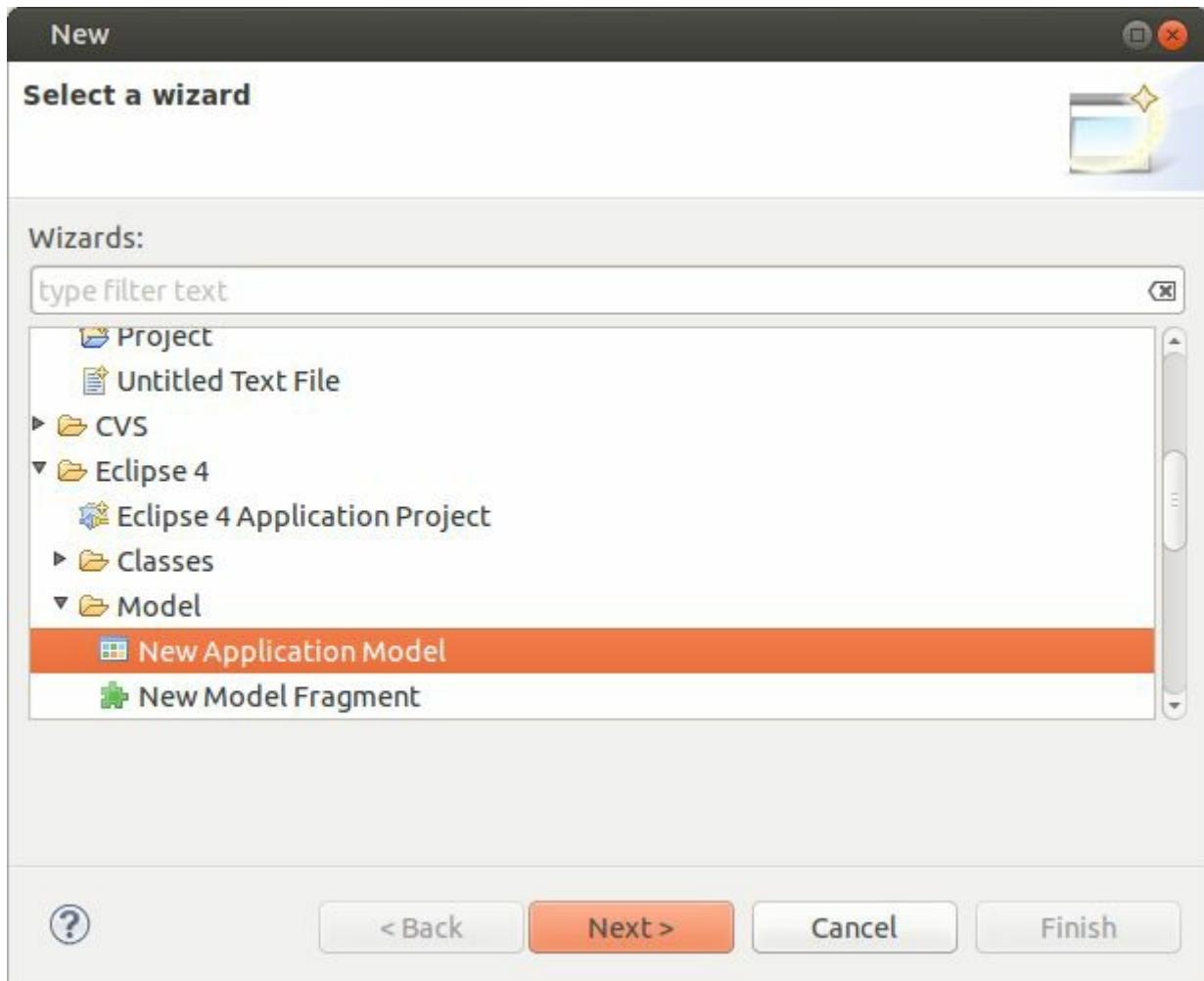


The result should look similar to the following screenshot.

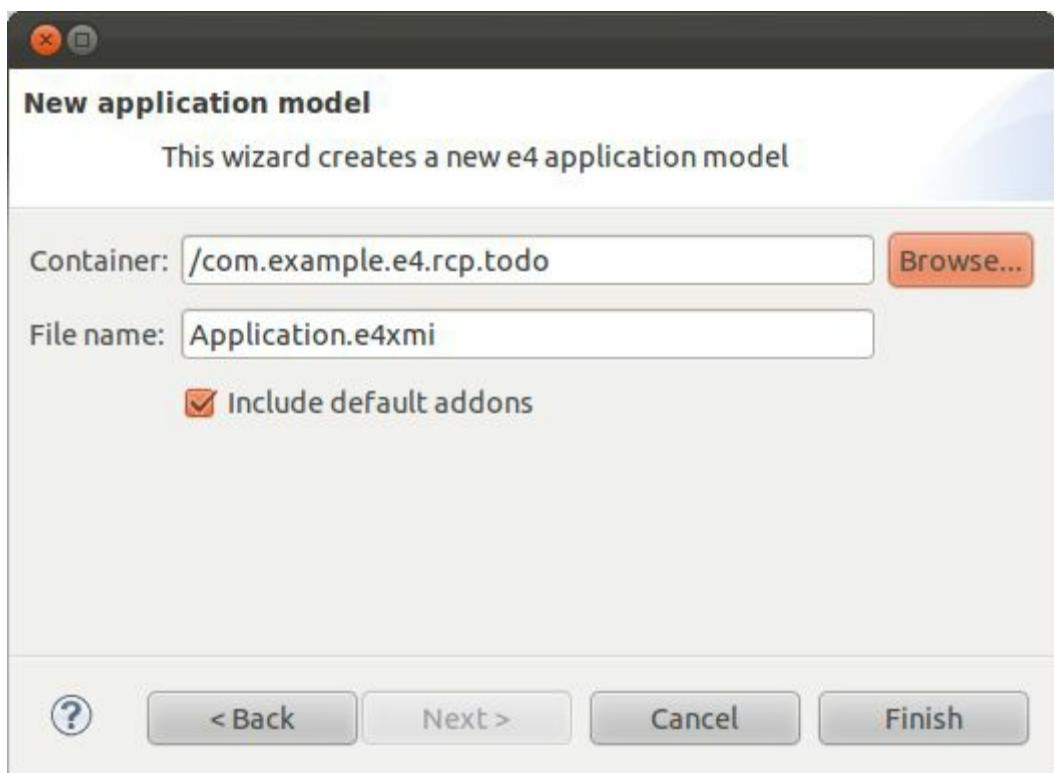
The screenshot shows the "Dependencies" tab of the "todo.product" editor. It lists features: "com.example.e4.rcp.todo.feature", "org.eclipse.e4.rcp", "org.eclipse.emf.ecore", and "org.eclipse.emf.common". To the right of the list are buttons for "Add...", "Add Required", "Remove", "Remove All", "Properties...", "Up", and "Down". A total count of "Total: 4" is displayed at the bottom. Below the main area is a navigation bar with tabs: Dependencies, Configuration, Launching, Splash, Branding, and "»2".

29.8. Create an application model

Select File → New → Other... → Eclipse 4 → Model → New Application Model to open a wizard to create an application model file.



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

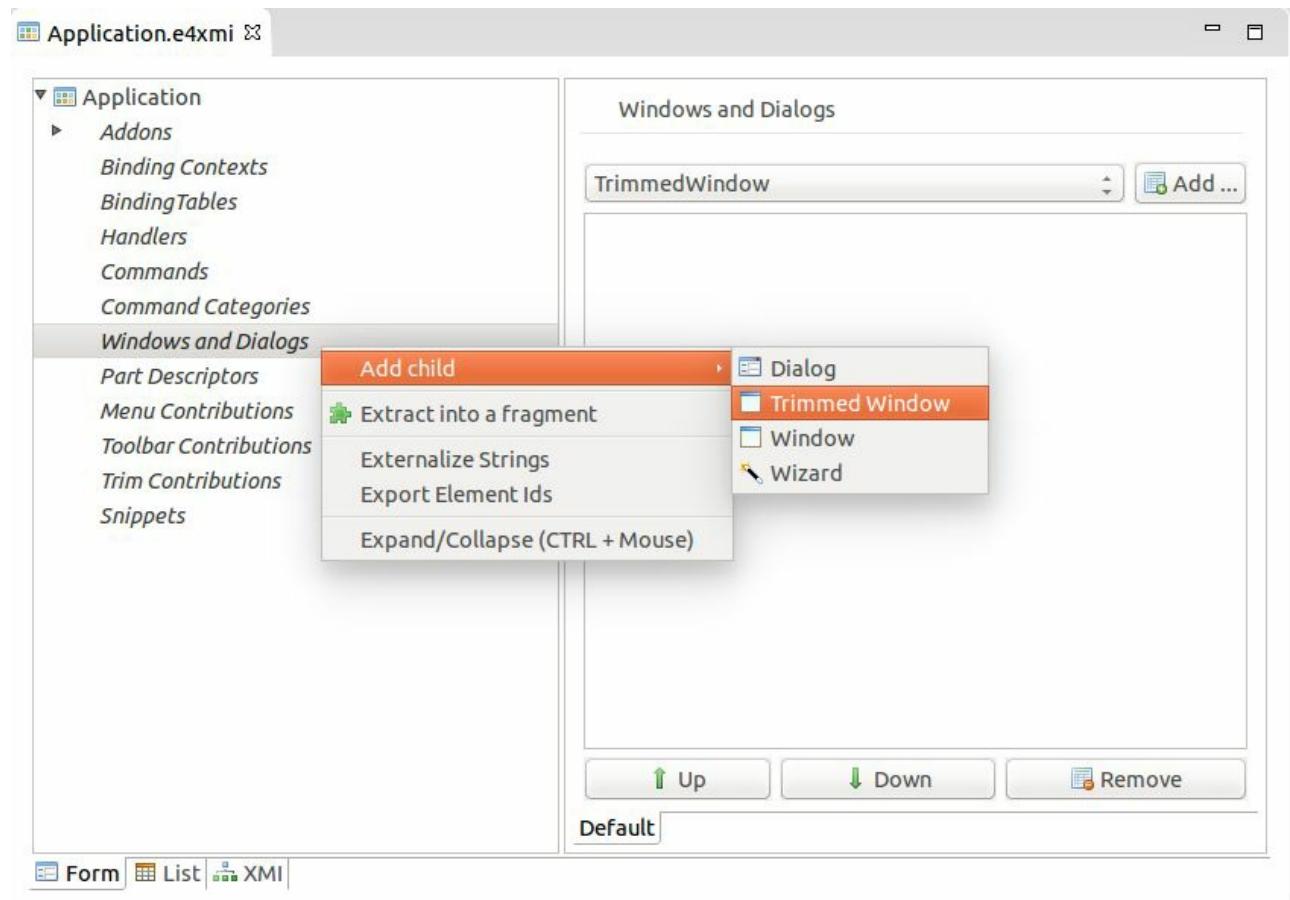


Press the *Finish* button. This triggers the creation of the `Application.e4xmi` file inside the `com.example.e4.rcp.todo` plug-in and opens it.

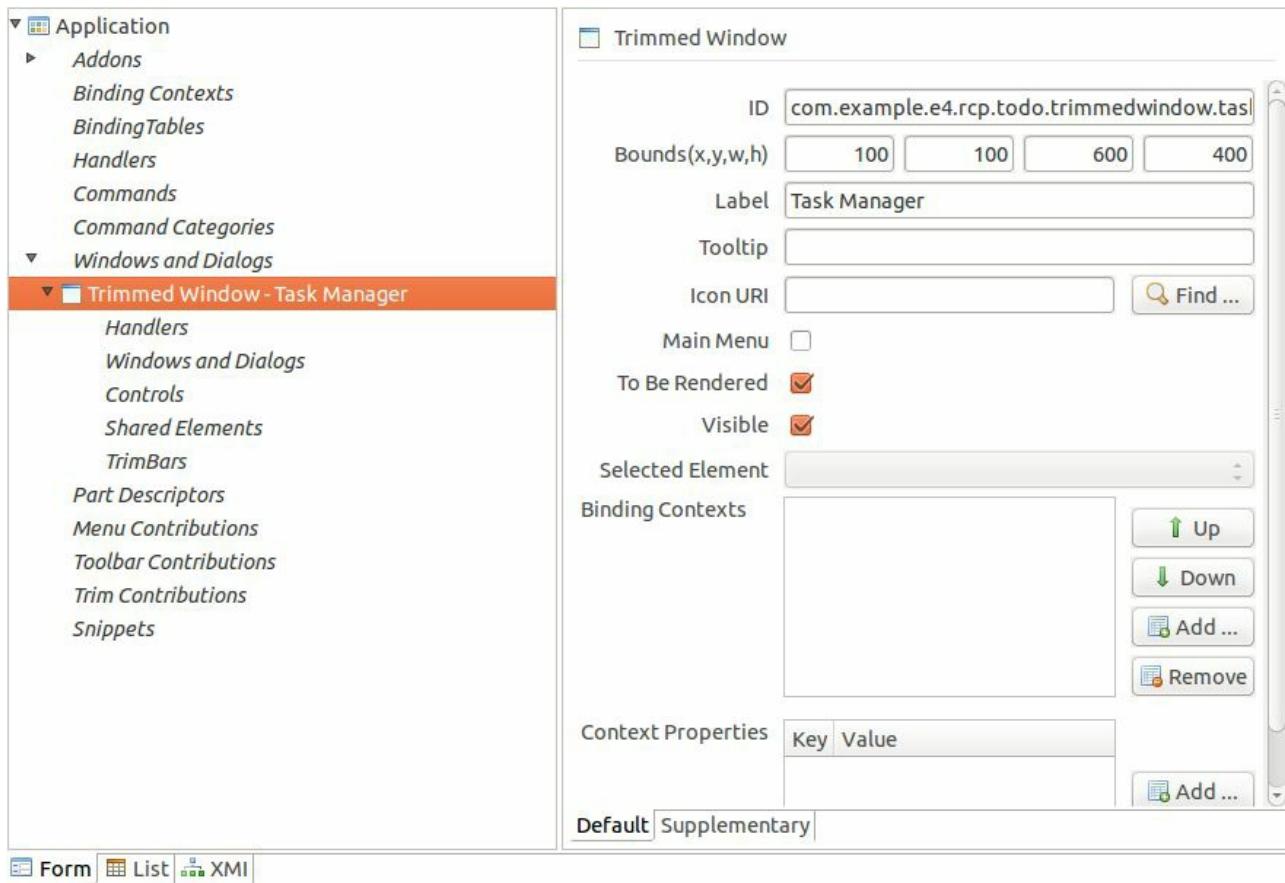
29.9. Add a window to the application model

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

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



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



Tip

If you start and close your application the last state of the application is persisted by the framework and restored the next time you start this application. This is undesired during development, as the latest state from the application model file should be used. In [Section 30.1, “Delete the persisted user changes at startup”](#) you configure your product to remove all the persisted changes from the last run.

29.10. Start the 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](#)

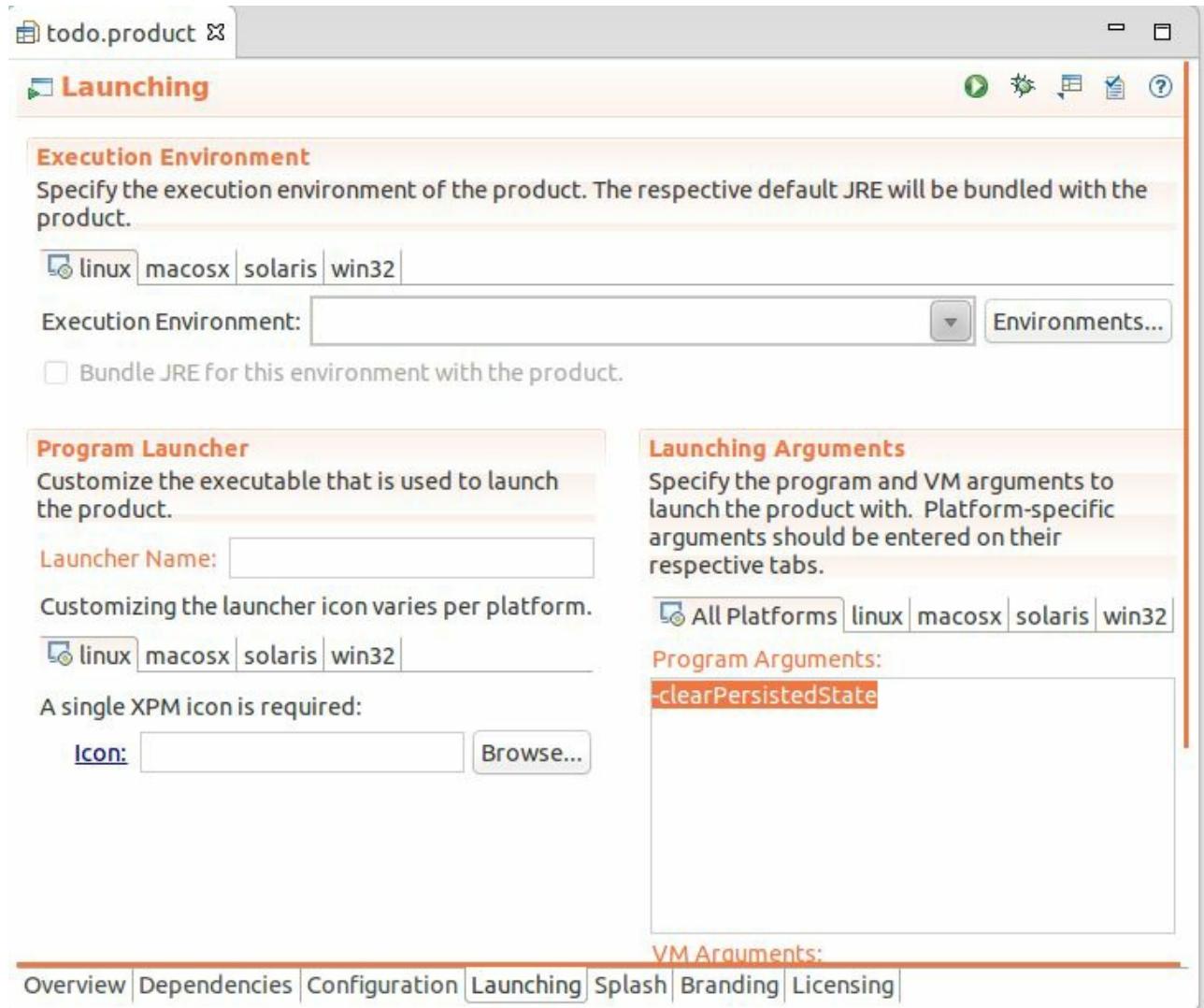
Validate that your application starts. You should see an empty window, which can be moved, resized, minimized, maximized and closed.

Chapter 30. Exercise: Configure the deletion of persisted model data

30.1. Delete the persisted user changes at startup

To ensure that always the latest version of your application model is used, add the `-clearPersistedState` parameter to your product configuration file.

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



Note

Please note that *Program Arguments* must be specified via the `-` sign, e.g., `-clearPersistedState`.

30.2. Why is this setting necessary?

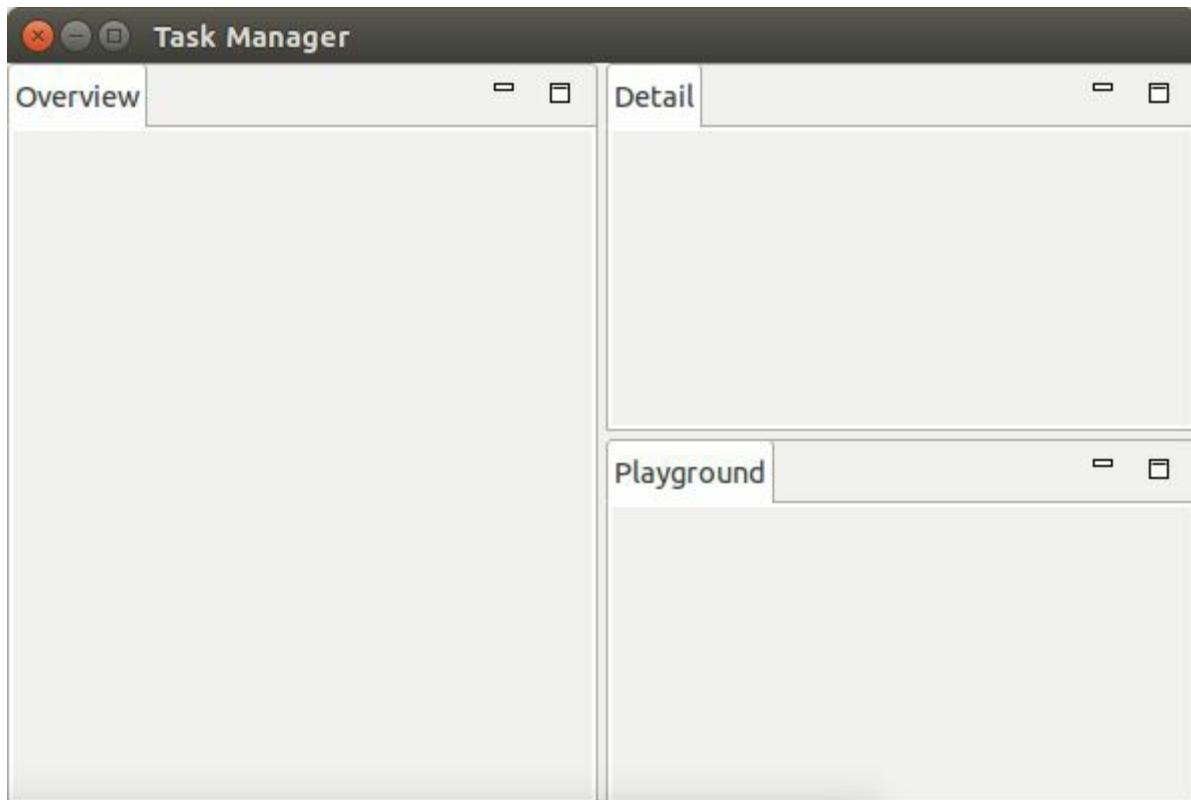
At startup of an Eclipse application the Eclipse platform restores the state in which you left the application before the last shutdown. During development this might lead to situations where changes are not correctly applied and displayed, e.g., you define a new menu entry and this entry is not displayed in your application.

Alternatively to the approach described above, you can also set the *Clear* flag on the *Main* tab in your Run configuration. This would also delete other persisted data, like for example preference values.

Chapter 31. Exercise: Modeling a user interface

31.1. Desired user interface

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



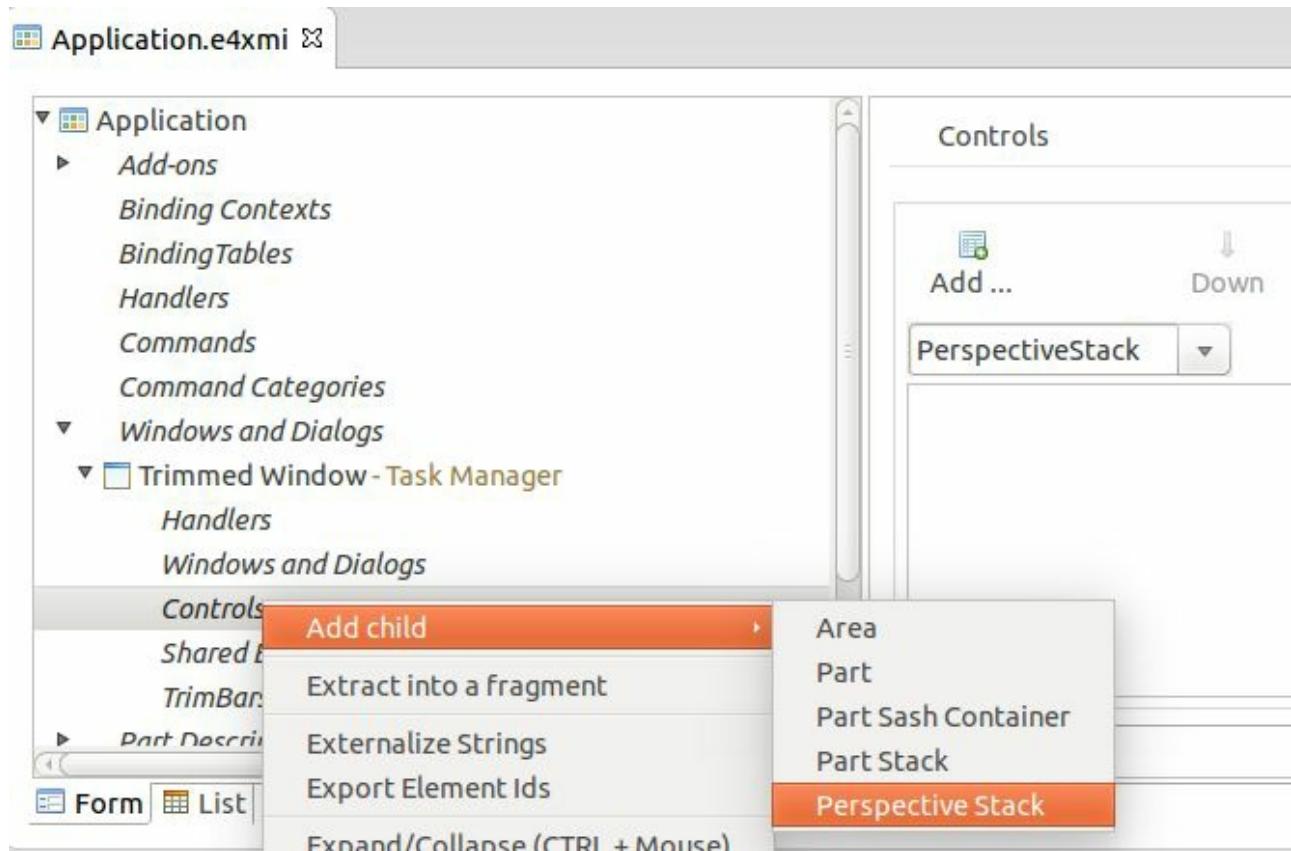
31.2. Open the Application.e4xmi file

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

31.3. Add a perspective

Add a perspective stack with one perspective to your application model so that you can later easily add more of them.

Navigate to your window inside the `Application.e4xmi` file. Select the *Controls* node. Add a *Perspective Stack* via the context menu on the *Controls* entry as indicated in the following screenshot.



Tip

Alternatively to the context menu you can also use the *Add...* button on the detail page to add child elements to the selected element.

After creating the perspective stack add a *Perspective* to it, either via the context menu or the *Add...* button.

▼ Windows and Dialogs
 ▼ Trimmed Window - Task Manager
Handlers
Windows and Dialogs
 ▼ Controls

Perspective Stack

Shared Elements

TrimBars

► Part Descriptors

Menu Contributions

Toolbar Contributions

Add child

Import 3x

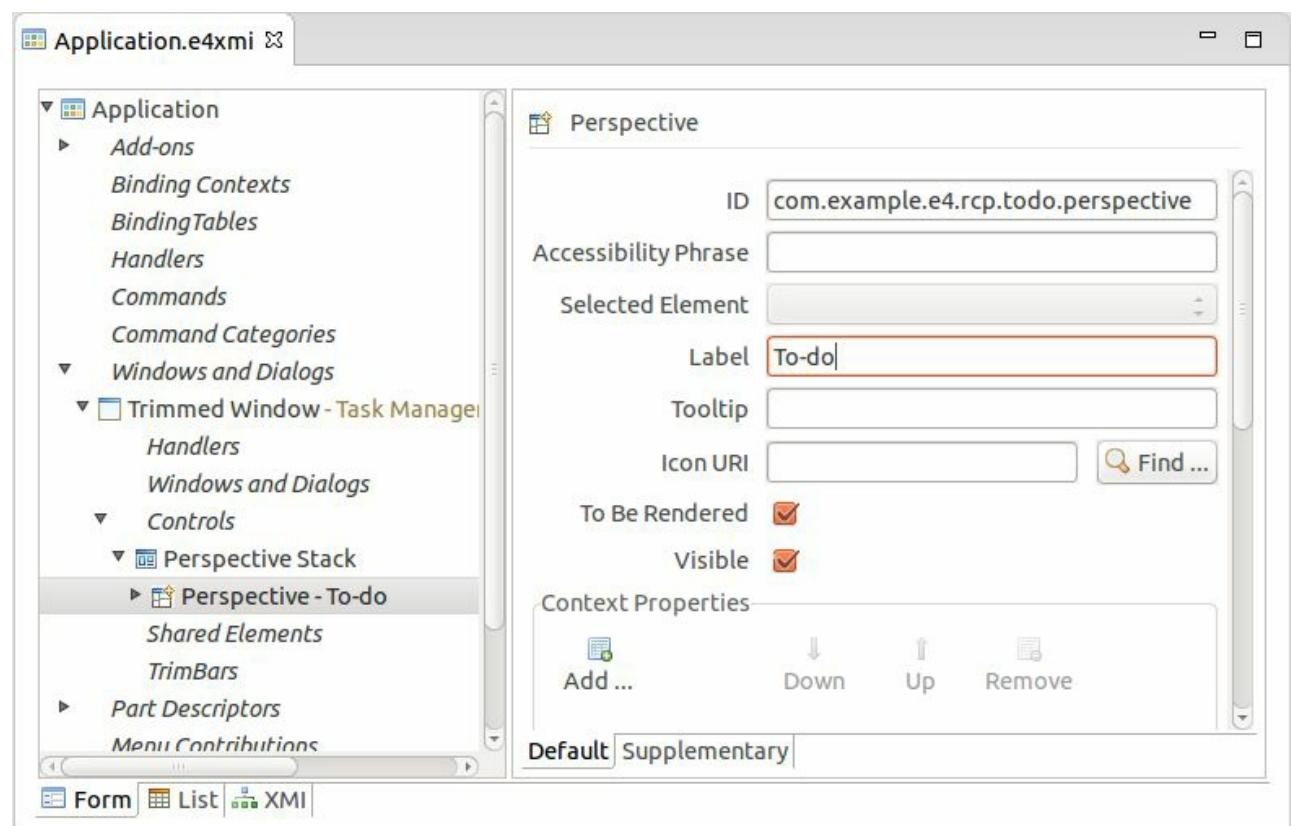
Remove

Extract into a fragment

Execute Script

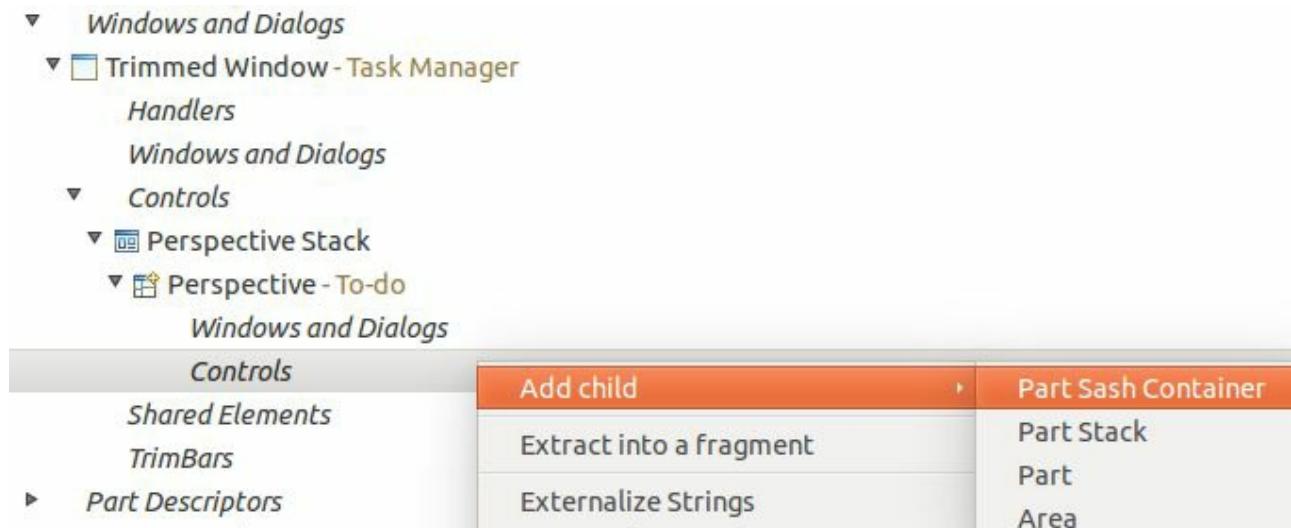
Perspective

Enter the *To-do* value in the *Label* field and the *com.example.e4.rcp.todo.perspective* value in the *ID* field.

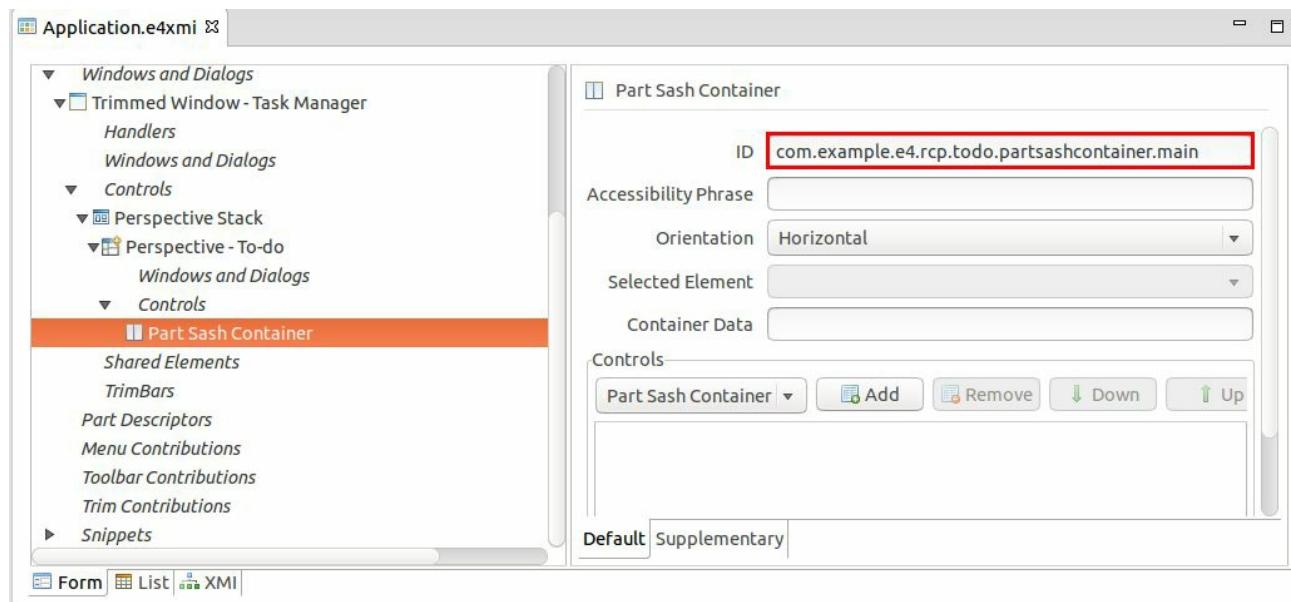


31.4. Add part sash and part stack containers

Select *Controls* below the newly created perspective and add a part sash container element.



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



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

Re-select the parent part sash container and add another part sash container element. Now add two part stacks to the second part sash container.

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

- ▼ *Windows and Dialogs*
 - ▼  Trimmed Window - **Task Manager**
 - Handlers*
 - Windows and Dialogs*
 - ▼ **Controls**
 - ▼  Perspective Stack
 - ▼  Perspective - **To-do**
 - Windows and Dialogs*
 - ▼ **Controls**
 - ▼  Part Sash Container
 -  Part Stack
 - ▼  Part Sash Container
 -  Part Stack
 -  Part Stack

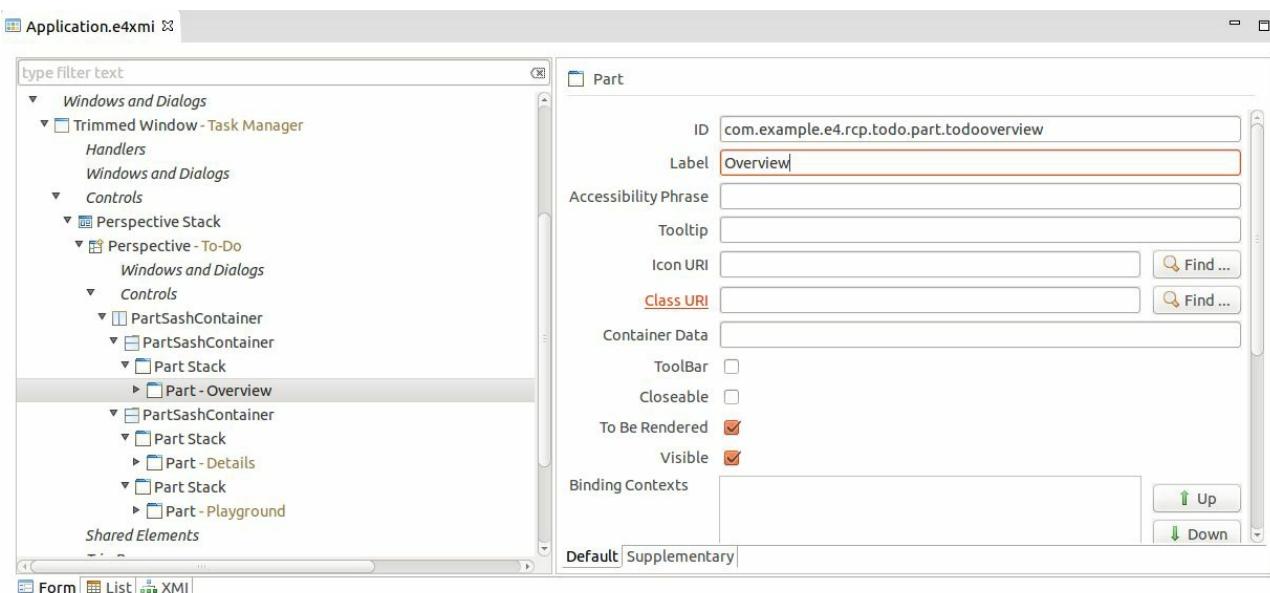
31.5. Create the parts

Add a part model element to each part stack. As ID for the parts use the prefix `com.example.e4.rcp.todo.part` and the suffix from the following table. Also put the label from the table into the appropriate field of the part editor.

Table 31.1. Label and ID from the Parts

ID Suffix	Label
.todooverview	Overview
.tododetails	Details
.playground	Playground

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



31.6. Validate the user interface

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

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

Chapter 32. Exercise: Connect Java classes with the parts

32.1. Create a new package and some Java classes

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

Create three Java classes called `TodoOverviewPart`, `TodoDetailsPart` and `PlaygroundPart` in this package.

Tip

You can create the classes by clicking on the *Class URI* hyperlink in the detail pane of the model editor for the part. This also connects the created class to the model object. If you do this, you can skip [Section 32.2, “Connect the Java classes with your parts”](#) of this exercise.

The following code shows the `TodoDetailsPart` class. All classes should not extend another class, nor do they implement any interface.

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

public class TodoDetailsPart {
```

32.2. Connect the Java classes with your parts

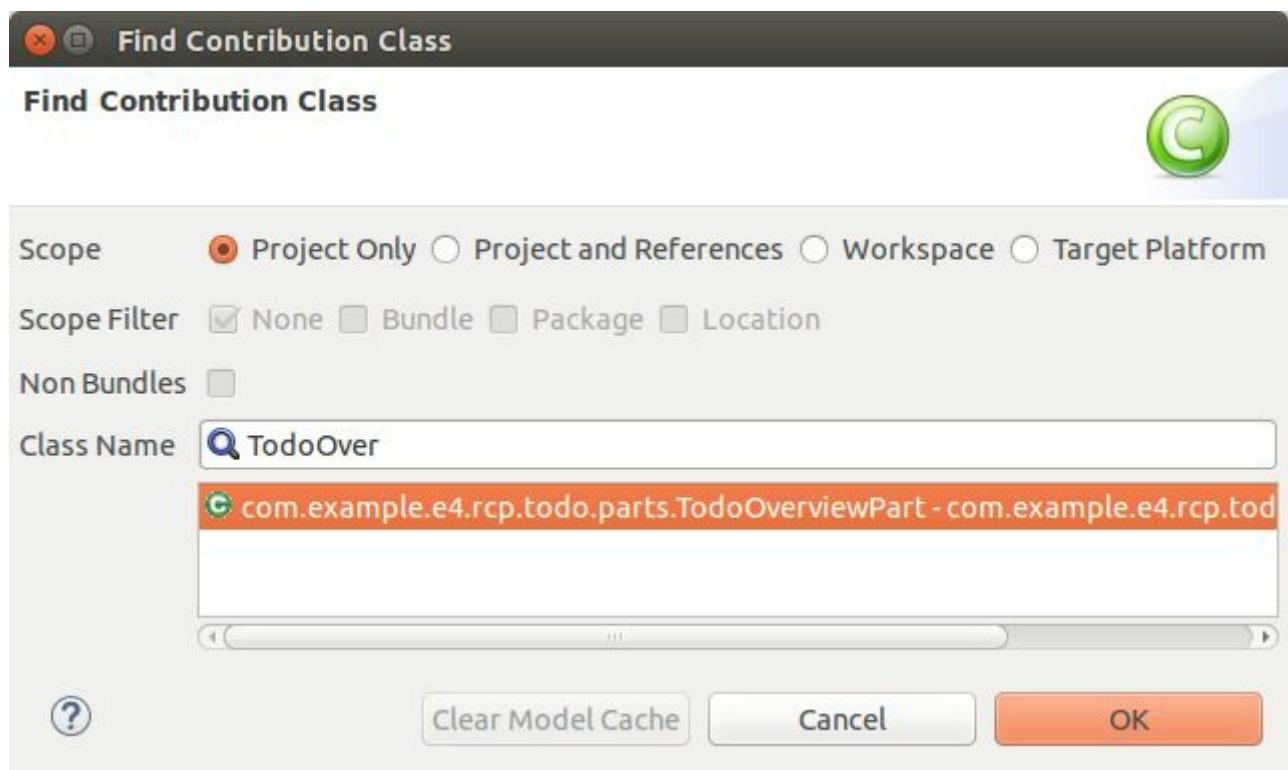
Open the `Application.e4xmi` file and connect the class with the corresponding part model element. You can do this via the *Class URI* property of the part model element.

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

Table 32.1. Mapping Java classes with part model element

Class	Part ID suffix
TodoOverviewPart	<code>*.todooverview</code>
TodoDetailsPart	<code>*.tododetails</code>
PlaygroundPart	<code>*.playground</code>

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



The following screenshot shows the result for the overview part.

Part

ID	com.example.e4.rcp.todo.part.todooverview
Label	Overview
Accessibility Phrase	
Tooltip	
Icon URI	<input type="text"/>
<u>Class URI</u>	bundleclass://com.example.e4.rcp.todo/com.example.e4.rcp.todo.parts.TodoOverviewPart
Container Data	
ToolBar	<input checked="" type="checkbox"/>
Closeable	<input checked="" type="checkbox"/>
To Be Rendered	<input checked="" type="checkbox"/>
Visible	<input checked="" type="checkbox"/>

32.3. Validating

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

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

Chapter 33. Exercise: Enter the dependencies

33.1. Add the 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.

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

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

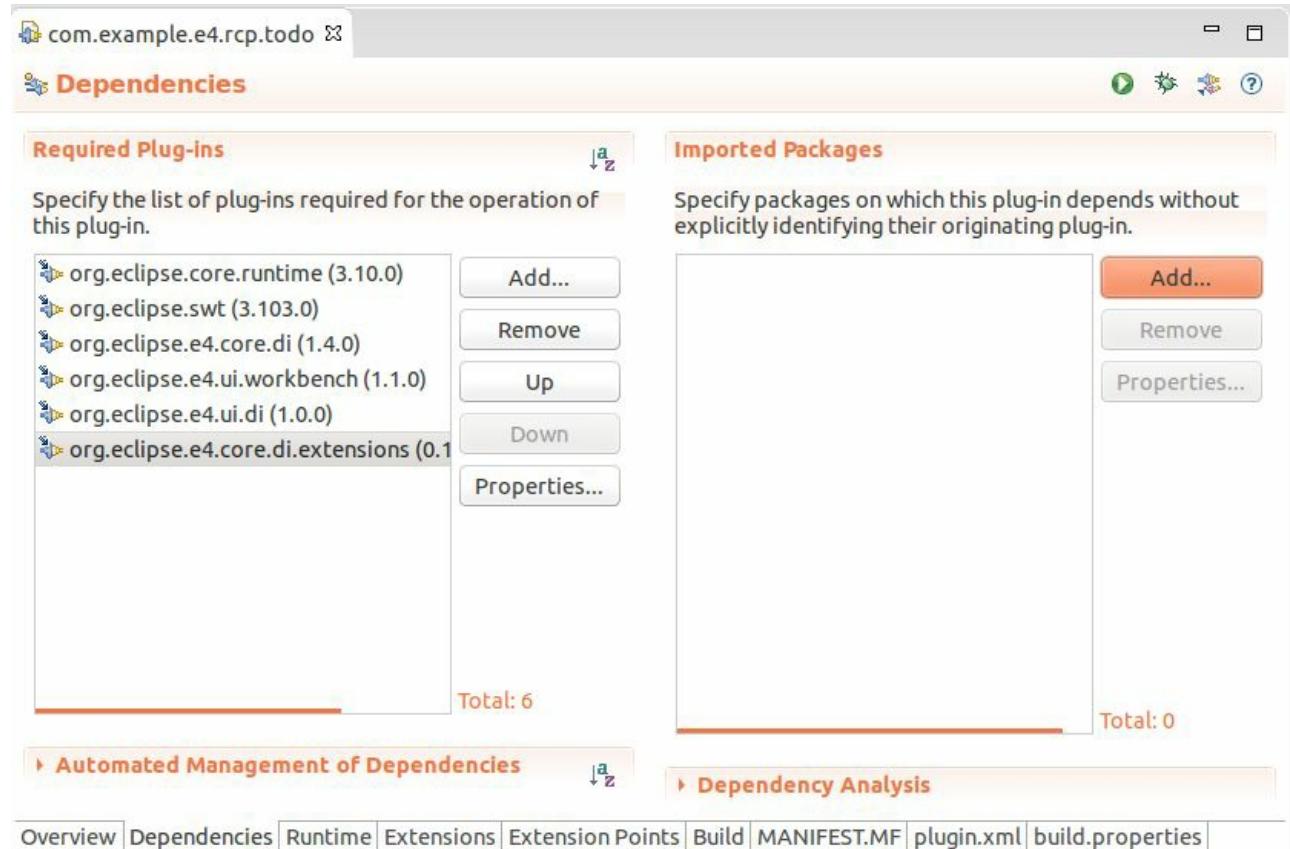
- `org.eclipse.core.runtime`
- `org.eclipse.swt`
- `org.eclipse.e4.core.di`
- `org.eclipse.e4.ui.workbench`
- `org.eclipse.e4.ui.di`
- `org.eclipse.e4.core.di.extensions`

Note

Before Eclipse 4.4 you had to define `javax.annotation` as package dependency with the minimum version of 1.1.0. As of Eclipse 4.4 the `javax.annotation` package is re-exported by the `org.eclipse.core.runtime` plug-in. Also the `javax.inject` plug-in is re-exported by `org.eclipse.core.runtime` therefore it is not required to define a dependency to it.

33.2. Validating

The result should be similar to the following screenshot.



33.3. More on dependencies

The exact details of applying this modular approach is covered in [Part X, “Eclipse modularity based on OSGi”](#).

Chapter 34. Optional Exercise: Using the model spy

34.1. Target

To test the application model dynamics, integrate the model spy and its dependencies into the run configuration of your application.

Warning

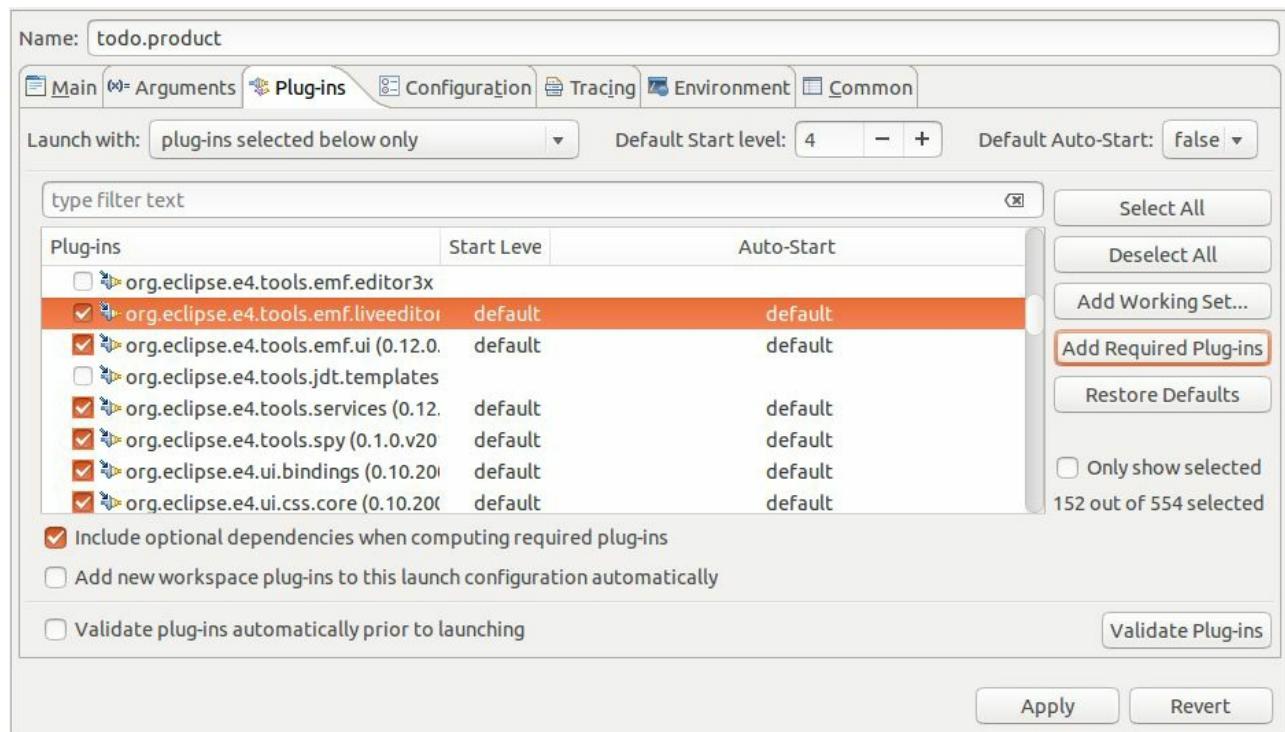
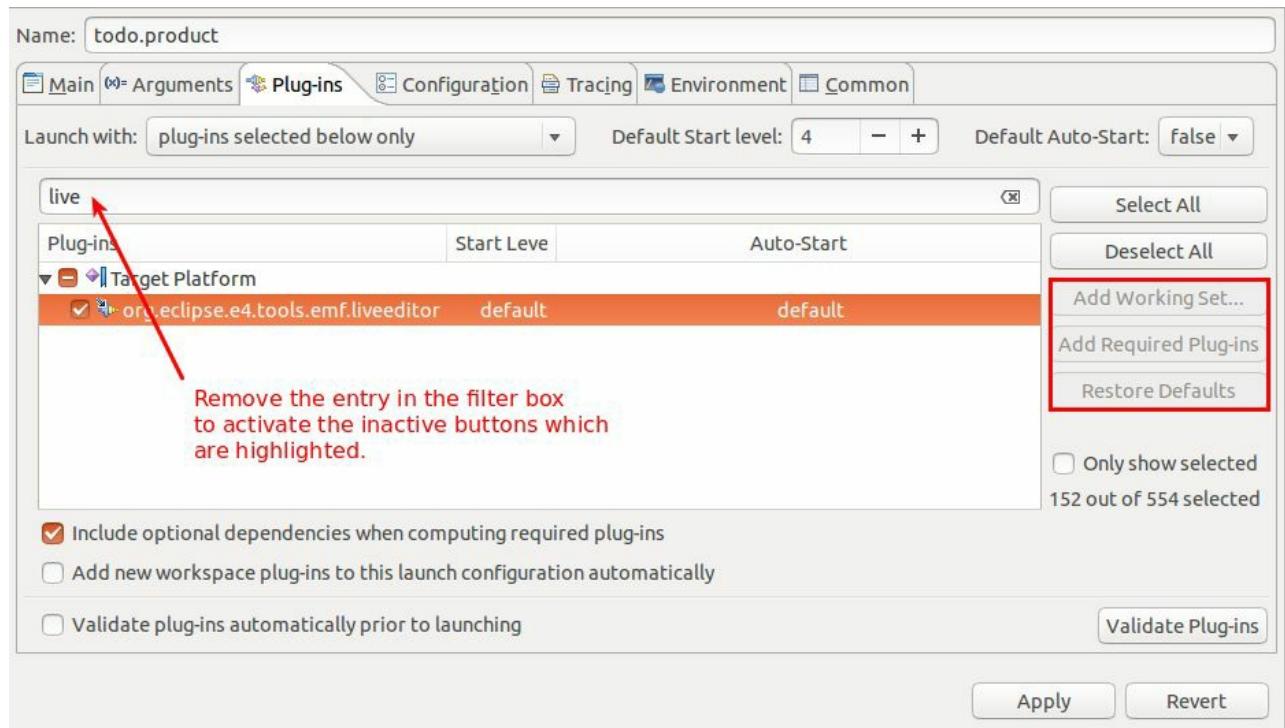
Do not modify your product configuration file for this exercise. You should not include the model spy in your exported application as it is only a test and debug tool. It is not intended to be used by end users.

Note

The model spy is a debug tool and not intended to be used in a productive application. So it might happen that this exercise does not work, or that you see additional exceptions in your log file, caused by the tool.

34.2. Adding the model spy to your runtime configuration

Open the run configuration of your product via the Run → Run Configurations... menu entry and add the `org.eclipse.e4.tools.emf.liveeditor` plug-in and its dependencies to it.



Press the *Validate* button to ensure that all dependencies are included.

34.3. Use the model spy

After you have added the new plug-ins to your run configuration press the *Run* button in the run configuration you just added.

Warning

Do not start the application now via the product. Starting via the product resets your changes in the runtime configuration.

Start your application and use the **Shift+Alt+F9** shortcut to open the model spy.

Warning

The code of the model spy adds this key binding to your application by modifying your application model at runtime. If you already defined key bindings this process might fail.

Use the model spy to change the runtime application model. For example, add a part (with a label) or change the size of the window. All changes should become immediately visible (except new menu entries, these changes are not updated automatically due to a bug).

34.4. Cleanup - remove model spy from your run configuration

After testing with the tool enabled, start your application again from your product configuration file to remove the live model editor plug-in from your run configuration.

Warning

If you leave the model spy included in your run configuration, it might cause side effects, e.g., in the past it created exceptions if key bindings were present in the application.

Part VII. Annotations and dependency injection

Chapter 35. The concept of dependency injection

35.1. What is dependency injection?

Dependency injection is a concept which is not limited to Java. But we will look at dependency injection from a Java point of view. The general concept behind dependency injection is called *Inversion of Control*. According to this concept a class should not configure its dependencies statically but should be configured from the outside.

A Java class has a dependency on another class if it uses an instance of this class. We call this a *class dependency*. For example a class which accesses a logger service has a dependency on this service class.

Ideally Java classes should be as independent as possible from other Java classes. This increases the possibility of reusing these classes and to be able to test them independently from other classes.

If the Java class creates an instance of another class via the `new` operator, it cannot be used (and tested) independently from this class and this is called a hard dependency. The following example shows a class which has no hard dependencies.

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

import java.util.logging.Logger;

import org.eclipse.e4.core.services.events.IEventBroker;

public class MyClass {

    private final static Logger logger;

    public MyClass(Logger logger) {
        this.logger = logger;
        // write an info log message
        logger.info("This is a log message.")
    }
}
```

Note

Please note that this class is just a normal Java class, there is nothing special about it, except that it avoids direct object creation.

A framework class, usually called the *dependency container*, could analyze the dependencies of this class. With this analysis it is able to create an instance of the class and inject the objects into the defined dependencies, via Java reflection.

This way the Java class has no hard dependencies, which means it does not rely on an instance of a certain class. This allows you to test your class in isolation, for example by using *mock* objects.

Mock objects (mocks) are objects which behave similar as the real object. But these mocks are not programmed; they are configured to behave in a certain predefined way. Mock is an English word which means to mimic or imitate.

If dependency injection is used, a Java class can be tested in isolation.

35.2. Using annotations to describe class dependencies

Different approaches exist to describe the dependencies of a class. The most common approach is to use Java annotations to describe the dependencies directly in the class.

The standard Java annotations for describing the dependencies of a class are defined in the Java Specification Request 330 (JSR330). This specification describes the `@Inject` and `@Named` annotations.

The following listing shows a class which uses annotations to describe its dependencies.

```
// import statements left out

public class MyPart {

    @Inject private Logger logger;

    // inject class for database access
    @Inject private DatabaseAccessClass dao;

    @Inject
    public void createControls(Composite parent) {
        logger.info("UI will start to build");
        Label label = new Label(parent, SWT.NONE);
        label.setText("Eclipse 4");
        Text text = new Text(parent, SWT.NONE);
        text.setText(dao.getNumber());
    }
}
```

Note

Please note that this class uses the `new` operator for the user interface components. This implies that this part of the code is nothing you plan to replace via your tests and that you made the decision to have a hard coupling to the corresponding user interface toolkit.

35.3. Where can objects be injected into a class?

Dependency injection can be performed on:

- the constructor of the class (construction injection)
- a field (field injection)
- the parameters of a method (method injection)

Dependency injection can be performed on static and on non-static fields and methods.

It is good practice to avoid using dependency injection on static fields and methods as this typically leads to confusion and has the following restrictions:

- Static fields will be injected after the first object of the class was created via DI, which means no access to the static field in the constructor
- Static fields can not be marked as final, otherwise the compiler or the runtime complains about them
- Static methods are called only once after the first object of the class was created

35.4. Order in which dependency injection is performed on a class

According to JSR330 the injection is done in the following order:

- constructor injection
- field injection
- method injection

The order in which the methods or fields annotated with `@Inject` are called is not defined by JSR330. You cannot assume that the methods or fields are called in the order of their declaration in the class.

Note

As fields and method parameters are injected after the constructor is called, you cannot use injected member variables in the constructor.

Chapter 36. Dependency injection in Eclipse

36.1. Define class dependencies in Eclipse

The programming model in Eclipse supports constructor, method and field injection according to the Java Specification Request 330 (JSR330). Eclipse also defines additional annotations for the purpose of dependency injection. The most important annotations are covered in [Section 36.2, “Annotations to define class dependencies in Eclipse”](#), other more special annotations are covered in there corresponding chapters.

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

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

36.2. Annotations to define class dependencies in Eclipse

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

Table 36.1. Basic annotations for dependency injection

Annotation	Description
<code>@javax.inject.Inject</code>	Defined by JSR330, can be added to a field, a constructor or a method. The Eclipse framework tries to inject the corresponding objects into the fields or the parameters of the instance.
<code>@javax.inject.Named</code>	Defined by JSR330, defines the key for the value which should be injected. By default, the fully qualified class name is used as the key. Several keys for default values are defined as constants in the <code>IServiceConstants</code> interface.
<code>@Optional</code>	Eclipse specific annotation, marks an injected value to be optional. If no valid object can be determined for the given key (and type), the framework does not throw an exception. The specific behavior depends on where the <code>@Optional</code> is placed: <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.
<code>@GroupUpdates</code>	Eclipse specific annotation, indicates that updates for this <code>@Inject</code> should be batched. If you change such objects in the Eclipse context , the update is triggered by the <code>processWaiting()</code> method of the <code>IEclipseContext</code> object. This annotation is intended to be used by the platform for performance optimization and should rarely be necessary in RCP applications.

Note

The Eclipse platform supports additional annotations for special purposes, e.g., for receiving events (sent by the event service) or working with preferences. For a summary of all standard annotations defined in the Eclipse platform see [Section A.1, “Standard annotations in Eclipse”](#).

36.3. On which objects does Eclipse perform dependency injection?

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

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

36.4. Dynamic dependency injection based on key / value changes

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

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

The re-injection only works on methods and fields which are marked with `@Inject`. It will not work on parameters injected into constructors and methods which are marked with `@PostConstruct`, as these methods are only executed once. This will be explained later in [Section 37.4, “Life cycle of the Eclipse context”](#).

Note

This does not mean that Eclipse tracks the fields of the value to which the key points. For example if the `mykey1` key points to a `Todo` object as value, and the key points to a new object, this triggers the re-injection of the value to all objects which have a relevant class dependency. But if a field inside the existing `Todo` object changes, it does not trigger a re-injection.

Part VIII. Dependency injection and the Eclipse context

Chapter 37. The Eclipse context

37.1. What is the Eclipse context?

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

The context is similar to a `Map` data structure, in which objects can be placed under a certain key. The key is a `String` and in several cases the fully qualified class name is used as key. The value (to which the key points) can be injected into other objects.

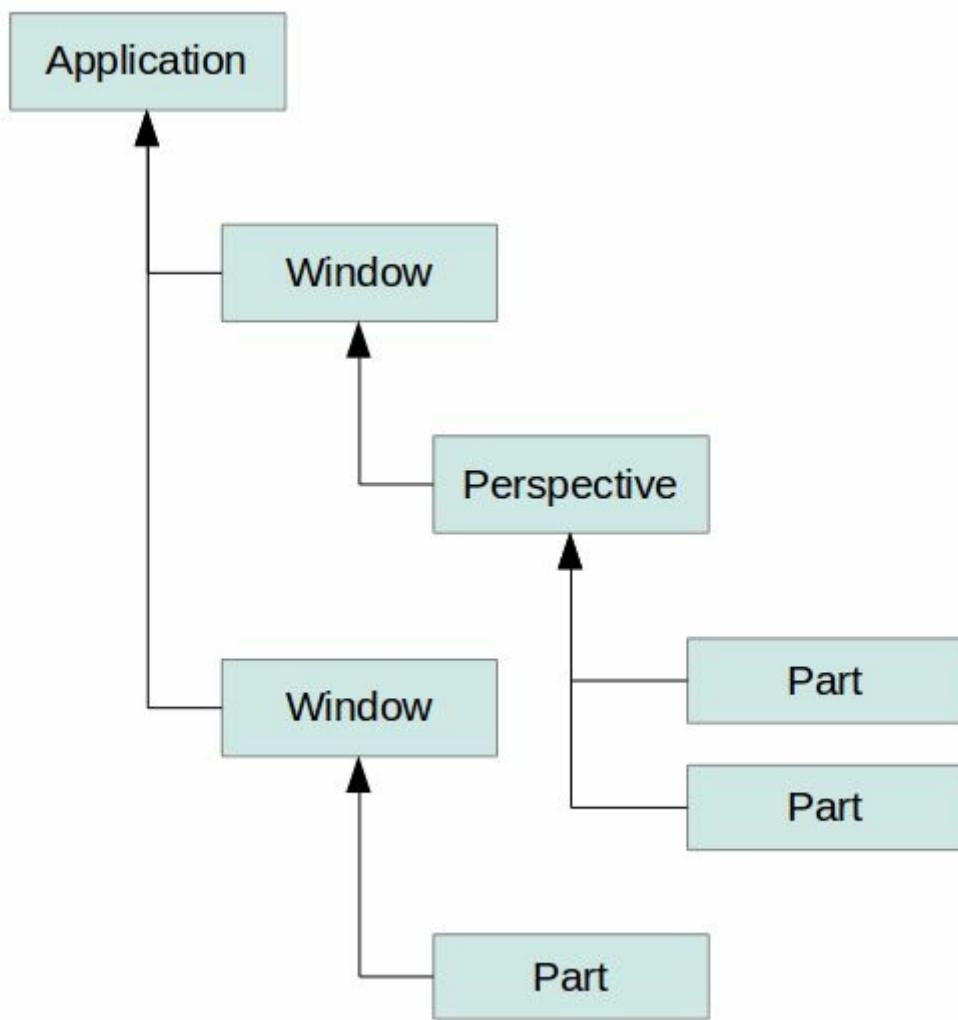
37.2. Relationship definition in the Eclipse context

As described earlier the Eclipse context is similar to a `Map` data structure. But unlike a map, the Eclipse context is hierarchical and can also dynamically compute values for requested keys.

For certain model objects (see [Section 37.3, “Which model elements have a local context?”](#)) a local context is created. Such a context is associated with an application model object.

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

A sample context hierarchy is depicted in the following picture.



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

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

37.3. Which model elements have a local context?

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

- `MApplication`
- `MWindow`
- `MPerspective`
- `MPart`
- `MPopupMenu`

37.4. Life cycle of the Eclipse context

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

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

Note

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

For example, the `ContributedPartRenderer` class is the default renderer for part model objects. This renderer creates a `Composite` for every part and puts this `Composite` into the local context of the part.

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

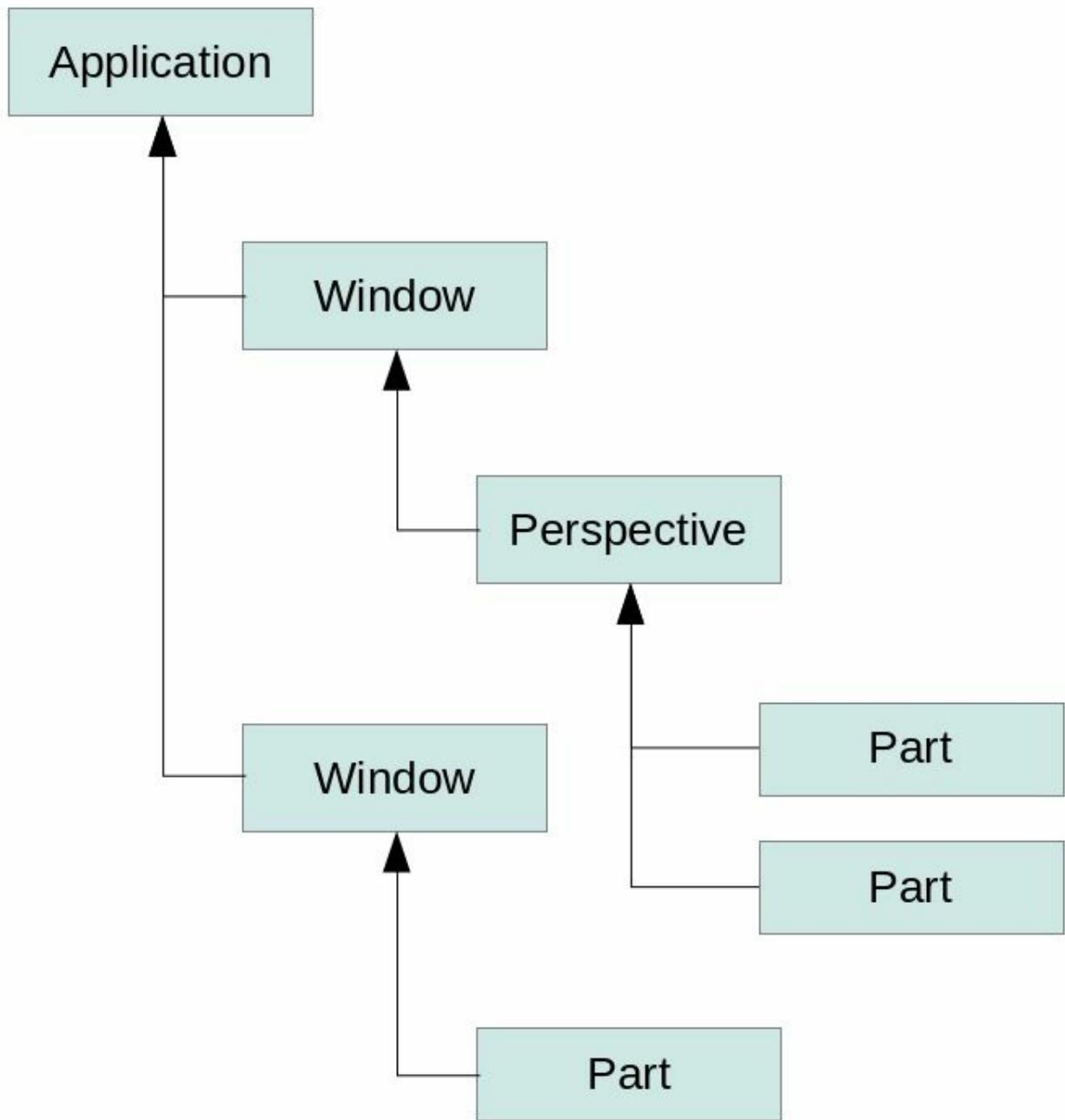
Objects in the context are persisted in memory (transient), i.e., once the application is stopped the context gets destroyed.

Chapter 38. Object selection during dependency injection

38.1. How are objects selected for dependency injection

As described in [Section 36.1, “Define class dependencies in Eclipse”](#) an object which is created by Eclipse can use annotations to describe its class dependencies.

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



As you learn in later chapters the Eclipse context is not the only possible source of objects which can get injected. Other examples which are covered later are OSGi services, preferences, events and custom objects. The search happens (mostly) transparently for the caller of the injection.

38.2. How to access the model objects?

For the class references in the application model (see [Section 25.2, “Connect model elements to classes”](#)), the Eclipse framework creates the corresponding objects when needed. Such an object has access to its corresponding model object (see [Section 26.1, “Model objects”](#)) via dependency injection.

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

38.3. Default entries in the Eclipse context

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

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

The context can be modified by the application code and the framework. As the Eclipse framework automatically tracks the dependencies of the objects it creates, it can update them as described in [Section 36.4, “Dynamic dependency injection based on key / value changes”](#).

38.4. Qualifiers for accessing the active part or shell

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

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

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

To track the active shell use the `IServiceConstants.ACTIVE_SHELL` key.

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

Note

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

38.5. Tracking a child context with @Active

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

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

The usage of this annotation is demonstrated by the following code snippet.

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

Note

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

Part IX. Behavior annotations

Chapter 39. Using annotations to define behavior

39.1. API definition

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

For this purpose every framework defines an Application Programming Interface (API). This API defines how you can interact with the framework from your code. The API also defines the interaction of application objects created or controlled by the framework. Typically, a framework uses inheritance or annotations for this purpose.

39.2. API definition via inheritance

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

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

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

39.3. API definition via annotations

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

The following table lists the available behavior annotations for parts.

Table 39.1. Eclipse life cycle annotations for parts

Annotation	Description
<code>@PostConstruct</code>	Is called after the class is constructed and the field and method injection has been performed.
<code>@PreDestroy</code>	Is called before the class is destroyed. Can be used to clean up resources.
<code>@Focus</code>	Is called whenever the part gets the focus.
<code>@Persist</code>	Is called if a save request on the part is triggered by the Eclipse framework.
<code>@PersistState</code>	Is called before the model object is disposed, so that the part is able to save its instance state. This method is called before the <code>@PreDestroy</code> method.

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

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

39.4. Behavior annotations imply method dependency injection

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

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

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

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

39.6. Why is the @PostConstruct method not called?

Both Java 7 and the Eclipse platform expose the `@PostConstruct` annotation. In your Eclipse application you need to tell the framework that the annotation from the Eclipse platform should be used.

In case your `@PostConstruct` method is not called, ensure that you have defined a dependency to `org.eclipse.core.runtime` in the `MANIFEST.MF` file. See <http://wiki.eclipse.org/Eclipse4/RCP/FAQ> for details on this issue.

Note

`org.eclipse.core.runtime` exports `javax.annotation` in the correct version. If you have a dependency to `org.eclipse.core.runtime` in your `MANIFEST.MF` file, no additional package dependency is needed. If, for some reasons, you want to avoid a dependency to `org.eclipse.core.runtime`, you could define a package dependency to the `javax.annotation` package and set the version to 1.0.0.

Chapter 40. Exercise: Using @PostConstruct

40.1. Implement an @PostConstruct method

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

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

// more code

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

40.2. Validating

Run your application and validate that the `@PostConstruct` method is called. If this does not work, see [Section 39.6, “Why is the `@PostConstruct` method not called?”](#) for a solution.

Part X. Eclipse modularity based on OSGi

Chapter 41. Software modularity with OSGi

41.1. What is software modularity?

An application consists of different parts, these are typically called *software components* or *software modules*.

These components interact with each other via an Application Programming Interface (API). The API is defined as a set of classes and methods which can be used from other components. A component also has a set of classes and methods which are considered as internal to the software component.

If a component uses an API from another component, it has a dependency to the other component, i.e., it requires the other component exists and works correctly.

A component which is used by other components should try to keep its API stable to avoid that a change affects other components. But it should be free to change its internal implementation.

Java, in its current version (Java 8), provides no structured way to describe software component dependencies. Java only supports the usage of access modifiers, but every public class can be called from another software component. What is desired is a way to explicitly define the API of a software component. The OSGi specification fills this gap.

41.2. What is OSGi?

OSGi is a set of specifications which, in its core specification, defines a component and service model for Java. A practical advantage of OSGi is that every software component can define its API via a set of exported Java packages and that every component can specify its required dependencies.

The components and services can be dynamically installed, activated, de-activated, updated and de-installed.

41.3. OSGi implementations

The OSGi specification has several implementations, for example Eclipse Equinox, Knopflerfish OSGi or Apache Felix.

Eclipse Equinox is the reference implementation of the base OSGi specification. It is also the runtime environment on which Eclipse applications are based.

41.4. Plug-in or bundles as software component

The OSGi specification defines a bundle as the smallest unit of modularization, i.e., in OSGi a software component is a bundle. The Eclipse programming model typically calls them *plug-in* but these terms are interchangeable. A valid plug-in is always a valid bundle and a valid bundle is always a valid plug-in. In this book the usage of *plug-in* is preferred, to be consistent with the terminology of Eclipse plug-in development.

A plug-in is a cohesive, self-contained unit, which explicitly defines its dependencies to other components and services. It also defines its API via Java packages.

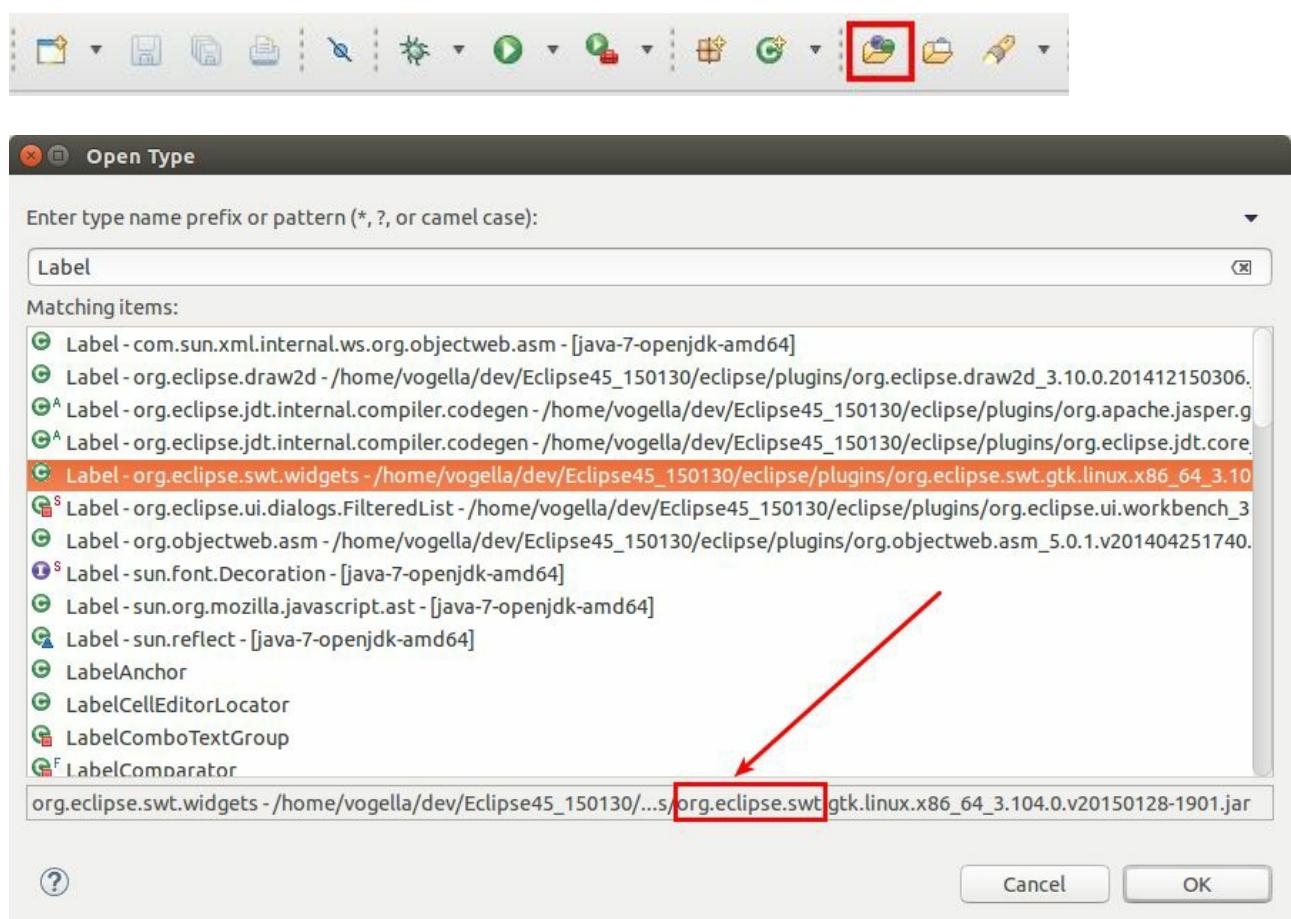
41.5. Naming convention: simple plug-in

A plug-in can be generated by Eclipse via the File → New → Other... → Plug-In Development → Plug-In Project menu entry. The corresponding wizard allows specifying several options. This book calls plug-ins generated with the following options a *simple plug-in* or *simple bundle*.

- No Activator
- No contributions to the user interface
- Not a 3.x rich client application
- Generated without a template

41.6. Find the plug-in for a certain class

You frequently have to find the plug-in for a given class. The Eclipse IDE makes it easy to find the plug-in for a class. After enabling the *Include all plug-ins from target into Java Search* setting in the Eclipse IDE preferences you can use the *Open Type* dialog (**Ctrl+Shift+T**) to find the plug-in for a class. The *JAR* file is shown in this dialog and the prefix of the JAR file is typically the plug-in which contains this class.



Chapter 42. OSGi metadata

42.1. The manifest file (MANIFEST.MF)

Technically OSGi plug-ins are `.jar` files with additional meta information. This meta information is stored in the `META-INF/MANIFEST.MF` file. This file is called the *manifest* file and is part of the standard Java specification and OSGi adds additional metadata to it. According to the Java specification, any Java runtime must ignore unknown metadata. Therefore, plug-ins can be used without restrictions in other Java environments.

The following listing is an example for a manifest file.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Popup Plug-in
Bundle-SymbolicName: com.example.myosgi; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: com.example.myosgi.Activator
Require-Bundle: org.eclipse.ui,
  org.eclipse.core.runtime
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

The following table gives an explanation of the identifiers in the manifest file. For additional information on the unique identifier see [Section 42.2, “Bundle-SymbolicName and Version”](#) and for information on the version schema which is typically used in OSGi see [Section 42.3, “Semantic Versioning with OSGi”](#).

Table 42.1. OSGi identifiers in the manifest file

Identifier	Description
Bundle-Name	Short description of the plug-in.
Bundle-SymbolicName	The unique identifier of the plug-in. If this plug-in is using the extension point functionality of Eclipse, it must be marked as Singleton. You do this by adding the following statement after the <code>Bundle-SymbolicName</code> identifier: ; <code>singleton:=true</code>
Bundle-Version	Defines the plug-in version and must be incremented if a new version of the plug-in is

	<p>published.</p> <p>Defines an optional activator class which implements the <code>BundleActivator</code> interface. An instance of this class is created when the plug-in gets activated. Its <code>start()</code> and <code>stop()</code> methods are called whenever the plug-in is started or stopped. An OSGi activator can be used to configure the plug-in during startup. The execution of an activator increases the startup time of the application, therefore this functionality should be used carefully.</p>
Bundle-Activator	
Bundle-RequiredExecutionEnvironment (BREE)	<p>Specify which Java version is required to run the plug-in. If this requirement is not fulfilled, then the OSGi runtime does not load the plug-in. Setting this to <code>lazy</code> will tell the OSGi runtime that this plug-in should only be activated if one of its components, i.e. classes and interfaces are used by other plug-ins. If not set, the Equinox runtime does not activate the plug-in, i.e., services provided by this plug-in are not available.</p>
Bundle-ActivationPolicy	
Bundle-ClassPath	<p>The Bundle-ClassPath specifies where to load classes from the bundle. The default is '.' which allows classes to be loaded from the root of the bundle. You can also add JAR files to it, these are called <i>nested JAR files</i>.</p>

42.2. Bundle-SymbolicName and Version

Each plug-in has a symbolic name which is defined via the `Bundle-SymbolicName` property. The name starts by convention with the reverse domain name of the plug-in author, e.g., if you own the "example.com" domain then the symbolic name would start with "com.example".

Each plug-in has also a version number in the `Bundle-Version` property.

The `Bundle-Version` and the `Bundle-SymbolicName` uniquely identifies a plug-in.

42.3. Semantic Versioning with OSGi

OSGi recommends to use a $<\text{major}>.<\text{minor}>.<\text{patch}>$ schema for the version number which is defined via the `Bundle-Version` field identifier. If you change your plug-in code you increase the version according to the following rule set.

- $<\text{patch}>$ is increased if all changes are backwards compatible.
- $<\text{minor}>$ is increased if public API has changed but all changes are backwards compatible.
- $<\text{major}>$ is increased if changes are not backwards compatible.

For more information on this version scheme see the [Eclipse Version Numbering Wiki](#).

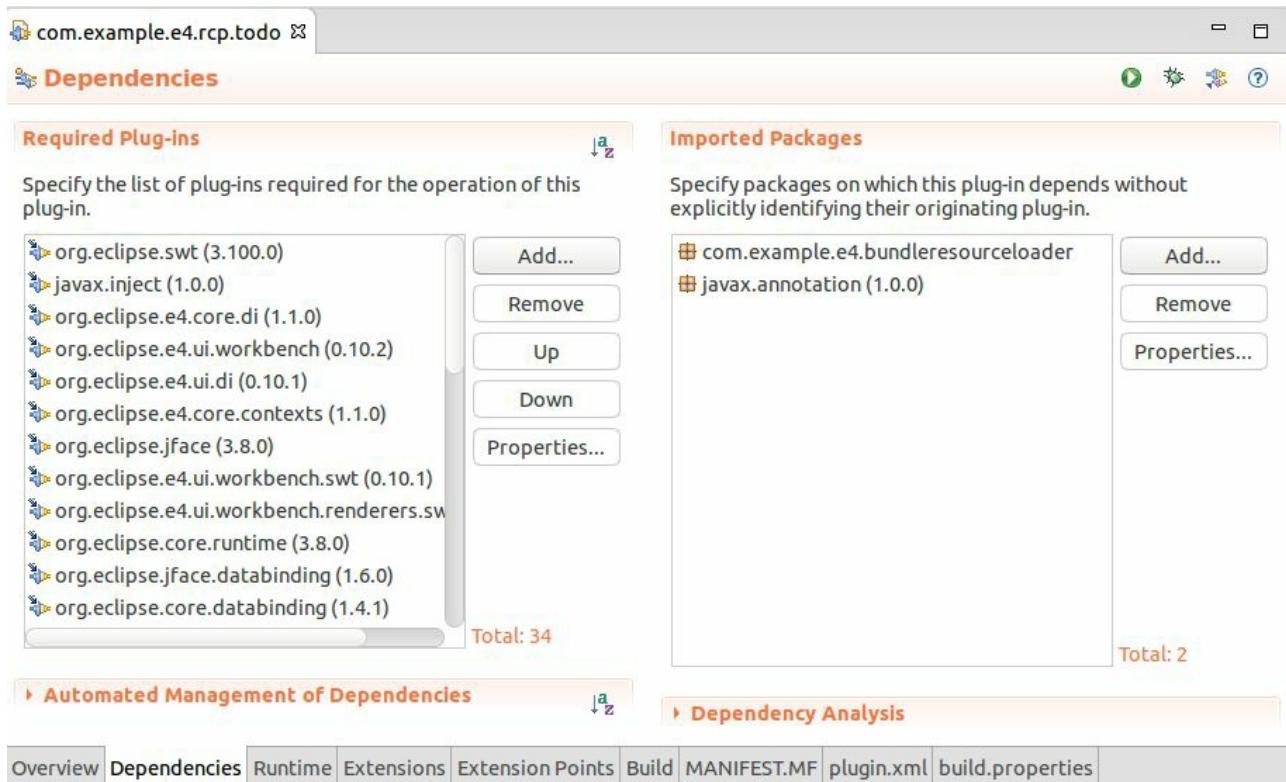
Chapter 43. Defining the dependencies of a plug-in

43.1. Specifying plug-in dependencies via the manifest file

A plug-in can define dependencies to other software components via its manifest file. OSGi prevents access to classes without a defined dependency and throws a `ClassNotFoundException`. The only exception are packages from the Java runtime environment (based on the `Bundle-RequiredExecutionEnvironment` definition of the plug-in); these packages are always available to a plug-in without an explicitly defined dependency.

If you add a dependency to your manifest file, the Eclipse IDE automatically adds the corresponding `JAR` file to your project classpath.

You can define dependencies either as plug-in dependencies or package dependencies. If you define a plug-in dependency your plug-in can access all exported packages of this plug-in. If you specify a package dependency you can access this package. Using package dependencies allows you to exchange the plug-in which provides this package at a later point in time. If you require this flexibility prefer the usage of package dependencies.



A plug-in can define that it depends on a certain version (or a range) of another

bundle, e.g., plug-in A can define that it depends on plug-in C in version 2.0, while plug-in B defines that it depends on version 1.0 of plug-in C.

The OSGi runtime ensures that all dependencies are present before it starts a plug-in. OSGi reads the manifest file of a plug-in during its installation. It ensures that all dependent plug-ins are also resolved and, if necessary, activates them before the plug-in starts.

43.2. Life cycle of plug-ins in OSGi

With the installation of a plug-in in the OSGi runtime the plug-in is persisted in a local bundle cache. The OSGi runtime then tries to resolve its dependencies.

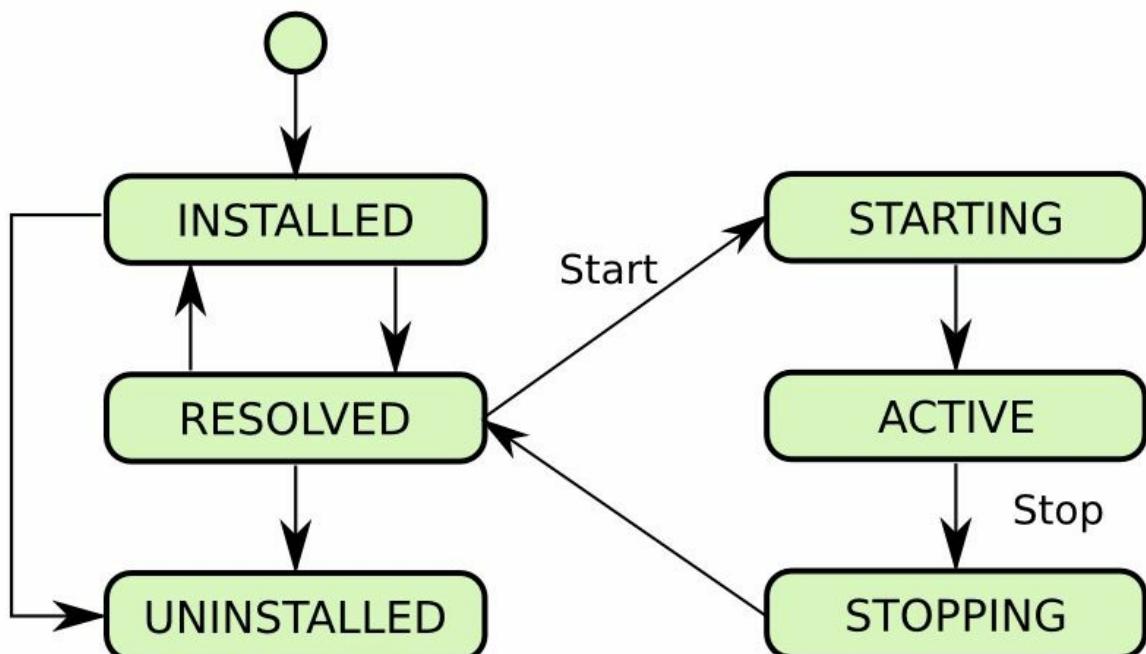
If all required dependencies are resolved, the plug-in is in the *RESOLVED* status otherwise it stays in the *INSTALLED* status.

In case several plug-ins exist which can satisfy the dependency, the plug-in with the highest valid version is used.

If the versions are the same, the plug-in with the lowest unique identifier (ID) is used. Every plug-in gets this ID assigned by the framework during the installation.

When the plug-in starts, its status is *STARTING*. After a successful start, it becomes *ACTIVE*.

This life cycle is depicted in the following graphic.



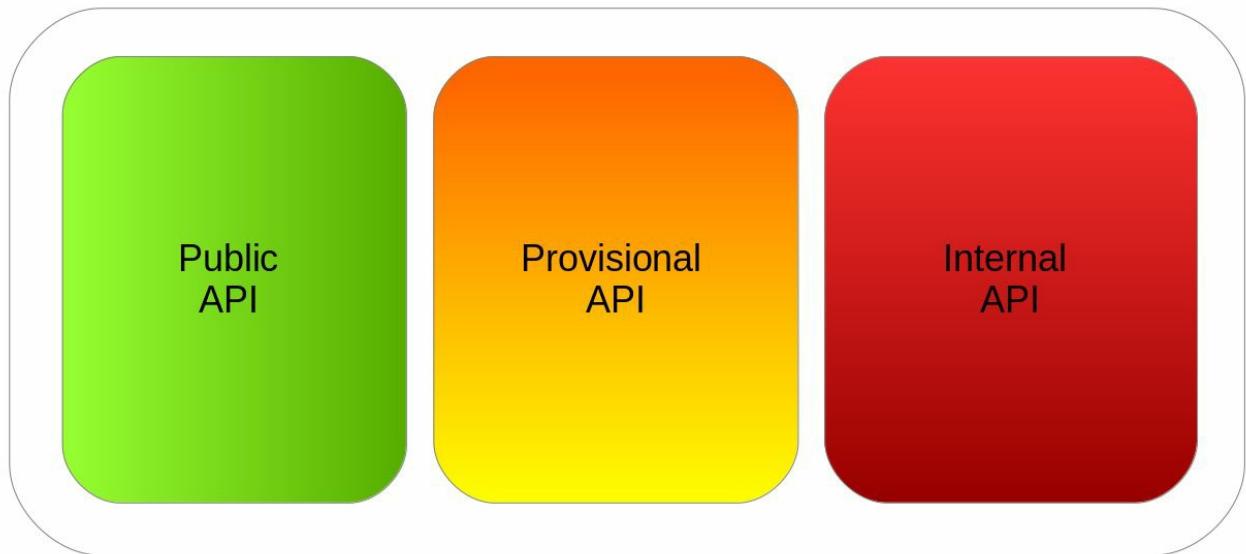
43.3. Dynamic imports of packages

For legacy reasons OSGi supports a dynamic import of packages. See [OSGi Wiki for dynamic imports](#) for details. You should not use this feature, it is a symptom of a non-modular design.

Chapter 44. Defining an API

44.1. Specifying the API of a plug-in

In the `MANIFEST.MF` file a plug-in also defines its API via the Export-Package Identifier. All packages which are not explicitly exported are not visible to other plug-ins.



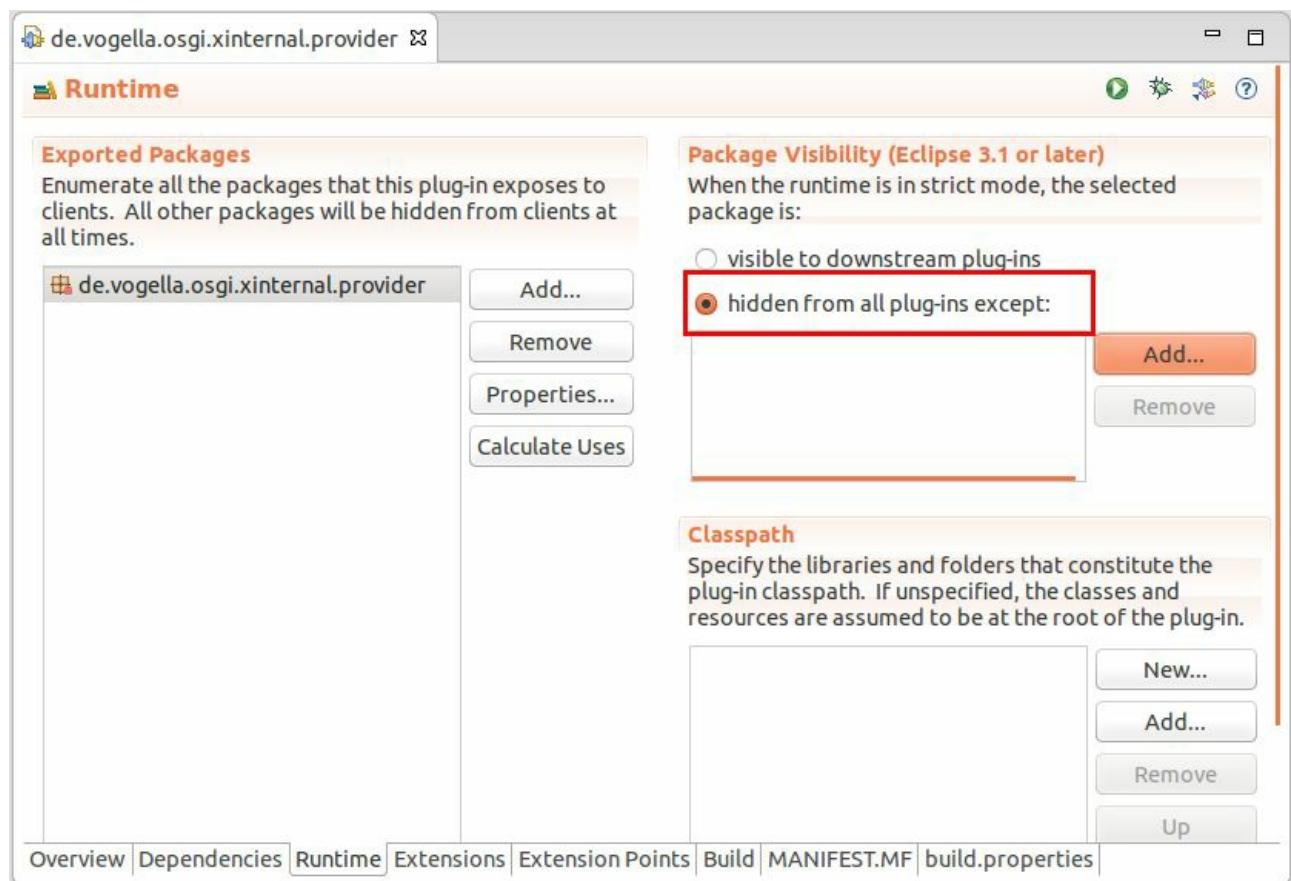
All these restrictions are enforced via a specific OSGi classloader. Each plug-in has its own classloader. Access to restricted classes is not possible.

Unfortunately OSGi can not prevent you from using Java reflection to access these classes. This is because OSGi is based on the Java runtime which does not yet support a modularity layer.

44.2. Provisional API

Via the `x-internal` flag the OSGi runtime can mark an exported package as provisional. This allows other plug-ins to consume the corresponding classes, but indicates that these classes are not considered as official API.

The following screenshot shows how to set a package as `x-internal` in the manifest editor.

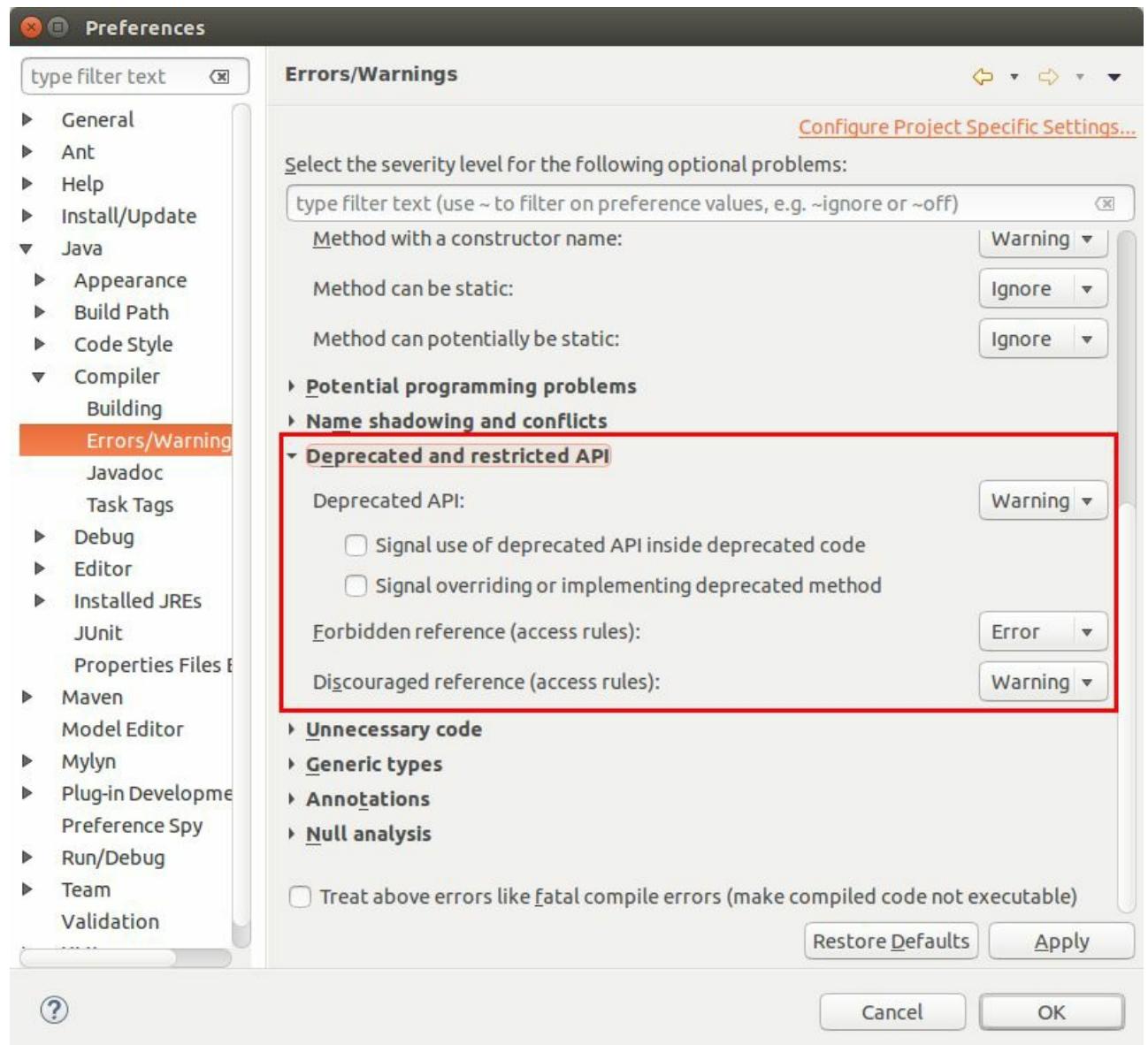


This is how the corresponding manifest file looks like.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Provider
Bundle-SymbolicName: de.vogella.osgi.xinternal.provider
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Export-Package: de.vogella.osgi.xinternal.provider;x-internal:=true
```

You can configure how the Eclipse Java editor shows the usage of provisional API. Such an access can be configured to be displayed as, error, warning or if such access should be result in no additional message.

The default is to display a warning message. You can adjust this in the Eclipse preferences via the Window → Preferences → Java → Compiler → Errors/Warnings preference setting.



44.3. Provisional API with exceptions via x-friends

You can define that a set of plug-ins can access provisional API without a warning or error message. This can be done via the `x-friends` directive. This flag is added if you add a plug-in to the *Package Visibility* section on the *Runtime* tab of the manifest editor.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Provider
Bundle-SymbolicName: de.vogella.osgi.xinternal.provider
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Export-Package: de.vogella.osgi.xinternal.provider;x-friends:="another.bundle"
```

The `x-friends` setting has the same effect as `x-internal` but all plug-ins mentioned in the `x-friends` setting can access the package without receiving an error or warning message.

Chapter 45. Using the OSGi console

45.1. The OSGi console

The OSGi console is like a command-line shell. In this console you can type a command to perform an OSGi action. This can be useful to analyze problems on the OSGi layer of your application.

Use, for example, the command `ss` to get an overview of all bundles, their status and bundle-id. The following table is a reference of the most important OSGi commands.

Table 45.1. OSGi commands

Command Description

<code>help</code>	Lists the available commands.
<code>ss</code>	Lists the installed bundles and their status.
<code>ss <i>vogella</i></code>	Lists bundles and their status that have <code>vogella</code> within their name.
<code>start <i><bundle-id></i></code>	Starts the bundle with the <code><bundle-id></code> ID.
<code>stop <i><bundle-id></i></code>	Stops the bundle with the <code><bundle-id></code> ID.
<code>diag <i><bundle-id></i></code>	Diagnoses a particular bundle. It lists all missing dependencies.
<code>install <i>URL</i></code>	Installs a bundle from a URL.
<code>uninstall <i><bundle-id></i></code>	Uninstalls the bundle with the <code><bundle-id></code> ID.
<code>bundle <i><bundle-id></i></code>	Shows information about the bundle with the <code><bundle-id></code> ID, including the registered and used services.
<code>headers <i><bundle-id></i></code>	Shows the MANIFEST.MF information for a bundle.
<code>services <i>filter</i></code>	Shows all available services and their consumer. Filter is an optional LDAP filter, e.g., to see all services which provide a ManagedService implementation use the "services

(objectclass=*ManagedService)" command.

45.2. Required bundles

You have to add the following bundles to your runtime configuration to use the OSGi console.

- org.eclipse.equinox.console
- org.apache.felix.gogo.command
- org.apache.felix.gogo.runtime
- org.apache.felix.gogo.shell

45.3. Telnet

If you specify the `-console` parameter in your launch configuration Eclipse will allow you to interact with the OSGi console. An OSGi launch configuration created with the Eclipse IDE contains this parameter by default. Via the following parameter you can open a port to which you can connect via the telnet protocol.

```
-console 5555
```

If you open a telnet session to the OSGi console, you can use tab completion and a history of the commands similar to the *Bash* shell under Linux.

45.4. Access to the Eclipse OSGi console

You can also access the OSGi console of your running Eclipse IDE. In the *Console View* you find a menu entry with the tooltip *Open Console*. If you select *Host OSGi Console*, you will have access to your running OSGi instance.

Please note that interfering with your running Eclipse IDE via the OSGi console, may put the Eclipse IDE into a bad state.

Chapter 46. Exercise: Data model plug-in

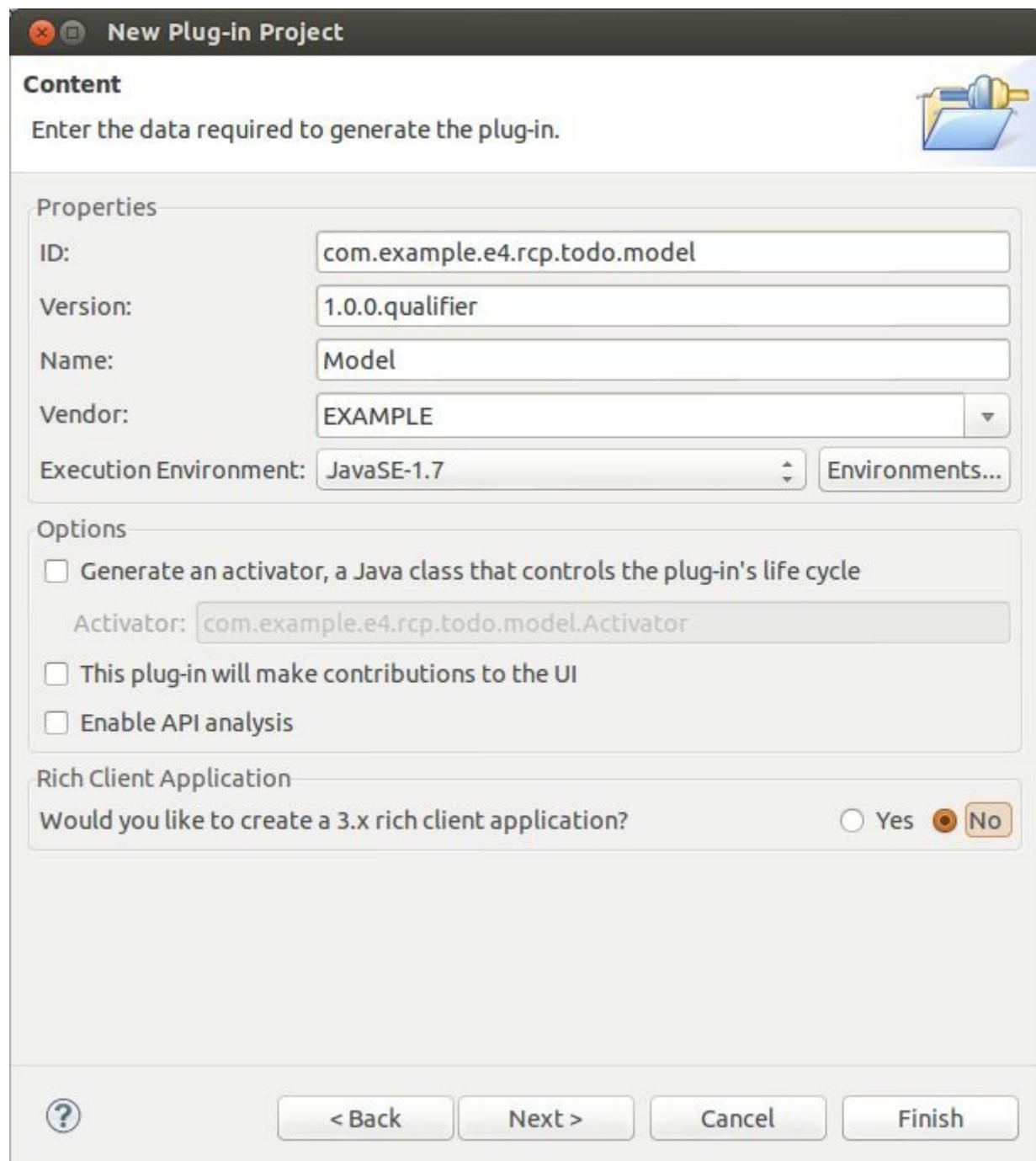
46.1. Target of the exercise

In this exercise you create a plug-in for the definition of your data model. You also make this data model available to other plug-ins.

46.2. Create the plug-in for the data model

Create a simple plug-in project (see [Section 41.5, “Naming convention: simple plug-in”](#)) called `com.example.e4.rcp.todo.model`.

The following screenshot depicts the second page of the plug-in project wizard and its corresponding settings. Press the *Finish* button on this page to avoid the usage of templates.



46.3. Create the base class

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

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

import java.util.Date;

public class Todo {

    private final long id;
    private String summary = "";
    private String description = "";
    private boolean done = false;
    private Date dueDate;

}
```

Note

You see an error for your final id field. This error is solved in the [Section 46.4, “Generate constructors”](#) section.

46.4. Generate constructors

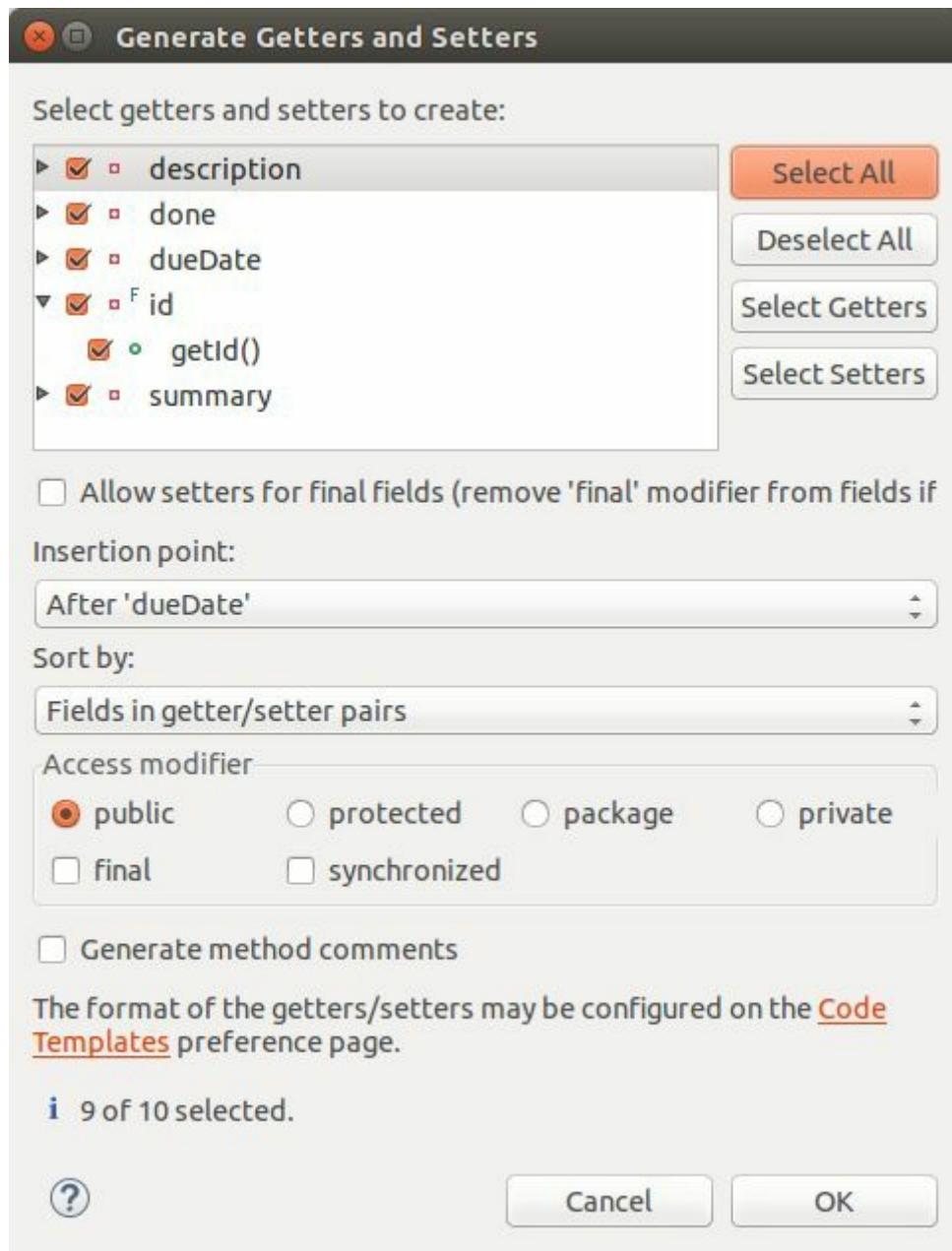
Select Source → Generate Constructor using Fields... to generate a constructor using all fields. Use the same approach to create another constructor using only the `id` field.

Warning

Ensure that you have created both constructors, because they are required in the following exercises.

46.5. Generate getter and setter methods

Use the Source → Generate Getter and Setter... menu to create getters and setters for all fields except a setter for `id`. The `id` is final and cannot be changed after the creation of a `Todo` object.



Adjust the generated getter and setter for the `dueDate()` field to make defensive copies. The `Date` class is not immutable and we want to avoid that an instance of `Todo` can be changed from outside without the corresponding setter.

```
public Date getDueDate() {  
    return new Date(dueDate.getTime());
```

```

}

public void setDueDate(Date dueDate) {
    this.dueDate = new Date(dueDate.getTime());
}

```

The resulting class should look like the following listing.

```

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

import java.util.Date;

public class Todo {

    private final long id;
    private String summary = "";
    private String description = "";
    private boolean done = false;
    private Date dueDate;

    public Todo(long id) {
        this.id = id;
    }

    public Todo(long id, String summary, String description, boolean done,
               Date dueDate) {
        this.id = id;
        this.summary = summary;
        this.description = description;
        this.done = done;
        this.dueDate = dueDate;
    }

    public long getId() {
        return id;
    }

    public String getSummary() {
        return summary;
    }

    public void setSummary(String summary) {
        this.summary = summary;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}

```

```
public boolean isDone() {
    return done;
}

public void setDone(boolean done) {
    this.done = done;
}

public Date getDueDate() {
    return new Date(dueDate.getTime());
}

public void setDueDate(Date dueDate) {
    this.dueDate = new Date(dueDate.getTime());
}

}
```

Note

Why is the `id` field marked as `final`? We will use this field to generate the `equals` and `hashCode()` methods therefore it should not be mutable. Changing a field which is used in the `equals` and `hashCode()` methods can create bugs which are hard to identify, i.e., an object is contained in a `HashMap` but not found.

46.6. Generate `toString()`, `hashCode()` and `equals()` methods

Use Eclipse to generate a `toString()` method for the `Todo` class based on the `id` and `summary` field. This can be done via the Eclipse menu Source → Generate `toString()`....

Also use Eclipse to generate a `hashCode()` and `equals()` method based on the `id` field. This can be done via the Eclipse menu Source → Generate `hashCode()` and `equals()`....

46.7. Write a copy() method

Add the following `copy()` method to the class.

```
public Todo copy() {  
    return new Todo(this.id, this.summary,  
        this.description, this.done,  
        this.dueDate);  
}
```

46.8. Create the interface for the todo service

Create the following `ITodoService` interface.

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

import java.util.List;

public interface ITodoService {

    Todo getTodo(long id);

    boolean saveTodo(Todo todo);

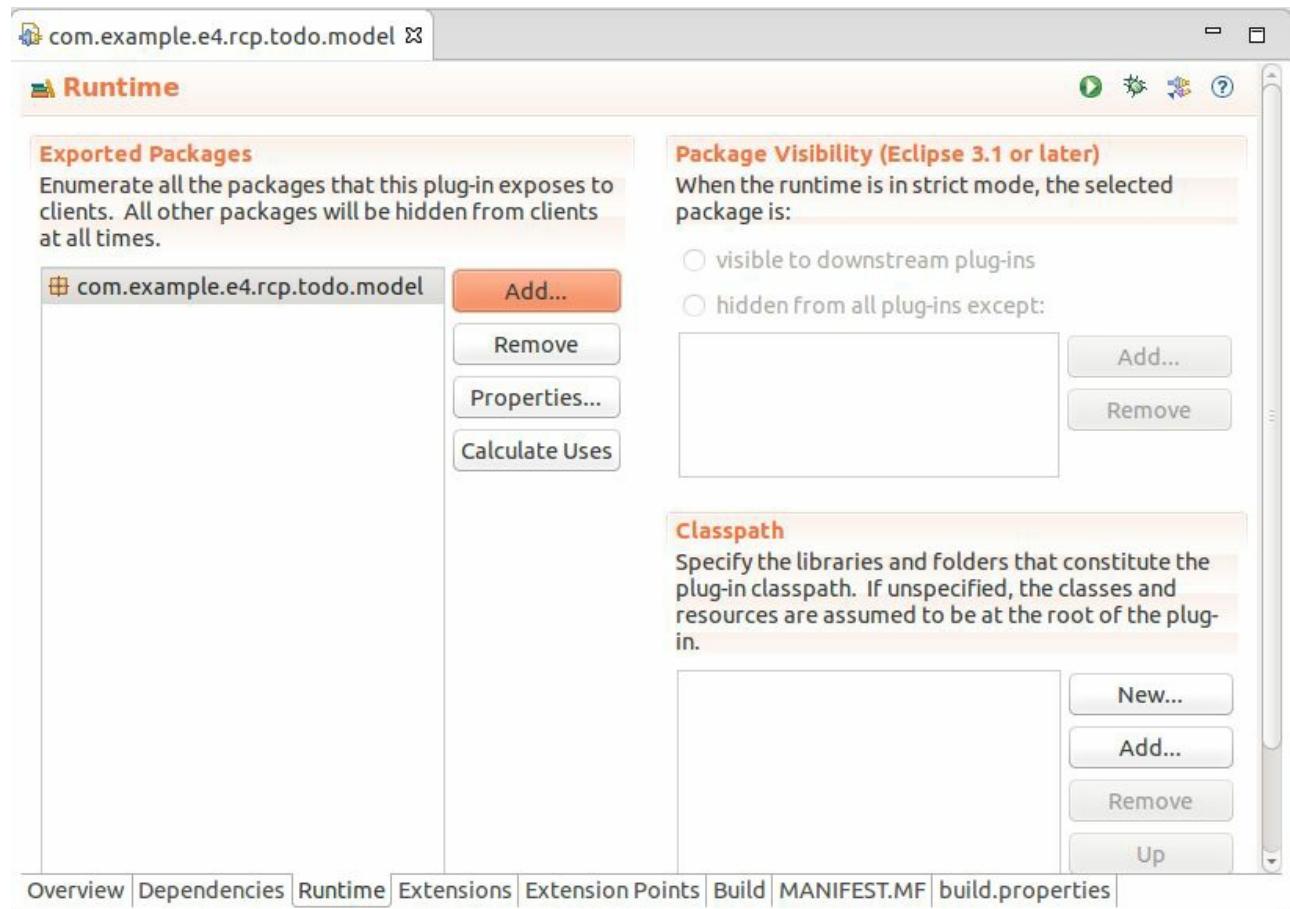
    boolean deleteTodo(long id);

    List<Todo> getTodos();
}
```

46.9. Define the API of the model plug-in

Define that the `com.example.e4.rcp.todo.model` package is released as API. This requires that you export this package.

For this open the `MANIFEST.MF` file and select the *Runtime* tab. Add `com.example.e4.rcp.todo.model` to the exported packages.



Chapter 47. Exercise: Service plug-in

47.1. Target of the exercise

In this exercise you create a plug-in for a service implementation which provides access to the data.

This service implementation uses transient data storage, i.e., the data is not persisted between application restarts. To persist the data you could extend this class to store the data for example in a database or the file system. As this storage is not special for Eclipse RCP applications, it is not covered in this book.

47.2. Create a data model provider plug-in (service plug-in)

Create a new simple plug-in (see [Section 41.5, “Naming convention: simple plug-in”](#)) project called `com.example.e4.rcp.todo.services`. This plug-in is called *todo service* plug-in in the following description.

Tip

The MacOS operating system treats folders ending with `.service` special, therefore we use the `.services` ending.

47.3. Define the dependencies in the service plug-in

Add the `com.example.e4.rcp.todo.model` plug-in as dependency to your service plug-in. To achieve this open the `MANIFEST.MF` file and select the *Dependencies* tab and add the `com.example.e4.rcp.todo.model` package to the *Imported Packages*.

47.4. Provide an implementation of the ITodoService interface

Create the `com.example.e4.rcp.todo.services.internal` package in your service plug-in and create the following class.

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

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

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;

/**
 * example service impl, data is not persisted
 * between application restarts
 */
public class MyTodoServiceImpl implements ITodoService {

    private static int current = 1;
    private List<Todo> todos;

    public MyTodoServiceImpl() {
        todos = createInitialModel();
    }

    // always return a new copy of the data
    @Override
    public List<Todo> getTodos() {
        List<Todo> list = new ArrayList<Todo>();
        for (Todo todo : todos) {
            list.add(todo.copy());
        }
        return list;
    }

    // create a new or update an existing Todo object
    @Override
    public synchronized boolean saveTodo(Todo newTodo) {
        Todo updateTodo = findById(newTodo.getId());
        if (updateTodo == null) {
            updateTodo= new Todo(current++);
            todos.add(updateTodo);
        }
        updateTodo.setSummary(newTodo.getSummary());
        updateTodo.setDescription(newTodo.getDescription());
        updateTodo.setDone(newTodo.isDone());
        updateTodo.setDueDate(newTodo.getDueDate());
    }
}
```

```

        return true;
    }

@Override
public Todo getTodo(long id) {
    Todo todo = findById(id);

    if (todo != null) {
        return todo.copy();
    }
    return null;
}

@Override
public boolean deleteTodo(long id) {
    Todo deleteTodo = findById(id);

    if (deleteTodo != null) {
        todos.remove(deleteTodo);
        return true;
    }
    return false;
}

// example data
private List<Todo> createInitialModel() {
    List<Todo> list = new ArrayList<Todo>();
    list.add(createTodo("Application model", "Flexible and extensible"));
    list.add(createTodo("DI", "@Inject as programming mode"));
    list.add(createTodo("OSGi", "Services"));
    list.add(createTodo("SWT", "Widgets"));
    list.add(createTodo("JFace", "Especially Viewers!"));
    list.add(createTodo("CSS Styling", "Style your application"));
    list.add(createTodo("Eclipse services", "Selection, model, Part"));
    list.add(createTodo("Renderer", "Different UI toolkit"));
    list.add(createTodo("Compatibility Layer", "Run Eclipse 3.x"));
    return list;
}

private Todo createTodo(String summary, String description) {
    return new Todo(current++, summary, description, false, new Date());
}

private Todo findById(long id) {
    for (Todo todo : todos) {
        if (id == todo.getId()) {
            return todo;
        }
    }
    return null;
}
}

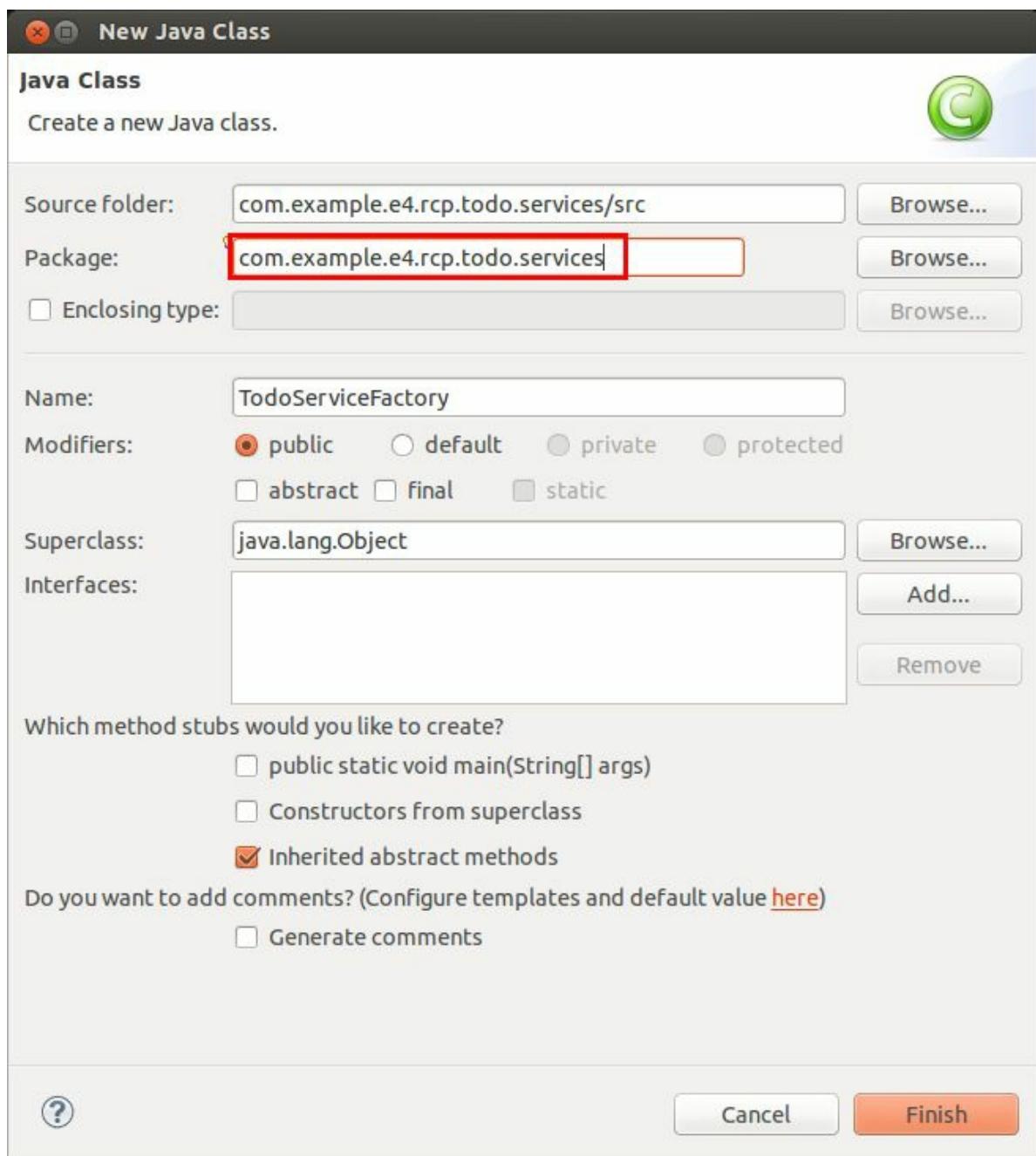
```

47.5. Create a factory

Create a new class called *TodoServiceFactory* in the `com.example.e4.rcp.todo.services` package. For this you might need the following tip.

Tip

In its default configuration the Eclipse IDE hides parent packages if they don't contain any classes. During the specification of your class you can define the correct package. This is depicted in the following screenshot.



The class provides access to your `ITodoService` implementation via a static method. It can be considered to be a *factory* for the `ITodoService` interface. A factory hides the creation of the concrete instance of a certain interface.

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

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.services.internal.MyTodoServiceImpl;

/**
 * factory provides access to the todo service provider
 */

```

```
public class TodoServiceFactory {  
  
    private static ITodoService todoService = new MyTodoServiceImpl();  
  
    public static ITodoService getInstance() {  
        return todoService;  
    }  
  
}
```

47.6. Export the package in the service plug-in

Export the `com.example.e4.rcp.todo.services` package via the `MANIFEST.MF` file on the *Runtime* tab, so that it is available for other plug-ins.

Note

Please note that the Eclipse tooling does not support the export of empty packages. You have to create at least one class in the package before you are able to export it.

Chapter 48. Exercise: Use the new plug-ins

48.1. Update the plug-in dependencies

In the *MANIFEST.MF* file of your `com.example.e4.rcp.todo` plug-in define a dependency to the two plug-ins you have created before.

48.2. Update the product configuration (via your feature)

Also add the two new plug-ins to your `com.example.e4.rcp.todo.feature` feature. Ensure that you use the *Plug-ins* tab on the `feature.xml` file.

Warning

Every time you create a new plug-in and refer to it in your `MANIFEST.MF` file you have to add it to your product configuration file (via your feature project).

48.3. Use the data model provider in your parts

In your `TodoOverviewPart` class use the `@PostConstruct` method to access the data model and to write the number of `Todo` objects to the console.

```
ITodoService todoService = TodoServiceFactory.getInstance();  
System.out.println("Number of Todo objects " +  
    todoService.getTodos().size());
```

48.4. Validate your implementation

Start your application via the product and ensure that you see the number of tasks printed to the console. If this does not work, use the following list to solve the error.

- Ensure you have saved everything (ensure that the Save All button is inactive).
- If you get compile errors trying to access your classes, ensure that:
 - you have defined all required dependencies
 - you have exported the required packages as API
- If you get an error during start up, ensure that:
 - you have added the two new plug-ins to your feature
 - you have used the product to start your application A start from the product updates the *Run Configuration* with the additional plug-ins defined in your feature

48.5. Review your implementation

Using the factory and the access via the static method makes your `TodoOverviewPart` part dependent on a concrete implementation. This makes your parts harder to test and you cannot easily change the data model implementation.

To improve your implementation, you will replace the factory with an OSGi service in [Part XI, “Using the service layer of OSGi”](#).

Chapter 49. OSGi fragments and fragment projects

49.1. What are fragments in OSGi?

A *fragment* is an optional attachment to another plug-in. This other plug-in is called the *host plug-in*. At runtime the fragment is merged with its host plug-in and for the runtime both projects are just one. Fragments are always optional for their host plug-in and the host plug-in doesn't even know that it exists.

The Eclipse IDE supports the creation of fragments via *fragment projects*. To create a fragment project select File → New → Other... → Plug-in Development → Fragment Project.

49.2. Typical use cases for fragments

Fragments can be used to contain test classes. This way the tests can access the host plug-in classes, even if the host plug-in does not define them as external API. Sometimes tests are also contained in normal plug-ins, in this case they can only test the external API of other plug-ins.

Fragments can also be used to contribute property files for additional translations. This allows contributing support for a new language without the need to adjust any other part of the code.

In addition, fragments can also supply native code which is specific to a certain system environment. A platform filter can be used to define a fragment as valid for a certain platform. The Eclipse user interface library SWT uses this approach. More information on the platform filter can be found in the Eclipse help system if you search for the following term: "OSGi Bundle Manifest Headers".

Last but not the least, fragments can be used to contain resources like icon sets or other images. This allows you to customize your application icons via the provided fragment.

Part XI. Using the service layer of OSGi

Chapter 50. OSGi service introduction

50.1. What are OSGi services?

A *service* in OSGi is defined by a standard Java class or interface. A plug-in can register new services and consume existing services via the OSGi runtime. OSGi provides a central *service registry* for this purpose.

A service can be dynamically started and stopped, and plug-ins which use services must be able to handle this dynamic behavior. The plug-ins can register listeners to be informed if a service is started or stopped.

50.2. Life cycle status for providing services

To provide a service a plug-in needs to be in the ACTIVE life cycle status of OSGi.

This requires that the service plug-in has the *Activate this plug-in when one of its classes is loaded* flag set in the manifest file.

The screenshot shows the Eclipse Marketplace interface for a plugin named "com.example.e4.rcp.todo.services". The "Overview" tab is selected. In the "Activator" section, there are two checkboxes: one checked ("Activate this plug-in when one of its classes is loaded") and one unchecked ("This plug-in is a singleton"). A red box highlights the checked checkbox.

ID:	com.example.e4.rcp.todo.services
Version:	1.0.0.qualifier
Name:	Service
Vendor:	EXAMPLE
Platform Filter:	
Activator:	<input checked="" type="checkbox"/> Browse...

Activate this plug-in when one of its classes is loaded
 This plug-in is a singleton

Overview Dependencies Runtime Build MANIFEST.MF build.pr

50.3. Best practices for defining services

It is good practice to define a service via a plug-in which only contains the interface definition. Another plug-in would provide the implementation for this service. This allows you to change the implementation of the service via a different plug-in.

50.4. Service properties

During the declaration of a service it is possible to specify key / values which can be used to configure the service.

50.5. Service priorities

It is possible to define a service ranking for a service via a service property. OSGi assigns by default a value of zero as the service ranking. The higher the ranking the better. Frameworks like the Eclipse dependency injection framework automatically inject the service with the highest service ranking.

The `Constants` class from the `org.osgi.framework` package defines the `service.ranking` value via the `Constants.SERVICE_RANKING` constant. This constant can be used to set the integer property of the service ranking.

Chapter 51. Defining OSGi services

51.1. Defining declarative services

The OSGi *declarative services* (DS) functionality allows you to define and consume services via metadata (XML) without any dependency in your source code to the OSGi framework.

The *OSGi service component* is responsible for starting the service (service component). For the service consumer it is not visible if the service has been created via declarative services or by other means.

Service components consist of an XML description (component description) and an object (component instance). The component description contains all information about the service component, e.g., the class name of the component instance and the service interface it provides. Plug-ins typically define component descriptions in a directory called *OSGI-INF*.

A reference to the component description file is entered in the *MANIFEST.MF* file via the *Service-Component* property. If the OSGi runtime finds such a reference, the *org.eclipse.equinox.ds* plug-in creates the corresponding service.

