Migration Eclipse 3 to Eclipse 4

June 6th 2013

EclipseCon France 2013

Table des matières

Presentation

OPCoach

Image 1

- **Training** : RCP, E4, Modeling, Build, given in French, English and ... Spanish (2013)
- **Consulting**
- **Recruitment service to link companies and job applicants**
- Web site : *http://www.opcoach.com/en*
- Twitter : **@OPCoach_Eclipse**, **@OPCoach_Job**

E3 to E4 migration.

Outline

- Main concepts in Eclipse 4
- What, Why, Who, When should we migrate ?
- Method for migration

Main concepts of Eclipse 4

- Application model
- Injection
- Compatiblity Layer

The E4 application model

- It is a global model that brings together the usual extension points:
  - view, perspective, menus (visual)
  - command handlers, key bindings (non-visual)
- It simply describes the structure of the UI without detailing its contents
- Its structure is defined by an Ecore meta model
- It can be edited with a dedicated editor
- It can be modified and UI is refreshed
- This model is independent of the display
  - a 'renderer' allows specific display (swt and javafx)
- Classes referenced in the application model are simple annotated POJOs

Application model

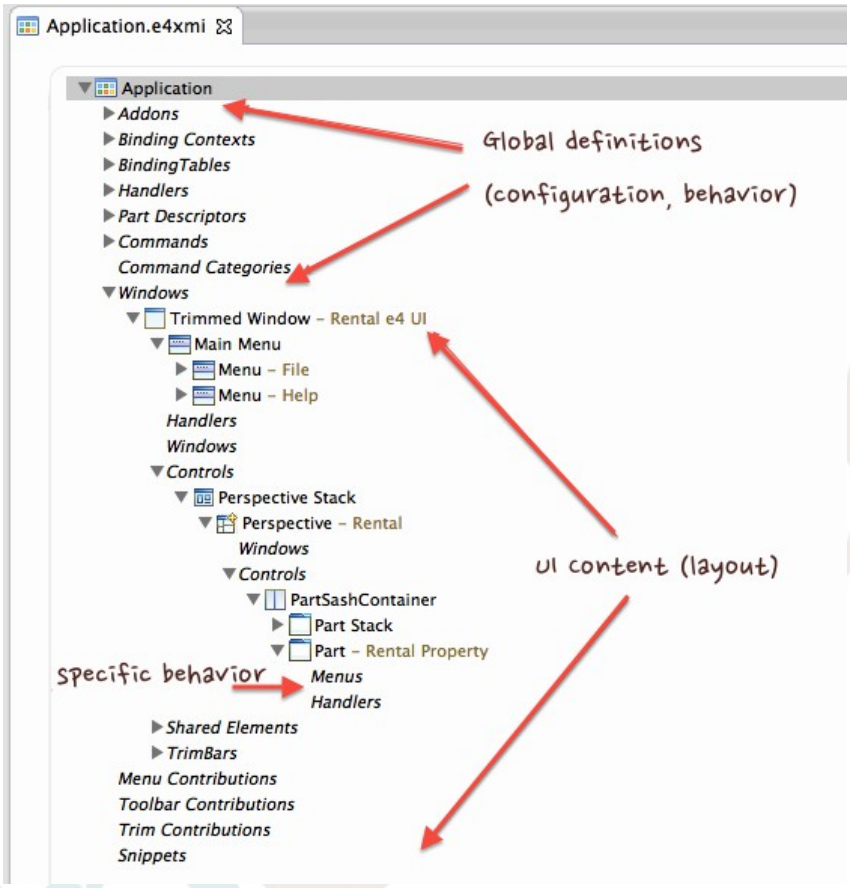

Image 2 Application Model

Injection

- The goal of injection is to delegate object initializations to a framework
- Injection works by using a **context** that contains the values
- We use the annotation **@ Inject** (Javax.inject) to inject the values
- It can be applied to a constructor, a method or a field.

