

Eclipse 4 Tutorial

The Eclipse 4 Application Model

From the Application Model to the Implementation of Views

Extending the Application Model

Dependency Injection Basics

Behavior Annotations

Eclipse 4 Services

Appendix A: Migrating from Eclipse 3.x to Eclipse 4 (e4)

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

Jonas Helming
Maximilian Koegel

September 2013



Table of Contents

1. The e4 Application Model.....	4
1.1. Application Model vs. Views	4
1.2. Installation	4
1.3. The first step.....	5
1.4. The Editor.....	5
1.5. Live Editing.....	6
1.6. Programmatical Access to the Model.....	7
1.7. Scripting	8
1.8. Conclusion.....	8
2. From the Application Model to the Implementation of Views	9
2.1. An application model without views?	9
2.2. A view without an application model?	9
2.3. The Reunion	10
2.4. Another Separation (Adding a Handler)	11
2.5. Conclusion.....	13
3. Extending the Application Model	14
3.1. Only one model?	14
3.2. Warm-Up	15
3.3. Model Fragment	16
3.4. Model Processor.....	17
3.5. Conclusion.....	18
4. Dependency Injection Basics	19
4.1. Dependency Injection?	19
4.2. The Eclipse Context	20
4.3. @Named.....	21
4.4. @Optional	21
4.5. @Active.....	22
4.6. Injecting Objects	22
4.7. Constructors	22
4.8. Fields	23
4.9. Methods.....	23
4.10. Conclusion.....	23

5. Behavior Annotations.....	25
5.1. When To Inject?	25
5.2. @PostConstruct and @PreDestroy	25
5.3. @Focus.....	26
5.4. @Persist.....	26
5.5. @PersistState.....	27
5.6. @Execute and @CanExecute.....	27
5.7. Lifecycle Annotations	27
5.8. Conclusion.....	28
6. Services	30
6.1. Why Services?.....	30
6.2. Selection Service (ESelectionService)	31
6.3. Model Service (EModelService)	33
6.4. Part Service (EPartService)	35
6.5. Conclusion.....	36
7. Appendix A: Migrating from Eclipse 3.x to Eclipse 4 (e4)	37
7.1. Option 1: Use the Compatibility Layer	37
7.2. Option 2: A pure Eclipse 4 (e4) Application	37
7.3. Option 3: Compatibility Layer and Eclipse 4 (e4) Plugins	38
7.4. An Eclipse 4 (e4) Application including some 3.x components	39
8. Appendix B: Soft migration from 3.x to Eclipse 4 (e4)	40
8.1. POJOs in 3x.....	40
8.2. Dependency Injection in 3.x.....	43
8.3. Conclusion.....	45
For more information, contact us:	46

Eclipse 4 Tutorial

This tutorial introduces the core concepts of the Eclipse 4 Application Platform, aka RCP 2.0. One of the key innovations of e4 is the separation between the application model and the implementation of the application's parts, such as views. In the first chapter of this tutorial we provide an overview of the application model as well as the different ways to modify it using the editor or the API. In the second chapter of the tutorial, we explain how to work with the second part, the implementation of elements such as views. In chapter 3 we describe how to extend an existing application model with new elements, for example, adding a new entry to a menu. In chapter 4 we describe the basics about dependency injection in Eclipse 4. In chapter 5, we describe how Eclipse 4 uses annotations to specify when exactly parameters are injected, or more precisely, when certain methods of a class are called by the framework. In chapter 6, we describe the general idea of platform services as well as the most important ones in detail. Finally, we have included a brief discussion in the appendices on migrating from Eclipse 3.x to Eclipse 4. [1]

[1] Please note that the parts of this tutorial are numbered differently on eclipsesource.com because they were published in a different order.

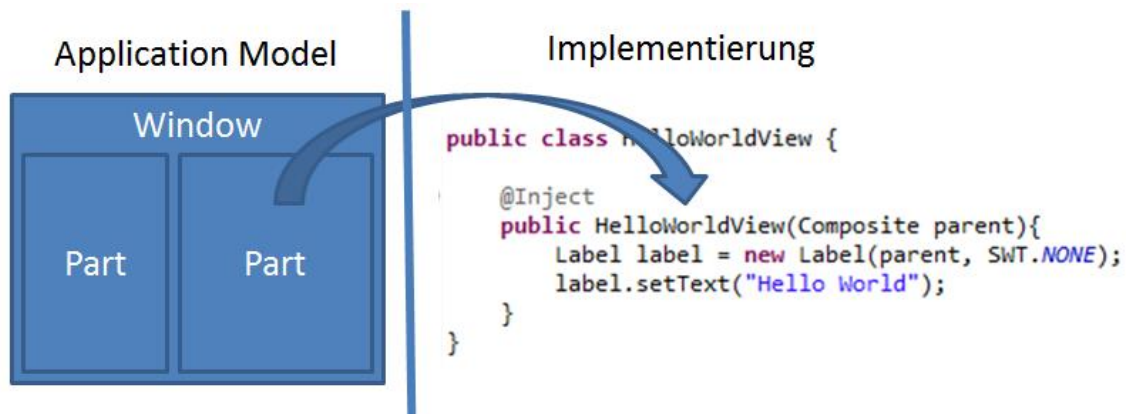
1. The e4 Application Model

In chapter 1 we start with the foundation of every Eclipse 4 application, the application model. We will show you how to install and get started with Eclipse 4 and then introduce the different options for modifying the model.

1.1. Application Model vs. Views

In Eclipse 4, the application model defines the workbench, including views, menu contributions and key bindings. The model doesn't require that you first implement the single components. For example, you can work with the model without implementing a view.

