

What every Eclipse Developer should know about Eclipse 4

Jonas Helming
EclipseSource

Preparation

- Get a fresh Eclipse 4.2 SR1 Modeling Edition including e4 tools
(<http://downloads.efxclipse.org/tutorial/sdks/>)
- Please download the example solutions
- Import start.zip
 - File => Import
 - Existing Projects into Workspace
 - Select Archive and Import
- Run the product once to check the set-up

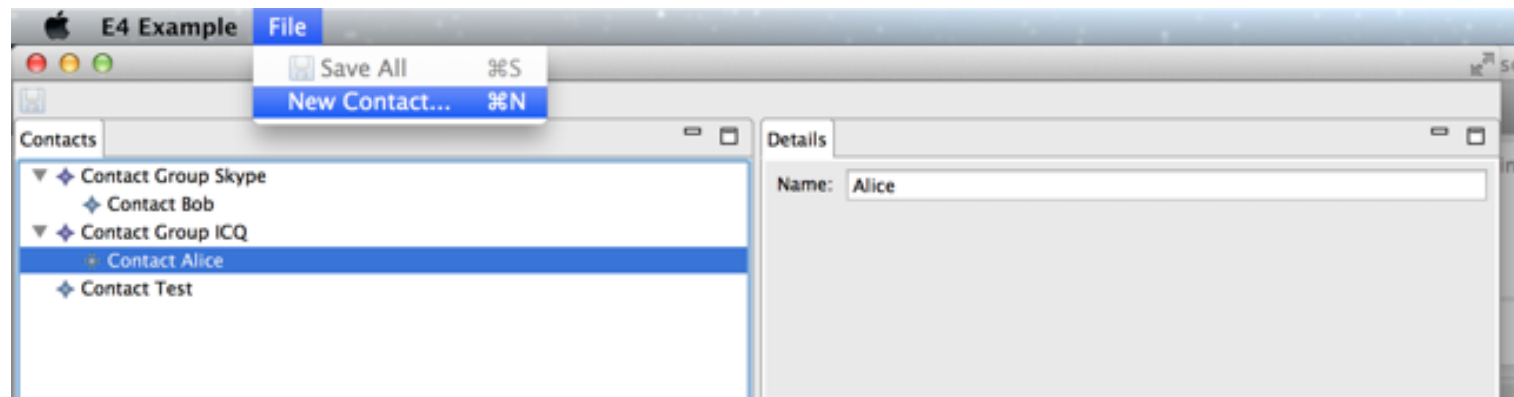
What we will not do today

- Compare Eclipse 4 with 3.x
- Show all parts of Eclipse 4, e.g.:
 - CSS
 - All Services
 - All Application Model Elements
 - Event Brooker
 - (...)
- Learn about SWT and JFace

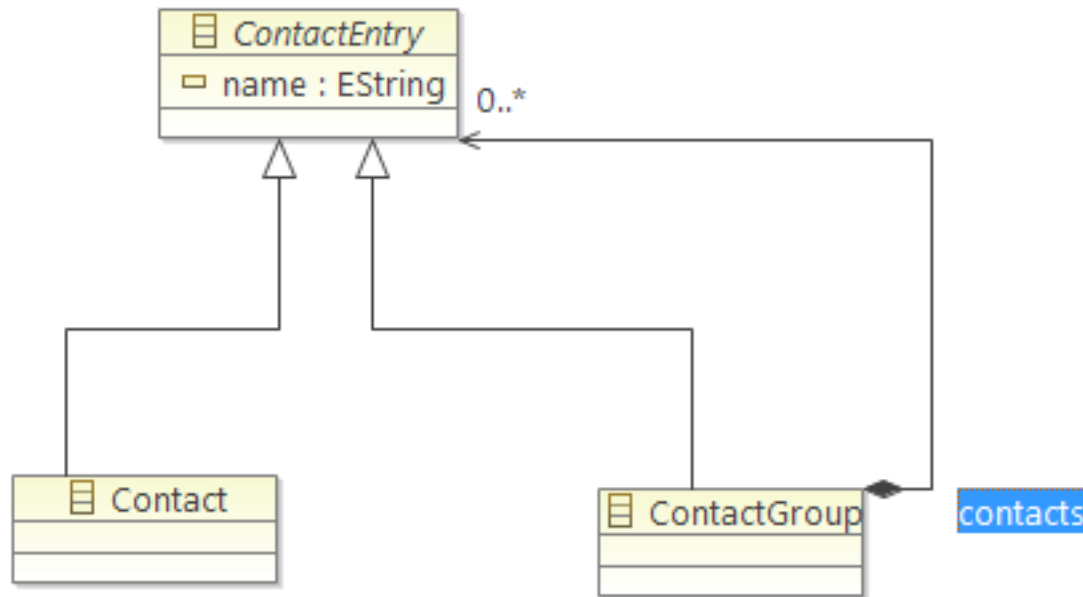
Outline

- The Application Model
- Implementing Views
- Selection Service
- Handlers, Commands and Items
- Editors
- Dependency Injection Details
- Modularity
- Migration