SampleInjectedClass.java

```
1  package com.opcoach.training.e4.codesamples;
2
3  import javax.inject.Inject;
4
5  import org.eclipse.e4.core.di.annotations.Optional;
6
7  public class SampleInjectedClass
8  {
9      // An injected field
10     @Inject
11     private MyService service;
12
13     @Inject
14     public SampleInjectedClass(Object1 object, @Optional Object2 o2)
15     {
16         // An injected constructor with 2nd parameter optional
17     }
18
19     @Inject
20     public void anInjectedMethod(Object2 o2)
21     {
22         // An injected method
23     }
24
25 }
26
```

Image 3 Sample injected class

Basic rule of injection

If an object is modified in the context, it will be reinjected automatically :

- in a field
- as a parameter of a method that will be called again

A lot of listeners can be replaced !

```
116    @Inject
117    public void setSelection(@Optional @Named(IServiceConstants.ACTIVE_SELECTION) Object o,
118                             Adapter adapter)
119    {
120        Rental r = adapter.adapt(o, Rental.class);
121        setRental(r);
122
123    }
124
```

Image 4 @Inject @Named @Optional

Contexts for an application

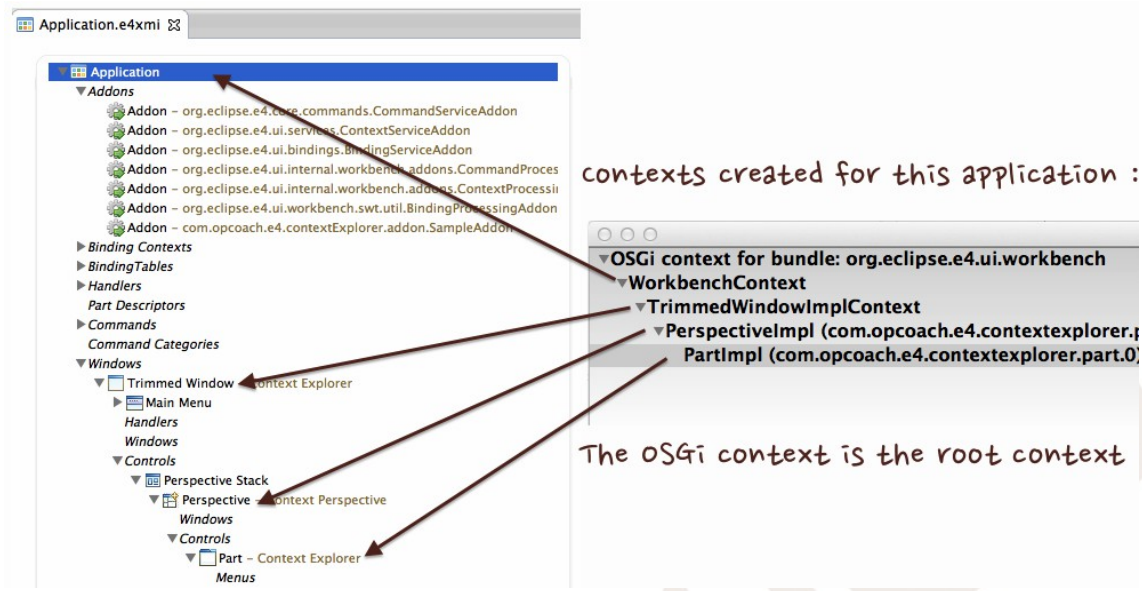

Image 5 Contexts in application

Context explorer

- The context explorer is used to display all the contexts and their contents
- Available on github : **https://github.com/opcoach/contextExplorer**
- Instructions to use it in your application :
  - in french : *http://www.opcoach.com/2013/03/e4-context-explorer/*
  - in english : *http://www.opcoach.com/en/2013/03/e4-context-explorer/*
  - in spanish : coming soon (check e4 opcoach blog) !

Compatibility layer

The compatibility layer uses the E4 engine to run E3 based application

It provides a default application model : the LegacyIDE.e4xmi

It transforms extensions into objects in the application model:

- a view extension becomes an instance of **MPart** in the model
- a perspective extension becomes an instance of **MPerspective** in the model
- etc ...

