Eclipse 4 Tutorial – Part 1-3

Part 1: The Eclipse 4 Application Model

Part 2: From the Application Model to the Implementation of Views

Part 3: Extending the Application Model

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

Jonas Helming

Maximilian Koegel

http://eclipsesource.com

Eclipse 4 Tutorial

This tutorial series 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 part 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 part of the tutorial, we explain how to work with the second part, the implementation of elements such as views. In Part 3 we describe how to extend an existing application model with new elements, for example, adding a new entry to a menu. Finally, we have included a brief discussion in an Appendix on migrating from Eclipse 3.x to Eclipse 4.

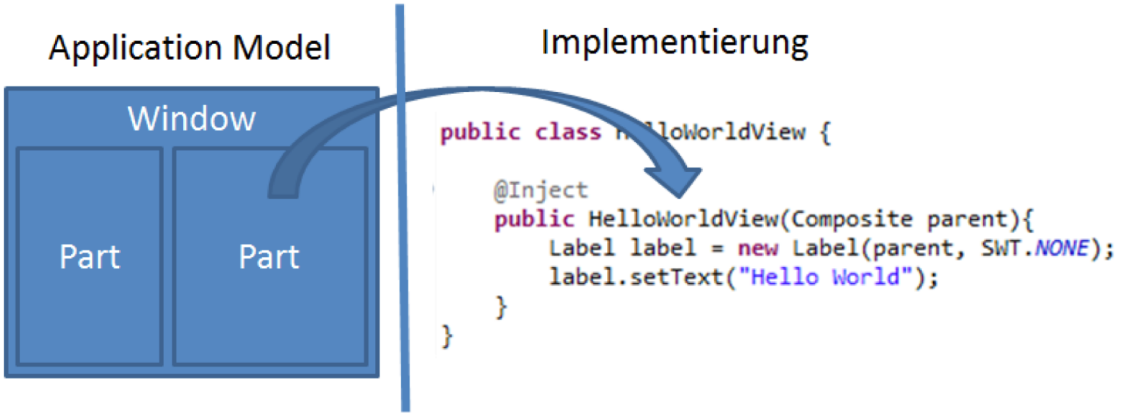
Part 1: The e4 Application Model

In Part 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.

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.

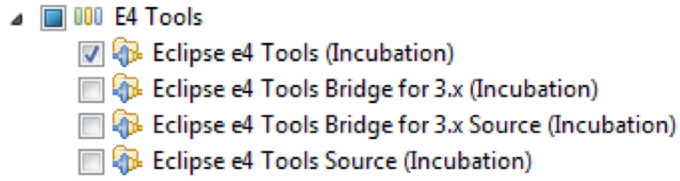
The cornerstones of the application model are windows and parts. Contrary to the eclipse 3.x platform, e4 has combined views and editors into the concept of Parts, which represent views inside a window. If you add a part in the model, you can later connect it to your implementation of the selected view.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

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



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.

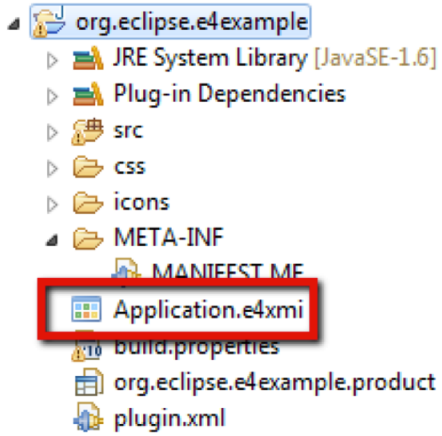
Click here to start the product.

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.

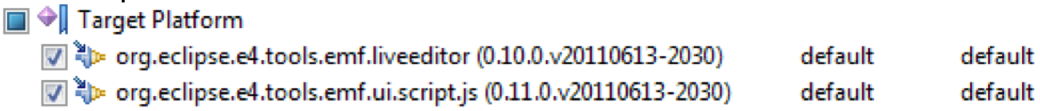
With a right click in the tree, new elements can be added and existing ones can be removed. As an example you can remove the existing PerspectiveStack and just add a single Part instead. After a restart of the application you will notice that the main area of the application does not have a border anymore. 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.



