Eclipse 4.2 – Introduction

Jonas Helming

EclipseSource Munich

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

Who is Jonas Helming?

- Software Engineer / Trainer / General Manager at EclipseSource Munich
- Committer at e4, EMFStore, EMF Client Platform, EDAPT
- Author for Eclipse Magazines and Eclipse 4 Book

- Blog: http://eclipsesource.com/blogs/author/jhelming/

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

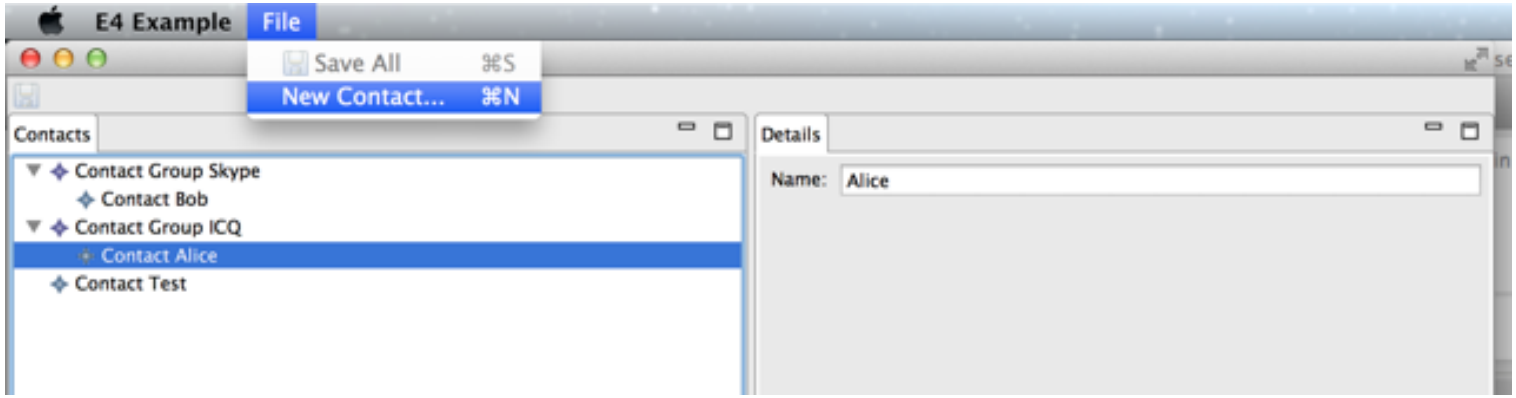
- The Application Model
- Implementing Views
- Selection Service