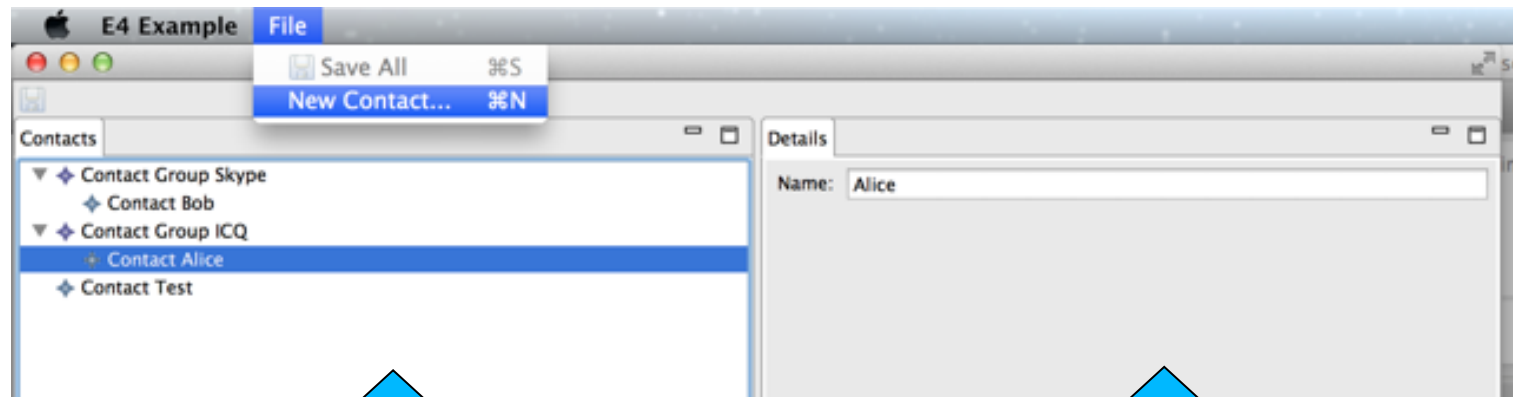
Example Application



The Application entity model



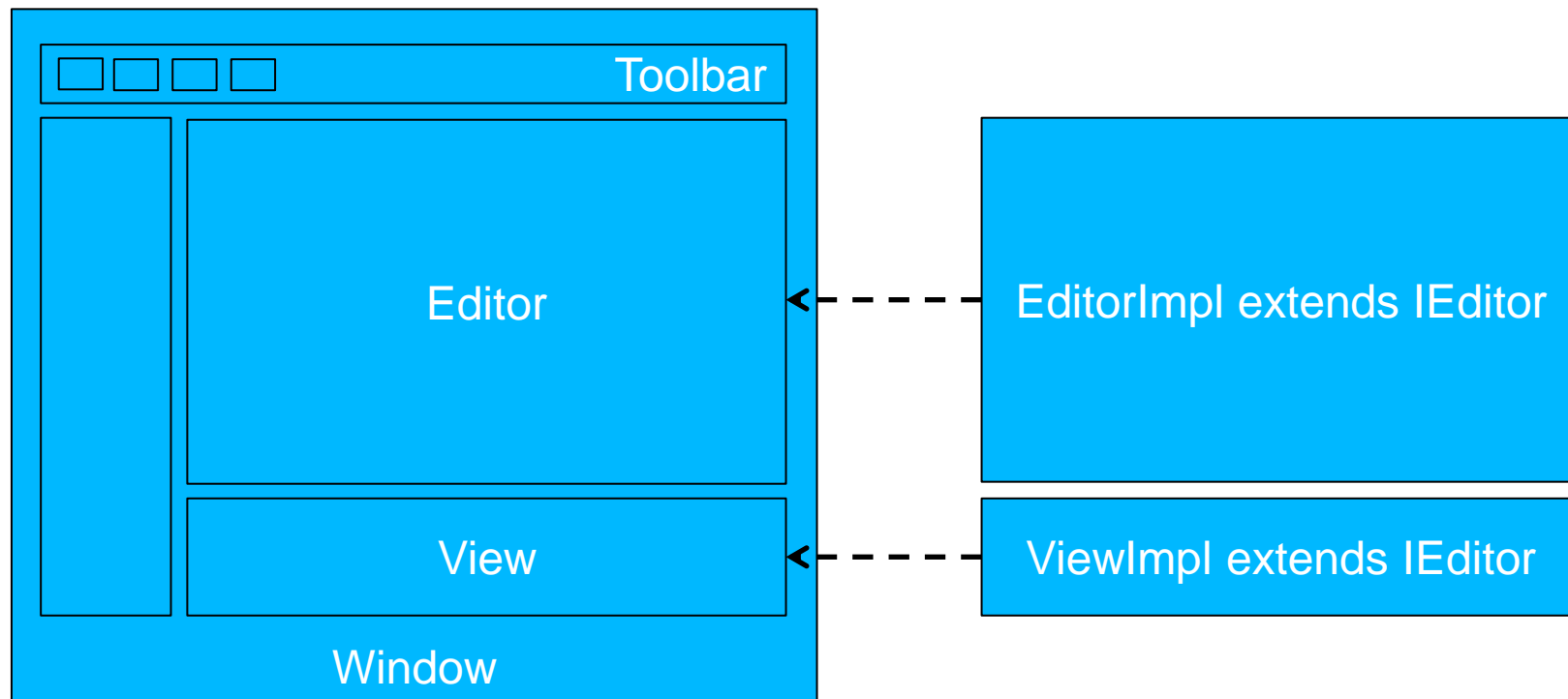
Goal 1: Two Views



ContactsView

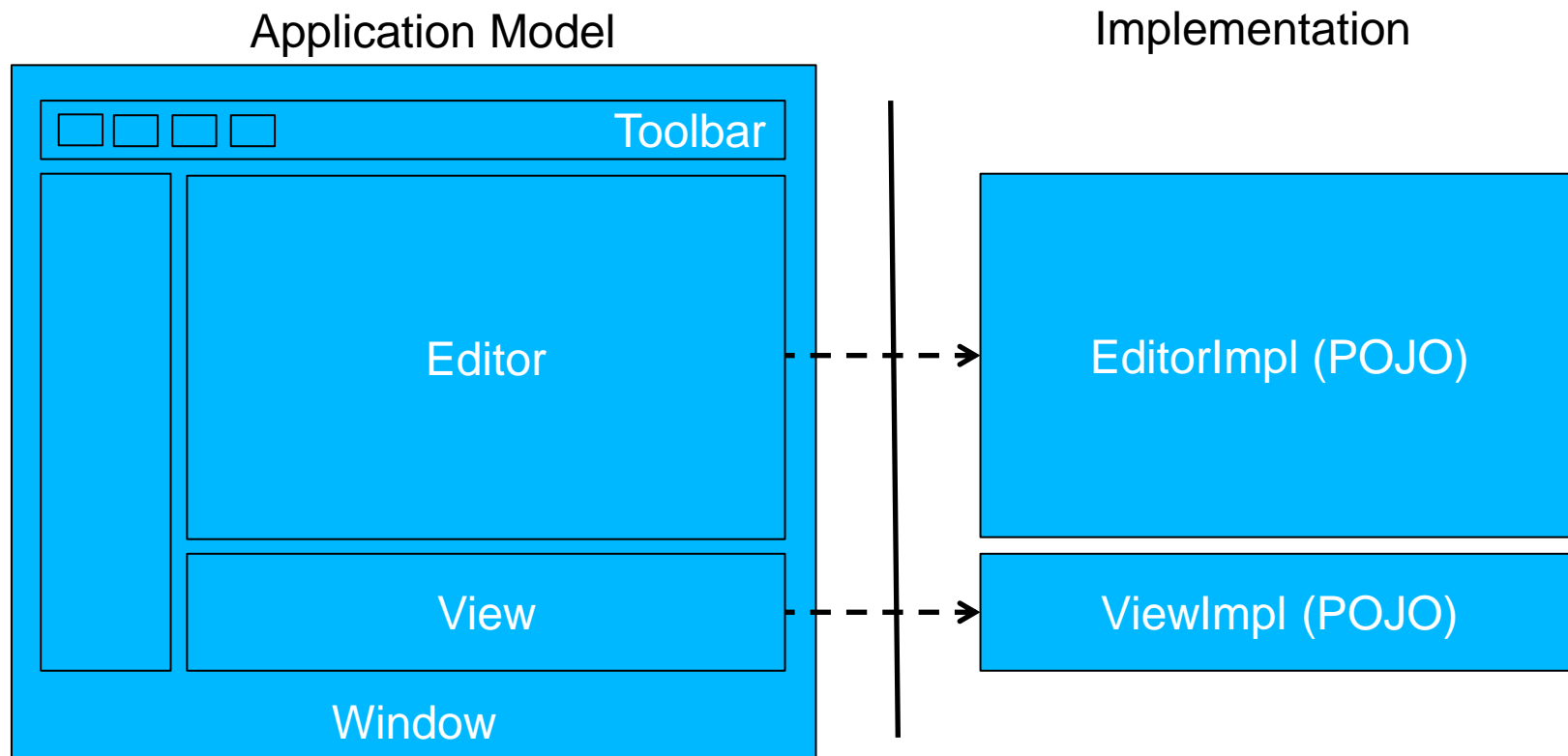
DetailsView

The 3.x Workbench



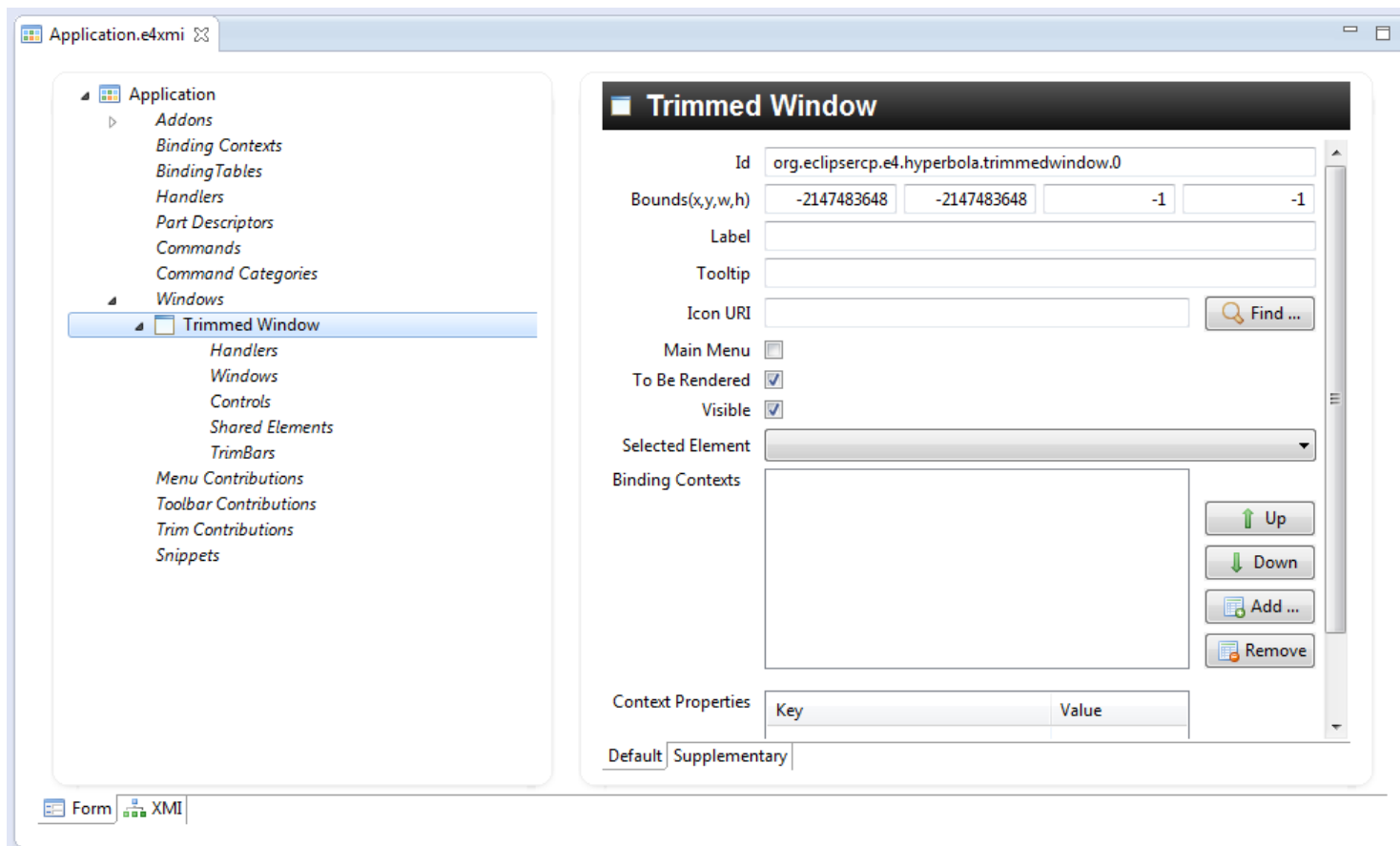
=> Strong coupling between Workbench and Implementation

The Eclipse 4 Workbench



=> Loose coupling between Workbench and Implementation

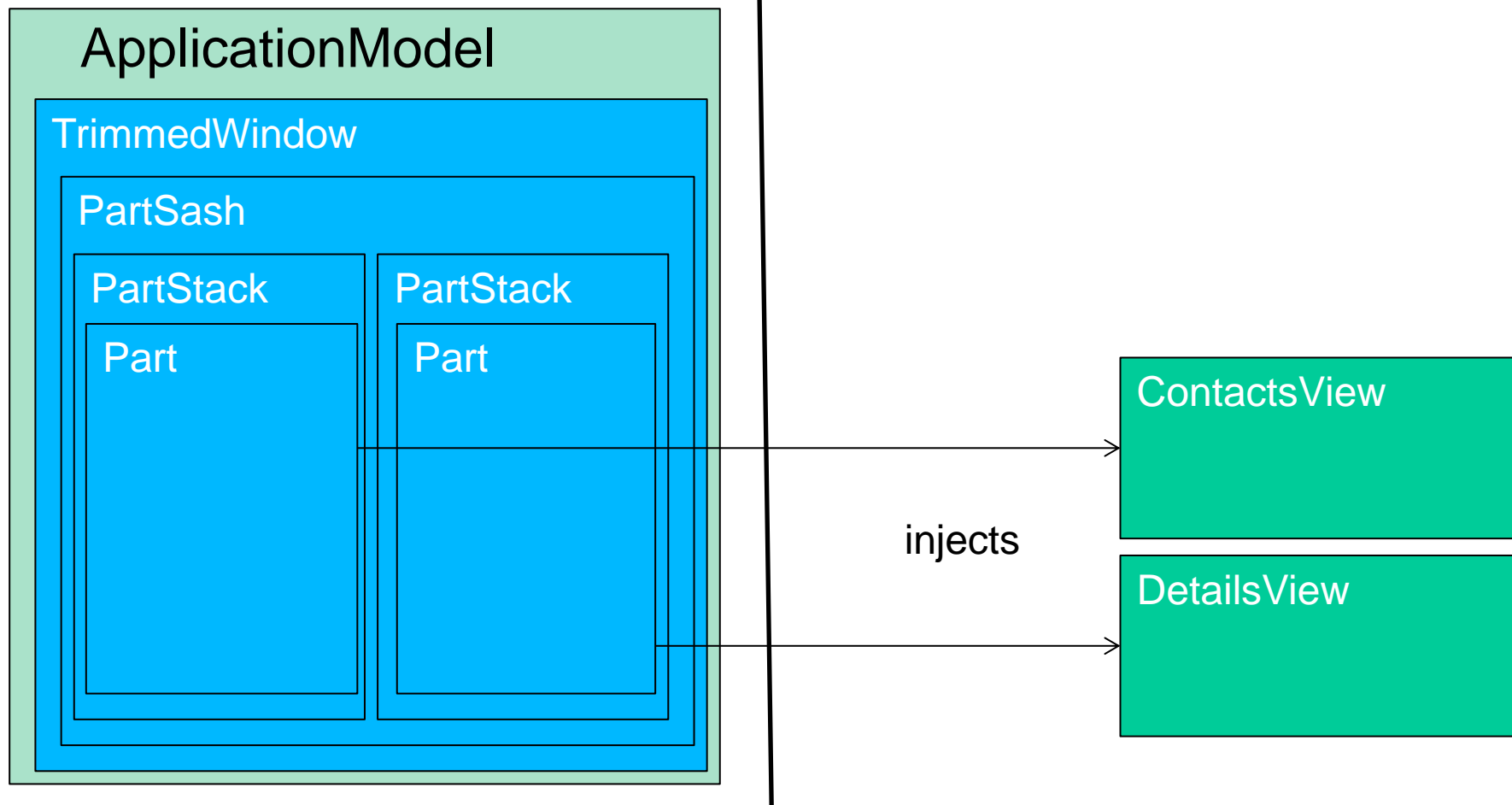
The e4 tools editor



Elements of the Application Model

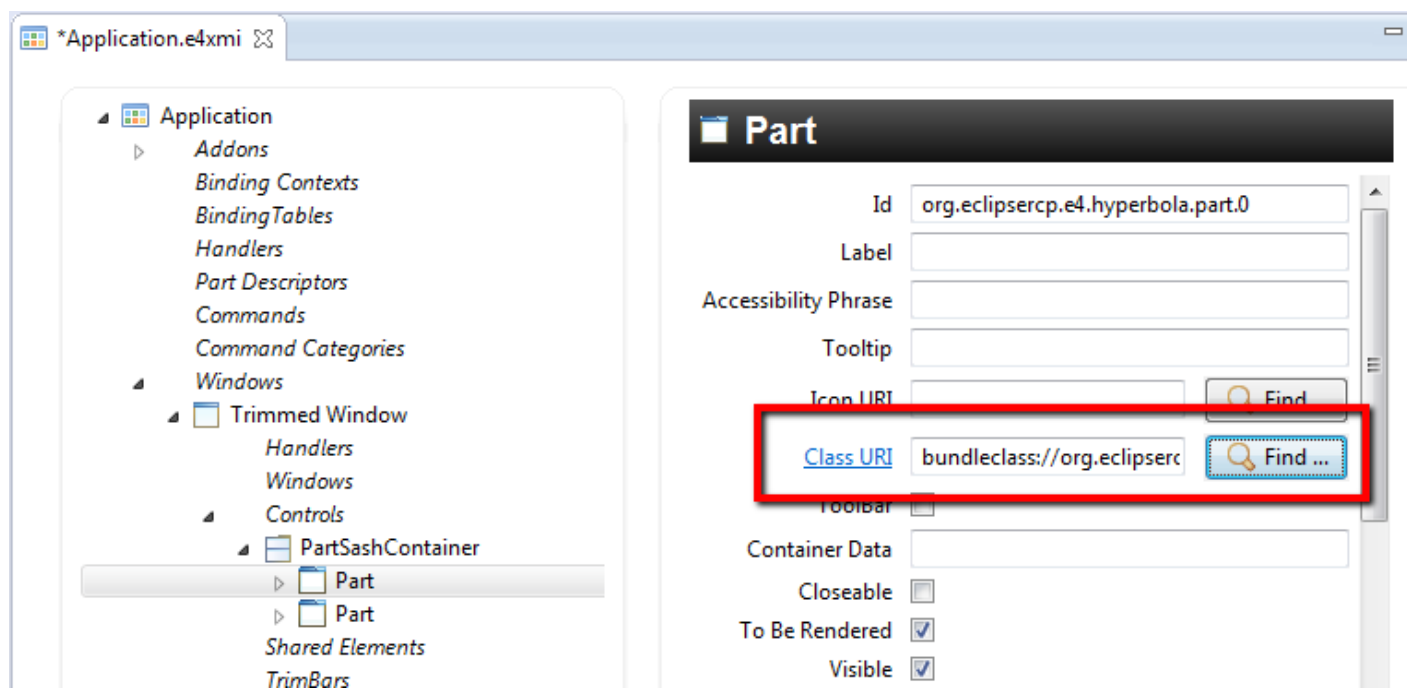
- Parts
- Windows, PartSash, PartStacks
- Handler, Commands, MenuItem, ToolBarItems
- KeyBindings
- Perspectives
- Placeholder
- PartDescriptors
- Addons
- ...

Application model

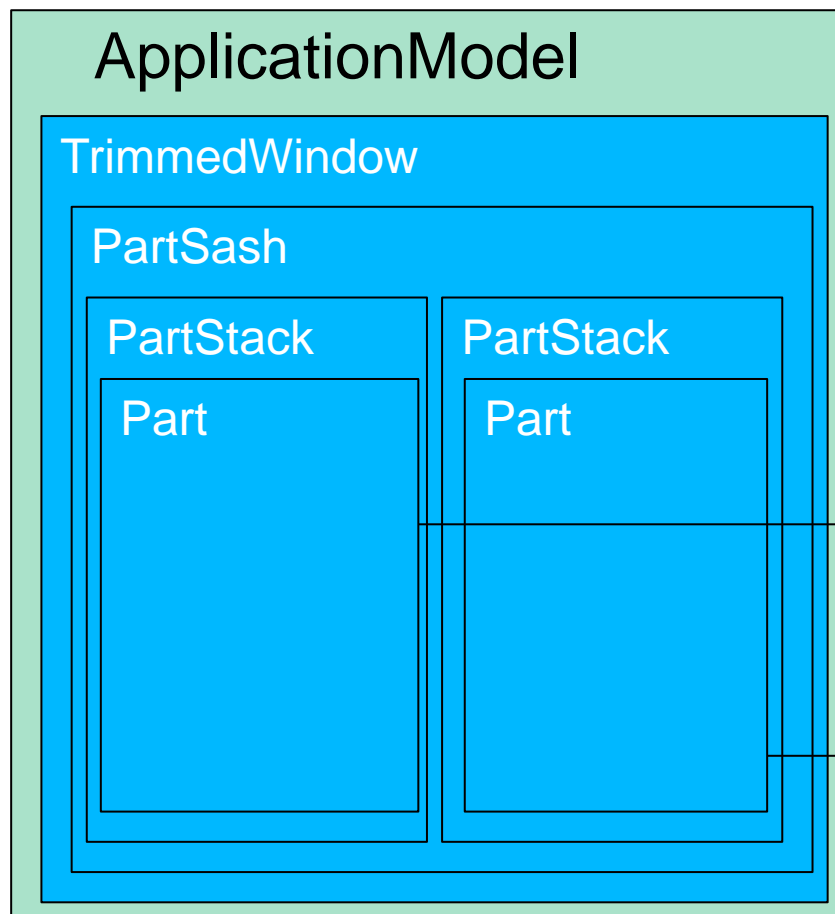


Connect views to the model

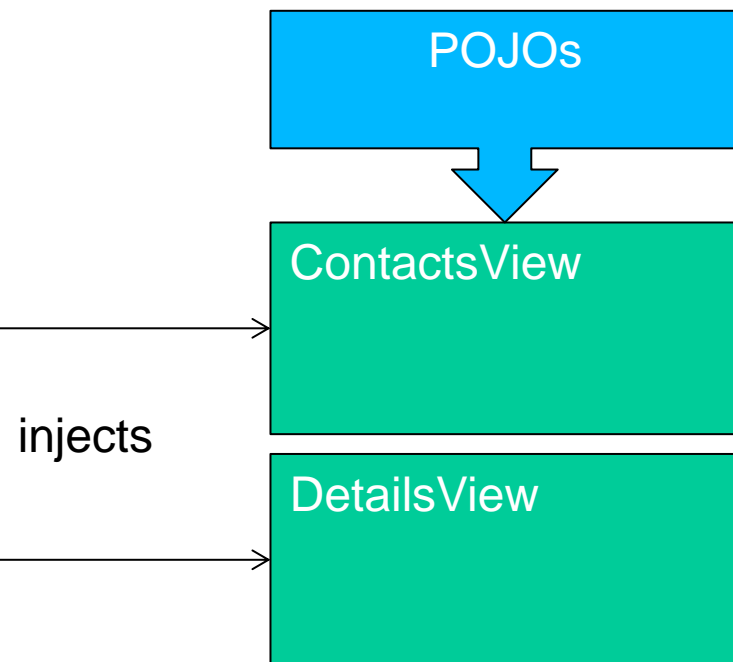
The class URI attribute binds a Part to its implementation