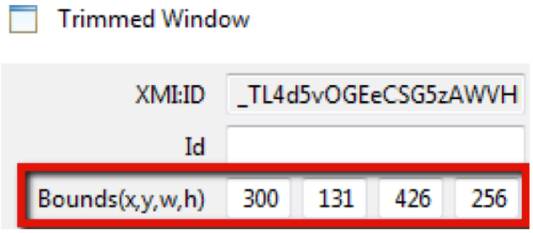
Open the application model to modify the workbench

Live Editing

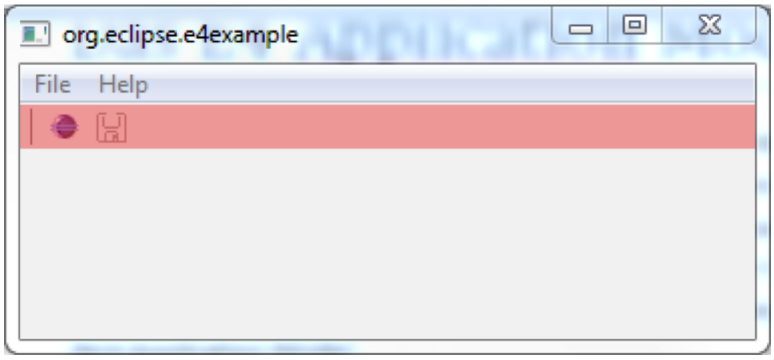
Eclipse allows you to define the workbench using the application model even without providing implementations. However, this is sometimes hard to work with because empty Parts are often hard to identify. To resolve this Tom Schindl introduced the idea of a live editor. It allows you to access the application 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.

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);
	}
```

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)
```

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.

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 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.

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.

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 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 tutorial in this series.

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

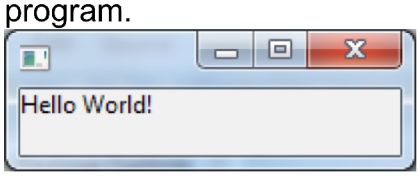
```
public class ExampleView {
```

```
	@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();
	    }
	}

}
```

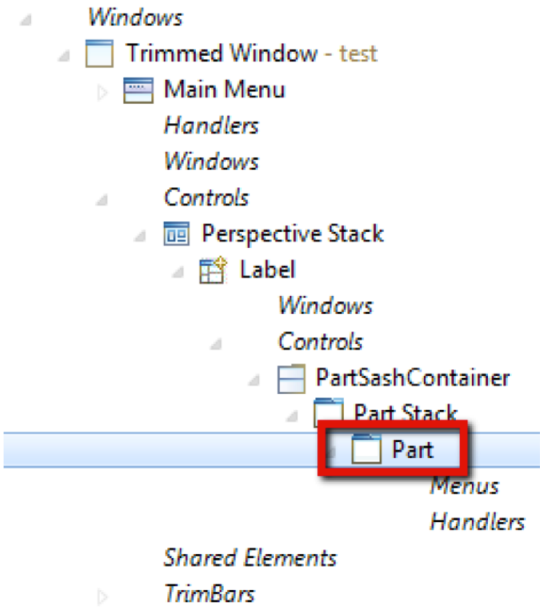
This screenshot shows the running Hello World application started from a plain Java program.



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 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 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.

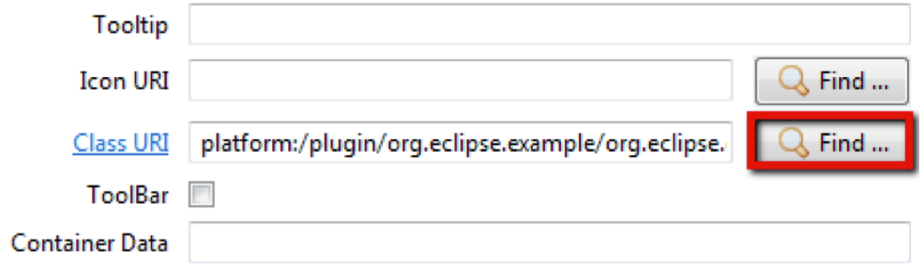


Figure : Parts are linked to the implementation of views using the Class URI

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” (see Figure). 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.