The Eclipse 4.2 IDE is launched with the compatibility layer

Launching the compatibility layer

We need to create a new launch configuration

We launch a regular E3 RCP application (which uses advisors)

Simply add the following plugins in the launch configuration:

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

By adding the live editor, you can also consult the E4 generated model:

- add **org.eclipse.e4.tools.emf.liveeditor** your launch configuration

- trigger the shortcut: **Alt Shift F9**

What kind of migrations can we do ?

Full Migration

Goals :

- all plugins are running with Eclipse 4 platform
- application model is defined in your main plugin
- compatibility layer is no more used

Conditions :

- your application contains a lot of swt/jface and/or core business model code
- you do not use standard views of Eclipse like console, properties,...
- there is a low coupling with E3 APIs

Examples of candidate RCP applications :

- a simulation application
- a computation application
- a database display application
- ...

Partial migration

Goals :

- only some features of your application are migrated
- model fragment will provide the migrated code
- application model is provided by the compatibility layer which is still used

Conditions :

- your application is composed of isolated features that can be easily migrated
- you plan to develop new features using Eclipse 4

Examples of candidate RCP applications:

- The Eclipse IDEs : Juno, Kepler...
- An application composed of independant features with low coupling with E3 APIs

No Migration

No migration is also a solution !

Several reasons to refuse the migration :

- your application is old and will not change
- your application is not well designed
- your application uses too much E3 stuff that has not migrated yet
- you have no time and E3 platform is still sufficient for your needs

Why should we migrate ?

Reasons for migration

- Since June 2012, Eclipse 4 has been the official architecture for new developments !
- Application model is dynamic and platform agnostic (SWT, Java FX...) thanks to POJOs
- Injection is pretty cool and reduces the amount of code
- Eclipse 4 event notification system (**IEventBroker**) is very concise and easy to use with

injection
- You want to use the CSS styling capability of Eclipse 4
- Your application will still live several years and it is time to refactor it

What are the prerequisites for migration ?

A good and clean architecture is necessary

Your application **must be clean** !
- UI and core plugins must be clearly separated
- Features must be clearly defined
- Coupling with Eclipse 3 must be clearly identified
- Packages in ui plugins must be sorted correctly : views, handlers,...

What should we migrate ?

What should we migrate or not ?

You have to migrate:
- At least UI plugins
- Core plugins if you want to use injection
- Some unit test fragments
- The maven parent pom file (to change the target platform)

You don't have to migrate :
- the core plugins that do not use injection
- the fragments containing native code or i18n files
- other unit test fragments
- the features
- the maven pom files in each artifact

What is not available today ?

Unavailable Eclipse 4 items

In a pure Eclipse 4 application, those items are not available easily :
- preference pages (there is a 'hack'. Cf github opcoach training e4)
- wizards
- property pages
- online help (static and contextual)
- standard views (console, log, error, progress)
- common navigator framework
- the update manager
- standard commands (you must redefine all)

When should we migrate ?

Scheduling

- you can start now if conditions are ok (good architecture, isolated features, low E3 coupling...)
- use the Kepler release with the latest E4 tools
- try some samples and use git branches !

Who can migrate ?

Actors for migration

Obviously a trained team !

Developper who :

- knows the E4 concepts
- has already written a simple E4 application
- has a global view of the application to migrate
- has software architecture skill
- is curious and patient !

How can we migrate ?