Application model

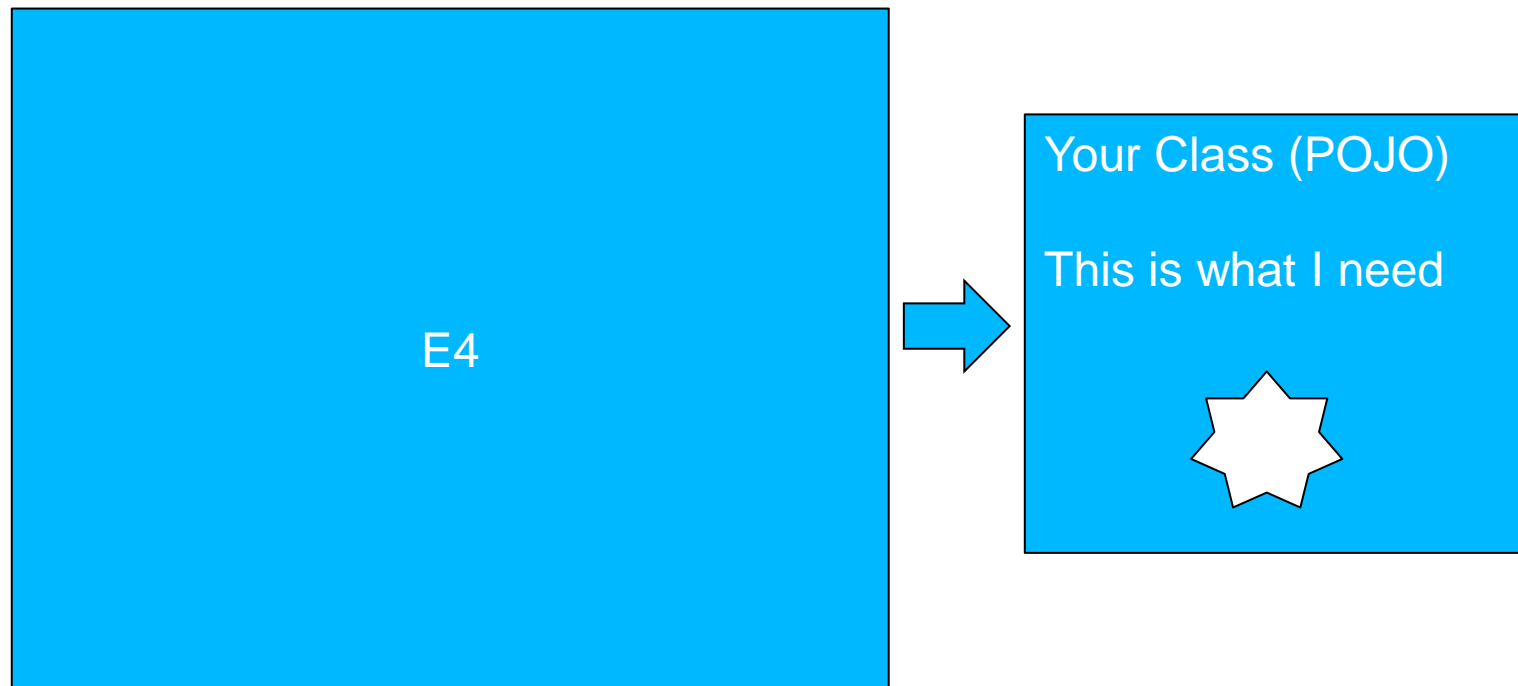


If views are POJOs, where do they get parameters from?



Dependency Injection

Hollywood principle: „Don't call us, we call you!“



What can be injected

- Elements related to the current application model element, e.g. the parent composite
- Application model elements, e.g. the main window
- Services
- Products of services, e.g. the active selection
- Preferences (@Preference)
- Events
- Own objects
- Objects marked with @Creatable

How can Objects be injected

1. Constructor:

`@Inject`

```
public void MyClass(Composite parent)
```

2. Fields:

`@Inject`

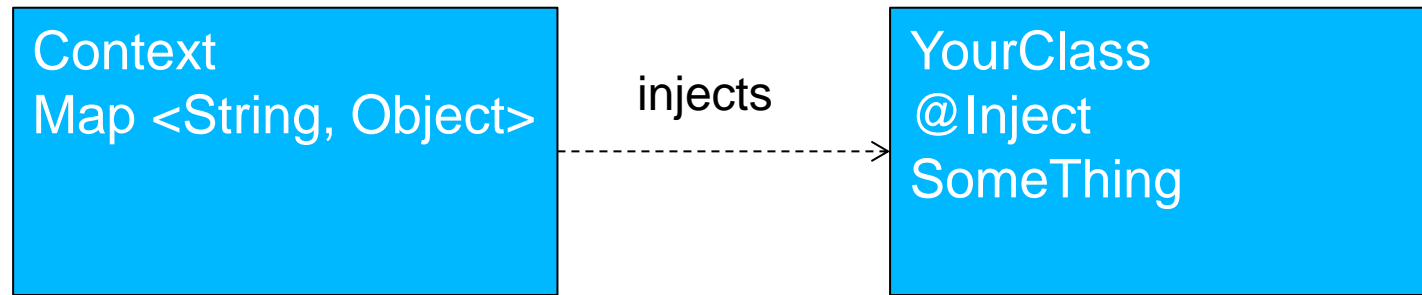
```
Service service
```

3. Methods:

`@Inject`

```
public void myMethod(SomeClass class)
```

Who injects objects?



Injects are updated if the value changes

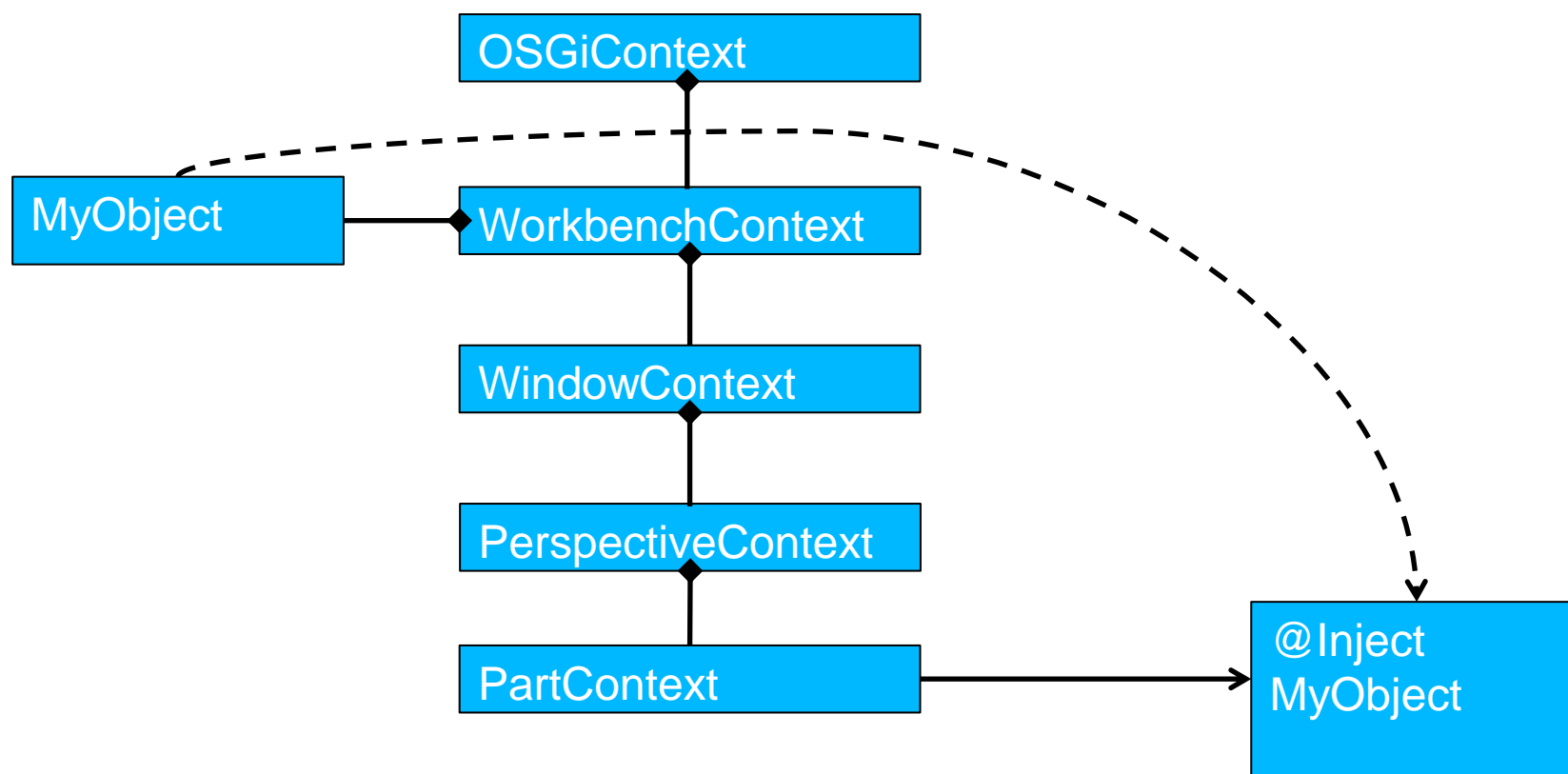
Contexts are hierarchical

`@Inject`: Tries to satisfy the requested type

`@Named`: Define a key, which is used for look-up

`@Optional`: Injects null, if not available

Hierarchy of contexts



Behaviour Annotations

- @Inject, @Named and @Optional specify, what is injected, not when
- There is often a need for a „init“ and „dispose“ method
- Custom components, such as views need to be notified, e.g. if they receive the focus

Available Behaviour Annotations

@PostConstruct: After an object is created

@PreDestroy: Before an object is destroyed

@Focus: When a UI element is focused, Views must forward the focus to a SWT widget

@Execute: Called to execute a handler

@CanExecute: Checks if a handler can be executed

All annotations include @Inject, therefore, parameters get injected

Views in Eclipse 4

- No need to implement any interface
- Start to design the view as you would do it without knowing about Eclipse
- Needed parameters are injected later on, e.g. the parent composite:
@Inject
MyView(Composite parent){
//Implement View
- Services are injected, too, e.g.:
@Inject
ContactService contactService

Annotations in Views

```
public class ContactsView {
```

```
    @Inject
    ContactService contactService;
```

```
    @Inject
    ESelectionService selectionService;
```

```
    @Inject
    public ContactsView(Composite parent) {
    }

```

```
    @PostConstruct
    public void init() {
    }

```

```
    @Focus
    public void setFocus(){
    }

```

```
    @PreDestroy
    public void dispose(){
    }

```

```
}
```

Services

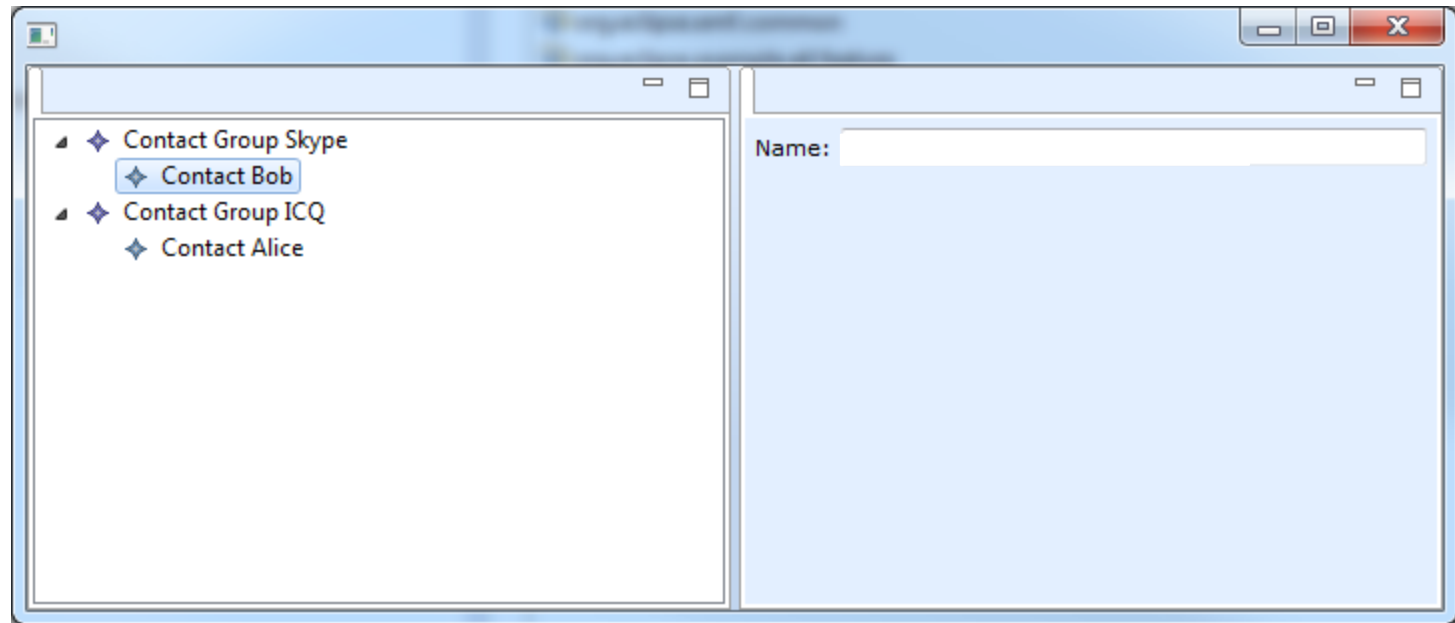
Create UI

Init UI

Set Focus

Dispose UI

Result task 1



SWT Helper in model.edit

- SWTExampleHelper.**createTreeViewer**(Composite):
creates a TreeViewer with provider
- SWTExampleHelper.**createTextWithLabel**
(Composite); to create a Label showing „Name:“ and
a SWT Text
- SWTExampleHelper.**dispose**(TreeViewer)

