What about your Eclipse 4 migration strategy ?

Eclipse Day Milan

22 September 2017

Table of contents

What about your Eclipse 4 migration strategy ?

This talk will explain :

- some general issues about migration
- the key points of your migration strategy
- tooling that could be used
- the model fragments and processors
- some injection information
- how to migrate some standard extensions (extra)

A. Presentation

OPCoach

Image 1

- **Eclipse Training** : RCP, E4, Modeling, Build, given in French, English and ... Spanish
- **Eclipse Consulting**
- **Recruitment service on Eclipse technologies**
- **ICPF & PSI quality certification**
- Web site : *http://www.opcoach.com/en*[1]
- Twitter : **@OPCoach_Eclipse**

B. Migration concerns

It is time to migrate / software reasons

- The official **Eclipse RCP Runtime** is 4.X since June 2012
- Injection is pretty cool, reduces the amount of code, simplifies testing and reduces coupling
- Application model is dynamic and platform agnostic (SWT, Java FX...) thanks to POJOs`
- Eclipse 4 event notification system (**IEventBroker**) is very concise and easy to use with injection

It is time to migrate / other reasons

- UI customization:
  - CSS
  - renderers that can be overriden
- Tooling:
  - E4 spies will help to develop your application
  - Plug-in templates have been defined to help you

1 - http://www.opcoach.com/en

- Long term and strategical:
  - Your application will still live several years :
  - It will provide an opportunity to refactor and decouple your components
  - The application will be better every year with the new runtime version

The global prerequisites / team

Be sure of your team's knowledge:

- do they know Eclipse 3 and Eclipse 4 ?
- do they know the application ! ! ?
- do they know how to migrate ?
- are they really involved in the migration process ?

The global prerequisites / application

Be sure of your software's quality :

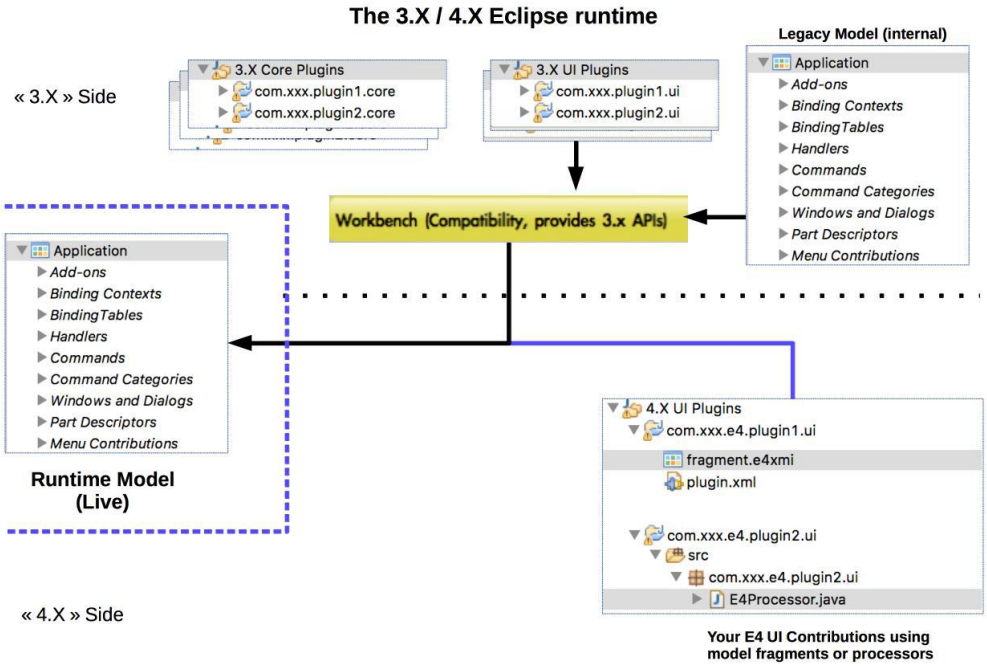
- do you manage properly your target platform ?
- is your architecture clean regarding ui and core separation ?
- do you have a continuous integration build (really better) ?

The technical prerequisites

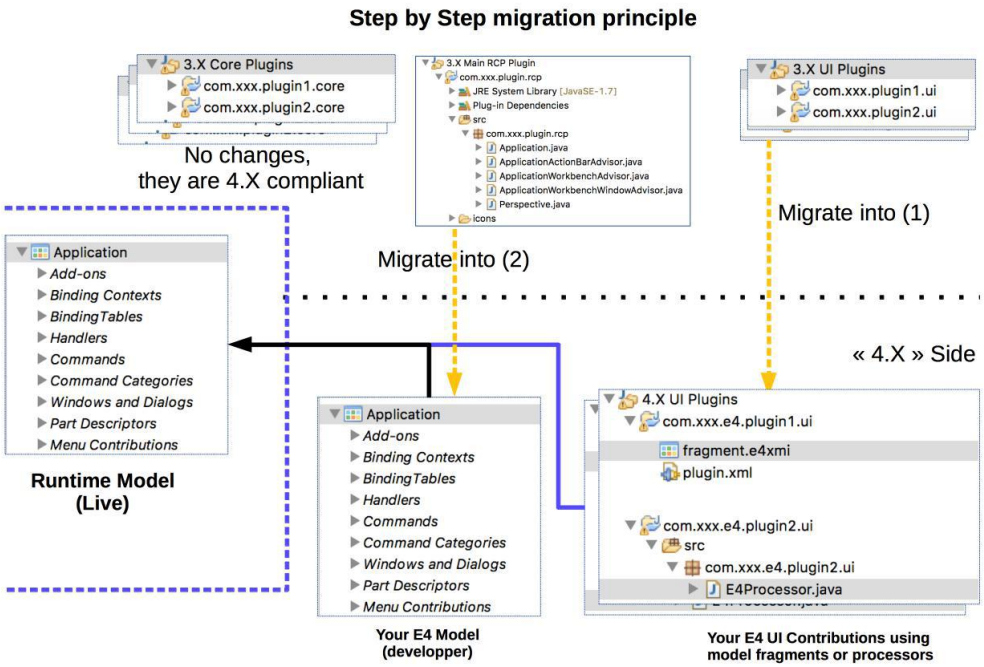
To prepare your E3 plugin/application migration you have to :