- Handlers, Commands and Items
- Editors

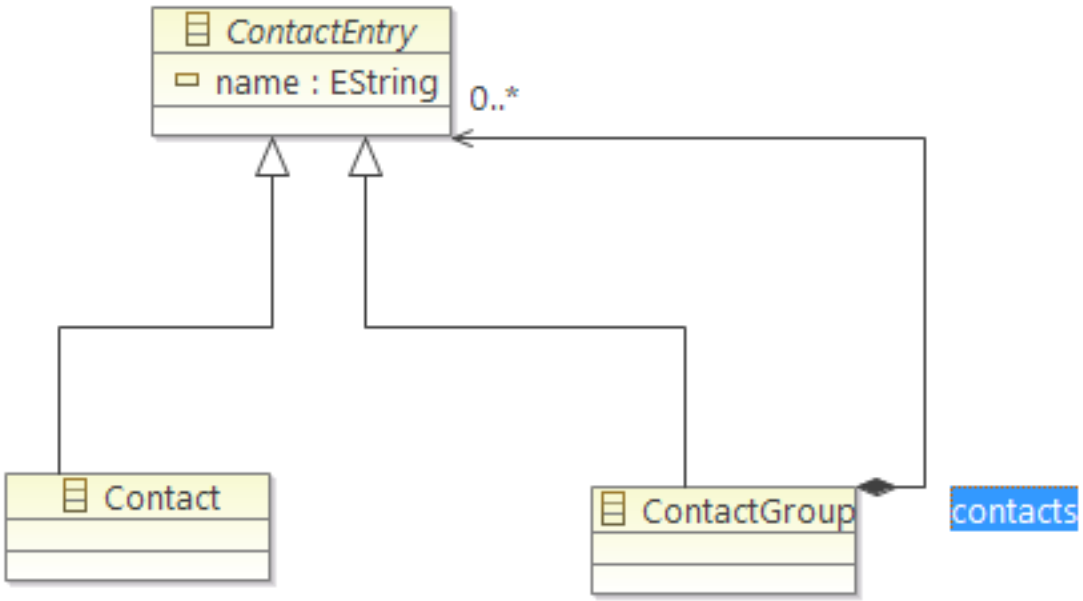
- Modularity
- Migration

http://goo.gl/Xn2Bp

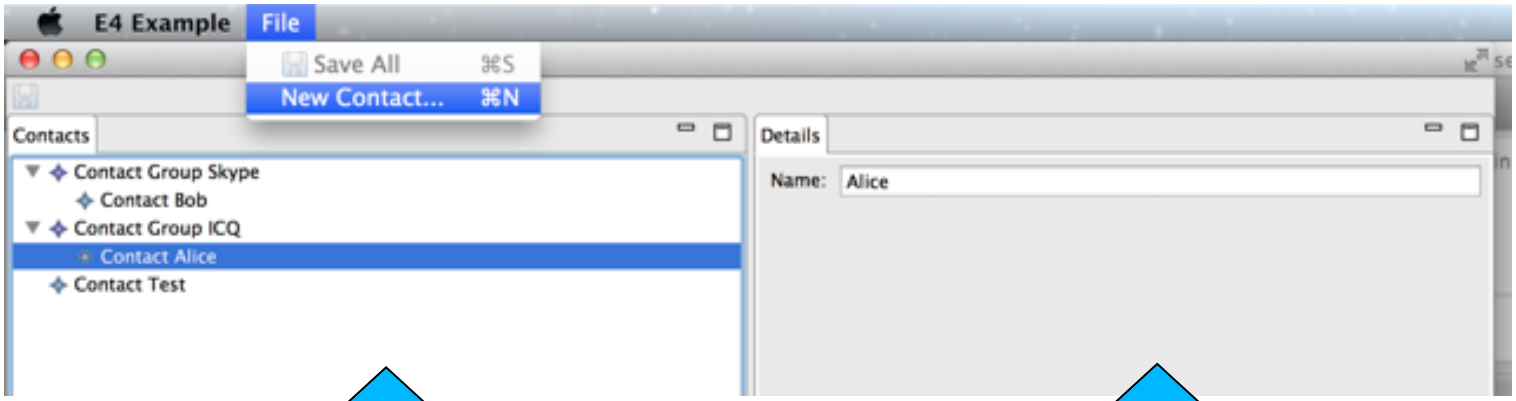
Example Application



The Application entity model