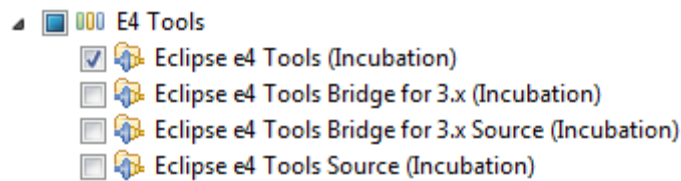
The cornerstones of the application model are windows and parts. Contrary to the eclipse 3.x platform, e4 has combined views and editors into the concept of Parts, which represent views inside a window. If you add a part in the model, you can later connect it to your implementation of the selected view. To show the resulting separation between the general workbench design and the implementation of single parts, we will not show any SWT code in this section. Instead we'll focus on the model and how to connect the model to code.



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

1.2. Installation

You can get the latest version of Eclipse 4 here: <http://www.eclipse.org/eclipse4/>. The IDE itself is based on Eclipse 4 and also contains several useful tools to create RCP and RCP 2 applications. Additionally, we recommend installing the e4 Tools, which, thanks to Tom Schindl, provide a very useful template for creating applications as well as an editor to modify the application model. At writing, these tools are still in incubator status and can be installed from this update site: <http://download.eclipse.org/e4/updates/0.11>



1.3. The first step

After installing Eclipse 4 the easiest way to get started is to use the e4 template to create a new e4 application. To create a project, choose the „e4 Application Project“ entry within the „new Project“ wizard. For this application you don't have to change anything except the name of the application. The template creates a product definition and you can start the application simply by starting this product. To start it, open the *.product file and click on run or debug in the upper right corner of the editor. As you can see below, the generated template application already contains a window, two menus, a toolbar and a perspective stack.



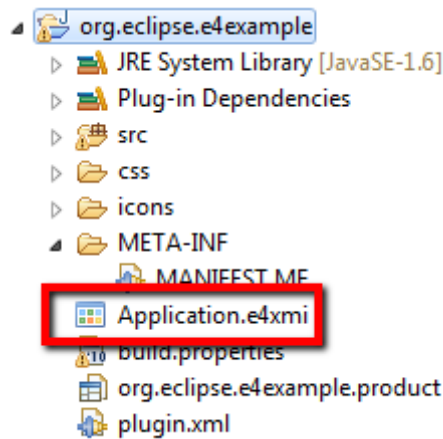
[Click here to start the product.](#)

1.4. The Editor

To modify the application model you'll need an editor which you can start by opening the Application.e4xmi located in the root level of the project. On the left side you see a tree showing the complete contents of the model. By double-clicking an element in the tree, a detailed view will be opened on the right side allowing you to modify the properties of that element.

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

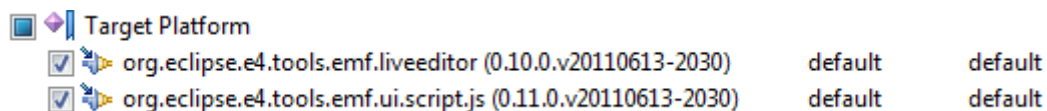
With a right click in the tree, new elements can be added and existing ones can be removed. As an example you can remove the existing PerspectiveStack and just add a single Part instead. After a restart of the application you will notice that the main area of the application does not have a border anymore. However, the new part isn't visible and it would be nice to have some control over the result. I'll describe how to do that in the next section.



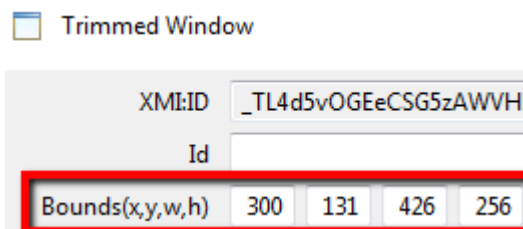
Open the application model to modify the workbench

1.5. Live Editing

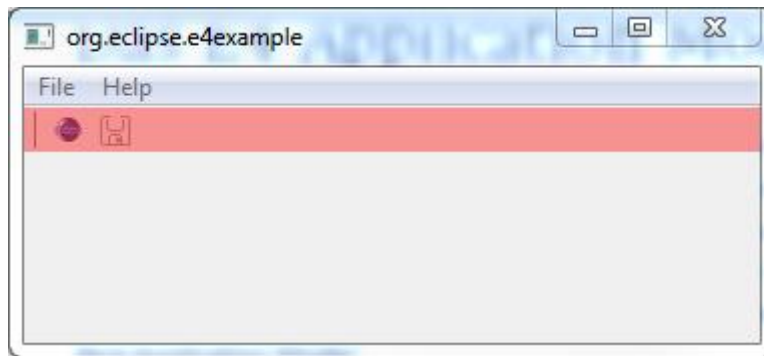
Eclipse allows you to define the workbench using the application model even without providing implementations. However, this is sometimes hard to work with because empty Parts are often hard to identify. To resolve this Tom Schindl introduced the idea of a live editor. It allows you to access the application model of a running application, modify it and highlight selected components. To enable the live editor you need to start two plug-ins along with your application. You can check them in the run configuration and 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 tree-node windows), 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.



Elements of the application models can be colored 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, 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.

1.6. Programmatical Access to the Model

One of the major advantages of the application model is the ability to modify it via API. As the application model is represented in EMF, the API is very familiar to anyone who has used EMF before. Using this API you can create or modify parts of the application programmatically, for example, reacting to a user action. To test this in the template application you can use 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 executed if the connected `ToolItem` is pressed by the user.

Dependency injection, which we'll go into more detail on later, allows the programmer to easily define which parameters are needed within this method. In the following code example the method requires the application window as parameter so it will be injected by the framework. In the first line a new part is created and in the second line this part is added to the window. You can check the result by using the live editor described above. First you'll need to start the application and the live editor. Then click the open button in the toolbar of the example application. In the live editor you can confirm that the new part has been added correctly and even color it in the application.

@Execute

```
public void execute(MWindow mWindow) {
    MPart newPart = MBasicFactory.INSTANCE.createPart();
    mWindow.getChildren().add(newPart);
}
```


In the second 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. Restart the application and press the button again 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);  
}
```

1.7. Scripting

Another nice feature of the live editor is the ability to apply scripting and access parts of the model API during runtime. As this code will be dynamically interpreted, JavaScript is used. Scripts can be executed on any part of the application model. To do so, start the application and the live editor (ALT+SHIFT+F9). Right click any element, e.g. a window, and select “Execute script”. In the open window, you can enter JavaScript, which will be wrapped to the Java API. The following code example will set the label of a window – during runtime.