- clearly separate core and ui plug-ins
- have packages for each entities to migrate : views, handlers, etc...
- remove the **org.eclipse.ui** internal package uses and imports

Big picture of 4.X runtime application with 3.X components



Big picture of a full migration



Define a migration strategy

Basically you will have to follow these steps:

1. adapting and rebuilding against the newest target platform (use CL)
2. migrating some or all features of your application (use CL)
3. migrating fully to E4 (no CL)

1. Adapting and rebuilding against the newest target platform

Change the Target Platform (use the latest one)

Use the target platform editor

Add these dependencies (in top level plugin or in launch configuration)

- **org.eclipse.equinox.ds**
- **org.eclipse.equinox.event**
- **org.eclipse.equinox.util**
- **org.eclipse.e4.ui.workbench.addons.swt**

2. Migrating some or all features of your application (use CL)

- Identify the features you want to migrate and the reasons why
  - Evaluate the UI E3 coupling (use migration view)
  - Evaluate the lifetime of your component
  - Prefer the components that have no tests to test them now !
- **Be aware that you will may not be able to migrate the entire application !**

3. Migrating fully to E4 (no CL)

Once features are migrated you can tend to a pure E4 application :

- remove all dependencies to **org.eclipse.ui**
- remove the E3 RCP feature in the TP
- create your top level plugin containing your own application model

C. Migration tooling and advices

Tooling

.

E4 Spies

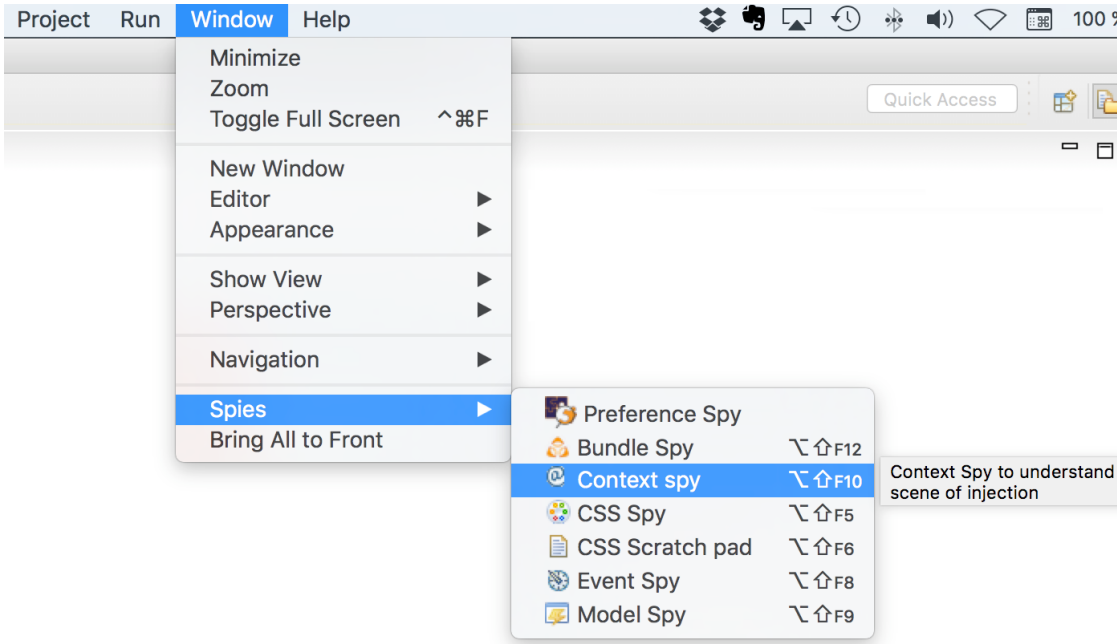
- The E4 spies are essential to develop an E4 application
- They help in browsing the application model, injection contexts, events, css....
- It is possible to write its own spy for any specific data
- To install them, search for 'E4 spies' in the market place
- **Most of the spies are pure E4 plugins**

Using the spies

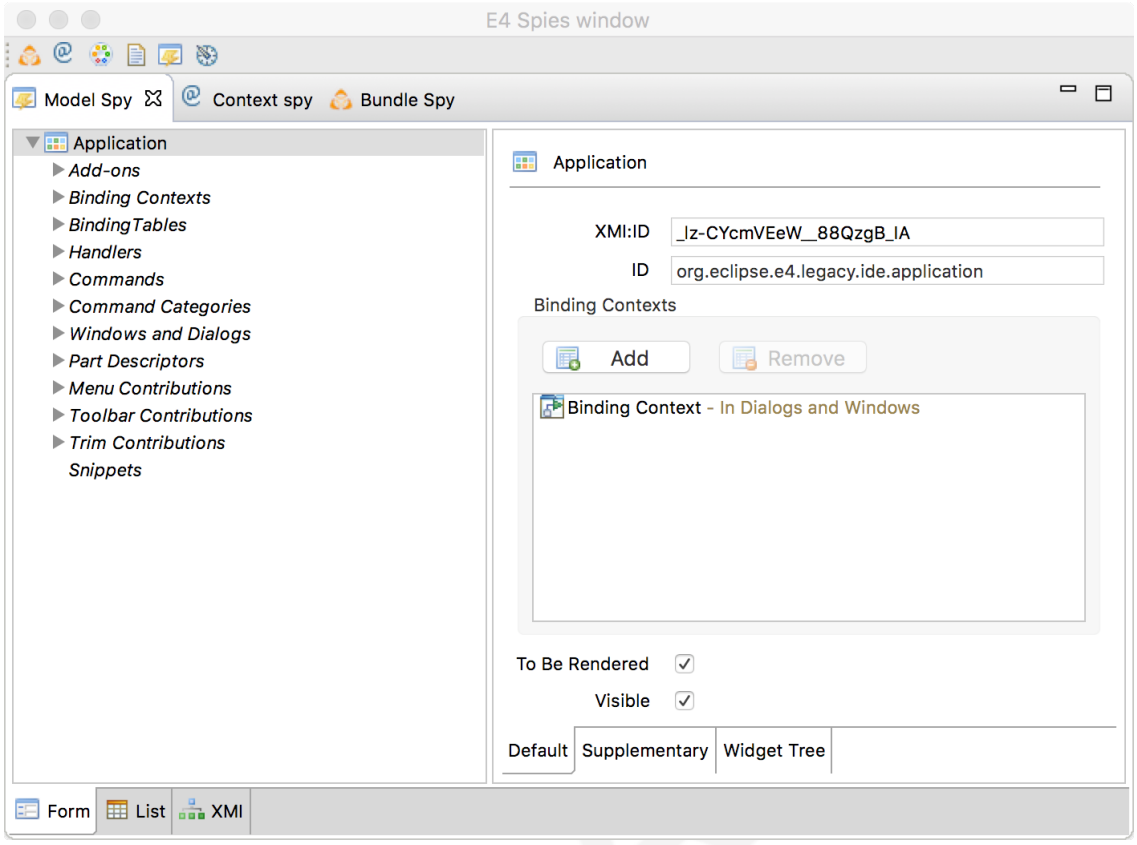
There are 3 different ways to open the spy window :