The following example *MANIFEST.MF* file demonstrates how a reference to a component definition file looks like.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Service
Bundle-SymbolicName: com.example.e4.rcp.todo.service
Bundle-Version: 1.0.0.qualifier
Bundle-Vendor: EXAMPLE
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Bundle-ActivationPolicy: lazy
Service-Component: OSGI-INF/service.xml
```

51.2. Required bundles

To use declarative services the following plug-ins must be available at runtime.

- `org.eclipse.equinox.util` - contains utility classes
- `org.eclipse.equinox.ds` - is responsible for reading the component metadata and for creating and registering the services based this information
- `org.eclipse.osgi.services` - service functionality used by declarative services

51.3. Activating the declarative service plug-in

A plug-in which provides service must be in its ACTIVE life cycle status. Therefore, ensure that the *Activate this plug-in when one of its classes is loaded* flag is set on the *MANIFEST.MF* file. See [Section 50.2, “Life cycle status for providing services”](#) for details.

Chapter 52. Steps to declare an OSGi service

52.1. Defining the service interface

The first step to define an OSGi service is to define the class or interface for which you want to provide a service. This is called the *service interface*, even though it can also be a Java class.

52.2. Providing a service implementation

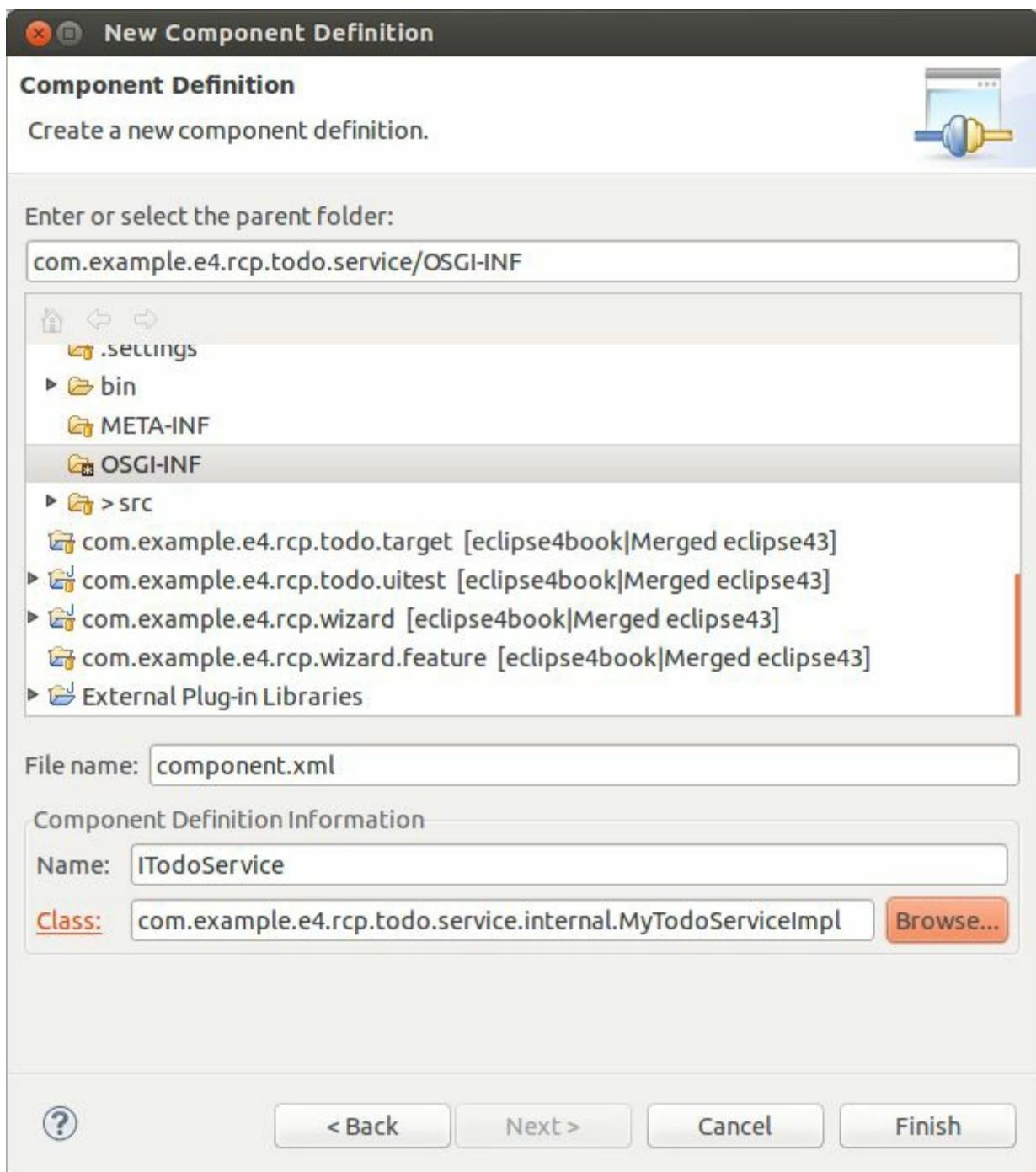
As a second step you write the implementation class for the service interface.

52.3. Service declaration with a component definition file

After you provided the implementation you need to register it for the service interface. In OSGi DS this is done via a component definition file.

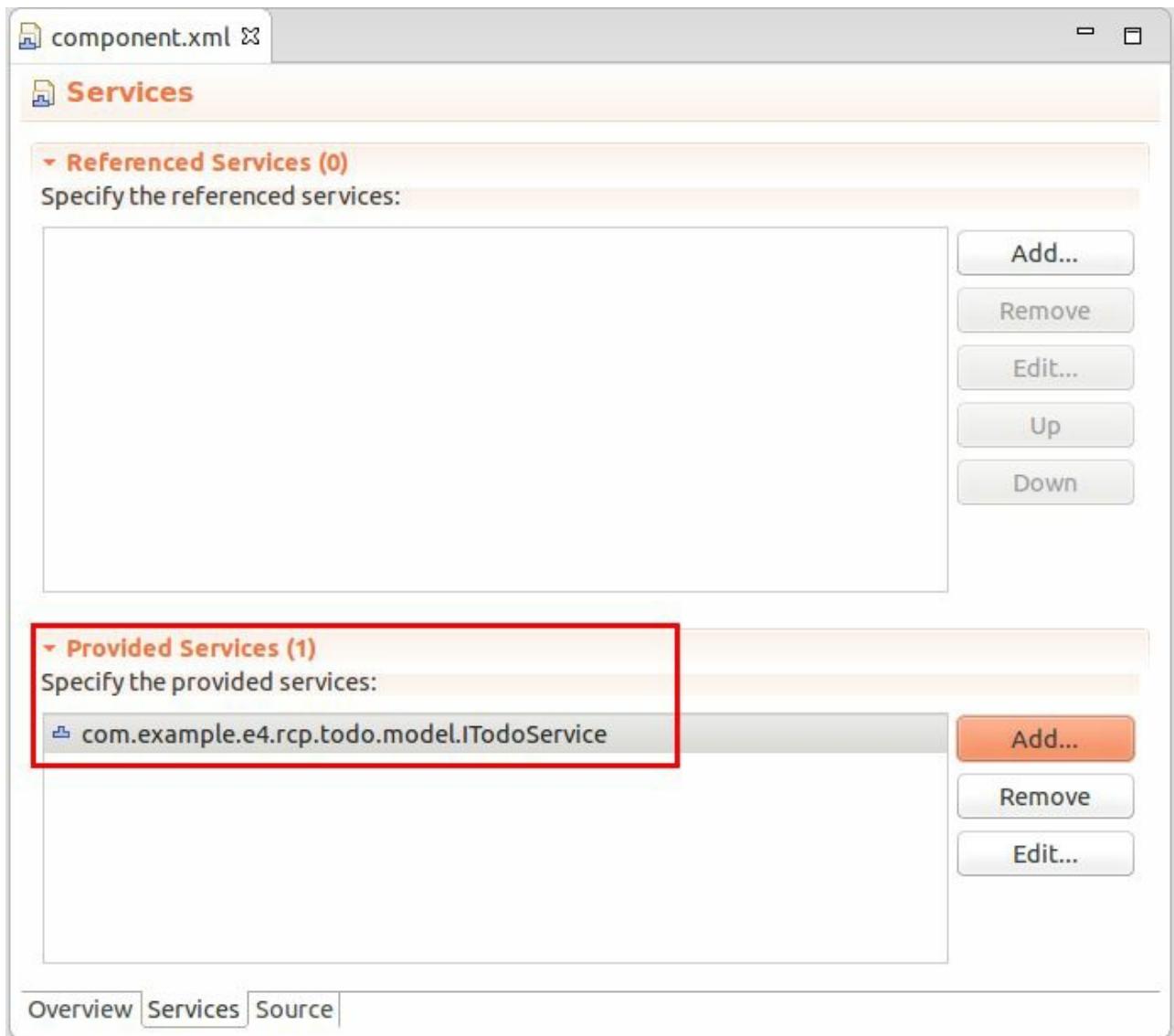
The Eclipse IDE provides a wizard for creating such files via the New → Other... → Plug-in Development → Component Definition menu entry. This wizard also adds the `Service-Component` entry to the `MANIFEST.MF` file.

On the first page of the wizard, you can enter the filename of the component definition file, a component name and the class which implements the service interface.



If you press the *Finish* button, the file is created and the corresponding editor opens.

In this editor you can specify the provided and required service on the *Services* tab. To provide a service you press the *Add...* button under *Provided Services* and select the service interface you want to provide.



For example assume that you want to provide a service for the *ITodoService* interface via the *MyTodoServiceImpl* class. A correctly maintained *component.xml* XML file would look like the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="ITodoService"
    implementation class="com.example.e4.rcp.todo.service.internal.MyTodoService">
    <service>
        <provide interface="com.example.e4.rcp.todo.model.ITodoService"/>
    </service>
</scr:component>
```

52.4. Reference to the service in the MANIFEST.MF file

After the definition of the component your *MANIFEST.MF* file contains an entry to the service component.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Service
Bundle-SymbolicName: com.example.e4.rcp.todo.service
Bundle-Version: 1.0.0.qualifier
Bundle-Vendor: EXAMPLE
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Require-Bundle: com.example.e4.rcp.todo.model;bundle-version="1.0.0"
Bundle-ActivationPolicy: lazy
Service-Component: OSGI-INF/component.xml
```

52.5. Low-level OSGi service API

OSGi also provides a low-level API for starting, stopping and tracking services. See [Section E.1, “Using the service API”](#) for a reference.

Chapter 53. Eclipse RCP and OSGi services

53.1. Using declarative services in RCP applications

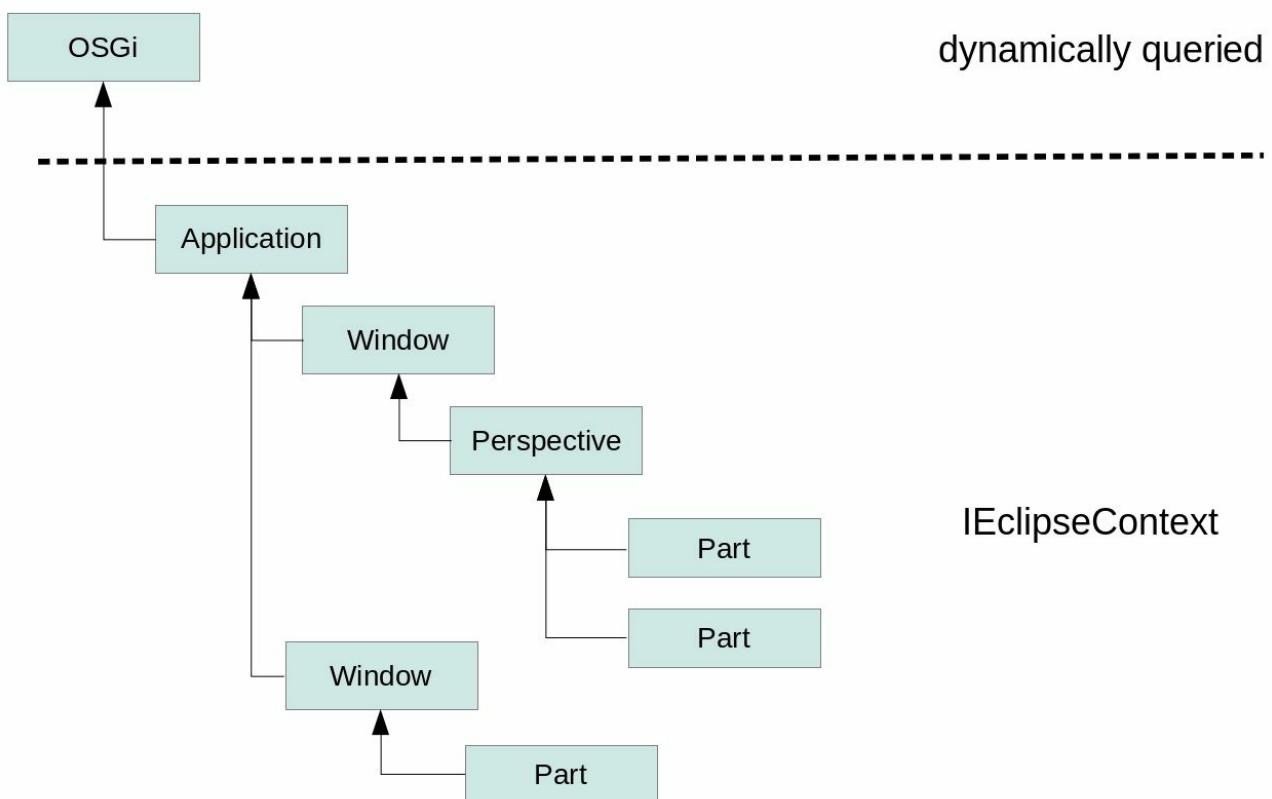
The Eclipse RCP framework automatically starts the required plug-ins for using declarative OSGi service. It is not required to manually set a start level in your product configuration file.

A plug-in which provides a service must be activated. See [Section 51.3, “Activating the declarative service plug-in”](#).

53.2. OSGi services and Eclipse dependency injection

OSGi services are available for dependency injection in Eclipse applications. If you define your custom OSGi services, you can inject them into your model objects. This removes the need to create singleton or factory implementations in your application to access data.

If a requested key is not found in the Eclipse context hierarchy, the Eclipse framework dynamically queries for a fitting OSGi service in the OSGi registry.



For example, if you have an OSGi service declared for the `ITodoService` interface you can inject it via the following code snippet into a field of an Eclipse part.

```
// optional in case the service is deregistered  
@Inject @Optional ITodoService todoService;
```

Alternatively you can inject the service into a method of such a Eclipse part as demonstrated in the following listings.

```
@Inject ITodoService todoService;/  
@Inject
```

```
public void setTodoService(@Optional ITodoService todoService) {  
    this.todoService = todoService;  
    // TODO reload data  
}
```

Chapter 54. Exercise: Define and use an OSGi service

54.1. Target of this exercise

In this exercise you use the OSGi declarative services functionality to provide a service. The provided services is for the `ITodoService` interface and is implemented by the existing `MyTodoServiceImpl` class.

Tip

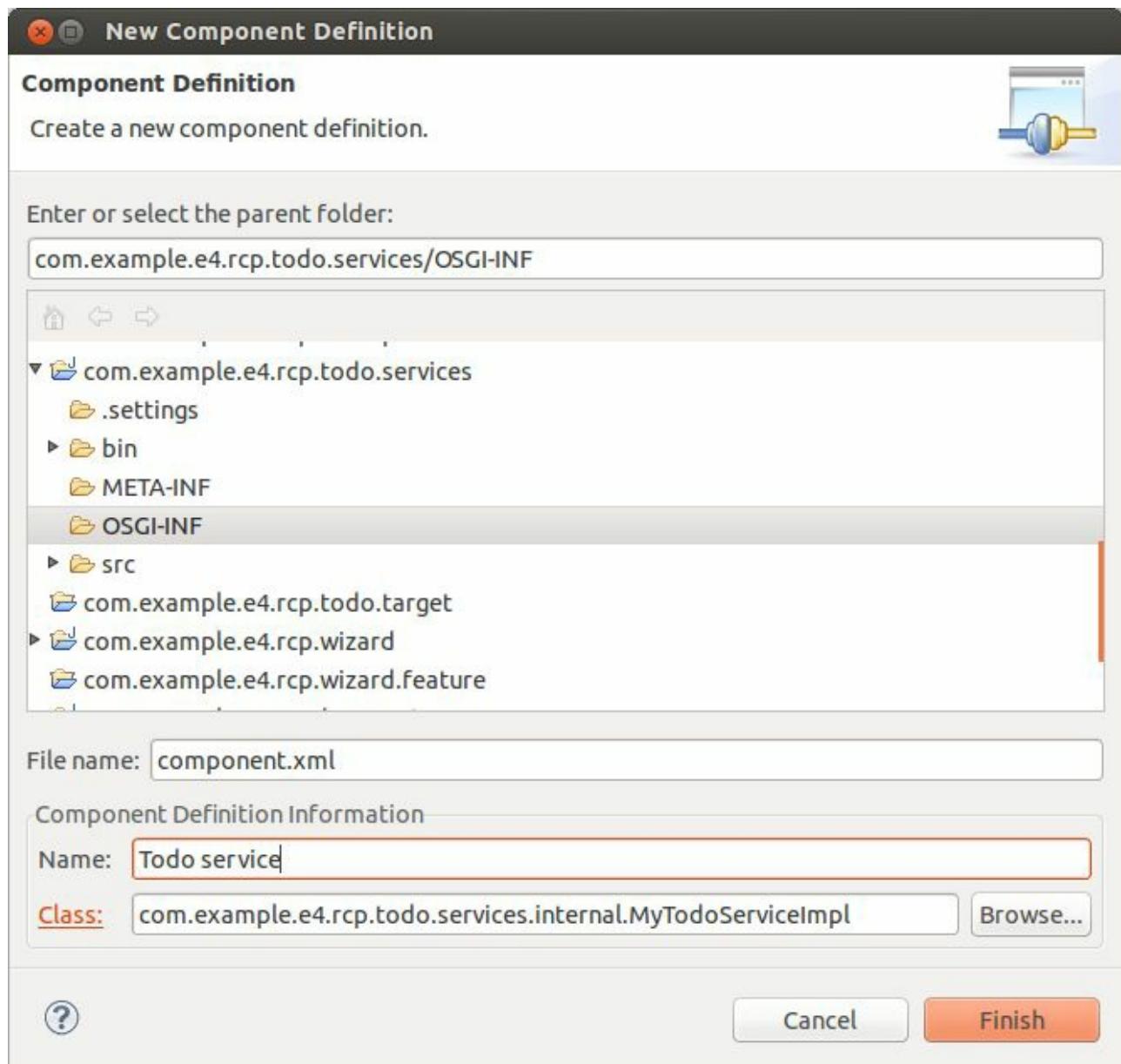
Your product is based on the `org.eclipse.e4.rcp` feature. This feature includes the required plug-ins to use OSGI declarative services already.

54.2. Define the component definition file

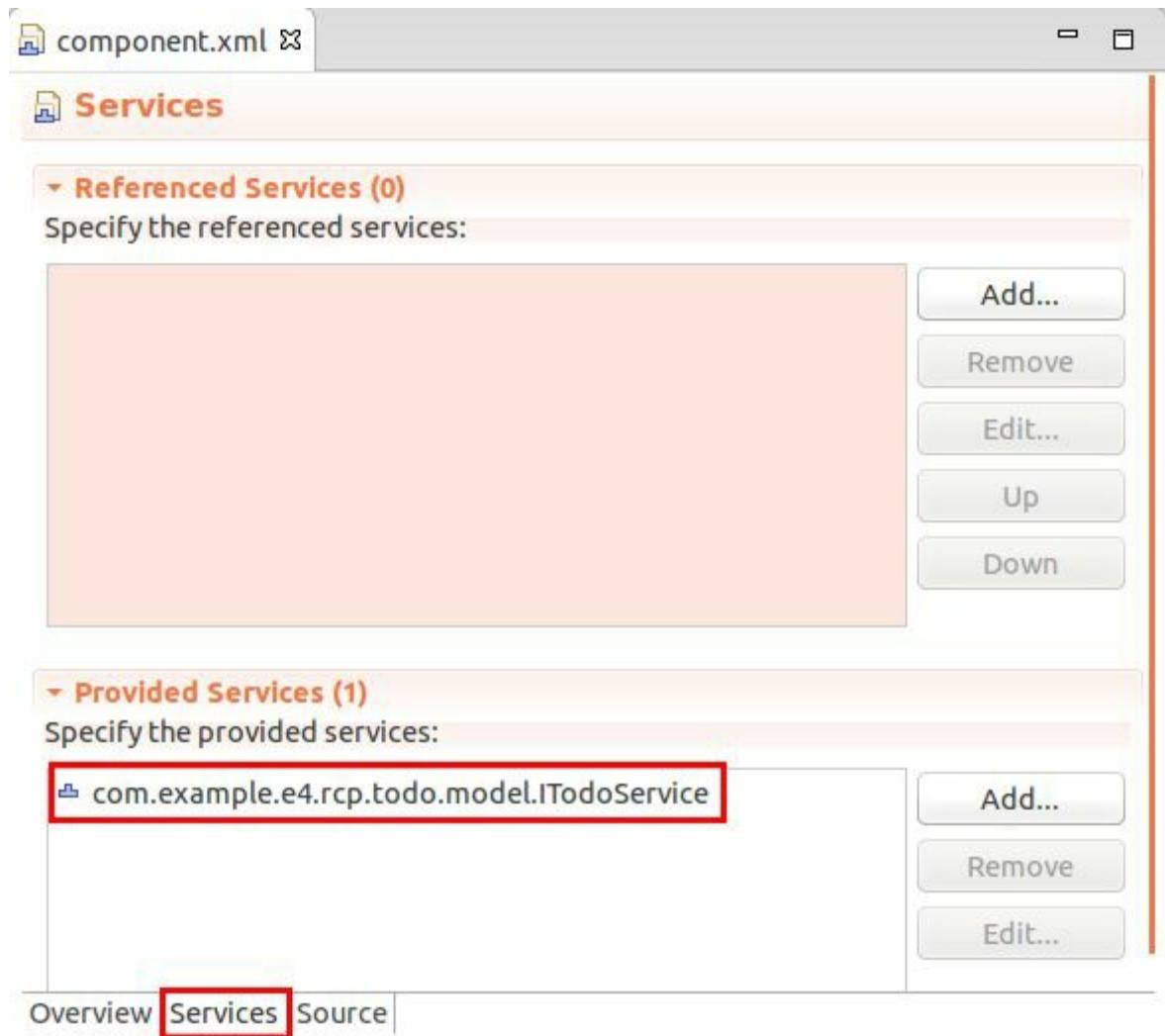
Create a new folder called `OSGI-INF` in your service plug-in. Do this via a right-click on your `com.example.e4.rcp.todo.services` project and by selecting New → Folder.

Select the `OSGI-INF` folder, right-click on it and select New → Other... → Plug-in Development → Component Definition.

In this wizard use the `MyTodoServiceImpl` class as the class which provides the service implementation and press the *Finish* button.



In the generated file, select the *Services* tab and specify that you provide a service for the `ITodoService` interface as depicted in the following screenshot.



Warning

Ensure that you provide the service for `ITodoService` and not for your `MyTodoServiceImpl` implementation class.

The wizard creates a reference to the `component.xml` file in the `MANIFEST.MF` file under the Service-Component reference.

Warning

If you move or rename the `component.xml` file, you need to adjust this reference. It should look like the following.

```
Service-Component: OSGI-INF/component.xml
```

The `component.xml` should look like the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="Todo servi
<implementation
  class="com.example.e4.rcp.todo.services.internal.MyTodoServiceImpl"/>
<service>
  <provide interface="com.example.e4.rcp.todo.model.ITodoService"/>
</service>
</scr:component>
```

54.3. Set the lazy activation flag for the service plug-in

Ensure that the *Activate this plug-in when one of its classes is loaded* flag for the service plug-in is set in the `MANIFEST.MF` file. Depending on your setup, this flag might already be set.

The screenshot shows the 'Overview' tab of the PDE interface for the plugin `com.example.e4.rcp.todo.services`. The 'General Information' section contains fields for ID, Version, Name, Vendor, Platform Filter, Activator, and two checkboxes: 'Activate this plug-in when one of its classes is loaded' (which is checked and highlighted with a red box) and 'This plug-in is a singleton'. The 'Plug-in Content' section describes the two main parts of the plugin: Dependencies and Runtime. The 'Extension / Extension Point Content' section lists Extensions and Extension Points. The 'Testing' section is partially visible at the bottom. Navigation tabs at the bottom include Overview, Dependencies, Runtime, Build, MANIFEST.MF, and build.properties.

54.4. Get the ITodoService injected

Change your `TodoOverviewPart` class so that the `ITodoService` service implementation is injected into it by using your `@PostConstruct` method. Print the number of todos on the console as demonstrated in the following example.

```
@PostConstruct  
public void createControls(Composite parent, ITodoService todoService) {  
    System.out.println("Number of Todo objects " +  
        todoService.getTodos().size());  
}
```

54.5. Possible issues: ITodoService cannot get injected

If the `ITodoService` object cannot be injected ensure that the *Activate this plug-in when one of its classes is loaded* flag for the plug-in has been set in the `MANIFEST.MF` file of your service plug-in.

Also ensure that the `com.example.e4.rcp.todo.services` and `com.example.e4.rcp.todo.model` plug-ins are part of your feature project and that you started the application via the product file.

54.6. Clean-up the service implementation

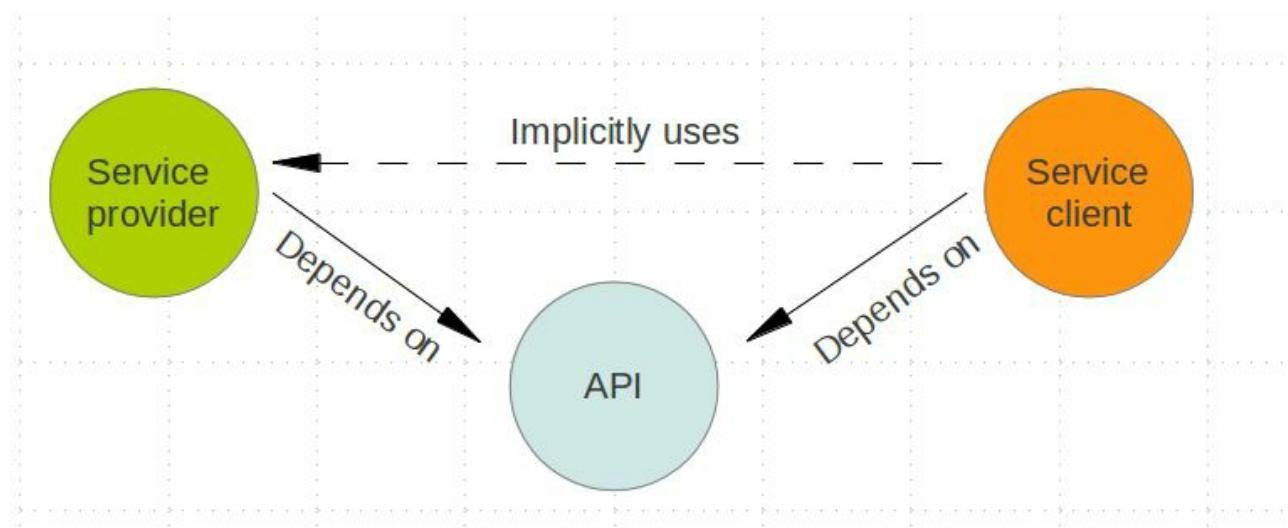
Remove any access to the `TodoServiceFactory` from the `com.example.e4.rcp.todo.services` plug-in. This access is not required anymore, as the `ITodoService` implementation can now be injected.

Also ensure that the `com.example.e4.rcp.todo.services` plug-in does not export any packages. If you still use this class you get compile errors. Fix these errors by using dependency injection to access the implementation of `ITodoService`.

Afterwards delete the `TodoServiceFactory` class.

54.7. Review the service implementation

With this setup you have completely hidden the concrete implementation of the service from the usage. This design is depicted in the following screenshot.



The current service implementation does not persist the data, as it is just a test service. You could replace it with another service implementation without changing your source code or the dependencies in the consuming plug-ins.

Note

The only change required to use another service implementation in your application is to exchange the corresponding service plug-in in your product configuration file (via your feature).

Chapter 55. Optional exercise: Create an image loader service

55.1. Target of this exercise

In this exercise you create a service for your application which allows you to load images. We use the same plug-in (bundle) for the interface definition and the service definition. This is a valid setup if you plan to have only one service implementation.

55.2. Creating a new plug-in

Create a new simple plug-in called `com.example.e4.bundleresourceloader`.

Add the following plug-ins as dependencies to your new plug-in.

- `org.eclipse.core.runtime`
- `org.eclipse.jface`

Note

The `org.eclipse.jface` plug-in re-exports the `org.eclipse.swt` hence if you define a dependency to `*.jface` you can also use the classes from `*.swt`.

Create the following interface called `IBundleResourceLoader`.

```
package com.example.e4.bundleresourceloader;

import org.eclipse.jface.resource.ImageDescriptor;

public interface IBundleResourceLoader {
    public ImageDescriptor getImageDescriptor(Class<?> clazz, String path);
}
```

55.3. Creating a service implementation

Create the following class which allows, via its `getImageDescriptor()` method, to load an `ImageDescriptor` from a plug-in based on the class provided as parameter.

```
package com.example.e4.bundleresourceloader.internal;

import java.net.URL;

import org.eclipse.core.runtime.FileLocator;
import org.eclipse.core.runtime.Path;
import org.eclipse.jface.resource.ImageDescriptor;
import org.osgi.framework.Bundle;
import org.osgi.framework.FrameworkUtil;

import com.example.e4.bundleresourceloader.IBundleResourceLoader;

public class BundleResourceLoaderImpl implements IBundleResourceLoader {

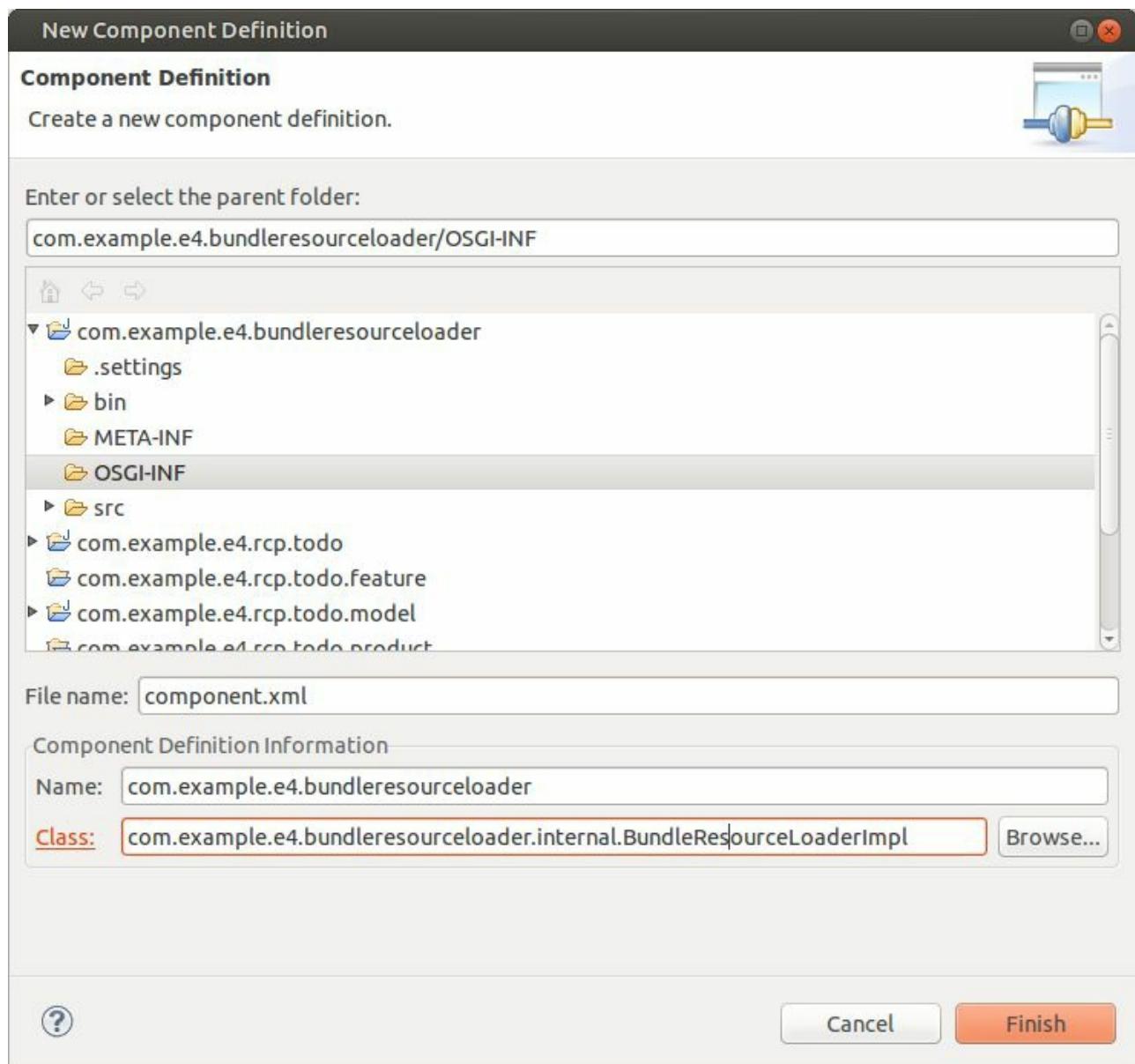
    // service does NOT recycle the provided image
    // consumer of the image is responsible to call the dispose()
    // method on the created Image object

    @Override
    public ImageDescriptor getImageDescriptor(Class<?> clazz, String path) {
        Bundle bundle = FrameworkUtil.getBundle(clazz);
        URL url = FileLocator.find(bundle, new Path(path), null);
        return ImageDescriptor.createFromURL(url);
    }
}
```

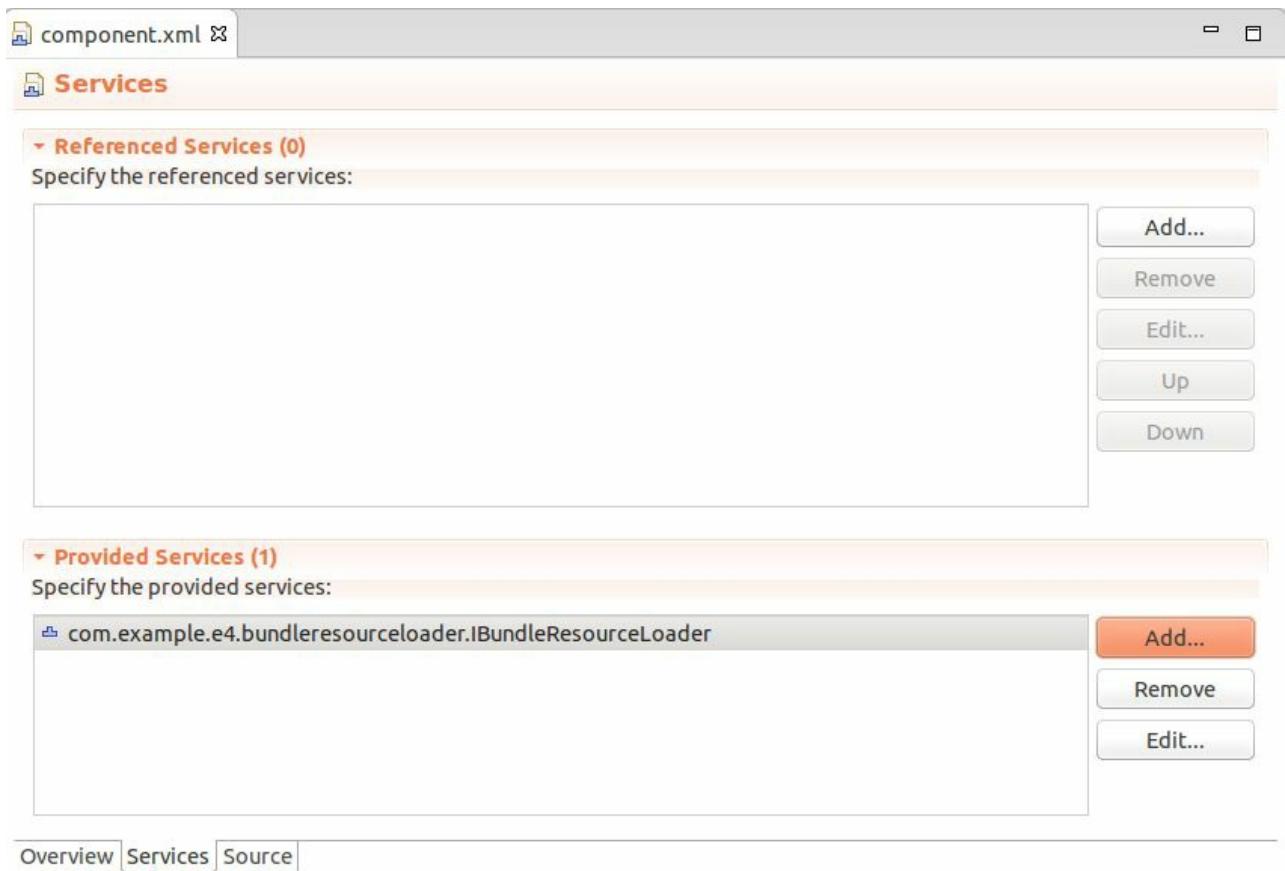
55.4. Defining a new service

Define the implementation class as service in the `*.bundleresourceloader` plug-in similar to [Section 54.2, “Define the component definition file”](#).

The following shows a screenshot of the wizard used to create a service component definition file.



Ensure that this component definition provides the `IBundleResourceLoader` interface as service.



Also ensure that *Activate this plug-in when one of its classes is loaded* flag is set on the `MANIFEST.MF` file of the `*.bundleresourceloader` plug-in.

55.5. Adding the new plug-in to your feature project

Add the new plug-in to your product configuration file via your feature project.

55.6. Exporting the API

To use the service in other plug-ins, export the package which contains the interface in the corresponding *MANIFEST.MF* file.

55.7. Add a MANIFEST.MF dependency to your new plug-in

Define a dependency to this package (or the whole plug-in) in the *MANIFEST.MF* file of your application plug-in.

55.8. Using the new service

Create a directory called *images* in your application plug-in and put an image with the file name *vogella.png* into it. A Google search for *vogella.png* should find such an image, but you can use any *.png file you like.

Use the image loader service in your `PlaygroundPart` class. You can assign an image to an SWT Label via the `setImage(image)` method. Use your service to get an `ImageDescriptor` and create an image with a `LocalResourceManager` and assign it to a `Label` in your user interface. The following code demonstrates the usage of your service based on the path name from above.

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

import javax.annotation.PostConstruct;

import org.eclipse.jface.resource.LocalResourceManager;
import org.eclipse.jface.resource.ResourceManager;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;

import com.example.e4.bundleresourceloader.IBundleResourceLoader;

public class PlaygroundPart {
    @PostConstruct
    public void createControls(Composite parent, IBundleResourceLoader loader) {
        Label label = new Label(parent, SWT.NONE);
        // the following code assumes that you have a vogella.png file
        // in a folder called "images" in this plug-in
        ResourceManager resourceManager =
            new LocalResourceManager(JFaceResources.getResources(), label);
        Image image =
            resourceManager.
            createImage(loader.loadImage(this.getClass(), "images/vogella.png"));
        label.setImage(image);
    }
}
```

The result might look like the following.



Warning

As you added plug-ins to your product via the feature, remember to start via the product configuration file. Using the existing run configuration does not work.

55.9. Reviewing the implementation

In this example the service interface and the implementation class is contained in one plug-in. Therefore, this plug-in needs to export the interface and consuming plug-ins must define a dependency to this service plug-in. In the first service we separated the implementation and the service interface as this allows you to replace the service easily.

Both approaches are valid. You typically put the interface and the implementation in one plug-in, if you do not expect an alternative implementation of the service interface.

Note

In general you should always use a JFace ResourceManager in order to avoid problems with system resources or even a "No more handles" SWTException.

Part XII. User interface development with SWT

Chapter 56. Standard Widget Toolkit

56.1. What is SWT?

The Standard Widget Toolkit (*SWT*) is the default user interface library used by Eclipse. It provides widgets, e.g., buttons and text fields, as well as layout managers. Layout managers are used to arrange the widgets according to a certain rule set.

SWT supports several platforms, e.g., Windows, Linux and Mac OS X. The design target of SWT is to stay closely to the operating system (OS) and the SWT API (Application Programming Interface) is very close to the native API of the OS.

SWT uses the native widgets of the platform whenever possible. The native widgets of the OS are accessed by the SWT framework via the *Java Native Interface* (JNI) framework. JNI is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call, and to be called by, native applications and libraries written in other languages such as C, C++ and assembler.

The approach of using native widgets can also be found in *AWT*, a standard user interface library available in Java. But SWT provides more widgets than AWT, e.g., tree and table widgets. In case a widget is not natively available on one platform, SWT emulates this widget on this platform.

56.2. Eclipse applications and SWT

Eclipse applications typically use SWT for the user interface. If you develop Eclipse plug-ins which extend the Eclipse IDE itself, you have to use SWT as the Eclipse IDE Workbench uses an SWT renderer.

For RCP applications it is possible to use other user interface toolkits than SWT, like JavaFX. This is possible because the Eclipse platform provides a flexible rendering framework which allows you to replace the user interface toolkit.

56.3. Display and Shell

The `Display` and `Shell` classes are key components of SWT applications.

A `org.eclipse.swt.widgets.Shell` class represents a window.

The `org.eclipse.swt.widgets.Display` class is responsible for managing event loops, fonts, colors and for controlling the communication between the user interface thread and other threads. `Display` is therefore the base for all SWT capabilities.

Every SWT application requires at least one `Display` and one or more `Shell` instances. The main `Shell` gets, as a default parameter, a `Display` as a constructor argument. Each `Shell` is constructed with a `Display` and if none is provided during construction it will use either the `Display` which is currently used or a default one.

56.4. Event loop

An event loop is needed to transfer user input events from the underlying native operating system widgets to the SWT event system.

SWT does not provide its own event loop. This means that the programmer explicitly starts and checks the event loop to update the user interface. The loop executes the `readAndDispatch()` method which reads events from the native OS event queue and dispatches them to the SWT event system. The loop is executed until the main shell is closed. If this loop would be left out, the application would terminate immediately.

For example the following creates a SWT application which creates and executes the event loop.

```
Display display = new Display();
Shell shell = new Shell(display);
shell.open();

// run the event loop as long as the window is open
while (!shell.isDisposed()) {
    // read the next OS event queue and transfer it to a SWT event
    if (!display.readAndDispatch())
    {
        // if there are currently no other OS event to process
        // sleep until the next OS event is available
        display.sleep();
    }
}

// disposes all associated windows and their components
display.dispose();
```

If SWT is used in an Eclipse plug-in or an Eclipse RCP application, this event loop is provided by the Eclipse framework.

56.5. Relationship to JFace

JFace is a set of APIs which builds on top of SWT and provides higher level abstraction APIs and commonly used functions.

JFace extends the SWT API for certain uses case but does not hide it. Therefore even if you frequently work with JFace you need a solid understanding of SWT.

56.6. Using SWT in a plug-in project

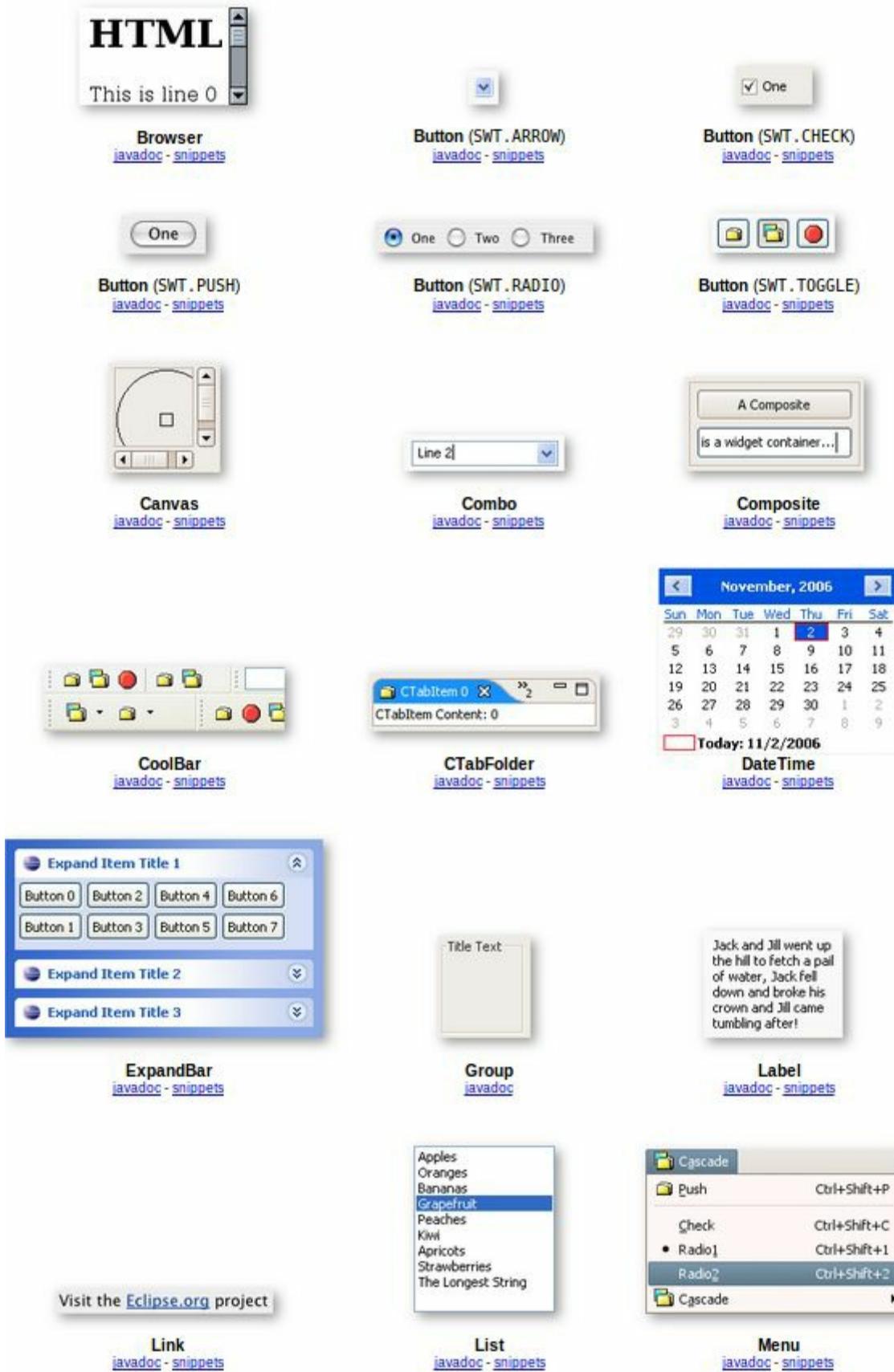
The SWT library is packaged as an Eclipse plug-in. If you create an Eclipse plug-in and want to use SWT you have to specify a dependency to the `org.eclipse.swt` plug-in in the corresponding manifest file.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: My SWT-based Plug-in
Bundle-SymbolicName: de.vogella.swt
Bundle-Version: 1.0.0.qualifier
Require-Bundle: org.eclipse.swt
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

Chapter 57. Using SWT widgets

57.1. Available widgets in the SWT library

SWT widgets are located in the packages `org.eclipse.swt.widgets` and `org.eclipse.swt.custom`. Widgets extend either the `Widget` or the `Control` class. Several of these widgets are depicted in the following graphic. This graphic is a screenshot of the [SWT widget homepage](#).



While SWT tries to use native widgets as much as possible, it can not fulfill all

common requirements with the native widgets. Therefore some widgets extend the capabilities of the native platform. These are part of the `org.eclipse.swt.custom` package and usually start with the additional prefix `C` to indicate that they are custom widgets, e.g. `CCombo`.

Compared to the `Combo` class, the `CCombo` class provides the ability to set the height of the widget.

Another example is `StyledText`, a class which provides advanced features for displaying text, e.g. drawing a background.

Widgets from the package `org.eclipse.swt.custom` are implemented in pure Java while widgets from `org.eclipse.swt.widgets` use native code. Custom widgets are not supposed to use the internal classes of SWT as these classes may be different on the various platforms. Every custom widget must extend the `Composite` or `Canvas` class. API compliance is guaranteed only for these base classes.

If the new custom widget is supposed to contain other widgets, it should extend `Composite`. Otherwise it should extend the `Canvas` class.

In this book the words "widget" and "control" are used interchangeable for user interface elements.

57.2. Memory management

SWT widgets are not automatically garbage collected. If you release an SWT widget, you have to call its `dispose()` method. Fortunately if a container is disposed, e.g., a `Shell`, this container also releases all its children.

The automatic release does not work for Color, Cursor, Display, Font, GC, Image, Printer, Region, Widget and subclasses. All of these SWT objects need to be manually disposed. JFace provides a simplification for this via its `LocalResourceManager`.

57.3. Constructing widgets

SWT widgets, except the `Shell` object, are always constructed with a parent widget which contains them. This is similar to AWT and different to Swing, where the `add()` method is used.

The second parameter of the widget constructor contains the *stylebits*. Depending on the provided stylebits the widget adjusts its look and feel as well as its behavior. Each widget documents the supported stylebits in its Javadoc.

The possible stylebits are predefined in the `SWT` class. If no special style is required you can pass `SWT.NONE`.

For example the following code snippet creates a push button.

```
new Button(shell, SWT.PUSH);
```

The following example creates a checkbox button. The only difference is the usage of another stylebit.

```
new Button(shell, SWT.CHECK);
```

57.4. Basic containers

The `Composite` class is a container which is capable of containing other widgets. The `Group` class is another container which is able to contain other widgets but it additionally draws a border around itself and allows you to set a header for the grouped widgets.

57.5. Event Listener

You can register listeners for specific events on SWT controls, e.g., a `ModifyListener` to listen to changes in a `Text` widget or a `SelectionListener` for selection (click) events on a `Button` widget. The following code demonstrates the implementation, it uses `SelectionAdapter` which is an implementation of the `SelectionListener` interface.

```
Button button = new Button(shell, SWT.PUSH);

//register listener for the selection event
button.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
        System.out.println("Called!");
    }
});
```

`FocusListener` is another examples for such a listener interface.

Tip

The Listener interfaces sometimes contain several methods and you only want to implement one of them. For this purpose Eclipse provides empty default implementations for these interfaces. This allows you to implement only the methods you are interested in. These implementations follow the naming convention:

`Name Listener` → `Name Adapter`

For example `SelectionListener` has the abstract class `SelectionAdapter` which pre-implements the methods of `SelectionListener`.

If you want to add a listener to the whole application, you can use the `Display` class. For example to add a global mouse listener use `addFilter(SWT.MouseMove, listener)`. If you add filters to the `Display` you may interfere with existing listeners. Ensure to test your code accordingly.

Chapter 58. Using layout managers in SWT

58.1. The role of a layout manager

A layout manager is responsible for arranging the user interface components of a container, e.g., a `Composite`, on the screen. SWT offers several standard layout managers. The following table gives an overview of them. It is sorted by complexity of the layout manager, i.e. the simplest layout manager is listed first and the most complex one as the last entry.

Table 58.1. Layout Manager

Layout Manager	Description
AbsoluteLayout	Allows you to specify the exact position, the width and the height of components. As user interfaces may be used on screens with different sizes this layout manager should be avoided.
FillLayout	Arranges equal-sized widgets in a single row or column.
RowLayout	Puts the widgets in rows or columns and allows you to control the layout with options, e.g., wrap, spacing, fill and so on.
GridLayout	Arranges widgets in a grid.
FormLayout	Arranges the widgets with the help of the associated attachments.

58.2. Layout Data

Each SWT widget can have a layout specific settings class assigned to it, e.g. `GridData` for a `GridLayout`. This allows the developer to control the arrangement of the widgets within the layout.

In the following example you specify that a certain widget should take two columns in a `GridLayout`.

```
button = new Button(parent, SWT.PUSH);  
GridData gridData = new GridData();  
gridData.horizontalSpan = 2;  
button.setLayoutData(gridData);
```

Warning

The used layout data must match the layout manager, otherwise an exception is thrown at runtime.

The layout will be automatically calculated when the container is displayed. You can tell a `Composite` to recalculate the layout with the `composite.layout()` method.

Warning

Layout data objects should not be reused as the layout manager expects that every user interface element has a unique layout data object.

58.3. FillLayout

FillLayout divides the available space provided by the container equally to all widgets and can be set to arrange the widgets either horizontally (SWT.HORIZONTAL) or vertically (SWT.VERTICAL). It also allows you to set the space between the widgets (attribute `spacing`) and the margins of the widgets to the container via the `marginWidth` and `marginHeight` attributes.

58.4. RowLayout

RowLayout orders UI components in a row (`SWT.HORIZONTAL`) or in a column (`SWT.VERTICAL`). RowLayout supports wrapping of fields (field `wrap`) by default. You can define if widgets should have their preferred size (default) or if they should grab the available space via the field `pack`. It is also possible to set margins at the top, bottom, left and right. If you set `justify`, the widgets will be spread through the available space.

Each element can define its height and width via a `RowData` element.

58.5. GridLayout

GridLayout allows you to arrange the user interface components in a Grid with a certain number of columns. It is also possible to specify column and row spanning.

You can use `new GridData()` and assign properties to the new object. Alternatively you can use one of its richer constructors to define certain attributes during construction. For example via the following constructor.

```
new GridData(horizontalAlignment,  
             verticalAlignment,  
             grabExcessHorizontalSpace,  
             grabExcessVerticalSpace,  
             horizontalSpan,  
             verticalSpan)
```

The most important attributes are defined in the following table.

Table 58.2. GridData

Parameter	Description
horizontalAlignment	Defines how the control is positioned horizontally within a cell (one of: <code>SWT.LEFT</code> , <code>SWT.CENTER</code> , <code>SWT.RIGHT</code> , or <code>SWT.FILL</code>).
verticalAlignment	Defines how the control is positioned vertically within a cell (one of: <code>SWT.TOP</code> , <code>SWT.CENTER</code> , <code>SWT.END</code> , <code>SWT.BOTTOM</code> (treated the same as <code>SWT.END</code>), or <code>SWT.FILL</code>).
grabExcessHorizontalSpace	Defines whether the control is extended by the layout manager to take all the remaining horizontal space.
grabExcessVerticalSpace	Defines whether the control grabs any remaining vertical space.
horizontalSpan	Defines the number of column cells that the control will take up.
verticalSpan	Defines the number of row cells that the control will take up.
heightHint	Defines the preferred height in pixels.
widthHint	Defines the preferred width in pixels.

If the widget has the `grabExcessHorizontalSpace` attribute set to true, it will grab available space in its container. `SWT.FILL` tells the widget to fill the available space. Therefore, `grabExcessHorizontalSpace` and `SWT.FILL` are often used together.

Tip

The `GridDataFactory` class provides static methods for creating `GridData` objects. The Javadoc of this class contains several examples for it.

58.6. Using GridDataFactory

The `GridDataFactory` class can be used to create `GridData` objects. This class provides a convenient shorthand for creating and initializing `GridData`. The following listing demonstrates its usage and compares it with the direct usage of `GridData`.

```
// listBox is an SWT widget

// GridDataFactory version
GridDataFactory.fillDefaults().grab(true, true).hint(150, 150).applyTo(listBox);

// Equivalent SWT version
GridData listBoxData = new GridData(GridData.FILL_BOTH);
listBoxData.widthHint = 150;
listBoxData.heightHint = 150;
listBoxData.minimumWidth = 1;
listBoxData.minimumHeight = 1;
listBox.setLayoutData(listBoxData);
```

Unfortunately the *SWT Designer* does currently not support `GridDataFactory`, hence the following examples avoid using them.

58.7. Tab order of elements

You can specify the tab order of controls via the `setTabList()` method of a `Composite`. For this you provide an array of the controls where the order of the controls in the array specify the tab order.

```
package com.example.swt.widgets

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class TabExample {
    public static void main(String[] args) {
        Display display = Display.getDefault();
        Shell shell = new Shell(display);
        shell.setLayout(new RowLayout());
        Button b1 = new Button(shell, SWT.PUSH);
        b1.setText("Button1");
        Button b2 = new Button(shell, SWT.PUSH);
        b2.setText("Button2");
        Button b3 = new Button(shell, SWT.PUSH);
        b3.setText("Button3");

        Control[] controls = new Control[] { b2, b1, b3 };
        shell.setTabList(controls);
        shell.pack();
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }
        display.dispose();
    }
}
```

Tip

Defining a tab order for your controls is important to increase the accessibility and to allow advanced users a quick navigation.

58.8. Example: Using layout manager

The following shows an example for the usage of the `GridLayout` class in the `com.example.swt.widgets` project.

```
package com.example.swt.widgets.layouts;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Group;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Spinner;
import org.eclipse.swt.widgets.Text;

public class GridLayoutSWT {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);

        // create a new GridLayout with two columns
        // of different size
        GridLayout layout = new GridLayout(2, false);

        // set the layout to the shell
        shell.setLayout(layout);

        // create a label and a button
        Label label = new Label(shell, SWT.NONE);
        label.setText("A label");
        Button button = new Button(shell, SWT.PUSH);
        button.setText("Press Me");

        // create a new label that will span two columns
        label = new Label(shell, SWT.BORDER);
        label.setText("This is a label");
        // create new layout data
        GridData data = new GridData(SWT.FILL, SWT.TOP, true, false, 2, 1);
        label.setLayoutData(data);

        // create a new label which is used as a separator
        label = new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL);

        // create new layout data
        data = new GridData(SWT.FILL, SWT.TOP, true, false);
        data.horizontalSpan = 2;
        label.setLayoutData(data);
```

```

// create a right aligned button
Button b = new Button(shell, SWT.PUSH);
b.setText("New Button");

data = new GridData(SWT.LEFT, SWT.TOP, false, false, 2, 1);
b.setLayoutData(data);

// create a spinner with min value 0 and max value 1000
Spinner spinner = new Spinner(shell, SWT.READ_ONLY);
spinner.setMinimum(0);
spinner.setMaximum(1000);
spinner.setSelection(500);
spinner.setIncrement(1);
spinner.setPageIncrement(100);
GridData gridData = new GridData(SWT.FILL, SWT.FILL, true, false);
gridData.widthHint = SWT.DEFAULT;
gridData.heightHint = SWT.DEFAULT;
gridData.horizontalSpan = 2;
spinner.setLayoutData(gridData);

Composite composite = new Composite(shell, SWT.BORDER);
gridData = new GridData(SWT.FILL, SWT.FILL, true, false);
gridData.horizontalSpan = 2;
composite.setLayoutData(gridData);
composite.setLayout(new GridLayout(1, false));

Text txtTest = new Text(composite, SWT.NONE);
txtTest.setText("Testing");
gridData = new GridData(SWT.FILL, SWT.FILL, true, false);
txtTest.setLayoutData(gridData);

Text txtMoreTests = new Text(composite, SWT.NONE);
txtMoreTests.setText("Another test");

Group group = new Group(shell, SWT.NONE);
group.setText("This is my group");
gridData = new GridData(SWT.FILL, SWT.FILL, true, false);
gridData.horizontalSpan = 2;
group.setLayoutData(gridData);
group.setLayout(new RowLayout(SWT.VERTICAL));
Text txtAnotherTest = new Text(group, SWT.NONE);
txtAnotherTest.setText("Another test");

shell.pack();
shell.open();
while (!shell.isDisposed()) {
    if (!display.readAndDispatch()) {
        display.sleep();
    }
}
display.dispose();
}

```

}

Start your application, it should look similar to the following screenshot.



Resize the window and see how the arrangement of the widgets change.

Chapter 59. SWT widget examples and controls

59.1. SWT snippets and examples

The *SWT Snippets* are examples for stand-alone SWT applications using different kinds of SWT widgets. The SWT snippet site is located at the following URL: <http://www.eclipse.org/swt/snippets/>.

You can copy these snippets and paste them directly into a Java package inside of the Eclipse IDE. Eclipse automatically creates the Java class based on the content of the clipboard for you.

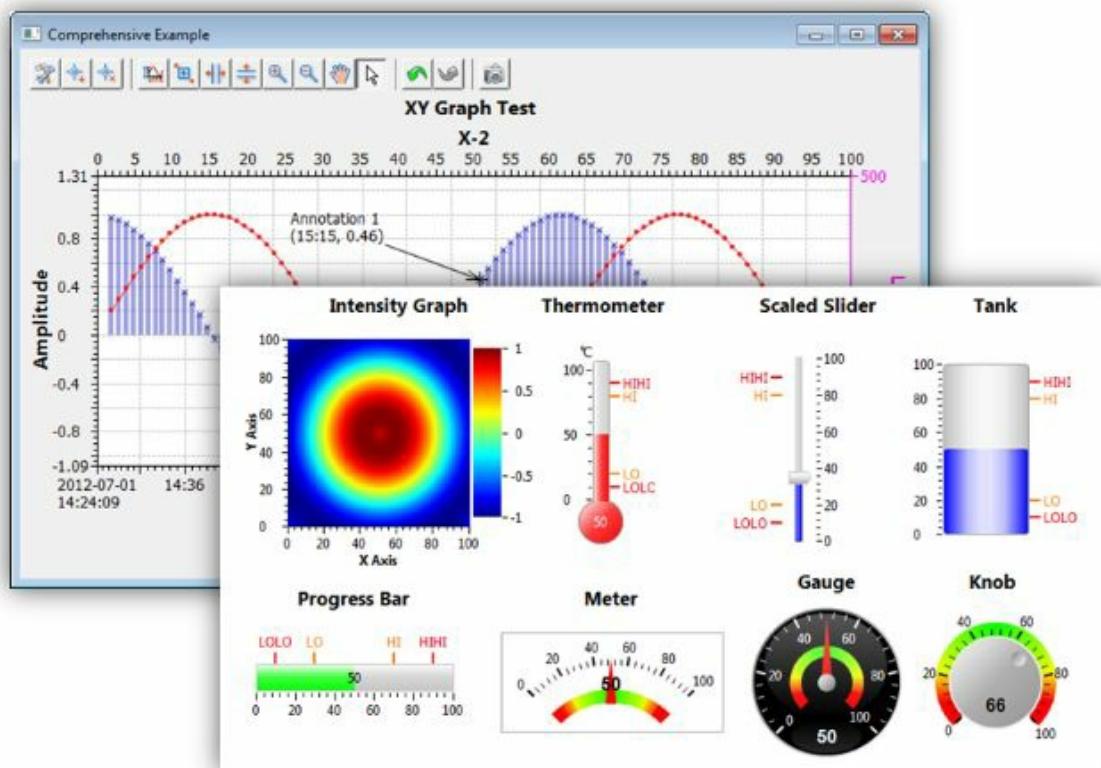
If you do this on a project which has SWT already available, the snippet can be started immediately.

The [SWT Examples](#) page provides useful programs that are written in SWT. These are typically much larger and more comprehensive than the SWT Snippets.

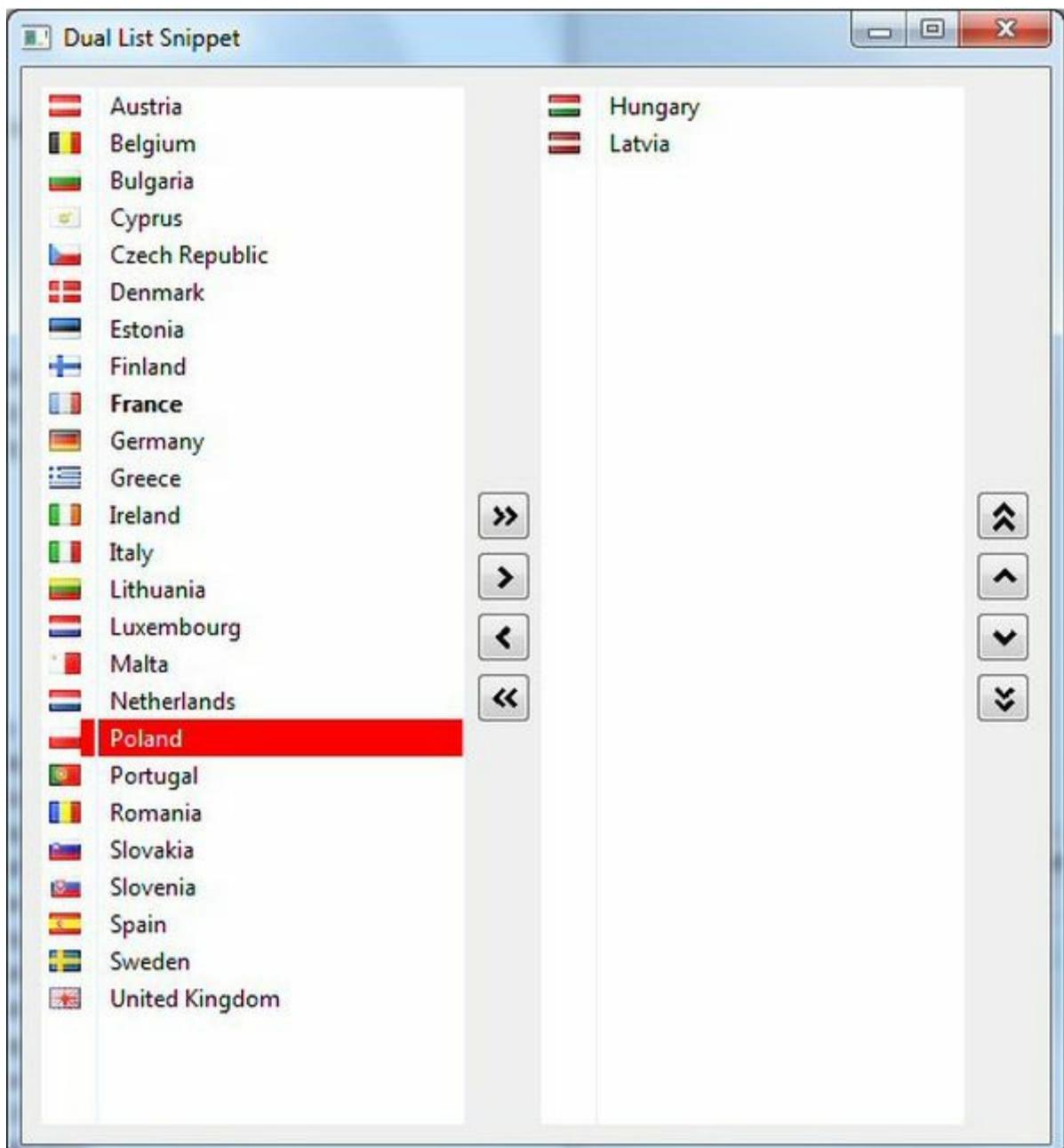
59.2. The Nebula and Opal widgets

The [Eclipse Nebula](#) project provides additional widgets for SWT. For example, it provides several visualization widgets as depicted in the following screenshot.

Visualization :



Another source for SWT widgets is the [Opal widget homepage](#). The following screenshot shows a widgets for displaying two lists.



Chapter 60. SWT Designer

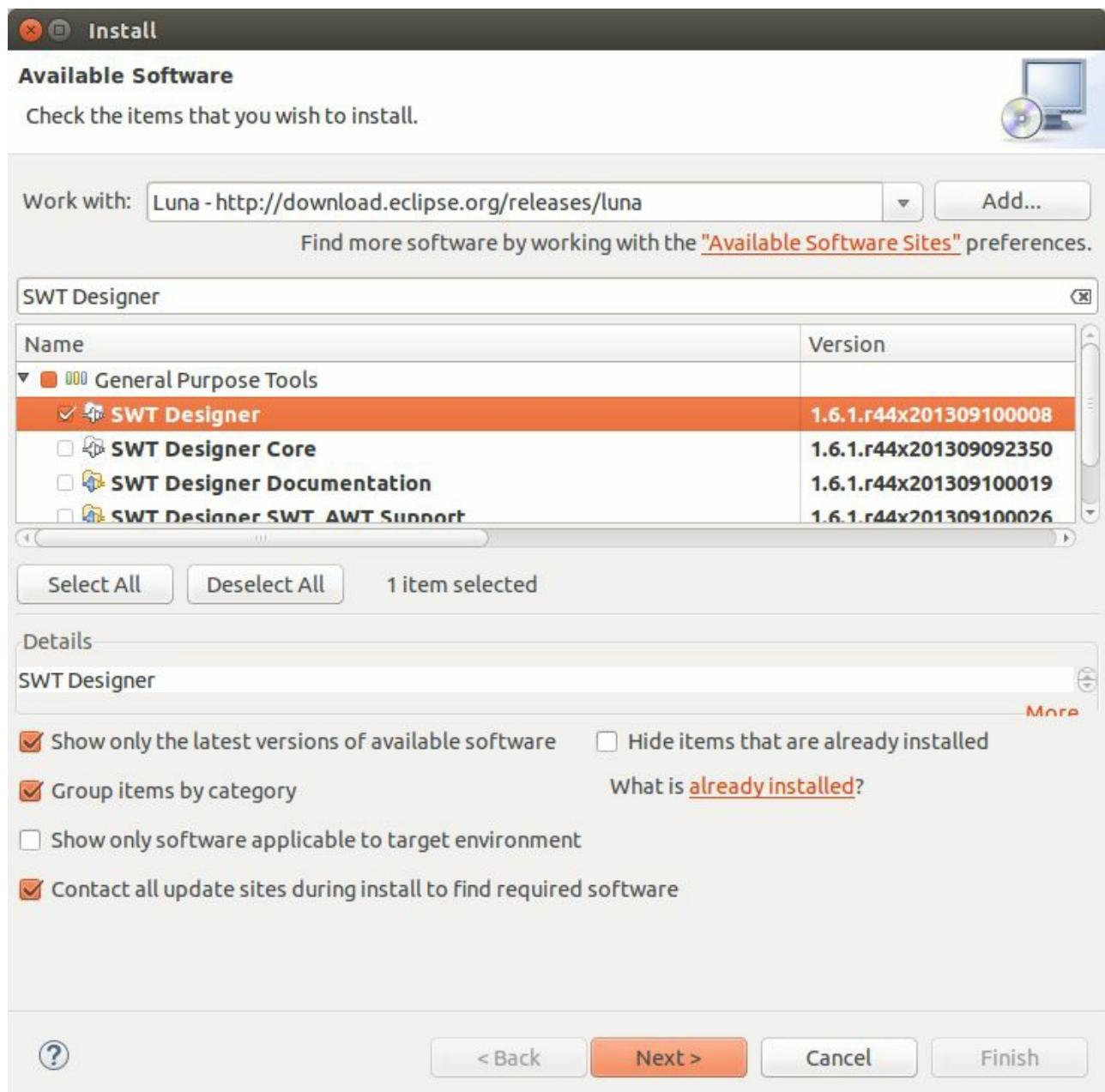
60.1. SWT Designer (WindowBuilder)

SWT Designer is a visual editor used to create graphical user interfaces. It is a two way parser, e.g., you can edit the source code or use a graphical editor to modify the user interface and SWT Designer will synchronize between both representations.

SWT Designer is part of the WindowBuilder project. WindowBuilder provides the foundation and SWT Designer adds the support for working with SWT based applications. SWT Designer supports Eclipse 3.x and Eclipse 4 RCP applications.

60.2. Install SWT Designer

You can install SWT Designer via the Eclipse update manager from the main Eclipse update site.



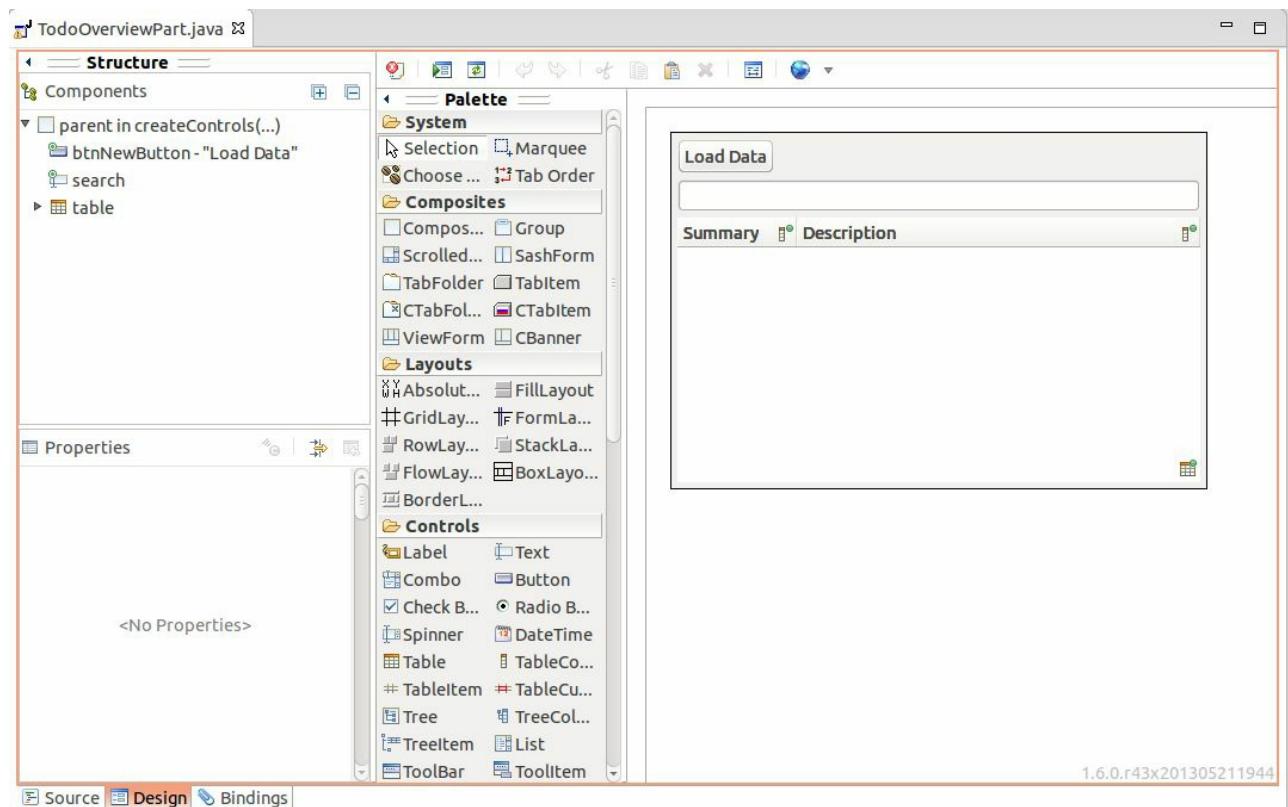
Tip

The latest release of SWT Designer can be found on the following website: [WindowBuilder download](#).

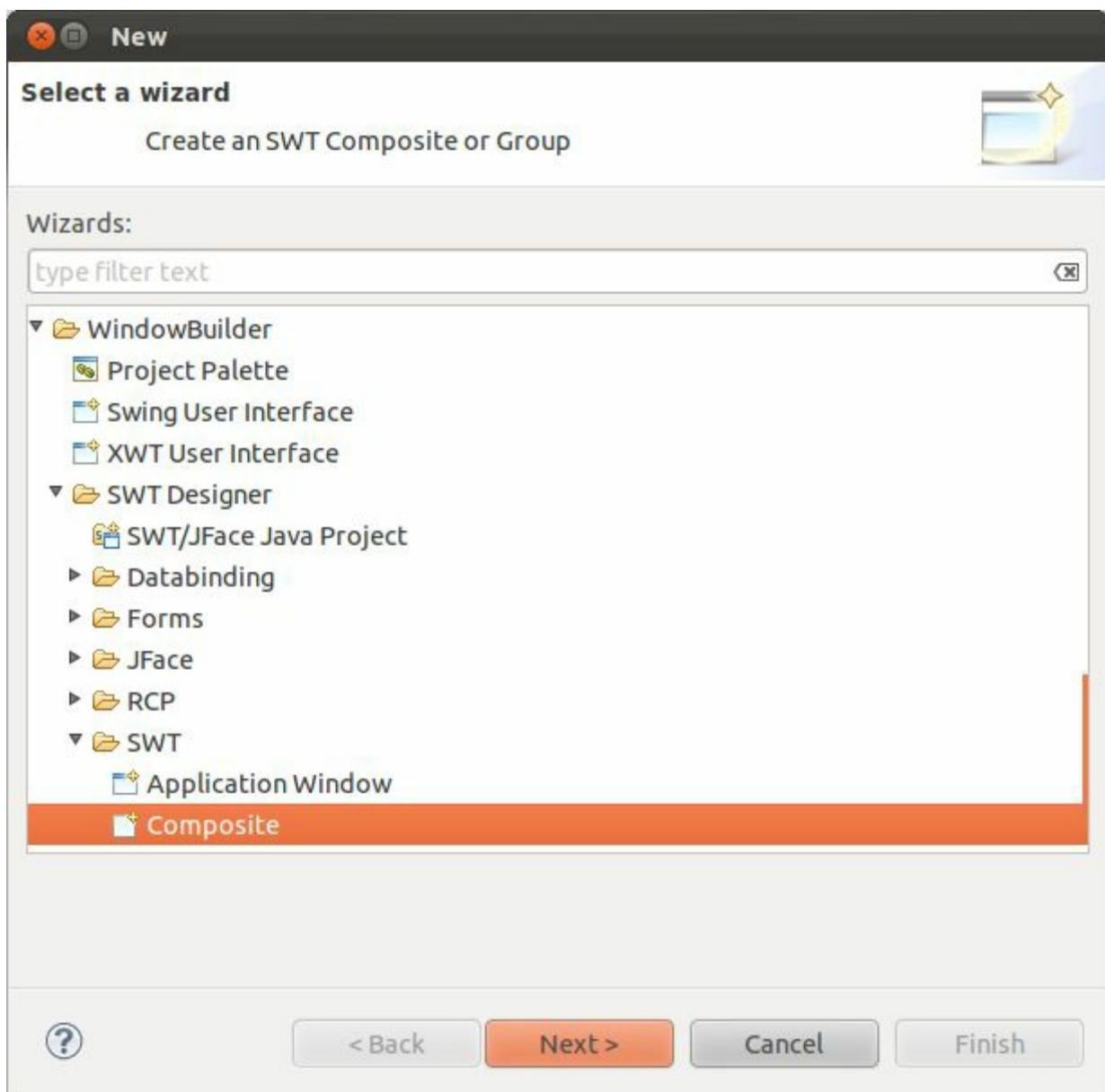
60.3. Using SWT Designer

SWT Designer allows you to open Java components in a special editor. This editor can work with parts, SWT and JFace dialogs, JFace wizards, etc.

SWT Designer allows to drag-and-drop SWT components into an existing layout, change layout settings and create event handlers for your widgets.



You can also use the SWT and JFace templates which SWT Designer contributes to the Eclipse IDE. For example you can use it to create `Composites` and add these to the Eclipse user interface. To create a new `Composite` select `File → New → Other... → WindowBuilder → SWT → Composite`.



SWT Designer has excellent support to establish data binding between your data model and your user interface components via the JFace Data Binding framework.

Chapter 61. Exercise: Getting started with SWT Designer

61.1. Installation

Install SWT Designer if you have not already done this. See [Section 60.2, “Install SWT Designer”](#) for a description.

61.2. Building an user interface

Right-click on your `PlaygroundPart` class and select Open With → WindowBuilder Editor.

Note

WindowBuilder uses the `@PostConstruct` method to identify that a class is an Eclipse part. This method needs to specify at least a `Composite` as parameter.

```
// the WindowBuilder / SWTDesigner tooling
// uses methods to figure out that the
// class is an Eclipse 4 part

// one method must be annotated with @PostConstruct and
// must receive a least a Composite

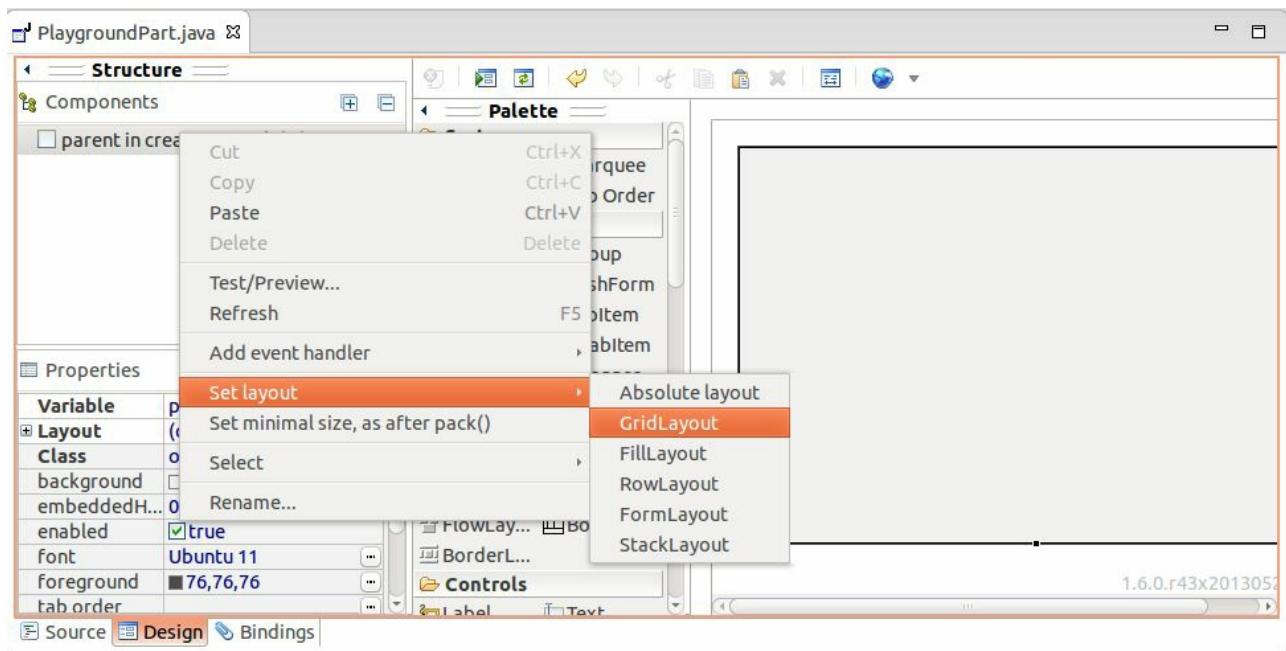
@PostConstruct
public void createControls(Composite parent) {

}
```

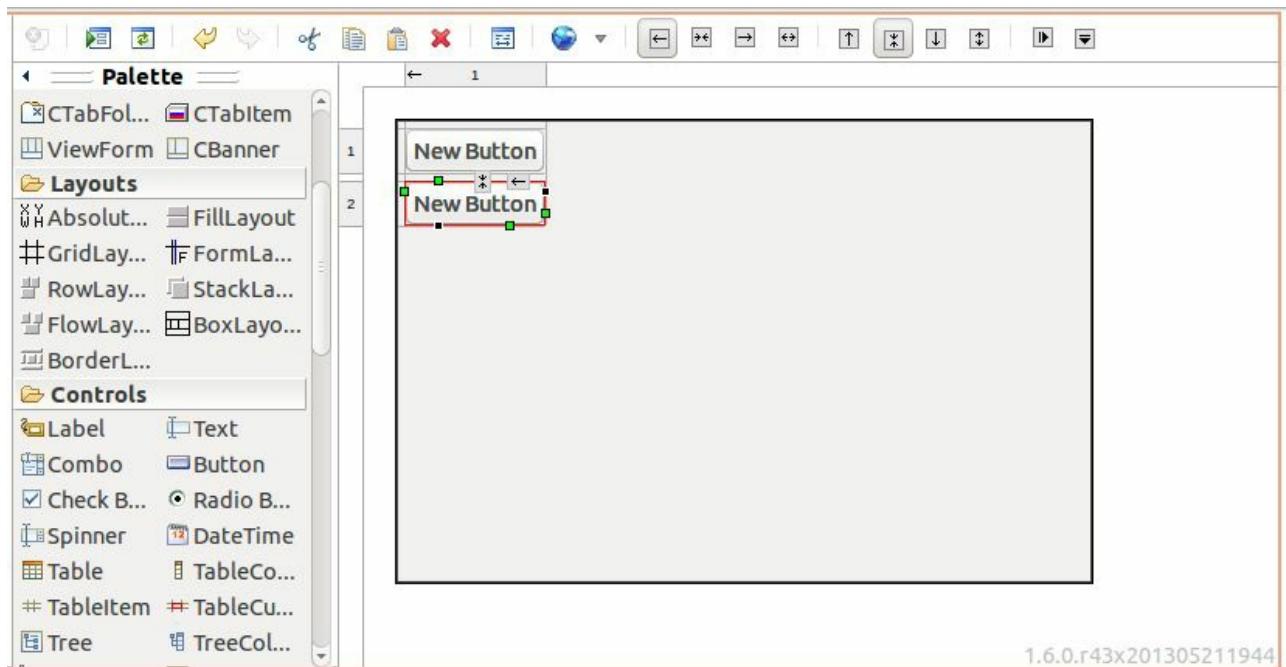
Switch to the *Design* tab in the WindowBuilder editor. This selection is highlighted in the following screenshot.



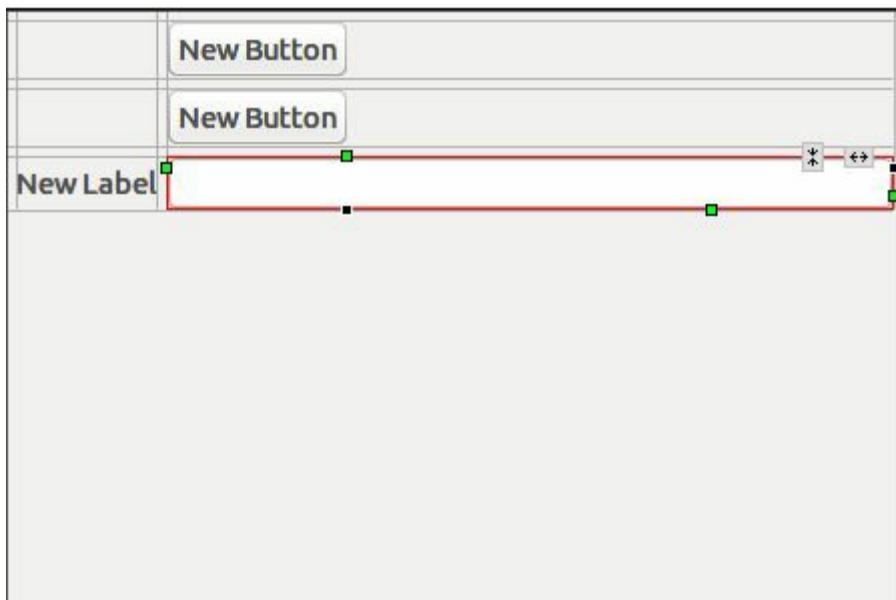
Use SWT Designer to change the layout of the `Composite` of the part to a `GridLayout`.



Click in the Palette on *Button* and add a few buttons to your user interface.

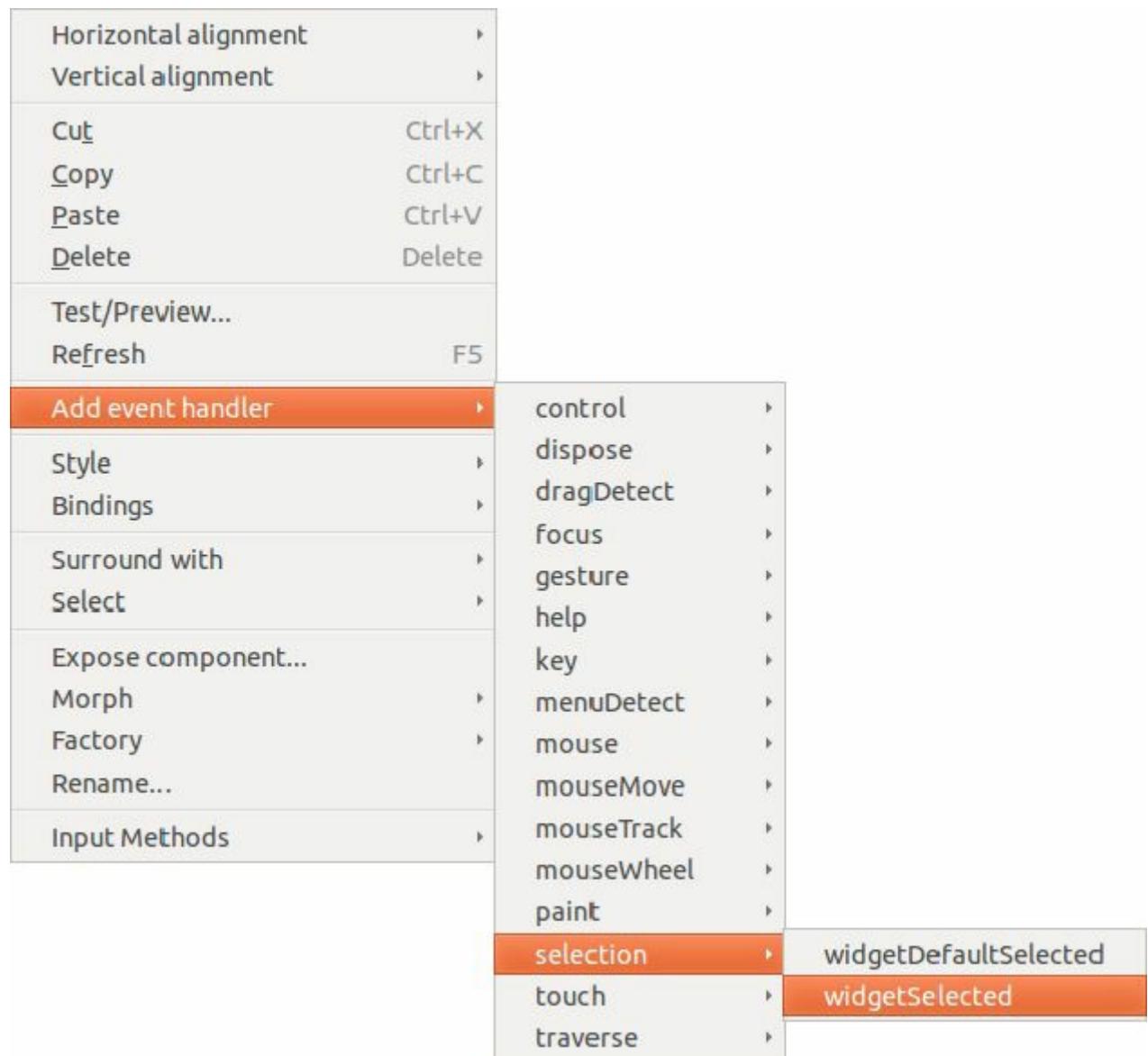


Add a Label and a Text field.



61.3. Creating an event handler

Assign an event handler to one of your buttons via a right-click on the button. Select Add event handler → selection → widgetSelected.



61.4. Review the generated code

Switch to the *Source* tab and review the code generated by the SWT Designer.

Chapter 62. Exercise: Build a first SWT UI

62.1. TodoOverviewPart

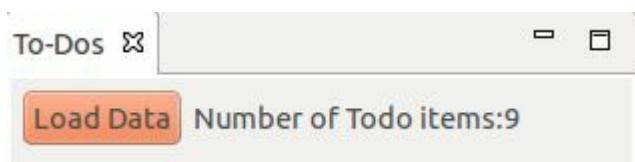
Add a button to your `TodoOverviewPart` class. Besides the button add a label with the default text set to: "Data not available."

Attach a `SelectionListener` to the button. If the button is pressed, read the number of Todos from your `ITodoService` service and assign the number to the text of the label.

The following code may serve as an entry point. See [Section 62.2, “Example code for TodoOverviewPart”](#) for the complete example coding.

```
btnLoadData.addSelectionListener(new SelectionAdapter() {  
    @Override  
    public void widgetSelected(SelectionEvent e) {  
        //TODO update the Label  
    }  
});
```

The resulting user interface should look similar to the following screenshot.



62.2. Example code for TodoOverviewPart

Your resulting implementation of `TodoOverviewPart` should be similar to the following classes.

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

import javax.annotation.PostConstruct;

import org.eclipse.e4.ui.di.Focus;
import org.eclipse.swt.SWT;
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.Label;

import com.example.e4.rcp.todo.model.ITodoService;

public class TodoOverviewPart {

    private static final String NUMBER_OF_TODO_ITEMS = "Number of Todo items: ";
    private Button btnLoadData;
    private Label lblNumberOfTodo;

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

        btnLoadData = new Button(parent, SWT.PUSH);
        btnLoadData.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {
                lblNumberOfTodo.setText(NUMBER_OF_TODO_ITEMS
                    + todoService.getTodos().size());
            }
        });
        btnLoadData.setText("Load Data");

        lblNumberOfTodo = new Label(parent, SWT.NONE);
        lblNumberOfTodo.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true,
            false, 1, 1));
        lblNumberOfTodo.setText("Data not available.");
    }

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

}

Chapter 63. Exercise: Implement a UI for TodoDetailsPart

63.1. TodoDetailsPart

Change the `TodoDetailsPart` class so that you create a user interface similar to the following. Use a `DateTime` widget for the "Due Date" selection.



See [Section 63.3, “Example code for TodoDetailsPart”](#) for the complete example coding.

63.2. Implement focus setting for one of your widgets

Ensure that your @Focus method sets the focus to one of the SWT widgets. This is an optional step but it is a good practice to put the focus explicitly on one SWT widget.

The following code shows an example implementation of this method.

```
@Focus  
public void onFocus() {  
    // The following assumes that you have a Text field  
    // called summary  
    summary.setFocus();  
}
```

Note

If you forget to put focus on an SWT widget in your method annotated with @Focus the framework call this method twice in certain situations. This is done by the Eclipse framework to ensure consistency within the framework. If you correct places the focus on one SWT widget, the method is only called once if the part is activated by the user.

Ensure that you also put the focus on an SWT widget in your TodoOverviewPart class.

63.3. Example code for TodoDetailsPart

The following listing contains a possible implementation of this exercise as reference for the reader.

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

import javax.annotation.PostConstruct;

import org.eclipse.e4.ui.di.Focus;
import org.eclipse.swt.SWT;
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.DateTime;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;

public class TodoDetailsPart {
    private Text txtSummary;
    private Text txtDescription;
    private DateTime dateTime;
    private Button btnDone;

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

        Label lblSummary = new Label(parent, SWT.NONE);
        lblSummary.setText("Summary");

        txtSummary = new Text(parent, SWT.BORDER);
        txtSummary.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true,
            false, 1, 1));

        Label lblDescription = new Label(parent, SWT.NONE);
        lblDescription.setText("Description");

        txtDescription = new Text(parent, SWT.BORDER| SWT.MULTI| SWT.V_SCROLL);
        txtDescription.setLayoutData(new GridData(SWT.FILL, SWT.FILL, true,
            true, 1, 1));

        Label lblDueDate = new Label(parent, SWT.NONE);
        lblDueDate.setText("Due Date");

        dateTime = new DateTime(parent, SWT.BORDER);
        dateTime.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, false, false,
            1, 1));
        new Label(parent, SWT.NONE);

        btnDone = new Button(parent, SWT.CHECK);
```

```
    btnDone.setText("Done");  
}  
  
@Focus  
public void setFocus() {  
    txtSummary.setFocus();  
}  
}
```

Chapter 64. Exercise: Prepare TodoDetailsPart for data

64.1. Preparing for data

The `TodoDetailsPart` class will be used to modify data of `Todo` objects. In this exercise we prepare the class so it contains a `Todo` setter method, which can be called by the Eclipse framework.

64.2. Add dependency

Add the `org.eclipse.e4.ui.services` plug-in as dependency to your application plug-in via the `MANIFEST.MF` file. This is required to use the `IServiceConstants` interface in the next section.

64.3. Prepare for dependency injection

To be able to set the `Todo` object from outside create a `setTodo(Todo todo)` method in the `TodoDetailsPart` class.

Use the following example code to create the method. This method uses annotations so that the current active selected `Todo` object can be injected into it. It uses a special String key defined by the `IServiceConstants.ACTIVE_SELECTION` constant. The `setTodo` method updates your user interface via the `updateUserInterface(Todo todo)` method, whenever a `Todo` object is set. It also disables the widgets if the `Todo` object is not yet set.

```
// more code
// ....
// ....

// define a new field
private Todo todo;

// more code
// ....
// ....

@PostConstruct
public void createControls(Composite parent) {
    // ... existing code for creating user interface
    // more code
    // ....
    // ....
    // NEW! NEW ! NEW ! Add this at the end of the method
    // disable the user interface in case
    // the Todo object is not set
    updateUserInterface(todo);
}

@Inject
public void setTodo(@Optional
    @Named(IServiceConstants.ACTIVE_SELECTION) Todo todo) {
    if (todo != null) {
        // remember todo as field
        this.todo = todo;
    }
    // update the user interface
    updateUserInterface(todo);
}

// allows to disable/ enable the user interface fields
// if no todo is set
```

```

private void enableUserInterface(boolean enabled) {
    if (txtSummary != null && !txtSummary.isDisposed()) {
        txtSummary.setEnabled(enabled);
        txtDescription.setEnabled(enabled);
        dateTIme.setEnabled(enabled);
        btnDone.setEnabled(enabled);
    }
}

private void updateUserInterface(Todo todo) {
    // if Todo is null disable user interface
    // and leave method
    if (todo == null) {
        enableUserInterface(false);
        return;
    }

    // the following check ensures that the user interface is available,
    // it assumes that you have a text widget called "txtSummary"
    if (txtSummary != null && !txtSummary.isDisposed()) {
        enableUserInterface(true);
        txtSummary.setText(todo.getSummary());
        // more code to fill the widgets with data from your Todo object
        // more code
        // ....
        // ....
    }
}

```

In this method you have to check the availability of the SWT widgets because you cannot be sure that the `@PostConstruct` method has already been called.

64.4. Usage of this method

At the moment the `setTodo()` method is not called with a `Todo` object. Neither by the Eclipse framework, nor by your coding. You call this method in [Chapter 87, Exercise: Create a wizard](#) and the Eclipse framework uses this method after you implement the changes in [Chapter 99, Exercise: Selection service](#).

Chapter 65. Exercise: Using the SWT Browser widget

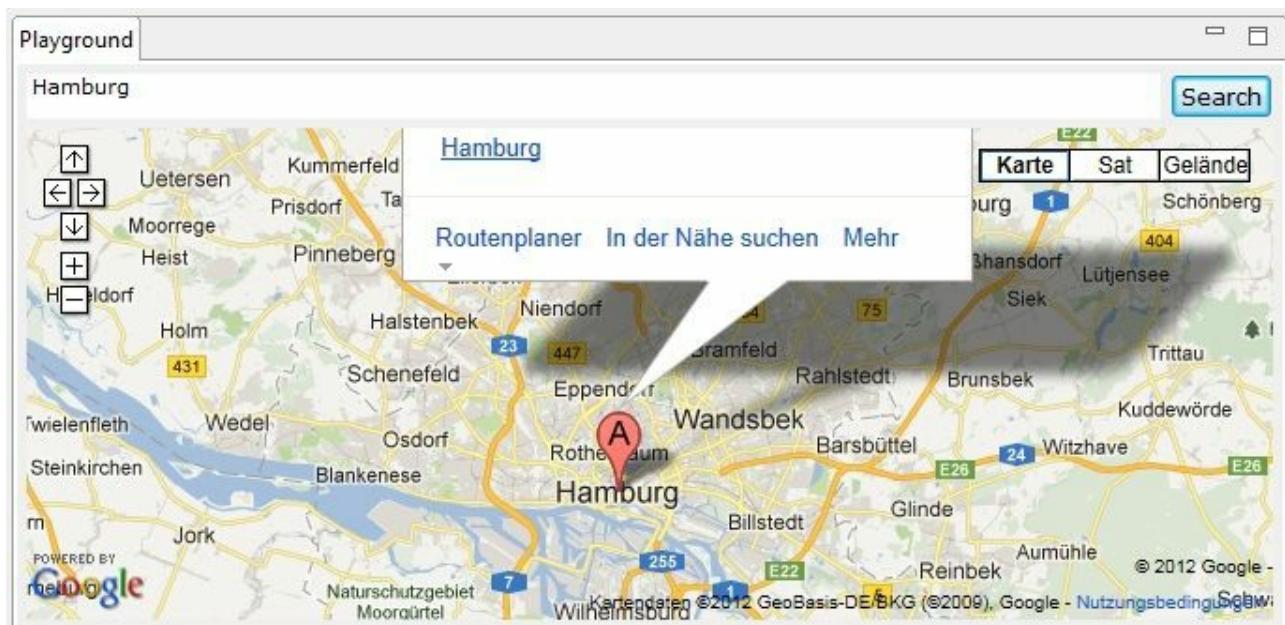
65.1. Implementation

In this exercise you will display Google Maps in an SWT Browser widget.

Note

This exercise does not always work on a Linux system because on certain Linux versions the `Browser` widget does not work. See the Eclipse SWT FAQ answered at [How do I use the WebKit renderer on Linux-GTK](#) for details.

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



Note

This example might not work, in case Google changes its API.

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

65.2. Solution

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

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

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

import javax.annotation.PostConstruct;

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

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

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

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

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

```
        } 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();
}
}
```

Part XIII. User interface development with JFace

Chapter 66. JFace

66.1. What is Eclipse JFace?

Eclipse *JFace* is a set of plug-ins based upon the user interface toolkit SWT. JFace provides classes and frameworks which simplify common SWT use cases. JFace does not hide the SWT API; therefore, SWT knowledge is still required.

JFace provides the *viewers* framework, which simplifies the mapping of a data model to a visual representation. For example you find viewers for Combo Boxes, Tables and Trees.

JFace also provides helper classes to effectively manage your system resources, like colors, images and fonts.

In addition JFace provides support for handling preferences, preference pages, wizards and dialogs. It also contains functionality to support icon decorations and user-input help for SWT controls.

JFace Data Binding is a framework which connects properties of objects. It is typically used to synchronize fields of the user interface with properties of model objects and allows you to include validation and conversion in this synchronization process.

66.2. JFace resource manager for Colors, Fonts and Images

SWT is based on the native widgets of the OS. Whenever an SWT widget is allocated, a corresponding OS specific widget is created. The Java garbage collector cannot automatically clean-up these OS-specific widget references.

Fortunately all widgets which are created based on a parent widget are automatically disposed when the parent `Composite` is disposed. If you develop Eclipse plug-ins, the `Composite` of a part is automatically disposed once the part is closed. Therefore, these SWT widgets are handled automatically in Eclipse plug-in projects.

This rule does not apply for colors, fonts and images, as these may be reused in other places. For this reason they need to be explicitly disposed. Fortunately JFace provides the `LocalResourceManager` class.

An instance of the `LocalResourceManager` class is created with a reference to a `Composite`. If this `Composite` is disposed, the resources created by the `LocalResourceManager` are also disposed.

```
// create the manager and bind to a widget
LocalResourceManager resManager =
    new LocalResourceManager(JFaceResources.getResources(), composite);

// create resources
Color color = resManager.createColor(new RGB(200, 100, 0));
Font font = resManager.
    createFont(FontDescriptor.createFrom("Arial", 10, SWT.BOLD));
// get an imageDescriptor and create Image object
Image image = resManager.createImage(imageDescriptor);
```

The `createImage()` method expects an `ImageDescriptor` class. To get one `ImageDescriptor` from an image file stored in your current plug-in use the following:

```
Bundle bundle = FrameworkUtil.getBundle(this.getClass());
// use the org.eclipse.core.runtime.Path as import
URL url = FileLocator.find(bundle,
    new Path("icons/alt_window_32.gif"), null);
ImageDescriptor imageDescriptor = ImageDescriptor.createFromURL(url);
```

To create the `Image` directly from the `ImageDescriptor` you can use the `createImage()` method on the `ImageDescriptor` object.

66.3. ControlDecoration

The `ControlDecoration` class allows you to place image decorations on SWT controls to show additional information about the control. These decorations can also have a description text which is displayed once the user places the mouse over them.

During the layout of your screen you need to make sure that enough space is available to display these decorations.

The following code snippet shows how to create `ControlDecoration` and how to set a description and an icon to it.

```
// create the decoration for the text UI component
final ControlDecoration deco =
    new ControlDecoration(text, SWT.TOP | SWT.RIGHT);

// re-use an existing image
Image image = FieldDecorationRegistry.
    getDefault().
    getFieldDecoration(FieldDecorationRegistry.DEC_INFORMATION).
    getImage();

// set description and image
deco.setDescriptionText("This is a tooltip text");
deco.setImage(image);
// hide deco if not in focus
deco.setShowOnlyOnFocus(true);
```

You can hide and show the decoration via the corresponding methods as demonstrated in the following code snippet.

```
deco.hide();
deco.show();
```

66.4. User input help with field assistance

The `org.eclipse.jface.fieldassist` package provides assistance to define user-input help for a widget, e.g., a text field or combo box. The `ContentProposalAdapter` class is responsible for providing the possible input values.

In the following example the content proposal should get activated via certain keys ("." and "#") as well as the **Ctrl+Space** key combination.

The following code demonstrates the usage of the field assistance functionality. It also uses the `ControlDecoration` class.

```
GridLayout layout = new GridLayout(2, false);
// parent is a Composite
parent.setLayout(layout);
Label lblPleaseEnterA = new Label(parent, SWT.NONE);
lblPleaseEnterA.setText("Please enter a value:");

Text text = new Text(parent, SWT.BORDER);
GridData gd_text = new GridData(SWT.FILL, SWT.CENTER, true, false);
gd_text.horizontalIndent = 8;
text.setLayoutData(gd_text);
GridData data = new GridData(SWT.FILL, SWT.TOP, true, false);

// create the decoration for the text component
final ControlDecoration deco = new ControlDecoration(text, SWT.TOP
    | SWT.LEFT);

// use an existing image
Image image = FieldDecorationRegistry.getDefault()
    .getFieldDecoration(FieldDecorationRegistry.DEC_INFORMATION)
    .getImage();

// set description and image
deco.setDescriptionText("Use CTRL + SPACE to see possible values");
deco.setImage(image);

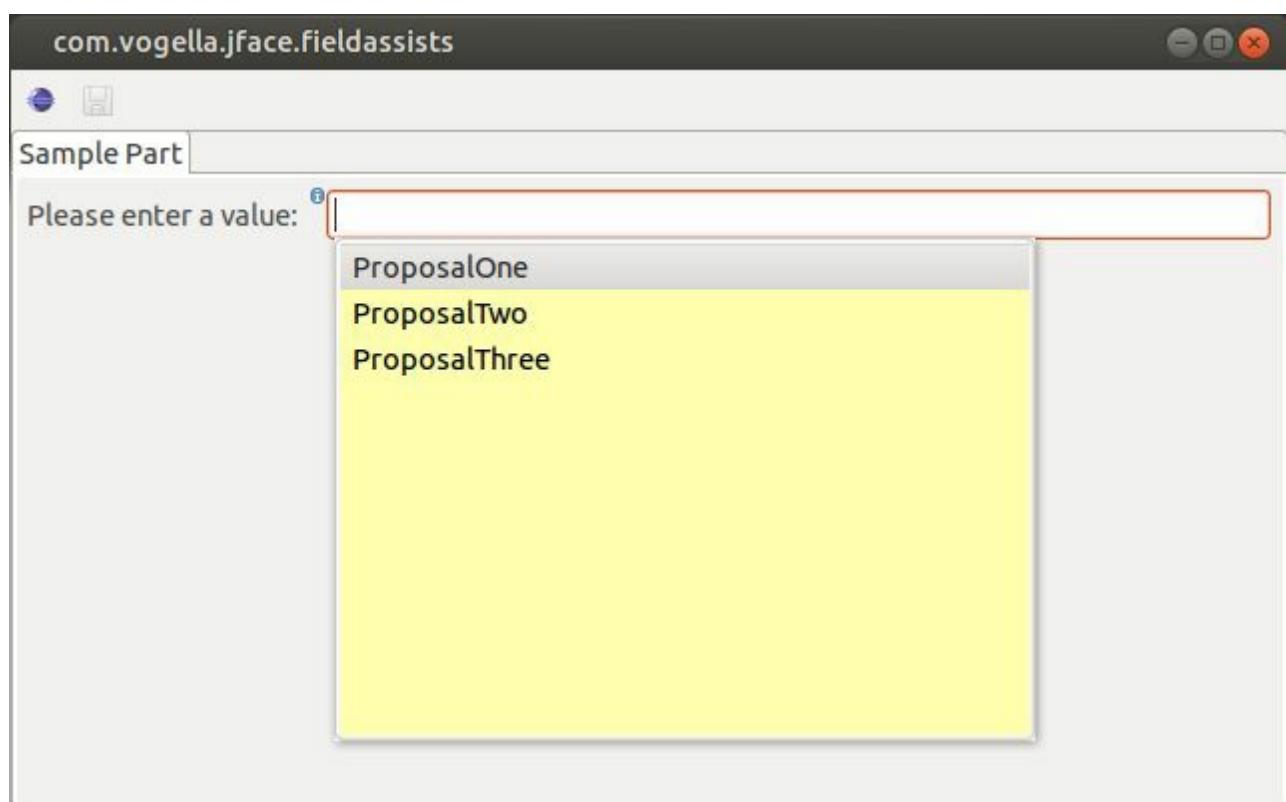
// always show decoration
deco.setShowOnlyOnFocus(false);

// hide the decoration if the text component has content
text.addModifyListener(new ModifyListener() {
    @Override
    public void modifyText(ModifyEvent e) {
        Text text = (Text) e.getSource();
        if (!text.getText().isEmpty()) {
            deco.hide();
        } else {
```

```
        deco.show();
    }
}
});

// help the user with the possible inputs
// "." and "#" activate the content proposals
char[] autoActivationCharacters = new char[] { '.', '#' };
KeyStroke keyStroke;
//
try {
    keyStroke = KeyStroke.getInstance("Ctrl+Space");
    ContentProposalAdapter adapter = new ContentProposalAdapter(text,
        new TextContentAdapter(),
        new SimpleContentProposalProvider(new String[] {
            "ProposalOne", "ProposalTwo", "ProposalThree" })),
        keyStroke, autoActivationCharacters);
} catch (ParseException e1) {
    e1.printStackTrace();
}
```

If used the result should look similar to the following.



Chapter 67. The JFace viewer framework

67.1. Purpose of the JFace viewer framework

The JFace viewer framework allows you to display a domain model in a standard SWT widget like list, combo, tree or table without converting the domain model beforehand.

A viewer allows you to set a *content provider* which provides the data for the viewer. The content provider makes no assumption about the presentation of the data model.

You can also assign at least one *label provider* to a viewer. The label provider defines how the data from the model will be displayed in the viewer.

67.2. Standard JFace viewer

JFace provides several standard viewer implementations. These viewers are part of the `org.eclipse.jface.viewers` package. The following list contains the most important ones.

- ComboViewer
- ListViewer
- TreeViewer
- TableViewer

67.3. Standard content and label provider

The related interfaces for defining a content provider are described in the following table.

Table 67.1. Content providers

Interface	Default implementation	Description
IStructuredContentProvider	ArrayContentProvider	Used for the List-, Combo- and TableViewer. JFace provides implementation for Collection with the <code>ArrayContentProvider</code> . Because the <code>ArrayContentProvider</code> class does not store any data, it is safe to share an instance with several viewers. To get a shared instance use the <code>ArrayContentProvider.getSharedInstance()</code> method.
ITreeContentProvider	Not available	Used for the TreeViewer class. It adds additional methods to determine the children and the parents of the tree.

Important standard label providers are listed in the following table.

Table 67.2. Label providers

Required class	Standard label providers	Description
ILabelProvider	LabelProvider	Used for lists and trees, can return an icon and a label per element.
CellLabelProvider	ColumnLabelProvider	Used for tables. Defines a label provider per column.

67.4. JFace ComboViewer

A simple example for a JFace Viewer framework is the `ComboViewer` class. Assume the following data model.

```
package com.vogella.eclipse.viewer;

public class Person {
    private String firstName;
    private String lastName;
    private boolean married;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public boolean isMarried() {
        return married;
    }

    public void setMarried(boolean married) {
        this.married = married;
    }
}
```

The following example snippet shows you how you could use this given data model in a `ComboViewer`.

```
// the following code is executed by the method which
// creates the user interface
// assumes parent is an SWT Composite

GridLayout layout = new GridLayout(2, false);
parent.setLayout(layout);
```

```

Label label = new Label(parent, SWT.NONE);
label.setText("Select a person:");
final ComboViewer viewer = new ComboViewer(parent, SWT.READ_ONLY);

// the ArrayContentProvider object does not store any state,
// therefore, you can re-use instances

viewer.setContentProvider(ArrayContentProvider.getInstance());
viewer.setLabelProvider(new LabelProvider() {
    @Override
    public String getText(Object element) {
        if (element instanceof Person) {
            Person person = (Person) element;
            return person.getFirstName();
        }
        return super.getText(element);
    }
});
});

Person[] persons = new Person[] { new Person("Lars", "Vogel"),
    new Person("Tim", "Taler"), new Person("Jim", "Knopf") };

// set the input of the Viewer,
// this input is send to the content provider

viewer.setInput(persons);

// note: the order of setContentProvider and setInput is important

```

You can register a listener which is notified whenever the selection of the viewer changes via the following code.

```

// react to the selection change of the viewer
// note that the viewer returns the actual object

viewer.addSelectionChangedListener(new ISelectionChangedListener() {
    @Override
    public void selectionChanged(SelectionChangedEvent event) {
        IStructuredSelection selection = (IStructuredSelection) event
            .getSelection();
        if (selection.size() > 0){
            System.out.println(((Person) selection.getFirstElement())
                .getLastName());
        }
    }
});

```

You can get and set selections using Java objects based on your domain model.

```

// you can select an object directly via the domain object
Person person = persons[0];

```

```
viewer.setSelection(new StructuredSelection(person));  
  
// retrieves the selection, returns the data model object  
IStructuredSelection selection =  
    (IStructuredSelection) viewer.getSelection();  
Person p = (Person) selection.getFirstElement();
```

Chapter 68. Using tables

68.1. Using the JFace TableViewer

You can use the `TableViewer` class to create tables using the JFace framework. The SWT `Table` widget is wrapped into the `TableViewer` and can still be accessed to set its properties.

```
// define the TableViewer
viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL
    | SWT.V_SCROLL | SWT.FULL_SELECTION | SWT.BORDER);

// create the columns
// not yet implemented
createColumns(viewer);

// make lines and header visible
final Table table = viewer.getTable();
table.setHeaderVisible(true);
table.setLinesVisible(true);
```

68.2. Content provider for JFace tables

As with other JFace viewers a content provider supplies the data which should be displayed in the `TableViewer`.

Eclipse provides an implementation of this interface via the `ArrayContentProvider` class. The `ArrayContentProvider` class supports Arrays or Collections as input, containing the domain data. You can implement your own content provider for a table by implementing the interface `IStructuredContentProvider` from the `org.eclipse.jface.viewers` package.

The `getElements()` method of the content provider is used to translate the input of the viewer into an array of elements. Once the `setInput()` method on the viewer is called, it uses the content provider to convert it. This is the reason why the content provider must be set before the `setInput()` method is called.

Each object in the Array returned by the content provider is displayed as individual element by the viewer. In case of the table viewer each object is displayed in an individual row.

The usage of the content provider is demonstrated with the following code snippet.

```
// this code is placed after the definition of  
// the viewer  
  
// set the content provider  
viewer.setContentProvider(ArrayContentProvider.getInstance());  
  
// provide the input to the viewer  
// setInput() calls getElements() on the  
// content provider instance  
viewer.setInput(someData...);
```

68.3. Columns and label provider

Columns for a JFace `TableViewer` object are defined by creating instances of the `TableViewerColumn` class.

Each `TableViewerColumn` object needs to get a label provider assigned to it via the `setLabelProvider()` method. The label provider defines which data is displayed in the column. The label provider for a table viewer column is called per row and gets the corresponding object as input. It uses this input to determine which data is displayed in the column for this row.

Typically the label provider returns a `String`, but more complex implementations are possible.

The `setLabelProvider()` method on the `TableViewerColumn` expects an instance of the abstract `CellLabelProvider` class. A default implementation of this class is provided by the `ColumnLabelProvider` class. Its usage is demonstrated in the following code snippet.

```
// create a column for the first name
TableViewerColumn colFirstName = new TableViewerColumn(viewer, SWT.NONE);
colFirstName.getColumn().setWidth(200);
colFirstName.getColumn().setText("Firstname");
colFirstName.setLabelProvider(new ColumnLabelProvider() {
    @Override
    public String getText(Object element) {
        Person p = (Person) element;
        return p.getFirstName();
    }
});

// create more text columns if required...

// create column for married property of Person
// uses getImage() method instead of getText()
// CHECKED andUNCHECK are fields of type Image

TableViewerColumn colMarried = new TableViewerColumn(viewer, SWT.NONE);
colMarried.getColumn().setWidth(200);
colMarried.getColumn().setText("Married");
colMarried.setLabelProvider(new ColumnLabelProvider() {
    @Override
    public String getText(Object element) {
        return null; // no string representation, we only want to display the image
    }
}

@Override
```

```

public Image getImage(Object element) {
    if (((Person) element).isMarried()) {
        return CHECKED;
    }
    return UNCHECKED;
}
);

```

The above code uses two fields which contain `Image` instances. These fields could for example be initialized via the following code. Using the classes in this code requires a dependency to the `org.eclipse.core.runtime` plug-in.

```

// fields for your class
// assumes that you have these two icons
// in the "icons" folder
private final Image CHECKED = getImage("checked.gif");
private final Image UNCHECKED = getImage("unchecked.gif");

// more code...

// helper method to load the images
// ensure to dispose the images in your @PreDestroy method
private static Image getImage(String file) {

    // assume that the current class is called View.java
    Bundle bundle = FrameworkUtil.getBundle(View.class);
    URL url = FileLocator.find(bundle, new Path("icons/" + file), null);
    ImageDescriptor image = ImageDescriptor.createFromURL(url);
    return image.createImage();

}

```

68.4. Reflect data changes in the viewer

To reflect data changes in the data model that is displayed by the viewer, you can call the `viewer.refresh()` method. This method updates the viewer based on the data which is assigned to it.

To change the data which is displayed use the `viewer.setInput()` method.

68.5. Selection change listener

Via the `addSelectionChangedListener` method you can add a listener to a viewer. This listener is an implementation of the `ISelectionChangedListener` interface. The following code shows an example that gets the selected element of the viewer.

```
viewer.addSelectionChangedListener(new ISelectionChangedListener() {
    @Override
    public void selectionChanged(SelectionChangedEvent event) {
        IStructuredSelection selection = (IStructuredSelection)
            viewer.getSelection();
        Object firstElement = selection.getFirstElement();
        // do something with it
    }
});
```

68.6. Column editing support

To make a column in a table editable, you need an object of type `EditingSupport`.

The following code shows an example of an `EditingSupport` implementation.

```
package de.vogella.jface.tableviewer.edit;

import org.eclipse.jface.viewers.CellEditor;
import org.eclipse.jface.viewers.EditingSupport;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.jface.viewers.TextCellEditor;

import com.vogella.eclipse.viewer.Person;

public class FirstNameEditingSupport extends EditingSupport {

    private final TableViewer viewer;
    private final CellEditor editor;

    public FirstNameEditingSupport(TableViewer viewer) {
        super(viewer);
        this.viewer = viewer;
        this.editor = new TextCellEditor(viewer.getTable());
    }

    @Override
    protected CellEditor getCellEditor(Object element) {
        return editor;
    }

    @Override
    protected boolean canEdit(Object element) {
        return true;
    }

    @Override
    protected Object getValue(Object element) {
        return ((Person) element).getFirstName();
    }

    @Override
    protected void setValue(Object element, Object userInputValue) {
        ((Person) element).setFirstName(String.valueOf(userInputValue));
        viewer.update(element, null);
    }
}
```

The `EditingSupport` implementation defines how the content can be changed.

The `getCellEditor()` method returns an object of type `CellEditor`. This object creates the controls to change the data.

The `canEdit()` method defines if the cell can be edited. The `getValue()` method receives the current object and returns the value which should be edited.

The method `setValue()` in `EditingSupport` receives the changed value based on the user input. In this method you assign the value to your data object.

JFace provides the following default implementations for cell editors:

- `TextCellEditor`
- `ColorCellEditor`
- `CheckboxCellEditor`
- `DialogCellEditor`
- `ComboBoxViewerCellEditor`

You can assign the instance of `EditingSupport` to your `TableColumn` via the `setEditingSupport()` method of your `TableViewerColumn` object.

```
colFirstName.setEditingSupport(new FirstNameEditingSupport(viewer));
```

From an application design perspective, editing within a table can be cumbersome for the user. If the end user has to edit a lot of data, you should also offer a dialog, wizard or part to edit the data.

68.7. Filtering data

A JFace viewer supports filtering of data via the `setFilters()` or `addFilter()` methods. These methods expect `ViewerFilter` objects as arguments.

For each registered `ViewerFilter` object the `select()` method is called. The method returns `true` if the data should be shown and `false` if it should be filtered.

```
package de.vogella.jface.tableviewer.filter;

import org.eclipse.jface.viewers.Viewer;
import org.eclipse.jface.viewers.ViewerFilter;

import de.vogella.jface.tableviewer.model.Person;

public class PersonFilter extends ViewerFilter {

    private String searchString;

    public void setSearchText(String s) {
        // ensure that the value can be used for matching
        this.searchString = ".*" + s + ".*";
    }

    @Override
    public boolean select(Viewer viewer,
        Object parentElement,
        Object element) {
        if (searchString == null || searchString.length() == 0) {
            return true;
        }
        Person p = (Person) element;
        if (p.getFirstName().matches(searchString)) {
            return true;
        }
        if (p.getLastName().matches(searchString)) {
            return true;
        }
        return false;
    }
}
```

All filters are checked whenever the input of the viewer changes, or whenever its `refresh()` method is called.

If more than one filter is defined for a viewer, all filters must return `true` to

display the data.

68.8. Sorting data with ViewerComparator

JFace supports sorting of the viewer content via the `setComparator()` method on the viewer object. This method expects a `ViewerComparator` object. By default, it will sort based on the `toString()` method of the objects in the viewer.

```
// sort according to due date
viewer.setComparator(new ViewerComparator() {
    public int compare(Viewer viewer, Object e1, Object e2) {
        Todo t1 = (Todo) e1;
        Todo t2 = (Todo) e2;
        return t1.getDueDate().compareTo(t2.getDueDate());
    }
});
```

68.9. TableColumnLayout

With the `TableColumnLayout` class you can define the width of the columns in the table. This can be done based on a fixed or percentage value.

Using `TableColumnLayout` requires a `Composite` which only contains the table widget. This `Composite` gets the `TableColumnLayout` assigned.

```
Composite tableComposite = new Composite(parent, SWT.NONE);
TableColumnLayout tableColumnLayout = new TableColumnLayout();
tableComposite.setLayout(tableColumnLayout);
TableViewer viewer =
    new TableViewer(tableComposite,
        SWT.MULTI |
        SWT.H_SCROLL |
        SWT.V_SCROLL |
        SWT.FULL_SELECTION |
        SWT.BORDER);
```

The `TableColumnLayout` requires that you define a fixed or relative size for all columns.

```
// fixed size
tableColumnLayout.
    setColumnData(colFirstName.getColumn(),
        new ColumnPixelData(50));

// alternatively use relative size
// last parameter defines if the column is allowed
// to be resized
tableColumnLayout.
    setColumnData(colFirstName.getColumn(),
        new ColumnWeightData(20, 200, true));
```

68.10. StyledCellLabelProvider and OwnerDrawLabelProvider

It is possible to use a `StyledCellLabelProvider` for a very flexible styling of your text. `StyledCellLabelProvider` extends `CellLabelProvider` and allows you to style the text which is displayed in the cell.

The following example shows how to use a `StyledCellLabelProvider`. In this example a portion of a pre-defined text is highlighted.

```
// define column
TableViewerColumn colTesting = new TableViewerColumn(viewer, SWT.NONE);
colTesting.getColumn().setText("Testing");
colTesting.getColumn().setWidth(200);

// set label provider
colTesting.setLabelProvider(new StyledCellLabelProvider() {
    @Override
    public void update(ViewerCell cell) {
        cell.setText("This is a test (15)");
        StyleRange myStyledRange =
            new StyleRange(16, 2, null,
                Display.getCurrent().getSystemColor(SWT.COLOR_YELLOW));
        StyleRange[] range = { myStyledRange };
        cell.setStyleRanges(range);
        super.update(cell);
    }
});
```

As a result the number "15" will be highlighted as depicted in the following screenshot.

To-dos		
Summary	Description	Testing
SWT	Learn Widget API	This is a test (15)
JFace	Especially Viewers	This is a test (15)
DI	@Inject looks cool	This is a test (15)
OSGi	Services	This is a test (15)
Compatibility	Run Eclipse 3.4	This is a test (15)

The `OwnerDrawLabelProvider` class is a label provider that handles custom draws.

The following example draws a text and an image into the cell.

```

TableViewerColumn colHelloAndIcon = new TableViewerColumn(viewer, SWT.NONE);
colHelloAndIcon.getColumn().setText("Column 1");
colHelloAndIcon.getColumn().setWidth(200);

// ICON is an Image
colHelloAndIcon.setLabelProvider(new OwnerDrawLabelProvider() {
    @Override
    protected void measure(Event event, Object element) {
        Rectangle rectangle = ICON.getBounds();
        event.
        setBounds(new Rectangle(event.x,
            event.y,
            rectangle.width + 200 ,
            rectangle.height));
    }

    @Override
    protected void paint(Event event, Object element) {
        Rectangle bounds = event.getBounds();
        event.gc.drawText("Hello", bounds.x, bounds.y);
        Point point = event.gc.stringExtent("Hello");
        event.gc.drawImage(ICONS, bounds.x + 5 + point.x, bounds.y);
    }
});

});

```



68.11. Table column menu and hiding columns

You can add a menu to your table. This menu can get a menu entry for each column. This allow you to add arbitrary popup actions to your columns.

You can use it, for example, to hide and show columns based on the width setting as demonstrated in the following code snippet.

```
// define the menu and assign to the table
contextMenu = new Menu(viewer.getTable());
viewer.getTable().setMenu(contextMenu);

// create your columns as usual...

// afterwards add a MenuItem for each column to the table menu
for (TableColumn TableColumn : viewer.getTable().getColumns()) {
    createMenuItem(contextMenu, TableColumn);
}

// the createMenuItem() method adds a MenuItem
// per column to the menu

private void createMenuItem(Menu parent, final TableColumn column) {
    final MenuItem itemName = new MenuItem(parent, SWT.CHECK);
    itemName.setText(column.getText());
    itemName.setSelection(column.getResizable());
    itemName.addSelectionListener(new SelectionAdapter() {
        @Override
        public void widgetSelected(SelectionEvent e) {
            if (itemName.getSelection()) {
                column.setWidth(150);
                column.setResizable(true);
            } else {
                column.setWidth(0);
                column.setResizable(false);
            }
        }
    });
}
```

68.12. Tooltips for viewers

You can use tooltips for the cells of the viewer. To achieve this, you have to activate the tooltips for a viewer.

```
// activate the tooltip support for the viewer
ColumnViewerToolTipSupport.enableFor(viewer, ToolTip.NO_RECREATE);
```

In your `CellLabelProvider` you specify the related methods for displaying the tooltip.

```
colFirstName.setLabelProvider(new ColumnLabelProvider() {
    @Override
    public String getText(Object element) {
        Person p = (Person) element;
        return p.getFirstName();
    }

    @Override
    public String getToolTipText(Object element) {
        return "Tooltip (" + ((Person)element).getLastName() + ")";
    }

    @Override
    public Point getToolTipShift(Object object) {
        return new Point(5, 5);
    }

    @Override
    public int getToolTipDisplayDelayTime(Object object) {
        return 100; // msec
    }

    @Override
    public int getToolTipTimeDisplayed(Object object) {
        return 5000; // msec
    }
});
```

68.13. Virtual tables with LazyContentProvider

If you have a huge number of lines which you want to display in the table, you can use a `LazyContentProvider`. This provider allows you to fetch the data when they are needed instead of loading everything into memory. As a result of the lazy loading you gain a better memory footprint and improve the performance for a large set of data. The following code demonstrates its usage.

```
private class MyLazyContentProvider implements ILazyContentProvider {
    private TableViewer viewer;
    private Person[] elements;

    public MyLazyContentProvider(TableViewer viewer) {
        this.viewer = viewer;
    }

    public void dispose() {
    }

    public void inputChanged(Viewer viewer, Object oldInput, Object newInput) {
        this.elements = (Person[]) newInput;
    }

    public void updateElement(int index) {
        viewer.replace(elements[index], index);
    }
}

// create your table with the virtual flag
final TableViewer v = new TableViewer(shell, SWT.VIRTUAL);

// create TableColumns with label providers as before...
// ...

v.setContentProvider(new MyLazyContentProvider(v));

// special settings for the lazy content provider
v.setUseHashlookup(true);

// create the model and set it as input
Person[] model = createModel();
v.setInput(model);
// you must explicitly set the items count
v.setItemCount(model.length);
```

68.14. Alternative table implementations

It is possible to use other table implementations. Most notable is the [NatTable](#) implementation which is a flexible and powerful framework for creating tables, grids and trees that show great performance for a huge number of rows and columns.

Chapter 69. Exercise: Using TableViewer

69.1. Create a JFace TableViewer for the Todo items

In this exercise you create a JFace table in the `TodoOverviewPart` class which displays the `Todo` items.

69.2. Add dependencies

Add the `org.eclipse.jface` plug-in as a dependency to the `com.example.e4.rcp.todo` plug-in.

69.3. Create implementation

Create a JFace table implementation with the `summary` and `description` fields as table columns.

Your part should still have the "Load Data" button. Load the data into the viewer once this button is pressed.

You can use the following example code as base for this exercise.

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

import javax.annotation.PostConstruct;
import javax.inject.Inject;

import org.eclipse.e4.ui.di.Focus;
import org.eclipse.jface.viewers.ArrayContentProvider;
import org.eclipse.jface.viewers.ColumnLabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.jface.viewers.TableViewerColumn;
import org.eclipse.swt.SWT;
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.Table;

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;

public class TodoOverviewPart {
    @Inject
    ITodoService todoService;
    private TableViewer viewer;

    @PostConstruct
    public void createControls(Composite parent) {
        parent.setLayout(new GridLayout(1, false));

        Button button = new Button(parent, SWT.PUSH);
        button.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {
                // update the table content, whenever the button is pressed
                viewer.setInput(todoService.getTodos());
            }
        });
        button.setText("Load Data");
    }
}
```

```

viewer = new TableViewer(parent, SWT.MULTI | SWT.FULL_SELECTION);
Table table = viewer.getTable();
table.setHeaderVisible(true);
table.setLinesVisible(true);
table.setLayoutData(new GridData(SWT.FILL, SWT.FILL, true, true));
viewer.setContentProvider(ArrayContentProvider.getInstance());

// create column for the summary property
TableViewerColumn colSummary = new TableViewerColumn(viewer, SWT.NONE);
colSummary.setLabelProvider(new ColumnLabelProvider() {
    @Override
    public String getText(Object element) {
        Todo todo = (Todo) element;
        return todo.getSummary();
    }
});
colSummary.getColumn().setWidth(100);
colSummary.getColumn().setText("Summary");

// create column for description property
TableViewerColumn colDescription = new TableViewerColumn(viewer, SWT.NONE);
colDescription.setLabelProvider(new ColumnLabelProvider() {
    @Override
    public String getText(Object element) {
        Todo todo = (Todo) element;
        return todo.getDescription();
    }
});
colDescription.getColumn().setWidth(200);
colDescription.getColumn().setText("Description");

// initially the table is also filled
// the button is used to update the data if the model changes
viewer.setInput(todoService.getTodos());
}

@Focus
public void setFocus() {
    viewer.getControl().setFocus();
}
}

```

The result should look similar to the following screenshot.

To-Dos	
Load Data	
Summary	Description
SWT	Learn Widgets
JFace	Especially Viewers!
DI	@Inject looks interesting
OSGi	Services
Compatibility Layer	Run Eclipse 3.x

Tip

The "Load Data" button is later used, once you implement the possibility to delete data.

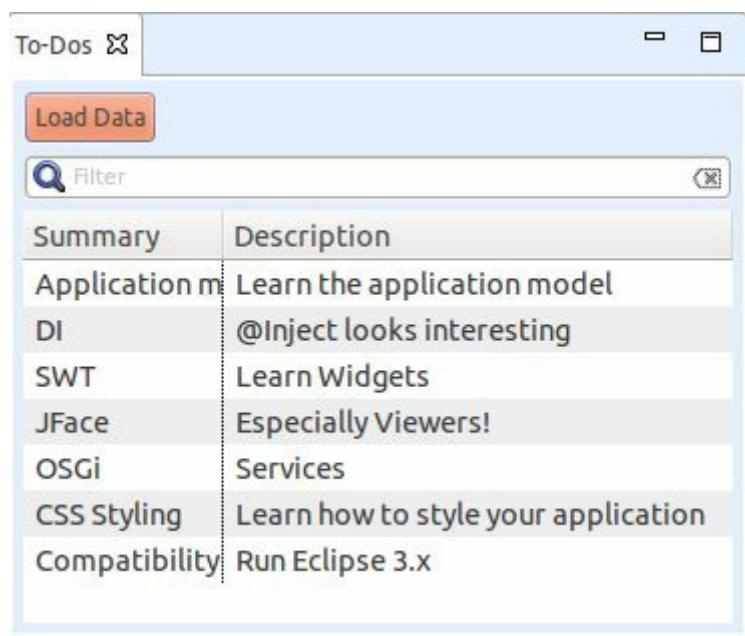
Chapter 70. Exercise: Using more viewer functionality

Note

The following exercise is optional.

70.1. Add a filter to the table

Add also a filter text box to the part. Use the text in the filter text box as filter for the displayed items in the table. Ensure that you only show the table entries which contain the text of the filter box in their summary or description field.



The following code snippet can be used as a starting point.

```
// TableViewer must be a field
private TableViewer viewer;

// define this new field
protected String searchString="";

@PostConstruct
public void createControls(Composite parent) {

    // EXISTING CODE TO BUILD THE UPDATE BUTTON
    // ...
    // ...

    Text search = new Text(parent, SWT.SEARCH | SWT.CANCEL
```

```

| SWT.ICON_SEARCH);

// assuming that GridLayout is used
search.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, false, false,
    1, 1));
search.setMessage("Filter");

// filter at every keystroke
search.addModifyListener(new ModifyListener() {
    @Override
    public void modifyText(ModifyEvent e) {
        Text source = (Text) e.getSource();
        searchString = source.getText();
        // trigger update in the viewer
        viewer.refresh();
    }
});

// SWT.SEARCH | SWT.CANCEL is not supported under Windows7 and
// so the following SelectionListener will not work under Windows7
search.addSelectionListener(new SelectionAdapter() {
    public void widgetDefaultSelected(SelectionEvent e) {
        if (e.detail == SWT.CANCEL) {
            Text text = (Text) e.getSource();
            text.setText("");
            //
        }
    }
});
}

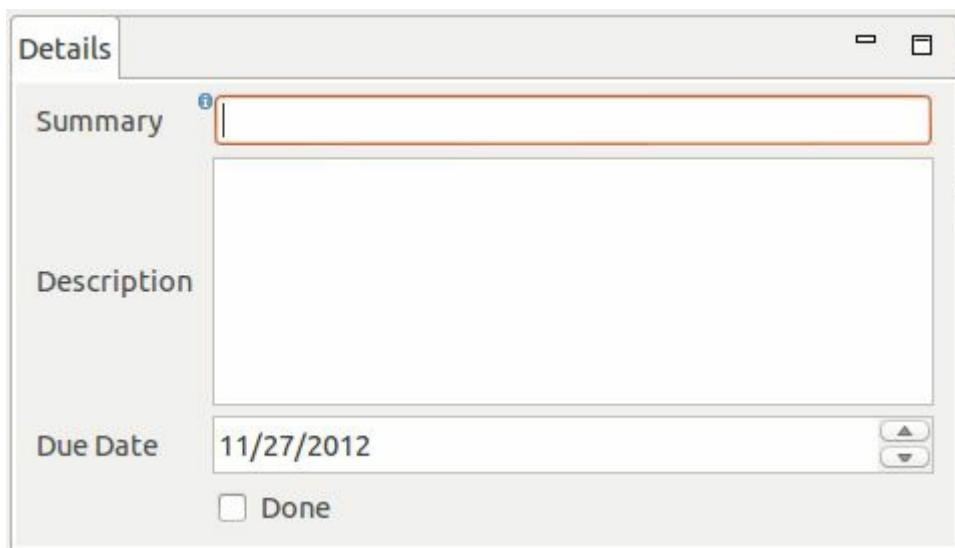
// EXISTING CODE WHICH DEFINES THE VIEWER.....
// ...
// ...
// ...

