427

Free tutorial, donate to support

Donate

by Lars Vogel

Eclipse 4 RCP - Tutorial

Building Eclipse RCP applications based on Eclipse 4

Lars Vogel

Version 6.7

Copyright © 2009 , 2010 , 2011 , 2012 Lars Vogel

05.11.2012

Revision History			
Revision 0.1	14.02.2009	Lars Vogel	created
Revision 0.2 - 6.7	16.02.2009 - 05.11.2012	Lars Vogel	bug fixes and enhancements

Eclipse e4

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

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

Table of Contents

1. Eclipse 4

1.1. What is Eclipse 4?

Eclipse 4 introduces a new set of technologies which increase the flexibility of Eclipse plug-in development. The simultaneous release of Eclipse named *Juno* in June 2012 was based on Eclipse 4.2.

Eclipse 3.8 was released in parallel to Eclipse 4.2 but the Eclipse 3.x series will be discontinued after the Eclipse 3.8 release, i.e. there will not be an Eclipse 3.9 release. Eclipse 3.8 receives bug fixes for a year via service releases.

Eclipse 4 was an opportunity to rationalize the best parts of the Eclipse 3.x APIs and to fix pain points of Eclipse 3.x development. Eclipse 3.x changed from 3.0 to 3.8, and tough lessons were learned about the difficulties and flexibility of the APIs.

Eclipse 3.x developers will find that many of the concepts that know are carried across and found in Eclipse 4.

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

- The Eclipse application is described via a defined structure called the application model
- This application model can be modified at development and runtime
- This application model can be extended
- Eclipse 4 supports dependency injection
- Eclipse widgets can be styled via external CSS files, similar to webpages
- The application model is decoupled from its presentation, e.g. different user interface toolkits such as SWT or JavaFX, can be used to render the model

Eclipse 4 provides a compatibility layer which allows Eclipse 3.x plug-ins to run unmodified.

Eclipse 4 primarily contains enhancements for Eclipse based development. The user of the Eclipse IDE, e.g. a web developer or a Swing developer, will only indirectly benefit because tool development of Eclipse becomes easier. This allows Eclipse developers to create better tools for Eclipse users.

1.2. 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 Platform project is to enable other developers to easily build and deliver integrated tools and applications.

1.3. What is the Eclipse e4 project?

Eclipse e4 is the name used for the *Eclipse Platform's* incubator project. This incubator is used for exploratory projects relating to improving the Eclipse Platform. The new Eclipse 4 platform was originally developed within the incubator and the incubator name became synonymous with the new Eclipse Platform. Today the term *e4* it is only the name of the project.

The e4 project includes several technology evaluations. Some of these evaluations have been ported back to the core Eclipse framework. All functionality described in this document are part of the official Eclipse 4 release, except the *Eclipse e4 tooling* project.

The subproject *Eclipse e4 tooling* provides tools to develop Eclipse 4 applications. They are very useful, but have not been added to the Eclipse core because they are not as mature as the other components.

Projects such as XWT, TM or OpenSocial Gadgets, which are also part of the Eclipse e4 project, are not included in the standard Eclipse 4.2 core platform and are not described in this document.

1.4. Provisional API

Currently the Application Programming Interface (API) for Eclipse 4 is still marked as provisional, i.e. this means that the API might be changed in the future.

Eclipse 4.2 already runs the complete Eclipse IDE therefore it is unlikely that dramatic changes will occur in the future.

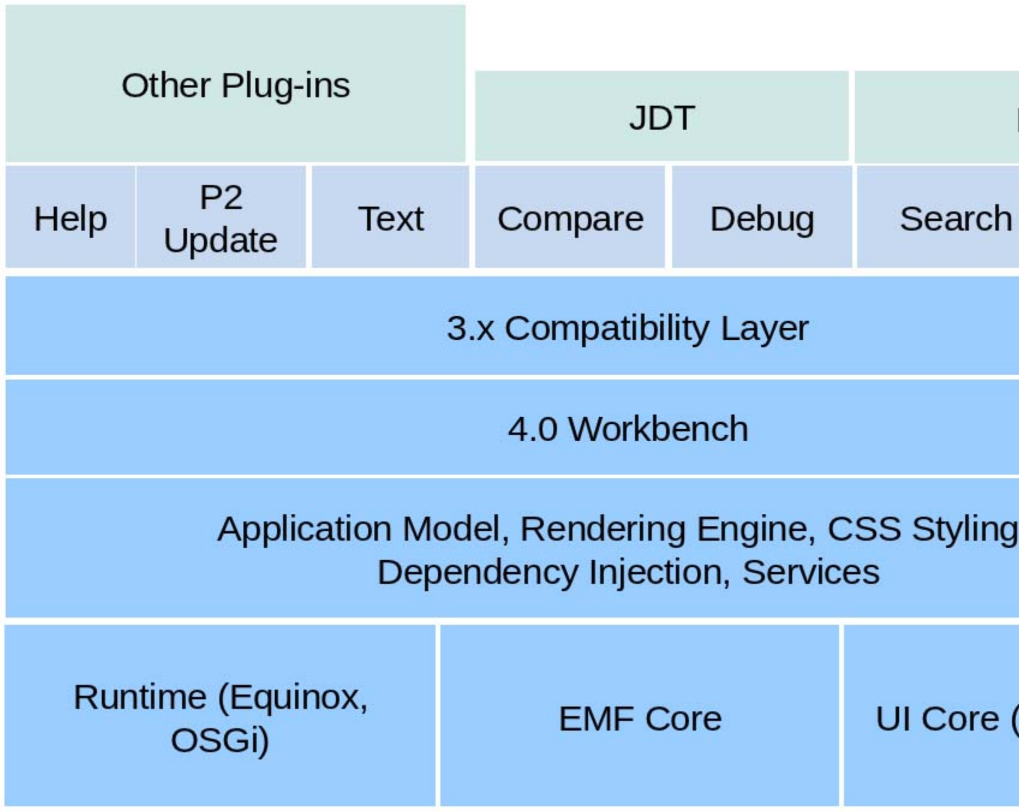
It is therefore relatively safe to start using the API now, but nevertheless you must be prepared that you might have to make some adjustments to your application.

2. The Architecture of Eclipse

2.1. Eclipse based applications

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

The components of the Eclipse IDE are primarily the following. Please note that the graph should display the concept, the displayed relationship is not 100 % accurate.



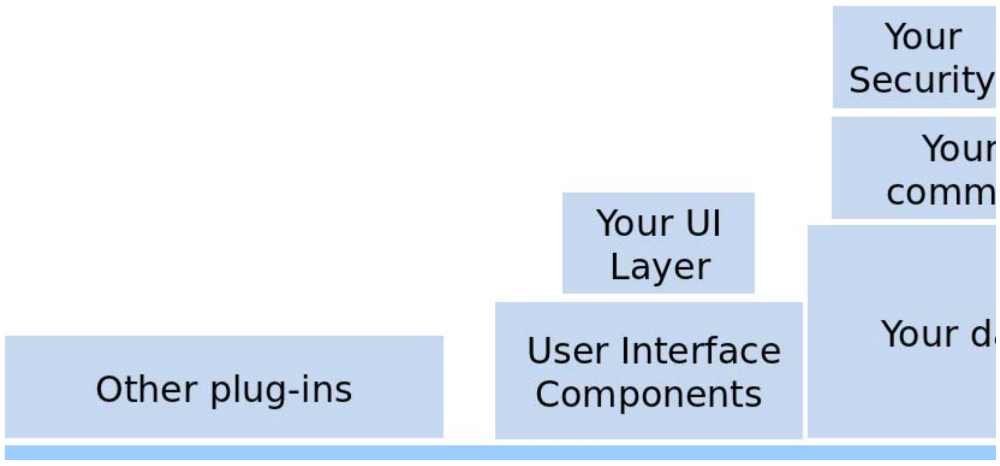
OSGi is a specification which describes a modular approach for Java application. Equinox is one implementation of OSGi and is used by the Eclipse platform. The Equinox runtime provides the necessary framework to run a modular Eclipse application.

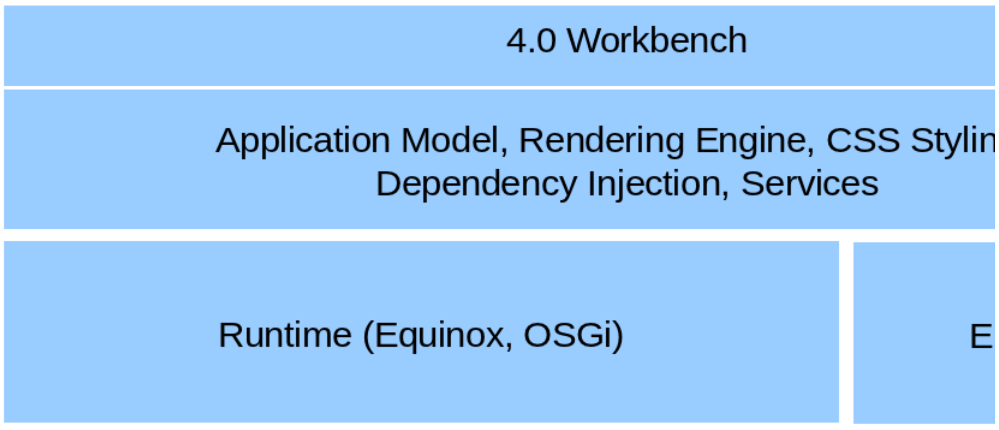
SWT is the standard user interface component library used by Eclipse. JFace provides some convenient APIs on top of SWT. The workbench provides the framework for the application. The workbench is responsible for displaying all other UI components.

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

Eclipse 4 has a different programming model then Eclipse 3.x. Eclipse 4 provides the *3.x Compatibility Layer* component which maps the 3.x API to the 4.0 API. This allows Eclipse 3.x based components to run unmodified on Eclipse 4.

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




4.0 Workbench
Application Model, Rendering Engine, CSS Stylin
Dependency Injection, Services
Runtime (Equinox, OSGi)
E

The programming model of OSGi (Equinox) allows you to define dynamic software components, i.e. OSGi services, which can also be part of an Eclipse based application.

2.2. Terminology

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

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

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

2.3. Important configuration files

An Eclipse plug-in has the following main configuration files.

- MANIFEST.MF - Contains the OSGi configuration information.
- plugin.xml - Contains information about Eclipse specific extension mechanisms

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

The `plugin.xml` file provides the possibility to define *extension points* and *extensions*. *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.

In Eclipse 4 the usage of extension points and extensions is very limited. They are mostly used to define pointers to other Eclipse 4 configuration files.

3. Tutorial: Install Eclipse 4.2 for RCP development

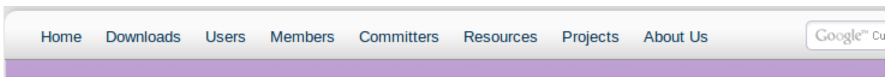
3.1. Prerequisites

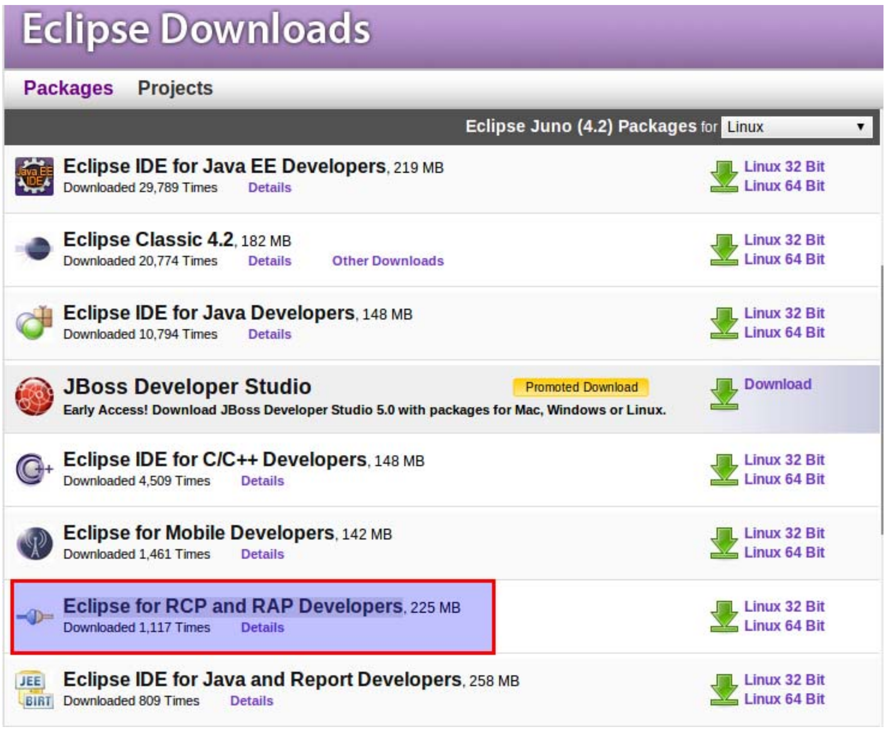
The following assumes that you have Java installed in at least version 1.6.

3.2. Download and Install Eclipse 4.2

This tutorial is based on the Eclipse 4.2 release. Download the *Eclipse for RCP and RAP Developers* package from the following URL.

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




Eclipse Downloads

Packages Projects

Eclipse Juno (4.2) Packages for Linux

Eclipse IDE for Java EE Developers, 219 MB Downloaded 29,789 Times Details		Linux 32 Bit Linux 64 Bit
Eclipse Classic 4.2, 182 MB Downloaded 20,774 Times Details Other Downloads		Linux 32 Bit Linux 64 Bit
Eclipse IDE for Java Developers, 148 MB Downloaded 10,794 Times Details		Linux 32 Bit Linux 64 Bit
JBoss Developer Studio Early Access! Download JBoss Developer Studio 5.0 with packages for Mac, Windows or Linux.	Promoted Download	Download
Eclipse IDE for C/C++ Developers, 148 MB Downloaded 4,509 Times Details		Linux 32 Bit Linux 64 Bit
Eclipse for Mobile Developers, 142 MB Downloaded 1,461 Times Details		Linux 32 Bit Linux 64 Bit
Eclipse for RCP and RAP Developers, 225 MB Downloaded 1,117 Times Details		Linux 32 Bit Linux 64 Bit
Eclipse IDE for Java and Report Developers, 258 MB Downloaded 809 Times Details		Linux 32 Bit Linux 64 Bit

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

Do not extract Eclipse to a directory with a path which contains spaces as this might lead to problems in the usage of Eclipse.

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

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

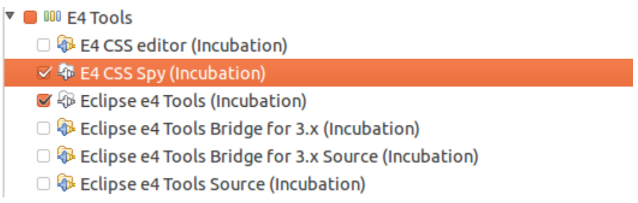
3.3. Install the Eclipse 4 tooling

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

The update site of the *Eclipse e4 tooling* for the Eclipse 4.2 release is the following.

```
http://download.eclipse.org/e4/updates/0.12
```

From this update site, install only the *E4 CSS Spy* and the *Eclipse e4 Tools*.


E4 Tools
- [ ] E4 CSS editor (Incubation)
- [✓] E4 CSS Spy (Incubation)
- [✓] Eclipse e4 Tools (Incubation)
- [ ] Eclipse e4 Tools Bridge for 3.x (Incubation)
- [ ] Eclipse e4 Tools Bridge for 3.x Source (Incubation)
- [ ] Eclipse e4 Tools Source (Incubation)

4. Tutorial: Eclipse 4 application using the wizard

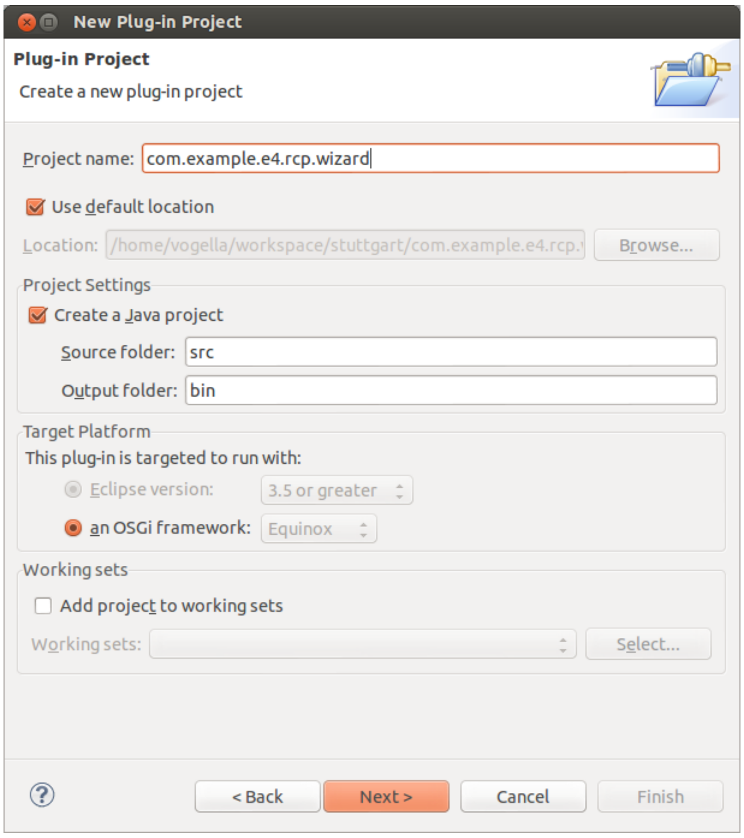
4.1. Overview

The Eclipse 4 tooling project provides a project generation wizard which allows you to create a working Eclipse 4 based RCP application.

4.2. Create project

Select *File → New → Others → Eclipse 4 → Eclipse 4 Application Project* .

Create a project called `com.example.e4.rcp.wizard` using the default settings. This should be similar to the following screenshots.


New Plug-in Project

Plug-in Project

Create a new plug-in project

Project name: com.example.e4.rcp.wizard

[✓] Use default location

Location: /home/vogella/workspace/stuttgart/com.example.e4.rcp.