- use one of the shortcut (**Alt Shift F4** to **Alt Shift F10** for instance) depending on the installed spies
- look for "spy" in the quick access field

- use the Window->Spies menu:



It will open a specific E4 Spies Window with a toolbar to display each spy.
For instance the **Model Spy** :

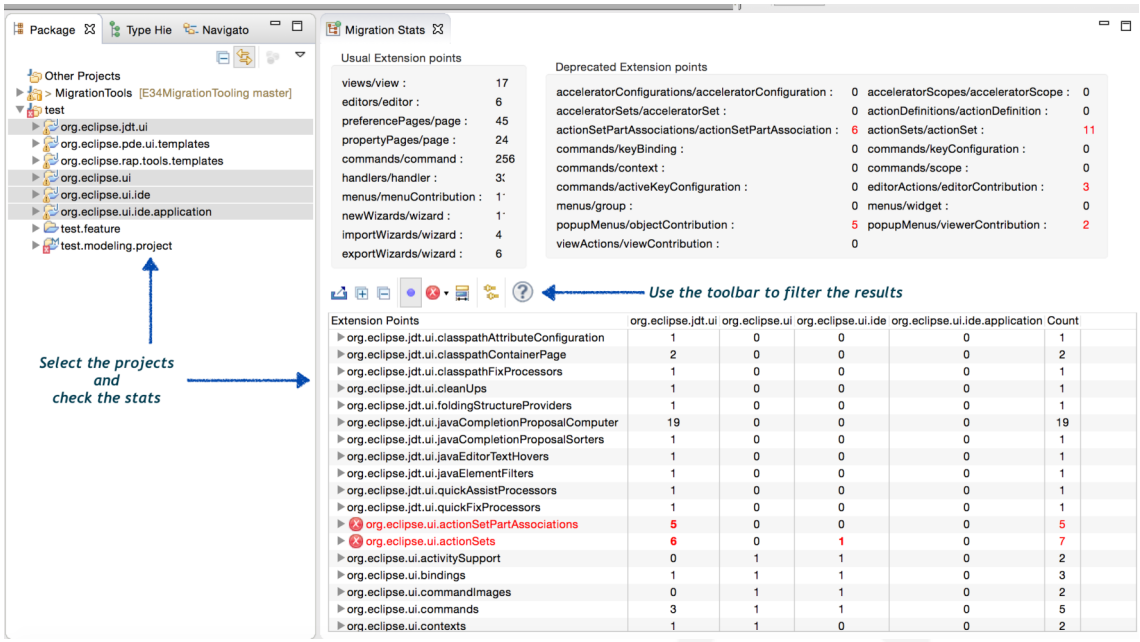


e4 Spies Window

A tooling to help to evaluate the migration cost

- OPCoach developed a specific statistic view dedicated to migration
- This plugin is available on github : **http://opcoach.github.io/E34MigrationTooling/**
- It is delivered under EPL license and it is free

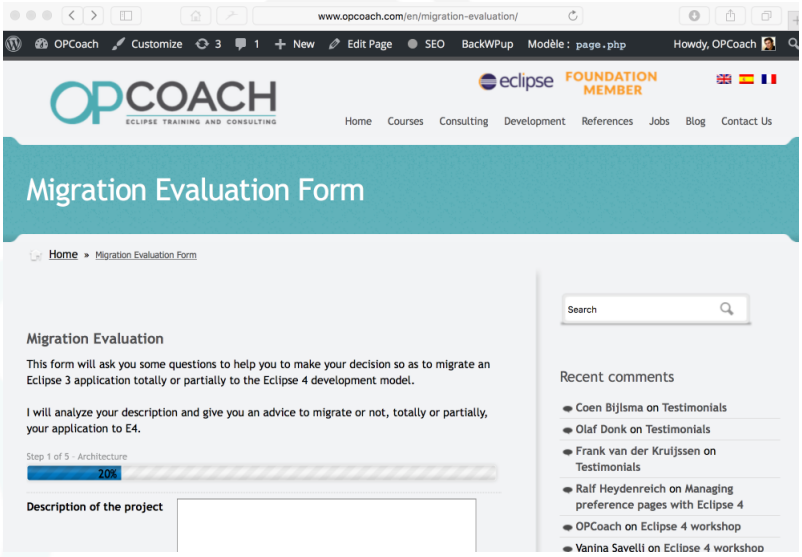
- Select the projects in the workspace and get some statistics about used ui extension points :