// add a filter which will search in the summary and description field
viewer.addFilter(new ViewerFilter() {
    @Override
    public boolean select(Viewer viewer, Object parentElement,
        Object element) {
        Todo todo = (Todo) element;
        return todo.getSummary().contains(searchString)
            || todo.getDescription().contains(searchString);
    }
});

```

70.2. Add a ControlDecoration to the TodoDetailsPart

Add a ControlDecoration to the *Summary* text fields in your TodoDetailsPart. This ControlDecoration should only be active if the field is selected and should provide a tooltip for the field, e.g., *Short description of the task.*

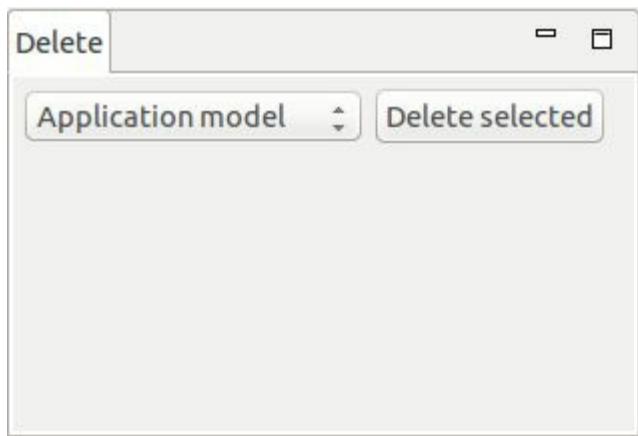


See [Section 66.3, “ControlDecoration”](#), for more information on how to achieve this.

Chapter 71. Optional exercise: Using ComboViewer

71.1. Target

In this exercise you create a new part which allows you to delete a `Todo` object from your model based on the selection in your `ComboViewer`. The final part should look similar to the following.



71.2. Implementation

Ensure that the `org.eclipse.jface` plug-in is available as a dependency to your `com.example.e4.rcp.todo` plug-in. If you followed the previous exercises, this should be already the case.

Create a new class called `TodoDeletionPart` and a new part in your application model which refers to this new class.

Rearrange the user interface so that the left area of your application contains two parts. The `TodoDeletionPart` part should be below the `TodoOverviewPart` part.

Tip

If a certain part of your user interface is not displayed, check the container data of your model elements in the application model. If one of the children of a container uses container data, all elements need to define container data.

Implement a `ComboViewer` which allows you to select a Todo item. Add a button to the part. If this button is pressed the `deleteTodo()` method of your `ITodoService` should be called. You can use the following example code as reference.

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

// NOTE: the import statements have been removed from
// this example

public class TodoDeletionPart {
    @Inject
    private ITodoService todoService;
    private ComboViewer viewer;

    @PostConstruct
    public void createControls(Composite parent) {
        parent.setLayout(new GridLayout(2, false));
        viewer = new ComboViewer(parent, SWT.READ_ONLY);
        viewer.setLabelProvider(new LabelProvider() {
            @Override
            public String getText(Object element) {
                Todo todo = (Todo) element;
                return todo.getSummary();
            }
        });
    }
}
```

```

    });
    viewer.setContentProvider(ArrayContentProvider.getInstance());

    List<Todo> todos = todoService.getTodos();
    updateViewer(todos);

    Button button = new Button(parent, SWT.PUSH);
    button.addSelectionListener(new SelectionAdapter() {
        @Override
        public void widgetSelected(SelectionEvent e) {
            ISelection selection = viewer.getSelection();
            IStructuredSelection sel = (IStructuredSelection) selection;
            if (sel.size() > 0) {
                Todo firstElement = (Todo) sel.getFirstElement();
                todoService.deleteTodo(firstElement.getId());
                updateViewer(todoService.getTodos());
            }
        }
    });
    button.setText("Delete selected");
}

private void updateViewer(List<Todo> todos) {
    viewer.setInput(todos);
    if (todos.size() > 0) {
        viewer.setSelection(new StructuredSelection(todos.get(0)));
    }
}

@Focus
public void focus() {
    viewer.getControl().setFocus();
}
}

```

Tip

If you face issues with the selection in the viewer or the deletion of a `Todo` object, ensure that `equals()` and `hashCode()` methods in `Todo` are correct (they should be based on the `id` field, see [Section 46.6, “Generate `toString\(\)`, `hashCode\(\)` and `equals\(\)` methods”](#)).

71.3. Validation

Delete entries in your data model. The `ComboViewer` should be updated automatically.

Validate that the number of entries in your data model has been reduced, e.g., by pressing the *Load Data* button in the `TodoOverviewPart` part.

Chapter 72. Using trees

72.1. Using viewers to display a tree

The `TreeViewer` class provides viewer support for displaying trees. The usage of this class is similar to the `TableViewer` class. The main difference is that the `TreeViewer` class requires a structured content provider. Typically your content provider has to implement the `ITreeContentProvider` interface to be used with your `TreeViewer` class.

The usage of the `TreeViewer` class is demonstrated in [Chapter 73, Optional exercises: Using TreeViewer.](#)

72.2. Selection and double-click listener

JFace allows you to access the SWT controls to define listeners on your viewer. For example you can add a `SelectionListener` implementation to the SWT control which is wrapped in the JFace object. The following code snippet demonstrates how to expand a tree with a mouse click.

```
// the viewer field is an already configured TreeViewer
Tree tree = (Tree) viewer.getControl();
tree.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
        TreeItem item = (TreeItem) e.item;
        if (item.getItemCount() > 0) {
            item.setExpanded(!item.getExpanded());
            // update the viewer
            viewer.refresh();
        }
    }
});
```

Viewers allows you to add certain listeners directly to them. The following example shows how to expand an instance of a `TreeViewer` with a double click.

```
viewer.addDoubleClickListener(new IDoubleClickListener() {
    @Override
    public void doubleClick(DoubleClickEvent event) {
        TreeViewer viewer = (TreeViewer) event.getViewer();
        IStructuredSelection thisSelection = (IStructuredSelection) event.getSelection();
        Object selectedNode = thisSelection.getFirstElement();
        viewer.setExpandedState(selectedNode,
            !viewer.getExpandedState(selectedNode));
    }
});
```

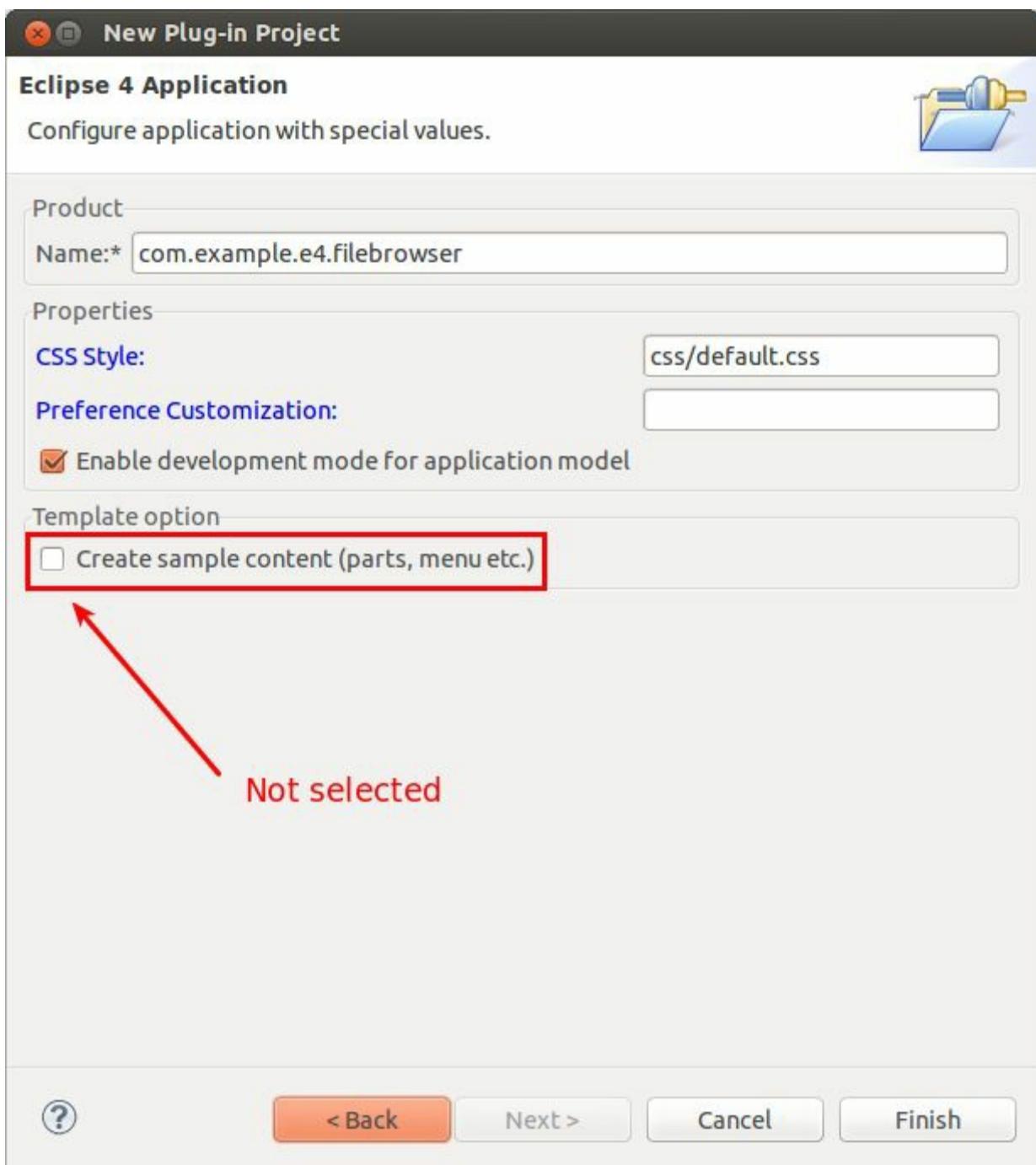
Chapter 73. Optional exercises: Using TreeViewer

73.1. Create a new application

This exercise is a stand-alone exercise and can be used to repeat the steps of creating an Eclipse 4 application.

Use the *Eclipse 4 wizard* from File → New → Other... → Eclipse 4 → Eclipse 4 Application Project to create a new Eclipse 4 application without sample data called *com.example.e4.filebrowser*.

The important selection in the last wizard page is highlighted in the following screenshot.



73.2. Add an image file

Download or create an icon called folder.png and place it into the "icons" folder of your plug-in.

You find an example icon under the following URL: [Folder icon](#).

This icon is taken from the following icon collection: [FamFamFam icons](#)

73.3. Create a part

Add a part stack with a part to your application model and display a TreeViewer in this part.

Implement a class for the `ITreeContentProvider` interface which allows you to browse the file system. Review the Javadoc of this class to understand the methods of this interface.

Also implement your custom `LabelProvider` for the tree.

Use `viewer.setInput(File.listRoots());` to set the initial input to the viewer.

The following listing contains an example implementation for this exercise. It assumes that you added the "folder.png" icon to the "icons" folder. It also demonstrates the usage of a `StyledLabelProvider`.

```
package com.example.e4.filebrowser.parts;

import java.io.File;
import java.net.URL;

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

import org.eclipse.core.runtime.FileLocator;
import org.eclipse.core.runtime.Path;
import org.eclipse.e4.ui.di.Focus;
import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.jface.viewers.ITreeContentProvider;
import org.eclipse.jface.viewers.StyledCellLabelProvider;
import org.eclipse.jface.viewers.StyledString;
import org.eclipse.jface.viewers.TreeViewer;
import org.eclipse.jface.viewers.Viewer;
import org.eclipse.jface.viewers.ViewerCell;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.widgets.Composite;
import org.osgi.framework.Bundle;
import org.osgi.framework.FrameworkUtil;

public class FileBrowserPart {
    private TreeViewer viewer;
    private Image image;

    @PostConstruct
    public void createControls(Composite parent) {
```

```

createImage();
viewer = new TreeViewer(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
viewer.setContentProvider(new ViewContentProvider());
viewer.setLabelProvider(new ViewLabelProvider());
viewer.setInput(File.listRoots());
}

private void createImage() {
    Bundle bundle = FrameworkUtil.getBundle(ViewLabelProvider.class);
    URL url = FileLocator.find(bundle, new Path("icons/folder.png"), null);
    ImageDescriptor imageDcr = ImageDescriptor.createFromURL(url);
    this.image = imageDcr.createImage();
}

class ViewContentProvider implements ITreeContentProvider {
    public void inputChanged(Viewer v, Object oldInput, Object newInput) {
    }

    @Override
    public void dispose() {
    }

    @Override
    public Object[] getElements(Object inputElement) {
        return (File[]) inputElement;
    }

    @Override
    public Object[] getChildren(Object parentElement) {
        File file = (File) parentElement;
        return file.listFiles();
    }

    @Override
    public Object getParent(Object element) {
        File file = (File) element;
        return file.getParentFile();
    }

    @Override
    public boolean hasChildren(Object element) {
        File file = (File) element;
        if (file.isDirectory()) {
            return true;
        }
        return false;
    }
}

class ViewLabelProvider extends StyledCellLabelProvider {
    @Override
    public void update(ViewerCell cell) {
        Object element = cell.getElement();

```

```

StyledString text = new StyledString();
File file = (File) element;
if (file.isDirectory()) {
    text.append(getFileName(file));
    cell.setImage(image);
    String[] files = file.list();
    if (files != null) {
        text.append(" (" + files.length + ") ",
                    StyledString.COUNTER_STYLER);
    }
} else {
    text.append(getFileName(file));
}
cell.setText(text.toString());
cell.setStyleRanges(text.getStyleRanges());
super.update(cell);

}

private String getFileName(File file) {
    String name = file.getName();
    return name.isEmpty() ? file.getPath() : name;
}

@Focus
public void setFocus() {
    viewer.getControl().setFocus();
}

@PreDestroy
public void dispose() {
    image.dispose();
}
}

```

Link from your part in the application model to your new class.

73.4. Validating

Start your new application ensure that you see the content of your file system in your tree.

Part XIV. Defining menus and toolbars

Chapter 74. Menu and toolbar application objects

74.1. Adding menu and toolbar entries

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

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. They contain 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 *Handled MenuItem* and *Handled ToolItem* model elements, you refer to a *Command* model element. The Command model elements are described in [Section 74.2, “What are commands and handlers?”](#).

The application model also supports the creation of menus at runtime via the *DynamicMenuContribution* model elements.

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

Menus and toolbars support separators and can have submenus.

74.2. What are commands and handlers?

The Eclipse application model allows you to specify *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 element points to a class via the `contributionURI` property of the handler. This attribute is displayed as *Class URI* in the model editor. Such a class is called *handler class* in this book.

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

Tip

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

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

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

74.4. Standard commands

Eclipse 4 does not provide standard commands, i.e., you have to create all required commands in your application model.

74.5. 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 letters.

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

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

Table 74.1. Default IDs for commonly used commands

Command ID

Save	<code>org.eclipse.ui.file.save</code>
Save All	<code>org.eclipse.ui.file.saveAll</code>
Undo	<code>org.eclipse.ui.edit.undo</code>
Redo	<code>org.eclipse.ui.edit.redo</code>
Cut	<code>org.eclipse.ui.edit.cut</code>
Copy	<code>org.eclipse.ui.edit.copy</code>
Paste	<code>org.eclipse.ui.edit.paste</code>
Delete	<code>org.eclipse.ui.edit.delete</code>
Import	<code>org.eclipse.ui.file.import</code>
Export	<code>org.eclipse.ui.file.export</code>
Select All	<code>org.eclipse.ui.edit.selectAll</code>
About	<code>org.eclipse.ui.help.aboutAction</code>
Preferences	<code>org.eclipse.ui.window.preferences</code>
Exit	<code>org.eclipse.ui.file.exit</code>

Chapter 75. Dependency injection for handler classes

75.1. Handler classes and their behavior annotations

Direct menu, tool items and handler model elements point to a class. This class uses behavior annotations to mark 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 75.1. Behavior annotations for handler classes

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

Annotation Description

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

Warning

According to the Javadoc only one method is allowed to be annotated with `@Execute`. The same applies for `@CanExecute`. While the framework currently does not complain about several methods marked with these annotation, you should avoid this, as it is otherwise undefined which method is called.

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;

// import statements cut out
// ..

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

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

75.2. Which context is used for a handler class?

The handler class is executed with the `IEclipseContext` in which the handler is called, i.e., the one which is currently marked as active in the window. In most common cases this is 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.

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

Warning

To ensure that you get the expected values from the active context injected into your handler class, NEVER use field or constructor injection in it.

75.3. Scope of handlers

If a command is selected, the runtime will determine the relevant handlers for the command.

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

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

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

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.

75.4. Evaluation of @CanExecute

The methods annotated with `@CanExecute` are called by the framework if a change in the context happens, i.e., if the context is modified or if the active context changes. Application code can request the evaluation of the `@CanExecute` methods by the framework by sending out an event via the event broker.

```
// evaluate all @CanExecute methods
eventBroker.send(UIEvents.REQUEST_ENABLEMENT_UPDATE_TOPIC, UIEvents.ALL_ELEMENT_

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

//See https://bugs.eclipse.org/bugs/show\_bug.cgi?id=427465 for details
```

75.5. Learn about the event system

See [Part XXIV, “Event service for message communication”](#) to learn how to send out events in your Eclipse application.

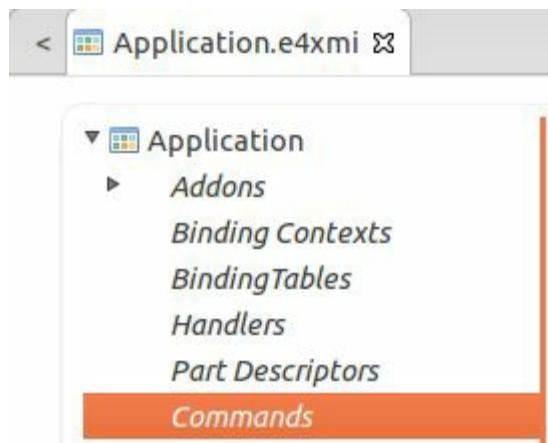
Chapter 76. Exercise: Adding a menu and menu entries

76.1. Target of this exercise

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

76.2. Create command model elements

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



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

Table 76.1. Commands

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

76.3. Creating the handler classes

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

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

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

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

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

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

- `SaveAllHandler`
- `ExitHandler`
- `NewTodoHandler`
- `RemoveTodoHandler`
- `TestHandler`

76.4. Creating handler model elements

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

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

Table 76.2. Handlers

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

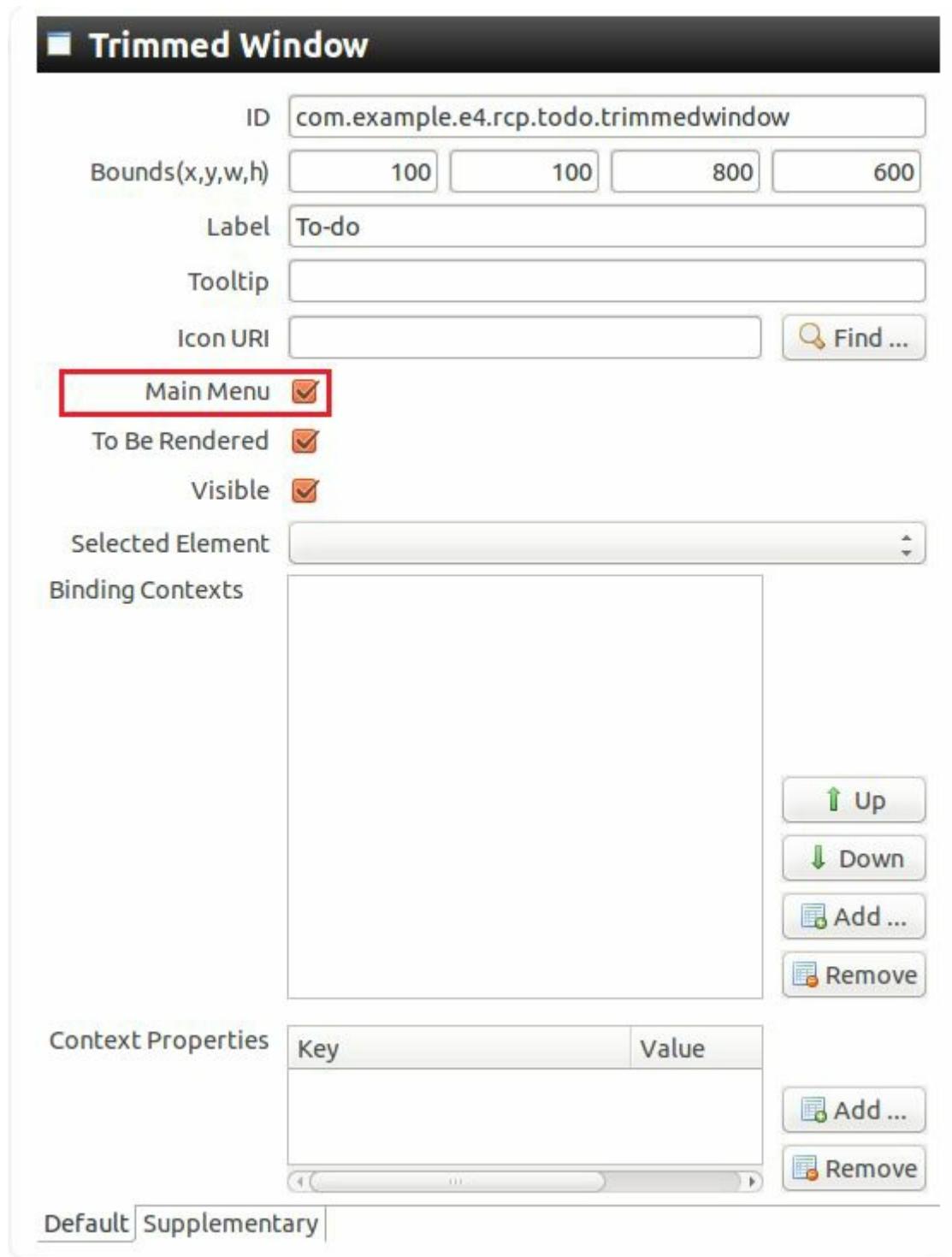
The application model editor shows both the name and the ID of the command. The class URI follows the `bundleclass://` schema, the table only defines the class name to make the table more readable. For example, for the save handler this looks like the following:

```
bundleclass://com.example.e4.rcp.todo/com.example.e4.rcp.todo.handlers.SaveAllHa
```



76.5. Adding a menu

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



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

Warning

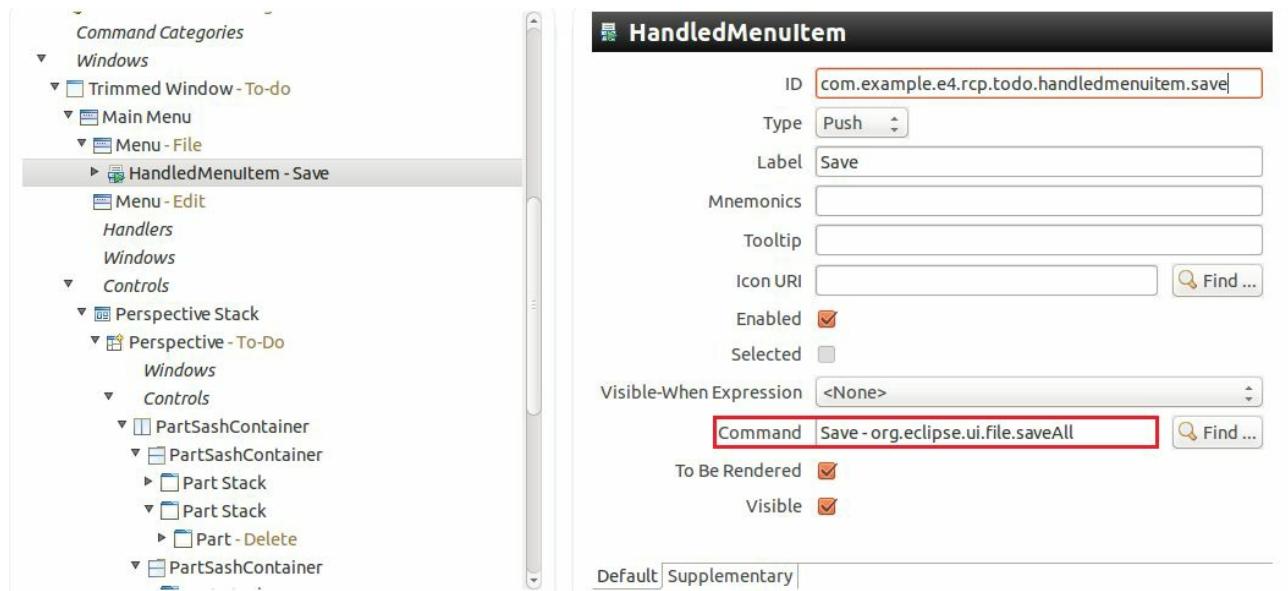
Ensure that this ID of the main menu is correct. You use it later to contribute another menu entry via another plug-in.

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

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



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



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

Add all other commands to the Edit menu.

76.6. Implement a handler class for exit

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

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

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

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

76.7. Validating

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

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

76.8. Possible issue: Exit menu entry on a MacOS

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

Chapter 77. Exercise: Adding a toolbar

77.1. Target of this exercise

In this exercise you add a toolbar to your application.

77.2. Adding a toolbar

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



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

Set the label for this entry to *Save*.

Application.e4xmi

The screenshot shows the Eclipse RCP Application Editor interface. On the left, there is a tree view of command contributions under the 'Command Categories' section. The 'Handled Tool Item - Save' item is selected and highlighted with an orange background. On the right, the properties for this selected item are displayed in a configuration form.

Handled Tool Item

ID	com.example.e4.rcp.todo.handledtoolitem.0
Type	Push
Label	Save
Accessibility Phrase	
Tooltip	
Icon URI	
Menu	<input type="checkbox"/>
Enabled	<input checked="" type="checkbox"/>
Selected	<input type="checkbox"/>
Visible-When Expression	<None>
Command	Save - org.eclipse.ui.file.saveAll
To Be Rendered	<input checked="" type="checkbox"/>
Visible	<input checked="" type="checkbox"/>
Default	Supplementary

Part Descriptors
Menu Contributions
Toolbar Contributions
Trim Contributions
Snippets

Form XMI

77.3. Validating

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

Chapter 78. View, popup and dynamic menus

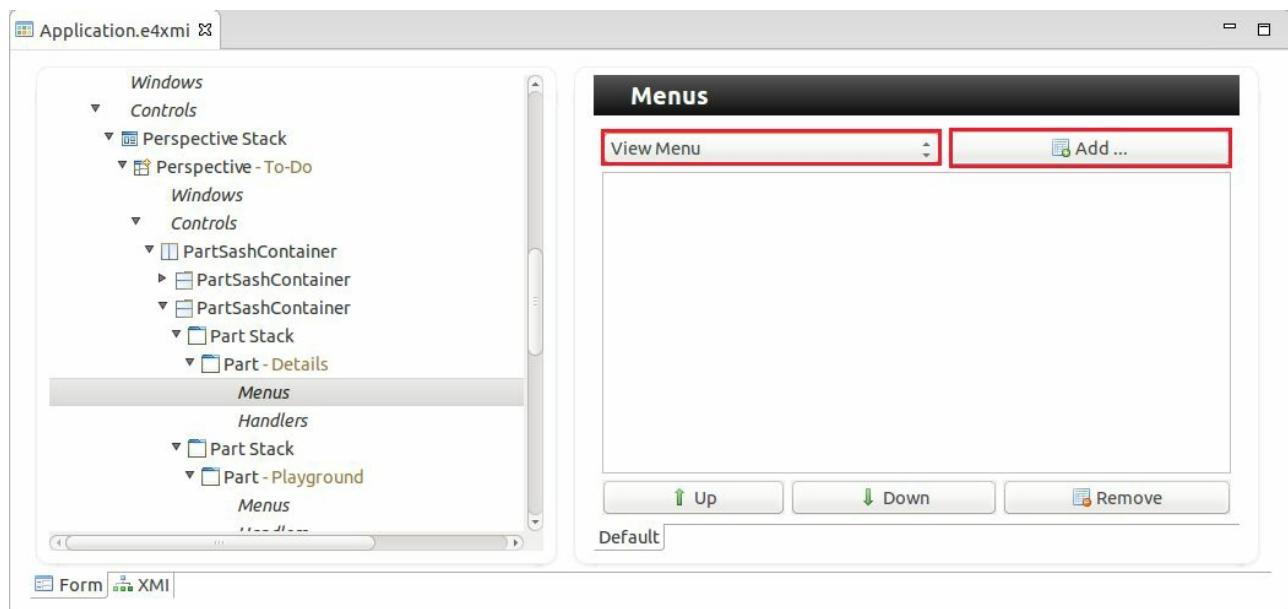
78.1. View menus

One menu in a part can be defined as a *view menu*.

Note

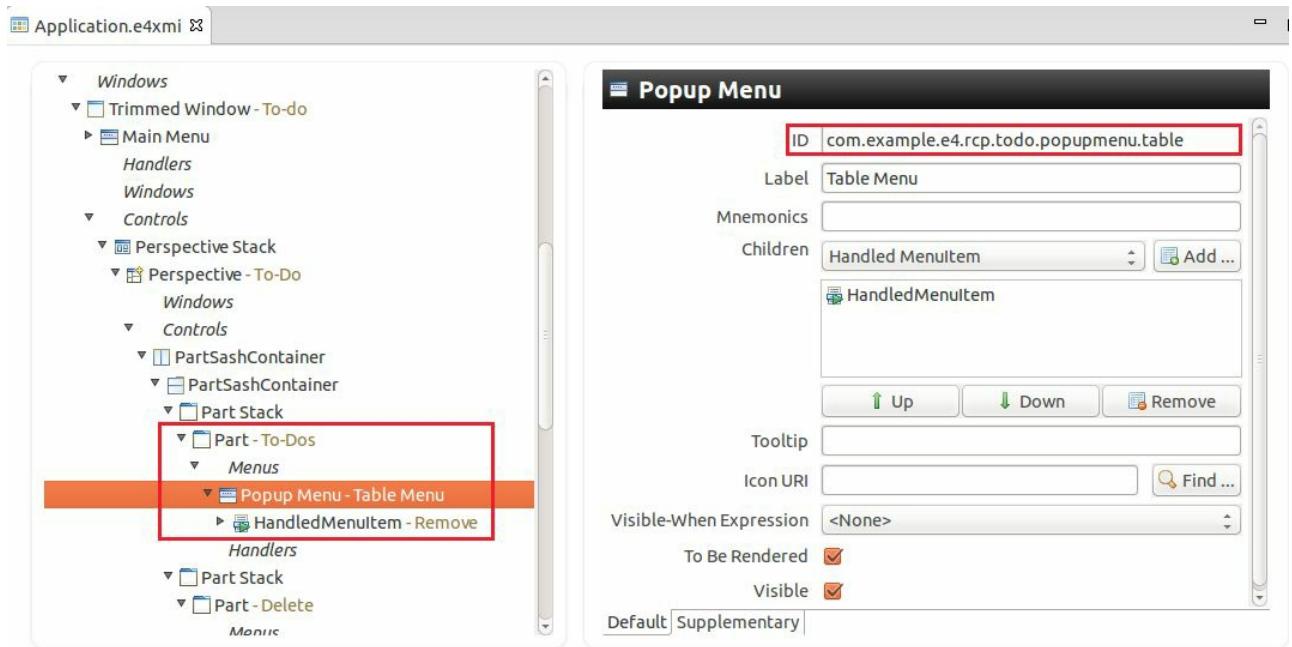
Please note that the default Eclipse (renderer) framework supports only one menu for a part.

To add a view menu entry, select the *Menus* entry under the part and append a *ViewMenu* model entry to it.

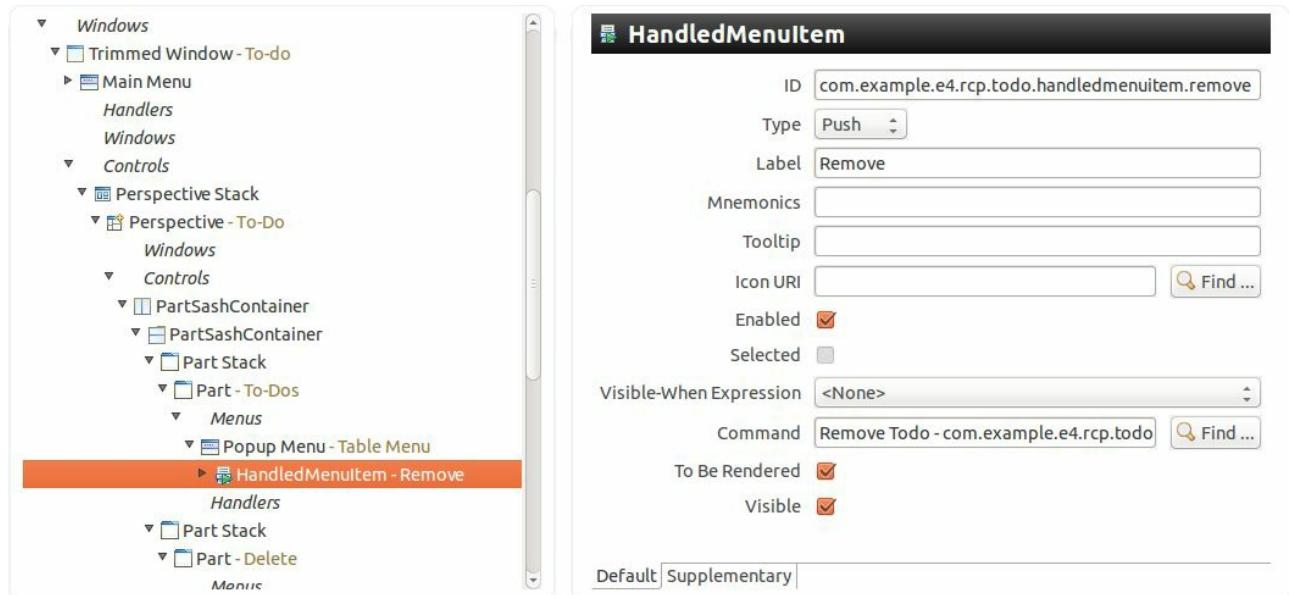


78.2. Popup menu (context menu)

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



The popup menu contains entries, as, for example, a HandledMenuItem.



After this the pop menu can be assigned to an SWT control with the `EMenuService` service which can be accessed via dependency injection. This class provides the `registerContextMenu(control, id)` method for this

purpose. The *id* parameter of the `registerContextMenu` method must be the ID attribute of your *Popup Menu* model element.

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

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

import javax.annotation.PostConstruct;

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

public class TodoOverviewPart {

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

        // more code

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

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

78.3. Dynamic menu and toolbar entries

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

This model element points to a class in which you annotate a method with the annotation. The annotated method is called if the user selects the user interface element.

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

In these methods you can dynamically adjust the application model.

78.4. Creating dynamic menu entries

See [Part XX, “Model service and model modifications at runtime”](#) for information about creating and changing application model elements dynamically.

Chapter 79. Exercise: Add a context menu to a table

Tip

This exercise is optional.

79.1. Target

Implement a context menu for the table in your `TodoOverviewPart` and connect it with the command for deleting todo items.

79.2. Dependencies

Add the following plug-ins as dependencies to your application plug-in.

- org.eclipse.e4.ui.workbench.swt
- org.eclipse.e4.ui.model.workbench

79.3. Implementation

For the implementation of a popup menu, check [Section 78.2, “Popup menu \(context menu\)”](#) to learn how to create popup (context) menus and assign them to SWT controls.

Change the `RemoveTodoHandler` so that it writes, "Soon, I will really delete your todo".

Note

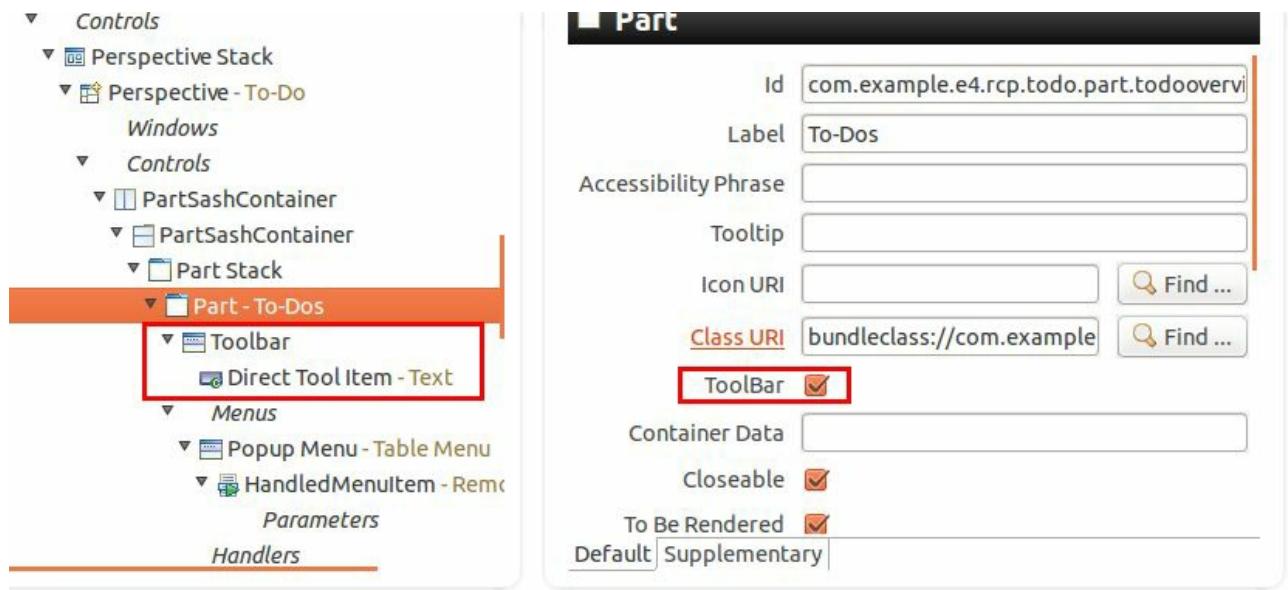
You will change the `RemoveTodoHandler` to delete the selected Todo once you have learned about the selection service in [Chapter 99, *Exercise: Selection service*](#).

Chapter 80. Toolbars, ToolControls and drop-down tool items

80.1. Adding toolbars to parts

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

Such an example setup is displayed in the following screenshot.



80.2. ToolControls

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

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

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

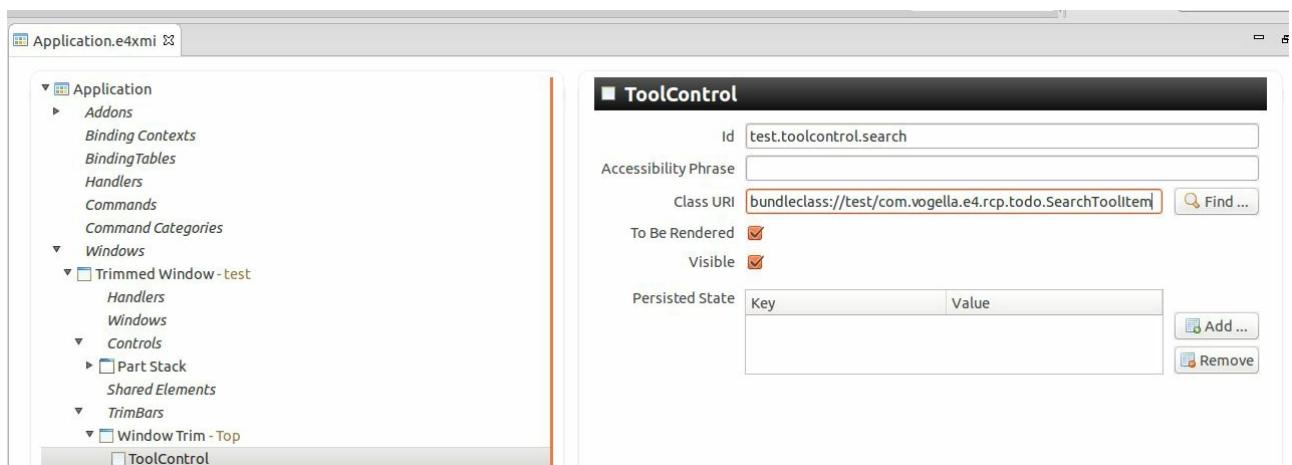
import javax.annotation.PostConstruct;

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

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

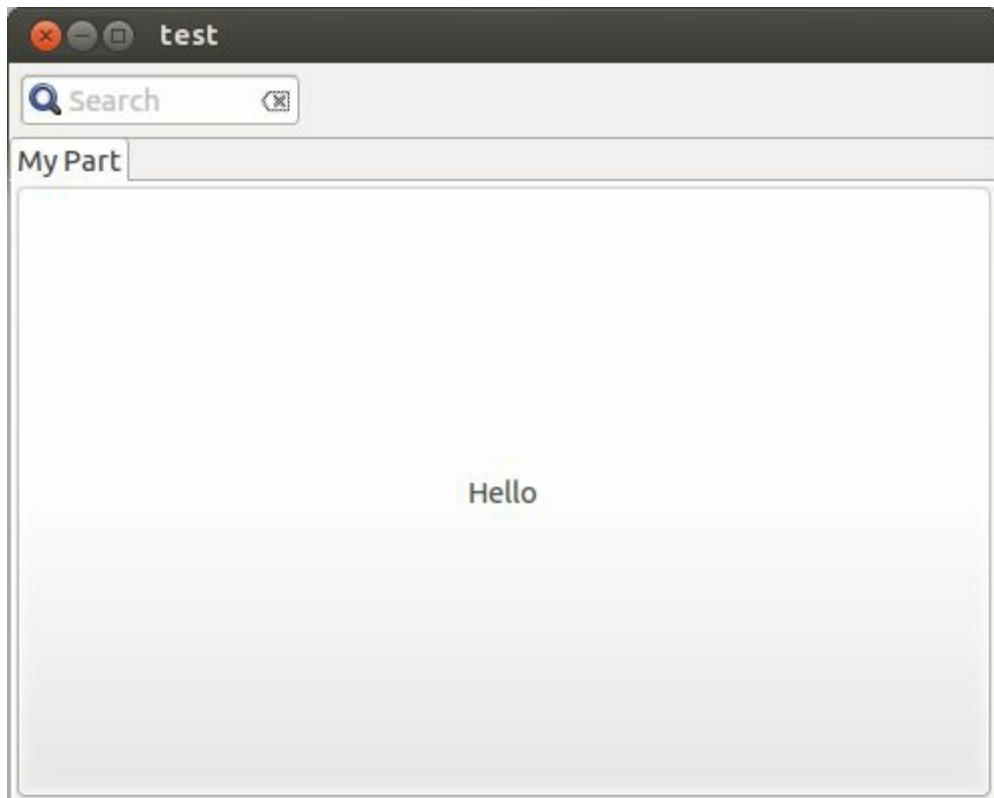
    }
}
```

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



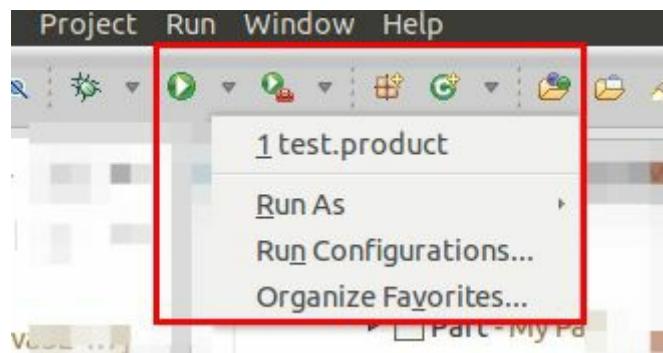
The following screenshot shows this `ToolControl` used in an example RCP

application.



80.3. Drop-down tool items

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

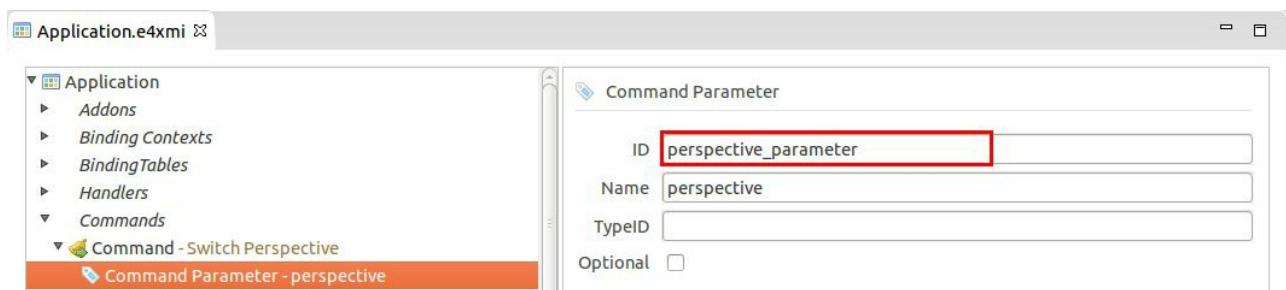


Chapter 81. More on commands and handlers

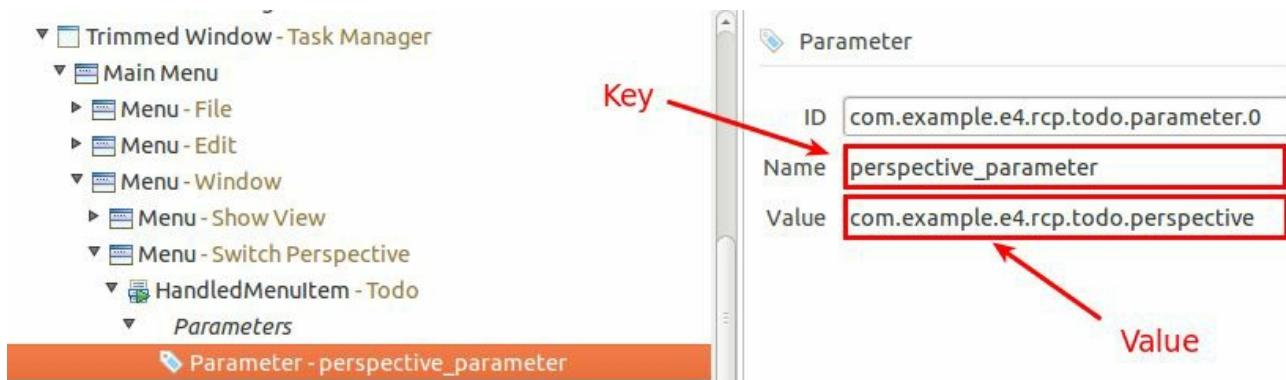
81.1. Passing parameters to commands

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

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



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



Warning

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

The screenshot shows the Eclipse RCP Workbench interface. On the left, there is a tree view under the 'Application' node. A red box highlights the 'Command Parameter - perspective' entry under the 'Commands' node. To the right, two dialog boxes are displayed side-by-side. The top dialog is titled 'Command Parameter' and contains fields: 'ID' (perspective_parameter), 'Name' (perspective), 'TypeID' (empty), and 'Optional' (unchecked). A green arrow points from the 'Name' field in this dialog down to the 'Name' field in the bottom dialog. The bottom dialog is titled 'Parameter' and contains fields: 'ID' (com.example.e4.rcp.todo.parameter.0), 'Name' (perspective_parameter), and 'Value' (com.example.e4.rcp.todo.perspective).

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

```

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

import java.util.List;

import javax.inject.Named;

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

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

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

        // switch to perspective with the ID if found
        if (!perspectives.isEmpty()) {
            partService.switchPerspective(perspectives.get(0));
        }
    }
}

```

```
    }  
}
```

Tip

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

```
package com.example.e4.rcp.todo.handlers;  
  
import javax.inject.Named;  
  
import org.eclipse.e4.core.di.annotations.Execute;  
  
public class TestHandlerWithCommandInjected {  
    private String parametername = "com.example.e4.rcp.todo" +  
        ".commandparameter.input";  
    @Execute  
    public void execute(ParameterizedCommand command) {  
        Object queryId = command.getParameterMap().get(parametername);  
        // more...  
    }  
}
```

81.2. Usage of core expressions

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

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

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

HandledMenuItem

Id	com.example.e4.rcp.todo.menusitems.test
Type	Push
Label	Test
Mnemonics	
Tooltip	
Icon URI	<input type="text"/> Find ...
Enabled	<input checked="" type="checkbox"/>
Selected	<input type="checkbox"/>
Visible-When Expression	CoreExpression
Command	For testing - com.example.e4.rcp.todo.test
To Be Rendered	<input checked="" type="checkbox"/>
Visible	<input checked="" type="checkbox"/>

Default **Supplementary**

Part Descriptors

- ▶ Commands
- Command Categories
- ▼ Windows
 - Trimmed Window - To-do
 - Main Menu
 - Menu - File
 - Menu - Edit
 - HandledMenuItem - New Todo
 - HandledMenuItem - Remove Todo
 - HandledMenuItem - Test

Core Expression

Expression Id	com.example.e4.rcp.todo.selectionset	Find ...
---------------	--------------------------------------	----------

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

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

81.3. Evaluate your own values in core expressions

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

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

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

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

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

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



81.4. Example for dynamic model contributions

See [Chapter 104, Exercise: Creating dynamic menu entries](#) for an example how to create dynamically menu entries.

Part XV. Using key bindings

Chapter 82. Key bindings

82.1. Using key bindings in your application

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 *Binding Context* node of your application model.

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

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

Warning

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

82.2. JFace default values for binding contexts

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

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

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

Table 82.1. Default BindingContext values

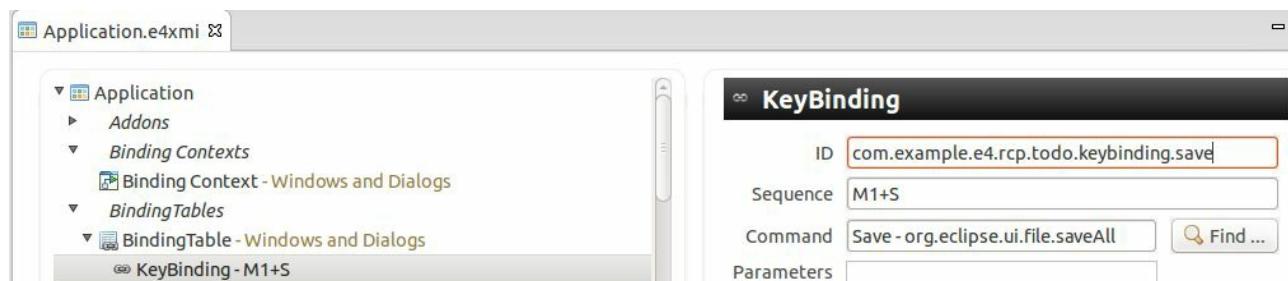
Context ID	Description
<code>org.eclipse.ui.contexts.dialogAndWindow</code>	Key bindings valid for dialogs and windows
<code>org.eclipse.ui.contexts.dialog</code>	Key bindings valid for dialogs
<code>org.eclipse.ui.contexts.window</code>	Key bindings valid for windows

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.

82.3. Define Shortcuts

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

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



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

Table 82.2. 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.

82.4. Activate bindings

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

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

82.5. Key bindings for a part

You can assign a specific binding context to be active while a part is active.

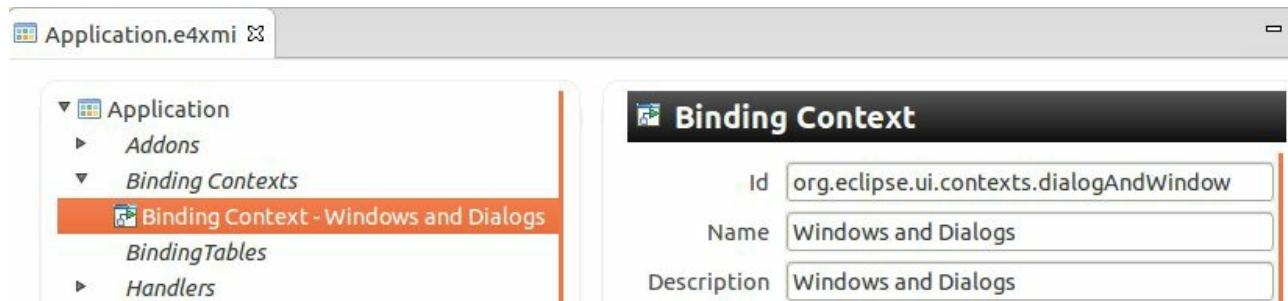
The screenshot shows the 'Part' configuration dialog. At the top, there are several input fields: 'Id' (empty), 'Label' ('Todo Overview'), 'Accessibility Phrase' (empty), 'Tooltip' (empty), 'Icon URI' (empty) with a 'Find ...' button, 'Class URI' ('bundleclass://com.example.e4.rcp.todo/com.e') with a 'Find ...' button, 'ToolBar' (checkbox unchecked), 'Container Data' (empty), 'Closeable' (checkbox unchecked), 'To Be Rendered' (checkbox checked), and 'Visible' (checkbox checked). Below these is a section titled 'Binding Contexts' which contains a table with one row labeled 'Binding Context - test'. To the right of the table are four buttons: 'Up', 'Down', 'Add ...', and 'Remove'. At the bottom, there is a table with two rows: 'Persisted State' (checkboxes for 'Key' and 'Value') and 'Default' (checkboxes for 'Supplementary' and 'Global').

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

Chapter 83. Exercise: Define key bindings

83.1. Create binding context entries

Select the *Binding Contexts* entry in your application model. Define a new entry with the `org.eclipse.ui.contexts.dialogAndWindow` ID. As name use "Windows and Dialogs".

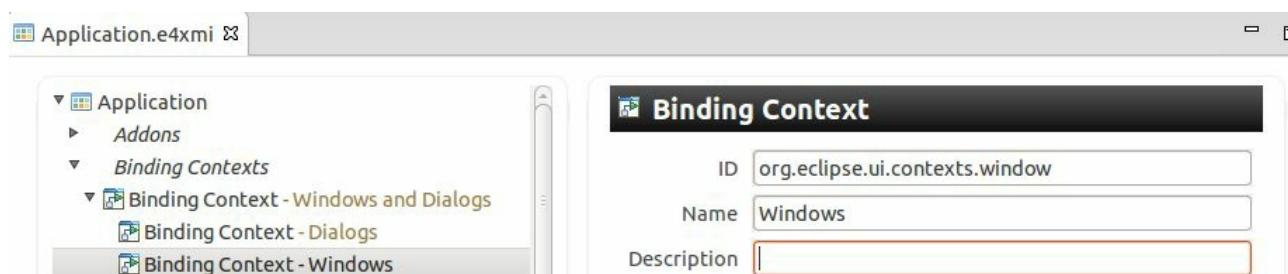


Afterwards add the following sub-entries:

Table 83.1. Binding context entries

ID	Name
<code>org.eclipse.ui.contexts.dialog</code>	Dialogs
<code>org.eclipse.ui.contexts.window</code>	Windows

The resulting entry is depicted in the following screenshot.



Warning

The name attribute of a binding context entry is not allowed to be empty.

83.2. Create key bindings for a BindingContext

Create a key binding for your Save command to the **M1+S** shortcut. Afterwards create a keybinding for your Exit command to the **M1+X** shortcut.

The required entries are depicted in the following screenshots.

The image consists of two side-by-side screenshots of the Eclipse RCP application editor. The left screenshot shows the 'Application.e4xmi' editor with a tree view of 'Binding Contexts'. One specific binding context is selected, highlighted with a red border. The right screenshot shows the 'BindingTable' editor, which contains fields for 'Id' (set to 'com.example.e4.rcp.todo.bindingtable.mybinding'), 'Context Id' (set to 'Windows and Dialogs'), and a 'Keybindings' section containing a single entry labeled 'KeyBinding'. The bottom screenshot shows the 'KeyBinding' editor, which contains fields for 'Id' (set to 'com.example.e4.rcp.todo.keybinding.save'), 'Sequence' (set to 'M1+S'), 'Command' (set to 'Save - org.eclipse.ui.file.saveAll'), and a 'Find ...' button. The 'Parameters' field is empty.

Run your program and test it.

Part XVI. Dialogs and wizards

Chapter 84. Dialogs

84.1. Dialogs in Eclipse

Eclipse allows you to use *dialogs* to prompt the user for additional information or provide the user with feedback. The Eclipse platform offers several standard dialogs via SWT and JFace.

84.2. SWT dialogs

SWT provides an API to use the native dialogs from the OS platform. The default SWT dialogs are listed below.

- `ColorDialog` - for selecting a color
- `DirectoryDialog` - for selecting a directory
- `FileDialog` - for selecting a file
- `FontDialog` - for selecting a font
- `MessageBox` - for opening a message dialog

The following code demonstrates the usage of the `MessageBox` class to open a message dialog.

```
// create a dialog with ok and cancel buttons and a question icon
MessageBox dialog =
    new MessageBox(shell, SWT.ICON_QUESTION | SWT.OK| SWT.CANCEL);
dialog.setText("My info");
dialog.setMessage("Do you really want to do this?");

// open dialog and await user selection
returnCode = dialog.open();
```

84.3. JFace Dialogs

Dialogs from JFace

JFace contains several frequently used dialogs which are not based on the native dialogs as well as a framework for building custom dialogs.

Note

Even though JFace dialogs are not native, they follow the native platform semantics for things like the button order.

MessageDialog

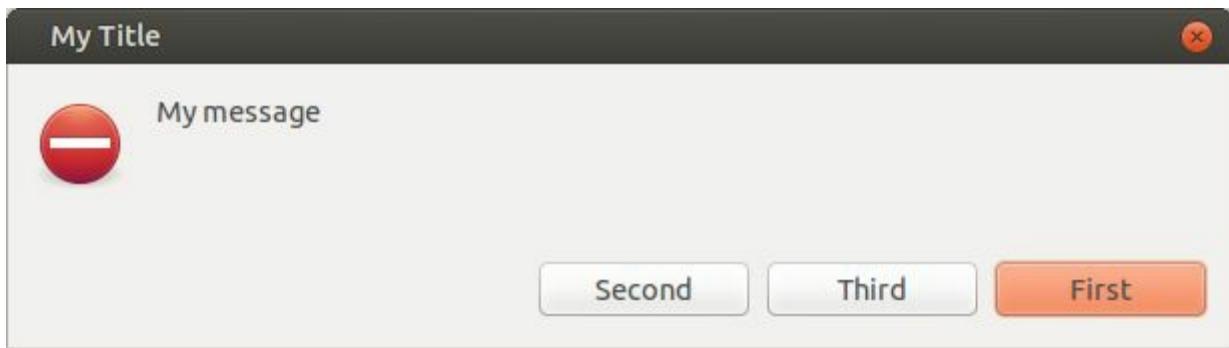
The `MessageDialog` class provides static methods to open commonly used dialogs, for example an info or a warning dialog. The following code demonstrates the usage of these static methods.

```
// standard message dialogs
MessageDialog.openConfirm(shell, "Confirm", "Please confirm");
MessageDialog.openError(shell, "Error", "Error occurred");
MessageDialog.openInformation(shell, "Info", "Info for you");
MessageDialog.openQuestion(shell, "Question", "Really, really?");
MessageDialog.openWarning(shell, "Warning", "I am warning you!");
```

The `MessageDialog` class also allows the customization of the buttons in the dialog. The following code demonstrates its usage.

```
// customized MessageDialog with configured buttons
MessageDialog dialog = new MessageDialog(shell, "My Title", null,
    "My message", MessageDialog.ERROR, new String[] { "First",
    "Second", "Third" }, 0);
int result = dialog.open();
System.out.println(result);
```

If you open this dialog, it looks similar to the following screenshot.



Several of these dialogs return the user selection, e.g. the `openConfirm()` method returns true if the user selected the *OK* button. The following example code prompts the user for confirmation and handles the result.

```
boolean result =  
    MessageDialog.openConfirm(shell, "Confirm", "Please confirm");  
  
if (result){  
    // OK Button selected do something  
} else {  
    // Cancel Button selected do something  
}
```

ErrorDialog

The `ErrorDialog` class can be used to display one or more errors to the user. If an error contains additional detailed information then a button is automatically added, which shows or hides this information when pressed by the user.

The following snippet shows a handler class which uses this dialog.

```
package com.example.e4.rcp.todo.handlers;  
  
import java.util.ArrayList;  
import java.util.List;  
  
import org.eclipse.core.runtime.IStatus;  
import org.eclipse.core.runtime.MultiStatus;  
import org.eclipse.core.runtime.Status;  
import org.eclipse.e4.core.di.annotations.Execute;  
import org.eclipse.e4.ui.model.application.ui.basic.MWindow;  
import org.eclipse.jface.dialogs.ErrorDialog;  
import org.eclipse.swt.widgets.Shell;  
  
public class ShowErrorDialogHandler {  
    @Execute  
    public void execute(final Shell shell, MWindow window) {  
        // create exception on purpose to demonstrate ErrorDialog  
        try {
```

```

        String s = null;
        System.out.println(s.length());
    } catch (NullPointerException e) {
        // build the error message and include the current stack trace
        MultiStatus status = createMultiStatus(e.getLocalizedMessage(), e);
        // show error dialog
        ErrorDialog.openError(shell, "Error", "This is an error", status);
    }
}

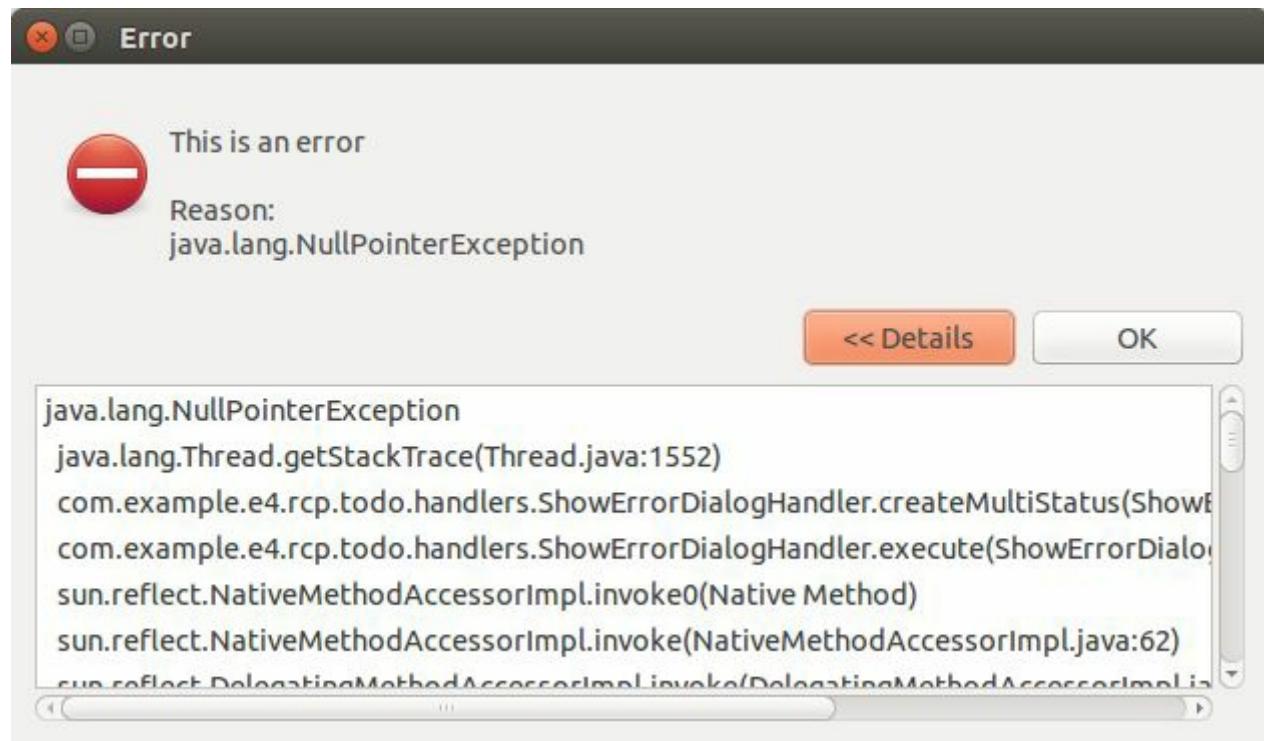
private static MultiStatus createMultiStatus(String msg, Throwable t) {
    List<Status> childStatuses = new ArrayList<Status>();
    StackTraceElement[] stackTraces = Thread.currentThread().getStackTrace();

    for (StackTraceElement stackTrace: stackTraces) {
        Status status = new Status(IStatus.ERROR,
            "com.example.e4.rcp.todo", stackTrace.toString());
        childStatuses.add(status);
    }

    MultiStatus ms = new MultiStatus("com.example.e4.rcp.todo",
        IStatus.ERROR, childStatuses.toArray(new Status[] {}),
        t.toString(), t);
    return ms;
}
}

```

If you trigger this handler, the dialog shows the exception messages and the detail page contains the stacktrace, as depicted in the following screenshot.



Creating a custom dialog

The `org.eclipse.jface.dialogs.Dialog` class can be extended to create your own dialog implementation. This class creates an area in which you can place controls and add an *OK* and *Cancel* button (or other custom buttons).

Your class needs to implement the `createDialogArea()` method. This method gets a `Composite` which expects to get a `GridData` object assigned as its layout data. Via the `super.createDialogArea(parent)` method call, you can create a `Composite` to which you can add your controls. This is demonstrated by the following example code.

```
package com.vogella.plugin.dialogs.custom;

import org.eclipse.jface.dialogs.Dialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Shell;

public class MyDialog extends Dialog {

    public MyDialog(Shell parentShell) {
        super(parentShell);
    }

    @Override
    protected Control createDialogArea(Composite parent) {
        Composite container = (Composite) super.createDialogArea(parent);
        Button button = new Button(container, SWT.PUSH);
        button.setLayoutData(new GridData(SWT.BEGINNING, SWT.CENTER, false,
                false));
        button.setText("Press me");
        button.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {
                System.out.println("Pressed");
            }
        });
    }

    return container;
}

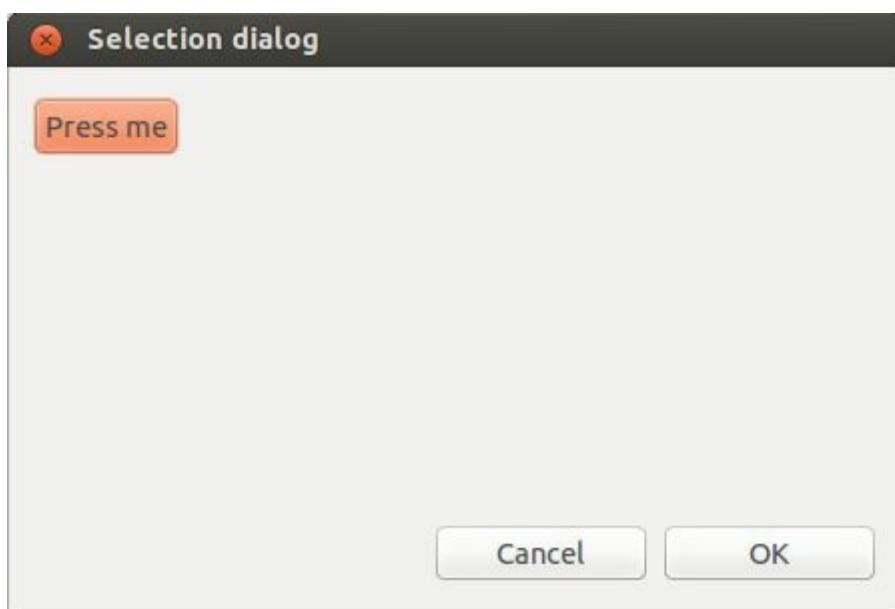
// overriding this methods allows you to set the
// title of the custom dialog
@Override
```

```
protected void configureShell(Shell newShell) {  
    super.configureShell(newShell);  
    newShell.setText("Selection dialog");  
}  
  
@Override  
protected Point getInitialSize() {  
    return new Point(450, 300);  
}  
  
}
```

Tip

The example code demonstrates how to set the title of your custom dialog via the `configureShell()` method.

If you open this dialog it looks similar to the following screenshot.



TitleAreaDialog

You can also implement your custom dialog based on the `TitleAreaDialog` class.

`TitleAreaDialog` has a reserved space for providing feedback to the user. You can set the text in this space via the `setMessage()` and `setErrorMessageBox()` methods.

The following example shows a custom defined `TitleAreaDialog`.

```

package com.vogella.plugin.dialogs.custom;

import org.eclipse.jface.dialogs.IMessageProvider;
import org.eclipse.jface.dialogs.TitleAreaDialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class MyTitleAreaDialog extends TitleAreaDialog {

    private Text txtFirstName;
    private Text lastNameText;

    private String firstName;
    private String lastName;

    public MyTitleAreaDialog(Shell parentShell) {
        super(parentShell);
    }

    @Override
    public void create() {
        super.create();
        setTitle("This is my first custom dialog");
        setMessage("This is a TitleAreaDialog", IMessageProvider.INFORMATION);
    }

    @Override
    protected Control createDialogArea(Composite parent) {
        Composite area = (Composite) super.createDialogArea(parent);
        Composite container = new Composite(area, SWT.NONE);
        container.setLayoutData(new GridData(SWT.FILL, SWT.FILL, true, true));
        GridLayout layout = new GridLayout(2, false);
        container.setLayout(layout);

        createFirstName(container);
        createLastName(container);

        return area;
    }

    private void createFirstName(Composite container) {
        Label lbtFirstName = new Label(container, SWT.NONE);
        lbtFirstName.setText("First Name");

        GridData dataFirstName = new GridData();
        dataFirstName.grabExcessHorizontalSpace = true;
        dataFirstName.horizontalAlignment = GridData.FILL;
    }
}

```

```

txtFirstName = new Text(container, SWT.BORDER);
txtFirstName.setLayoutData(dataFirstName);
}

private void createLastName(Composite container) {
    Label lbtLastName = new Label(container, SWT.NONE);
    lbtLastName.setText("Last Name");

    GridData dataLastName = new GridData();
    dataLastName.grabExcessHorizontalSpace = true;
    dataLastName.horizontalAlignment = GridData.FILL;
    lastNameText = new Text(container, SWT.BORDER);
    lastNameText.setLayoutData(dataLastName);
}

@Override
protected boolean isResizable() {
    return true;
}

// save content of the Text fields because they get disposed
// as soon as the Dialog closes
private void saveInput() {
    firstName = txtFirstName.getText();
    lastName = lastNameText.getText();

}

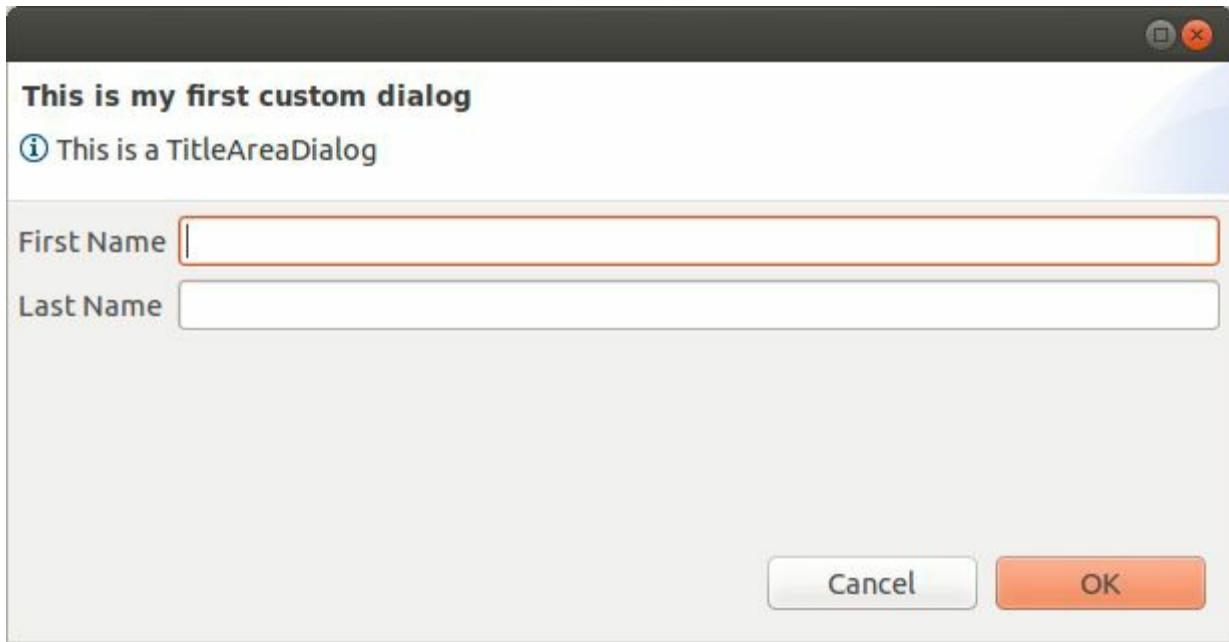
@Override
protected void okPressed() {
    saveInput();
    super.okPressed();
}

public String getFirstName() {
    return firstName;
}

public String getLastname() {
    return lastName;
}
}

```

This dialog is depicted in the following screenshot.



The usage of this dialog is demonstrated in the following code snippet. This code might for example be used in a handler.

```
MyTitleAreaDialog dialog = new MyTitleAreaDialog(shell);
dialog.create();
if (dialog.open() == Window.OK) {
    System.out.println(dialog.getFirstName());
    System.out.println(dialog.getLastName());
}
```

Chapter 85. Exercise: Dialogs

85.1. Confirmation dialog at exit

Ensure that the `org.eclipse.jface` plug-in is defined as dependency of your application plug-in.

In your `ExitHandler` class ensure that the user wants to exit the application via a confirmation dialog. This is done in the following code.

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

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

public class ExitHandler {
    @Execute
    public void execute(IWorkbench workbench, Shell shell) {
        boolean result = MessageDialog.openConfirm(shell, "Close",
            "Close application?");
        if (result) {
            workbench.close();
        }
    }
}
```

Warning

A handler is executed in the context in which it is called. It should receive all its input via its `@Execute` method. Field or constructor injection in a handler does only work for special cases, i.e. objects provided by an `ExtendedObjectSupplier`.

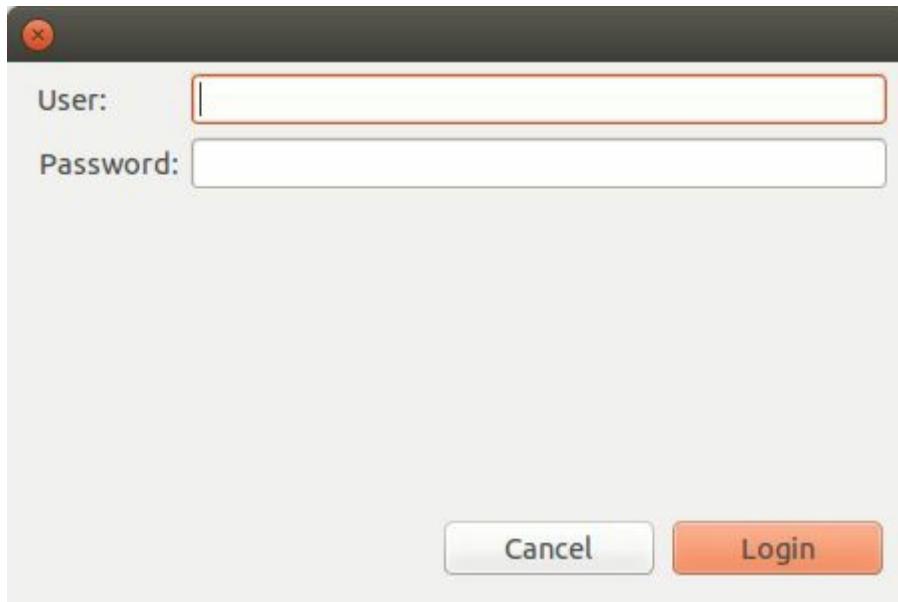
Test your implementation. If you select the *Cancel* button on the confirmation dialog the application should not terminate.



85.2. Create a password dialog

Create a `com.example.e4.rcp.todo.dialogs` package in your application plug-in.

Create a new class called `PasswordDialog` which extends the `Dialog` class. This dialog allows you to enter a user name and a password.



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

import org.eclipse.jface.dialogs.Dialog;
import org.eclipse.jface.dialogs.IDialogConstants;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.ModifyEvent;
import org.eclipse.swt.events.ModifyListener;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class PasswordDialog extends Dialog {
    private Text txtUser;
    private Text txtPassword;
    private String user = "";
    private String password = "";

    public PasswordDialog(Shell parentShell) {
        super(parentShell);
```

```

}

@Override
protected Control createDialogArea(Composite parent) {
    Composite container = (Composite) super.createDialogArea(parent);
    GridLayout layout = new GridLayout(2, false);
    layout.marginRight = 5;
    layout.marginLeft = 10;
    container.setLayout(layout);

    Label lblUser = new Label(container, SWT.NONE);
    lblUser.setText("User:");

    txtUser = new Text(container, SWT.BORDER);
    txtUser.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true, false,
        1, 1));
    txtUser.setText(user);
    txtUser.addModifyListener(new ModifyListener() {

        @Override
        public void modifyText(ModifyEvent e) {
            Text textWidget = (Text) e.getSource();
            String userText = textWidget.getText();
            user = userText;
        }
    });

    Label lblPassword = new Label(container, SWT.NONE);
    GridData gd_lblNewLabel = new GridData(SWT.LEFT, SWT.CENTER, false,
        false, 1, 1);
    gd_lblNewLabel.horizontalIndent = 1;
    lblPassword.setLayoutData(gd_lblNewLabel);
    lblPassword.setText("Password:");

    txtPassword = new Text(container, SWT.BORDER| SWT.PASSWORD);
    txtPassword.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true,
        false, 1, 1));
    txtPassword.setText(password);
    txtPassword.addModifyListener(new ModifyListener() {

        @Override
        public void modifyText(ModifyEvent e) {
            Text textWidget = (Text) e.getSource();
            String passwordText = textWidget.getText();
            password = passwordText;
        }
    });
    return container;
}

// override method to use "Login" as label for the OK button
@Override
protected void createButtonsForButtonBar(Composite parent) {
    createButton(parent, IDialogConstants.OK_ID, "Login", true);
}

```

```

        createButton(parent, IDialogConstants.CANCEL_ID,
                    IDialogConstants.CANCEL_LABEL, false);
    }

@Override
protected Point getInitialSize() {
    return new Point(450, 300);
}

@Override
protected void okPressed() {
    user = txtUser.getText();
    password = txtPassword.getText();
    super.okPressed();
}

public String getUser() {
    return user;
}

public void setUser(String user) {
    this.user = user;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

}

```

To use the dialog in your application, create a new menu entry (and a command with a handler) which allows you to open this dialog. The class associated with the handler should be called *EnterCredentialsHandler*.

```

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

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.jface.window.Window;
import org.eclipse.swt.widgets.Shell;

import com.example.e4.rcp.todo.dialogs.PasswordDialog;

public class EnterCredentialsHandler {

    @Execute
    public void execute(Shell shell) {
        PasswordDialog dialog = new PasswordDialog(shell);

        // get the new values from the dialog
    }
}

```

```
if (dialog.open() == Window.OK) {  
    String user = dialog.getUser();  
    String pw = dialog.getPassword();  
    System.out.println(user);  
    System.out.println(pw);  
}  
}  
}
```

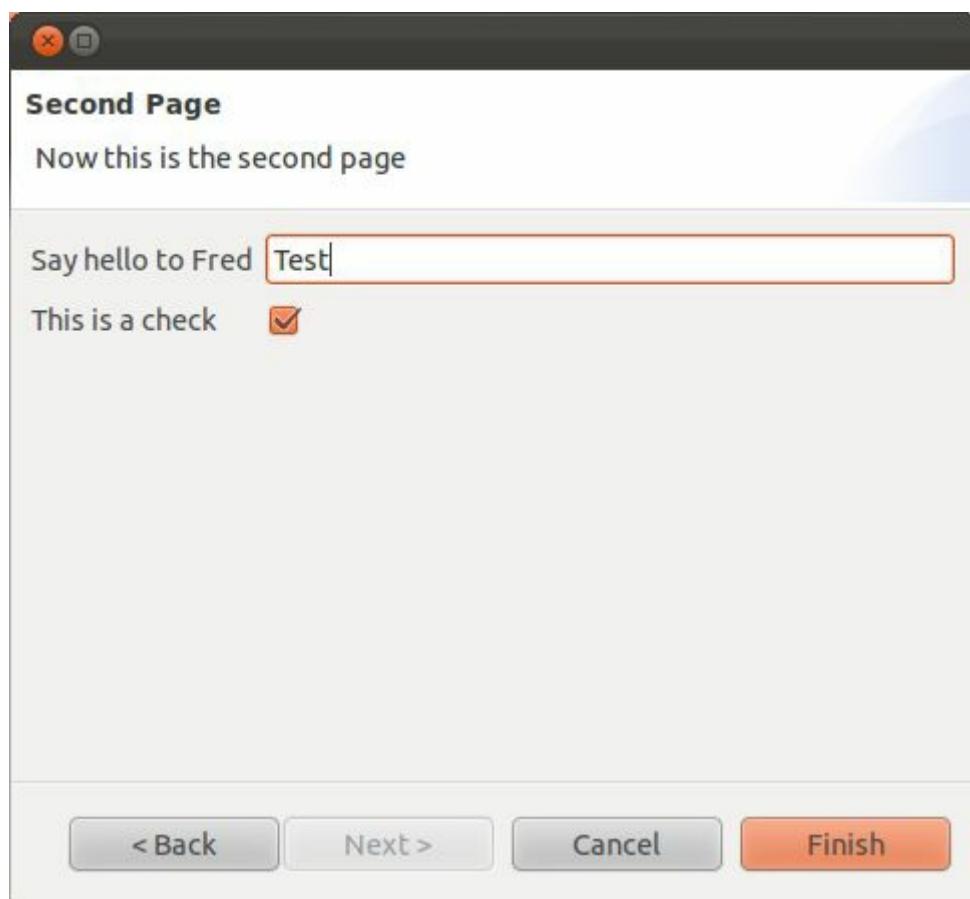
Chapter 86. Wizards

86.1. What is a wizards

In Eclipse, a *wizard* is commonly used for the creation of new elements, imports or exports. It can also be used for the execution of any task involving a sequential series of steps. A wizard should be used if there are many steps in the task, and they must be completed in a specific order.

Wizards are meant to take the hassle out of standard, repetitive, or tedious user tasks. If a wizard is started, typically a new modal window is opened. This window blocks the remaining user interface until the user finishes the task in the wizard or cancels the wizard.

The following screenshot shows the second page of a wizard.



Tip

JFace orders the buttons differently depending on the platform.

86.2. Wizards and WizardPages

The `Wizard` class from the `org.eclipse.jface.wizard` package provides the functionality to build custom wizards. This class controls the navigation between the different pages and provides the base user interface, for example, an area for error and information messages.

A wizard contains one or several pages of the type `WizardPage`. Such a page is added to a `Wizard` object via the `addPage()` method.

A `WizardPage` must create a new `Composite` in its `createControl()` method. This new `Composite` must use the `Composite` of the method parameter as parent. It also must call the `setControl()` method with this new `Composite` as parameter. If this is omitted, Eclipse will throw an error.

86.3. Starting the Wizard

To open a wizard, you use the `WizardDialog` class from the `org.eclipse.jface.wizard` package.

```
WizardDialog dialog = new WizardDialog(shell, new YourWizardClass());
dialog.open();
```

Typically, a wizard is opened via a menu or toolbar entry or via a `SelectionListener` on a button.

86.4. Changing the page order

To control the order of the `WizardPage` object, you can override the `getNextPage()` method in the `Wizard`. This allows you to change the order of the pages depending on the data in the wizard and pages.

The following code snippet demonstrates this approach.

```
// todo is an object available in the wizard

private TodoWizardPage1 page1 = new TodoWizardPage1(todo);
private TodoWizardPage2 page2 = new TodoWizardPage2();
private TodoWizardPage3 specialPage = new TodoWizardPage3(todo);

@Override
public void addPages() {
    addPage(page1);
    addPage(page2);
    addPage(specialPage);
}

@Override
public IWizardPage getNextPage(IWizardPage currentPage) {
    if (todo.isDone()) {
        return specialPage;
    }
    if (currentPage == page1) {
        return page2;
    }
    return null;
}
```

86.5. Working with data in the wizard

To use the same data in different pages of your wizard, pass them to the wizard pages via their constructor parameters.

The `isVisible()` method is called whenever the `WizardPage` either gets visible or invisible. Call the `super.isVisible()` method and check the current status of the page. If the page is visible, assign the data of your object to the user interface components.

86.6. Updating the Wizard buttons from a WizardPage

The `WizardPage` class defines the `canFlipToNextPage()` and `setPageComplete()` methods to control if the *Next* or the *Finish* button in the wizard becomes active.

The `Wizard` class defines the `canFinish` method in which you can define if the wizard can be completed.

If the status changes, you can update the buttons in the `Wizard` (e.g., the *Next* and the *Finish* button) from a `WizardPage` via the `getWizard().getContainer().updateButtons()` method call.

Chapter 87. Exercise: Create a wizard

87.1. Create classes for the wizard

In this exercise you create a wizard in your `com.example.e4.rcp.todo` plug-in which allows you to create a new Todo item. This exercise demonstrates how you can reuse your part implementation in a wizard.

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

Create the following `TodoWizardPage1` class which is your first wizard page. This page reuses the `TodoDetailsPart` for defining the user interface.

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

import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;

import com.example.e4.rcp.todo.model.Todo;
import com.example.e4.rcp.todo.parts.TodoDetailsPart;

public class TodoWizardPage1 extends WizardPage {

    private Todo todo;

    public TodoWizardPage1(Todo todo) {
        super("page1");
        this.todo = todo;
        setTitle("New Todo");
        setDescription("Enter the todo data");
    }

    public void createControl(Composite parent) {
        Composite container = new Composite(parent, SWT.NONE);
        // We reuse the part and
        // inject the values
        TodoDetailsPart part = new TodoDetailsPart();
        part.createControls(container);
        part.setTodo(todo);
        setControl(container);
    }
}
```

Note

As you create this instance of the `TodoDetailsPart` class, there is no automatic dependency injection performed by the framework. Therefore the above code calls the methods directly to construct the user interface.

Create the following class for the second page of the wizard. It contains only a *Confirm* checkbox button.

Warning

You will get a temporary compile error once you get this class because it uses the `TodoWizard` class which you create in the next step.

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

import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;

public class TodoWizardPage2 extends WizardPage {

    private boolean checked = false;

    /**
     * create the wizard.
     */
    public TodoWizardPage2() {
        super("wizardPage");
        setTitle("Validate");
        setDescription("Check to create the todo item");
    }

    /**
     * create contents of the wizard.
     *
     * @param parent
     */
    public void createControl(Composite parent) {
        Composite container = new Composite(parent, SWT.NONE);
        GridLayout layout = new GridLayout(2, true);
        container.setLayout(layout);
        Label label = new Label(container, SWT.NONE);
        label.setText("Create the todo");
        Button button = new Button(container, SWT.CHECK);
```

```

button.setText("Check");
button.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
        checked = ((Button) e.getSource()).getSelection();
        TodoWizard wizard = (TodoWizard) getWizard();
        wizard.finish = checked;
        // the following code updates the
        // buttons in the wizard
        wizard.getContainer().updateButtons();
    }
});
setControl(container);
}

public boolean isChecked() {
    return checked;
}
}

```

Create the TodoWizard class which extends the Wizard class and add TodoWizardPage1 and TodoWizardPage2 as pages to it.

```

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

import javax.inject.Inject;

import org.eclipse.jface.wizard.Wizard;

import com.example.e4.rcp.todo.model.Todo;

public class TodoWizard extends Wizard {

    private Todo todo;
    boolean finish = false;

    @Inject
    public TodoWizard(Todo todo) {
        this.todo = todo;
        setWindowTitle("New Wizard");
    }

    @Override
    public void addPages() {
        addPage(new TodoWizardPage1(todo));
        addPage(new TodoWizardPage2());
    }

    @Override
    public boolean performFinish() {
        return true;
    }
}

```

```
}

@Override
public boolean canFinish() {
    return finish;
}

}
```

87.2. Adjust part

To capture the changes in the user interface in your `Todo` object you need to extend your `TodoDetailsPart` class.

Add a `ModifyListener` to your text fields in your `TodoDetailsPart` class and write the content of the text fields back to the `Todo` object if the text fields are modified.

```
// NOTE: your variable names may be different!!!

// add modification listeners to your text fields in
// the @PostConstruct method

// this is already in your @PostConstruct method
txtSummary = new Text(parent, SWT.BORDER);
txtSummary.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true,
    false, 1, 1));

// NEW! NEW! add a ModifyListener
txtSummary.addModifyListener(new ModifyListener() {
    @Override
    public void modifyText(ModifyEvent e) {
        if (todo!=null){
            todo.setSummary(txtSummary.getText());
        }
    }
});

// this is already in your @PostConstruct method
Label lblDescription = new Label(parent, SWT.NONE);
lblDescription.setText("Description");

txtDescription = new Text(parent, SWT.BORDER | SWT.MULTI | SWT.V_SCROLL);
txtDescription.setLayoutData(new GridData(SWT.FILL, SWT.FILL, true,
    true, 1, 1));

//NEW! NEW! ModifyListener
txtDescription.addModifyListener(new ModifyListener() {
    @Override
    public void modifyText(ModifyEvent e) {
        if (todo!=null){
            todo.setDescription(txtDescription.getText());
        }
    }
});

// more code
```

Note

You could also add similar listeners to your check button and `DateTime` widget to write changes in these widgets back to the `Todo` object. We skip this step here because in [Part XVII, “Data Binding with JFace”](#) you learn an easier way to do this.

87.3. Adjust handler implementation

Call the wizard from your `NewTodoHandler` class, which you already added to the menu in an earlier exercise.

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

import java.util.Date;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.jface.wizard.WizardDialog;
import org.eclipse.swt.widgets.Shell;

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;
import com.example.e4.rcp.todo.wizards.TodoWizard;

public class NewTodoHandler {
    @Execute
    public void execute(Shell shell, ITodoService todoService) {
        // use -1 to indicate a not existing id
        Todo todo = new Todo(-1);
        todo.setDueDate(new Date());
        WizardDialog dialog = new WizardDialog(shell, new TodoWizard(todo));
        if (dialog.open() == WizardDialog.OK) {
            // call service to save Todo object
            todoService.saveTodo(todo);
        }
    }
}
```

87.4. Validating

Validate that you can create new Todo objects via your wizard. Start your application and create a new Todo via the wizard.

Afterwards refresh your table via the "Load Data" button to see the entry.

Warning

If you did not implement the modification listeners correctly, you may get a new empty in the table.

Note

In [Part XXIV, “Event service for message communication”](#) you learn how to update all components of your application immediately after a change in the data model.

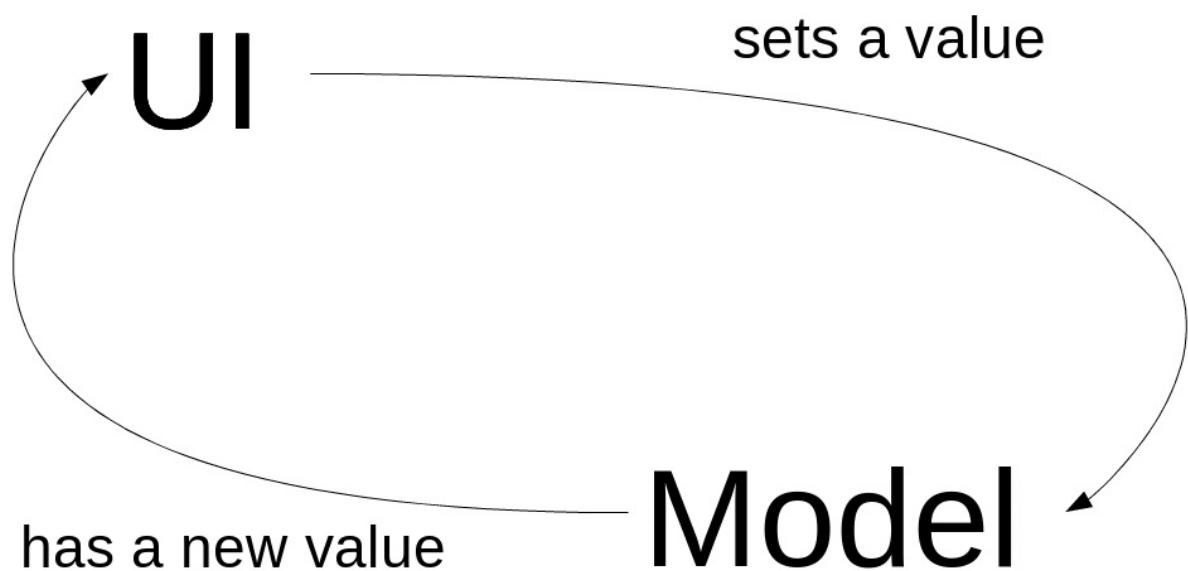
Part XVII. Data Binding with JFace

Chapter 88. Data Binding with JFace

88.1. What are Data Binding frameworks?

A *Data Binding* framework connects properties of objects. It is typically used to synchronize properties of user interface widgets with properties of other Java objects. These Java objects are typically called the *data model* or the *domain model*.

Data Binding frameworks synchronize changes of these properties. They typically support validation and conversion during the synchronization process. This synchronization is depicted in the following graphic.



For example you could bind the String property called *firstName* of a Java object to a *text* property of the SWT `Text` widget. If the user changes the text in the user interface, the corresponding property in the Java object is updated.

88.2. JFace Data Binding

JFace Data Binding is an Eclipse framework for data binding and is frequently used in Eclipse based applications.

88.3. JFace Data Binding Plug-ins

The following plug-ins are required to use JFace Data Binding.

- org.eclipse.core.databinding
- org.eclipse.core.databinding.beans
- org.eclipse.core.databinding.property
- org.eclipse.jface.databinding

Chapter 89. Listening to changes

89.1. Ability to listen to changes in UI components

To observe changes in an attribute of a Java object, a data binding framework needs to be able to register itself as a listener to this attribute.

The SWT and JFace widgets support this, therefore it is possible to use JFace data binding to update SWT and JFace user interface components.

89.2. Ability to listen to changes in the domain model

JFace Data Binding can be used to observe attributes of a domain model. It can register listeners for these Java objects and gets notified if a change in the model happens. This change notification from the domain model requires that the model objects provide `PropertyChangeSupport` according to the Java Bean specification.

89.3. Property change support

The data model is typically described as a *Plain Old Java Object (POJO)* model or a *Java Bean* model.

The term POJO is used to describe a Java object which does not follow any of the major Java object models, conventions, or frameworks, i.e. a Java object which does not have to fulfill any specific requirements. For example the following is a POJO.

```
package com.vogella.databinding.example;

public class Person {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

A Java Bean is a Java object which follows the Java Bean specification. This specification requires that the class implements `getter` and `setter` methods for all its attributes. It must also implement property change support via the `PropertyChangeSupport` class and propagate changes to registered listeners.

A Java class which provides `PropertyChangeSupport` looks like the following example.

```
package com.vogella.databinding.example;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class ModelObject {
    private final PropertyChangeSupport changes =
        new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener
        listener) {
        changes.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener
```

```
        listener) {  
    changes.removePropertyChangeListener(listener);  
}  
}
```

Other domain classes could extend this class. The following example demonstrates that.

```
package com.vogella.databinding.example;  
  
public class Person extends ModelObject {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        changes.firePropertyChange("name", this.name, this.name = name);  
    }  
}
```

89.4. Data Binding and Java objects without change notification

If the Java object (POJO) does not support change notification, you can still use Data Binding to connect fields of other objects to fields of this object.

For example you can connect the `text` property of a SWT `Text` field to the `summary` field of a `Todo` object. While updates in fields of the `Todo` object will not update the `Text` widget, relevant change in the user interface will update the domain model (`Todo`).

Chapter 90. Create bindings

90.1. Observing properties with the `IObservableValue` interface

The `IObservableValue` interface is used to observe properties of objects. On widgets you typically observe the *text* property but you can also observe other values. For example, you could bind the *enabled* property to a boolean value of the data model.

90.2. Creating instances of the `IObservableValue`

JFace Data Binding provides the *Properties API* for creating `IObservableValue` instances. The *Properties API* provides factories to create `IObservableValue` objects.

The main factories are `PojoProperties`, `BeanProperties`, `WidgetProperties` and `ViewerProperties`.

Table 90.1. Factories

Factory	Description
<code>PojoProperties</code>	Used to create <code>IObservableValues</code> for Java objects.
<code>BeanProperties</code>	Used to create <code>IObservableValues</code> objects for Java Beans.
<code>WidgetProperties</code>	Used to create <code>IObservableValues</code> for properties of SWT widgets.
<code>ViewerProperties</code>	Used to create <code>IObservableValues</code> for properties of JFace Viewer.
<code>Properties</code>	Used to create <code>IObservables</code> for properties of any type like Objects, Collections or Maps.
<code>Observables</code>	Used to create <code>IObservables</code> for properties of special Objects, Collections, Maps and Entries of an <code>IObservableMap</code> .

90.3. Connecting properties with the DataBindingContext

The `DataBindingContext` class provides the functionality to connect `IObservableValues` objects.

Via the `DataBindingContext.bindValue()` method two `IObservableValues` objects are connected. The first parameter is the target and the second is the model. During the initial binding the value from the model is copied to the target.

```
// create new Context
DataBindingContext ctx = new DataBindingContext();

// define the IObservables
IObservableValue target = WidgetProperties.text(SWT.Modify).
    observe(firstName);
IObservableValue model= BeanProperties.
    value(Person.class,"firstName").observe(person);

// connect them
ctx.bindValue(target, model);
```

Note

The initial copying from model to target is useful for the initial synchronization. For example if you have an attribute of a `Person p` object and the text attribute of a `Text txtName` widget, you typically want to copy the value from `p` to `txtName` at the beginning.

90.4. Example: how to observe properties

The following code demonstrates how to create an `IObservableValue` object for the `firstName` property of a Java object called `person`.

```
// if person is a POJO
IObservableValue myModel = PojoProperties.value("firstName").
    observe(person)

// prefer using beans if your data model provides property change support
IObservableValue myModel = BeansProperties.value("firstName").
    observe(person)
```

The next example demonstrates how to create an `IObservableValue` for the `text` property of an SWT `Text` widget called `firstNameText`.

```
IObservableValue target = WidgetProperties.text(SWT.Modify).
    observe(firstNameText);
```

In case you do not want to manipulate the object properties directly you can place those attributes into an `IObservableMap` and observe the `MapEntries` with the `Observables.observeMapEntry()` method.

```
import javax.annotation.PostConstruct;
import javax.inject.Inject;
import javax.inject.Named;

import org.eclipse.core.databinding.DataBindingUtil;
import org.eclipse.core.databinding.observable.Observables;
import org.eclipse.core.databinding.observable.map.IObservableMap;
import org.eclipse.core.databinding.observable.map.WritableMap;
import org.eclipse.core.databinding.observable.value.IObservableValue;
import org.eclipse.e4.core.di.annotations.Optional;
import org.eclipse.e4.ui.di.Persist;
import org.eclipse.e4.ui.services.IServiceConstants;
import org.eclipse.jface.databinding.swt.ISWTObservableValue;
import org.eclipse.jface.databinding.swt.WidgetProperties;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Text;

import com.example.e4.rcp.todo.model.Todo;

public class ObservableMapEntry {

    private static final String SECOND_ATTRIBUTE = "secondAttribute";
    private static final String FIRST_ATTRIBUTE = "firstAttribute";

    private IObservableMap attributesMap = new WritableMap();
```

```

private DataBindingContext dbc;
private Todo todo;

@PostConstruct
public void createUI(Composite parent) {
    dbc = new DataBindingUtil();

    Text txtFirstAttribute = new Text(parent, SWT.BORDER);
    Text txtSecondAttribute = new Text(parent, SWT.BORDER);

    // create observables for the Text controls
    ISWT/ObservableValue txtFirstAttributeObservable =
        WidgetProperties.text(SWT.Modify).observe(txtFirstAttribute);
    ISWT/ObservableValue txtSecondAttributeObservable = WidgetProperties.text(SWT
        .observe(txtSecondAttribute);

    // create observables for the Map entries
    IObservableValue firstAttributeObservable =
        Observables.observeMapEntry(attributesMap, FIRST_ATTRIBUTE);
    IObservableValue secondAttributeObservable =
        Observables.observeMapEntry(attributesMap, SECOND_ATTRIBUTE);

    dbc.bindValue(txtFirstAttributeObservable, firstAttributeObservable);
    dbc.bindValue(txtSecondAttributeObservable, secondAttributeObservable);
}

@Inject
@Optional
public void setModel(@Named(IServiceConstants.ACTIVE_SELECTION) Todo todo) {
    if (todo != null) {
        this.todo = todo;
        // Set new values for the map entries from a model object
        attributesMap.put(FIRST_ATTRIBUTE, todo.getSummary());
        attributesMap.put(SECOND_ATTRIBUTE, todo.getDescription());
    }
}

@Persist
public void save() {
    if (todo != null) {
        // only store the actual values on save and not directly
        todo.setSummary((String) attributesMap.get(FIRST_ATTRIBUTE));
        todo.setDescription((String) attributesMap.get(SECOND_ATTRIBUTE));
    }
}
}

```

90.5. Observing nested properties

You can also observe nested model properties, e.g., attributes of classes which are contained in another class.

The following code demonstrates how to access the *country* property in the *address* field of the object *person*.

```
IObservable model = PojoProperties.value(Person.class,  
    "address.country").observe(person);
```

Chapter 91. Updates, convertors and validators

91.1. UpdateValueStrategy

The `bindValue()` method allows you to specify `UpdateValueStrategy` objects as third and fourth parameters. These objects allow you to control how and when the values are updated. The following values are permitted:

Table 91.1. UpdateValueStrategy

Value	Description
<code>UpdateValueStrategy.POLICY_NEVER</code>	Policy constant denoting that the source observable's state should not be tracked and that the destination observable should never be updated.
<code>UpdateValueStrategy.POLICY_ON_REQUEST</code>	Policy constant denoting that the source observable's state should not be tracked but that validation, conversion and updating the destination observable should be performed when explicitly requested. You can call <code>DataBindingContext.updateModelToTarget</code> or <code>DataBindingContext.updateTargets</code> to update all bindings at once. Or call <code>Binding.updateTargetToModel</code> or <code>Binding.updateModelToTarget</code> to update a single binding.
<code>UpdateValueStrategy.POLICY_CONVERT</code>	Policy constant denoting that the source observable's state should be tracked including validating changes except <code>validateBeforeSet(Object)</code> , but the destination observable's value should be updated on request.
<code>UpdateValueStrategy.POLICY_UPDATE</code>	Policy constant denoting that the source observable's state should be tracked and that validation, conversion and updating the destination observable's value should be performed automatically on every change of the source observable.

If no `UpdateValueStrategy` is specified, the `UpdateValueStrategy.POLICY_UPDATE` is used by default.

You can register converters and validators in the `UpdateValueStrategy` object.

91.2. Converter

Converters allow you to convert the values between the model and the target. Converters are defined based on the `IConverter` interface.

91.3. Validator

Validators allow you to implement validation of the data before it is propagated to the other connected property. A class which wants to provide this functionality must implement the org.eclipse.core.databinding.validation.IValidator interface.

```
// define a validator to check that only numbers are entered
IValidator validator = new IValidator() {
    private final Pattern numbersOnly = Pattern.compile("\\d*");
    @Override
    public IStatus validate(Object value) {
        if (value != null && numbersOnly.matcher(value.toString()).matches()) {
            return ValidationStatus.ok();
        }
        return ValidationStatus.error(value + " is not a number");
    }
};

// create an UpdateValueStrategy and assign it to the binding
UpdateValueStrategy strategy = new UpdateValueStrategy();
strategy.setBeforeSetValidator(validator);

Binding bindValue =
    ctx.bindValue(widgetValue, modelValue, strategy, null);
```

Tip

The WizardPageSupport class provides support to connect the result from the given data binding context to the given wizard page, updating the wizard page's completion state and its error message accordingly.

Chapter 92. More on bindings

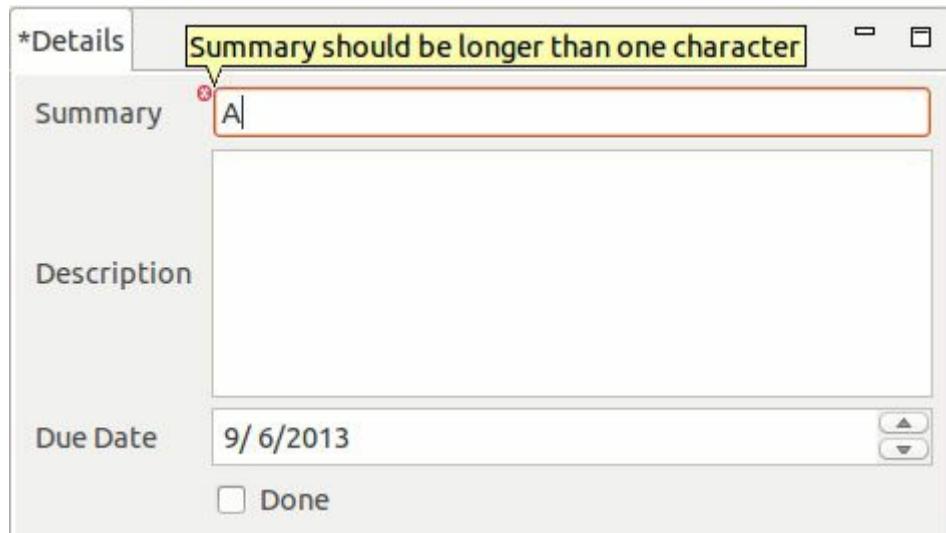
92.1. ControlDecorators

JFace Data Binding allows you to use icon decorators in the user interface which reflect the status of the field validation. This allows you to provide immediate feedback to the user. For the creation of the control decoration you use the return object from the `bindValue()` method of `DataBindingContext` object.

```
// The following code assumes that a Validator is already defined
Binding bindValue =
    ctx.bindValue(widgetValue, modelValue, strategy, null);

// add some decorations to the control
ControlDecorationSupport.create(bindValue, SWT.TOP | SWT.LEFT);
```

The result might look like the following screenshot.



92.2. Placeholder binding with WritableValue

You can create bindings to a `WritableValue` object. A `WritableValue` object can hold a reference to another object.

You can exchange this reference in `WritableValue` and the databinding will use the new (reference) object for its binding. This way you can create the binding once and still exchange the object which is bound by databinding.

To bind to a `WritableValue` you use the `observeDetail()` method, to inform the framework that you would like to observe the contained object.

```
WritableValue value = new WritableValue();

// create the binding
DataBindingContext ctx = new DataBindingContext();
IObservableValue target = WidgetProperties.
    text(SWT.Modify).observe(text);
IObservableValue model = BeanProperties.value("firstName").
    observeDetail(value);

ctx.bindValue(target, model);

// create a Person object called p
Person p = new Person();

// make the binding valid for this new object
value.setValue(p);
```

92.3. Listening to all changes in the binding

You can register a listener to all bindings of the `DataBindingContext` class. Your listener will be called when something has changed.

For example this can be used to determine the status of a part which behaves like an editor. If its data model changes, this editor marks itself as dirty.

```
// define your change listener
// dirty holds the state for the changed status of the editor
IChangeListener listener = new IChangeListener() {
    @Override
    public void handleChange(ChangeEvent event) {
        // Ensure dirty is not null
        if (dirty!=null){
            dirty.setDirty(true);
        }
    }
};

private void updateUserInterface(Todo todo) {

    // check that the user interface is available
    if (txtSummary != null && !txtSummary.isDisposed()) {

        // Deregister change listener to the old binding
        IObservableList providers = ctx.getValidationStatusProviders();
        for (Object o : providers) {
            Binding b = (Binding) o;
            b.getTarget().removeChangeListener(listener);
        }

        // dispose the binding
        ctx.dispose();

        // NOTE
        // HERE WOULD BE THE DATABINDING CODE
        // INTENTIALLY LEFT OUT FOR BREVITY

        // get the validation status provides
        IObservableList bindings =
            ctx.getValidationStatusProviders();

        // not all validation status providers
        // are bindings, e.g. MultiValidator
        // otherwise you could use
        // context.getBindings()
```

```
// register the listener to all bindings
for (Object o : bindings) {
    Binding b = (Binding) o;
    b.getTarget().addChangeListener(listener);
}
}
```

92.4. More information on Data Binding

Data Binding provides lots of examples for other use cases via its version control system. See [Wiki on JFace Data Binding](#) for instructions how to access this information.

Chapter 93. Exercise: Data Binding for SWT widgets

93.1. Add the plug-in dependencies

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

- org.eclipse.core.databinding
- org.eclipse.core.databinding.beans
- org.eclipse.core.databinding.property
- org.eclipse.jface.databinding

93.2. Implement the property change support

To have a type safe access to your field names in your `Todo` class, you should add constants for them. Also, change your `Todo` class to implement the property change support.

The resulting code should look like the following listing.

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

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.util.Date;

public class Todo {

    private PropertyChangeSupport changes = new PropertyChangeSupport(this);

    public static final String FIELD_ID = "id";
    public static final String FIELD_SUMMARY = "summary";
    public static final String FIELD_DESCRIPTION = "description";
    public static final String FIELD_DONE = "done";
    public static final String FIELD_DUEDATE = "dueDate";

    public final long id;
    private String summary;
    private String description;
    private boolean done;
    private Date dueDate;

    public Todo(long i) {
        id = i;
    }

    public Todo(long i, String summary, String description, boolean b, Date date)
        this.id = i;
        this.summary = summary;
        this.description = description;
        this.done = b;
        this.dueDate = date;
    }

    public long getId() {
        return id;
    }

    public String getSummary() {
        return summary;
    }

    public void setSummary(String summary) {
```

```

        changes.firePropertyChange(FIELD_SUMMARY, this.summary,
            this.summary = summary);
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        changes.firePropertyChange(FIELD_DESCRIPTION, this.description,
            this.description = description);
    }

    public boolean isDone() {
        return done;
    }

    public void setDone(boolean isDone) {
        changes.firePropertyChange(FIELD_DONE, this.done, this.done = isDone);
    }

    public Date getDueDate() {
        return new Date(dueDate.getTime());
    }

    public void setDueDate(Date dueDate) {
        changes.firePropertyChange(FIELD_DUEDATE, this.dueDate,
            this.dueDate = new Date(dueDate.getTime()));
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + (int) (id ^ (id >>> 32));
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Todo other = (Todo) obj;
        if (id != other.id)
            return false;
        return true;
    }

    @Override
    public String toString() {

```

```
        return "Todo [id=" + id + ", summary=" + summary + "]";  
    }  
  
    public Todo copy() {  
        return new Todo(id, summary, description, done, dueDate);  
    }  
  
    public void addPropertyChangeListener(PropertyChangeListener l) {  
        changes.addPropertyChangeListener(l);  
    }  
  
    public void removePropertyChangeListener(PropertyChangeListener l) {  
        changes.removePropertyChangeListener(l);  
    }  
}
```

Warning

The constants must reflect the bean property names in the class. Otherwise, the data binding will not find the right property to bind to.

93.3. Remove the modification listeners in your code

Delete your `addModifyListener()` methods in the `TodoDetailsPart` class. The synchronization between the `Todo` object and the user interface will be handled by the JFace Data Binding framework.

Warning

Ensure to remove these listeners to avoid unwanted side-effects. The easiest way to find them, is with a text search. Search for the `ModifyListener` term in your `TodoDetailsPart` class.

93.4. Add a field for the data binding context to TodoDetailsPart

Add a new field to your `TodoDetailsPart` class for the `DataBindingContext` which is used in the next step for doing the data binding.

```
// more code  
// ....  
  
// define DataBindingContext as field  
DataBindingContext ctx = new DataBindingContext();  
  
// more code  
// ....
```

93.5. Implement data binding in TodoDetailsPart

Change your implementation of the `TodoDetailsPart` class to use data binding based on the following example code.

```
private void updateUserInterface(Todo todo) {
    // if Todo is null disable user interface
    // and leave method
    if (todo == null) {
        enableUserInterface(false);
        return;
    }

    // the following check ensures that the user interface is available,
    // it assumes that you have a text widget called "txtSummary"
    if (txtSummary != null && !txtSummary.isDisposed()) {
        enableUserInterface(true);

        // ctx is defined as field in the class!!!
        // disposes existing bindings
        ctx.dispose();

        // summary
        IObservableView oWidgetSummary = WidgetProperties.text(SWT.Modify)
            .observe(txtSummary);
        IObservableView oTodoSummary = BeanProperties.value(Todo.FIELD_SUMMARY)
            .observe(todo);
        ctx.bindValue(oWidgetSummary, oTodoSummary);

        // description
        IObservableView oWidgetDescription =
            WidgetProperties.text(SWT.Modify).observe(txtDescription);
        IObservableView oTodoDescription =
            BeanProperties.value(Todo.FIELD_DESCRIPTION).observe(todo);
        ctx.bindValue(oWidgetDescription, oTodoDescription);

        // done status
        IObservableView oWidgetButton = WidgetProperties.selection().observe(btnD
        IObservableView oTodoDone = BeanProperties.value(Todo.FIELD_DONE).observe
        ctx.bindValue(oWidgetButton, oTodoDone);

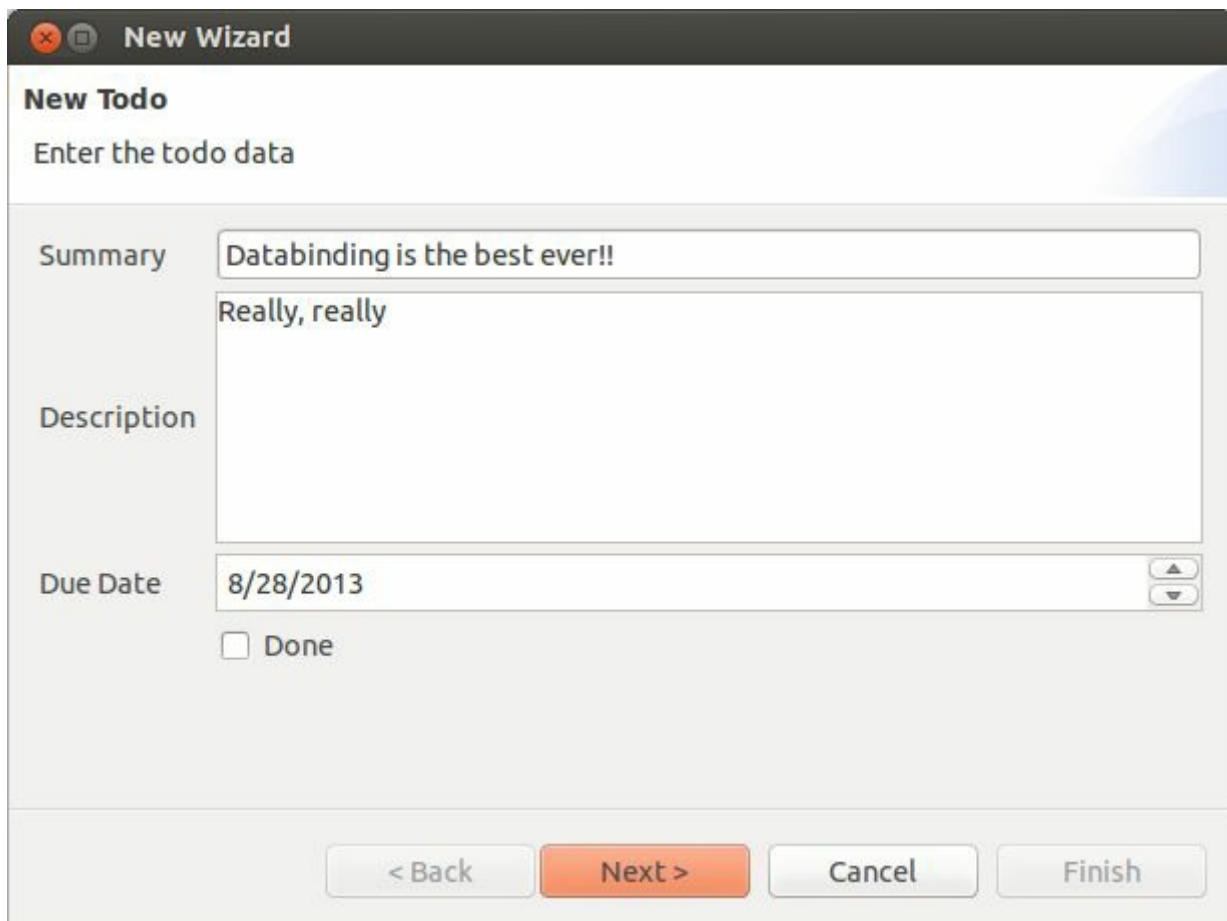
        // due date
        IObservableView oWidgetSelectionDateTime = WidgetProperties
            .selection().observe(dateTime);
        IObservableView oTodoDueDate = BeanProperties.
            value(Todo.FIELD_DUEDATE).observe(todo);
        ctx.bindValue(oWidgetSelectionDateTime,
            oTodoDueDate);

    }
}
```

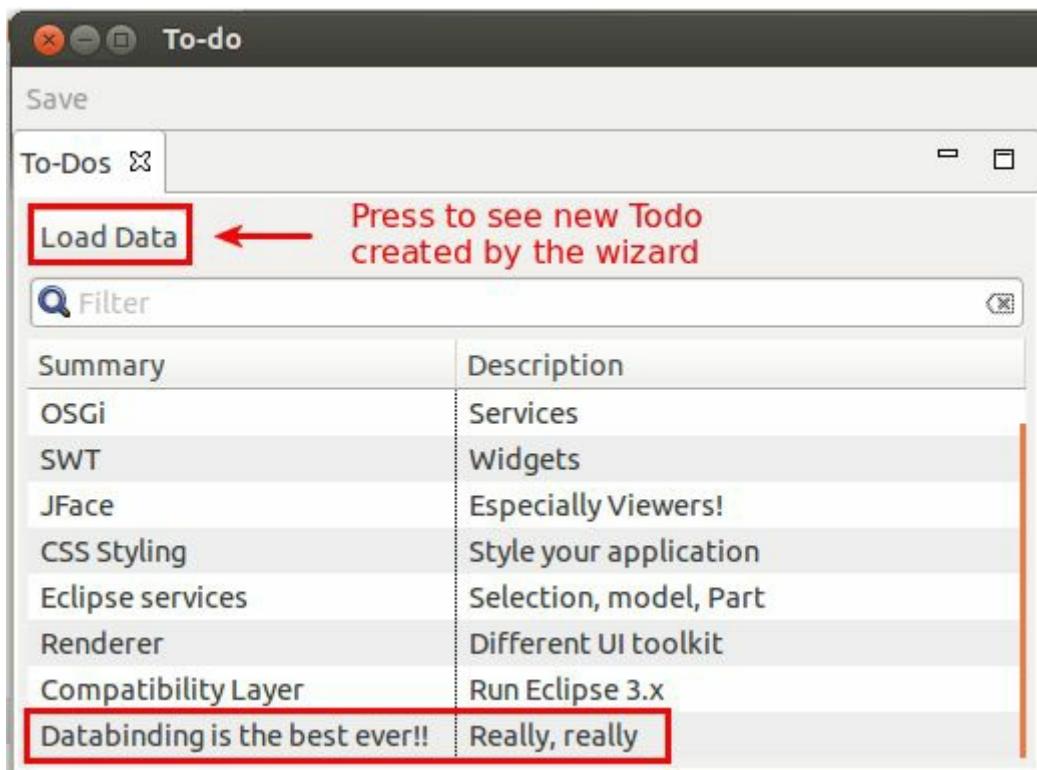
}

93.6. Validating

Use your wizard to create new `Todo` objects. The following screenshot shows the first page of the wizard.



Test if your implementation behaves correctly. Ensure that new `Todo` objects are shown in the table after pressing the *Load data* button.



Note

Your detail part is currently not supplied with a Todo, only the wizard can currently be used to test the databinding. This will be change in a later exercise.

Chapter 94. Data Binding for JFace viewer

94.1. Binding Viewers

JFace Data Binding provides functionality to bind the data of JFace viewers.

Data binding for these viewers distinguish between changes in the collection and changes in the individual object.

In the case that Data Binding observes a collection, it requires a content provider which notifies the viewer, once the data in the collection changes.

The `ObservableListContentProvider` class is a content provider which requires a list implementing the `IObservableList` interface. The `Properties` class allows you to wrap another list with its `selfList()` method into an `IObservableList`.

The following snippet demonstrates the usage:

```
// use ObservableListContentProvider
viewer.setContentProvider(new ObservableListContentProvider());

// create sample data
List<Person> persons = createExampleData();

// wrap the input into a writable list
IObservableList input =
    Properties.selfList(Person.class).observe(persons);

// set the IObservableList as input for the viewer
viewer.setInput(input);
```

94.2. Observing list details

You can also use the `ObservableMapLabelProvider` class to observe changes of the list elements.

```
ObservableListContentProvider contentProvider =
    new ObservableListContentProvider();

// create the label provider which includes monitoring
// of changes to update the labels

IobservableSet knownElements = contentProvider.getKnownElements();

final IobservableMap firstNames = BeanProperties.value(Person.class,
    "firstName").observeDetail(knownElements);
final IobservableMap lastNames = BeanProperties.value(Person.class,
    "lastName").observeDetail(knownElements);

IobservableMap[] labelMaps = { firstNames, lastNames };

ILabelProvider labelProvider =
    new ObservableMapLabelProvider(labelMaps) {
        public String getText(Object element) {
            return firstNames.get(element) + " " + lastNames.get(element);
        }
    };
};
```

94.3. ViewerSupport

ViewerSupport simplifies the setup for JFace viewers in cases where selected columns should be displayed. It registers changes listeners on the collection as well as on the individual elements.

ViewerSupport creates via the bind() method the LabelProvider and ContentProvider for a viewer automatically.

```
// the MyModel.getPersons() method call returns a List<Person> object
// the WritableList object wraps this object in an IObservableList

input = new WritableList(MyModel.getPersons(), Person.class);

// The following creates and binds the data
// for the Table based on the provided input
// no additional label provider /
// content provider / setInput required

ViewerSupport.bind(viewer, input,
    BeanProperties.
    values(new String[] { "firstName", "lastName", "married" }));
```

94.4. Master Detail binding

The `ViewerProperties` class allows you to create `IObservableValues` for properties of the viewer. For example you can track the current selection, e.g., which data object is currently selected. This binding is called *Master Detail* binding as you track the selection of a master.

To access fields in the selection you can use the `PojoProperties` or the `BeanProperties` class. Both provide the `value().observeDetail()` method chain, which allows you to observe a detailed value of an `IObservableValue` object.

For example the following will map the *summary* property of the `Todo` domain object to a `Label` based on the selection of a `ComboViewer`.

```
// assume we have Todo domain objects
// todos is a of type: List<Todo>
final ComboViewer viewer = new ComboViewer(parent, SWT.DROP_DOWN);
viewer.setContentProvider(ArrayContentProvider.getInstance());
viewer.setLabelProvider(new LabelProvider() {
    public String getText(Object element) {
        Todo todo = (Todo) element;
        return todo.getSummary();
    }
});
viewer.setInput(todos);

// create a Label to map to
Label label = new Label(parent, SWT.BORDER);
// parent has a GridLayout assigned
label.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true, false));

DataBindingContext dbc = new DataBindingContext();

// for binding to the label
IObservableValue target = WidgetProperties.text().observe(label);

// observe the selection
IVIEWERObservableValue selectedTodo = ViewerProperties
    .singleSelection().observe(viewer);
// observe the summary attribute of the selection
IObservableValue detailValue =
    PojoProperties
        .value("summary", String.class)
        .observeDetail(selectedTodo)

dbc.bindValue(target, detailValue);
```

94.5. Chaining properties

You can chain properties together to simplify observing nested properties. The following examples demonstrate this usage.

```
IObservableValue viewerSelectionSummaryObservable =
    ViewerProperties.singleSelection()
        .value(BeanProperties.value("summary", String.class))
        .observe(viewer);

IListProperty siblingNames = BeanProperties.
    value("parent").list("children").values("name");
IObservableList siblingNamesObservable =
    siblingNames.observe(node);
```

Chapter 95. Exercise: Data Binding for viewers

95.1. Implement Data Binding for the viewer

Change your implementation of `TodoOverviewPart` to use Data Binding.

For this you can delete the `LabelProvider` of your `TableViewerColumn` objects. Also remove the `viewer.setInput()` and the `ContentProvider`.

The following code demonstrates this implementation.

Tip

The following code snippet only contains the data binding relevant part of the code. If you did any of the optional exercises your implementation may also include a search box, the registration of the popup menu, a filter or more.

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

import java.util.List;

import javax.annotation.PostConstruct;
import javax.inject.Inject;

import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.observable.list.WritableList;
import org.eclipse.e4.ui.di.Focus;
import org.eclipse.jface.databinding.viewers.ViewerSupport;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.jface.viewers.TableViewerColumn;
import org.eclipse.swt.SWT;
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.Table;

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;

public class TodoOverviewPart {

    private Button btnLoadData;
    private TableViewer viewer;
```

```

@Inject
ITodoService todoService;

private WritableList writableList;

@PostConstruct
public void createControls(Composite parent) {
    parent.setLayout(new GridLayout(1, false));

    btnLoadData = new Button(parent, SWT.PUSH);
    btnLoadData.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            List<Todo> todos = todoService.getTodos();
            updateViewer(todos);
        }
    });
    btnLoadData.setText("Load Data");

    // more code, e.g. your search box
    // ...
    // ...

    viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL
        | SWT.V_SCROLL | SWT.FULL_SELECTION);
    Table table = viewer.getTable();
    table.setLayoutData(new GridData(SWT.FILL, SWT.FILL, true, true, 1, 1));

    table.setHeaderVisible(true);
    table.setLinesVisible(true);

    TableViewerColumn column = new TableViewerColumn(viewer, SWT.NONE);

    column.getColumn().setWidth(100);
    column.getColumn().setText("Summary");
    column = new TableViewerColumn(viewer, SWT.NONE);

    column.getColumn().setWidth(100);
    column.getColumn().setText("Description");

    // more code for your table, e.g. filter, etc.

    // use data binding to bind the viewer
    writableList = new WritableList(todoService.getTodos(), Todo.class);
    ViewerSupport.bind(viewer,
        writableList,
        BeanProperties.values(new String[] { Todo.FIELD_SUMMARY,
            Todo.FIELD_DESCRIPTION }));
}

public void updateViewer(List<Todo> list) {
    if (viewer != null) {
        writableList.clear();
        writableList.addAll(list);
    }
}

```

```
    }  
}  
  
@Focus  
private void setFocus() {  
    btnLoadData.setFocus();  
}  
}
```

95.2. Clean up old code

Warning

You should not have any call to the `setInput`, `setContentProvider` and `setLabelProvider` methods anymore in your code. Use the text search to validate this in case you are having problems.

95.3. Validating

Start your application and load the data into the table via the *Load data* button. Ensure that the data is displayed correctly in the table.

Note

Currently the selection of the table is not sent to the TodoDetailsPart. This will be implemented in [Chapter 99, Exercise: Selection service](#).

Part XVIII. Using Eclipse services

Chapter 96. Eclipse platform services

96.1. What are Eclipse platform services?

Services are software components (based on an interface or a class) which provide functionality. The Eclipse platform defines several services. The classes which are created based on the application model can access these services via dependency injection.

To use an Eclipse service you specify the service dependency via an `@Inject` annotation and the Eclipse framework injects this component into your object.

The typical naming convention for Eclipse service interfaces is to start with an *E* and end with *Service* e.g. *E*Service*.

96.2. Overview of the available platform services

The following table gives an overview of the most important available platform services.

Table 96.1. Platform services

Service	Description
EModelService	Used to search for elements in the model, create new model elements, clone existing snippets and insert new elements into the runtime application model.
ESelectionService	Used to retrieve and set the current active selection in the user interface.
ECommandService	Gives access to existing commands and allows you to create and change commands.
EHandlerService	Allows you to access, change and trigger handlers.
EPartService	Provides API to access and modify parts. It also allows you to switch perspectives and can be used to trigger that a method annotated with <code>@Persist</code> in dirty parts, i.e. if the corresponding part behaves like an editor.
IEventBroker	Provides functionality to send event data and to register for specified events and event topics.
EContextService	Activate and deactivate key bindings defined as <code>BindingContext</code> in the application model. The content referred to in this service is the <code>BindingContext</code> and not the <code>IEclipseContext</code> .
IThemeEngine	Allows to switch the styling of the application at runtime.
EMenuService	Registers a popup menu (<code>MPopupMenu</code>) for a control

Other available services are:

- `org.eclipse.e4.core.services.Adapter` - An adapter can adapt an object to the specified type, allowing clients to request domain-specific behavior for an object. It integrates `IAdaptable` and `IAdapterManager`. See the [Adapter wiki](#) for details.
- `org.eclipse.e4.core.services.Logger` - Provides logging functionality
- `org.eclipse.jface.window.IShellProvider` - allows access to a `Shell`, depends on SWT.

Chapter 97. Implementation

97.1. How are Eclipse platform services implemented?

Usually services have two parts: the interface definition and the implementation. How these two are linked is defined by a context function, an OSGi service or plain context value setting (`IEclipseContext`). Please note that there can be more than one service implementation for an interface.

97.2. References

How context functions are implemented as OSGI services is described in [Part XXVI, “Eclipse context functions”](#). The dedicated section [Part XLII, “Good development practices”](#) summarize the different approaches for communicating inside your application.

Part XIX. Selection Service

Chapter 98. Selection service

98.1. Usage of the selection service

The `ESelectionService` service allows you to retrieve and set the global selection in your current application window. Other classes in the application model can use the dependency injection mechanism to retrieve the relevant active selection directly.

A client can get the selection service via: `@Inject ESelectionService`.

Tip

The selection is window specific, i.e. stored in the context of the `MWindow` model object.

98.2. Changing the current selection

You can change the current selection with the `setSelection()` method of the `ESelectionService` class. This is demonstrated in the following code.

```
// use field injection for the service
@Inject ESelectionService selectionService;

// viewer is a JFace Viewer
viewer.addSelectionChangedListener(new ISelectionChangedListener() {
    @Override
    public void selectionChanged(SelectionChangedEvent event) {
        IStructuredSelection selection = (IStructuredSelection)
            viewer.getSelection();
        selectionService.setSelection(selection.getFirstElement());
    }
});
```

98.3. Getting the selection

A client can retrieve the last selection for the current window directly from the `ESelectionService` via the `getSelection()` method. The `getSelection(partId)` method allows you to retrieve the selection of a specific part.

Tip

The possibility of retrieving the selection of a part is based on the hierarchy of the `IEclipseContext`. With this hierarchy it is possible to store a selection per window and also per part.

It is preferred that a class which is part of the application model uses dependency injection to retrieve the selection. The selection is stored under the key based on the `IServiceConstants.ACTIVE_SELECTION` constant. This key can be specified via the `@Named` annotation. The Eclipse framework ensures that selections are only injected if they have the fitting type.

The usage of the `@Named` annotation to retrieve the selection is demonstrated with the following method.

```
@Inject
public void setTodo(@Optional
    @Named(IServiceConstants.ACTIVE_SELECTION) Todo todo) {
    if (todo != null) {
        // do something with the value
    }
}
```

Chapter 99. Exercise: Selection service

99.1. Target of this exercise

In this section you propagate the selection of the `TableViewer` in the `TodoOverviewPart` to your `TodoDetailsPart`.

The `org.eclipse.e4.ui.services` plug-in must be defined as dependency in the `MANIFEST.MF` file of your application plug-in. You have done this in [Chapter 64, Exercise: Prepare TodoDetailsPart for data](#). Validate that this is the case.

99.2. Retrieving the selection service

In your `TodoOverviewPart` class, acquire the `ESelectionService` service via dependency injection.

```
@Inject ESelectionService service;
```

99.3. Setting the selection in TodoOverviewPart

Use the selection service to set the selection, once the user selects an entry in the list. The following code demonstrates that.

```
// after the viewer is instantiated
viewer.addSelectionChangedListener(new ISelectionChangedListener() {
@Override
public void selectionChanged(SelectionChangedEvent event) {
    IStructuredSelection selection =
        (IStructuredSelection) viewer.getSelection();
    // selection.getFirstElement() returns a Todo object
    service.setSelection(selection.getFirstElement());
}
});
```

99.4. Review TodoDetailsPart

If you followed the exercises in [Chapter 64, Exercise: Prepare TodoDetailsPart for data](#), your `TodoDetailsPart` class should already be correctly implemented. The relevant part of the code is repeated in the following snippet.

```
@Inject
public void setTodo(@Optional @Named(IServiceConstants.ACTIVE_SELECTION) Todo to
    if (todo != null) {
        // remember todo as field
        this.todo = todo;
    }
    // update the user interface
    updateUserInterface(todo);
}
```

99.5. Validate selection propagation

Start your application. If you select a row in the JFace table, the corresponding `Todo` object is propagated via the selection service to the `TodoDetailsPart` class.

Warning

Please keep in mind that ALL injections run before `@PostConstruct`, e.g., the annotated constructor, the annotated fields and the annotated setter methods. `@PostConstruct` is executed afterwards. Hence the `updateUserInterface()` method needs to check if your user interface is already available.

Note

It might also happen that `setTodo()` is called before the `@PostConstruct` method. For example, your part may be in a stack but is not selected. Eclipse does not construct the object for the part before it becomes selected by the user. If the selection is set before the part is constructed, its `@Inject` method is called before the `@PostConstruct` method is called.

The `updateUserInterface()` method call at the end of your `@PostConstruct` method covers this case.

Tip

If you see unwanted updates of `Todo` items in your table, you did not implement the data binding correctly in the `TodoOverviewPart` class. Ensure that you call the `dispose()` method on your instance of the `DataBindingContext` before creating a new binding. Otherwise the binding between the user interface widgets and the old `Todo` objects stays active. A new `Todo` will update the user interface and the user interface binding will update all the old `Todo` objects, which is not what you want.

Chapter 100. Exercise: Selection service for deleting data

100.1. Implement the RemoveTodoHandler handler

Note

This exercise is optional.

Implement `RemoveTodoHandler` so that the selected `Todo` is deleted.

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

import javax.inject.Named;

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

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;

public class RemoveTodoHandler {
    @Execute
    public void execute(ITodoService model,
        @Optional @Named(IServiceConstants.ACTIVE_SELECTION) Todo todo,
        Shell shell)
    {
        if (todo != null) {
            model.deleteTodo(todo.getId());
        } else {
            MessageDialog.openInformation(shell, "Deletion not possible",
                "No todo selected");
        }
    }
}
```

100.2. Validate that the deletion works

Run your application and select a line in your JFace table. Select your Edit → Remove Todo menu entry.

Press the "Load Data" button to update the table. Validate that the selected Todo is not available anymore.

Note

Currently your parts are not reacting to changes in your data model and you have to press the *Load Data* button in your TodoOverviewPart part. You will fix that after you learned about the event bus of Eclipse in [Part XXIV, “Event service for message communication”](#).

