



Eclipse 4 (e4) Tutorial

Table of Contents

1. The e4 Application Model
2. Implementing Views
3. Extending the Application Model
4. Dependency Injection Basics
5. Behavior Annotations
6. Services
7. Eclipse 3.x vs. Eclipse 4 - Which Platform to use?
8. Soft migration from 3.x to Eclipse 4 (e4)

The e4 Application Model

This tutorial series introduces the new concepts in the Eclipse 4 Application Platform, aka RCP 2.0. While some projects still use the compatibility layer, it is worthwhile to look at and benefit from the new concepts. This tutorial and all other parts of the series are available as a downloadable PDF.

We will start with the foundation of every Eclipse 4 application, the application model. In this first part, we give an overview of the most important elements of this model to define the layout of your application. Subsequently, we will introduce the different options for modifying this model.

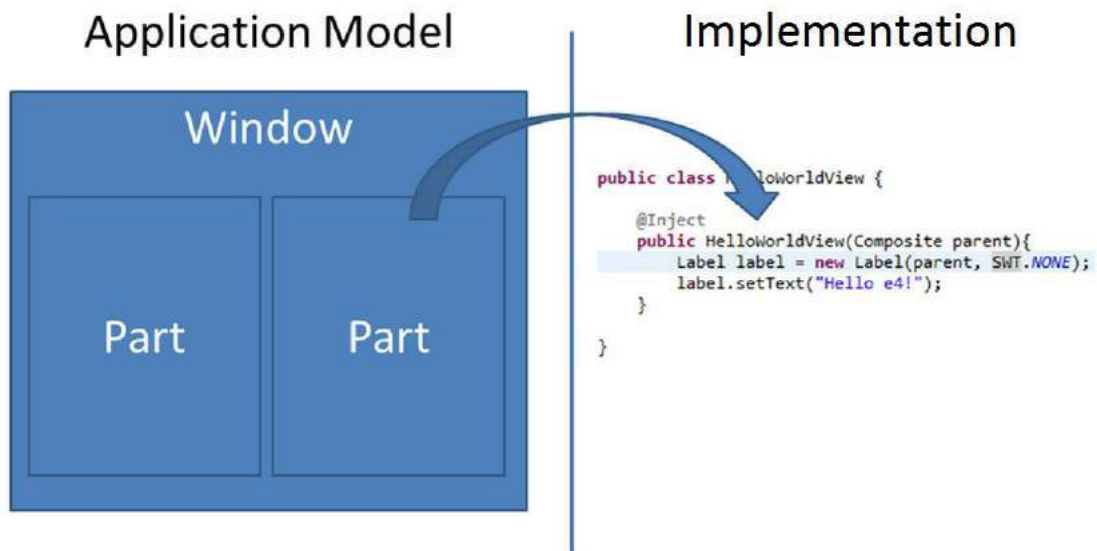
Application Model vs. Views

In Eclipse 4, the workbench is defined in the so-called Application Model. This includes windows, views, perspectives, menu contributions, handlers and key bindings. Using the model you define the general design or “skeleton” of your application. Defining a model doesn’t already require that you implement the single components. For example, you can add a view to the model without implementing its contents.

To show the resulting separation between the general workbench design and the implementation of single parts, I will not show any SWT or JavaFX code in this section. Instead, we’ll focus on the model and how to connect the model to code.

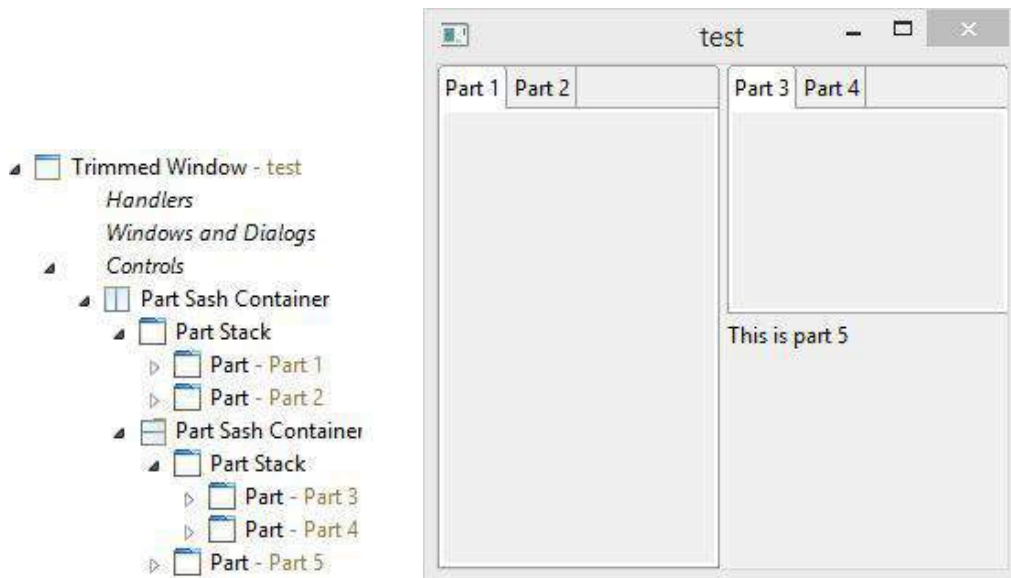
Windows, Parts and PartContainers

The cornerstones of the application model are windows, parts and part containers. Contrary to the eclipse 3.x platform, e4 has combined views and editors into the concept of Parts, which represent views inside a window. If you add a part in the model, you can later connect it to your implementation of the selected view.



The parts of an application model are connected later to their implementations

As parts cannot exist on their own, they are always contained in PartContainers. Those PartContainers create an application layout. Besides windows, there are two elements which can contain Parts: PartSashes and PartStacks. A PartSash will arrange contained elements either vertically or horizontally. A PartStack adds an area, where the contained elements are stacked behind each other, only one element is shown on top. PartSash and PartStack can contain each other to create any kind of layout within a window. The following example shows an example layout. The left side shows the respective Application Model, the right side the rendered result. The first PartStack is set to “horizontal” orientation, the second one to “vertical”. Please note, that part 5 would actually not be visible, if parts are not contained in PartStacks, their label is not shown. For the example, we added an implementation for part 5, which shows a label to make it identifiable in the screenshot. All other parts do not have any implementation, yet, so you can design your application without already thinking about the contents of views. In the following section, we describe how to create a first application model using Windows, PartSashes, PartStacks and Parts, only.

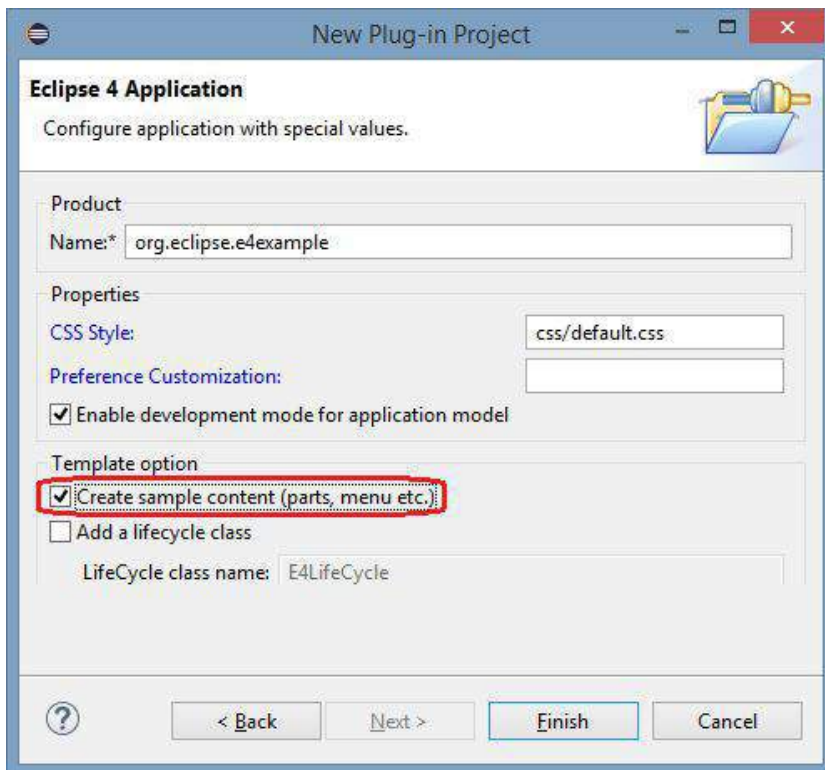


Installation

The most convenient way to modify an Application Model is the e4 Model Editor, which will be described in the following. Starting from Mars, it is already part of some Eclipse Packages (Eclipse for RCP Developers, Eclipse Modeling Tools and Eclipse for Committers). For carrying on with the tasks from this tutorial, we recommend you to download the latest version of “Eclipse for RCP developers” from here: <https://eclipse.org/downloads/>.

Creating an e4 Application

The easiest way to get started is to use a template to create a new e4 application. This will create a bundle containing all necessary artefacts for an Eclipse 4 Application including the Application Model. To create such a project, choose the "Eclipse 4 Application Project" entry within the "New Project" wizard. Give a name to your project on the first page of the wizard and select the “Create sample content” checkbox on the last page. This option will already fill the template Application Model with some elements.

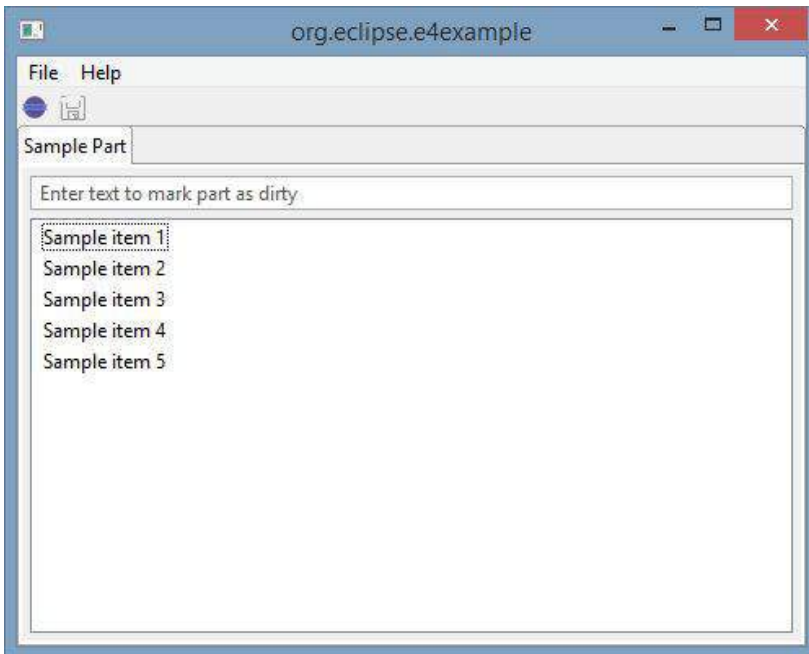


Before we have a look at the Application Model, we will start the template application once. For this purpose, the template wizard creates a product definition and you can start the application simply by starting this product. To do so, open the *.product file and click on run or debug in the upper right corner of the editor.



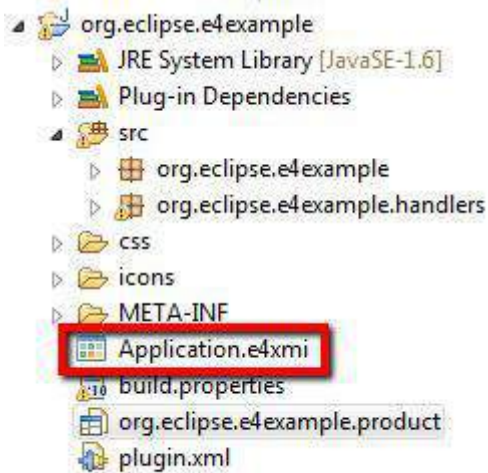
Click here to start the product.

As you can see below, the generated template application already contains a window, two menus, a toolbar and a PartStack and a Part within it. In fact, the model also already contains a PerspectiveStack and a perspective, but this is currently not visible.



The e4 Model Editor

To modify the application model and therefore the layout of your application, Eclipse provides the e4 Model Editor. You can open it by double clicking the `Application.e4xmi` located in the root level of the project.