A. Partial migration to Eclipse 4

Keep the 3.X code but migrate some code to E4

- So as to prepare a future migration
- The compatibility layer is still used
- We continue to write E3 code but some code can be written as POJOs
- Example: writing a 4.X POJO part but binding it on a 3.X view.
- You must install the E4 tools bridge for 3.X:

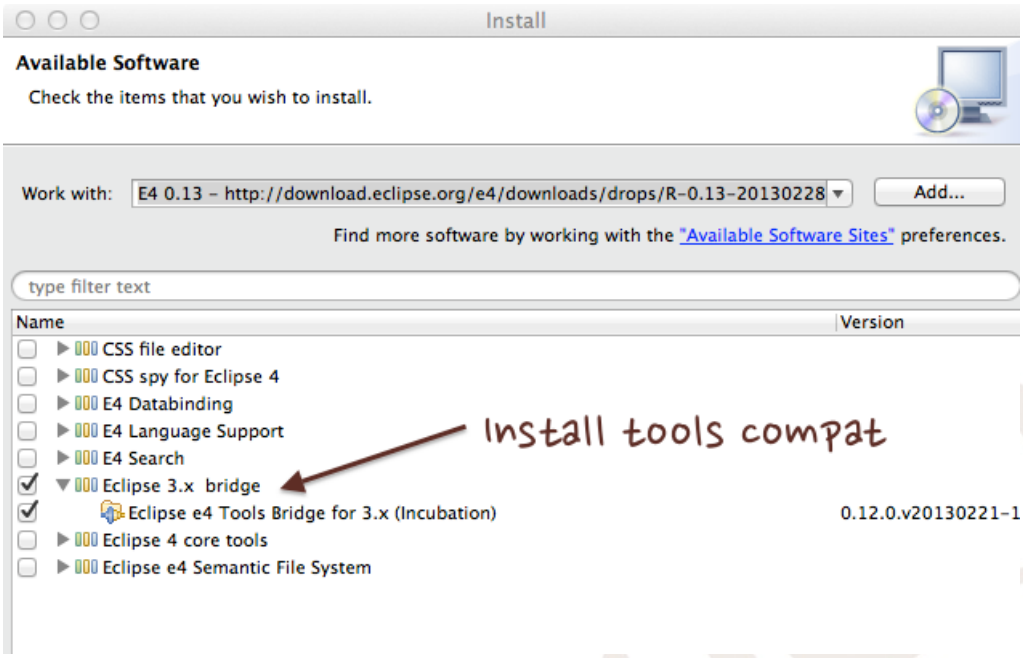

Image 6 Tools bridge for 3.X

Writing a DIViewPart

DIViewPart<T> class can make the bridge between:

- one Eclipse 3 ViewPart
- one Eclipse 4 Pojo

```java
public abstract class DIViewPart<C> extends ViewPart {
    private IEclipseContext context;
    private Class<C> clazz;
    private C component;

    public DIViewPart(Class<C> clazz) {
        this.clazz = clazz;
    }

    @Override
    public void init(IViewSite site) throws PartInitException {
        super.init(site);
        context = PartHelper.createPartContext(this);

        context.declareModifiable(IViewPart.class);

        context.set(IViewPart.class, this);
    }

    @Override
    public void createPartControl(Composite parent) {
        component = PartHelper.creatComponent(parent, context, clazz, this);
    }

    protected IEclipseContext getContext() {
        return context;
    }

    public C getComponent() {
        return component;
    }

    @Override
    public void setFocus() {
        try {
            ContextInjectionFactory.invoke(component, Focus.class, context);
        } catch (InjectionException e) {
            // TODO: handle exception
        }
    }
}
```

receive the Pojo as class parameter

This view part has its own context

call the @Focus method

Image 7 DIViewPart

DIViewPart use

The Eclipse 3 wrapper is written in a few lines of code.

It will disappear completely during migration, and POJO will survive

```java
import org.eclipse.e4.tools.compat.parts.DIViewPart;

public class ViewAsViewPart extends DIViewPart<ViewAsPojo>{

    public ViewAsViewPart() {
        super(ViewAsPojo.class);
    }

}
```

must have an empty constructor !