100.3. Test the context menu for deletion

Tip

This exercise is optional

In [Chapter 79, Exercise: Add a context menu to a table](#) you defined a context (popup) menu for your table. As the `RemoveTodoHandler` class is already correctly implemented, you should be able to delete selected items with the *Delete* context menu without any changes in the `RemoveTodoHandler` handler class.

Ensure that this works correctly. To validate that the select item got deleted use the *Load Data* button.

Part XX. Model service and model modifications at runtime

Chapter 101. Model service

101.1. What is the model service?

The *model service* gives you access to the application model at runtime and allows you to modify it. For example, you can add and remove model elements. It also contains functionality to clone application model snippets which can be added to the application.

101.2. How to access the model service

This service can be accessed via dependency injection. For example, via field injection with the following statement: @Inject EModelService modelService;

101.3. Cloning elements or snippets

In the application model you can create `Snippet` model elements which can be used to create model objects at runtime. It is also possible to copy existing application model elements via the model service.

You can use the `cloneElement()` and `cloneSnippet()` methods of the model service to copy an existing element or snippet. The resulting object can be assigned to another model element.

101.4. Searching model elements

The `findElements()` method allows you to search for specific model elements. The following code shows an example for using the `findElements()` method.

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

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

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.model.application.MApplication;
import org.eclipse.e4.ui.model.application.ui.MUIElement;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.e4.ui.workbench.modeling.EModelService;

public class ModelServiceExampleHandler {

    @Execute
    public void execute(MApplication application, EModelService service) {

        // find objects by ID
        findPartsById(application, service);

        // find objects by type
        findParts(application, service);

        // find objects by tags
        findObjectsByTag(application, service);
    }

    // example for search by ID
    private void findPartsById(MApplication application, EModelService service) {
        List<MPart> parts = service.findElements(application, "mypart",
            MPart.class, null);
        System.out.println("Found part(s) : " + parts.size());
    }

    // example for search by type
    private void findParts(MApplication application,
        EModelService service) {
        List<MPart> parts = service.findElements(application, null,
            MPart.class, null);
        System.out.println("Found parts(s) : " + parts.size());
    }

    // example for search by tag
    private void findObjectsByTag(MApplication application,
        EModelService service) {
```

```

        List<String> tags = new ArrayList<String>();
        tags.add("justatag");
        List<MUIElement> elementsWithTags = service.findElements(application,
            null, null, tags);
        System.out.println("Found parts(s) : " + elementsWithTags.size());
    }
}

```

Here is an example of how to find the perspective for a part.

```

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

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

public class FindPerspectiveHandler {
    @Execute
    public void execute(MApplication application, EModelService service) {
        // search for a part with the following ID
        String ID = "com.example.e4.rcp.parts.tododetail";
        MUIElement element = service.find(ID, application);
        MPerspective perspective = service.getPerspectiveFor(element);
        System.out.println(perspective);
        // TODO do something useful with the perspective
    }
}

```

You can also use the `findElements` method with a fifth parameter which allows you to specify additional search flags, e.g., `IN_ACTIVE_PERSPECTIVE`, `OUTSIDE_PERSPECTIVE`, `IN_ANY_PERSPECTIVE`. See the Javadoc of the `findElements` method for further details.

Chapter 102. Modifying the application model at runtime

102.1. Creating model elements

As the application model is interactive, you can change it at runtime. For example you can add parts to your application or remove menu entries.

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

102.2. Modifying existing model elements

You can also access existing model elements, via the model service or via dependency injection and adjust their attributes.

The Eclipse framework automatically keeps track of the application model and changes in the model are reflected immediately in your application.

For example, if you add a new window to your application, it becomes visible instantly. Or if you inject an MPart object and call its `setLabel()` method, the text of the part in a `PartStack` changes immediately.

Chapter 103. Example for application model modifications

103.1. Example: Search for a perspective and change its attributes

The following code shows how to access a `PartSashContainer` with the `mypartsashcontainer` ID. It also demonstrates how to modify model attributes.

In this example it changes the `container data` parameter for its children. This will arrange (layout) the parts in the container.

```
@Execute
public void execute(EModelService service, MWindow window) {
    MPartSashContainer find = (MPartSashContainer) service.
        find("mypartsashcontainer", window);
    List<MPartSashContainerElement> list = find.getChildren();

    int i = 0;
    // make the first part in the container larger
    for (MPartSashContainerElement element : list) {

        if (i > 0) {
            element.setContainerData("20");
        } else {
            element.setContainerData("80");
        }
        i++;
    }
}
```

103.2. Example: Dynamically create a new window

To create new model objects you can use the `createModelElement()` of the model service. After you created such an object you can add it to your application model at runtime.

For example, the creation of a new window for a running application is demonstrated by the following code snippet.

```
// create a new window and set its size
MWindow window = modelService.createModelElement(MWindow.class);
window.setWidth(200);
window.setHeight(300);

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

103.3. Example: Dynamically create a new part

The following code demonstrates how to create and add a new part to the currently active window.

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

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.model.application.MApplication;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.e4.ui.workbench.modeling.EModelService;
import org.eclipse.e4.ui.workbench.modeling.EPartService;
import org.eclipse.e4.ui.workbench.modeling.EPartService.PartState;

public class DynamicPartHandlerCode {
    // used as reference
    @Execute
    public void execute(MApplication application, EPartService partService,
        EModelService modelService) {

        // create new part
        MPart mPart = modelService.createModelElement(MPart.class);
        mPart.setLabel("Testing");
        mPart.setElementId("newid");
        mPart.setContributionURI("bundleclass://com.example.e4.rcp.todo/"
            + "com.example.e4.rcp.todo.parts.DynamicPart");
        partService.showPart(mPart, PartState.ACTIVATE);
    }
}
```

Tip

Typically you would add the part to a pre-defined `PartStack`. Use the model service to search for the correct one.

Chapter 104. Exercise: Creating dynamic menu entries

104.1. Target of this exercise

Note

This exercise is optional.

In this exercise you create dynamic menu entries with the model service. For this you use the *DynamicMenuContribution* model element which was introduced in the [Section 78.3, “Dynamic menu and toolbar entries”](#) chapter.

104.2. Create handler class

Create the following class in your application plug-in. This simple class is later used as example handler.

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

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

public class PrintMessageHandler {

    @Execute
    public void printMessage() {
        System.out.println("message printed....");
    }
}
```

104.3. Create class for DynamicMenuContribution

Create the following implementation for the *DynamicMenuContribution* in your application plug-in.

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

import java.util.Date;
import java.util.List;

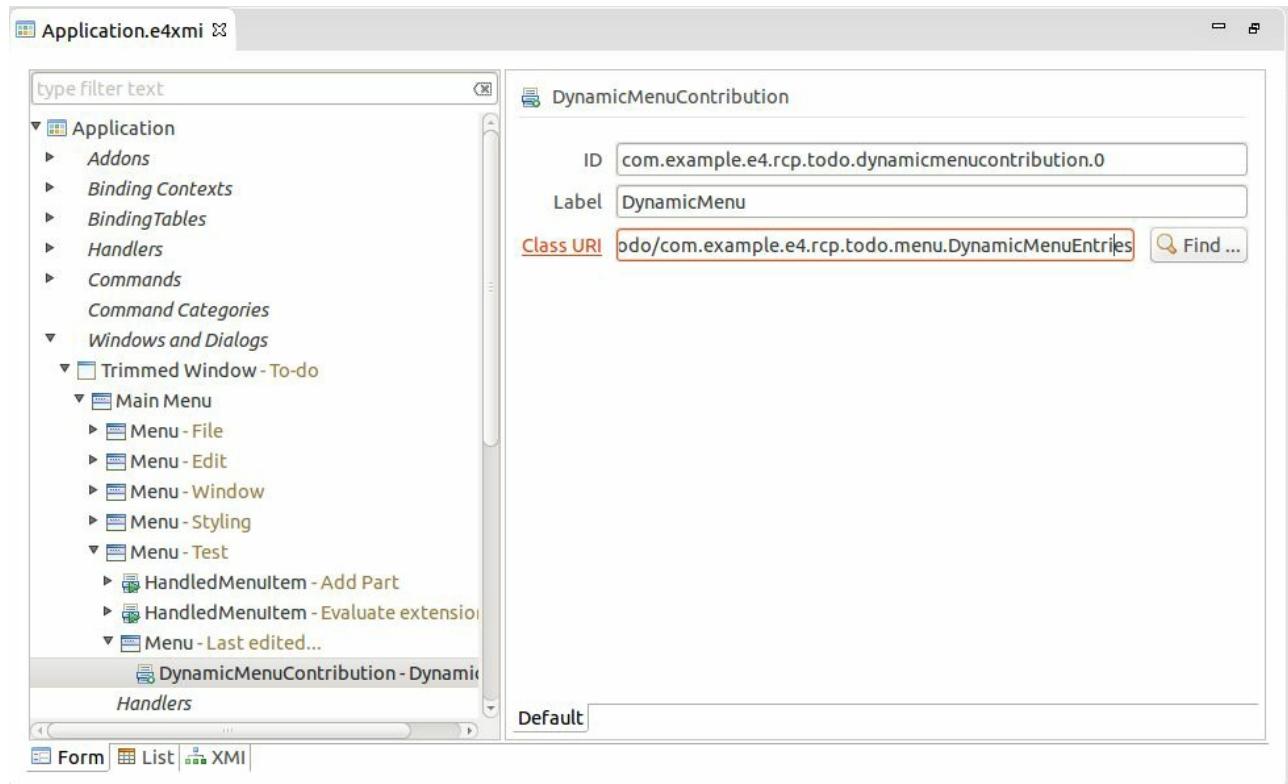
import org.eclipse.e4.ui.di.AboutToShow;
import org.eclipse.e4.ui.model.application.ui.menu.MDirectMenuItem;
import org.eclipse.e4.ui.model.application.ui.menu.MMenuElement;
import org.eclipse.e4.ui.workbench.modeling.EModelService;

public class DynamicMenuEntries {
    private String bundleprefix = "bundleclass://com.example.e4.rcp.todo/";
    private String classname = "com.example.e4.rcp.todo.handlers.PrintMessageHandl

    @AboutToShow
    public void aboutToShow(List<MMenuElement> items, EModelService modelService)
        for (int i = 0; i < 10; i++) {
            MDirectMenuItem dynamicItem = modelService
                .createModelElement(MDirectMenuItem.class);
            dynamicItem.setLabel("Dynamic Menu Item (" + new Date() + ")");
            // ensure that you really use setContributionURI
            // and NOT setContributorURI
            dynamicItem
                .setContributionURI(bundleprefix+classname);
            items.add(dynamicItem);
        }
    }
}
```

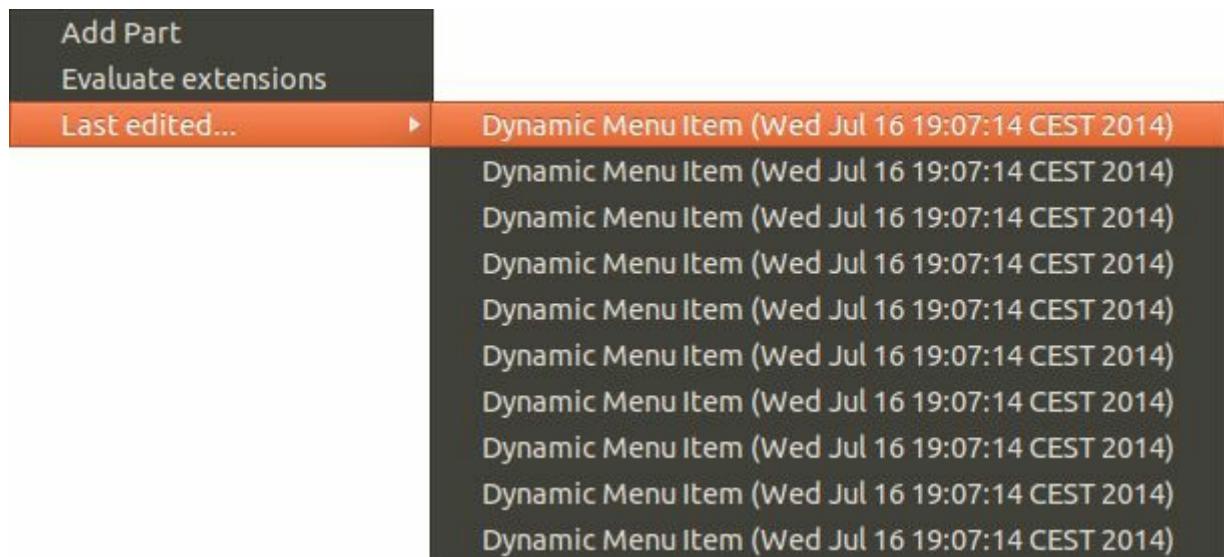
104.4. Add DynamicMenuContribution model element

Add to one of for menu another menu entry called "Recently opened..." and add a *DynamicMenuContribution* to this menu, pointing to your *DynamicMenuEntries.java* class.



104.5. Validating

Start your application and validate that the menu entries are dynamically calculated.



Part XXI. Part service and implementing editors

Chapter 105. Using the part service

105.1. What is the part service?

The *part service* allows you to find and perform actions on parts in the application model.

It also allows you to switch perspectives and to create and activate new parts based on part descriptors in the application model.

105.2. How to access the part service

Use dependency injection to get access to the part service. For example via the `@Inject EPartService partService;` statement.

105.3. Example: Showing and hiding parts

The following example shows how you can find parts, hide or show them. If the `Visible` attribute of the part was initially set to `false` (not visible), you need to call the `setVisible(true)` method of the model element to ensure that the part gets displayed.

```
@Inject private EPartService partService;

// search part with ID "com.example.todo.rcp.parts.tododetails"
// assume that a part with this ID exists
detailsTodoPart = partService.findPart("com.example.todo.rcp.parts.tododetails")

// hide the part
partService.hidePart(detailsTodoPart);

//show the part
detailsTodoPart.setVisible(true); // required if initial not visible
partService.showPart(detailsTodoPart, PartState.VISIBLE);
```

105.4. Example: Switching perspectives

The following example shows how you can switch to another perspective with the part service.

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

import java.util.List;

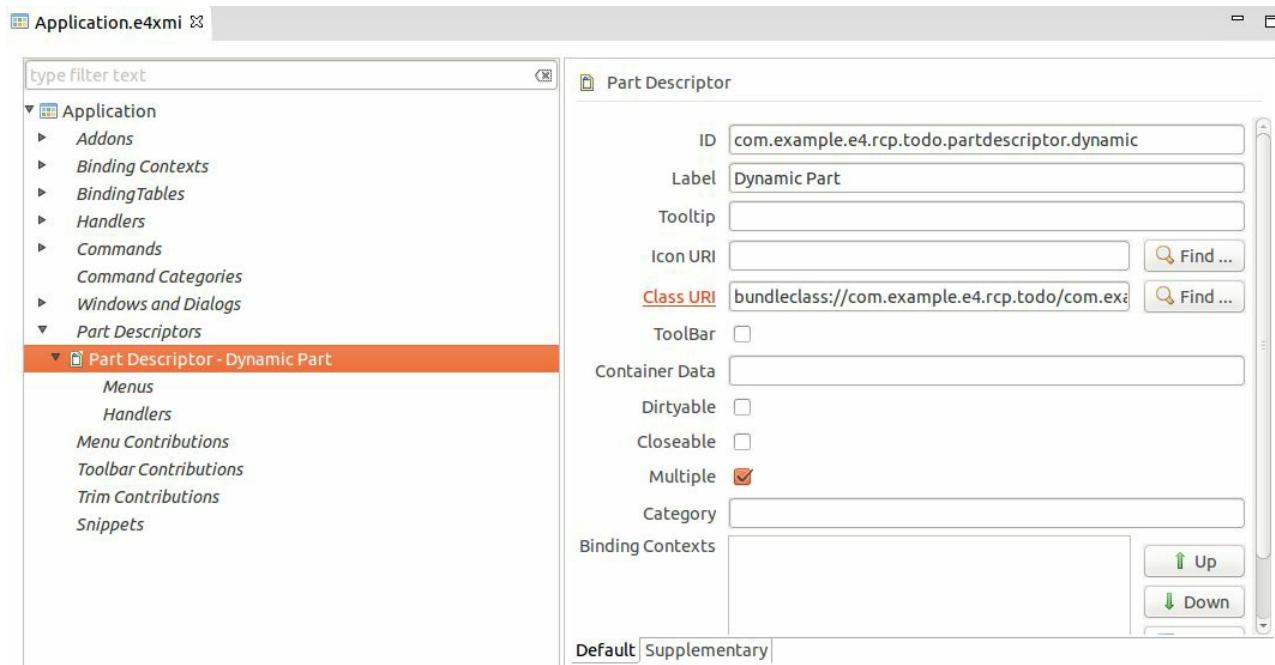
import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.model.application.MApplication;
import org.eclipse.e4.ui.model.application.ui.MUIElement;
import org.eclipse.e4.ui.model.application.ui.advanced.MPerspective;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import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

public class SwitchPerspectiveHandler {
    @Execute
    public void execute(MApplication app, EPartService partService,
        EModelService modelService) {
        MPerspective element =
            (MPerspective) modelService.find("secondperspective", app);
        // now switch perspective
        partService.switchPerspective(element);
    }
}
```

105.5. Using part descriptors

The *part descriptor* model element is a template for the creation of a part. By defining a common set of attributes via such a blueprint it is possible to create concrete instances of it via the part service.

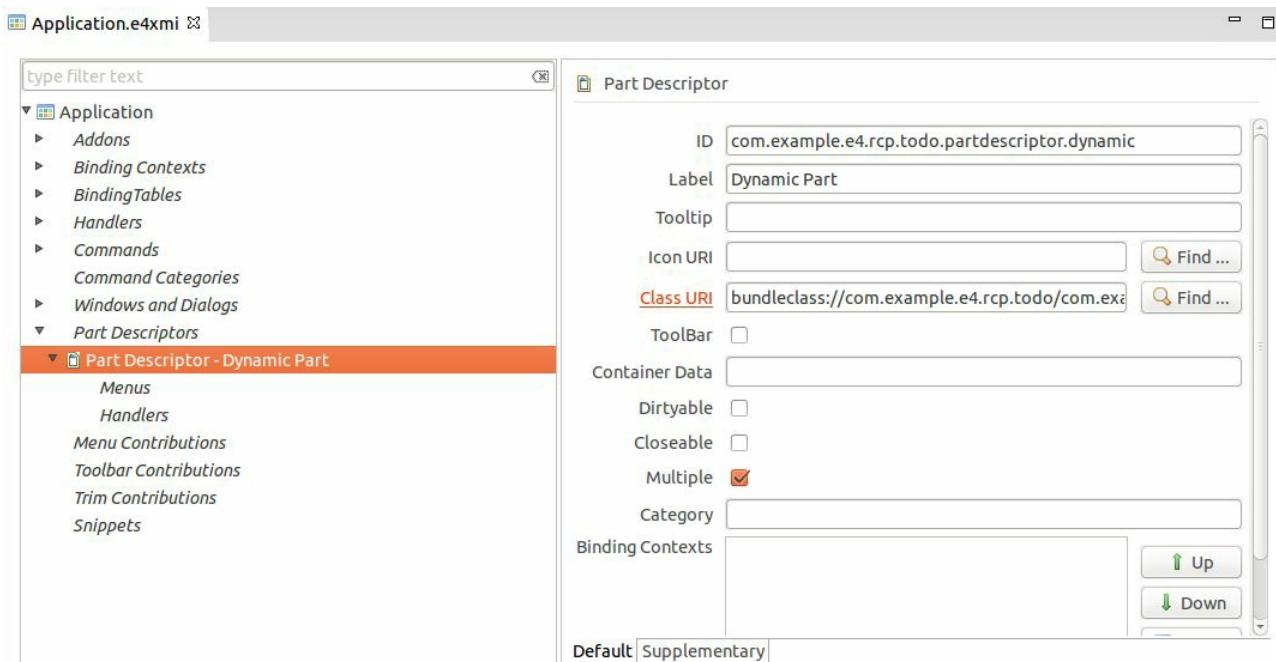
Via the *Multiple* of the part descriptor you configure if multiple instances of this part can be created or not. Such a model element is depicted in the following screenshot.



105.6. Example: Part descriptors and creating parts dynamically

If you define a part descriptor in your application model, you can use the `EPartService` to create a part from it.

The following screenshot shows the definition of a part descriptor in the application model. As the `Multiple` parameter is set, it is possible to create several parts based on this template.



The part service allows you to create a new part based on this template. This is demonstrated by the following example code.

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

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.e4.ui.workbench.modeling.EPartService;
import org.eclipse.e4.ui.workbench.modeling.EPartService.PartState;

public class OpenPartHandler {

    // the following code assumes that
    // the "com.example.e4.rcp.todo.partdescriptor.fileeditor" ID
    // is used for the part descriptor
    @Execute
    public void execute(EPartService partService) {

        // create a new part based on a part descriptor
        // if multiple parts of this type are allowed a new part
    }
}
```

```
// is always generated

MPart part = partService
    .createPart("com.example.e4.rcp.todo.partdescriptor.fileeditor");
part.setLabel("New Dynamic Part");

// the provided part is be shown
partService.showPart(part, PartState.ACTIVATE);
}
}
```

Chapter 106. Implementing editors

106.1. Parts which behave similar to editors

An editor is a part which requires that the user triggers a save operation to persist data changes in the editor. Editors that contain data, which can be saved, are typically called *dirty*.

The part service allows you to save dirty parts. Every part can mark itself as dirty, hence behave like an editor.

106.2. MDirtyable and @Persist

A part has the `MDirtyable` attribute which indicates that it can be marked as dirty. Dirty indicates that the part contains data which has been changed but not yet saved. The `MDirtyable` object can get injected into a part.

You can use the `setDirty(boolean)` method to mark the part as dirty.

The following snippet demonstrates how to use the `MDirtyable` model property in a part to flag it as dirty after a button was pressed.

```
import javax.annotation.PostConstruct;
import javax.inject.Inject;

import org.eclipse.e4.ui.di.Persist;
import org.eclipse.e4.ui.model.application.ui.MDirtyable;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;

public class MySavePart {

    @Inject
    MDirtyable dirty;

    @PostConstruct
    public void createControls(Composite parent) {
        Button button = new Button(parent, SWT.PUSH);
        button.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {
                dirty.setDirty(true);
            }
        });
    }
}
```

The part service allows you to query the dirty parts and to call a method annotated with `@Persist` on the dirty parts. This method saves the data of the part and sets the dirty flag back to false if the save operation was successful.

```
@Persist
public void save(MDirtyable dirty, ITodoService todoService) {
    // save changes via ITodoService for example
    todoService.saveTodo(todo);
```

```
// save was successful  
dirty.setDirty(false);  
}
```

Note

Every part is responsible for saving itself. Hence every part which behaves like an editor must have one method annotated with `@Persist`.

106.3. Use part service to trigger save in editors

The part service allows you to trigger the `@Persist` method on the dirty parts via the `saveAll()` method.

The `EPartService` searches in each part which is marked as dirty for a method annotated with `@Persist`. This method is called by the framework and has to save the data which the editor holds. If saving the data was successful it should call the `setDirty(false)` method on the `MDirtyable` object.

The following example demonstrates that.

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

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

public class SaveHandler {

    @Execute
    void execute(EPartService partService) {
        partService.saveAll(false);
    }
}
```

106.4. MPart and multiple editors

You can use the *MPart* model element to create multiple editors. Every model element can get persisted data assigned which can be accessed via the `getPersistedState()` method.

In its `@PostConstruct` method the implementation class can get the `MPart` injected and access its persisted state. This information can be used to configure the editor.

106.5. MInputPart

Eclipse used to have the `MInputPart` model element. This model element has been deprecated and should not be used anymore. Use the approach described in [Section 106.4, “MPart and multiple editors”](#).

106.6. Code examples for editor implementations

How to implement editor like behavior in a part is demonstrated in [Chapter 107, Exercise: Implement an editor](#). An example implementation of multiple editors can be found in exercise [Chapter 112, Exercise: Implement multiple editors](#).

Chapter 107. Exercise: Implement an editor

107.1. Add the plug-in dependencies

Note

If you did the optional exercise [Chapter 79, Exercise: Add a context menu to a table](#) you should have already the required dependencies in your `com.example.e4.rcp.todo` plug-in. Skip this step in this case.

Add the following plug-ins as dependency to the `MANIFEST.MF` file of the `com.example.e4.rcp.todo` plug-in.

- `org.eclipse.e4.ui.model.workbench`
- `org.eclipse.e4.ui.workbench.swt`

107.2. Convert TodoDetailsPart to an editor

Use the `MDirtyable` attribute to mark the `TodoDetailsPart` as dirty once the user changes data via the user interface.

The following example code shows how to get the attribute and how you can define a listener to your existing data binding (which you created in [Chapter 93, Exercise: Data Binding for SWT widgets](#)).

```
// more import statements

// relevant databinding import statements
import org.eclipse.core.databinding.Binding;
import org.eclipse.core.databinding.DataBindingContext;
import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.observable.ChangeEvent;
import org.eclipse.core.databinding.observable.IChangeListener;
import org.eclipse.core.databinding.observable.list.IObservableList;
import org.eclipse.core.databinding.observable.value.IObservableValue;
import org.eclipse.e4.core.di.annotations.Optional;

//more import statements

// define dirty as field
@Inject MDirtyable dirty;

// new Listener field for the notifications of
// data binding changes
IChangeListener listener = new IChangeListener() {
    @Override
    public void handleChange(ChangeEvent event) {
        if (dirty!=null){
            dirty.setDirty(true);
        }
    }
};

// more code in TodoDetailsPart
// ...
// ...


// UPDATED METHOD
private void updateUserInterface(Todo todo) {
    // if Todo is null disable user interface
    // and leave method
    if (todo == null) {
        enableUserInterface(false);
        return;
    }
}
```

```

// the following check ensures that the user interface is available,
// it assumes that you have a text widget called "txtSummary"
if (txtSummary != null && !txtSummary.isDisposed()) {
    enableUserInterface(true);

    // NEW!
    // unregister change listener from the existing binding
    // must be done before the ctx.dispose() call
    IObservableView providers = ctx.getValidationStatusProviders();
    for (Object o : providers) {
        Binding b = (Binding) o;
        b.getTarget().removeChangeListener(listener);
    }

    // disposes existing bindings
    ctx.dispose();

    IObservableValue oWidgetSummary = WidgetProperties.text(SWT.Modify)
        .observe(txtSummary);
    IObservableValue oTodoSummary = BeanProperties.value(Todo.FIELD_SUMMARY)
        .observe(todo);
    ctx.bindValue(oWidgetSummary, oTodoSummary);

    IObservableValue oWidgetDescription = WidgetProperties.text(SWT.Modify)
        .observe(txtDescription);
    IObservableValue oTodoDescription = BeanProperties.value(Todo.FIELD_DESCRIPTION)
        .observe(todo);
    ctx.bindValue(oWidgetDescription, oTodoDescription);

    IObservableValue oWidgetButton = WidgetProperties.selection().observe(btnDone);
    IObservableValue oTodoDone = BeanProperties.value(Todo.FIELD_DONE).observe(todo);
    ctx.bindValue(oWidgetButton, oTodoDone);

    IObservableValue oWidgetSelectionDateTime = WidgetProperties
        .selection().observe(dateTime);
    IObservableValue oTodoDueDate = BeanProperties.value(Todo.FIELD_DUEDATE).observe(todo);
    ctx.bindValue(oWidgetSelectionDateTime,
        oTodoDueDate);

    // register listener for any changes
    providers = ctx.getValidationStatusProviders();
    for (Object o : providers) {
        Binding b = (Binding) o;
        b.getTarget().addChangeListener(listener);
    }
}

```

Tip

The null check of the dirty field in the `IChangeListener` is required as you reuse the `TodoDetailsPart` class in your wizard page. The wizard page does not set the dirty field.

Add the following method to your `TodoDetailsPart` class to save the changed data. This method will be used by the part service.

```
// dirty already injected as field  
  
@Persist  
public void save(ITodoService todoService) {  
    todoService.saveTodo(todo);  
    dirty.setDirty(false);  
}
```

107.3. Implement the save handler

Change your `SaveAllHandler` handler class, so that the part service is used to call the `@Persist` method of all parts which are marked as dirty.

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

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

public class SaveAllHandler {

    @Execute
    void execute(EPartService partService) {
        partService.saveAll(false);
    }
}
```

In [Section 108.1, “Enable the save handler only if necessary”](#) you implement that the save handler can only be executed if a dirty editor exists.

107.4. Validating

Start your application and select an entry in the table of your *TodoOverviewPart* element. Modify the selected `Todo` in the *Details* part. Ensure that the part is marked as dirty.

Select File → Save and validate that the editor is not marked as dirty anymore.

Reload the data in the table to ensure that the data was correctly saved.

Create a new `Todo` item, with the wizard and ensure that the wizard still works.

107.5. Confirmation dialog for modified data

Note

This exercise is optional. It is a repetition of the usage of dialogs.

In your `TodoDetailsPart` class trigger a JFace confirmation dialog if the selection in the table of `TodoOverviewPart` changes and if `TodoDetailsPart` contains modified data.

The confirmation dialog should allow the user to prevent an update in `TodoDetailsPart` in case it has unsaved data.

Chapter 108. Exercise: Enable handlers and avoiding data loss

108.1. Enable the save handler only if necessary

Change your `SaveAllHandler` class so that the `save` handler is only active if there is a part in the model which has its `dirty` attribute set to true.

```
@CanExecute
boolean canExecute(@Optional EPartService partService) {
    if (partService != null) {
        return !partService.getDirtyParts().isEmpty();
    }
    return false;
}
```

108.2. Enable the deletion handler only if necessary

Note

This exercise is optional.

Similar to [Section 108.1, “Enable the save handler only if necessary”](#) enable the deletion handler for todos only if a `Todo` is selected.

108.3. Avoid data loss in your ExitHandler

Change your `ExitHandler` class so that in the case of a dirty editor the user is prompted if he wants to save the data.

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

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.workbench.IWorkbench;
import org.eclipse.e4.ui.workbench.modeling.EPartService;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Shell;

public class ExitHandler {
    @Execute
    public void execute(EPartService partService,
        IWorkbench workbench, Shell shell) {
        if (!partService.getDirtyParts().isEmpty()) {
            boolean confirm = MessageDialog.openConfirm(shell, "Unsaved",
                "Unsaved data, do you want to save?");
            if (confirm) {
                partService.saveAll(false);
                // we close the workbench here to avoid
                // the second popup
                workbench.close();
                // the Workbench continues to run, until we leave this handler
                // so we return;
                return;
            }
        }

        boolean result = MessageDialog.openConfirm(shell, "Close",
            "Close application?");
        if (result) {
            workbench.close();
        }
    }
}
```

Chapter 109. Exercise: Using multiple perspectives

109.1. Target of this exercise

In this exercise you define another perspective and implement a menu to switch between your perspectives.

109.2. Create a new perspective

Define a new perspective in your application. You should know by now how to do this, therefore it is not described in this exercise. Ensure that your new perspective has at least one part in it (contained in a PartStack so that the part label is visible).

Warning

Ensure to add the perspective to the existing PerspectiveStack.

109.3. Create new menu entries

Create a command and a handler to switch between perspectives. The handler class for switching perspectives might look like the following.

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

import java.util.List;

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

public class PerspectiveSwitchHandler {
    @Execute
    public void switchPersepctive(MPerspective activePerspective,
        MApplication app, EPartService partService,
        EModelService modelService) {
        List<MPerspective> perspectives = modelService.findElements(app, null,
            MPerspective.class, null);
        // assume you have only two perspectives and you always
        // switch between them
        for (MPerspective perspective : perspectives) {
            if (!perspective.equals(activePerspective)) {
                partService.switchPerspective(perspective);
            }
        }
    }
}
```

Create or update a *Windows* menu entry in your menu main with a sub-menu called *Switch Perspective*.

Add a new menu entry in your application model to switch between your perspectives.



Test if you can switch between your perspectives with your new handler.

109.4. Using command parameters to define the perspective ID

Tip

This exercise is optional

Currently your perspective switcher only allows to switch between two perspectives. Use parameters in your command and menu entries to specify to which perspective you want to switch. See [Section 81.1, “Passing parameters to commands”](#) for an example how to do this.

Chapter 110. Exercise: Sharing elements between perspectives

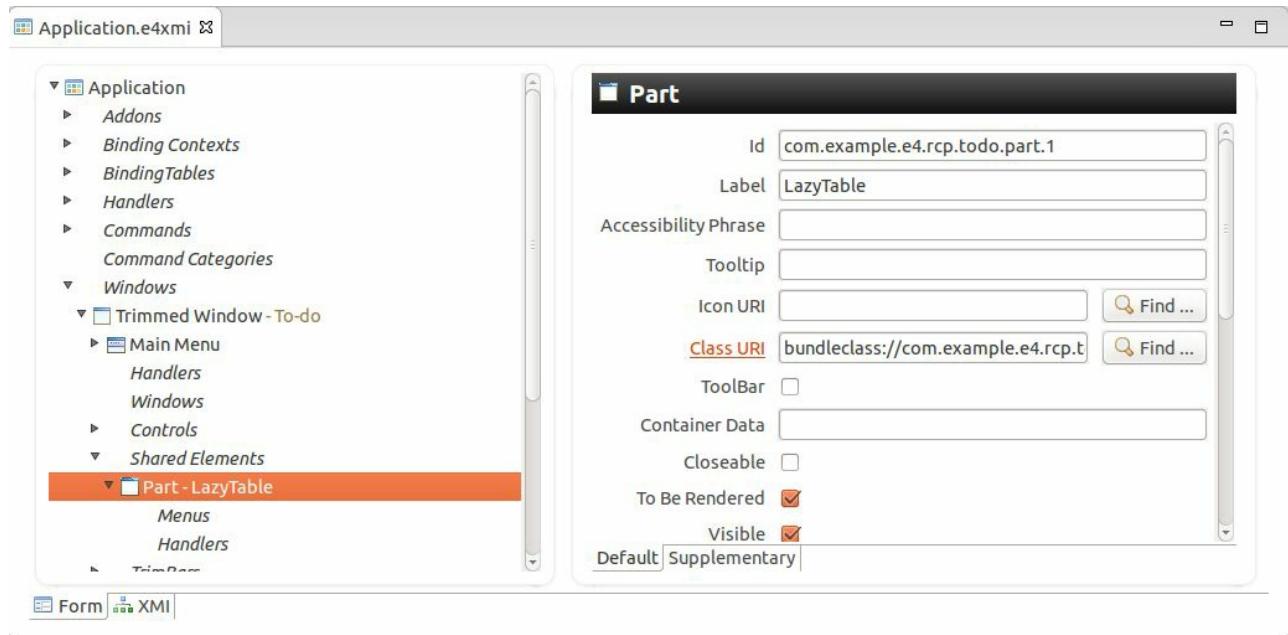
110.1. Target

This tutorial demonstrates how you can share parts between perspectives.

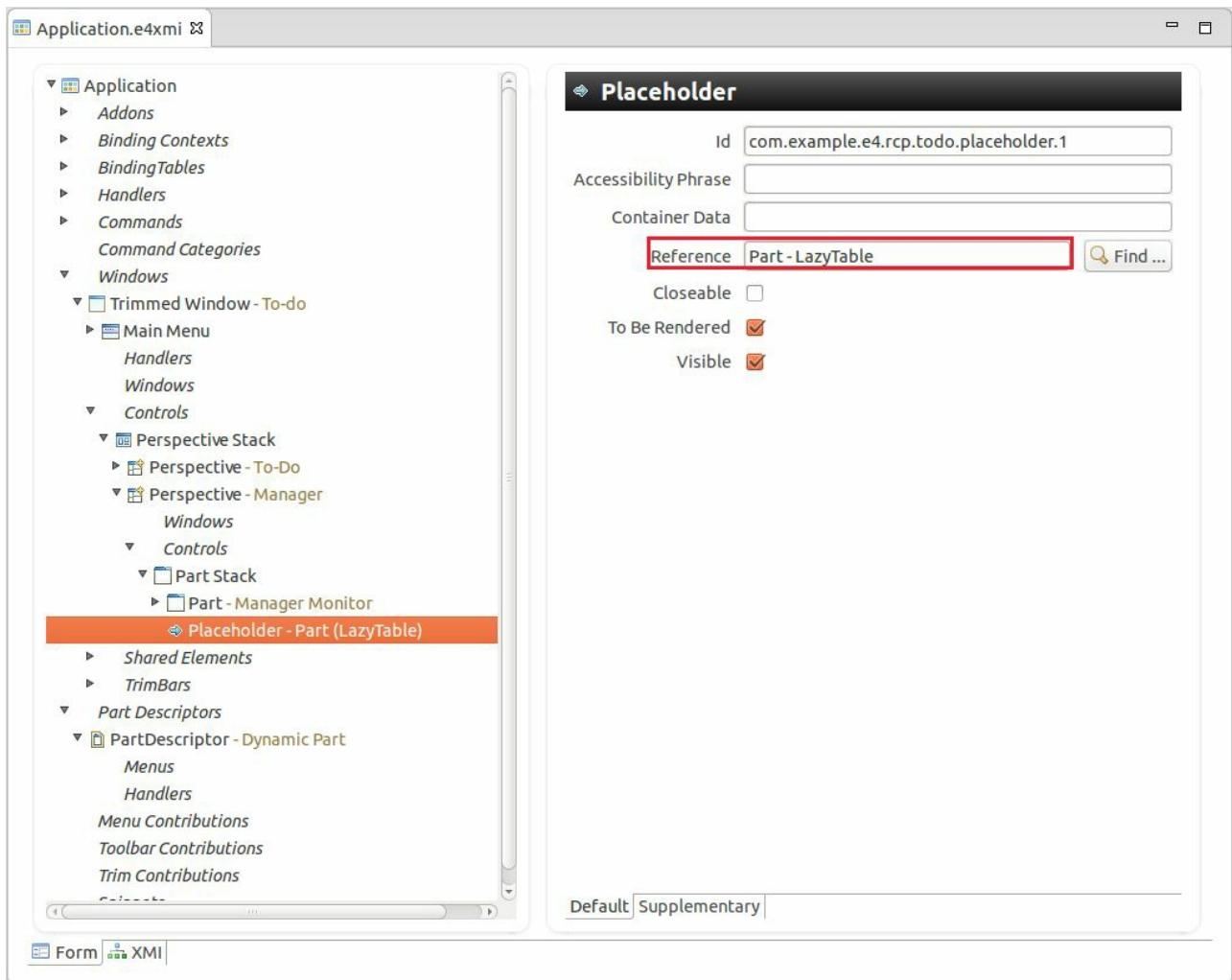
110.2. Using shared parts between perspectives

Using the *Placeholder* model element you are able to define elements which should be shared between perspectives. In this exercise you define a shared part which is displayed in both perspectives.

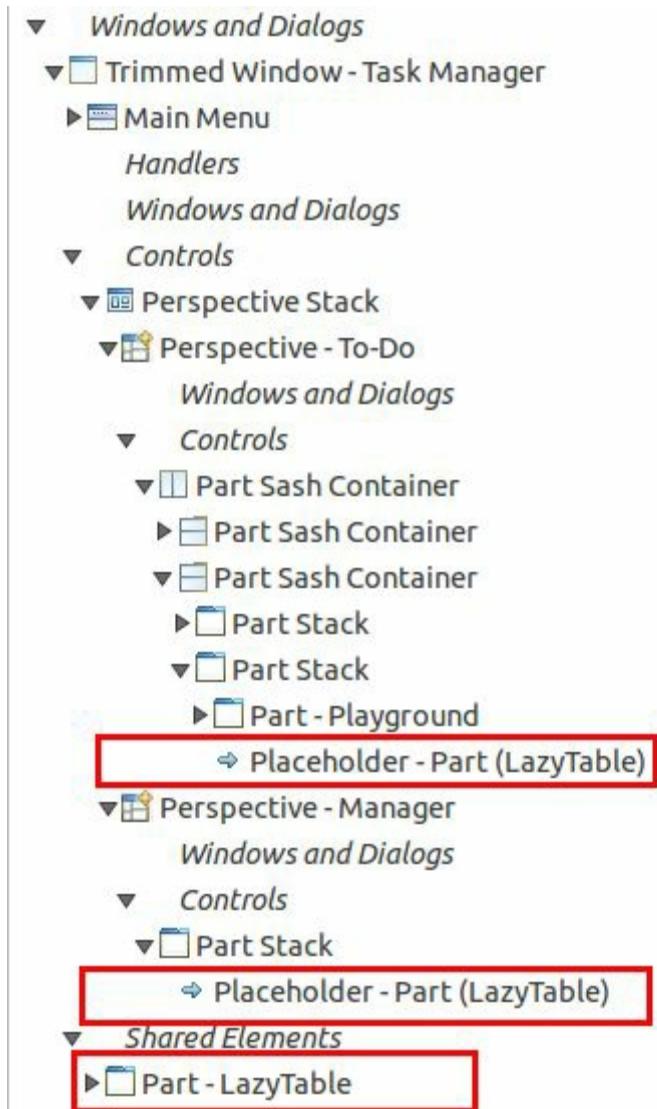
Add a part as *Shared Elements* to your application model.



Add a *Placeholder* entry to both perspectives into a stack. Point to the *Shared Elements* part in this placeholder.



As result you should have two placeholders pointing to the shared element.



If you switch between your perspectives, this part should be the same.

Tip

By using this approach the part displayed in two perspectives is the same instance, i.e. the data in the part is the same. If you define two parts using the same class reference, the Eclipse runtime creates two instances (objects).

Chapter 111. Optional exercise: Dynamic creation of parts based on a part descriptor

111.1. Create a class for the part descriptor

Create the following new class.

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

import javax.annotation.PostConstruct;

import org.eclipse.e4.ui.di.Focus;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Shell;

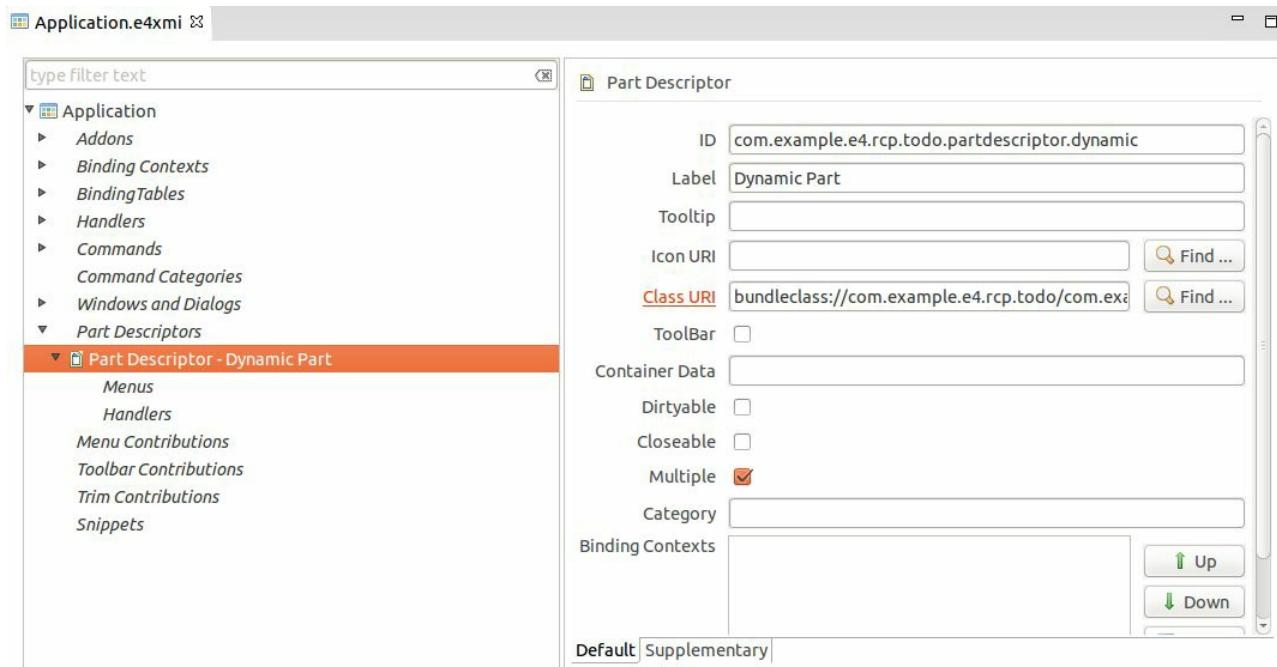
public class DynamicPart {
    private Button button;

    @PostConstruct
    public void createControls(Composite parent, final Shell shell) {
        button = new Button(parent, SWT.PUSH);
        button.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {
                MessageDialog.openInformation(shell, "Dynamic", "Dynamics are working");
            }
        });
        button.setText("Validate");
    }

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

111.2. Create the part descriptor model element

Create a *Part Descriptor* model element in your application model using the the `com.example.e4.rcp.todo.partdescriptor.dynamic` ID. In the *Class URI* field point to your new `DynamicPart` class.



The *Multiple* attribute should be flagged.

111.3. Create a handler for creating new parts

Add a new handler and a new command for creating a new part to your application model. The code for the handler class should be similar to the following code.

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

import java.util.List;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.model.application.MApplication;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.e4.ui.model.application.ui.basic.MPartStack;
import org.eclipse.e4.ui.workbench.modeling.EModelService;
import org.eclipse.e4.ui.workbench.modeling.EPartService;
import org.eclipse.e4.ui.workbench.modeling.EPartService.PartState;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Shell;

public class DynamicPartHandler {
    // Used as reference
    private final String STACK_ID = "com.example.e4.rcp.todo.partstack.bottom";
    @Execute
    public void execute(MApplication application,
        EPartService partService,
        EModelService modelService,
        Shell shell) {
        // create part based on part descriptor
        MPart part = partService.createPart("com.example.e4.rcp.todo.partdescriptor.

        // search for the part stack to add the part to
        // this means you must have a part stack with in ID in your application mode
        List<MPartStack> stacks = modelService.findElements(application, STACK_ID,
            MPartStack.class, null);
        if (stacks.size() < 1) {
            MessageDialog.openError(shell, "Error ",
                "Part stack not found. Is the following ID correct?" + STACK_ID);
            return;
        }
        stacks.get(0).getChildren().add(part);
        // activates the part
        partService.showPart(part, PartState.ACTIVATE);
    }
}
```

Add a new menu called *Dynamic Contributions* to your application model.

Also add a new menu entry to your new menu, pointing to your new command.

Start your application and validate that you can dynamically create and display new parts. As you have set the `Multiple` flag on the part descriptor, you can also open multiple instances of the part.

Chapter 112. Exercise: Implement multiple editors

Tip

This exercise is optional.

112.1. Prerequisites

This optional exercise is based on [Chapter 79, Exercise: Add a context menu to a table](#) as it reuses the context menu of the table you created there. If you skipped this exercise, you need to go back and perform the exercise before going on with this one.

112.2. New menu entries

Create a new command called *Open Editor*, create a new handler for it, pointing to the following new class `OpenEditorHandler`.

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

import javax.inject.Named;

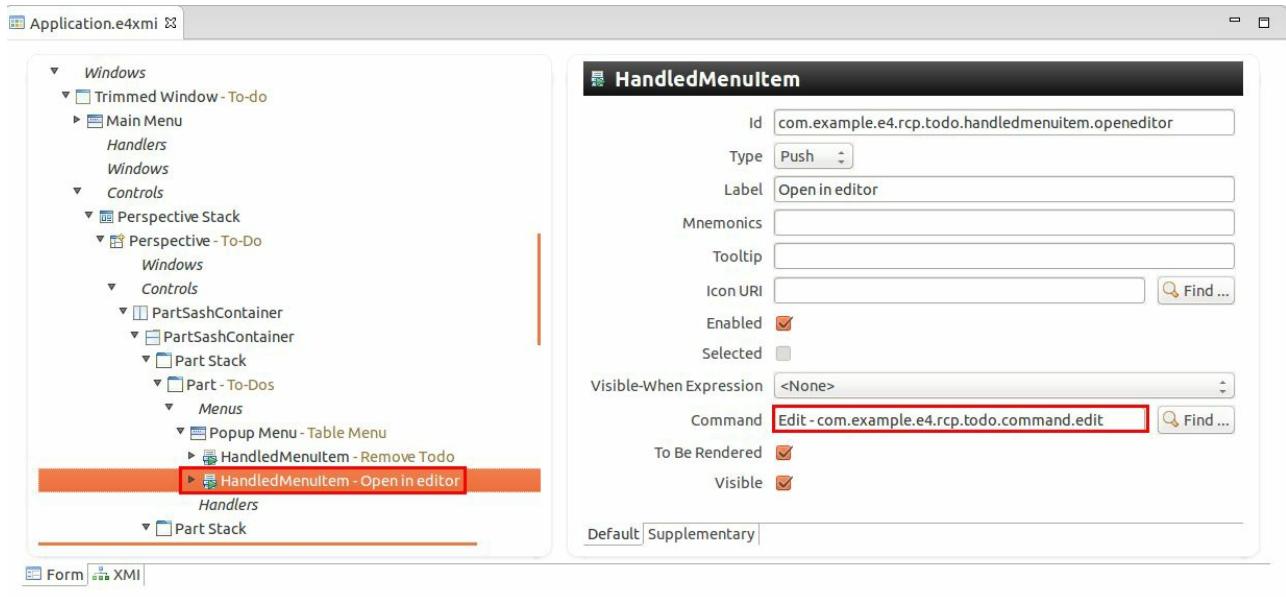
import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.core.di.annotations.Optional;
import org.eclipse.e4.ui.services.IServiceConstants;

import com.example.e4.rcp.todo.model.Todo;

public class OpenEditorHandler {
    @Execute
    public void execute(@Optional @Named(IServiceConstants.ACTIVE_SELECTION) Todo
        System.out.println("Will open a new editor");

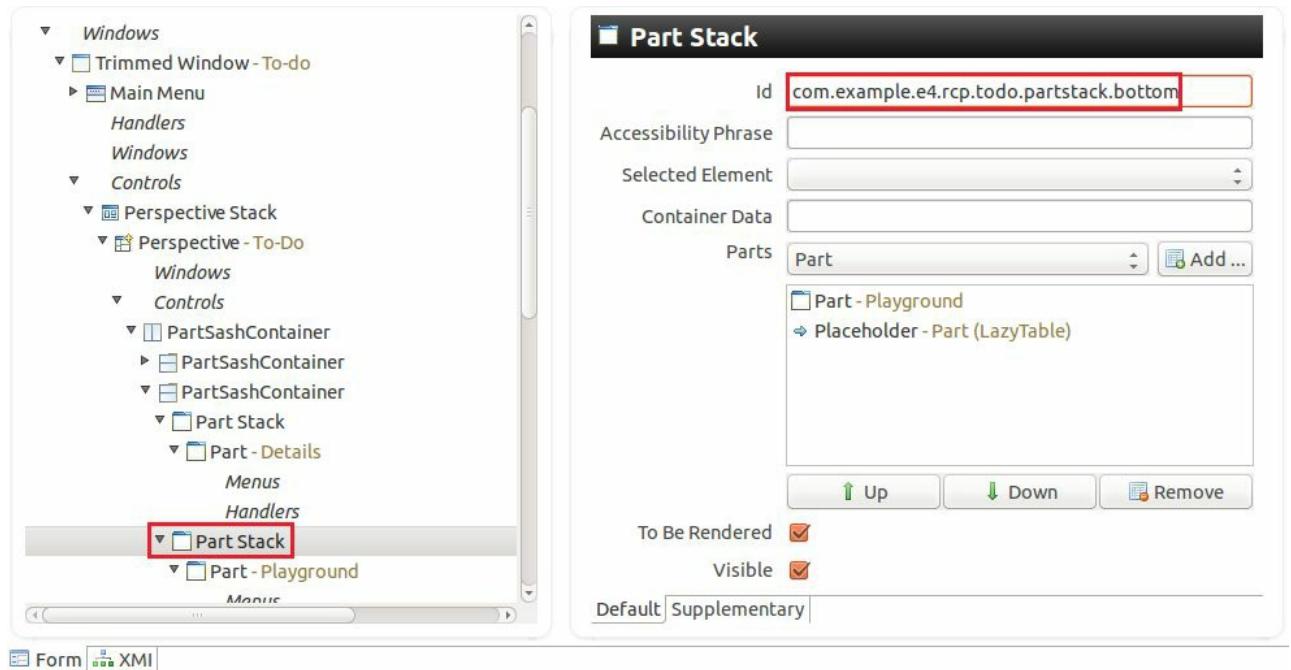
    }
}
```

Add another entry to the context menu of the table which you have created and registered with the SWT table in [Chapter 79, Exercise: Add a context menu to a table](#).



112.3. Validate ID of the PartStack

Your editors should be opened in the right lower corner of the part stack. Ensure that the ID of the part stack in the application model is set to com.example.e4.rcp.todo.partstack.bottom.



112.4. Add a handler and a part implementation

Create the following implementation for the part which represents the editor. It is based on the `TodoDetailsPart` code but avoids listening to the current active selection.

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

import javax.annotation.PostConstruct;
import javax.inject.Inject;

import org.eclipse.core.databinding.Binding;
import org.eclipse.core.databinding.DataContext;
import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.observable.ChangeEvent;
import org.eclipse.core.databinding.observable.IChangeListener;
import org.eclipse.core.databinding.observable.list.IObservableList;
import org.eclipse.core.databinding.observable.value.IObservableValue;
import org.eclipse.e4.ui.di.Focus;
import org.eclipse.e4.ui.di.Persist;
import org.eclipse.e4.ui.model.application.ui.MDirtyable;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.jface.databinding.swt.WidgetProperties;
import org.eclipse.swt.SWT;
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.DateTime;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;

public class EditorPart {

    @Inject
    MDirtyable dirty;

    private Text txtSummary;
    private Text txtDescription;
    private Button btnDone;
    private DateTime dateTime;

    private DataContext ctx = new DataContext();

    // define listener for the data binding
    IChangeListener listener = new IChangeListener() {
        @Override
        public void handleChange(ChangeEvent event) {
```

```

        if (dirty!=null){
            dirty.setDirty(true);
        }
    }
};

private Todo todo;

@PostConstruct
public void createControls(Composite parent, MPart part, ITodoService todoServ
    // extract the id of the todo item
    String string = part.getPersistedState().get(Todo.FIELD_ID);
    Long idOfTodo = Long.valueOf(string);

    // retrieve the todo based on the id stored in the persisted state
    todo = todoService.getTodo(idOfTodo);

    GridLayout gl_parent = new GridLayout(2, false);
    gl_parent.marginRight = 10;
    gl_parent.marginLeft = 10;
    gl_parent.horizontalSpacing = 10;
    gl_parent.marginWidth = 0;
    parent.setLayout(gl_parent);

    Label lblSummary = new Label(parent, SWT.NONE);
    lblSummary.setText("Summary");

    txtSummary = new Text(parent, SWT.BORDER);
    txtSummary.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true, false));

    Label lblDescription = new Label(parent, SWT.NONE);
    lblDescription.setText("Description");

    txtDescription = new Text(parent, SWT.BORDER | SWT.MULTI);
    GridData gd = new GridData(SWT.FILL, SWT.CENTER, true, false);
    gd.heightHint = 122;
    txtDescription.setLayoutData(gd);

    Label lblNewLabel = new Label(parent, SWT.NONE);
    lblNewLabel.setText("Due Date");

    dateDateTime = new DateTime(parent, SWT.BORDER);
    dateDateTime.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, false, false,
        1, 1));
    new Label(parent, SWT.NONE);

    btnDone = new Button(parent, SWT.CHECK);
    btnDone.setText("Done");

    updateUserInterface(todo);
}

@Persist
public void save(MDirtyable dirty, ITodoService model) {
    model.saveTodo(todo);
}

```

```

    dirty.setDirty(false);
}

private void updateUserInterface(Todo todo) {

    // the following check ensures that the user interface is available,
    // it assumes that you have a text widget called "txtSummary"
    if (txtSummary != null && !txtSummary.isDisposed()) {

        // Deregister change listener to the old binding
        IObservableView providers = ctx.getValidationStatusProviders();
        for (Object o : providers) {
            Binding b = (Binding) o;
            b.getTarget().removeChangeListener(listener);
        }

        // Remove bindings
        ctx.dispose();

        IObservableValue oWidgetSummary = WidgetProperties.text(SWT.Modify)
            .observe(txtSummary);
        IObservableValue oTodoSummary = BeanProperties.value(Todo.FIELD_SUMMARY)
            .observe(todo);
        ctx.bindValue(oWidgetSummary, oTodoSummary);

        IObservableValue oWidgetDescription = WidgetProperties.text(SWT.Modify)
            .observe(txtDescription);
        IObservableValue oTodoDescription = BeanProperties.value(Todo.FIELD_DESCR)
            .observe(todo);
        ctx.bindValue(oWidgetDescription, oTodoDescription);

        IObservableValue oWidgetButton = WidgetProperties.selection().observe(btnD
        IObservableValue oTodoDone = BeanProperties.value(Todo.FIELD_DONE).observe
        ctx.bindValue(oWidgetButton, oTodoDone);

        IObservableValue oWidgetSelectionDateTime = WidgetProperties
            .selection().observe(dateTime);
        IObservableValue oTodoDueDate = BeanProperties.value(Todo.FIELD_DUEDATE).o
        ctx.bindValue(oWidgetSelectionDateTime,
            oTodoDueDate);

        // register listener for any changes
        providers = ctx.getValidationStatusProviders();
        for (Object o : providers) {
            Binding b = (Binding) o;
            b.getTarget().addChangeListener(listener);
        }

    }
}

```

```

@Focus
public void onFocus() {
    // the following assumes that you have a Text field
    // called summary
    txtSummary.setFocus();
}
}

```

Tip

You could also adjust the `TodoDetailsPart` code to handle both use cases. The approach to create a new class for the multiple editors has been chosen in this book to "protect" your existing implementation, i.e. if you make an error in this exercise, the rest of your application should still work fine.

This implementation uses the `getPersistedState()` to retrieve the `Todo` id which will be passed to it via the handler.

Change your handler to open a new editor if there is currently not an open editor for the selected `Todo` object. If the editor is already open, set the focus to it.

```

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

import java.util.List;

import javax.inject.Named;

import org.eclipse.e4.core.di.annotations.CanExecute;
import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.core.di.annotations.Optional;
import org.eclipse.e4.ui.model.application.MApplication;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.e4.ui.model.application.ui.basic.MPartStack;
import org.eclipse.e4.ui.services.IServiceConstants;
import org.eclipse.e4.ui.workbench.modeling.EModelService;
import org.eclipse.e4.ui.workbench.modeling.EPartService;

import com.example.e4.rcp.todo.model.Todo;

public class OpenEditorHandler {

    private static final String bundleName = "com.example.e4.rcp.todo";
    private static final String className = "com.example.e4.rcp.todo.parts.EditorP

    @Execute
    public void execute(@Optional @Named(IServiceConstants.ACTIVE_SELECTION) Todo
        MApplication application, EModelService modelService,
        EPartService partService) {

```

```

// sanity check
if (todo == null) {
    return;
}

String id = String.valueOf(todo.getId());

// maybe the editor is already open?
List<MPart> parts = (List<MPart>) partService.getParts();

// if the editor is open show it
for (MPart mPart : parts) {
    String currentId = mPart.getPersistedState().get(Todo.FIELD_ID);
    if (currentId != null && currentId.equals(id)) {
        partService.showPart(mPart, EPartService.PartState.ACTIVATE);
        return;
    }
}

// editor was not open, create it
MPart part = modelService.createModelElement(MPart.class);

// pointing to the contributing class
part.setContributionURI("bundleclass://" + bundleName + "/" + className);
part.getPersistedState().put(Todo.FIELD_ID, id);

// create a nice label for the part header
String header = "ID:" + id + " " + todo.getSummary();
part.setLabel(header);
part.setElementId(id);
part.setCloseable(true);

// add it an existing stack and show it
MPartStack stack =
    (MPartStack) modelService.
        find("com.example.e4.rcp.todo.partstack.bottom", application);
stack.getChildren().add(part);
partService.showPart(part, EPartService.PartState.ACTIVATE);

}

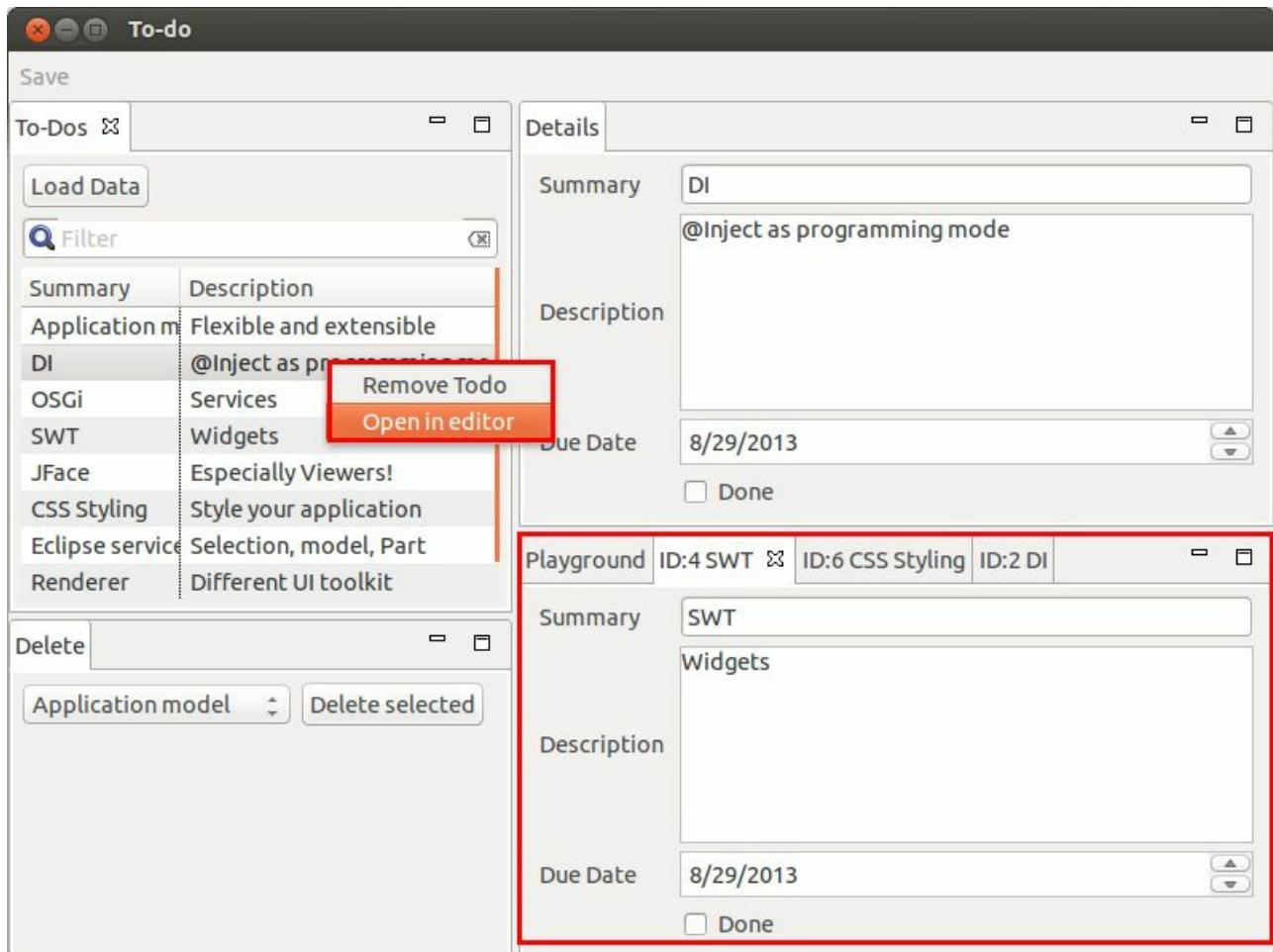
@CanExecute
public boolean canExecute(@Optional @Named(IServiceConstants.ACTIVE_SELECTION)
    if (todo != null) {
        return true;
    }
    return false;
}

}

```

112.5. Validate multiple editor implementation

Ensure that you can open multiple instances of your editor with your context menu and that an existing editor gets focused if you re-select it. The result should look similar to the following screenshot.



Change a `Todo` in the new editor and save. Reload the data in your table via your *Load Data* button and ensure that the data has changed.

Part XXII. Handler and command services

Chapter 113. Command and handler service

113.1. Purpose of the command and handler service

The command and handler services provide the functionality to work with commands and handlers.

Via the handler service you can create, activate and trigger handlers based on commands. The command service allows you to access, and configure commands, i.e. by setting the parameters.

113.2. Access to command and handler service

You can use dependency injection to access the services. The relevant interfaces are `ECommandService` and `EHandlerService`.

113.3. Example for executing a command

The following example shows how to execute a handler for an existing command.

```
Command command = commandService.getCommand("com.example.mycommand");

// check if the command is defined
System.out.println(command.isDefined());

// activate handler, assumption: the AboutHandler() class exists already
handlerService.activateHandler("com.example.mycommand",
    new AboutHandler());

// prepare execution of command
ParameterizedCommand cmd =
    commandService.createCommand("com.example.mycommand", null);

// check if the command can get executed
if (handlerService.canExecute(cmd)) {
    // execute the command
    handlerService.executeHandler(cmd);
}
```

113.4. Example for assigning a handler to a command

The following example shows how to add a new handler to an existing command. It assumes that the `AboutHandler` class already exists.

```
Command command = commandService.getCommand("com.example.mycommand");

// check if the command is defined
System.out.println(command.isDefined());

// activate handler, assumption: the AboutHandler() class exists already
handlerService.activateHandler("com.example.mycommand",
    new AboutHandler());

// prepare execution of command
ParameterizedCommand cmd =
    commandService.createCommand("com.example.mycommand", null);

// check if the command can get executed
if (handlerService.canExecute(cmd)) {
    // execute the command
    handlerService.executeHandler(cmd);
}
```

Chapter 114. Optional exercise: Using handler service

114.1. Delete Todos only via the handler

Change your `TodoDeletionPart` class so that it uses the `EHandlerService` to trigger the existing deletion handler to delete the selected `Todo` from the `ComboViewer`.

Add the following dependencies to your application plug-in:

- `org.eclipse.e4.core.commands`
- `org.eclipse.e4.core.contexts`

114.2. Implementation

Change your `TodoDeletionPart` based on the following example code.

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

import java.util.List;

import javax.annotation.PostConstruct;
import javax.inject.Inject;

import org.eclipse.core.commands.ParameterizedCommand;
import org.eclipse.e4.core.commands.ECommandService;
import org.eclipse.e4.core.commands.EHandlerService;
import org.eclipse.e4.core.contexts.IEclipseContext;
import org.eclipse.e4.ui.di.Focus;
import org.eclipse.e4.ui.workbench.modeling.ESelectionService;
import org.eclipse.jface.viewers.ArrayContentProvider;
import org.eclipse.jface.viewers.ComboViewer;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.LabelProvider;
import org.eclipse.jface.viewers.StructuredSelection;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;

public class TodoDeletionPart {
    @Inject
    private ITodoService todoService;

    @Inject
    ESelectionService selectionService;

    @Inject
    EHandlerService handlerService;
    @Inject
    ECommandService commandService;
    @Inject
    IEclipseContext ctx;

    private ComboViewer viewer;

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

```

viewer = new ComboViewer(parent, SWT.READ_ONLY | SWT.DROP_DOWN);
viewer.setLabelProvider(new LabelProvider() {
    @Override
    public String getText(Object element) {
        Todo todo = (Todo) element;
        return todo.getSummary();
    }
});
viewer.setContentProvider(ArrayContentProvider.getInstance());

List<Todo> todos = todoService.getTodos();
updateViewer(todos);

Button button = new Button(parent, SWT.PUSH);
button.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
        ISelection selection = viewer.getSelection();
        IStructuredSelection sel = (IStructuredSelection) selection;
        if (sel.size() > 0) {
            selectionService.setSelection(sel.getFirstElement());

            // ensure that "com.example.e4.rcp.todo.command.remove" is
            // the command ID which deletes Todo objects
            ParameterizedCommand cmd =
                commandService.
                createCommand("com.example.e4.rcp.todo.command.remove",
                             null);

            handlerService.executeHandler(cmd, ctx);

            // update the viewer and remove the selection
            List<Todo> todos = todoService.getTodos();
            updateViewer(todos);
            selectionService.setSelection(null);
        }
    }
});
button.setText("Delete selected");
}

private void updateViewer(List<Todo> todos) {
    viewer.setInput(todos);
    if (todos.size() > 0) {
        viewer.setSelection(new StructuredSelection(todos.get(0)));
    }
}

@Focus
public void focus() {
    viewer.getCombo().setFocus();
}
}

```

Note

The handler and command service have not been released as official API. See [Section 6.4, “Eclipse API and internal API”](#).

Part XXIII. Asynchronous processing

Chapter 115. Threading in Eclipse

115.1. Concurrency

What is concurrency?

Concurrency is the ability to run several programs or several parts of a program in parallel. If a time consuming task can be performed asynchronously or in parallel, this improves the throughput and the interactivity of the program.

A modern computer has several CPU's or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application.

Process vs. threads

A *process* runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.

A *thread* is a so called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data.

A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

115.2. Main thread

An Eclipse RCP application runs in one process but can create multiple threads.

By default the Eclipse framework uses a single thread to run all the code instructions. This thread runs the event loop for the application and is the only thread that is allowed to interact with the user interface (UI). It is called the *main thread*. Sometimes it is also called the *UI thread*, but this is a misnomer as it handles all events not only the ui events.

If another thread tries to update the UI, the Eclipse framework throws an `SWTException` exception.

```
org.eclipse.swt.SWTException: Invalid thread access
```

All events in the user interface are executed one after another. If you perform a long running operation in the main thread, the application does not respond to user interaction during the execution time of this operation.

Blocking the user interaction is considered a bad practice. Therefore it is important to perform all long running operations in a separate thread. Long running operations are, for example, network or file access.

As only the main thread is allowed to modify the user interface, the Eclipse framework provides ways for a thread to synchronize itself with the main thread. It also provides the Eclipse Jobs framework which allows you to run operations in the background and providing feedback of the job status to the Eclipse platform.

115.3. Using dependency injection and UISynchronize

The `org.eclipse.e4.ui.di` plug-in contains the `UISynchronize` class. An instance of this class can be injected into an Eclipse application via dependency injection.

`UISynchronize` provides the `syncExec()` and `asyncExec()` methods to synchronize with the main thread.

115.4. Eclipse Jobs API

The Eclipse Jobs API provides support for running background processes and providing feedback about the progress of the Job.

The important parts of the Job API are:

- IJobManager - schedules jobs
- Job - the individual task to perform
- IProgressMonitor - interface to communicate information about the status of your Job.

The creation and scheduling of a Job is demonstrated in the following code snippet.

```
// get UISynchronize injected as field
@Inject UISynchronize sync;

// more code

Job job = new Job("My Job") {
    @Override
    protected IStatus run(IProgressMonitor monitor) {
        // do something long running
        //...

        // If you want to update the UI
        sync.asyncExec(new Runnable() {
            @Override
            public void run() {
                // do something in the user interface
                // e.g. set a text field
            }
        });
        return Status.OK_STATUS;
    }
};

// Start the Job
job.schedule();
```

If you want to update the user interface from a Job, you need to synchronize the corresponding action with the user interface similar to the direct usage of threads.

115.5. Priorities of Jobs

You can set the `Job` priority via the `job.setPriority()` method. The `Job` class contains predefined priorities, e.g. `Job.SHORT`, `Job.LONG`, `Job.BUILD` and `Job.DECORATE`.

The Eclipse job scheduler will use these priorities to determine in which order the `Jobs` are scheduled. For example, jobs with the priority `Job.SHORT` are scheduled before jobs with the `Job.LONG` priority . Check the JavaDoc of the `Job` class for details.

115.6. Blocking the UI and providing feedback

Sometimes you simply want to give the user the feedback that something is running without using threads.

The easiest way to provide feedback is to change the cursor via the `BusyIndicator.showWhile()` method call.

```
// Show a busy indicator while the runnable is executed  
BusyIndicator.showWhile(display, runnable);
```

If this code is executed, the cursor will change to a busy indicator until the `Runnable` is done.

Chapter 116. Progress reporting

116.1. IProgressMonitor

The `IProgressMonitor` object can be used to report progress. Use the `beginTask()` method to specify the total units of work. Use the `worked()` method to report that a certain number of units of work have been finished. The `worked()` method is called with the units of work done since the last call, it is not called with the total amount of finished work. The usage of `IProgressMonitor` in a `Job` is demonstrated in the following code.

```
Job job = new Job("My Job") {
    @Override
    protected IStatus run(IProgressMonitor monitor) {
        // set total number of work units
        monitor.beginTask("Doing something time consuming here", 100);
        for (int i = 0; i < 5; i++) {
            try {
                // sleep a second
                TimeUnit.SECONDS.sleep(1);

                monitor.subTask("I'm doing something here " + i);

                // report that 20 additional units are done
                monitor.worked(20);
            } catch (InterruptedException e1) {
                e1.printStackTrace();
                return Status.CANCEL_STATUS;
            }
        }
        return Status.OK_STATUS;
    }
};

job.schedule();
```

116.2. Reporting progress in Eclipse RCP applications

In Eclipse applications you can report progress by implementing the `IProgressMonitor` interface.

You can, for example, add a tool control to a toolbar in your application model. This tool control can implement the `IProgressMonitor` interface to show the progress.

This is demonstrated in the following example.

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

import java.util.Objects;

import javax.annotation.PostConstruct;
import javax.inject.Inject;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.NullProgressMonitor;
import org.eclipse.core.runtime.jobs.IJobChangeEvent;
import org.eclipse.core.runtime.jobs.Job;
import org.eclipse.core.runtime.jobs.JobChangeAdapter;
import org.eclipse.core.runtime.jobs.ProgressProvider;
import org.eclipse.e4.ui.di.UISynchronize;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.ProgressBar;

public class ProgressMonitorControl {

    private final UISynchronize sync;

    private ProgressBar progressBar;
    private GobalProgressMonitor monitor;

    @Inject
    public ProgressMonitorControl(UISynchronize sync) {
        this.sync = Objects.requireNonNull(sync);
    }

    @PostConstruct
    public void createControls(Composite parent) {
        progressBar = new ProgressBar(parent, SWT.SMOOTH);
        progressBar.setBounds(100, 10, 200, 20);

        monitor = new GobalProgressMonitor();

        Job.getJobManager().setProgressProvider(new ProgressProvider() {
```

```

@Override
public IProgressMonitor createMonitor(Job job) {
    return monitor.addJob(job);
}
});

}

private final class GobalProgressMonitor extends NullProgressMonitor {

    // thread-Safe via thread confinement of the UI-Thread
    // (means access only via UI-Thread)
    private long runningTasks = 0L;

    @Override
    public void beginTask(final String name, final int totalWork) {
        sync.syncExec(new Runnable() {

            @Override
            public void run() {
                if(runningTasks <= 0) {
                    // --- no task is running at the moment ---
                    progressBar.setSelection(0);
                    progressBar.setMaximum(totalWork);

                } else {
                    // --- other tasks are running ---
                    progressBar.setMaximum(progressBar.getMaximum() + totalWork);
                }

                runningTasks++;
                progressBar.setToolTipText("Currently running: " + runningTasks +
                    "\nLast task: " + name);
            }
        });
    }

    @Override
    public void worked(final int work) {
        sync.syncExec(new Runnable() {

            @Override
            public void run() {
                progressBar.setSelection(progressBar.getSelection() + work);
            }
        });
    }

    public IProgressMonitor addJob(Job job) {
        if(job != null){
            job.addJobChangeListener(new JobChangeAdapter() {
                @Override
                public void done(IJobChangeEvent event) {
                    sync.syncExec(new Runnable() {

```

```
        @Override
        public void run() {
            runningTasks--;
            if (runningTasks > 0) {
                // --- some tasks are still running ---
                progressBar.setToolTipText("Currently running: " + runningTask
                } else {
                    // --- all tasks are done (a reset of selection could also be
                    progressBar.setToolTipText("No background progress running.");
                }
            }
        });

        // clean-up
        event.getJob().removeJobChangeListener(this);
    }
});
```

This new element can be accessed via the model service and used as an `IProgressMonitor` for the job.

```
    Job job = new Job("My Job") {  
        // code as before  
    };  
    job.schedule();
```

Tip

A more advanced implementation could, for example, implement a progress monitoring OSGi Service and report progress to the user interface via the event service.

Chapter 117. Exercise: Using asynchronous processing

117.1. Simulate delayed access

Change your `getTodos()` method in the `MyTodoServiceImpl` class to the following example code. This code simulates a slow server access with the `TimeUnit` class.

```
// always return a new copy of the data
@Override
public List<Todo> getTodos() {
    // Simulate Server access delay
    try {
        TimeUnit.SECONDS.sleep(5);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    List<Todo> list = new ArrayList<Todo>();
    for (Todo todo : model) {
        list.add(todo.copy());
    }
    return list;
}
```

Start your application. What happens if you press the `Button` to fill the JFace table?

117.2. Use asynchronous processing

You have a *Load Data* button in your `TodoOverviewPart`. Write a custom `Job` implementation to load the data in the background whenever this button is pressed. Once the data is loaded, update the table with the new data. Make sure that this table update happens in the main interface thread.

The following example code can help.

```
// get UISynchronize injected as field
@Inject UISynchronize sync;

// more code.....

// replace the existing button implementation with
// the following code

btnLoadData = new Button(parent, SWT.PUSH);
btnLoadData.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
        Job job = new Job("loading") {
            @Override
            protected IStatus run(IProgressMonitor monitor) {
                // assumes you get todoService (ITodoService) injected
                final List<Todo> list = todoService.getTodos();
                sync.asyncExec(new Runnable() {
                    @Override
                    public void run() {
                        updateViewer(list);
                    }
                });
                return Status.OK_STATUS;
            }
        };
        job.schedule();
    }
});

btnLoadData.setText("Load Data");

public void updateViewer(List<Todo> list) {
    if (viewer != null) {
        // if you use databinding for the viewer
        writableList.clear();
        writableList.addAll(list);
        // in case you skipped the data binding exercise
    }
}
```

```
// use the following  
// viewer.setInput(list);  
}  
}
```

117.3. Validating

Note

If you start your application, it takes longer to start if you still read the data somewhere synchronously during startup. This is done for example if you implemented [Chapter 71, Optional exercise: Using ComboViewer](#). If you want to avoid this you need to perform this reading also asynchronously. We leave this change as an optional exercise to the reader.

Use the wizard to create a new `Todo` object. Press the *Load Data* button to update the table. The application should be usable and the table should update after the delay defined in the `ITodoService` implementation class.

117.4. Remove delay

After finishing this exercise and validating that everything works correctly, change the delay to one second so avoid delays during future exercises.

Part XXIV. Event service for message communication

Chapter 118. Eclipse event notifications

118.1. Event based communication

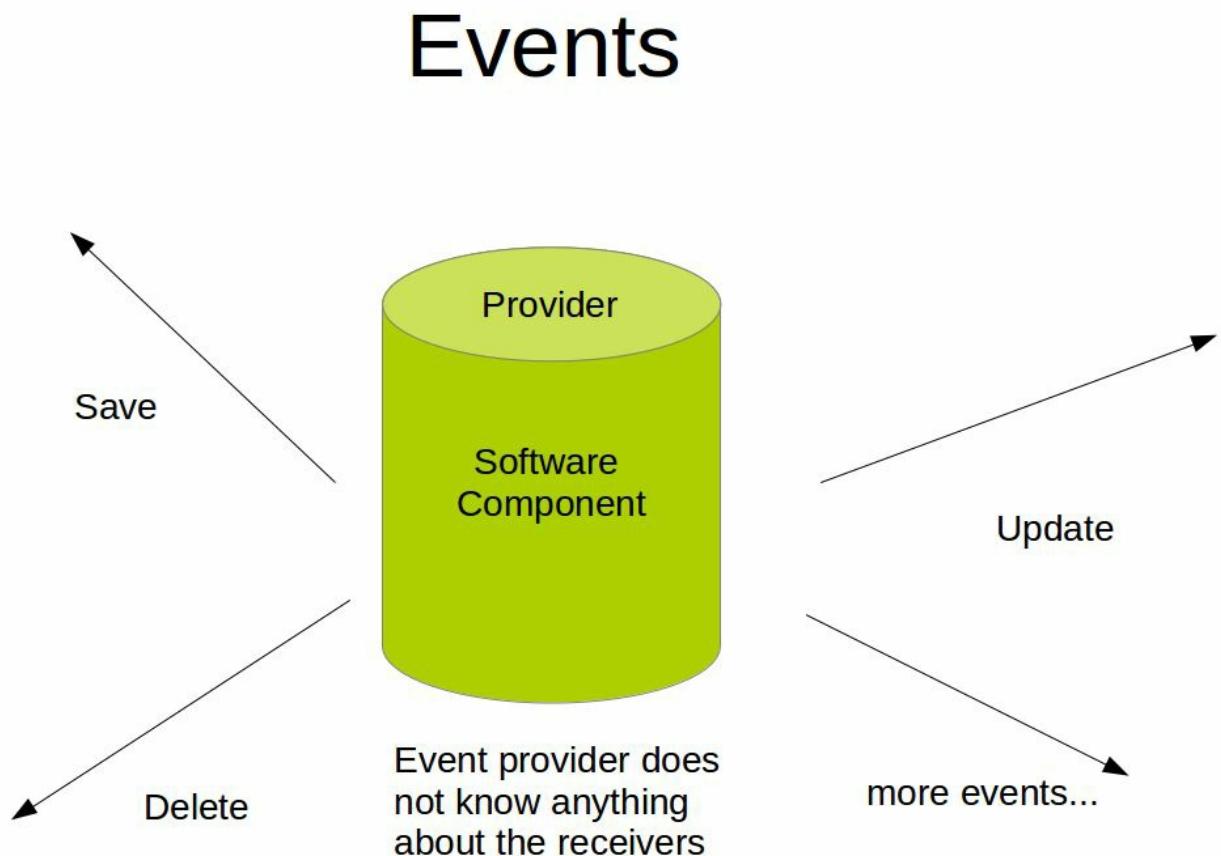
In your application you frequently have the need to communicate between application components. This communication should be loosely coupled, to remove dependency between the components and to increase reuse of these components.

This can be achieved by the subscriber/ publisher model implemented as an event system. Software components can register for specified events and other components can send events out. The event system ensures that all registered components receive the event they registered for. .

118.2. The event bus of Eclipse

For this purpose the Eclipse platform provides a global event based communication system, called the *event bus*.

Any software component which has access to the event system can send out arbitrary events as depicted in the following graphic.



The Eclipse platform will make sure that registered components receive the messages. The Eclipse platform uses this event system for the internal communication.

118.3. Event service

The Eclipse framework provides the event service for event communication. This service can be accessed via dependency injection based on the `IEventBroker` interface. This communication service can also be used to communicate between your own application components.

The Eclipse event service is based on the OSGi *EventAdmin* service but has been wrapped with a simpler API.

118.4. Required plug-ins to use the event service

The following plug-ins are required to use the event service functionality:

- org.eclipse.e4.core.services
- org.eclipse.osgi.services

118.5. Sending and receiving events

Sending

The event service can be injected via dependency injection.

```
@Inject  
private IEventBroker eventBroker;
```

The following code examples assume that you have a class named *MyEventConstants* defined which contains a static final field (constant) for the `TOPIC_TODO_NEW` string.

The event service collects all events and sends them to the registered components. This can be done asynchronously or synchronously.

```
@Inject IEventBroker broker;  
  
...  
// asynchronously  
broker.post(MyEventConstants.TOPIC_TODO_NEW,  
    new Event(MyEventConstants.TOPIC_TODO_NEW,  
        new HashMap<String, String>());  
  
@Inject IEventBroker broker;  
...  
// synchronously sending a todo  
// the calling code is blocked until delivery  
  
broker.send(MyEventConstants.TOPIC_TODO_NEW,  
    new Event(MyEventConstants.TOPIC_TODO_NEW,  
        new HashMap<String, String>())
```

You can now send arbitrary Java objects or primitives through the event bus.

Annotations for receiving events

You can use dependency injection to register and respond to events. If dependency injection is used, the Eclipse framework automatically removes all event subscriptions when the model class is disposed.

The `@EventTopic` and `@UIEventTopic` annotations tag methods and fields that should be notified on event changes. The `@UIEventTopic` ensures the event notification is performed in the user interface thread.

```

import java.util.Map;

// MyEventConstants.TOPIC_TODO_UPDATE is
// a String constant

@Inject
@Optional
private void subscribeTopicTodoUpdated
    (@UIEventTopic(MyEventConstants.TOPIC_TODO_UPDATE)
     Map data) {
    if (viewer!=null) {
        // this example assumes that you do not use data binding
        viewer.setInput(todoService.getTodos());
    }
}

```

Registering listeners for events

An object can also register an instance of the `EventHandler` directly with the `IEventBroker` interface via the `subscribe()` method.

Tip

Using dependency injection for subscribing should be preferred compared to the direct subscription as this way the framework handles the listener registration and de-registration automatically for you.

Which objects should be send out?

The event system allows sending and receiving objects of an arbitrary type. In Eclipse applications it is good practice to send out events with a `java.util.Map` type because a similar type is also used by the Eclipse framework for sending out events. A `Map` class can be configured with properties to contain more data about the event.

Subscribing to sub-topics

You can subscribe to specific topics or use wildcards to subscribe to all sub-events. Sub-events are separated by `/`. The following example code defines several topics and also the `TOPIC_TODO_ALLTOPICS` constant which can be used to register for all sub-events.

```

package com.example.e4.rcp.todo.events;
/**
 *

```

```

* @noimplement This interface is not intended to be implemented by clients.
*
* Only used for constant definition
*/

```

```

public interface MyEventConstants {

    // topic identifier for all todo topics
    String TOPIC_TODO = "TOPIC_TODOS";

    // this key can only be used for event registration, you cannot
    // send out generic events
    String TOPIC_TODO_ALLTOPICS = "TOPIC_TODOS/*";

    String TOPIC_TODOS_CHANGED = "TOPIC_TODOS/CHANGED";

    String TOPIC_TODO_NEW = "TOPIC_TODOS/TODO/NEW";

    String TOPIC_TODO_DELETE = "TOPIC_TODOS/TODO/DELETED";

    String TOPIC_TODO_UPDATE = "TOPIC_TODOS/TODO/UPDATED";
}

```

Eclipse framework events

It is also possible to register for Eclipse framework events. The `UIEvents` class from the `org.eclipse.e4.ui.workbench` contains all framework events. This class also contains information about the purpose of events in its Javadoc. The Eclipse framework sends out objects of type `org.osgi.service.event.Event`.

For example the `UIEvents.UILifecycle.APP_STARTUP_COMPLETE` event is triggered once the application has started.

Another example is the `UIEvents.UILifecycle.ACTIVATE` event which is triggered if a part is activated. Registering for this event is demonstrated in the following code snippet.

```

@Inject
@Optional
public void partActivation(@UIEventTopic(UIEvents.UILifecycle.ACTIVATE)
    Event event) {
    // do something
    System.out.println("Got Part");
}

```

118.6. Usage of the event system

The `IEventBroker` is a global event bus and is unaware of the `IEclipseContext` hierarchy. The `IEventBroker` service supports sending event information without knowing who is receiving it. Interested classes can register for events without knowing who is going to provide them. This is known as the whiteboard pattern and this pattern supports the creation of very loosely coupled application components.

The disadvantage is that it is a global bus, i.e. there is no scoping of the events. Publishers have to ensure they provide enough information in the topic and the send object to allow subscribers to discriminate and decide that the particular event is applicable to a subscriber.

118.7. Asynchronous processing and the event bus

Your threads can use the `IEventBroker` to send event data. Every listener will be automatically called and if a method is annotated with the `UIEventTopic` annotation, it will be called in the main thread.

```
private static final String UPDATE ="update";

// get the IEventBroker injected
@Inject
IEventBroker broker;

// somewhere in you code you do something
// performance intensive

button.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
        Runnable runnable = new Runnable() {
            public void run() {
                for (int i = 0; i < 10; i++) {
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    // send out an event to update
                    // the UI
                    broker.send(UPDATE, i);
                }
            }
        };
        new Thread(runnable).start();
    }
});

// more code
// ....

// get notified and sync automatically
// with the UI thread

@Inject @Optional
public void getEvent(@UIEventTopic(UPDATE) int i) {
    // text1 is a SWT Text field
    text1.setText(String.valueOf(i));
    System.out.println(i);
}
```

Chapter 119. Exercise: Event notifications

119.1. Creating a plug-in for event constants

Create a new simple plug-in called `com.example.e4.rcp.todo.events`.

Define the following `MyEventConstants` interface in this new plug-in. This interface stores the constants for the event topics.

```
package com.example.e4.rcp.todo.events;
/**
 *
 * @noimplement This interface is not intended to be implemented by clients.
 *
 * Only used for constant definition
 */

public interface MyEventConstants {

    // topic identifier for all todo topics
    String TOPIC_TODO = "TOPIC_TODO";

    // this key can only be used for event registration, you cannot
    // send out generic events
    String TOPIC_TODO_ALLTOPICS = "TOPIC_TODO/*";

    String TOPIC_TODO_CHANGED = "TOPIC_TODO/CHANGED";

    String TOPIC_TODO_NEW = "TOPIC_TODO/TODO/NEW";

    String TOPIC_TODO_DELETE = "TOPIC_TODO/TODO/DELETED";

    String TOPIC_TODO_UPDATE = "TOPIC_TODO/TODO/UPDATED";
}
```

Export the `com.example.e4.rcp.todo.events` package in the `MANIFEST.MF` file (via the *Runtime* tab) in your new plug-in.

119.2. Add the new plug-in to your product

Add the *event* plug-in to your feature so that it is available in your product configuration file.

119.3. Enter the plug-in dependencies

Add `com.example.e4.rcp.todo.events` as a dependency to your `com.example.e4.rcp.todo` plug-in via its `MANIFEST.MF` file.

Start your application to validate that you did everything correctly.

Warning

Remember to start your application via the product to update the runtime configuration. If it fails ensure that you have added the `*.todo.events` plug-in to the feature.

119.4. Send out notifications

Ensure that the following two plug-ins are included as dependency into your `com.example.e4.rcp.todo` plug-in.

- `org.eclipse.e4.core.services`
- `org.eclipse.osgi.services`

Update your `NewTodoHandler` and your `SaveAllHandler` classes to send out an event to communicate the data change. The following example code demonstrates that for the `NewTodoHandler` class.

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

import java.util.HashMap;
import java.util.Map;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.core.services.events.IEventBroker;
import org.eclipse.jface.wizard.WizardDialog;
import org.eclipse.swt.widgets.Shell;

import com.example.e4.rcp.todo.events.MyEventConstants;
import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;
import com.example.e4.rcp.todo.wizards.TodoWizard;

public class NewTodoHandler {
    @Execute
    public void execute(Shell shell, ITodoService todoService, IEventBroker broker
        // use -1 to indicate a not existing id
        Todo todo = new Todo(-1);
        WizardDialog dialog = new WizardDialog(shell, new TodoWizard(todo));
        if (dialog.open() == WizardDialog.OK) {
            todoService.saveTodo(todo);
            // asynchronously
            String todoId = String.valueOf(todo.getId());
            broker.post(MyEventConstants.TOPIC_TODO_NEW,
                        createEventData(MyEventConstants.TOPIC_TODO_NEW, todoId));
        }
    }

    private Map<String, String> createEventData(String topic, String todoId) {
        Map<String, String> map = new HashMap<String, String>();
        // in case the receiver wants to check the topic
        map.put(MyEventConstants.TOPIC_TODO, topic);
        // which todo has changed
        map.put(Todo.FIELD_ID, todoId);
        return map;
    }
}
```

}

119.5. Receive updates in your parts

Register for this event in all relevant parts of your application. If the event is received, update your user interface. The following code demonstrates that for your `TodoOverviewPart` class.

```
import java.util.Map;
// more code
// assumes that ITodoService is injected somewhere as todoService

@Inject
@Optional
private void subscribeTopicTodoAllTopics
    (@UIEventTopic(MyEventConstants.TOPIC_TODO_ALLTOPICS) Map<String, String> even
     if (viewer != null) {
        // code if you use databinding for your viewer
        writableList.clear();
        writableList.addAll(todoService.getTodos());
        // if you do not use databinding, use the following snippet:
        // viewer.setInput(todoService.getTodos());
    }
}
```

119.6. Validating

Create a new `Todo` object via your wizard. The table in `TodoOverviewPart` should get updated automatically, e.g., the new entry should be visible in the table after you finish the wizard.

119.7. Review the implementation

The current implementation works but is not perfect. Each user interface component, which modifies `Todo` objects, needs to send out events.

It would be preferable if the `ITodoService` implementation could send out these events directly.

In the [Part XXVI, “Eclipse context functions”](#) chapter we improve our current solution and move the event sending to the implementation class of the `ITodoService` service.

Part XXV. Extending and modifying the Eclipse context

Chapter 120. Accessing and extending the Eclipse context

120.1. Accessing the context

You can place objects directly in the `IEclipseContext` hierarchy to make them available to other model objects.

To access an existing context you can use dependency injection if the relevant object is managed by the Eclipse runtime. This is the case for all model objects. The following code demonstrates how to get access to the active `IEclipseContext`, in which the handler is called.

```
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 the active local context injected into
        // this handler
        // ...
    }
}
```

If a model object implements `MContext`, you can use dependency injection to get the model object injected and call the `getContext()` method to access its context. For example, `MPart`, `MWindow`, `MApplication` and `MPerspective` extend `MContext`.

The following code demonstrates how to get the `MApplication` injected and how to access its `IEclipseContext`.

```
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;

public class TodoDetailsPart {
```

```
@PostConstruct
public void createControls(Composite parent,
    MApplication application) {

    // getting the IEclipseContext of the application
    // via the MApplication object
    IEclipseContext context = application.getContext();

    // add or access objects to and from the application context
    // ...
}

}
```

120.2. Objects and context variables

You can add key / value pairs directly to the `IEclipseContext`.

Adding objects to a context can be done via the `set()` method of the `IEclipseContext` interface. The following example creates a new context via the `EclipseContextFactory.create()` factory method call and adds some objects to it. Via the `setParent()` method call, the new context is connected to the context hierarchy.

```
@Inject
public void addingContext(IEclipseContext context) {

    // create a new IEclipseContext instance
    IEclipseContext myContext = EclipseContextFactory.create();

    // add objects to context
    myContext.set("mykey1", "Hello1");
    myContext.set("mykey2", "Hello2");

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

    // alternatively you can create a new
    // context which has a parent/child
    // relationship via the
    // context.createChild() method call

}
```

Such a context can be used to instantiate an object via the Eclipse framework. See [Section 121.2, “Using dependency injection to create objects”](#) for a detailed description.

A *context variable* is a key which is 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");
```

}

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()` method searches up the chain to find the `IEclipseContext` defining the variable. If no entry is found in the context hierarchy, the value will be set in the `IEclipseContext` in which the call started. 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/value pairs and *Context variables* at different levels of the context hierarchy to supply different objects in your application.

120.3. Replacing existing objects in the IEclipseContext

Instead of adding new objects to the `IEclipseContext` hierarchy, you can also override existing objects by using the same key.

You can change behavior of your application by overriding certain entries in the context. For example, you can modify the context of the `MWindow` model element. Its `IEclipseContext` 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 `MWindow` model element. The `IWindowCloseHandler` object is responsible for the behavior once the `MWindow` model element is closed. The default `IWindowCloseHandler` prompts the user if he wants to save dirty parts (editors with changed content). You can change this default implementation by replacing the object in the context. 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);
}
```

Tip

You could use this example in your life cycle handler and subscribe to the `UIEvents.UILifeCycle.APP_STARTUP_COMPLETE` event. In the event handler you would replace the `IwindowCloseHandler` handler in the context.

120.4. Accessing the IEclipseContext hierarchy from OSGi services

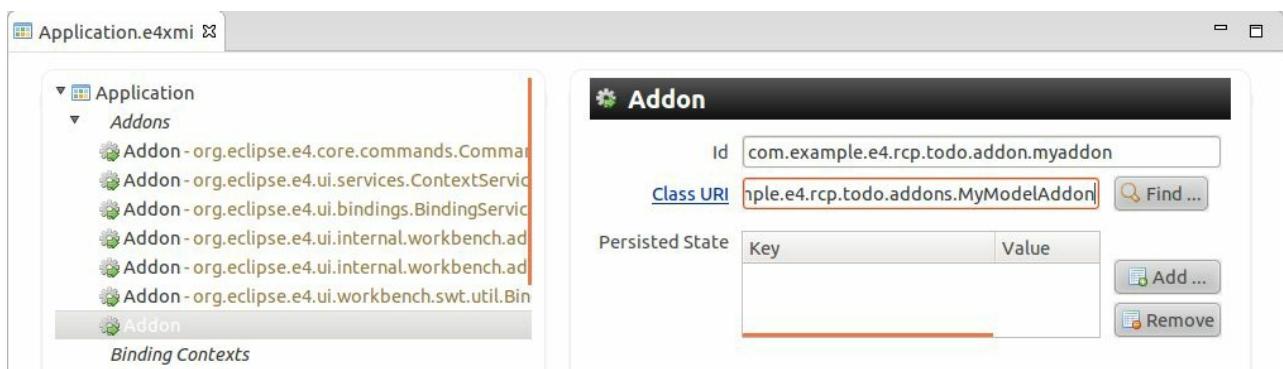
OSGi services are not directly part of the `IEclipseContext` hierarchy and are created by the OSGi runtime. The OSGi runtime does not support dependency injection based on the `@Inject` annotation.

The Eclipse framework registers the implementation of the `MApplication` interface also as an OSGi service. This allows OSGi services to use the OSGi API to access the `MApplication` and its context via the `getContext()` method. As the `EModelService` is part of the `MApplication` context you can search for other context elements via it.

120.5. Model add-ons

To participate in dependency injection with your custom Java objects you can add them as model add-ons to the application model. The classes referred to by the model add-ons can access and modify the `IEclipseContext` or interact with other services, e.g., the event system.

The following screenshot shows a custom model add-on registered in the application model.



The following code shows an example implementation for the model addon class. This addon places an object into the `IEclipseContext`

```
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) {
        // injected IEclipseContext comes from the application
        context.set("test1", "Hello");
    }
}
```

120.6. RunAndTrack

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

A `RunAndTrack` object is basically a `Runnable` which has access to the context.

If the context changes, the `RunAndTrack` is called by the Eclipse framework. The runnable does not need to be explicitly unregistered from this context when it is no longer interested in tracking changes. If the `RunAndTrack` is invoked by the Eclipse platform and it returns `false` from its `RunAndTrack.changed()` method, it is automatically unregistered from change tracking on this context.

Such a `RunAndTrack` object allows a client to keep some external state synchronized with one or more values in this context.

Chapter 121. Using dependency injection for your Java objects

121.1. Creating and injecting custom objects

Using dependency injection for your custom objects has two flavors.

1. You want to create objects which declare their dependencies with `@Inject` based on a `I EclipseContext` context. See [Section 121.2, “Using dependency injection to create objects”](#) for details.
2. You want the Eclipse dependency container to create your custom objects automatically on demand and then get them injected into your model objects. See [Section 121.3, “Create your custom objects automatically with `@Creatable`”](#) and [Section 121.4, “Create automatically objects in the application context with `@Singleton`”](#).

121.2. Using dependency injection to create objects

Using dependency injection is not limited to the objects created by the Eclipse runtime. You can use the same approach to create an instance of a given class based on a given `IEclipseContext`. The given class can contain `@Inject` annotations. For this you use the `ContextInjectionFactory` class as demonstrated in the following code example.

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

The `ContextInjectionFactory.make()` method creates the object. You can also put it into the `IEclipseContext` hierarchy after the creation. If you place it into the `IEclipseContext` of the application, the created object is globally available.

For this you can either use an existing `IEclipseContext` or create a new `IEclipseContext`. The new context object can be connected to the context hierarchy. Using a new context might be preferable to avoid collision of keys and to isolate your changes in a local context. Call the `dispose` method on your local context, if the object is not needed anymore.

The following code demonstrates how to create a new `IEclipseContext` object and to place values into it. This context can be used to create a new object.

```
IEclipseContext context = EclipseContextFactory.create();  
  
// add your Java objects to the context  
context.set(MyDataObject.class.getName(), data);  
context.set(MoreStuff.class, moreData);  
  
// dispose the context if you are done with it  
context.dispose();
```

The next code example demonstrates how you can connect your new `IEclipseContext` object with an existing context hierarchy. The factory searches the hierarchy upwards to find values, requested by the class which is instantiated.

```
@Inject  
public void createObjectInPart(IEclipseContext ctx) {  
    // create a new local_ context  
    IEclipseContext localCtx =  
        EclipseContextFactory.create();
```

```
localCtx.set(String.class, "Hello");

// connect new local context with context hierarchy
localCtx.setParent(ctx);

// create object of type MyJavaObject via DI
// uses the localCtx and searches upwards for required objects
MyJavaObject o = ContextInjectionFactory.make(MyJavaObject.class,
    localCtx);

//TODO do something with the "o" object
}
```

The `ContextInjectionFactory.inject(Object, IEclipseContext)` method allows you to perform injection on an existing object. For example, if you created the object with the `new()` operator, you can still run dependency injection on it.

121.3. Create your custom objects automatically with @Creatable

If you want the Eclipse framework to create your custom 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 automatically-generated instance is not stored in the context.

If you have a non default-constructor, you must use the `@Inject` annotation on the constructor to indicate that Eclipse should try to run dependency injection on it.

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

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

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

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

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

import javax.inject.Inject;

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

import com.example.e4.rcp.todo.model.ITodoService;

@Creatable
public class YourObject {
    // constructor
    @Inject
    public YourObject(Dependent depend, ITodoService service) {
        // placeholder
    }
}
```

As the Eclipse framework is allowed to create instances of the `Dependent` and the `YourObject` class, it can create them if an instance is requested via dependency injection. In this example the arguments of the constructors can be satisfied. If no fitting constructor is found, the Eclipse framework throws an exception.

Assuming that you have defined the `ITodoService` OSGi service in your application, you can get an instance of your `YourObject` class injected into a part. The following example code demonstrates that.

```
// add this for example to your playground part
@Inject
public void setYourObject(YourObject object) {
    System.out.println(object);
}
```

121.4. Create automatically objects in the application context with @Singleton

If the object should be created in the application context, use the `@Singleton` annotation in addition to the `@Creatable` annotation. This ensures that only one instance of the object is created in your application.

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

import javax.inject.Inject;

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

import com.example.e4.rcp.todo.model.ITodoService;

@Creatable
@Singleton
public class YourObject {
    // constructor
    @Inject
    public YourObject(Dependent depend, ITodoService service) {
        // placeholder
    }
}
```

Chapter 122. Exercise: Dependency injection for your objects

Note

This exercise is optional.

122.1. Target of this exercise

The following demonstrates the usage of dependency injection framework of Eclipse for creating objects.

In this exercise you create an instance of the `TodoWizard` via the `ContextInjectionFactory.make()` method.

122.2. Prepare the wizard classes for dependency injection

Add the `@Inject` annotation to the constructor of your `TodoWizardPage1` class.

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

import javax.inject.Inject;

import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;

import com.example.e4.rcp.todo.model.Todo;
import com.example.e4.rcp.todo.parts.TodoDetailsPart;

public class TodoWizardPage1 extends WizardPage {

    private Todo todo;

    @Inject
    public TodoWizardPage1(Todo todo) {
        super("page1");
        this.todo = todo;
        setTitle("New Todo");
        setDescription("Enter the todo data");
    }

    public void createControl(Composite parent) {
        Composite container = new Composite(parent, SWT.NONE);
        // we could also create this class via DI but
        // in this example we stay with the next operator
        TodoDetailsPart part = new TodoDetailsPart();
        part.createControls(container);
        part.setTodo(todo);

        setControl(container);
    }
}
```

Add the `@Inject` annotation to the constructor of your wizard and to the two fields for the wizard pages.

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

import javax.inject.Inject;

import org.eclipse.jface.wizard.IWizardPage;
import org.eclipse.jface.wizard.Wizard;
```

```
import com.example.e4.rcp.todo.i18n.Messages;
import com.example.e4.rcp.todo.model.Todo;

public class TodoWizard extends Wizard {

    boolean finish = false;

    @Inject
    TodoWizardPage1 page1;
    @Inject
    TodoWizardPage2 page2;

    @Inject
    public TodoWizard() {
        setWindowTitle("New Wizard");
    }

    @Override
    public void addPages() {

        addPage(page1);
        addPage(page2);
    }

    @Override
    public boolean performFinish() {
        return true;
    }

    @Override
    public boolean canFinish() {
        return finish;
    }

    @Override
    public IWizardPage getNextPage(IWizardPage page) {
        return super.getNextPage(page);
    }

}
```

122.3. Create the wizard via dependency injection

In your handle which opens the wizard get the `IEclipseContext` injected and create a new `IEclipseContext` with the `createChild()` method.

Create a new Todo via the `new()` Operator. Add the new `Todo` to your local context under the `Todo.class` key.

Also create the two wizard pages and add them to the created `IEclipseContext`. Afterwards created the wizard via the `ContextInjectionFactory` class as demonstrated by the following code.

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

import java.util.Date;

import org.eclipse.e4.core.contexts.ContextInjectionFactory;
import org.eclipse.e4.core.contexts.IEclipseContext;
import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.jface.wizard.WizardDialog;
import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.widgets.Shell;

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;
import com.example.e4.rcp.todo.wizards.TodoWizard;
import com.example.e4.rcp.todo.wizards.TodoWizardPage1;
import com.example.e4.rcp.todo.wizards.TodoWizardPage2;

public class NewTodoHandler {
    @Execute
    public void execute(Shell shell, ITodoService model, IEclipseContext ctx) {
        // create new context
        IEclipseContext wizardCtx = ctx.createChild();

        // create todo and store in context
        // use -1 to indicate a not existing id
        Todo todo = new Todo(-1);
        todo.setDueDate(new Date());
        wizardCtx.set(Todo.class, todo);

        // create WizardPages via CIF
        TodoWizardPage1 page1 = ContextInjectionFactory.make(TodoWizardPage1.class,
            wizardCtx.set(TodoWizardPage1.class, page1));
        // no context needed for the creation
        TodoWizardPage2 page2 = ContextInjectionFactory.make(TodoWizardPage2.class,
            wizardCtx.set(TodoWizardPage2.class, page2));

        TodoWizard wizard = ContextInjectionFactory.make(TodoWizard.class, wizardCtx
```

```
WizardDialog dialog = new WizardDialog(shell, wizard);
if (dialog.open() == WizardDialog.OK) {
    model.saveTodo(todo);
}
}
```

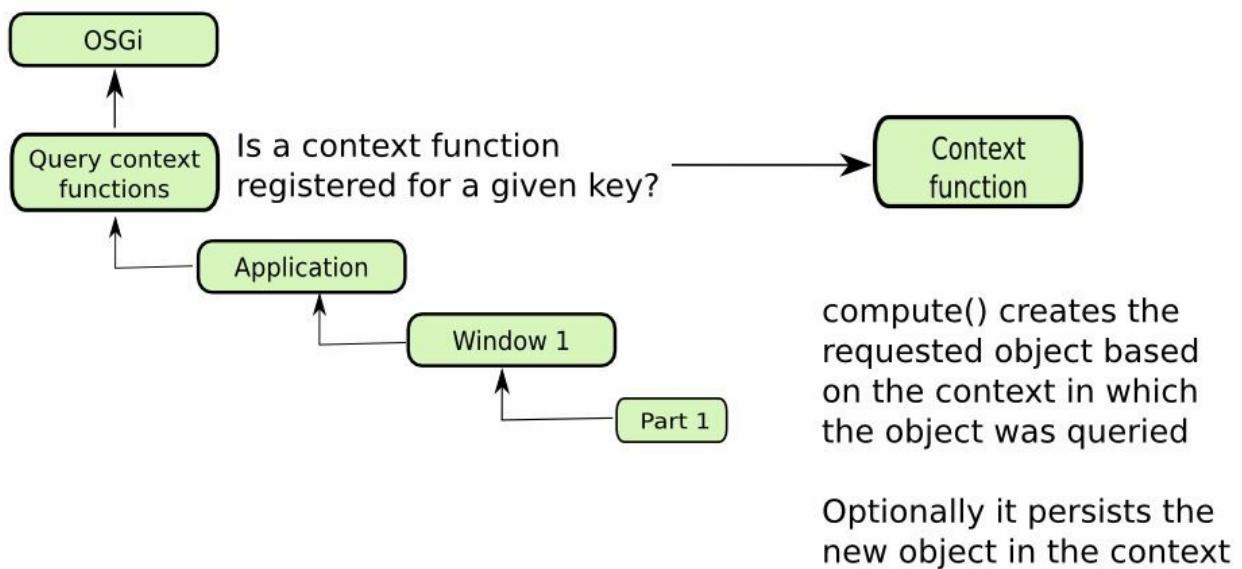
Part XXVI. Eclipse context functions

Chapter 123. Context functions

123.1. What are context functions?

A *context function* is an OSGi service which has access to the `IEclipseContext`. It allows you to lazily create an object for a given key if this key is not contained in the relevant `IEclipseContext`.

The context function registers itself for a certain key, for example a class name. Whenever the Eclipse dependency injection does not find an existing object under this key, it calls the `compute()` method of the object registered as context function.



In this `compute()` method the context function creates the requested object. As input you receive the local `IEclipseContext` in which the injection was invoked and the requested key.

It can also persist the created object into the context so that successive calls will return the same generated object.

123.2. Creation of a context function

Context functions are typically contributed as OSGi services. They extend the `ContextFunction` class from the `org.eclipse.e4.core.contexts` package. The Eclipse runtime adds context functions by default to the application context.

They register itself as OSGi service for the `IContextFunction` interface.

Tip

Question: Why should you register your service for the `IContextFunction` interface but implement the `ContextFunction` class?

Answer: If the interface gets another method in a later Eclipse release your service continues to work, as the abstract class `ContextFunction` can provide a default implementation.

You need to specify the key the context function is responsible for. This is done via the `service.context.key` property in the service definition.

If the key is a class you have to point to the fully qualified class. This key can be used for dependency injection. If you register a key which is not a class name, a consumer of the injection would have to use the `@Named` annotation to specify the key.

123.3. Examples for context function registrations

The following example shows the declaration of a context function. This function is available in the application context and responsible for the `vogella` key.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    name="com.example.e4.rcp.todo.contextservice.translate">

    <implementation class="com.example.e4.rcp.todo.contextservice.Test"/>

    <service>
        <provide interface="org.eclipse.e4.core.contexts.IContextFunction"/>
    </service>

    <property name="service.context.key" type="String"
        value="vogella"/>

</scr:component>
```

This key can be used for dependency injection as demonstrated in the following example.

```
// field injection
@Inject @Named("vogella") Todo todo;
```

The next example shows how to register the service for the `ITodoService` interface from the `com.example.e4.rcp.todo.model` package.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    name="ITodoService context function">
    <implementation
        class="com.example.e4.rcp.todo.service.internal.TodoServiceContextFunction">
        <property
            name="service.context.key"
            type="String"
            value="com.example.e4.rcp.todo.model.ITodoService"/>
    <service>
        <provide interface="org.eclipse.e4.core.contexts.IContextFunction"/>
    </service>
</scr:component>
```

The following example demonstrates a possible implementation of this context function.

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

```
import org.eclipse.e4.core.contexts.ContextFunction;
import org.eclipse.e4.core.contexts.ContextInjectionFactory;
import org.eclipse.e4.core.contexts.IEclipseContext;
import org.eclipse.e4.ui.model.application.MApplication;

import com.example.e4.rcp.todo.model.ITodoService;

public class TodoServiceContextFunction extends ContextFunction {
    @Override
    public Object compute(IEclipseContext context, String contextKey) {
        ITodoService todoService = ContextInjectionFactory.make(MyTodoServiceImpl.class,
            context);
        // add the new object to the application context
        MApplication application = context.get(MApplication.class);
        IEclipseContext ctx = application.getContext();
        ctx.set(ITodoService.class, todoService);
        return todoService;
    }
}
```

123.4. When to use context functions?

Using context functions instead of a OSGi services has the advantage that these functions have access to the `IEclipseContext` hierarchy. They have access to the context in which the dependency injection was called.

This allows them to lazily create objects using values from the context.

Standard OSGi services have no direct access to the `IEclipseContext`.

123.5. Publishing to the OSGi service registry from a context function

OSGi services may need to access the objects created by a context function. In this case you can also publish an OSGi service from a context function.

The following example demonstrates how to publish an implementation into the `IEclipseContext` and to the OSGi service registry.

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

import org.eclipse.e4.core.contexts.ContextFunction;
import org.eclipse.e4.core.contexts.ContextInjectionFactory;
import org.eclipse.e4.core.contexts.IEclipseContext;
import org.eclipse.e4.ui.model.application.MApplication;

import com.example.e4.rcp.todo.model.ITodoService;

public class TodoServiceContextFunction extends ContextFunction {
    @Override
    public Object compute(IEclipseContext context, String contextKey) {

        // create instance of ITodoService with dependency injection
        ITodoService todoService =
            ContextInjectionFactory.make(MyTodoServiceImpl.class, context);

        // add instance of ITodoService to context so that
        // test next caller gets the same instance
        MApplication app = context.get(MApplication.class);
        IEclipseContext appCtx = app.getContext();
        appCtx.set(ITodoService.class, todoService);

        // in case the ITodoService is also needed in the OSGi layer, e.g.
        // by other OSGi services, register the instance also in the OSGi service layer
        Bundle bundle = FrameworkUtil.getBundle(this.getClass());
        BundleContext bundleContext = bundle.getBundleContext();
        bundleContext.registerService(ITodoService.class, service, null);

        // return model for current invocation
        // next invocation uses object from application context
        return todoService;
    }
}
```

Chapter 124. Exercise: Create a context function

124.1. Target

OSGi services are created by the OSGi runtime and the OSGi runtime does not support the `@Inject` annotation. In addition OSGi services do not have direct access to the `IEclipseContext` hierarchy.

In this exercise you create a context function via OSGi declarative services. This allows you to use dependency injection to create an instance of the `ITodoService` service and to persist this instance in the Eclipse context. During the creation the event service is injected into your service instance; this allows you to send events from this service.

124.2. Add dependencies to the service plug-in

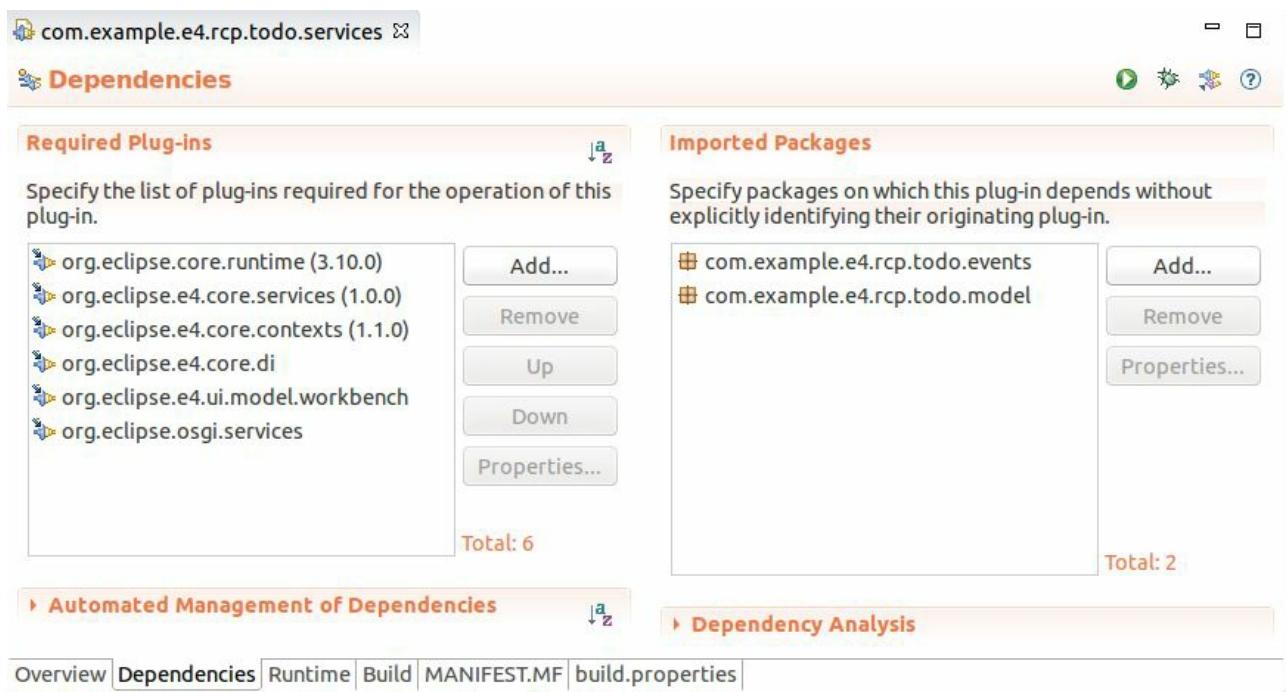
Ensure that the following framework dependencies are present in your `com.example.e4.rcp.todo.services` plug-in. Some of them are already part of your manifest file but the list is complete so that you can use this as a reference for other implementations.

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

For this exercise the service plug-in also needs to import the following packages (or plug-ins):

- `com.example.e4.rcp.todo.model`
- `com.example.e4.rcp.todo.events`

The resulting `MANIFEST.MF` file should look like the following screenshot.



124.3. Create a class for the context function

Create the following new class in your *.service plug-in.

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

import org.eclipse.e4.core.contexts.ContextFunction;
import org.eclipse.e4.core.contexts.ContextInjectionFactory;
import org.eclipse.e4.core.contexts.IEclipseContext;
import org.eclipse.e4.ui.model.application.MApplication;

import com.example.e4.rcp.todo.model.ITodoService;

public class TodoServiceContextFunction extends ContextFunction {
    @Override
    public Object compute(IEclipseContext context, String contextKey) {

        // create an instance of ITodoService
        ITodoService todoService =
            ContextInjectionFactory.make(MyTodoServiceImpl.class, context);

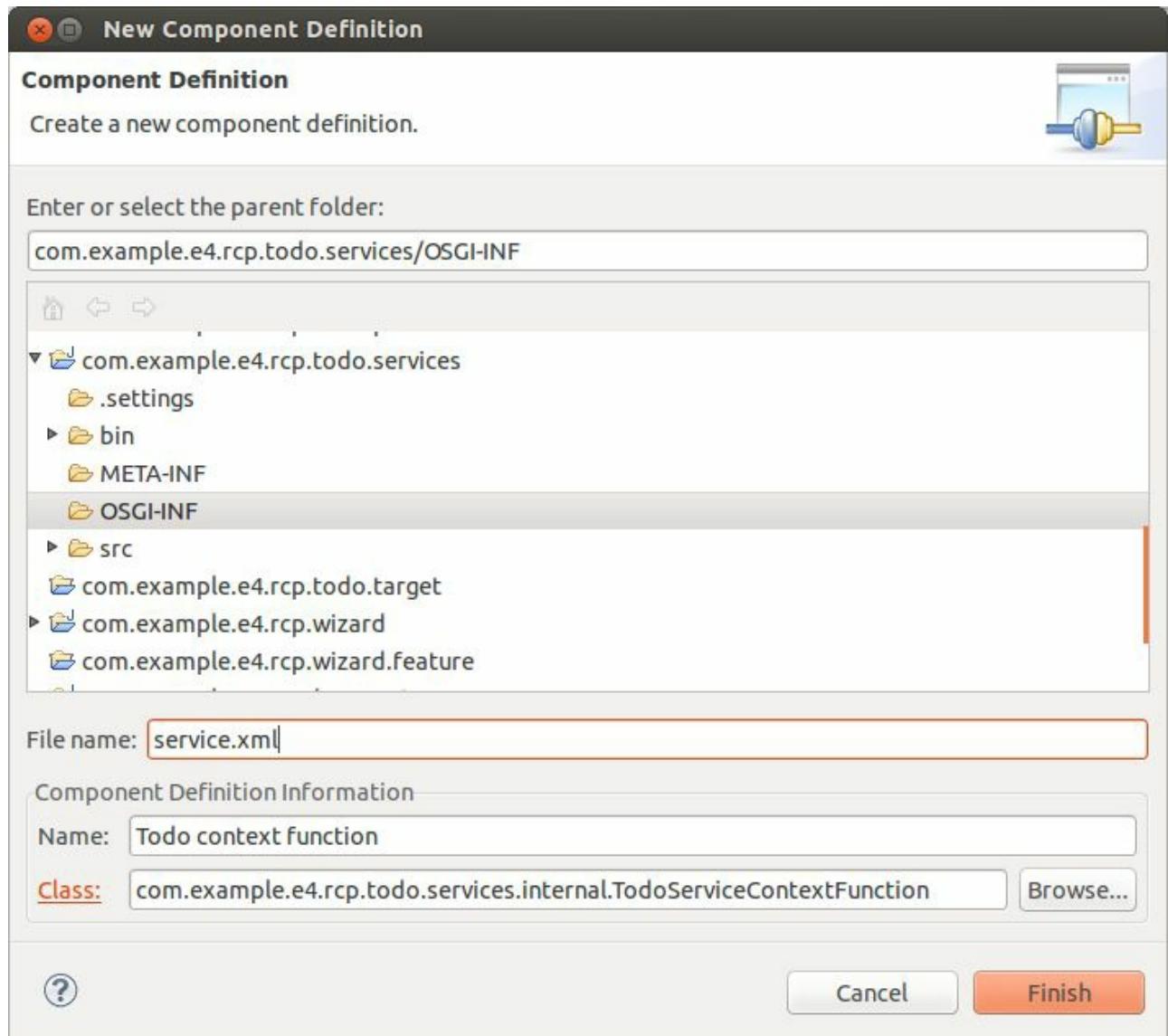
        // add this instance to the application context
        // next invocation uses the instance from the application context
        MApplication app = context.get(MApplication.class);
        IEclipseContext appCtx = app.getContext();
        appCtx.set(ITodoService.class, todoService);

        // return instance for the current invocation
        return todoService;
    }
}
```

124.4. Register the context function

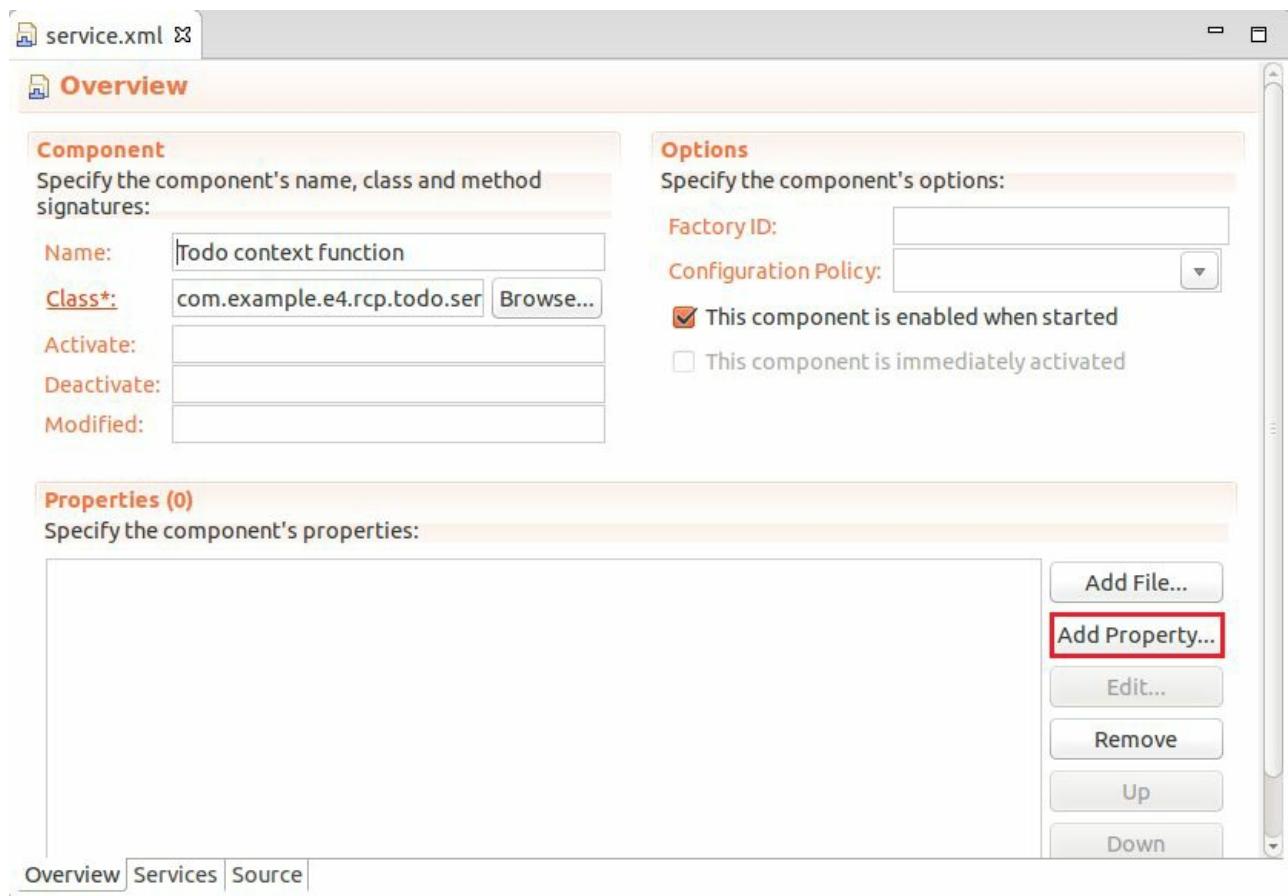
Ensure that you are in the *Plug-in Development* perspective, as the entry New → Component Definition is only available in this perspective.

Create a new *Component Definition* file called `service.xml`. To create this file, select your `OSGI-INF` folder, right-click on it and select New → Component Definition.

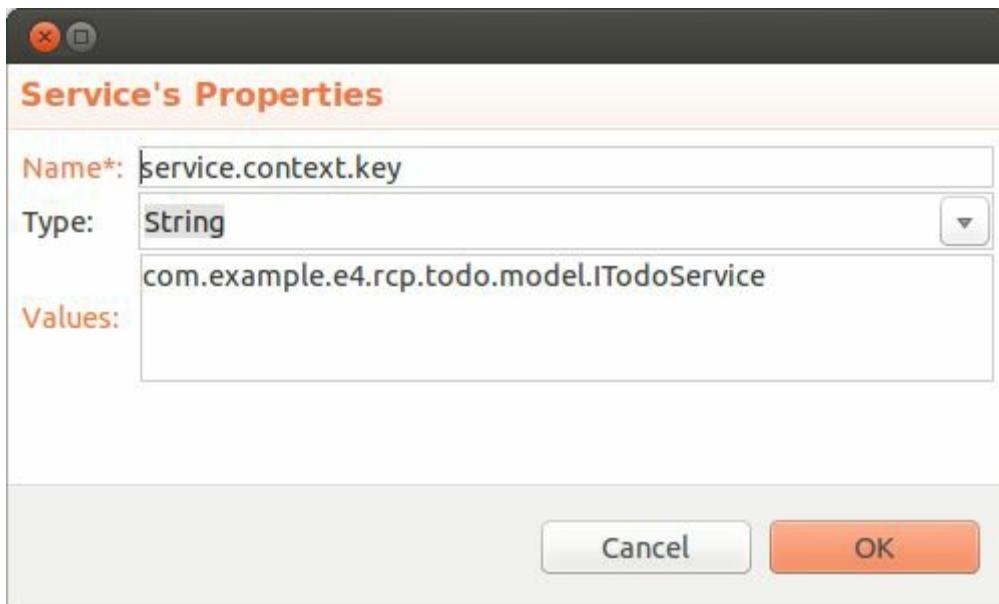


124.5. Specify the responsibility of the context function

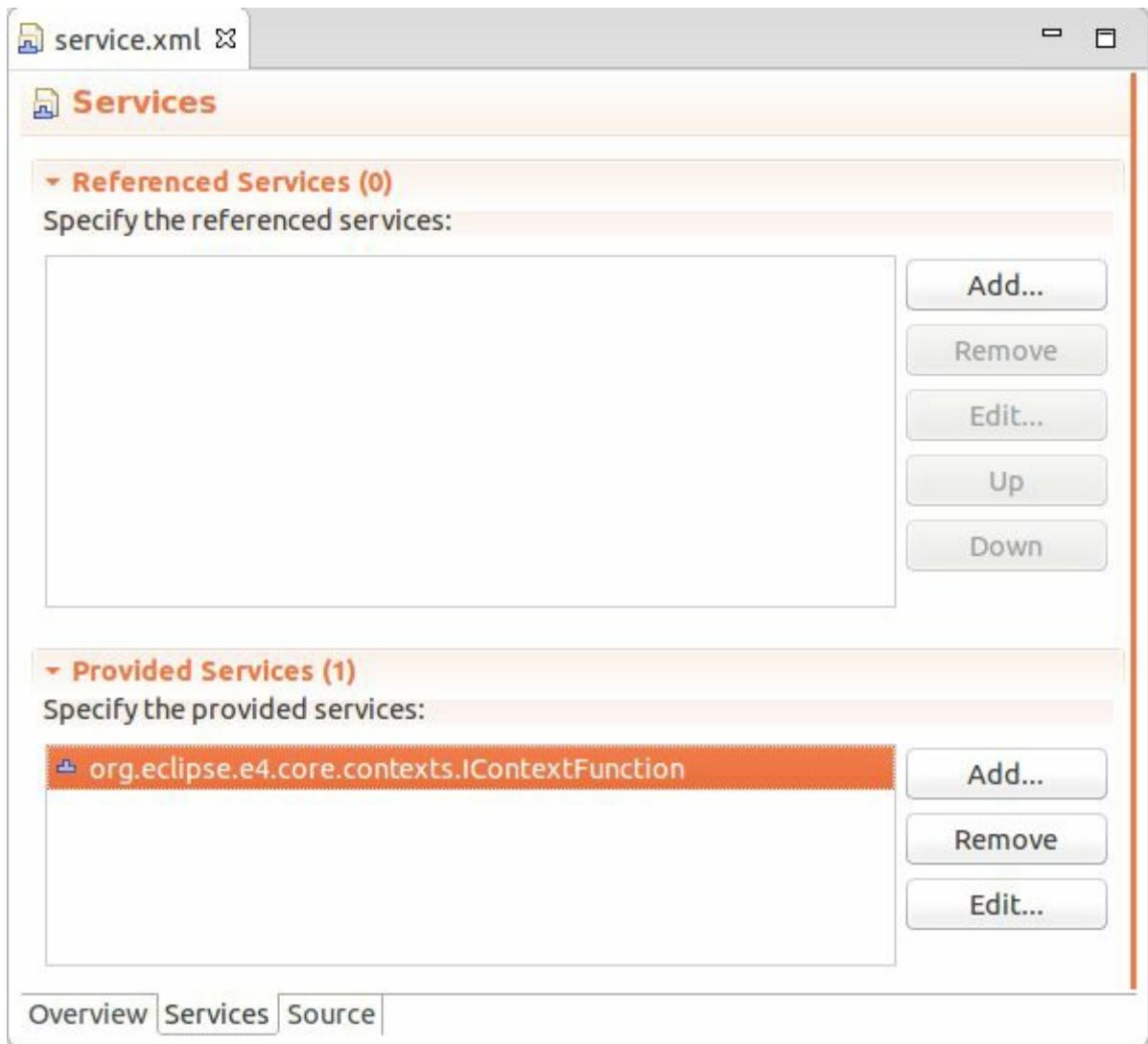
To define the key for which this context function is responsible, press the *Add Property...* button in the editor.



Use `com.example.e4.rcp.todo.model.ITodoService` as value for the `service.context.key` key.



Switch to the *Services* tab on the editor and configure that your service provides a service for the `org.eclipse.e4.core.contexts.IContextFunction` interface.



The resulting `service.xml` file should look similar to the following (except the additional whitespace).

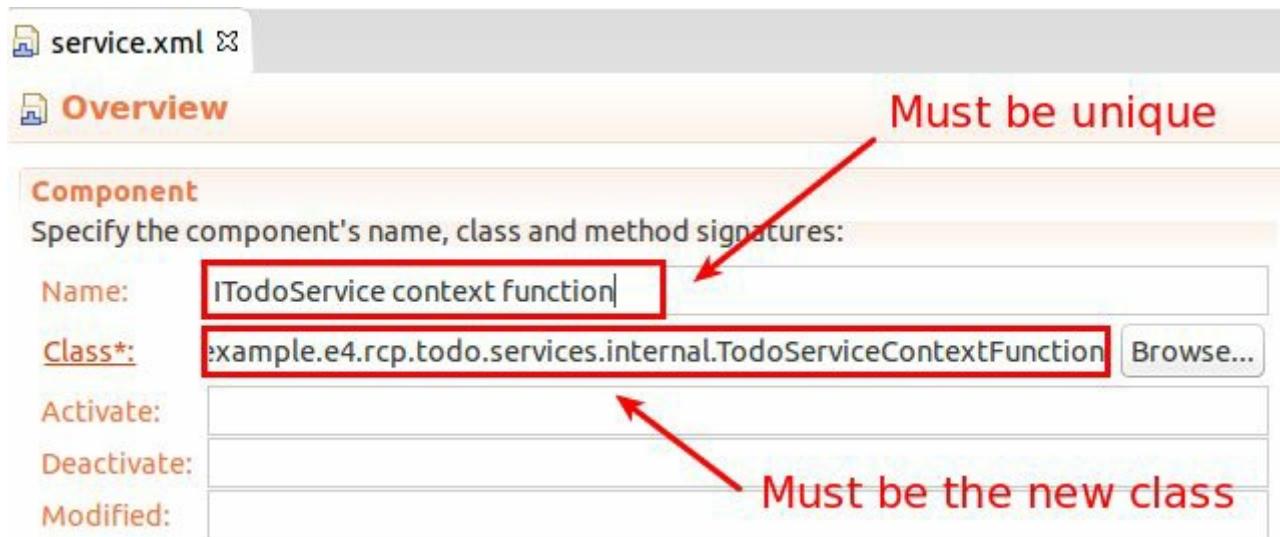
```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="Todo context function">
    <implementation
        class="com.example.e4.rcp.todo.services.internal.TodoServiceContextFunction"/>
    <property
        name="service.context.key"
        type="String"
        value="com.example.e4.rcp.todo.model.ITodoService"/>
    <service>
        <provide
            interface="org.eclipse.e4.core.contexts.IContextFunction"/>
    </service>
</scr:component>
```

124.6. Error analysis

Every service defined by OSGi declarative services must have a unique name.

Start your application. If you see the error that you have duplicate names, change the name of the new service to avoid a name collision with your existing OSGi service.

Also ensure that you are pointing to the correct class.



If the service still fails ensure that the service property is correctly entered. See [Section 124.5, “Specify the responsibility of the context function”](#) for details.

124.7. Deactivate your ITodoService OSGi service

Deactivate your OSGi service for the `ITodoService` which you have created in [Chapter 54, Exercise: Define and use an OSGi service](#).

To remove this service, delete the `OSGI-INF/component.xml` file and remove the reference to this file in the `MANIFEST.MF` file. This file should only contain a reference to your new context function as shown in the following listing.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Service
Bundle-SymbolicName: com.example.e4.rcp.todo.services
Bundle-Version: 1.0.0.qualifier
Bundle-Vendor: EXAMPLE
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Require-Bundle: org.eclipse.core.runtime;bundle-version="3.10.0",
  org.eclipse.e4.core.services;bundle-version="1.0.0",
  org.eclipse.e4.core.contexts;bundle-version="1.1.0",
  org.eclipse.e4.core.di,
  org.eclipse.e4.ui.model.workbench,
  org.eclipse.osgi.services
Bundle-ActivationPolicy: lazy
Service-Component: OSGI-INF/service.xml
Import-Package: com.example.e4.rcp.todo.events,
  com.example.e4.rcp.todo.model
```

Note

You could leave your OSGi service for the `ITodoService` active. Eclipse evaluates context functions before checking for other OSGi services. But it is good practice to remove things which are not used anymore.