Migration Stat View

An evaluation form to check your migration

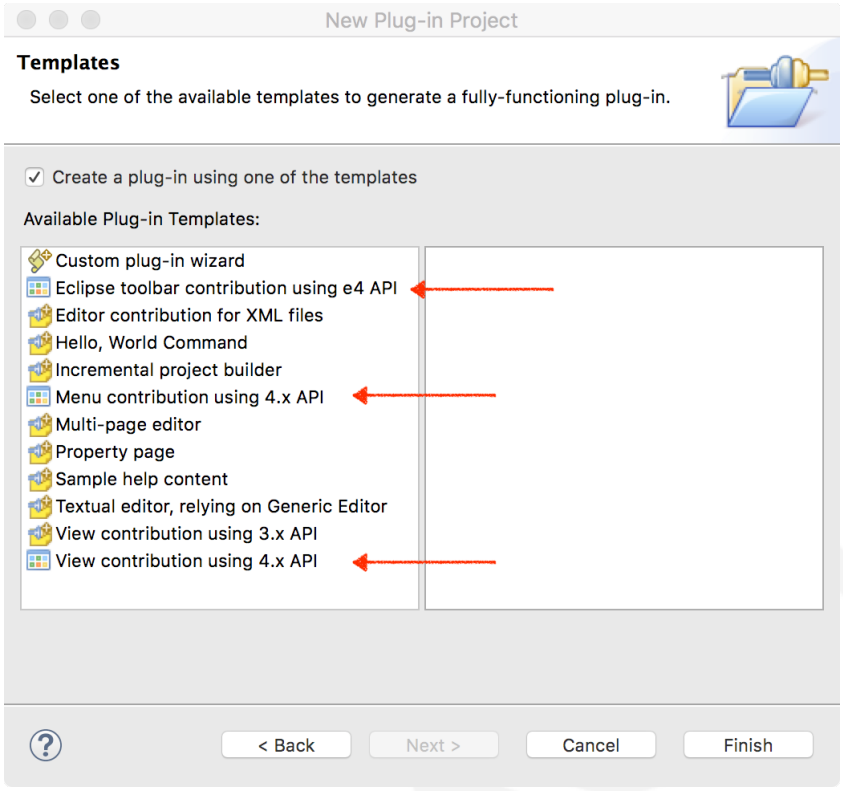
- OPCoach provides a form to help you to evaluate the work
- http://www.opcoach.com/en/migration-evaluation/



http://www.opcoach.com/en/migration-evaluation/

The plugin templates with model fragments

Since Neon, it is now possible to create pure E4 plugins that are using model fragments.
You can use them to start to write your plugins



Organize yourself to manage your migration

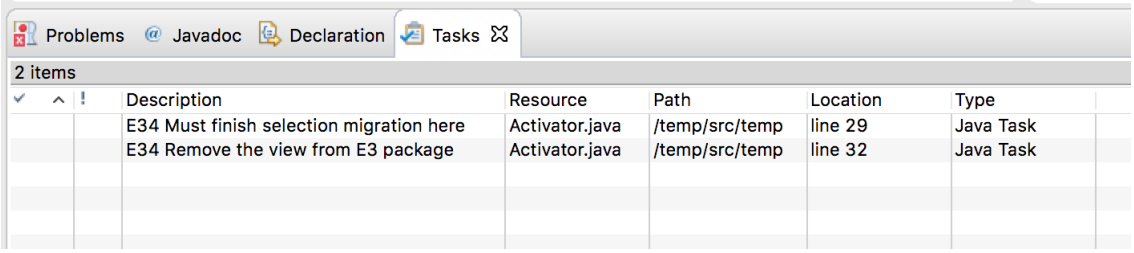
- Create a xxx.e4.xxx package to put the migrated class, **in current migrated plug-in**
  - for instance : xxx.e4.handlers or xxx.e4.parts
- Copy the E3 class and its dependencies in this package and keep the names
- Set the E3 classes as 'deprecated'
- Annotate with a **//E34** comment the current migrated areas when they are not finished
- Remove the old E3 packages when the migration is finished
- **Use git (of course) !**

These tips help to maintain existing plugins and the build process

Displaying the //E34 tasks

It is possible to display the **//E34** comments in the task view :

- open the 'Tasks' view
- add a E34 tag in the preference page of Java->Compiler->Task



E34 tasks

D. Model Fragments and Processors

Fragments and processors

The key concepts to manage your migration

Introduction

You can contribute to an E4 application model by using two mechanisms :

- a **model fragment** : with the ID or xpath of model objects
- a **processor** : with a piece of code modifying the injected application

Model fragment

- The model fragment adds content to the live application model
- To create a model fragment,
  - use the model fragment wizard (Ctrl N + 'fragment')
  - extract a piece of model into a fragment (contextual menu on application model editor)

Application fragment / top level

It is possible to add any contribution to any object using its ID/feature name

If you contribute on the top level application, you can use:

- the ID of the application
- the ID of the legacy E4 application : **org.eclipse.e4.legacy.ide.application**
- the '**xpath:/**' to get any application whatever its ID (see bug #437958)
  - **This is the best practice for the top level contributions**

- Example for the spy fragment:

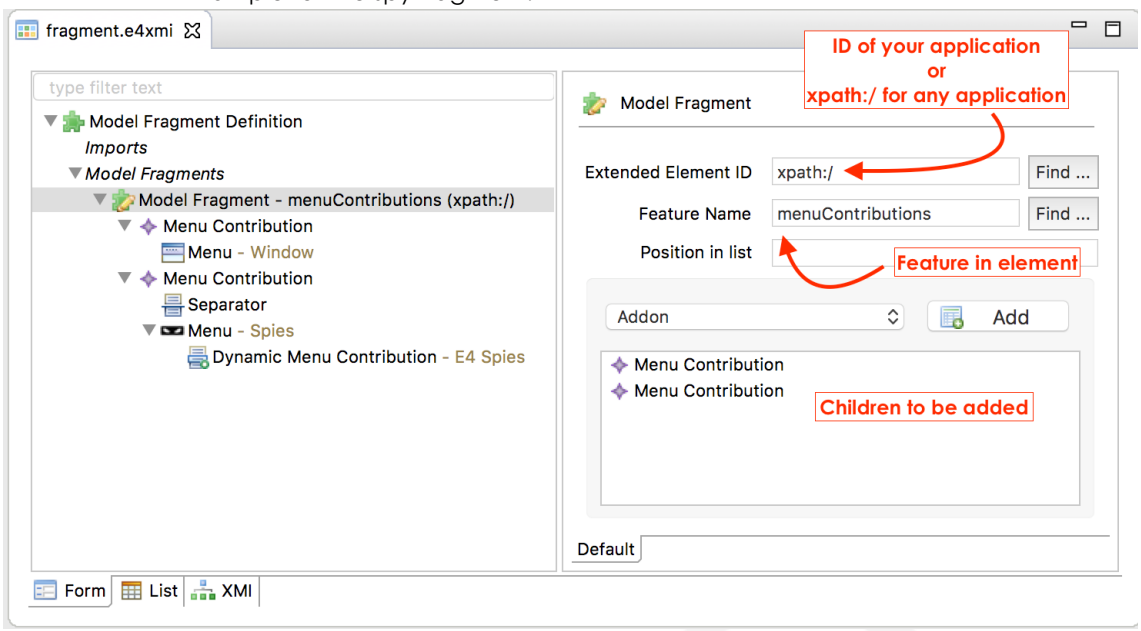


Image 2

Model fragment

Remind that model fragments must be declared in an extension (**org.eclipse.e4.workbench.model**)