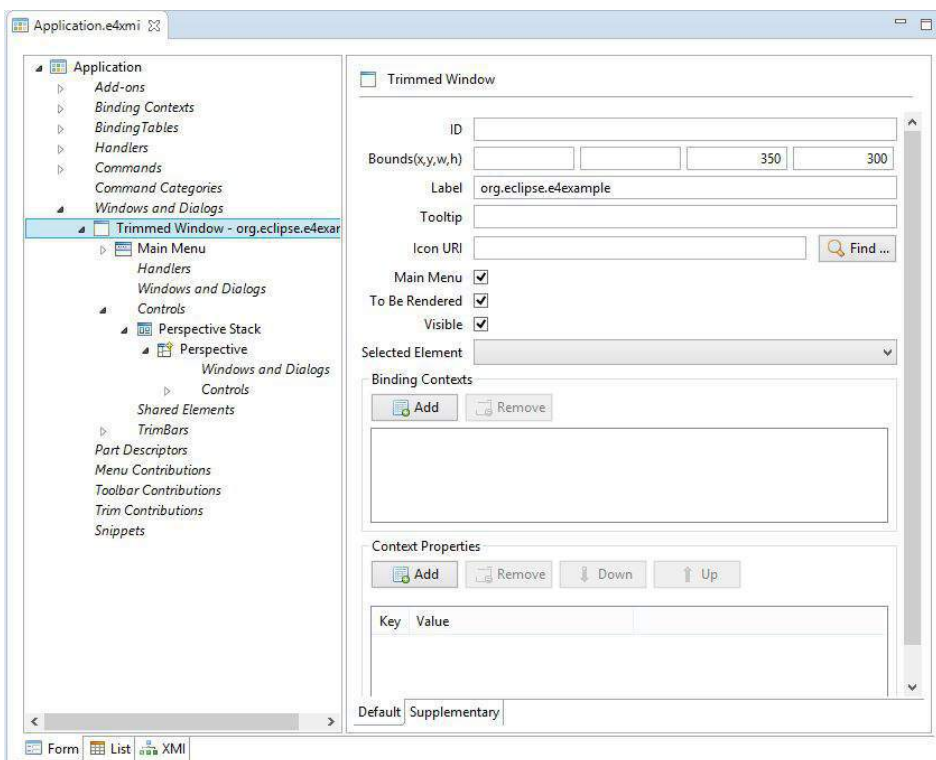


Open the application model to modify the workbench

On the left side you see a tree showing the complete contents of the model. Tree nodes with icons are model elements, e.g. the root element “Application”. In the level below those elements you find

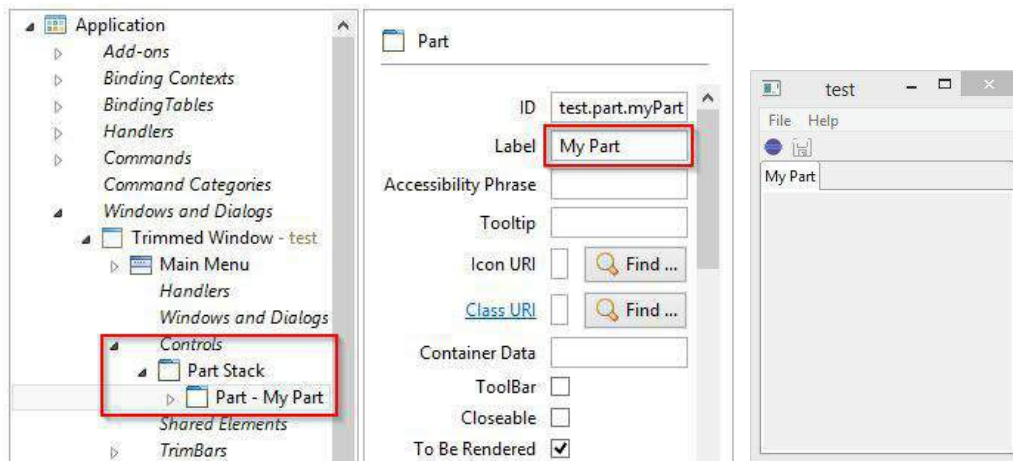
tree nodes without icons. Those are like folders and structure the elements, which are contained by a parent elements. As an example, the application has a folder called “Windows and Dialogs”. By expanding this node, you see all windows and dialogs, which are contained by the application. By selecting any element in the tree, a detailed view will be opened on the right side, allowing you to modify the properties of that element. If you select a folder, the right side will present a list of elements which are contained in that folder.

The top-level elements of an application are usually one or more windows that you can find in the application model in the folder “Windows and Dialogs”. The template project already contains a TrimmedWindow. By selecting this element you can, for instance, modify the size of this window, in the “Bounds” fields. Check the result by restarting the application.



With a right click in the on folders in the tree, new elements can be added within those folders. Using the delete action in the right click menu of an element, you can remove them. As an example, you can remove the existing PerspectiveStack and just add a single Part instead. To do so, you need to expand the “Controls” item of the Trimmed Window in the Application tree. After a restart of the application, you will notice that the main area of the application does not

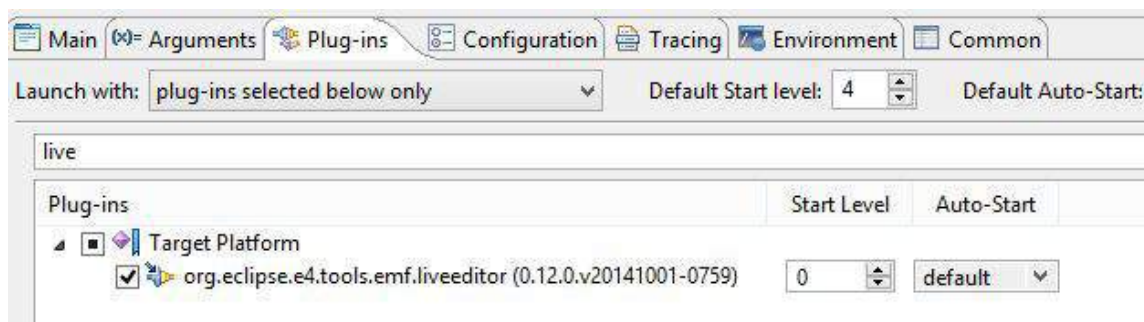
have a border anymore. However, the new part isn't visible, as it shows no tab anymore. Tabs of Parts are only visible, if parts are contained in PartStacks. Try to add a PartStack as a child element of the TrimmedWindow and move the part in it. Adapt the label property of the Part, restart the application and check the result.



Model Spy (Live Editing)

Eclipse allows you to define the workbench using the application model even without providing implementations. However, this is sometimes hard to work with, because empty Parts are often hard to identify. To resolve this, there is a special version of the Model Editor called Model Spy. It allows you to access the application model of a running application, modify it and highlight selected components. The Model Spy is not yet part of the standard Eclipse Packages, it can be installed from [the e4 update site](#).

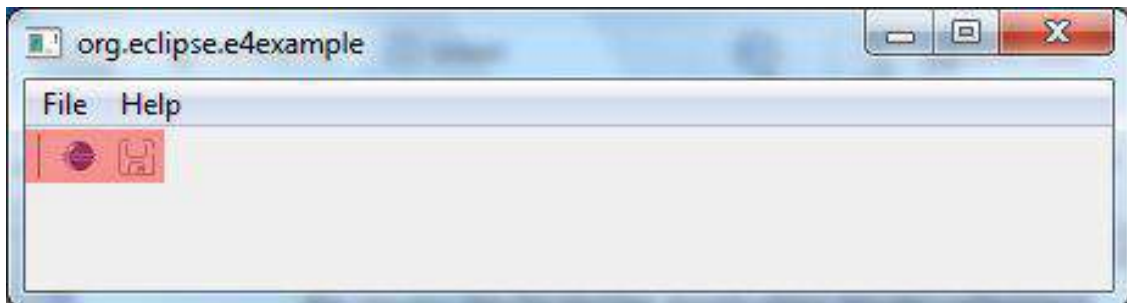
To enable the Model Spy you need to start an additional plugin along with your application. Please note that you typically do not want to make the Model Spy a permanent part of your application, so it should not be added to the product. Instead, open the run configuration and add the bundle "" to it. Additionally click on "Add required" to include the required dependencies. A run configuration should have been created for you when you first started the product.



In the running application you can start the live editor via ALT+SHIFT+F9. This editor works exactly like the editor in your IDE, however, it directly accesses the application model of the running application. If you, for instance, open the TrimmedWindow in the editor and change its size or position, the changes are directly applied in the running application.



The live editor is not only capable of modifying elements, you can even add new ones. As an example, if you add a new window to the application model (right-click on the folder “Windows and Dialogs”), a new window will be opened in the application. To maintain an overview of which components are visible in the application, these components can be colored. By right-clicking an element in the live editor, e.g. the TrimBar and selecting “Show Control”, the control will be colored in red in the running application.

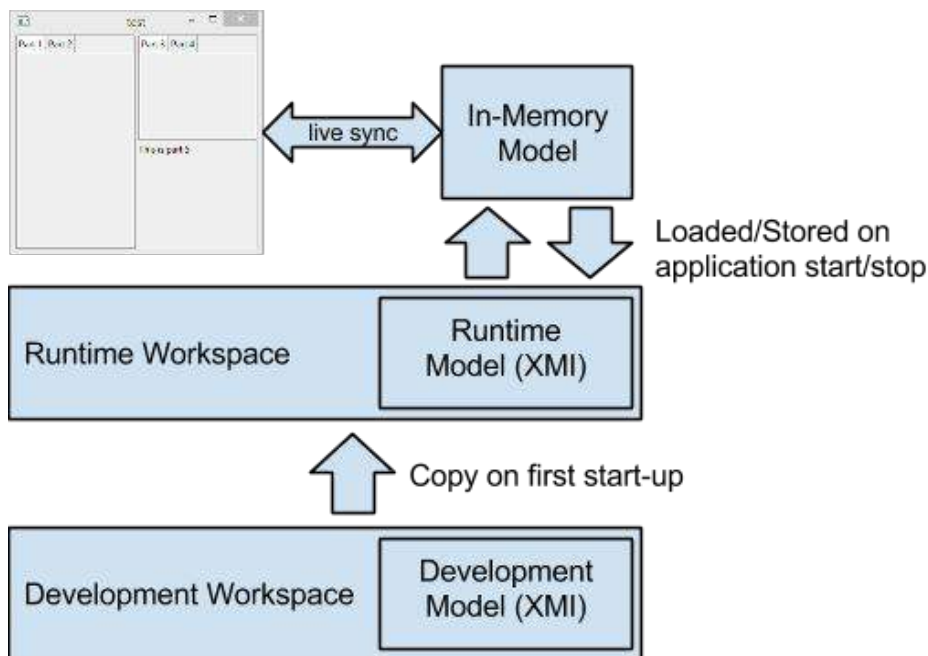


Using this feature, one can easily visualize changes within the application model. This is especially useful for elements which are not directly visible in the UI. As an example, if you add a new Part in a Window (without a containing PartStack), it will not be visible without coloring, as it does not have any content yet.

If you use the live editor to change the application model, the changes will only be reflected in the running application. To transfer them into the deployable application, you can copy the modified version of the model using the tab “XMI” and copy it into the model available in your IDE.

Development Model vs. Runtime Model

So far, we have created and modified the Application Model in our IDE, meaning during development time. This was done by using the e4 Model Editor, the resulting model was persisted in the Application.e4xmi file. When a e4 application is started, this file is deployed along with the containing bundle. When an application is started, a copy of this file will be placed in the workspace of the running application. From there on, this copy will always store the latest state of the running application, e.g. the arrangement of Parts. Consequently, when re-starting the application from within the IDE, this copy will not be overridden by default. While developing an application, you typically want the new state of the Application Model, e.g. when you have added some new elements. You can achieve this by either deleting the workspace of the running application completely or by adding the command parameter “-clearPersistedState” to the product/run configuration. Please note that the product we have created before using the e4 wizard already contains this parameter (see the .product file on the tab “Launching”).



The default e4 application de-serializes this file and creates an in-memory representation of it. A rendering component interprets this model and creates the workbench describe by it, i.e. windows, parts, etc. If the application is started again with a adapted model, this will be reflected in the UI. Even more, the application model can even be changed at runtime using it's API, e.g. the label of a window can be changed or new parts can be created. The rendering component is listening to all changes to the model at runtime and will reflect those changes in the running application. We will describe in the following section how to access the model at runtime.