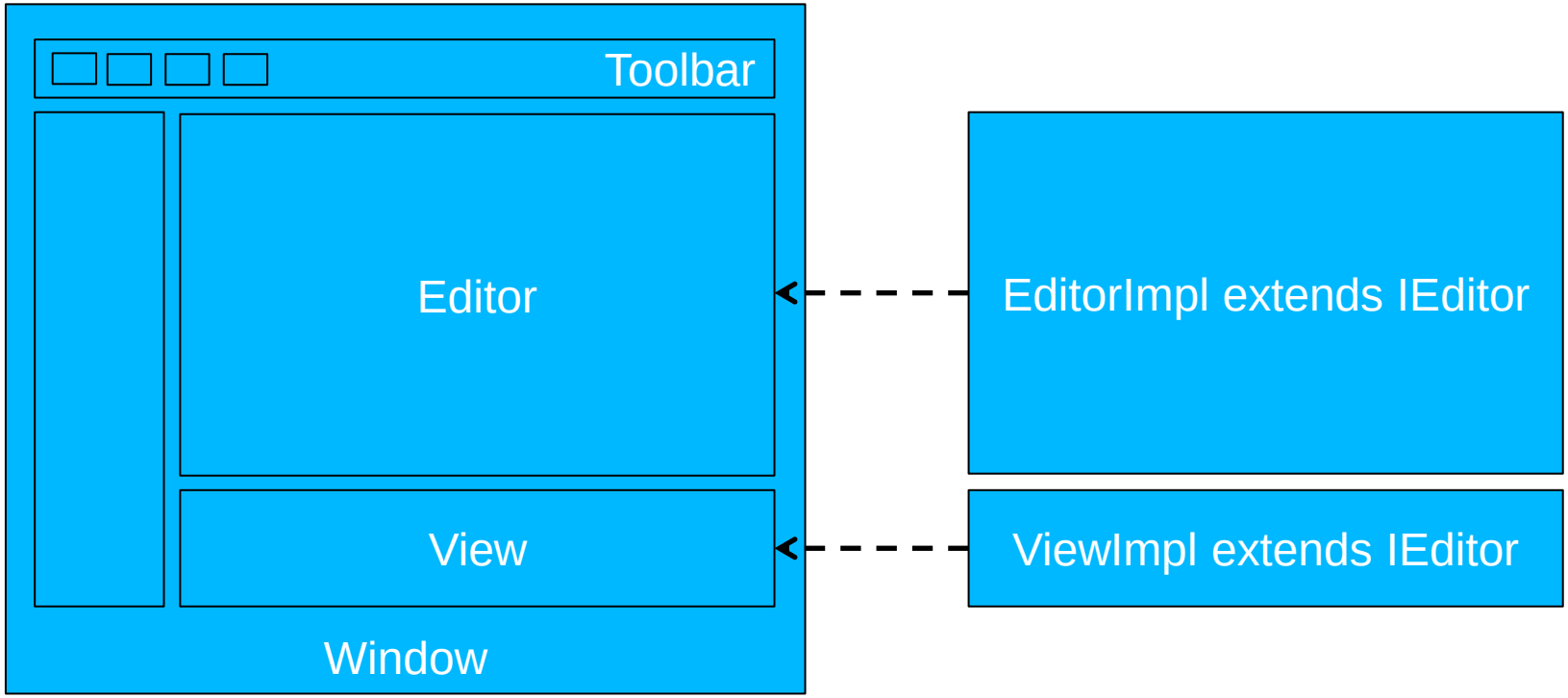
Goal 1: Two Views



ContactsView

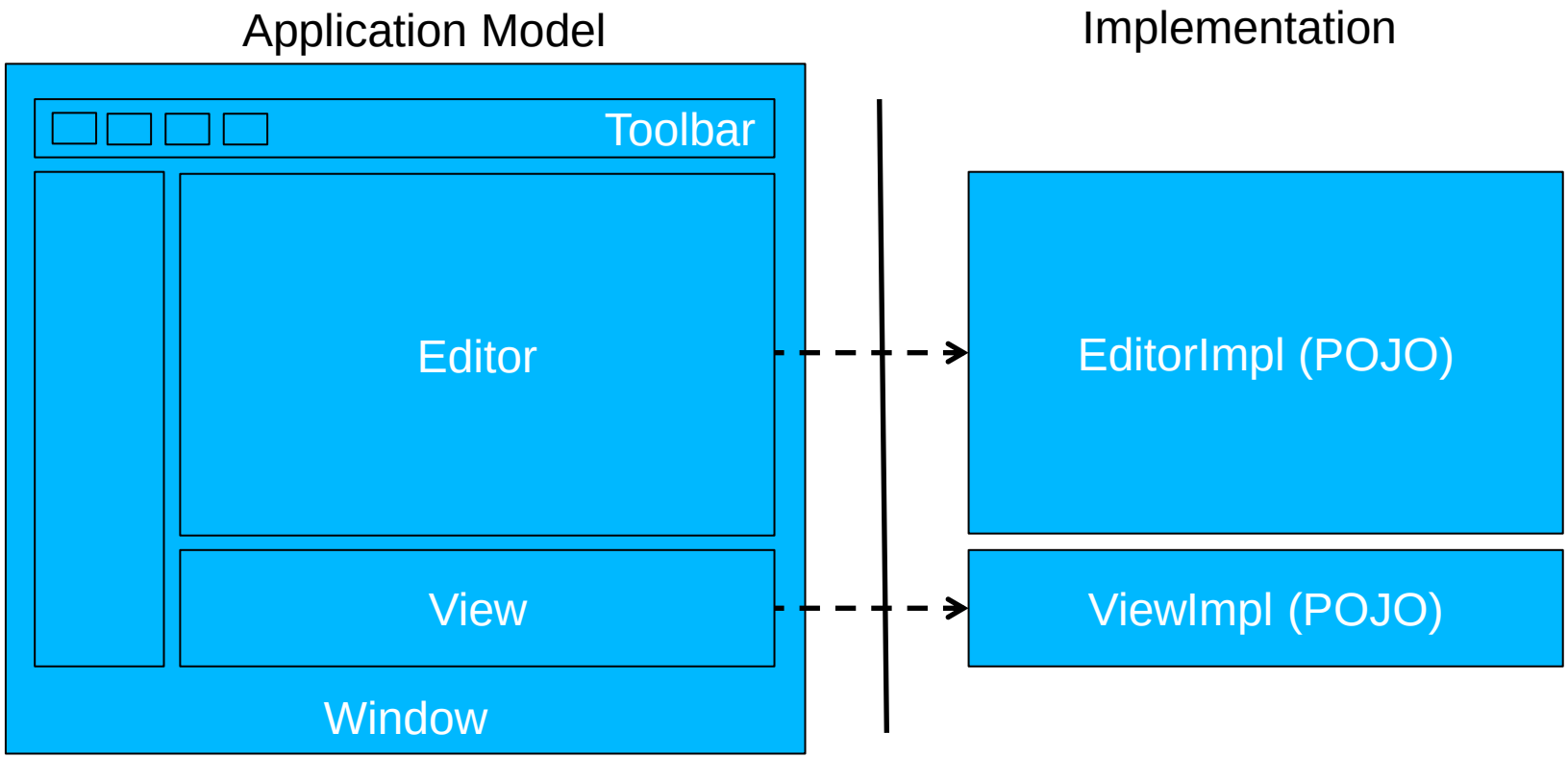
DetailsView

The 3.x Workbench



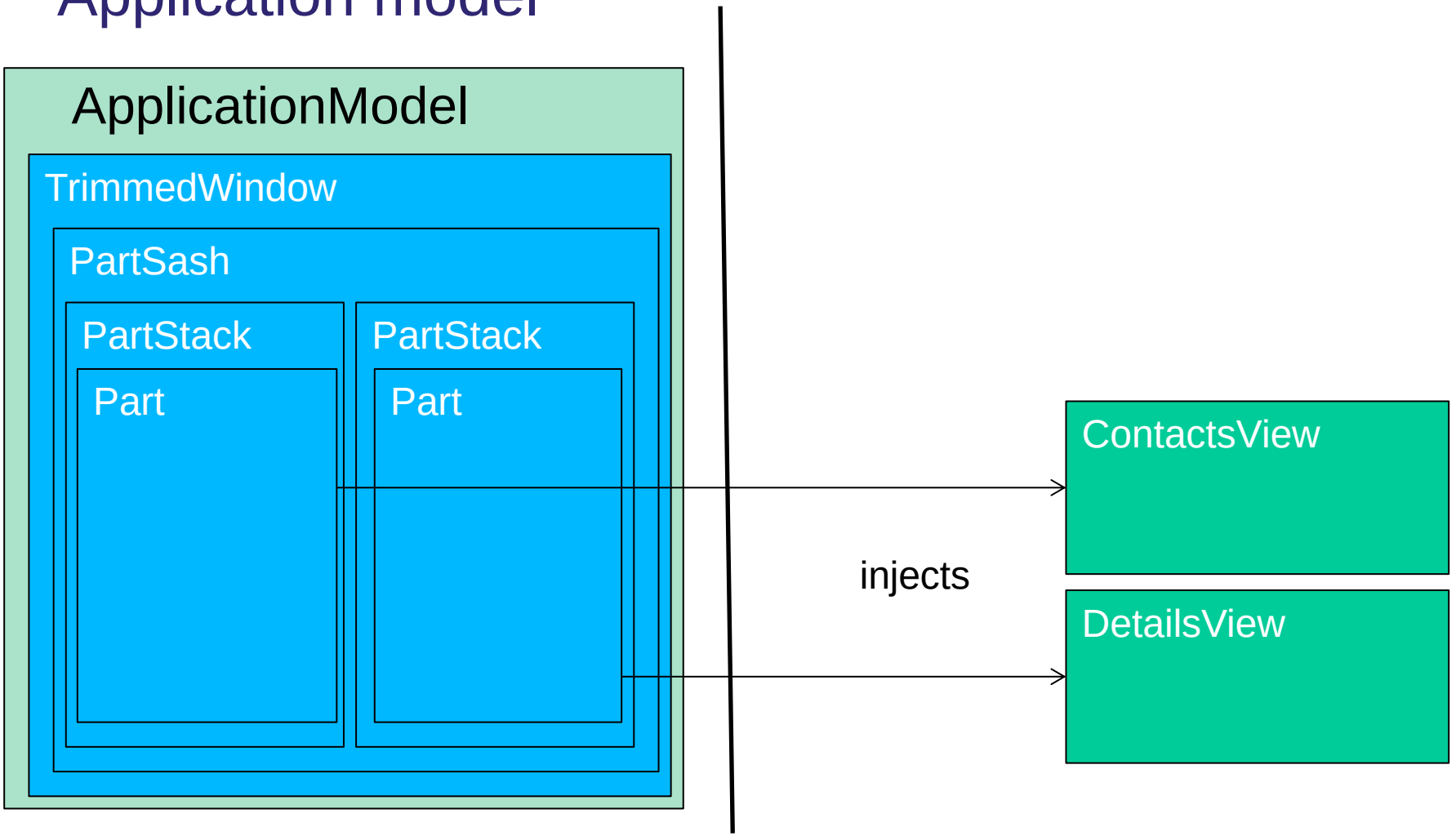
=> Strong coupling between Workbench and Implementation