Project Settings

[✓] Create a Java project

Source folder: src

Output folder: bin

Target Platform

This plug-in is targeted to run with:

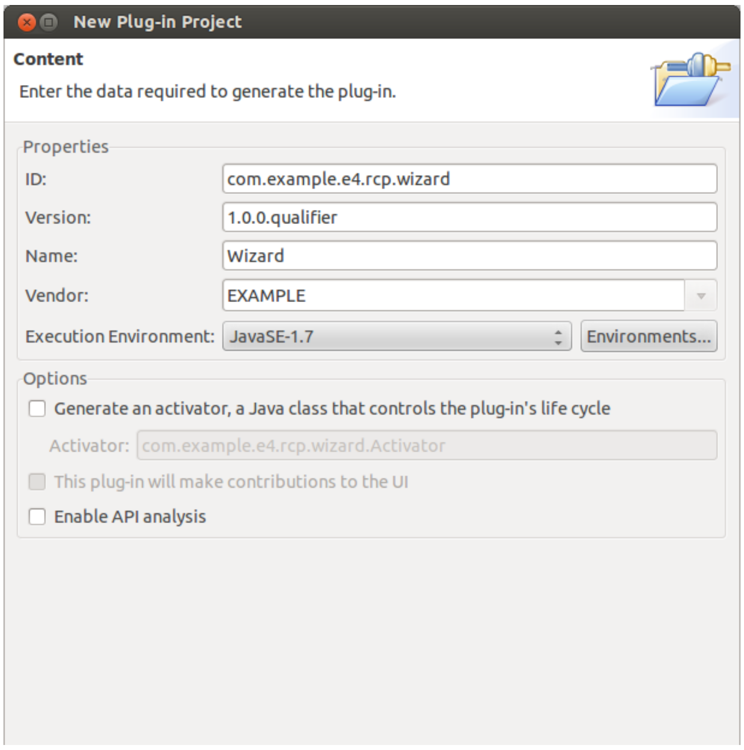
○ Eclipse version: 3.5 or greater

● an OSGi framework: Equinox

Working sets

☐ Add project to working sets

Working sets:


New Plug-in Project

Content

Enter the data required to generate the plug-in.

Properties

ID:	com.example.e4.rcp.wizard
Version:	1.0.0.qualifier
Name:	Wizard
Vendor:	EXAMPLE
Execution Environment:	JavaSE-1.7

Options

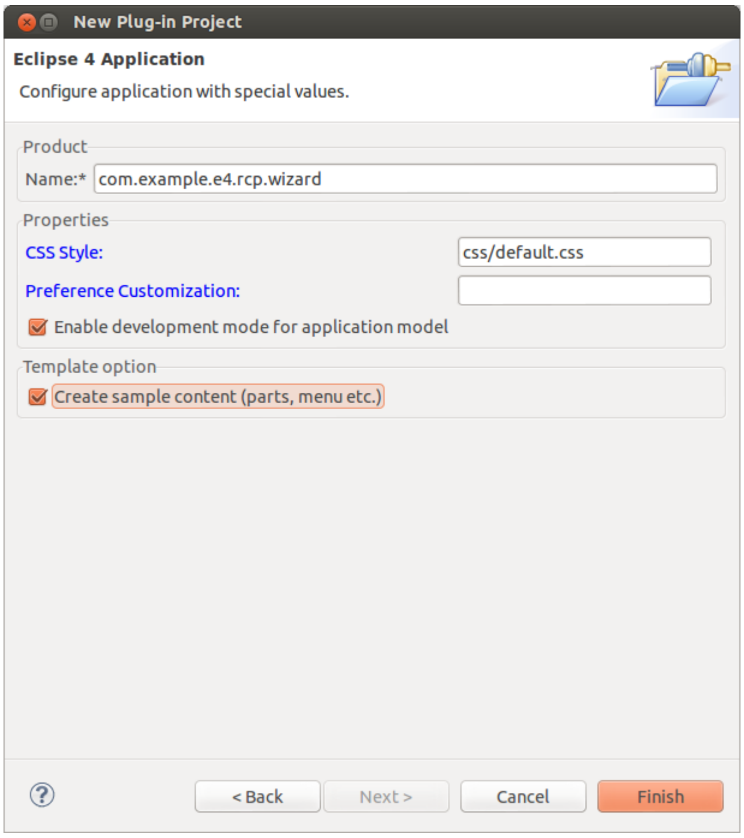
☐ Generate an activator, a Java class that controls the plug-in's life cycle

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

☐ This plug-in will make contributions to the UI

☐ Enable API analysis

Select the development mode and the creation of sample context checkbox on the last page.


New Plug-in Project

Eclipse 4 Application

Configure application with special values.

Product

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

Properties

CSS Style: css/default.css

Preference Customization:

☑ Enable development mode for application model

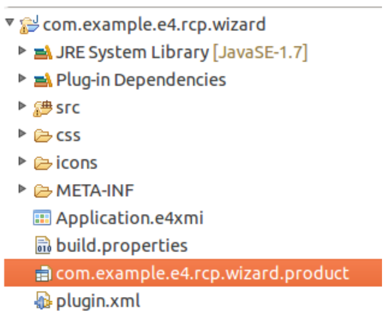
Template option

☑ Create sample content (parts, menu etc.)

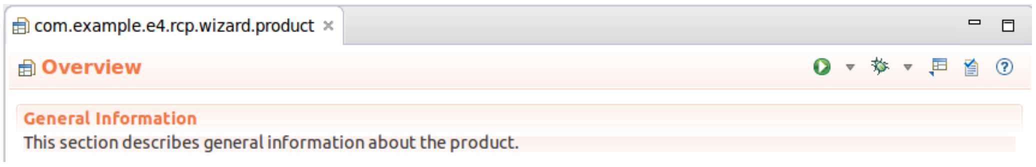
This wizard creates all the necessary files to start your application. The central file for starting your application is the `.product` file, created in your project folder.

4.3. Launch

Open your `com.example.e4.rcp.wizard.product` product configuration file by double-clicking on the file in package explorer.


```
▼ com.example.e4.rcp.wizard
  ▶ JRE System Library [JavaSE-1.7]
  ▶ Plug-in Dependencies
  ▶ src
  ▶ css
  ▶ icons
  ▶ META-INF
    Application.e4xmi
    build.properties
    com.example.e4.rcp.wizard.product
    plugin.xml
```

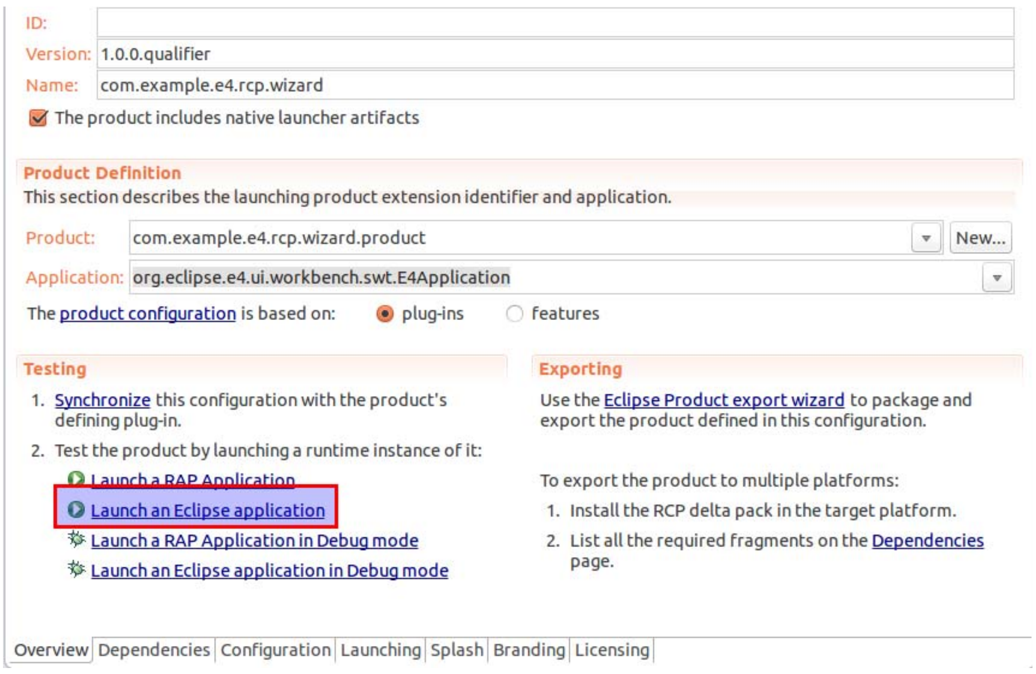
Switch to the *Overview* tab and launch your product by pressing the *Launch an Eclipse application* hyperlink. This should start the generated Eclipse application.


com.example.e4.rcp.wizard.product

Overview

General Information

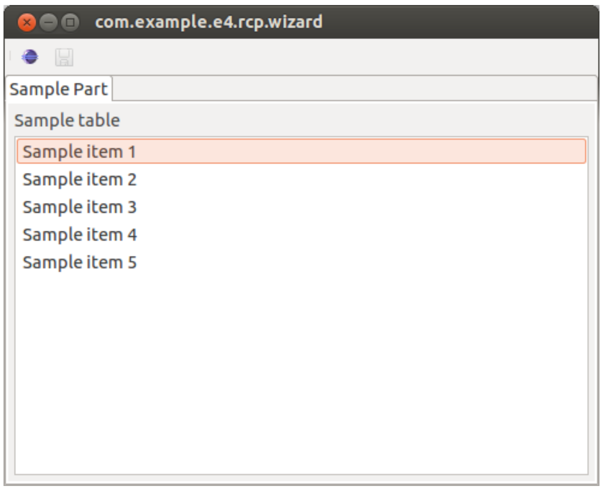
This section describes general information about the product.


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


com.example.e4.rcp.wizard
Sample Part
Sample table
Sample item 1
Sample item 2
Sample item 3
Sample item 4
Sample item 5

5. Eclipse 4 application model

5.1. What is the application model?

The visual part of an Eclipse application consists of *Perspectives*, *Parts* (Views and Editors), Menus, Toolbars, etc. An Eclipse application also includes non-visual components, e.g. *Handlers*, *Commands* and *Key bindings* .

Eclipse 4 uses an abstract description, called the *application model*, to describe the structure of an application. This application model contains the visual elements as well as some non-visual elements of the Eclipse 4 application.

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

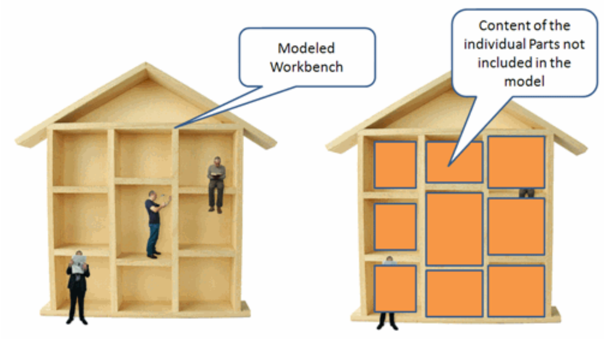
The application model defines the structure of the application; it does not describe the content of the individual user interface components.

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

Parts properties, e.g. if a *Part* is closable, its label, ID, etc.

But it does not describe the content of the *Part*, e.g. the labels, text fields and button it consists of. The content of the *Part* is still defined by your source code.

If the application model was a house, it would describe the available rooms (*Parts*) and their arrangement (*Perspectives*, *PastStacks*, *PartSashContainer*) but not the furniture of the rooms. This is illustrated by the following image.



5.2. Where is the application model defined?

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

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

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

5.3. How is the model connected to my Java classes?

Application model elements can contain references to Java classes via an Uniform Resource Identifier (URI).

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

For example a model description for a *Part* contains attributes such as labels, tooltips and icon URIs. It also contains a class URI which points to a Java class for this element. This class provides the behavior of the *Part* — using the house/rooms metaphor from earlier, the class is responsible for defining the furnishings and the layout of the room, and how the interactive objects behave.

If the model elements get activated this class will get instantiated.

The objects created based on the application model are called *model objects*.

5.4. URI in the model

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

Table 1. URI pattern

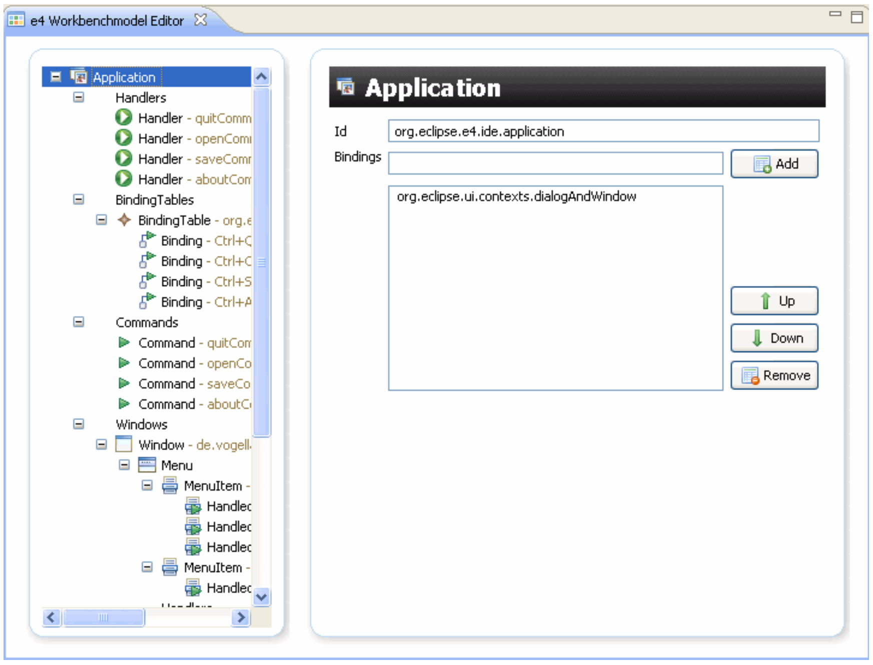
Pattern	Description
bundleclass://Bundle-SymbolicName/ package.classname Example:	Used to identify Java classes. It consists of the following parts: "bundleclass://" is a fixed schema, Bundle-SymbolicName as defined in the *`MANIFEST.MF`* file, and the fully qualified classname

Example: bundleclass://test/test.parts.MySavePart	the fully qualified classname.
platform:/plugin/Bundle-SymbolicName/path/filename.extension Example: platform:/plugin/test/icons/save_edit.gif	Identifier for a resource in the plug-in. "platform:/plugin/" is a fixed schema, followed by the Bundle-SymbolicName of the `MANIFEST.MF` file, followed by the path to the file and the filename.

5.5. Application model editor

The Eclipse tooling project provides an editor which makes it easier to work on an application model.

To open the model editor double-click on your `Application.e4xmi` file (or right click on it and select *Open With → Eclipse 4 model editor* .

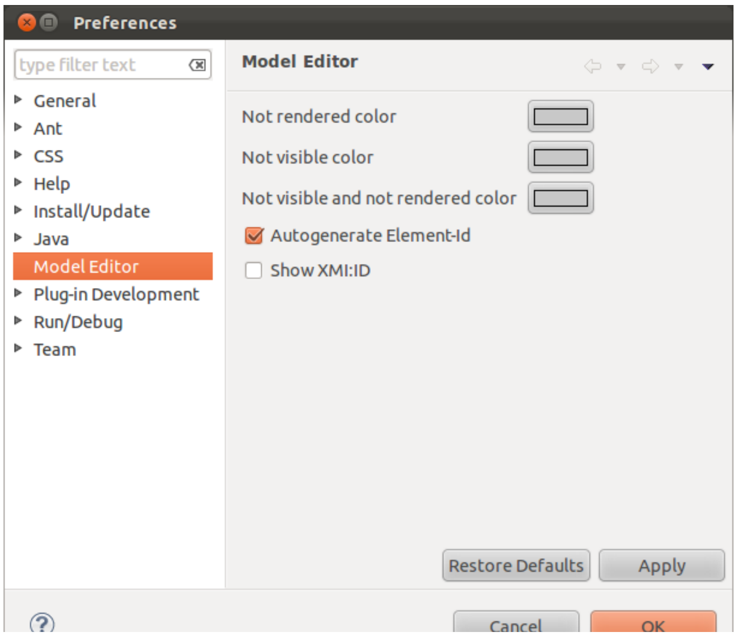


Id org.eclipse.e4.ide.application

Bindings

org.eclipse.ui.contexts.dialogAndWindow

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



Not rendered color

Not visible color

Not visible and not rendered color

☑ Autogenerate Element-Id

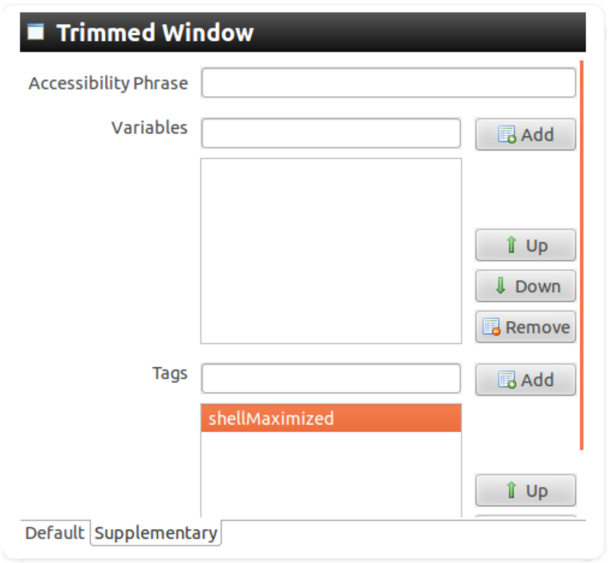
☐ Show XMI:ID

5.6. Supplementary

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

All model elements allow to enter *`Tags`*. These *`Tags`* can be used by the Eclipse platform to select the related model elements via the *ModelService*. 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*.



shellMaximized

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

```
// modelObject is the model object
// retrieved via dependency injection

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

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

5.7. Model access at runtime

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

Add the `org.eclipse.e4.tools.emf.liveeditor` plug-in and its dependencies to your launch configuration to make the model editor available in your application.

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

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

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

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

5.8. Meta-model of the application model

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

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

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

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

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

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

5.9. Creation of the runtime application model

During startup the Eclipse runtime creates the application model based on the *Application.e4xmi* and instantiates the referred classes in the model if required.

The life cycle of every model object is therefore controlled by the Eclipse runtime. The Eclipse runtime instantiates and destroys the model objects.

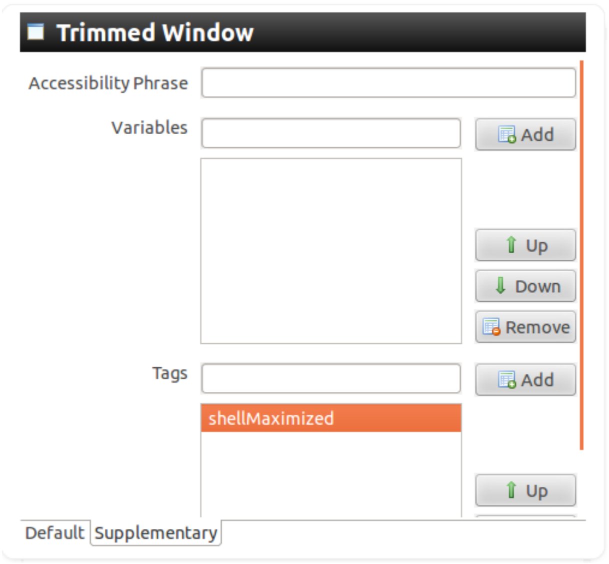
6. Supplementary model attributes

6.1. Tags

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

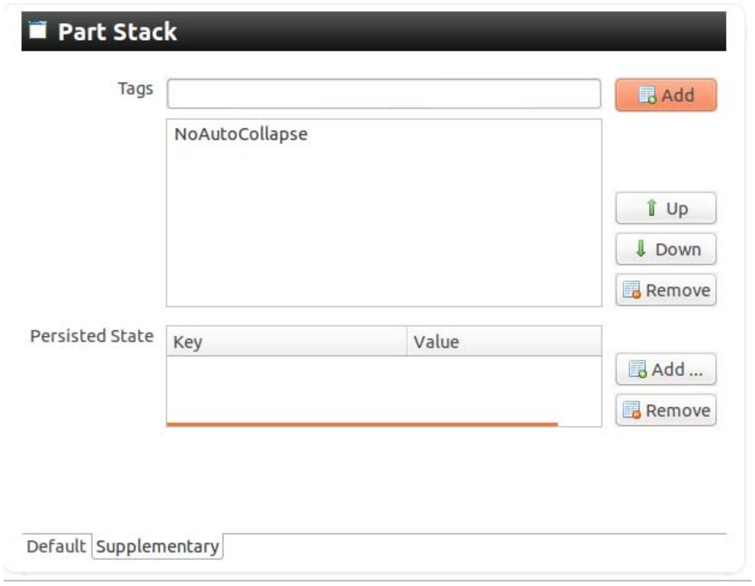
All model elements allow to enter *Tags*. These *Tags* can be used by the Eclipse platform to select the related model elements via the *ModelService*. By default Eclipse uses some predefined *Tags* to determine the state of certain model elements. For example the *shellMaximized* and *shellMinimized* 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*.



shellMaximized

Another tag which is used frequently is the *NoAutoCollapse* tag which you can add to a *PartStack* container. With this flag the *PartStack* will not collapse even if you remove all parts from it.



Check the appendix of this {tutorial} for more tags.

6.2. Persisted State

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

```
// modelObject is the model object
// retrieved via dependency injection

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

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

7. Overview of available model elements

Model elements have Java classes associated with them. As the model is interactive you can use these model elements to change its attributes or children.