Programmatic Access to the Application Model

During runtime, there is a Java Object for every element of the deserialized Application Model. Technically, those objects are EMF objects (EObject), so the API is very familiar to anyone who has worked with EMF before. By using this API you can create or modify parts of the application programmatically, like, for example, reacting to a user action. To test this in the template

application created before, you can take a look at one of the existing handlers, such as the class `OpenHandler`. As you can see in this handler, there is a method `execute()` marked with the annotation `@Execute`, which will be called if the connected `MenuItem` is clicked by the user.

Dependency injection allows the programmer to easily define which parameters are needed within this method. We will go into detail about dependency injection and the implementation of handlers later, for now, we will just use the existing handler to execute some example code, which modifies the Application Model at runtime.

In the following code example, the method requires the application itself as parameter, so it will be injected by the framework. Please note, that the Java Objects (more precisely their interfaces) to access the Application Model at runtime are all prefixed with a "M". As an example, you can access a Window using the interface "MWindow" or a Part using the interface "MPart".

In the following code example, a new window is created. To add this new window into the application, the application is required as a parameter. Using the API, the window is sized, a new part is added into the window and the window is added to the application. By adding the window to the application, it is opened in the running application. Start the application and press the toolbar button to check the result.

```
@Execute
public void execute(MApplication application) {
    MWindow mWindow = MBasicFactory.INSTANCE.createTrimmedWindow();
    mWindow.setHeight(200);
    mWindow.setWidth(400);
    mWindow.getChildren().add(MBasicFactory.INSTANCE.createPart());
    application.getChildren().add(mWindow);
}
```

Conclusion

The e4 application models allows you to define the general design of an application in a consistent way, without implementing single parts in advance. We described different methods to modify the application model, including how to modify the model during runtime using the live editor or the API. At this point we have only created placeholders in the application. The next part of this series describes how to connect the application model with the implementation of UI



components, that is, how to create the connection between a part and the implementation of a view filling this part.

Implementing Views

From the Application Model to the Implementation of Views

This tutorial series introduces the core concepts of the Eclipse 4 Application Platform (e4). One of the key innovations of e4 is the separation and independence between the application model and the implementation of the application's parts, such as view. In the first part of this tutorial we provided an overview of the application model, as well as the different ways to modify it, using the editor or the API. With the application model, it is possible to define and test the basic design of an application without having already implemented single views. In this second part of the tutorial, we explain how to create the missing part, the implementation of views, for which we have thus far created only placeholders in the application model. This tutorial and all other parts of the series are now available as a [downloadable PDF](#).

An application model without views?

At first glance, it might be confusing as to why Eclipse 4 facilitates such a clear separation between the application model and the implementation of UI components. This is especially true, as one part doesn't really make sense without the other. In Eclipse 3.x and also in other frameworks, implementations of UI components, such as views, have to implement given interfaces. This approach defines exactly which methods a developer has to implement to create a view. However, this approach also restricts the ability to reuse the implementation of UI components.

A well-known example for this problem is the differentiation between views and editors in Eclipse 3.x, which required different interfaces to be implemented. If you want to reuse a view as an editor or vice versa, you had to refactor it. Another example would be the reuse of a view in a modular dialog. Finally, when RCP applications are transferred to another context, e.g. on a mobile device ([see RAP mobile / Tabris](#)), the design of the workbench has to be changed to fit smaller screens. Therefore one of the goals of Eclipse 4 is to implement UI components in a modular and independent way. The UI consists of small, independent parts, which are not bound to any framework classes such as editor or view, and can be reused in any context. Finally, the application model itself has no UI toolkit dependencies at all. Therefore, it can also be used for implementing applications in other technologies than SWT, for example in JavaFX as provided by the [e\(fx\)clipse project](#).

A view without an application model?

To demonstrate the modularity of the application model and the implementation of views, we started in the first part of this tutorial with the creation of an application model without any implementations. Using the e4 tools, you can even visualize the "empty" application model.

Before we fill the application model with implementations, we'll demonstrate the opposite, that is, implementing views using SWT without an existing application model. We'll develop the modular parts of our application before we know the exact design of the workbench to illustrate Eclipse 4's modular UI development.

In Eclipse 4, views do not have to implement a given interface. Instead, views define the parameters that the workbench needs to provide. In one of the simplest cases, an SWT view just requires a parent composite, on which the view can be placed. The annotation "@Inject" will be used later on by the Eclipse 4 framework to determine if the parameters of the view should be "injected". We will go into more detail about dependency injection later in the tutorial.

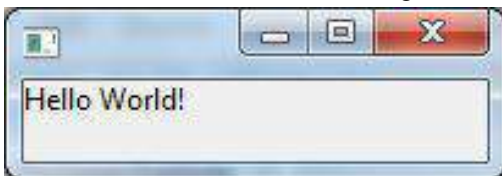
The following code example shows a very easy "Hello World!" view in SWT.

```
public class ExampleView {
    @Inject
    public ExampleView(Composite parent) {
        Label label = new Label(parent, SWT.NONE);
        label.setText("Hello World!");
    }
}
```

Using the application model, this view can be shown later on as part of the workbench in an application. To demonstrate the flexible reusability and testability of such a view, we will first use it without any workbench. The following code example shows how to open the "HelloWorld" view just using plain SWT. It is worth mentioning that this is a plain Java program. To run this, we only need the relevant SWT libraries. This shows, that the view can be reused anywhere, e.g. in a dialog, a wizard or even outside of the Eclipse workbench.

```
public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setLayout(new FillLayout());
    new ExampleView(shell);
    shell.open();
    while( !shell.isDisposed() ) {
        if( ! display.readAndDispatch() ) {
            display.sleep();
        }
    }
}
```

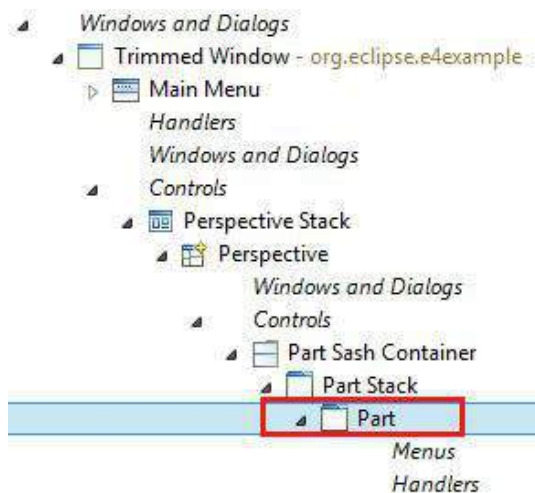
This screenshot shows the running Hello World application started from a plain Java program.



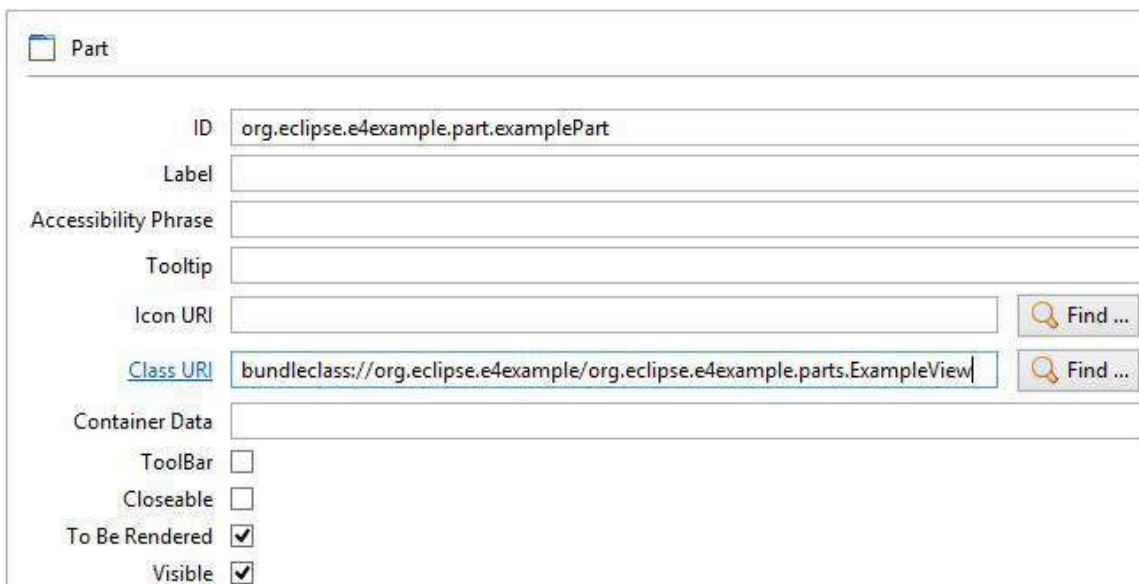
The Reunion

So far we have created and tested an application model and the implementation of a view separately from each other. Now it's time to bring both parts together. We do this by adding an element ("Part") in the application model representing the view within the workbench (we can also of course use an existing one). The part will be linked to the implementation of the view that we have just created. Using the e4 template application created with the e4 tools (see section on Installation in the Eclipse 4 Tutorial – Part 1), a part can, for example, be created within the existing "PartStack".

You see in this screenshot that parts are added to the application model as placeholders for views and editors.



To link the part to the implementation of the view, the view's class has to be selected in the properties of the part under "Class URI". When you start the application, Eclipse 4 will create a part within the workbench and the linked view implementation will be initialized. That means, the constructor of the view implementation will be called. Parameters, which are required by the view, will be taken from the current context and will be injected into the view. As an example, Eclipse 4 will use the content area of the part as a parent composite for the view and therefore place the view within the part.



Part

ID

Label

Accessibility Phrase

Tooltip

Icon URI

Class URI

Container Data

ToolBar

Closeable

To Be Rendered

Visible

Figure : Parts are linked to the implementation of views using the Class URI

Handlers

For our next step, we want to add some behavior to our application. Therefore, we will implement a Handler, which is triggered by a button in the toolbar of the application. Handlers are the components, which define some specific behavior to be triggered. Similar to the implementation of UI components, Eclipse 4 allows a clean separation between the framework and the implementation of a handler, which enables reusability and testability. To demonstrate, we'll follow a similar workflow to the previous sections, implementing and testing the handler independently from the integration into the application model. We'll then integrate it in a following step.

Also parallel to how UI components work in Eclipse 4, handlers don't have to implement a given interface. Instead, they define the required parameters. This reduces the number of required parameters to the minimum needed, making it also easier to test the handler. The following code example shows the implementation of a very basic handler for opening a "Hello World!" dialog. The handler needs only one parameter, a shell, to open the dialog. Using the annotation "@Execute", the handler tells the Eclipse 4 framework which method to execute. In addition, the annotation has the same effect as "@Inject". That means that the required parameters of the method execute(), in this case a shell, will be injected by the framework. As the example handler does not have a state, the execute() method can be static.


```
public class MyHandler {  
  
    @Execute  
    public static void execute(Shell shell){  
        MessageDialog.openInformation(shell, "", "Hello World!");  
    }  
  
}
```

Handlers in Eclipse 4 are very easy to test, reuse and even chain, as they only require the parameters they really use. The following code example shows a simple Java program which tests the implemented handler.

```
public static void main(String[] args) {  
    Display display = new Display();  
    Shell shell = new Shell(display);  
    shell.open();  
    MyHandler.execute(shell);  
    while( !shell.isDisposed() ) {  
        if( ! display.readAndDispatch() ) {  
            display.sleep();  
        }  
    }  
}
```

The code example shows a complete Java program again. However, these tests could also be written in plain JUnit. If you imagine a handler implementing some behavior (in contrast to just open a dialog), it makes sense to have a JUnit test for the method marked with `@Execute`.

To integrate the handler with a button in a toolbar, we need another element in the application model. The easiest way to integrate the handler is using a "Direct ToolItem" (see Figure). The ToolItem is placed in the ToolBar and therefore specifies, that there is another button to be clicked. Analogous to the Part, the implementation of the handler can be bound to the element by setting the Class URI. This specifies, that the implementation of the handler is executed, when a user clicks on the toolbar button. Finally, we need to set a label or icon for the tool item to make it visible in the example application.