Task: Application Model

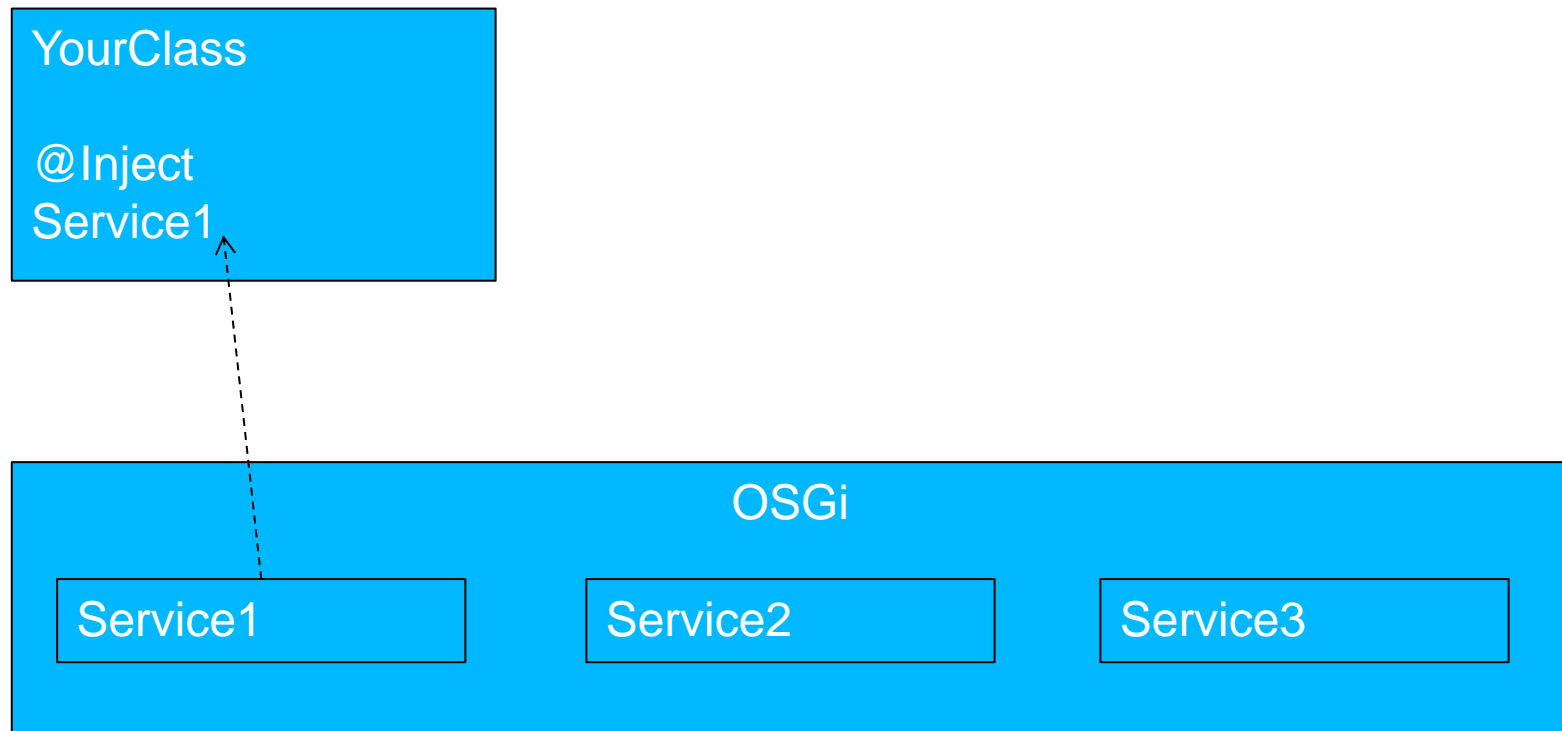
- Open the Application model (Application.e4xmi)
- Add a PartSashContainer to the window
- Add two PartStacks and Parts within them
- Implement two POJOs for the parts
 - Left: ContactsView, Right: DetailsView
 - Use the ContactService to retrieve an input for the TreeViewer (in a @PostConstruct method)
- Connect the parts to the POJOs (class URI)
- *Optional: Create different layouts*
- *Use parameter –clearPersistedState or clear the workspace*

Services

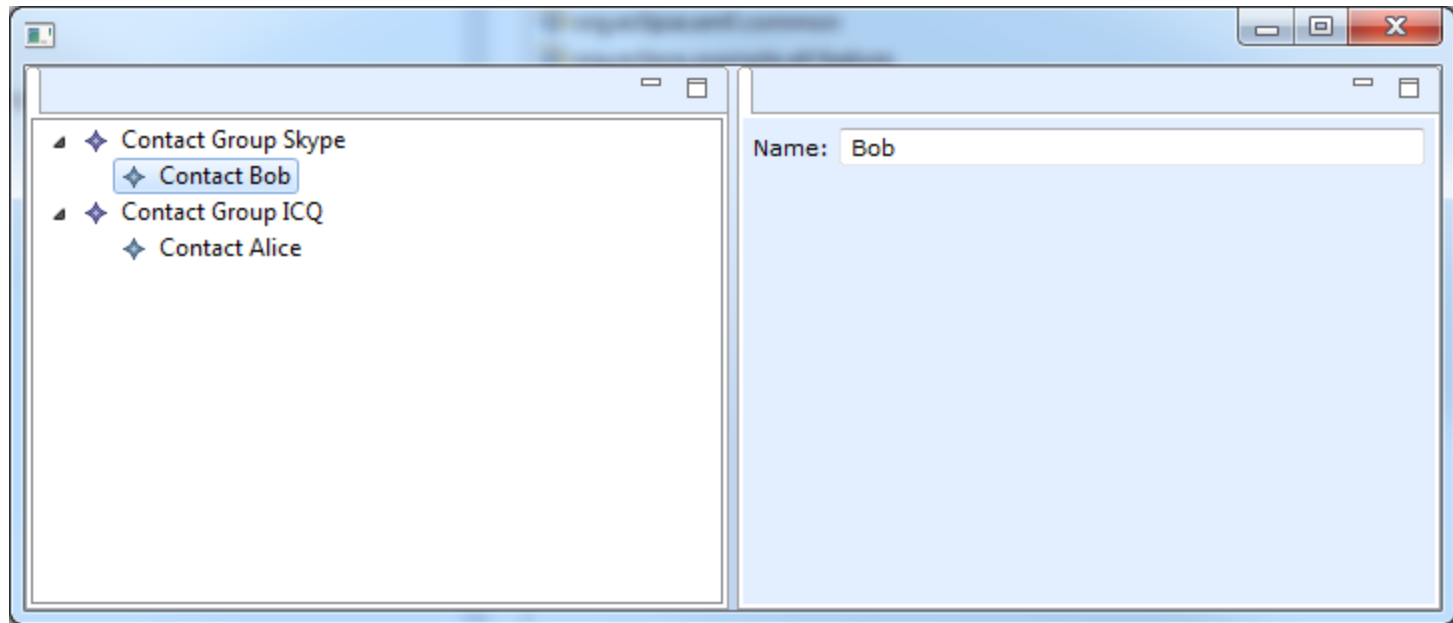


Eclipse 4 includes a lot of services, we will look at one example...

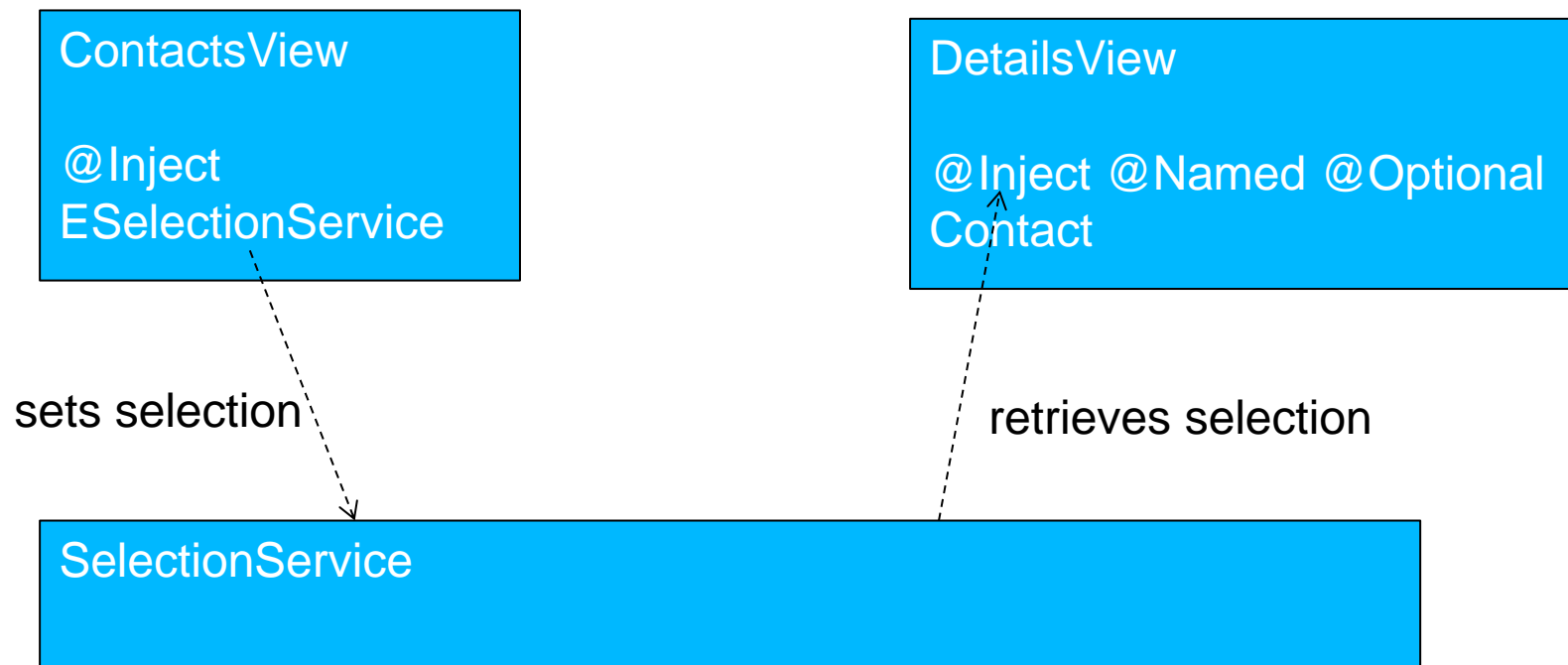
Services



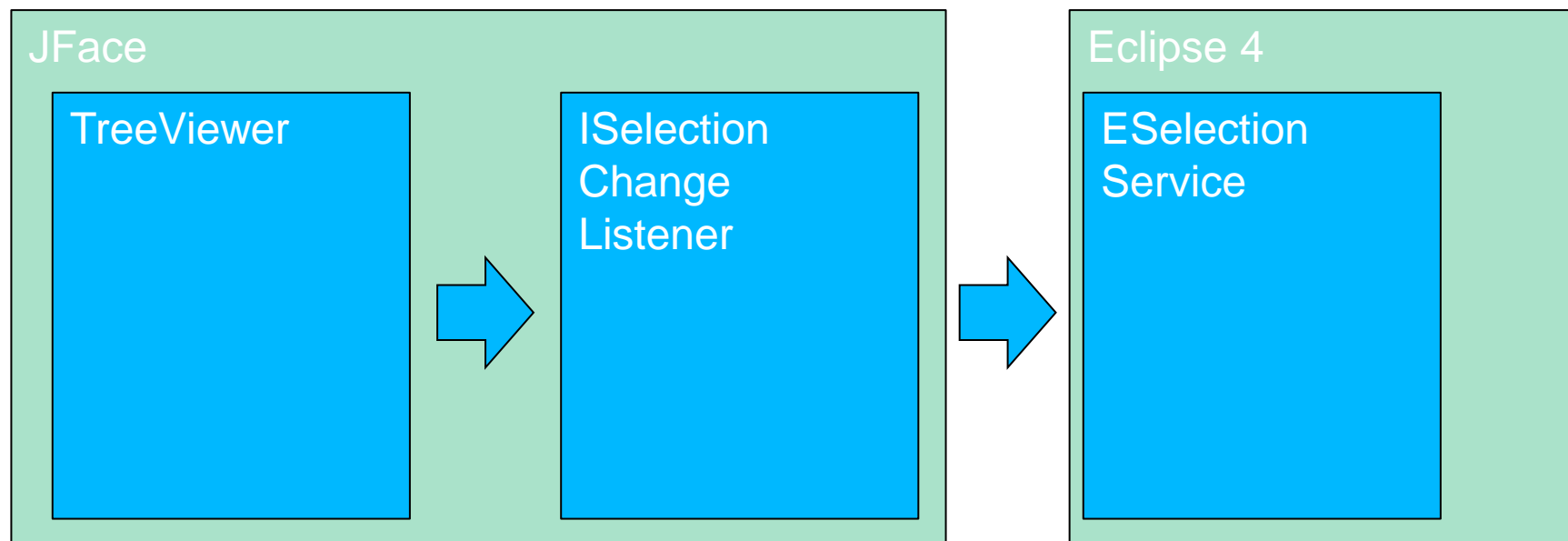
Goal task 2



The selection service



Connecting a Viewer to ESelectionService



```
private final class ISelectionChangedListenerImplementation implements
    ISelectionChangedListener {
    @Override
    public void selectionChanged(SelectionChangedEvent event) {
        Object firstElement = ((IStructuredSelection)event.getSelection()).getFirstElement();
        selectionService.setSelection(firstElement);
    }
}
```