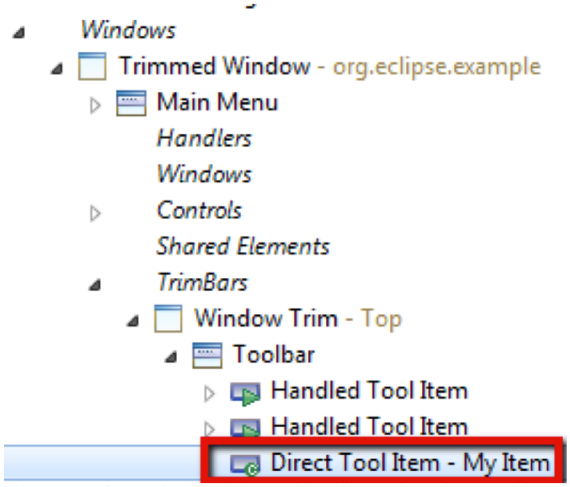


Figure : Creating a Direct ToolItem in the toolbar of the application

Using a „Direct ToolItem“ is the fastest way to integrate a handler with an item in the toolbar. However, for 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 (see Figure). 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.

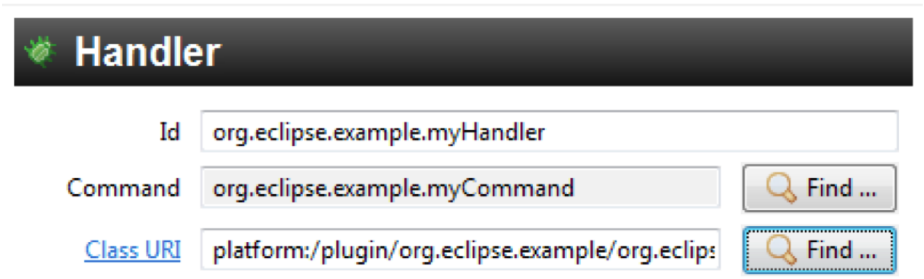


Figure : A handler is bound to its implementation as well as to a command.

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] eclipsesource.com/mobilerap
[3] http://eclipsesource.com/en/services/eclipse-training/introduction-to-eclipse-e4/

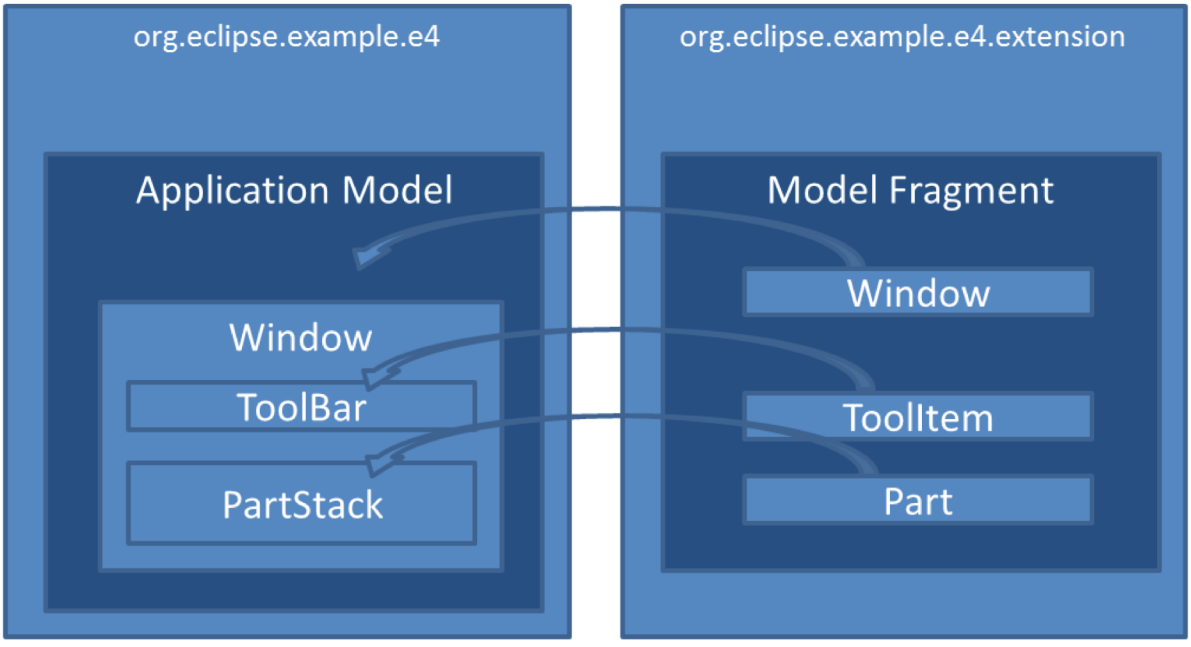
Part 3: Extending the Application Model

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.