```
mainObject.setLabel("Hello Eclipse")
```

This second example will make an element invisible. You can try executing this example on the ToolBar, which you can find in the model tree under *TrimmedWindow => TrimBar => WindowTrim => ToolBar*

```
mainObject.setVisible(false)
```

1.8. Conclusion

The e4 application model 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 chapter of this tutorial 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.

2. From the Application Model to the Implementation of Views

As we have seen, with the application model, it is possible to define and test the basic design of an application without implementing single views. In this second chapter 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.

2.1. 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. Another example would be to reuse a view in a modular dialog. Finally, when RCP applications are transferred to another context, e.g. on a mobile device (see RAP mobile [2]), 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.

2.2. A view without an application model?

To demonstrate the modularity of the application model and the implementation of views, we started in the first chapter 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 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 in a future chapter in this 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!");
    }
}

```

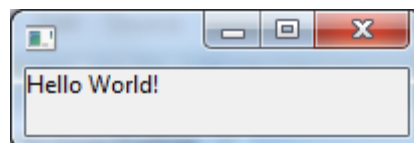
Using the application model, this view can be shown later on as part of the workbench in an application. To demonstrate the flexible reusability and testability of such a view, we will first use it without any workbench. The following code example shows how to open the “HelloWorld” view just using SWT.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

```

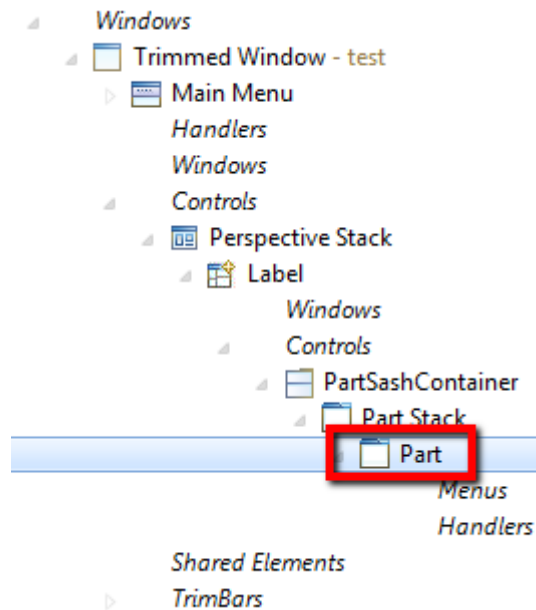
This screenshot shows the running Hello World application started from a plain Java program.



2.3. 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. The part will be linked to the implementation of the view. Using the e4 template application created with the e4 tools (see section 1.2 on Installation), 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 under “Class URI”. When you start the application, Eclipse 4 will create a part within the workbench and the linked view will be initialized. 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.

Tooltip	<input type="text"/>	
Icon URI	<input type="text"/>	<input type="button" value="Find ..."/>
Class URI	<input type="text" value="platform:/plugin/org.eclipse.example/org.eclipse."/>	<input type="button" value="Find ..."/>
ToolBar	<input type="checkbox"/>	
Container Data	<input type="text"/>	

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

2.4. Another Separation (Adding a Handler)

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. 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 show 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 {

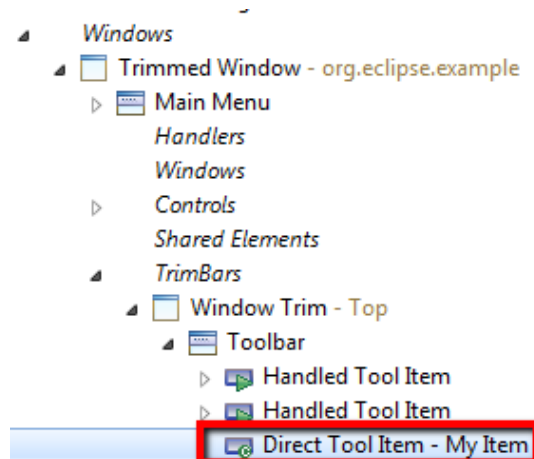
    @Execute
    public static void execute(Shell shell){
        MessageDialog.openInformation(shell, "", "Hello World!");
    }
}
```

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. These tests could also be written in JUnit.

```
public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);

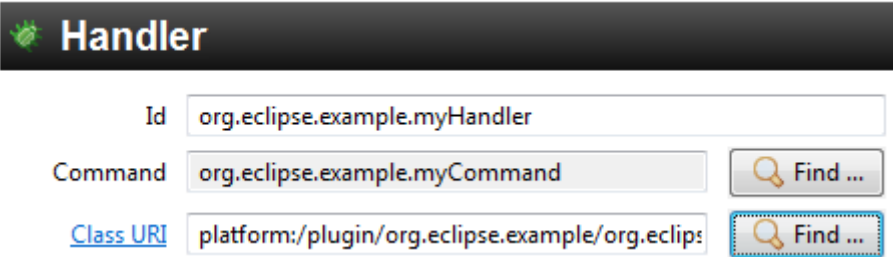
    shell.open();
    MyHandler.execute(shell);
    while( !shell.isDisposed() ) {
        if( ! display.readAndDispatch() ) {
            display.sleep();
        }
    }
}
```

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). Analogous to the part, the implementation of the handler can be bound to the element by setting the Class URI. We'll need to set a label or icon for the tool item to make it visible in the example application.



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 maximum reuse we recommend using commands. We do this by simply creating a handler and a command on the root level of the application model. The handler is bound to the implementation as before, but also to the command. Instead of using a “Direct ToolItem”, which is directly bound to the implementation, a “Handled ToolItem” will be bound to the command. This way, commands can be reused within the application, as an example, key bindings can be used to trigger the execution of a handler.