The following is a table of important model elements based on their Java classes. These model elements can also get injected.

Table 2. Eclipse 4 model element

Model element	Description
MApplication	Describes the application object. Can be used for example to add new windows to your application
MWindow	Represents a Window in your application.
MTrimmedWindow	Represents a Window in your application. The underlying SWT shell has been created with the SWT.SHELL_TRIM attribute which means, it has a tile, a minimize, maximize and resize button.
MPerspective	Object for the perspective model element.
MPart	Represents the model element part, e.g. a View or a Editor.
MDirtyable	Property of MPart which can be injected. If set to true, this property informs the Eclipse platform that this Part contains unsaved data (is dirty). In a Handler you can query this property to trigger a save.

MPartDescriptor	MPartDescriptor is a template for new Parts. You define in your application model a PartDescriptor. A new Part based on this PartDescriptor can be created via the EPartService and shown its `showPart()` method.
Snippets	Snippets can be used to pre-configure model parts which you want to create via your program. You can use `EcoreUtil.copy` to copy a Snippet and assign it to another model element, e.g. on a `MSash` you can add a newly copied `MStack` called *copy*, via `getChildren().add(copy)`. Also the model service has some methods to create and clone a snippet.

8. Identifiers for model elements

8.1. Identifiers for model elements

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

Therefore make sure that all your model elements have an ID assigned to them.

8.2. Best practices for naming conventions

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

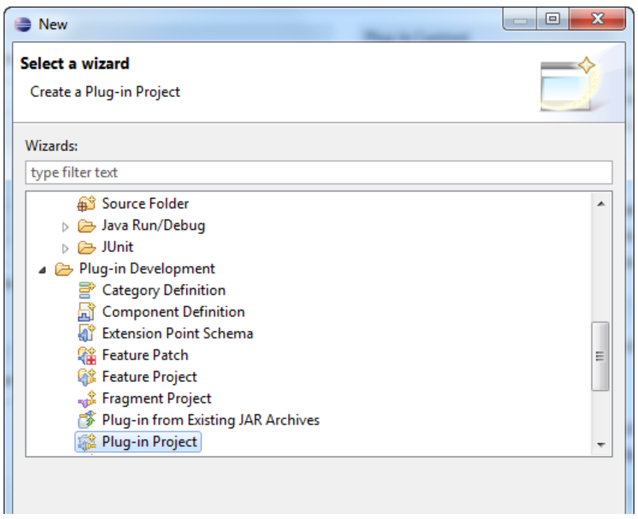
Table 3. Naming conventions

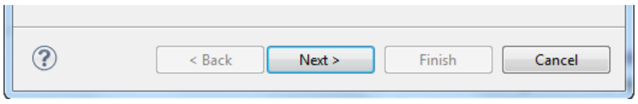
Object	**Description**
Project Names	The plug-in project name is the same as the top-level package name.
Packages	For plug-in containing lots of user interface components use sub-packages based on the primary purpose of the components. For example the `com.example` package may have the `com.example.parts` and `com.example.handler` 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 which represents a Part which displays `Todo` objects, might be called `TodoOverviewPart`.
IDs	Define clear rules for naming IDs in plugin.xml. IDs should always start with the top-level package. If appropriate use the sub-package of the implementing class also. The remainder of the ID should be descriptive for the purpose of the component. For example: "com.example.parts.todolist". ID should be only lower cases (some Eclipse projects also use camelCase for the last part of the ID).

9. Tutorial: Create an Eclipse plug-in

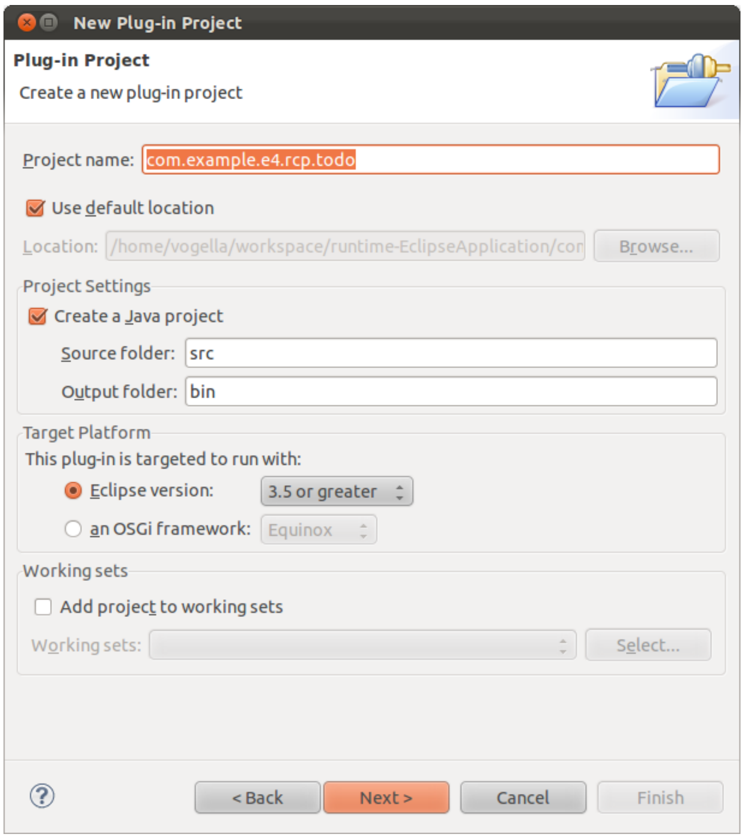
9.1. Create a plug-in project

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





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


New Plug-in Project

Plug-in Project
Create a new plug-in project

Project name: com.example.e4.rcp.todo

[✓] Use default location

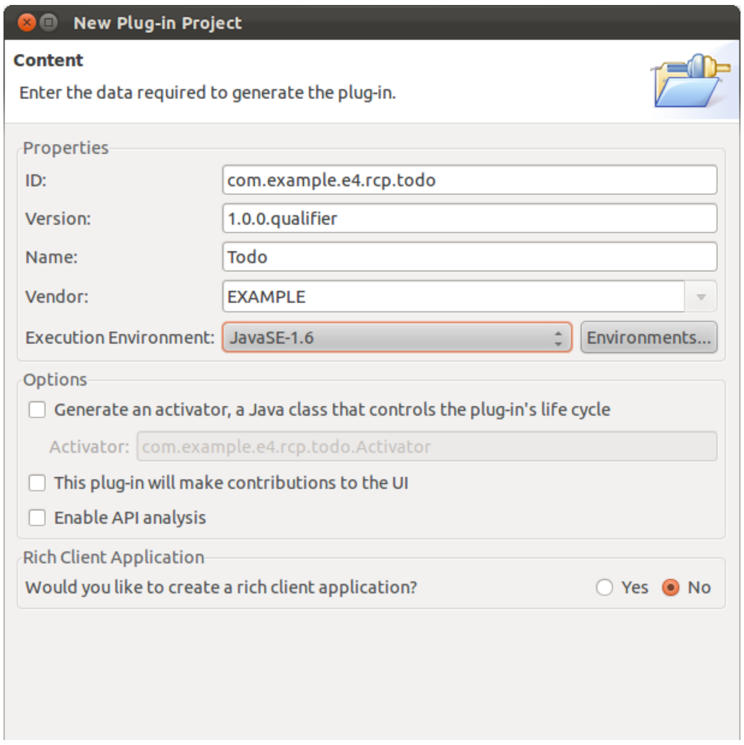
Location: /home/vogella/workspace/runtime-EclipseApplication/cor

Project Settings
[✓] Create a Java project
Source folder: src
Output folder: bin

Target Platform
This plug-in is targeted to run with:
[✓ Eclipse version: 3.5 or greater]
an OSGi framework: Equinox

Working sets
[ ] Add project to working sets
Working sets:

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


New Plug-in Project

Content
Enter the data required to generate the plug-in.

Properties

ID:	com.example.e4.rcp.todo
Version:	1.0.0.qualifier
Name:	Todo
Vendor:	EXAMPLE
Execution Environment:	JavaSE-1.6

Options
[ ] Generate an activator, a Java class that controls the plug-in's life cycle
Activator: com.example.e4.rcp.todo.Activator
[ ] This plug-in will make contributions to the UI
[ ] Enable API analysis

Rich Client Application
Would you like to create a rich client application? Yes [✓ No]

Press the *Finish* button; we will not use a template.

9.2. Validate the result

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

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

10. Feature project

An Eclipse feature project contains *features*. A feature has a name, version number, a software license and contains a list of plug-ins and other features which can be understood as a logical unit.

A feature is described via a `feature.xml` file in the Eclipse feature project.

Features are 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. `org.eclipse.rcp` for Eclipse 3.x based RCP applications or `org.eclipse.e4.rcp` for Eclipse 4 based RCP applications.

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

The `feature.xml` file is a simple text file. For example the file might look like the following in a text editor.

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="de.vogella.featuretest.feature"
      label="Feature"
      version="1.0.0.qualifier">

   <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="de.vogella.featuretest.testplugin"
         download-size="0"
         install-size="0"
         version="0.0.0"
         unpack="false"/>

</feature>
```

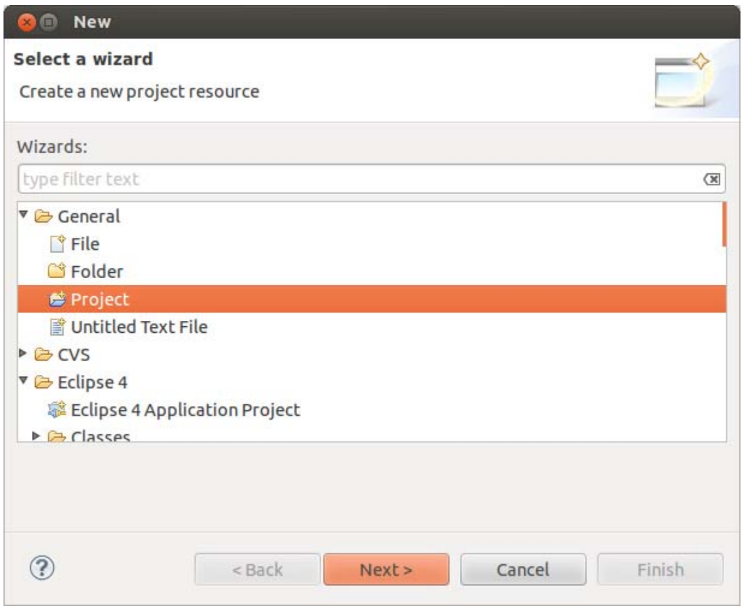
11. Tutorial: From Plug-in to Eclipse 4 application

application

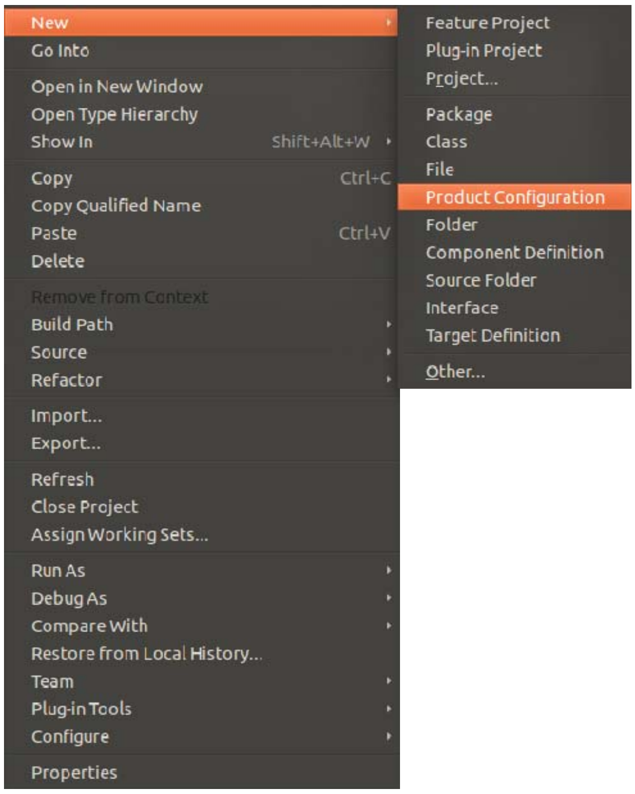
In this chapter we convert the Eclipse plug-in into an Eclipse 4 application.

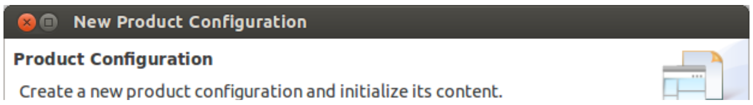
11.1. Create product configuration

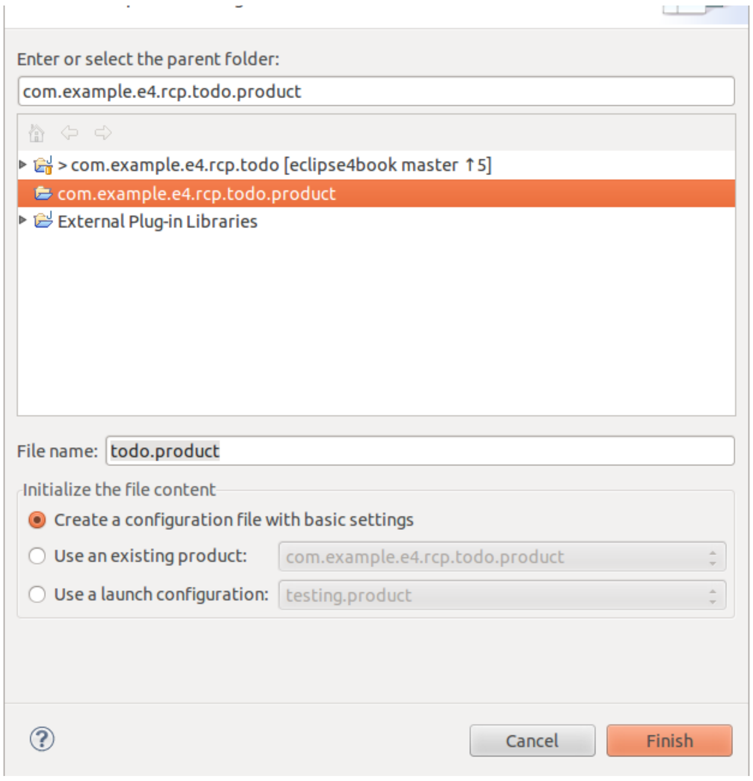
Create a new project called *com.example.e4.rcp.todo.product* of type *General → Project*.



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




New Product Configuration
Product Configuration
Create a new product configuration and initialize its content.



Enter or select the parent folder:

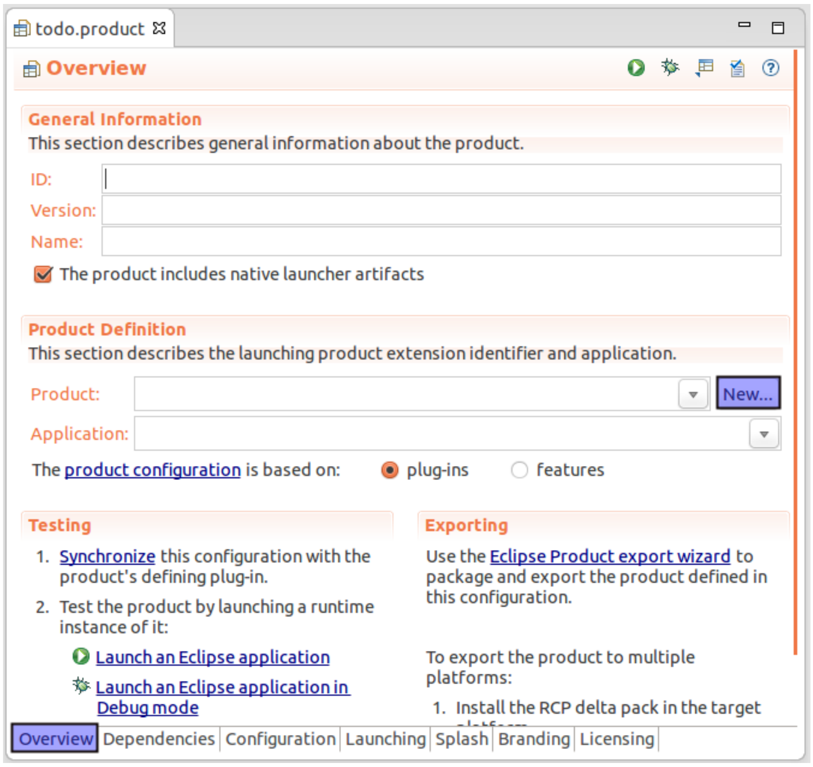
com.example.e4.rcp.todo.product

> com.example.e4.rcp.todo [eclipse4book master ↑5]
com.example.e4.rcp.todo.product
External Plug-in Libraries

File name: todo.product

Initialize the file content
[✓ Create a configuration file with basic settings]
Use an existing product: com.example.e4.rcp.todo.product
Use a launch configuration: testing.product

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



todo.product

Overview

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.

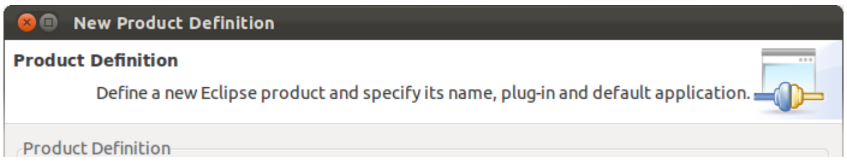
Product:
Application:
The product configuration is based on: [✓ plug-ins] features

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
   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

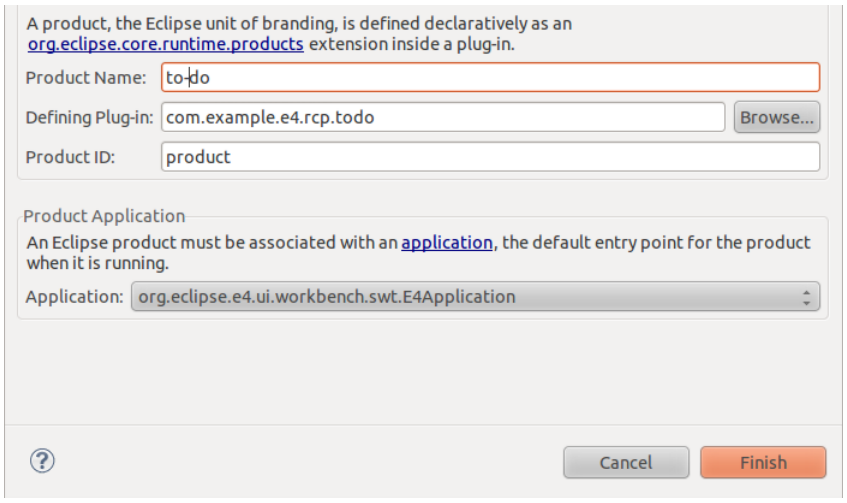
Enter *to-do* as the name, the defining plug-in is your plug-in and use the name *product* as the ID. Select as *Application* the `E4Application` application class.



New Product Definition

Product Definition
Define a new Eclipse product and specify its name, plug-in and default application.

Product Definition


A product, the Eclipse unit of branding, is defined declaratively as an org.eclipse.core.runtime.products extension inside a plug-in.

Product Name: todo

Defining Plug-in: com.example.e4.rcp.todo

Product ID: product

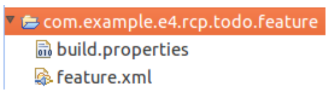
Product Application

An Eclipse product must be associated with an application, the default entry point for the product when it is running.

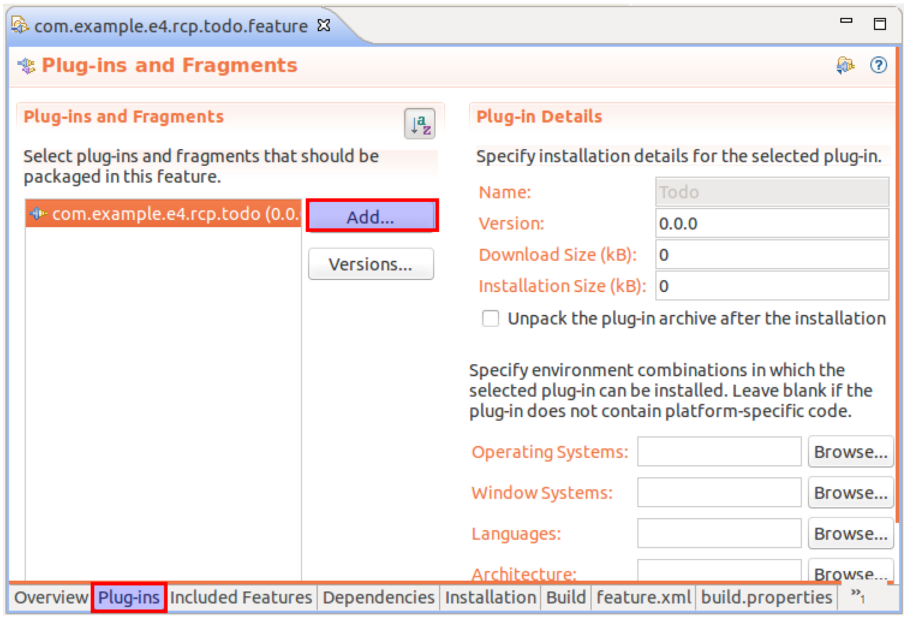
Application: org.eclipse.e4.ui.workbench.swt.E4Application

11.2. Create a feature project

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