124.8. Notifications from the ITodoService

Change the `MyTodoServiceImpl` class to get the `IEventBroker` injected and to send out events in case of changes in the data model.

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

import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.inject.Inject;

import org.eclipse.e4.core.services.events.IEventBroker;

import com.example.e4.rcp.todo.events.MyEventConstants;
import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;

public class MyTodoServiceImpl implements ITodoService {

    private static int current = 1;
    private List<Todo> todos;

    // use dependency injection in MyTodoServiceImpl
    @Inject
    private IEventBroker broker;

    public MyTodoServiceImpl() {
        todos = createInitialModel();
    }

    // always return a new copy of the data
    @Override
    public List<Todo> getTodos() {
        List<Todo> list = new ArrayList<Todo>();
        for (Todo todo : todos) {
            list.add(todo.copy());
        }
        return list;
    }

    // create or update an existing instance of Todo
    @Override
    public synchronized boolean saveTodo(Todo newTodo) {
        boolean created = false;
        Todo updateTodo = findById(newTodo.getId());
        if (updateTodo == null) {
            created = true;
            updateTodo = new Todo(current++);
        }
        updateTodo.setTitle(newTodo.getTitle());
        updateTodo.setDescription(newTodo.getDescription());
        updateTodo.setDueDate(newTodo.getDueDate());
        updateTodo.setPriority(newTodo.getPriority());
        updateTodo.setCompleted(newTodo.isCompleted());
        broker.postEvent(new TodoEvent(updateTodo));
        return created;
    }
}
```

```

        todos.add(updateTodo);
    }
    updateTodo.setSummary(newTodo.getSummary());
    updateTodo.setDescription(newTodo.getDescription());
    updateTodo.setDone(newTodo.isDone());
    updateTodo.setDueDate(newTodo.getDueDate());

    // configure the event

    // send out events
    if (created) {
        broker.post(MyEventConstants.TOPIC_TODO_NEW,
            createEventData(MyEventConstants.TOPIC_TODO_NEW,
                String.valueOf(updateTodo.getId())));
    } else {
        broker.
        post(MyEventConstants.TOPIC_TODO_UPDATE,
            createEventData(MyEventConstants.TOPIC_TODO_UPDATE,
                String.valueOf(updateTodo.getId())));
    }
    return true;
}

@Override
public Todo getTodo(long id) {
    Todo todo = findById(id);

    if (todo != null) {
        return todo.copy();
    }
    return null;
}

@Override
public boolean deleteTodo(long id) {
    Todo deleteTodo = findById(id);

    if (deleteTodo != null) {
        todos.remove(deleteTodo);
        // configure the event
        broker.
        post(MyEventConstants.TOPIC_TODO_DELETE,
            createEventData(MyEventConstants.TOPIC_TODO_DELETE,
                String.valueOf(deleteTodo.getId())));
        return true;
    }
    return false;
}

// Example data, change if you like
private List<Todo> createInitialModel() {
    List<Todo> list = new ArrayList<Todo>();
    list.add(createTodo("Application model", "Flexible and extensible"));
    list.add(createTodo("DI", "@Inject as programming mode"));
}

```

```

list.add(createTodo("OSGi", "Services"));
list.add(createTodo("SWT", "Widgets"));
list.add(createTodo("JFace", "Especially Viewers!"));
list.add(createTodo("CSS Styling", "Style your application"));
list.add(createTodo("Eclipse services", "Selection, model, Part"));
list.add(createTodo("Renderer", "Different UI toolkit"));
list.add(createTodo("Compatibility Layer", "Run Eclipse 3.x"));
return list;
}

private Todo createTodo(String summary, String description) {
    return new Todo(current++, summary, description, false, new Date());
}

private Todo findById(long id) {
    for (Todo todo : todos) {
        if (id == todo.getId()) {
            return todo;
        }
    }
    return null;
}

private Map<String, String> createEventData(String topic, String todoId) {
    Map<String, String> map = new HashMap<String, String>();
    // in case the receiver wants to check the topic
    map.put(MyEventConstants.TOPIC_TODO, topic);
    // which todo has changed
    map.put(Todo.FIELD_ID, todoId);
    return map;
}
}

```

124.9. Clean-up your user interface code

In the previous exercise you send out events directly from your `NewTodoHandler`. This is considered bad practice as the user interface would be responsible for sending out updates for data model changes.

Now that you move this logic to the implementation of the `ITodoService` you can remove this code. Thus remove all event notification calls from your plug-in. The following code shows the new `NewTodoHandler` handler implementation.

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

import java.util.Date;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.jface.wizard.WizardDialog;
import org.eclipse.swt.widgets.Shell;

import com.example.e4.rcp.todo.model.ITodoService;
import com.example.e4.rcp.todo.model.Todo;
import com.example.e4.rcp.todo.wizards.TodoWizard;

public class NewTodoHandler {
    @Execute
    public void execute(Shell shell, ITodoService model) {
        // use -1 to indicate a not existing id
        Todo todo = new Todo(-1);
        todo.setDueDate(new Date());
        WizardDialog dialog = new WizardDialog(shell, new TodoWizard(todo));
        if (dialog.open() == WizardDialog.OK) {
            model.saveTodo(todo);
        }
    }
}
```

124.10. Validating

Validate that the user interface still receives all events for model changes. For example, does the table in the `TodoOverviewPart` class still update when a new `Todo` is created by the wizard?

If you delete a `Todo` object via your menu, this should also update the table.

124.11. Review implementation

This implementation is better compared to the event notification via the user interface code. It places the responsibility of sending out the relevant events into the corresponding service.

The developer needs only to register for the correct events to update the relevant user interface controls.

Part XXVII. Model add-ons

Chapter 125. Using model add-ons

125.1. What are model add-ons?

The application model allows you to create model objects called *add-ons*. These components can enhance the application with additional functionality.

Add-ons add flexibility to the application model. They allow you to extend or change the default Eclipse behavior without having to modify existing Eclipse code. You just have to replace the related model component.

For example, the default drag and drop support of parts in Eclipse is implemented via an add-on. Other examples are the keybinding or the command processing. If this functionality is undesired, you can simply remove the component from the application model. If you want a different behavior, you simply register your own add-on in your application model.

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

125.2. Add-ons from the Eclipse framework

Currently the following standard add-ons are useful for Eclipse applications. Their class names give an indication of their provided functionality.

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

125.3. Additional SWT add-ons

Additional add-ons are available. 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` classes from this bundle as add-ons in your application model. This plug-in contains also the `MinMaxAddon` class which allows adding the functionality to minimize or maximize your application.

The `org.eclipse.e4.ui.workbench.addons.swt` plug-in contributes these add-ons automatically to your application model via a model processor at runtime.

125.4. Relationship to other services

Add-ons are created before the Eclipse rendering framework renders the model and after the event service has been created.

This allows add-ons to alter the user interface that is produced by the rendering engine. For example, the `MinMaxAddon` add-on changes the tab folders created for the part stacks to have the min/max buttons in the corner.

The Eclipse platform uses events sent via the event service to communicate changes in the application model. For example, if a part is activated, the Eclipse platform sends out an event for this. Add-ons can subscribe to these events from the Eclipse platform and react to them.

Chapter 126. Exercise: Model add-on to change the close behavior

126.1. Target

Note

This exercise is optional.

In this exercise you ensure that the user is prompted if he wants to leave the application with some unsaved changes. This should also work if he presses the close button of the application corner.

126.2. Creating a plug-in

Create a new simple plug-in called com.vogella.e4.addon.exitconfirmation.

Add the following dependencies to your plug-in.

- org.eclipse.core.runtime
- org.eclipse.e4.ui.di
- org.eclipse.e4.ui.model.workbench
- org.eclipse.e4.ui.workbench
- org.eclipse.e4.core.di
- org.eclipse.jface
- org.eclipse.e4.core.contexts
- org.osgi.services

126.3. Creating the model add-on

Define the following class which is used as model add-on

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

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

import javax.inject.Inject;

import org.eclipse.e4.core.di.annotations.Optional;
import org.eclipse.e4.ui.di.UIEventTopic;
import org.eclipse.e4.ui.model.application.MApplication;
import org.eclipse.e4.ui.model.application.ui.basic.MWindow;
import org.eclipse.e4.ui.workbench.IWorkbench;
import org.eclipse.e4.ui.workbench.UIEvents;
import org.eclipse.e4.ui.workbench.modeling.EModelService;
import org.eclipse.e4.ui.workbench.modeling.EPartService;
import org.eclipse.e4.ui.workbench.modeling.ElementMatcher;
import org.eclipse.e4.ui.workbench.modeling.IWindowCloseHandler;
import org.eclipse.jface.dialogs.IDialogConstants;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Shell;

// @PostConstruct does not work as the workbench gets
// instantiated after the processing of the add-ons
// hence this approach uses method injection

public class WindowCloseListenerAddon {

    @Inject
    @Optional
    private void subscribeApplicationCompleted
        (@UIEventTopic(UIEvents.UILifeCycle.APP_STARTUP_COMPLETE)
        final MApplication application,
        final IWorkbench workbench) {
        // only if a close of the main window is requested
        MWindow mainWindow = findMainWindow(application);

        mainWindow.getContext().set(IWindowCloseHandler.class, new IWindowCloseHandler {
            @Override
            public boolean close(MWindow window) {
                Collection<EPartService> allPartServices = getAllPartServices(application);

                if (containsDirtyParts(allPartServices)) {
                    Shell shell = (Shell) window.getWidget();
                    MessageDialog msgDialog = new MessageDialog(shell, "Close Application"
                        "Do you want to save the changes before you close the entire application?"
                        MessageDialog.QUESTION_WITH_CANCEL, new String[] { IDialogConstants.NO_LABEL, IDialogConstants.CANCEL_LABEL },
                    );
                }
            }
        });
    }

    private boolean containsDirtyParts(Collection<EPartService> services) {
        for (EPartService service : services) {
            if (service.isDirty())
                return true;
        }
        return false;
    }

    private MWindow findMainWindow(MApplication application) {
        for (MWindow window : application.getWindowList()) {
            if (window instanceof IWorkbenchWindow)
                return window;
        }
        return null;
    }

    private Collection<EPartService> getAllPartServices(MApplication application) {
        List<EPartService> services = new ArrayList<>();
        for (MWindow window : application.getWindowList()) {
            if (window instanceof IWorkbenchWindow) {
                IWorkbenchWindow workbenchWindow = (IWorkbenchWindow) window;
                for (IWorkbenchPage page : workbenchWindow.getPageList()) {
                    EPartService partService = page.getPartService();
                    if (partService != null)
                        services.add(partService);
                }
            }
        }
        return services;
    }
}
```

```

        0);
    switch (msgDialog.open()) {
        case 0: // YES: save all and close
            saveAll(allPartServices);
            return workbench.close();

        case 1: // NO: save nothing and close
            return workbench.close();

        case 2: // CANCEL: prevent close
        default:
            return false;
    }
} else {
    Shell shell = (Shell) window.getWidget();
    if (MessageDialog.openConfirm(shell, "Close Application",
        "Do you really want to close the entire application?")) {
        return workbench.close();
    }
}

return false;
}
);
}
}

private static MWindow findMainWindow(MApplication application) {
    // instead of using the index you could also use a tag on the MWindow to
    // mark the main window
    return application.getChildren().get(0);
}

private static Collection<EPartService> getAllPartServices(MApplication applic
List<EPartService> partServices = new ArrayList<EPartService>();

EModelService modelService = application.getContext().get(EModelService.clas
List<MWindow> elements = modelService.findElements(application, MWindow.clas
    EModelService.IN_ACTIVE_PERSPECTIVE,
    new ElementMatcher(null, MWindow.class, (List<String>) null));
for (MWindow w : elements) {
    if (w.isVisible() && w.isToBeRendered()) {
        EPartService partService = w.getContext().get(EPartService.class);
        if (partService != null) {
            partServices.add(partService);
        }
    }
}

return partServices;
}

private static boolean containsDirtyParts(Collection<EPartService> partService
for (EPartService partService : partServices) {
    if (!partService.getDirtyParts().isEmpty())

```

```
        return true;
    }

    return false;
}

private static void saveAll(Collection<EPartService> partServices) {
    for (EPartService partService : partServices) {
        partService.saveAll(false); // false: save without prompt
    }
}
```

Part XXVIII. Supplementary application model data

Chapter 127. Supplementary model data

127.1. Adding additional information on the model elements

The *Supplementary* tab in the application model editor allows you to enter additional information about the selected model element. This data can also be modified and accessed via Eclipse API.

127.2. Tags

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

Tags are automatically persisted by the Eclipse runtime between application restarts and are represented as a `List` of type `String`.

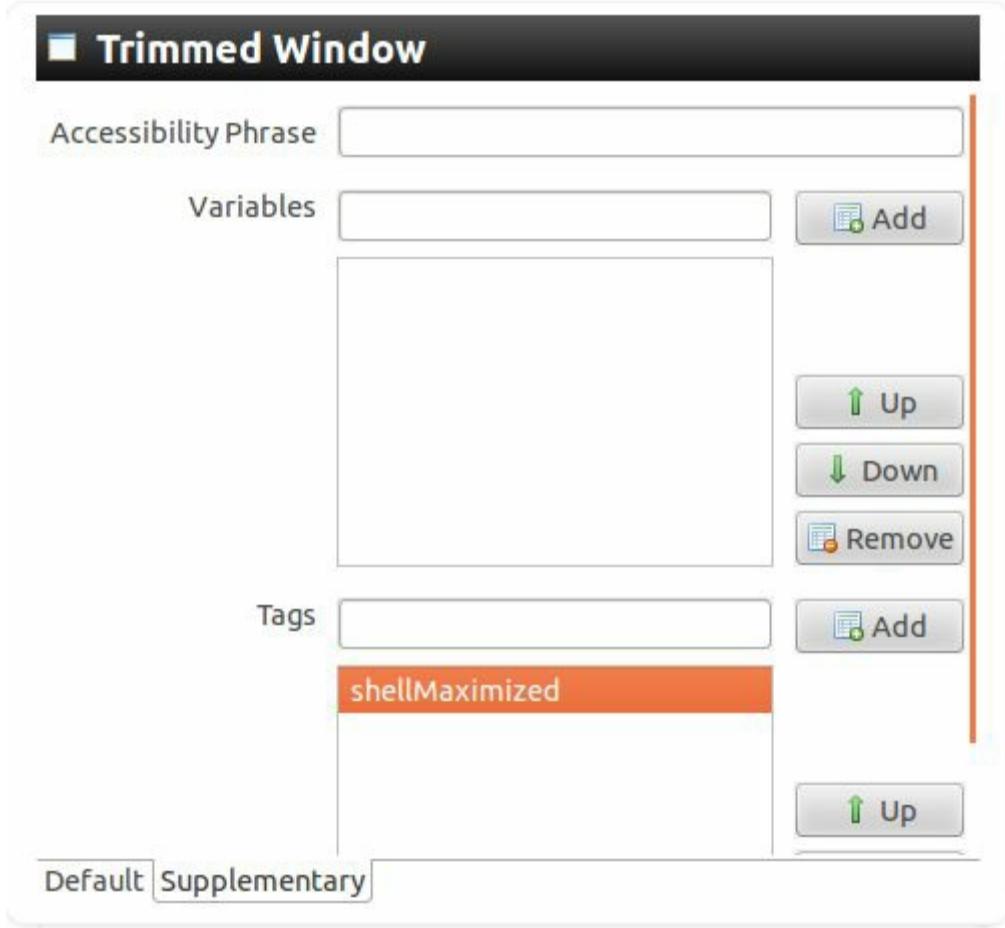
Tip

Tags are also available to the CSS Engine as additional class selectors. For example, the currently active part is tagged as `active` at runtime and the CSS engine allows to style components based on this with the selector `.MPartStack.active`. If you want to use tag as CSS selectors, don't use whitespace in them.

You can define your own tags and define CSS for this. This is a great way to allow custom CSS to be integrated into the model.

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 127.6, “Relevant tags and persisted state keys in the application model”](#).

127.3. Variables

You can also define *Variables* in the *Supplementary* tab which can be used as *context variables*. If you use this approach, Eclipse creates keys in the context which are marked as modifiable (by descendants). See [Chapter 36, Dependency injection in Eclipse](#) for the concept of dependency injection and see [Chapter 120, Accessing and extending the Eclipse context](#) to learn more about context variables.

127.4. Persisted state

Model elements can have persisted state key/value pairs assigned to them. If you retrieve the model element, you can get and set this persisted state.

```
String yourKey = "key1";  
  
// modelObject is the model object  
// retrieved via dependency injection (e.g., a 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 allows to store key/values pairs based on Strings.

Persisted state data can also be used by the Eclipse framework to define or change the application behavior.

127.5. Transient data

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

Transient data is not persisted between application restarts and needs to be generated at runtime.

As transient data is not persisted, it can not be assigned to model elements via the application model editor, you need to use the `getTransientData()` method.

127.6. Relevant tags and persisted state keys 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` interface. It is up to the renderer implementation and model add-ons 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 127.1. Relevant tags of application model elements

Tag	Model element	Description
<i>shellMaximized</i>	<i>Window</i> or <i>Trimmed Window</i>	Window is maximized at start of the application.
<i>shellMinimized</i>	<i>Window</i> or <i>Trimmed Window</i>	Window is minimized at start of the application.
<i>NoAutoCollapse</i>	<i>PartStack</i>	Can be added to a <i>PartStack</i> container. With this flag the <i>PartStack</i> is not collapsed by the <i>MinMax</i> add-on even if you remove all parts from it.
<i>FORCE_TEXT</i>	<i>ToolItem</i>	Enforces that text and icon are shown for a toolbar item.
<i>NoMove</i>	<i>Part</i>	Prevents the user from moving the part. This tag is evaluated by the <i>DnDAddon</i> class which handles drag and drop by default in the Eclipse platform.
<i>styleOverride</i>	<i>MWindow</i> or <i>MPartStack</i>	Allows to define the style bits of the user interface control. For example, to create an SWT shell which cannot be resized, you need to supply the integer value for <code>SWT.SHELL_TRIM & (~SWT.RESIZE)</code> , which is 1248 on the <i>MWindow</i> application element. If, for example, your <i>MPartStack</i> should render its tab below the parts, use 1024.

Chapter 128. Exercise: Using model tags and persisted data

128.1. Target

In this exercise you are about to learn how to use tags and persisted state data on the application model elements.

Tip

Use [Section 127.6, “Relevant tags and persisted state keys in the application model”](#) for information on the tags or look directly at the `IPresentationEngine` interface.

128.2. Use tags

Add the `NoMove` tag to your parts and ensure that you cannot move it via drag and drag.

Make all your parts closable via the corresponding model attribute. Assign the `NoAutoCollapse` to a stack and close all parts in this stack. Validate that your stack is not closed.

128.3. Optional: Use persisted state

Add to the `styleOverride` parameter with the value 1024 as *Persisted State* to on of your *PartStacks* model elements.

This causes this part stack to render its tabs at the bottom.

Part XXIX. Application model modularity

Chapter 129. Contributing to the application model

129.1. Modularity support in Eclipse RCP

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

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

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

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

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

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

129.2. Contributing to the application model

Model fragments

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

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

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

Tip

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

Model processors

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

Position of new model elements

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

Table 129.1. Position in list

Value	Description
<code>first</code>	Positions the element on the beginning of the list.
<code>index=theIndex</code>	Places the new model elements at position <code>theIndex</code> . Example: <code>index=0</code>
<code>before=theOtherElementsId</code>	Places the new model elements before the model element with the ID <code>theOtherElementsId</code> .

after=theotherelementsid Places the new model elements after the model element with the ID *theotherelementsid*.

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

Usage of IDs

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

Tip

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

Comparison with Eclipse 3.x

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

129.3. Constructing the runtime application model

User Changes

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

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

Tip

User changes can be deleted at start of your application via the `clearPersistedState` parameter as a launch parameter. In most cases this is undesired behavior for an exported application and only used during development.

Runtime application model

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

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

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

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

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

Note

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

129.4. Fragment extension elements

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

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

Table 129.2. Contribution, Featurename and Element id

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

Chapter 130. Exercise: Contributing via model fragments

130.1. Target

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

130.2. Create a new plug-in

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

130.3. Add the dependencies

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

- `org.eclipse.core.runtime`
- `org.eclipse.swt`
- `org.eclipse.jface`
- `org.eclipse.e4.core.di`
- `org.eclipse.e4.ui.workbench`
- `org.eclipse.e4.ui.di`

130.4. Create a handler class

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

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

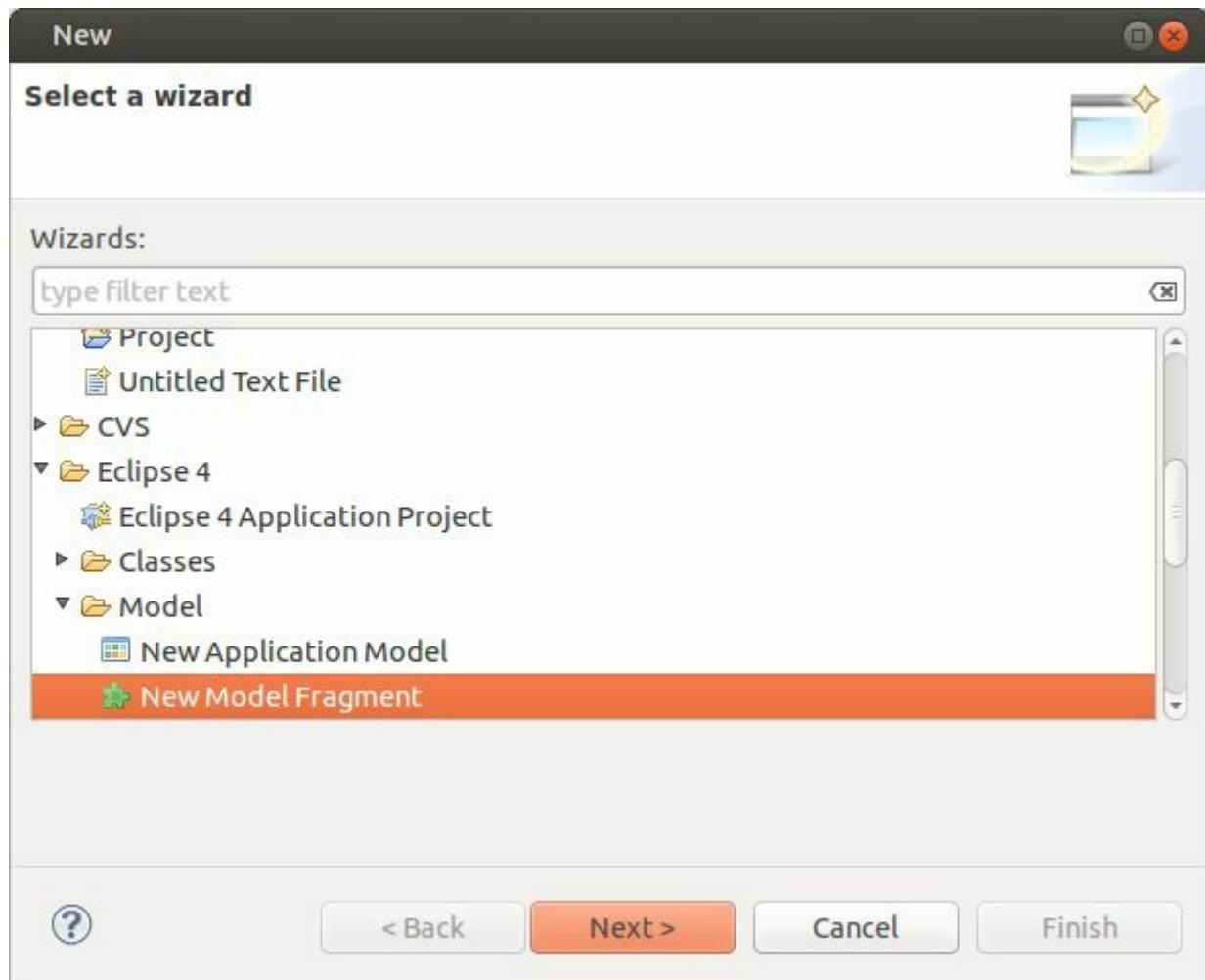
import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Shell;

public class OpenMapHandler {

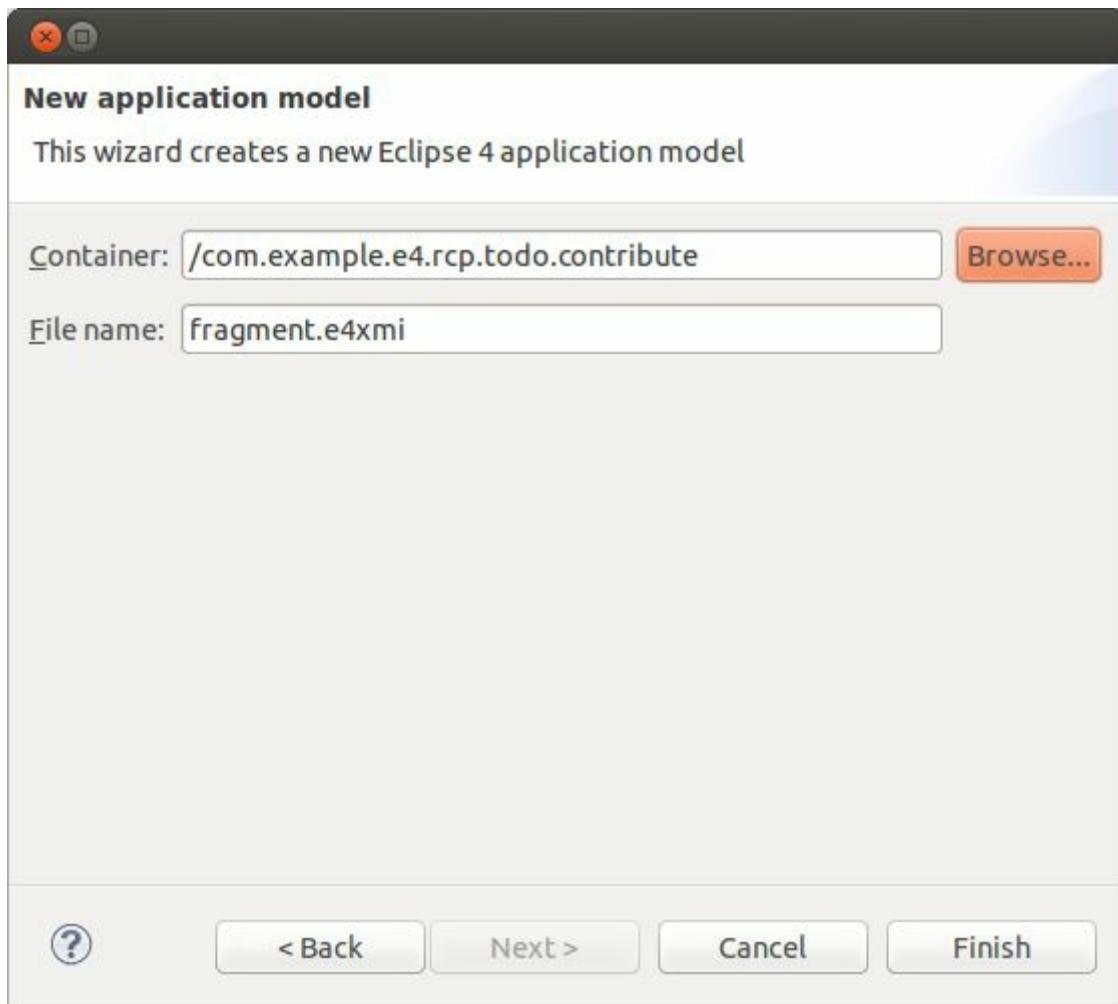
    @Execute
    public void execute(Shell shell) {
        MessageDialog.openInformation(shell, "Test", "Just testing");
    }
}
```

130.5. Create a model fragment

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



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



Press the *Finish* button.

130.6. Validate that the fragment is registered as extension

This wizard which creates the fragment also adds the `org.eclipse.e4.workbench.model` extension to your `contribute` plug-in. To review this open the `plugin.xml` file.

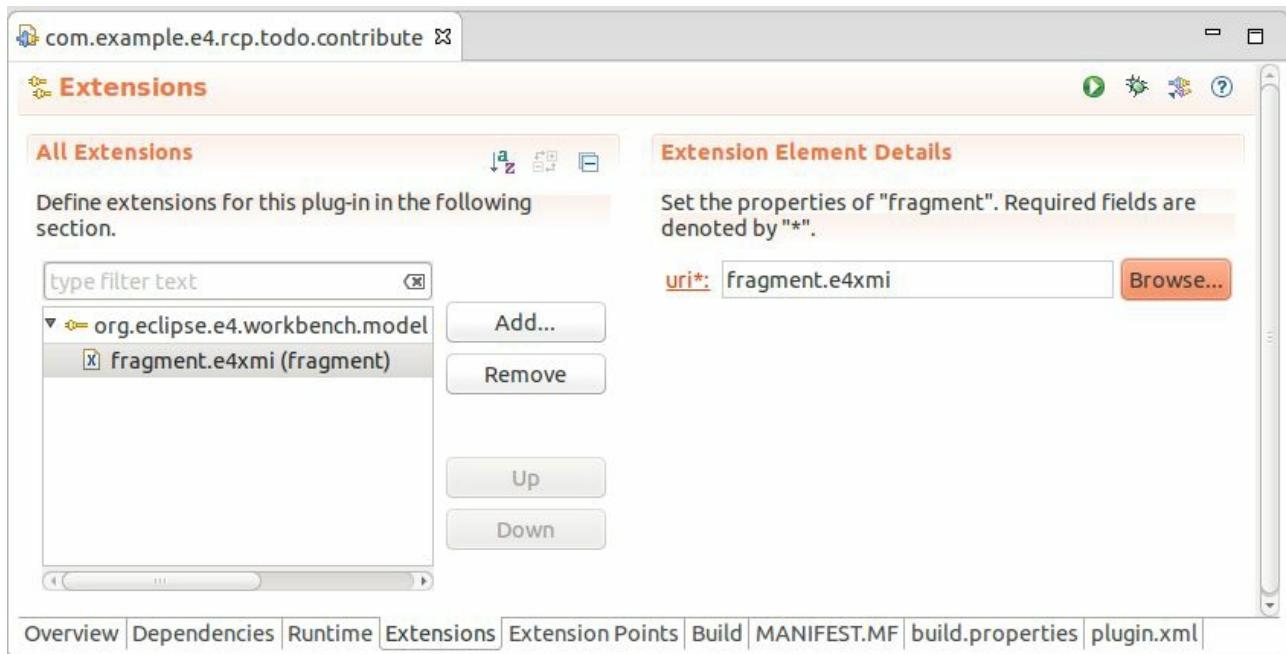
Tip

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

Warning

At the time of this writing the wizard deletes existing content in the `plugin.xml`, if you create a new fragment. This is a bug and might already be solved once you read this.

On the *Extensions* tab validate that you have an entry similar to the following screenshot.



Tip

If entry in the `plugin.xml` is missing you can create it by clicking on

the *Add...* button and by adding add a new extension for the `org.eclipse.e4.workbench.model` extension point. Afterwards you the right mouse click to add a fragment to it.

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

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

130.7. Adding model elements

Open the `fragment.e4xmi` file in its editor. Select the *Model Fragments* node and press the *Add...* button.

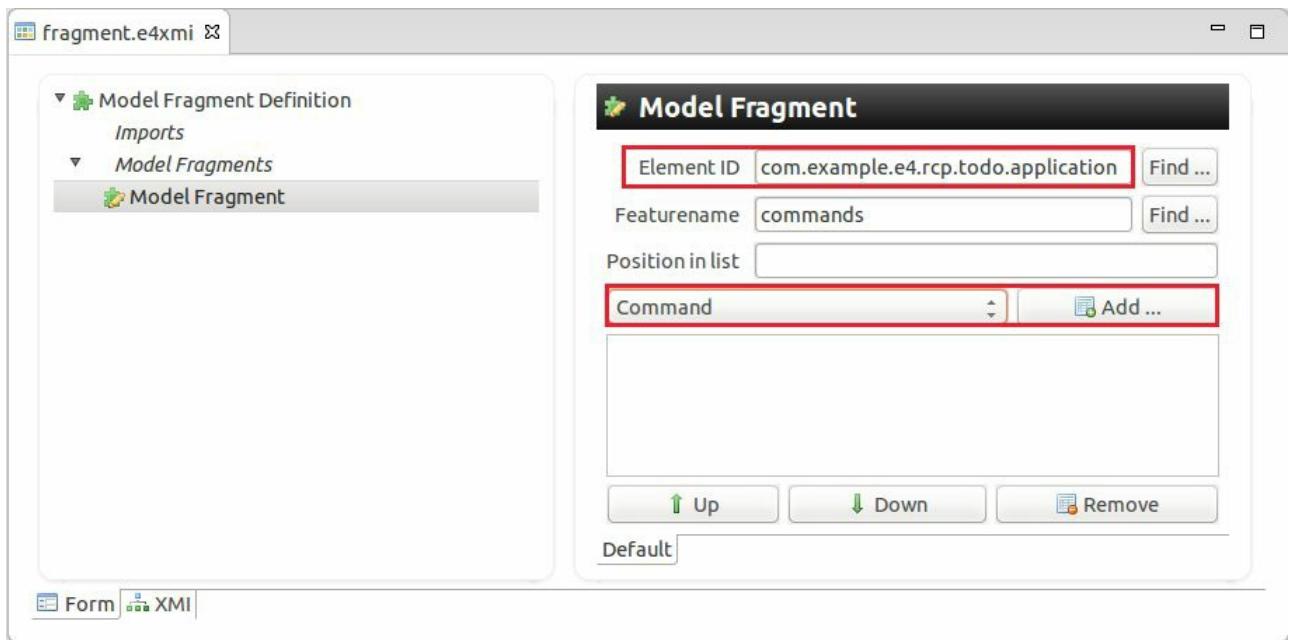


Use `com.example.e4.rcp.todo.application` as the *Element ID*. This is the ID of the *Application* model element in your `Application.e4xmi` file.

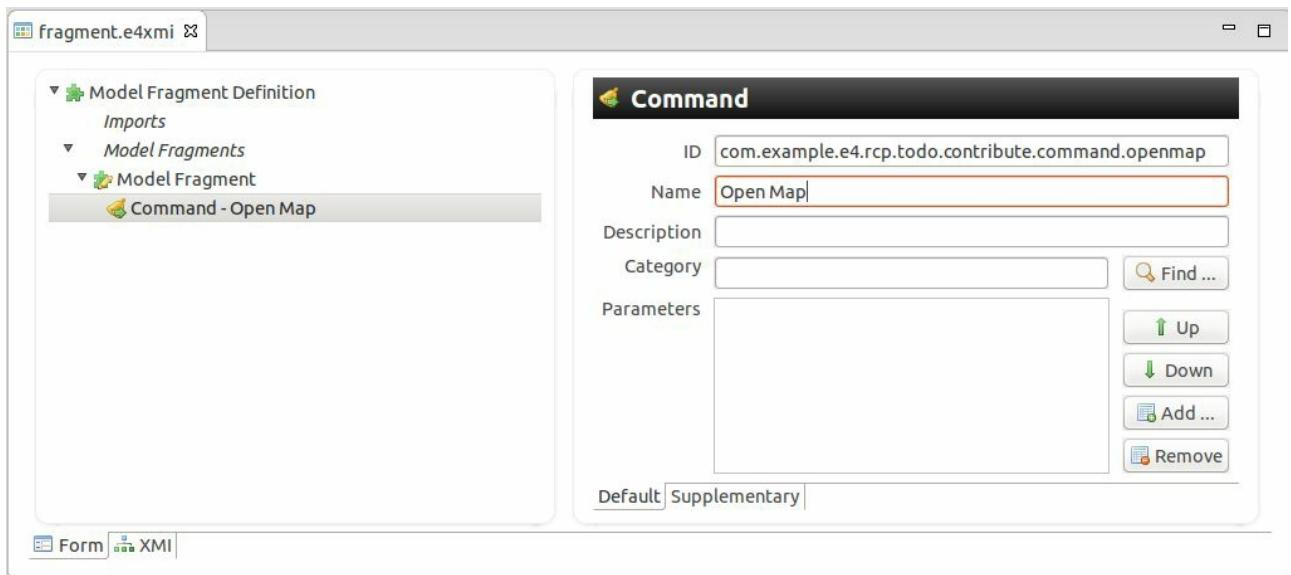
Warning

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

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

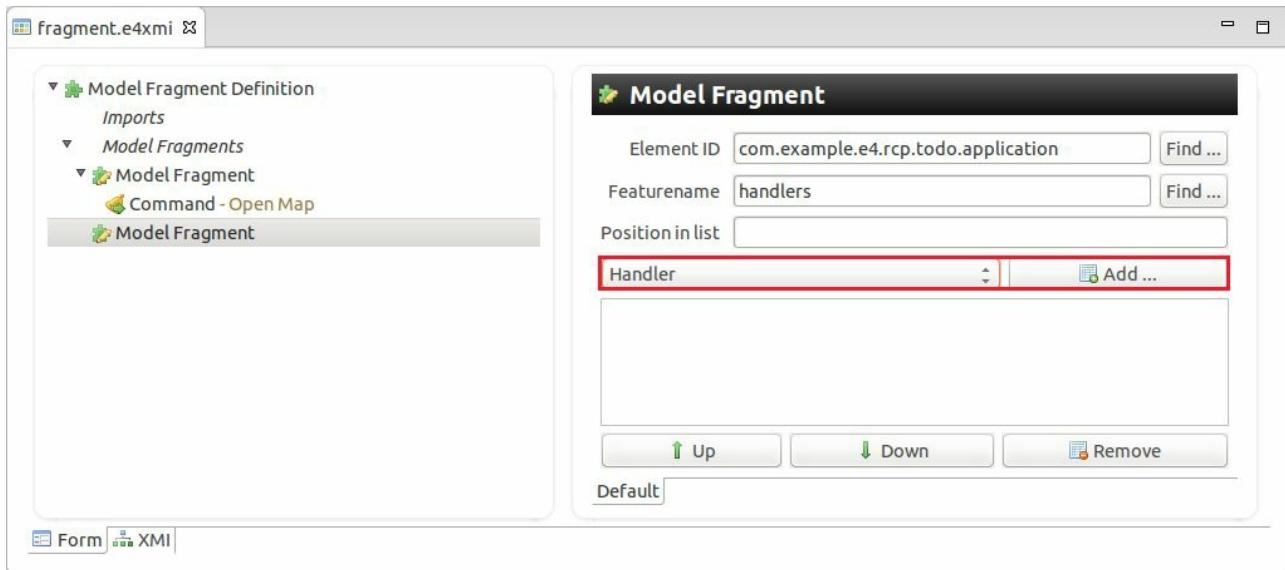


Use `com.example.e4.rcp.todo.contribute.command.openmap` for the *ID* field and Open Map for the *Name* field.

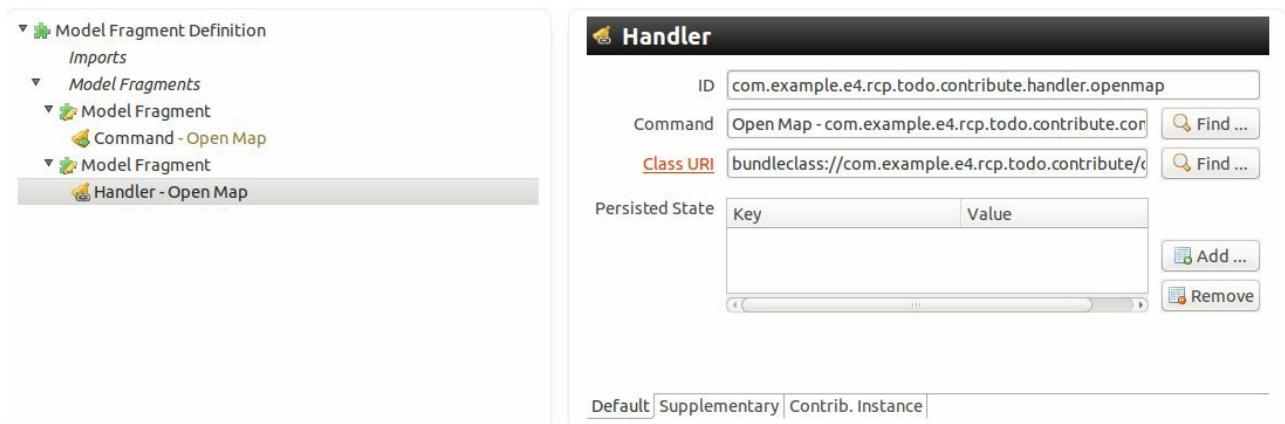


Create a new model fragment for the handler. The *Element ID* is again your application ID, the *Featurename* is `handlers`.

Add a *Handler* to this model fragment.



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



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

The left pane shows a tree structure of model fragments:

- Model Fragment Definition
 - Imports
 - Model Fragments
 - Model Fragment (selected)
 - Command - Open Map
 - Model Fragment
 - Handler - Open Map
 - Model Fragment

The right pane shows the details for the selected "Model Fragment" entry:

Model Fragment

Element ID: org.eclipse.ui.main.menu
Find ...

Featurename: children
Find ...

Position in list:

Menu:

Up | Down | Remove

Default

Warning

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

The left pane shows the structure of the *Application.e4xmi* file:

- Application
 - Addons
 - Binding Contexts
 - BindingTables
 - Handlers
 - Commands
 - Command Categories
 - Windows
 - Trimmed Window - To-do
 - Main Menu (highlighted with a red box)
 - Menu - File
 - Menu - Edit
 - Menu
 - Menu - Styling
 - Menu - Dynamic

The right pane shows the properties for the highlighted "Main Menu" entry:

Main Menu

Id: org.eclipse.ui.main.menu

Mnemonics:

Children:

Menu - File | Menu - Edit | Menu | Menu - Styling | Menu - Dynamic

Up | Down | Remove

Visible-When Expression: <None>

To Be Rendered:

Default: Supplementary

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

Model Fragment Definition

- Imports
- Model Fragments
 - Model Fragment
 - Command - Open Map
 - Model Fragment
 - Handler - Open Map
 - Model Fragment

Model Fragment

Element ID Find ...

Featurename Find ...

Position in list

Menu

Default

Menu

ID

Label

Mnemonics

Children Add ...

Tooltip

Icon URI

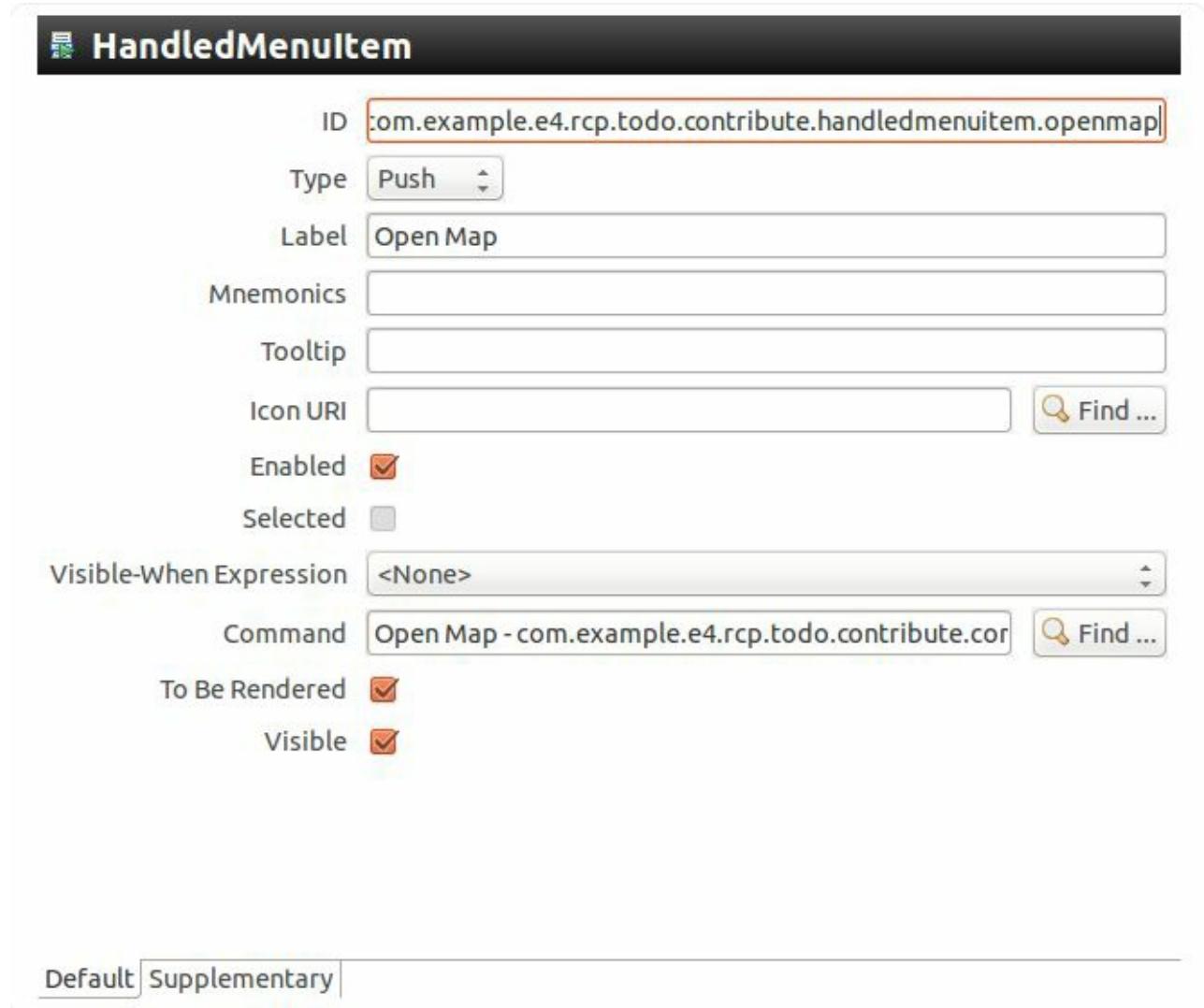
Visible-When Expression

To Be Rendered

Visible

Default

Add a *HandledMenuItem* which points to your new command. The process of defining these entries is the same as defining menus in the *Application.e4xmi* file. See [Part XIV, “Defining menus and toolbars”](#) for further information. The created entry should be similar to the following screenshot.



130.8. Update the product configuration (via the feature)

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

Warning

Ensure that you added this new plug-in to your feature and saved the changes.

130.9. Validating

Start your application.

Warning

Remember to start via the product to update the launch configuration.

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

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

130.10. Exercise: Contributing a part

Note

This exercise is optional.

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

Chapter 131. Exercise: Implementing a model processor

131.1. Target

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

131.2. Enter the dependencies

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

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

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

131.3. Create the Java classes

Create the following dialog and handler classes.

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

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

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

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

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

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

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

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

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

```
}
```

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

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

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

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

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.model.application.ui.menu.MDirectMenuItem;
import org.eclipse.e4.ui.model.application.ui.menu.MMenu;
import org.eclipse.e4.ui.model.application.ui.menu.MMenuItemElement;
import org.eclipse.e4.ui.workbench.modeling.EModelService;

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

public class MenuProcessor {

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

    @Execute
    public void execute(EModelService modelService) {
        // remove the old exit menu entry
        if (!menu.getChildren().isEmpty()) {
            List<MMenuItemElement> list = new ArrayList<>();
            for (MMenuItemElement element : menu.getChildren()) {
                // use ID instead of label as label is later translated
                if (element.getElementId() != null) {
                    if (element.getElementId().contains("exit")) {
                        list.add(element);
                    }
                }
            }
            menu.getChildren().removeAll(list);
        }

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

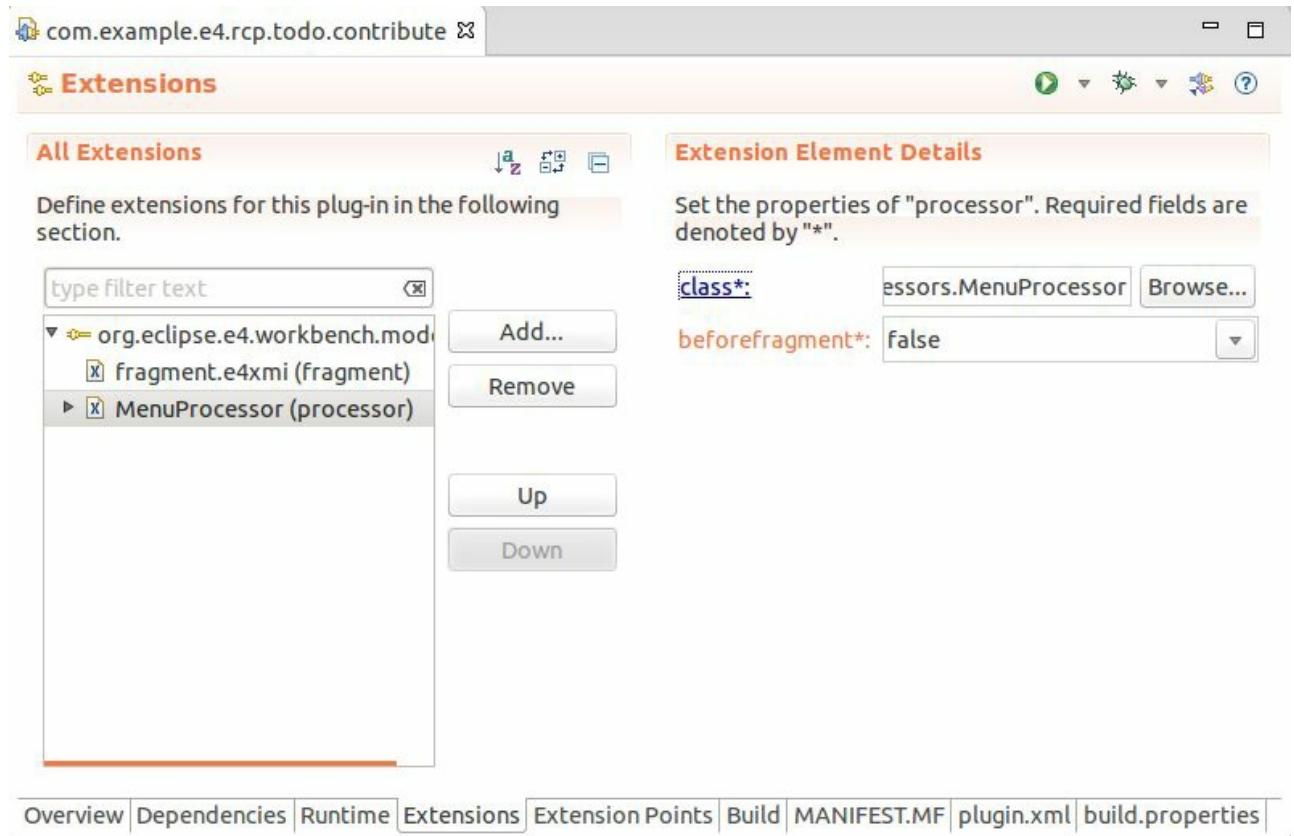
```
        menu.getChildren().add(menuItem);
    }
}
```

Warning

Ensure that your menu entry labeled with "Exit" in the *Application.e4xmi* file, contains "exit" in its ID.

131.4. Register processor via extension

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



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

The screenshot shows the Eclipse E4 RCP Extensions view for the plugin 'com.example.e4.rcp.todo.contribute'. The 'Extensions' tab is selected. In the center, there's a list of extension points under 'All Extensions'. One item, 'org.eclipse.ui.file.menu', is highlighted with an orange background. To the right, the 'Extension Element Details' panel is open, showing configuration fields for this extension point. The 'id*' field is set to 'org.eclipse.ui.file.menu' and the 'contextKey' field is empty.

All Extensions

Define extensions for this plug-in in the following section.

type filter text

▼ org.eclipse.e4.workben

Add...

fragment.e4xmi (fragr)

▼ MenuProcessor (proce

Remove

org.eclipse.ui.file.me

Edit...

Down

Extension Element Details

Set the properties of "element". Required fields are denoted by "*".

id*: org.eclipse.ui.file.menu

contextKey:

Warning

This assumes that you used *org.eclipse.ui.file.menu* as ID for your *File* menu in the main application model.

Warning

The ID of the element defined in the extension point must match the `@Named` value in the processor, otherwise your menu is not injected into the processor.

131.5. Validating

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

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

Part XXX. Eclipse application life cycle

Chapter 132. Registering for the application life cycle

132.1. Connecting to the Eclipse application life cycle

If an application is started or stopped, it typically requires some central setup or shutdown. For example you want to connect to a database or close the database connection.

The Eclipse platform allows you to register a class for predefined events of this life cycle. For example you can use these life cycle hooks to create a login screen or an interactive splash screen before the application is started.

132.2. Accessing application startup parameters

The `IApplicationContext` object which can be injected into your life cycle class contains information about the startup parameters of your application. You can access these parameters via the `getArguments()` method.

The access is demonstrated by the following snippet.

```
// access the command line arguments
String[] args = (String[])
    applicationContext.
        getArguments().
            get(IApplicationContext.APPLICATION_ARGS);
```

A parameter can be a flag or can have a parameter and a value. You can use a method similar to the following to evaluate this.

```
private String getArgValue(String argName, IApplicationContext appContext,
    boolean singledCmdArgValue) {
    // Is it in the arg list ?
    if (argName == null || argName.length() == 0)
        return null;

    if (singledCmdArgValue) {
        for (String arg : args) {
            if ("-" + argName).equals(arg))
                return "true";
        }
        return "false";
    }
    // not a singleCmdArgValue
    for (int i = 0; i < args.length; i++) {
        if ("-" + argName).equals(args[i]) && i + 1 < args.length)
            return args[i + 1];
    }
}
```

132.3. Close static splash screen

If you configured in your product configuration file that a static splash screen should be used, you can call the `applicationRunning()` method on the `IApplicationContext` object. You typically call this method if you want to replace the static splash screen with a dynamic screen developed with SWT.

132.4. How to implement a life cycle class

The `org.eclipse.core.runtime.product` extension point allows you to define a class as life cycle callback via a property.

The key for this property is `lifeCycleURI` and it points to the class via the `bundleclass://` schema.

In the class you can annotate methods with the following annotations. These methods are called by the framework depending on the life cycle of your application.

Table 132.1. Life cycle annotations

Annotation	Description
<code>@PostContextCreate</code>	Is called after the Application's <code>IEclipseContext</code> is created, can be used to add objects, services, etc. to the context. This context is created for the <code>MApplication</code> class.
<code>@ProcessAdditions</code>	Is called directly before the model is passed to the renderer, can be used to add additional elements to the model.
<code>@ProcessRemovals</code>	Same as <code>@ProcessAdditions</code> but for removals.
<code>@PreSave</code>	Is called before the application model is saved. You can modify the model before it is persisted.

132.5. Example life cycle implementation

The following example shows how to register a class in your `plugin.xml` file as life cycle handler. To avoid that the code wraps we use testing as `BundleSymbolicName` and package, you need to replace that with the correct values for your life cycle class.

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

    <extension
        id="product"
        point="org.eclipse.core.runtime.products">
        <product
            name="testing"
            application="org.eclipse.e4.ui.workbench.swt.E4Application">
            <property
                name="appName"
                value="testing">
            </property>
            <property
                name="applicationXMI"
                value="testing/Application.e4xmi">
            </property>
            <property
                name="applicationCSS"
                value="platform:/plugin/testing/css/default.css">
            </property>
            <property
                name="lifeCycleURI"
                value="bundleclass://testing/testing.LifeCycleManager">
            </property>
        </product>
    </extension>
</plugin>
```

The following class displays a `Shell` until the startup process of your application has finished. You could extend this example to show a progress bar.

```
package testing;

import org.eclipse.e4.core.services.events.IEventBroker;
import org.eclipse.e4.ui.workbench.UIEvents;
import org.eclipse.e4.ui.workbench.lifecycle.PostContextCreate;
import org.eclipse.equinox.app.IApplicationContext;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Shell;
import org.osgi.service.event.Event;
import org.osgi.service.event.EventHandler;
```

```
// for a extended example see
// https://bugs.eclipse.org/382224

public class LifeCycleManager {
    @PostContextCreate
    void postContextCreate(final IEventBroker eventBroker, IApplicationContext context) {
        final Shell shell = new Shell(SWT.SHELL_TRIM);

        // register for startup completed event and close the shell
        eventBroker.subscribe(UIEvents.UILifeCycle.APP_STARTUP_COMPLETE,
            new EventHandler() {
                @Override
                public void handleEvent(Event event) {
                    shell.close();
                    shell.dispose();
                    eventBroker.unsubscribe(this);
                }
            });
        // close static splash screen
        context.applicationRunning();
        shell.open();
    }
}
```

Chapter 133. Exercise: Life cycle hook and a login screen

133.1. Target

In this exercise you create a splash screen which allows the user to enter his or her credentials. This assumes that you have created a `PasswordDialog` class which allows the user to enter this user and password.

133.2. Create a new class

The following development is done again in the `com.example.e4.rcp.todo` plug-in.

Create the `com.example.e4.rcp.todo.lifecycle` package and add the following class.

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

import org.eclipse.e4.ui.workbench.lifecycle.PostContextCreate;
import org.eclipse.equinox.app.IApplicationContext;
import org.eclipse.jface.window.Window;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Rectangle;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Monitor;
import org.eclipse.swt.widgets.Shell;

import com.example.e4.rcp.todo.dialogs.PasswordDialog;

// for a extended example see
// https://bugs.eclipse.org/382224
public class Manager {

    @PostContextCreate
    void postContextCreate(IApplicationContext appContext, Display display) {
        final Shell shell = new Shell(SWT.SHELL_TRIM);
        PasswordDialog dialog = new PasswordDialog(shell);

        // close the static splash screen
        appContext.applicationRunning();

        // position the shell
        setLocation(display, shell);

        if (dialog.open() != Window.OK) {
            // close the application
            System.exit(-1);
        }
    }

    private void setLocation(Display display, Shell shell) {
        Monitor monitor = display.getPrimaryMonitor();
        Rectangle monitorRect = monitor.getBounds();
        Rectangle shellRect = shell.getBounds();
        int x = monitorRect.x + (monitorRect.width - shellRect.width) / 2;
        int y = monitorRect.y + (monitorRect.height - shellRect.height) / 2;
        shell.setLocation(x, y);
    }
}
```

Note

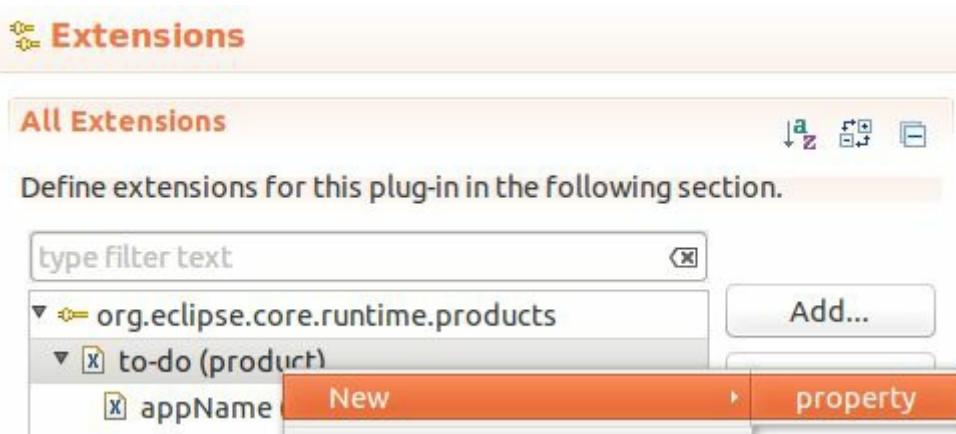
The life cycle annotations are not yet released as official API. See [Section 6.4, “Eclipse API and internal API”.](#)

133.3. Register life cycle hook

Register the Manager class as life cycle handler in your `plugin.xml` file.

For this, open the `plugin.xml` in your application plug-in and select the *Extensions* tab.

Open the `org.eclipse.core.runtime.products` extension, right click on the existing entry and create a new property.



Enter `lifeCycleURI` as name for the property and use `bundleclass://com.example.e4.rcp.todo/com.example.e4.rcp.todo.li` as the value.

Extension Element Details

Set the properties of "property". Required fields are denoted by "*".

name* :	lifeCycleURI
value* :	bundleclass://com.example.e4.rcp.todo/com.example.e4.rcp.todo.lifecycle.Manager

The following snippet shows the resulting `plugin.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
    <extension
        id="product"
        point="org.eclipse.core.runtime.products">
        <product
            name="to-do"
            application="org.eclipse.e4.ui.workbench.swt.E4Application">
            <property
                name="appName"
                value="to-do">
```

```
        </property>
<property
    name="lifeCycleURI"
    value=
"bundleclass://com.example.e4.rcp.todo/com.example.e4.rcp.todo.lifecycle.Manager
</property>
<!-- other properties omitted -->
</product>
</extension>

</plugin>
```

133.4. Validating

Start your application. The login dialog should get displayed and your application should be started after the user selects the *Login* button.

Part XXXI. Handling preferences

Chapter 134. Eclipse preference basics

134.1. Preferences and scopes

The Eclipse platform supports *preferences* for persisting data between application restarts. Preferences are stored as key / value pairs. The key is an arbitrary String. The value can be a boolean, String, int or another primitive type. For example the *user* key may point to the value *vogella*.

The preference support in Eclipse is based on the `Preferences` class from the `org.osgi.service.prefs` package. Eclipse preferences are very similar to standard Java Preferences but use the Eclipse framework to save and retrieve the configuration and support *scopes*.

The scope defines how the preference data is stored and how it is changeable. The Eclipse runtime defines three scopes. The different scopes are explained in the following table.

Table 134.1. Eclipse Preference scope

Scope	Description
Instance scope	Preferences in this scope are specific to a single Eclipse workspace. If the user runs the same program twice with different workspaces, the settings between the two programs may be different.
Configuration scope	If the user runs the same program twice, then the settings between the two programs are identical. Preferences stored in this scope are shared by all workspaces that are launched using a particular configuration of Eclipse plug-ins.
Default scope	Default values which can not be changed. This scope is not stored on disk at all but can be used to store default values for all your keys. Supplied via configuration files in plug-ins and product definitions.
BundleDefaultsScope	Similar to the default scope, these values are not written to disk. They are read from a particular bundle's <code>preferences.ini</code> file.

134.2. Storage of the preferences

Eclipse stores the preferences in the workspace of your application in the `.metadata/.plugins/org.eclipse.core.runtime/.settings/` directory in the `<nodePath>.prefs` file.

The `<nodePath>` is by default the Bundle-SymbolicName of the plug-in but can be specified via the preference API. The workspace is by default the directory in which the application starts.

You can configure the storage location of the preferences via the `-data path` launch parameter in Eclipse. To place the preferences in the user home directory use the `-data @user.home` parameter setting.

134.3. Eclipse preference API

You can create and manipulate preferences directly via Singletons provided by the Eclipse runtime. You have the `InstanceScope`, `ConfigurationScope` and `DefaultScope` classes which give access to the corresponding instance via the `INSTANCE` field.

Preference values are read and saved by `get()` and `put()` methods. In the `get()` method you specify a default value in case the key can not be found. The `clear()` method removes all preferences and the `remove()` method allows you to delete a selected preference value. Via the `flush()` method you persist the preferences to the file system.

```
// We access the instanceScope
Preferences preferences = InstanceScope.INSTANCE
    .getNode("com.vogella.eclipse.preferences.test");

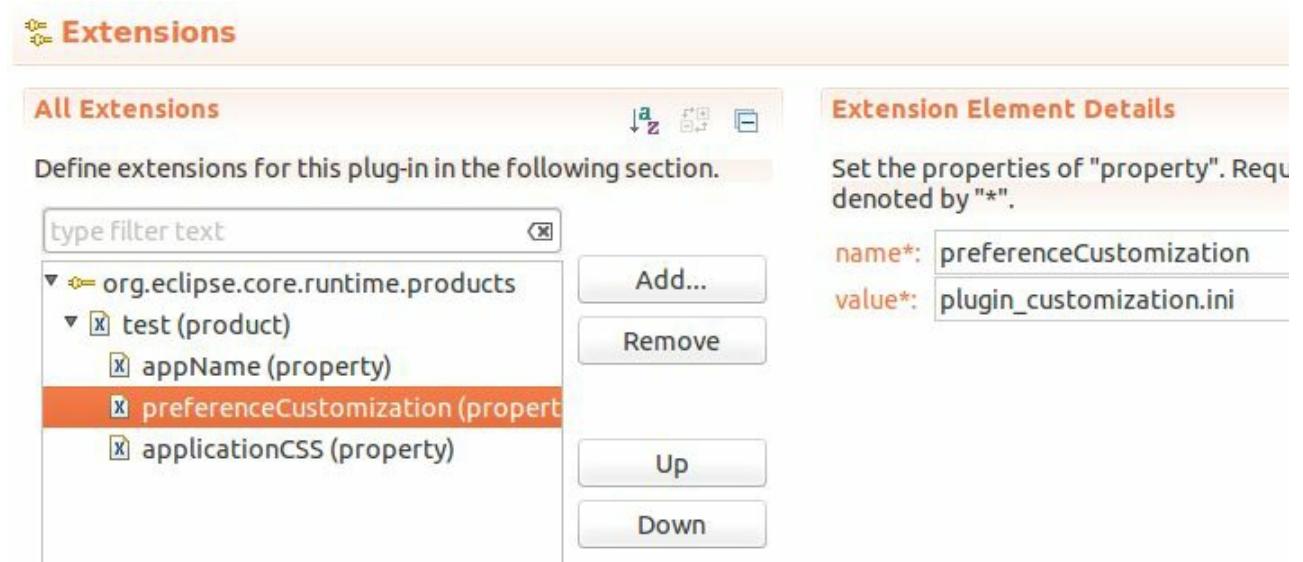
Preferences sub1 = preferences.node("node1");
Preferences sub2 = preferences.node("node2");
sub1.put("h1", "Hello");
sub1.put("h2", "Hello again");
sub2.put("h1", "Moin");
try {
    // forces the application to save the preferences
    preferences.flush();
} catch (BackingStoreException e) {
    e.printStackTrace();
}
}

// read values from the instance scope
Preferences preferences = InstanceScope.INSTANCE
    .getNode("com.vogella.eclipse.preferences.test");
Preferences sub1 = preferences.node("node1");
Preferences sub2 = preferences.node("node2");
sub1.get("h1", "default");
sub1.get("h2", "default");
sub2.get("h1", "default");
```

134.4. Setting preferences via `plugin_customization.ini`

You can use a file to set the default values of preferences. The file which contains these defaults is typically named `plugin_customization.ini`.

Such a file needs to be registered via the `preferenceCustomization` property on the product extension point in the `plugin.xml` file. This is demonstrated in the following screenshot.



The format to use is `<plugin id>/<setting>=<value>`, e.g. `com.example.e4.rcp.todo/user=vogella`.

Tip

To find the correct keys, just start your Eclipse application, switch to the `.metadata` directory in your workspace directory (by default the directory your application is starting in) and search for files ending with `.pref`.

Chapter 135. Preferences and dependency injection

135.1. Preferences and dependency injection

The Eclipse platform allows you to use dependency injection for preferences handling.

Preference values are not stored directly in the `IEclipseContext` but contributed by an *ExtendedObjectSupplier*. This implies that you have to use a special annotation `@Preference` to access them. The `@Preference` must be used together with `@Inject` or one of the other annotations which implies dependency injection, e.g. `@Execute`.

The `@Preference` annotation allows you to specify the `nodePath` and the `value` as optional parameters.

The `nodePath` is the file name used to save the preference values to disk. By default this is the Bundle-SymbolicName of the plug-in. The `value` parameter specifies the preference key for the value which should be injected.

Eclipse can also inject the `IEclipsePreference` object. You can use this object for storing values. If you use the `value` parameter, Eclipse injects the value directly. Use the `value` parameter for read access, while for storing or changing values, use the `IEclipsePreference` object.

Warning

For accessing the `IEclipsePreference` object you still have to use the `@Preference` annotation. `@Inject` alone is not sufficient.

The following code snippet demonstrates how to put values into the preferences store.

```
// get IEclipsePreferences injected to change a value
@Execute
public void execute
    (@Preference(nodePath = "com.example.e4.rcp.todo") IEclipsePreferences prefs)
    // more stuff...
    prefs.put("user", "TestUser");
    prefs.put("password", "Password");
    // Persists
    try {
        prefs.flush();
```

```

} catch (BackingStoreException e) {
    e.printStackTrace();
}
}

```

The next snippet demonstrates the read access of preference values.

```

@Inject
@Optional
public void trackUserSettings
    (@Preference(nodePath = "com.example.e4.rcp.todo",
    value = "user")
    String user) {
    System.out.println("New user: " + user);
}

@Inject
@Optional
public void trackPasswordSettings
    (@Preference(nodePath = "com.example.e4.rcp.todo",
    value = "password")
    String password) {
    System.out.println("New password: " + password);
}

```

The Eclipse platform automatically tracks the values and re-injects them into fields and methods if they change. Eclipse tracks changes of preferences in the `InstanceScope` scope. Preference values in the `ConfigurationScope` and `DefaultScope` are not tracked.

If you use the injected `IEclipsePreference` to store new preference values, these values are stored in the instance scope.

135.2. Persistence of part state

Eclipse provides the `@PersistState` annotation. This annotation can be applied to a method in a class referred to a part.

Such an annotated method can be used to store the instance state of the part. The Eclipse framework calls such a method whenever the part or the application closes. The stored information can be used in the method annotated with the `@PostConstruct` annotation. A typical use case for such a method would be to store the state of a checkbox.

The usage of this annotation is demonstrated in the following example code.

```
@PostConstruct
public void createControl(MPart part) {
    Map<String, String> state = part.getPersistedState();
    String value = state.get("key");
    ...
}

@PersistState
public void persistState(MPart part) {
    Map<String, String> state = part.getPersistedState();
    state.put("key", "newValue");
    ...
}
```

Chapter 136. Exercise: Using preferences in the life cycle class

136.1. Target

In this exercise you store the last logged in user name in the preferences.

You continue to use the dialog which you created in [Section 85.2, “Create a password dialog”](#). In this exercise you modify your life cycle handler so that the previously entered user is displayed again in the password dialog.

136.2. Dependency

The `@Preference` annotation is provided by the `org.eclipse.e4.core.di.extensions` plug-in. Ensure that you have entered a dependency to this plug-in in the `MANIFEST.MF` file of your application plug-in. If you followed the previous exercises correctly this dependency is already entered.

136.3. Create interface for the preference constants

Create the `com.example.e4.rcp.todo.preferences` package and the following interface in this package.

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

public interface PreferenceConstants {
    public static final String NODEPATH = "com.example.e4.rcp.todo";
    public static final String USER_PREF_KEY = "user";
    public static final String PASSWORD_PREF_KEY = "password";
}
```

136.4. Using preferences in the life cycle class

Tip

This exercise is optional.

Extend the Manager class to get the user preference injected during application startup.

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

import javax.inject.Inject;

import org.eclipse.core.runtime.preferences.IEclipsePreferences;
import org.eclipse.e4.core.di.extensions.Preference;
import org.eclipse.e4.ui.workbench.lifecycle.PostContextCreate;
import org.eclipse.equinox.app.IApplicationContext;
import org.eclipse.jface.window.Window;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Rectangle;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Monitor;
import org.eclipse.swt.widgets.Shell;
import org.osgi.service.prefs.BackingStoreException;

import com.example.e4.rcp.todo.dialogs.PasswordDialog;
import com.example.e4.rcp.todo.preferences.PreferenceConstants;

public class Manager {

    // example uses the optional nodePath parameter
    // allows to move the handler to another plug-in
    @Inject
    @Preference(nodePath = PreferenceConstants.NODEPATH,
        value = PreferenceConstants.USER_PREF_KEY)
    private String user;

    @PostContextCreate
    public void postContextCreate(@Preference IEclipsePreferences prefs,
        IApplicationContext appContext, Display display) {

        final Shell shell = new Shell(SWT.SHELL_TRIM);
        PasswordDialog dialog = new PasswordDialog(shell);
        if (user != null) {
            dialog.setUser(user);
        }

        // close the static splash screen
        appContext.applicationRunning();

        // position the shell
    }
}
```

```

        setLocation(display, shell);

        // open the dialog
        if (dialog.open() != Window.OK) {
            // close the application
            System.exit(-1);
        } else {
            // get the user from the dialog
            String userValue = dialog.getUser();
            // store the user values in the preferences
            prefs.put(PreferenceConstants.USER_PREF_KEY, userValue);
            try {
                prefs.flush();
            } catch (BackingStoreException e) {
                e.printStackTrace();
            }
        }
    }

private void setLocation(Display display, Shell shell) {
    Monitor monitor = display.getPrimaryMonitor();
    Rectangle monitorRect = monitor.getBounds();
    Rectangle shellRect = shell.getBounds();
    int x = monitorRect.x + (monitorRect.width - shellRect.width) / 2;
    int y = monitorRect.y + (monitorRect.height - shellRect.height) / 2;
    shell.setLocation(x, y);
}
}

```

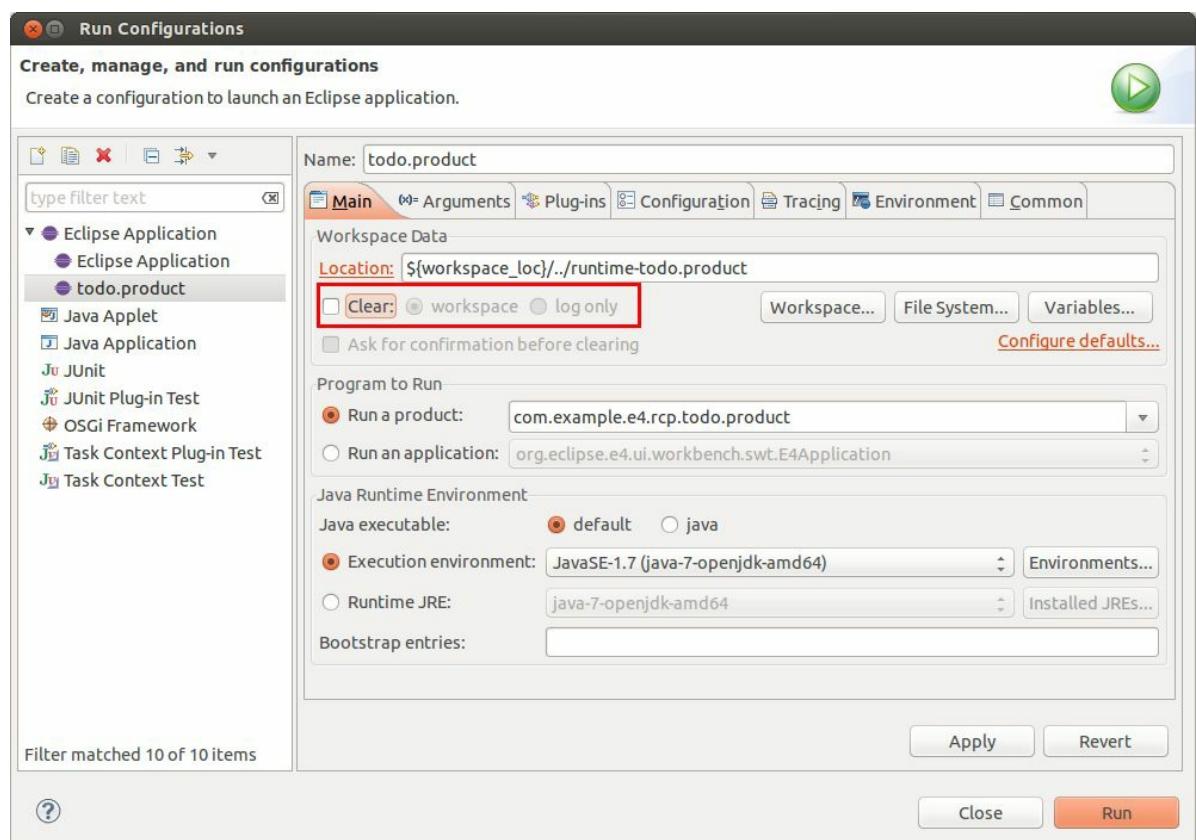
136.5. Validate life cycle handling

Start your application and enter a user. Stop and the application and start the application again. The login dialog should display the last entered user.

Tip

If persisting the user does not work, you potentially have an incorrect setting in your launch configuration.

Ensure that the *Clear* flag in the Project → Run Configurations... is not set. Otherwise your preference data is deleted during the start of the application.



Chapter 137. Exercise: Using preferences in a handler and in a part

137.1. Target

In this exercise you modify the handler within the application to enter the user and password and listen to changes in the corresponding preference values in a part.

Note

In a real application you would not store the password without any encryption on the file system.

137.2. Using preferences in your handler

Extend your `EnterCredentialsHandler` handler to get and store the user and password via the preference values.

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

import javax.inject.Inject;

import org.eclipse.core.runtime.preferences.IEclipsePreferences;
import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.core.di.extensions.Preference;
import org.eclipse.jface.window.Window;
import org.eclipse.swt.widgets.Shell;
import org.osgi.service.prefs.BackingStoreException;

import com.example.e4.rcp.todo.dialogs.PasswordDialog;

public class EnterCredentialsHandler {

    @Inject
    @Preference(nodePath = PreferenceConstants.NODEPATH,
               value = PreferenceConstants.USER_PREF_KEY)
    String userPref;

    @Inject
    @Preference(nodePath = PreferenceConstants.NODEPATH,
               value = PreferenceConstants.PASSWORD_PREF_KEY)
    String passwordPref;

    @Execute
    public void execute(Shell shell, @Preference IEclipsePreferences prefs) {
        PasswordDialog dialog = new PasswordDialog(shell);

        if (userPref != null) {
            dialog.setUser(userPref);
        }

        if (passwordPref != null) {
            dialog.setPassword(passwordPref);
        }

        // get the new values from the dialog
        if (dialog.open() == Window.OK) {
            prefs.put(PreferenceConstants.USER_PREF_KEY, dialog.getUser());
            prefs.put(PreferenceConstants.PASSWORD_PREF_KEY,
                      dialog.getPassword());
            try {
                prefs.flush();
            } catch (BackingStoreException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
    }  
}  
  
}
```

Note

As `nodePath` is not explicitly set, it uses `com.example.e4.rcp.todo` which is the Bundle-SymbolicName of your application plug-in.

Note

The `@Preference` annotation is not yet released as official API. See [Section 6.4, “Eclipse API and internal API”](#).

For test purpose listen to preference changes in your playground view and print the values to the console.

```
@Inject  
@Optional  
public void trackUserSettings(@Preference(value = "user") String user) {  
    System.out.println("New user: " + user);  
}  
  
@Inject  
@Optional  
public void trackPasswordSettings(@Preference(value = "password") String password)  
{  
    System.out.println("New password: " + password);  
}
```

137.3. Validate menu entry

Start your application. Use the menu entry which uses the `EnterCredentialsHandler` handler and enter a user into the dialog. Close the dialog and open it again via the menu. The last entered user should be displayed.

Part XXXII. Internationalization

Chapter 138. Internationalization and localization in Eclipse

138.1. Translation of a Java applications

The process of preparing an application for being translated into several languages is called *internationalization*. This term is typically abbreviated to *i18n*.

The process of translating the application is called *localization* and is abbreviated to *l10n*.

138.2. Property files

Java applications are typically translated via *property* files. Such a file contains key/values. The key can be used in your application and is substituted at runtime with the corresponding value. The following listing demonstrates an example content of such a file.

```
#Properties file
part.overview=Overview
part.detail=Details
```

Based on the language of the user, the Java runtime searches for the corresponding resource bundle using language identifiers. If a certain language identifier is not provided, the Java runtime will fall back to the next general resource bundle.

For example:

- messages.properties: default language file, if nothing else is available
- messages_de.properties: used for German
- messages_en.properties: default for English
- messages_en_US.properties: US English file
- messages_en_UK.properties: British English file

The Eclipse platform provides functionality to support translations based on properties files but also allows the usage of alternative approaches.

138.3. Encoding of property files in Java

Resource bundles in Java are always LATIN-1 (ISO 8859-1) encoded. This is defined by the Java specification.

138.4. Relevant files for translation in Eclipse applications

The following table lists the relevant elements for translating an Eclipse RCP application.

Table 138.1. Translation relevant entities for Eclipse plug-ins

Entity	Description
application model (Application.e4xmi and model fragment files)	Describes the application model in Eclipse RCP.
plugin.xml	Primarily important for Eclipse 3.x based plug-ins.
Source Code	The source code contains text, e.g., labels which must be translated.

138.5. Where to store the translations?

It is not uncommon that a plug-in also contains its translations.

If a translation should be used by several plug-ins it is good practice to have one central plug-in which contains the translations. This plug-in contains at least the main language and potentially more.

It is also possible to provide translation files via fragment projects. Fragment projects extend their host plug-in. This approach allows that the text files can be maintained in separate plug-ins, which is sometimes easier to handle for the translation team. This approach also allows configuring the included languages via the product configuration file.

138.6. Setting the language in the launch configuration

By default Eclipse uses the language configured in the operating system of the user.

For testing you can set the language manually. In your Eclipse launch configuration on the *Arguments* tab you can specify the runtime parameter `-nl` to select the language, e.g. `-nl en`.

138.7. Translation service

Eclipse uses a translation service via the `TranslationService` interface. The default implementation of this class is the `BundleTranslationProvider` which uses property files as input for the translations. But you can provide your own OSGi service which uses a different source, e.g., a database to get the translations.

This service is stored in the application context. You can replace it there, e.g., via a life cycle hook or via a model-addon which modifies the application context once the application startup is completed.

By storing it in the `IEclipseContext` context hierarchy you can have different translation services for different local contexts, e.g., for different windows you can use different translation services.

Chapter 139. Translating the application model and plugin.xml

139.1. OSGi resource bundles

For the translation of the plugin.xml and application model files the Eclipse runtime uses by default OSGi resource bundles. These are property files in a specified location. OSGi expects at least one resource bundle in this location.

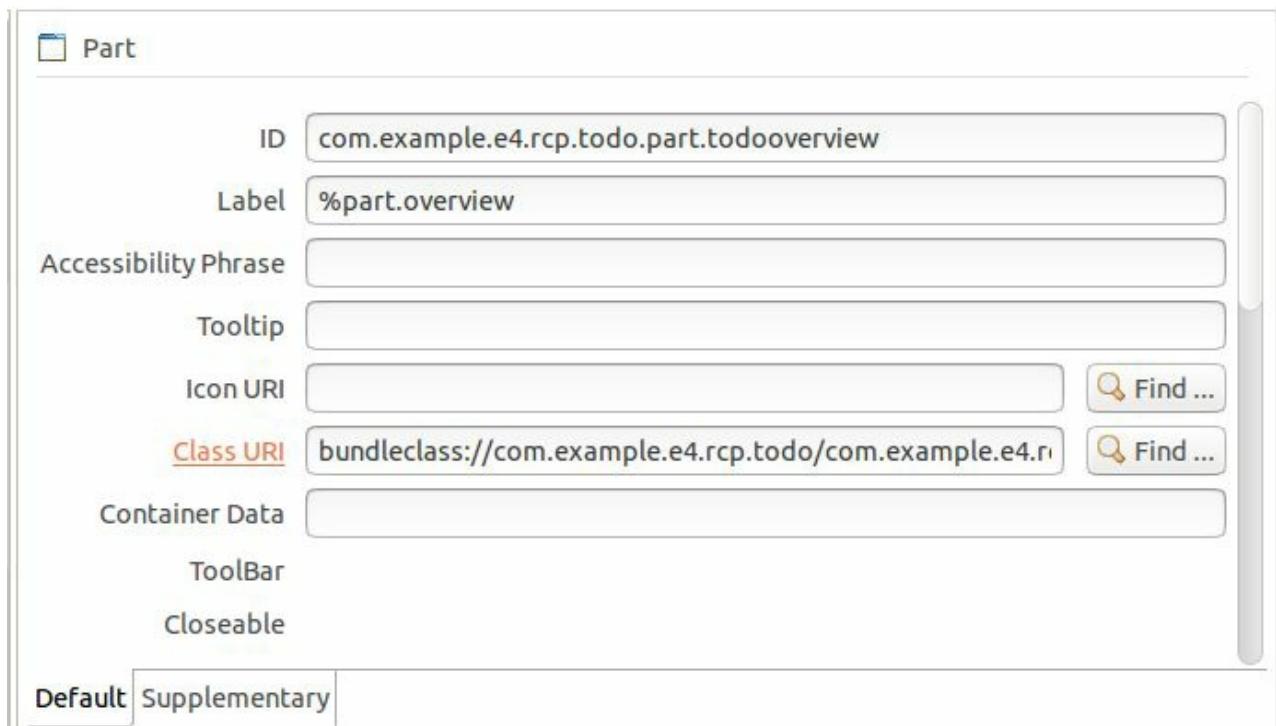
OSGI-INF/110n/bundle is the default location and file prefix which the Eclipse translation service is using. Via the *Bundle-Localization* attribute in the manifest file, you can specify an alternative location for the bundle resources. The usage of this attribute is demonstrated in the following listing.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: I18n
Bundle-SymbolicName: de.vogella.rcp.i18n; singleton:=true
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: de.vogella.rcp.i18n.Activator
Require-Bundle: org.eclipse.ui,
    org.eclipse.core.runtime
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Bundle-Localization: OSGI-INF/anotherlocation/bundle
```

Alternative languages are defined via additional property files. The filename of these files can include a language and optional a country variant as described in [Section 138.2, “Property files”](#). For example *bundle_en.properties* or *bundle_en_UK.properties*.

139.2. Translating the application model

The translation key can be specified in your application model via the `%key` reference. This usage is demonstrated via the following screenshot of the model data for a part.



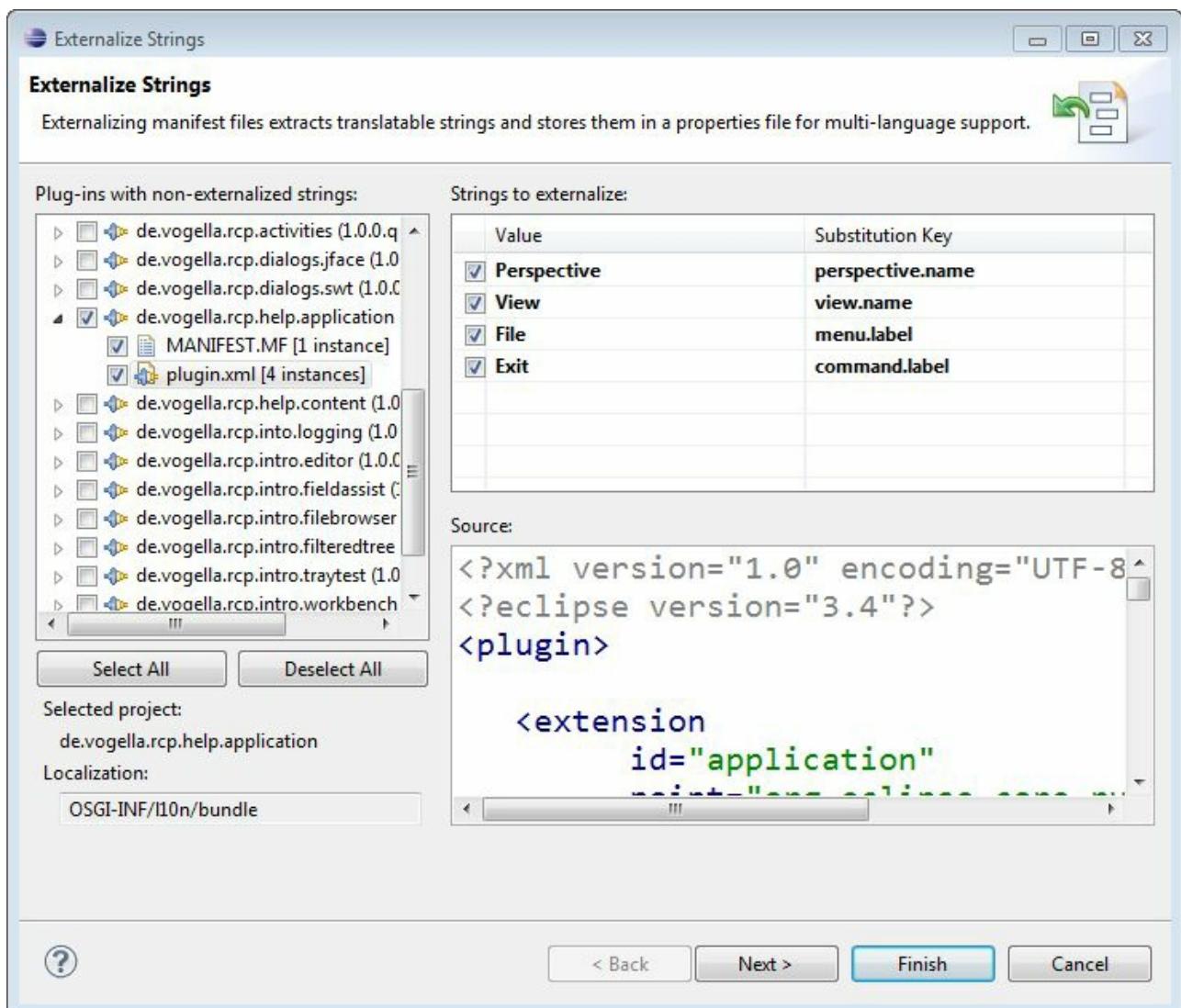
Currently the application model tooling includes rudimentary support for extracting Strings. Press the right mouse button on an empty space in the application model editor and select *Externalize Strings*.

Warning

This action extracts all Strings from the application model. If you want to replace only a set of relevant Strings, you have to do this manually. If you do it manually you can use the PDE wizard as described in [Section 139.3, “Translating plugin.xml”](#) to create the `bundle.property` file and the OSGi reference.

139.3. Translating plugin.xml

Similar to the usage of keys in the application model, you can use the key prefixed with % in the *plugin.xml* file. The Eclipse tooling supports extracting existing String values from the *plugin.xml* file. Select your *plugin.xml* file, right-click on it and select Plug-in Tools → Externalize Strings.



Via PDE Tools → Internationalize you can also directly create one or several fragments for the result of the internationalization.

Chapter 140. Source code translation with the Eclipse translation service

140.1. Translation with POJOs

To define your translations you simply define a Java object with several public fields of type String.

```
public class Messages {  
    public String labelSummary;  
    public String labelDescription;  
    public String labelDone;  
    public String labelDueDate;  
    public String labelWithPlaceholder = {0} says {1}  
}
```

The fields of such a Java object are initialized when an instance is injected via the `@Translation` annotation. The `org.eclipse.e4.core.services` plug-in contains the annotations which are used for translations.

```
@Inject  
@Translation  
Messages messages;  
  
// more code  
myLabel.setText(messages.label_message);
```

To find the property files for the key/value pairs, it searches for property files based on the rules described in the next section.

140.2. Search process for translation files

The default translation service searches for property files with the translations in the following order.

1. Check the `@Message` annotation in the `Messages` class for the correct location. This annotation allows defining the location of the file, see [the section called “Using the optional `@Message` annotation”](#). If not present, go to the next step.
2. Check if a property file with a corresponding name is available relatively to the `Message` class. If not present, go to the next step.
3. Check if a property file is configured via the `OSGi-ResourceBundle` header in the `MANIFEST.MF` file. resource bundle. If this entry is not present check the `OSGI-INF/110n` folder otherwise use the entered location. Such a property file is called OSGi resource bundle (see [Section 139.1, “OSGi resource bundles”](#)).

Using the optional `@Message` annotation

The optional `@Message` annotation can be used to define the location of the resource bundle. It allows also to define how and if translations should be cached by the Eclipse platform. See the Javadoc of the `@Message` annotation for more information on this topic.

Using the optional `@PostConstruct` in message POJOs

It is possible to initialize fields via the `@PostConstruct` method in a message object as demonstrated in the following snippet.

```
public class Messages {  
    public String labelWithPlaceholder1 = {0} says {1}  
    public String labelWithPlaceholder2 = {0} says {1}  
  
    // initialize the first message  
  
    @PostConstruct  
    public void format() {  
        labelWithPlaceholder2 =  
            MessageFormat.  
                format(labelWithPlaceholder2, "Test", "this is fun.");  
    }  
}
```

Note

`@PostConstruct` does not support parameter injection in the context of a message object.

140.3. Dynamic language switch

The currently active locale, e.g., "de" or "en", is represented by an object of type `Locale`. The Eclipse runtime supports a dynamic switch of this locale at runtime.

Note

The type of the `TranslationService.LOCALE` key was changed from `String` to `Locale` in Eclipse 4.4.2.

To support a dynamic locale switch the application must be able to react on changes in the messages. This requires that the widgets are declared as public fields and that a method exists which updates them. This is called the *Eclipse Translation Pattern*.

```
public class TranslationExamplePart {  
    private Label myLabel;  
  
    @PostConstruct  
    public void postConstruct(Composite parent, @Translation Messages messages) {  
        // create the UI  
        myLabel = new Label(parent, SWT.NONE);  
        // as @PostConstruct is executed after field injection, we need to  
        // call translate here manually to fill the UI initially  
        translate(messages);  
    }  
  
    // the method that will perform the dynamic locale changes  
    @Inject  
    public void translate(@Translation Messages messages) {  
        if (myLabel != null && !myLabel.isDisposed())  
            myLabel.setText(messages.labelSummary);  
    }  
}
```

To change the active locale at runtime use the `ILocaleChangeService` service. This is demonstrated by the following class which could be used as part of a tool control.

```
public class SwitchLanguageToolControl {  
  
    Button button;  
  
    @Inject  
    ILocaleChangeService lcs;  
  
    @PostConstruct
```

```

public createPartControls(Composite parent) {

    final Text input = new Text(parent, SWT.BORDER);

    input.addKeyListener(new KeyAdapter() {
        @Override
        public void keyPressed(KeyEvent event) {
            if (event.keyCode == SWT.CR
                || event.keyCode == SWT.KEYPAD_CR) {
                lcs.changeApplicationLocale(input.getText());
            }
        }
    });

    button = new Button(parent, SWT.PUSH);
    button.addSelectionListener(new SelectionAdapter() {
        @Override
        public void widgetSelected(SelectionEvent e) {
            lcs.changeApplicationLocale(input.getText());
        }
    });
}

@Inject
public void translate(@Translation Messages messages) {
    // button localization via Eclipse Translation Pattern
    button.setText(messages.button_change_locale);
}
}

```

To get notified you can use dependency injection.

```

@Inject
@Optional
private void getNotified(@Named(TranslationService.LOCALE) Locale s) {
    System.out.println("Injected via context: " + s);
}

@Inject
@Optional
private void getNotified(@UIEventTopic(ILocaleChangeService.LOCALE_CHANGE) L
    System.out.println("Injected via event broker: " + s);
}

```

Chapter 141. Source code translation with NLS support

141.1. NLS compared to the Eclipse translation service

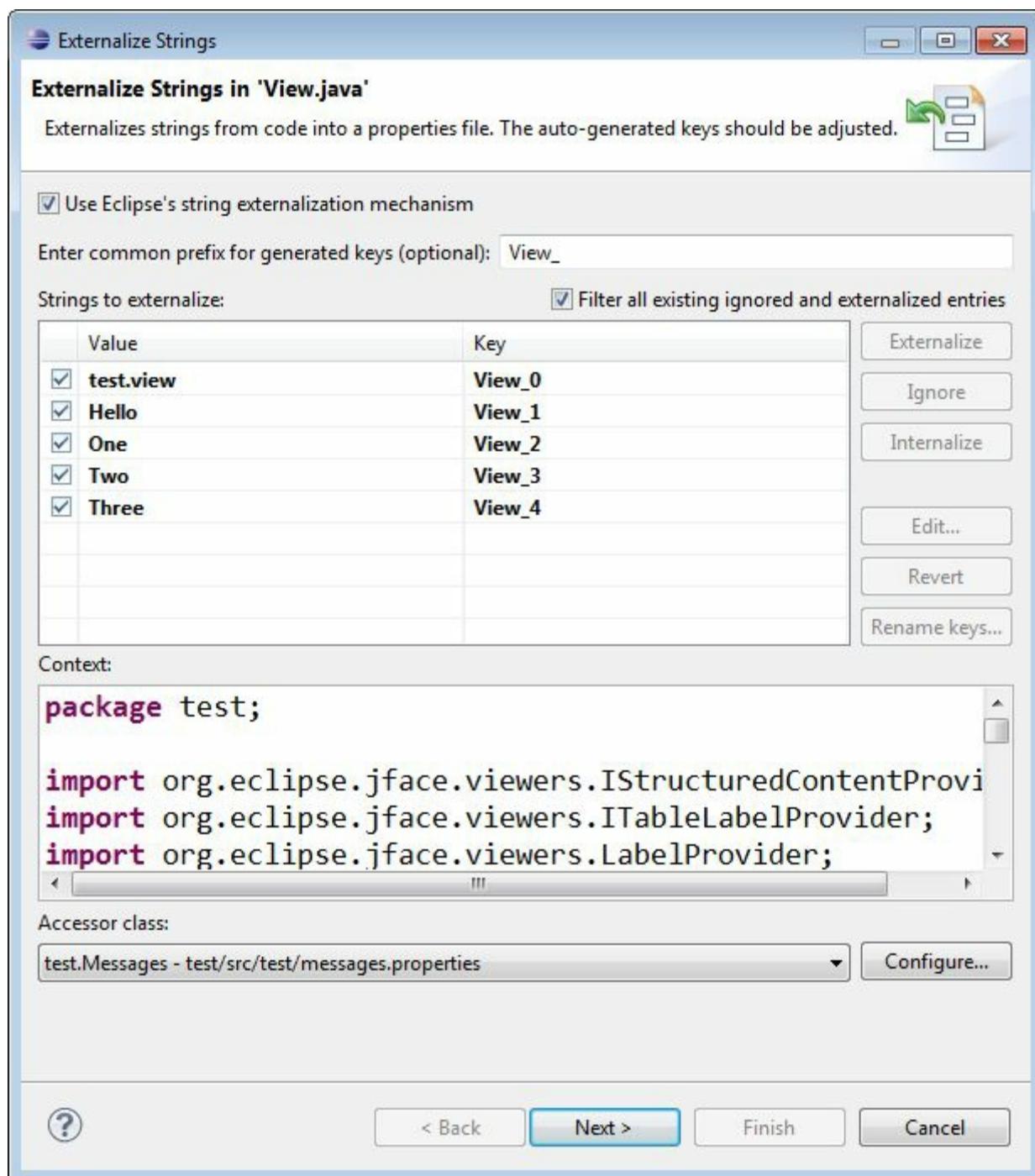
Warning

This part of the description is included for completeness. It describes the NLS approach for translation. This approach still works but the Eclipse translation mechanism provides more flexibility, e.g., dynamic switching of languages, less memory consumption and a translation service which can be exchanged. This superior approach is described in [Section 140.1, “Translation with POJOs”](#).

141.2. Translating your custom code

It is possible to translate the Java source with two different approaches, based on Strings and based on constants. The approach based on constants is more reliable and should be preferred. To enable this support you need to configure the `org.eclipse.core.runtime` plug-in as dependency in your related plug-in.

To translate Strings in the source code, select the file you want to translate and select Source → Externalize Strings.

A screenshot of the "Externalize Strings" dialog box. The title bar says "Externalize Strings". The main area is titled "Externalize Strings in 'View.java'". It contains a message: "Externalizes strings from code into a properties file. The auto-generated keys should be adjusted." with a "Filter" icon. A checked checkbox "Use Eclipse's string externalization mechanism" is present. An optional prefix "Enter common prefix for generated keys (optional): View_" is shown. A table lists strings to externalize with their corresponding keys:

Value	Key
<input checked="" type="checkbox"/> test.view	View_0
<input checked="" type="checkbox"/> Hello	View_1
<input checked="" type="checkbox"/> One	View_2
<input checked="" type="checkbox"/> Two	View_3
<input checked="" type="checkbox"/> Three	View_4

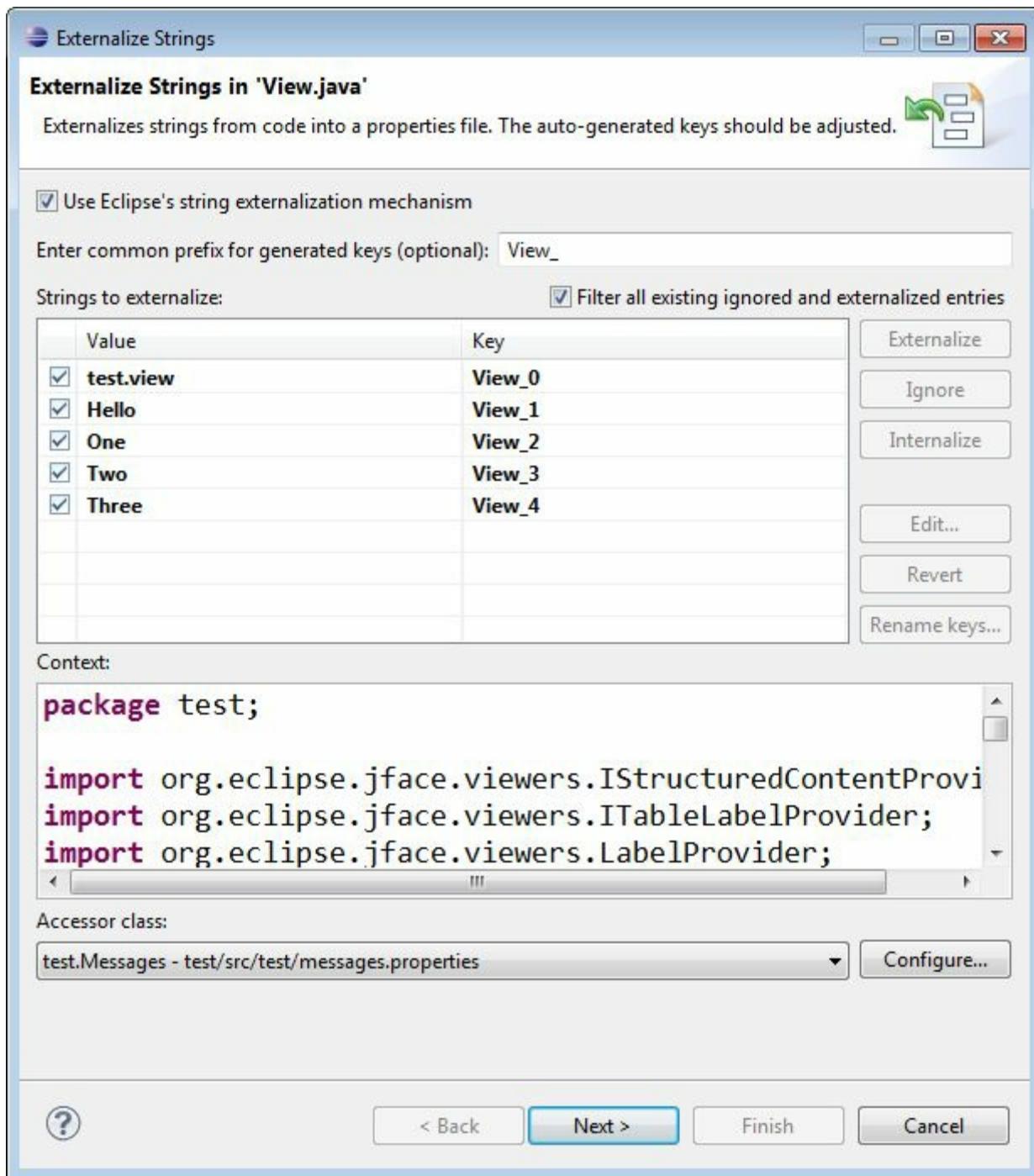
Buttons on the right include "Externalize", "Ignore", "Internalize", "Edit...", "Revert", and "Rename keys...". A "Filter all existing ignored and externalized entries" checkbox is also present. Below the table is a "Context:" section containing Java code:

```
package test;

import org.eclipse.jface.viewers.IStructuredContentProvi
import org.eclipse.jface.viewers.ITableLabelProvider;
import org.eclipse.jface.viewers.LabelProvider;
```

Accessories class is set to "test.Messages - test/src/test/messages.properties" with a "Configure..." button. At the bottom are buttons for "?", "< Back", "Next >" (highlighted in blue), "Finish", and "Cancel".

Please note that the *Use Eclipse's string externalization mechanism* option is only visible if you have the `org.eclipse.core.runtime` plug-in configured as a dependency in your plug-in.



In this wizard you can select which Strings should be translated, which should be skipped and which should be marked as not translatable.

If you select that a String should not be translated, Eclipse marks the occurrence with a `$NON-NLS` comment in the source code.

As the result a `Messages` class is generated which serves as an access point for the properties file.

```

package test;

import org.eclipse.osgi.util.NLS;

public class Messages extends NLS {
    private static final String BUNDLE_NAME
        = "test.messages"; //NON-NLS-1$
    public static String View_0;
    public static String View_1;
    static {
        // initialize resource bundle
        NLS.initializeMessages(BUNDLE_NAME, Messages.class);
    }

    private Messages() {
    }
}

```

In this example the `Messages` class uses a constant called `BUNDLE_NAME` to point to the `message*.properties` file in the `test` package. * is a placeholder for your locale, e.g., `_de`, `_en`, etc.

```

View_0=test.view
View_1=Hello

```

In your code you access the translations via the `Message` class.

```
label.setText(Messages.View_1);
```

You can also use placeholders in the messages and evaluate them with the `NLS.bind()` method.

```

MyMessage = {0} says {1}

// NLS bind will call the toString method on obj
NLS.bind(Message.MyMessage, obj, obj);

```

If translations should be used over several plug-ins to ensure consistency, it is good practice to create separate plug-ins or fragments for the translations. All plug-ins which want to use the translations define a dependency to the corresponding plug-in.

Additional languages are typically contributed via Eclipse fragment projects to this message plug-in.

141.3. Translating SWT and JFace code

The SWT and JFace plug-ins include their resource translation files. You can change these default texts and their translations because resource bundles have an override support built in.

If you supply a different file of the SWT or JFace translation bundles in a fragment or plug-in (such as `SWTMessage_de.properties`), then the most local bundle matching the current locale is used.

To change the text supplied by SWT and JFace do the following:

- Create two fragments projects, which will hold the translations. One with `org.eclipse.jface` and one with `org.eclipse.swt` as Host-Plug-in.
- Create the `org.eclipse.jface` and `org.eclipse.swt.internal` packages in the corresponding fragments.
- Copy the `org/eclipse/jface/message.properties` file from the `org.eclipse.jface` plug-in and the `org/eclipse/swt/internal/SWTMessage.properties` file from the SWT plug-in to the corresponding packages in your project. See the appendix for the links to their location in the Eclipse Git repository.
- Provide the files with your desired language extensions, e.g. `message_de.properties`.
- Edit the files to contain your text properties.

Warning

The original `message.properties` in JFace or `SWTMessage.properties` in SWT cannot be overridden by a fragment. Therefore, you can only provide additional `*.properties` files for a certain locale like `message_de.properties`.

You can translate selected properties of the original `message.properties` file. The following listing shows how you would override the buttons of a JFace Wizard class in a `message_de.properties` file.

```
▼ src
  ▼ org.eclipse.jface
    messages_de.properties
    messages_en.properties
```

Chapter 142. Exporting and common problems with translations

142.1. Exporting Plug-ins and Products

The *build.properties* file in a plug-in defines which of the files are included in the exported product.

You must include your property files in the *build.properties* file, otherwise your translations will be missing in the exported product.

142.2. Common problems with i18n

- If the translations are not available in the exported product, ensure that the property files are included in the export. You do this via the *build.properties* file of the plug-in which provides the property files.
- If the translations are not displayed in the application, select *Clear Configuration* in your launch configuration. OSGi caches text information sometimes.

Chapter 143. Optional Exercise: Internationalization for the application model

143.1. Target

In this exercise you translate the relevant parts of your application model and use the translation service.

143.2. Create the translations for the application model

Create the `OSGI-INF/110n` folder in your `com.example.e4.rcp.todo` plug-in.

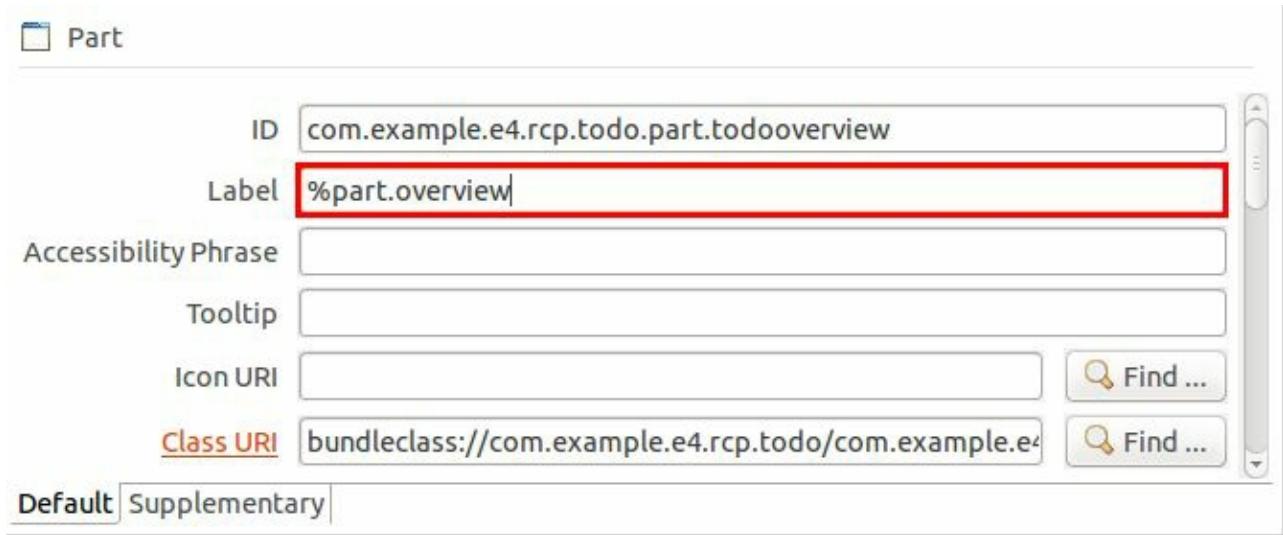
Create the `bundle.properties` file with key/values for the parts in your application model in this folder. The file might look like the following.

```
part.overview=Overview  
part.details=Details  
command.save=Save
```

Create at least one additional translation file for another language. For example, if English is your default language, create the `bundle_de.properties` file to provide a German translation. Ensure that the translation values are different.

143.3. Translate the application model

Use the defined keys with the % prefix in your application model. The following screenshot demonstrates that for a part.



143.4. Test the translation of the application model

Ensure that the application works well for the languages you provided translations for. You can set the language in your launch configuration by using the `-nl` start parameter as described in [Section 138.6, “Setting the language in the launch configuration”](#)

Warning

You have to change the launch configuration. Setting the language via the product file does not work, as the `nl` parameter is currently not correctly transferred to the launch configuration. See [PDE Bug report](#) for details.

143.5. Application model and translations

By default, it is not possible to define that the translations for the application model are located in another plug-in. If you want this, you have to implement your custom translation service.

Chapter 144. Optional exercise: Internationalization for the source code

144.1. Target

In this exercise you translate the relevant parts of your application source code via the Eclipse 4 translation service.

144.2. Creating a plug-in to host the translations

Create a new simple plug-in called `com.example.e4.rcp.todo.i18n`. Add the dependency `org.eclipse.core.runtime` to the new plug-in in its `MANIFEST.MF` file.

144.3. Create a Message class for the source code translations

Create a `Messages` class which contains your translation keys. This might look like the following (keys are just examples).

```
package com.example.e4.rcp.todo.i18n;

public class Messages {
    public String buttonLoadData;
    public String txtSummary;
    public String txtDescription;
}
```

144.4. Create the translations for the source code

Create the `OSGI-INF/110n` folder in your `com.example.e4.rcp.todo.i18n` plug-in.

Create a `bundle.properties` file with key/values for the text in your source code. The file might look like the following.

```
button_load_data=Load Data  
txt_summary=Summary  
txt_description>Description
```

Create one additional translation file for another language, e.g., via a `bundle_de.properties` file. Ensure that the translation values are different.

Note

The translations for the application model as created in [Chapter 143, *Optional Exercise: Internationalization for the application model*](#) cannot be moved to this new plug-in. See [Section 143.5, “Application model and translations”](#) as reminder.

144.5. Export the translations as API

Publish the `com.example.e4.rcp.todo.i18n` package as API via the `MANIFEST.MF` file of the translation plug-in.

144.6. Define dependencies to the translation plug-in

Open the *MANIFEST.MF* file of the `com.example.e4.rcp.todo` plug-in. Add a dependency to the `com.example.e4.rcp.todo.i18n` plug-in.

144.7. Update the product (via the feature)

Add the `com.example.e4.rcp.todo.i18n` plug-in to your feature project. This includes it in your product.

Start your application via the product configuration file and validate that the application still starts and works correctly.

144.8. Using @Translation to get the messages injected

Use dependency inject to the translations injected. Prepare your TodoOverviewPart for a dynamic language switch.

```
@PostConstruct
public void createControls(Composite parent,
    EMenuService menuService, @Translation Messages message) {

    // more source code

    // new
    translateTable(message);

}

// more source code

@Inject
public void translateTable(@Translation Messages message) {
    if (viewer !=null && !viewer.getTable().isDisposed()) {
        colSummary.getColumn().setText(message.todo_summary);
        colDescription.getColumn().setText(message.todo_description);
    }
}
```

144.9. Test your translation

Repeat the test as described in [Section 143.4, “Test the translation of the application model”](#) but this time ensure that your source code is correctly translated.

Part XXXIII. Custom annotations and ExtendedObjectSupplier

Chapter 145. Usage of custom annotations

145.1. Custom annotations

Custom annotations in Eclipse

The Eclipse platform uses dependency injection as the primary programming model. This mechanism is extensible with custom annotations. The following code demonstrates how you can define your custom annotation.

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

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@javax.inject.Qualifier
@Documented
@Target({ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface UniqueTodo {

}
```

Define an annotation processor via an OSGi service

To process these custom annotations you can register an OSGi service for the abstract `ExtendedObjectSupplier` class from the `org.eclipse.e4.core.di.suppliers` package. In this registration you define via the `dependency.injection.annotation` property the annotation for which the service is responsible.

After this registration the Eclipse framework calls this annotation processor if it encounters the configured annotation.

The following snippet shows an example `UniqueTodoSupplier.xml` file which registers an `ExtendedObjectSupplier` for the `UniqueTodo` annotation.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  name="com.example.e4.rcp.todo.ownannotation">
  <implementation
    class="com.example.e4.rcp.todo.ownannotation.internal.UniqueTodoObjectSuppl
```

```

<service>
    <provide
        interface="org.eclipse.e4.core.di.suppliers.ExtendedObjectSupplier"/>
</service>
<property
    name="dependency.injection.annotation"
    type="String"
    value="com.example.e4.rcp.todo.ownannotation.UniqueTodo"/>
</scr:component>

```

The annotation processor registered via the `UniqueTodoSupplier.xml` file returns an object. The following code shows an example for an annotation processor which returns an instance of the `Todo` class.

```

package com.example.e4.rcp.todo.ownannotation.internal;

import java.util.Date;

import org.eclipse.e4.core.di.suppliers.ExtendedObjectSupplier;
import org.eclipse.e4.core.di.suppliers.IObjectDescriptor;
import org.eclipse.e4.core.di.suppliers IRequestor;

import com.example.e4.rcp.todo.model.Todo;

public class UniqueTodoObjectSupplier extends ExtendedObjectSupplier {
    @Override
    public Object get(IObjectDescriptor descriptor, IRequestor requestor,
                      boolean track, boolean group) {
        System.out.println("Own annotation processor");
        // for the purpose of providing a simple example here
        // we return a hard-coded Todo
        Todo todo = new Todo(15, "Checked", "Checked", false, new Date());
        return todo;
    }
}

```

145.2. Restrictions

Extended object suppliers do not have access to the `IEclipseContext` in which they are called. This limits them to search for objects independent of the current Eclipse context.

For example preferences are implemented as *extended object suppliers*. They look for the preference values on the file system and do not know about the `IEclipseContext`.

145.3. Example usage

An example for using your custom annotations would be the Java Persistence API (JPA). You could create a custom annotation for the JPA persistence manager. The JPA persistence manager could be injected into the Eclipse 4 application based on that custom annotation and a corresponding annotation processor.

Chapter 146. Exercise: Defining custom annotations

146.1. Target

The following exercise demonstrates how to create a custom annotation and a processor for it.

The annotation processor is just a sample implementation. If used it returns a `Todo` object with a fixed ID.

146.2. Creating a new plug-in

Create a new simple plug-in project called *com.example.e4.rcp.todo.ownannotation*.

Add the following plug-ins as dependencies to your new plug-in.

- `org.eclipse.e4.core.di`
- `com.example.e4.rcp.todo.model`

Also configure that the `javax.inject` package should be used with `1.0.0` as a minimum version.

146.3. Define and export annotations

Define the following annotation.

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

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@javax.inject.Qualifier
@Documented
@Target({ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface UniqueTodo {

}
```

Export the `com.example.e4.rcp.todo.ownannotation` package via your `MANIFEST.MF` file.

146.4. Create class

Create the `com.example.e4.rcp.todo.ownannotation.internal` package and the following class.

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

import java.util.Date;

import org.eclipse.e4.core.di.suppliers.ExtendedObjectSupplier;
import org.eclipse.e4.core.di.suppliers.IObjectDescriptor;
import org.eclipse.e4.core.di.suppliers IRequestor;

import com.example.e4.rcp.todo.model.Todo;

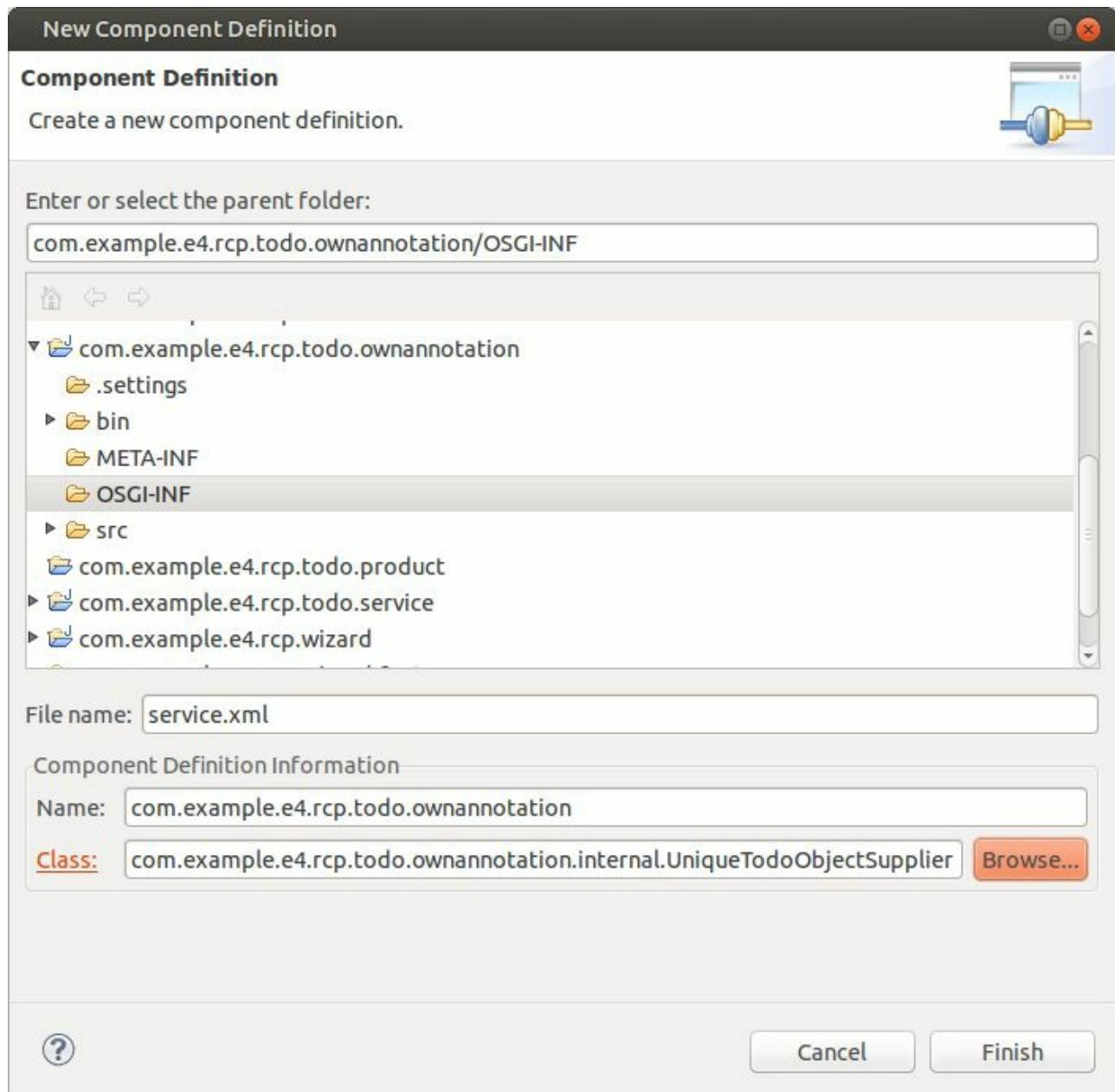
public class UniqueTodoObjectSupplier extends ExtendedObjectSupplier {
    @Override
    public Object get(IObjectDescriptor descriptor, IRequestor requestor,
        boolean track, boolean group) {
        System.out.println("Own annotation processor");
        // for the purpose of providing a simple example here
        // we return a hard-coded Todo
        Todo todo = new Todo(15, "Checked", "Checked", false, new Date());
        return todo;
    }
}
```

Note

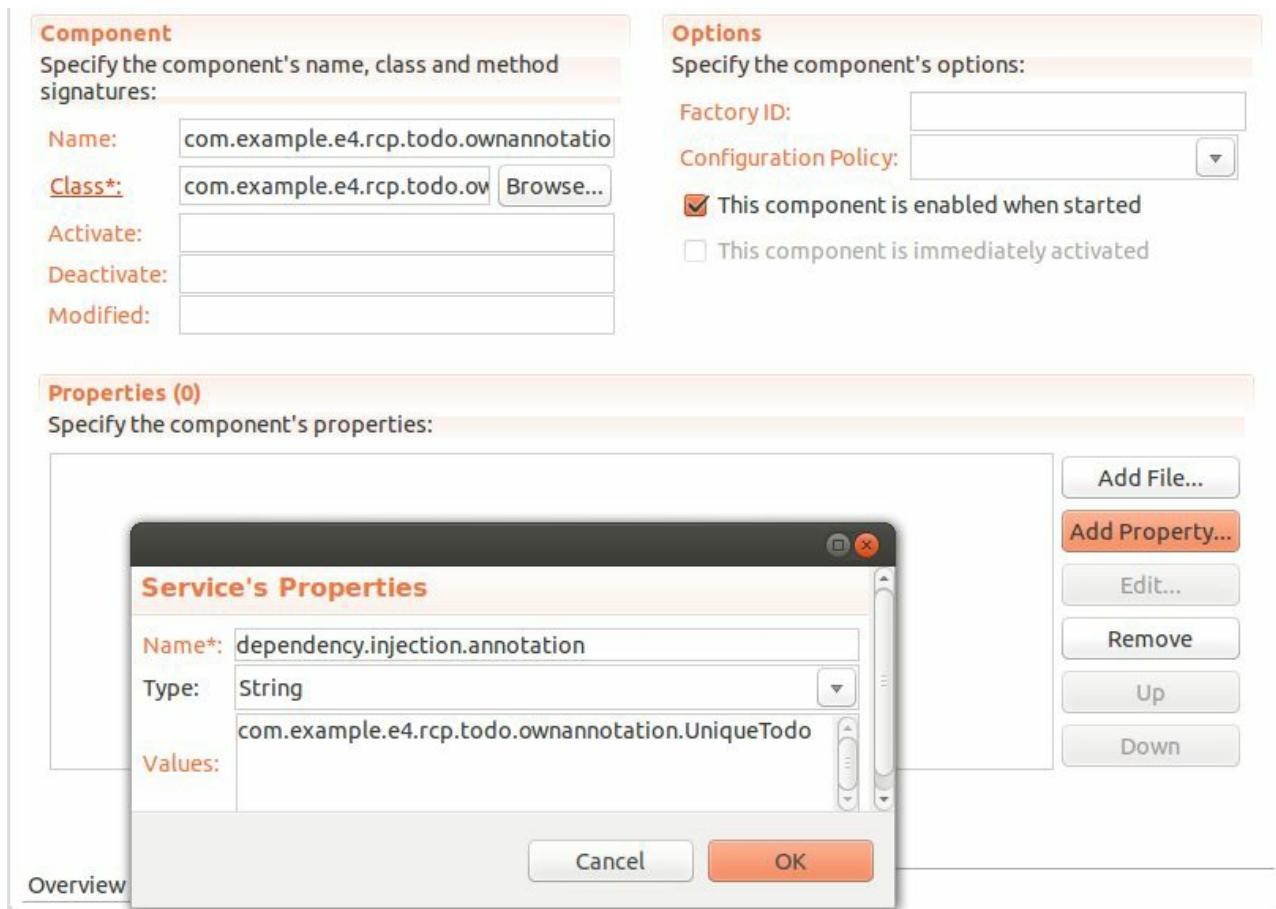
The `ExtendedObjectSupplier` class is not yet released as official API. See [Section 6.4, “Eclipse API and internal API”](#).

146.5. Register the annotation processor

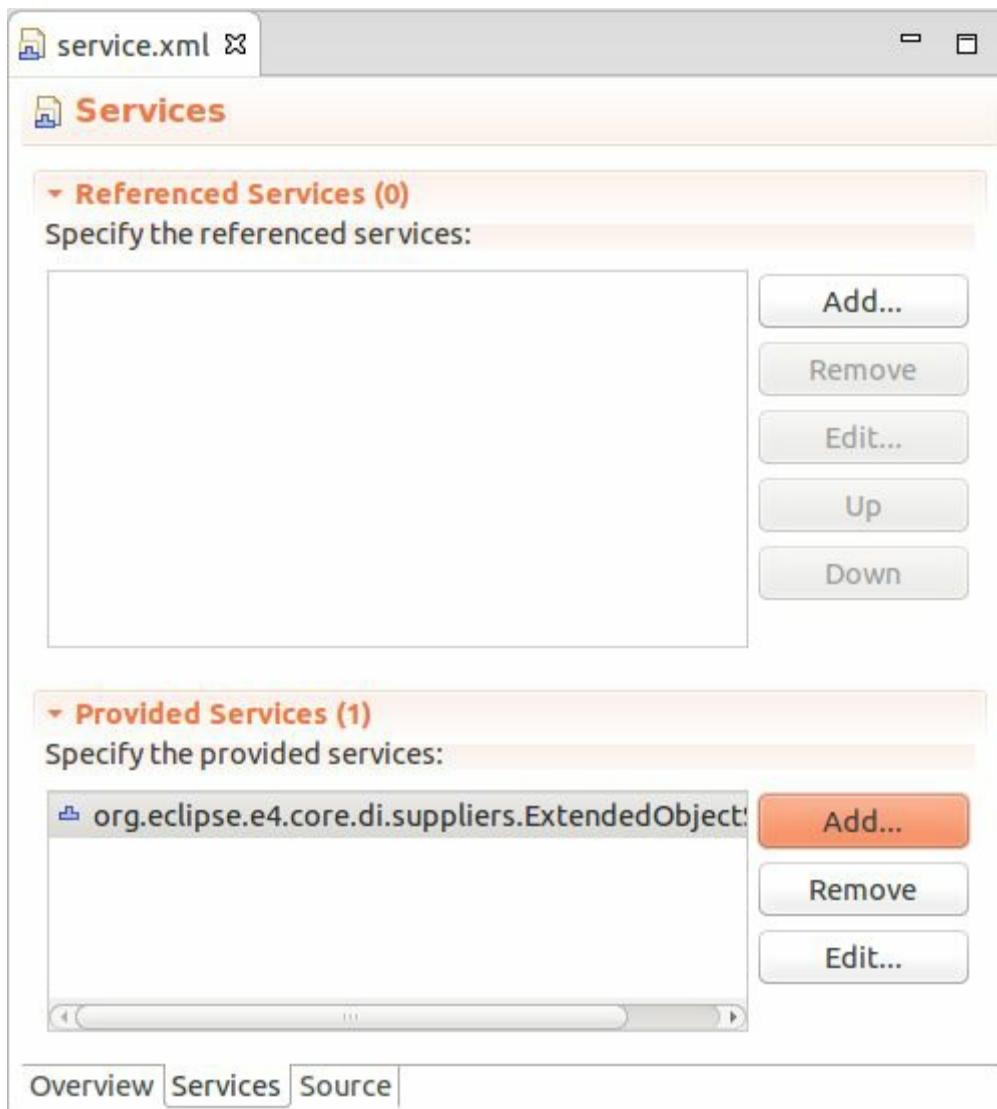
Register this class as service for the `ExtendedObjectSupplier` abstract class. Use OSGi declarative services for this. The dialog for the component definition file should be similar to the following screenshot.



Add a property to the component to identify the annotation for which this annotation processor is responsible. Use `dependency.injection.annotation.name` pointing to `com.example.e4.rcp.todo.ownannotation.UniqueTodo` and `dependency.injection.annotation.value` to `com.example.e4.rcp.todo.ownannotation.UniqueTodoObjectSupplier`.



On the *Services* tab define that you provide the service based on the `org.eclipse.e4.core.di.suppliers.ExtendedObjectSupplier` class.



The resulting component definition file should be similar to the following file.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    name="com.example.e4.rcp.todo.ownannotation">
    <implementation
        class="com.example.e4.rcp.todo.ownannotation.internal.UniqueTodoObjectSupplier">
        <service>
            <provide
                interface="org.eclipse.e4.core.di.suppliers.ExtendedObjectSupplier"/>
        </service>
        <property
            name="dependency.injection.annotation"
            type="String"
            value="com.example.e4.rcp.todo.ownannotation.UniqueTodo"/>
    </scr:component>
```

Warning

Ensure that the component definition file is registered in your *MANIFEST.MF* file. You can use the following code for comparison; in this example the component definition file is called *component.xml* and it is located in the *OSGI-INF* folder.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Ownannotation
Bundle-SymbolicName: com.example.e4.rcp.todo.ownannotation
Bundle-Version: 1.0.0.qualifier
Bundle-Vendor: EXAMPLE
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Require-Bundle: org.eclipse.e4.core.di;bundle-version="1.3.0",
    com.example.e4.rcp.todo.model;bundle-version="1.0.0"
Import-Package: javax.inject;version="1.0.0"
Export-Package: com.example.e4.rcp.todo.ownannotation
Service-Component: OSGI-INF/UniqueTodoSupplier.xml
Bundle-ActivationPolicy: lazy
```

Warning

Ensure that the *Activate this plug-in when one of its classes is loaded* is set on the *MANIFEST.MF* file of the *com.example.e4.rcp.todo.ownannotation* plug-in. This is required for the OSGi declarative service functionality to work.

146.6. Add the plug-in as dependency

Add your new plug-in as dependency to your `com.example.e4.rcp.todo` application plug-in via its *MANIFEST.MF* file.

146.7. Update the product configuration (via the features)

Include the new plug-in into your feature to make it available for your product.

146.8. Update the build.properties

Add the `component.xml` file to the `build.properties` file, to include it into the exported application.

146.9. Validate: Use your custom annotation

In order to use your new annotation, add it to a field or method parameter in one of your parts. For example add the following method to your `PlaygroundPart` class.

```
@Inject
public void setTodo(@UniqueTodo Todo todo) {
    // do something with the _unique_ Todo
}
```

Start your application (via the product as you added a new plug-in). Ensure that you get a `Todo` injected based on your `@UniqueTodo` annotation.

Part XXXIV. Using Java libraries in Eclipse applications

Chapter 147. Defining and using libraries in Java

147.1. What is a JAR file?

A JAR file is a Java archive based on the pkzip file format. JAR files are the deployment format for Java. A JAR can contain Java classes and other resources (icons, property files, etc.) and can be executable.

You can distribute your program in a jar file or you can use existing java code via jars by putting them into your classpath.

147.2. Using Java libraries

If you add a JAR file to your classpath, you can use its classes in your Java application.

Chapter 148. Using JAR files in Eclipse applications

148.1. JAR files without OSGi meta-data

If a JAR file does not contain the OSGi meta-data in the *META-INF/MANIFEST.MF* file, it cannot be directly consumed by other Eclipse plug-ins.

If you want to use such standard Java libraries in other Eclipse plug-ins you have to convert them also into a plug-in. After the conversion the resulting JAR file can still be used in a non OSGi runtime, e.g., a Java webserver. The Java runtime ignores the additional OSGi meta-data.

Eclipse provides a wizard to convert a JAR file with OSGi meta-data to a plug-in. The usage of this wizard is demonstrated in [Section 148.2, “Integrating external jars / third party libraries”](#).