Handler

Id

Command

Class URI

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

2.5. Conclusion

In Eclipse 4, elements of the UI 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. 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.

[2] <http://developer.eclipsesource.com/tabriss/>

[3] <http://eclipsesource.com/en/services/training/training-details/introduction-to-eclipse-4-e4/>

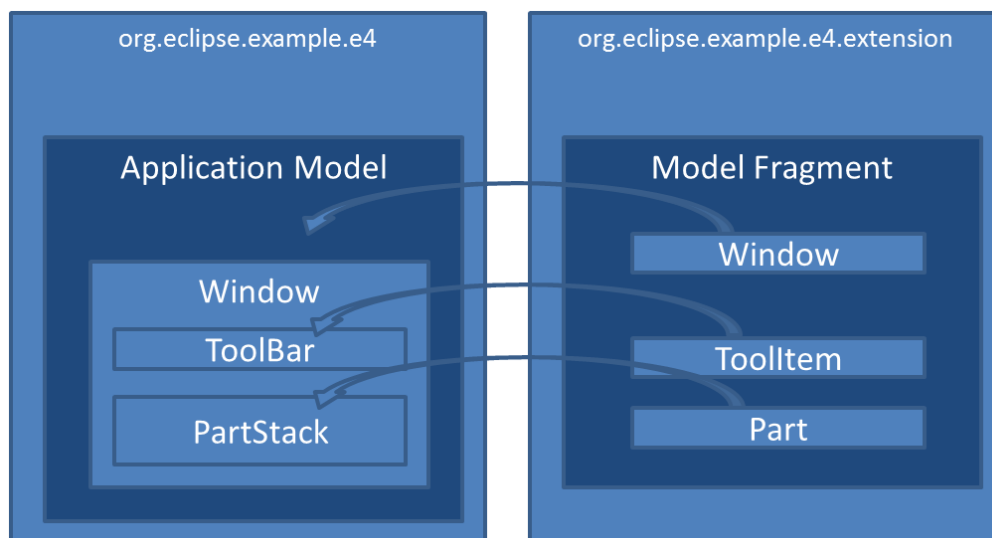
3. Extending the Application Model

In the previous chapters of this tutorial 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. In this chapter, we describe how to extend an existing application model with new elements, for example, adding a new entry to a menu.

3.1. Only one model?

One of the major advantages of Eclipse RCP development is the modular design of applications. The plugin concept, based on OSGi, enables the development of features independently and deployment of same, only if required. A very good example of such a modular application is the Eclipse IDE, where many additional plugin 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, while Eclipse 4 offers fragments and processors to extend the application model. A fragment is a small application model 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“.



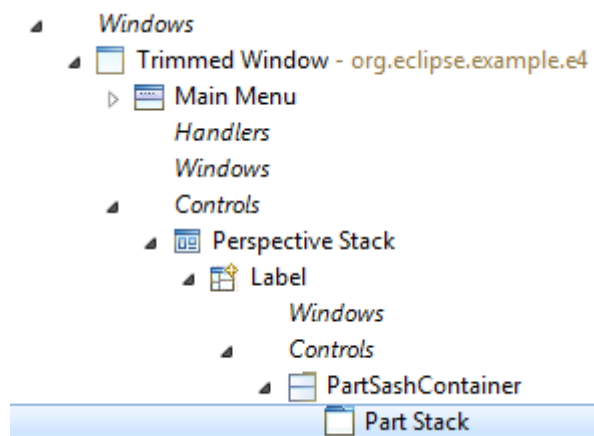
The application model can be extended using fragments.

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

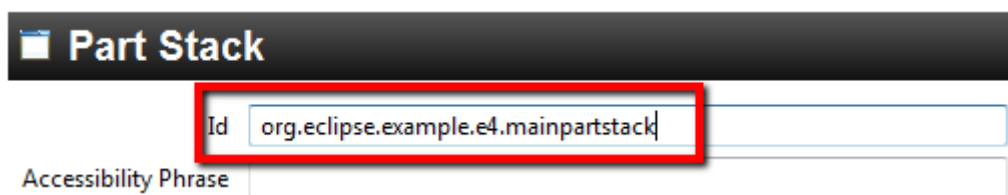
new button if another contribution is installed or if you can remove existing elements. In this tutorial, we describe both ways of extending an application model, fragments and processors. In both cases elements of the application model are linked to their implementations as described in the previous 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“.

3.2. Warm-Up

The first step is to create a main plugin and an application model that can be extended. As in the previous chapters 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 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



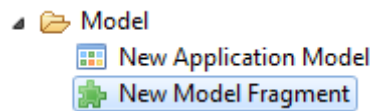
The ID allows referencing of an existing element from an extending fragment

Additionally, we need a second plugin which extends the first one. This second plugin needs the following dependencies:

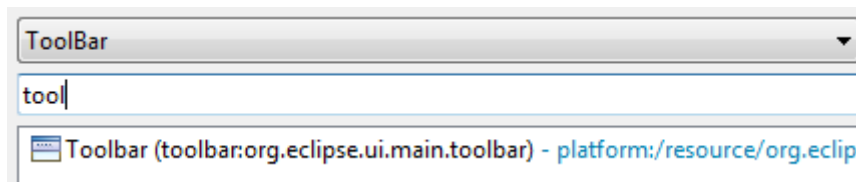
- org.eclipse.e4.ui.model.workbench
- org.eclipse.e4.core.di
- javax.inject
- the plugin to be extended

3.3. Model Fragment

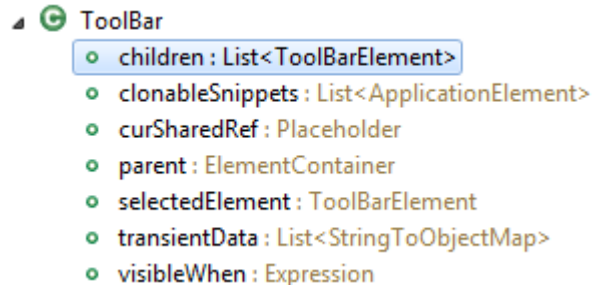
A fragment is nothing more than a small application model. A file containing fragments can be created using the wizard provided by the e4 tools.