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“.

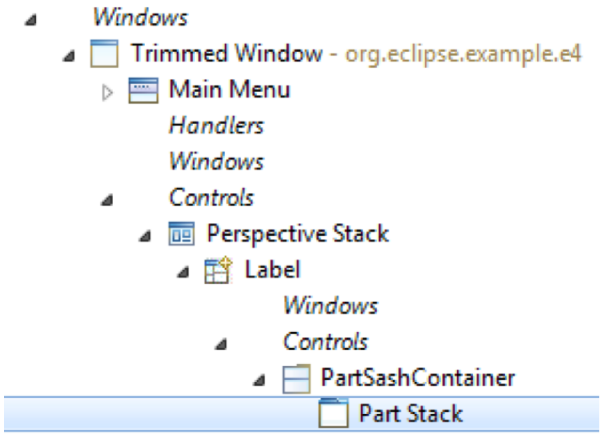


The application model can be extended using fragments.

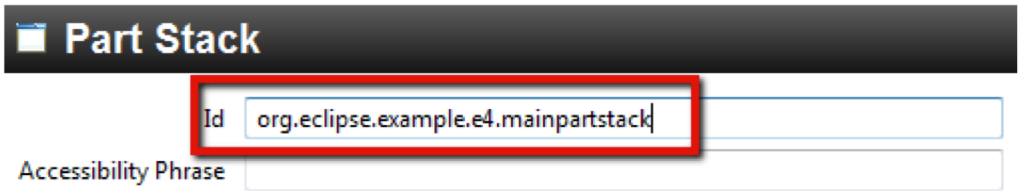
Processors offer a mechanism to programmatically extend an application model. This allows the application to react to the current state of the model. As an example, you can only add a new button if another contribution is installed or if you can remove existing elements. In this tutorial, we describe 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“.

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 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

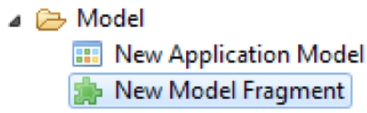
Additionally, we need a second plugin which extends the first one. This second plugin needs the following dependencies:

- org.eclipse.e4.ui.model.workbench
- org.eclipse.e4.core.di
- javax.inject

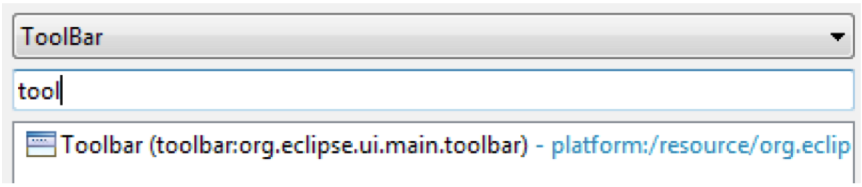
the plugin to be extended

Model Fragment

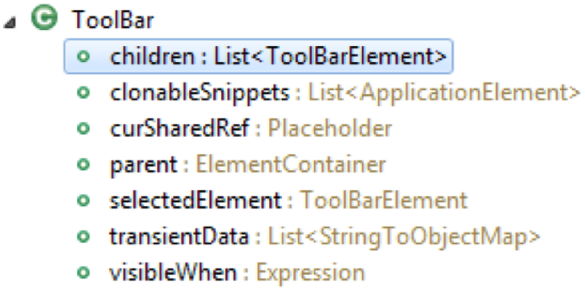
A fragment is nothing more than a small application model. A file containing fragments can be created using the wizard provided by the e4 tools.



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!”.

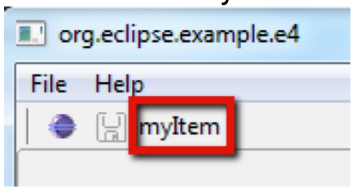
```
public class MyHandler {
	@Execute
	public void execute(Shell parent) {
```

```
            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.



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>
```

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.

Appendix: 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 post I will review the options for developing Eclipse plugins and applications with the new platform. Looking forward to your comments and additions.

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.

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.

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 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 DIViewPart 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 DIViewPart<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.

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 I will gladly add it to this post.

For more information, contact us:

Maximilian Koegel and Jonas Helming
EclipseSource Munich leads

Email: e4@eclipsesource.com

http://eclipsesource.com/munich