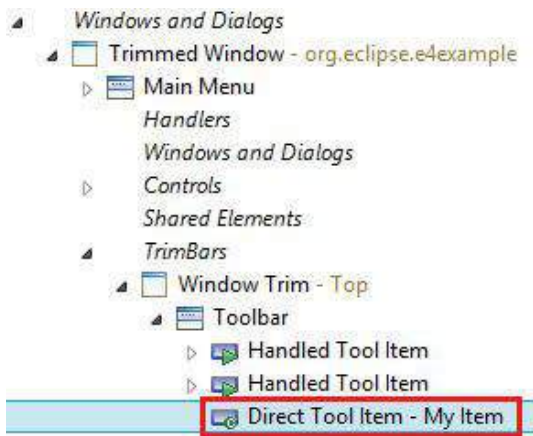
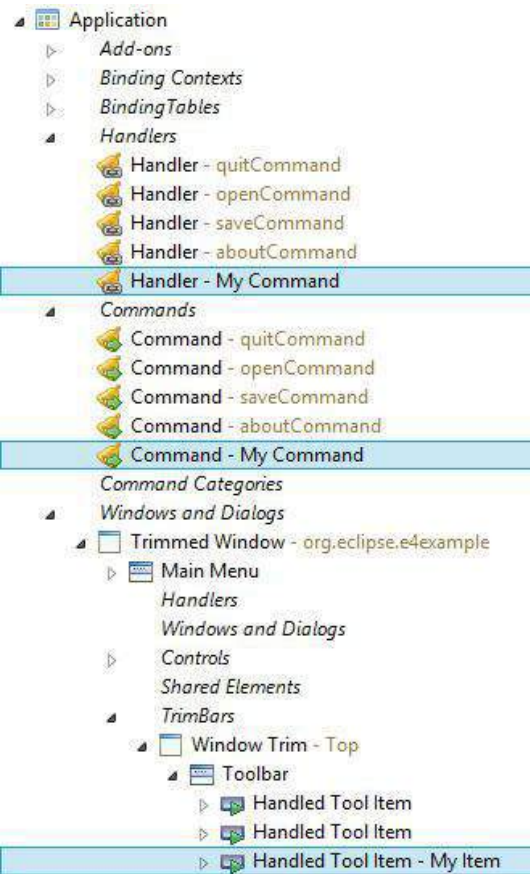


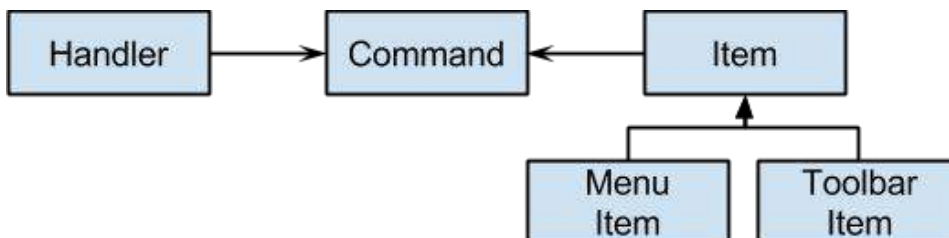
Figure : Creating a Direct ToolItem in the toolbar of the application

Using a "Direct ToolItem" is the fastest way to integrate a handler with an item in the toolbar. However, for more flexibility and better reuse we recommend using commands and handlers. This enables to have alternative implementations of a certain user action and also to use key bindings for triggering handlers.


Commands, Handlers and Items already existed in Eclipse 3.x. They facilitate a separation between a visible item to trigger this action (MenuItem or ToolbarItem) and the implementation of the action (Handler). As a binding in between, Eclipse defines the concept of a Command. All three elements, Handlers, Commands and Items are created as part of the application model. Items are created at the place, they should be displayed to the user, e.g. in a toolbar (as shown in the following screenshot) or in a menu. For both item types (Menu and Toolbar) use handled item, e.g. HandleToolbarItem. Commands are defined on the application level (see also the following screenshot) and are therefore valid for the whole application. Handler can be created on three levels, the application itself, on a window level or for a specific part, only.



As the following diagram shows, items and handlers reference a certain command and are thereby bound to each other. It is possible to bind several items to the same command, e.g. to show an action in a menu and in a toolbar at the same time. It is also possible to bind multiple handlers to the same command providing alternative implementations of an action. However, in this case, only one handler must be active at the same time. The Eclipse 4 Application platform will automatically activate handlers in the current application context. As an example, if a certain part is focussed, its specific handlers will be activated. If you have only one implementation for a handler, you can place it on the application level, if you have alternative implementation for different parts or windows, you place them in those elements respectively.




After all three elements are created, both the handler and the item need to be bound to the command (see the following screenshots. Commands can now be reused within the application, as an example, key bindings can be used to trigger the execution of a command or other items can be bound to it.

 Handler

ID

Command


[Class URI](#)

 Command

ID

Name

Description

 Handled Tool Item

ID

Type

Label

Accessibility Phrase

Tooltip

Icon URI

Menu

Enabled

Selected

Visible-When Expression

Command

To Be Rendered

Visible

Figure : A handler is bound to its implementation as well as to a command.



Conclusion

Eclipse 4 facilitates a clear separation between the definition of a workbench (i.e. application model) and the implementation of concrete parts of it (e.g. views or handler implementations). Implementations use dependency injection to define which parameters they require. This approach leads to minimal interfaces and implementations that are very easy to test and reuse. This tutorial showed how to create and test views and handlers without having a corresponding element for them in the application model and how to integrate them into a real application afterwards. In the next installment of this tutorial, we describe how to extend and modularize the application model, that is, how to contribute views and handlers from several plug-ins.

Extending the Application Model

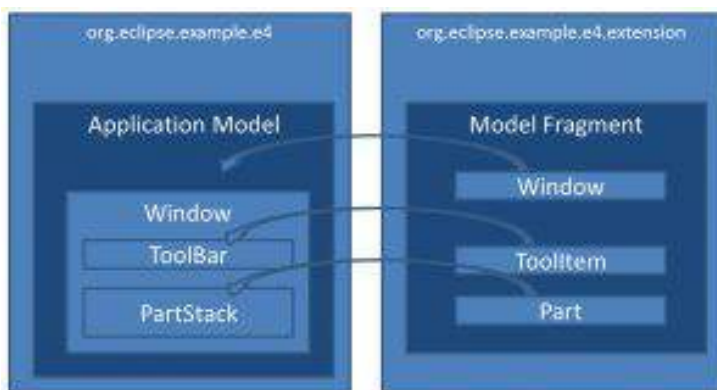
In the previous parts of this tutorial series we described how to create an application model and link those elements to implementations. Until now we have only worked with one application model. However, Eclipse applications usually follow a modular design. That means, an application consists of an arbitrary number of bundles, which add functionality and are loosely coupled.

In this part, we describe how to extend an existing application model with new elements, for example, adding a new entry to a menu. This tutorial and all other parts of the series are now available as a [downloadable PDF](#).

Only one model?

One of the major advantages of Eclipse RCP development is the modular design of applications. The module concept, based on OSGi, enables the independent development of features, as well as their independent deployment. A very good example of such a modular application is the Eclipse IDE, where many additional plugins can be installed. Many of these extensions affect the workbench design of an application, that is, they add additional buttons, menu items and views. In e4, the application model is the central and consistent approach to designing the workbench. However, there needs to be a way to extend the application model from new plugins. Eclipse 3.x uses extensions points for this. Eclipse 4 offers model fragments and model processors. A model fragment is a small application model in itself and defines elements which need to be added to the root application model. Fragments can add anything that can be part of the application model, for example handlers, menu items or even windows.

The following diagrams show an example of such an extension. The application model of the plugin "org.eclipse.example.e4" is extended by a fragment from the plugin "org.eclipse.example.e4.extension". For every element, in this case a Window, a ToolItem and a Part, you need to define the place, where it gets added in the core model.



The application model can be extended using fragments.

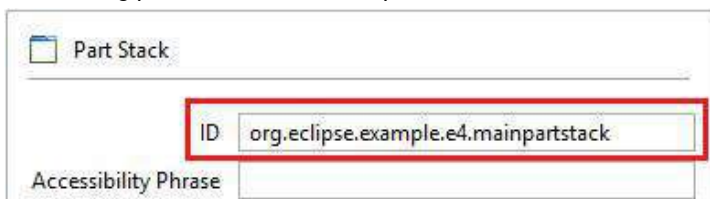
Processors offer a mechanism to programmatically extend an application model. This allows the application to react to the current state of the model. As an example, you can only add a new button if another contribution is installed or if you can remove existing elements. In this tutorial, we describe both ways of extending an application model, fragments and processors. In both cases, elements of the application model are linked to their implementations as described in the previous chapters of this tutorial. The implementation is usually part of the plugin doing the contribution, as in the previous example "org.eclipse.example.e4.extension".

Warm-Up

The first step is to create a main plugin and an application model which can be extended. As in the previous parts of this tutorial, we will use the e4 template application, which can be created using a wizard. It is important that elements in the application model which will be extended have a unique id. This id is used to reference elements from the extending fragment. In the template application, both the application and the toolbar already have an id. As we want to add a new part to the existing part stack, the part stack also has to have an id. Therefore, the field "id" has to be set for the part stack in the application model (Application.e4xmi).



The existing part stack needs a unique ID



The ID allows referencing of an existing element from an extending fragment

Additionally, we need a second plugin to extend the first one. For this example, this second plugin needs the following dependencies:

- org.eclipse.e4.ui.model.workbench
- org.eclipse.e4.core.di
- javax.inject
- the plugin to be extended

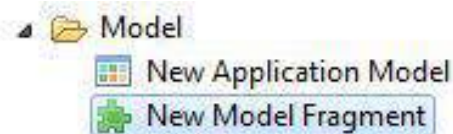
Model Fragment

A model fragment is nothing more than a small application model. It contains elements, which are supposed to be added to another application model.

The most comfortable way of creating a fragment is provided by the “Extract Fragment” wizard of the e4 model editor. It allows you to add elements in the application model at the place you want them to appear at the end, and extract them to an extending fragment afterwards. To try the wizard, add a new Part to the existing PartStack in your application model. Now right click the new Part and select “Extract into a Fragment”. In the following wizard, you can select “New” for the model fragment file and its container. Please note, that it usually does not make sense to place a fragment in the same bundle as the application model, but for modularity reasons in an additional bundle.

Alternatively to using the extract wizard, you can also create a model fragment from scratch. This might be necessary, e.g. if you cannot directly access the application model you want to extend.

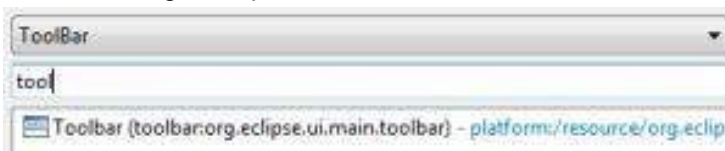
A file containing model fragments can be created using the wizard provided by the e4 tools. Although the editor says “New Model Fragment”, it will actually create a file, which can contain an arbitrary number of model fragments.



The extending plugin is set as a container for the model fragment file. After finishing the wizard, the model fragment file is opened in an editor which works similarly to the editor used to modify an application model.

The next step is to add a model fragment to the model fragment file. A model fragment has to define at which place the main application model is extended. This is done through an Element ID and a feature name. The Element ID defines which element of the main application model is extended, e.g. a tool bar. The feature name defines the containment reference to which the new element is added. For elements such as toolbars, menus, windows, or even the application, the feature is usually defined as “children”. If you are looking for the right containment reference, take a look at the core application model you want to extend. The containment references are typically shown as “folders” (child elements without an icon).

In the following example, a new element is added as a child of the existing toolbar.