The viewpart is only a wrapper to the pojo view class

Image 8 DIViewPart usage

DIEditorPart

There is a similar class for Editors : **DIEditorPart<T>**

```
34 public abstract class DIEditorPart<C> extends EditorPart implements IDirtyProviderService {
35     private IEclipseContext context;
36     private C component;
37     private Class<C> clazz;
38     private boolean dirtyState;
39
40     private int features;
41
42     protected static final int COPY = 1;
43     protected static final int PASTE = 1 << 1;
44     protected static final int CUT = 1 << 2;
45
46
47     public DIEditorPart(Class<C> clazz) {
48         this(clazz, SWT.NONE);
49     }
50
51     public DIEditorPart(Class<C> clazz, int features) {
52         this.clazz = clazz;
53         this.features = features;
54     }
55
56     @Override
57     public void doSave(IProgressMonitor monitor) {
58         IEclipseContext saveContext = context.createChild();
59         ContextInjectionFactory.invoke(component, Persist.class, saveContext);
60         saveContext.dispose();
61     }
62
```

Receive the POJO editor as class parameter

call the @persist method

Image 9 DIEditorPart

But there are some limitations

- Warning, there are some discussions about the selection management :
  - See forum (selection with tool compat wrapper) :
    - *http://www.eclipse.org/forums/index.php/mv/msg/367926/897287/*
  - Or bug 386329 : *https://bugs.eclipse.org/bugs/show_bug.cgi?id=386329*
  - Or bug 403930: *https://bugs.eclipse.org/bugs/show_bug.cgi?id=403930*[1][2]
- To sum up :
  - there are 2 different selection services (E3 and E4) which are not connected
  - the selection must be fixed by an E3 mechanism
  - the selection may be received using injection
  - **it seems that the selection can not be set by E4 mechanism yet**
  - follow these 2 bugs !

B. Full Migration

Assumptions

- In this case, the compatibility layer is no more used
- We must now launch a **org.eclipse.e4.ui.workbench.swt.E4Application**
- We must create an Application model
- Each component must be migrated step by step according to their type (control, view, etc. ..)

1 - https://bugs.eclipse.org/bugs/show_bug.cgi?id=386329
2 - https://bugs.eclipse.org/bugs/show_bug.cgi?id=403930

Migration to the application model

Rewrite an implementation model of an E3 application can be tedious

To simplify this task, you can follow this procedure:

- in the E3 launch configuration add the following plugins:
  - the compatibility layer
  - the live model editor
- launch your E3 application
- open the live editor (Alt Shift F9)
- find the XMI code for your component (perspective, control ...)
- copy this extract in the application model
- adapt the extract according to the context (for view, put an URL to a POJO for example)

General Eclipse 3 Code migration

For all classes you must:

- remove all singletons and get them by injection
- inject the services
- replace the E3 selection mechanism with E4 pattern

selection code migration

Selection in Eclipse 3 is defined with an implemention of a **ISelectionProvider** and **ISelectionListener**.

In Eclipse 4, you must:

- intercept selection earliest (on addSelectionListener of a **TreeViewer** for example)
- set the current selection in the **ESelectionService**
- we can 'digest' the **IStructuredSelection** to get the selected object
- (Also choose whether to extract multiple selection as an array or Collection)
- The **ESelectionService** is received by injection

```
106  @Inject
107  private ESelectionService selectionService;
108
109  private void provideSelection()
110  {
111      // attach a selection listener to the viewer
112      agencyViewer.addSelectionChangedListener(new ISelectionChangedListener() {
113          public void selectionChanged(SelectionChangedEvent event)
114          {
115              // Get the selection in event
116              IStructuredSelection sel = (IStructuredSelection) event.getSelection();
117              // set the selection to the service
118              selectionService.setSelection(sel.size() == 1 ? sel.getFirstElement() : sel.toArray());
119          }
120      });
121  }
122
123
```

listen to Selection and extract it as object or Array

Set the selection in SelectionService

Image 10 Selection migration

The active selection is also received by injection:

```java
    @Inject
    public void receiveSelection(@Optional @Named(IServiceConstants.ACTIVE_SELECTION) Object o)
    {
        if (o instanceof Rental)
            setRental((Rental) o);
    }
```