The extending plugin is set as a container. The model fragment file is opened in an editor which works similarly to the editor used to modify an application model. The first step is to add a fragment. A 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 EMF 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 a “child”. In the following example, a new element is added as a child of the existing toolbar.



The element ID defines which element is extended



The feature name defines where the new element is added

Within the new fragment, another new element can now be created. As a simple example, we will add a `DirectToolItem` as a child of the fragment. To make it visible, a label or icon should be set. To trigger some action on a click on the new tool item, 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!");  
    }  
}
```

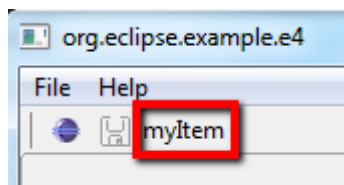
To connect the `DirectToolItem` to the handler, the “Class URI” of the `DirectToolItem` is set to the implementing class. This class is located in the extending plugin.

Of course it is also possible to add more than one element to a fragment. For example, a tool item, a handler and a command can be added and linked to each other.

In any case the new model fragment has to be registered via an extension point.

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

Finally, the new plugin adding the model fragment has to be added to the existing product configuration. After restarting the application the tool item should be visible in the same way as a new part would be added to the existing part stack.



3.4. 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. 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 [1] The following code shows the implementation of the described processor where the modified application model (`MApplication`) can be injected.

```
public class Processor {

    @Inject
    MApplication application;

    @Execute
    public void execute(){
        MWindow existingWindow = application.getChildren().get(0);
        existingWindow.setX(200);
        MTrimmedWindow newWindow =
        MBasicFactory.INSTANCE.createTrimmedWindow();
        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. 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>

```

3.5. 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 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.

4. Dependency Injection Basics

In most of the programming examples provided so far, we implicitly used a further innovation of Eclipse 4: dependency injection (DI). DI plays a central role in Eclipse 4, reason enough to devote a whole tutorial to this. This tutorial describes:

- Which objects can be injected.
- How the Eclipse Context works.
- Which annotations can be used to influence the injection.
- How the Eclipse Context can be extended.
- How the injection can be triggered manually.

4.1. 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 a view that needs a parent composite, an input object or a service, such as a logger. 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. 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 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. 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. 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.

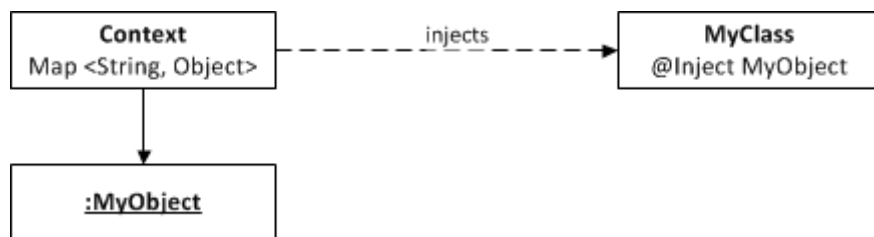
Injected objects can either be fields of a class, parameters of a constructor or parameters of a method that is called by the framework. In the simplest case, required objects are marked by the annotation `@Inject`. There are a number of additional annotations that control the behavior and the timing 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  
}
```

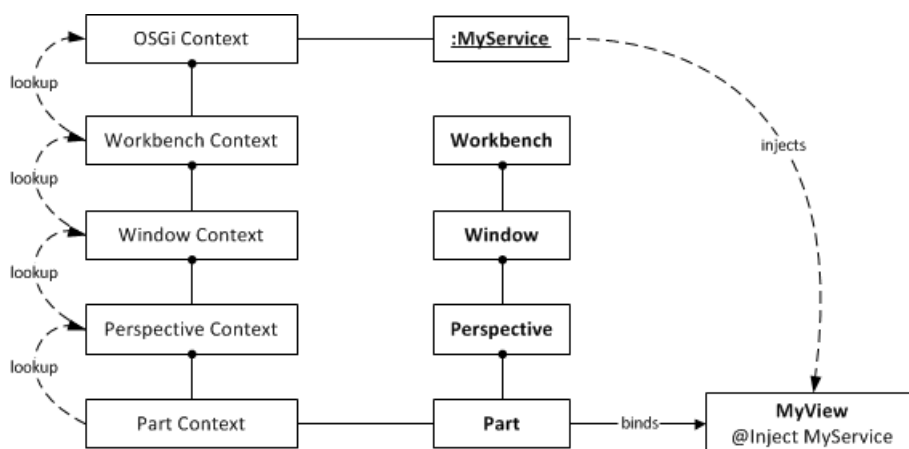
4.2. The Eclipse Context

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, 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 included 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 as well as 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.

4.3. @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 includes 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 the next installment of this tutorial.

4.4. @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, it has to be checked, if it is null.

4.5. `@Active`

In certain use cases, it is necessary to access not only a specific type of an element from the application model, but the currently active item. With the annotation `@Active`, the currently active element is used for the injection, for example, to access the active part.

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

4.6. Injecting Objects

In the simplest case the injection is triggered via the `@Inject` annotation. It can be placed in front of each parameter, prior 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 a future installment 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. 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.

4.7. Constructors

Constructors should include parameters that are essential for the existence of an object. Any unnecessary parameter limits the testability and as well, the reuse 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
}
```

```
}
```

4.8. 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.

4.9. Methods

After the constructor and the fields, while initializing a class, all annotated methods are sequentially called. This also applies to methods that have no parameters. If one of the injected parameters of a method changes in 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
}
```

4.10. Conclusion

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

The next chapter of the tutorial is dedicated to a more in-depth description of dependency injection, for example, how the injection can be triggered manually or how custom elements can be added into the context. In addition to **@Inject**, 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.