```
com.example.e4.rcp.todo.feature
  build.properties
  feature.xml
```

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


com.example.e4.rcp.todo.feature

Plug-ins and Fragments

Plug-ins and Fragments

Select plug-ins and fragments that should be packaged in this feature.

com.example.e4.rcp.todo (0.0.

Plug-in Details

Specify installation details for the selected plug-in.

Name:	Todo
Version:	0.0.0
Download Size (kB):	0
Installation Size (kB):	0

Unpack the plug-in archive after the installation

Specify environment combinations in which the selected plug-in can be installed. Leave blank if the plug-in does not contain platform-specific code.

Operating Systems:

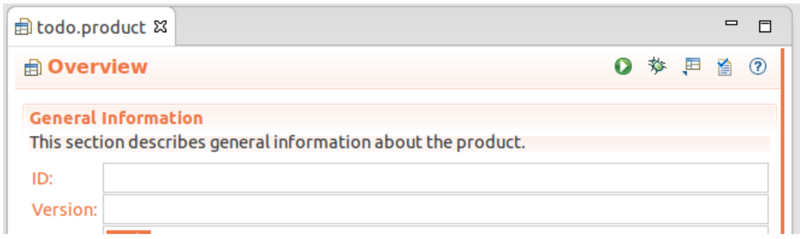
Window Systems:

Languages:

Architecture:

11.3. Enter feature dependencies in product

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


todo.product

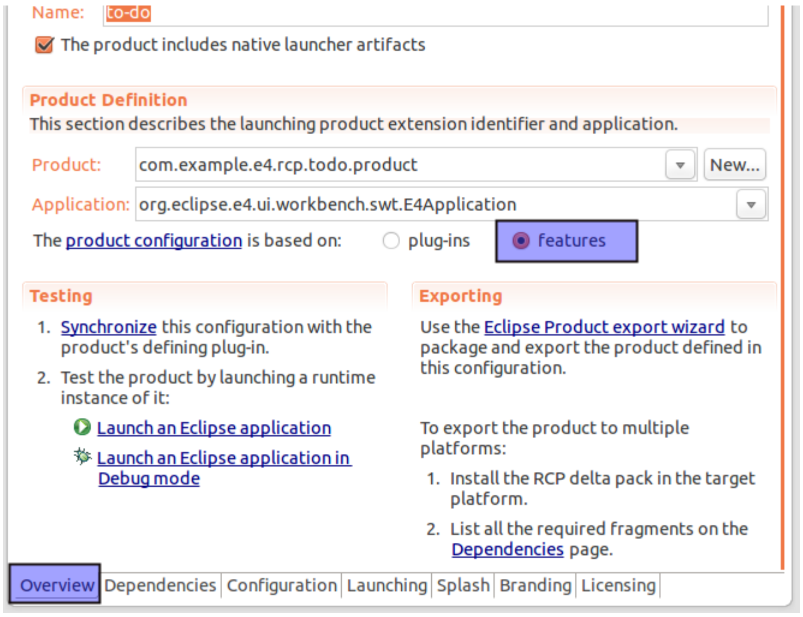
Overview

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

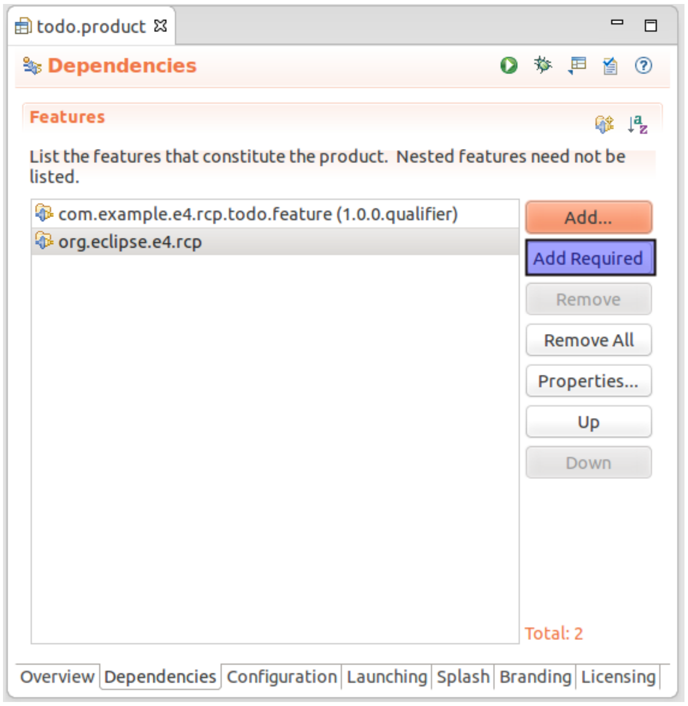
Exporting

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.

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


todo.product

Dependencies

Features

List the features that constitute the product. Nested features need not be listed.

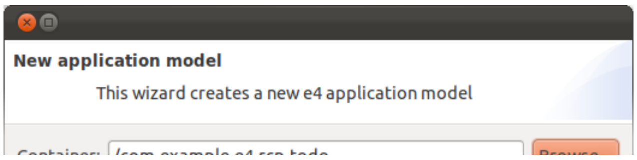
com.example.e4.rcp.todo.feature (1.0.0.qualifier)
org.eclipse.e4.rcp

Total: 2

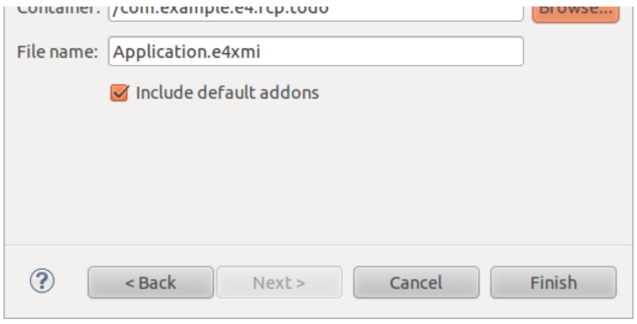
Press the *Add Required* button. This will add the `org.eclipse.emf.common` and `org.eclipse.emf.ecore` features to the dependencies.

11.4. Create Application model

Select *File → New → Other → Eclipse 4 → Model → New Application Model* to open a wizard. Use the project as the container and the filename suggested by the wizard.


New application model

This wizard creates a new e4 application model

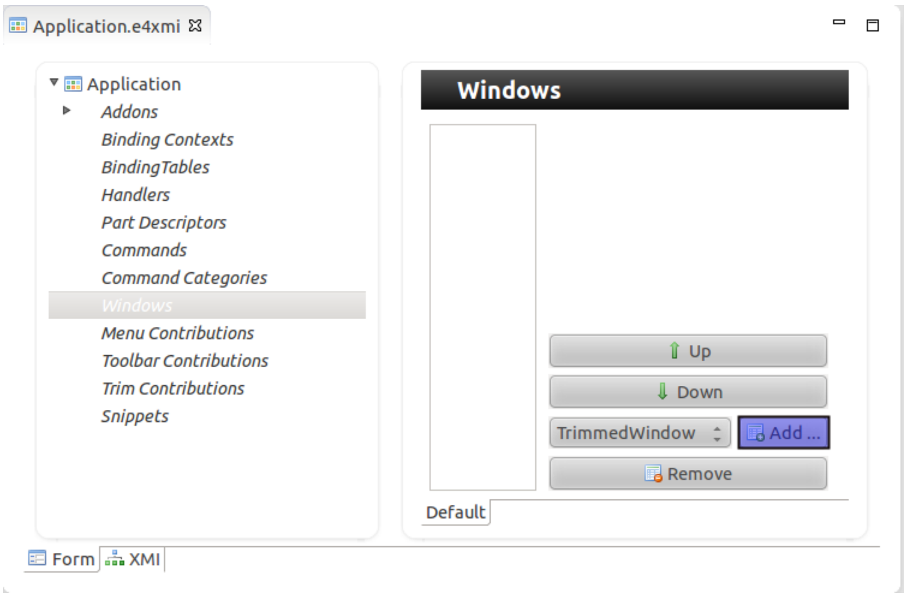

Container: /com.example.e4.rcp.todo
File name: Application.e4xmi
Include default addons

This will create the `Application.e4xmi` file and open this file with the application model editor.

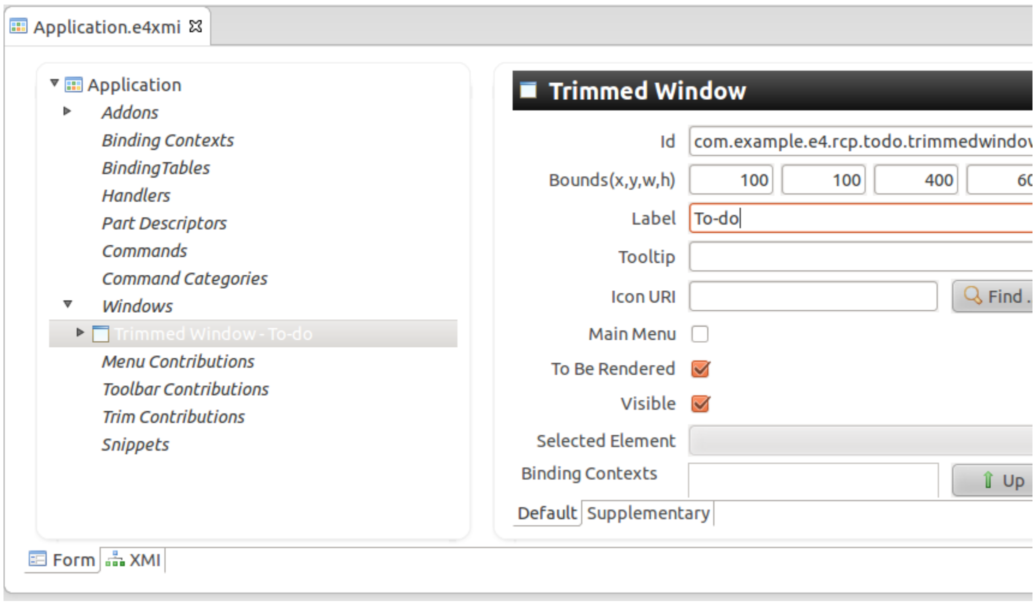
11.5. Add model elements to the application model

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

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



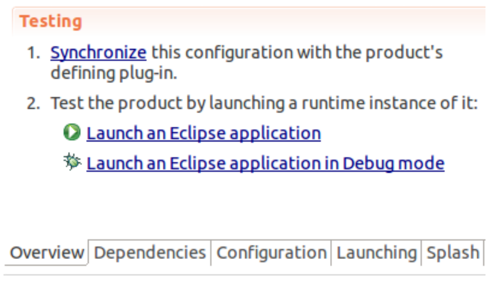
Enter an ID, the position and size of the window and a label as shown in the screenshot below.



Id: com.example.e4.rcp.todo.trimmedwindo
Bounds(x,y,w,h): 100 100 400 60
Label: To-do
Tooltip:
Icon URI:
Main Menu: ☐
To Be Rendered: ✓
Visible: ✓
Selected Element:
Binding Contexts:

11.6. Start application

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



Testing

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

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

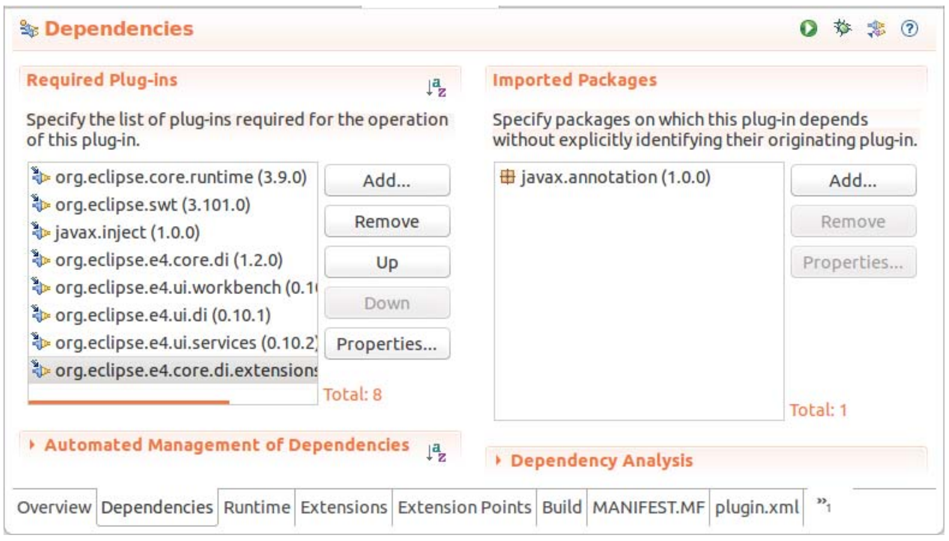
11.7. Add plug-in dependencies

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

Open your *META-INF/MANIFEST.MF* file 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
- javax.inject
- org.eclipse.e4.core.di
- org.eclipse.e4.ui.workbench
- org.eclipse.e4.ui.di
- org.eclipse.e4.ui.services
- org.eclipse.e4.core.di.extensions

Also add the `javax.annotation` package as package dependency.



Dependencies

Required Plug-ins

Specify the list of plug-ins required for the operation of this plug-in.

- org.eclipse.core.runtime (3.9.0)
- org.eclipse.swt (3.101.0)
- javax.inject (1.0.0)
- org.eclipse.e4.core.di (1.2.0)
- org.eclipse.e4.ui.workbench (0.1
- org.eclipse.e4.ui.di (0.10.1)
- org.eclipse.e4.ui.services (0.10.2)
- org.eclipse.e4.core.di.extension

Total: 8

Imported Packages

Specify packages on which this plug-in depends without explicitly identifying their originating plug-in.

- javax.annotation (1.0.0)

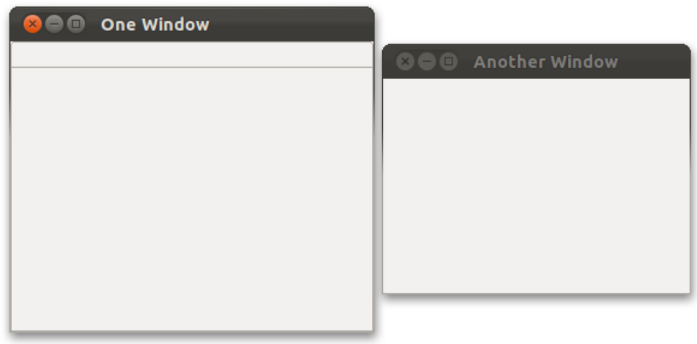
Total: 1

12. Important user interface model elements

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

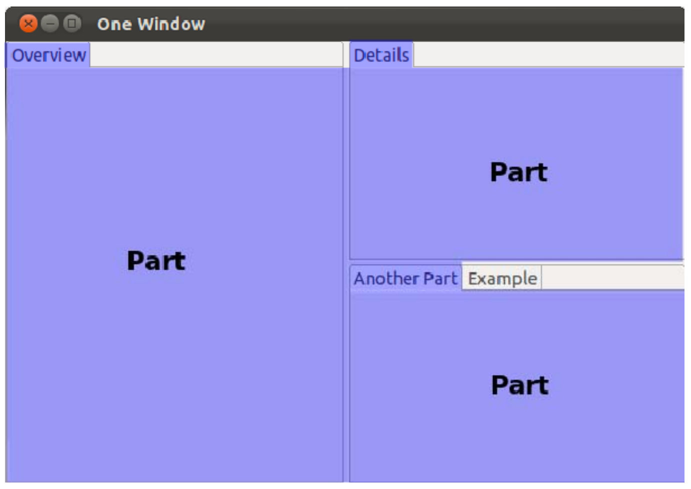
12.1. Windows

Eclipse applications consist of one or more *Windows*. Typically an application has only one *Window* but you are not limited to that, e.g. if you want to support multiple displays for two connected monitors.



12.2. Parts

Parts are user interface components which allow you to navigate and modify data. *Parts* are typically divided into *Views* and *Editors*.



The distinction into *Views* and *Editors* is primarily not based on technical differences, but on a different concept of using and arranging these *Parts*.

A *View* is typically used to work on a set of data, which might be a hierarchical structure. If data is changed via the *View*, this change is typically directly applied to the underlying data structure. A *View* sometimes allows us to open an *Editor* for a selected set of data.

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

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

Editors were traditionally placed in a certain area, called the *editor area*. Until Eclipse 4 this was a hard limitation. Eclipse 4 applies no technical restrictions on *Editors*, they can be freely positioned