The element ID defines which element is extended


```

ToolBar
├── children : List<ToolBarElement>
├── clonableSnippets : List<ApplicationElement>
├── curSharedRef : Placeholder
├── parent : ElementContainer
├── selectedElement : ToolBarElement
├── transientData : List<StringToObjectMap>
└── visibleWhen : Expression

```

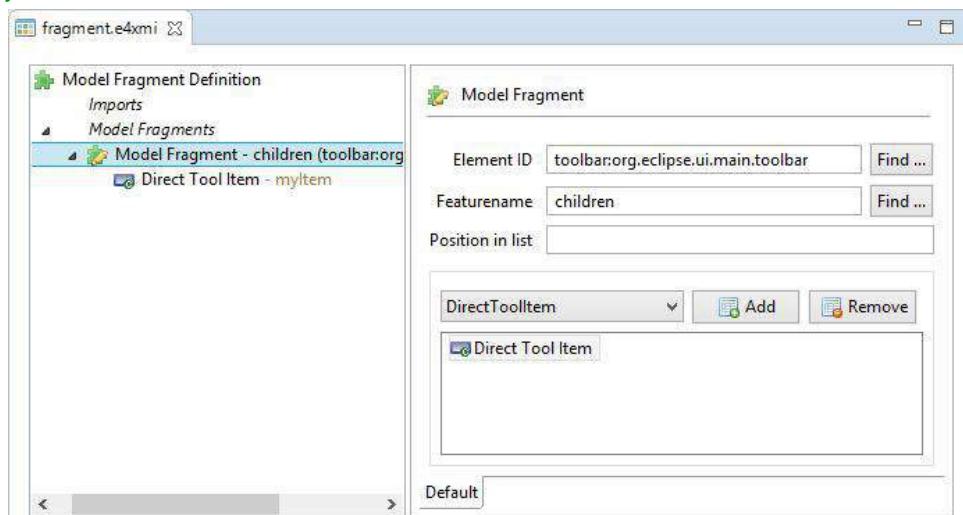
The feature name defines where the new element is added

The fragment now defines the location in the core application model, where elements will be added. As child elements of the new fragment, now application model elements can be created. All elements in a fragment will be added to the specified location in the core application model. As a simple example, we will add a Direct Tool Item as a child of the fragment (see Figure). To make it visible, a label or icon should be set. To trigger some action when the tool item is clicked, it should be linked to a handler. In the example, it is linked to a handler saying "Hello Eclipse!".

```

public class MyHandler {
    @Execute
    public void execute(Shell parent) {
        MessageDialog.openInformation(parent, "", "Hello Eclipse!");
    }
}

```



To connect the Direct Tool Item to the handler, the "Class URI" needs to point to the implementing class, located in the extending plugin. Of course, it is also possible to add more than one element to the core application model. If they are supposed to be added to the same

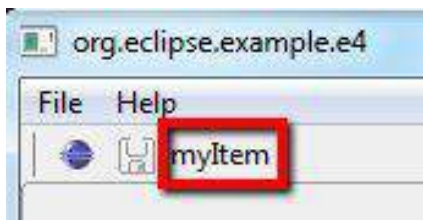
location, they can all be placed within one model fragment. If they need to be added to different locations, you will need several model fragments specifying those locations. However, the model fragments can still be placed in one model fragment file. For example, to add a tool item, a handler and a command, you can add three model fragments to one model fragment file.

In any case, the every model fragment has to be registered via an extension point. This will usually happen automatically, e.g. if you use the “Extract” wizard or the “New model fragment wizard”. There is an optional attribute “apply” for the extension, which controls if model fragments are merged into the core application model. There are three possible values:

- initial: The model fragments is only added to the core application model, if there is no persistent application model, typically, if the application is started the first time. On a second start-up, if the former state of the application model is loaded, the fragment will not be merged again
- notexist: The model fragments is only added to the core application model, if the elements added by the fragment are not already existing in the core application model
- always: The model fragments are always added. Please note, that this might lead to duplicated elements in the core, as elements in the model fragment are added again on every start-up

```
<extension id="id" point="org.eclipse.e4.workbench.model">  
  <fragment  
    apply="initial"  
    uri="fragment.e4xmi">  
  </fragment>  
</extension>
```

Finally, the new plugin adding the model fragment has to be added to the existing product configuration. Please note, that this will not happen automatically, nor will there be an error, if you forget this. The advantage is, that our application and the core application model has no dependency to our model fragment and its containing bundle. We can add, remove or replace it without breaking anything. After restarting the application, the tool item should be visible in the same way as a new part would be when added to the existing part stack.



Model Processor

In addition to being able to use fragments, it is also possible to programmatically extend the application model. In e4 this is accomplished using processors. Processors are especially useful if the extension needs to react to conditions within the existing application model, or if the existing application model is to be modified by an extension. Please note that you should prefer model fragments for all cases, where those two pre-conditions are not met. In general, programmatic modifications on the application model should be an exception, as they typically rely on some preconditions and are therefore not as robust as model fragments. Another disadvantage will be discussed in the conclusion of this chapter.

In the example application, we will add a new window that is positioned relatively to the existing window. The new window has the same height as the existing one and is positioned to the left of it. To free space for the new window, the existing window is moved right. To modify the application model, some experience with EMF is useful. A tutorial on EMF can be found under [this link](#). The following code shows the implementation of the described processor. The method to be executed is marked with the annotation `@Execute`, like in the implementation of handlers before.

Like in other implementing classes, required parameters can be injected, in this case the modified application model (`MApplication`) can be injected. All elements of the application model provide a corresponding Java interface to access it programmatically. All these interfaces are prefixed with an "M" (for "Model"). As an example, to access the application model element itself (the root node), you use `MApplication`, for a window "MWindow", respectively. All properties of an application model element are then accessible using simple getters and setter, e.g. "setHeight". References are accessible via a modifiable list, as an example the last line of the example adds the new window in the containment reference of the application. As processors have no context they are bound to, it is not possible to directly inject a window, it would not be unique for the platform, which window to inject. Instead, the model service is used to retrieve a window with a certain ID within the application. The ID to identify the existing window can be found in the application model editor. Please have a look at this tutorial to learn more about services in e4.

After moving the existing window, a new window is created by using the model service again.

```
public class Processor {  
  
    @Execute  
    public void execute(MApplication application, EModelService  
modelService){  
        MWindow existingWindow = modelService.find(String  
"IdOfExistingWindow", MUIElement application);  
        existingWindow.setX(200);  
        MTrimmedWindow newWindow =  
        modelService.createModelElement(MTrimmedWindow.class);  
        newWindow.setWidth(200);  
        newWindow.setHeight(existingWindow.getHeight());  
        application.getChildren().add(newWindow);  
    }  
}
```

Finally, the same as we did for the model fragment, the processor has to be registered via an extension point. The “beforefragment” attribute specifies, if the processors should be executed before or after all model fragments have been merged. In the example, after a restart of the application, the second window should open.

```
<extension id="id" point="org.eclipse.e4.workbench.model">  
    <processor  
        beforefragment="true"  
        class="org.eclipse.example.e4.extension.Processor">  
    </processor>  
</extension>
```

Conclusion

Model fragments and processors allow the extension of an existing application model. This supports the modular design of an application as new features including UI contributions, can be easily added or removed from an existing application. The definition of model fragments works in the same way as the definitions of the application model itself and does not require additional knowledge. The programmatic extension using processors uses a consistent EMF API and offers full flexibility.

In general, model fragments should be preferred over processors. First, they follow the same model-based approach to define elements as done in the core model using the application model editor. Second, model fragments are defined declaratively, so the platform can



understand its contents. Thereby, the platform can decide if elements must be merged based on a given directive (always, initial, etc.). To get this behavior with a model processor, it has to be implemented manually.

The next chapter of this tutorial will describe dependency injection in Eclipse 4. We will describe how to influence the injected parameters using different annotations, as well as how to trigger the injection manually.

Dependency Injection Basics

In the previous parts of this tutorial series we described how to create an application model, link those elements to implementations and how to extend the application model. This tutorial and all other parts of the series are now available as a [downloadable PDF](#).

In most of the programming examples provided so far, we implicitly used a common concept of Eclipse 4: dependency injection (DI). DI plays a central role in Eclipse 4, reason enough to devote a whole part of this tutorial to it. In this part, we describe:

- How dependency injection works in general
- Which objects can be injected.
- How the Eclipse Context works.
- Which annotations can be used to influence the injection.

Dependency Injection?

In web programming, dependency injection has been a hot topic for some time, with prominent representatives such as Google Guice or Spring. With Eclipse Version 4, Dependency Injection enters into the Eclipse world. It is essentially about how certain objects can access other objects from the outside. An Eclipse example would be the implementation of a view that needs a parent composite, an input object or a service, such as a logger.

To understand the concept and the motivation behind DI, we will use a metaphore, which has nothing to do with programming. If you are already familiar with DI in general, you might skip this section and continue with the next one already focussing on how to use DI in Eclipse.

So again, DI deals with the problem of how to retrieve certain objects while implementing something. A comparable example from the real world would be grocery shopping based on a shopping list. On the shopping list are all the things you need, in programming that would be all objects that you need to retrieve. Now, there are three ways to make the purchase.

Variant one would be to go to the appropriate stores and take the necessary goods from the shelves. In the world of programming this would correspond to the access to singletons. One must, however, know exactly where you can buy the required goods. Additionally, stores can close or relocate, the person using the store might move to another city, not being able to use the store anymore. In all cases, your way of getting goods would not work anymore. Therefore this option is very inflexible and requires detailed knowledge.

The second variant would be to order a pre-made food box that covers the shopping list as well as possible. The box is delivered to your door and you do not care where it comes from. Even if you move, you just need to change your address to still get the same box. However, you might get more or less than you really want. Missing something, you still need to go to the store. Second variant corresponds to the implementation of interfaces that are defined by a framework and thus “filled” by it. You are not that much bound to the framework (to the interface, though), but you need to be happy with the parameter set defined by the framework.

Version three, a hypothetical one, would mean that you just hung a piece of paper on your fridge, describing exactly what is needed. When you returned home, the fridge would be filled with exactly these goods. You get exactly what is needed and you do not have to be concerned about where it comes from. This is not (yet) possible in the real world, but through Dependency Injection it is possible in programming with Eclipse 4. The basic idea is therefore that classes specify for themselves which objects they need from the outside. The framework will then “inject” these objects.

Injected objects can be fields of a class, parameters of a constructor or parameters of a method that is called by the framework. In the simplest case, required objects are marked by the annotation `@Inject`. There are a number of additional annotations that control the behavior and the timing of the injection. For instance, the annotation `@Execute` marks a method in a handler which is called during the execution of the handler. The required parameters for this method will be injected:

```
@Execute
public void execute (MyObject requiredObject) {
    // Here is the actual handler code
}
```

The Eclipse Context

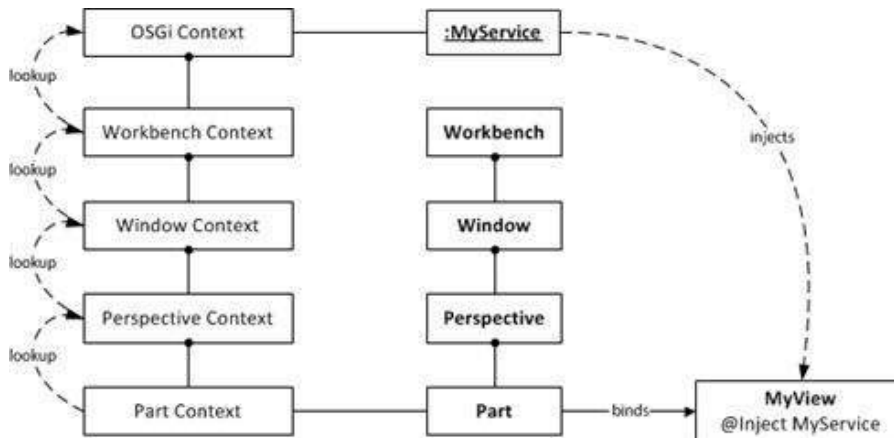
Having objects injected seems intuitive and practical, but leaves open the question where injected objects actually come from and how they are identified. So how does the framework determine which objects to inject at a certain place? In Eclipse 4 there is the so-called Eclipse context. This is a kind of list of objects that are ready for injection. Technically this context is a map of strings and objects. Without further information, an object is saved under its full class name, for example “org.eclipse.swt.Composite”. Now, when an object of a particular type is requested, the appropriate context will be searched. If it contains an object of the required type, this object is then used to call a constructor or a method, or to fill a field.



Dependency Injection using the Eclipse Contexts

However, in Eclipse 4, there is more than one global context – otherwise, it would be very difficult to identify the correct object to inject. In the example of a view that requires a composite as a parent, which composite to inject is ambiguous. Therefore, some elements of the Application Model have their own context during runtime, such as a window, a perspective or a part. These are hierarchically linked. For example, if an object is not found in the context of a part, the context of the perspective, or the window, the workbench and the OSGi context are

searched. The OSGi context contains objects that are valid for the entire application, such as services.



The Eclipse context is hierarchically linked.