Note

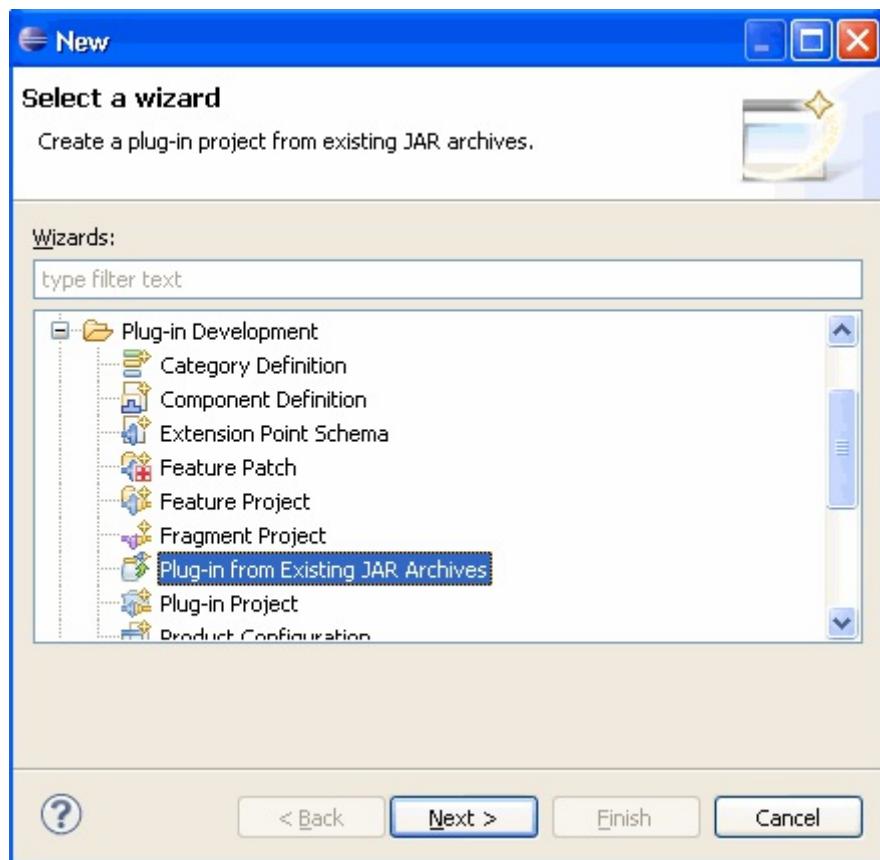
If you repackage a JAR into a plug-in it is wise to check if the license allows this. You should also try to contact the author of the software and ask if he can integrate the OSGi meta-data directly in his library.

148.2. Integrating external jars / third party libraries

The following gives an example how to convert a standard Java JAR to an Eclipse plug-in.

Creating a plug-in project for your jar

Create a new plug-in project by selecting File → New → Project → Plug-in Development → Plug-in from Existing JAR Archives.

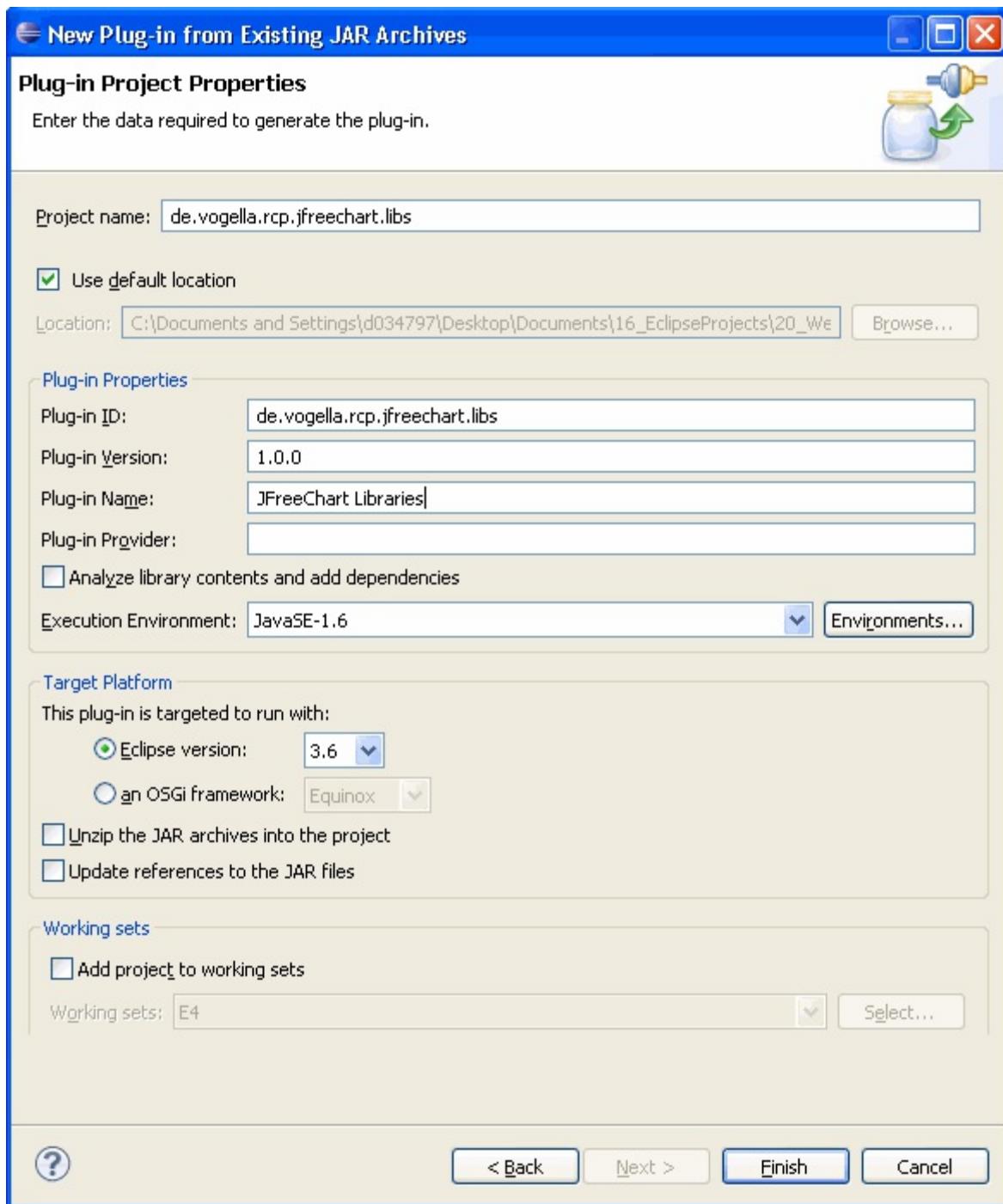


Add the JAR files you want to have in this new plug-in. Press next.



Enter a name and a version for your new plug-in. Uncheck the *Unzip the JAR archive into the project* flag. Unchecking this flag prevents that the class files are extracted from the JAR file which is not necessary to use them.

Afterwards press the *Finish* button in the wizard.



You have created a new plug-in for the selected JAR files. Open the file *MANIFEST.MF* and validate that all required packages are exported on the tab *Runtime*. All the packages from your JAR files should be included in the exported packages as OSGi will otherwise prevent other plug-ins from accessing them.

Using the new plug-in project

In the plug-in project which should access the library, open the *MANIFEST.MF*

file and select the *Dependencies* tab. Add the new plug-in as dependency.

The screenshot shows the Eclipse IDE interface with the 'Automated Management of Dependencies' view open. At the top, there's a header bar with tabs: Overview, Dependencies (which is highlighted with a red box), Runtime, Extensions, Extension Points, Build, and MANIFEST.MF. Below the tabs, there are two main sections: 'Required Plug-ins' and 'Automated Management of Dependencies'.

Required Plug-ins: This section lists the required plug-ins for the operation of the current plugin. It includes three entries: 'org.eclipse.ui', 'org.eclipse.core.runtime', and 'de.vogella.jfreechart.libs (1.0.0)'. The entry 'de.vogella.jfreechart.libs (1.0.0)' is highlighted with a red box. To the right of the list are several buttons: 'Add...', 'Remove', 'Up', 'Down', and 'Properties...'. Below the list, it says 'Total: 3'.

Automated Management of Dependencies: This section contains tabs for Overview, Dependencies, Runtime, Extensions, Extension Points, Build, and MANIFEST.MF. The 'Dependencies' tab is currently selected and highlighted with a red box.

Part XXXV. Testing of Eclipse plug-ins and applications

Chapter 149. Introduction to JUnit

149.1. The JUnit framework

JUnit in version 4.x is a test framework which uses annotations to identify methods that specify a test.

The main websites for JUnit are the [JUnit homepage](#) and the [GitHub project page](#).

149.2. How to define a test in JUnit?

A JUnit *test* is a method contained in a class which is only used for testing. This is called a *Test class*.

To write a test with the JUnit 4.x framework you annotate a method with the `@org.junit.Test` annotation.

In this method you use a method provided by the JUnit framework to check the expected result of the code execution versus the actual result.

149.3. Example JUnit test

The following code shows a JUnit test method.

```
@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {

    // MyClass is tested
    MyClass tester = new MyClass();

    // Tests
    assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
    assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
    assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
}
```

149.4. JUnit naming conventions

There are several potential naming conventions for JUnit tests. In widespread use is to use the name of the class under test and to add the "Test" suffix to the test class.

Tip

You should prefer the "Test" suffix over "Tests" as the Maven build system (via its surfice plug-in) automatically includes such classes in its test scope.

For the test method names it is frequently recommended to use the word "should" in the test method name, as for example "ordersShouldBeCreated" or "menuShouldGetActive" as this gives a good hint what should happen if the test method is executed.

As a general rule, a test name should explain what the test does so that it can be avoided to read the actual implementation.

149.5. JUnit test suites

If you have several test classes, you can combine them into a *test suite*. Running a test suite will execute all test classes in that suite in the specified order.

The following example code shows a test suite which defines that two test classes (MYClassTest and MySecondClassTest) should be executed. If you want to add another test class you can add it to `@Suite.SuiteClasses` statement.

```
package com.vogella.junit.first;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ MyClassTest.class, MySecondClassTest.class })
public class AllTests {

}
```

A test suite can also contain other test suites.

149.6. Run your test from the command line

You can also run your JUnit tests outside Eclipse via standard Java code. Build frameworks like Apache Maven or Gradle in combination with a Continuous Integration Server (like Hudson or Jenkins) are typically used to execute tests automatically on a regular basis.

The `org.junit.runner.JUnitCore` class provides the `runClasses()` method which allows you to run one or several tests classes. As a return parameter you receive an object of the type `org.junit.runner.Result`. This object can be used to retrieve information about the tests.

The following class demonstrates how to run the `MyClassTest`. This class will execute your test class and write potential failures to the console.

```
package de.vogella.junit.first;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class MyTestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(MyClassTest.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
    }
}
```

This class can be executed like any other Java program on the command line. You only need to add the JUnit library JAR file to the classpath.

Chapter 150. Basic JUnit code constructs

150.1. Available JUnit annotations

JUnit 4.x uses annotations to mark methods and to configure the test run. The following table gives an overview of the most important available annotations.

Table 150.1. Annotations

Annotation	Description
<code>@Test public void method()</code>	The <code>@Test</code> annotation identifies a method as a test method.
<code>@Test(expected = Exception.class)</code>	Fails if the method does not throw the named exception.
<code>@Test(timeout=100)</code>	Fails if the method takes longer than 100 milliseconds.
<code>@Before public void method()</code>	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
<code>@After public void method()</code>	This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
<code>@BeforeClass public static void method()</code>	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as <code>static</code> to work with JUnit.
<code>@AfterClass public static void method()</code>	This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as <code>static</code> to work with JUnit.
<code>@Ignore or @Ignore("Why disabled")</code>	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional

150.2. Assert statements

JUnit provides static methods in the `Assert` class to test for certain conditions. These *assertion methods* typically start with `assert` and allow you to specify the error message, the expected and the actual result. An *assertion method* compares the actual value returned by a test to the expected value, and throws an `AssertionException` if the comparison test fails.

The following table gives an overview of these methods. Parameters in [] brackets are optional and of type String.

Table 150.2. Test methods

Statement	Description
<code>fail(message)</code>	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
<code>assertTrue([message,] boolean condition)</code>	Checks that the boolean condition is true.
<code>assertFalse([message,] boolean condition)</code>	Checks that the boolean condition is false.
<code>assertEquals([message,] expected, actual)</code>	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
<code>assertEquals([message,] expected, actual, tolerance)</code>	Test that float or double values match. The tolerance is the number of decimals which must be the same.
<code>assertNull([message,] object)</code>	Checks that the object is null.
<code>assertNotNull([message,] object)</code>	Checks that the object is not null.
<code>assertSame([message,] expected, actual)</code>	Checks that both variables refer to the same object.
<code>assertNotSame([message,] expected, actual)</code>	Checks that both variables refer to different objects.

Note

You should provide meaningful messages in assertions so that it is

easier for the developer to identify the problem. This helps in fixing the issue, especially if someone looks at the problem, who did not write the code under test or the test code.

150.3. Test execution order

JUnit assumes that all test methods can be executed in an arbitrary order. Well-written test code should not assume any order, i.e., tests should not depend on other tests.

As of JUnit 4.11 you can use an annotation to define that the test methods are sorted by method name, in lexicographic order.

To activate this feature, annotate your test class with the `@FixMethodOrder(MethodSorters.NAME_ASCENDING)` annotation.

Note

The default in JUnit 4.11 is to use a deterministic, but not predictable, order which can also be explicitly specified via the `MethodSorters.DEFAULT` parameter in the above annotation. You can also use `MethodSorters.JVM` which uses the JVM defaults, which may vary from run to run.

Chapter 151. Eclipse IDE support for JUnit tests

151.1. Creating JUnit tests

You can write the JUnit tests manually, but Eclipse supports the creation of JUnit tests via wizards.

For example, to create a JUnit test or a test class for an existing class, right-click on your new class, select this class in the *Package Explorer* view, right-click on it and select New → JUnit Test Case.

Alternatively you can also use the JUnit wizards available under File → New → Other... → Java → JUnit.

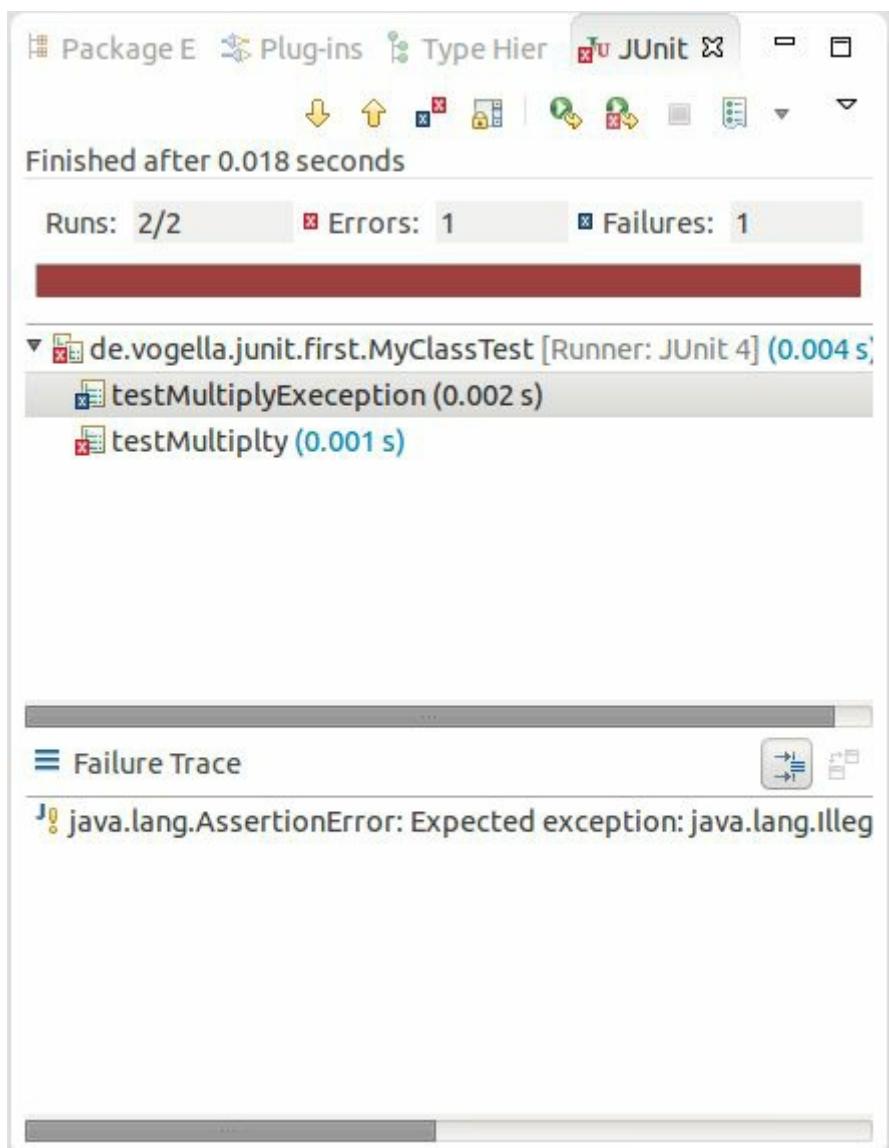
151.2. Running JUnit tests

The Eclipse IDE also provides support for executing your tests interactively.

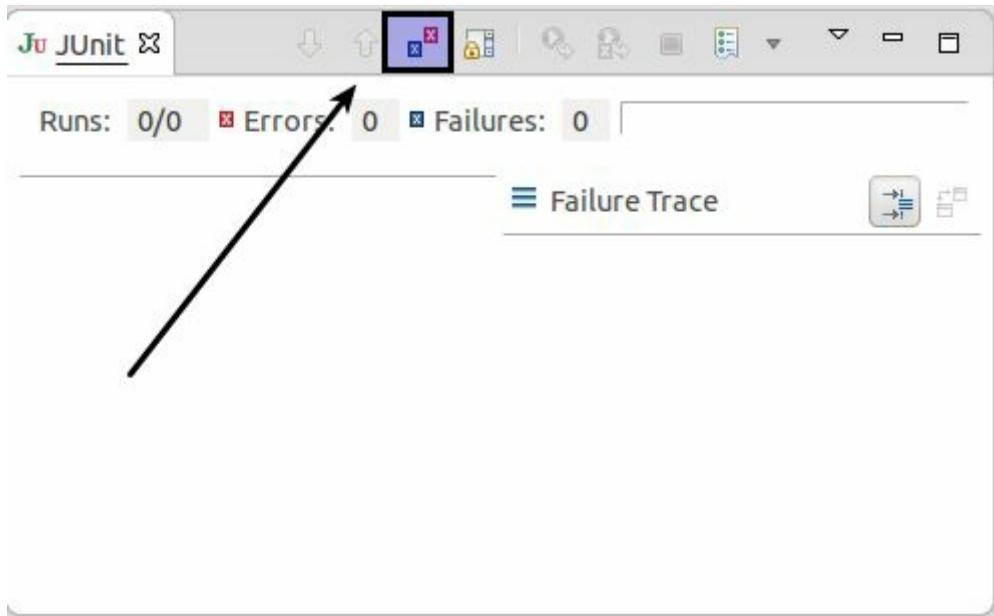
To run a test, select the class which contains the tests, right-click on it and select Run-as → JUnit Test. This starts JUnit and executes all test methods in this class.

Eclipse provides the **Alt+Shift+X, ,T** shortcut to run the test in the selected class. If you position the cursor in the Java editor on one test method name, this shortcut runs only the selected test method.

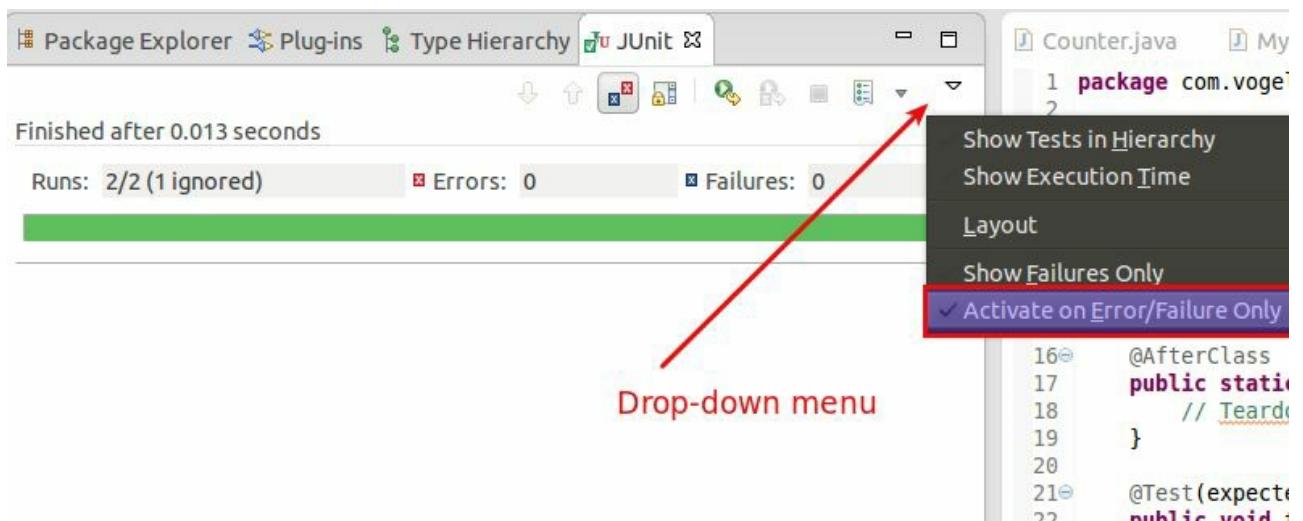
To see the result of an JUnit test, Eclipse uses the *JUnit* view which shows the results of the tests. You can also select individual unit tests in this view , right-click on them and select *Run* to execute them again.



By default this view shows all tests. You can also configure, that it only shows failing tests.



You can also define that the view is only activated if you have a failing test.



Note

Eclipse creates run configurations for tests. You can see and modify these via the Run → Run Configurations... menu.

151.3. JUnit static imports

Static import is a feature that allows fields and methods) defined in a class as `public static` to be used in Java code without specifying the class in which the field is defined.

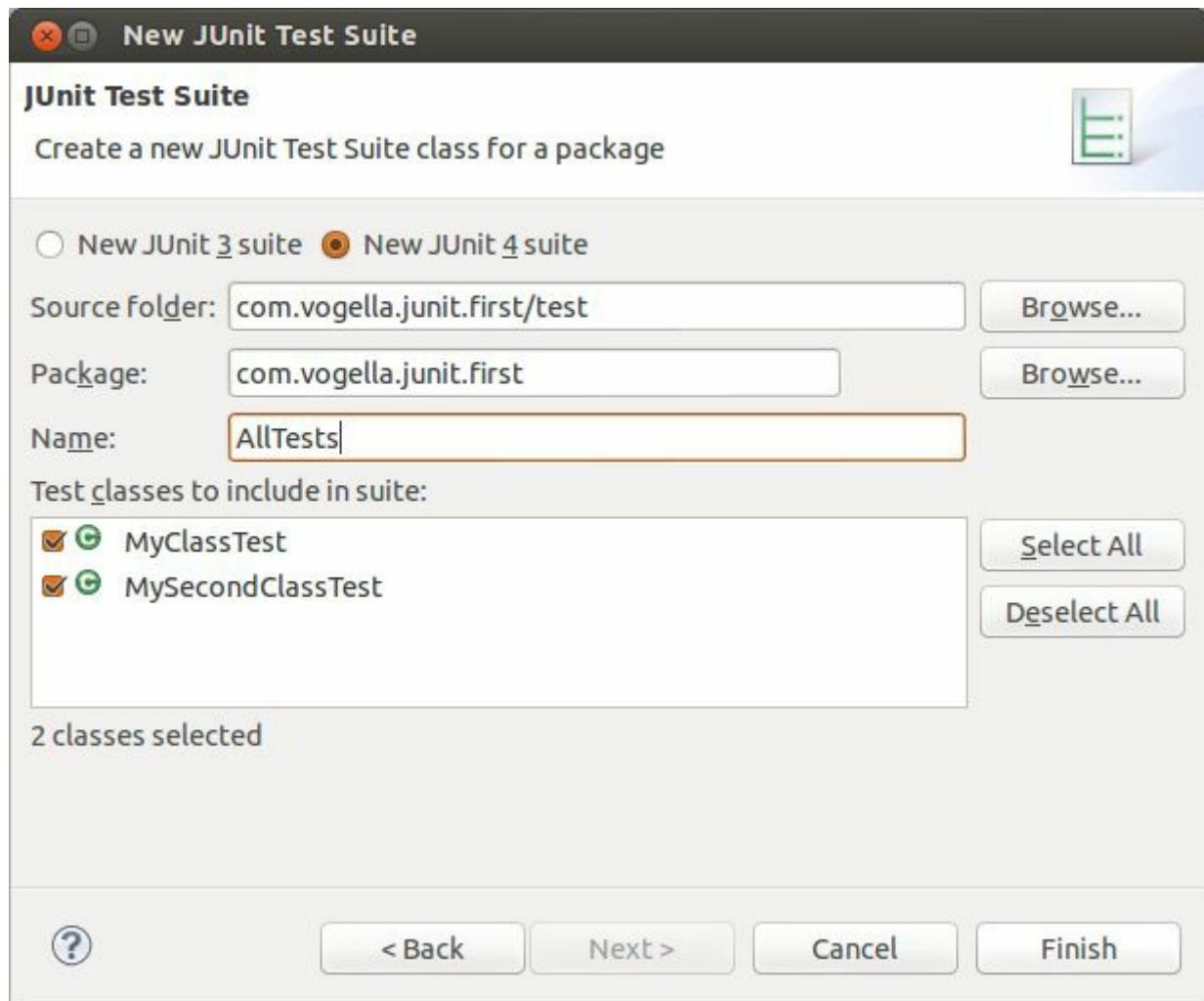
JUnit assert statement are typically defined as `public static` to allow the developer to write short test statements. The following snippet demonstrates an assert statement with and without static imports.

```
// without static imports you have to write the following statement  
Assert.assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));  
  
// alternatively define assertEquals as static import  
import static org.junit.Assert.assertEquals;  
  
// more code  
  
// use assertEquals directly because of the static import  
assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
```

The Eclipse IDE cannot always create the corresponding static import statements automatically. You can make the JUnit test methods available via the *Content Assists*. *Content Assists* is a functionality in Eclipse which allows the developer to get context sensitive code completion in an editor upon user request. See [Section 151.7, “Setting Eclipse up for using JUnits static imports”](#) for the required setup.

151.4. Wizard for creating test suites

To create a test suite in Eclipse, you select the test classes which should be included into this in the *Package Explorer* view, right-click on them and select New → Other... → JUnit → JUnit Test Suite.



151.5. Testing exception

The `@Test (expected = Exception.class)` annotation is limited as it can only test for one exception. To test exceptions, you can use the following test pattern.

```
try {
    mustThrowException();
    fail();
} catch (Exception e) {
    // expected
    // could also check for message of exception, etc.
}
```

151.6. JUnit Plug-in Test

JUnit Plug-in tests are used to write unit tests for your plug-ins. These tests are executed by a special test runner that launches another Eclipse instance in a separate VM—just and executes the test methods within that instance.

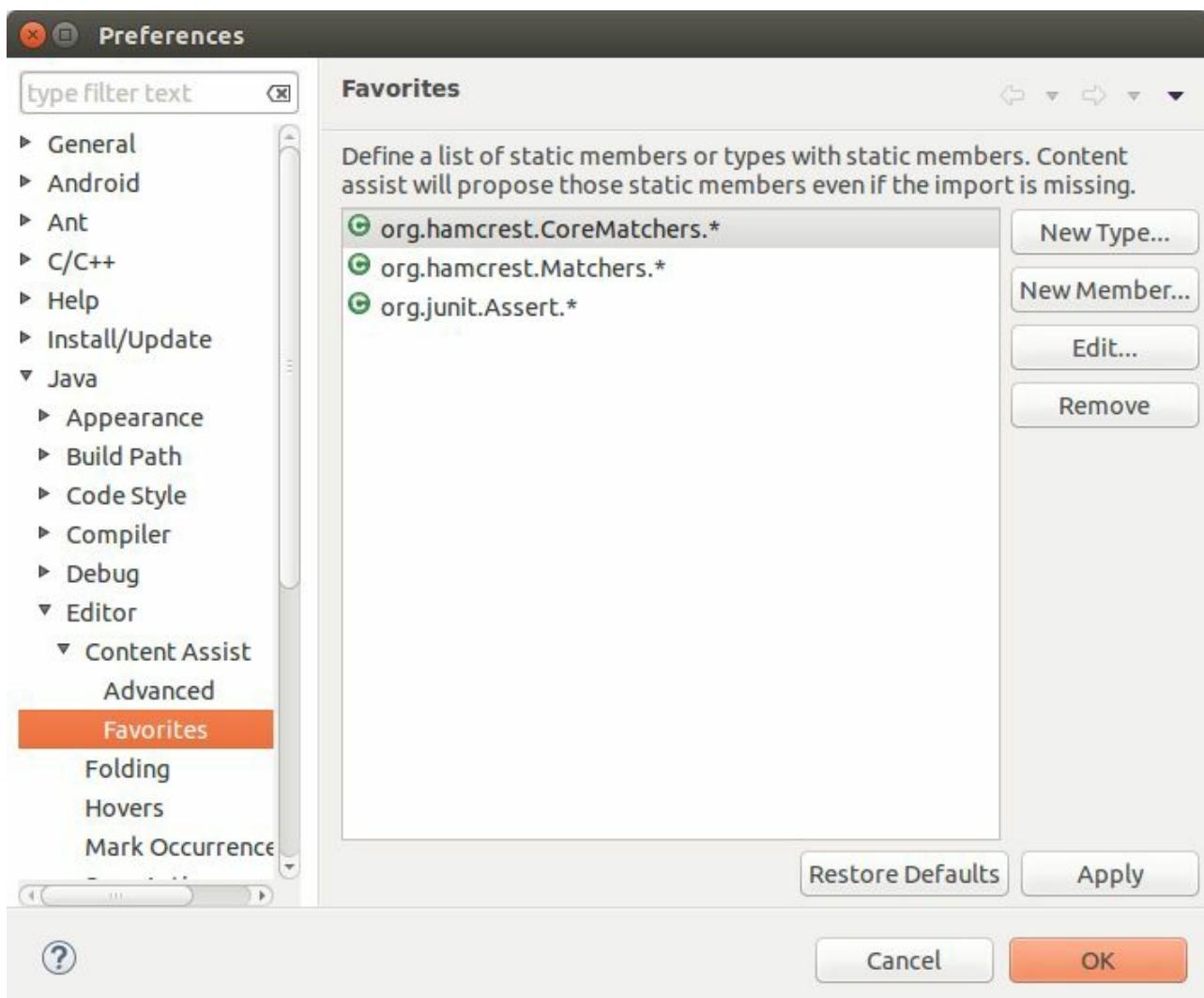
151.7. Setting Eclipse up for using JUnits static imports

You can configure Eclipse to allow you to use code completion to insert typical JUnit method calls. For this open the Preferences via Window → Preferences and select Java → Editor → Content Assist → Favorites.

Use the *New Type* button to add the following entries to it:

- org.junit.Assert
- org.hamcrest.CoreMatchers
- org.hamcrest.Matchers

This makes, for example, the `assertTrue`, `assertFalse` and `assertEquals` methods directly available in the *Content Assists*.



You can now use *Content Assists* (shortcut: **Ctrl+Space**) to add the method and

the import.

Chapter 152. Testing Eclipse 4 applications

152.1. General testing

In general all Java classes in an Eclipse 4 application can be tested similarly to other Java applications. This description highlights the special Eclipse 4 constructs.

152.2. Fragment projects

Tests for Eclipse plug-ins are typically contained in a fragment project. This way the tests can access all classes in their host plug-in.

152.3. Testing user interface components

Eclipse classes that are using the application model have no hard dependency on the Eclipse framework. Therefore you can test these components directly with JUnit.

For example take the following part.

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

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

import org.eclipse.swt.SWT;
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.Label;

public class TodoOverviewPart {

    @PostConstruct
    public void createControls(Composite parent) {
        parent.setLayout(new GridLayout(2, false));
        Button button = new Button(parent, SWT.PUSH);
        button.setLayoutData(new GridData(SWT.BEGINNING, SWT.CENTER, false,
            false));
        button.setText("Load Data");
        Label label = new Label(parent, SWT.NONE);
        label.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true,
            false));
        label.setText("Data not available");

    }

    @PreDestroy
    public void dispose() {
    }
}
```

This part can be created via a simple Java Program which has the SWT library included in its classpath.

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

import org.eclipse.swt.widgets.Display;
```

```
import org.eclipse.swt.widgets.Shell;

public class TodoOverviewPartTest {
    public static void main(String... main) {
        Display display = new Display();
        Shell shell = new Shell(display);
        TodoOverviewPart part = new TodoOverviewPart();
        part.createControls(shell);
        shell.open();
        // create and check the event loop
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
                display.sleep();
        }
        display.dispose();
    }
}
```

The above code can be easily changed to a unit test. Your test class can create the class, provide the required dependencies and run the tests.

152.4. Testing dependency injection

You can include the process of dependency injection into the test. Create your own `IEclipseContext` and use the `ContextInjectionFactory.make()` method to create the object which should be tested.

The following code shows an example of how to create your own context and construct the object based on this construct. This test needs to run as *JUnit Plug-in* test.

```
package testing;

import org.eclipse.e4.core.contexts.ContextInjectionFactory;
import org.eclipse.e4.core.contexts.EclipseContextFactory;
import org.eclipse.e4.core.contexts.IEclipseContext;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class CreateContextText {
    @Test
    public void testCreation() {
        IEclipseContext context = EclipseContextFactory.create();
        // prepare the context for the test
        context.set("myvalue1", "For testing");
        // more things, for example a LayoutManager
        MyClass test = ContextInjectionFactory.make(MyClass.class, context);
    }
}
```

Chapter 153. UI testing with SWTBot

153.1. User interface testing with SWTBot

SWTBot is an Eclipse project which supports testing the user interface of an SWT based application.

SWTBot provides an API which allows you to interact with the user interface of the application and to validate certain conditions on the user interface.

The Eclipse or SWT application is started and controlled by SWTBot.

If SWTBot is used to test Eclipse based applications, you need to run the JUnit test with a *JUnit Plug-in Test* run configuration. A JUnit plug-in test allows you to start and test Eclipse bundles.

153.2. Installation

Install SWTBot via the Eclipse Update manager which you find under Help → Install New Software.... The update site for the SWTBot is:
<http://download.eclipse.org/technology/swtbot/releases/latest/>

Install the *SWTBot Eclipse Features* and the *SWTBot SWT Features*.

153.3. SWTBot API

SWTBot is the base class for testing SWT applications.

The `SWTWorkbenchBot` class provides API to interact with Eclipse 3.x applications. The `SWTWorkbenchBot` class extends `SWTBot`. To test Eclipse 4.x RCP application you use the `SWTBot` class.

An user interface interaction may take some time, e.g., if the application reads some data. Therefore SWTBot waits, by default, 5 seconds before throwing an exception. You can change the timeout via the following:

```
// Set the timeout to 6 seconds  
SWTBotPreferences.TIMEOUT = 6000;
```

For details on the API usage please see [SWTBot wiki](#).

Note

The support for Eclipse 4 RCP application is at the time of this writing fairly new. Therefore this chapter is intentionally short, as improvements and changes are expected in the future. Please see the project homepage for the most recent information.

Warning

Testing an application with a login screen is currently not supported by SWTBot. See [SWTBot FAQ](#).

Part XXXVI. Eclipse application updates

Chapter 154. Implementing updates in your application

154.1. Eclipse application updates

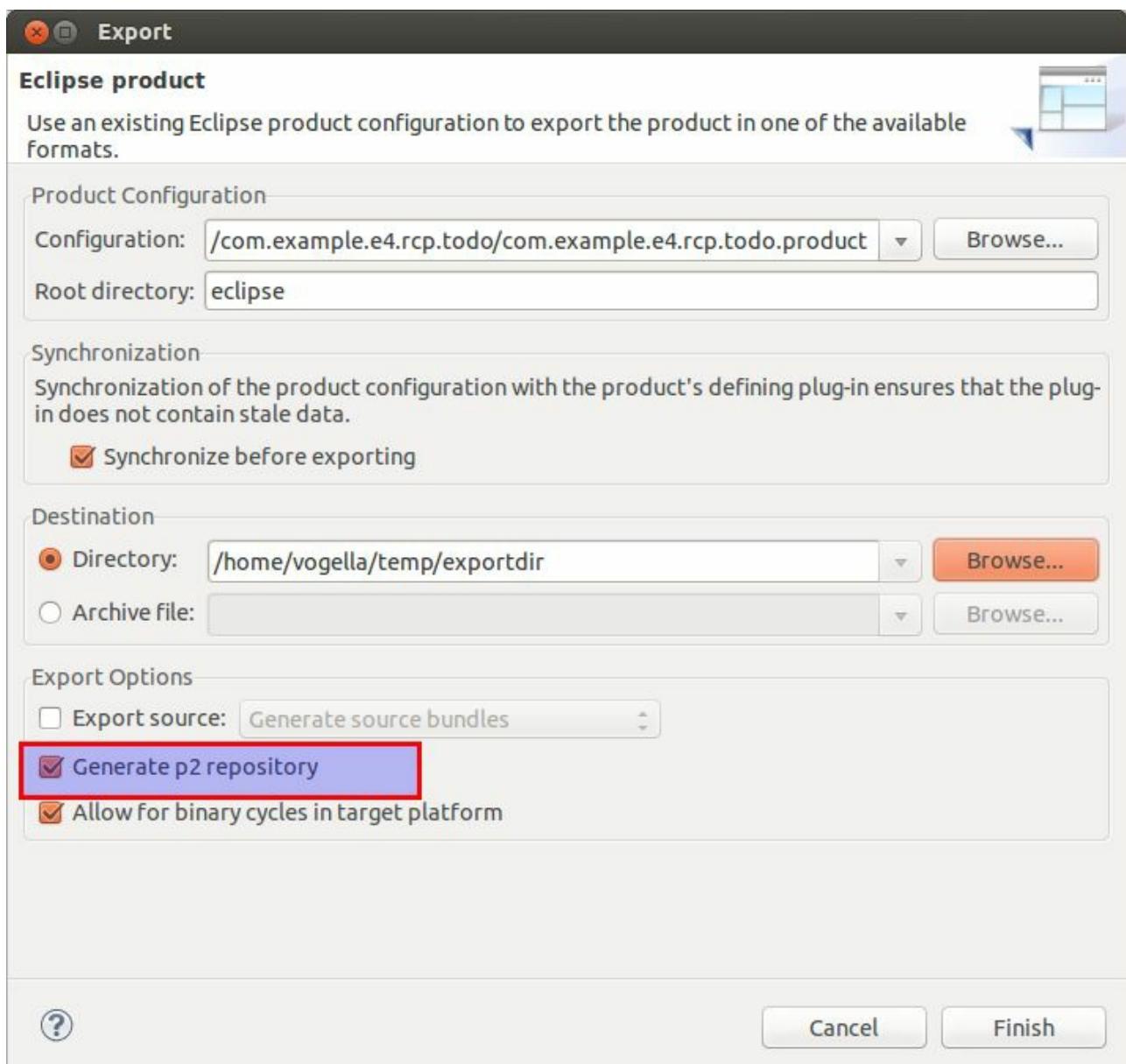
The Eclipse platform provides an installation and update mechanism called *Eclipse p2* (short: *p2*). This update mechanism allows you to update Eclipse applications and to install new functions.

The update and installation of functions with p2 is based on *feature projects* (short: *features*). It is possible to update complete products or individual features. In the terminology of p2 these features are *installable units*.

154.2. Creating p2 update sites

Installable units can be grouped into a p2 repository. A repository is defined via its URI and can point to a local file system or to a web server. A p2 repository is also frequently called *update site*.

During the export of an Eclipse application you can select the *Generate p2 repository* option in the Eclipse product export dialog. This option is highlighted in the following screenshot.



If you select this option, an update site is created in a sub-folder called *repository* of the export directory.

Typically this directory is copied to a web server so users can install new functionality or upgrades using the p2 mechanism. The Eclipse update mechanism also supports file based update sites. File based update sites are useful for testing the update functionality.

An update site contains an artifact and a metadata repository.

154.3. p2 composite repositories

It is possible to create p2 composite repositories. Such a repository wraps the information about other repositories. If a client connects to such an update site, the other update sites are contacted and the user can install features from the other update site.

To build such a composite repository you have to create two files, one called *compositeContent.xml* and another called *compositeArtifacts.xml*. The following example demonstrates the content of these files.

The following listing is an example for an *compositeContent.xml* file.

```
<?xml version='1.0' encoding='UTF-8'?>
<?compositeMetadataRepository version='1.0.0'?>
<repository name='"oclipse development environment"' type='org.eclipse.equinox.internal.p2.metadata.repository.CompositeMetadataRep version='1.0.0'>
<properties size='1'>
    <property name='p2.timestamp' value='1243822502499' />
</properties>
<children size='2'>
    <child location='http://download.eclipse.org/releases/luna/' />
    <child location='http://dl.bintray.com/vogella/company/eclipse-preference-s' />
</children>
</repository>
```

The following listing is an example for a fitting *compositeArtifacts.xml* file.

```
<?xml version='1.0' encoding='UTF-8'?>
<?compositeArtifactRepository version='1.0.0'?>
<repository name='"voclipse development environment"' type='org.eclipse.equinox.internal.p2.artifact.repository.CompositeArtifactRep version='1.0.0'>
<properties size='1'>
    <property name='p2.timestamp' value='1243822502440' />
</properties>
<children size='2'>
    <child location='http://download.eclipse.org/releases/luna/' />
    <child location='http://dl.bintray.com/vogella/company/eclipse-preference-spy' />
</children>
</repository>
```

Chapter 155. Using the p2 update API

155.1. Required plug-ins for updates

The following table lists the core plug-ins and the feature which provide the non-user interface functionality of p2.

Table 155.1. Eclipse p2 plug-ins and features

Plug-in	Description
org.eclipse.equinox.p2.core	Core p2 functionality
org.eclipse.equinox.p2.engine	The engine carries out the provisioning operation.
org.eclipse.equinox.p2.operations	Layer over the core and engine API to describe updates as an atomic install.
org.eclipse.equinox.p2.metadata.repository	Contains the definition of p2 repositories.
org.eclipse.equinox.p2.core.feature	Feature containing the p2 bundles.

To use the Eclipse update API you need to include these plug-ins as dependencies to your manifest file and you must add the feature to your product configuration file.

155.2. Updating Eclipse RCP applications

To update your Eclipse 4 application you use the p2 API. The following code shows an example handler which performs an update of a complete Eclipse RCP application. The method annotated with `@Execute` configures the update operation and runs it.

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

import java.net.URI;
import java.net.URISyntaxException;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.core.runtime.jobs.IJobChangeEvent;
import org.eclipse.core.runtime.jobs.Job;
import org.eclipse.core.runtime.jobs.JobChangeAdapter;
import org.eclipse.core.di.annotations.Execute;
import org.eclipse.e4.core.services.log.Logger;
import org.eclipse.e4.ui.di.UISynchronize;
import org.eclipse.e4.ui.workbench.IWorkbench;
import org.eclipse.equinox.p2.core.IProvisioningAgent;
import org.eclipse.equinox.p2.operations.ProvisioningJob;
import org.eclipse.equinox.p2.operations.ProvisioningSession;
import org.eclipse.equinox.p2.operations.UpdateOperation;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Shell;

// Require-Bundle: org.eclipse.equinox.p2.core|engine|operation|metadata.repository
// Feature: org.eclipse.equinox.p2.core.feature
//
// !!! do not run from within IDE, the update only works in an exported product
//
public class UpdateHandler {
    private static final String REPOSITORY_LOC =
        System.getProperty("UpdateHandler.Repo", "http://localhost/repository");

    @Execute
    public void execute(final IProvisioningAgent agent, final Shell shell,
        final UISynchronize sync, final IWorkbench workbench,
        final Logger logger) {
        Job j = new Job("Update Job") {
            @Override
            protected IStatus run(final IProgressMonitor monitor) {
                return checkForUpdates(agent, shell, sync, workbench, monitor,
                    logger);
            }
        };
        j.schedule();
    }
}
```

```

private IStatus checkForUpdates(final IProvisioningAgent agent,
    final Shell shell, final UISynchronize sync,
    final IWorkbench workbench, IProgressMonitor monitor, Logger logger) {

    // configure update operation
    final ProvisioningSession session = new ProvisioningSession(agent);
    final UpdateOperation operation = new UpdateOperation(session);
    configureUpdate(operation, logger);

    // check for updates, this causes I/O
    final IStatus status = operation.resolveModal(monitor);

    // failed to find updates (inform user and exit)
    if (status.getCode() == UpdateOperation.STATUS NOTHING_TO_UPDATE) {
        showMessage(shell, sync);
        return Status.CANCEL_STATUS;
    }

    // run installation
    final ProvisioningJob provisioningJob = operation
        .getProvisioningJob(monitor);

    // updates cannot run from within Eclipse IDE!!!
    if (provisioningJob == null) {
        logger.error("Trying to update from the Eclipse IDE? This won't work!");
        return Status.CANCEL_STATUS;
    }
    configureProvisioningJob(provisioningJob, shell, sync, workbench);

    provisioningJob.schedule();
    return Status.OK_STATUS;
}

private void configureProvisioningJob(ProvisioningJob provisioningJob,
    final Shell shell, final UISynchronize sync,
    final IWorkbench workbench) {

    // register a job change listener to track
    // installation progress and notify user upon success
    provisioningJob.addJobChangeListener(new JobChangeAdapter() {
        @Override
        public void done(IJobChangeEvent event) {
            if (event.getResult().isOK()) {
                sync.syncExec(new Runnable() {

                    @Override
                    public void run() {
                        boolean restart = MessageDialog
                            .openQuestion(shell,
                                "Updates installed, restart?",
                                "Updates have been installed. Do you want to restart?");
                        if (restart) {

```

```

        workbench.restart();
    }
}
}) ;

}

super.done(event);
}
);
}

private void showMessage(final Shell parent, final UISynchronize sync) {
    sync.syncExec(new Runnable() {

        @Override
        public void run() {
            MessageDialog
                .openWarning(parent, "No update",
                    "No updates for the current installation have been found.");
        }
    );
}
}

private UpdateOperation configureUpdate(final UpdateOperation operation,
    Logger logger) {
    // create uri and check for validity
    URI uri = null;
    try {
        uri = new URI(REPOSITORY_LOC);
    } catch (final URISyntaxException e) {
        logger.error(e);
        return null;
    }

    // set location of artifact and metadata repo
    operation.getProvisioningContext().setArtifactRepositories(new URI[] { uri })
    operation.getProvisioningContext().setMetadataRepositories(new URI[] { uri })
    return operation;
}
}
}

```

Chapter 156. Exercise: Performing an application update

Note

This exercise is optional.

156.1. Preparation: Ensure the exported product works

Export your product as described in [Section 19.1, “Creating a stand-alone version of the application”](#) and ensure that the exported application starts correctly outside of the Eclipse IDE.

If you encounter problems during the export or during the start of the exported application, check [Chapter 20, Common product export problems](#) for common problems and their solutions.

156.2. Select an update location

Choose a local file system path as the location for the update site. This path will be referred to as `REPOSITORY_LOC`.

156.3. Add dependencies

Add the plug-in dependencies from [Section 155.1, “Required plug-ins for updates”](#) to your application plug-in.

156.4. Add the p2 feature to the product

Add the `org.eclipse.equinox.p2.core.feature` feature to your product configuration file. Use the *Dependencies* tab for that.

156.5. Create a user interface

Create a new *Update* menu entry in your application. Implement a new handler for this menu entry.

See [Section 155.2, “Updating Eclipse RCP applications”](#) for an example implementation. Ensure to adjust the `REPOSITORY_LOC` constant in this code. It should point to your file location from [Section 156.2, “Select an update location”](#).

156.6. Enter a version in your product configuration file

Ensure that you have entered a version number for the product and to append the ".qualifier" suffix to the product version.

The screenshot shows the Eclipse Product Configuration Editor interface. The title bar says "todo.product". The main area is the "Overview" tab, which contains the following sections:

- General Information**: Describes the product. Fields include:
 - ID: [empty]
 - Version:** 1.0.0.qualifier (highlighted with a red border)
 - Name: to-do
- Product Definition**: Describes the launching product extension identifier and application.
 - Product: com.example.e4.rcp.todo.product
 - Application: org.eclipse.e4.ui.workbench.swt.E4Application
 - The product configuration is based on:
 - plug-ins
 - features
- Testing**:
 - Synchronize this configuration with the product's defining plug-in.
 - Test the product by launching a runtime instance of it:
 - [Launch an Eclipse application](#)
 - [Launch an Eclipse application in Debug mode](#)
- Exporting**:

Use the [Eclipse Product export wizard](#) to package and export the product defined in this configuration.

To export the product to multiple platforms:

 - Install the RCP delta pack in the target platform.
 - List all the required fragments on the [Dependencies](#) page.

At the bottom, there is a navigation bar with links: Overview | Dependencies | Configuration | Launching | Splash | Branding | Licensing.

156.7. Create the initial product export

Export your product via the *Eclipse Product export wizard* on the *Overview* tab of the product file.

Warning

Do not choose the `REPOSITORY_LOC` path as destination.

This application will be updated later.

156.8. Start the exported application and check for updates

Start your exported application from the export directory. Check for updates by invoking your update handler. Your handler should give you the feedback that no updates are available.

156.9. Make a change and export the product again

Change a (visible) label in your application and increment the product version on the overview page of the product editor to indicate a functional change.

Export your product again. Use a different export folder than you did before. You don't want to override the existing exported application. Make sure to select the *Generate p2 repository* option on the export dialog.

Warning

Again, do not export to the `REPOSITORY_LOC` path.

In the new export folder you find a sub-folder called *repository*. Copy this sub-folder to the `REPOSITORY_LOC` path.

156.10. Update the application

Start your exported application from [Section 156.7, “Create the initial product export”](#) and check again for updates via your menu entry. If everything was implemented correctly, your handler should report that updates are available. Install these updates and restart the application.

Tip

In case your handler does not find the update, restart your application to clear the caches of p2. p2 caches the meta information of an update side via a weak HashMap. So as long as the application is running and it has enough memory the information is cached.

Verify that all updates have been applied.

Part XXXVII. Using target platforms

Chapter 157. Target Platform

157.1. Defining available plug-ins for development

The set of plug-ins which you can use for your development or your build process to resolve the project dependencies is defined by the plug-ins in your workspace in addition with the plug-ins defined by your *target platform*.

For example your plug-ins use classes from the SWT and JFace plug-ins.

By default the plug-in installed in your Eclipse IDE installation are used as target platform.

157.2. Target platform definition

You can specify your target platform with a *target definition file*. With such a file you define the available plug-ins and features.

Tip

A target platform can be defined based on software sites (p2 update sites) or other means. As the new build system like Maven Tycho support only p2 update sites, it is recommended to use only these.

A target definition file is typically shared between the developers to ensure that everyone is using the same basis for development.

157.3. Using an explicit target definition file

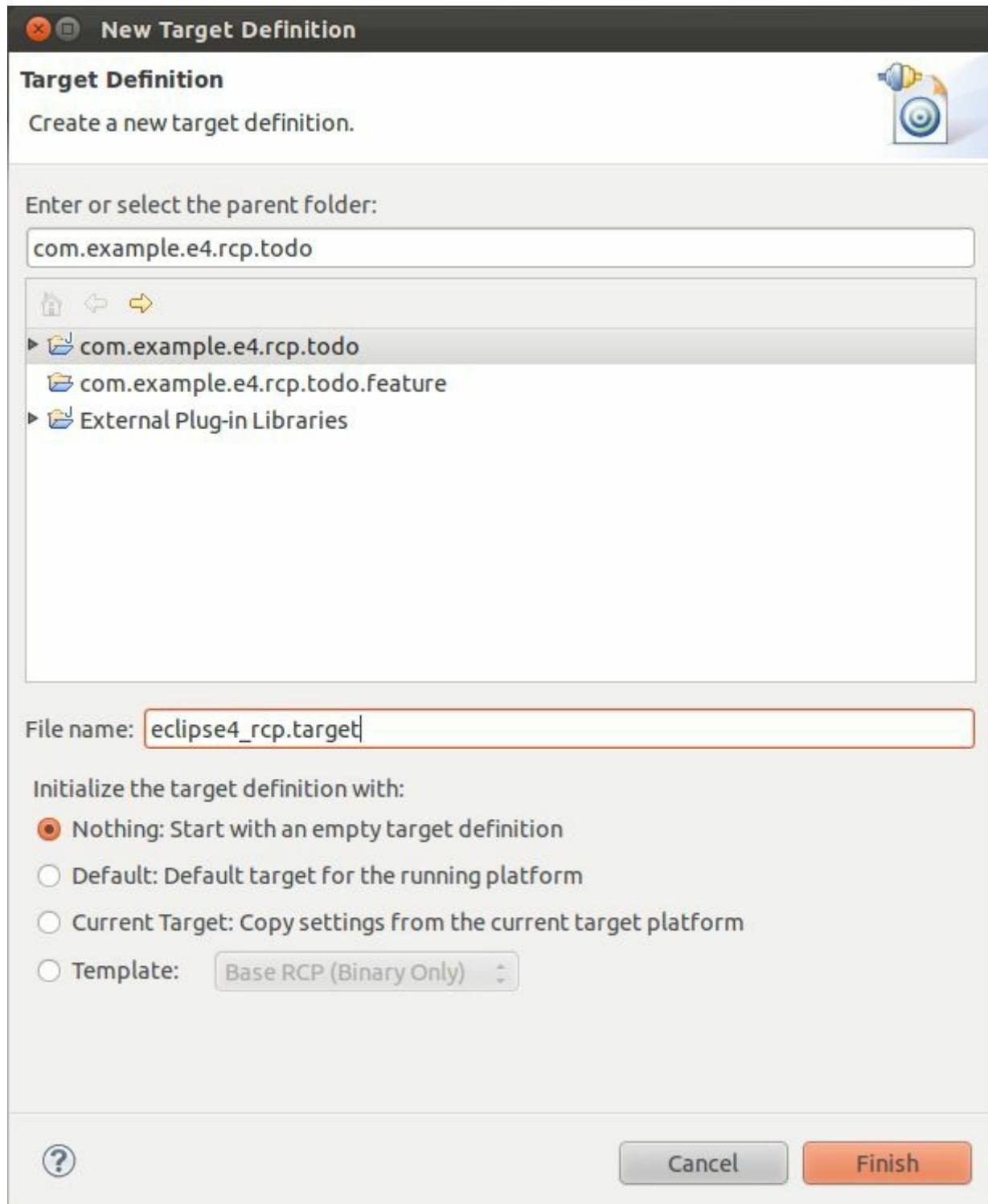
It is good practice to develop and build against a specific target definition. This way it can be ensured that dependencies and their versions doesn't change during the development. Via an explicit target definition it can also be ensured that all developers in a team are using the same dependencies and versions, rather than being dependent on the versions installed in the IDE of every developer.

Developing against your IDE installation has the following disadvantages:

- it makes you dependent on your version of the Eclipse IDE
- it can lead to problems if developers are using different versions of Eclipse because the API might be different
- it makes it difficult to upgrade the set of available plug-ins for every developer at the same time
- it also requires that you install every plug-in required for your product either in your workspace or in your Eclipse IDE
- it can happen that you might unintentionally add plug-ins from the Eclipse IDE to your development

157.4. Defining a target platform

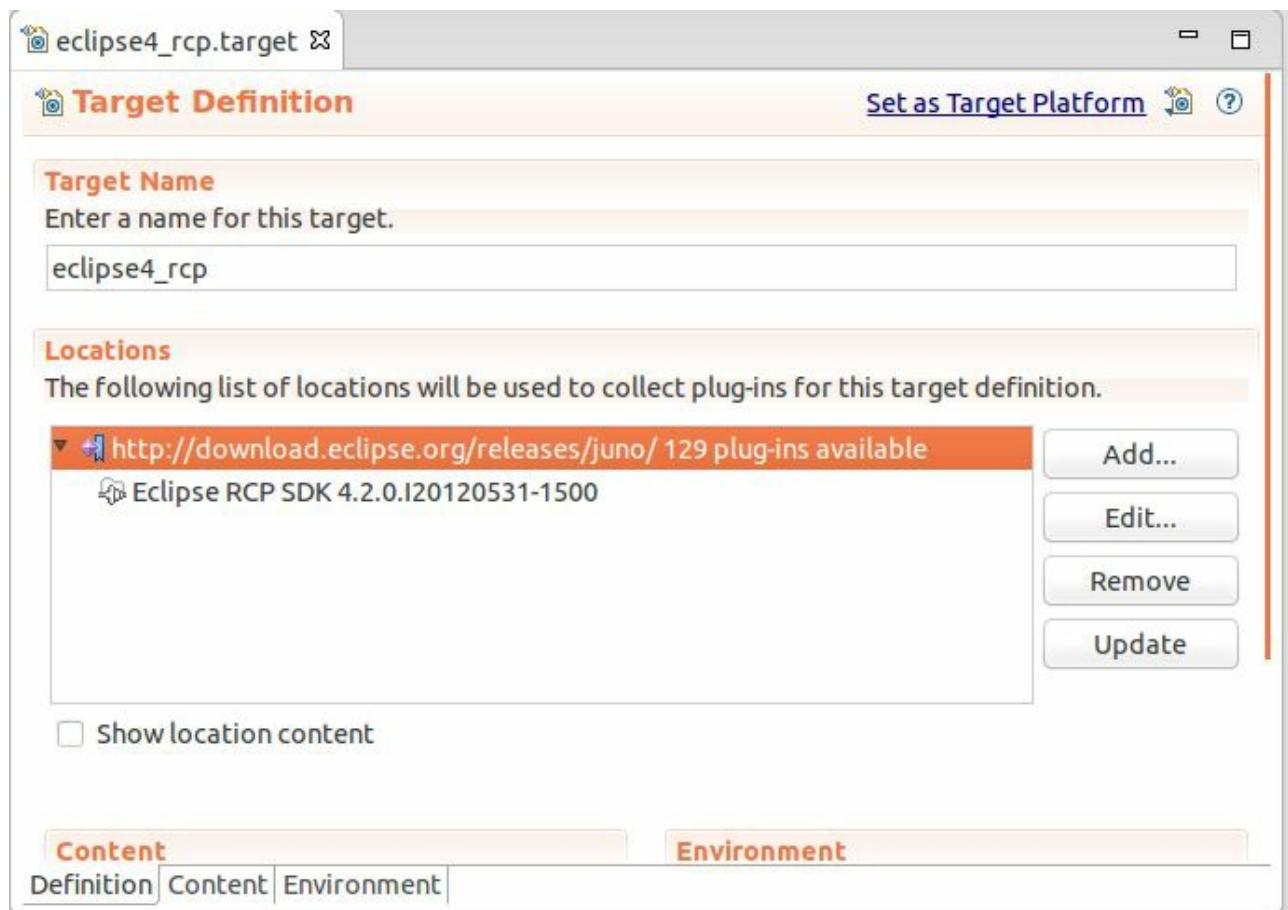
A target definition file can be created via File → New → Other... → Plug-in Development → Target Definition.



You can add new locations via the *Add...* button in the Locations section. To add

an Eclipse p2 update site, select *Software Site* and specify the URL.

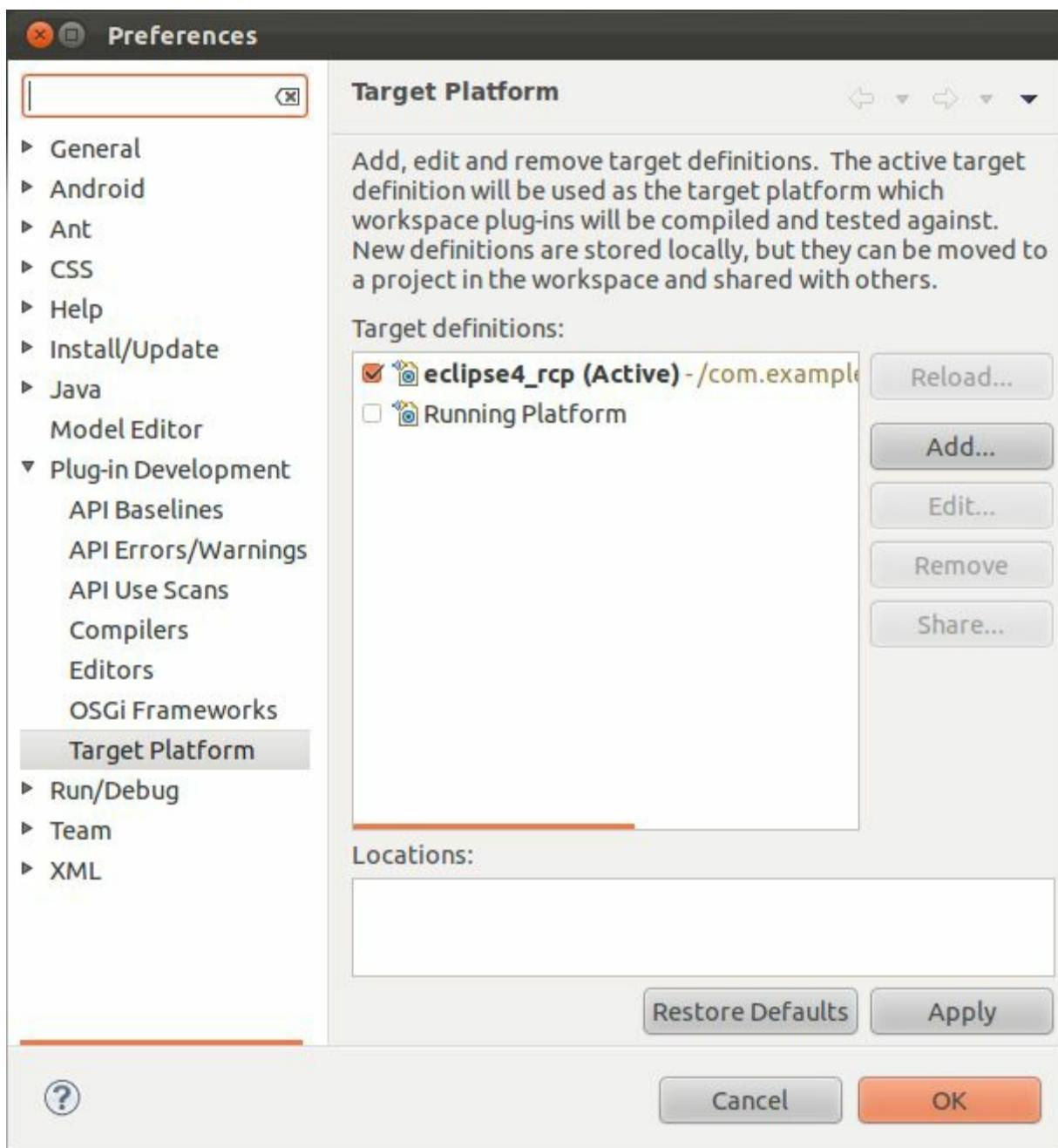
Once you have a target definition file, you can set it as the target platform in your Eclipse IDE. This can be done via the *Set as Target Platform* link in the Target definition editor as depicted in the following screenshot.



Tip

Wait until the target platform is completely resolved before setting it as target platform.

You can switch the target platform in the Eclipse Preferences. Select Window → Preferences → Plug-in Development → Target Platform.



The most effective way of defining your target platform is to use p2 update sites. These are of the same type as the update sites that you used to install a new set of plug-ins. If the content in the update sites defined by your target platform changes, your local set of plug-ins can be updated.

It is also possible to define your target platform based on plug-ins in your file system, but this is not recommended as certain build system like Maven/Tycho do not support file based target definition files.

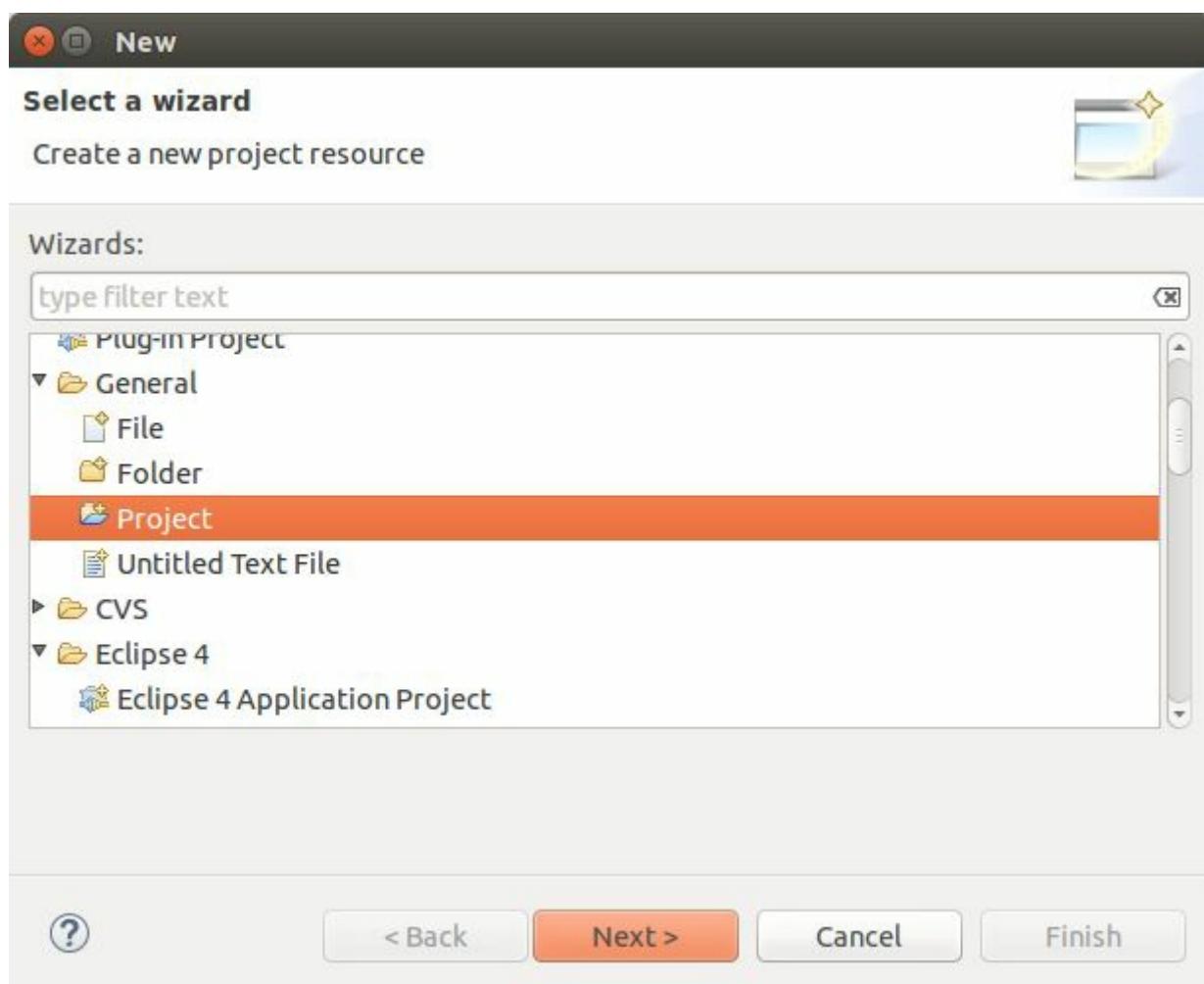
Chapter 158. Exercise: Defining a target platform

Note

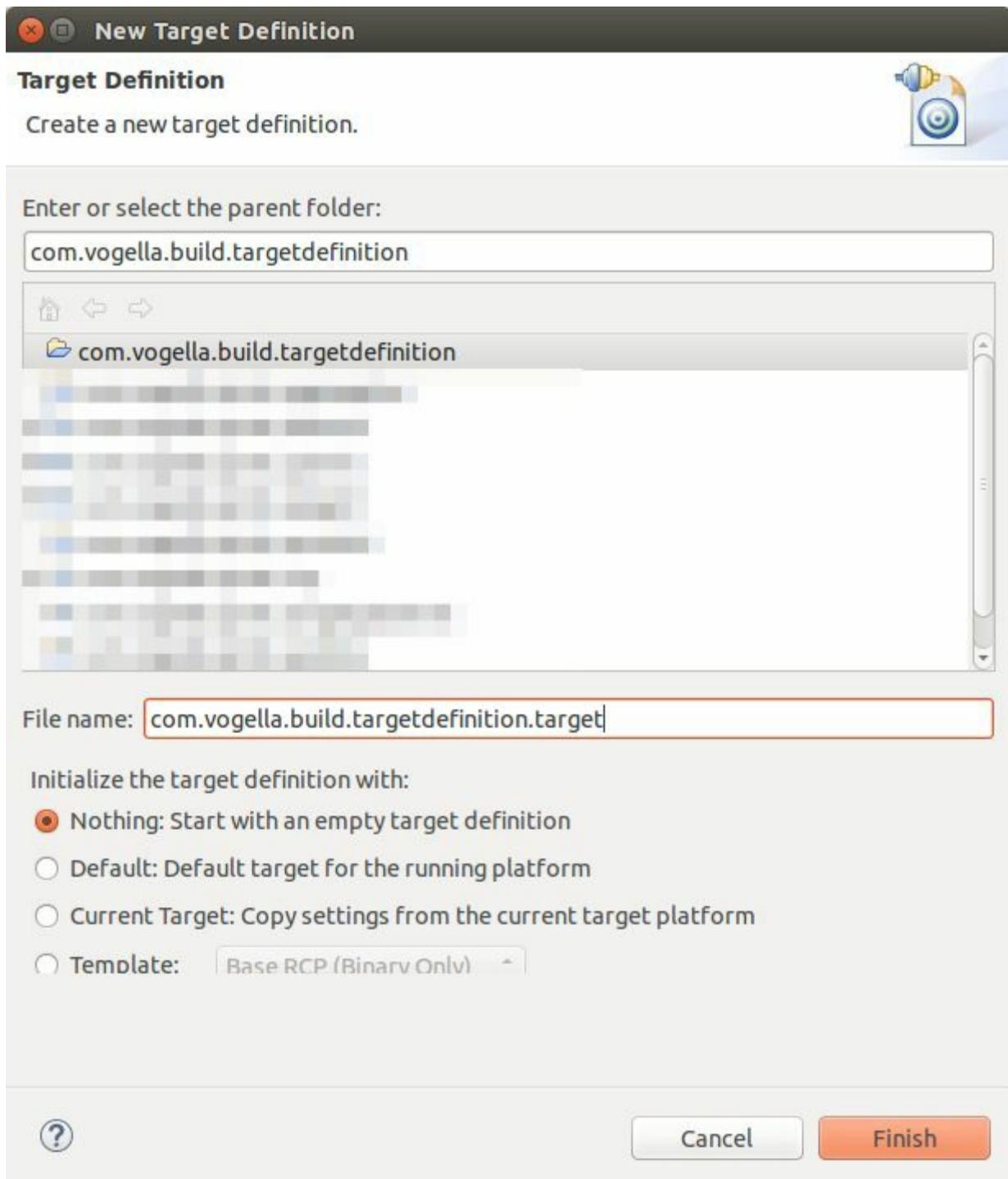
This exercise is optional.

158.1. Creating a target definition file

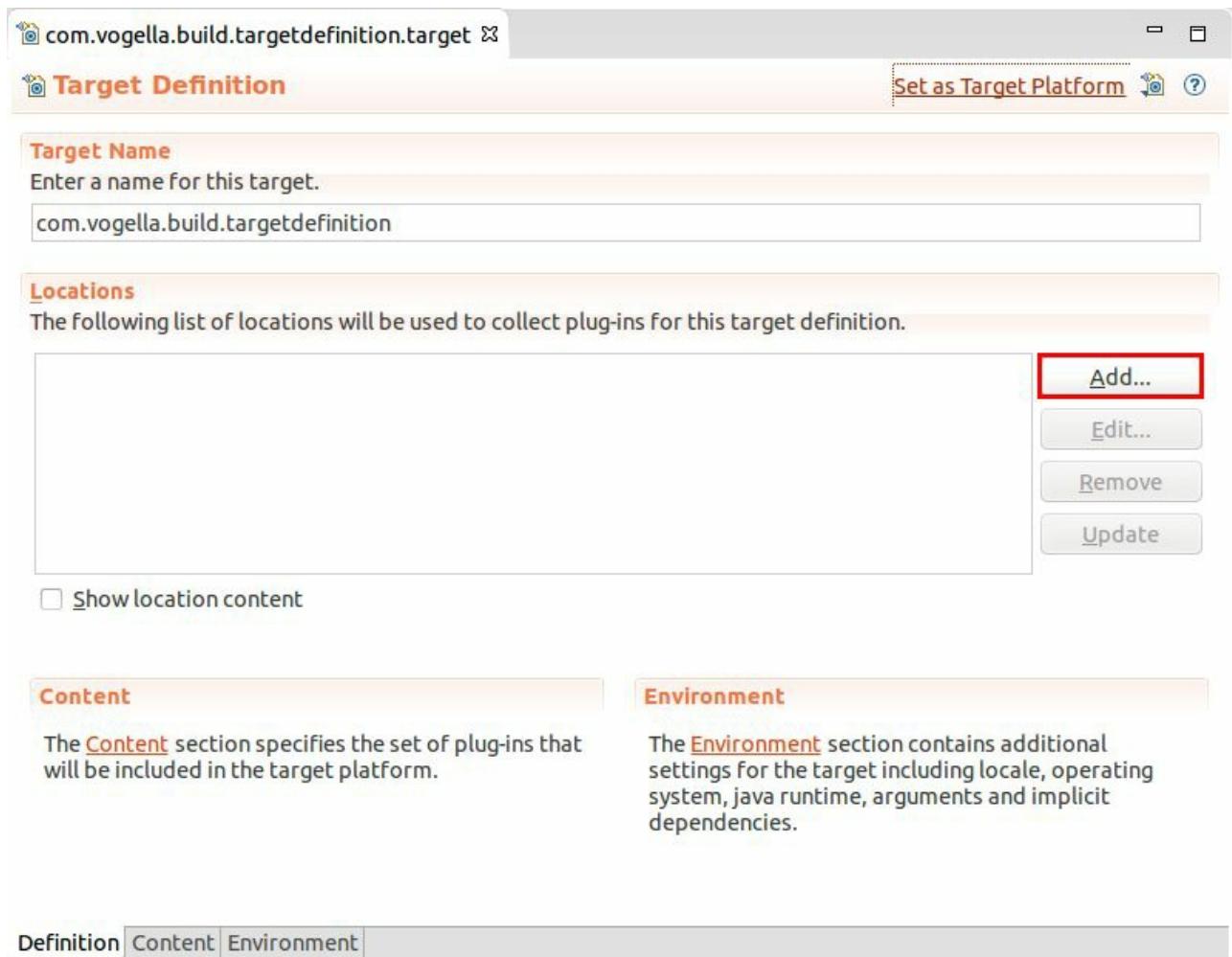
Create a new project called *com.vogella.build.targetdefinition* of type *General* via File → New → Other... → General → Project.



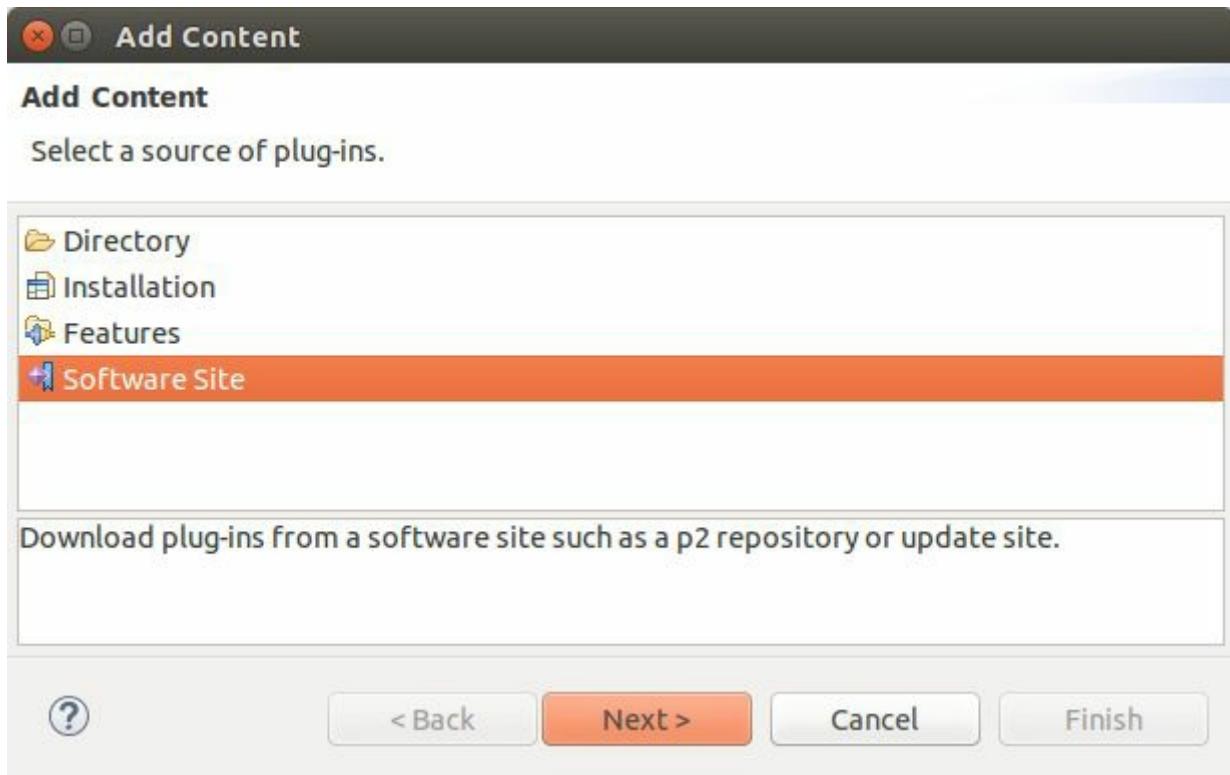
Create a new target definition file via the File → New → Other... → Plug-in Development → Target Definition menu path. Use the *com.vogella.build.targetdefinition.target* as file name.



Press the *Add...* button.



Select *Software site* in the following dialog.



Press the *Next* button and enter the <http://download.eclipse.org/eclipse/updates/4.4> URL in the Work with selection box. This is the update site for the Eclipse 4.4 (Luna) release.

Remove the *Group by Category* flag and add the following components:

Table 158.1. Target components

Component	Description
Eclipse RCP SDK	Components for RCP
Eclipse Platform Launcher Executables	The native launchers for the platform, ensure to select the non "black and white" entry if this exists. This entry is empty.
Eclipse JDT Plug-in	
Developer Resources	Required for JUnit
Equinox p2, Provisioning for IDEs	Update functionality

The result should look similar to the following screenshot. Please note that your version numbers might be different.

com.vogella.build.targetdefinition.target

Target Definition

[Set as Target Platform](#)

Target Name
Enter a name for this target.
com.vogella.build.targetdefinition

Locations
The following list of locations will be used to collect plug-ins for this target definition.

▼ http://download.eclipse.org/eclipse/updates/4.4_333 plug-ins available

- ↳ Eclipse JDT Plug-in Developer Resources 3.10.0.v20140606-1536
- ↳ Eclipse Platform Launcher Executables 3.6.100.v20140603-1326
- ↳ Eclipse RCP SDK 4.4.0.I20140606-1215
- ↳ Equinox p2, Provisioning for IDEs. 2.2.0.v20140523-0116

Show location content

Content
The [Content](#) section specifies the set of plug-ins that will be included in the target platform.

Environment
The [Environment](#) section contains additional settings for the target including locale, operating system, java runtime, arguments and implicit dependencies.

Definition Content Environment

Note

If you use SWTBot for unit tests, add *SWTBot for SWT Testing* features from <http://download.eclipse.org/technology/swtbot/releases/latest/> to your target platform.

Add Content

Add Software Site

Select content from a software site to be added to your target

Work with: <http://download.eclipse.org/technology/swtbot/releases/latest/> [Add...](#)

[Work with the list of software sites](#)

Name	Version
▼ SWTBot - API	
<input type="checkbox"/> SWTBot for Eclipse Forms Testing	2.2.1.201402241301
<input type="checkbox"/> SWTBot for Eclipse Testing	2.2.1.201402241301
<input type="checkbox"/> SWTBot for GEF Testing	2.2.1.201402241301
<input checked="" type="checkbox"/> SWTBot for SWT Testing	2.2.1.201402241301
▶ <input type="checkbox"/> SWTBot - Headless Support	
▶ <input type="checkbox"/> SWTBot - IDE Integration	
▶ <input type="checkbox"/> SWTBot - Sources	
▶ <input type="checkbox"/> SWTBot – Dependencies	

1 item selected

Details

SWTBot for testing SWT based applications

[Properties...](#)

Group by Category Show only the latest version

Included Software

By default, all required software is added to the target based on its environment settings. Turning this option off allows software to be added with missing requirements and multiple environments. This setting applies to the entire target definition.

Include required software

Include all environments

Include source if available

Include configure phase

[?](#) [< Back](#) [Next >](#) [Cancel](#) [Finish](#)

158.2. Activate your target platform for development

Afterwards press the *Set as Target Platform* to activate it. See [Section 158.4, “Solving potential issues for development”](#) in case you have problems with your new target platform.

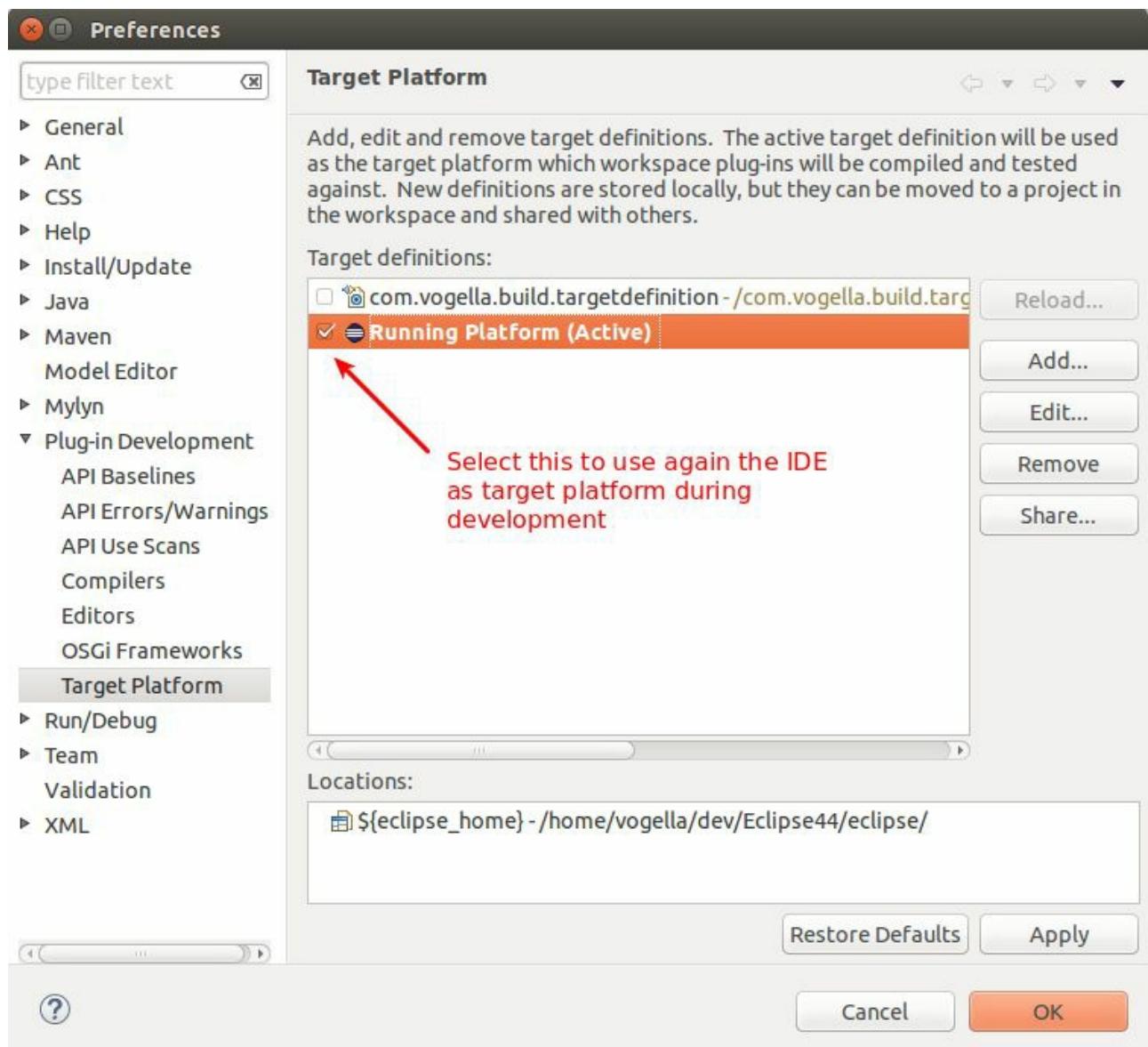
158.3. Validate that target platform is active

Try to add the `org.eclipse.pde.ui` plug-in as dependency to one of your plug-ins. This should not be possible.

158.4. Solving potential issues for development

Your target platform depends on external update sites and their packaging of plug-ins. These update sites might change over time, so you should be able to revert your target platform settings in case you face issues.

You can switch back to your Eclipse IDE as target platform via Window → Preferences → Plug-in Development → Target Platform.



Part XXXVIII. Using custom extension points

Chapter 159. Creating and evaluating extension points

159.1. Extensions and extension points

Eclipse provides the concept of *extensions* to contribute functionality to a certain type of API by one or more plug-ins. The type of API is defined by another plug-in as *extension point*.

These extensions and extension points are defined via the `plugin.xml` file.

This approach is depicted in the following graphic and the terminology is described in the following table.

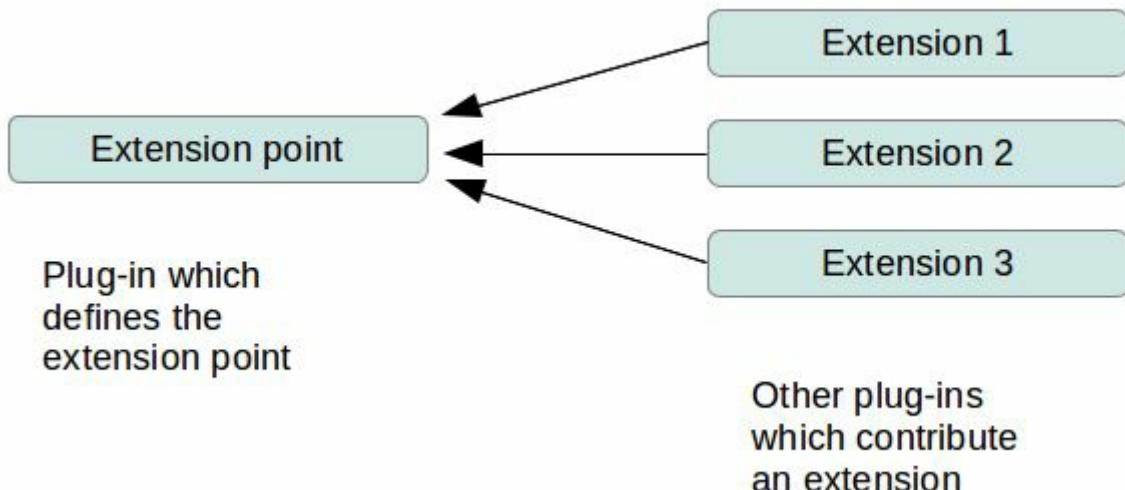


Table 159.1. Extensions and extension points table

Term	Description
Plug-in defines extension point	The plug-in defines a contract (API) with the definition of an extension point. This allows other plug-ins to add contributions (extensions) to the extension point. The plug-in which defines the extension point is also responsible for evaluating the extensions. Therefore, it typically contains the necessary code to do that.
A plug-in provides an extension	This plug-in provides an extension (contribution) based on the contract defined by an existing extension point. Contributions can be code or data.

159.2. Creating an extension point

A plug-in which declares an extension point must declare the extension point in its *plugin.xml* file. You use the *Extension Points* tab for the definition.

Extension points are described via an XML schema file which is referred to in the *plugin.xml* file. This XML schema file defines the details and the contract of the extension point. The following code demonstrates the link to the schema file in the *plugin.xml* file.

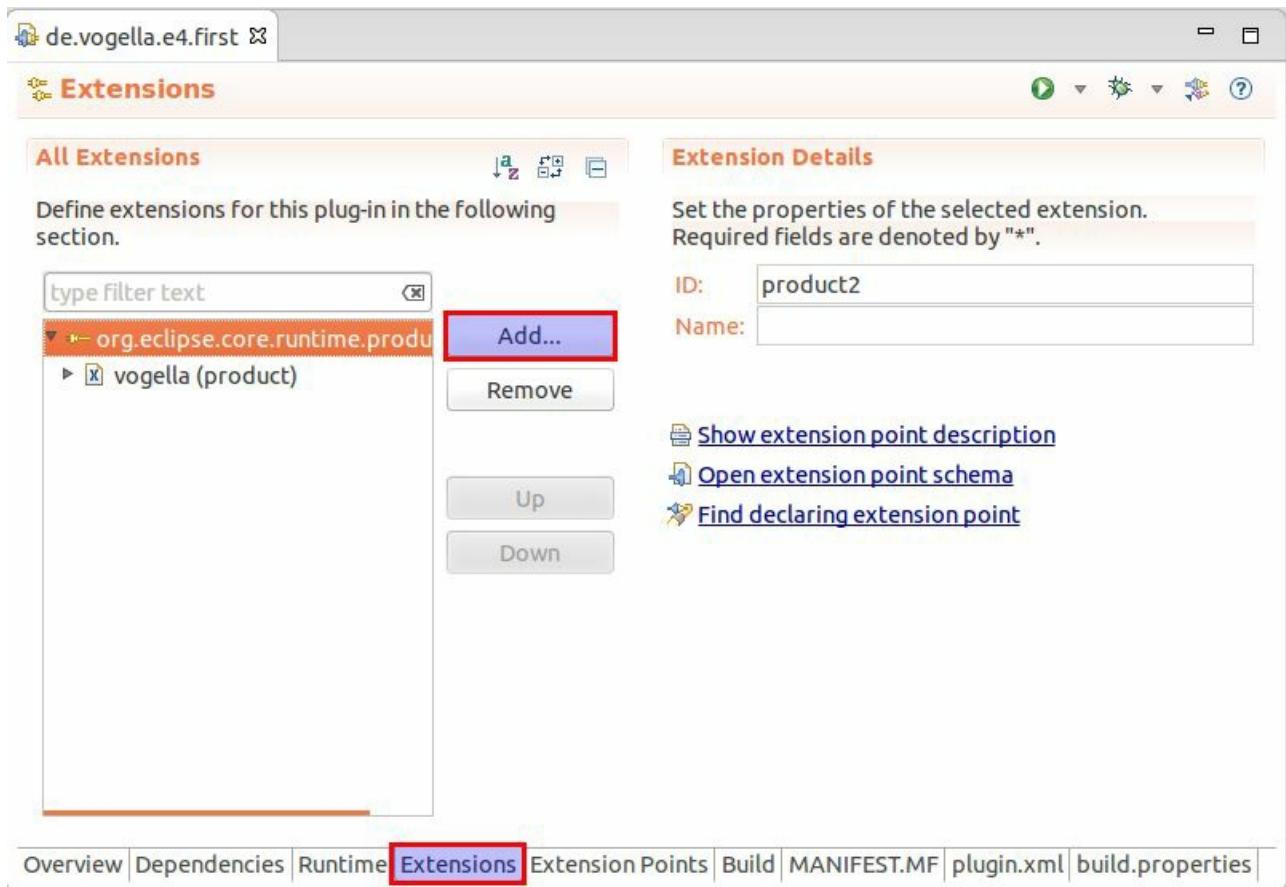
```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
    <extension-point id="de.vogella.extensionpoint.greeters"
        name="Greeters"
        schema="schema/de.vogella.extensionpoint.greeters.exsd" />
</plugin>
```

The plug-in typically contains code to evaluate the extensions for this extension point.

159.3. Adding extensions to extension points

The Eclipse IDE provides an editor for the `plugin.xml` file. It is the same editor which is used for modifying the `MANIFEST.MF` file.

Select the *Extensions* tab in this editor to create new extensions. On the *Extension Points* tab you can define new extension points.



By using the *Add...* button you can add a new extension. After adding one extension you can right-click on it to add elements or attributes.

159.4. Accessing extensions

The information about the available extension points and the provided extensions are stored in a class of type `IExtensionRegistry`.

Note

The Eclipse platform reads the extension points and provided extensions once the plug-in is in the *RESOLVED* life cycle status as defined by the OSGi specification.

In Eclipse applications you can use the dependency injection mechanism to get the `IExtensionRegistry` class injected.

```
@Inject
public void doSomething(IExtensionRegistry registry) {
    // do something
    registry.getConfigurationElementsFor("yourexextension");
}
```

In Eclipse 3.x based plug-ins you query for a certain extension via static methods of the `Platform` class. An example is listed in the following code.

```
// The following will read all existing extensions
// for the defined ID
Platform.getExtensionRegistry().
    getConfigurationElementsFor("yourexension");
```

Tip

You can access extensions of all extension points via the `IExtensionRegistry`.

159.5. Extension Factories

If you create your class directly via the `IConfigurationElement` class of the extension you are limited to classes which have a default constructor. To avoid this restriction you can use extension factories.

To use a factory implement the interface `IExecutableExtensionFactory` in the class attribute of your extension point definition. The factory receives the configuration elements and can construct the required object.

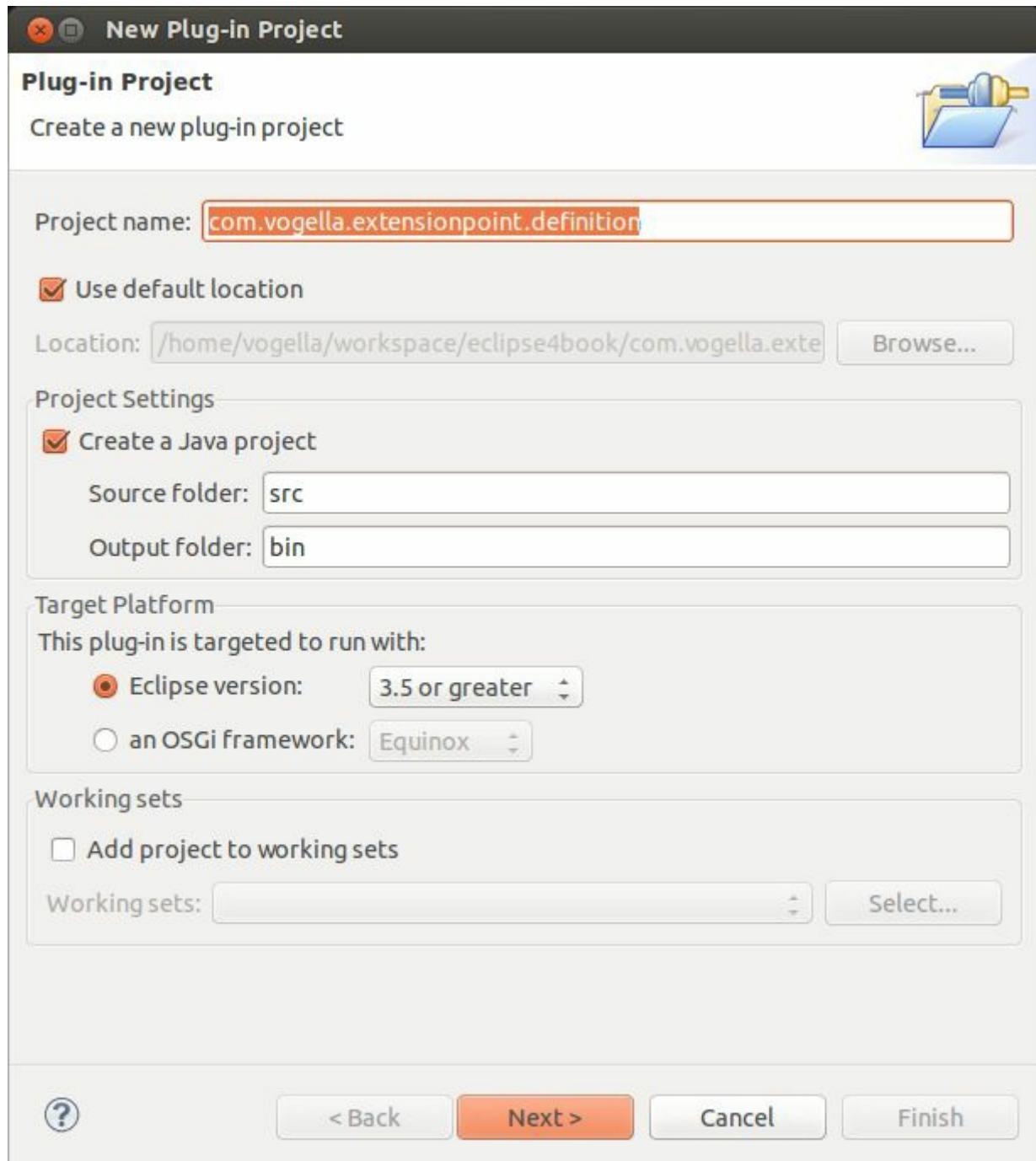
Chapter 160. Exercise: Create and evaluate extension point

160.1. Target for this exercise

In this exercise you create two new plug-ins. The first one contains an extension point and the second contributes an extension to this new extension point. This exercise is to demo the creation and usage of extension points. The result is not intended to be a realistic feature.

160.2. Creating a plug-in for the extension point definition

Create a new plug-in project called `com.vogella.extensionpoint.definition` based on the settings of the following screenshot.



New Plug-in Project

Content

Enter the data required to generate the plug-in.

Properties

ID: com.vogella.extensionpoint.definition

Version: 1.0.0.qualifier

Name: Definition

Vendor: VOGELLA

Execution Environment: JavaSE-1.7

Options

Generate an activator, a Java class that controls the plug-in's life cycle
Activator: com.vogella.extensionpoint.definition.Activator

This plug-in will make contributions to the UI

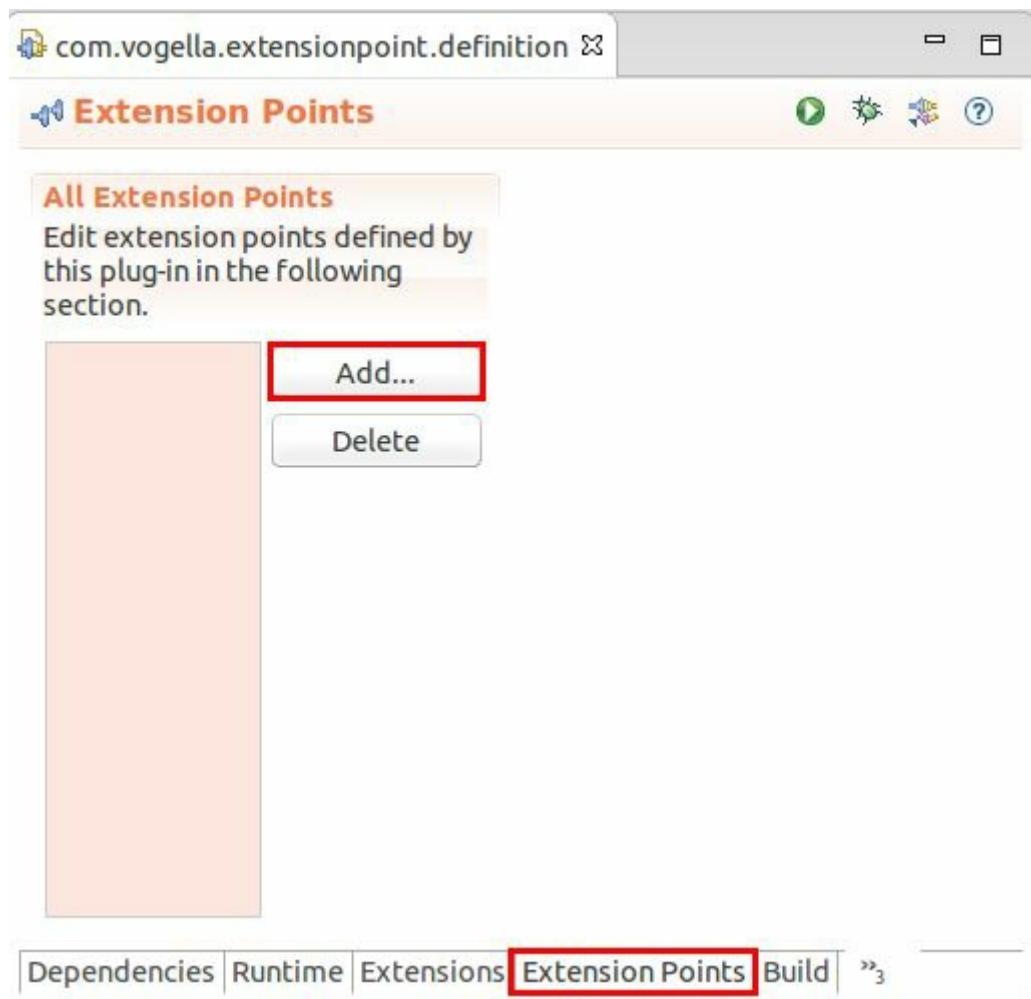
Enable API analysis

Rich Client Application

Would you like to create a 3.x rich client application? Yes No

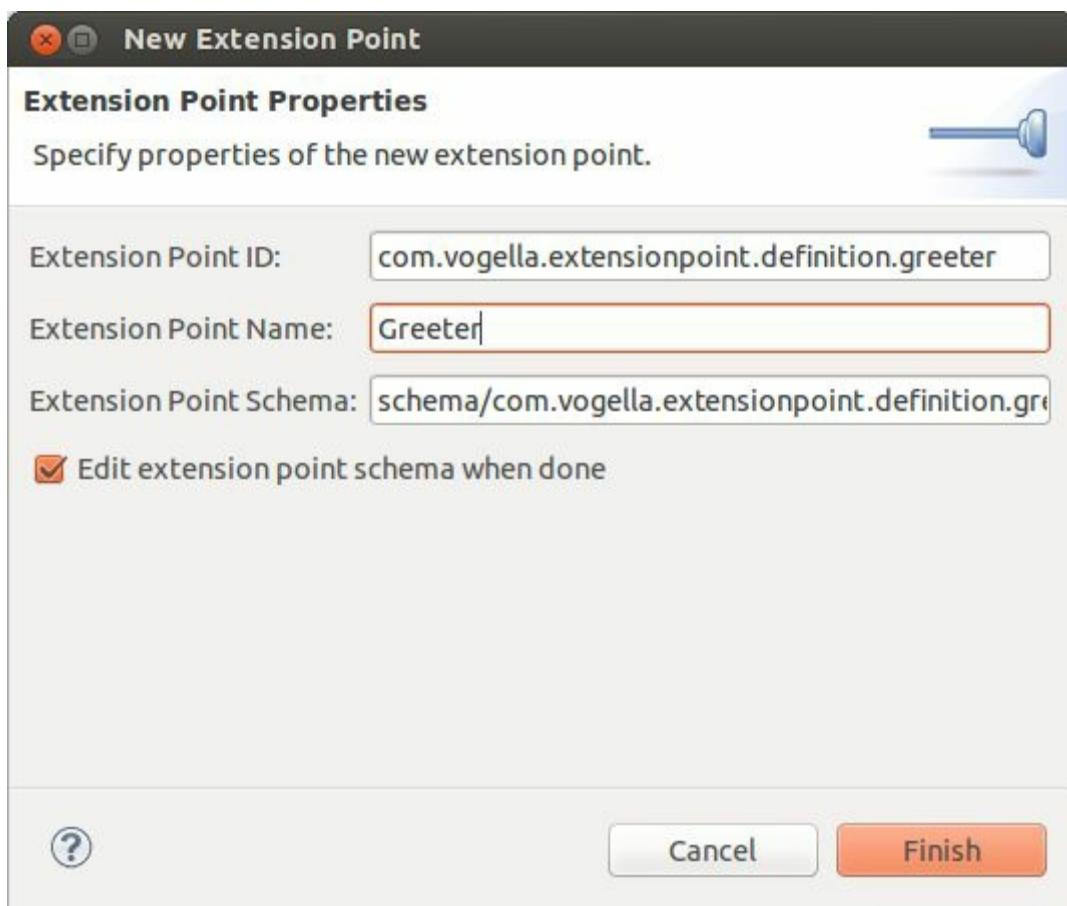
160.3. Create an extension point

Open the `MANIFEST.MF` file or the `plugin.xml` file and select the *Extension Points* tab.



Press the *Add...* button.

Enter "com.vogella.extensionpoint.definition.greeter" as *Extension Point ID* and "Greeter" as *Extension Point Name* in the dialog. The *Extension Point Schema* field is automatically populated based on your input.



Press the *Finish* button. The definition of the extension is generated and the schema editor opens. Switch to the *Definition* tab.

The screenshot shows the Eclipse XML Editor interface for defining an extension point. The title bar says "com.vogella.extensionpoint.definition.greeter.exsd". The left panel, titled "Extension Point Elements", lists an "extension" element under "Greeter". On the right, the "Extension Element Details" panel shows properties for the "extension" element:

- Name: extension
- Internal: true false
- Deprecated: true false
- Replacement: [empty]
- Description: [empty]

The "DTD Approximation" panel shows "EMPTY". At the bottom, there are tabs: Overview, **Definition**, Source.

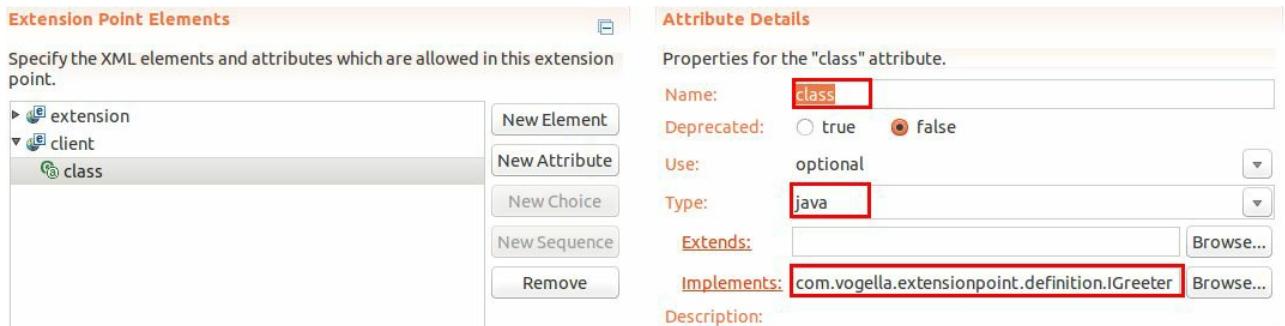
Add attributes to the extension point. For that click on the *New Element* button. Give the new element the name "client".

The screenshot shows the Eclipse XML Editor interface after adding a "client" element to the extension point. The title bar says "com.vogella.extensionpoint.definition.greeter.exsd". The left panel now lists both "extension" and "client" elements. The "Element Details" panel for the "client" element shows:

- Name: client
- Deprecated: true false
- Translatable: true false
- Description: [empty]

The "DTD Approximation" panel shows "(#PCDATA)". At the bottom, there are tabs: Overview, Definition, Source.

Select the "client" element and press *New Attribute*. Give the new element the name "class" and the type "java". Enter com.vogella.extensionpoint.definition.IGreeter as interface name

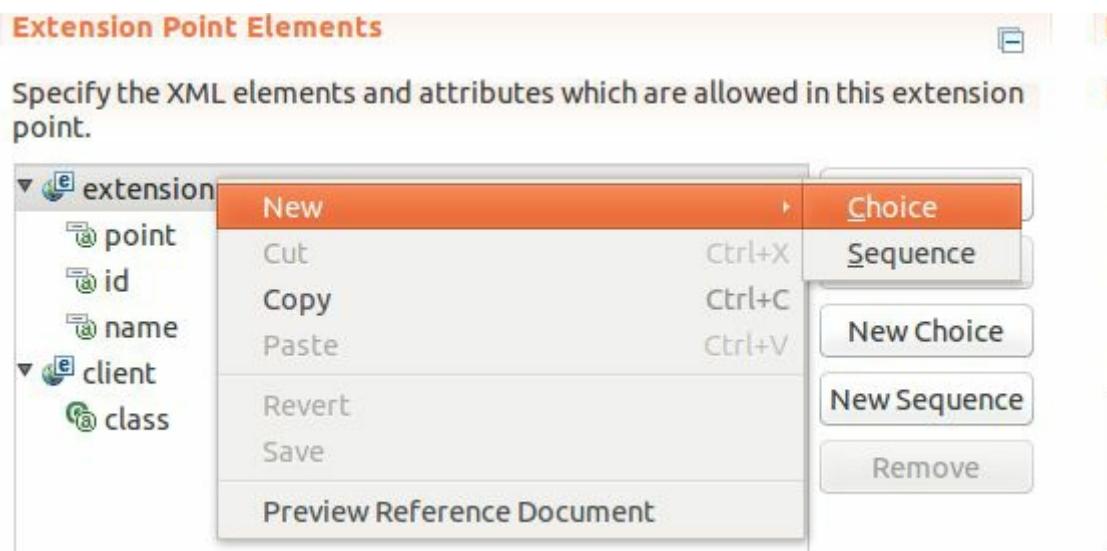


This interface doesn't exist yet. Press on the hyperlink called *Implements* to create it based on the following code.

```
package com.vogella.extensionpoint.definition;

public interface IGreeter {
    void greet();
}
```

Go back to your extension point definition and add a choice to the extension point. For this select the *extension* entry, right-click on it and select *New → Choice*. This defines how often the extension "client" can be provided by contributing plug-ins. We will set no restrictions (unbound).



com.vogella.extensionpoint.definition.greeter.exsd

Greeter

[Preview Reference Document](#)

Extension Point Elements

Specify the XML elements and attributes which are allowed in this extension point.

- extension
 - Choice (1 - *)
 - point
 - id
 - name
- client
 - class

New Element
New Attribute
New Choice
New Sequence
Remove

Composer Details

Properties for the Choice compositor.

Min Occurrences: 1
Max Occurrences: 1 Unbounded
Type: choice

DTD Approximation

(EMPTY+)

Overview Definition Source

Extension Point Elements

Specify the XML elements and attributes which are allowed in this extension point.

- extension
 - Choice (1 - *)
 - point
 - id
 - name
- client
 - class

New Element

New

- Choice
- Sequence
- choice
- client

Min Occurrences: 1
Max Occurrences: 1 Unbounded
Type: choice

com.vogella.extensionpoint.definition.greeter.exsd ✎

Greeter

[Preview Reference Document](#) [?](#)

Extension Point Elements

Specify the XML elements and attributes which are allowed in this extension point.

- extension
 - Choice (1 - *)
 - client
 - point
 - id
 - name
 - client
 - class

New Element
New Attribute
New Choice
New Sequence
Remove

Element Reference Details

Properties for the "client" element reference.

Min Occurrences: 1

Max Occurrences: 1 Unbounded

Reference: [client](#)

DTD Approximation

EMPTY

160.4. Export the package

Select the `MANIFEST.MF` file switch to the *Runtime* tab and export the package which contains the `IGreeter` interface.



160.5. Add dependencies

Add the following dependencies via the *MANIFEST.MF* file of your new plug-in:

- org.eclipse.core.runtime
- org.eclipse.e4.core.di

.

160.6. Evaluating the registered extensions

The defining plug-in also evaluates the existing extension points. In the following example you create a handler class which can evaluate the existing extensions.

Create the following class.

```
package com.vogella.extensionpoint.definition;

import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IConfigurationElement;
import org.eclipse.core.runtime.IExtensionRegistry;
import org.eclipse.core.runtime.ISafeRunnable;
import org.eclipse.core.runtime.SafeRunner;
import org.eclipse.e4.core.di.annotations.Execute;

public class EvaluateContributionsHandler {
    private static final String IGREETER_ID =
        "com.vogella.extensionpoint.definition.greeter";
    @Execute
    public void execute(IExtensionRegistry registry) {
        evaluate(registry);
    }

    private void evaluate(IExtensionRegistry registry) {
        IConfigurationElement[] config =
            registry.getConfigurationElementsFor(IGREETER_ID);
        try {
            for (IConfigurationElement e : config) {
                System.out.println("Evaluating extension");
                final Object o =
                    e.createExecutableExtension("class");
                if (o instanceof IGreeter) {
                    executeExtension(o);
                }
            }
        } catch (CoreException ex) {
            System.out.println(ex.getMessage());
        }
    }

    private void executeExtension(final Object o) {
        ISafeRunnable runnable = new ISafeRunnable() {
            @Override
            public void handleException(Throwable e) {
                System.out.println("Exception in client");
            }
        };

        @Override
        public void run() throws Exception {
            ((IGreeter) o).greet();
        }
    }
}
```

```
        }
    };
    SafeRunner.run(runnable);
}
}
```

The code above uses the `ISafeRunnable` interface. This interface protects the plug-in which defines the extension point from malfunction extensions. If an extension throws an `Exception`, it will be caught by `ISafeRunnable` and the remaining extensions will still get executed.

Review the `.exsd` schema file and the reference to this file in the `plugin.xml` file.

160.7. Create a menu entry and add it to your product

Add a dependency to the `com.vogella.extensionpoint.definition` plug-in in the `MANIFEST.MF` file of your application plug-in.

Afterwards create a new menu entry called *Evaluate extensions* and define a command and handler for this menu entry. In the handler point to the `EvaluateContributionsHandler` class.

Also update your product (via your features) so that the new plug-in is part of your application.

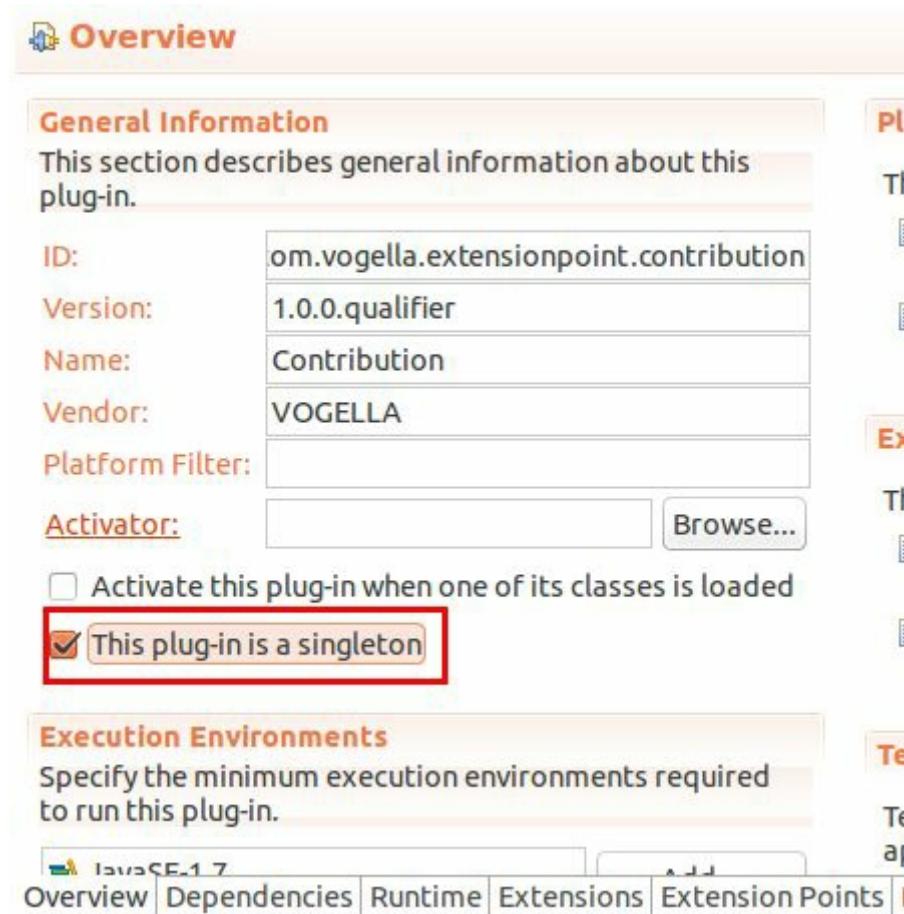
Note

A better approach would be to add the menu entry via a model fragment or a model processor, but I leave that as an additional exercise to the reader.

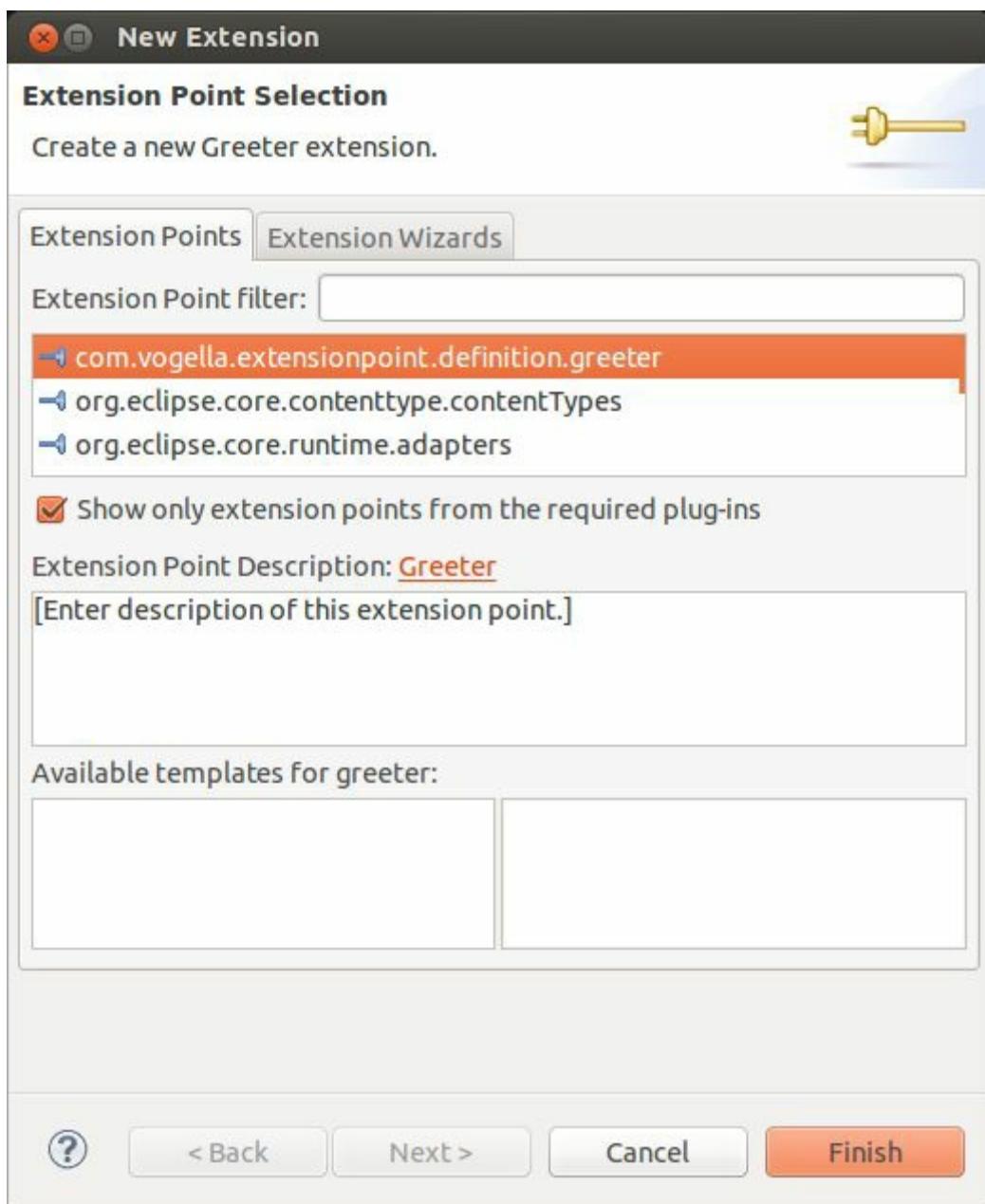
160.8. Providing an extension

Create a new simple plug-in called `com.vogella.extensionpoint.contribution`.

Open the `MANIFEST.MF` editor of this new plug-in and select the *Dependencies* tab. Add the `com.vogella.extensionpoint.definition` and `org.eclipse.core.runtime` plug-ins as dependencies. Make sure your plug-in has the *This plug-in is a singleton* flag set on the *Overview* tab for your `MANIFEST.MF` file.



Switch to the *Extensions* tab and select *Add....* Select your custom extension point and press the *Finish* button.



Add a client to your extension point via right-click.

The screenshot shows the 'All Extensions' view under the 'Extensions' tab. It lists 'com.vogella.extensionpoint.definition.greeter' and includes a 'type filter text' input and sorting icons. A context menu is open over the listed extension, with options 'New' and 'Remove'. The 'New' option is highlighted in orange. To the right, there is an 'Extension Details' section with a note about setting properties for selected fields. A 'client' entry is visible in the details area.

Extension Element Details

Set the properties of "client". Required fields are denoted by "*".

class: com.vogella.extensionpoint.contribution.GreeterGerman

[Browse...](#)

Create the GreeterGerman class with the following code.

```
package com.vogella.extensionpoint.contribution;

import com.vogella.extensionpoint.definition.IGreeter;

public class GreeterGerman implements IGreeter {

    @Override
    public void greet() {
        System.out.println("Moin, moin!");
    }
}
```

160.9. Add the plug-in to your product

Add the `com.vogella.extensionpoint.definition` plug-in to your product (via your features).

160.10. Validating

Start your application via the product configuration file to update the plug-ins included in the run configuration.

In your running application select your new menu entry. The handler class write the output of your extensions to the *Console* view of your Eclipse IDE.

Part XXXIX. Eclipse styling with CSS

Chapter 161. Introduction to CSS styling

161.1. Cascading Style Sheets (CSS)

What is CSS?

Cascading Style Sheets (CSS) allow you to define styles, layouts and spacing separate from the content which should be styled.

The CSS information is typically contained in an external file. Other code, e.g., an HTML page, can reference to the CSS file for its layout information. For example you define in an external file the fonts and colors used for certain elements.

CSS is defined as a standard. Currently the versions *CSS 2.1* and *CSS 3* are most widely used.

CSS selectors and style rules

The CSS standard defines *selectors* and *style rules*. The syntax is defined as follows:

```
selector { property:value; }
```

A *selector* can be one of a predefined identifier(, e.g., H1), a class name (e.g. .myclass) or an identifier (e.g. #myuniqueid).

In CSS an identifier is supposed to be unique across all of the elements in a page (or window in our case) while a class can be assigned to several elements.

For example the following CSS file defines the size and color of the h1 tag.

```
h1 { color:red; font-size:48px; }
```

CSS pseudo classes

CSS pseudo classes are used to qualify attributes of selectors. For example you can select an visited link in HTML and style is differently.

```
a:visited { color:red; }
```

161.2. Styling Eclipse applications

The visual appearance of your Eclipse application can be styled via external files similar to the CSS 2.1 standard.

CSS selectors used in Eclipse are identifiers, which relate to widgets or other elements, for example predefined pseudo classes. Non-standard properties are prefixed with either *swt-* or *eclipse-*.

The following example shows a CSS file which defines a styling for an Eclipse application.

```
Label {
    font: Verdana 8px;
    color: black;
}
Composite Label {
    color: black;
}
Text {
    font: Verdana 8px;
}
/* Identifies only those SWT Text elements
appearing within a Composite */
Composite Text {
    background-color: white;
    color: black;
}
SashForm {
    background-color: #c1d5ef;
}
/* background uses a gradient */
.MTrimBar {
    background-color: #e3efff #c1d5ef;
    color: white;
    font: Verdana 8px;
}

Shell {
    background-color: #e3efff #c1d5ef 60%;
}
```

The Eclipse platform supports static as well as dynamic styling of an application. In the case of static styling the styling cannot be exchanged at runtime, while in the case of dynamic styling it is possible to switch the styling at runtime.

161.3. Limitations

SWT has certain limitations concerning component styling. These limitations are based on the restrictions of the underlying operating system or limitation in SWT itself. For example it is not possible to style menus and table headers because this is not supported by the SWT implementation. Also, some platforms do not allow the styling of certain widgets, e.g. the `Button` or the `ScrollBar` widget.

Chapter 162. How to style in Eclipse

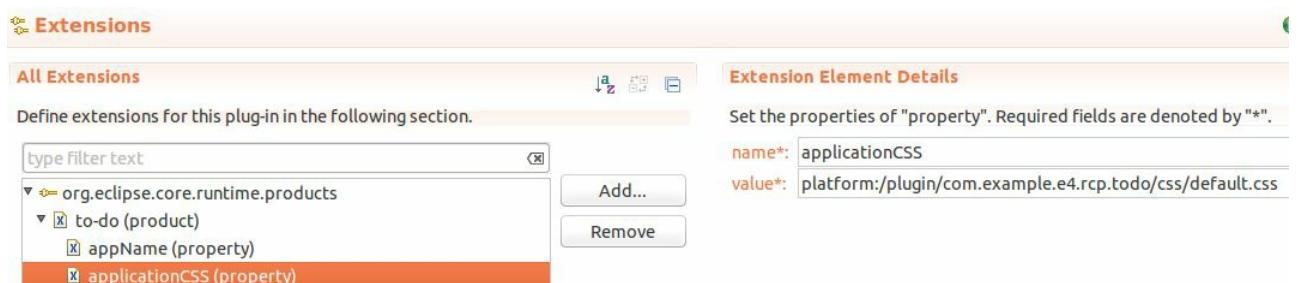
162.1. Fixed styling in Eclipse

You can specify a fixed styling of your application via a property in your extension for the `org.eclipse.core.runtime.products` extension point. Like all extensions this extension is contributed via the `plugin.xml` file.

The value of the `applicationCSS` property should point to your CSS file via an URI. The URI follows the `platform:/plugin/BundleSymbolicName/path/file` convention. The following example demonstrates such a URI:

```
platform:/plugin/com.example.e4.rcp.todo/css/default.css
```

The screenshot shows an example of the `plugin.xml` file with a defined `applicationCSS` property.



The corresponding file is shown below. It leaves out possible other properties you have set, e.g., for a life cycle handler.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>

<extension
    id="product"
    point="org.eclipse.core.runtime.products">
<product
    application="org.eclipse.e4.ui.workbench.swt.E4Application"
    name="to-do">
<property
    name="appName"
    value="to-do">
</property>
<property
    name="applicationCSS"
    value="platform:/plugin/com.example.e4.rcp.todo/css/default.css">
```

```
</property>
</product>
</extension>

</plugin>
```

162.2. Dynamic styling using themes

You can define multiples *themes* via an extension in the `plugin.xml` file. The Eclipse platform provides the theme service with an instance of the `IThemeEngine` interface. Via this instance you can change the active theme at runtime.

To create new themes you define extensions for the `org.eclipse.e4.ui.css.swt.theme` extension point. Such an extension defines an ID for the style and a pointer to the CSS file.

You must also define the default theme via the `cssTheme` property in your `org.eclipse.core.runtime.products` extension. This can also be used to define a fixed styling.

Warning

If `cssTheme` is not set, the `IThemeEngine` service is not registered as service by the Eclipse platform.

The switching of themes is demonstrated in the following handler code.

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

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.css.swt.theme.IThemeEngine;

public class ThemeSwitchHandler {
    private static final String DEFAULT_THEME = "com.example.e4.rcp.todo.default";
    private static final String RAINBOW_THEME = "com.example.e4.rcp.todo.rainbow";

    @Execute
    public void switchTheme(IThemeEngine engine) {
        if (!engine.getActiveTheme().getId().equals(DEFAULT_THEME)) {
            // second argument defines that change is
            // persisted and restored on restart
            engine.setTheme(DEFAULT_THEME, true);
        } else {
            engine.setTheme(RAINBOW_THEME, true);
        }
    }
}
```

Chapter 163. More details on Eclipse styling

163.1. CSS attributes and selectors

The CSS attributes for SWT widgets which can be styled are listed under the following link:

http://wiki.eclipse.org/E4/CSS/SWT_Mapping

Eclipse application model elements, e.g., *MPartStack* or *MPart* can also be styled. The CSS selectors are based on the Java classes for the model elements. Some examples are given in the following table.

Table 163.1. Model elements and CSS selectors

Model Element	CSS Selector
MPart	.MPart
MPartStack	.MPartStack
MPartSashContainer	.MPartSashContainer

For example you could hide the minimize and maximize button of a *MPartStack* via the following CSS rule.

```
.MPartStack {  
    swt-maximize-visible: false;  
    swt-minimize-visible: false;  
}
```

Eclipse also supports *CSS pseudo classes*. The following table lists several of these pseudo classes.

Table 163.2. CSS pseudo classes in Eclipse

CSS pseudo classes	Description
Button:checked	Selects checked buttons
:active	For example shell:active selects the active shell\
:selected	Allows to style a selected element, e.g., a part in a PartStack.

163.2. Styling based on identifiers and classes

You can specify an identifier or a class on widgets in your source code and use these as selectors for styling. An identifier is supposed to be unique while a class can be assigned to several elements.

The following example demonstrates how to set the identifier and the class on SWT widgets.

```
// IStylingEngine is injected
@Inject IStylingEngine engine;

// more code.....

Label label = new Label(parent, SWT.NONE);
Text text = new Text(parent, SWT.BORDER);

// set the ID, must be unique in the same window
engine.setID(label, "MyCSSTagForLabel");

// set the class, can be used several times
engine.setClassname(text, "error");
```

Sometimes the `IStylingEngine` cannot be accessed easily, for example if you want to style existing dialog which are not created via dependency injection. In this case you can set a tag on SWT widget directly.

```
// more code.....

Label label = new Label(parent, SWT.NONE);
Text text = new Text(parent, SWT.BORDER);

// set the ID, must be unique in the same window
text.setData("org.eclipse.e4.ui.css.id", "MyCSSTagForLabel");

// set the class, can be used several times
label.setData("org.eclipse.e4.ui.css.CssClassName", "MyCSSTagForLabel")
```

These ids or classes can be addressed in the CSS file via the `#` or the `.` selector.

```
#MyCSSTagForLabel{
    color: blue;
}

.error {
    border: 1px red;
}
```

163.3. Colors and gradients

Colors can be defined in different ways, e.g., the color white can be described as `white`, `rgb(255, 255, 255)` or `#ffffff`.

Styling supports gradients for the background color of user interface widgets. Linear and radial gradients are supported. The definition of gradients is demonstrated in the following listing.

```
// linear gradient
// background-color: gradient linear color_1 [color_2]* [percentage]*
// for example
background-color: gradient linear white black
background-color: gradient linear white black blue
background-color: gradient linear white black 50%
background-color: gradient linear white black 50% 20%

// radial gradient
// background-color: gradient radial color_1 [color_2]* [percentage]*
// for example
background-color: gradient radial white black
background-color: gradient radial white black 50%
```

If you use the optional percentage in the definition, then the color change will be done after the defined percentage. The following picture shows the result of different settings.

linear orange black



linear orange black
60%



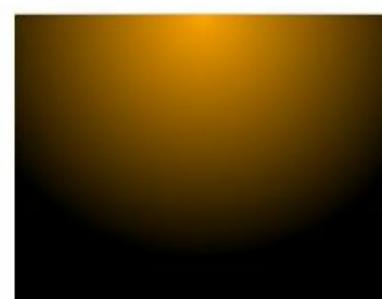
linear orange black
orange 50% 100%



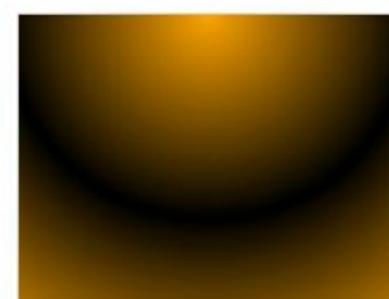
radial orange black



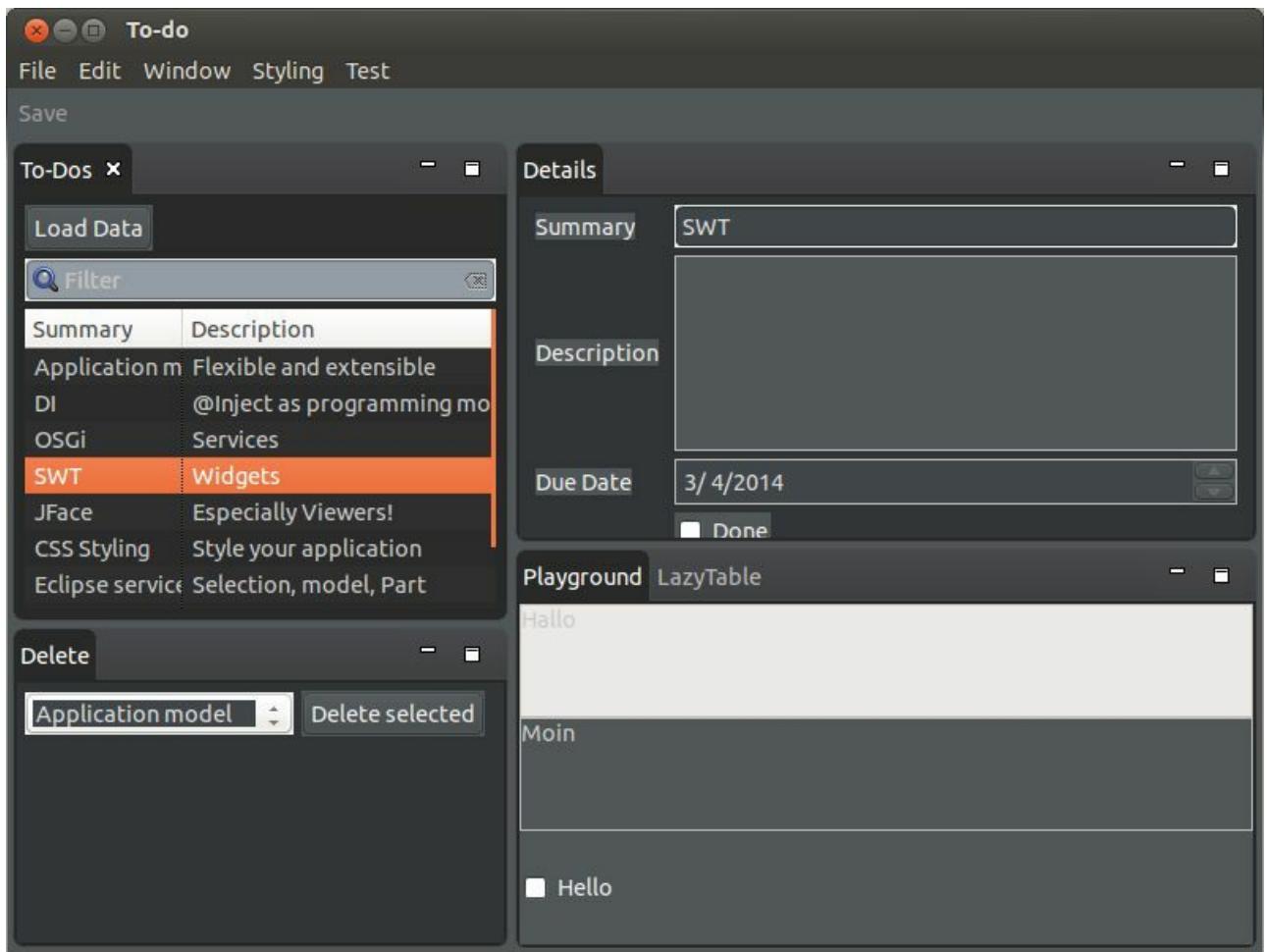
radial orange black
60%



radial orange black
orange 50% 100%



Using this import results in an application styling similar to the following screenshot.



Note

Please note that the current gradient specification is not compliant with the CSS specification and that it might change in the future.

163.4. CSS imports

Eclipse supports also the import of existing stylesheets via the `@import` directive using the platform URI notation.

For example the following stylesheet would import the existing dark theme of the `org.eclipse.ui.themes` plug-in from the Eclipse IDE.