Image 11 Get the selection

AbstractUIPlugin migration

UI Activators in E3 extends **AbstractUIPlugin** (E3 workbench).
This inheritance must be removed but the Activator has no access to injection contexts
These initializations should be moved into an **Addon** :

- extension point initialization
- **ImageRegistry** initialization
- **PreferenceStore** management
- Singletons should be removed (and put in the context)

The **Addon** has these features :

- it is instanciated using injection
- it can get the **IExtensionRegistry** using injection
- it can fill an **IEclipseContext**

```java
public class RentalAddon implements RentalUIConstants
{

    @PostConstruct
    void startRentalFeature(IEclipseContext ctx, IExtensionRegistry reg)
    {
        // Put objects in context
        ctx.set(RentalAgency.class, RentalAgencyGenerator.createSampleAgency());
        ctx.set(RENTAL_UI_IMG_REGISTRY, getLocalImageRegistry());

        ctx.set(RENTAL_UI_PREF_STORE, getPreferenceStore());

        // Read the palettes extensions and publish it in context
        readPaletteExtensions(reg);
        ctx.set(PALETTE_MANAGER, paletteManager);

        // Set the current palette in context...
        String palId = prefStore.getString(PREF_PALETTE);
        ctx.set(Palette.class, paletteManager.get(palId));

    }
```

Image 12 Addon Sample

Addon use

To use the Addon you can :

- reference it in the application model in the Addon part
- add a model fragment added in current application model

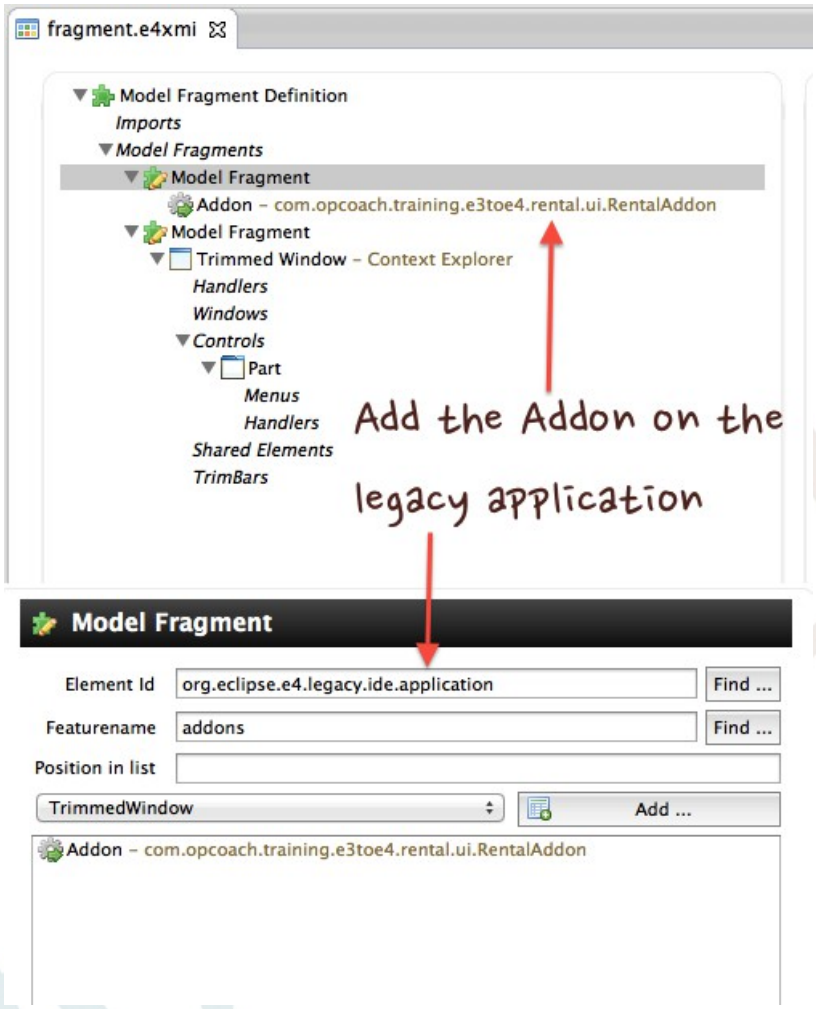

Image 13 Addon in fragment

Migrating views

The views become simple POJO and extensions disappear.