positioned.

For example the *Java Editor* is used to modify Java source files. Changes to the source file are applied once the user selects the *Save* command.

12.3. Perspective

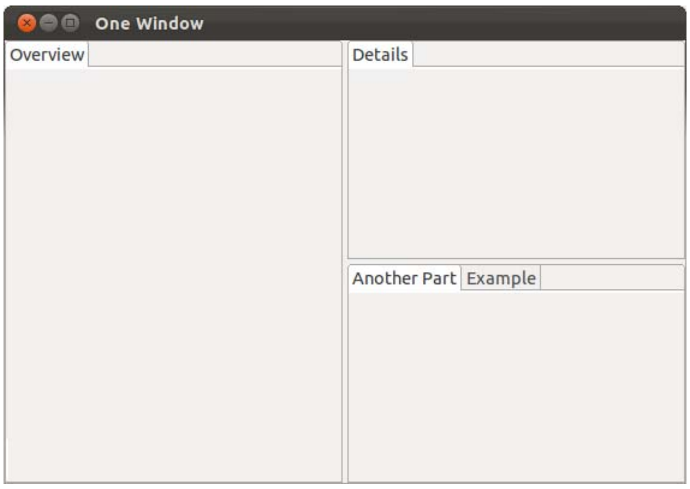
A *Perspective* is a visual container for a set of *Parts*. You can switch *Perspectives* in your Eclipse RCP application via the the *EPartService* service.

12.4. PartStacks and PartSashContainers

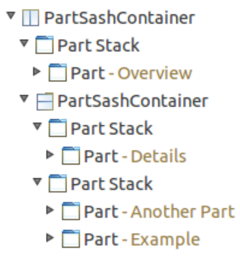
Parts can be directly assigned to a *Window* or a *Perspective*. If you want to group and arrange *Parts* you can use *PartStacks* and *PartSashContainers*.

PartStacks contain a stack of *Parts* of which only one is visible at the same time and can be selected via tabs. A *PartSashContainer* displays all its children at the same time either horizontally or vertically.

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



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



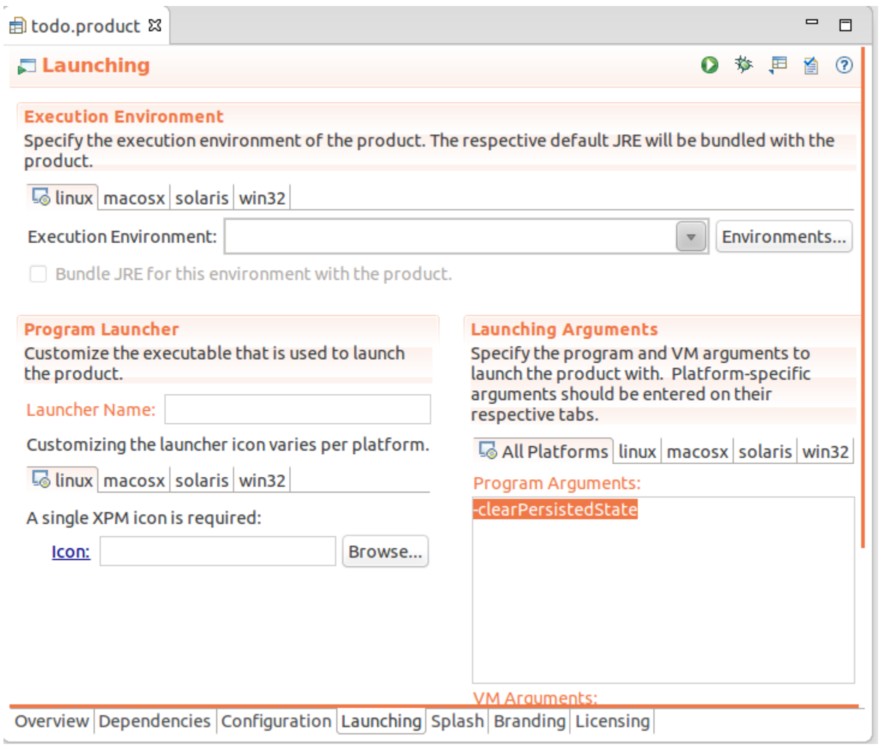
```
▼ PartSashContainer
  ▼ Part Stack
    ▸ Part - Overview
  ▼ PartSashContainer
    ▼ Part Stack
      ▸ Part - Details
    ▼ Part Stack
      ▸ Part - Another Part
      ▸ Part - Example
```

13. Configure the deletion of persisted model data

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

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

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


todo.product

Launching

Execution Environment
Specify the execution environment of the product. The respective default JRE will be bundled with the product.

linux | macosx | solaris | win32

Execution Environment: Environments...

Bundle JRE for this environment with the product.

Program Launcher
Customize the executable that is used to launch the product.

Launcher Name:

Customizing the launcher icon varies per platform.

linux | macosx | solaris | win32

A single XPM icon is required:

Icon: Browse...

Launching Arguments
Specify the program and VM arguments to launch the product with. Platform-specific arguments should be entered on their respective tabs.

All Platforms | linux | macosx | solaris | win32

Program Arguments:
-clearPersistedState

VM Arguments:

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

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

14. Tutorial: Modeling a User Interface

14.1. Create modeled user interface

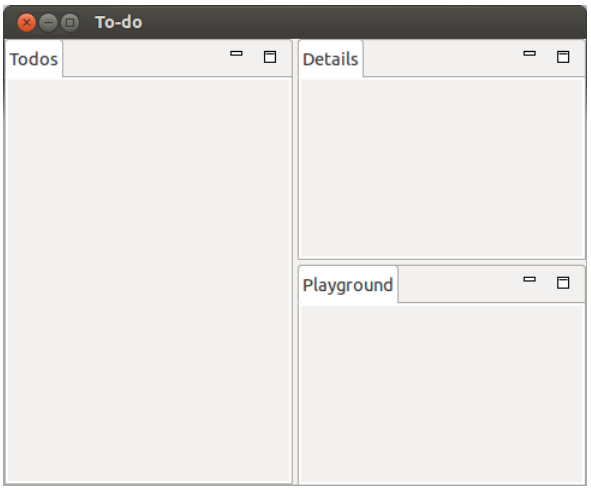
You will extend the modeled user interface of your Eclipse 4 application.

You will add a *Perspective* to the Window. This *Perspective* will contain a *PartSashContainer*. This container will divide its children horizontally.

The *PartSashContainer* will contain a *PartStack* and another *PartSashContainer*. This second *PartSashContainer* will contain two *PartStacks* and divide them vertically.

Each *PartStacks* will contain one *Part*.

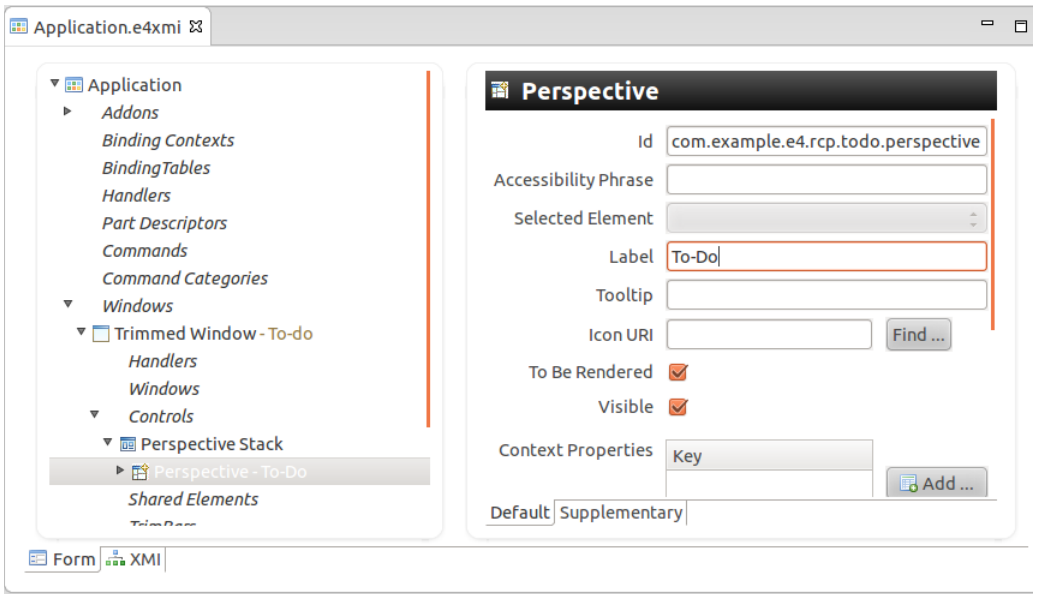
After these changes, your user interface should look like the following screenshot.



14.2. Adding model elements

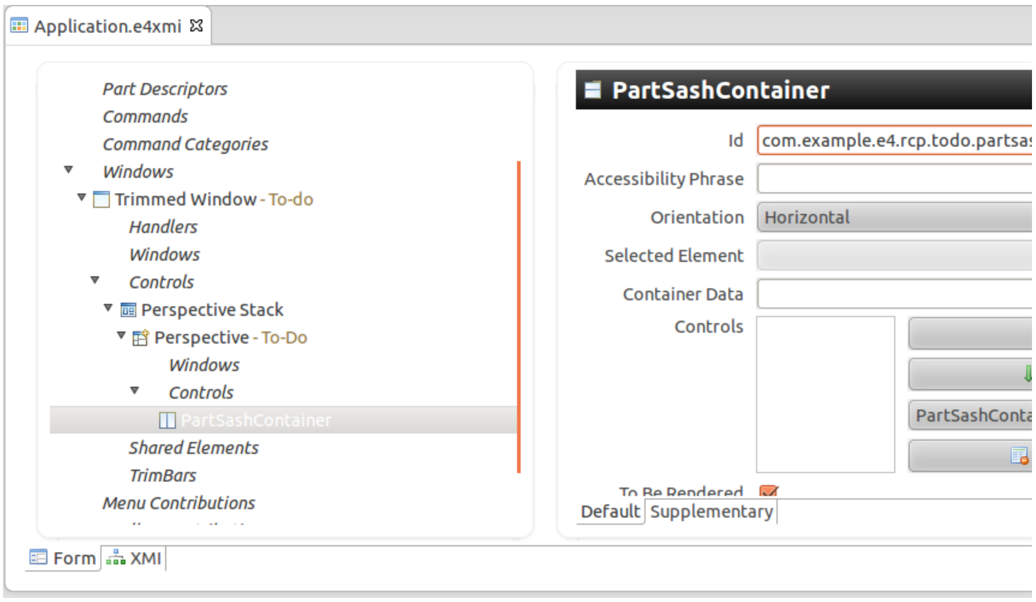
Open the `Application.e4xmi` file. Go to your *To-do* Window and select the *Controls* node. Add a *PerspectiveStack*. Press the *Add* button to create a *Perspective* entry.

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



Id	com.example.e4.rcp.todo.perspective
Accessibility Phrase	
Selected Element	
Label	To-Do
Tooltip	
Icon URI	
To Be Rendered	✓
Visible	✓
Context Properties	Key

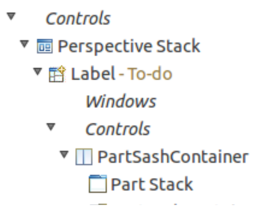
Select *Controls* below the newly created *Perspective* and add a *PartSashContainer*. Change its *Orientation* attribute to *Horizontal*.

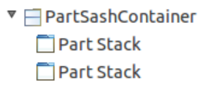


Id	com.example.e4.rcp.todo.partsa
Accessibility Phrase	
Orientation	Horizontal
Selected Element	
Container Data	
Controls	
To Be Rendered	✓

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

Re-select the parent *PartSashContainer* and add another *PartSashContainer*. Now add two *PartStacks* to the second *PartSashContainer*. This should result in a tree in your application model similar to the following screenshot.



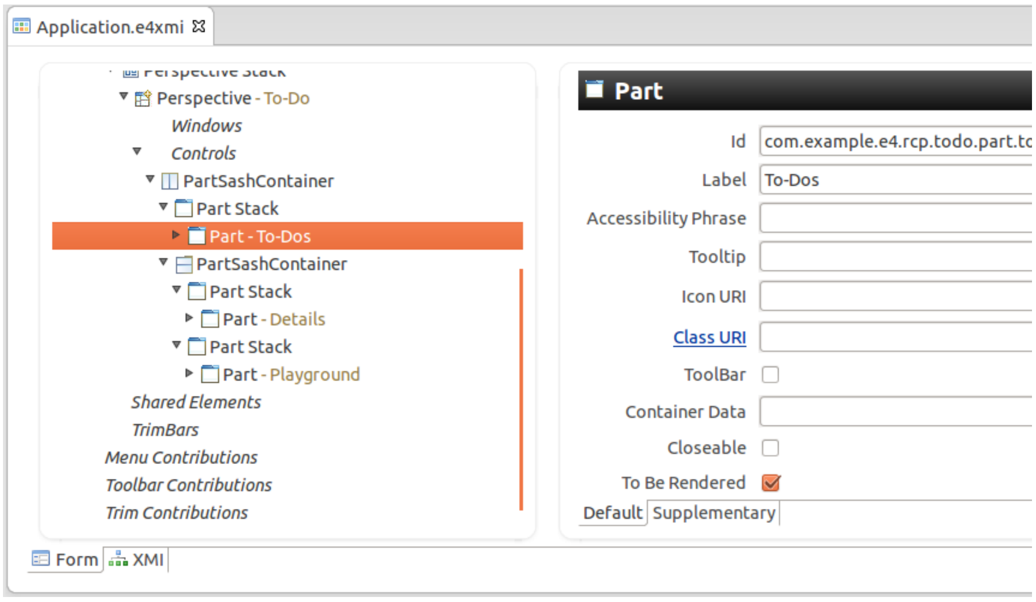


Add a *Part* to each *PartStack*. As ID for the *Parts* use the prefix `com.example.e4.rcp.todo.part` and the suffix from the following table.

Table 4. Label and ID from the Parts

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

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



Id: com.example.e4.rcp.todo.part.to
Label: To-Dos
Accessibility Phrase:
Tooltip:
Icon URI:
Class URI:
ToolBar: ☐
Container Data:
Closeable: ☐
To Be Rendered: ✓

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

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

14.3. Create Java classes and connect to the model

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

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

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

The following code shows the `TodoDetailsPart` class.

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

public class TodoDetailsPart {

}
```