The Eclipse 4 Workbench



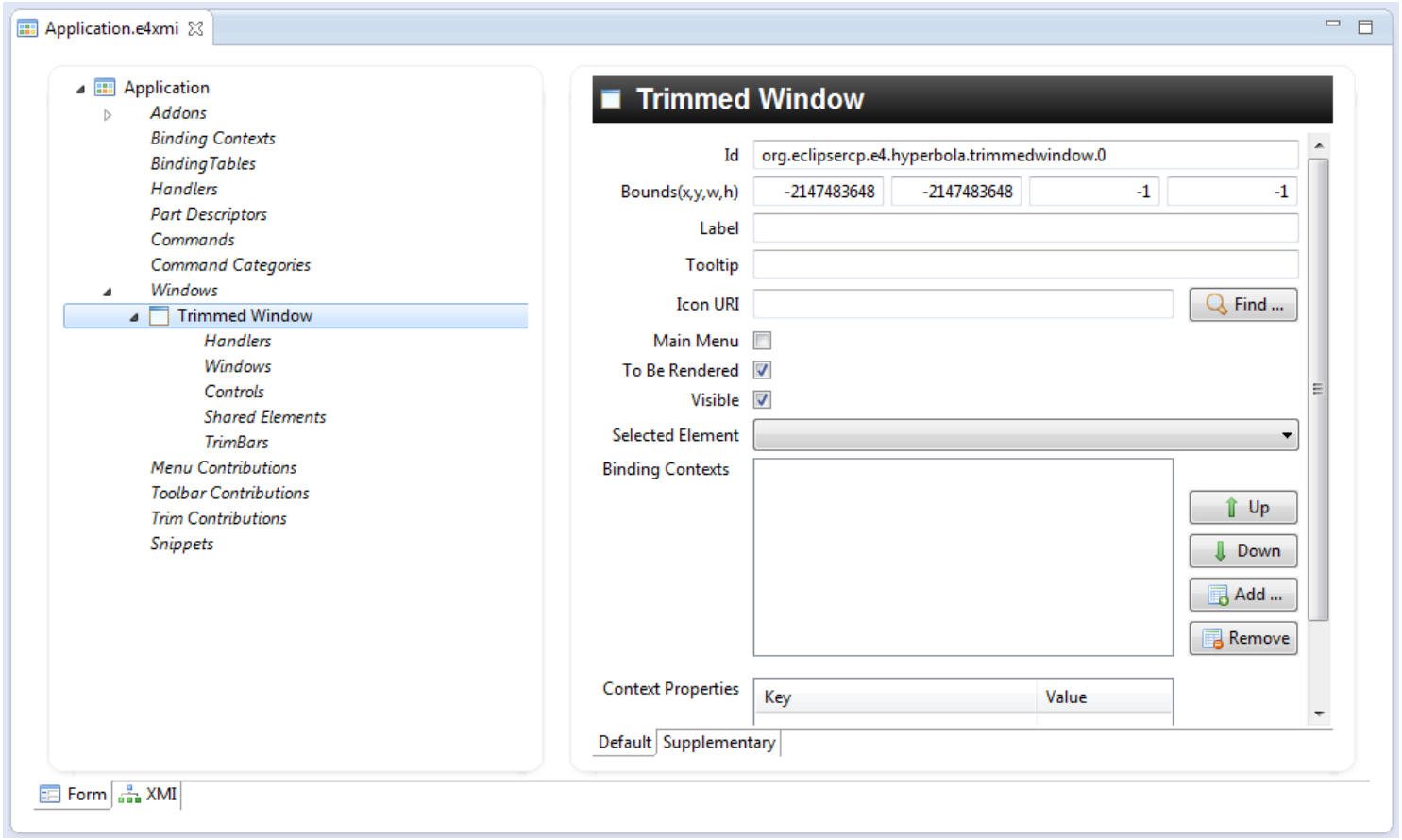
=> Loose coupling between Workbench and Implementation