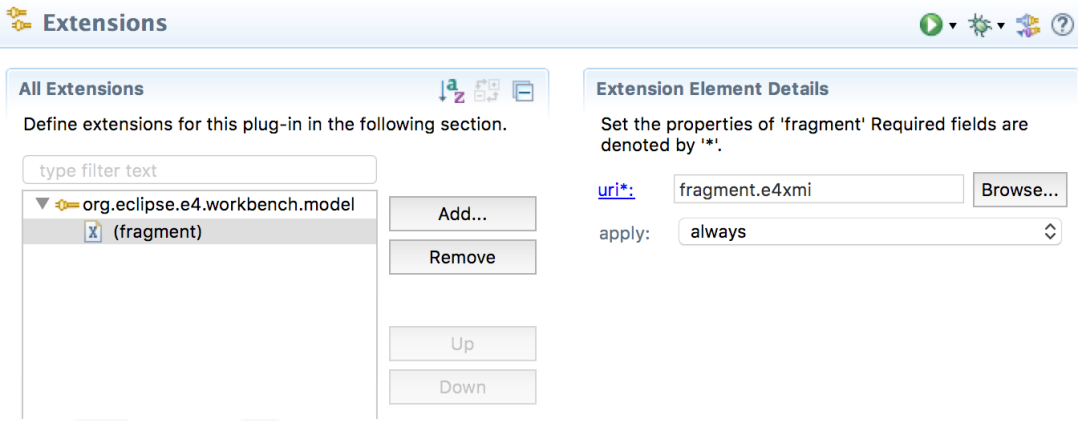


Image 3 Model Fragment

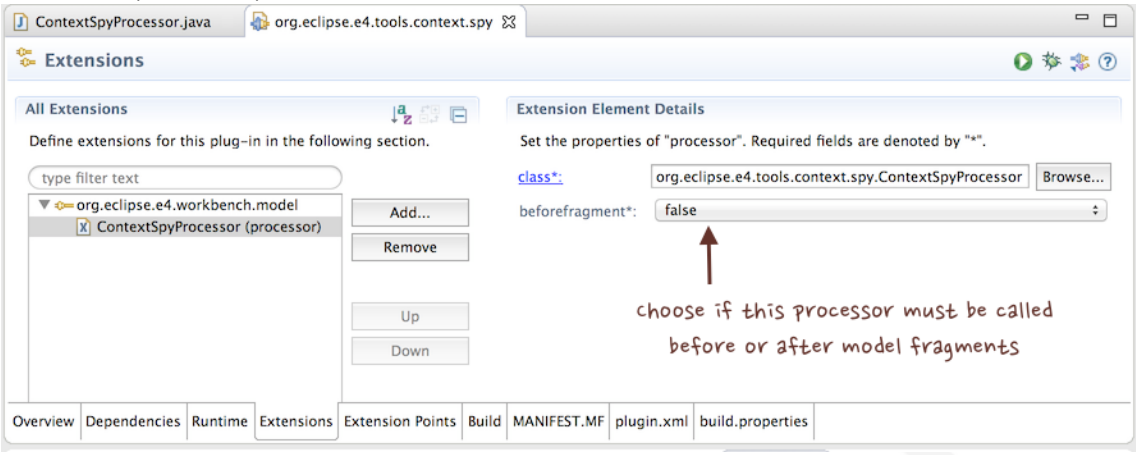
Remarque : Eclipse version for the model fragment editor

To edit the model fragments, use the latest Eclipse version and at least 'Oxygen' which provides a lot of enhancements

Processor declaration

- The processor is used when the object's ID is not known (application for instance)
- The application is received using injection so as to be modified

- It must be declared in the **org.eclipse.e4.workbench.model** extension using a processor parameter:



Extension for a processor

Processor code

- The processor code is a POJO with a **@Execute** annotation
- The method receives the application and needed services as parameters
- Use the **modelService.createElement** method to create instances

```
20
21 /** A sample E4 processor adding a command in application  */
22 public class SampleE4Processor
23 {
24
25     @Execute
26     public void process(MApplication application, EModelService modelService)
27     {
28         // Just create a command and add it in the application
29         MCommand command = modelService.createModelElement(MCommand.class);
30         command.setElementId("id.of.my.command");
31         command.setCommandName("Launch My Commad");
32         String contributorURI = "platform:/plugin/" + FrameworkUtil.getBundle(getClass()).getSymbolicName();
33         command.setContributorURI(contributorURI);
34         command.setDescription("A sample command added in application");
35
36         application.getCommands().add(command);
37
38     }
39 }
40
```

Code for a processor

E. Injection concerns

Introduction

This presentation is not a course on injection.