```
@import url("platform:/plugin/org.eclipse.ui.themes/css/e4-dark.css");  
  
/* add new CSS rules here. You can also override setting defined in the imported  
by using the same selector-property combination */
```

Warning

Using CSS import statements requires that the plug-in which contains the imported CSS file is included in your product configuration file.

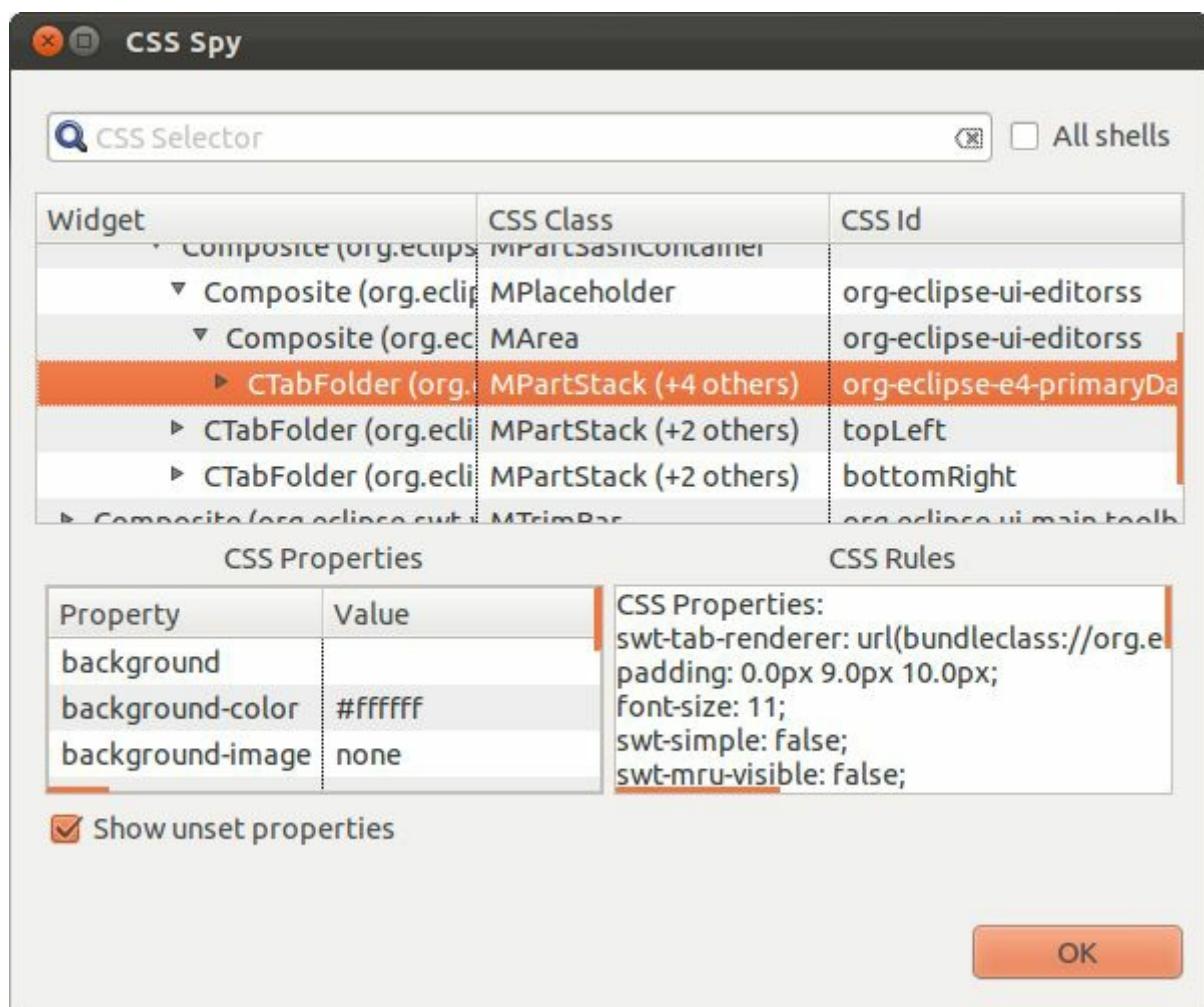
163.5. CSS Tools

CSS Spy

The *CSS spy* tool helps to see the possible styling options for selected elements. *CSS spy* is part of the *e4 tooling* project and can be installed from its update site. See the Eclipse 4 installation description for details.

You can open the *CSS spy* via the **Shift+Alt+F5** shortcut.

CSS spy allows you to see the available properties and the current style settings.



CSS Scratchpad

The *CSS Scratchpad* allows to change CSS rules interactively. is also part of the *e4 tooling* project.

You open it via the **Ctrl+Shift+Alt+F6** shortcut.

If you click the *Apply* button, the entered CSS is applied at runtime to your application.

Chapter 164. Exercise: Styling with CSS files

164.1. Target

In this exercise you style your application with a CSS file.

164.2. Create a CSS file

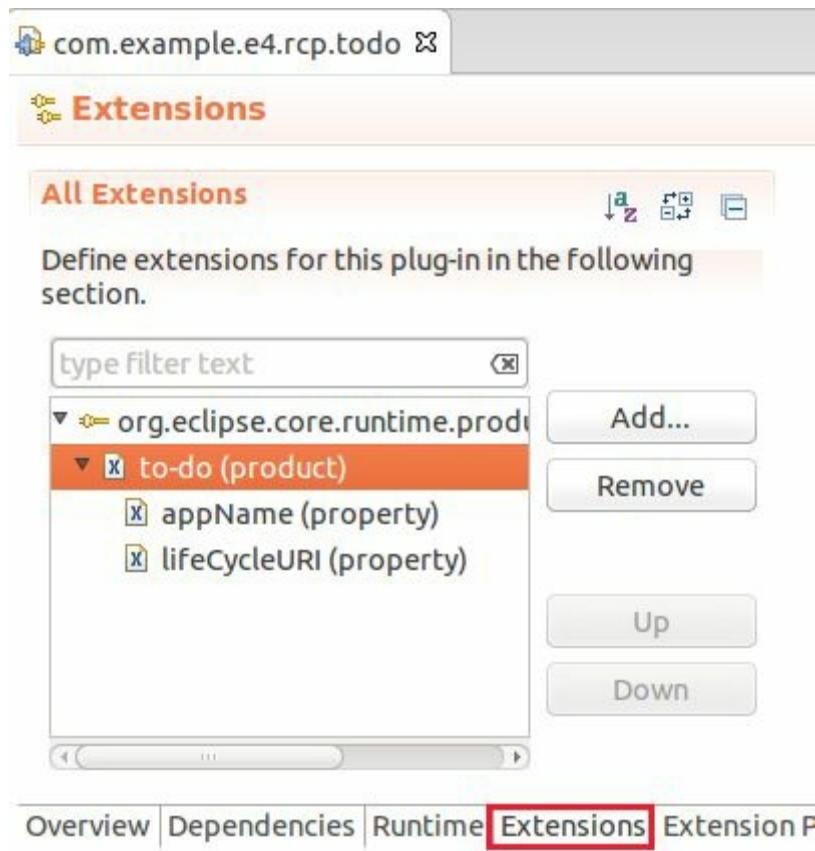
Create a folder called `css` in your `com.example.e4.rcp.todo` plug-in and create the following `default.css` file.

```
Label {
    font: Verdana 8px;
    color: black;
}
Composite Label {
    color: black;
}
Text {
    font: Verdana 8px;
}
/* Identifies only those SWT Text elements
appearing within a Composite */
Composite Text {
    background-color: white;
    color: black;
}
SashForm {
    background-color: #c1d5ef;
}
/* background uses a gradient */
.MTrimBar {
    background-color: #e3efff #c1d5ef;
    color: white;
    font: Verdana 8px;
}

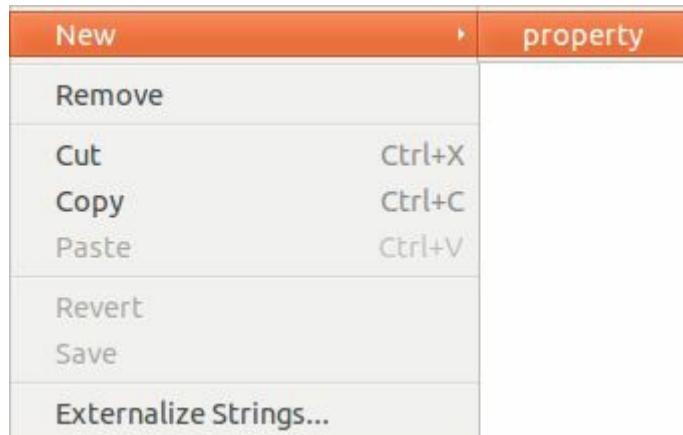
Shell {
    background-color: #e3efff #c1d5ef 60%;
}
```

164.3. Define the applicationCSS property

Open the `plugin.xml` file in your `com.example.e4.rcp.todo` plug-in. Select the *Extensions* tab.

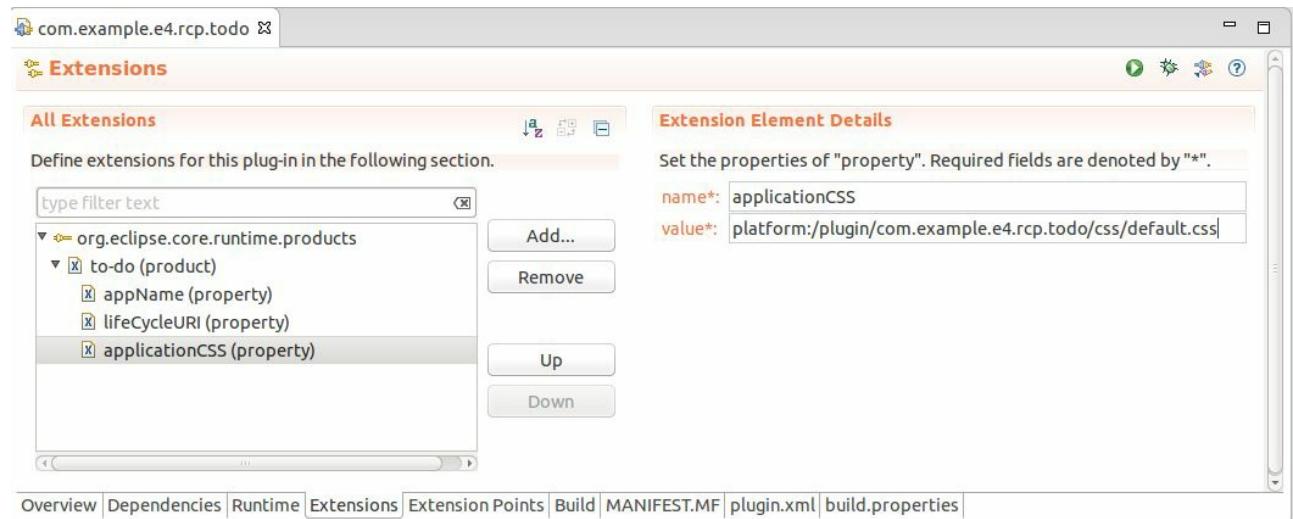


Right-click on your product entry and select New → property.



Use `applicationcss` as name for the property. The value of this property should use the following identifier.

platform:/plugin/com.example.e4.rcp.todo/css/default.css



164.4. Validating

Start your application. The application should be styled according to your CSS file.

Change the CSS style sheet and restart your application to see the effect.

164.5. Adjust the build.properties file

Add the created CSS file to your *build.properties* file in the `com.example.e4.rcp.todo` plug-in to make it available in an exported application.

Chapter 165. Exercise: Using the theme service

165.1. Target

The following exercise assumes that you have already implemented CSS support for your application based on the last exercise. In this exercise you use the `IThemeEngine` to introduce the ability to switch the application styling at runtime.

165.2. Add dependencies

Add the `org.eclipse.e4.ui.css.swt.theme` plug-in as dependency to your application plug-in.

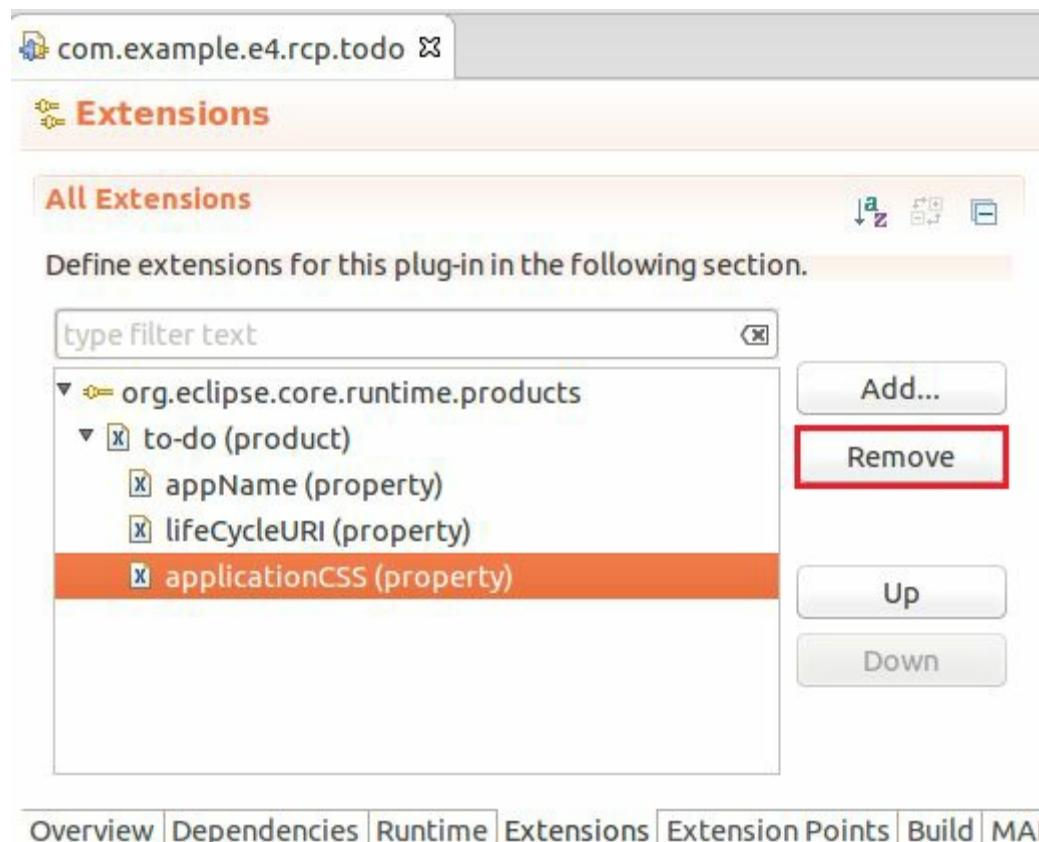
165.3. Create a CSS file

Create the following `css/rainbow.css` file.

```
Shell {  
    swt-background-mode: default;  
    background-color: red orange yellow green orange 30% 60% 75% 95%  
}  
  
.MPartStack {  
    font-size: 22px;  
    color: orange;  
}
```

165.4. Remove the applicationCSS property

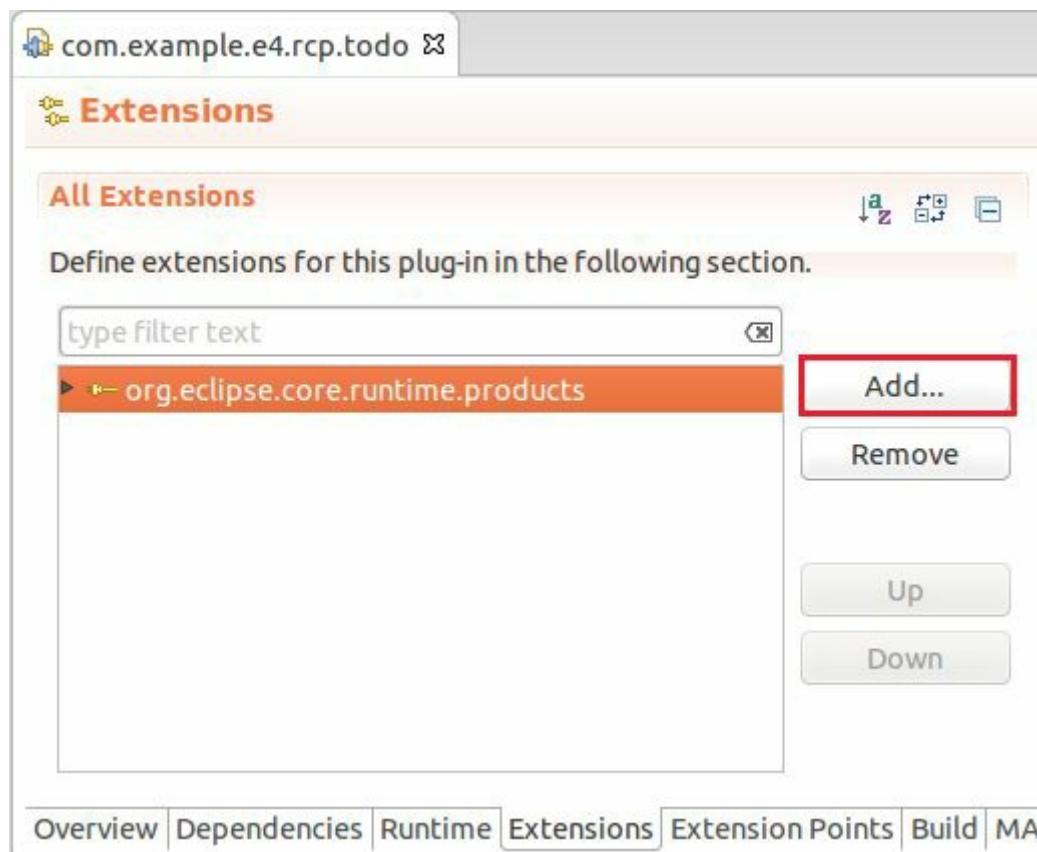
Remove the `applicationCSS` property from your product extension. For this select the property and press the *Remove* button as depicted in the following screenshot.



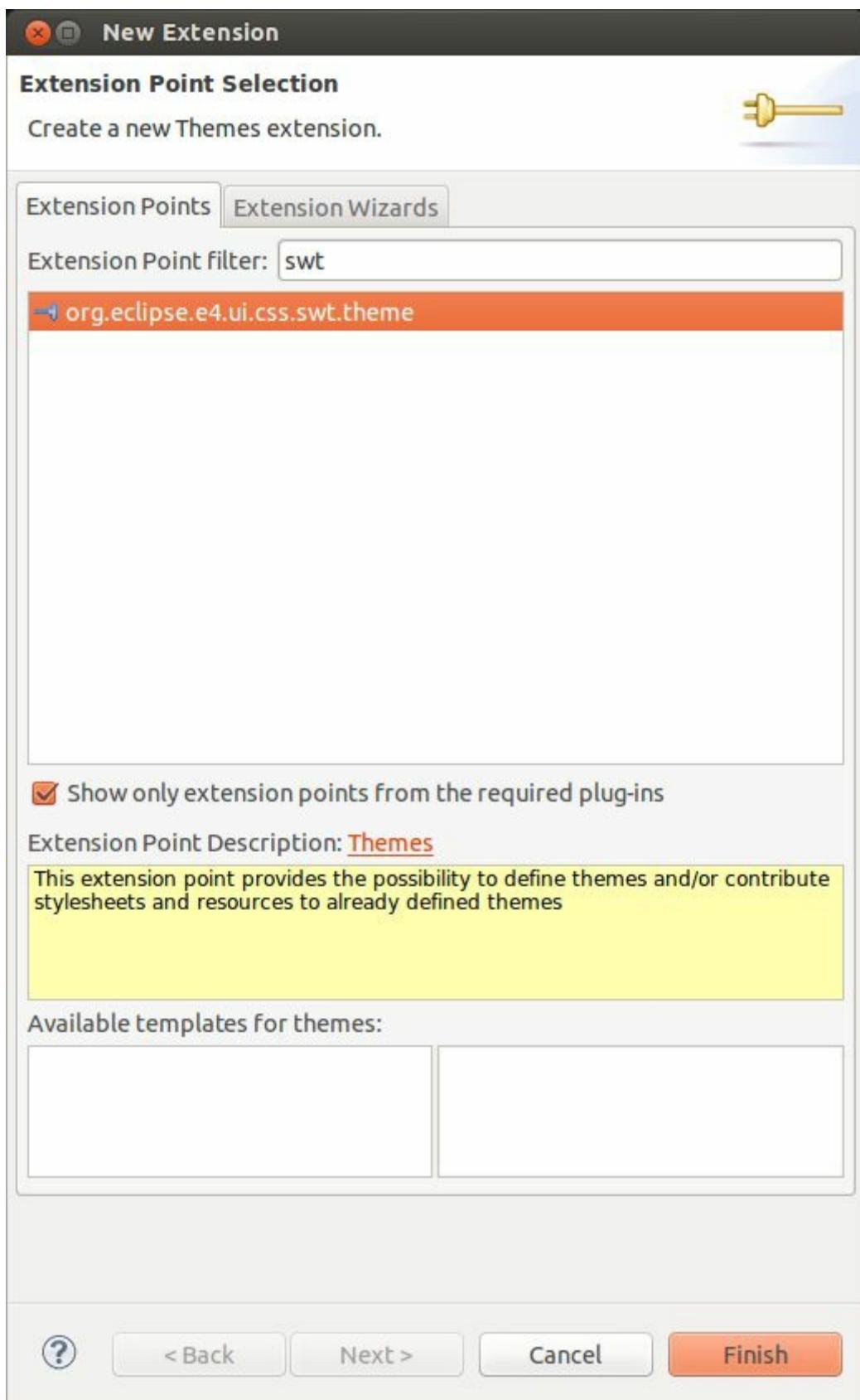
165.5. Create the theme extensions

Select the *Extensions* tab in the `plugin.xml` editor and create two extensions for the `org.eclipse.e4.ui.css.swt.theme` extension point.

You create an extension in the *Extensions* tab by pressing the *Add...* button.



Select the `org.eclipse.e4.ui.css.swt.theme` extension point and press the *Finish* button.



Right-click on the `org.eclipse.e4.ui.css.swt.theme` extension to create a new theme. The following screenshots show the extensions which you should create.

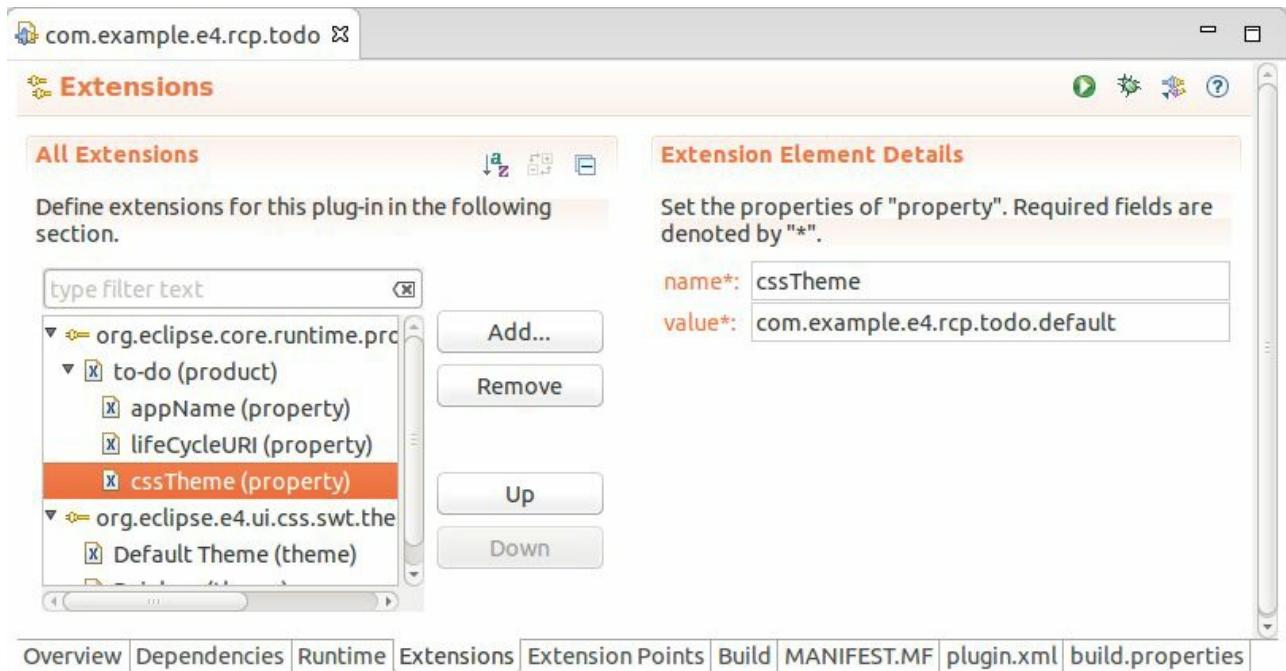
The screenshot shows the Eclipse Extensions dialog. On the left, under 'All Extensions', there is a tree view with nodes like 'org.eclipse.core.runtime.products' and 'org.eclipse.e4.ui.css.swt.theme'. Under 'org.eclipse.e4.ui.css.swt.theme', 'Default Theme (theme)' is selected. On the right, the 'Extension Element Details' panel shows configuration for this theme. The 'id*' field is set to 'com.example.e4.rcp.todo.default', 'label*' is 'Default Theme', and 'basestylesheeturi*' is 'css/default.css'. Buttons for 'Add...', 'Remove', 'Up', and 'Down' are visible between the tree and the details panel.

This screenshot is similar to the first one but shows the 'Rainbow' theme selected in the tree view under 'org.eclipse.e4.ui.css.swt.theme'. The 'Extension Element Details' panel shows the 'id*' field as 'com.example.e4.rcp.todo.rainbow', 'label*' as 'Rainbow', and 'basestylesheeturi*' as 'css/rainbow.css'.

The relevant part in the `plugin.xml` file should look like the following.

```
<extension
    point="org.eclipse.e4.ui.css.swt.theme">
    <theme
        basestylesheeturi="css/default.css"
        id="com.example.e4.rcp.todo.default"
        label="Default Theme">
    </theme>
    <theme
        basestylesheeturi="css/rainbow.css"
        id="com.example.e4.rcp.todo.rainbow"
        label="Rainbow">
    </theme>
</extension>
```

Add the `cssTheme` property to your product and point to the `com.example.e4.rcp.todo.default` theme.



Warning

Ensure that the `cssTheme` property is correctly set. Without this initial setting the `IThemeEngine` does not work.

165.6. Validating

Start your application and ensure that the application is styled according to the *cssTheme* property setting.

165.7. Implement a new menu entry

Create a new Java class, which allows you to switch between the themes.

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

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.css.swt.theme.IThemeEngine;

public class ThemeSwitchHandler {
    private static final String DEFAULT_THEME = "com.example.e4.rcp.todo.default";
    private static final String RAINBOW_THEME = "com.example.e4.rcp.todo.rainbow";

    @Execute
    public void switchTheme(IThemeEngine engine) {
        if (!engine.getActiveTheme().getId().equals(DEFAULT_THEME)) {
            // second argument defines that change is
            // persisted and restored on restart
            engine.setTheme(DEFAULT_THEME, true);
        } else {
            engine.setTheme(RAINBOW_THEME, true);
        }
    }
}
```

Define a new command and a handler in your application model for switching the style. Assign the above class to the handler.

Add a menu entry to your application model which uses your handler for switching the style.

165.8. Validate theme switching

Start the application and select your new menu entry. Afterwards the styling of your application should use the *rainbow* theme.



Note

Changes in styling may require a restart of your application if your new style does not override all properties of the old style.

165.9. Optional: Reusing the dark theme of Eclipse

You can also reuse other CSS themes. To use the dark theme of the Eclipse IDE, add the `org.eclipse.ui.themes` plug-in to your feature.

Create the following file called `dark.css` in your `css` folder.

```
@import url("platform:/plugin/org.eclipse.ui.themes/css/e4-dark.css");
```

Create a new theme pointing to the following new CSS file. Start your application and validate that you can switch to the dark theme.

165.10. Adjust the build.properties file

Ensure that all CSS files are selected in your *build.properties* file in the `com.example.e4.rcp.todo` plug-in to make them available in an exported application.

Chapter 166. Exercise: Styling the widgets created by the life cycle class

166.1. Target

In this exercise you will ensure that the widget created by the life cycle handler of your application are also styled.

166.2. Implement styling

Change your Manager class to the following to one of your CSS files for the widgets created in this class.

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

import javax.inject.Inject;

import org.eclipse.core.runtime.preferences.IEclipsePreferences;
import org.eclipse.e4.core.contexts.IEclipseContext;
import org.eclipse.e4.core.di.extensions.Preference;
import org.eclipse.e4.ui.internal.workbench.E4Workbench;
import org.eclipse.e4.ui.internal.workbench.swt.PartRenderingEngine;
import org.eclipse.e4.ui.workbench.IWorkbench;
import org.eclipse.e4.ui.workbench.lifecycle.PostContextCreate;
import org.eclipse.equinox.app.IApplicationContext;
import org.eclipse.jface.window.Window;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Rectangle;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Monitor;
import org.eclipse.swt.widgets.Shell;
import org.osgi.service.prefs.BackingStoreException;

import com.example.e4.rcp.todo.dialogs.PasswordDialog;
import com.example.e4.rcp.todo.preferences.PreferenceConstants;

public class Manager {

    // We add the nodePath in case you move the lifecycle handler to
    // another plug-in later
    @Inject
    @Preference(nodePath = PreferenceConstants.NODEPATH,
        value = PreferenceConstants.USER_PREF_KEY)
    private String user;

    @PostContextCreate
    public void postContextCreate(@Preference IEclipsePreferences prefs,
        IApplicationContext appContext, Display display, IEclipseContext context)

        final Shell shell = new Shell(SWT.SHELL_TRIM);
        PasswordDialog dialog = new PasswordDialog(shell);
        if (user != null) {
            dialog.setUser(user);
        }

        // close the static splash screen
        appContext.applicationRunning();

        // position the shell
        setLocation(display, shell);
```

```

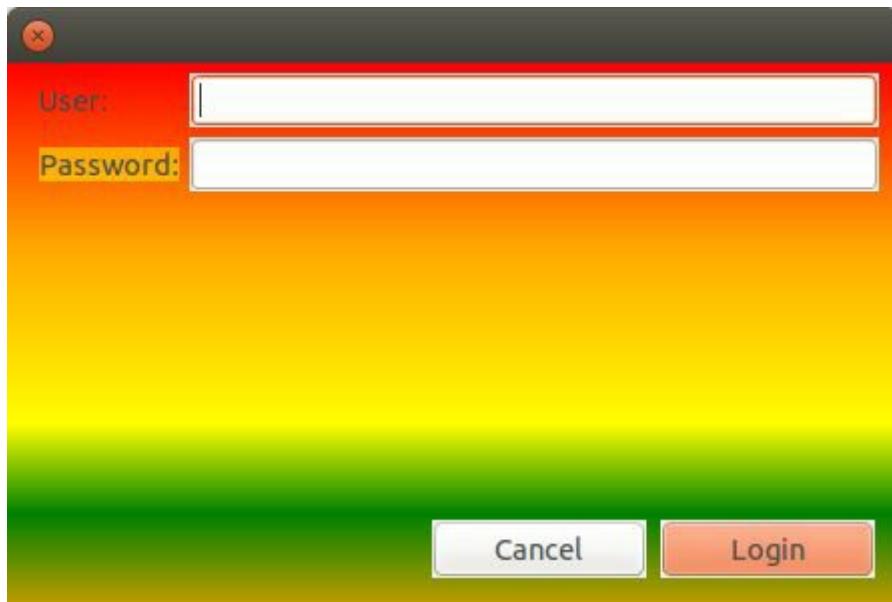
String cssURI = "platform:/plugin/com.example.e4.rcp.todo/css/rainbow.css";
context.set(E4Workbench.CSS_URI_ARG, cssURI);
PartRenderingEngine.initializeStyling(shell.getDisplay(), context);
// open the dialog
if (dialog.open() != Window.OK) {
    // close the application
    System.exit(-1);
} else {
    // get the user from the dialog
    String userValue = dialog.getUser();
    // store the user values in the preferences
    prefs.put(PreferenceConstants.USER_PREF_KEY, userValue);
    try {
        prefs.flush();
    } catch (BackingStoreException e) {
        e.printStackTrace();
    }
}

private void setLocation(Display display, Shell shell) {
    Monitor monitor = display.getPrimaryMonitor();
    Rectangle monitorRect = monitor.getBounds();
    Rectangle shellRect = shell.getBounds();
    int x = monitorRect.x + (monitorRect.width - shellRect.width) / 2;
    int y = monitorRect.y + (monitorRect.height - shellRect.height) / 2;
    shell.setLocation(x, y);
}
}

```

166.3. Validating

Start your application and ensure the splash screen is styled according to your settings.



Part XL. The renderer framework

Chapter 167. The usage of renderer

167.1. Renderer

An Eclipse application is modeled without a hard dependency to a specific user interface toolkit. Eclipse allows you to define a *renderer factory*. This renderer factory determines for every model element the responsible *renderer object*.

A renderer object is responsible for creating the user interface objects for its model element, monitoring their changes and reacting to these changes.

By default the Eclipse platform uses the SWT based `WorkbenchRendererFactory` as renderer factory. The `org.eclipse.e4.ui.workbench.renderers.swt` plug-in contains this default implementation.

167.2. Renderer factory and renderer objects

The renderer factory determines per model object the renderer object which is responsible for rendering the corresponding model object.

Renderer factories implement the `IRendererFactory` interface. In the `getRenderer()` method it returns a renderer object of type `AbstractPartRenderer`.

For example, if the model element is an instance of `MPart`, then an instance of the `ContributedPartRenderer` class is created by the default Eclipse renderer factory. The relevant code in the `WorkbenchRendererFactory` class is shown below.

```
// Other if statements above
else if (uiElement instanceof MPart) {
    if (contributedPartRenderer == null) {
        contributedPartRenderer = new ContributedPartRenderer();
        initRenderer(contributedPartRenderer);
    }
    return contributedPartRenderer;
}
```

167.3. Context creation of model objects

During the rendering the `IEclipseContext` of those model elements which extend `MContext` (`MApplication`, `MPart`, etc) is created and populated.

For example an object of type `ContributedPartRenderer` is the default renderer for a `MPart` model element. This object creates a `Composite` and injects it into the local context of each part.

167.4. Using a custom renderer

The Eclipse platform allows you to extend or exchange the default renderer classes. For example you can change the stack to use tabs at the bottom instead of the top.

You can also implement renderer for other user interface toolkits than SWT. For example you could use JavaFX or Swing for your application.

Tip

If you switch from SWT to another user interface technology your non user interface related logic, e.g., services, the model application, etc. can be reused. But you need to re-implement your user interface components. For example a part written in SWT must be re-written in JavaFX to work together with a JavaFX Renderer.

The custom renderer factory is registered via the `org.eclipse.core.runtime.products` extension point. You use an additional property called `rendererFactoryUri` to point to the new factory class.

167.5. Using a custom renderer for one model element

If you want to replace the renderer for only one model element, there is a key CUSTOM_RENDERER_URI which you can add to the persistentData map of a model element. The 'value' for this key would be the URI of the renderer you want to use.

Chapter 168. Existing alternative renderer implementations

168.1. Alternatives to SWT

You can use any user interface technology for your RCP application, as long as you provide a renderer implementation for it. There are already several default implementations besides SWT available. This section lists the most popular ones.

A popular misconception about a different renderer is that you can port your SWT applications unchanged to it, for example to Vaadin or JavaFX. That is not the case. Alternative renderer implementations do not port automatically your user interface implementation which relate to code. For example in the exercises of this book your parts or your tool controls are implemented with SWT classes.

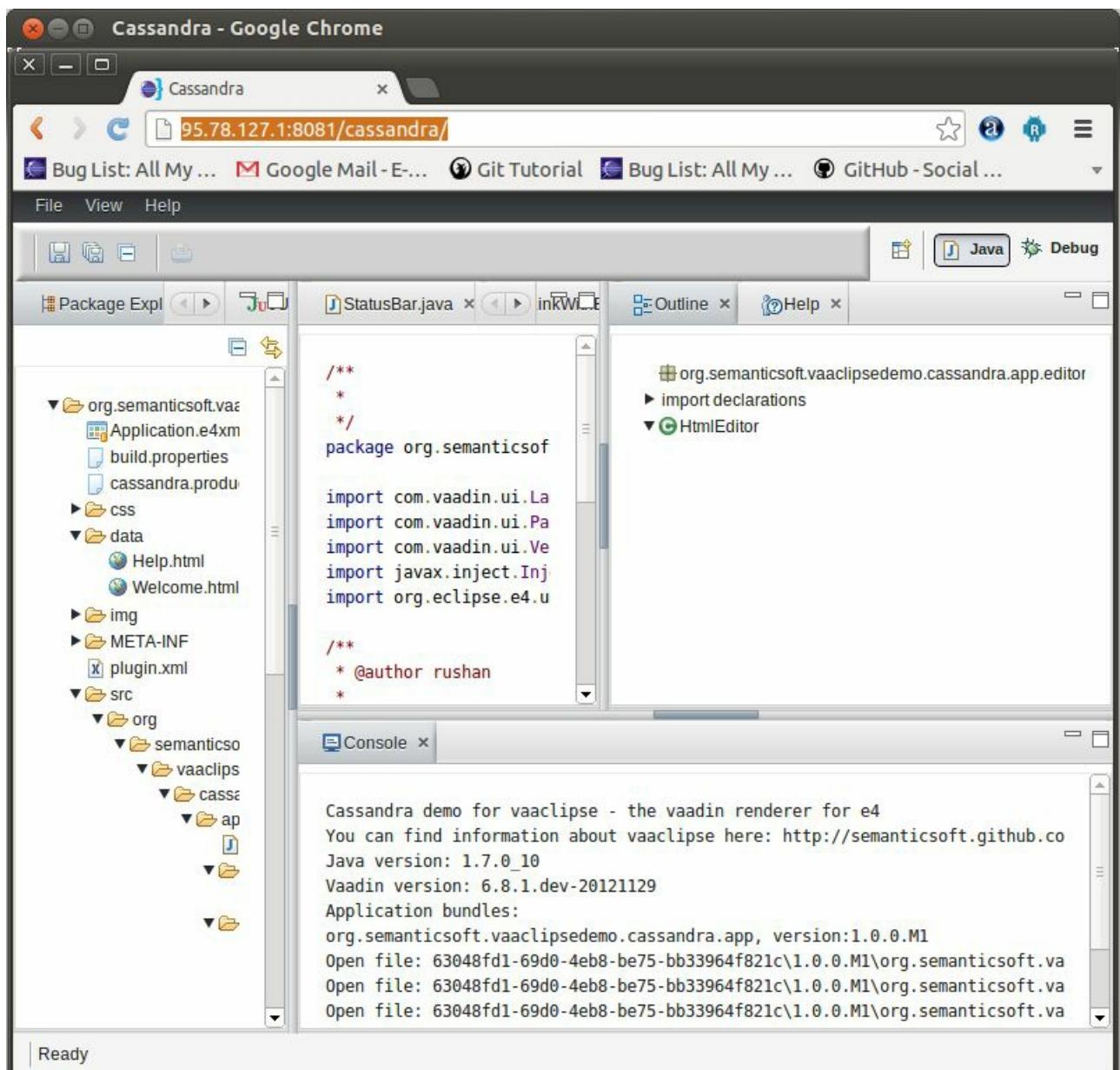
You have to re-implement these user interface components. The value of alternative renderer implementations lies in the fact that you have an application framework with a modeled application and dependency injection of which your user interface framework can take advantage. This allows you to use the Eclipse 4 application framework with the user interface toolkit of your choice.

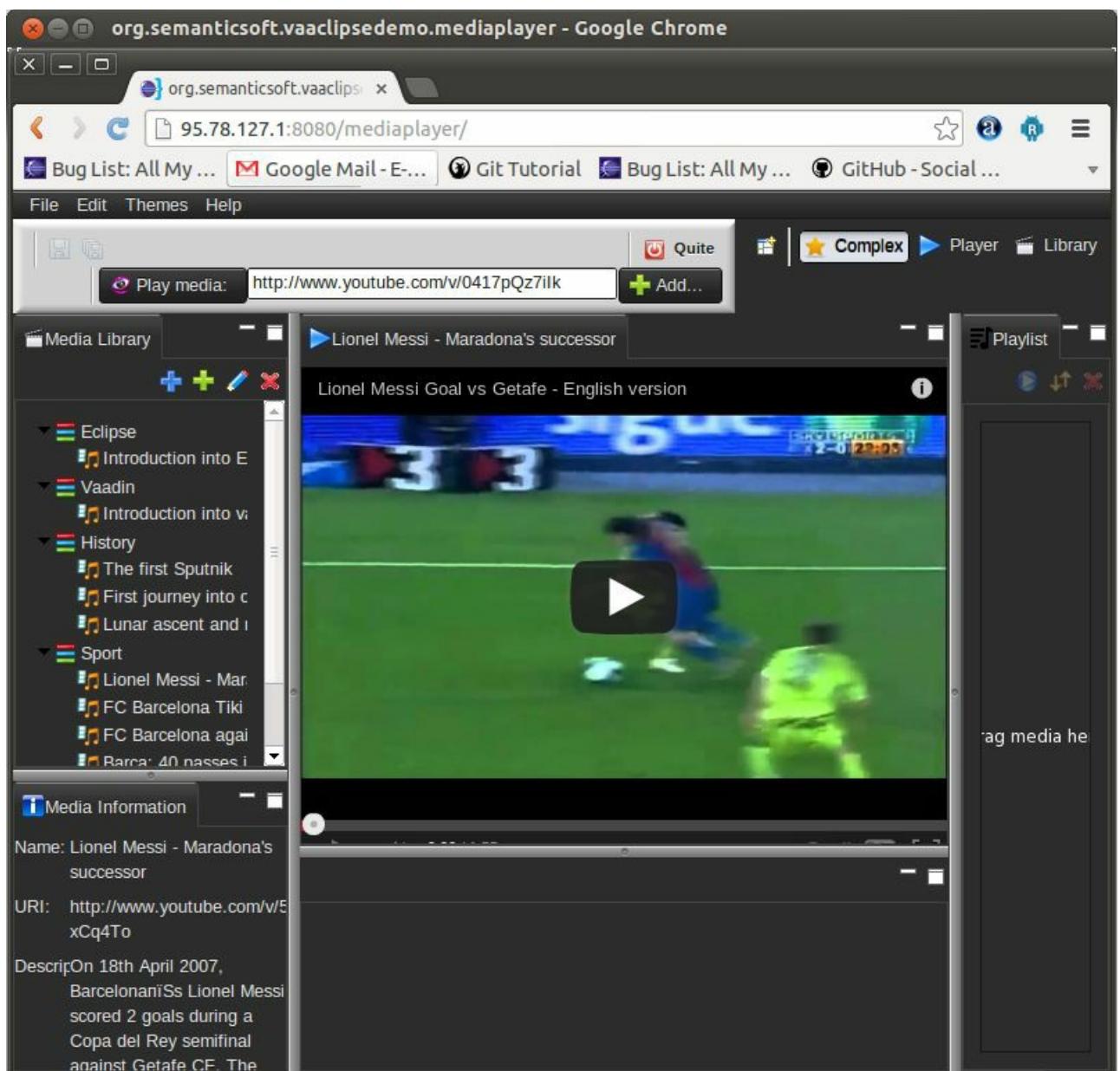
168.2. Vaadin renderer - Vaaclipse

Vaaclipse is a framework for building web applications using the Eclipse 4 platform and Vaadin. It allows to use the power of the Eclipse framework in web development.

You find more information about this project on the following homepage:
[Vaaclipse](#)

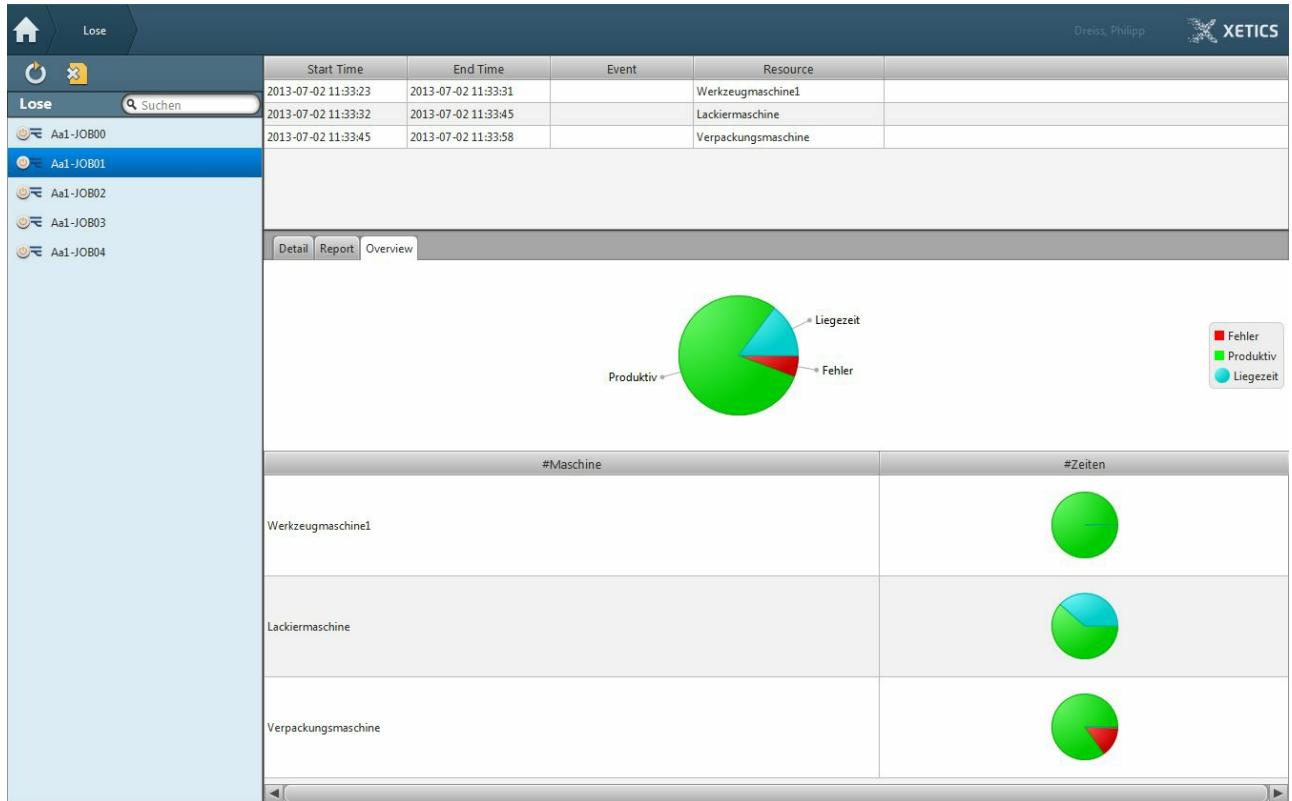
The following screenshots show demo applications of Vaaclipse.





168.3. JavaFX renderer - e(fx)clipse

The e(fx)clipse provides a JavaFX renderer for Eclipse 4 RCP application. For more information see the [e\(fx\)clipse homepage](#). The following screenshot shows an example Eclipse RCP application using JavaFX as a renderer.



168.4. Eclipse RAP

Eclipse RAP is an approach to provide a web implementation for the SWT API and makes it relatively easy to run an Eclipse 3.x RCP application in the web without code modifications.

Eclipse RAP is special case as the renderer is still the default SWT renderer but there is a translation layer between SWT and the actual widget that is produced. This allows you to reuse your SWT user interface code to run as a web application.

Currently the Eclipse RAP team does not officially support the Eclipse 4 API but several open source ports have been made. See for example [Eclipse 4 on RAP wiki](#).

168.5. Additional UI toolkits

Writing source code is the current default approach by the Eclipse platform to create your user interfaces.

Currently several open source frameworks are developed to simplify the creation of user interface components or to remove a hard dependency to a specific rendering framework. The following description is a limited selection of possible approaches.

The jo-widget project provides components which can be rendered to SWT, Swing and JavaFX. See the [jo-widget homepage](#) for more information.

The Eclipse XWT project supports the creation of user interface components based on XML files. See the [XWT wiki](#) for documentation.

The Eclipse Sapphire project allows you to create user interfaces based on a property editor. See the [Eclipse Sapphire homepage](#) for details.

The Wazzabi project provides functionality to model the user interface via EMF and to generate the SWT user interface at runtime based on this model. See the [Wazzabi homepage](#) for more information.

The EMF Parsley project plans to provide model based user interface components. See the [EMF Parsley project page](#) for more information.

Chapter 169. Exercise: Defining a renderer

169.1. Target

The following exercise serves as a simple example for creating a custom renderer. It implements a custom stack renderer which uses "* Demo *" instead of "*" to mark an editor dirty.

169.2. Creating a plug-in

Create a new simple plug-in called `com.example.e4.renderer.swt`. This plug-in is called the `renderer` plug-in in the following description.

169.3. Enter the dependencies

Add the following plug-ins as dependencies to your `renderer` plug-in.

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

169.4. Create the renderer implementation

In your renderer plug-in create the `com.example.e4.renderer.swt` package and the following classes. The new stack renderer extends the stack renderer from the Eclipse platform.

```
package com.example.e4.renderer.swt;

import org.eclipse.e4.ui.model.application.ui.MDirtyable;
import org.eclipse.e4.ui.model.application.ui.MUILabel;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.e4.ui.workbench.UIEvents;
import org.eclipse.e4.ui.workbench.renderers.swt.StackRenderer;
import org.eclipse.swt.custom.CTabItem;

public class MyStackRenderer extends StackRenderer {

    private String getLabel(MUILabel itemPart, String newName) {
        if (newName == null) {
            newName = ""; //NON-NLS-1$
        }
        if (itemPart instanceof MDirtyable && ((MDirtyable) itemPart).isDirty()) {
            newName = '*' + newName; //NON-NLS-1$
        }
        return newName;
    }

    protected void updateTab(CTabItem cti, MPart part, String attName,
                           Object newValue) {
        if (UIEvents.UILabel.LABEL.equals(attName)) {
            String newName = (String) newValue;
            cti.setText(getLabel(part, newName));
        } else if (UIEvents.UILabel.ICONURI.equals(attName)) {
            cti.setImage(getImage(part));
        } else if (UIEvents.UILabel.TOOLTIP.equals(attName)) {
            String newTTip = (String) newValue;
            cti.setToolTipText(newTTip);
        } else if (UIEvents.Dirtyable.DIRTY.equals(attName)) {
            Boolean dirtyState = (Boolean) newValue;
            String text = cti.getText();
            boolean hasExclamationMark = text.length() > 0 && text.charAt(0) == '!';
            if (dirtyState.booleanValue()) {
                if (!hasExclamationMark) {
                    cti.setText("* Demo *" + text);
                }
            } else if (hasExclamationMark) {
                cti.setText(text.substring(1));
            }
        }
    }
}
```

```
package com.example.e4.renderer.swt;

import org.eclipse.e4.ui.internal.workbench.swt.AbstractPartRenderer;
import org.eclipse.e4.ui.model.application.ui.MUIEelement;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.e4.ui.model.application.ui.basic.MPartStack;
import org.eclipse.e4.ui.workbench.renderers.swt.WorkbenchRendererFactory;

public class MyRendererFactory extends WorkbenchRendererFactory {
    private MyStackRenderer stackRenderer;

    @Override
    public AbstractPartRenderer getRenderer(MUIEelement uiElement, Object parent) {
        if (uiElement instanceof MPartStack) {
            if (stackRenderer == null) {
                stackRenderer = new MyStackRenderer();
                super.initRenderer(stackRenderer);
            }
            return stackRenderer;
        }
        return super.getRenderer(uiElement, parent);
    }
}
```

169.5. Register the renderer

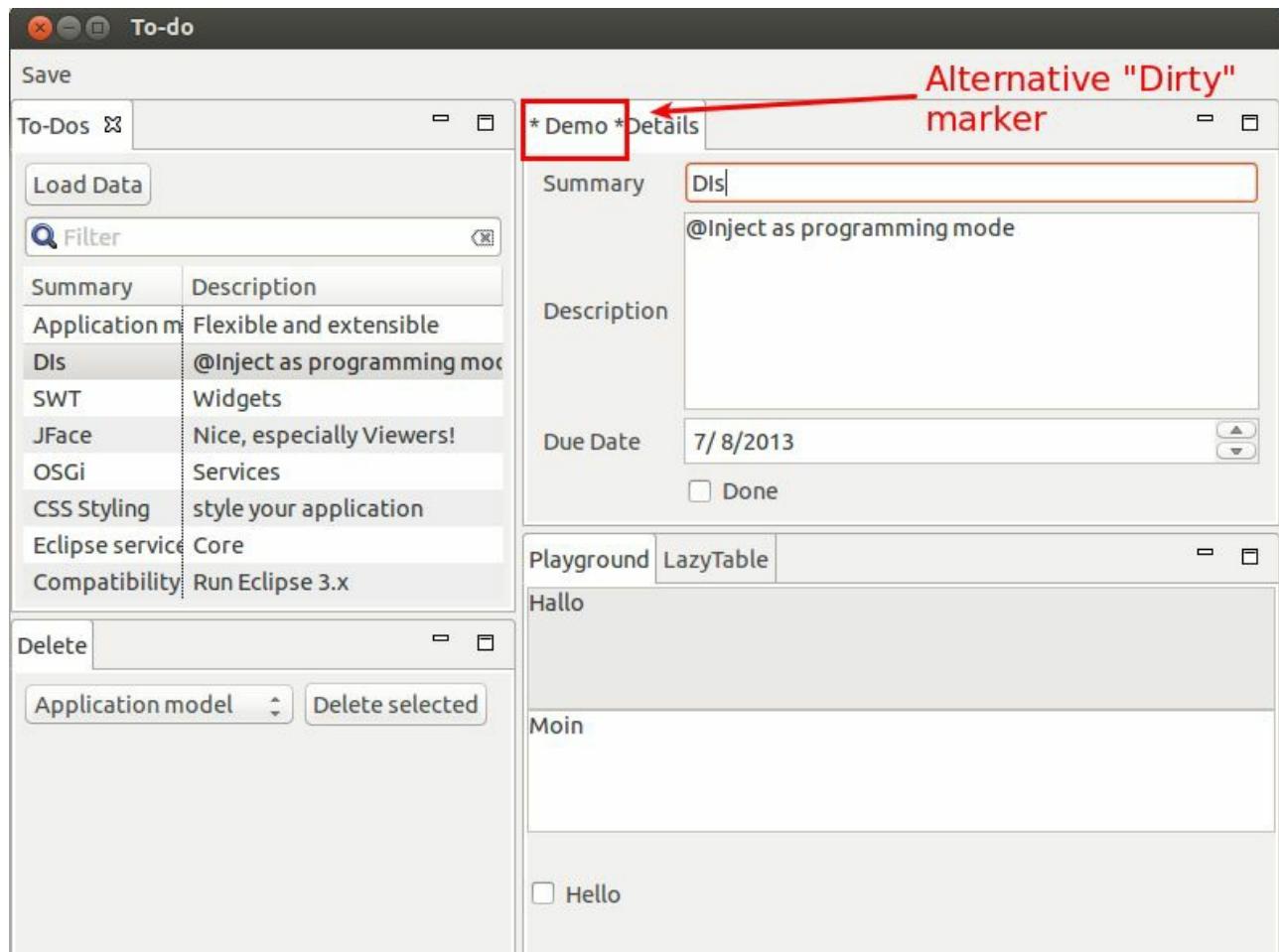
Register your renderer in the `plugin.xml` file of your `com.example.e4.rcp.todo` plug-in. For this create a new property on the extension for your product. Use `rendererFactoryUri` as name and `bundleclass://com.example.e4.renderer.swt/com.example.e4.renderere` as value. This setting is similar to the setting of the `applicationCSS` or the `lifeCycleURI` property hence you should know how to do this.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>

    <extension
        id="product"
        point="org.eclipse.core.runtime.products">
        <product
            application="org.eclipse.e4.ui.workbench.swt.E4Application"
            name="to-do">
            <property
                name="applicationXMI"
                value="com.example.e4.rcp.todo/Application.e4xmi">
            </property>
            <property
                name="appName"
                value="to-do">
            </property>
            <property
                name="rendererFactoryUri"
                value="bundleclass://com.example.e4.renderer.swt/[CONTINUE...]
com.example.e4.renderer.swt.MyRendererFactory">
            </property>
        </product>
    </extension>
</plugin>
```

169.6. Validating

Start your application and change a Todo in your `TodoDetailPart`. The dirty indicator should now be different as indicated in the following screenshot.



169.7. Exercise: A custom part renderer

In this exercise you create a custom part renderer which renders all parts with an SWT browser widget.

Warning

This example might not work on Linux, as the SWT browser widget sometimes has issues on Linux.

```
package com.example.e4.renderer.swt;

import org.eclipse.e4.ui.model.application.ui.MUIElement;
import org.eclipse.e4.ui.workbench.renderers.swt.SWTPartRenderer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.browser.Browser;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;

public class MyPartRenderer extends SWTPartRenderer {

    @Override
    public Object createWidget(MUIElement element, Object parent) {
        final Composite mapComposite = new Composite((Composite) parent,
            SWT.NONE);
        System.out.println(parent.getClass());
        mapComposite.setLayout(new GridLayout(1, false));
        final Browser browser = new Browser(mapComposite, SWT.NONE);
        browser.setUrl("http://www.vogella.com");
        GridData data = new GridData(SWT.FILL, SWT.FILL, true, true);
        browser.setLayoutData(data);
        return mapComposite;
    }
}
```

Extend the `MyRendererFactory` to the following to use your own part renderer.

```
package com.example.e4.renderer.swt;

import org.eclipse.e4.ui.internal.workbench.swt.AbstractPartRenderer;
import org.eclipse.e4.ui.model.application.ui.MUIElement;
import org.eclipse.e4.ui.model.application.ui.basic.MPart;
import org.eclipse.e4.ui.model.application.ui.basic.MPartStack;
import org.eclipse.e4.ui.workbench.renderers.swt.WorkbenchRendererFactory;

public class MyRendererFactory extends WorkbenchRendererFactory {

    private MyStackRenderer stackRenderer;
```

```

private MyPartRenderer partRenderer;

@Override
public AbstractPartRenderer getRenderer(MUIElement uiElement, Object parent) {

    if (uiElement instanceof MPart) {
        if (partRenderer == null) {
            partRenderer = new MyPartRenderer();
            super.initRenderer(partRenderer);
        }
        return partRenderer;
    } else if (uiElement instanceof MPartStack) {
        if (stackRenderer == null) {
            stackRenderer = new MyStackRenderer();
            super.initRenderer(stackRenderer);
        }
        return stackRenderer;
    }
    return super.getRenderer(uiElement, parent);
}

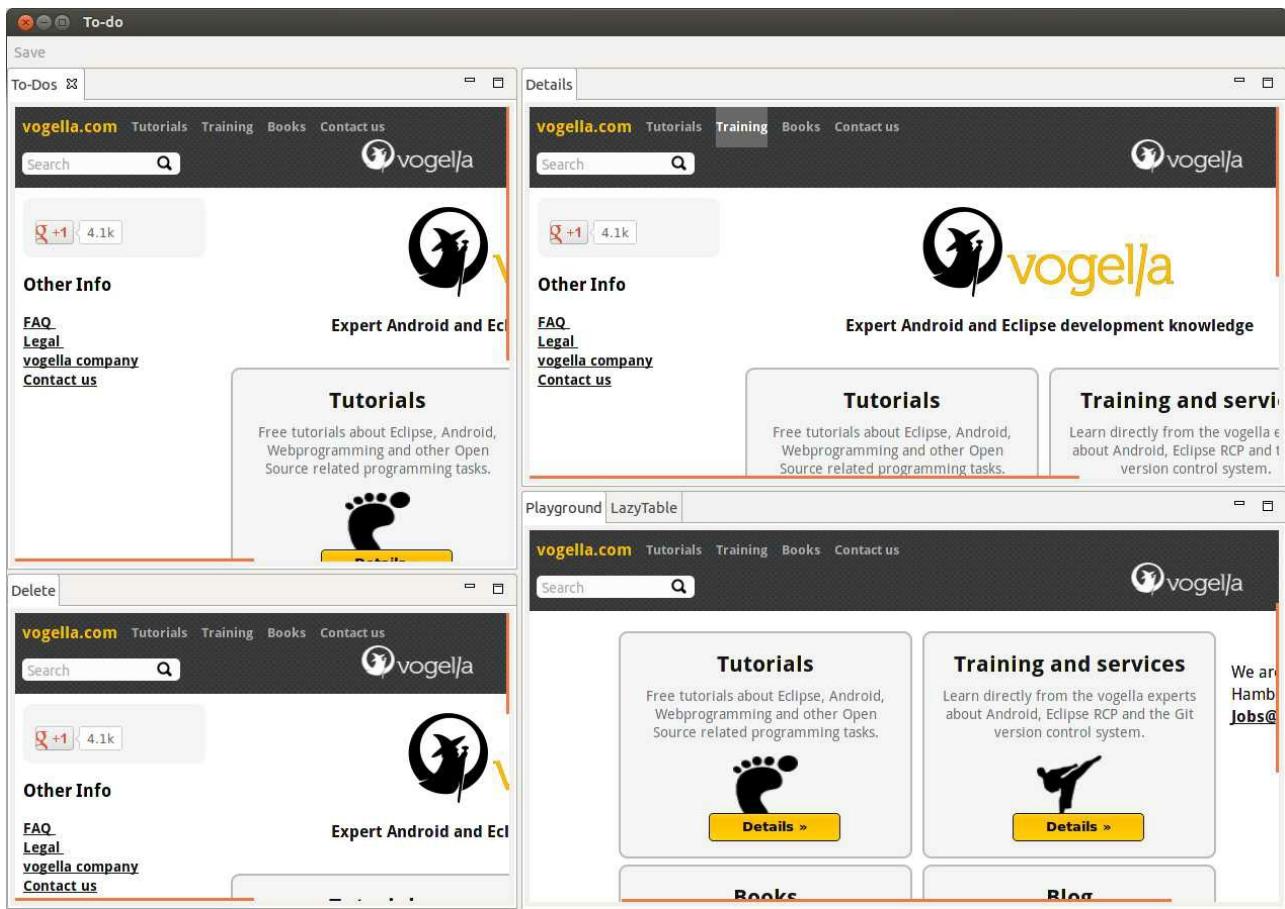
}

```

Run your application and test that every part is now rendered as browser.

Note

Currently all parts show the same URL, but you could extend your renderer to access for example tags at the parts to determine the URLs which should be used.



Warning

After finishing the test, remove the `rendererFactoryUri` property from the `plugin.xml` file of your application plug-in, so that your application works again as before.

Part XLI. Application model persistence and model extensions

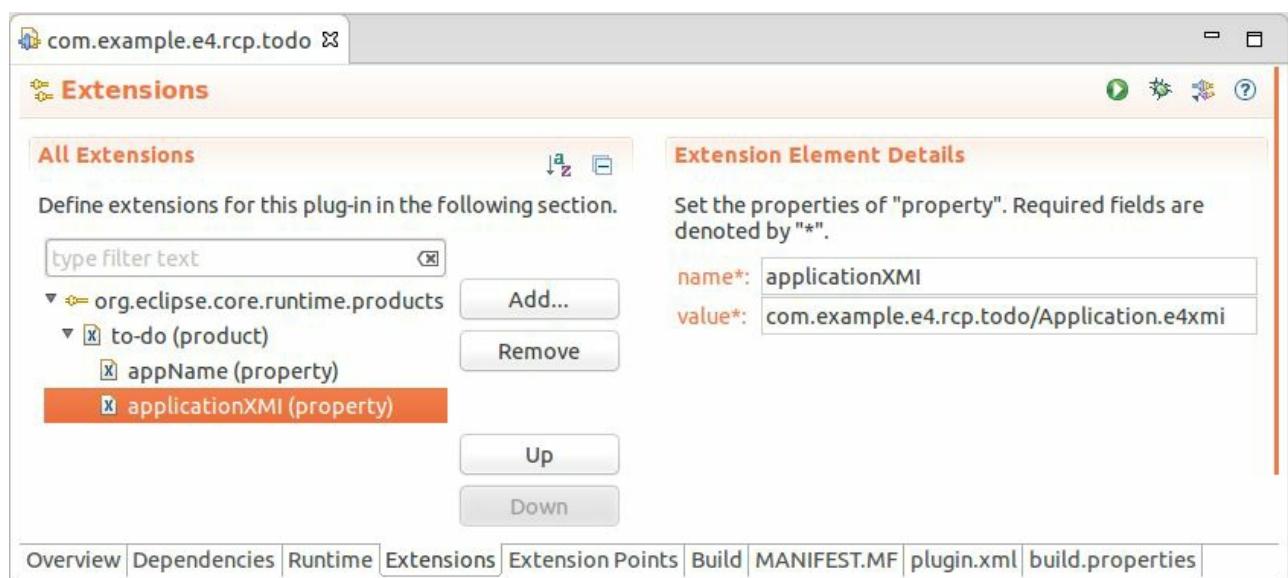
Chapter 170. Custom persistence for the application model

170.1. Specifying the location of the application model file

The file name and location of the application model can be specified in the `plugin.xml` file. This is done via the `applicationXMI` property. The path to the file follows the "bundleSymbolicName"/"filename" pattern, e.g. `test/ApplicationDifferent.e4xmi`.

To do this open the `plugin.xml` file, select the *Extensions* tab and open the `org.eclipse.core.runtime.products` contribution.

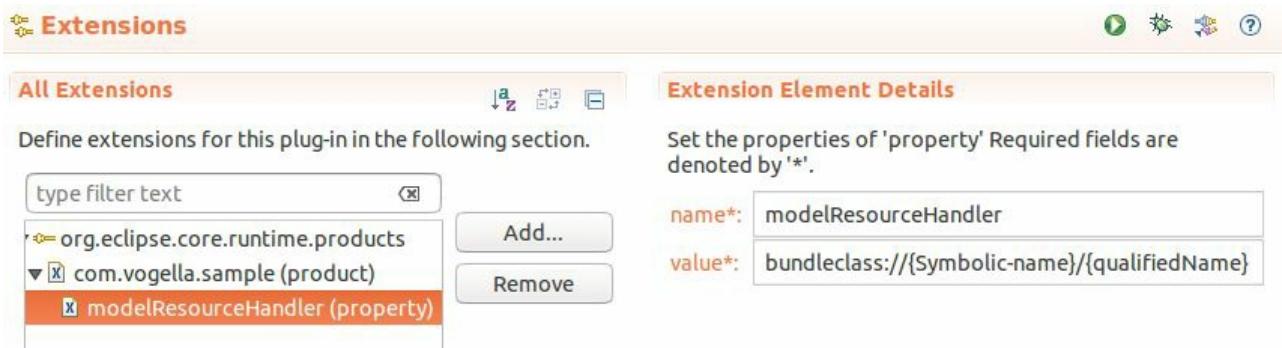
Right-click on the *product* entry and select `New → Property`. Use `applicationXMI` as parameter name and `com.example.e4.rcp.todo/ApplicationNew.e4xmi` as value. This value consists of the `BundleSymbolicName` of the corresponding plug-in and the file name of your application model file.



170.2. Custom application model persistence handler

Eclipse uses an instance of the `IModelResourceHandler` interface to load and save the application model at startup and shutdown. The default implementation of this interface is provided by the `ResourceHandler` class from the `org.eclipse.e4.ui.internal.workbench` package.

You can specify the `modelResourceHandler` as parameter on your product extension to point to another class via the `bundleclass://` notation.



This allows you to construct the initial application model from any source you desire. For example, you could use property files or construct the application model based on information from a database.

Implementing this interface requires knowledge about the Eclipse Modeling Framework (EMF). You can use the `ResourceHandler` as a template to see how the `IModelResourceHandler` interface could be implemented.

Chapter 171. Saving and restoring parts of the application model

171.1. Store certain model elements

By default the Eclipse platform stores and restores the complete runtime application model, unless the startup parameter `clearPersistedState` is used. If this parameter is used, the whole user changes are deleted.

It is possible to store selected model elements via the `E4XMIResourceFactory` class. This allows a more flexible handling of the model persistence.

The following snippet shows a handler implementation, where the corresponding model elements of the active perspective are persisted in an xmi file.

```
package com.vogella.e4.model.persistence.handlers;

import java.io.FileOutputStream;
import java.io.IOException;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.internal.workbench.E4XMIResourceFactory;
import org.eclipse.e4.ui.model.application.ui.MUIElement;
import org.eclipse.e4.ui.model.application.ui.advanced.MPerspective;
import org.eclipse.e4.ui.model.application.ui.basic.MWindow;
import org.eclipse.e4.ui.workbench.modeling.EModelService;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.resource.Resource;

public class SaveHandler {

    @Execute
    public void execute(EModelService modelService, MWindow window) {

        // store model of the active perspective
        MPerspective activePerspective = modelService.getActivePerspective(window);

        // create a resource, which is able to store e4 model elements
        E4XMIResourceFactory e4xmiResourceFactory = new E4XMIResourceFactory();
        Resource resource = e4xmiResourceFactory.createResource(null);

        // You must clone the perspective as snippet, otherwise the running
        // application would break, because the saving process of the resource
        // removes the element from the running application model
        MUIElement clonedPerspective = modelService.cloneElement(activePerspective,

        // add the cloned model element to the resource so that it may be stored
        resource.getContents().add((EObject) clonedPerspective);

        FileOutputStream outputStream = null;
```

```
try {

    // Use a stream to save the model element
    outputStream = new FileOutputStream("/home/example/snippet.xmi");

    resource.save(outputStream, null);
} catch (IOException ex) {
    ex.printStackTrace();
} finally {
    if (outputStream != null) {
        try {
            outputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}
```

171.2. Load stored model elements

The previously stored xmi file can be restored by using the `load` method of the `Resource` class.

The following snippet is an example, where a stored perspective is applied to an existing application model. In case an existing perspective with the same `elementId` as the one of the loaded perspective is found, it is removed. Afterwards the code adds and switches to the loaded perspective.

```
package com.vogella.e4.model.persistence.handlers;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.List;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.e4.ui.internal.workbench.E4XMIResourceFactory;
import org.eclipse.e4.ui.model.application.ui.MElementContainer;
import org.eclipse.e4.ui.model.application.ui.MUIElement;
import org.eclipse.e4.ui.model.application.ui.advanced.MPerspective;
import org.eclipse.e4.ui.model.application.ui.basic.MWindow;
import org.eclipse.e4.ui.workbench.modeling.EModelService;
import org.eclipse.e4.ui.workbench.modeling.EPartService;
import org.eclipse.emf.ecore.resource.Resource;

public class LoadHandler {

    @Execute
    public void execute(EPartService partService,
        EModelService modelService,
        MWindow window) throws IOException {

        // create a resource, which is able to store e4 model elements
        E4XMIResourceFactory e4xmiResourceFactory = new E4XMIResourceFactory();
        Resource resource = e4xmiResourceFactory.createResource(null);

        FileInputStream inputStream = null;
        try {

            inputStream = new FileInputStream("/home/example/snippet.xmi");

            // load the stored model element
            resource.load(inputStream, null);

            if (!resource.getContents().isEmpty()) {

                // after the model element is loaded it can be obtained from the
                // contents of the resource
                MPerspective loadedPerspective = (MPerspective) resource.getContents().g
```

```
// get the parent perspective stack, so that the loaded
// perspective can be added to it.
MPerspective activePerspective = modelService.getActivePerspective(windo
MElementContainer<MUIElement> perspectiveParent = activePerspective.getP

// remove the current perspective, which should be replaced by
// the loaded one
List<MPerspective> alreadyPresentPerspective = modelService.findElements
    loadedPerspective.getElementId(), MPerspective.class, null);
for (MPerspective perspective : alreadyPresentPerspective) {
    modelService.removePerspectiveModel(perspective, window);
}

// add the loaded perspective and switch to it
perspectiveParent.getChildren().add(loadedPerspective);

partService.switchPerspective(loadedPerspective);
}
} finally {
    if (inputStream != null) {
        inputStream.close();
    }
}
}
```

Chapter 172. Extend the Eclipse application model

It is also possible to extend the application model. This requires knowledge about the Eclipse EMF framework. Describing EMF is currently out of scope for this book. See [Extending the Eclipse 4 application model](#) for an introduction into the concepts.

Part XLII. Good development practices

Chapter 173. Eclipse development good practices

173.1. Project, package and class names

It is good practice to use your reverse web domain name as top-level package identifier. For example if you own the domain *mycompany.com* you package name space would start with `com.mycompany`. This avoids name collision with packages contributed by other plug-ins.

The following table suggests good practices for naming conventions for projects, packages and classes.

Table 173.1. Naming conventions

Object	Description
Project Names	The plug-in project name is the same as the top-level package name.
Packages	Plug-ins which contain a lot 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 class name. For example, a class used as a part implementation, should be called <code>[PurposeDescription]Part</code> .

173.2. Naming conventions for model identifiers (IDs)

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 element and ensure that these IDs are unique whenever it makes sense. Reusing IDs is sometimes required. For example you typically use the same ID for the main menu in every window of your application. This allows menu contributions to contribute to every window its menu entries.

Unintentionally using the same ID for a model element may result in unexpected behavior. For example if you search for a element via the ID.

A good convention is to start IDs with the *top level package* name of your project followed by a group descriptor and a name which gives an idea about the purpose of the element. For example, *com.example.e4.rcp.todo.part.todooverview*, where *com.example.e4.rcp.todo* is the top level package, *part* is the group descriptor for all visible parts (views and editors) in your application and *todooverview* gives an idea about the purpose of this part.

Also note that the entire ID is written only in lower case characters.

Some Eclipse projects also use camelCase for the last part of the ID, but that is more for historical reasons.

173.3. Create isolated components

It is important to separate your application components into isolated modules. This reduces the complexity of development and allows you to reuse these components in different applications.

User interface related and core functionalities should be separated into different plug-ins. For example, if you develop a new SWT widget, you should place this widget in a separate plug-in.

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.

173.4. Usage of your custom extension points

The programming model of Eclipse 4 has reduced the need for using extensions and extension points but they 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.

173.5. Avoid releasing unnecessary API

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.

173.6. Packages vs. plug-in dependencies

OSGi allows you to define dependencies via plug-ins (Require-Bundle) or via packages (Import-Package) in the manifest file.

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 intend 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.

Chapter 174. Application communication and context usage

174.1. Application communication

A simple way of propagating user interface selections is the selection service (`ISelectionService`) of the Eclipse platform. Use this service to propagate the current selection of a window.

For general scoped communication it is recommended to use the Eclipse context (`IEclipseContext`).

The event service (`IEventBroker`) is a good choice, if there is no scope involved in the communication and if the events should not be persisted in the Eclipse context. The strengths of the event service is that arbitrary listeners can listen to events and that the publish and subscribe mechanism is relatively simple.

Frequently the event service is also used together with a modification of the Eclipse context. The following section contains an example for this.

174.2. Example: Using events together with the IEclipseContext

The following example demonstrates the registration for events and how the provided information is used to modify the context.

Assume that on every change of the active part you want to place a variable called `myactivePartId` into the `IEclipseContext`. This can be done via the following code.

```
// set a context variable
@Inject
@Optional
public void partActivation(@UIEventTopic(UIEvents.UILifeCycle.ACTIVATE)
    Event event,
    MApplication application) {

    // determine the context of the application
    MPart activePart = (MPart) event.
        getProperty(UIEvents.EventTags.ELEMENT);
    IEclipseContext context = application.getContext();
    if (activePart != null) {
        context.set("myactivePartId", activePart.getElementId());
    }
}
```

This context variable can be used to define a visible-when clause which can be used to restrict the visibility of menus and toolbars.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>

    <extension
        id="product"
        point="org.eclipse.core.runtime.products">
        <product
            application="org.eclipse.e4.ui.workbench.swt.E4Application"
            name="to-do">
            <property
                name="applicationXMI"
                value="com.example.e4.rcp.todo/Application.e4xmi">
            </property>
            <property
                name="appName"
                value="to-do">
            </property>
        </product>
    </extension>
```

```

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

</plugin>

```



174.3. Which dependency injection approach to use for your implementation

The application can add or modify the behavior of the dependency injection mechanisms. You have several options for this. The most common approaches are the following:

- OSGi services
- Context modifications of the Eclipse context
- Context functions
- Extended object suppliers for custom annotations
- Model add-ons

All are valid options for your implementation.

The programming model of Eclipse 4 makes it easy to create and use OSGi services. OSGi services do not have access to the Eclipse application context. OSGi services are a good approach for infrastructure services.

Context functions allow you to create objects based on the Eclipse context. They also allow you to persist the created object in the context hierarchy. If access to the context is required, you use context functions instead of pure OSGi services.

Storing the key/value pairs in the Eclipse context (as context elements or context variables) allows you to persist them in the hierarchy and to propagate them to elements down in the relevant hierarchy. You can also replace (override) Eclipse platform implementations in the context. For example you can change the default window close handler.

Via extended object suppliers you can define your custom annotations and provide processors for them. Extended object supplier do not have access to the Eclipse context, but can access data from different sources. For example, you could implement a provider for preferences which are stored in the network instead of the file system.

Model add-ons allow you to contribute Java objects to the application model. A model add-on registers itself typically to events from the `IEclipseBroker` and can contribute functionality based on these events.

Part XLIII. Migrating Eclipse 3.x components and RCP applications

Chapter 175. Why migrating an Eclipse 3.x RCP application?

175.1. Using the Eclipse 3.x API on top of an 4.x runtime

It is possible to run Eclipse 3.x API based plug-ins and RCP applications on top of the Eclipse 4.x platform. The Eclipse 3.x API has been reworked to use the 4.x underneath. This is sometimes called the *compatibility mode*. See [Section 176.1, “Using the compatibility mode”](#) for more information about it.

The Eclipse platform team plans to support this layer for an unlimited period of time. Therefore, if you have an existing Eclipse 3.x application, you do not have to migrate to the Eclipse 4.x API.

175.2. Technical reasons for migrating to the 4.x API

The Eclipse 3.x API is effectively frozen, therefore it is recommended to migrate to the latest 4.x API to participate and benefit from further enhancements in the Eclipse platform.

Another important technical reason to migrate to the Eclipse 4.x API is that it provides a consistent and well-designed API. The concept of the application model and dependency injection makes your application code concise and flexible. Services and the renderer toolkit can be replaced, which adds more flexibility to your implementation.

The event service and the extensible Eclipse context hierarchy provide very powerful ways to communicate within your application.

Chapter 176. Running Eclipse 3.x plug-ins on top of Eclipse 4

176.1. Using the compatibility mode

The Eclipse platform team has re-implemented the existing 3.x API on top of the 4.x API. This re-implementation is sometimes called the compatibility mode. Therefore, you can run an Eclipse 3.x RCP application nearly unmodified on the Eclipse 4 runtime. In this case your application code calls the 3.x API. Call to the 3.x API use internally the 4.x API. Relevant extensions are converted into application model elements at runtime.

Eclipse 3.x API based plug-ins

Eclipse 3.x API

Eclipse 3.x API impl
(uses 4.x API)

Eclipse 4.x runtime
(provides the 4.x API)

This approach requires that your Eclipse 3.x application is based on released 3.x API. If it uses 3.x internals, you might have to adjust your code as this code might have been changed.

If a 3.x RCP application runs on top of an Eclipse 4.x runtime, the platform

translates the information contained in the plugin.xml based extension into model elements. For example the extension point information for views is read and part descriptors are generated based on this information.

In this scenario you can start to convert certain parts of your application into Eclipse 4.x API based parts. Afterwards you have migrated sufficient parts of your application you should switch over completely.

176.2. The e4 API and e4 runtime terminology

This book uses the term *e4 API* as a synonym for *Eclipse 4.x API* and the term *e4 runtime* as a synonym for the *Eclipse 4.x runtime* term.

176.3. Running 3.x RCP applications on top of an e4 runtime

To migrate your Eclipse RCP application to e4 API you have to exchange the target platform to an Eclipse 4.x version.

The following features includes the necessary Eclipse platform plug-ins required for this.

- org.eclipse.rpc
- org.eclipse.emf.ecore
- org.eclipse.emf.common

Add these features (or the included plug-ins) to your product configuration file. Afterwards the Eclipse 3.x API based RCP application starts on top of an e4 runtime without requiring any change in your code.

Note

This assumes that your Eclipse 3.x application does not use any 3.x internal API. Eclipse marks internal packages with the *x-internal* flag in the *MANIFEST.MF* file.

176.4. Benefit of adjusting the runtime to Eclipse 4.x

Running your Eclipse 3.x RCP application on top of an e4 runtime allows you to start migrating components of your application to the e4 API.

It also ensures that you do not use 3.x internal API and that you benefit from the latest Eclipse platform developments, e.g., security updates.

Chapter 177. Excursion: Extending the Eclipse IDE

177.1. Extending the Eclipse IDE

The Eclipse IDE is basically an Eclipse RCP application to support development activities. Even core functionality of the Eclipse IDE is provided via a plug-in, for example the Java development or the C development tools are contributed as a set of plug-ins. Only if these plug-ins are present the Java or C development capabilities are available.

The Eclipse IDE functionality is heavily based on the concept of extensions and extension points. For example the Java Development Tools provide an extension point to register new code templates for the Java editor.

Via additional plug-ins you can contribute to an existing functionality, for example new menu entries, new toolbar entries or provide completely new functionality. But you can also create completely new programming environments.

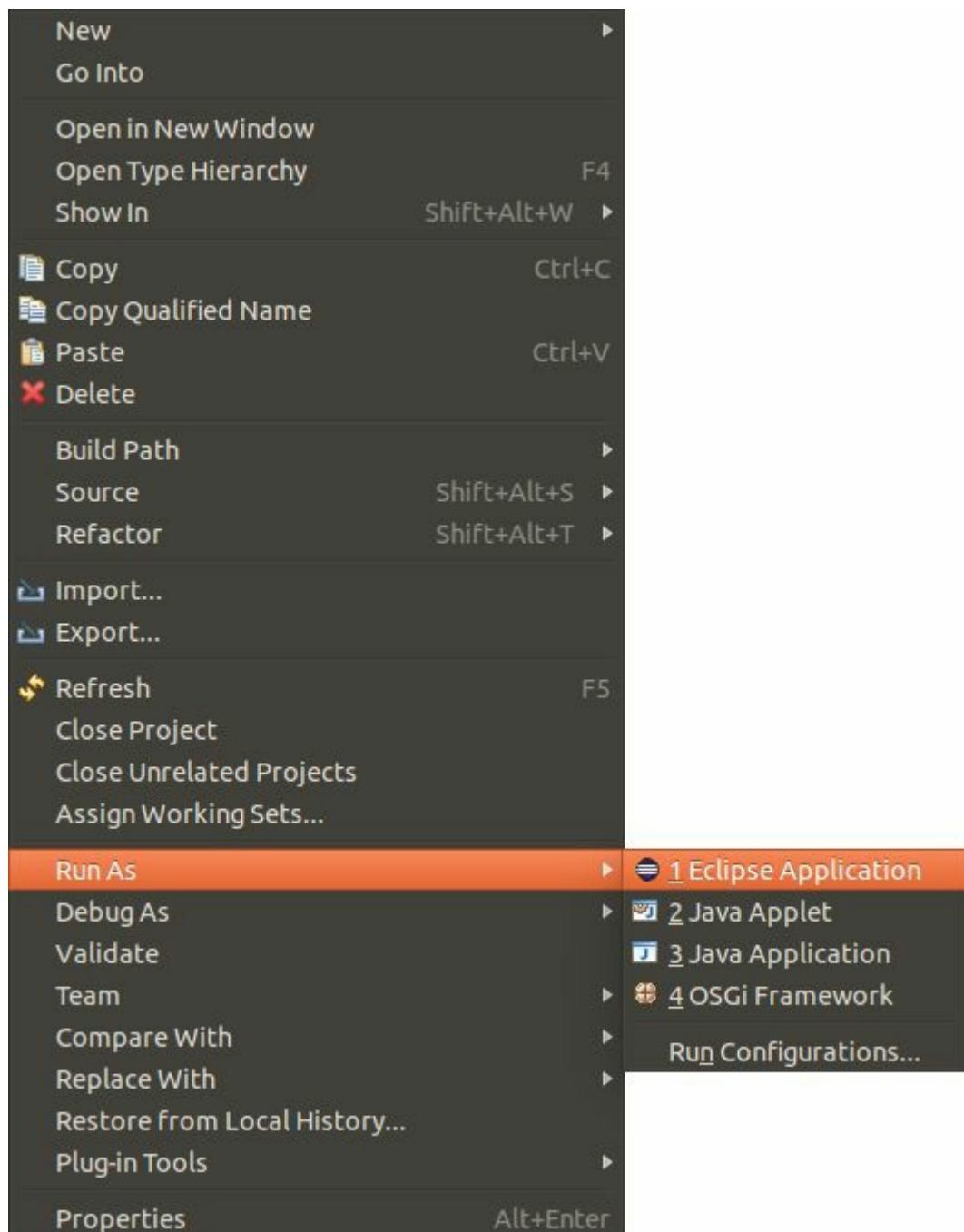
177.2. Starting the Eclipse IDE from Eclipse

During the development of plug-ins targeting the Eclipse IDE you frequently have to deploy your plug-ins into a new Eclipse IDE instance to test your development.

As this is a very common requirement, it is possible to start an instance of the Eclipse IDE from another Eclipse IDE instance. This is sometimes called a *runtime Eclipse IDE*. For this you specify the set of plug-ins which should be included in the Eclipse instance via a run Configuration.

177.3. Starting a new Eclipse instance

The easiest way to create a run configuration for an Eclipse IDE is by selecting Run As → Eclipse Application from the context menu of an existing plug-in or your manifest file. This takes (by default) all your plug-ins from the workspace and your target environment and start an Eclipse IDE instance with these plug-ins.



177.4. Debugging the Eclipse instance

You can debug the Eclipse IDE itself if you want to see how certain things are done. Put a breakpoint in a line of the source code which you want to inspect, right-click your plug-in and select Debug As → Eclipse Application. This creates a run configuration if it does not already exist.

If a statement is reached which contains a breakpoint, Eclipse stops at it and allows you to debug the Eclipse IDE instance.

Chapter 178. Partial migrating to the Eclipse 4.x API

178.1. Using e4 API in 3.x applications

Using e4 API based components in a 3.x applications

It is possible to define new e4 API based components and use them in an application which still uses the Eclipse 3.x API. This allows contributing e4 API based views, menus and toolbars to such an application. By using this approach, you can convert these 3.x components step by step to e4 API. How this can be done is shown in the following chapters.

Extending the Eclipse IDE

The Eclipse IDE uses the same approach as an Eclipse 3.x application on top of an e4 runtime. The e4 API based extensions described in this chapter can, of course, also be used to extend the Eclipse IDE.

178.2. Adding e4 commands, menus and toolbars to 3.x based applications

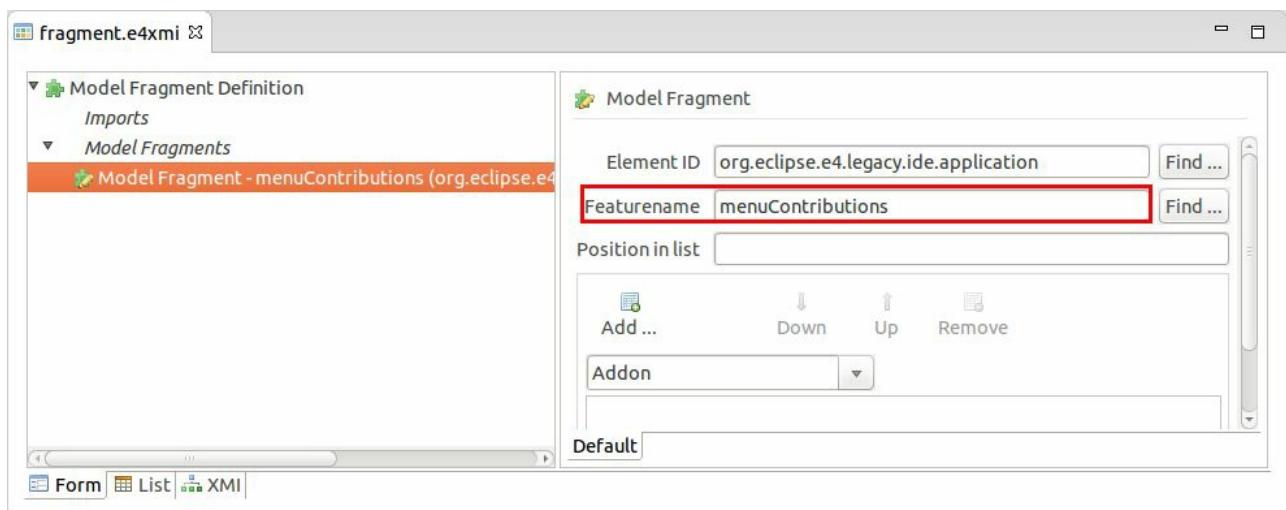
Adding e4 menu entries

Menus, handlers and commands can be contributed to an Eclipse application via model fragments. For this you only need to know the ID of the element to which you want to contribute. The ID of the Eclipse IDE and Eclipse 3.x RCP applications is hard coded to the `org.eclipse.e4.legacy.ide.application` value.

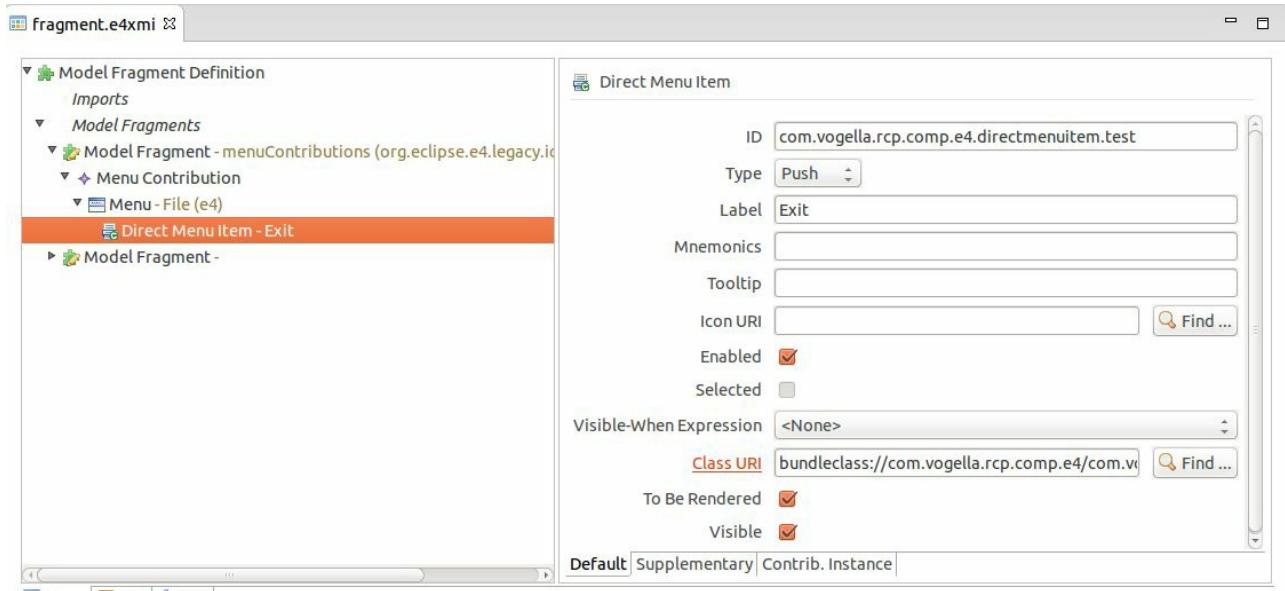
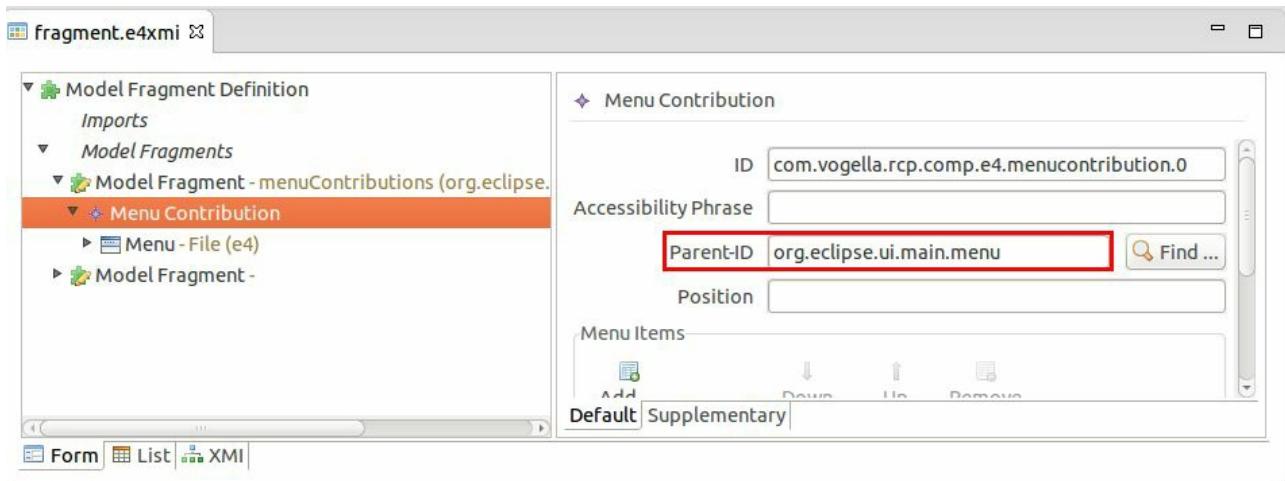
Tip

You can use the model spy from the e4 tools project to identify the ID of the element you want to contribute too. See [Section 27.1, “Analyzing the application model with the model spy”](#).

With the correct ID you can create model fragments that contribute to the corresponding application model element. This is demonstrated by the following screenshots.



The Parent-ID must be the ID of the menu your are contributing to.



The model fragment must be registered in the plugin.xml file via an extension to the `org.eclipse.e4.workbench.model` extension point, as demonstrated in the following listing.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
  <extension
    id="id1"
    point="org.eclipse.e4.workbench.model">
    <fragment
      apply="notexists"
      uri="fragment.e4xmi">
    </fragment>
  </extension>
</plugin>
```

Adding e4 toolbar entries

Similar to menus you can contribute toolbar contributions. This is demonstrated in [Section 179.7, “Adding a toolbar contribution”](#).

178.3. Adding Eclipse 4.x parts and perspectives to 3.x based applications

Adding Eclipse 4.x parts to Eclipse 3.x applications

As a reminder, the term POJO (Plain Old Java Object) is used to describe a Java class without inheritance of a framework class.

As of the Eclipse 4.4 release you can also use a POJO in an extension for the `org.eclipse.ui.views` extension point. Use the `e4view` entry in the context menu of the extension to archive this. The resulting object is created via dependency injection.

For such a kind of view the existing toolbar and view extension point contributions do not work. To add for example a toolbar to your `e4view`, get the `MToolbar` injected into its implementation and construct the entries in your source code.

Supporting part descriptors in an Eclipse 4.5 IDE

The Eclipse 4.5 IDE release will support the contribution of part descriptor model elements via fragments or processors.

If you use the `org.eclipse.e4.legacy.ide.application` ID to contribute your part descriptors, the views can be opened via the Window → Show View → Other... dialog or via the *Quick Access*. This requires that you add the `view` tag to such a part descriptor.

Eclipse 3.x API RCP applications running on top of a 4.5 or higher runtime can use the same approach.

Adding perspectives to the Eclipse IDE via model snippets

Your model fragment or processor can also contribute a perspective to an Eclipse 4.x IDE. For this add a perspective via a snippet. This snippet must again contribute to the application via the `org.eclipse.e4.legacy.ide.application` ID.

This approach can also be used to contribute a perspective to your Eclipse 3.x API based RCP application running on top of an Eclipse 4.x runtime but this depends on the specifics of your application.

178.4. Accessing the IEclipseContext from 3.x API

The Eclipse 4.x IDE uses the `IEclipseContext` data structure to store central information about the IDE. You can access this information also via the Eclipse 3.x API. This approach is demonstrated with the following snippets. These code snippets assume that you are familiar with the 3.x API, if that is not the case you can skip this section.

For example, to access the context from an Eclipse 3.x API view, you can use the following snippet.

```
// get the context of a part
IEclipseContext partContext =
    (IEclipseContext) getViewSite().getService(IEclipseContext.class);

// or access directly a value in the context based on its key
EModelService service =
    (EModelService) getViewSite().getService(EModelService.class);
```

This snippet demonstrates the access via an Eclipse 3.x API handler.

```
// the following example assumes you are in a handler

// get context from active window
IEclipseContext windowCtx =
    (IEclipseContext)
        HandlerUtil.getActiveWorkbenchWindow(event) .
            getService(IEclipseContext.class);

// get context from active part
IEclipseContext ctx =
    (IEclipseContext) HandlerUtil.getActivePart(event) .
        getSite().getService(IEclipseContext.class);
```

Chapter 179. Exercise: Adding an e4 menu and toolbar to a 3.x based application

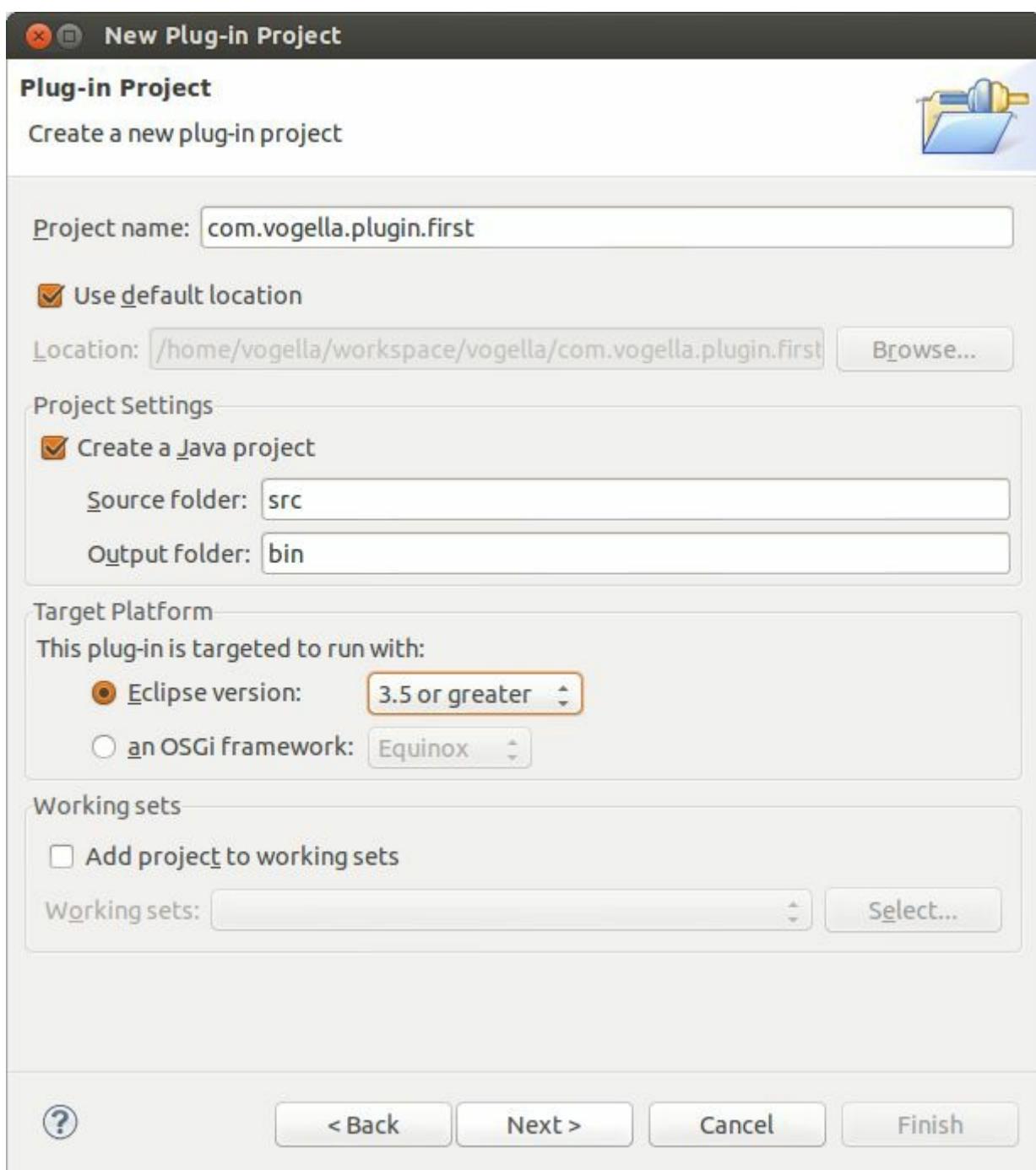
179.1. Target of this exercise

In this exercise you create a plug-in which contributes an e4 menu entry to a 3.x based application menu.

179.2. Creating a plug-in project

Create a new plug-in project called `com.vogella.plugin.first` via File → New → Project... → Plug-in Development → Plug-in Project.

Enter the data as depicted in the following screenshots.



Press the *Next* button.



Press the *Next* button.

Select the *Hello, World Command* template and press the *Next* button. This template uses the 3.x API, which you convert to the e4 API in [Section 179.6, “Creating a model contribution”](#).

New Plug-in Project

Templates

Select one of the available templates to generate a fully-functioning plug-in.

[Create a plug-in using one of the templates](#)

Available Templates:

 Custom plug-in wizard	This wizard creates standard plug-in directory structure and adds the following: <ul style="list-style-type: none">• Command contribution. This template creates a simple command contribution that adds Sample Menu to the menu bar and a button to the tool bar. Both the menu item in the new menu and the button invoke the same Sample Action. Its role is to open a simple message dialog with a message of your choice.
 Hello, World	
 Hello, World Command	
 Plug-in with a multi-page editor	
 Plug-in with a popup menu	
 Plug-in with a property page	
 Plug-in with a view	
 Plug-in with an editor	
 Plug-in with an incremental project builder	
 Plug-in with sample help content	

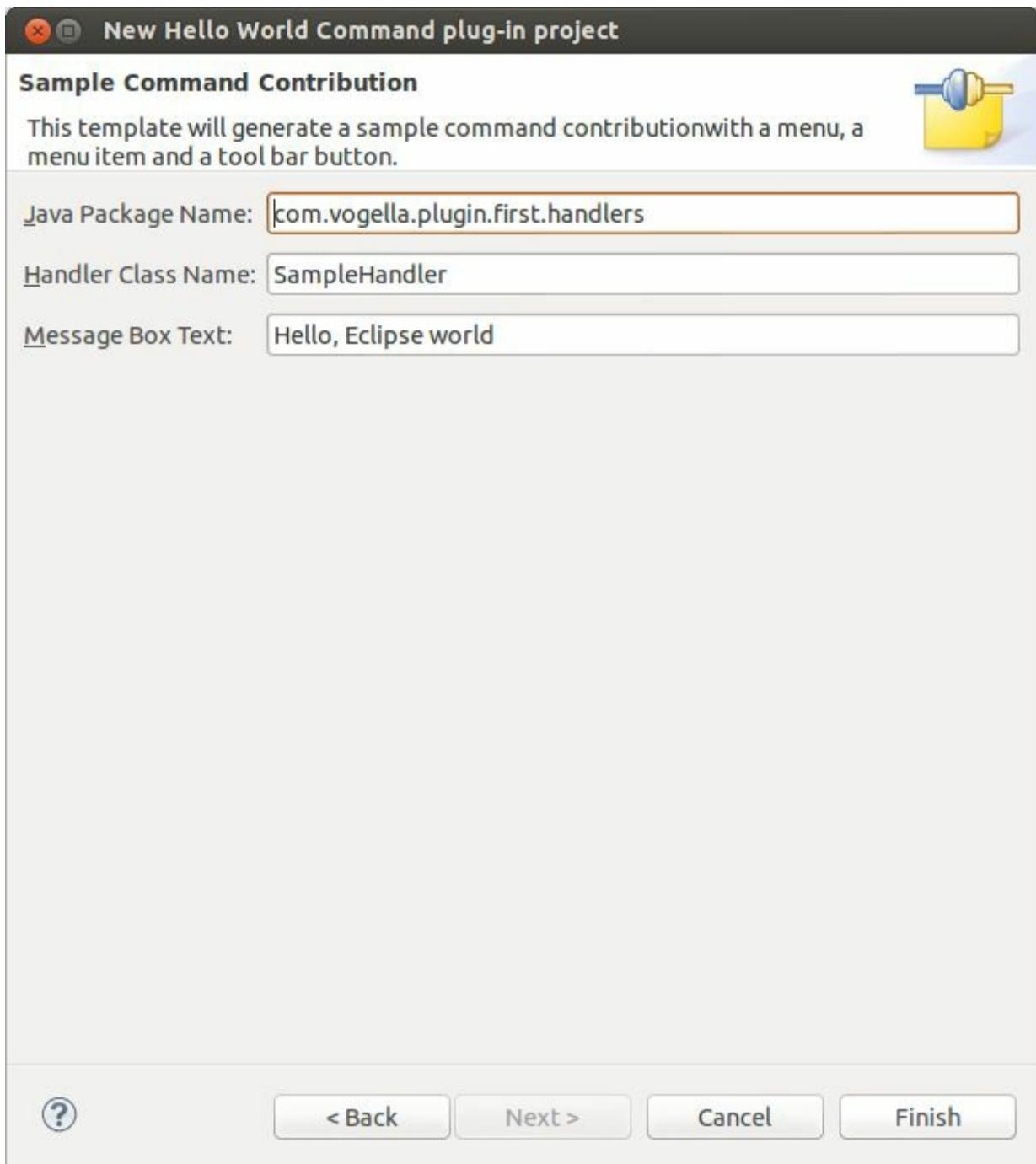
Extensions Used

- org.eclipse.ui.commands
- org.eclipse.ui.handlers
- org.eclipse.ui.bindings
- org.eclipse.ui.menus

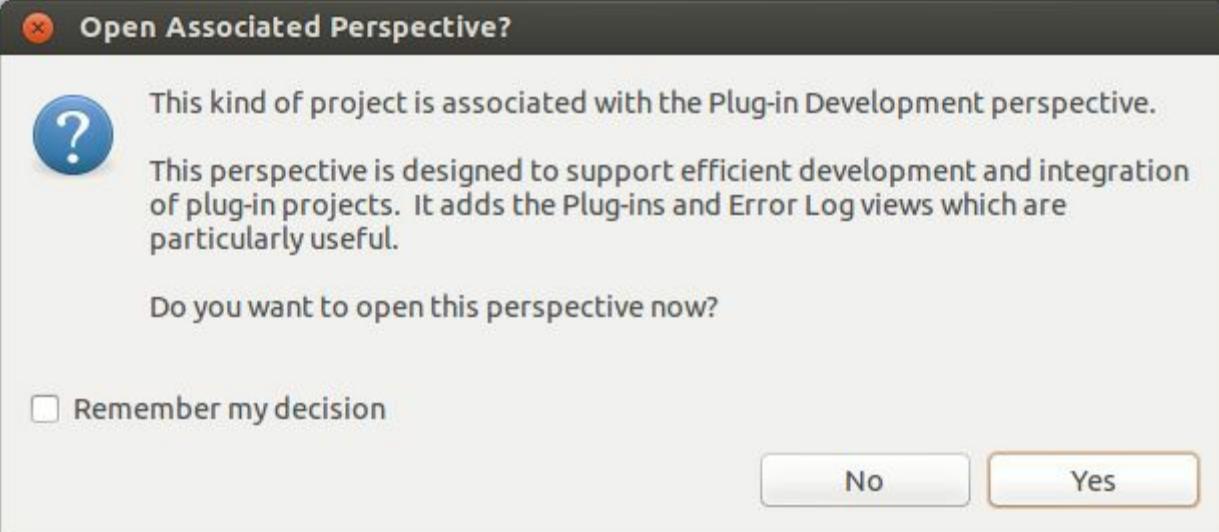
Buttons:

- ? (Help)
- < Back
- Next >
- Cancel
- Finish

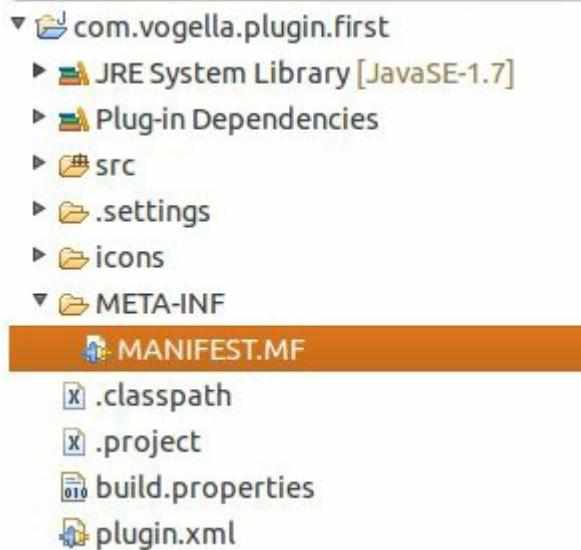
The last page of the wizard allows you to customize the values of the wizard. You can leave the default values and press the *Finish* button.



Eclipse may ask you if you want to switch to the plug-in development perspective. Answer *Yes* if you are prompted.



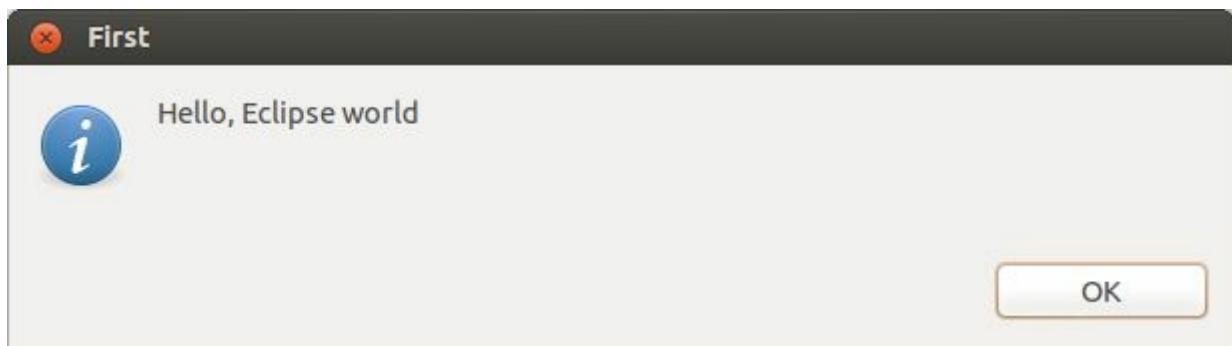
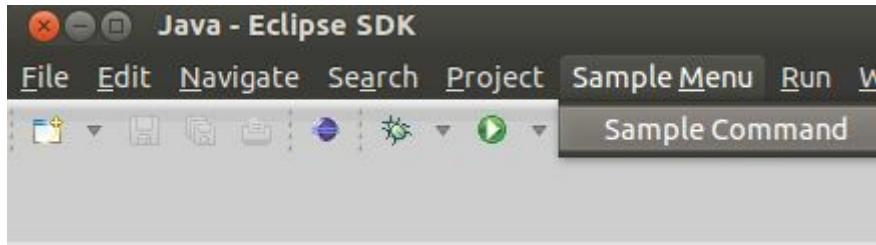
As a result the following project is created.



179.3. Starting an Eclipse IDE with your plug-in

Start a new Eclipse IDE instance and validate that your menu and toolbar entry is available. See [Section 177.3, “Starting a new Eclipse instance”](#) for more information on how to start an instance of the Eclipse IDE with your new plug-in.

A new Eclipse IDE starts, which has your new menu entry included. If you select this menu entry, a message box is displayed.



179.4. Adding the plug-in dependencies for the e4 API

Add a dependency to the `org.eclipse.e4.core.di` plug-in in the manifest file of the newly created plug-in.

179.5. Creating the handler class

Create the following class based on the generated handler class.

```
package com.vogella.plugin.first.handlers;

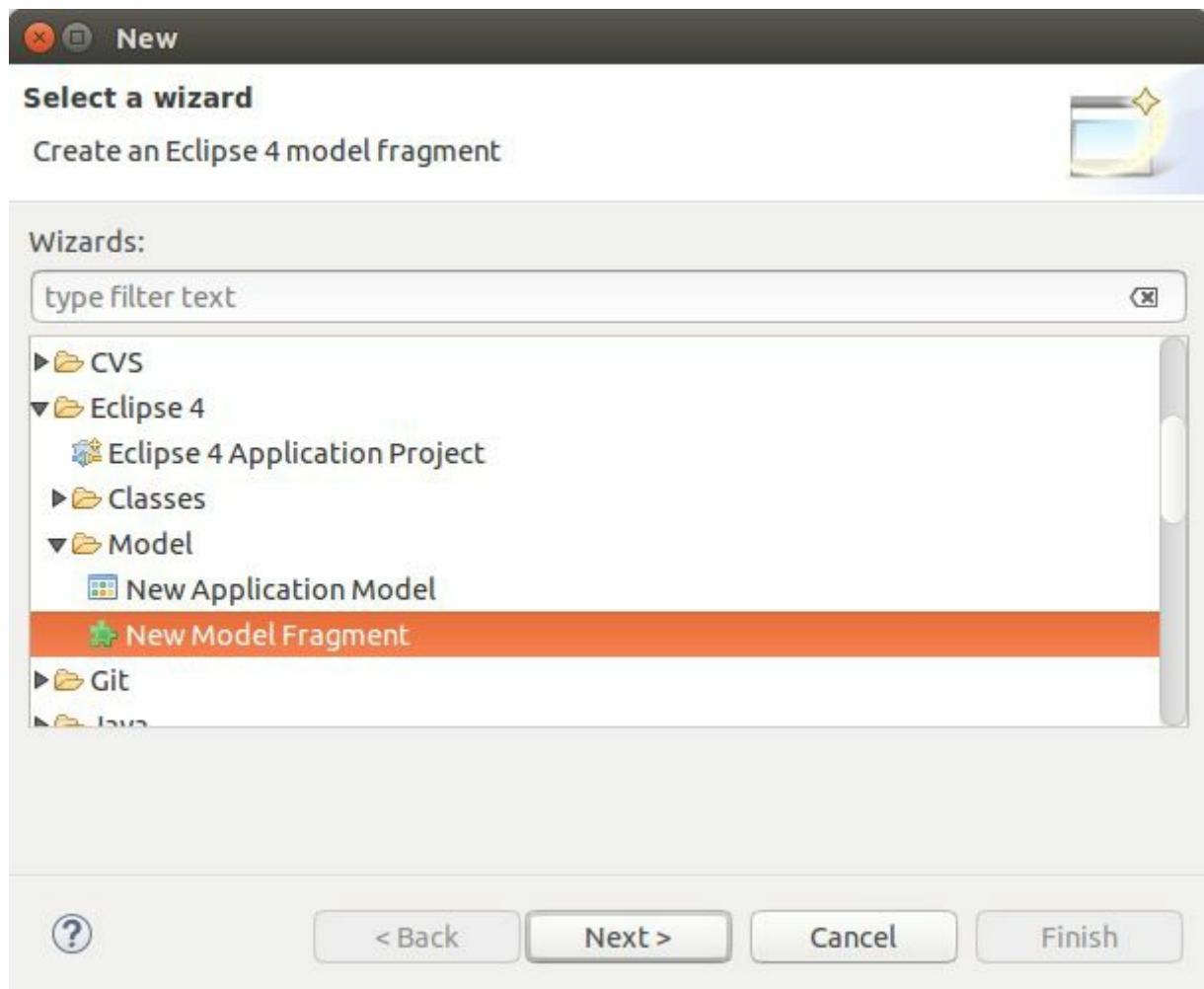
import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Shell;

public class SampleE4Handler {

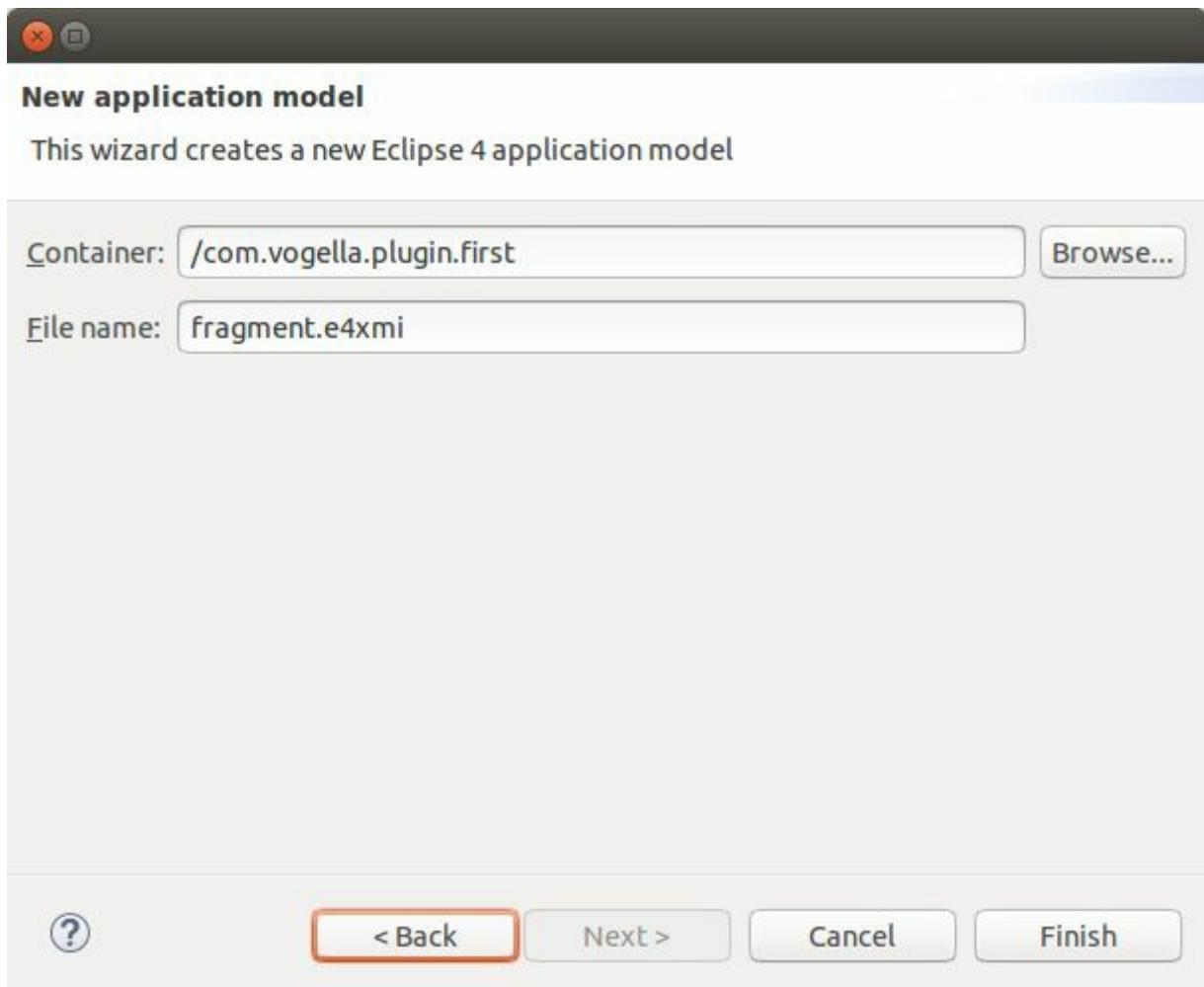
    @Execute
    public void execute(Shell shell) {
        MessageDialog.openInformation(shell, "First", "Hello, e4 API world");
    }
}
```

179.6. Creating a model contribution

Select New → Other... → Eclipse 4 → Model → New Model Fragment from the context menu of the plug-in project.



Press the *Finish* button.



Create three model fragment entries in your new file, all of them should be contributing to the `org.eclipse.e4.legacy.ide.application` element id.

Use the following screenshots to define the new contribution.

The screenshot shows the 'Model Fragments' editor. On the left, there is a tree view with nodes like 'Model Fragments', 'Model Fragment - commands (org.eclipse.e4.legacy.commands)', 'Command - Sample', 'Model Fragment - handlers (org.eclipse.e4.legacy.handlers)', and 'Model Fragment - menuContributions (org.eclipse.e4.legacy.menuContributions)'. The 'Model Fragment - commands' node is selected and highlighted with an orange border. On the right, there are several configuration fields: 'Element ID' set to 'org.eclipse.e4.legacy.ide.application' with a 'Find ...' button; 'Featurename' set to 'commands' with a 'Find ...' button; 'Position in list' with a dropdown menu showing 'Addon' and buttons for 'Add', 'Remove', and 'Down'. Below these fields is a list box containing 'Command - Sample'.

Model Fragment Definition

- Imports
- Model Fragments
 - Model Fragment - commands (org.eclipse.e4.legacy)
 - Command - Sample**
 - Model Fragment - handlers (org.eclipse.e4.legacy)
 - Model Fragment - menuContributions (org.eclipse.e4.legacy)

Command

ID: com.vogella.plugin.first.command.sample
 Name: Sample
 Description:
 Category:
 Parameters:

Model Fragment Definition

- Imports
- Model Fragments
 - Model Fragment - commands (org.eclipse.e4.legacy)
 - Command - Sample**
 - Model Fragment - handlers (org.eclipse.e4.legacy)
 - Handler - Sample**
 - Model Fragment - menuContributions (org.eclipse.e4.legacy)

Model Fragment

Element ID: org.eclipse.e4.legacy.ide.application
 Featurename: handlers
 Position in list:
 Addon:
Handler - Sample

Model Fragment Definition

- Imports
- Model Fragments
 - Model Fragment - commands (org.eclipse.e4.legacy)
 - Command - Sample**
 - Model Fragment - handlers (org.eclipse.e4.legacy)
 - Handler - Sample**
 - Model Fragment - menuContributions (org.eclipse.e4.legacy)

Handler

ID: com.vogella.plugin.first.handler.0
 Command: Sample - com.vogella.plugin.first.command.sample
 Class URI: bundleclass://com.vogella.plugin.first/com.vogella.plugin.first.handlers.SampleE4Handler
 Persisted State:
 Value:

Model Fragment Definition

- Imports
- Model Fragments
 - Model Fragment - commands (org.eclipse.e4.legacy)
 - Command - Sample**
 - Model Fragment - handlers (org.eclipse.e4.legacy)
 - Handler - Sample**
 - Model Fragment - menuContributions (org.eclipse.e4.legacy)
 - Menu Contribution
 - Menu - e4 sample menu

Model Fragment

Element ID: org.eclipse.e4.legacy.ide.application
 Featurename: menuContributions
 Position in list:
 Addon:
Menu Contribution

Model Fragment Definition

- Imports
- Model Fragments
 - Model Fragment - commands (org.eclipse.e4.legacy)
 - Command - Sample**
 - Model Fragment - handlers (org.eclipse.e4.legacy)
 - Handler - Sample**
 - Model Fragment - menuContributions (org.eclipse.e4.legacy)
 - Menu Contribution
 - Menu - e4 sample menu

Menu Contribution

ID: com.vogella.plugin.first.menucontribution.0
 Accessibility Phrase:
 Parent-ID: org.eclipse.ui.main.menu
 Position: after=additions
 Menu Items:

Model Fragment Definition

- Imports
- Model Fragments
 - Model Fragment - commands (org.eclipse.e4.core.commands)
 - Command - Sample
- Model Fragment - handlers (org.eclipse.e4.core.services)
- Handler - Sample
- Model Fragment - menuContributions (org.eclipse.e4.ui.menu)
- Menu Contribution
- Menu - e4 sample menu
 - Handled Menu Item - e4 sample menu

Menu

ID: com.vogella.plugin.first.menu.newMenu

Label: e4 sample menu

Mnemonics:

Children:

- Handled Menu Item
- Add
- Remove
- Down
- Up

Handled Menu Item

ID: id.e4Sample

Type: Push

Label: e4 sample menu

Mnemonics:

Tooltip:

Icon URI: platform:/plugin/com.vogella.plugin.first/icons/sample.gif

Enabled:

Selected:

Visible-When Expression: <None>

Command: Sample - com.vogella.plugin.first.command.sample

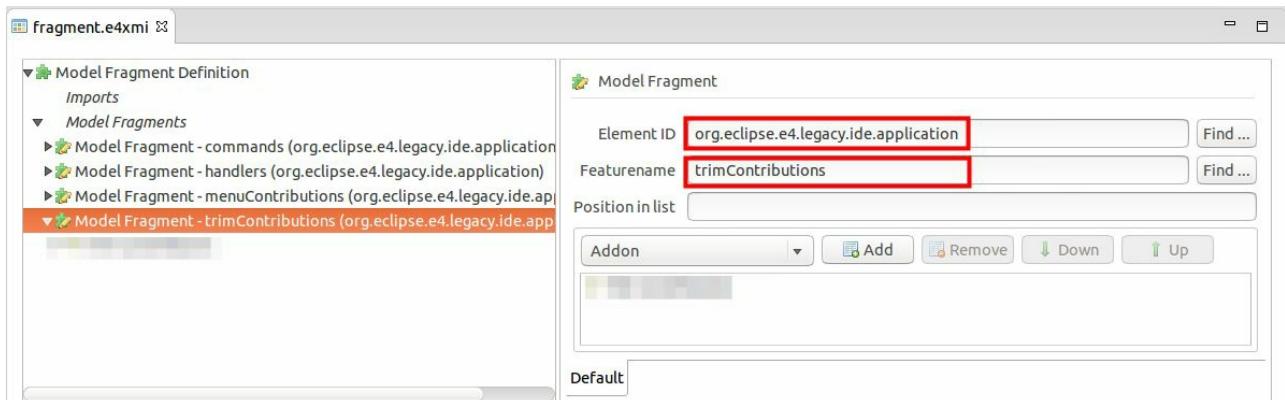
To Be Rendered:

Visible:

Default: Supplementary

179.7. Adding a toolbar contribution

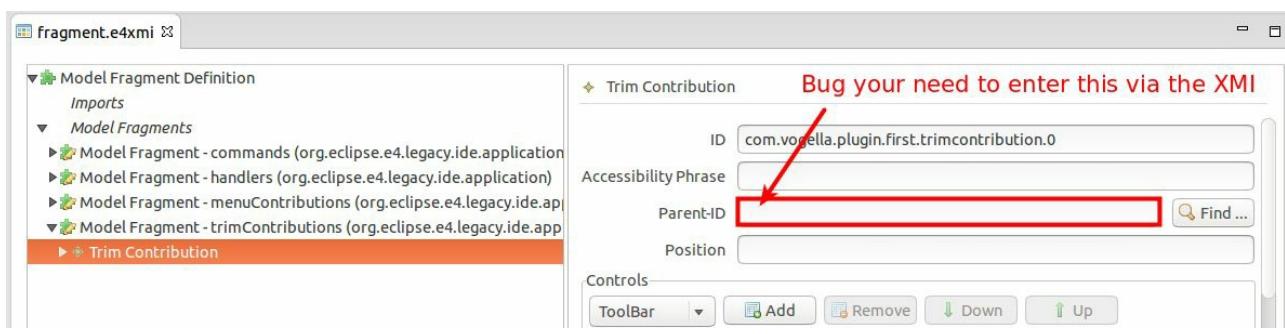
Also add a toolbar contribution for the same command.



Warning

When this document was written, the model editor had a bug. If you enter a *Parent-ID* to the toolbar contribution, that information is not persisted in the xmi code. Therefore, ensure that your settings are actually reflected in the fragment file. You can do this by closing the file and opening it again.

If you read this and the bug is still present, you have to enter the Parent-ID directly in the xmi. For this switch to the XMI tab of the editor and enter the parent ID directly. Use the `org.eclipse.ui.main.toolbar` value. The file is only saved, if you enter the information syntactically correct.



```
<fragments xsi:type="fragment:StringModelFragment" xmi:id="_xr
<elements xsi:type="menu:TrimContribution"
  xmi:id="_4LRJcMaBEESYi86KPuHLFg"
  elementId="com.vogella.plugin.first.trimcontribution.0"
  parentId="org.eclipse.ui.main.toolbar">
```

Add a toolbar and a handled tool item to your contribution.

The screenshot shows the Eclipse Model Fragment editor interface. On the left, there is a tree view of the model fragment structure:

- Model Fragment - trimContributions (org.eclipse.e4.legacy.ide.app)
- Trim Contribution
- Toolbar

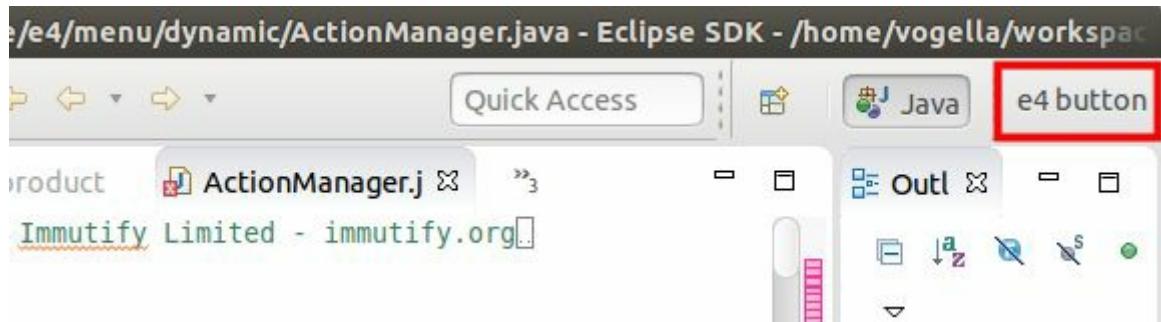
The "Toolbar" node is selected, highlighted with an orange bar at the bottom of the tree. On the right, the properties for this selection are displayed in a form:

Handled Tool Item

ID	com.vogella.plugin.first.handledtoolitem.e4Button
Type	Push
Label	e4 button
Accessibility Phrase	
Tooltip	
Icon URI	
Find ...	
Menu	<input type="checkbox"/>
Enabled	<input checked="" type="checkbox"/>
Selected	<input type="checkbox"/>
Visible-When Expression	<None>
Command	Sample - com.vogella.plugin.first.command.sample
Find ...	

179.8. Validating the presence of the menu and toolbar contribution

Start a new instance of the Eclipse IDE and validate that your menu and the toolbar are contributed. If they are not visible in the window, check via the model spy for potential issues.



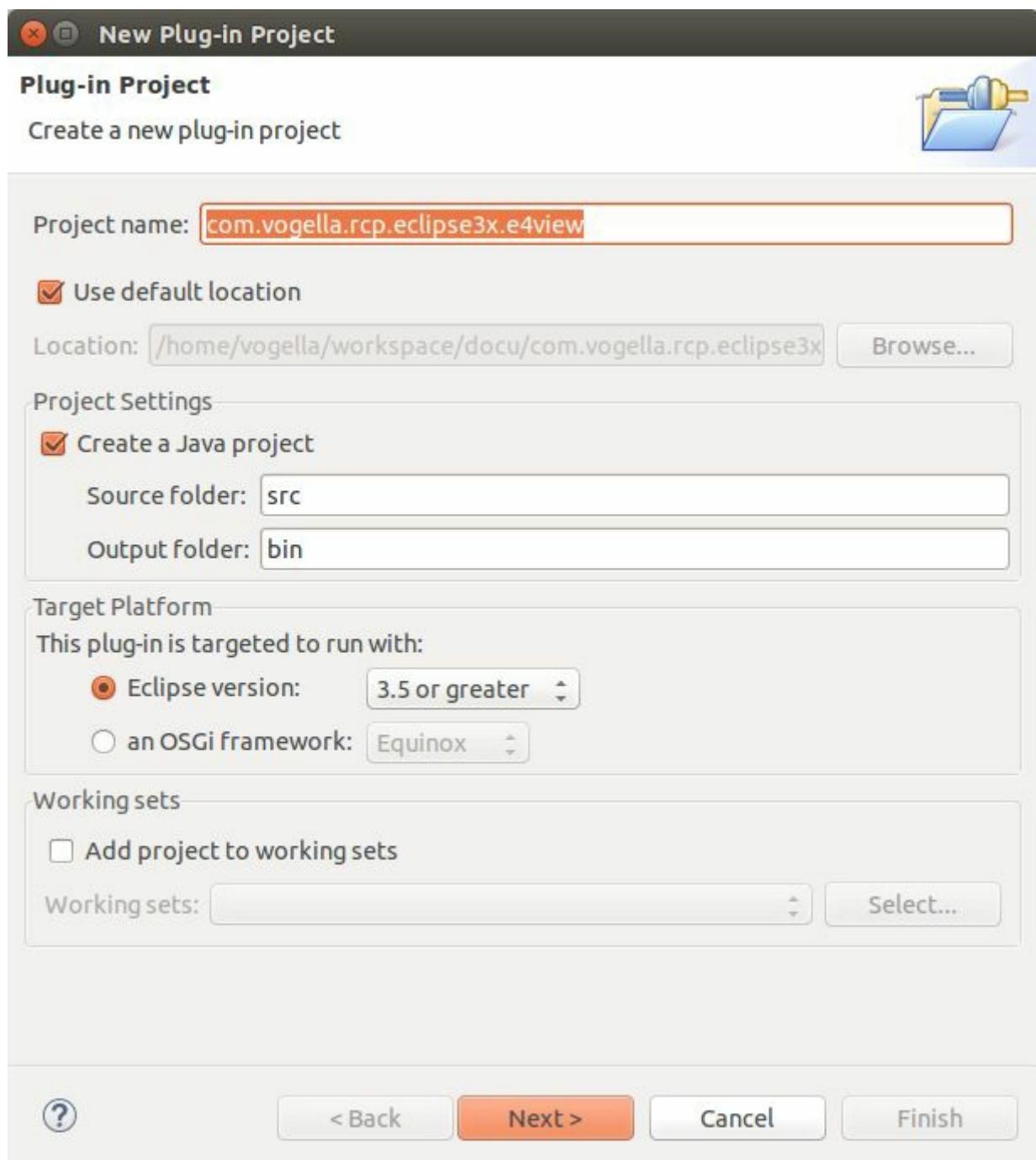
Chapter 180. Exercise: Using POJOs to contribute views to a 3.x based application

180.1. Target

In this exercise you create an Eclipse 3.x RCP application and add a POJO part to it based on the e4view element available to contribute an extension point based view.