Application model



http://goo.gl/Xn2Bp

The e4 tools editor

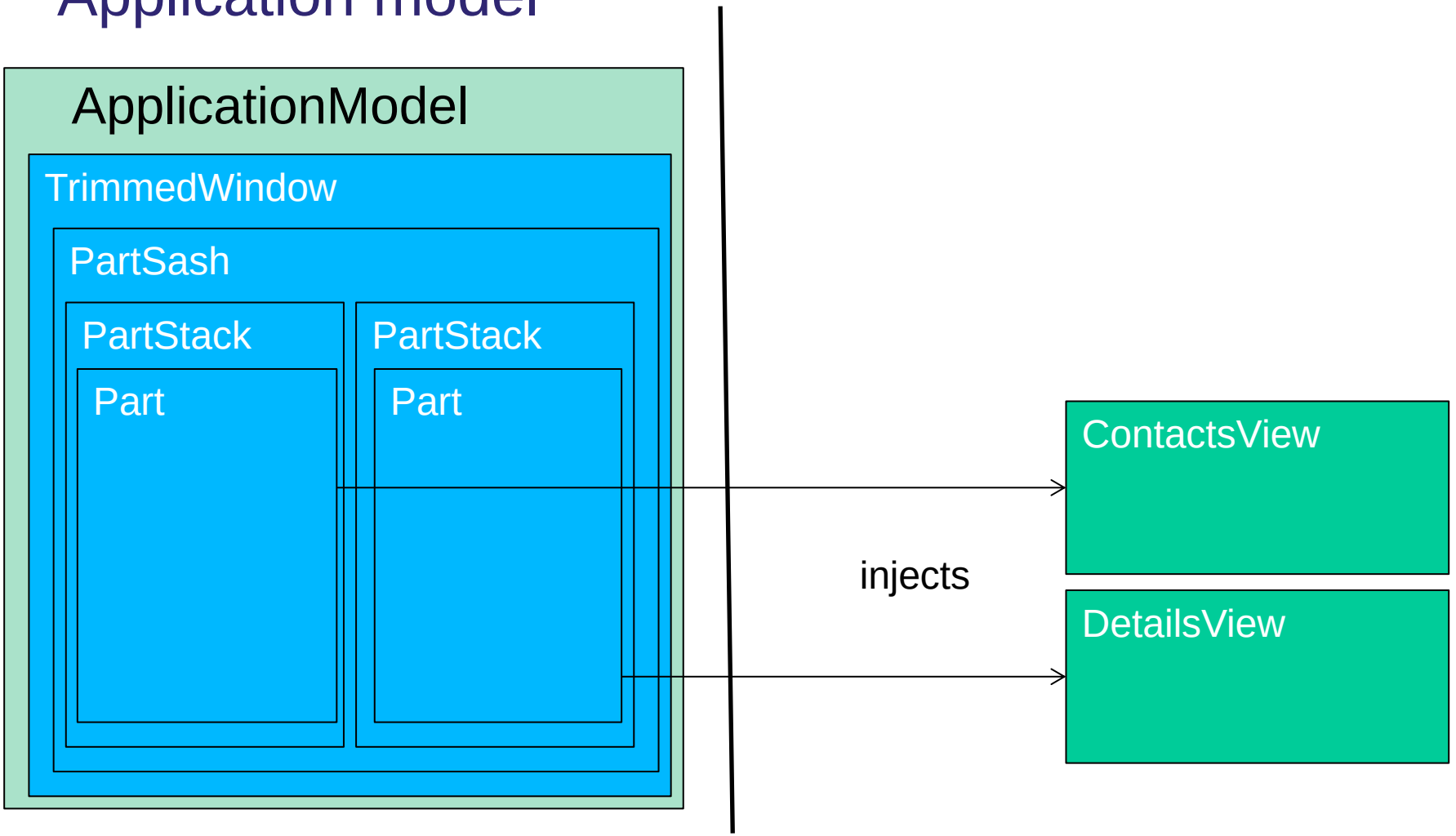


Elements of the Application Model

- Parts
- Windows, PartSash, PartStacks
- Handler, Commands, MenueItems, ToolBarItems
- KeyBindings

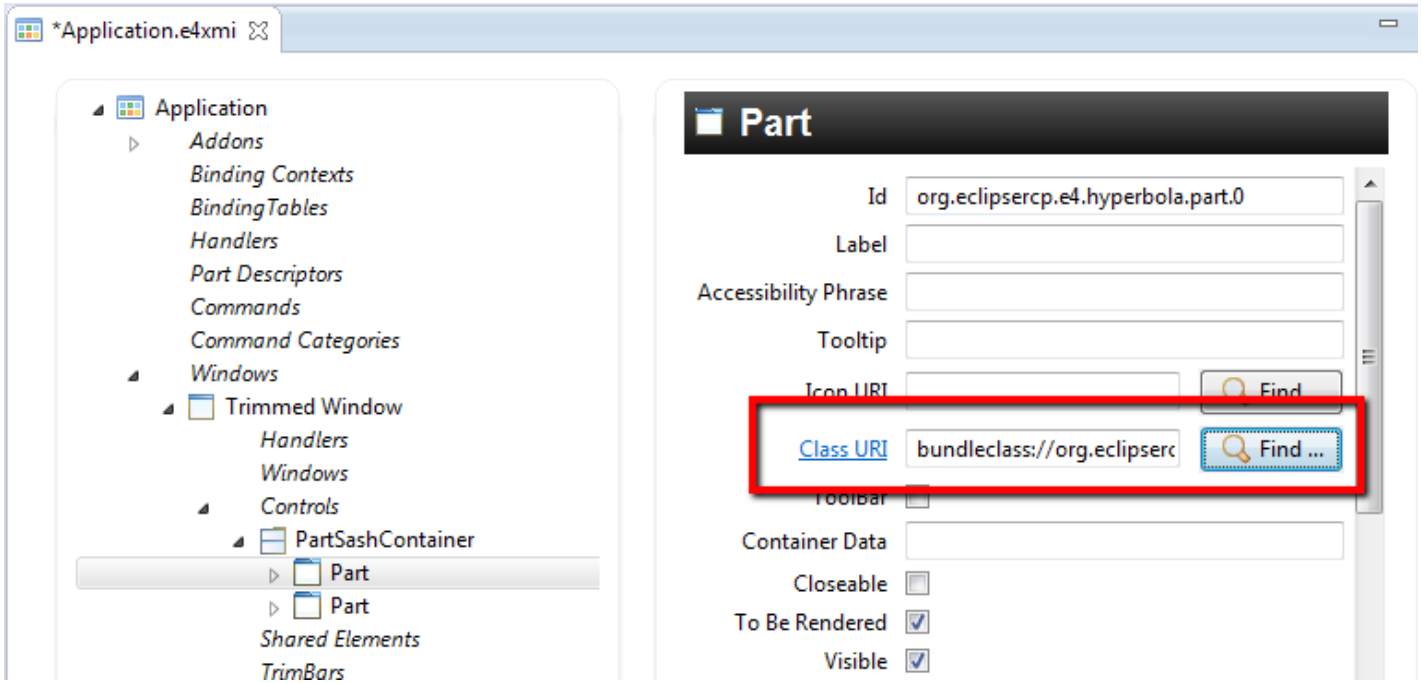
- Perspectives
- Placeholder
- PartDescriptors
- Addons
- …