Generally, all elements of the Application Context Models can be accessed along the ascending hierarchy. So, for example, in the context of a part, the window in which the part is contained can be injected:

```
@Inject
public void myMethod (MWindow window) {
}
```

Additionally, the context contains some SWT elements associated with the Application Model such as the composite of a part or the shell of the running application. Eclipse 4 services also ensure that certain commonly accessible objects, such as the current selection or the Eclipse Workbench Preferences are available in the context. The root context contains all OSGi services. Last but not least, you can insert your own items into the context.

@Named

If you wanted access to not only a specific type, but to the specific instance of a type, you can specify a name for the injection. This is done via the additional annotation `@Named`, which is used in combination with `@Inject`. `@Named` allows the additional specification of a string that defines the name of the object to be injected. In this case, the context will not search for the type of object to be injected, but for the corresponding string. Specifically, an injection without `@Named` is just a shortcut, where the type of a parameter is assumed to be the name of the variable to be injected. Conversely, objects which are placed in context with no further indication of a name, are available under the full class path. In the following example `@Named` therefore has no effect and could be omitted:


```
@ Inject
@ Named ("org.eclipse.swt.widgets.Composite")
Composite parent;
```

Eclipse 4 services automatically include some objects in the context under certain IDs. The valid names are found in the interface `IServiceConstants`. In this manner, for example, the active shell is injected.

```
@ Inject
@Named(IServiceConstants.ACTIVE_SHELL)
Shell shell;
```

Furthermore, it is possible to store custom objects with a name in context. We will describe this in more detail in a following chapter of this tutorial.

@Optional

Of course, it is always possible that a context does not contain a matching object that can be injected. In this case, the dependency injection mechanism of Eclipse 4 displays an error. More specifically, an exception is thrown. A class that needs the missing parameter in a constructor, a method or as fields that can not be injected, will not be initialized correctly.

However, some parameters are not needed in every case, for example, the active selection of specific services. For these parameters, the annotation `@Optional` can be used. If an object marked with `@Optional` is not available in the context, null will be injected. In this case, prior to access to the injected object in your custom code, it has to be checked, if it is null.

@Active

In certain use cases, it is necessary to access not only a specific type of an element from the application model, but an object from the active context. With the annotation `@Active`, the currently active context is used for the injection. Context usually get activated, when the corresponding UI element gets activated. The following example injects the part of the active context, which is typically also the active part.

```
@Inject
public void save (@Active MPart part) {
    partService.save (part);
}
```

Injecting Objects

In the simplest case, the injection is triggered via the `@Inject` annotation. It can be placed prior to methods, before the constructor or class fields. If you mark a method or constructor with `@Inject`, all their parameters are injected. Without further information with `@Named`, the lookup for the right object will be according to the corresponding type of the parameter or field. There are a number of additional annotations, which control the exact time of the injection, but they behave in

principle the same as `@Inject`. We will describe the additional annotations in another part of this tutorial.

A crucial role is played by the order of injection. When a class is instantiated, for example, a view, the constructor and its parameters are injected first. Immediately following, the relevant fields are

injected. As a consequence, fields can not be accessed in the constructor. Parameters of methods are injected when these methods are called by the framework. All methods marked with `@Inject` are also called to initialize the object after the constructor and the fields. If the injected object changes in context, it is re-injected. Methods are therefore called again, when the injected values change.

Constructors

Constructors should include parameters that are essential for the existence of an object. Any unnecessary parameter limits the testability and as well, the reusability of an object. Particularly the object initializations should be done in separate methods that are called after the constructor. A typical example of dependency injection in the constructor is the injection of the Parent composite of a view, as has been described previously in this tutorial. Since views are initialized in the context of a part of the application model, the specification of the type composite in this case is clear and no additional annotations are required.

```
@Inject
public void MyView (Composite parent) {
    //Implement the View placed on the Parent
}
```

Fields

After the constructor of a class, the class' fields are injected. A typical application is the injection of services that will be available globally in the class. An example of this is the Selection Service to set the current selection of a view. As services usually exist only once per application, as in this case, the indication of the type is sufficient.

```
@Inject
ESelectionService service;
...
service.setSelection (mySelection);
```

Injected fields must not be marked as final, as they can potentially be re-injected. Final fields must be explicitly set through the constructor and its injection.

Methods

After the constructor and the fields, while initializing a class, all annotated methods (with `@Inject`) are sequentially called. This also applies to methods that have no parameters. If one of



the injected parameters of a method changes afterwards in the context, the method will be called again with the new parameters. A good example of an injection in methods is the current

selection, on which you often want to respond in a view or in a handler. In this case, however, specifying the type of the parameter is not enough, the parameter must also be marked with the annotation `@Named`. The following example also uses `@Optional` because, for example, when the application is started no selection will be in the context. In the following example, the injection is repeated every time and the method is called again when the selection changes.

```
@ Inject
public void setSelection (@ Named (IServiceConstants.ACTIVE_SELECTION) @ Optional
MyObject myObject) {
//Process Selection
}
```

Conclusion

Dependency injection reduces dependencies on singletons and framework interfaces. Objects define exactly which parameters or services they use. This also makes testing comparatively easy. The use of additional annotations allows more precise specification of objects to be injected, for example, marking certain parameters as optional.

A next part of the series is dedicated to even more details about dependency injection. We will cover additional annotations such as `@PreDestroy` and `@PostConstruct`. With these, you can instruct the framework, at which time certain methods should be invoked without creating a direct dependency on particular framework classes.

Behavior Annotations

In the previous parts of this tutorial series, we described how to create an application model, link those elements to implementations and how to extend the application model. This tutorial and all other parts of the series are now available as a [downloadable PDF](#). In the last part of this tutorial, we provided details about dependency injection. However, we focused on how to influence which parameter is injected at a certain place. In many cases, it is additionally important to specify when exactly parameters are injected, or more precisely, when certain methods of a class are called by the framework. Eclipse 4 uses annotations for this purpose. This tutorial describes the most important annotations used in Eclipse 4.

When To Inject?

The annotation `@Inject`, in combination with `@Named` and `@Optional`, described in a previous part of this tutorial, is sufficient to control dependency injection for constructors and fields. In both cases, the point in time when objects are injected is clear (instantiation of the class). When methods are marked only with `@Inject`, these methods are called once after the class is initialized and again every time a parameter changes in the context. However, there are many use cases wherein the developer may want to react to certain events, e.g., if a view gets the focus or if an object is disposed. The Eclipse 3.x interfaces defined methods for these events, e.g., `setFocus()`, that were called by the framework when a certain event was triggered. In Eclipse 4, views are POJOs, and methods can be named arbitrarily. Therefore, methods that need to be called by the framework at a certain point in time must be marked with corresponding annotations, e.g., `@Focus`. All the described annotations include the dependency injection as `@Inject` does. That means that if a method is marked with any of the annotations below, all parameters of the method will be injected without an explicit addition of `@Inject`.

`@PostConstruct` and `@PreDestroy`

In many cases, objects need additional initialization after the constructor has been called. This is especially relevant if fields are used. Since fields are injected after the constructor is called, any initialization dependent on fields cannot be done in the constructor.

A typical task for an initialization of an object is the registration of listeners. These listeners typically need to be unregistered if the object is not needed anymore. Eclipse 3.x interfaces typically provided methods such as `init()` and `dispose()` for this use case. Eclipse 4 uses two standard annotations defined in `javax.annotation`: `@PostConstruct` and `@PreDestroy`.

A method annotated with `@PostConstruct` is called after a class is initialized with its constructor and after all fields have been injected. A method annotated with `@PreDestroy` is called when an object is not needed anymore, e.g., when the corresponding view is closed but before the object is destroyed. As mentioned before, all annotations allow the use of additional parameters in these methods, but that is not mandatory. The following code example shows a typical use case. A service is injected as a field and can therefore not be accessed in the constructor. The

`@PostConstruct` method is used to register a listener on the service, the `@PreDestroy` method to deregister the listener.

```
@Inject
MyService service;

@PostConstruct
public void postConstruct() {
    service.addListener(this);
}

@PreDestroy
public void preDestroy() {
    service.removeListener(this);
}
```

`@PostConstruct` and `@PreDestroy` can be used for all classes, which are initialized by the framework or manually using the Injection Factory.

@Focus

For visual elements, e.g., parts, there are additional events to which an implementation should react. A method marked with `@Focus` is called when the corresponding UI element receives the focus. In SWT applications, the focus must be forwarded to the central SWT element, e.g., a text field or a tree. If the implementation of a view contains several SWT controls, the developer has to choose a control, typically the first text field if it is a form editor.

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

@Persist

The annotation `@Persist` marks a method to be called if a save is triggered on a part. For example, if the parts represent a text editor, the content of the text control is saved into a file.

```
@Persist
public void save(){
    //save the context of the part
}
```

The method is typically called from another place than the part itself, e.g., from a handler. The `EPartService` provides helper methods to save a specific part or all parts that are dirty:

```
@Execute
public void execute(@Named(IServiceConstants.ACTIVE_PART
MPart part, EPartService partService) {
    partService.savePart(part, false);
}
```

@PersistState

A method marked with `@PersistState` is called before an object is disposed and before the method marked with `@PreDestroy` is called. The purpose of this method is to persist the latest state of an element if required. If the method is a view, the latest input by the user could be stored for convenience.

@Execute and @CanExecute

There are two additional annotations used especially for handlers, `@Execute` and `@CanExecute`. `@Execute` marks the method to be executed if the handler itself is executed. `@CanExecute` marks the method responsible for the enable state of the handler. Therefore, the `@CanExecute` method needs to return a Boolean value, which tells the framework whether the implementation action is currently available or not. As a consequence, Eclipse 4 will enable or disable all menu and toolbar items linked to this handler. As for all annotations, all required parameters are injected.

However, the annotation `@CanExecute` works quite differently than other annotations. It is not called on a certain event or on a change of one parameter in the context. In fact, in version 4.4, it is called continuously and is timer-based, so it is important to not spend too much execution time within this method.

A very common example for the implementation of a `@CanExecute` method is a check for the current selection, the active part or the active perspective. The following example checks whether

the current selection is of a certain type and enables the handler if it is. The `@Execute` method invokes a certain action on the current selection:

```
@CanExecute
public boolean canExecute(@Named(IServiceConstants.ACTIVE_SELECTION)
@Optional Object selection) {
    if (selection != null && selection instanceof MyObject)
        return true;
    return false;
}
```

Lifecycle Annotations

Finally, Eclipse 4 offers the possibility to hook into the lifecycle of a running application. To do this, a lifecycle handler needs to be registered as a property of the registered application:

```
<property name="lifeCycleURI"  
value="platform:/plugin/helloworld/helloworld.LifecycleHandler">
```

The implementation of the lifecycle handler itself is a POJO. It supports the following specific annotations:

@PostContextCreate

Is called after the application's context has been created. Can be used to add or remove objects from the context.

@ProcessAdditions and @ProcessRemovals

Allows the modification of the application model before it is passed to the renderer that will display the application on screen. Allows the addition and removal of application model elements before the application is actually shown.

@PreSave

Is called before the application model is persisted. Allows the modification of the model before saving it.

Conclusion

Behavior annotations of Eclipse 4 allow the specification of the precise point in time when objects are injected. Annotated methods can require parameters but don't have to. For example, a method annotated with `@Focus` often does not require any parameters. In this case, it is more important that a focus method is called at a certain point in time when the corresponding UI element gets the focus. Some annotations, such as `@Inject`, `@Named`, `@PostConstruct` and `@PreDestroy`, are Java standards. Additional annotations, such as `@Optional` or `@Persist`, are specific for Eclipse 4.

To get an overview of the source of the available annotations, the following list shows all described annotations with the bundle defining them. If you use any of these annotations, you will need a dependency or a package import to these bundles.

@Active	org.eclipse.e4.core.contexts
@Creatable	org.eclipse.e4.core.di.annotations
@CanExecute	org.eclipse.e4.core.di.annotations
@Execute	org.eclipse.e4.core.di.annotations
@Inject	javax.inject
@Named	javax.inject
@Optional	org.eclipse.e4.core.di.annotations
@Persist	org.eclipse.e4.ui.di
@PersistState	org.eclipse.e4.ui.di
@PostConstruct	javax.annotation
@ProcessAdditions	org.eclipse.e4.ui.workbench.lifecycle
@ProcessRemovals	org.eclipse.e4.ui.workbench.lifecycle
@PostContextCreate	org.eclipse.e4.ui.workbench.lifecycle
@PreDestroy	javax.annotation
@PreSave	org.eclipse.e4.ui.workbench.lifecycle

Services

In the previous parts of this tutorial series, we described how to create an application model, link those elements to implementations, how to extend the application model, details about dependency injection and how to use behavior annotations. This tutorial and all other parts of the series are now available as a [downloadable PDF](#).