It has already been presented in different talks and articles

Refer to this articles to be aware of this powerful mechanism

- Talk in boston :
  - *https://www.eclipsecon.org/2013/sites/eclipsecon.org.2013/files/E4_Injection_OPCoach_talk_0.pdf*
- Tutorial about injection :
  - *http://eclipsesource.com/blogs/tutorials/eclipse-4-e4-tutorial-part-4-dependency-injection-basics/*
- Eclipse 4 context usage :
  - *http://www.vogella.com/tutorials/Eclipse4ContextUsage/article.html*
- Eclipse Wiki
  - *https://wiki.eclipse.org/Eclipse4/RCP/Dependency_Injection*

E4 Injection

Principles of E4 injection

- The injector gathers hierarchically all common objects
- Listeners and initializations are simplified :
  - Methods annotated with **@Inject** are called automatically if a parameter changes in the context
  - Fields annotated with **@Inject** are automatically initialized if the value changes in the context
- Allows to have a framework independant of an external library (UI Agnostic)
- Simplify unit tests
- Example for the selection management :

Usage of injection for the selection

Just receive the selection object in the expected type and you will be notified !

```
11
12      /** This method will be invoked only if current selection is a Rental instance */
13      @Inject @Optional
14      public void receiveSelection(@Named(IServiceConstants.ACTIVE_SELECTION) Rental r)
15      {
16          setRental(r);
17      }
18
```

Get the selection

Use case : mixed selection between E3 and E4

In mixed mode, the selection can have different types:

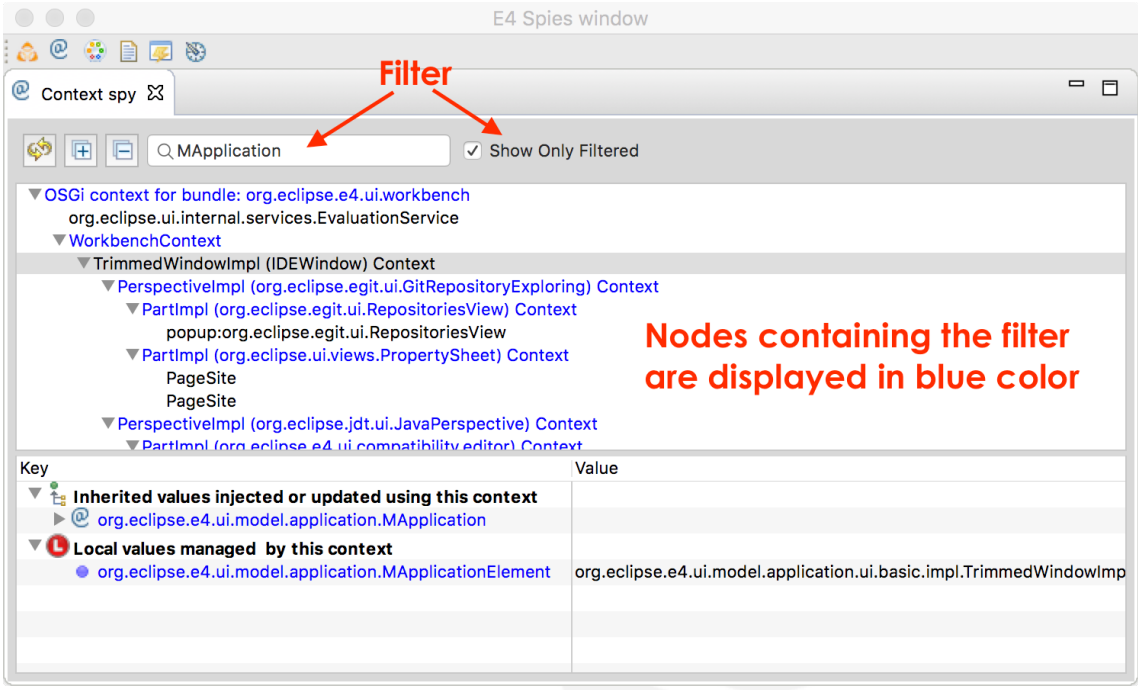
- from the E3 code it is still an **ISelection**
- from the E4 code it is directly the selected type

Be aware to receive the both types in the E4 code

A full example is provided in the 'View contribution using 4.X API'.

Use the context spy to explore your contexts

Open the spy window with the shortcut **Alt Shift F10**:



Use case : E3 and E4 context sharing

During a migration you will need :

- to get an E4 managed value in E3 code
- to get an E3 managed value in E4 code
- listen between E3 and E4 worlds

Everything is possible and you will be able to achieve a smooth migration

Example for E3 code:

```
//.....................................
// Get the application E4 context
//.....................................
IWorkbench workbench = PlatformUI.getWorkbench();
IEclipseContext appliCtx = workbench.getService(IEclipseContext.class);
appliCtx.set("myKeyInAppli", "value");
```

F. Resources

Articles about migration

- Eclipse magazin about migration (german) :
  - *https://jaxenter.de/ausgaben/eclipse-magazin-6-15*
- Recipes for your Eclipse 4 migration (english)
  - https://jaxenter.com/eclipse-migration-tutorial-124527.html
- OPCoach's article in eclipse magazin (german)
  - *http://www.opcoach.com/wp-content/uploads/2015/09/Migration34_EclipseMagazine_final.pdf*
- Comment migrer vers eclipse 4 (french)
  - *http://opcoach.developpez.com/tutoriels/eclipse/migration-e3-e4*

Ask your questions

Feel free to ask your questions

- in E4 forum
- by email :
  - **olivier@opcoach.com**
- or just after this talk

Evaluate the session

Thank you to evaluate this talk

G. Extension Migration (extra)

Reminder

- The compatibility layer transforms E3 concepts to E4 concepts
- To guess how to migrate an element, you can :
  - launch your application using the model and context spy
  - check what the compatibility layer has generated in the model or context.

Migration steps / Core

To migrate a core plugin you must :

- do nothing !
- because there is no dependencie to **org.eclipse.ui**

Migration steps / UI

To migrate an UI plug-in, you must :

- move the ui E3 extensions to a model fragment (or to the application model)
- migrate the relevant code
- remove all E3 extensions
- remove the **org.eclipse.ui** dependency when it is not used anymore
  - add the jface dependency and others instead
- for the RCP main plugin (containing application), create the source application model.

Then, once all the plug-ins have been migrated, it is possible to remove the compatibility layer.