SWTExampleHelper

SWT. connectTreeViewWithSelectionService
to connect the TreeViewer with the selection service

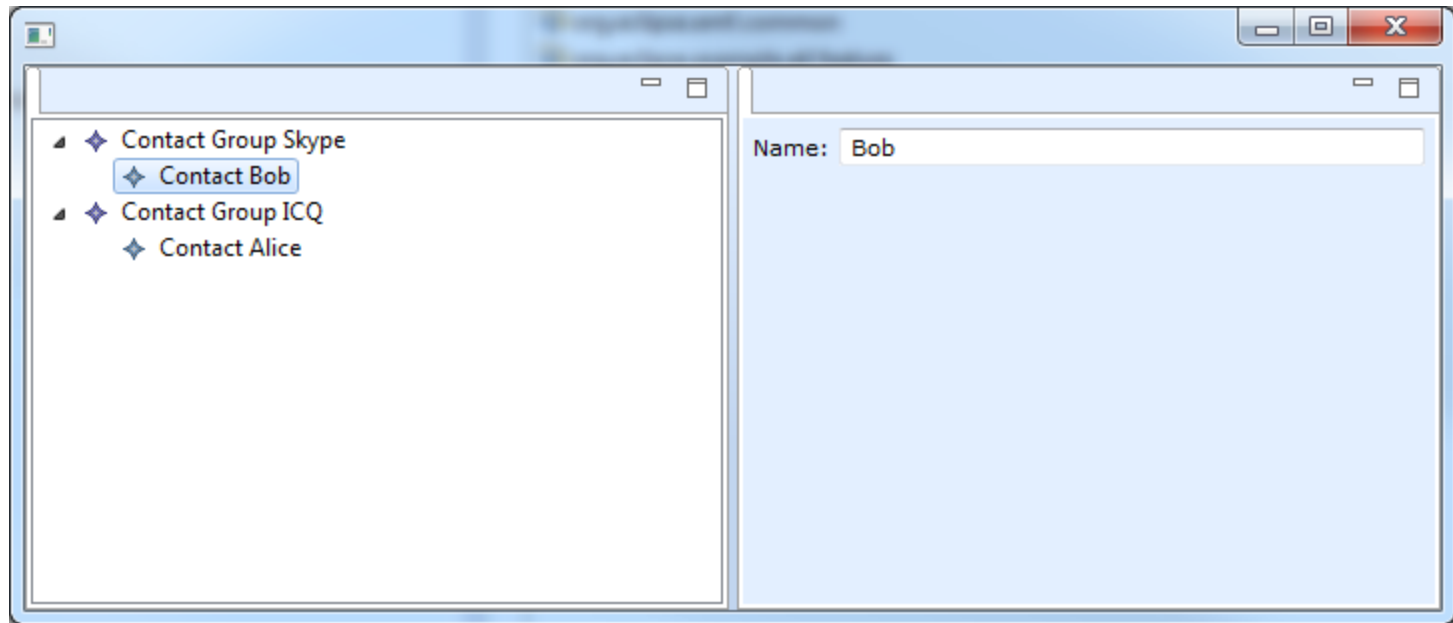
Inject the Selection

- Use @Inject to get the selection injected
- Use @Optional to accept „null“
- Use @Named to specify that the injected object is the selection
- Use the parameter type to specify the type of selection you want to retrieve

@Inject

```
public void setInput(@Optional  
@Named(IServiceConstants.ACTIVESELECTION Contact contact)
```

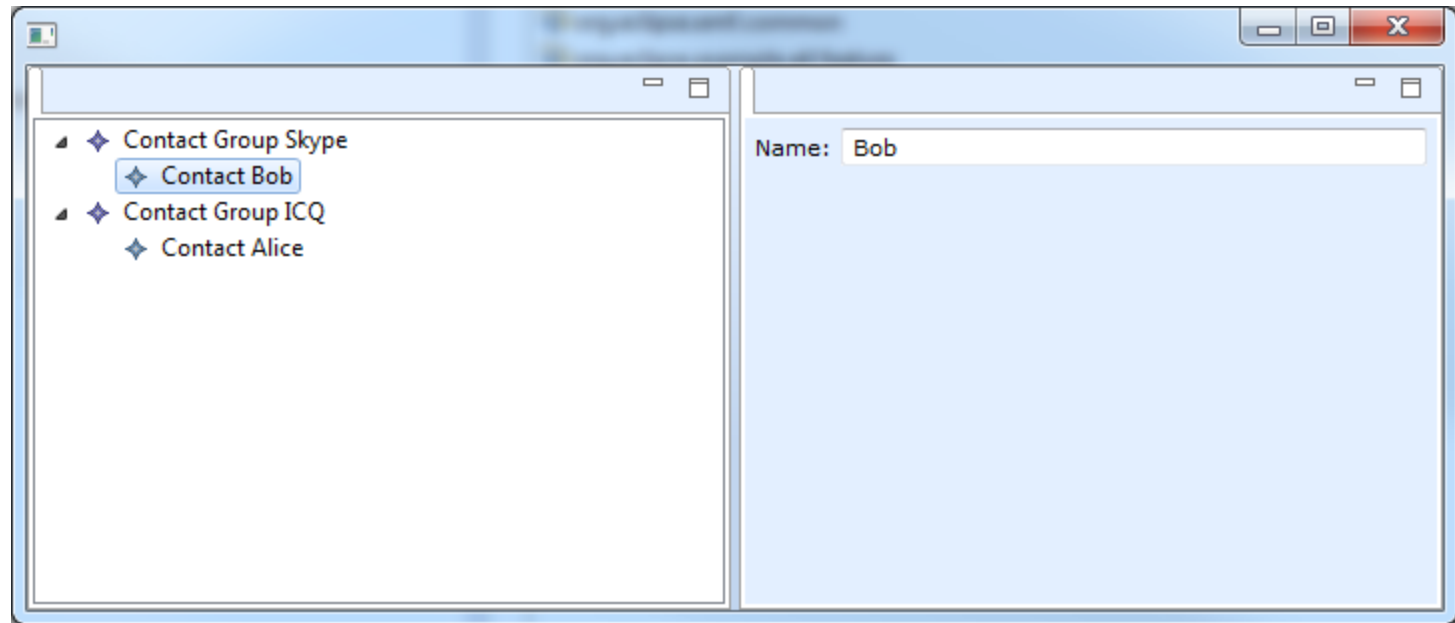
Goal task 2



Task: Selection Service

- Inject the ESelectionService in the ContactsView and connect it with the TreeViewer (you can use SWTExampleHelper.connectTreeViewer...)
- Retrieve the selection in the DetailsView:
 - Create a new method to set the selection
 - Inject the selection using @Inject, @Named and @Optional

Result task 1



```
public class ContactsView {

    @Inject
    ContactService contactService;

    @Inject
    ESelectionService selectionService;

    private TreeViewer treeViewer;

    @Inject
    public ContactsView(Composite parent) {
        treeViewer = SWTExampleHelper.createTreeViewer(parent);
    }

    @PostConstruct
    public void init() {
        treeViewer.setInput(contactService.getInput());
        SWTExampleHelper.connectTreeViewerWithSelectionService(treeViewer, selectionService);
    }

    @Focus
    public void setFocus(){
        treeViewer.getTree().setFocus();
    }

    @PreDestroy
    public void dispose(){
        SWTExampleHelper.dispose(treeViewer);
    }
}
```

```
public class DetailsView {  
  
    private Text text;  
  
    @Inject  
    public DetailsView(Composite parent) {  
        text = SWTExampleHelper.createTextWithLabel(parent);  
    }  
  
    @Inject  
    public void setInput(  
        @Optional @Named(IServiceConstants.ACTIVE_SELECTION) ContactEntry contactEntry) {  
        if(contactEntry==null){  
            text.setText("");  
            return;  
        }  
        text.setText(contactEntry.getName());  
    }  
}
```

Outline

- The Application Model
- Implementing Views
- Selection Service

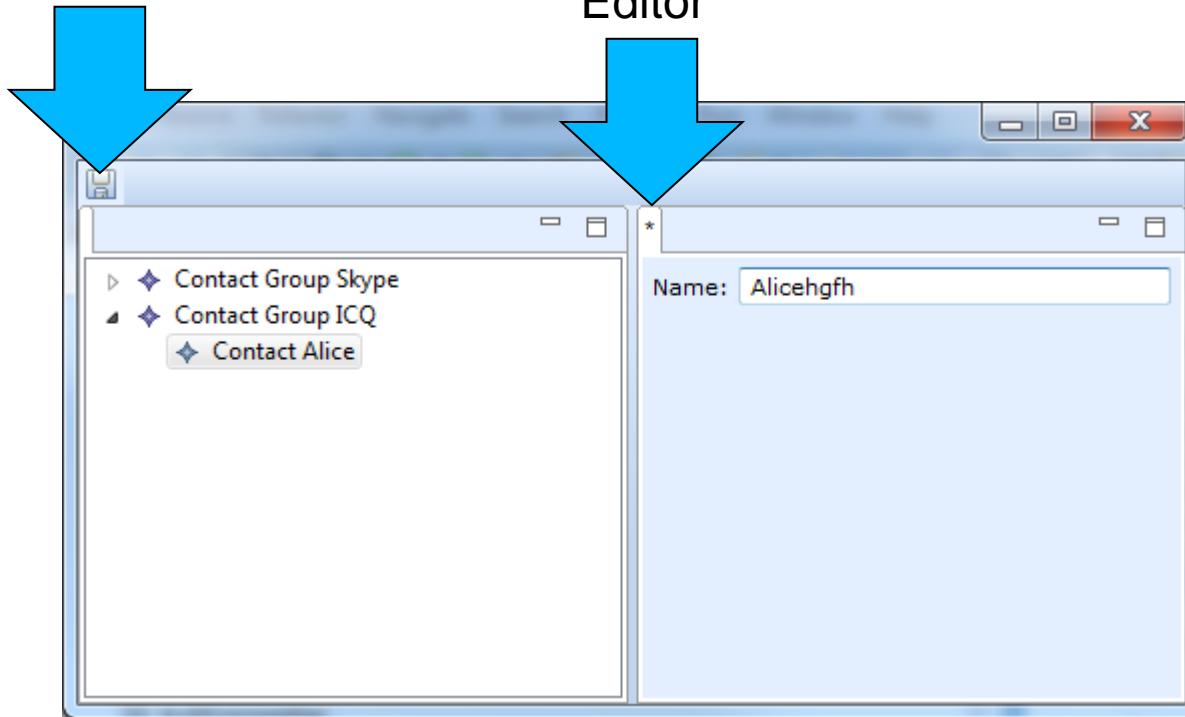
- Handlers, Commands and Items
- Editors