In the last two parts of this tutorial, we described a lot of details about dependency injection. However, dependency injection is only a technique; the major goal is to get access to certain objects you want to use in a class. As we learned in part 4 of the tutorial, one type of objects you can retrieve using dependency injection are services. Services play a very central role in Eclipse 4. They provide framework features such as managing a selection or opening a perspective. In this

part of the tutorial, we described the three most important Eclipse 4 services: the selection service (ESelectionService), the model service (EModelService) and the part service (EPartService). Using these services also serves as a blueprint for how to use any other services in Eclipse 4, too.

Why Services?

One of the key strengths of a framework such as Eclipse has always been the possibility of reusing of a lot of framework functionality. That means Eclipse as a framework already implements a lot of features typically required in applications. Therefore, developers don't have to reinvent the wheel and can focus on implementing specific and valuable parts of an application. In Eclipse 3.x, a lot of these framework features were provided in the workbench API and in the use of singletons. For example, it was possible to retrieve the current selection of an application using this line of code:

```
PlatformUI.getWorkbench().getActiveWorkbenchWindow().getSelectionService().getSelection();
```

This approach had several drawbacks. Since we described the issues in more detail in part 5 of this tutorial, here we will recap only the three major problems:

- You need to know exactly where in the workbench API a certain method can be accessed. There is no real separation of concerns, so you basically need to know the complete workbench API. This high level of complexity has always been a problem, especially for beginners.
- It is very difficult to create a mock for a certain object of the workbench, which is required for testing.
- It is difficult, sometimes even impossible, to replace an existing implementation with an own one if you want to adapt or extend the default behavior of the framework.

Those are the main reasons why Eclipse 4 has chosen a different concept of providing framework functionality. Instead of providing one big API, framework features have been split into a number of services. Every Eclipse 4 service has a specific focus, e.g., managing the



selection or dealing with parts and perspectives. All services can be accessed using dependency injection. The following line of code injects the Eclipse 4 selection service as a field:

```
@Inject  
ESelectionService selectionService;
```

See parts 4 and 6 for more details about dependency injection.

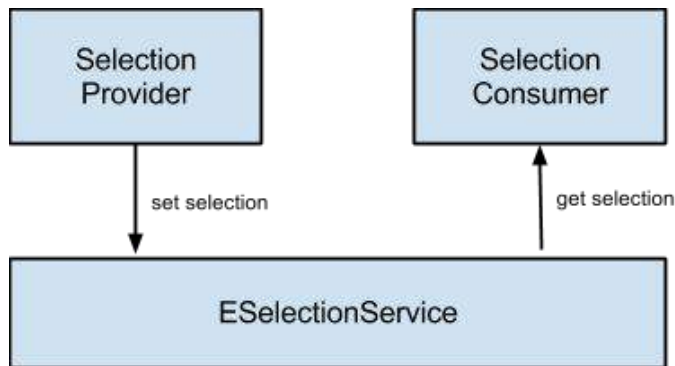
To be fair, the idea of using services is nothing new and nothing specific to Eclipse 4. However, Eclipse 4 has really adapted the concept of a service-oriented architecture. This provides mitigations to the above mentioned problems:

- As services can be injected, you only need to know which service you want to use, more precisely its Java interface, not where it comes from. This is far simpler since you can search for something you want to do, e.g., “managing a selection” and you will very likely find the selection service. Additionally, you only have to deal with the API of services you want to use, so there is a good separation of concerns.
- One focus while developing the services for Eclipse 4 was to create an easy to use and cleaner API than 3.x. The 3.x API has grown over the years and the gathered experience was used to design the API for Eclipse 4.
- Services can be replaced, even during runtime. That allows to create mock implementations for testing, as well as replacing existing services with custom implementations.

However, not all 3.x functionality has been transformed into Eclipse 4 services. The platform team has obviously focused on the most important things. Some features of Eclipse 3.x don't even have to be provided explicitly for Eclipse 4, since things such as the application model already provide them or at least make it much easier to implement them. In this tutorial, we focus on the most important services of Eclipse 4. The list is obviously not complete; we plan to add more in the future. If you would like a certain service to be described in more detail, please feel free to [get in contact with us](#).

Selection Service (ESelectionService)

The selection service is responsible for managing the active selection of an application. The selection of an application is typically an object a user can select from a view, e.g., a file for a file browser or an e-mail in an e-mail application. There are two different users of the selection service. Selection providers are elements that set the active selection. Selection providers are typically UI elements that allow the user to select an element, e.g., a tree viewer. The selection is then forwarded to the selection service. Selection consumers are interested in the element currently selected in an application. For example, the enabling of a handler can be dependent on the current selection. Thus, the enabling of the handler “Export File” could only be enabled if the current selection is a file.



Eclipse 3.x vs. Eclipse 4 - Which Platform to use?

In the previous parts of this tutorial, we introduced the Eclipse 4 Application Platform including its features such as the Application Model, Dependency Injection, and the available services. All of those features provide great support for developing applications and in general Eclipse 4 does many things better than Eclipse 3.x. However, that does not mean that Eclipse 3.x, more precisely

its API is decrepit. There are tons of existing plugins out there based on this API and they are still actively developed. The implementation of Eclipse 4 does not mean, that the 3.x API is no longer useful. This is especially true in the tools area, many of them, e.g. the java development tools, are and will probably stay on the 3.x API for a long time. Therefore, before you start an application, you will have to deal with the question of which API it is based. In the following sections, I will describe the most important options in detail.

Option 0 - Eclipse 3.x

This would mean you use an old version of the Eclipse platform (highest one is 3.8). Those versions do not contain any components from Eclipse 4. This option is only valid for existing tools, which are not actively developed anymore. There are no updates nor service releases for the 3.x stream anymore. Sooner or later, you will need to update to option 1, e.g. if the contained SWT version no longer supports your windowing system or if you need other fixes. The only alternative is to get [Long Term Support](#) for your Eclipse 3.x version. New projects should not use 3.x. We will not describe this option in more detail.

Option 1 - 3.x Compatibility Layer (3.x API) on Eclipse 4.x

This is the option used by most tools and applications which existed before Eclipse 4. The compatibility layer enables 3.x applications to run on the new Eclipse 4 platform without any code adaptation. Most existing projects use this option as a first step towards Eclipse 4. Besides the easy migration, you can still use all existing components and frameworks, even if they are not migrated to e4. Finally, your application stays backwards compatible, meaning it can still be run on 3.x



To ease migration, the compatibility layer provides the 3.x workbench API and translates all calls into the programming model of e4. In the background, it transparently creates an Application Model. For example, if the 3.x application registers a 3.x view using an extension point, the compatibility layer will create a Part for this view. One important criteria for existing applications to work well on the compatibility layer is that they should not use any internal workbench API. Aside from this, there should be no source code changes required. However, you will probably need to adapt the product or run a configuration of the application. Eclipse 4 needs additional plugins to work, and as there are no direct dependencies, this will not be automatically discovered. These are the plugins you will need to add:

org.eclipse.equinox.ds : The OSGi plugin enabling declarative services

org.eclipse.equinox.event: The OSGi event broker

org.eclipse.equinox.util: Required by the first two

org.eclipse.e4.ui.workbench.addons.swt: Enables features such as minimizing and maximizing parts.

We have mentioned before, that “there should be no source code changes required”. However, whether this is true, depends on the project and more precisely on the usage of the workbench API. In general, if you only rely on official API, your code should compile well and the basic functionality should generally work. However, there are things in Eclipse 4 which behave a little different than in Eclipse 3.x. The reason is obviously the reimplementations of all internals. These

differences could be slight changes in the look and feel or the behavior, which implicitly worked on 3.x, but is not really guaranteed in the API. The only way to find out about those issues is to try to test an existing application based on the compatibility layer. Sometimes it is required to fix some remaining minor issues on the Eclipse 4 platform to fully support your application. It is impossible to estimate the required effort for migrating an existing application on the compatibility layer, without evaluating it. Some applications run without any adaptations, for others you need a few days or even weeks of work. For supporting your migration project, we offer [developer support and sponsored development as professional services](#).

An obvious disadvantage of using the compatibility layer is that you won't benefit from the new concepts, such as the application model, dependency injection and annotations provided by e4. Although, some other improvements will still work, such as CSS styling.

Option 2 - 3.x Compatibility Layer on Eclipse 4.x with some e4 components

There can be good reasons to keep an application on the compatibility layer, meaning the 3.x API. This is especially true for existing applications, as it requires almost no migration effort. Additionally, the Eclipse IDE itself is still based on the 3.x API, so if you extend it, you automatically need to use the compatibility layer.

However, it would still be useful to use some of the benefits of the Eclipse 4 programming model, especially for newly developed components. Additionally, it would be nice, if components could be developed in a way such that they can be used in a 3.x as well as in a pure Eclipse 4 application.

There are ways to integrate Eclipse 4 components into an 3.x application based on Eclipse 4.x. That enables you to use the Eclipse 4 benefits for newly developed components and still reuse all existing plugins. More details are described in the section “Soft Migration to Eclipse 4.x” This approach allows you to develop new parts of the application using the benefits of the Eclipse 4 programming model and as well as reuse all existing components. Further, the views developed in this way can be integrated into any pure e4 application without any adaptations.

Option 3 - A “pure” or “native” Eclipse 4 Application

The third option, primarily interesting for new projects, is to build a pure Eclipse 4 (e4) application without any compatibility layer. Any existing parts of an application should be completely migrated to e4. The major disadvantage of this option is that many existing components and frameworks cannot be reused. This affects components doing UI contributions such as Views. Examples would be the Error Log, the Console View, or existing editors. To use them in an e4 application they would have to be migrated to e4 as well. However, components without any workbench contributions should work in a pure e4 application. The advantage of this approach is obviously that you will have very clean design and you benefit from all the concepts in Eclipse 4, such as the Application Model, dependency injection, and the e4 services.

Option 4 - A “pure” Eclipse 4 Application integrating some 3.x components

In this option you would develop a pure Eclipse 4 application and reuse some 3.x components. Components without workbench dependencies can typically easily be integrated into a pure e4 application. Components with workbench dependencies have to be adapted. Depending on the

component, this adaptation is often not much effort. Even UI components can often be easily reused or adapted to work with e4. However, this needs to be evaluated individually for each component. The good news is that if you rely on open source components, everybody is able to adapt those. To help you with evaluating and adopting existing components to Eclipse 4, we offer [developer support and sponsored development as a professional services](#).

Conclusion

In the end, when and how to migrate to e4 is still one of those “it depends...” decisions. Probably the most important criteria is the number of existing components and the number of reused third-party components. If you use many existing components you require the 3.x API, and you should probably go for option 1 or 2. If you do not use a lot of 3.x based components, or if all of those components can easily be migrated, you should go for option 3 or 4. If you have additional options for migrating or mixing the two technologies, let me know and I will gladly add it to this post.