5. Behavior Annotations

In the previous chapters of this tutorial, we described how to create an application model, link those elements to implementations and how to extend the application model. In the last chapter of this tutorial, we provide 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 chapter describes the most important annotations used in Eclipse 4.

5.1. When To Inject?

The annotation `@Inject` in combination with `@Named` and `@Optional`, described in a previous chapter 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 with only `@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 described annotations include the dependency injection as `@Inject` does. That means that if a method is marked with any of the below annotations, all parameters of the method will be injected without an explicit addition of `@Inject`.

5.2. @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 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.

5.3. @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();
}
```

5.4. @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);
}

```

5.5. @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.

5.6. @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 4.2, 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;
}

```

5.7. 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.

5.8. 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

6. Services

In the previous chapters of this tutorial series, we described how to create an application model and link those elements to implementations, how to extend the application model, details about dependency injection and how to use behavior annotations.

In the last two chapters, 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. In Appendix B of the tutorial, we explain that one type of object you can retrieve using dependency injection is 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 tutorial, we described the three most important Eclipse 4 services: the selection service (ESelectionService), the model service (EModelService) and the part service (EPartService).

6.1. 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 Appendix B 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 Appendix B and chapter 5 for more details about dependency injection.

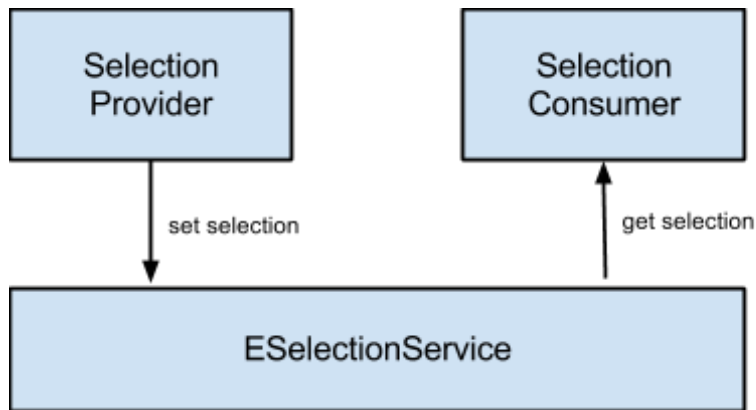
To be fair, the idea of using services is nothing new and nothing specific to Eclipse 4. However, Eclipse 4 has really adapted the concept of a service-oriented architecture very. This provides mitigations to the above mentioned problems:

- As services can be injected, you only need to know which service you want to use, 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, this experience was used to design the API for Eclipse 4
- Services can be replace, 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 already 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 try to describe 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.

6.2. 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 enablement of a handler can be dependent on the current selection. Thus, the enablement of the handler “Export File” could only be enabled if the current selection is a file.



The API to set a selection is very simple. The selection service is typically injected as a field. After that, it provides a method to set the current selection.

```

@Inject
ESelectionService selectionService;
//Within a method:
selectionService.setSelection(someObject);
  
```

In real applications, a change of the current selection is typically triggered by a click of the user, e.g., in a tree viewer. Therefore, the corresponding UI element has to be observed. This part is not specific to Eclipse 4 and depends on the used UI Toolkit. In the following example, we assume we want to set the selection of an application when an element is selected in a TreeViewer. The registration and implementation of this “TreeViewerSelectionListener” is therefore specific for a JFace TreeViewer and independent of Eclipse 4 (except the call to the selection service). The TreeViewerSelectionListener is registered in an init method marked with @PostConstruct, and we assume the TreeViewer has been created in the constructor.

```

@PostConstruct
public void init(){
    selectionChangeListener = new TreeViewerSelectionListener();
    viewer.addSelectionChangedListener(selectionChangeListener);
}
@Inject
ESelectionService selectionService;
private class SelectionListener implements ISelectionChangedListener {
    @Override
    public void selectionChanged(SelectionChangedEvent event) {
        Object first =
            ((IStructuredSelection)event.getSelection()).getFirstElement();
        selectionService.setSelection(firstElement);
    }
}
  
```

On the other side, there are two possibilities for implementing an selection consumer, i.e., an element that gets notified when the selection changes. The first, conservative possibility is to inject the selection listener again and register a listener on it; the selection service provides API for this. However, in Eclipse 4, there is an even better possibility, which is to inject the selection directly as a parameter of a method. Using dependency injection, the framework will check whether the selection is of the right type, so no checks and casts are necessary. For injecting the selection, two additional annotations are used. @Named in combination with the

corresponding constant (see below) specifies that the injected object should be the selection. `@Optional` specifies that if there is no current selection, null should be injected. This is necessary because there does not always need to be a current selection and the dependency injection will fail if `@Optional` is not used. The following example shows a method within a selection consumer. It will be called whenever the selection changes. If the selection is of type “MyObject”, it will be passed as a parameter, and if not, null will be passed.

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

A special and often used case of injecting the selection is the implementation of the `canExecute` method of a handler. This method is responsible for determining whether a certain handler and the corresponding menu or toolbar item is enabled depending on the current state of the application. Such a check often depends on the current selection, e.g., a delete button in the toolbar is only enabled when a file is selected. The following example shows the typical implementation of such a method; see [part 6](#) of this tutorial for more details about `@CanExecute`

```
@CanExecute
public boolean void canExecute(
    @Named(IServiceConstants.ACTIVE_SELECTION)@Optional MyObject myObject){
    return myObject!=null;
}
```

6.3. Model Service (EModelService)

The model service provides general support for working programmatically with the application model. As described in chapter 1 of this tutorial, the application model represents the current state of the application including windows, perspective, parts, etc. Changes on this state done by the user of an application, e.g., by moving a window, are updated in the model. Developers can also access the model programmatically, e.g., call `window.setX()`. This in turn will be reflected in the UI of the running application. The API to modify the application model consists of two parts. First, every model element provides methods to be modified, e.g., getters and setters. Second, services such as the model service provide generic methods to operate on the model. The model service provides methods for all types of application model elements. Other services, such as the `EPartService` provide methods for a specific subset of model elements. In the following, we describe the most important methods of the model service. This list is not complete; please look at the API of `EModelService` to find more useful methods. The model service can be retrieved using dependency injection:

```
@Inject
EModelService modelService;
```

Creating Application Model Elements

The model service provides a generic method to create any kind of application model element. As a parameter, you need to pass the class you want to create, e.g., create a new window. Note that every created model element has to become part of the existing model. The following example creates a new window and adds it to the application:

```
@Execute
void addNewWindow(MApplication application){
    MWindow window = modelService.createModelElement(MWindow.class);
    application.getChildren().add(window);
}
```

Navigating the Application Model

In many cases, you can get the corresponding model element injected when you want to access the application model. For example, if you want to access a window from a part, you can use:

```
@Inject
MWindow window;
```

This works because Eclipse uses hierarchical contexts for resolving objects; see [part 6](#) of this tutorial. However, there are use cases where the model elements to be accessed come from another context or where the type of model element does not uniquely identify the element. For these cases, the model service offers methods to navigate within an instance of the application model. A group of methods allows developers to find certain elements within the application model; all these methods start with “find”. You can search for the ID of a model element, for its type, for certain tags and a combination of all these parameters. If a parameter should be ignored, null is used instead. As an additional parameter, find methods require the root object to search in. This allows you to search along the hierarchy of the application model. This can be the complete application model, only in a certain window, or within a perspective. If you want to search within the complete application model, you pass an element of the type MApplication. The following method searches for all perspectives of an application and ignores IDs and tags. Please note that the service can also be injected as a parameter, so no field is needed in this case.

```
@Execute
void search(MApplication application, EModelService modelService){
    modelService.findElements(application, null, MPerspective.class, null);
}
```

By using additional search flags, the lookup can be restricted, e.g., to the active perspective. The following call searches for a part with the id “myID” in the active perspective.

```
modelService.findElements(application, "myID", MPart.class,
    EModelService.IN_ACTIVE_PERSPECTIVE);
Part Service (EPartService)
```

Additionally, the model service provides a few specific methods to navigate within the model, e.g., retrieving the window containing an application model element (e.g., a part).

```
modelService.getTopLevelWindowFor(part);
```

Please refer to the API documentation of the find methods in EModelService to learn about all available alternatives to search for model elements.

Detach Windows

This method allows you to detach a part, partstack or partsash to a new window. The following example detaches a part into a window, position 0,0 and size 800 by 600:

```
modelService.detach(part, 0, 0, 800, 600);
```

6.4. Part Service (EPartService)

The part service provides operations specific for parts and perspectives. For parts it allows you to:

- show parts: showPart()
- hide parts: hideParts()
- bring parts to top, e.g., on a part stack: bringToTop()
- retrieve all dirty parts: getDirtyParts()
- etc.

Please refer to the API documentation of the find methods in EPartService to learn about all available features for parts.

One central method of the part service allows you to trigger a save on one or all parts. This is used to save the content of views, which can become dirty (i.e., editors). The implementation of a part can influence its dirty state by calling the interface MDirtyable, e.g.,:

```
@Inject MDirtyable dirtyable;  
dirtyable.setDirty(true);
```

Additionally, parts with a dirty state should implement a method annotated with @Persist, e.g.,:

```
@Persist  
void save(){  
    //save the content of the view  
    dirtyable.setDirty(false);  
}
```

The EPartService allows you to trigger this method and therefore save the content of dirty editors of the application. This can either be done for a single part or for all dirty parts:

```
partService.save(part);
```

This will call the method in the part implementation, which has to be annotated with `@Persist`.

Finally, the `EPartService` allows you to switch the active perspective. The following example saves all dirty parts and switches the perspective afterwards.

```
@Execute
public void execute(EPartService partService) {
    partService.saveAll(true);
    partService.switchPerspective(perspective);
}
```

6.5. Conclusion

In Eclipse 4 common framework, features are provided as services. This provides a good separation of concerns. Additionally, services can easily be replaced by custom implementations. As they are part of the Eclipse Context, you can simply do that by calling `context.set(ServiceInterface.class, customImplementationOfService)`. Of course you need to select the right context, depending on whether you want to replace a service for the whole application or only for parts of it. The same scheme applies when it comes to testing. If a class to be tested requires a certain service, a mock can easily be provided by implementing the same interface but only simulating its behavior.

7. Appendix A: Migrating from Eclipse 3.x to Eclipse 4 (e4)

Eclipse Juno 4.2 is about to be released. It will be the first Release Train building on the new Eclipse 4 (e4) Application Platform. This raises the question of how to migrate existing Eclipse 3.x applications to Eclipse 4 (e4). In this appendix, we will review the options for developing Eclipse plugins and applications with the new platform. Looking forward to your comments and additions.

7.1. Option 1: Use the Compatibility Layer

The compatibility layer enables 3.x applications to run on the new Eclipse 4 platform without any code adaptation. In the beginning, most existing projects will probably use this option. 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 3.7.

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 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

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. Some other improvements will work though, such as CSS styling.

7.2. Option 2: A pure Eclipse 4 (e4) Application

The second 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.

7.3. Option 3: Compatibility Layer and Eclipse 4 (e4) Plugins

For this option, you would rely on the compatibility layer to reuse all existing components without any adaptations. New components would be developed following the e4 programming model, including dependency injection and annotations. There are three ways to integrate e4 elements into a compatibility layer application.

The first option is to use processors and fragments to add elements to the application model created by the compatibility layer. However, there are currently 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 handles and views, but currently it doesn't work for editors.

The second option for integrating Eclipse 4 components is to create a copy of the application model used by the compatibility layer, register it as the application model of your application and add new e4 components to it. The relevant model LegacyIDE.xmi can be found in the plugin org.eclipse.ui.workbench.

The third option is to use the 3.x e4 bridge from the e4 tools project, developed by Tom Schindl. The goal of the bridge is to ease single sourcing applications on 3.x and e4, which means that views and editors can be used in 3.x and e4 in parallel. To enable this, the plugin org.eclipse.e4.tools.compat provides wrapper classes that implement the interfaces of 3.x. For example, the wrapper DUIViewPart implements ViewPart. In the wrapper, you specify a class (POJO), which implements a view following the e4 programming model, including dependency injection. Essentially the wrapper is just a pointer to an e4 object. It will initialize the POJO using dependency injection.

A 3.x wrapper