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

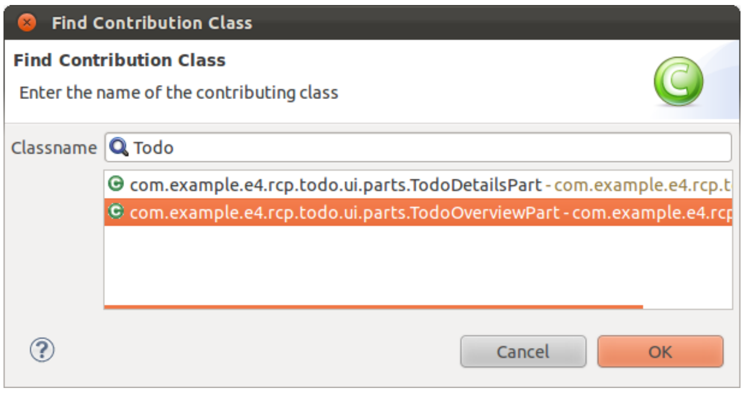
The following table gives an overview of which elements should be connected.

Table 5. Connecting Java classes with Model Element

Table 3. Connecting Java classes with Model Element

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

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


Find Contribution Class

Find Contribution Class

Enter the name of the contributing class

Classname: Todo

com.example.e4.rcp.todo.ui.parts.TodoDetailsPart - com.example.e4.rcp.t

com.example.e4.rcp.todo.ui.parts.TodoOverviewPart - com.example.e4.rcp

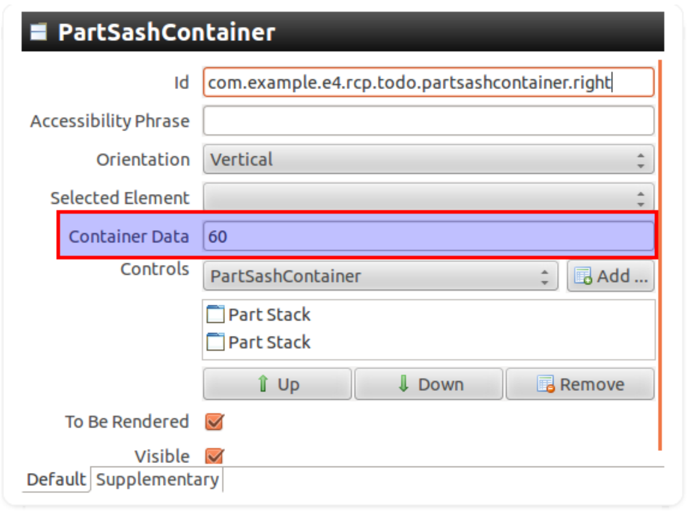
14.4. Test

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

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

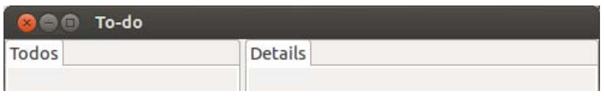
14.5. Layout

Use the model *`Container Data`* attribute to assign a layout weight of 40 to the left container and 60 to the right container.


PartSashContainer

Id	com.example.e4.rcp.todo.partsashcontainer.right
Accessibility Phrase	
Orientation	Vertical
Selected Element	
Container Data	60
Controls	PartSashContainer
	Part Stack
	Part Stack
To Be Rendered	✓
Visible	✓

As a result the left container should use 40 % of the avaialable space.



15. Dependency Injection and Annotations

15.1. What is Dependency Injection?

The general concept behind dependency injection is called *Inversion of Control* . A class should not configure its dependencies but should be configured from outside.

Dependency injection is a concept which is not limited to Java. But we will look at dependency injection from a Java point of view.

A Java class has a dependency on another class if it uses an instance of this class, e.g. via calling the constructor or via a static method call. 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, for example for unit testing.

If the Java class directly creates an instance of another class via the `new` operator, it cannot be used and tested independently from this class.

To decouple Java classes its dependencies should be fulfilled from the outside. A Java class would simply define its requirements like in the following example:

```
public class MyPart {

  @Inject private Logger logger;
  // DatabaseAccessClass would talk to the DB
  @Inject private DatabaseAccessClass dao;

  @Inject
  public void init(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());
  }

}
```

Another class could read these dependencies and create an instance of the class, injecting objects into the defined dependency. This can be done via the Java reflection functionality. This class is usually called the dependency container and is a framework class.

This way the Java class has no hard dependencies, i.e. it does not rely on an instance of a certain class. For example if you want to test a class which uses another object which directly uses a database, you could inject a *mock* object.

Mock objects are objects which act as if they are the real object but only simulate their behavior. Mock is an old English word meaning to mimic or imitate.

If dependency injection is used, a Java class can be tested in isolation, which is good.

Dependency injection can happen on:

- the constructor of the class (construction injection)
- a method (method injection)

- a field (field injection)

Dependency injection can happen on static as well as on non-static fields and methods.

15.2. Define dependencies in Eclipse

Eclipse 4 supports constructor, method and field injection. It uses the standard Java `@Inject` and `@Named` annotations, which were defined in the Java Specification Request 330 (JSR330). In addition to these standard annotations, it declares the `@Optional` and `@Preference` annotations.

The following table gives an overview of the dependency injection (DI) annotations.

Table 6. Annotations for Dependency Injection

Annotation	Description
@javax.inject.Inject	Marks a field, a constructor or a method. The Eclipse framework tries to inject the parameter.
@javax.inject.Named	Defines the name of the key for the value which should be injected. By default the fully qualified class name is used as key. Several default values are defined as constants in the `IServiceConstants` interface.
@Optional	Marks an injected value to be optional. If it can not be resolved, `null` is injected. Without @Optional the framework would throw an Exception. The specific behavior depends where @Optional is used: • for parameters: a `null` value will be injected; • for methods: the method calls will be skipped • for fields: the values will not be injected.
@Preference	Eclipse can store key/values pairs as so-called preferences. The @Preference annotation defines that the annotated value should be filled from the preference store.

The `org.eclipse.e4.core.di.annotations` package contains the `@Optional` annotation and the `@Preference` annotation is included in the `org.eclipse.e4.core.di.extensions` package. Both are part of the `org.eclipse.e4.core.di` plug-in.

15.3. Dependency injection in Eclipse 4

The Eclipse 4 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. First the constructor injection is performed and afterwards the field and method injections.

Constructors are called before the fields are injected. Accessing an injected field in the constructor will result in an Exception thrown by the Framework.

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

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

16. Scope of injection

16.1. What can be injected?

We covered how dependency injection works and which annotations can be used to define the behavior of the Java classes. But we have not yet covered what the scope of injection is, i.e. what you can inject into your model classes.

The Eclipse runtime creates a context in which the possible values for injection can be stored.

This context can be modified, e.g. the application and the framework can add elements to the context.

The Eclipse context contains:

- all objects associated with the application model
- all other objects which have explicitly been added to the context
- all Preferences - key/value pairs which typically used to configure the application
- OSGi services - software components which can be dynamically consumed

Applications can define their own injection suppliers too.

For the purpose of this tutorial, all elements of the context will be called *context elements*.

16.2. Key/values pairs and Context Variables

Objects can be places in the context via the Class name or via a String. If the key is a String, this key/value pair is called a *Context variable*.

Key/value pairs or *Context variables* can be present at different levels in the context hierarchy.

Context variables and key/value pairs in the context allow to separate the interesting state from the source of the state. For example in the Eclipse IDE a lot of components use the active editor to drive their views, but it means their views are coupled to the active editor; the editor could instead set a context variable to their state and the views react to that variable change instead.

The Eclipse platform places several key/value pairs and *Context variables* into the context. Several *Context variables* keys are defined in the `IServiceConstants` interface.

16.3. How are objects searched?

All possible values for dependency injection can be accessed via the context.

The context contains Java objects which can be accessed by keys. Keys can manifest themselves in two forms: Strings and Class objects.

If a String is used as a key, the corresponding key/value pair is called a *Context Variable*.

Access to the context works in a similar way to accessing a Java `Map` data structure.

The context is not a flat structure like a `Map`. It is hierarchical and can also dynamically compute values for requested keys.

A context can be local to an object and can have a parent context.

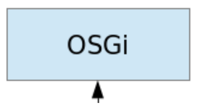
Model elements which implements the `MContext` interface have a local context. These are currently `MApplication`, `MWindow`, `MPerspective`, `MPart` and `MPopupMenu`.

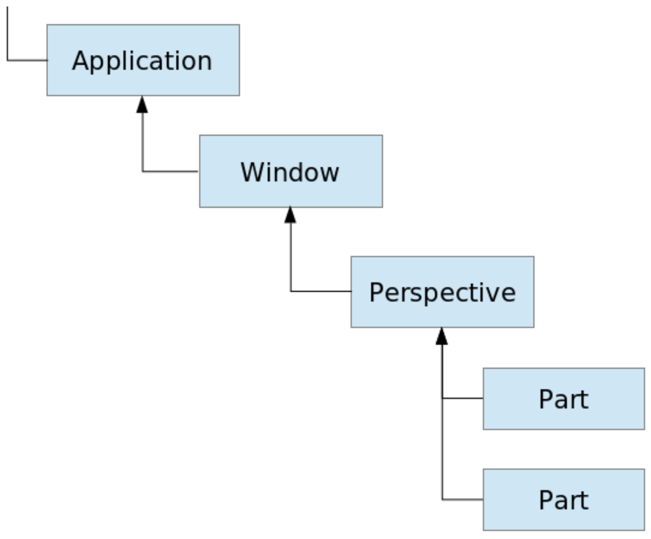
The main context is created by the Eclipse Framework and all context objects of the model elements are hierarchically connected to the main context object.

Model objects that implement the `MContext` interface have their own local context. This is for example the case for `MWindow` and `MPart`. Each context for a model element contains a reference to the Model object itself.

If for example a *Part* requests an object by a certain key from the context, Eclipse will first search for this object in the local context of the part. If it does not find the key in the local context it will search the parent context. This process continues until the main context has been reached. At this point the framework would check for fitting OSGi services in the OSGi registry.

The following picture visualizes this context hierarchy.





The context is hierarchical to avoid collisions in injected values and to increase the isolation of the classes in terms of what they need. For example a *Part* needs a unique `Composite` objects to create its user interfaces. Since *Parts* have different local contexts, their `@Inject` gets the needed implementation from the actual context.

The search happens transparently for the caller of the injection.

16.4. What are the relevant classes and interfaces

The interface for the context object is the `IEclipseContext` interface.

The Eclipse platform uses the `OSGiContextStrategy` class to search for OSGi services if the Eclipse framework does not find the requested key in the hierarchy of `IEclipseContext` objects.

16.5. Who creates the context for model elements?

Eclipse 4 has a flexible renderer framework. For each model element the framework determines a renderer class which is responsible for creating the Java object associated with the model element. This renderer class creates, if required, the local context for the model element via the `EclipseContextFactory` class and connects this local context to the context hierarchy.

For example the `ContributedPartRenderer` class is responsible for creating the Java objects for *Parts* in the model. The interface for *Parts* is `MPart`.

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

Another example is the `WBWRenderer` class which is responsible for creating a *Window*. This class puts an instance of the `IWindowCloseHandler` and the `ISaveHandler` interface into the context of the *Window*. The first is responsible for the behavior of a *Window* during close, the other one for saving. For example the default `IWindowCloseHandler` would prompt you if you want to save *Parts* which indicate that they have saveable content via the `MDirtyable` model attribute. You can change this default `IWindowCloseHandler` implementation via the `MWindow` model object. The following example shows this.

```
@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);
}
```

17. Behavior Annotations

17.1. API definition via inheritance

In general, every framework defines an application programming interface (API).

If you use a framework you need to have a convention for which methods are called at which point of the execution of your program. For example if a Java class is responsible for handling a toolbar button click, the framework needs to know which method of this class it should call.

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

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

For example in Eclipse 3.x a *View* class would extend the abstract `ViewPart` class. This class defines the `createPartControl()` method.

The Eclipse 3.x framework knows that `createPartControl()` is responsible for creating the user interface and calls this method once the *View* becomes visible.

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.

17.2. API definition via annotations

Eclipse 4 does not require that classes extend framework classes. To be precise Eclipse 4 does not provide any such classes.

Eclipse 4 uses annotations to indicate that a certain behavior is expected from the framework.

I will call these annotations *behavior annotations*.

Behavior annotations are used to indicate that certain methods should be called at certain events. The following tables list the available behavior annotations.

Table 7. Eclipse lifecycle annotations for Parts

Annotation	Description
@PostConstruct	Is called after the class is constructed and the field and method injection has been performed.
@PreDestroy	Is called before the class is destroyed. Can be used to clean up resources.
@Focus	Indicates that this method should be called, once the Part gets the focus. It is ***required*** to set the focus on one user interface control otherwise certain workbench functionality does not work.
@Persist	Is called if a save request on the Part is triggered. Can be used to save the data of the Part.
@PersistState	Is called before the model object is disposed, so that the Part can save its state.

Table 8. Other Eclipse Behavior Annotations

Annotation	Description
@Execute	Marks a method in a handler class to be executed. Only one method in a handler should be annotated with @Execute.
@CanExecute	Marks a method to be visited by the Command- Framework to check if a handler is enabled

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

All these annotations will also trigger dependency injections. Therefore you do not need to add the `@Inject` annotation if you use these annotations.

The `@PostConstruct`, `@PreDestroy` annotations are included in the `javax.inject` package.

The `org.eclipse.e4.core.di.annotations` package contains the `@Execute`, `@CanExecute` annotations.

`@Persists`, `@PersistState` and `@Focus` are part of the `org.eclipse.e4.ui.di` package.

17.3. Lifecycle Hooks

Typically some functionality should be triggered if an application is started or stopped. Eclipse 4 allows you to register a class to predefined events of this lifecycle. For example you can use these lifecycle hooks to create a login screen before the actual application is started.

The `org.eclipse.core.runtime.product` extension point allows you to define this lifecycle class with a property with the `lifeCycleURI` key.

This property points to a class via the `bundleclass://` schema.

In the lifecycle class you can use the following annotations. The annotated methods will be called by the framework depending on the life cycle of your application.

Table 9. Life Cycle Annotations

Annotation	Description
@PostContextCreate	Is called after the Application's `IEclipseContext` is created, can be used to add objects, services, etc. to the context. This context is created for the `MApplication` class.
@ProcessAdditions	Is called directly before the model is passed to the renderer, can be used to add additional elements to the model.
@ProcessRemovals	Same as @ProcessAdditions but for removals.
@PreSave	Is called before the application model is saved. You can modify the model before it is persisted.

The following example shows how to register a lifecycle class in your *plugin.xml* file.

```
<?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 coding for the lifecycle class will display a shell.

```
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.swt.SWT;
import org.eclipse.swt.widgets.Shell;
import org.osgi.service.event.Event;
import org.osgi.service.event.EventHandler;

public class LifeCycleManager {
  @PostContextCreate
  void postContextCreate() {
    final Shell shell = new Shell(SWT.TOOL | SWT.NO_TRIM);
    shell.open();
  }

}
```

18. Tutorial: Using dependency injection

18.1. Getting a Composite

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

Change the `TodoOverviewPart` class to the following:

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

import javax.inject.Inject;

import org.eclipse.swt.widgets.Composite;

public class TodoOverviewPart {

  @Inject
  public TodoOverviewPart(Composite parent) {

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

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

18.2. Validation

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

19. Exercise: Define Part Lifecycle

19.1. Overview

Although the user interface of a *Part* can be created in the constructor, it is recommended that the user interface should instead be created in a `@PostConstruct` method. When the constructor is fired, no marked fields or methods will have been injected, which can lead to subtle errors.

Realizing the user interface of a *Part* in a `@PostConstruct` requires that `@Inject` methods be aware that the user interface might not yet be created.

In the following tutorial we use the `@PostConstruct` and `@PreDestroy` annotations in our *Parts*.

19.2. Using @PostConstruct and @PreDestroy

Add the following method to your `TodoOverviewPart`, `TodoDetailsPart` and

PlaygroundPart classes.

```
// Declare a field label, required for @Focus
Label label;

public void createControls(Composite parent) {
  label = new Label(parent,SWT.NONE);
  label.setText("A text....");
}
```

Annotate the method with @PostContruct.

Create an empty public void dispose() method and annotate it with @PreDestroy.

Remove all constructors from your classes.

19.3. @Focus

Implement a @Focus method in each *Part*. Setting the focus to one of SWT controls is mandatory otherwise the Eclipse services do not work reliably.

```
@Focus
private void setFocus() {
  label.setFocus();
}
```

19.4. Validate

Run your application and validate that the @PostContruct method is called. Use either debugging or a System.out.println() statement. Every Part needs to assign focus to one of its controls, otherwise certain services will not work. Therefore we added a SWT label to be the focus control. Even though a label cannot take focus, it triggers the SWT.ACTIVATE event, which is sufficient for the Eclipse services to work correctly.

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

20. Commands, Handlers, Menus, Toolbars and Popups

20.1. Overview

The Eclipse application model can contain commands and handlers.

A command in Eclipse 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 defines a class via the contributionURI attribute of the handler. This attribute is displayed as "Class URI" in the model editor. The handler can be global to the application, or scoped to a window or even a part.

This class uses the @Execute annotation to define which method is called once the handler is executed. The @CanExecute annotation defines the method which evaluates if the handler is currently active. If the handler can always execute, it does not need to supply a @CanExecute method.

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

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

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

20.2. Default commands

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

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

20.3. Menus and Toolbar

You can add menus and toolbars to the application model for the Window and for Parts. Menu and toolbars items contain references to commands. If a command is selected, the runtime will determine the relevant handlers for the command.

For simple cases you can also use the "Direct MenuItem" or a "Direct ToolItem", which allows you to define a class to be executed directly. This is useful if you just want to respond to the user clicking on a menu item or tool item.

Using commands together with handlers gives you more flexibility, for example you can have different Handlers for different scopes (Applications or Views) and you can define key bindings for the handler's associated commands.

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

The related command is assigned to the menu and toolbar entry. Menus and toolbars support separators. Menus can have submenus.

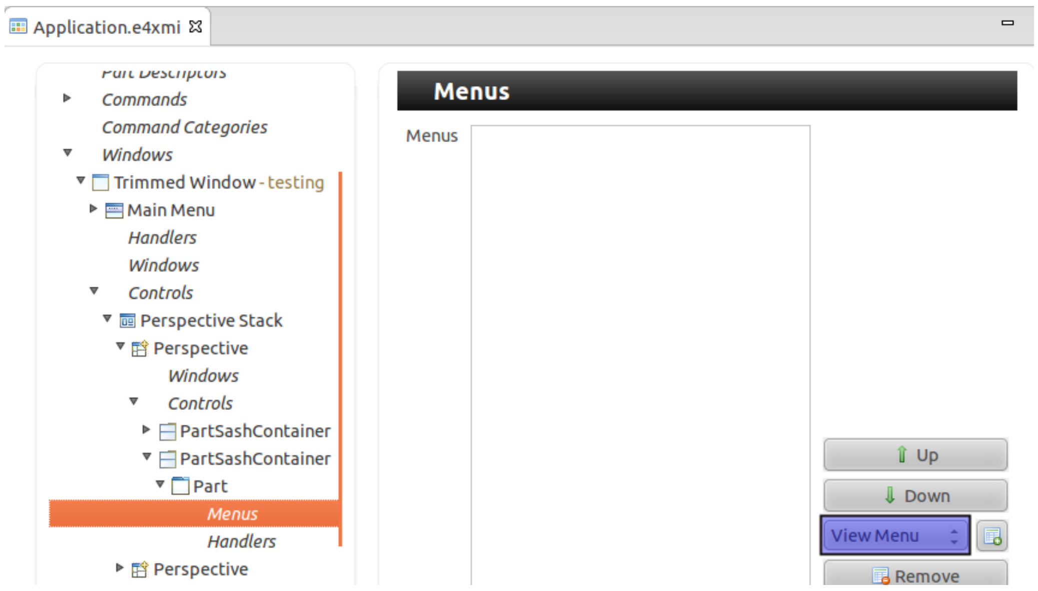
20.4. View Menus

To show a menu for a *View* you have to add a tag to the corresponding entry in the *`Application.e4xmi`* file. This tag must be defined before the menu children are defined and looks like the following.