Soft migration from 3.x to Eclipse 4 (e4)

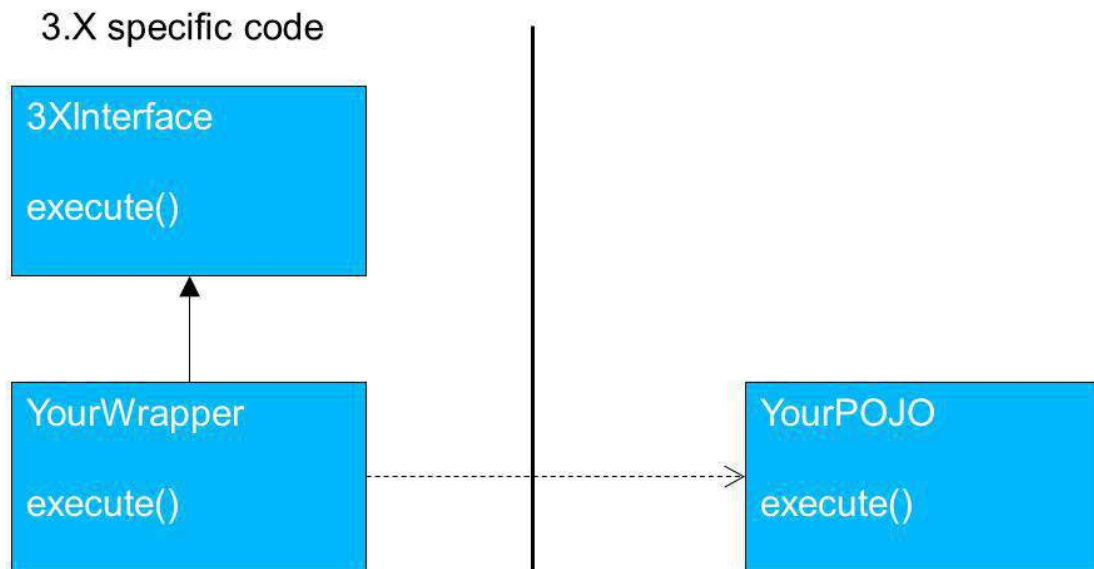
This tutorial describes how to do a soft migration to the Eclipse 4 (e4) programming model. The basic goal of the tutorial is to enable development using the new concepts such as Dependency Injection, and Annotations, but without first requiring a complete application migration. So the application is still based on the compatibility layer, but it includes some components following the Eclipse 4 programming model. As the compatibility layer is used, all existing plugins as well as frameworks which require the 3.x API can still be used as before. However, developing new UI components for an application following the e4 programming model has two major advantages:

1. The new components are POJOs and therefore very flexible, testable, and reusable.
2. If the application is migrated to the Eclipse 4 Application Platform, these components are ready to be used in e4.

Interestingly, the first point is worth taking advantage of, even if you are sure that Eclipse 4 will not be an option in the near future. The idea is actually pretty simple and isn't really new at all. There are basically two options for how to follow these concepts: with or without dependency injection. To explain the basic idea, we will first introduce the manual approach, without dependency injection and, in the subsequent section, introduce the Eclipse 4 like approach with dependency injection.

POJOs in 3.x (without Dependency Injection)

The basic concept is to make a clear separation between the code which you develop for a custom application and the code that binds your component into the Eclipse workbench. The second component depends on the workbench API and is therefore specific to a certain Eclipse version, i.e. 3.x or 4.x. The first group of code does not need to be specific to an Eclipse version and in fact, doesn't need to know about the workbench at all. Therefore, it is easy to test and reusable in any Eclipse version. In the following section we explain the basic idea based on the example of a handler implementation.



A good example for the separation is the implementation of a handler.

To implement a handler in Eclipse 3.x that is bound to a command, you need to implement the interface IHandler. Let's look at a typical example handler in 3.x, which does something with the current selection. In this example, the handler checks if the current selection is of type "MailAccount". If this is true, the handler checks if the user is already logged in and subsequently sends and receives mails.

```
public Object execute(ExecutionEvent event) throws ExecutionException {
    ISelection currentSelection = HandlerUtil.getCurrentSelection(event);
    if (currentSelection instanceof IStructuredSelection) {
        Object firstElement = ((IStructuredSelection) currentSelection)
            .getFirstElement();
        if (firstElement instanceof MailAccount) {
            MailAccount account = (MailAccount) firstElement;
            if(!account.isLoggedIn()){
                account.logIn();
            }
            account.sendMails();
            account.recieveMails();
        }
    }

    return null;
}
```

There are three major problems with this design: boilerplate code, lack of testability, and lack of re-usability. Let's imagine that you would like to write a test case for this handler. You need to manually create an ExecutionEvent and also make sure that the HandlerUtil is available in your test environment. Since the selection in this case is not a plain field, but rather a property, you would need to look at the implementation of HandlerUtil.getCurrentSelection() to find out how to properly prepare your Mock ExecutionEvent. Even if you manage to create a test case, let's imagine you want to trigger a timer-based mail synchronization, meaning that you want to directly call the execute method. In order to re-use the handler, you would again need to create an ExecutionEvent. If the handler is within your control, you will probably refactor at this time. However, the handler might be within a framework where you cannot refactor. The solution for this is pretty simple: we split the code into two methods. The first will deal with all workbench specific parts, i.e. unpacking the selection. The second method will execute the business logic itself and can, in this case, be static.


```
public Object execute(ExecutionEvent event) throws ExecutionException {
    ISelection currentSelection = HandlerUtil.getCurrentSelection(event);
    if (currentSelection instanceof IStructuredSelection) {
        Object firstElement = ((IStructuredSelection) currentSelection)
            .getFirstElement();
        if (firstElement instanceof MailAccount) {
            synchronizeAccount((MailAccount) firstElement);
        }
    }

    return null;
}
```

```
public static void synchronizeAccount(MailAccount account) {
    if (!account.isLoggedIn()){
        account.logIn();
    }
    account.sendMails();
    account.receiveMails();
}
```

With this design it is much easier to write a test case for the second method. Additionally, the method can be easily called from anywhere else, e.g. triggering the timer-based synchronization. Moreover, the code is easier to understand. As a next step, the second method can be moved out of the handler, for example, into a plugin which does not have any workbench dependencies.

Applying the same design pattern to views will result in the same advantages. We have one class implementing the workbench specific parts and one class which can be a POJO. In the following example, the WorkbenchView does all workbench specific parts, including handling the current selection, while the POJOView is completely independent.

```
public class WorkbenchView extends ViewPart {

    private POJOView pojoView;

    public WorkbenchView() {
        pojoView = new POJOView();
    }

    @Override
    public void createPartControl(Composite parent) {
        pojoView.createPartControl(parent);
    }
}
```

```
ISelectionService service = (ISelectionService) getSite().getService(  
ISelectionService.class);  
service.addSelectionListener(new ISelectionListener() {
```

```
@Override  
public void selectionChanged(IWorkbenchPart part,  
ISelection selection) {  
    if (selection instanceof IStructuredSelection) {  
        Object firstElement = ((IStructuredSelection) selection)  
            .getFirstElement();  
        pojoView.setInput(firstElement);  
    }  
}  
});  
  
}
```

```
@Override  
public void setFocus() {  
    pojoView.setFocus();  
}  
  
}  
public class POJOView {
```

```
private Text text;
```

```
public void createPartControl(Composite parent) {  
    text = new Text(parent, SWT.NONE);  
}
```

```
public void setFocus() {  
    text.setFocus();  
}  
  
}
```

```
public void setInput(Object object) {  
    if(object!=null){  
        text.setText(object.toString());  
    }  
}  
}
```

Again the POJOView is now very easy to understand, test and re-use. As an example, the POJOView could be embedded into a JFace Wizard. Until this point, we have not used any Eclipse 4 specific concepts, nor any dependency injection; the pattern can be used in plain 3.x. It is a very general pattern which we recommend to follow in order to make your custom components more reusable and testable.

As the wrapper classes (the one which implements the 3.x interface) always look pretty similar, it would be easy to provide a few generic implementations. We will introduce such a generic implementation later in this tutorial that uses dependency injection.

POJOs in 3.x (with Dependency Injection)

If you separate workbench specific code and custom components as POJOs, as shown before, components are easier to reuse and to test, even in 3.x. However, there are still two disadvantages compared to developing a component for the Eclipse 4 Application Platform:

1. The wrapper has to be manually implemented
2. The implementation of the component cannot use dependency injection and therefore, is not ready to be used in Eclipse 4.

Once your application is running on an Eclipse 4.x version there are solutions for this even if you still use the 3.x API (compatibility layer). That means, you can implement your POJO component as before, but additionally, you can use dependency injection and do not need to implement a wrapper to connect your POJO with the workbench.

There are three ways to connect POJO views using dependency injection into a compatibility layer based application:

1. Use the 3.x extension point (only available for views)
2. Use fragments or processors
3. Use the 3.x e4 bridge from the tools project

The first option is available only for views since Luna. The existing 3.x extension point has been extended by the possibility to register “e4views”. This entry in the default extension point does not point to an implementation of IViewPart (3.x interface to be implemented by view), but it points to a POJO class. This POJO class can use all the Eclipse annotations for dependency injection. The following example shows the two extensions available to register views. The first one is a traditional 3.x View implementing IViewPart. The second extension registers a POJO implementation using dependency injection. It will be added to the workbench, just like any other view.

```
<extension
  point="org.eclipse.ui.views">
  <view
    name="View"
    class="myrcpapp.3xViewImplementation"
    id="myRCApp.view">
  </view>
  <e4view
    class="myrcpapp.POJOViewImplementation"
    id="myRCApp.e4view"
    name="E4View"
    restorable="true">
  </e4view>
</extension>
```

The second option is to use processors and fragments to add elements to the application model created by the compatibility layer. This option is very appealing because not only can you use dependency injection for the implementation, but you can also model the things you want to contribute as elements in the fragment. However, there are currently still some timing problems. When processors and fragments are being processed, the compatibility layer has not yet created the complete application model. (See this bug report. (https://bugs.eclipse.org/bugs/show_bug.cgi?id=376486)). Therefore, this option might work for handlers and views, but currently it doesn't work for editors or things which extend perspectives defined by the IDE.

The third solution is provided by the 3.x e4 bridge from the e4 tools project. The plugin basically provides generic wrapper classes, which can be used in a 3.x application. The wrapper classes allow the definition of a second class, which is a POJO, and implements the corresponding component. The solution follows the same pattern we describe before, but works in a generic way and supports dependency injection. At the time of writing, implementations for Views, Editors, and Handlers are available. To create the 3.x workbench wrapper, one simply inherits from the respective type, e.g. from `DIViewPart` to implement a `View`. The wrapper class is almost empty. It only has to specify the POJO class that implements the component.

```
public class ExampleViewWrapper extends DIViewPart{
public ExampleViewWrapper() {
  super(ExampleView.class);
}
}
```

This class is now registered using the view's extension point as is usual in 3.x (not with the previously mentioned `e4view` extension).

The implementation of the view itself can be a POJO and dependency injection can therefore be used. In addition to being quite convenient to develop, in case the component is migrated to e4, it is ready to be used without any adaptation. In this case, you can remove the wrapper and the extension to integrate the POJOView into the application model. As you can see, the view can use all features of dependency injection, including injection into the current selection

```
public class ExampleView {
    private Label label;
    @Inject
    public ExampleView(Composite parent){
        label = new Label(parent, SWT.NONE);
        label.setText("Hello World");
    }

    @Inject
    public void setInput(@Optional @Named(IServiceConstants.ACTIVE_SELECTION)Object
input){
        if(input==null){
            return;
        }
        label.setText(input.toString());
    }
}
```

To understand how this works, we look at the simplest case, a wrapper for a Handler. To simplify the example, we will ignore the annotation `@CanEnable` for now. The DIHandler needs to implement the 3.x IHandler interface allowing it to be registered with the handler extension point, as is common in 3.x. Additionally, the DIHandler needs to know about the POJO class that it wraps. This POJO class should be instantiated by the wrapper. To do this, we use the e4 ContextInjectionFactory. As the application is running on the compatibility layer, we can retrieve the EclipseContext as a service and use it to create the class. This way, all fields expected by the Handler are being injected (as is standard in e4).

```
public class DIHandler extends AbstractHandler {  
  
    private Class clazz;  
    private C component;  
  
    public DIHandler(Class clazz) {  
        this.clazz = clazz;  
        IEclipseContext context = getActiveContext();  
        component = ContextInjectionFactory.make(clazz, context);  
    }  
    private static IEclipseContext getActiveContext() {  
        IEclipseContext parentContext = (IEclipseContext) PlatformUI.getWorkbench().getService(  
            IEclipseContext.class);  
        return parentContext.getActiveLeaf();  
    }  
}
```

The only missing piece now is the implementation of the execute method. It simply uses the InjectionFactory again to invoke the method of the POJO, which is marked with @Execute:

```
public Object execute(ExecutionEvent event) throws ExecutionException {  
    return ContextInjectionFactory.invoke(component, Execute.class,  
        getActiveContext());  
}
```

This DIHandler is not very complex and allows wrapping POJO handlers into the 3.x workbench.

Conclusion

This part of the tutorial described different approaches for a soft migration from 3.x to the Eclipse 4 programming model. We started with the concept of separating the implementation of custom UI components and workbench specific classes. This improved the re-usability and the testability of the components. It is a pattern you should follow, even if you never want to migrate to Eclipse 4. It can be used with or without dependency injection. When using dependency injection, there are three ways to integrate the POJOs into the 3.x workbench. For views, the 3.x extension point allows you to directly register them. For other elements, you will need to use the 3.x e4 bridge provided by the e4 tools or fragments.