Application model

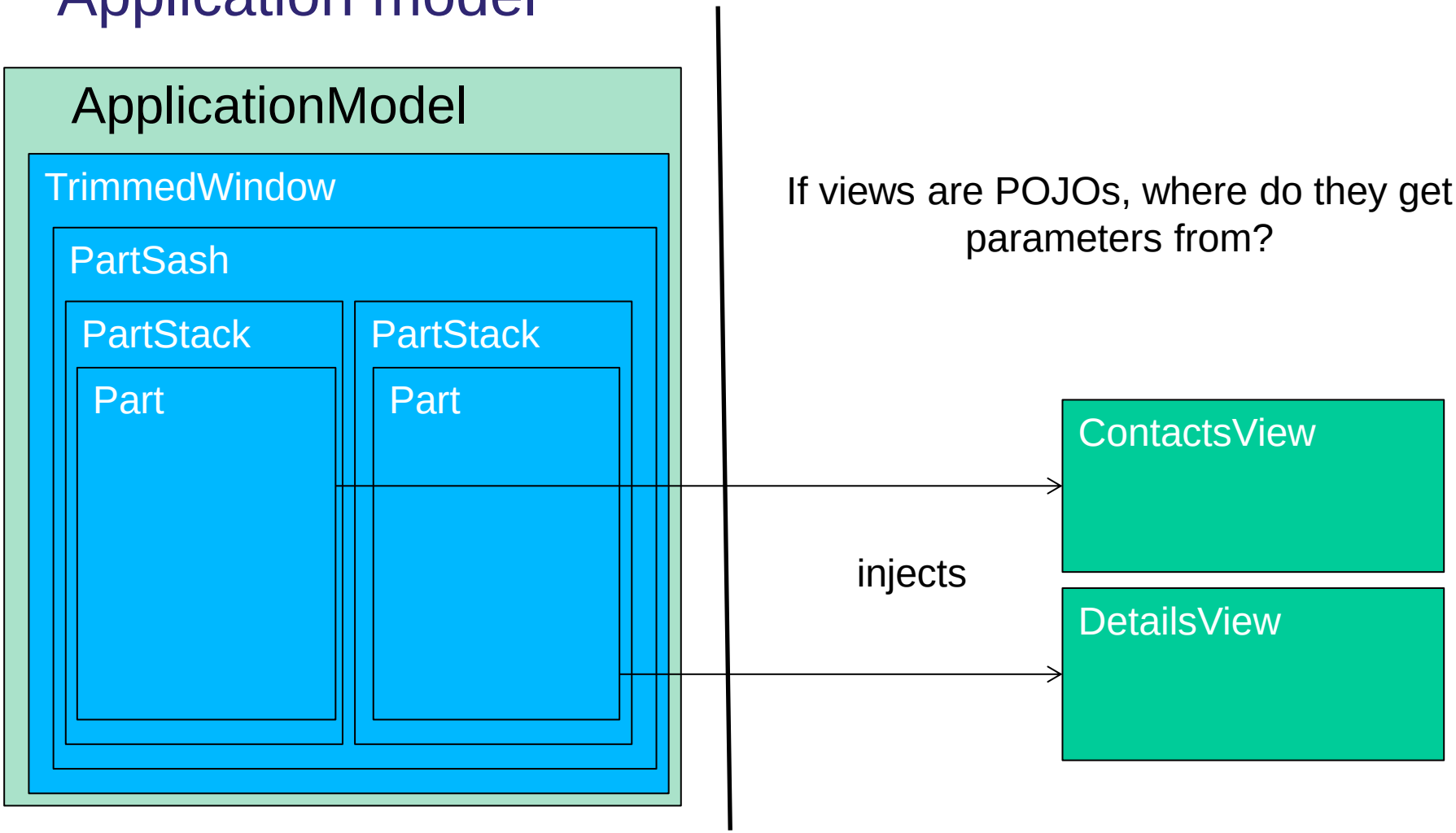


Connect views to the model

The class URI attribute binds a Part to its implementation

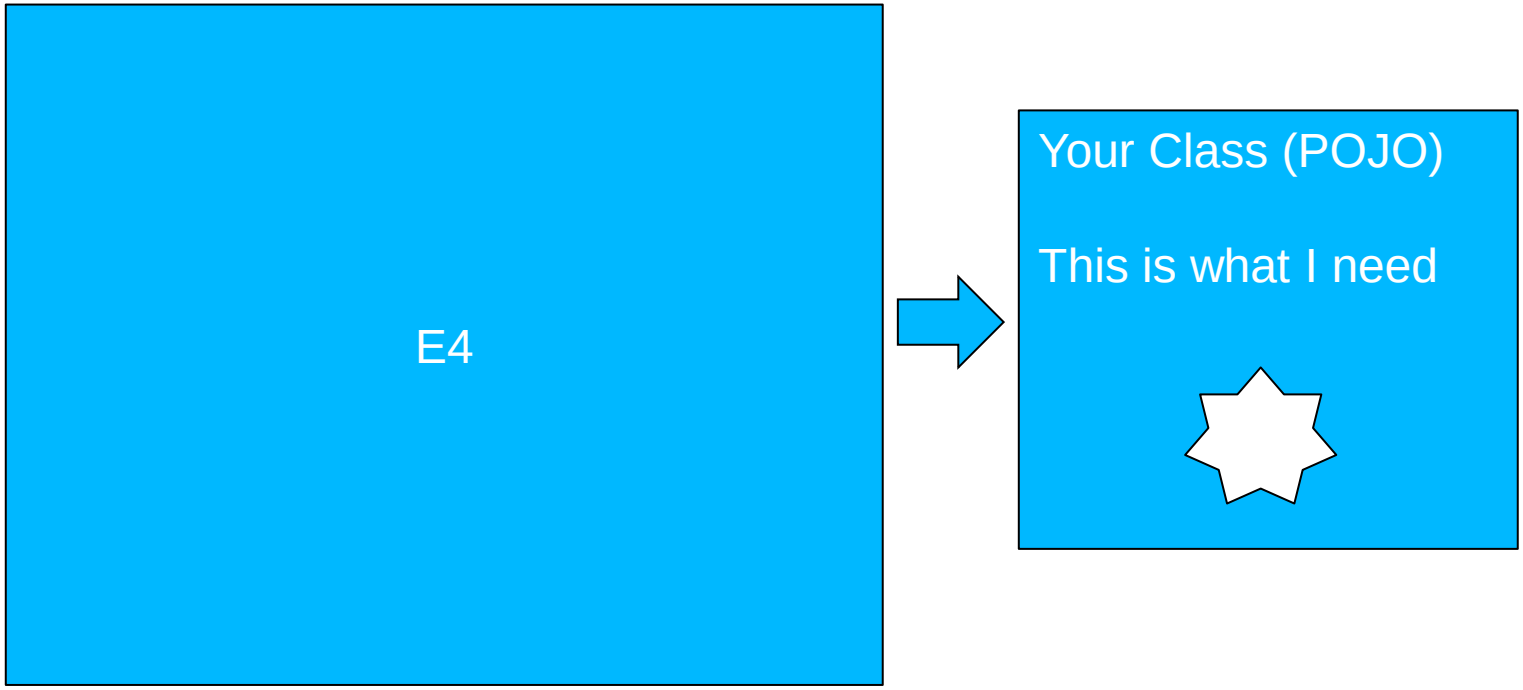


Application model



Dependency Injection

Hollywood principle: „Do not 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)
- Own objects
- Objects marked with @Creatable

How can Objects be injected

1. Constructor:

   @Inject

   public void MyClass(Composite parent)