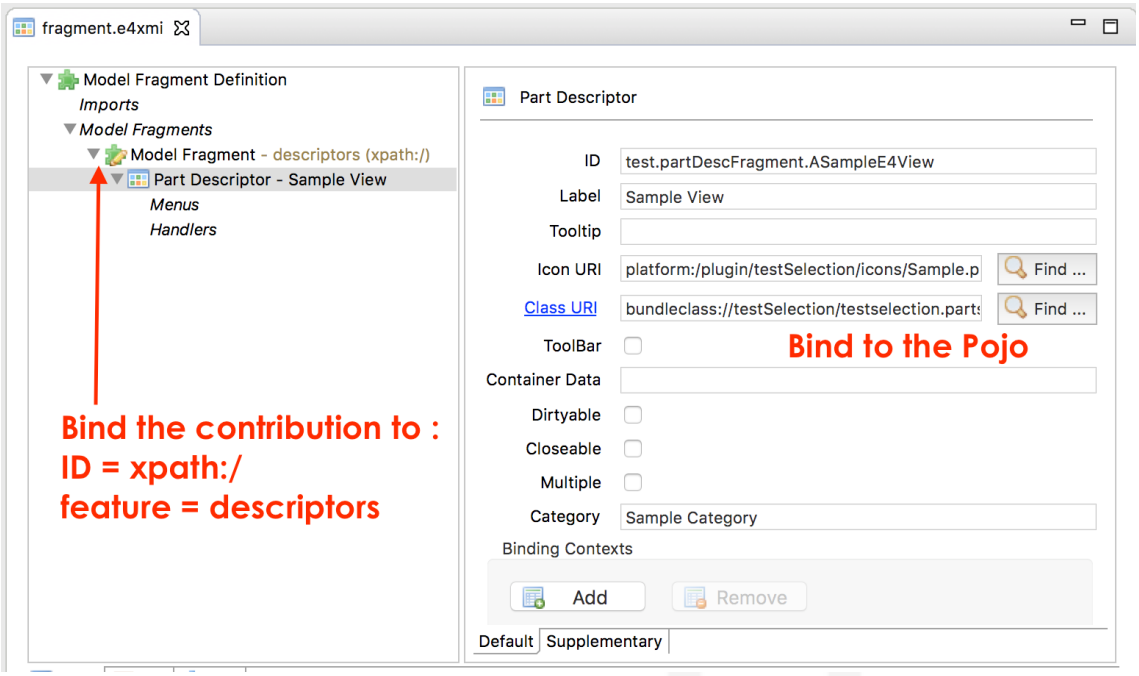
View migration

An **org.eclipse.ui.views** extension is actually a **PartDescriptor** in the application model

To migrate a view :

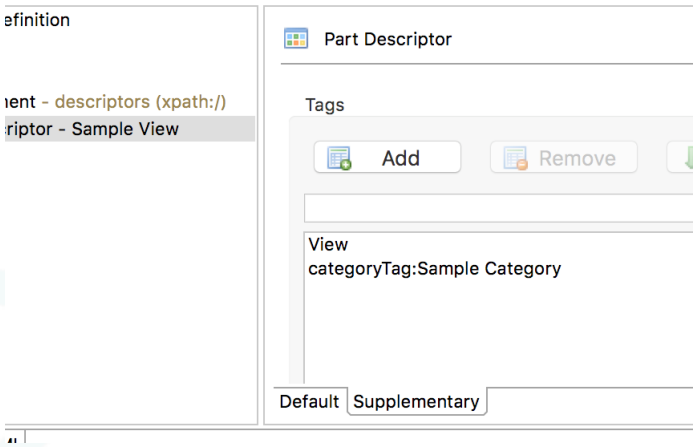
- Copy your ViewPart code in the xxx.e4.parts package
- Transform the code into a POJO :
  - remove inheritance to ViewPart
  - add **@PostConstruct** before the **createPartControl** method
  - add **@Focus** before the **setFocus** method
  - update the code to manage the selection using injection
  - remove the extension and the E3 code

- Bind this pojo in a model fragment :



To make the view appear in the 'Window -> Show view' menu :

- add this tags in the supplementary tab



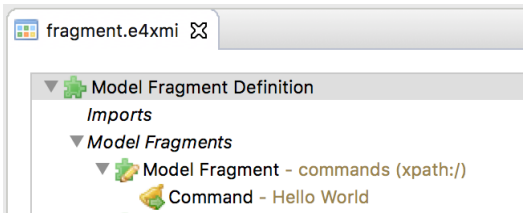
Perspective Migration

- Perspectives must be defined in the application model (using a fragment or a processor)
- There is no 'PerspectiveDescriptor' but you can add it in the application's snippets
- To migrate the Perspective
  - get the perspective description and put it in a model fragment bound in the snippets
  - remove the perspective extension
  - delete the perspective factory code.

Command Migration

An **org.eclipse.ui.command** extension can be defined in the 'commands' feature of the application model

- keep the same ID
- add the command in the fragment :



Handler Migration

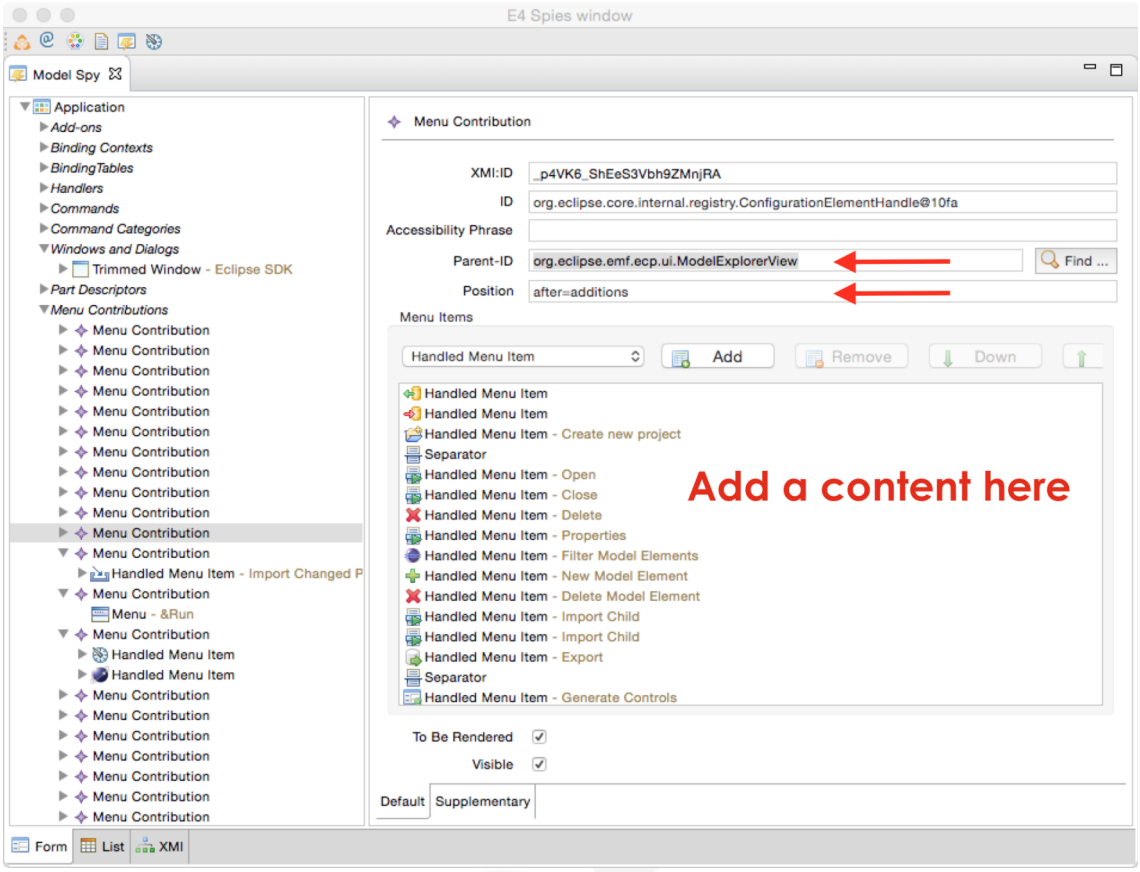
To migrate an **org.eclipse.ui.handlers** extension :