```
public class ExampleViewWrapper extends DUIViewPart<ExampleView>{
    public Example3xViewWrapper() {
        super(ExampleE4View.class);
    }
}
```

A e4 view:

```
public class ExampleView {
    private Label label;
    @Inject
```

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

This approach allows you to develop new parts of the application using all the benefits of e4 and as well, reuse all existing components. Further, the views developed in this way can be integrated into any pure e4 application without any adaptations.

7.4. An Eclipse 4 (e4) Application including some 3.x components

In this option you would develop an e4 application and reuse some 3.x components. If they don't access the workbench API, there shouldn't be any problems. In some cases, even UI components can be easily reused or adapted to work with e4. However, this needs to be evaluated individually for each component.

In the end, when and how to migrate to e4 is still one of those "it depends..." decisions. Probably the most important criteria are the number of existing components and the number of reused third-party components. If you have additional options for migrating or mixing the two technologies, let me know and we will gladly add it to this tutorial.

8. Appendix B: Soft migration from 3.x to Eclipse 4 (e4)

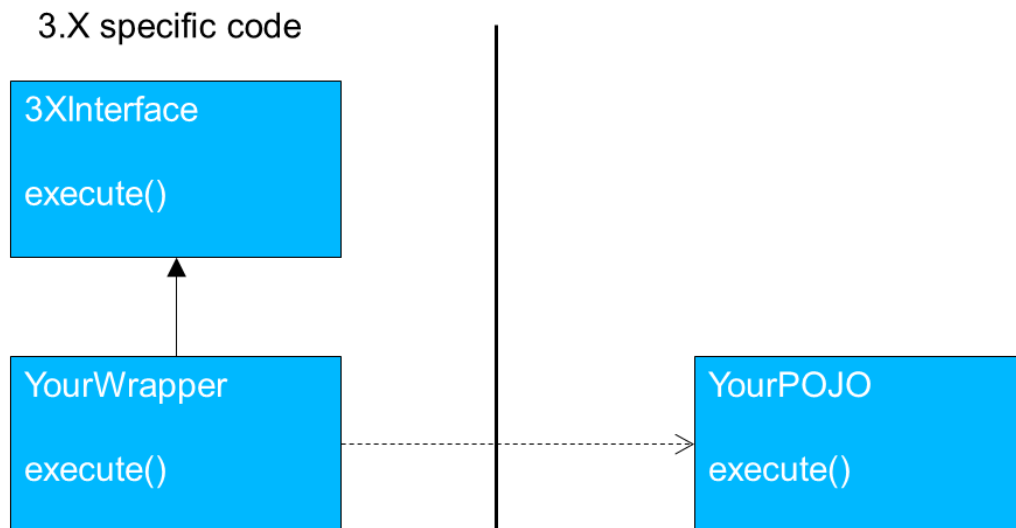
This tutorial describes how to do a soft migration to the Eclipse 4 (e4) programming model. The basic goal of the tutorial is to enable development using the new concepts such as Dependency Injection and Annotations, but without first requiring a complete application migration. Therefore, all existing plugins and frameworks which require 3.x, 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 re-usable.
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.

8.1. POJOs in 3x

The basic concept is to make a clear separation between the code that you develop for a custom application and the code that binds your component into the Eclipse workbench. The second group of code 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.



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 then 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.receiveMails();
        }
    }

    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. As the selection in this case is not a plain field, but a property, you would need to look at the implementation of HandlerUtil.getCurrentSelection() to find out how to properly prepare your Mock ExecutionEvent. In the last section, you have more lines of boilerplate code in the test case. 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 need to create an ExecutionEvent. If the handler is within your control, you will probably refactor at this time. But 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 deals with all workbench specific parts, i.e. unpacking the selection. The second method executes the business logic itself and can, as 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.recieveMails();
    }
}

```

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. Finally, 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 results 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; the pattern can be used in plain 3.x. 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. In the next section we introduce such a generic implementation that uses dependency injection.

8.2. Dependency Injection in 3.x

If you separate workbench specific code and custom components as POJOs, components are easier to re-use 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 and registered via an extension point.
2. The implementation of the component cannot use dependency injection and is therefore not ready to be used in Eclipse 4.

Once your application is running on an Eclipse 4.x version there is a solution for this, even if you still use the 3.x API. The solution is provided by the 3.x e4 bridge from the e4 tools project, developed by Tom Schindl. 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 writing, implementations for Views and Editors are available. To create the 3.x workbench wrapper, one simply inherits from the respective type, e.g. from DUIViewPart 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 DUIViewPart<ExampleView>{
    public ExampleViewWrapper() {
        super(ExampleView.class);
    }
}

```

This class is now registered using the view's extension point as is usual in 3.x.

```
<extension
  point="org.eclipse.ui.views">
  <view
    class="org.example.e3.ExampleViewWrapper"
    id="org.example.e3.view1"
    name="e4 View"
    restorable="true">
  </view>
</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 program today, 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 and 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, which we recently contributed. 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 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<C> extends AbstractHandler {

    private Class<C> clazz;
    private C component;

    public DIHandler(Class<C> 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.

8.3. Conclusion

This tutorial described an approach 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. Using the bridge plugin, the e4 tools wrapper classes didn't have to be implemented manually. Additionally the DIWrapper enabled the use of dependency injection even when programming in 3.x. Components developed in this way can be integrated without any adaptations, into any pure e4 application.

For more information, contact us:

Maximilian Koegel and Jonas Helming

EclipseSource Munich leads

Email: e4@eclipsesource.com

<http://eclipsesource.com/munich>