2. Fields:

   @Inject

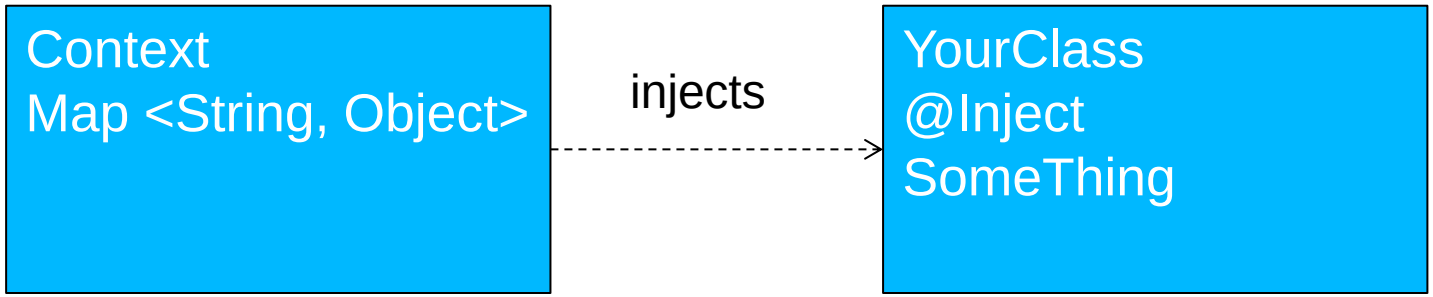
   Service service

3. Methods:

   @Inject

   Public void myMethod(SomeClass class)

Dependency Injection Details



Injections are updated if the value changes

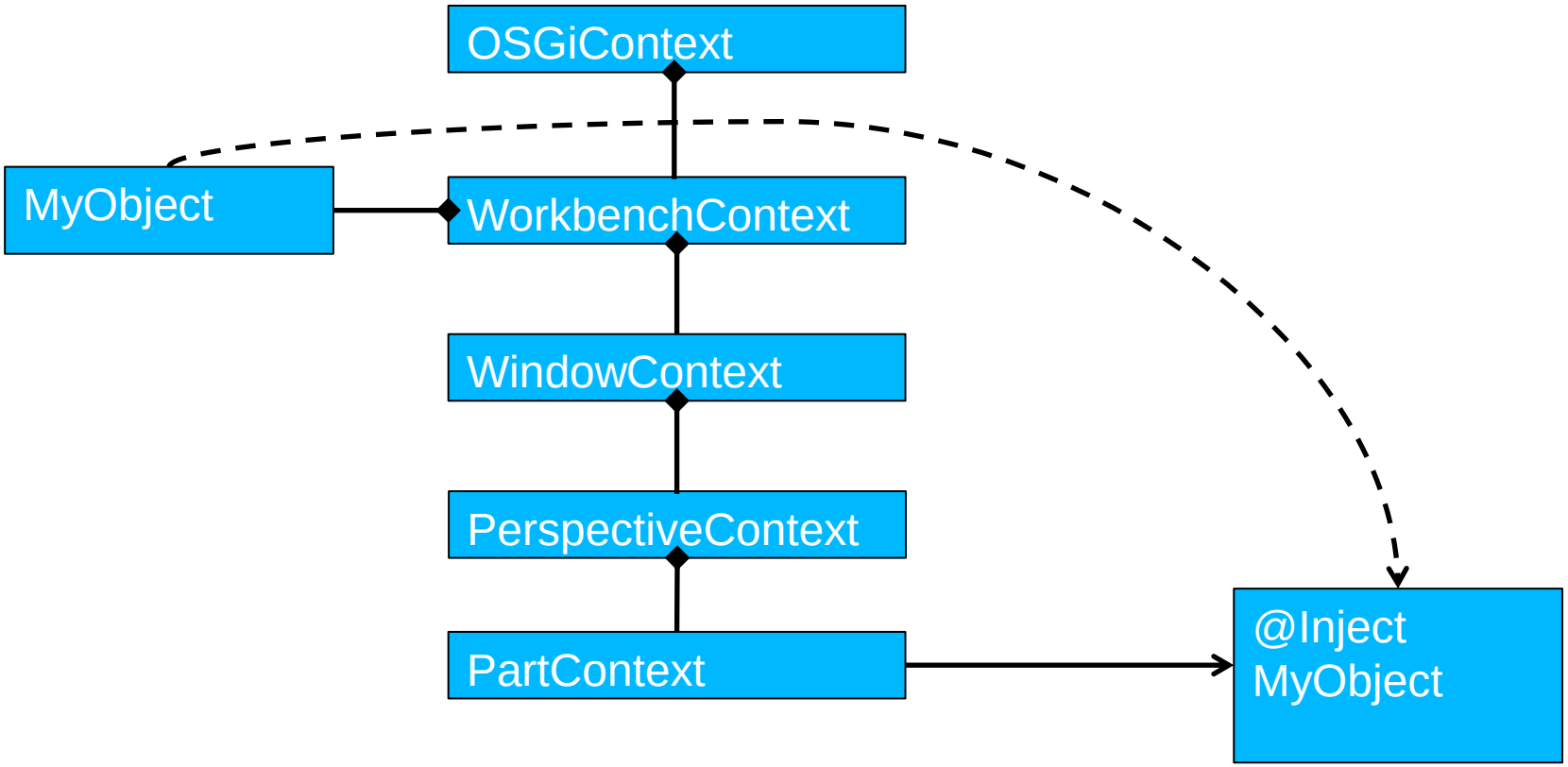
Contexts are hierachical

@Inject: Tries to satisfy the requested type

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

@Optional: Injects null, if not available

Hierachy 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 recieve 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
- View can be tested using plain SWT
- Needed parameters are injected later on, e.g. the parent composite:
  ```
  @Inject
  MyView(Composite parent){
  //Implement View
  ```
- Services are injected, too

OSGi Services can be injected

```
@Inject
ESelectionService selectionService;

@Inject
ContactService contactService;
```

Needs to be initialized! This is not possible in the constructor, as fields are injected after the constructor is called.