- Dependency Injection Details
- Modularity
- Migration

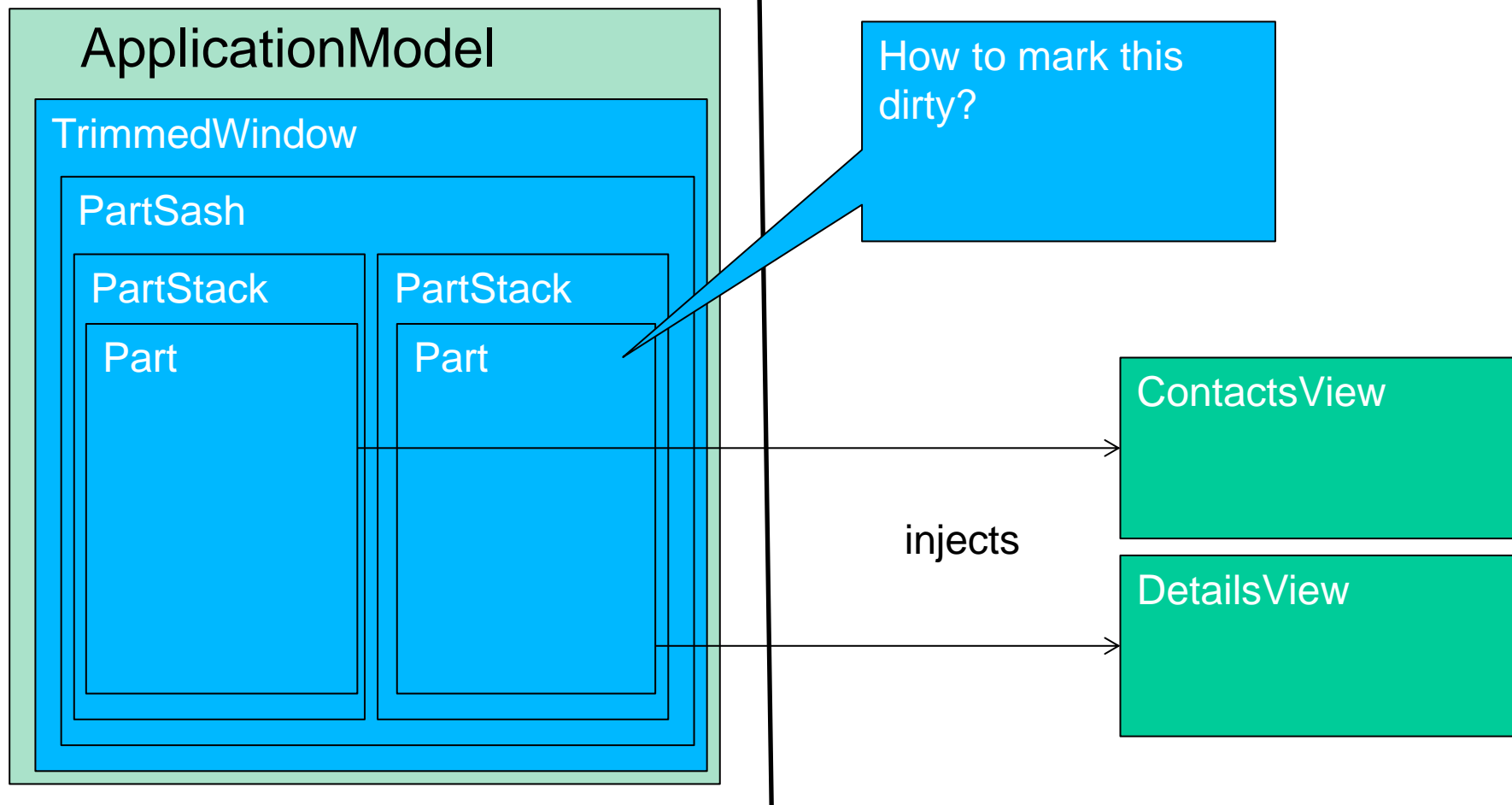
Goal 2

Save Button

Editor



Application model



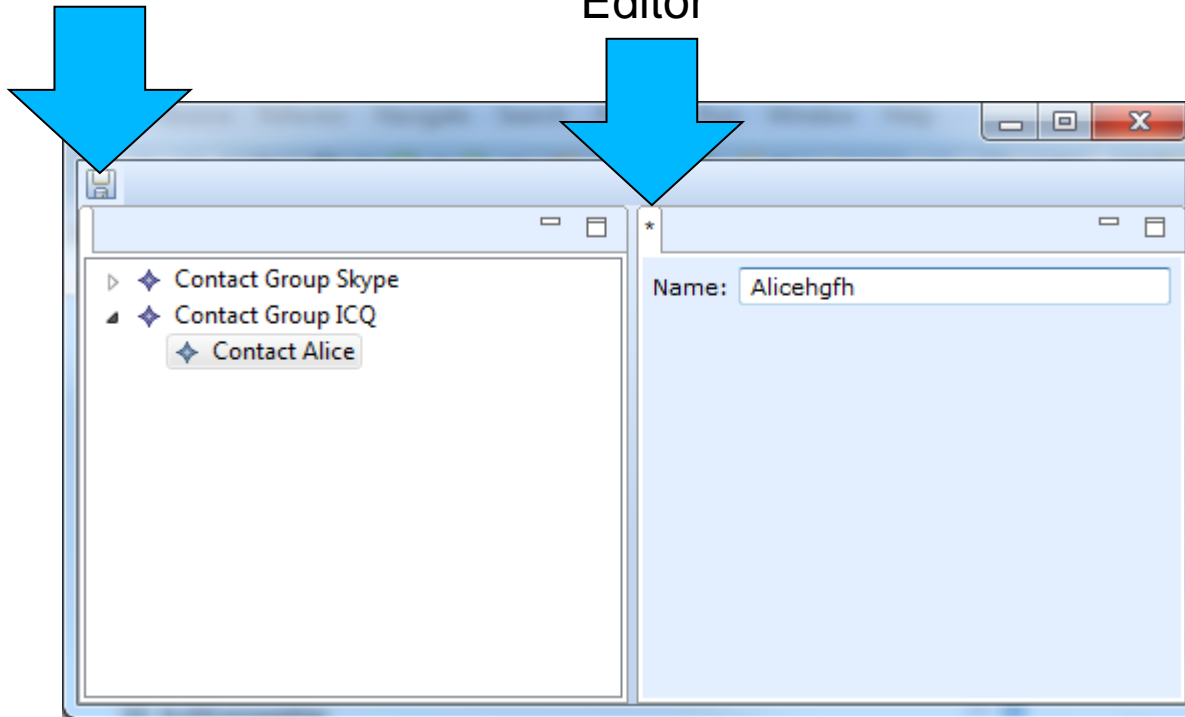
Editors in Eclipse 4

- All Parts are of type MDirtyable, get access through:
@Inject
MDirtyable dirtyable;
dirtyable.setDirty(true/false);
- Mark the save method with @Persist
 - In our case in the DetailsView
- You need to set dirty to false, once save is completed

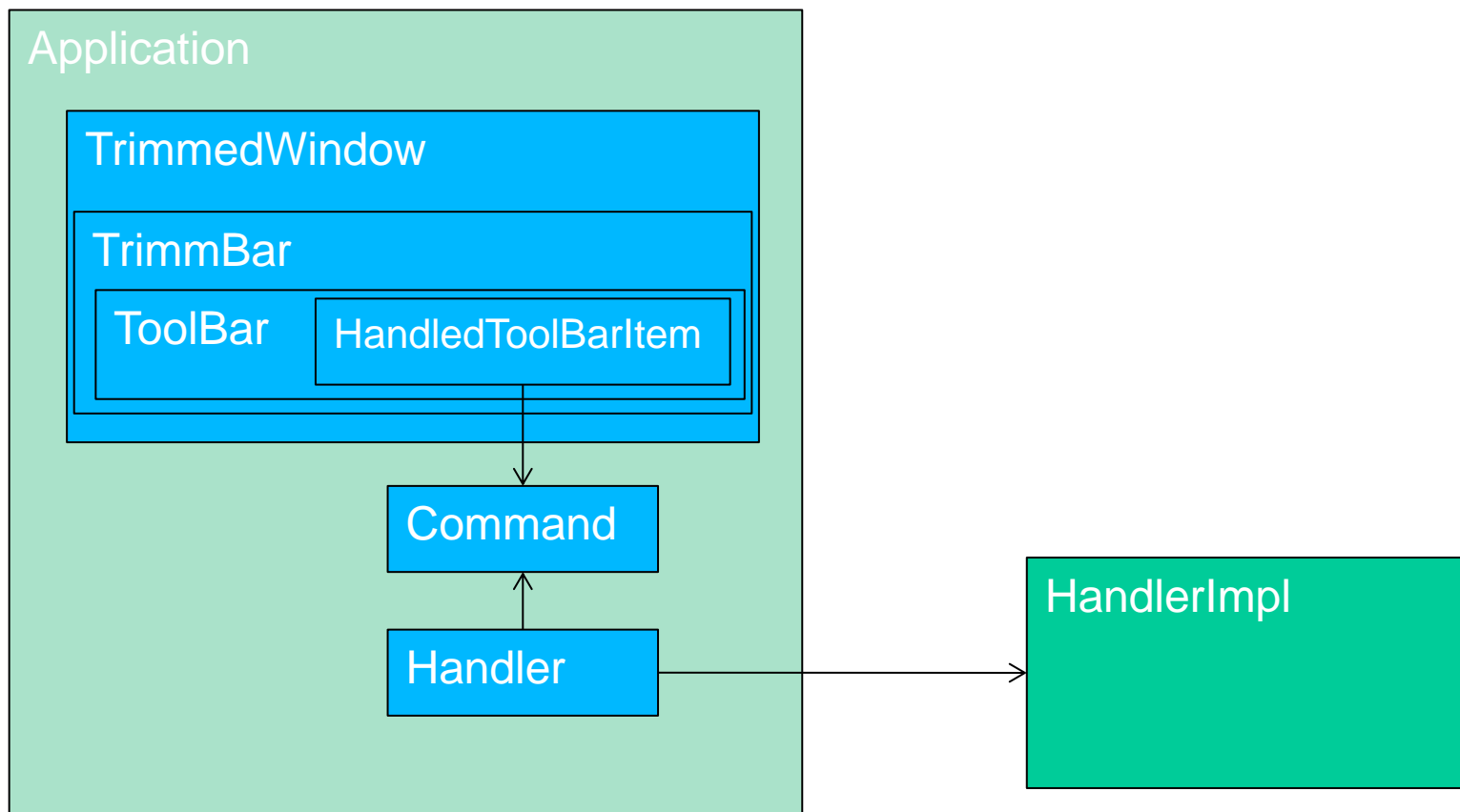
Goal 2

Save Button

Editor



In e4 handlers, commands and items are part of the workbench model



Handler

- Implementations are POJOs
- They can easily be tested
- Mark the method to be executed with `@Execute`
- Mark the method which is responsible for the enabled state with `@CanExecute`
 - Needs to return a boolean
- `@Execute` and `@CanExecute` include an `@Inject`, so parameters can be injected, e.g.:
 - Current selection
 - Active part

Save Button in e4

- Active part can be injected to check if it is dirty:

@CanExecute

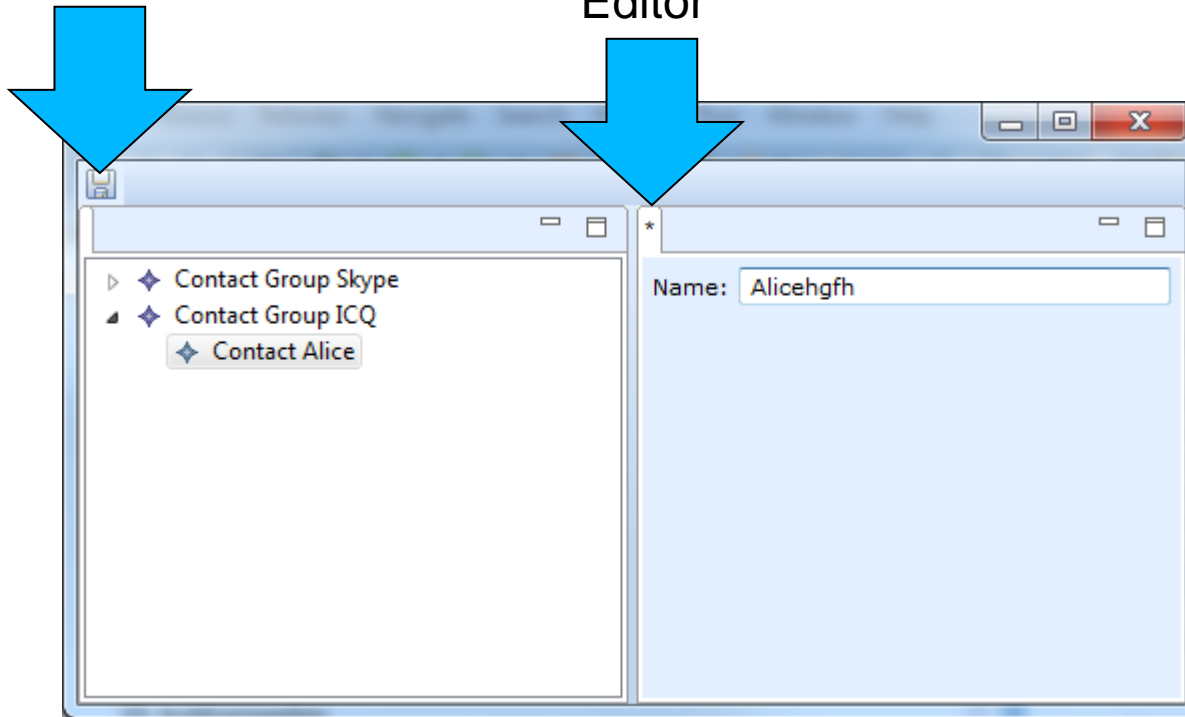
```
public boolean canExecute(@Named(IServiceConstants.ACTIVE_PART)
    @Optional MPart part) {
    //Check for null
    return part.isDirty();
}
```

- Save can be triggered using the EPartService:
partService.savePart(part, false);
 - Alternative: partService.saveAll()

Goal 2

Save Button

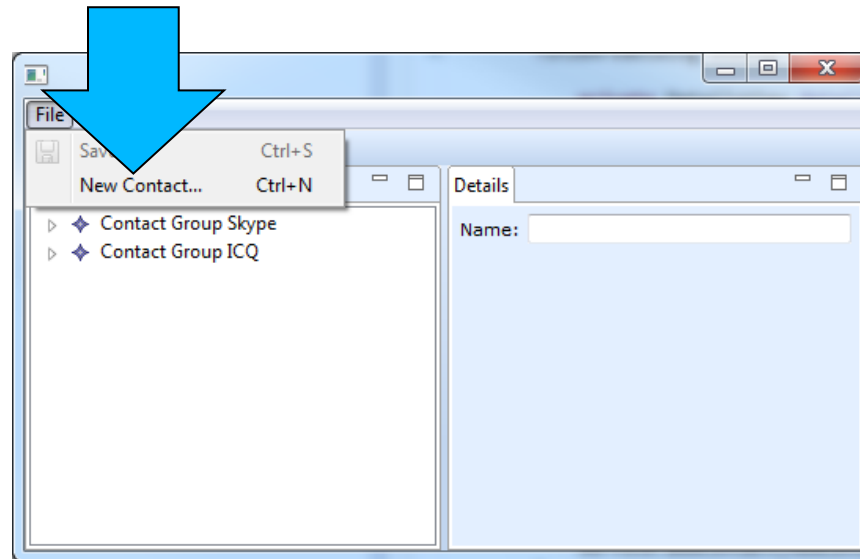
Editor



Task: Implement an Editor

- Let the EditorView handling it's dirty state:
 - Set dirty if (Text!=input)
 - Set not dirty in the save method
- Implement a save method to write changes to the contact. Mark it with @Persist
- Implement a save tool item, saving the active part
- Add a Command, a Handler and an Implementation
- The save tool item should only be enabled, if the active part is dirty

Optional Task



- Only enabled if a ContactGroup is selected
- Reuse the DetailsView in a dialog

```
contact= ModelFactory.eINSTANCE.createContact()  
contactGroup.getContacts.add(contact)
```

DetailsView

```
@Inject
MDirtyable dirtyable;

@Persist
public void save() {
    input.setName(text.getText());
    dirtyable.setDirty(false);
}

@Inject
public void setInput(
    @Optional @Named(IServiceConstants.ACTIVE_SELECTION) ContactEntry contactEntry) {
    if (contactEntry == null) {
        text.setText("");
        input = null;
    } else {
        text.setText(contactEntry.getName());
        input = contactEntry;
    }
    dirtyable.setDirty(false);
}

@PostConstruct
public void init() {
    text.addModifyListener(new ModifyListener() {

        @Override
        public void modifyText(ModifyEvent e) {
            dirtyable.setDirty(true);
        }
    });
}
```

```
public class SaveHandler {  
  
    @Execute  
    public void execute(EPartService partService){  
        partService.saveAll(false);  
    }  
  
    @CanExecute  
    public boolean canExecute(@Named(IServiceConstants.ACTIVE_PART) @Optional MPart part){  
        if(part==null){  
            return false;  
        }  
        return part.isDirty();  
    }  
}
```

Outline

- The Application Model
- Implementing Views
- Selection Service
- Handlers, Commands and Items
- Editors
- Dependency Injection Details
- Modularity
- Migration