For each ViewPart you must:

- remove inheritance on ViewPart
- annotate the createPartControl method with **@PostConstruct**
- annotate the setFocus method with **@Focus**
- apply the migration API (singletons, services, ...)
- reference the pojo in the application model with bundleclass ://pluginID/qualifiedName

It is always possible to edit the part with WindowBuilder !

Migrating editors

The editors also become simple POJO.

To migrate, you must apply the views migration

And you should:

- a **MDirtyable** injected as fields
- annotate with doSave **@Persist** which updates the dirtyable

Migrating perspectives

Perspective's Code disappears (IPerspectiveFactory)
The perspective appears directly in the application model.
Just copy the XMI code obtained with the live editor and readjust if it is complex
or recreate it by hand if it is easyer

Migrating org.eclipse.ui.command

Commands are now in the application model.
For each **org.eclipse.ui.commands** extension, you must:

- copy the extension in the application model (part command)
- remove the extension
- keep the same ID

It may help to copy the XMI model displayed by the live editor.

Migrating org.eclipse.ui.handlers

Handlers are classes but become POJOs
For each handler extension you must:

- rewrite the extension in the 'handler part' of the application model
- remove the extension

For each code deriving from **AbstractHandler** you must:

- suppress inheritance
- apply the migration API (singletons, services, ...)
- annotate the execute method with **@Execute**
- inject the necessary settings and remove the ExecutionEvent

Migrating org.eclipse.ui.menus

For each ui.menus extension, you must:

- add a **HandledMenuItem** at the location designated by the URI location
- reference the command
- remove the extension

Warning : if you add commands and unfortunately they stay disabled :

- Add this addon : org.eclipse.e4.ui.internal.workbench.addons.HandlerProcessingAddon

Migrating styles in the SWT code

If the SWT code contains calls to styling methods (like setForeground...), you must:

- remove it
- replace it with CSS referenced in the product extensions

C. Tips and Links

Some tips for your migration

You use an Eclipse 4 concept and you don't know where it is defined :

- See : *http://www.opcoach.com/en/2013/05/dependencies-to-use-for-eclipse-4/*

Your application seems to have a strange behaviour after some changes ?

- clear the runtime workspace when you relaunch it !

Your **@PostConstruct** never seems to be called :

- **ALWAYS use import-package for javax.inject and javax.annotations** instead of plugin dependencies
- see :
  - FAQ Eclipse 4 (§4.1) : *http://wiki.eclipse.org/Eclipse4/RCP/FAQ*[3]
  - Bug 348123 : *https://bugs.eclipse.org/bugs/show_bug.cgi?id=348123*

Some links to continue...

Interview of Paul Webster about Eclipse 4 evolutions :

- *http://www.infoq.com/interviews/paul-webster-eclipse-platform-UI*

An interesting post of Win Jongman about the reasons to use the Eclipse 4 architecture :

- *http://industrial-tsi-wim.blogspot.fr/2012/10/why-eclipse-e4-egg-laying-woolmilkpig.html*

Several migration tutorials :

- from lars : *http://www.vogella.com/articles/Eclipse4MigrationGuide/article.html*
- from eclipse source : *http://developer.eclipsesource.com/tutorials/#eclipse4*

Questions ?

Stay in touch

- Come to the booth !
- Register to mailing list on http:www.opcoach.com

@OPCoach_Eclipse : for OPCoach activity and Eclipse news
@OPCoach_Job : for Eclipse jobs (applicants, missions...)

3 - http://wiki.eclipse.org/Eclipse4/RCP/FAQ#Why_isn.27t_my_.40Inject-able.2F.40PostConstruct_methods_being_injected.3F