180.2. Using e4part and the org.eclipse.ui.views extension point

Create an Eclipse 3.x RCP application based on the *RCP application with a view template* via File → New → Other... → Plug-in Project. Call the project `com.vogella.rcp.eclipse3x.e4view` and select the options similar to the following screenshots.



 **New Plug-in Project**

Content 

Enter the data required to generate the plug-in.

Properties

ID: com.vogella.rcp.eclipse3x.e4view

Version: 1.0.0.qualifier

Name: E4view

Vendor: VOGELLA

Execution Environment: JavaSE-1.7  Environments...

Options

Generate an activator, a Java class that controls the plug-in's life cycle
Activator: com.vogella.rcp.eclipse3x.e4view.Activator

This plug-in will make contributions to the UI

Enable API analysis

Rich Client Application

Would you like to create a 3.x rich client application?  Yes  No

New Plug-in Project

Templates

Select one of the available templates to generate a fully-functioning plug-in.

Create a plug-in using one of the templates

Available Templates:

 Hello RCP	This wizard creates a standalone RCP application that consists of an application window with a single view.
 RCP application with a view	Extensions Used <ul style="list-style-type: none">org.eclipse.core.runtime.applicationsorg.eclipse.ui.perspectivesorg.eclipse.ui.views
 RCP application with an intro	

?

< Back Next > Cancel Finish

Add `org.eclipse.e4.ui.di` as a dependency to the manifest of your new plug-in.

Create the following class.

```
package com.vogella.rcp.eclipse3x.e4view;

import javax.annotation.PostConstruct;

import org.eclipse.e4.ui.di.Focus;
import org.eclipse.jface.viewers.ArrayContentProvider;
```

```

import org.eclipse.jface.viewers.LabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;

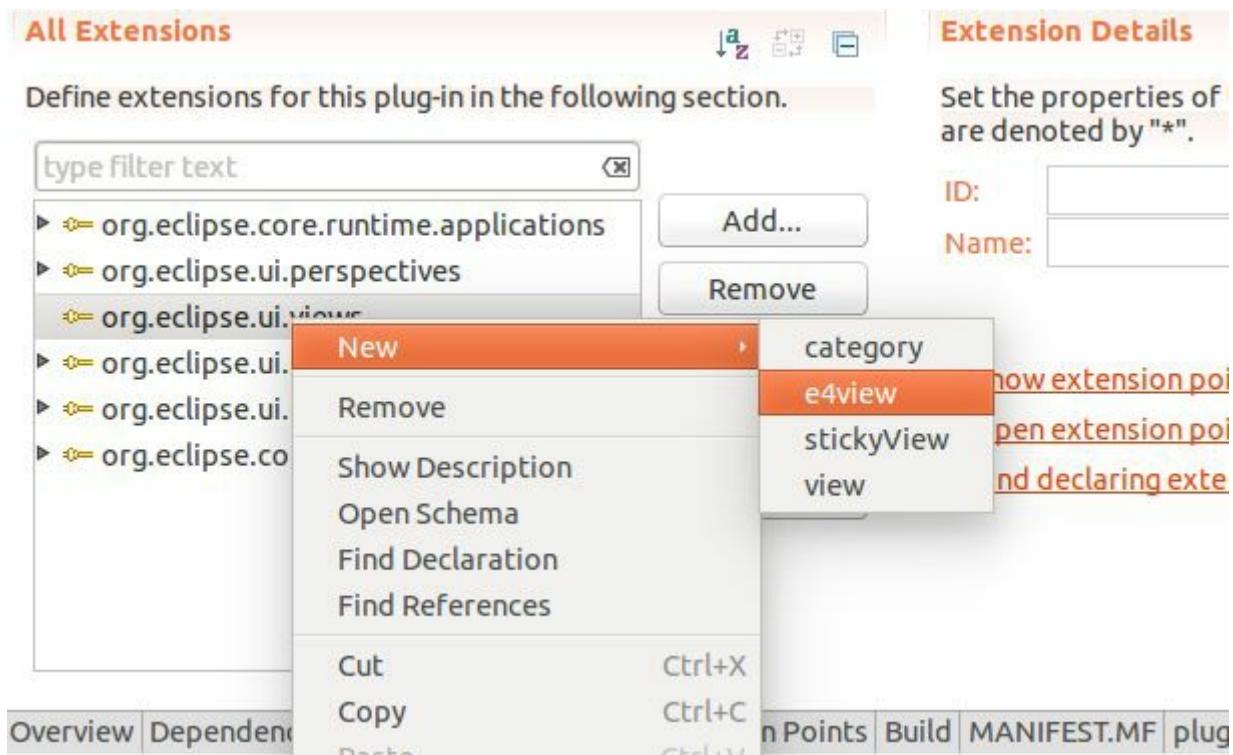
public class ViewEclipse4x {
    private TableViewer viewer;

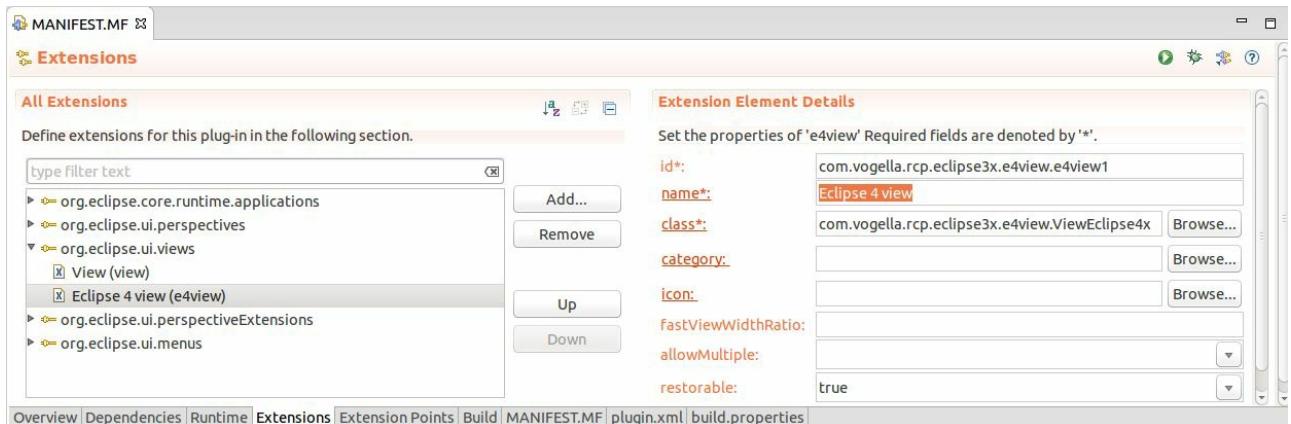
    @PostConstruct
    public void createPartControl(Composite parent) {
        viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL
            | SWT.V_SCROLL);
        viewer.setContentProvider(ArrayContentProvider.getInstance());
        viewer.setLabelProvider(new LabelProvider());
        viewer.setInput(new String[] {"One", "Two", "Three"});
    }

    @Focus
    public void setFocus() {
        viewer.getControl().setFocus();
    }
}

```

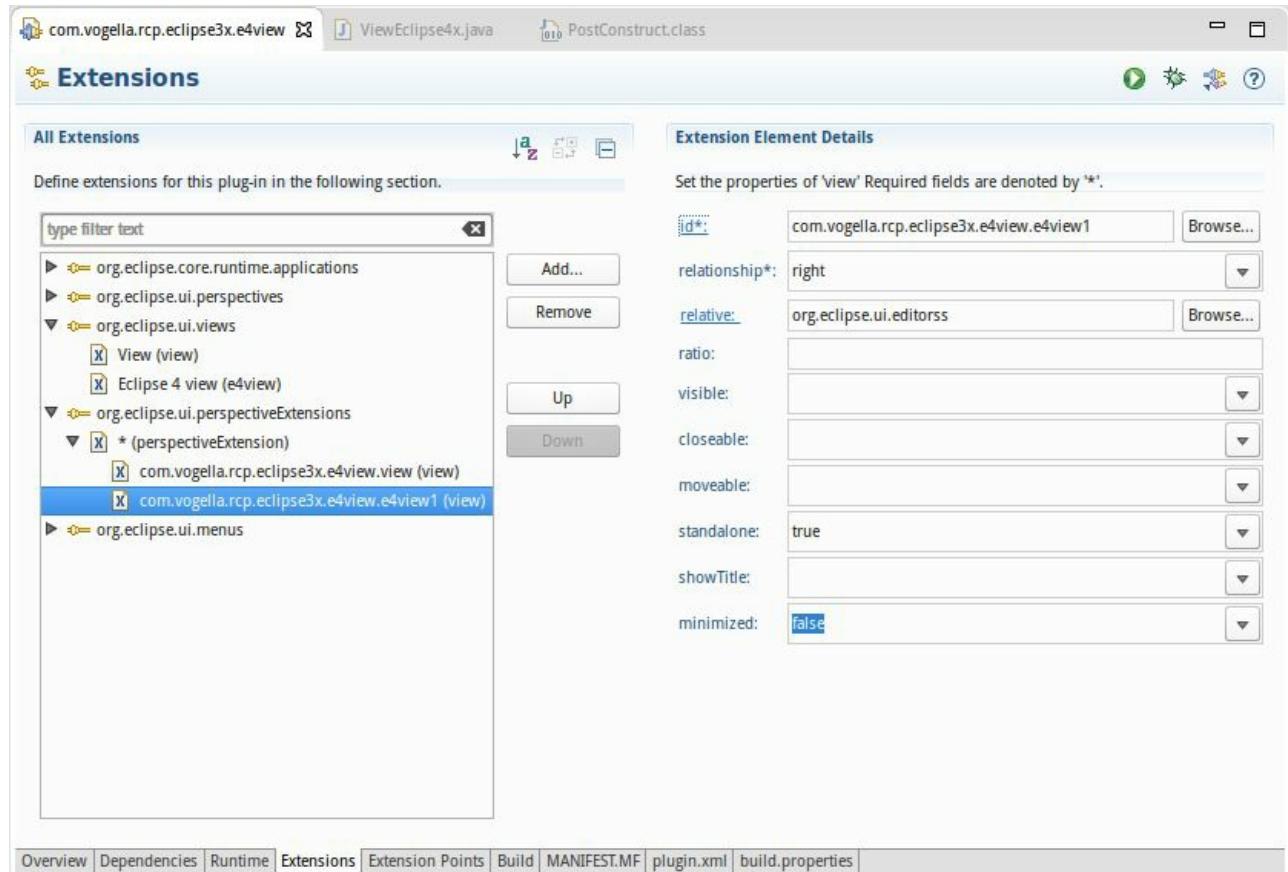
Afterwards add an *e4view* extension for the `org.eclipse.ui.views` extension point. For this use the *Extensions* tab of the `plugin.xml` editor.





180.3. Add the view to a perspective extension

To be able to see the new view, add it to the existing perspective extension as depicted in the following screenshot.



180.4. Validating

Start your Eclipse 3.x application and validate that the new part is displayed.

Chapter 181. Exercise: Adding e4 part descriptors to 3.x based applications

181.1. Target

In this exercise you add a model based part contribution to an Eclipse 3.x RCP application.

181.2. Adding a part descriptor

Create a simple plug-in called `com.vogella.plugin.partdescriptor`.

Add the following dependencies to your manifest file.

- `org.eclipse.core.runtime`
- `org.eclipse.jface`
- `org.eclipse.e4.ui.model.workbench`
- `org.eclipse.e4.ui.di`

Create the following class

```
package com.vogella.plugin.partdescriptor;

import javax.annotation.PostConstruct;

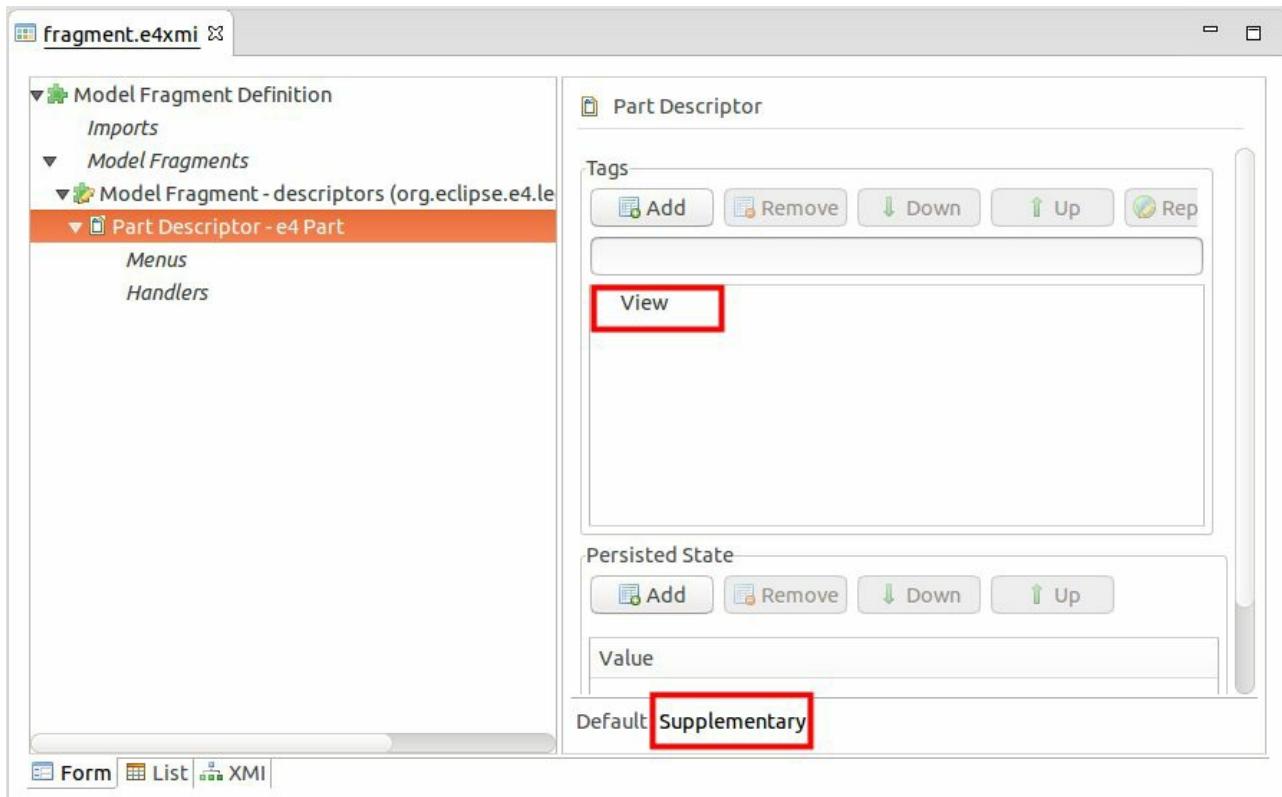
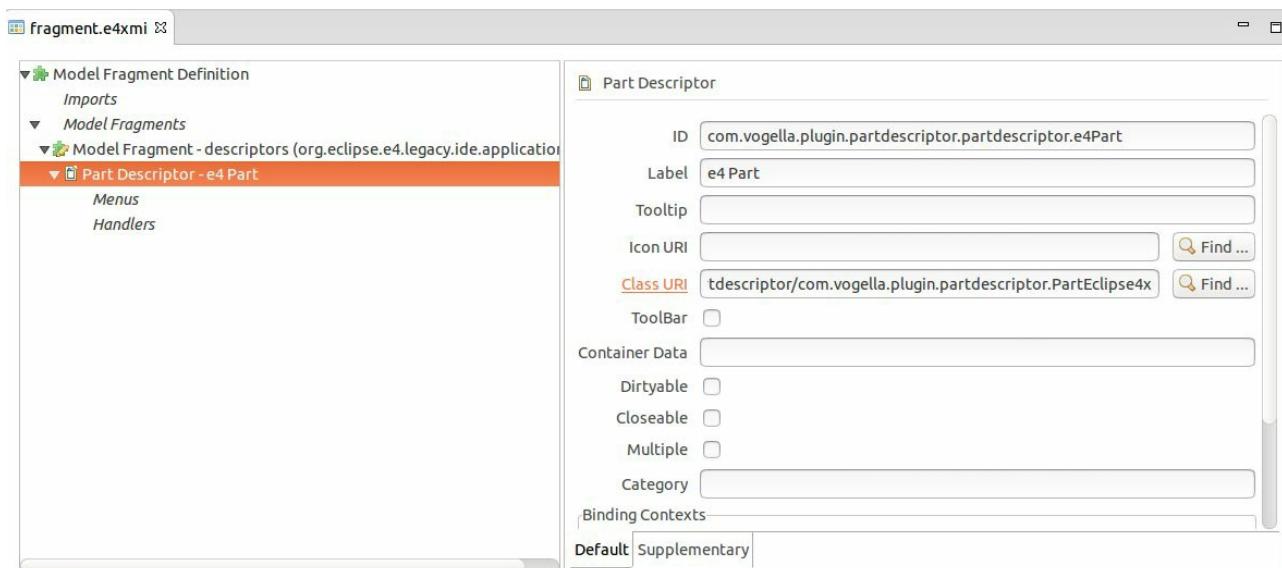
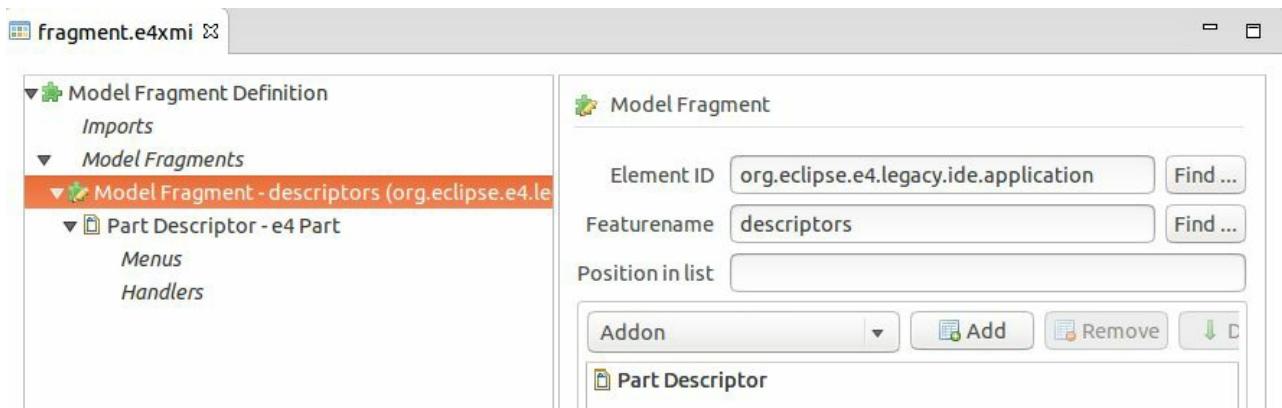
import org.eclipse.e4.ui.di.Focus;
import org.eclipse.jface.viewers.ArrayContentProvider;
import org.eclipse.jface.viewers.LabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;

public class PartEclipse4x {
    private TableViewer viewer;

    @PostConstruct
    public void createPartControl(Composite parent) {
        viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL
            | SWT.V_SCROLL);
        viewer.setContentProvider(ArrayContentProvider.getInstance());
        viewer.setLabelProvider(new LabelProvider());
        viewer.setInput(new String[] {"One", "Two", "Three"});
    }

    @Focus
    public void setFocus() {
        viewer.getControl().setFocus();
    }
}
```

Add a new model fragment to this plug-in via `File → New → Other... → Eclipse 4 → Model → New Model Fragment`.



181.3. Validating

Start an instance of the Eclipse IDE and validate that you can open the parts via the Quick Access box (shortcut Ctrl+3).

Note

In Eclipse 4.4 the part descriptor can only be opened via the Quick Access box. In Eclipse 4.5 it is planned that it can also be opened via the Window → Show View menu entry.

Chapter 182. Optional Exercise: Model add-on to change the close behavior of the IDE

182.1. Target

In this exercise you register an existing model add-on created in [Chapter 126, Exercise: Model add-on to change the close behavior](#) with the Eclipse IDE. This is done similar to the approach described in [Section 179.6, “Creating a model contribution”](#), the only difference is that you contribute a model add-on.

182.2. Register model add-on via model fragment with the Eclipse IDE

Create a model fragment which contributes the add-on to the application of the Eclipse IDE.

The screenshot shows two views of the Eclipse IDE interface, likely from the 'Model Fragment' editor.

Top View: A 'fragment.e4xmi' editor window. On the left, the tree view shows a 'Model Fragment Definition' node with 'Imports' and 'Model Fragments' children. One 'Model Fragment' child is selected, highlighted in orange, labeled 'Model Fragment - addons (org.eclipse.e4.legacy.ide.application)'. On the right, the details view shows:

- Model Fragment:** Element ID: org.eclipse.e4.legacy.ide.application
- Addon:** Featurename: addons
- Position in list:** (empty)
- Addon:** Add, Remove buttons
- Add-on:** A list containing one item: 'Add-on - com.vogella.e4.addon.exitconfirmation.WindowCloseListenerAddon'.

Bottom View: Another 'fragment.e4xmi' editor window. The tree view is identical to the top one. The details view on the right shows the configuration for the selected 'Add-on':

- Add-on:** ID: com.vogella.e4.addon.exitconfirmation.addon.0
- Class URI:** com.vogella.e4.addon.exitconfirmation.WindowCloseListenerAddon
- Persisted State:** Add, Remove, Down, Up buttons
- Value:** (empty)

182.3. Verifying

Verifying the model add-on in the Eclipse IDE

Start a runtime Eclipse IDE with your plug-in included in the launch configuration. Ensure that your new close dialog is used if you try to close the IDE with a dirty editor.

Chapter 183. Migrating to an e4 API application without 3.x API usage

183.1. Migrating an Eclipse 3.x application completely to the e4 API

If you want to migrate your Eclipse 3.x RCP application completely to the e4 API, you can't directly reuse existing `plugin.xml` based user interface components, e.g., 3.x API based views or editors.

Migrating these components involves:

- Converting existing user interface contributions via extension points to the application model.
- Removing the usage of Singleton objects of the Eclipse platform, e.g. `Platform` or `PlatformUI` and use the dependency injection programming model.

183.2. Using 3.x components in e4 API based applications

Using Eclipse 3.x API based components in e4 API based application is currently not officially supported. Customers which are willing to debug and fix potential issues can still use this approach. As the potential issues depend on the plug-ins which are re-used and as this approach is officially not supported, it is hard to predict which issues are encountered.

Note

The difference to [Section 178.1, “Using e4 API in 3.x applications”](#) is that in the scenario described in this chapter you have an application that uses only the e4 API, i.e., the compatibility layer is not present.

While not supported, some implementations still use this approach which various workarounds. See [Section 183.5, “Example for using 3.x components in e4 API based applications”](#) for a small example project.

183.3. Reusing platform components

Most Eclipse IDE components are still based on the Eclipse 3.x API. The Eclipse platform plans to migrate more and more existing components to the e4 API in future releases. This means that certain components which could be reused in Eclipse 3.x RCP application are not available by default.

Currently most components, for example, the IDE perspective switcher, the view switcher, the *Progress* view, the new project wizard are based on the Eclipse 3.x API and therefore cannot be used without modifications in an e4 API based application.

183.4. Existing replacements for 3.x components for e4 API based applications

The [vogella GmbH](#) provides an Open Source project for an e4 based perspective switcher. See the following URL for details: [e4 API based perspective switcher](#)

The Eclipse platform team provides also an Eclipse 4 based progress view. See the following URL for details: <https://wiki.eclipse.org/Eclipse4/ProgressView>

183.5. Example for using 3.x components in e4 API based applications

If you choose to try this approach despite the warning, you have to define a `LegacyIDE.e4xmi` file similar to the Eclipse IDE and add your model components to this file. This file can be found in the `org.eclipse.ui.workbench` plug-in.

Point to that file via the `applicationXMI` parameter in your `org.eclipse.core.runtime.products` extension.

A code example for this approach can be found under the following URL:

<https://github.com/fredrikattebrant/Simple-4x-RCP-app>

As said before, this approach is not officially supported by the Eclipse platform team. If you use this approach, you may find hard to solve issues.

Part XLIV. Questions, feedback and closing words

Chapter 184. Questions, feedback

184.1. Reporting Eclipse bugs and feature requests

The Eclipse bug and feature tracker is using the open source *Bugzilla* project from *Mozilla*. In this system you enter error reports for bugs you encounter with the usage of Eclipse and also to request new feature or improvements of existing features.

This bug tracker can be found under [Eclipse Bugzilla](#). Here you can search for existing bugs and review them.

184.2. Using the Eclipse bugzilla system

To participate actively in the Eclipse bug tracker, you need to create a new account. This can be done by pressing the *Create a New Account* link.

The screenshot shows the Eclipse Bugzilla main page. At the top, there's a header with the Eclipse logo and a 'DOWNLOAD' button. Below the header, a dark bar says 'Bugzilla - Main Page'. Underneath, there's a navigation bar with links: Home, New, Browse, Search, Search input field, Help (?), Reports, Requests, Log In, Forgot Password, and Terms of Use. The main content area has a title 'Welcome to Bugzilla' and three large buttons: 'File a Bug' (document icon), 'Search' (magnifying glass icon), and 'Create a New Account' (person icon). Below these buttons is a section titled 'Most requested bugs' with a search input field, a 'Quick Search' button, and a 'Quick Search help' link. At the bottom of the page, there are links to 'Bugzilla User's Guide' and 'Release Notes'.

Once you have a user account, you can login to the Eclipse bug tracker. This allows you to comment on existing bugs and report new ones.

Note

The user data for all Eclipse sites are the same, i.e., the forum, marketplace, bug tracker, etc. Only for the Gerrit access, different user data is used.

184.3. Eclipse bug priorities

The Eclipse Bugzilla system allows you and the Eclipse committer to enter the bug priority. But overall, it is up to each project to decide how they handle bugs so some variation from project to project will occur. The following rules can be used as guideline.

Table 184.1. Bug priorities

Priority	Description
blocker	The bug blocks development or testing of the build and no workaround is known.
critical	Implies "loss of data" or frequent crashes or a severe memory leak.
major	Implies a "major loss of function".
normal	This is the default value for new bug reports. Implies some loss of functionality under specific circumstances, typically the correct setting unless one of the other levels fit.
minor	Something is wrong, but doesn't affect function significantly or other problem where easy workaround is present.
trivial	This describes a cosmetic problem like misspelled words or misaligned text, but doesn't affect function.
enhancement	Represents a request for enhancement (also for "major" features that would be really nice to have).

184.4. Asking (and answering) questions

Due to the complexity and extensibility of Eclipse, you will need additional resources to help you solve your specific problems. Fortunately, the web contains several resources which can help you with your Eclipse problems.

Currently, the best places to find, ask and answer questions are the [Eclipse forums](#) and [Stack Overflow](#). Try to stay polite with your postings, as the Eclipse community values polite behavior.

The *Eclipse forums* offer several topic-specific forums in which you can post and answer questions. To post or to answer questions in the Eclipse forums, you need a valid user account in the Eclipse bug tracker. The advantage of the Eclipse forums is that, depending on the topic, developers of Eclipse projects (*Eclipse committers*) are also active there and might directly answer your question.

Stack Overflow also requires a user account and its community is also very active. *Stack Overflow* does not have separate forum sections for specific questions. *Stack Overflow* allows to tag questions with the relevant keyword, e.g., *Eclipse* and people search for them or subscribe to these theses.

Note

Ensure that you search the forums and mailing lists for solutions for your problem since often somebody else has asked the same question earlier and the answer is already available.

184.5. Eclipse 4 feedback

The *Eclipse forums* have a forum dedicated to Eclipse 4 questions which can be found under the following URL: <http://www.eclipse.org/forums/eclipse.e4>

To report bugs and feature requests specific to the Eclipse 4 platform please use the following link and the *UI* component:
https://bugs.eclipse.org/bugs/enter_bug.cgi?product=Platform

To report bugs and feature requests specific to the *Eclipse e4 tooling* platform please use the following link and the *UI* component:
https://bugs.eclipse.org/bugs/enter_bug.cgi?product=e4

Chapter 185. Closing words

I hope you enjoyed your learning experience with the Eclipse RCP framework.

The Eclipse platform provides a modern, flexible and powerful programming model. Building your application based on this framework enables you to leverage these capabilities and to create powerful native applications.

The Eclipse platform is also under active development. With developers from Google, Redhat, IBM and smaller companies and individuals the frameworks gets extended and improved in every release.

The development team is also open for feedback and code contributions, so you can help to enhance the framework in a direction you find important.

Please use the Eclipse Forum or the Eclipse bug tracker to provide feedback and bug reports and the Eclipse Gerrit code review system to provide code contributions.

I am personally very impressed by the ability of our Eclipse development community in providing this level of code stability combined with this level of innovation. I believe this ability and the existing ecosystem will ensure that the Eclipse programming model will remain one of the strongest programming models for creating native clients.

Lars Vogel
Eclipse Platform UI and e4 Co-Lead
vogella GmbH CEO

Part XLV. Appendix

Appendix A. Eclipse annotations, extension points

A.1. Standard annotations in Eclipse

The following table lists the default annotations of Eclipse and contains a link to the chapter which covers this annotation. This list is sorted by importance of the annotations, please see the index for an alphanumeric sorting.

Table A.1. Annotations for dependency injection

Annotation	Link
@javax.inject.Inject	See Section 36.1, “Define class dependencies in Eclipse” .
@javax.inject.Named	See Section 36.1, “Define class dependencies in Eclipse” .
@Optional	See Section 36.1, “Define class dependencies in Eclipse” .
@Execute	See Section 75.1, “Handler classes and their behavior annotations” .
@CanExecute	See Section 75.1, “Handler classes and their behavior annotations” .
@AboutToShow	See Section 78.3, “Dynamic menu and toolbar entries” .
@AboutToHide	See Section 78.3, “Dynamic menu and toolbar entries” .
@Persist	See Section 106.2, “MDirtyable and @Persist” .
@UIEventTopic	See the section called “Annotations for receiving events” .
@EventTopic	See the section called “Annotations for receiving events” .
@Preference	See Section 135.1, “Preferences and dependency injection” .
@PersistState	See Section 135.2, “Persistence of part state” .

@PostContextCreate	See Section 132.4, “How to implement a life cycle class” .
@PreSave	See Section 132.4, “How to implement a life cycle class” .
@ProcessAdditions	See Section 132.4, “How to implement a life cycle class” .
@ProcessRemovals	See Section 132.4, “How to implement a life cycle class” .
@Creatable	See Section 121.3, “Create your custom objects automatically with @Creatable” .
@Singleton	See Section 121.3, “Create your custom objects automatically with @Creatable” .
@GroupUpdates	See Section 36.1, “Define class dependencies in Eclipse” .
@Active	See Section 38.5, “Tracking a child context with @Active” .

A.2. Relevant extension points for Eclipse 4

The following table lists the relevant extension points for Eclipse 4 applications.

Table A.2. Extension Points relevant for Eclipse 4

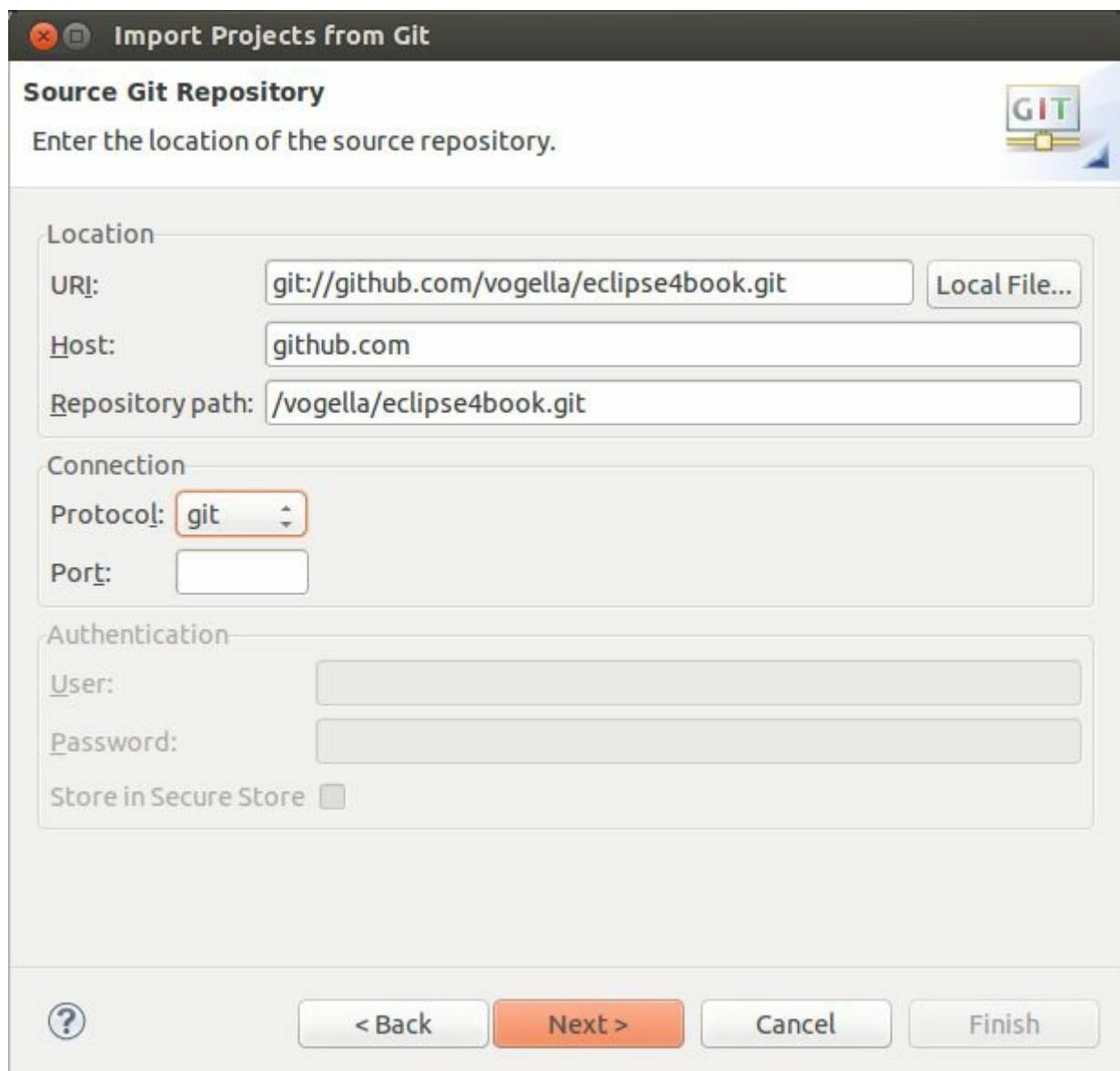
Name	Description
org.eclipse.core.runtime.products	Used to define the Eclipse application and the links to the Eclipse 4 configuration file.
org.eclipse.e4.workbench.model	Used to define contributions to the Eclipse application model.
org.eclipse.e4.ui.css.swt.theme	Used to define themes for CSS styling.

Appendix B. Solutions for the exercises

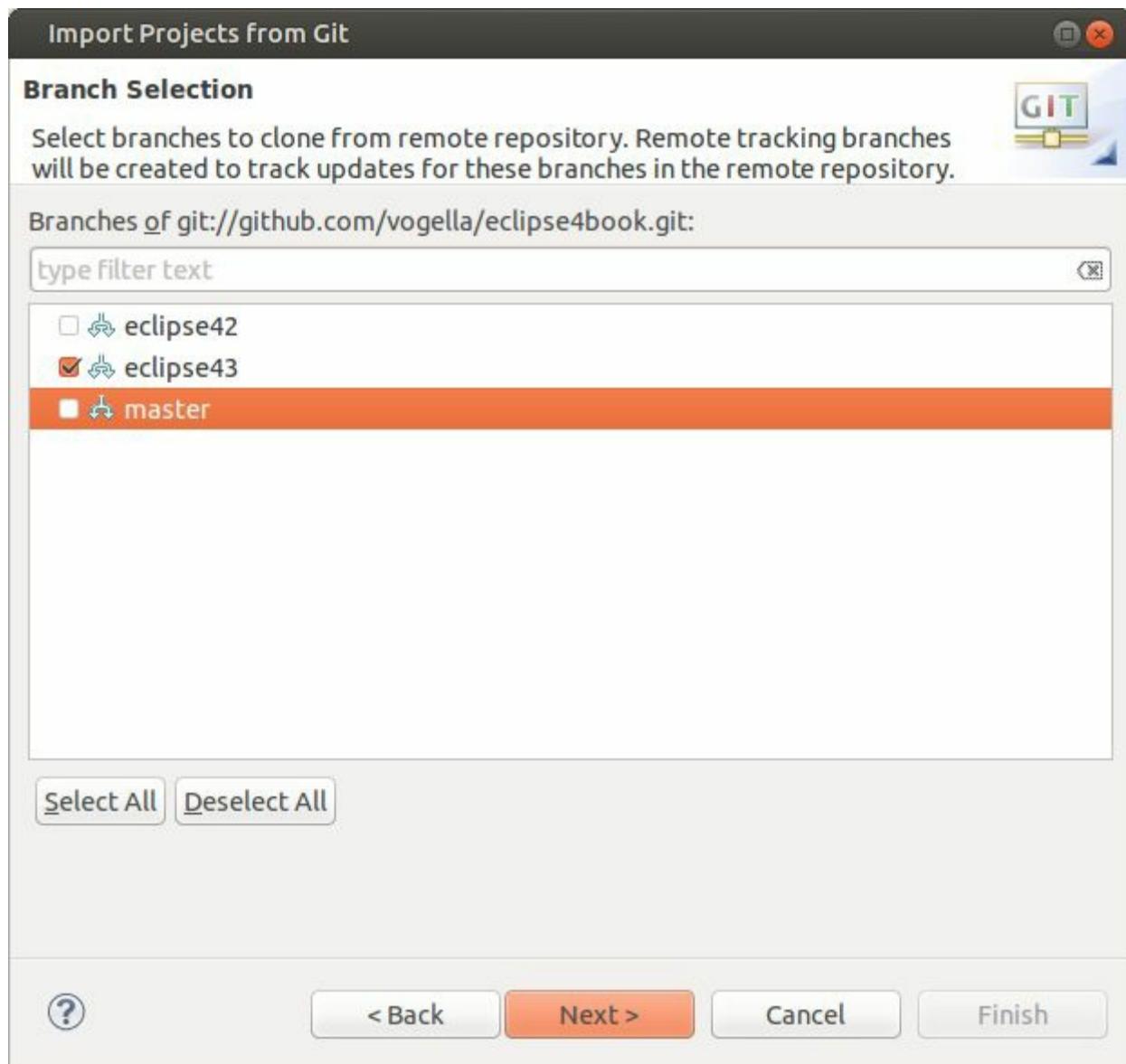
B.1. Getting the example implementation

The solutions for this version of the book are currently contained in the master branch and the eclipse43 branch. If a next edition of this book is published, the master branch will point to the latest solution but the eclipse43 branch will stay stable.

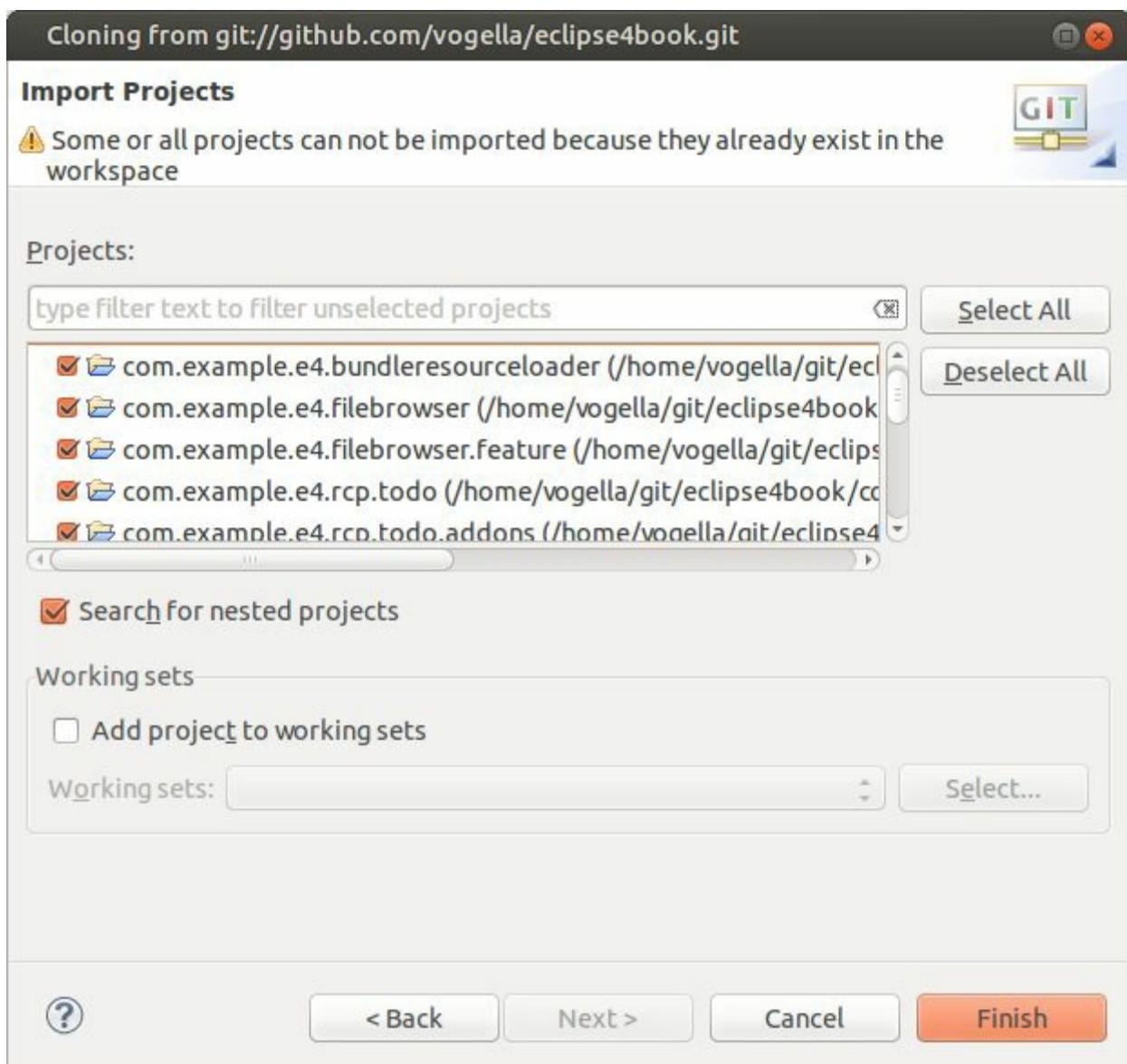
If you have EGit installed in your Eclipse, you can clone the repository via File → Import → Git → Projects from Git. Select *URI* in the next wizard page and press the *Next* button. In the following wizard page enter `git://github.com/vogella/eclipse4book.git` in the first line of the dialog.



Select the `eclipse43` branch for the solutions to the exercises of this book.



Continue the wizard. At some point, the wizard will prompt you to import the project from your local clone.



Tip

The `com.example.e4.rcp.todo.uitest` may show you an error, because you have not yet installed SWTBot. You can hide this error by closing this plug-in. For this right-click on it and select *Close Project*.

B.2. More information about Git and Eclipse

See [Eclipse Git tutorial](#) to learn more about the usage of the Eclipse Git tooling.

To learn about the concepts behind Git and its usage on the command line, see the *Distributed Version Control with Git: Mastering the Git command line* book (ISBN-10: 3943747069) from Lars Vogel.

Appendix C. Recipes

This chapter contains a few recipes for typical development tasks and some additional background material.

C.1. Eclipse update manager

The Eclipse IDE contains a software component called *Update Manager* which allows you to install and update software components. Installable software components are called `features` and consist of `plug-ins`.

These features are contained in so-called *update sites* or *software sites*. An *update site* contains installable software components and additional configuration files. It can be located in various places, e.g., on a web server or on the local filesystem.

The configuration files provide aggregated information about the software components in the *update site*. The update functionality in Eclipse uses this information to determine which software components are available in which version. This allows the Eclipse update functionality to download only components which are new or updated.

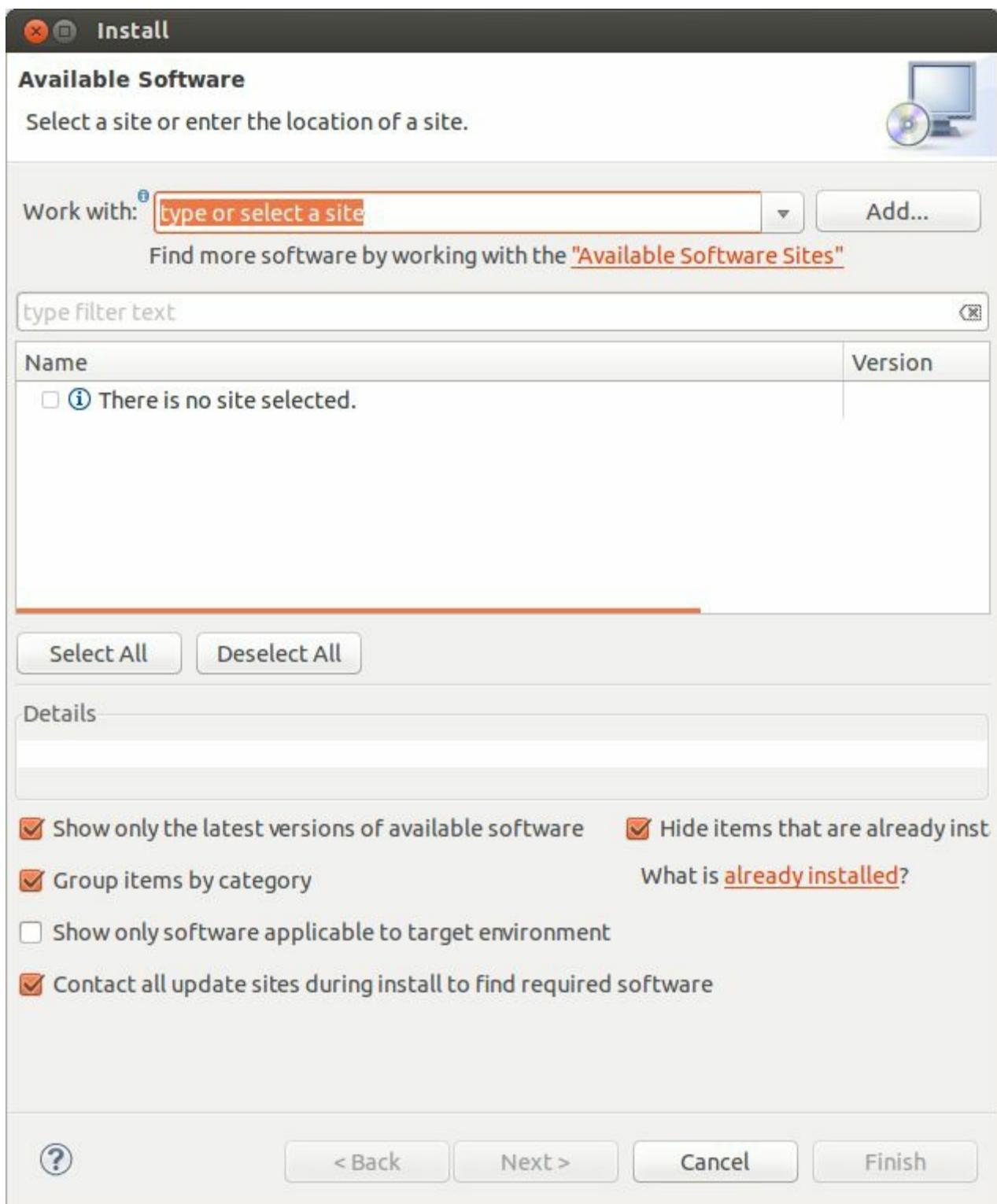
C.2. Performing an update and install new features

Warning

If you are behind a network proxy, you have to configure your proxy via the Window → Preferences → General → Network Connection preference setting. Otherwise, Eclipse may not able to reach the update sites.

To update your Eclipse installation, select Help → Check for Updates. The system searches for updates of the already installed software components. If it finds updated components, it will ask you to approve the update.

To install a new functionality, select Help → Install New Software....



From the *Work with* list, select or enter a URL from which you would like to install new software components. Entering a new URL adds this URL automatically to the list of available update sites.

To explicitly add a new update site, press the *Add...* button and enter the new URL as well as a name for the new update site.

The following update sites contain the official Eclipse components.

```
# Eclipse 4.4 (Luna release)
http://download.eclipse.org/releases/luna
```

```
# Eclipse 4.3 (Kepler release)
http://download.eclipse.org/releases/kepler
```

If you select a valid update site, Eclipse allows you to install the available components. Check the components which you want to install.

Install

Available Software

Check the items that you wish to install.

Work with: [Luna - http://download.eclipse.org/releases/luna](http://download.eclipse.org/releases/luna)

Find more software by working with the "[Available Software Sites](#)" preferences.

type filter text

Name	Version
► <input type="checkbox"/> Business Intelligence, Reporting and Charting	
► <input type="checkbox"/> Collaboration	
► <input type="checkbox"/> Database Development	
► <input type="checkbox"/> EclipseRT Target Platform Components	
► <input type="checkbox"/> General Purpose Tools	
► <input type="checkbox"/> Linux Tools	
► <input type="checkbox"/> Mobile and Device Development	
► <input type="checkbox"/> Modeling	
► <input type="checkbox"/> Programming Languages	

Details

Show only the latest versions of available software Hide items that are already installed

Group items by category [What is already installed?](#)

Show only software applicable to target environment

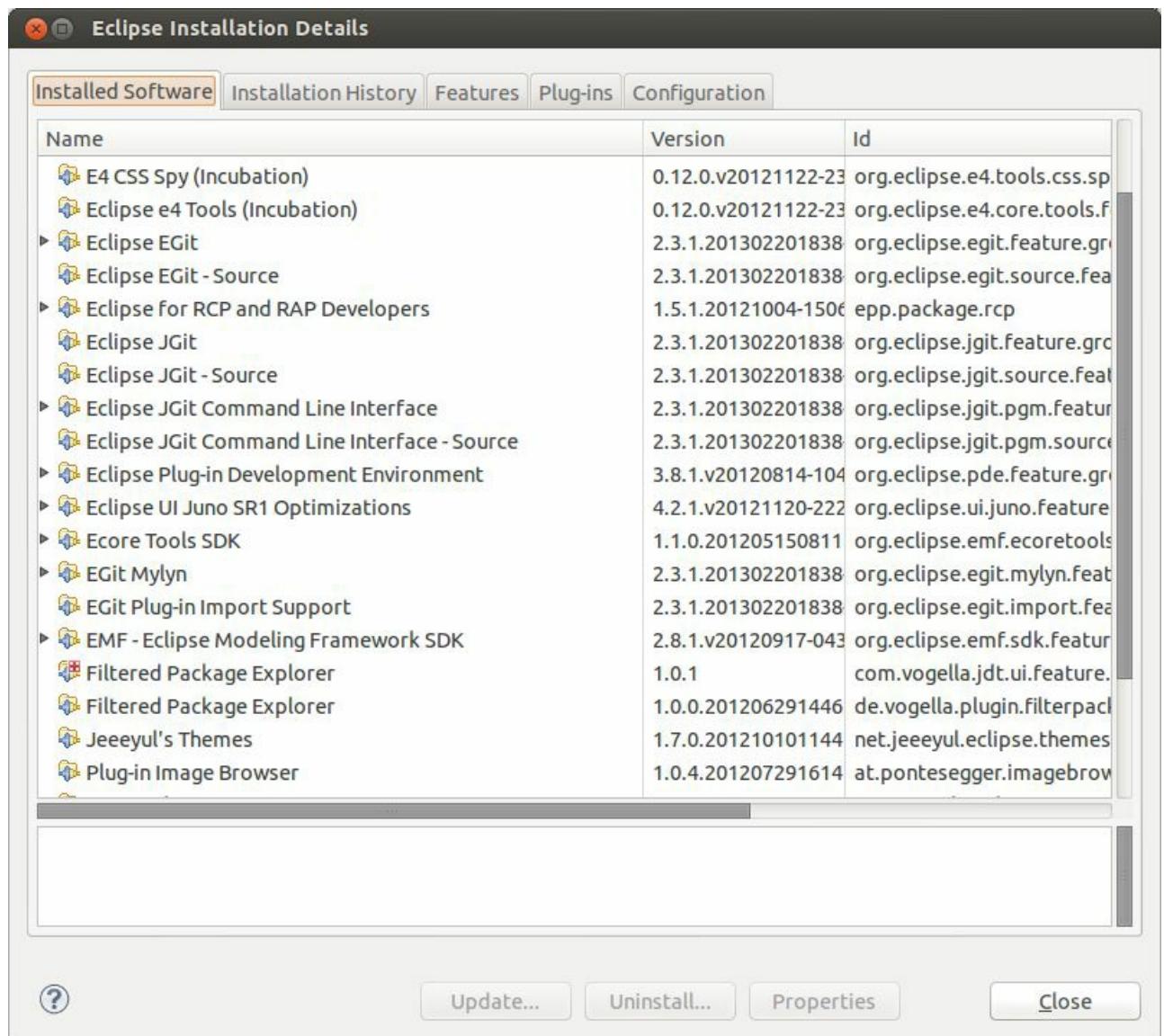
Contact all update sites during install to find required software

< Back Next > Cancel Finish

If you can't find a certain component, uncheck the *Group items by category* checkbox because not all available plug-ins are categorized. If they are not categorized, they will not be displayed, unless the grouping is disabled.

C.3. See the installed components

To see which components are installed, use Help → About Eclipse SDK → Installation Details.



C.4. Uninstalling components

If you select Help → About Eclipse SDK and then the *Installation Details* button, you can uninstall components from your Eclipse IDE.

C.5. Restarting Eclipse

After an update or an installation of a new software component, you should restart Eclipse to make sure that the changes are applied.

C.6. Reading resources from plug-ins

You can access files in your plug-in via two different approaches: via the `FileLocator` class or via an Eclipse specific URL. Both approaches require that your plug-in defines a dependency to the `org.eclipse.core.runtime` plug-in.

Both approaches return a URL which can be used to get an `InputStream`.

The following code shows an example with `FileLocator`.

```
Bundle bundle = FrameworkUtil.getBundle(this.getClass());
URL url = FileLocator.find(bundle,
    new Path("path_to_file"), null);
```

The following code shows an example for the usage of the Eclipse specific URL.

```
URL url = null;

try {
    url = new URL("platform:/plugin/"
        + "your_bundle-symbolicname"
        + "path_to_file");
} catch (MalformedURLException e1) {
    e1.printStackTrace();
}
```

Using the URL to create an `InputStream` and to read the file is demonstrated in the next code example.

```
public static String readTextFile(URL url) throws IOException {
    StringBuilder output = new StringBuilder();
    String lineSeparator = System.lineSeparator();

    try (InputStream in = url.openConnection().getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(in))) {

        String inputLine;

        while ((inputLine = br.readLine()) != null) {
            output.append(inputLine).append(lineSeparator);
        }

        if (output.length() > 0) {
            // remove last line separator
            output.setLength(output.length() - lineSeparator.length());
        }
    }
}
```

```
        }  
    }  
  
    return output.toString();  
}
```

C.7. Loading images from a plug-in

The `ImageDescriptor` class has a static method to create an `ImageDescriptor` instance directly from a URL. `ImageDescriptor` is a light-weight representation of an `Image`. Via the `createImage()` you can create the `Image` instance.

```
@Override
public Image loadImage(Class<Object> clazz, String path) {
    Bundle bundle = FrameworkUtil.getBundle(this.getClass());
    URL url = FileLocator.find(bundle,
        new Path("icons/alt_window_32.gif"), null);

    ImageDescriptor imageDescr = ImageDescriptor.createFromURL(url);
    return imageDescr.createImage();
}
```

C.8. Getting the command line arguments

The `IApplicationContext` gives access to the command line arguments which were used to start the Eclipse application, e.g., via the run configuration in the *Arguments* tab or via the `.ini` file of your exported product.

```
import java.util.Map;

import org.eclipse.e4.core.di.annotations.Execute;
import org.eclipse.equinox.app.IApplicationContext;

public class GetArgumentsHandler {
    @Execute
    public void execute(IApplicationContext context) {
        // OSGi consumes framework arguments
        String[] args = (String[]) context.
            getArguments().
            get(IApplicationContext.APPLICATION_ARGS);
        for (Object value : args) {
            System.out.println(value);
        }
    }
}
```

Appendix D. Architectural background of the application model

This section contains some background information about the design goals of the Eclipse application model and has been contributed by Eric Moffatt, Eclipse 4 SDK development lead.

D.1. Main areas of the model

Conceptually the model is structured into three areas which are described in the following table.

Table D.1. Architectural areas of the model

Area	Description
Mix-ins	Abstract interfaces containing the attributes necessary to support their specific roles in some derived concrete types. For example <code>MContext</code> contains only the reference to the context, <code>MContribution</code> contains the instantiated 'object' and <code>MUILabel</code> contains attributes to manage labels and images.
Containment	Every concrete container specifies a specific interface as its containment type. It allows the generic to be a specific type and the compiler to generate a compile time error if an attempt is made to add an invalid element to a container.
Types	It also means that when creating a domain specific model (or changing the base model) it is relatively easy to loosen the containment rules. For example to allow <code>MParts</code> in the <code>MTrim</code> area, you would only have to add the <code>MTrimElement</code> interface to <code>MPart</code> . Elements that the application model itself contains, defined by a combination of various mix-ins and containment types. These are further sub-divided based into packages in the model definition model.
Concrete Classes	

D.2. Advantages of using mix-ins

Mix-ins allow the Eclipse 4 internal code to manage the mix-ins specifically, regardless of the concrete class they're a part of. For example the `PartRenderingEngine` class handles all contexts by checking whether the element being rendered is an `MContext`. It also allows via the `findElements()` method of `EModelService` to locate all elements in the model of a specific type. This is useful for finding all `MDirtyable` elements for saving.

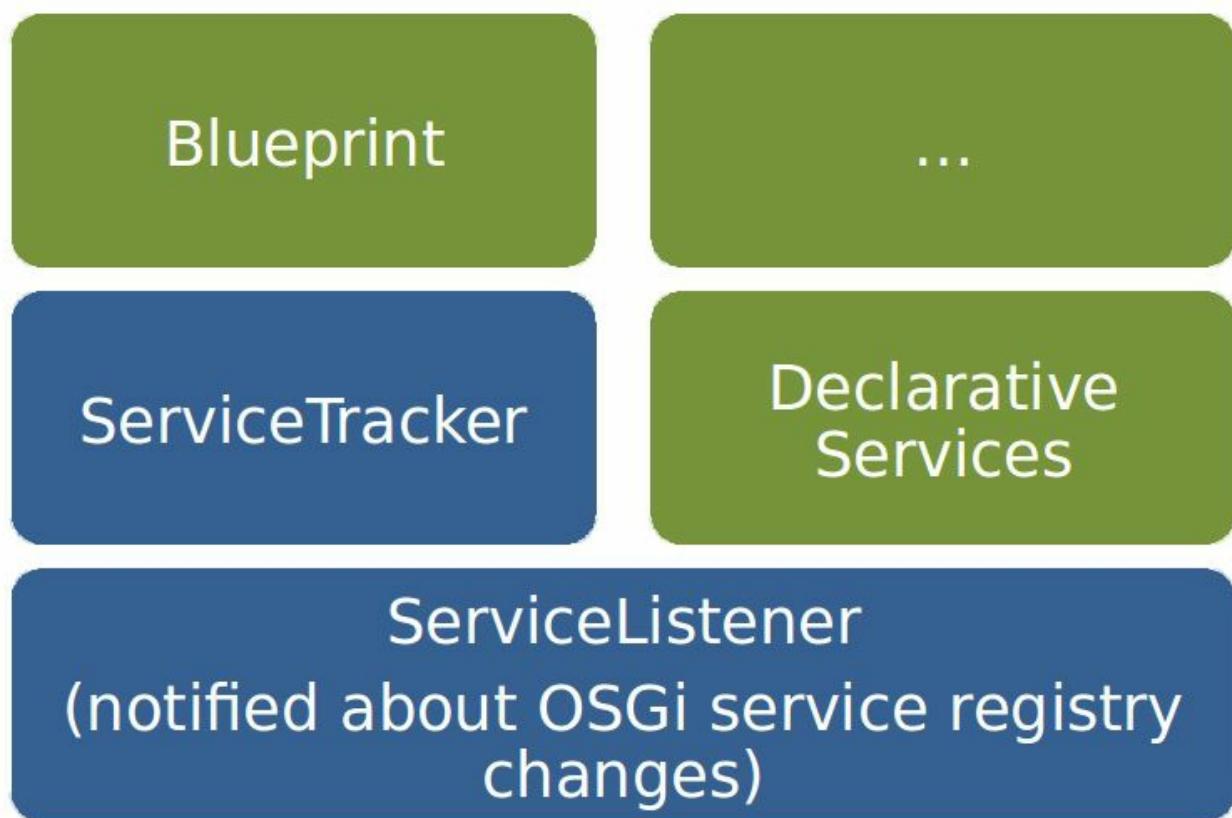
Mix-ins make domain specific modeling simpler. The Eclipse 4 team anticipate that more sophisticated clients create domain specific models (and the corresponding renderers) so that they could work directly against their own element types.

Having the mix-ins in place makes this much simpler as well as being able to reuse existing code written to manage a particular mix-in.

Appendix E. OSGi low level service API

E.1. Using the service API

OSGi provides several means of declaring services. This book focus on the OSGi declarative service functionality but it is also possible to use other means for defining services. These options are depicted in the following picture. Blueprint and Declarative Services provide high level abstractions for handling services.



This chapter describes the API to work directly with OSGi services but, if you have the option, you should prefer higher level abstractions as these simplify the handling of OSGi services.

E.2. BundleContext

Access to the service registry is performed via the `BundleContext` class.

A bundle can define a `BundleActivator` (Activator) in its declaration. This class must implement the `BundleActivator` interface.

If defined, OSGi injects the `BundleContext` into the `start()` and `stop()` methods of the implementing Activator class.

```
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

    public void start(BundleContext context) throws Exception {
        System.out.println("Starting bundle");
        // do something with the context, e.g.
        // register services
    }

    public void stop(BundleContext context) throws Exception {
        System.out.println("Stopping bundle");
        // do something with the context, e.g.
        // unregister service
    }
}
```

If you do not have an Activator, you can use the `FrameworkUtil` class from the OSGi framework which allows you to retrieve the `BundleContext` for a class.

```
BundleContext bundleContext =
    FrameworkUtil.
    getBundle(this.getClass()).
    getBundleContext();
```

E.3. Registering services via API

A bundle can also register itself for the events (`ServiceEvents`) of the `BundleContext`. These are, for example, triggered if a new bundle is installed or de-installed or if a new service is registered.

To publish a service in your bundle use:

```
public class Activator implements BundleActivator {  
    // ...  
    public void start(BundleContext context) throws Exception {  
        context.  
        registerService(IMyService.class.getName(),  
                      new ServiceImpl(), null);  
  
    }  
    // ...  
}
```

Once the service is no longer used, you must unregister the service with OSGi. OSGi counts the usage of services to enable the dynamic replacement of services. So once the service is no longer used by your implementation, you should tell the OSGi environment this by:

```
context.ungetService(serviceReference);
```

In the `registerService()` method from the `BundleContext` class you can specify arbitrary properties in the dictionary parameter.

You can use the `getProperty()` method of the `ServiceReference` class from the `org.osgi.framework` package, to access a specific property.

E.4. Accessing a service via API

A bundle can acquire a service via the `BundleContext` class. The following example demonstrates that.

```
ServiceReference<?> serviceReference = context.  
    getServiceReference(IMyService.class.getName());  
IMyService service = (IMyService) context.  
    getService(serviceReference);
```

E.5. Low-level API vs OSGi declarative services

OSGi services can be dynamically started and stopped. If you work with the OSGI low-level API you have to handle this dynamic in your code. This make the source code unnecessary complex. If you do not handle that correctly your service consumer can keep a reference to the service and the service cannot be removed via the OSGi framework.

To handle the dynamics automatically declarative services were developed. Prefer therefore the usage of OSGi *declarative services* over the low-level API.

Appendix F. Links and web resources

F.1. Eclipse RCP resources

Eclipse Bug Tracker and Eclipse forum

[Eclipse Forum](#) for asking questions and providing feedback.

[Eclipse Bug Tracker](#) for reporting errors or feature requests.

Eclipse RCP development resources

[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](#)

F.2. Links and Literature

Source Code

[Source Code of Examples](#)

Eclipse plug-in development resources

http://wiki.eclipse.org/Eclipse_Plug-in_Development_FAQ Eclipse Plug-in Development FAQ

[Wiki with link to Eclipse Articles, Tutorials, Demos, Books](#)

[Eclipse tutorials hosted at Eclipse.org.](#)

Special Eclipse plug-in development topics

<http://www.eclipse.org/articles/article.php?file=Article-Adapters/index.html>
Adapters in Eclipse

<http://www.eclipse.org/articles/Article-Resource-deltas/resource-deltas.html>
How to react to Eclipse resource deltas

[Using markers, annotations, and decorators in Eclipse \(IBM\)](#).

F.3. Eclipse product and export resources

[How to create ico files automatically](#)

F.4. OSGi Resources

[OSGi homepage](#)

[Equinox homepage](#)

[Equinox quick start guide](#)

[OSGi blueprint services](#)

[hawtio dependency visualization tool](#)

F.5. OSGi Resources

<http://www.osgi.org> OSGi Homepage

<http://www.eclipse.org/equinox> Equinox Homepage

[OSGi remote RESTful service tutorial](#) Equinox Homepage

[OSGi remove service tutorial with ECF](#) Equinox Homepage

<http://www.eclipse.org/equinox> Equinox Homepage

<http://www.eclipse.org/equinox/documents/quickstart.php> Equinox Quickstart guide

<http://www.ibm.com/developerworksopensource/library/os-osgiblueprint/>
OSGi Blueprint services

F.6. Eclipse SWT resources

[SWT Widgets](#)

[SWT Snippets \(Examples\)](#)

[Understanding Layout in SWT by Wayne Beaton](#)

[SWT design](#)

[Eclipse Nebula Project with additional SWT widgets](#)

[SWT Graphics](#)

[SWT and Java8](#)

[YARI tool suite to debug, spy, spider, inspect and navigate Eclipse based application GUIs \(Workbench or RCP\).](#)

[SWT Development Tools, include Sleak \(analyze memory leaks in SWT graphic resources\) and SWT Spy plug-in which prints out the info about the SWT widget under the cursor.](#)

F.7. JFace resources

<http://wiki.eclipse.org/index.php/JFaceSnippets> JFace snippets, e.g., small code examples

[NatTable - Flexible Table / Data Grid](#)

F.8. Eclipse Data Binding resources

[Wiki about the JFace Data Binding](#)

[WindowBuilder Data Binding Example](#)

[Using Bean Validation \(JSR 303\) with JFace Data Binding](#)

F.9. Eclipse Jobs resources

[Eclipse Jobs API tutorial](#)

F.10. Eclipse i18n resources

[Eclipse 4 internationalization by Dirk Fauth - Part 1](#)

[Eclipse 4 internationalization by Dirk Fauth - Part 2](#)

[Eclipse 4 internationalization by Dirk Fauth - Part 3](#)

[Eclipse 4 internationalization by Dirk Fauth - Part 4](#)

[Internationalization with the WindowBuilder](#)

[How to Internationalize your Eclipse Plug-In](#)

[Bug concerning overriding SWT and JFace resource bundles](#)

[Git location of the JFace message.properties](#)

[Git location of the SWT message.properties](#)

[Default SWT Language Packs](#)

F.11. CSS styling resources

[Eclipse Wiki for CSS attributes](#)

F.12. Eclipse p2 updater resources

[The Eclipse p2 wiki](#)

[Custom touch points in p2](#)

[Using p2 touch points to update the eclipse.ini](#)

[Defining a proxy for p2 headless operations](#)

[Installing Eclipse Plug-ins from an Update Site with a self-signed certificate](#)

[Adding p2 update to an RCP application](#)

F.13. Logging

[Blog series about Logging](#)

[EclipseZone Discussion about Logging](#)

Index

Symbols

@AboutToHide, [Dynamic menu and toolbar entries](#)
@AboutToShow, [Dynamic menu and toolbar entries](#)
@Active, [Tracking a child context with @Active](#)
@CanExecute, [Handler classes and their behavior annotations](#)
@Creatable, [Create your custom objects automatically with @Creatable](#)
@EventTopic, [Annotations for receiving events](#)
@Execute, [API definition via annotations](#), [Handler classes and their behavior annotations](#)
@GroupUpdates, [Annotations to define class dependencies in Eclipse](#)
@Inject, [Using annotations to describe class dependencies](#), [Annotations to define class dependencies in Eclipse](#)
@Named, [Using annotations to describe class dependencies](#), [Annotations to define class dependencies in Eclipse](#)
@Optional, [Annotations to define class dependencies in Eclipse](#)
@Persist, [API definition via annotations](#)
@Persists, [MDirtyable and @Persist](#)
@PersistState, [API definition via annotations](#), [Persistence of part state](#)
@PostConstruct method not called, [Why is the @PostConstruct method not called?](#)
@PostContextCreate, [How to implement a life cycle class](#)
@PostContract, [API definition via annotations](#)
@PreDestroy, [API definition via annotations](#)
@Preference, [Preferences and dependency injection](#)
@PreSave, [How to implement a life cycle class](#)
@ProcessAdditions, [How to implement a life cycle class](#)
@ProcessRemovals, [How to implement a life cycle class](#)
@Singleton, [Create automatically objects in the application context with @Singleton](#)
@UIEventTopic, [Annotations for receiving events](#)

A

AbsoluteLayout, [The role of a layout manager](#)
Add-ons, [What are model add-ons?](#)
Annotations, [Using annotations to describe class dependencies](#)
Behavior, [API definition via annotations](#)

Custom annotations processors, [Custom annotations](#)

Apache Felix, [OSGi implementations](#) API, [API definition](#) Application, [The Product configuration file and the application](#)

Icons, [Icons, launcher name and program arguments](#)

Launcher name, [Icons, launcher name and program arguments](#)

program arguments, [Icons, launcher name and program arguments](#)

Application life cycle hooks, [Connecting to the Eclipse application life cycle](#) Application model, [What is the application model?](#)

Add-ons, [Model add-ons](#)

Custom persistence handler, [Custom application model persistence handler](#)

Fragments, [Model fragments](#)

Modularity, [Modularity support in Eclipse RCP](#)

Processors, [Model processors](#)

Runtime modification, [Creating model elements](#)

Scope, [Scope of the application model](#)

Application Programming Interface, [API definition](#) Application updates, [Eclipse application updates](#) Application.e4xmi, [How do you define the application model?](#) applicationCSS, [Fixed styling in Eclipse](#)

B

Behavior annotations

Handler, [Handler classes and their behavior annotations](#)

Parts, [API definition via annotations](#)

Binding context, [Using key bindings in your application](#) Binding table, [Using key bindings in your application](#) BREE, [The manifest file](#)

([MANIFEST.MF](#)) build.properties, [Defining which artifacts are included in the export](#) Bundle, [What is OSGi?](#), [Plug-in or bundles as software component](#) Bundle

vs. plug-in, [Plug-in or bundles as software component](#) Bundle-ActivationPolicy, [The manifest file \(MANIFEST.MF\)](#) Bundle-Activator, [The manifest file](#)

([MANIFEST.MF](#)) Bundle-ClassPath, [The manifest file \(MANIFEST.MF\)](#) Bundle-Localization, [OSGi resource bundles](#) Bundle-Name, [The manifest file](#)

([MANIFEST.MF](#)) Bundle-RequiredExecutionEnvironment, [The manifest file](#)

([MANIFEST.MF](#)) Bundle-SymbolicName, [Bundle-SymbolicName and](#)

Version Bundle-Version, [The manifest file \(MANIFEST.MF\)](#) bundleclass, [URI](#)

[patterns or classes and resources](#)

C

Cascading Style Sheets (see CSS)

CIF, [Using dependency injection to create objects](#)

Class URI, [Connect model elements to classes](#)

bundleclass://, [URI patterns or classes and resources](#)

platform:/plugin/, [URI patterns or classes and resources](#)

clearPersistedState, [Create project](#)ComboViewer (JFace), [JFace](#)

ComboViewerCommand, [What are commands and handlers?](#)Command service,

[Purpose of the command and handler service](#)Commands

Parameters, [Passing parameters to commands](#)

Compatibility layer, [Compatibility layer for Eclipse 3.x plug-ins](#)Compatibility

mode, [Using the compatibility mode](#)Composite, [Basic containers](#)Concurrency,

[What is concurrency?](#)

Job, [Eclipse Jobs API](#)

Configuration files of plug-ins, [Important configuration files for Eclipse plug-ins](#)

Constructor injection, [Where can objects be injected into a class?](#)Context

function, [What are context functions?](#)Context variables, [Objects and context](#)

[variables](#)ContextInjectionFactory, [Using dependency injection to create](#)

[objects](#)ControlDecoration, [ControlDecoration](#)Core expressions, [Usage of core](#)

[expressions](#)Create sample content (parts, menu etc.), [Create project](#)CSS, [What is](#)

[CSS?](#)

Eclipse , [Styling Eclipse applications](#)

Import statement , [CSS imports](#)

Pseudo classes, [CSS pseudo classes](#)

Selector, [CSS selectors and style rules](#)

Styling of Eclipse applications, [Fixed styling in Eclipse](#)

Theming of Eclipse applications, [Dynamic styling using themes](#)

cssTheme, [Dynamic styling using themes](#)

D

Data Binding, [What are Data Binding frameworks?](#)
(see also JFace Data Binding)

DataBindingContext, [Connecting properties with the DataBindingContext](#) Declarative services, [Defining declarative services](#) Delta pack, [Export for multiple platforms via the delta pack](#) Dependency container, [What is dependency injection?](#) Dependency injection

annotations, [Using annotations to describe class dependencies](#)
Concept, [What is dependency injection?](#)

ContextInjectionFactory, [Using dependency injection to create objects](#)
Create objects with the Eclipse DI framework, [Using dependency injection to create objects](#)

Order, [Order in which dependency injection is performed on a class](#)

Deployment, [Creating a stand-alone version of the application](#)

Common problems, [Export problem number #1: export folder is not empty](#)
Headless build, [Headless build](#)

JRE, [Including the required JRE into the export](#)

Multiple platforms, [Export for multiple platforms via the delta pack](#)

Dialog, [Dialogs in Eclipse](#)

JFace, [JFace Dialogs](#)

SWT, [SWT dialogs](#)

Direct MenuItem, [Adding menu and toolbar entries](#) Direct ToolItem, [Adding menu and toolbar entries](#) Display (SWT), [Display and Shell](#) Download

Eclipse SDK, [Download the Eclipse Software Development Kit \(SDK\)](#)

DynamicMenuContribution

Example , [Exercise: Creating dynamic menu entries](#)

E

e4 API , [The e4 API and e4 runtime terminology](#)

e4 project, [The Eclipse e4 project](#)

e4 runtime , [The e4 API and e4 runtime terminology](#)

e4 tooling, [The e4 tools project](#)

Eclipse 3.x and 4.x plug-ins, [Using 3.x components in e4 API based applications](#)
Eclipse 3.x API support, [Using the Eclipse 3.x API on top of an 4.x runtime](#)
Eclipse Architecture, [Architecture of Eclipse based applications](#)
Eclipse context, [What is the Eclipse context?](#)
Hierarchy, [Relationship definition in the Eclipse context](#)
Life cycle, [Life cycle of the Eclipse context](#)
Modify, [Accessing the context](#)

Eclipse core components, [Core components of the Eclipse platform](#)
Eclipse Foundation, [Eclipse foundation](#)
Eclipse high level design, [Architecture of Eclipse based applications](#)
Eclipse IDE, [About the Eclipse IDE](#)
Eclipse open source project, [The Eclipse community and projects](#)
Eclipse Public License, [Source code and the Eclipse Public License](#)
Eclipse RCP, [What is the Eclipse Rich Client Platform \(Eclipse RCP\)?](#)
Eclipse releases, [Eclipse releases](#)
Eclipse services, [What are Eclipse platform services?](#)
Eclipse styling, [Styling Eclipse applications](#)

(see also CSS)

ECommandService, [Access to command and handler service](#)
EContextService, [Activate bindings](#)
Editor, [Parts behaving as views or as editors](#)

Implementation, [Parts which behave similar to editors](#)
Multiple instances, [MPart and multiple editors](#)
Save, [Use part service to trigger save in editors](#)

EHandlerService, [Access to command and handler service](#)
EMF Parsley, [Additional UI toolkits](#)
EModelService, [What is the model service?](#)
Enable development mode for application model, [Create project](#)
EPartService, [What is the part service?](#)
EPL, [Source code and the Eclipse Public License](#)
Equinox, [OSGi implementations](#)
ErrorDialog, [ErrorDialog](#)
ESelectionService, [Usage of the selection service](#)
Event listener (SWT), [Event Listener](#)
Event service, [Event service](#)

Annotations, [Annotations for receiving events](#)
Asynchronous processing, [Asynchronous processing and the event bus](#)
Framework events, [Eclipse framework events](#)
Listener, [Registering listeners for events](#)
Receiving events via annotations, [Annotations for receiving events](#)
Sending events, [Sending](#)
Sub-topics, [Subscribing to sub-topics](#)

Events, [Event based communication](#)Events bus, [Event based communication](#)Example code, [Getting the example implementation](#)Export (see Deployment)ExtendedObjectSupplier, [Define an annotation processor via an OSGi service](#)Extension points and extensions, [Extensions and extension points](#)

adding extensions , [Adding extensions to extension points](#)

F

Feature projects, [What are feature projects and features?](#)

Features, [What are feature projects and features?](#)

Field assistance (JFace), [User input help with field assistance](#)

Field injection, [Where can objects be injected into a class?](#)

FillLayout, [The role of a layout manager](#)

Find the plug-in for a certain class, [Find the plug-in for a certain class](#)

Finding missing plug-in dependencies during a product launch

, [Finding missing plug-in dependencies during a product launch](#)

FormLayout, [The role of a layout manager](#)Fragment projects

Tests, [Fragment projects](#)

Fragments, [What are fragments in OSGi?](#)

G

Git, [More information about Git and Eclipse](#)

GridLayout, [The role of a layout manager](#)

Group (SWT), [Basic containers](#)

H

Handled MenuItem, [Adding menu and toolbar entries](#)

Handled ToolItem, [Adding menu and toolbar entries](#)

Handler, [What are commands and handlers?](#)

Handler service, [Purpose of the command and handler service](#)

Headless build, [Headless build](#)

History, [What is the Eclipse Rich Client Platform \(Eclipse RCP\)?](#)

I

i18n, [Translation of a Java applications](#)

Application model, [Translating the application model](#)

Export, [Exporting Plug-ins and Products](#)

OSGi resource bundles, [OSGi resource bundles](#)

Problems, [Common problems with i18n](#)

Property files, [Property files](#)

Setting the language via launch parameter, [Setting the language in the launch configuration](#)

i22n

plugin.xml, [Translating plugin.xml](#)

IApplicationContext, [Accessing application startup parameters](#) IContextFunction, [Creation of a context function](#) IEclipseContext (see Eclipse context) IEclipsePreference, [Preferences and dependency injection](#)

(see also Preferences)

IEventBroker, [Event service](#)

(see also Event service)

IExtensionRegistry, [Accessing extensions](#) Include all plug-ins in Java search, [Include all plug-ins in Java search](#) Installation, [Install Eclipse IDE for RCP development](#)

e4 tools, [Install the e4 tools from the vogella GmbH update site](#)

Internal API, [Eclipse API and internal API](#) Internationalization (see i18n) IProgressMonitor, [IProgressMonitor](#) IRendererFactory, [Renderer factory and renderer objects](#) ISaveHandler, [Replacing existing objects in the IEclipseContext](#) IServiceConstants

ACTIVE_PART, [Qualifiers for accessing the active part or shell](#)

ACTIVE_SHELL, [Qualifiers for accessing the active part or shell](#)

IThemeEngine, [Dynamic styling using themes](#)

(see also CSS)

IWindowCloseHandler, [Replacing existing objects in the IEclipseContext](#)

J

JAR files, [What is a JAR file?](#)

Java Bean, [Property change support](#)

Java version requirements to run Eclipse, [Java requirements of the Eclipse IDE](#)

JavaFX, [JavaFX renderer - e\(fx\)clipse](#)

JFace, [What is Eclipse JFace?](#)

ComboViewer, [JFace ComboViewer](#)

ControlDecoration, [ControlDecoration](#)

Field assistance, [User input help with field assistance](#)

Resource manager, [JFace resource manager for Colors, Fonts and Images](#)

TableViewer, [Using the JFace TableViewer](#)

TreeViewer, [Using viewers to display a tree](#)

Viewer, [Purpose of the JFace viewer framework](#)

JFace Data Binding, [JFace Data Binding](#)

ControlDecorators, [ControlDecorators](#)

Converter, [Converter](#)

Listener, [Listening to all changes in the binding](#)

Validator, [Validator](#)

Viewer binding, [Binding Viewers](#)

WritableValue, [Placeholder binding with WritableValue](#)

jo-widget, [Additional UI toolkits](#)Job (Concurrency), [Eclipse Jobs API](#)JSR 330,

[Using annotations to describe class dependencies](#)JUnit

@Test, [How to define a test in JUnit?](#)

Annotations, [Available JUnit annotations](#)

Assert statements, [Assert statements](#)

Creating tests in Eclipse, [Creating JUnit tests](#)

Example test method, [Example JUnit test](#)

Exceptions, [Testing exception](#)

Static imports , [Setting Eclipse up for using JUnits static imports](#)

Static imports in Eclipse, [JUnit static imports](#)

Test execution order, [Test execution order](#)

Test suite, [JUnit test suites](#)

JUnit framework, [The JUnit framework](#)JUnit Plug-in Test, [JUnit Plug-in Test](#)

K

Key bindings, [Using key bindings in your application](#)
Activate, [Activate bindings](#)
Part, [Key bindings for a part](#)

Knopflerfish OSGi, [OSGi implementations](#)

L

Lars Vogel - Author , [About the author - Lars Vogel](#)
Launch configuration
Problems, [Finding missing plug-in dependencies during a product launch](#)

Launch parameters

Access, [Accessing application startup parameters](#)

Layout manager, [The role of a layout manager](#)Life cycle hooks, [Connecting to the Eclipse application life cycle](#)lifeCycleURI, [How to implement a life cycle class](#)Live model editor (see Model spy)Localization, [Translation of a Java applications](#)Log file

Exported RCP application, [Problems with the export and log files](#)

M

MAaddon, [Model objects](#)
MANIFEST.MF, [Important configuration files for Eclipse plug-ins](#), [The manifest file \(MANIFEST.MF\)](#)
MApplication, [Model objects](#)
MContext, [Accessing the context](#)
MDirtyable, [Model objects](#), [MDirtyable and @Persist](#)
Menu, [Adding menu and toolbar entries](#)
Dynamic, [Dynamic menu and toolbar entries](#)
Popup, [Popup menu \(context menu\)](#)
View / Part, [View menus](#)

MessageDialog, [MessageDialog](#)Method injection, [Where can objects be injected into a class?](#)Migration

Eclipse 3.x RCP application , [Using e4 API based components in a 3.x applications](#)

Migration of Eclipse 3.x RCP applications, [Technical reasons for migrating to the 4.x API](#)
[MInputPart](#)Mnemonics, [Mnemonics](#)Model add-ons,
[Model add-ons](#)

CleanupAddon, [Additional SWT add-ons](#)

Platform add-ons, [Add-ons from the Eclipse framework](#)

Model Add-ons, [What are model add-ons?](#)Model addons

DnDAddon, [Additional SWT add-ons](#)

MinMaxAddon, [Additional SWT add-ons](#)

Model fragments, [Model fragments](#)Model objects, [Model objects](#)Model objects with context, [Which model elements have a local context?](#)Model processor, [Model processors](#)Model service, [What is the model service?](#)Model spy, [Analyzing the application model with the model spy](#)MPart, [Model objects](#)MPartDescriptor, [Model objects](#)MPerspective, [Model objects](#)MTrimmedWindow, [Model objects](#)MWindow, [Model objects](#)

N

Naming conventions

Command IDs, [Naming schema for command and handler IDs](#)

Handler IDs, [Naming schema for command and handler IDs](#)

ID for model elements, [Naming conventions for model identifiers \(IDs\)](#)

Packages and classes, [Project, package and class names](#)

Projects, [Project, package and class names](#)

Naming conventions for tests, [JUnit naming conventions](#)Nebula widgets, [The Nebula and Opal widgets](#)

O

Observable (Data Binding), [Observing properties with the IObservableValue interface](#)

Online forums for Eclipse, [Asking \(and answering\) questions](#)

Opal widgets, [The Nebula and Opal widgets](#)

OSGi, [What is OSGi?](#)

Declarative services, [Defining declarative services](#)

Dependencies, [Specifying plug-in dependencies via the manifest file](#)

Life cycle, [Life cycle of plug-ins in OSGi](#)

provisional API, [Provisional API](#)
Services, [What are OSGi services?](#)

OSGi console, [The OSGi console](#)

P

p2 updates, [Eclipse application updates](#)

Part, [Parts](#)

Part descriptor, [Using part descriptors](#)

Part sash container

Layout weight, [Using layout weight data for children elements](#)

Part service, [What is the part service?](#)

Trigger save in dirty editors, [Use part service to trigger save in editors](#)

PartDescriptor (see Part descriptor)Parts

Creating at runtime, [What is the part service?](#)

PartSashContainer, [Part stack and part sash container](#)PartStack, [Part stack and part sash container](#)PDE, [The Eclipse Plug-in Development Environment \(PDE\)](#)Persisted state, [Persisted state](#)Perspective, [Perspective](#)

Switching with the part service, [What is the part service?](#)

Platform project, [The Eclipse Platform project](#)Plug-in, [Eclipse software components - Plug-ins](#)Plug-in development, [Eclipse software components - Plugins](#)Plug-in Development Environment, [The Eclipse Plug-in Development Environment \(PDE\)](#)Plug-in Test, [JUnit Plug-in Test](#)Plug-in vs. bundle, [Plug-in or bundles as software component](#)plugin.xml, [Important configuration files for Eclipse plug-ins](#)plugin_customization.ini, [Setting preferences via plugin_customization.ini](#)POJO, [Property change support](#)Popup menu, [Popup menu \(context menu\)](#)Preferences, [Preferences and scopes](#)

BundleDefaultsScope, [Preferences and scopes](#)

Configuration scope, [Preferences and scopes](#)

Default scope, [Preferences and scopes](#)

Default values, [Setting preferences via plugin_customization.ini](#)

Dependency injection, [Preferences and dependency injection](#)

Instance scope, [Preferences and scopes](#)

Process, [Process vs. threads](#) Product, [The Product configuration file and the application](#) Product Configuration

Start Level , [Configure the start levels](#)

Product configuration file, [The Product configuration file and the application](#)

Based on features, [Feature or plug-in based products](#)

Based on plug-ins, [Feature or plug-in based products](#)

Limitations, [Product configuration limitations](#)

Start level, [Using declarative services in RCP applications](#)

Progress reporting, [IProgressMonitor](#) Property change listeners, [Ability to listen to changes in the domain model](#) Provisional API (see Internal API)

R

RAP, [Eclipse RAP](#)

Renderer, [Renderer](#)

Custom implementation, [Using a custom renderer](#)

JavaFX, [JavaFX renderer - e\(fx\)clipse](#)

RAP, [Eclipse RAP](#)

Vaadin, [Vaadin renderer - Vaaeclipse](#)

rendererFactoryUri, [Using a custom renderer](#) Resource manager (JFace), [JFace resource manager for Colors, Fonts and Images](#) RowLayout, [The role of a layout manager](#) Run arguments (see Run configuration arguments) Run configuration, [What are run configurations?](#) Run configuration arguments, [Run arguments](#)

clearPersistedState, [Run arguments](#)

console, [Run arguments](#)

consoleLog, [Run arguments](#)

nl, [Run arguments](#)

noExit, [Run arguments](#)

RunAndTrack, [RunAndTrack](#) Runtime Eclipse IDE , [Starting the Eclipse IDE from Eclipse](#)

S

Sapphire, [Additional UI toolkits](#)
Selection services, [Usage of the selection service](#)
Services (OSGi), [What are OSGi services?](#)
(see also Eclipse services)

Shell (SWT), [Display and Shell](#)Simple plug-in, [Naming convention: simple plug-in](#)Snippets, [Model objects](#), [What is the model service?](#)Solutions for the exercises, [Getting the example implementation](#)Splash screen, [Splash screen](#)

Close, [Close static splash screen](#)

Start level in a product , [Configure the start levels](#)Starting the IDE from the Eclipse IDE , [Starting the Eclipse IDE from Eclipse](#)Static imports in Eclipse , [Setting Eclipse up for using JUnits static imports](#)Supplementary data, [Adding additional information on the model elements](#)SWT, [What is SWT?](#)

Composite (SWT), [Basic containers](#)
Display , [Display and Shell](#)
Event listener, [Event Listener](#)
Event loop , [Event loop](#)
Group, [Basic containers](#)
Layout manager, [The role of a layout manager](#)
Memory management, [Memory management](#)
Shell , [Display and Shell](#)
Widgets, [Available widgets in the SWT library](#)

SWT Designer, [SWT Designer \(WindowBuilder\)](#)SWT Examples, [SWT snippets and examples](#)SWT snippets, [SWT snippets and examples](#)SWTBot, [User interface testing with SWTBot](#)SWTEception - Invalid thread access, [Main thread](#)

T

TableViewer (JFace), [Using the JFace TableViewer](#)
Tags, [Tags](#)
Target definition file, [Target platform definition](#), [Defining a target platform](#)
Target platform, [Defining available plug-ins for development](#)
Definition, [Defining a target platform](#)

Test annotations from JUnit, [Available JUnit annotations](#)Testing

SWTBot, [User interface testing with SWTBot](#)

User interface components, [Testing user interface components](#)

User interface testing, [User interface testing with SWTBot](#)

Using dependency injection in your tests, [Testing dependency injection](#)

Theme service, [Dynamic styling using themes](#)

(see also CSS)

Threads, [Process vs. threads](#) Toolbar, [Adding menu and toolbar entries](#)

Drop-down, [Drop-down tool items](#)

View / Part, [Adding toolbars to parts](#)

ToolControl, [ToolControls](#) Transient data, [Transient data](#) Translation (see i18n) Translation service, [Translation with POJOs](#) TreeViewer (JFace), [Using viewers to display a tree](#)

U

UISynchronize, [Using dependency injection and UISynchronize](#)

Update manager

install new features , [Performing an update and install new features](#)

performing an update , [Performing an update and install new features](#)

see installed components , [See the installed components](#)

uninstall components , [Uninstalling components](#)

Update manager of the Eclipse IDE, [Eclipse update manager](#) Updates, [Eclipse application updates](#) UpdateValueStrategy, [UpdateValueStrategy](#)

V

Vaadin, [Vaadin renderer - Vaaeclipse](#)

View, [Parts behaving as views or as editors](#)

View menu, [View menus](#)

Viewer (JFace), [Purpose of the JFace viewer framework](#)

vogella GmbH, [About the vogella company](#)

W

Wazzabi, [Additional UI toolkits](#)

Web resources, [Links and web resources](#)
Widgets (SWT), [Available widgets in the SWT library](#)
Window, [Window](#)
WindowBuilder, [SWT Designer \(WindowBuilder\)](#)
Wizard (JFace), [What is a wizards](#)
WorkbenchRendererFactory, [Renderer](#)

X

x-friends, [Provisional API](#)
x-internal, [Provisional API](#)
XWT, [Additional UI toolkits](#)

目录

Foreword	38
Preface	40
1. Welcome	40
2. About the author - Lars Vogel	41
3. About the vogella company	43
4. Screenshots	44
5. Shortcuts on a Mac	45
6. How this book is organized	46
7. Prerequisites	47
8. Errata	48
9. Exercises, optional exercises and examples	49
10. Long lines	50
11. Example code	51
12. Acknowledgment	52
I. The Eclipse open source project	53
1. Introduction to the Eclipse project	54
1.1. The Eclipse community and projects	54
1.2. A short Eclipse history	55
1.3. Eclipse releases	56
2. Eclipse project structure	58
2.1. Eclipse foundation	58
2.2. Staff of the Eclipse foundation	59
3. The Eclipse Public License	60
3.1. Source code and the Eclipse Public License	60
3.2. Intellectual property cleansing of Eclipse code	61
II. The concepts behind Eclipse applications	62
4. Eclipse plug-ins and applications	63
4.1. What is an Eclipse RCP application?	63
4.2. Eclipse software components - Plug-ins	64
4.3. Advantages of developing Eclipse plug-ins	65
4.4. What is the Eclipse Rich Client Platform (Eclipse RCP)?	66

5. Important Eclipse projects for RCP	67
5.1. The Eclipse Platform project	67
5.2. The Eclipse e4 project	68
5.3. The Eclipse Plug-in Development Environment (PDE)	69
5.4. The e4 tools project	70
6. Architecture of Eclipse	71
6.1. Architecture of Eclipse based applications	71
6.2. Core components of the Eclipse platform	73
6.3. Compatibility layer for Eclipse 3.x plug-ins	74
6.4. Eclipse API and internal API	75
6.5. Important configuration files for Eclipse plug-ins	77
III. Setting up an Eclipse RCP development environment	78
7. Install Java for Eclipse RCP development	79
7.1. Java requirements of the Eclipse IDE	79
7.2. Check installation	80
7.3. Install Java on Ubuntu	81
7.4. Install Java on MS Windows	82
7.5. Installation problems and other operating systems	83
7.6. Validate installation	84
7.7. How can you tell you are using a 32 bit or 64 bit version of Java?	85
8. Install Eclipse IDE for RCP development	86
8.1. Download the Eclipse Software Development Kit (SDK)	86
8.2. Install the Eclipse IDE	87
8.3. Starting the Eclipse IDE	88
8.4. Appearance	91
9. Install the e4 tools	92
9.1. Requirements	92
9.2. Install the e4 tools from the vogella GmbH update site	93
9.3. Install the e4 tools from the Eclipse.org update site	95
10. Start Eclipse and use a new workspace	97
10.1. About the Eclipse IDE	97
10.2. Workspace	98
10.3. Review the Eclipse 4 model editor preferences	99

11. Enable Java access to all plug-ins	100
11.1. Filtering by the Java tools	100
11.2. Include all plug-ins in Java search	101
IV. Eclipse applications and product configuration files	102
12. Product configuration file	103
12.1. The Product configuration file and the application	103
12.2. Creating a new product configuration file	104
12.3. Using the product editor	105
12.4. Define the plug-ins which are included in this product	107
13. Branding	108
13.1. Splash screen	108
13.2. Icons, launcher name and program arguments	109
13.3. Product configuration limitations	111
14. The usage of run configurations	112
14.1. What are run configurations?	112
14.2. Reviewing run configurations	113
14.3. Run arguments	114
14.4. Launch configuration and Eclipse products	115
15. Common launch problems	116
15.1. Checklist for common launch problems	116
15.2. Finding missing plug-in dependencies during a product launch	118
16. Exercise: Create and run an RCP application	120
16.1. Target	120
16.2. Create project	121
16.3. Launch your Eclipse application via the product file	126
16.4. Validating	128
17. Features and feature projects	129
17.1. What are feature projects and features?	129
17.2. Creating a feature	130
17.3. Feature or plug-in based products	132
17.4. Advantages of using features	133
18. Exercise: Use a feature based product	134
18.1. Create a feature project	134

18.2. Include plug-in into feature project	135
18.3. Change product configuration file to use features	136
18.4. Add features as dependency to the product	137
18.5. Start the application via the product	138
V. Deploying Eclipse applications	139
19. Deployment of your RCP application	140
19.1. Creating a stand-alone version of the application	140
19.2. Exporting via the product file	141
19.3. Defining which artifacts are included in the export	143
19.4. Mandatory plug-in artifacts in build.properties	144
19.5. Export for multiple platforms via the delta pack	146
19.6. More about the target platform	147
19.7. Including the required JRE into the export	148
19.8. Headless build	149
20. Common product export problems	150
20.1. Problems with the export and log files	150
20.2. Export problem number #1: export folder is not empty	151
20.3. Checklist for common export problems	152
21. Exercise: Export your product	154
21.1. Export your product	154
21.2. Validate that the exported application starts	156
22. Exercise: Splash screen and launcher name	157
22.1. Using a static splash screen	157
22.2. Include a splash screen into the exported application	159
22.3. Change launcher name	161
VI. Application model	162
23. Eclipse application model	163
23.1. What is the application model?	163
23.2. Scope of the application model	164
23.3. How do you define the application model?	165
24. Important user interface model elements	166
24.1. Window	166
24.2. Parts	167

24.3. Part container	169
24.4. Perspective	172
25. Connecting model elements to classes and resources	173
25.1. URI patterns or classes and resources	173
25.2. Connect model elements to classes	174
25.3. Connect model elements to resources like icons	175
26. Model objects and the runtime application model	176
26.1. Model objects	176
26.2. Runtime application model	177
27. Using the model spy	178
27.1. Analyzing the application model with the model spy	178
27.2. Installation requirements	179
27.3. Model spy and the Eclipse IDE	180
28. Exercise: Create an Eclipse plug-in	181
28.1. Target	181
28.2. Creating a plug-in project	182
28.3. Validate the result	185
29. Exercise: From plug-in to Eclipse RCP	186
29.1. Target	186
29.2. Create a project to host the product configuration file	187
29.3. Create a product configuration file	189
29.4. Configure the start levels	193
29.5. Create a feature project	194
29.6. Enter the feature dependencies in product	196
29.7. Remove the version dependency from the features in product	198
29.8. Create an application model	200
29.9. Add a window to the application model	202
29.10. Start the application	204
30. Exercise: Configure the deletion of persisted model data	205
30.1. Delete the persisted user changes at startup	205
30.2. Why is this setting necessary?	206
31. Exercise: Modeling a user interface	207
31.1. Desired user interface	207

31.2. Open the Application.e4xmi file	208
31.3. Add a perspective	209
31.4. Add part sash and part stack containers	211
31.5. Create the parts	213
31.6. Validate the user interface	214
32. Exercise: Connect Java classes with the parts	215
32.1. Create a new package and some Java classes	215
32.2. Connect the Java classes with your parts	216
32.3. Validating	218
33. Exercise: Enter the dependencies	219
33.1. Add the plug-in dependencies	219
33.2. Validating	220
33.3. More on dependencies	221
34. Optional Exercise: Using the model spy	222
34.1. Target	222
34.2. Adding the model spy to your runtime configuration	223
34.3. Use the model spy	225
34.4. Cleanup - remove model spy from your run configuration	226
VII. Annotations and dependency injection	227
35. The concept of dependency injection	228
35.1. What is dependency injection?	228
35.2. Using annotations to describe class dependencies	230
35.3. Where can objects be injected into a class?	231
35.4. Order in which dependency injection is performed on a class	232
36. Dependency injection in Eclipse	233
36.1. Define class dependencies in Eclipse	233
36.2. Annotations to define class dependencies in Eclipse	234
36.3. On which objects does Eclipse perform dependency injection?	236
36.4. Dynamic dependency injection based on key / value changes	237
VIII. Dependency injection and the Eclipse context	238
37. The Eclipse context	239
37.1. What is the Eclipse context?	239
37.2. Relationship definition in the Eclipse context	240

37.3. Which model elements have a local context?	242
37.4. Life cycle of the Eclipse context	243
38. Object selection during dependency injection	244
38.1. How are objects selected for dependency injection	244
38.2. How to access the model objects?	246
38.3. Default entries in the Eclipse context	247
38.4. Qualifiers for accessing the active part or shell	248
38.5. Tracking a child context with @Active	249
IX. Behavior annotations	250
39. Using annotations to define behavior	251
39.1. API definition	251
39.2. API definition via inheritance	252
39.3. API definition via annotations	253
39.4. Behavior annotations imply method dependency injection	254
39.5. Use the @PostConstruct method to build the user interface	255
39.6. Why is the @PostConstruct method not called?	256
40. Exercise: Using @PostConstruct	257
40.1. Implement an @PostConstruct method	257
40.2. Validating	258
X. Eclipse modularity based on OSGi	259
41. Software modularity with OSGi	260
41.1. What is software modularity?	260
41.2. What is OSGi?	261
41.3. OSGi implementations	262
41.4. Plug-in or bundles as software component	263
41.5. Naming convention: simple plug-in	264
41.6. Find the plug-in for a certain class	265
42. OSGi metadata	266
42.1. The manifest file (MANIFEST.MF)	266
42.2. Bundle-SymbolicName and Version	268
42.3. Semantic Versioning with OSGi	269
43. Defining the dependencies of a plug-in	270
43.1. Specifying plug-in dependencies via the manifest file	270

43.2. Life cycle of plug-ins in OSGi	272
43.3. Dynamic imports of packages	273
44. Defining an API	274
44.1. Specifying the API of a plug-in	274
44.2. Provisional API	275
44.3. Provisional API with exceptions via x-friends	277
45. Using the OSGi console	278
45.1. The OSGi console	278
45.2. Required bundles	280
45.3. Telnet	281
45.4. Access to the Eclipse OSGi console	282
46. Exercise: Data model plug-in	283
46.1. Target of the exercise	283
46.2. Create the plug-in for the data model	284
46.3. Create the base class	285
46.4. Generate constructors	286
46.5. Generate getter and setter methods	287
46.6. Generate <code>toString()</code> , <code>hashCode()</code> and <code>equals()</code> methods	290
46.7. Write a <code>copy()</code> method	291
46.8. Create the interface for the todo service	292
46.9. Define the API of the model plug-in	293
47. Exercise: Service plug-in	294
47.1. Target of the exercise	294
47.2. Create a data model provider plug-in (service plug-in)	295
47.3. Define the dependencies in the service plug-in	296
47.4. Provide an implementation of the <code>ITodoService</code> interface	297
47.5. Create a factory	299
47.6. Export the package in the service plug-in	302
48. Exercise: Use the new plug-ins	303
48.1. Update the plug-in dependencies	303
48.2. Update the product configuration (via your feature)	304
48.3. Use the data model provider in your parts	305
48.4. Validate your implementation	306

48.5. Review your implementation	307
49. OSGi fragments and fragment projects	308
49.1. What are fragments in OSGi?	308
49.2. Typical use cases for fragments	309
XI. Using the service layer of OSGi	310
50. OSGi service introduction	311
50.1. What are OSGi services?	311
50.2. Life cycle status for providing services	312
50.3. Best practices for defining services	313
50.4. Service properties	314
50.5. Service priorities	315
51. Defining OSGi services	316
51.1. Defining declarative services	316
51.2. Required bundles	317
51.3. Activating the declarative service plug-in	318
52. Steps to declare an OSGi service	319
52.1. Defining the service interface	319
52.2. Providing a service implementation	320
52.3. Service declaration with a component definition file	321
52.4. Reference to the service in the MANIFEST.MF file	324
52.5. Low-level OSGi service API	325
53. Eclipse RCP and OSGi services	326
53.1. Using declarative services in RCP applications	326
53.2. OSGi services and Eclipse dependency injection	327
54. Exercise: Define and use an OSGi service	329
54.1. Target of this exercise	329
54.2. Define the component definition file	330
54.3. Set the lazy activation flag for the service plug-in	333
54.4. Get the ITodoService injected	334
54.5. Possible issues: ITodoService cannot get injected	335
54.6. Clean-up the service implementation	336
54.7. Review the service implementation	337
55. Optional exercise: Create an image loader service	338

55.1. Target of this exercise	338
55.2. Creating a new plug-in	339
55.3. Creating a service implementation	340
55.4. Defining a new service	341
55.5. Adding the new plug-in to your feature project	343
55.6. Exporting the API	344
55.7. Add a MANIFEST.MF dependency to your new plug-in	345
55.8. Using the new service	346
55.9. Reviewing the implementation	348
XII. User interface development with SWT	349
56. Standard Widget Toolkit	350
56.1. What is SWT?	350
56.2. Eclipse applications and SWT	351
56.3. Display and Shell	352
56.4. Event loop	353
56.5. Relationship to JFace	354
56.6. Using SWT in a plug-in project	355
57. Using SWT widgets	356
57.1. Available widgets in the SWT library	356
57.2. Memory management	359
57.3. Constructing widgets	360
57.4. Basic containers	361
57.5. Event Listener	362
58. Using layout managers in SWT	363
58.1. The role of a layout manager	363
58.2. Layout Data	364
58.3. FillLayout	365
58.4. RowLayout	366
58.5. GridLayout	367
58.6. Using GridDataFactory	369
58.7. Tab order of elements	370
58.8. Example: Using layout manager	371
59. SWT widget examples and controls	374

59.1. SWT snippets and examples	374
59.2. The Nebula and Opal widgets	375
60. SWT Designer	377
60.1. SWT Designer (WindowBuilder)	377
60.2. Install SWT Designer	378
60.3. Using SWT Designer	379
61. Exercise: Getting started with SWT Designer	381
61.1. Installation	381
61.2. Building an user interface	382
61.3. Creating an event handler	385
61.4. Review the generated code	386
62. Exercise: Build a first SWT UI	387
62.1. TodoOverviewPart	387
62.2. Example code for TodoOverviewPart	388
63. Exercise: Implement a UI for TodoDetailsPart	390
63.1. TodoDetailsPart	390
63.2. Implement focus setting for one of your widgets	391
63.3. Example code for TodoDetailsPart	392
64. Exercise: Prepare TodoDetailsPart for data	394
64.1. Preparing for data	394
64.2. Add dependency	395
64.3. Prepare for dependency injection	396
64.4. Usage of this method	398
65. Exercise: Using the SWT Browser widget	399
65.1. Implementation	399
65.2. Solution	400
XIII. User interface development with JFace	402
66. JFace	403
66.1. What is Eclipse JFace?	403
66.2. JFace resource manager for Colors, Fonts and Images	404
66.3. ControlDecoration	405
66.4. User input help with field assistance	406
67. The JFace viewer framework	408

67.1. Purpose of the JFace viewer framework	408
67.2. Standard JFace viewer	409
67.3. Standard content and label provider	410
67.4. JFace ComboViewer	411
68. Using tables	414
68.1. Using the JFace TableViewer	414
68.2. Content provider for JFace tables	415
68.3. Columns and label provider	416
68.4. Reflect data changes in the viewer	418
68.5. Selection change listener	419
68.6. Column editing support	420
68.7. Filtering data	422
68.8. Sorting data with ViewerComparator	424
68.9. TableColumnLayout	425
68.10. StyledCellLabelProvider and OwnerDrawLabelProvider	426
68.11. Table column menu and hiding columns	428
68.12. Tooltips for viewers	429
68.13. Virtual tables with LazyContentProvider	430
68.14. Alternative table implementations	431
69. Exercise: Using TableViewer	432
69.1. Create a JFace TableViewer for the Todo items	432
69.2. Add dependencies	433
69.3. Create implementation	434
70. Exercise: Using more viewer functionality	437
70.1. Add a filter to the table	437
70.2. Add a ControlDecoration to the TodoDetailsPart	439
71. Optional exercise: Using ComboViewer	440
71.1. Target	440
71.2. Implementation	441
71.3. Validation	443
72. Using trees	444
72.1. Using viewers to display a tree	444
72.2. Selection and double-click listener	445

73. Optional exercises: Using TreeViewer	446
73.1. Create a new application	446
73.2. Add an image file	448
73.3. Create a part	449
73.4. Validating	452
XIV. Defining menus and toolbars	453
74. Menu and toolbar application objects	454
74.1. Adding menu and toolbar entries	454
74.2. What are commands and handlers?	455
74.3. Mnemonics	456
74.4. Standard commands	457
74.5. Naming schema for command and handler IDs	458
75. Dependency injection for handler classes	459
75.1. Handler classes and their behavior annotations	459
75.2. Which context is used for a handler class?	461
75.3. Scope of handlers	462
75.4. Evaluation of @CanExecute	463
75.5. Learn about the event system	464
76. Exercise: Adding a menu and menu entries	465
76.1. Target of this exercise	465
76.2. Create command model elements	466
76.3. Creating the handler classes	467
76.4. Creating handler model elements	468
76.5. Adding a menu	469
76.6. Implement a handler class for exit	471
76.7. Validating	472
76.8. Possible issue: Exit menu entry on a Mac OS	473
77. Exercise: Adding a toolbar	474
77.1. Target of this exercise	474
77.2. Adding a toolbar	475
77.3. Validating	477
78. View, popup and dynamic menus	478
78.1. View menus	478

78.2. Popup menu (context menu)	479
78.3. Dynamic menu and toolbar entries	481
78.4. Creating dynamic menu entries	482
79. Exercise: Add a context menu to a table	483
79.1. Target	483
79.2. Dependencies	484
79.3. Implementation	485
80. Toolbars, ToolControls and drop-down tool items	486
80.1. Adding toolbars to parts	486
80.2. ToolControls	487
80.3. Drop-down tool items	489
81. More on commands and handlers	490
81.1. Passing parameters to commands	490
81.2. Usage of core expressions	493
81.3. Evaluate your own values in core expressions	495
81.4. Example for dynamic model contributions	496
XV. Using key bindings	497
82. Key bindings	498
82.1. Using key bindings in your application	498
82.2. JFace default values for binding contexts	499
82.3. Define Shortcuts	500
82.4. Activate bindings	501
82.5. Key bindings for a part	502
83. Exercise: Define key bindings	503
83.1. Create binding context entries	503
83.2. Create key bindings for a BindingContext	504
XVI. Dialogs and wizards	505
84. Dialogs	506
84.1. Dialogs in Eclipse	506
84.2. SWT dialogs	507
84.3. JFace Dialogs	508
85. Exercise: Dialogs	516
85.1. Confirmation dialog at exit	516

85.2. Create a password dialog	518
86. Wizards	522
86.1. What is a wizards	522
86.2. Wizards and WizardPages	523
86.3. Starting the Wizard	524
86.4. Changing the page order	525
86.5. Working with data in the wizard	526
86.6. Updating the Wizard buttons from a WizardPage	527
87. Exercise: Create a wizard	528
87.1. Create classes for the wizard	528
87.2. Adjust part	532
87.3. Adjust handler implementation	534
87.4. Validating	535
XVII. Data Binding with JFace	536
88. Data Binding with JFace	537
88.1. What are Data Binding frameworks?	537
88.2. JFace Data Binding	538
88.3. JFace Data Binding Plug-ins	539
89. Listening to changes	540
89.1. Ability to listen to changes in UI components	540
89.2. Ability to listen to changes in the domain model	541
89.3. Property change support	542
89.4. Data Binding and Java objects without change notification	544
90. Create bindings	545
90.1. Observing properties with the IObservableValue interface	545
90.2. Creating instances of the IObservableValue	546
90.3. Connecting properties with the DataBindingContext	547
90.4. Example: how to observe properties	548
90.5. Observing nested properties	550
91. Updates, convertors and validators	551
91.1. UpdateValueStrategy	551
91.2. Converter	553
91.3. Validator	554

92. More on bindings	555
92.1. ControlDecorators	555
92.2. Placeholder binding with WritableValue	556
92.3. Listening to all changes in the binding	557
92.4. More information on Data Binding	559
93. Exercise: Data Binding for SWT widgets	560
93.1. Add the plug-in dependencies	560
93.2. Implement the property change support	561
93.3. Remove the modification listeners in your code	564
93.4. Add a field for the data binding context to TodoDetailsPart	565
93.5. Implement data binding in TodoDetailsPart	566
93.6. Validating	568
94. Data Binding for JFace viewer	570
94.1. Binding Viewers	570
94.2. Observing list details	571
94.3. ViewerSupport	572
94.4. Master Detail binding	573
94.5. Chaining properties	574
95. Exercise: Data Binding for viewers	575
95.1. Implement Data Binding for the viewer	575
95.2. Clean up old code	578
95.3. Validating	579
XVIII. Using Eclipse services	580
96. Eclipse platform services	581
96.1. What are Eclipse platform services?	581
96.2. Overview of the available platform services	582
97. Implementation	583
97.1. How are Eclipse platform services implemented?	583
97.2. References	584
XIX. Selection Service	585
98. Selection service	586
98.1. Usage of the selection service	586
98.2. Changing the current selection	587

98.3. Getting the selection	588
99. Exercise: Selection service	589
99.1. Target of this exercise	589
99.2. Retrieving the selection service	590
99.3. Setting the selection in TodoOverviewPart	591
99.4. Review TodoDetailsPart	592
99.5. Validate selection propagation	593
100. Exercise: Selection service for deleting data	594
100.1. Implement the RemoveTodoHandler handler	594
100.2. Validate that the deletion works	595
100.3. Test the context menu for deletion	596
XX. Model service and model modifications at runtime	597
101. Model service	598
101.1. What is the model service?	598
101.2. How to access the model service	599
101.3. Cloning elements or snippets	600
101.4. Searching model elements	601
102. Modifying the application model at runtime	603
102.1. Creating model elements	603
102.2. Modifying existing model elements	604
103. Example for application model modifications	605
103.1. Example: Search for a perspective and change its attributes	605
103.2. Example: Dynamically create a new window	606
103.3. Example: Dynamically create a new part	607
104. Exercise: Creating dynamic menu entries	608
104.1. Target of this exercise	608
104.2. Create handler class	609
104.3. Create class for DynamicMenuContribution	610
104.4. Add DynamicMenuContribution model element	611
104.5. Validating	612
XXI. Part service and implementing editors	613
105. Using the part service	614
105.1. What is the part service?	614

105.2. How to access the part service	615
105.3. Example: Showing and hiding parts	616
105.4. Example: Switching perspectives	617
105.5. Using part descriptors	618
105.6. Example: Part descriptors and creating parts dynamically	619
106. Implementing editors	621
106.1. Parts which behave similar to editors	621
106.2. MDirtyable and @Persist	622
106.3. Use part service to trigger save in editors	624
106.4. MPart and multiple editors	625
106.5. MInputPart	626
106.6. Code examples for editor implementations	627
107. Exercise: Implement an editor	628
107.1. Add the plug-in dependencies	628
107.2. Convert TodoDetailsPart to an editor	629
107.3. Implement the save handler	632
107.4. Validating	633
107.5. Confirmation dialog for modified data	634
108. Exercise: Enable handlers and avoiding data loss	635
108.1. Enable the save handler only if necessary	635
108.2. Enable the deletion handler only if necessary	636
108.3. Avoid data loss in your ExitHandler	637
109. Exercise: Using multiple perspectives	638
109.1. Target of this exercise	638
109.2. Create a new perspective	639
109.3. Create new menu entries	640
109.4. Using command parameters to define the perspective ID	642
110. Exercise: Sharing elements between perspectives	643
110.1. Target	643
110.2. Using shared parts between perspectives	644
111. Optional exercise: Dynamic creation of parts based on a part descriptor	647
111.1. Create a class for the part descriptor	647

111.2. Create the part descriptor model element	648
111.3. Create a handler for creating new parts	649
112. Exercise: Implement multiple editors	651
112.1. Prerequisites	651
112.2. New menu entries	652
112.3. Validate ID of the PartStack	653
112.4. Add a handler and a part implementation	654
112.5. Validate multiple editor implementation	659
XXII. Handler and command services	660
113. Command and handler service	661
113.1. Purpose of the command and handler service	661
113.2. Access to command and handler service	662
113.3. Example for executing a command	663
113.4. Example for assigning a handler to a command	664
114. Optional exercise: Using handler service	665
114.1. Delete Todos only via the handler	665
114.2. Implementation	666
XXIII. Asynchronous processing	669
115. Threading in Eclipse	670
115.1. Concurrency	670
115.2. Main thread	671
115.3. Using dependency injection and UISynchronize	672
115.4. Eclipse Jobs API	673
115.5. Priorities of Jobs	674
115.6. Blocking the UI and providing feedback	675
116. Progress reporting	676
116.1. IProgressMonitor	676
116.2. Reporting progress in Eclipse RCP applications	677
117. Exercise: Using asynchronous processing	680
117.1. Simulate delayed access	680
117.2. Use asynchronous processing	681
117.3. Validating	683
117.4. Remove delay	684

XXIV. Event service for message communication	685
118. Eclipse event notifications	686
118.1. Event based communication	686
118.2. The event bus of Eclipse	687
118.3. Event service	688
118.4. Required plug-ins to use the event service	689
118.5. Sending and receiving events	690
118.6. Usage of the event system	693
118.7. Asynchronous processing and the event bus	694
119. Exercise: Event notifications	695
119.1. Creating a plug-in for event constants	695
119.2. Add the new plug-in to your product	696
119.3. Enter the plug-in dependencies	697
119.4. Send out notifications	698
119.5. Receive updates in your parts	700
119.6. Validating	701
119.7. Review the implementation	702
XXV. Extending and modifying the Eclipse context	703
120. Accessing and extending the Eclipse context	704
120.1. Accessing the context	704
120.2. Objects and context variables	706
120.3. Replacing existing objects in the IEclipseContext	708
120.4. Accessing the IEclipseContext hierarchy from OSGi services	709
120.5. Model add-ons	710
120.6. RunAndTrack	711
121. Using dependency injection for your Java objects	712
121.1. Creating and injecting custom objects	712
121.2. Using dependency injection to create objects	713
121.3. Create your custom objects automatically with @Creatable	715
121.4. Create automatically objects in the application context with @Singleton	717
122. Exercise: Dependency injection for your objects	718
122.1. Target of this exercise	718

122.2. Prepare the wizard classes for dependency injection	719
122.3. Create the wizard via dependency injection	721
XXVI. Eclipse context functions	723
123. Context functions	724
123.1. What are context functions?	724
123.2. Creation of a context function	725
123.3. Examples for context function registrations	726
123.4. When to use context functions?	728
123.5. Publishing to the OSGi service registry from a context function	729
124. Exercise: Create a context function	730
124.1. Target	730
124.2. Add dependencies to the service plug-in	731
124.3. Create a class for the context function	732
124.4. Register the context function	733
124.5. Specify the responsibility of the context function	734
124.6. Error analysis	737
124.7. Deactivate your ITodoService OSGi service	738
124.8. Notifications from the ITodoService	739
124.9. Clean-up your user interface code	742
124.10. Validating	743
124.11. Review implementation	744
XXVII. Model add-ons	745
125. Using model add-ons	746
125.1. What are model add-ons?	746
125.2. Add-ons from the Eclipse framework	747
125.3. Additional SWT add-ons	748
125.4. Relationship to other services	749
126. Exercise: Model add-on to change the close behavior	750
126.1. Target	750
126.2. Creating a plug-in	751
126.3. Creating the model add-on	752
XXVIII. Supplementary application model data	755
127. Supplementary model data	756

127.1. Adding additional information on the model elements	756
127.2. Tags	757
127.3. Variables	759
127.4. Persisted state	760
127.5. Transient data	761
127.6. Relevant tags and persisted state keys in the application model	762
128. Exercise: Using model tags and persisted data	763
128.1. Target	763
128.2. Use tags	764
128.3. Optional: Use persisted state	765
XXIX. Application model modularity	766
129. Contributing to the application model	767
129.1. Modularity support in Eclipse RCP	767
129.2. Contributing to the application model	768
129.3. Constructing the runtime application model	770
129.4. Fragment extension elements	772
130. Exercise: Contributing via model fragments	773
130.1. Target	773
130.2. Create a new plug-in	774
130.3. Add the dependencies	775
130.4. Create a handler class	776
130.5. Create a model fragment	777
130.6. Validate that the fragment is registered as extension	779
130.7. Adding model elements	781
130.8. Update the product configuration (via the feature)	787
130.9. Validating	788
130.10. Exercise: Contributing a part	789
131. Exercise: Implementing a model processor	790
131.1. Target	790
131.2. Enter the dependencies	791
131.3. Create the Java classes	792
131.4. Register processor via extension	795
131.5. Validating	797

XXX. Eclipse application life cycle	798
132. Registering for the application life cycle	799
132.1. Connecting to the Eclipse application life cycle	799
132.2. Accessing application startup parameters	800
132.3. Close static splash screen	801
132.4. How to implement a life cycle class	802
132.5. Example life cycle implementation	803
133. Exercise: Life cycle hook and a login screen	805
133.1. Target	805
133.2. Create a new class	806
133.3. Register life cycle hook	808
133.4. Validating	810
XXXI. Handling preferences	811
134. Eclipse preference basics	812
134.1. Preferences and scopes	812
134.2. Storage of the preferences	813
134.3. Eclipse preference API	814
134.4. Setting preferences via plugin_customization.ini	815
135. Preferences and dependency injection	816
135.1. Preferences and dependency injection	816
135.2. Persistence of part state	818
136. Exercise: Using preferences in the life cycle class	819
136.1. Target	819
136.2. Dependency	820
136.3. Create interface for the preference constants	821
136.4. Using preferences in the life cycle class	822
136.5. Validate life cycle handling	824
137. Exercise: Using preferences in a handler and in a part	825
137.1. Target	825
137.2. Using preferences in your handler	826
137.3. Validate menu entry	828
XXXII. Internationalization	829
138. Internationalization and localization in Eclipse	830

138.1. Translation of a Java applications	830
138.2. Property files	831
138.3. Encoding of property files in Java	832
138.4. Relevant files for translation in Eclipse applications	833
138.5. Where to store the translations?	834
138.6. Setting the language in the launch configuration	835
138.7. Translation service	836
139. Translating the application model and plugin.xml	837
139.1. OSGi resource bundles	837
139.2. Translating the application model	838
139.3. Translating plugin.xml	839
140. Source code translation with the Eclipse translation service	840
140.1. Translation with POJOs	840
140.2. Search process for translation files	841
140.3. Dynamic language switch	843
141. Source code translation with NLS support	845
141.1. NLS compared to the Eclipse translation service	845
141.2. Translating your custom code	846
141.3. Translating SWT and JFace code	850
142. Exporting and common problems with translations	851
142.1. Exporting Plug-ins and Products	851
142.2. Common problems with i18n	852
143. Optional Exercise: Internationalization for the application model	853
143.1. Target	853
143.2. Create the translations for the application model	854
143.3. Translate the application model	855
143.4. Test the translation of the application model	856
143.5. Application model and translations	857
144. Optional exercise: Internationalization for the source code	858
144.1. Target	858

144.2. Creating a plug-in to host the translations	859
144.3. Create a Message class for the source code translations	860
144.4. Create the translations for the source code	861
144.5. Export the translations as API	862
144.6. Define dependencies to the translation plug-in	863
144.7. Update the product (via the feature)	864
144.8. Using @Translation to get the messages injected	865
144.9. Test your translation	866
XXXIII. Custom annotations and ExtendedObjectSupplier	867
145. Usage of custom annotations	868
145.1. Custom annotations	868
145.2. Restrictions	870
145.3. Example usage	871
146. Exercise: Defining custom annotations	872
146.1. Target	872
146.2. Creating a new plug-in	873
146.3. Define and export annotations	874
146.4. Create class	875
146.5. Register the annotation processor	876
146.6. Add the plug-in as dependency	880
146.7. Update the product configuration (via the features)	881
146.8. Update the build.properties	882
146.9. Validate: Use your custom annotation	883
XXXIV. Using Java libraries in Eclipse applications	884
147. Defining and using libraries in Java	885
147.1. What is a JAR file?	885
147.2. Using Java libraries	886
148. Using JAR files in Eclipse applications	887
148.1. JAR files without OSGi meta-data	887
148.2. Integrating external jars / third party libraries	888
XXXV. Testing of Eclipse plug-ins and applications	892
149. Introduction to JUnit	893
149.1. The JUnit framework	893

149.2. How to define a test in JUnit?	894
149.3. Example JUnit test	895
149.4. JUnit naming conventions	896
149.5. JUnit test suites	897
149.6. Run your test from the command line	898
150. Basic JUnit code constructs	899
150.1. Available JUnit annotations	899
150.2. Assert statements	900
150.3. Test execution order	902
151. Eclipse IDE support for JUnit tests	903
151.1. Creating JUnit tests	903
151.2. Running JUnit tests	904
151.3. JUnit static imports	906
151.4. Wizard for creating test suites	907
151.5. Testing exception	908
151.6. JUnit Plug-in Test	909
151.7. Setting Eclipse up for using JUnits static imports	910
152. Testing Eclipse 4 applications	912
152.1. General testing	912
152.2. Fragment projects	913
152.3. Testing user interface components	914
152.4. Testing dependency injection	916
153. UI testing with SWTBot	917
153.1. User interface testing with SWTBot	917
153.2. Installation	918
153.3. SWTBot API	919
XXXVI. Eclipse application updates	920
154. Implementing updates in your application	921
154.1. Eclipse application updates	921
154.2. Creating p2 update sites	922
154.3. p2 composite repositories	924
155. Using the p2 update API	925
155.1. Required plug-ins for updates	925

155.2. Updating Eclipse RCP applications	926
156. Exercise: Performing an application update	929
156.1. Preparation: Ensure the exported product works	929
156.2. Select an update location	930
156.3. Add dependencies	931
156.4. Add the p2 feature to the product	932
156.5. Create a user interface	933
156.6. Enter a version in your product configuration file	934
156.7. Create the initial product export	935
156.8. Start the exported application and check for updates	936
156.9. Make a change and export the product again	937
156.10. Update the application	938
XXXVII. Using target platforms	939
157. Target Platform	940
157.1. Defining available plug-ins for development	940
157.2. Target platform definition	941
157.3. Using an explicit target definition file	942
157.4. Defining a target platform	943
158. Exercise: Defining a target platform	946
158.1. Creating a target definition file	946
158.2. Activate your target platform for development	952
158.3. Validate that target platform is active	953
158.4. Solving potential issues for development	954
XXXVIII. Using custom extension points	955
159. Creating and evaluating extension points	956
159.1. Extensions and extension points	956
159.2. Creating an extension point	957
159.3. Adding extensions to extension points	958
159.4. Accessing extensions	959
159.5. Extension Factories	960
160. Exercise: Create and evaluate extension point	961
160.1. Target for this exercise	961
160.2. Creating a plug-in for the extension point definition	962

160.3. Create an extension point	964
160.4. Export the package	970
160.5. Add dependencies	971
160.6. Evaluating the registered extensions	972
160.7. Create a menu entry and add it to your product	974
160.8. Providing an extension	975
160.9. Add the plug-in to your product	978
160.10. Validating	979
XXXIX. Eclipse styling with CSS	980
161. Introduction to CSS styling	981
161.1. Cascading Style Sheets (CSS)	981
161.2. Styling Eclipse applications	982
161.3. Limitations	983
162. How to style in Eclipse	984
162.1. Fixed styling in Eclipse	984
162.2. Dynamic styling using themes	986
163. More details on Eclipse styling	987
163.1. CSS attributes and selectors	987
163.2. Styling based on identifiers and classes	988
163.3. Colors and gradients	989
163.4. CSS imports	992
163.5. CSS Tools	993
164. Exercise: Styling with CSS files	995
164.1. Target	995
164.2. Create a CSS file	996
164.3. Define the applicationCSS property	997
164.4. Validating	999
164.5. Adjust the build.properties file	1000
165. Exercise: Using the theme service	1001
165.1. Target	1001
165.2. Add dependencies	1002
165.3. Create a CSS file	1003
165.4. Remove the applicationCSS property	1004

165.5. Create the theme extensions	1005
165.6. Validating	1009
165.7. Implement a new menu entry	1010
165.8. Validate theme switching	1011
165.9. Optional: Reusing the dark theme of Eclipse	1012
165.10. Adjust the build.properties file	1013
166. Exercise: Styling the widgets created by the life cycle class	1014
166.1. Target	1014
166.2. Implement styling	1015
166.3. Validating	1017
XL. The renderer framework	1018
167. The usage of renderer	1019
167.1. Renderer	1019
167.2. Renderer factory and renderer objects	1020
167.3. Context creation of model objects	1021
167.4. Using a custom renderer	1022
167.5. Using a custom renderer for one model element	1023
168. Existing alternative renderer implementations	1024
168.1. Alternatives to SWT	1024
168.2. Vaadin renderer - Vaaeclipse	1025
168.3. JavaFX renderer - e(fx)clipse	1027
168.4. Eclipse RAP	1028
168.5. Additional UI toolkits	1029
169. Exercise: Defining a renderer	1030
169.1. Target	1030
169.2. Creating a plug-in	1031
169.3. Enter the dependencies	1032
169.4. Create the renderer implementation	1033
169.5. Register the renderer	1035
169.6. Validating	1036
169.7. Exercise: A custom part renderer	1037
XLI. Application model persistence and model extensions	1040

170. Custom persistence for the application model	1041
170.1. Specifying the location of the application model file	1041
170.2. Custom application model persistence handler	1042
171. Saving and restoring parts of the application model	1043
171.1. Store certain model elements	1043
171.2. Load stored model elements	1045
172. Extend the Eclipse application model	1047
XLII. Good development practices	1048
173. Eclipse development good practices	1049
173.1. Project, package and class names	1049
173.2. Naming conventions for model identifiers (IDs)	1050
173.3. Create isolated components	1051
173.4. Usage of your custom extension points	1052
173.5. Avoid releasing unnecessary API	1053
173.6. Packages vs. plug-in dependencies	1054
174. Application communication and context usage	1055
174.1. Application communication	1055
174.2. Example: Using events together with the IEclipseContext	1056
174.3. Which dependency injection approach to use for your implementation	1058
XLIII. Migrating Eclipse 3.x components and RCP applications	1059
175. Why migrating an Eclipse 3.x RCP application?	1060
175.1. Using the Eclipse 3.x API on top of an 4.x runtime	1060
175.2. Technical reasons for migrating to the 4.x API	1061
176. Running Eclipse 3.x plug-ins on top of Eclipse 4	1062
176.1. Using the compatibility mode	1062
176.2. The e4 API and e4 runtime terminology	1064
176.3. Running 3.x RCP applications on top of an e4 runtime	1065
176.4. Benefit of adjusting the runtime to Eclipse 4.x	1066
177. Excursion: Extending the Eclipse IDE	1067
177.1. Extending the Eclipse IDE	1067
177.2. Starting the Eclipse IDE from Eclipse	1068

177.3. Starting a new Eclipse instance	1069
177.4. Debugging the Eclipse instance	1070
178. Partial migrating to the Eclipse 4.x API	1071
178.1. Using e4 API in 3.x applications	1071
178.2. Adding e4 commands, menus and toolbars to 3.x based applications	1072
178.3. Adding Eclipse 4.x parts and perspectives to 3.x based applications	1075
178.4. Accessing the IEclipseContext from 3.x API	1076
179. Exercise: Adding an e4 menu and toolbar to a 3.x based application	1077
179.1. Target of this exercise	1077
179.2. Creating a plug-in project	1078
179.3. Starting an Eclipse IDE with your plug-in	1083
179.4. Adding the plug-in dependencies for the e4 API	1084
179.5. Creating the handler class	1085
179.6. Creating a model contribution	1086
179.7. Adding a toolbar contribution	1090
179.8. Validating the presence of the menu and toolbar contribution	1092
180. Exercise: Using POJOs to contribute views to a 3.x based application	1093
180.1. Target	1093
180.2. Using e4part and the org.eclipse.ui.views extension point	1094
180.3. Add the view to a perspective extension	1099
180.4. Validating	1100
181. Exercise: Adding e4 part descriptors to 3.x based applications	1101
181.1. Target	1101
181.2. Adding a part descriptor	1102
181.3. Validating	1104
182. Optional Exercise: Model add-on to change the close behavior of the IDE	1105
182.1. Target	1105

182.2. Register model add-on via model fragment with the Eclipse IDE	1106
182.3. Verifying	1107
183. Migrating to an e4 API application without 3.x API usage	1108
183.1. Migrating an Eclipse 3.x application completely to the e4 API	1108
183.2. Using 3.x components in e4 API based applications	1109
183.3. Reusing platform components	1110
183.4. Existing replacements for 3.x components for e4 API based applications	1111
183.5. Example for using 3.x components in e4 API based applications	1112
XLIV. Questions, feedback and closing words	1113
184. Questions, feedback	1114
184.1. Reporting Eclipse bugs and feature requests	1114
184.2. Using the Eclipse bugzilla system	1115
184.3. Eclipse bug priorities	1116
184.4. Asking (and answering) questions	1117
184.5. Eclipse 4 feedback	1118
185. Closing words	1119
XLV. Appendix	1120
A. Eclipse annotations, extension points	1121
A.1. Standard annotations in Eclipse	1121
A.2. Relevant extension points for Eclipse 4	1123
B. Solutions for the exercises	1124
B.1. Getting the example implementation	1124
B.2. More information about Git and Eclipse	1127
C. Recipes	1128
C.1. Eclipse update manager	1128
C.2. Performing an update and install new features	1129
C.3. See the installed components	1133
C.4. Uninstalling components	1134
C.5. Restarting Eclipse	1135
C.6. Reading resources from plug-ins	1136
C.7. Loading images from a plug-in	1138

C.8. Getting the command line arguments	1139
D. Architectural background of the application model	1140
D.1. Main areas of the model	1140
D.2. Advantages of using mix-ins	1141
E. OSGi low level service API	1142
E.1. Using the service API	1142
E.2. BundleContext	1143
E.3. Registering services via API	1144
E.4. Accessing a service via API	1145
E.5. Low-level API vs OSGi declarative services	1146
F. Links and web resources	1147
F.1. Eclipse RCP resources	1147
F.2. Links and Literature	1148
F.3. Eclipse product and export resources	1149
F.4. OSGi Resources	1150
F.5. OSGi Resources	1151
F.6. Eclipse SWT resources	1152
F.7. JFace resources	1153
F.8. Eclipse Data Binding resources	1154
F.9. Eclipse Jobs resources	1155
F.10. Eclipse i18n resources	1156
F.11. CSS styling resources	1157
F.12. Eclipse p2 updater resources	1158
F.13. Logging	1159
Index	1160