Extending Contexts

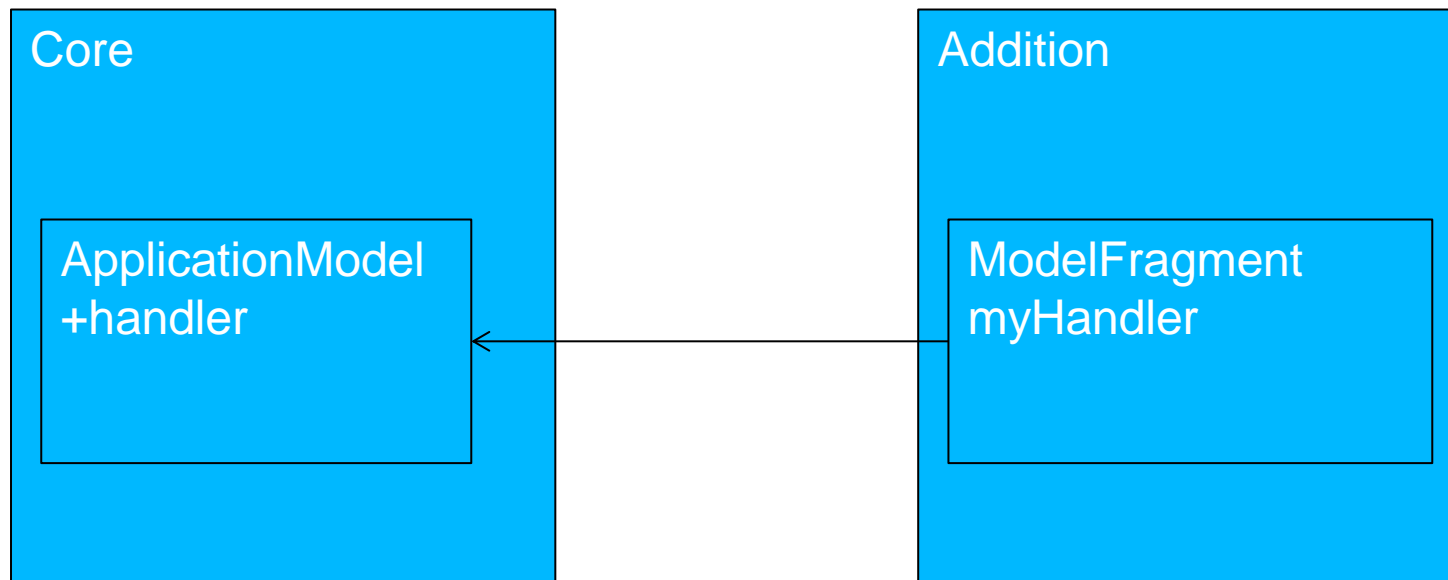
- IEclipseContext provides methods to browse and modify the Context, e.g.
`context.set(MyObject.class, new MyObject());`
- IEclipseContext.modify() walks up in the hierarchy and tries to replace an existing element
- Used to enhance low level contexts, e.g. to pass an input to an editor
- Used to create contexts for testing

Triggering Dependency Injection

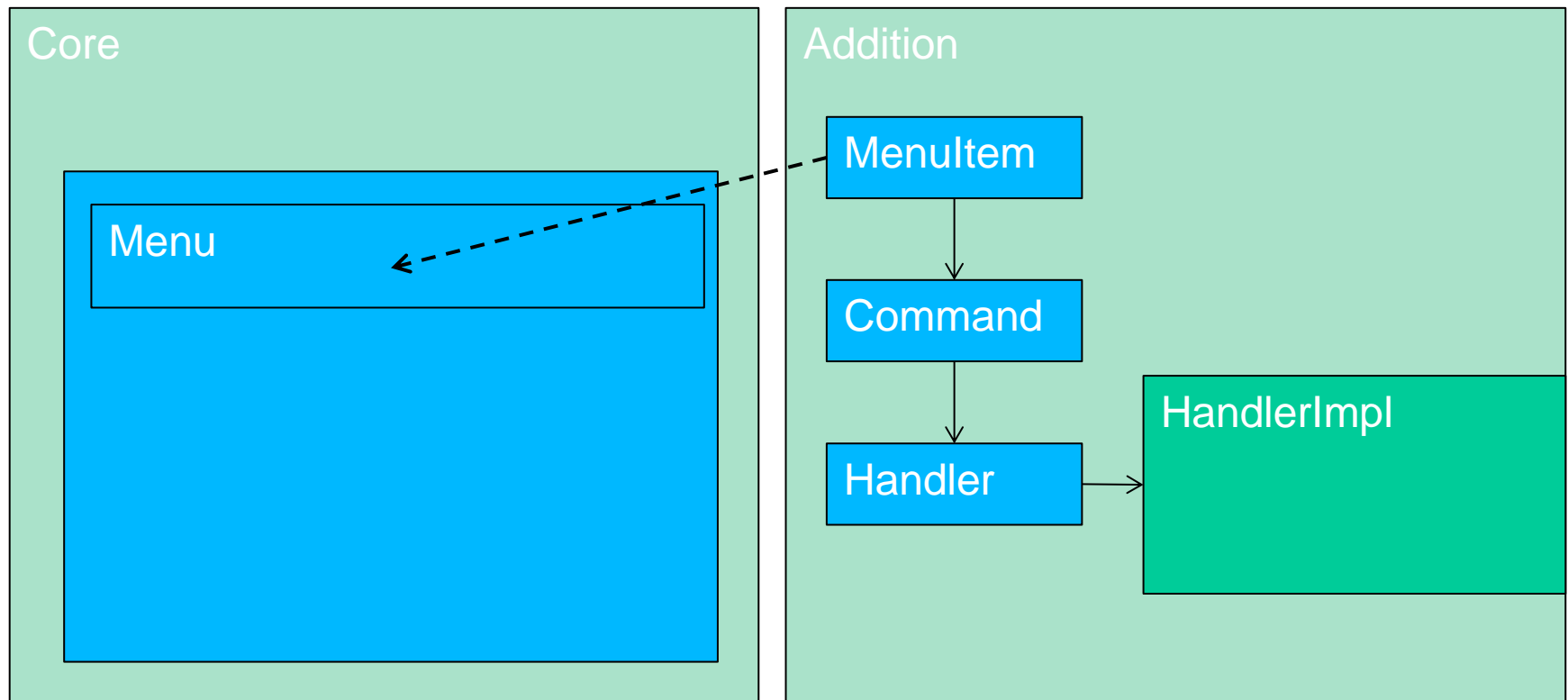
- Injection can be manually triggered:
`ContextInjectionFactory.make(DetailsView.class, context);`
- Methods with annotation can be called:
`ContextInjectionFactory.invoke(detailsView, Persist.class, context);`

Add contributions from other plugins

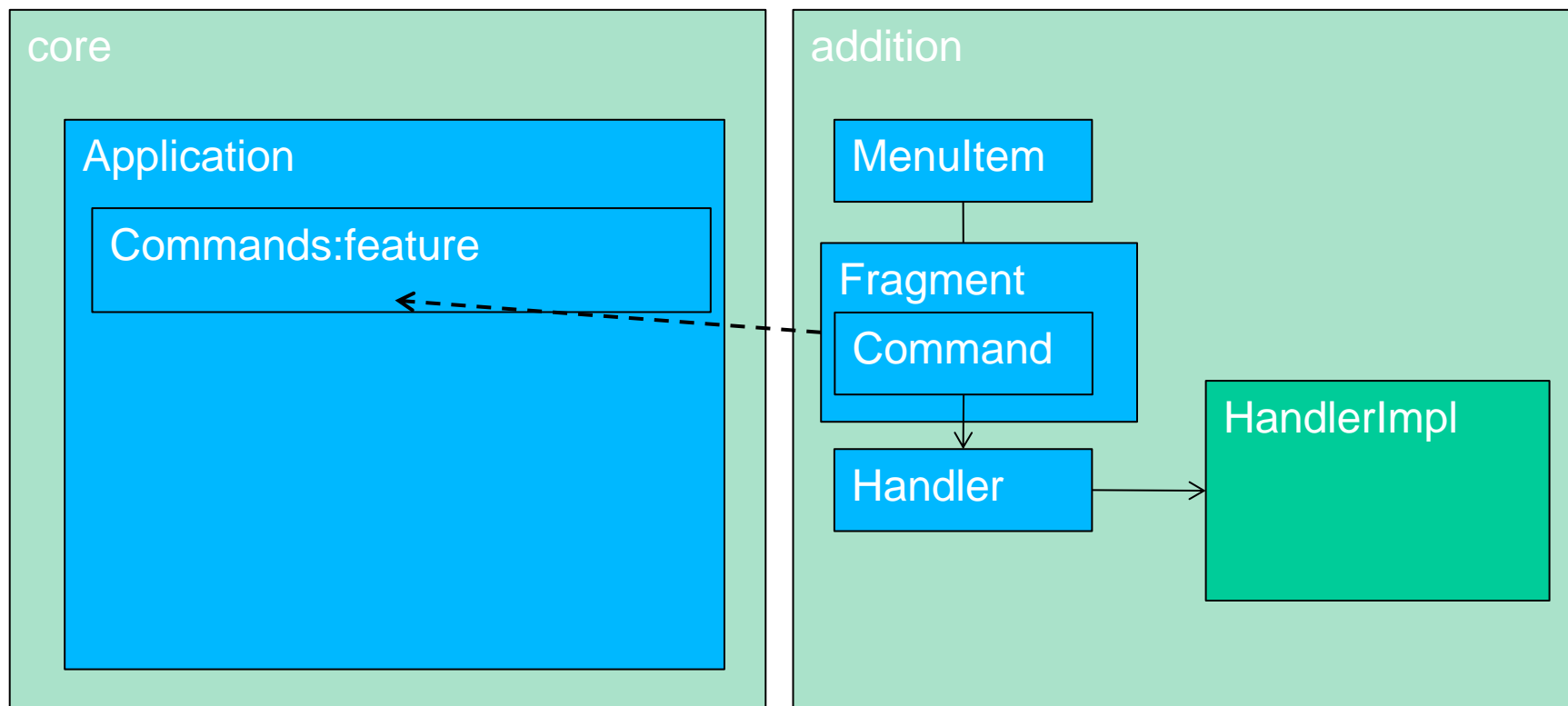
- Contributions can be added from plugins, e.g. to deploy an optional feature