- Copy the E3 handler code in the xxx.e4.handlers package
- Transform the code into a POJO :
  - remove inheritance to AbstractHandler
  - add **@Execute** before the **execute** method
  - add **@CanExecute** annotated method if needed
  - receive needed values as parameters (will be injected)
- Bind this pojo in a model fragment (**xpath:/** and **handlers**)

MenuContribution Migration

An **org.eclipse.ui.menus** extension must be redefined in the model fragment

- use '**xpath:/**' and '**menuContributions**' feature
- The link is done using the parent ID



Menu Contribution

MenuContribution / Parameters

The following parent ID can be used :

- **ID** of an existing view (it must have been registered using the **EMenuService**)
- **ID** of an existing menu
- **org.eclipse.ui.main.menu** : used for the main menu
- **popup** : used to be located in any part
- **org.eclipse.ui.main.toolbar** : used to be located in the main toolbar.

For the position :

- an ID of any existing object (command, menu, etc...)
- **after=additions** : the default location

It is possible to open the model Spy so as to check the values used by the IDE

Wizard migration

- **org.eclipse.ui.[???]Wizards**
- Wizards are not defined in the application model
- There is also no extension point outside of org.eclipse.ui
- Therefore, the main dialog to choose a wizard is not available in a pure E4 application
- Nevertheless it is possible to open a specific wizard in a pure E4 code
- Wizards are only JFace code and can be adapted to deal with injected selection
- They must not implement **INewWizard**, **IImportWizard** or **IExportWizard** anymore

- A command must be created to open the wizard, using the **WizardDialog** of Jface

Sample wizard

SampleWizard.java

```
1  package com.opcoach.training.e4.codesamples;
2
3  import javax.inject.Inject;
8
9  public class SampleWizard extends Wizard
10 {
11     private SampleWizardPage firstPage = null;
12     private IEclipseContext context;
13
14     @Inject
15     public SampleWizard(IEclipseContext ctx)
16     {
17         setWindowTitle("New Wizard");
18         context = ctx;
19     }
20
21     @Override
22     public void addPages()
23     {
24         firstPage = ContextInjectionFactory.make(SampleWizardPage.class, context);
25         addPage(firstPage);
26     }
27
28     @Override
29     public void dispose()
30     {
31         // Don't forget to uninject pages created with make !
32         ContextInjectionFactory.uninject(firstPage, context);
33         super.dispose();
34     }
35
36     @Override
37     public boolean performFinish()
38     {
39         // Do your stuff here by asking the pages...
40         // Use an AntRunner to create projects structures or to manipulate files !
41         return true;
42     }
43 }
```

Use the context to create the pages !

Uninject the injected pages

Sample wizard

Sample wizard page

SampleWizardPage.java

```
1  package com.opcoach.training.e4.codesamples;
2
3  import java.io.File;
13
14 public class SampleWizardPage extends WizardPage
15 {
16     private Object selection;
17     private Label filename;
18
19     @Inject
20     public SampleWizardPage(@Named(IServiceConstants.ACTIVE_SELECTION) Object currentSelection)
21     {
22         super("wizardPage");
23         setTitle("Wizard Page title");
24         setDescription("Wizard Page description");
25         selection = currentSelection;
26     }
27
28     @Override
29     public void createControl(Composite parent)
30     {
31         Composite container = new Composite(parent, SWT.NULL);
32
33         filename = new Label(container, SWT.BORDER);
34         if (selection instanceof File)
35            filename.setText(((File)selection).getName());
36
37         setControl(container);
38     }
39
40     @Override
41     public boolean isPageComplete()
42     {
43         return filename.getText().length() > 0;
44     }
45
46 }
47
```

Inject current selection for content init

Sample wizard page

Opening wizard

```
11 public class OpenSampleWizard {
12     @Execute
13     public void execute(IEclipseContext ctx, Shell s)
14     {
15         Wizard w = ContextInjectionFactory.make(MyWizard.class, ctx);
16         WizardDialog wd = new WizardDialog(s, w);
17         wd.open();
18     }
19
20 }
```

Open wizard

Preference pages Migration

- Like wizards, preference pages are not defined in the application model
- It is possible to use the plugin : *https://github.com/opcoach/e4Preferences*[2]
- You need to :
  - ensure that your preference pages are extending **FieldEditorPreferencePage**

2 - https://github.com/opcoach/e4Preferences

- change the extension **org.eclipse.ui.preferencePages** to **com.opcoach.e4.preferences.e4PreferencePages**
- add the handler and the command in your model

For the default values, you can keep the **org.eclipse.core.runtime.preferences** extensions.

Other migrations

- There are still plenty tips for your migration
- Try to put it in your model fragment
- If you can not describe your contribution in a model fragment, use a model processor