```
<tags>ViewMenu</tags>
```

Only one Menu in a Part should contain this tag.

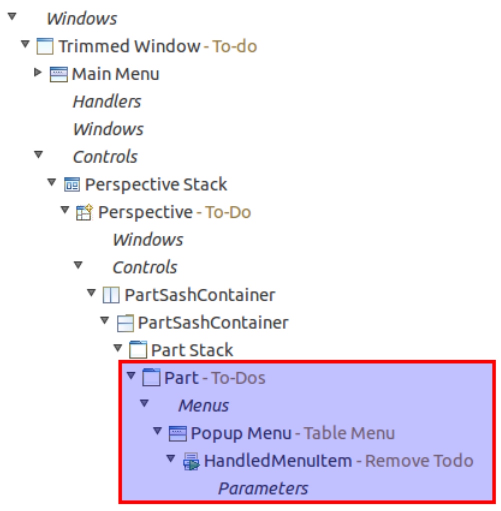
To add such a View menu entry, select the entry *Menu* under our Part, select *ViewMenu* and press the *Add* button.

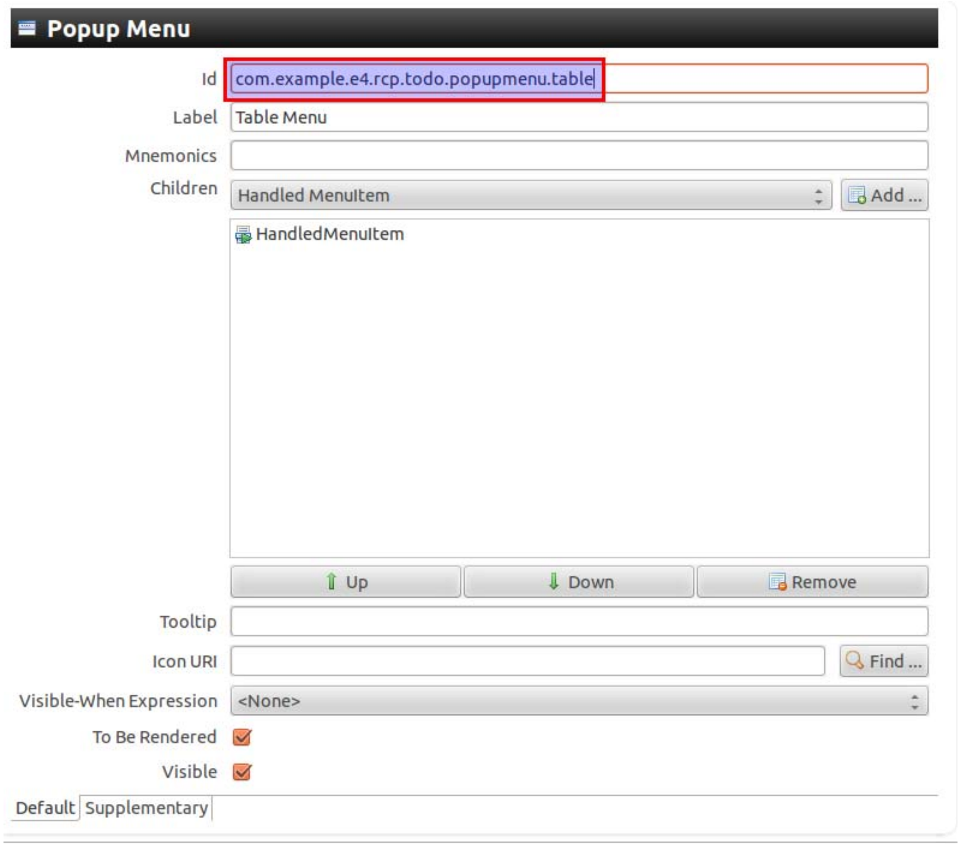


20.5. Popup Menu - Context Menu

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

You can then assign it via the `EMenuService` class to a *SWT* control. The *`ID`* parameter must be the *`elementId`* of your *Popup Menu* model element.





Id	com.example.e4.rcp.todo.popupmenu.table
Label	Table Menu
Mnemonics	
Children	Handled MenuItem
	HandledMenuItem
Tooltip	
Icon URI	
Visible-When Expression	<None>
To Be Rendered	✓
Visible	✓

The following code shows an example for the registration.

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

import javax.annotation.PostConstruct;

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

public class TodoDetailsPart {

  @PostConstruct
  public void createUi(Composite parent, EMenuService service) {
    final Text text = new Text(parent, SWT.BORDER);
    text.setText("Hello");
    // Make use to use the correct ID
    // from the application model
    menuService.registerContextMenu(text,
        "com.example.e4.rcp.todo.popupmenu.table");
  }
}
```

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

20.6. Scope of handlers

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

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

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

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

20.7. Several methods with @Execute

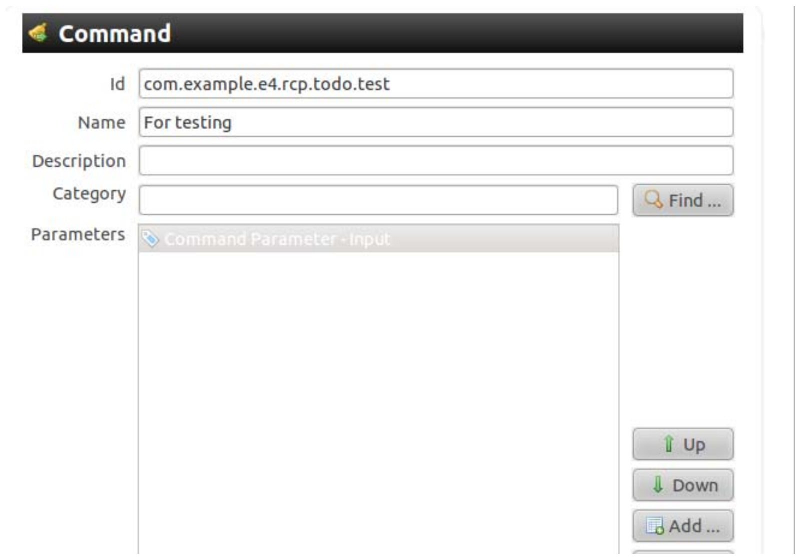
You should only annotate one methods of a handler with `@Execute`. If you annotate several methods the the Eclipse runtime picks one of them.

20.8. Passing parameters to commands

You can also pass parameters to commands.

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

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



Command

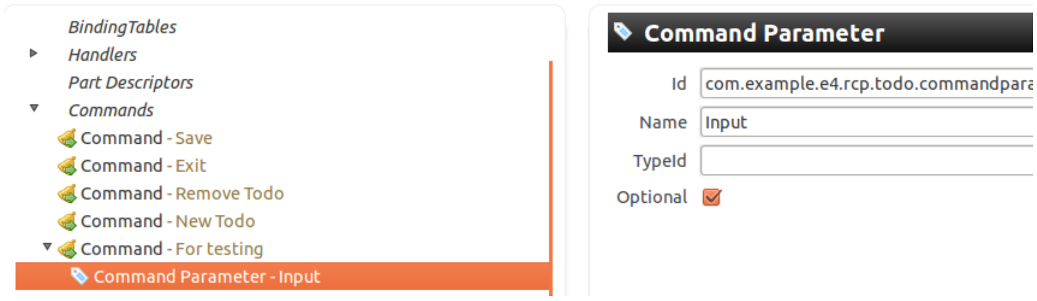
Id: com.example.e4.rcp.todo.test

Name: For testing

Description:

Category:

Parameters: Command Parameter - input

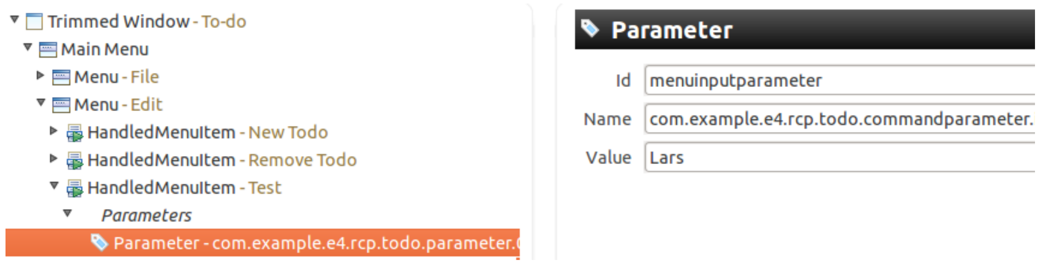


Id: com.example.e4.rcp.todo.commandpara
Name: Input
TypeId:
Optional: ✓

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

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

import javax.inject.Named;

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

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


}
```

In your menu or toolbar entries you use the ID of the command in the *Name* field. The entry in the *Value* field is passed to the command.


Trimmed Window - To-do
Main Menu
Menu - File
Menu - Edit
HandledMenuItem - New Todo
HandledMenuItem - Remove Todo
HandledMenuItem - Test
Parameters
Parameter - com.example.e4.rcp.todo.parameter.
Parameter

Id: menuinputparameter
Name: com.example.e4.rcp.todo.commandparameter.
Value: Lars

20.9. Core expressions

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

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

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

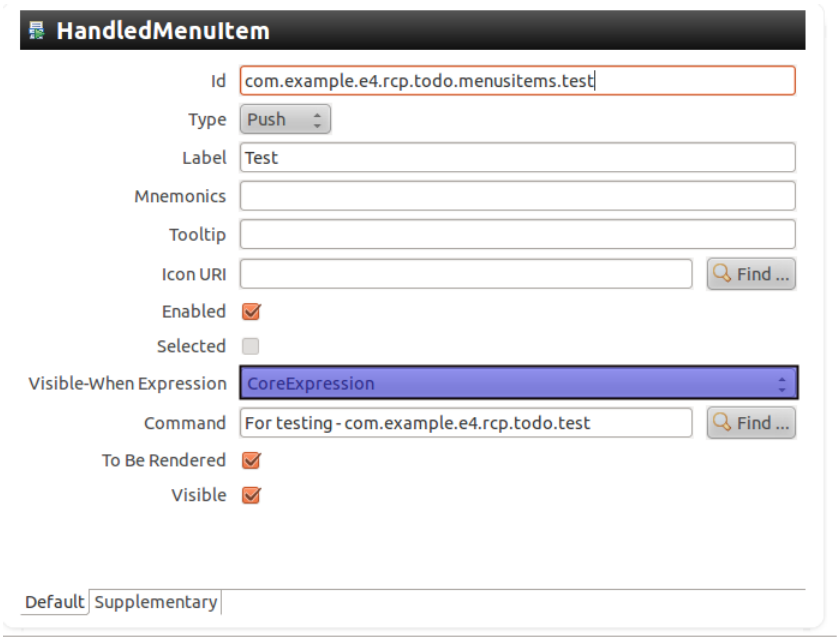
Via right mouse click on the extension you can start building your expression.

You can assign this core expression to your menu entry in the application model. The following screenshot shows this for the *Test* menu entry.

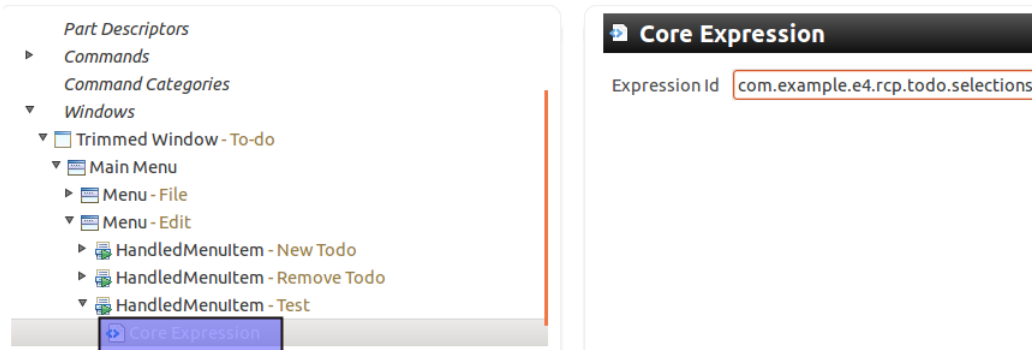
The following example can be used to restrict the visibility of a menu entry based on the type of the current selection. You will later learn how to set the current selection. Please note that the variable for the selection is currently called `org.eclipse.ui.selection`. In Eclipse 3.x this variable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

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

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


HandledMenuItem

Id	com.example.e4.rcp.todo.menusitems.test
Type	Push
Label	Test
Mnemonics	
Tooltip	
Icon URI	
Enabled	✓
Selected	
Visible-When Expression	CoreExpression
Command	For testing - com.example.e4.rcp.todo.test
To Be Rendered	✓
Visible	✓



Core Expression

Expression Id: com.example.e4.rcp.todo.selections

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

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

20.10. Evaluate your own values in Core expressions

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

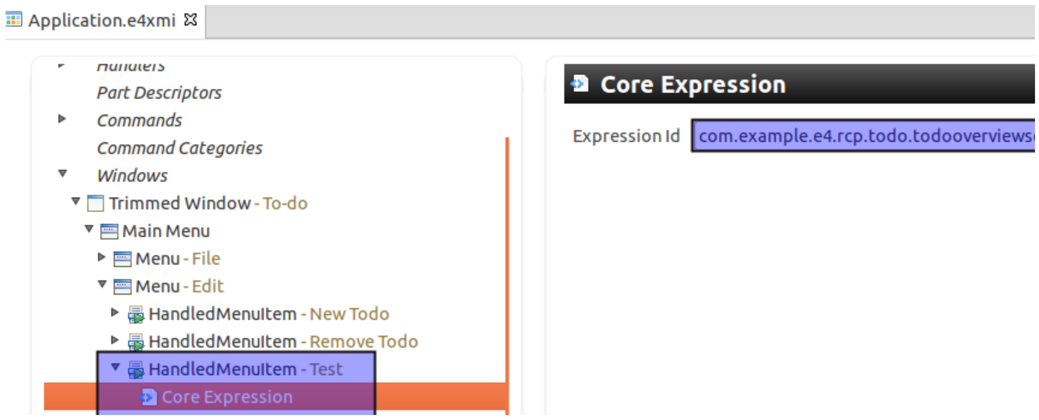
Later you will learn more about modifying the `IEclipseContext` but the following codes is an example `Handler` which places a value for the *myactivePartId* key in the context an example handler which places a value for the myactivePartId key in the context.

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

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

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

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



Expression Id: com.example.e4.rcp.todo.todooverviews

20.11. Naming schema for command and handler IDs

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

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

In case you implement commonly used functions, e.g. save, copy, you should use the existing platform IDs, as some Eclipse contributions expect these IDs. A more complete list of command IDs is available in `org.eclipse.ui.IWorkbenchCommandConstants`.

Table 10. Default IDs for commonly used commands

Command	ID
Save	org.eclipse.ui.file.save
Save All	org.eclipse.ui.file.saveAll
Undo	org.eclipse.ui.edit.undo
Redo	org.eclipse.ui.edit.redo
Cut	org.eclipse.ui.edit.cut
Copy	org.eclipse.ui.edit.copy

Paste	org.eclipse.ui.edit.paste
Delete	org.eclipse.ui.edit.delete
Import	org.eclipse.ui.file.import
Export	org.eclipse.ui.file.export
Select All	org.eclipse.ui.edit.selectAll
About	org.eclipse.ui.help.aboutAction
Preferences	org.eclipse.ui.window.preferences
Exit	org.eclipse.ui.file.exit

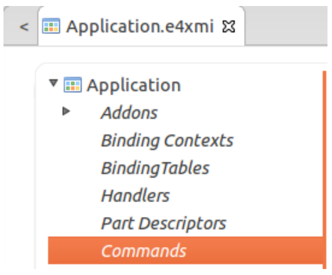
21. Tutorial: Defining and using Commands and Handlers

21.1. Overview

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

21.2. Defining Commands

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



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

Table 11. Commands

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

21.3. Defining Handler classes

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

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

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

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

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

  // Technically not needed
  // will default to true
  @CanExecute
  public boolean canExecute() {
    return true;
  }

}
```

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

- SaveAllHandler
- ExitHandler
- NewTodoHandler
- RemoveTodoHandler
- TestHandler

21.4. Defining Handlers in your model

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

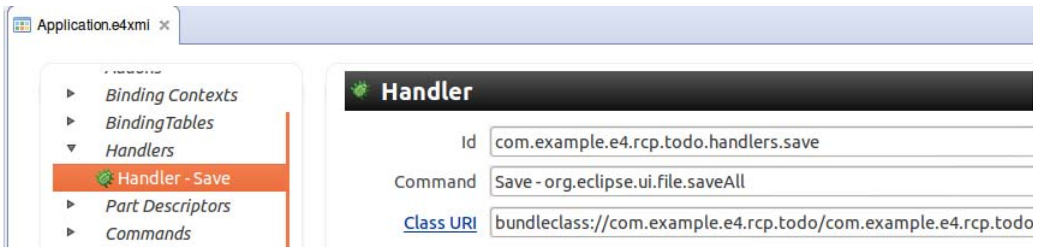
Use the *com.example.e4.rcp.todo.handlers* prefix for all handlers IDs.