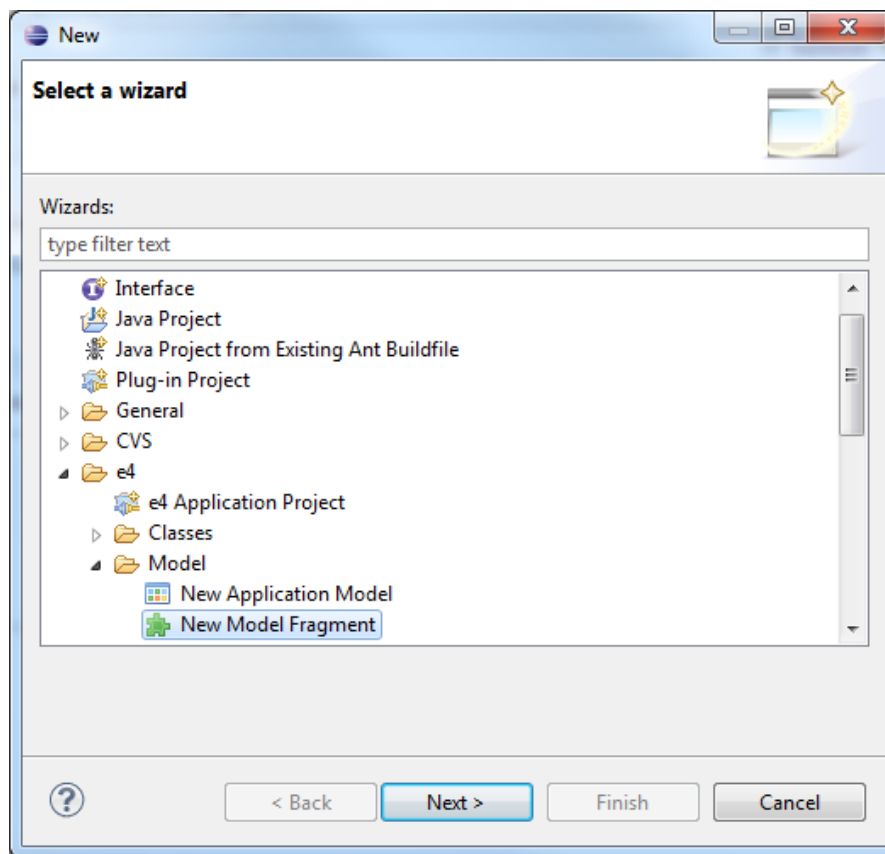
Extend the Application Model



Extend the Application Model

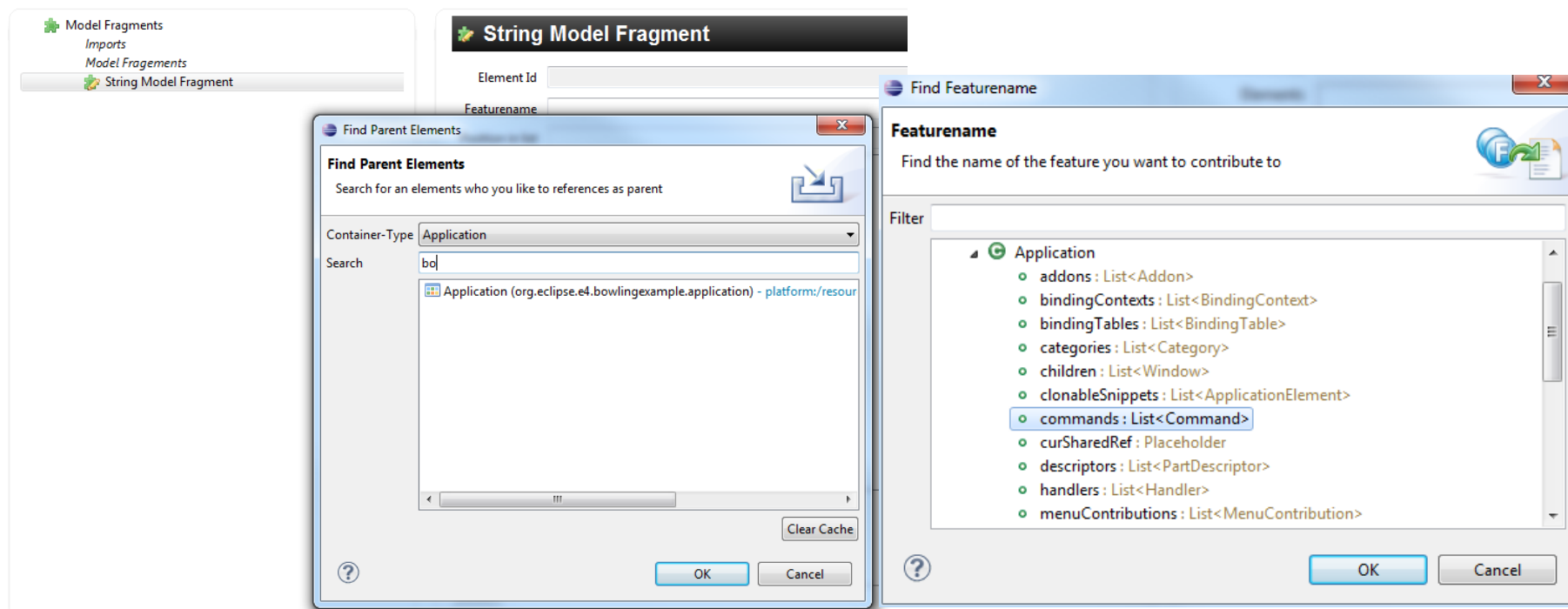


Fragments are files in the workspace

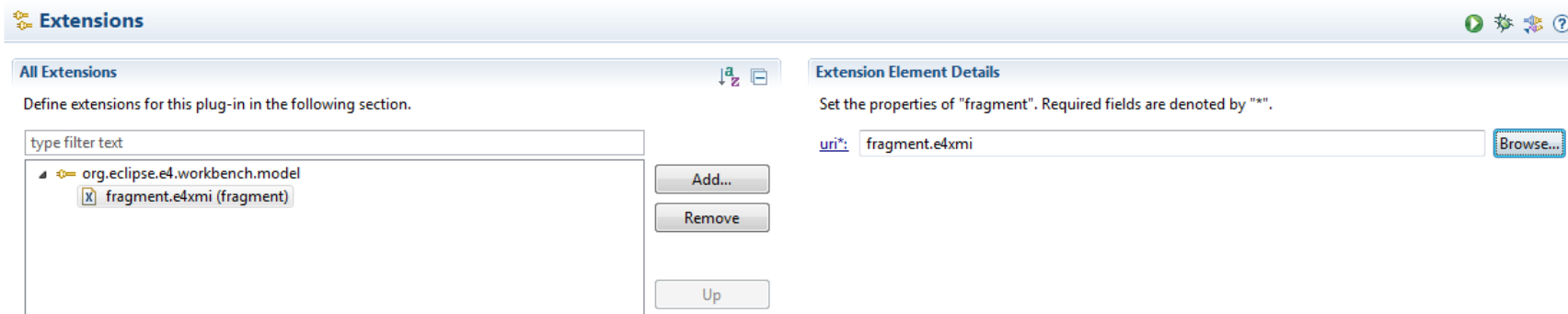


Extend the application model

Extended elements (container) are referenced by ID and EMF features



Fragments need to be registered

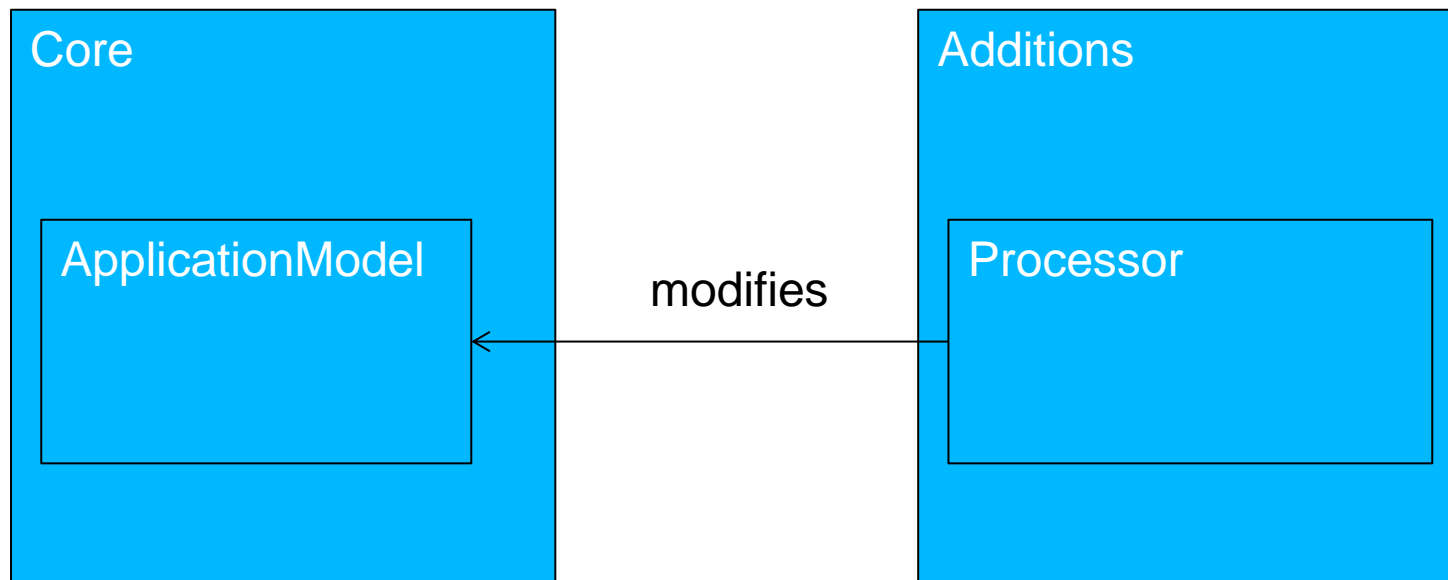


Additionally, the
Application Model
needs to be registered

```
<extension
  id="product"
  point="org.eclipse.core.runtime.products">
  <product
    application="org.eclipse.e4.ui.workbench.swt.E4Application"
    name="Hyperbola">
    <property
      name="appName"
      value="Hyperbola">
    </property>
    <property
      name="applicationXMI"
      value="org.eclipse.rcp.e4.hyperbola/Application.e4xmi">
    </property>
  </product>
</extension>
```

Programmatic model enhancements (Optional)

- Processors modify the Application Model using the EMF API



Processors

- Processors need to be registered (like fragments)
- Mark the method to be execute with `@Execute`
- Inject the elements to be extended or modified, e.g.: `MApplication` or `EModelService`
- Only root context is available
- Use `EModelService.findElement()` to retrieve Elements

Programmatically modify the application model

- Model classes are prefixed with a „M“
- They provide getter and setter methods for simple attributes as well as lists for references
- New Elements are created with a factory

@Execute

```
public void execute(MWindow window){  
    window.setHeight(200);  
}
```

@Execute

```
public void execute(MApplication application){  
    application.getChildren().add(MBasicFactory.INSTANCE.createWindow())  
}
```

Migration from 3.x?

- E4 offers a compatibility layer
- E4 only offers benefits, if you use the new concepts
- Many things like styling or dependency injection can be used in 3.x, too!

Option1: Pure e4 Application

- Benefits:
 - Use all concepts such as dependency injection, modeled workbench, CSS
 - Clean design
- Disadvantages:
 - Existing UI componentes need to be migrated
 - External UI componentes might not work at all!

Option 2: Pure Compatibility Layer

- Benefits:
 - Clean design
 - Reuse existing UI components
 - Reuse external UI components
- Disadvantages:
 - Concepts such as dependency injection, modeled workbench are not available

Option 3: Mixing e4 and 3.x

- Benefits:
 - Use all concepts such as dependency injection, modeled workbench for new components
 - Reuse existing UI components
 - Reuse external UI components
- Disadvantages:
 - Mix of technologies

Ways of mixing e4 with 3.x

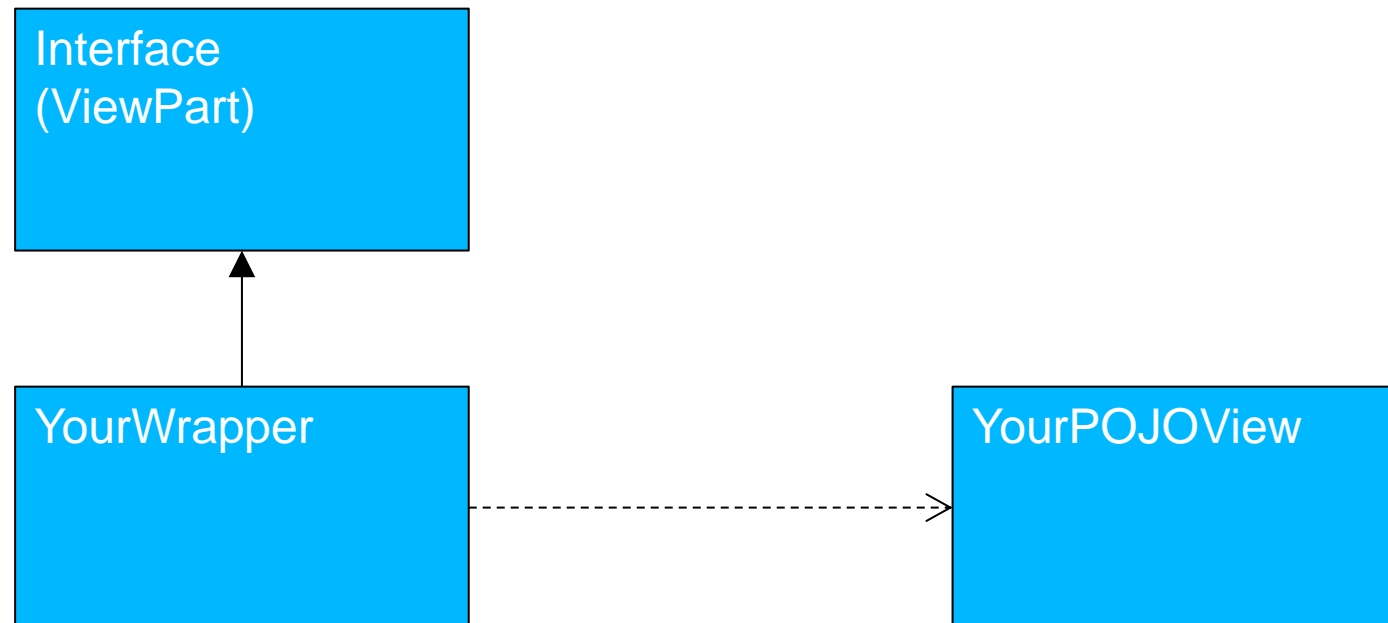
The compatibility layer mocks `org.eclipse.ui` and translates all calls and extensions to an Application Model in the background

Option 1: Add contributions to this model. Not explicitly supported yet

Option 2. Register contributions as in 3.x but use a wrapper for an e4 POJO

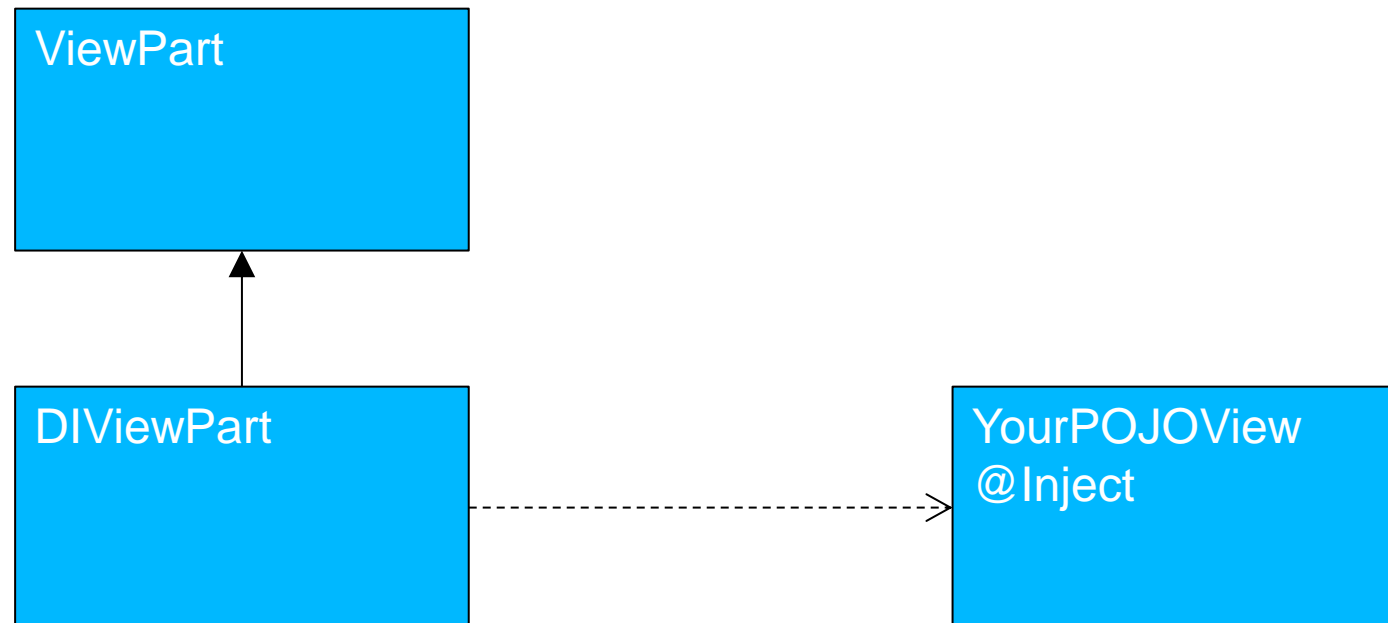
Single Sourcing

Keep your application independant from technologies



Single Sourcing

There are wrapper implementations supporting dependency injection (e4Tools Project)



More Information

- E4 Wiki
- E4 Newsgroup
- Blogs
- Books
- eclipsesource.com/eclipse4tutorial
- Professional Training => eclipsesource.com
- Jonas.Helming@eclipsesource.com