Table 12. Handlers

Handler ID	Command - Class
*.save	Save - SaveAllHandler
*.exit	Exit - ExitHandler
*.new	New Todo - NewTodoHandler
*.remove	Remove Todo - RemoveTodoHandler
*.test	For testing - TestHandler

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

```
bundleclass://com.example.e4.rcp.todo/[CONTINUE...]
com.example.e4.rcp.todo.handlers.SaveAllHandler
```

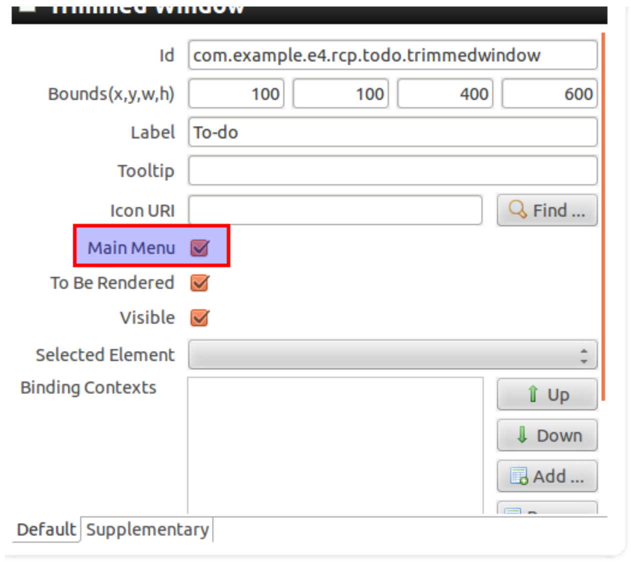


Id com.example.e4.rcp.todo.handlers.save
Command Save - org.eclipse.ui.file.saveAll
Class URI bundleclass://com.example.e4.rcp.todo/com.example.e4.rcp.todo

21.5. Adding a Menu

You will now add a Menu to your application model.

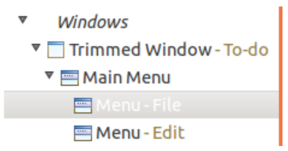
Select the *`Application.e4xmi`* file. To add a menu to a *Window* select your *TrimmedWindow* entry in the model and flag the *Main Menu* attribute.


Id com.example.e4.rcp.todo.trimmedwindow
Bounds(x,y,w,h) 100 100 400 600
Label To-do
Tooltip
Icon URI
Main Menu ✓
To Be Rendered ✓
Visible ✓
Selected Element
Binding Contexts

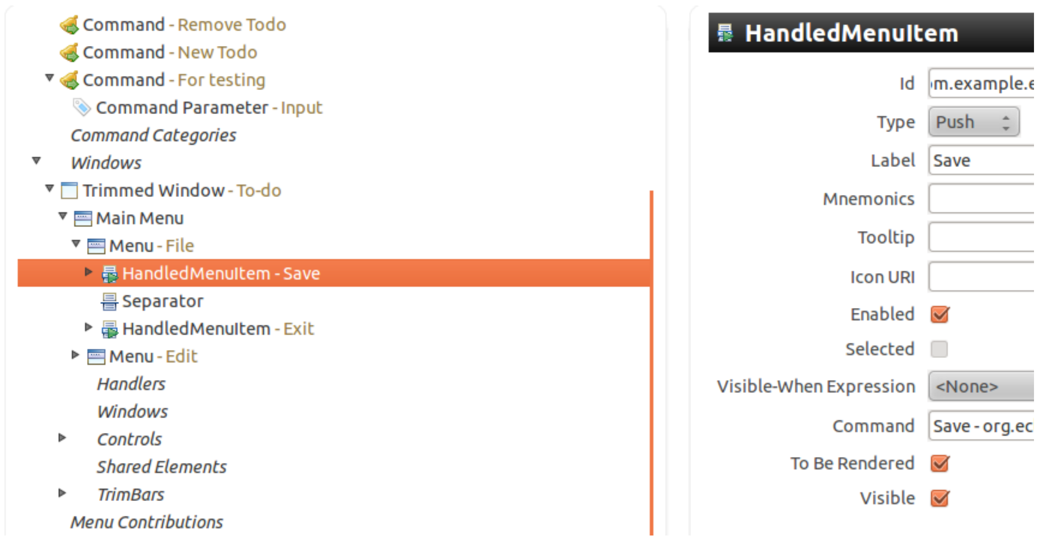
Assign the *org.eclipse.ui.main.menu* ID to your main menu.

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

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



Add a *HandledMenuItem* to the *File* menu. This item should point to the *Save* command via the `Command` attribute.



Id m.example.e
Type Push
Label Save
Mnemonics
Tooltip
Icon URI
Enabled ✓
Selected
Visible-When Expression <None>
Command Save - org.ec
To Be Rendered ✓
Visible ✓

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

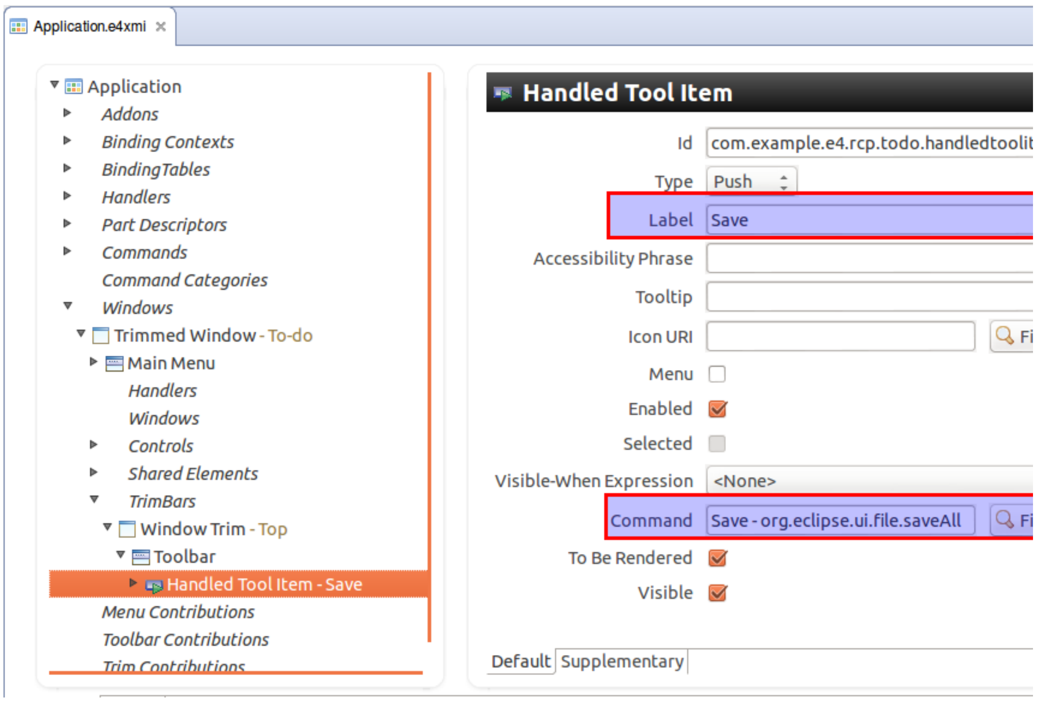
Add all other commmands to the *Edit* menu.

21.6. Adding a Toolbar

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

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

Set the label for this entry to *Save*.



Id: com.example.e4.rcp.todo.handledtoolit
Type: Push
Label: Save
Accessibility Phrase:
Tooltip:
Icon URI:
Menu: ☐
Enabled: ✓
Selected: ☐
Visible-When Expression: <None>
Command: Save - org.eclipse.ui.file.saveAll
To Be Rendered: ✓
Visible: ✓

21.7. Closing the application

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

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

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

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

21.8. Simulate save

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

22. Key bindings

22.1. Overview

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

Afterwards you need to enter the keybindings for the relevant *BindingContext* in the *BindingTable* node of your application model. A *BindingTable* is always assigned to a specific *BindingContext*. A *BindingContext* can have several *BindingTables* assigned to it.

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

22.2. BindingContext entries using by JFace

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

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

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

Table 13. Default BindingContext values

Context ID	Description
org.eclipse.ui.contexts.dialogAndWindow	Key bindings valid for Dialogs and Windows
org.eclipse.ui.contexts.dialog	Key bindings valid in Dialogs
org.eclipse.ui.contexts.window	Key bindings valid for Windows

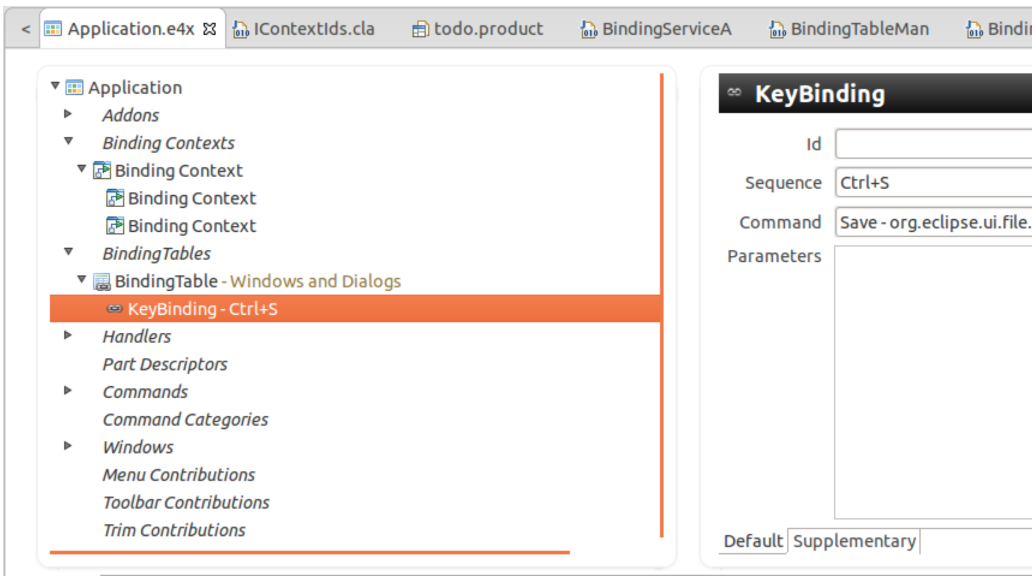
As an example, **Ctrl+C** (Copy) would be defined in *dialogAndWindows* as it is valid everywhere, but **F5** (Refresh) might only be defined for a Window and not for a Dialog.

22.3. Define Shortcuts

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

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

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



Id:
Sequence: Ctrl+S
Command: Save - org.eclipse.ui.file.
Parameters:

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

Table 14. Key Mapping

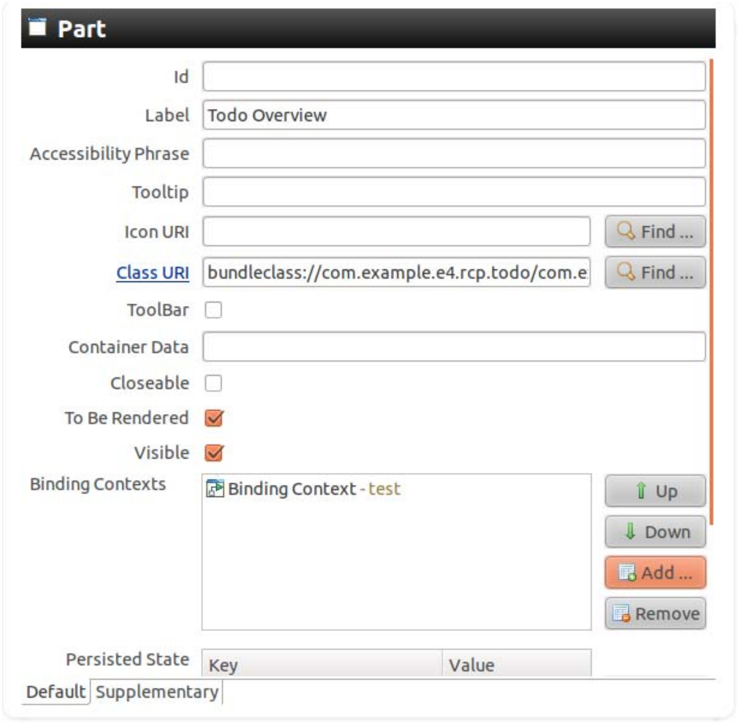
Control Key	Mapping for Windows and Linux	Mapping for Mac
M1	Ctrl	Command
M2	Shift	Shift

M3	Alt	Alt
M4	Undefined	Ctrl

These values are defined in the `SWTKeyLookup` class

22.4. Key Bindings for a Part

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


Part

Id	
Label	Todo Overview
Accessibility Phrase	
Tooltip	
Icon URI	
Class URI	bundleclass://com.example.e4.rcp.todo/com.e
ToolBar	☐
Container Data	
Closeable	☐
To Be Rendered	✓
Visible	✓
Binding Contexts	Binding Context - test

22.5. Activating Bindings

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

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

22.6. Issues with Keybinding

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

23. Application model modifications at runtime

23.1. Creating model elements

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

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

23.2. Modifying existing model elements

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

24. Example for changing the application model

24.1. Example: Dynamically create a new Window

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

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

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

24.2. Example: Dynamically create a new Part

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

```
package testing.handlers;

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

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

25. Model Addons

25.1. Overview

Eclipse 4 tries to keep the core framework as minimal as possible. `Addons` model object can be registered globally on the application model and can enhance the application with additional functionality. `Addons` are normal Java objects.

`Addons` use the `bundleclass://` URI convention in the application model to refer to the Java objects.

Having these Addons registered allows them to be enhanced or replaced by the Eclipse platform team or by a customer specific implementation in case the need arises.

25.2. Framework Addons

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

- CommandServiceAddon
- ContextServiceAddon
- BindingServiceAddon
- CommandProcessingAddon

- ContextProcessingAddon
- BindingProcessingAddon

25.3. Additional SWT addons

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

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

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

25.4. Relationship to other services

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

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

The `EventAdmin` services provides the notifications about Eclipse events, e.g. if a *Part* is activated. As the `EventAdmin` services is created before `Addons` are created , the `Addons` can to subscribe to events from the Eclipse platform.

26. Accessing and extending the Eclipse Context

26.1. Accessing the context

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

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

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

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

}
```

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

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

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

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

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

public class TodoDetailsPart {

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

}
```

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

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

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

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

26.2. OSGi services

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

26.3. Objects and Context Variables

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

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

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

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

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

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

}
```

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

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

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

}
```

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

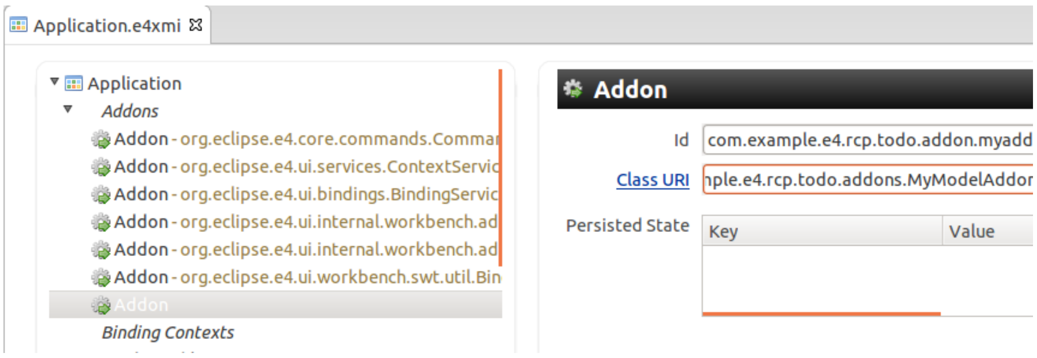
The `modify()` methods searches up the chain to find the `Context` defining the variable. If none of the `Context` on the parent chain have a value set for the name, the value will be set in this `Context`. If the key already exists in the context, then `modify()` requires that the in this `context`.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

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

26.4. Model Addons

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

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



Application.e4xmi

- Application
  - *Addons*
    - Addon - org.eclipse.e4.core.commands.Comma
    - Addon - org.eclipse.e4.ui.services.ContextServic
    - Addon - org.eclipse.e4.ui.bindings.BindingServic
    - Addon - org.eclipse.e4.ui.internal.workbench.ad
    - Addon - org.eclipse.e4.ui.internal.workbench.ad
    - Addon - org.eclipse.e4.ui.workbench.swt.util.Bin
    - Addon
  - *Binding Contexts*

Addon

Id: com.example.e4.rcp.todo.addon.myadd

Class URI: nple.e4.rcp.todo.addons.MyModelAddor

Persisted State:

Key	Value

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

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

import javax.annotation.PostConstruct;

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

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

26.5. RunAndTrack

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

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

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

The runnable does not need to be explicitly unregistered from this context when it is no longer interested in tracking changes. If a subsequent invocation of this runnable does not access any values in this context, it is automatically unregistered from change tracking on this context.

26.6. Context Functions

Context Functions implement the `IContextFunction` interface and allow you to lazily create an object.

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

Context Functions are contributed as OSGi Services. They implement the `IContextFunction` interface from the `org.eclipse.e4.core.contexts` package. The Eclipse 4 runtime adds them by default to the application context but you could package. The Eclipse 4 runtime adds them by default to the application context but you could add *Context Functions* to any context in the context hierarchy.

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

The following example shows the declaration of a *Context Function*. This function is available in the application context under the `com.example.e4.rcp.todo.contextservice.test` key and can be injected via the `@Inject @Named` 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.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="com.example.e4.rcp.todo.contextservice.test"/>

</scr:component>
```

Using *Context Functions* instead of plain OSGi services have the advantage that these functions has access to the application context, that they can be lazily created and that they are available in the application context. OSGi services are singletons and could not be used in a multi user environment.

27. Using dependency injection for your own Java objects

27.1. Overview

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

Both approaches are described here.

27.2. Using dependency injection to get your own objects

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

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

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

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

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

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

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

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

27.3. Using dependency injection to create objects

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

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

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

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

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

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

28. Relevant tags in the application model

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

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

Table 15. Relevant tags of application model elements

Tag	Model element	Description
shellMaximized	Window or Trimmed Window	Window is maximized at start of the application.
shellMinimized	Window or Trimmed Window	Window is minimized at start of the application.
NoAutoCollapse	PartStack	Avoids that the MinMax Addon minimizes this PartStack if all Parts have been removed from it.
FORCE_TEXT	ToolItem	Enforces that text and icon is shown for a toolbar item.
NoMove	Part	Prevents the user from moving the part (based on the DndAddON).

29. Eclipse 4 Best Practice

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

29.1. Extending the Eclipse Context

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

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

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

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

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

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

29.2. Application communication

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

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

29.3. Static vs. dynamic application model

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

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

29.4. Component based development

Eclipse and OSGi support component based development.

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

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

29.5. Usage of your own extension points

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

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

29.6. API definition

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

29.7. Packages vs. Plug-in dependencies

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

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

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

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

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

30. Closing words

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

31. Thank you

Please help me to support this article:

Flattr this! Donate

32. Questions and Discussion

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

33. Links and Literature

33.1. Source Code

Source Code of Examples

33.2. Eclipse 4

Eclipse 4 RCP Wiki

Eclipse 4 RCP FAQ

vogella Eclipse Tutorials

Eclipse wiki with Eclipse 4 tutorials

Eclipse 4 Dependency Injection Wiki

Eclipse 4 RCP Wiki for tags for the Application model

Eclipse 4 Build Schedule