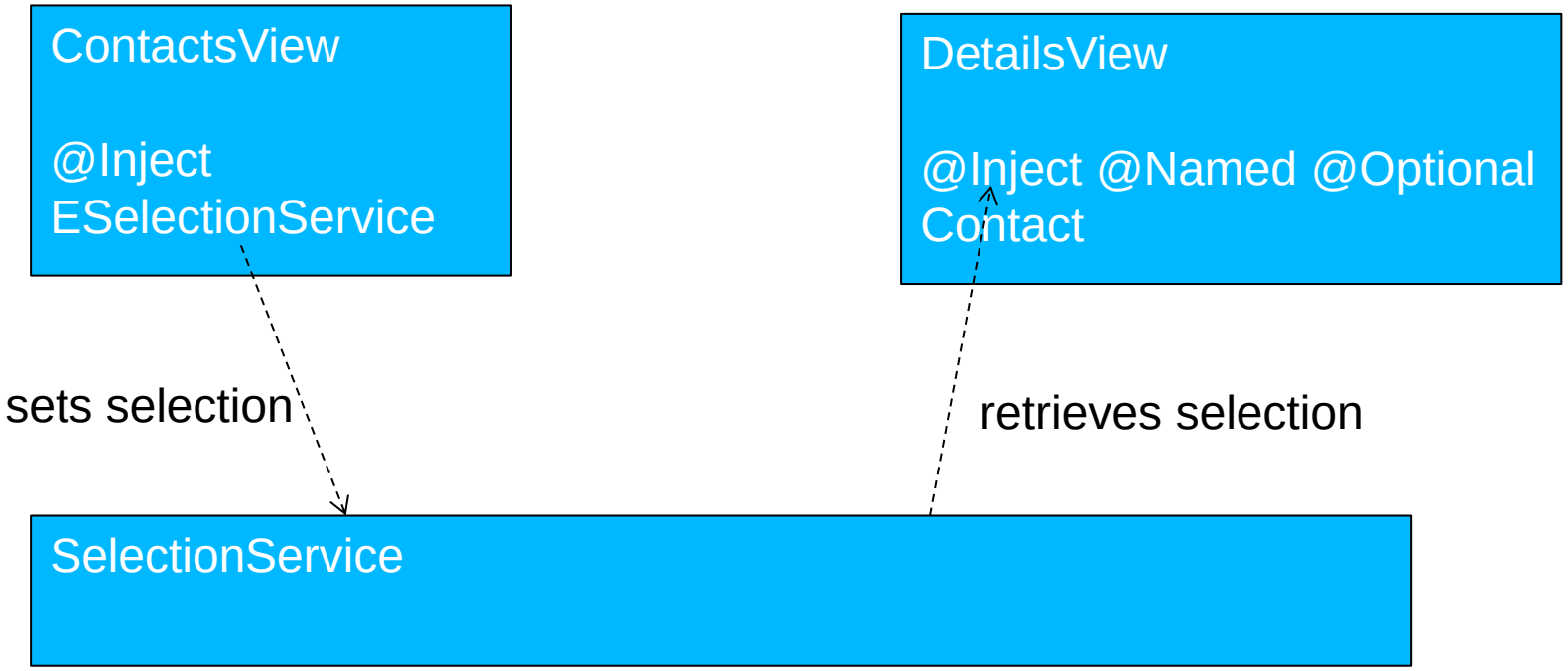
Annotations in Views

```
@Inject
ESelectionService selectionService;

@PostConstruct
public void init(){
    //Do sth. with service
}
@PreDestroy
public void dispose(){
    //Unregister listener
}

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

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 react to

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

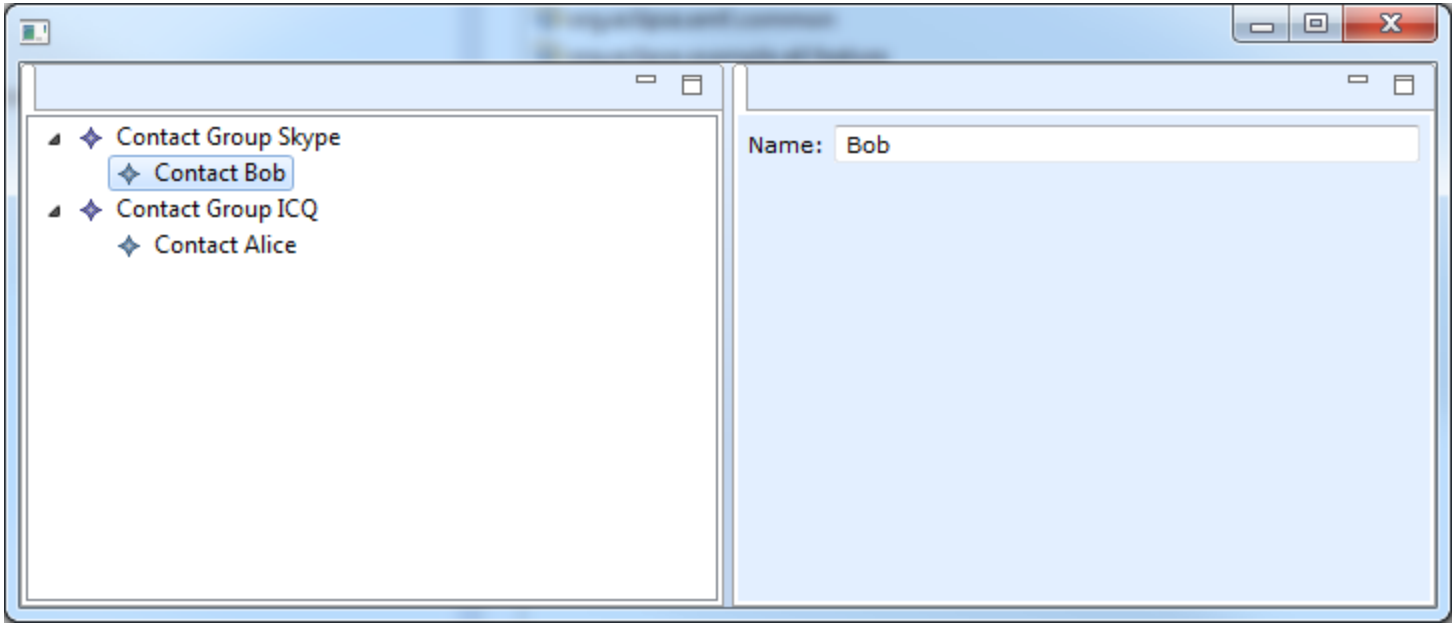
SWT Helper in model.edit

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

Task: Application Model

- Create a new application (empty model) org.eclipse.example.e4
- Add a PartSashContainer to the window
- Add two PartStacks and Parts within them
- Implement two POJOs for the parts
  - Left: ContactsView, Right: DetailsView
- Connect them to the parts
- Use the ContactService to retrieve an input for the TreeViewer (in a @PostConstruct method)
- Connect both views, using the selection service

Result taks 1



http://goo.gl/Xn2Bp

```
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);
    }

}
```

http://goo.gl/Xn2Bp

```
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());
    }

}
```

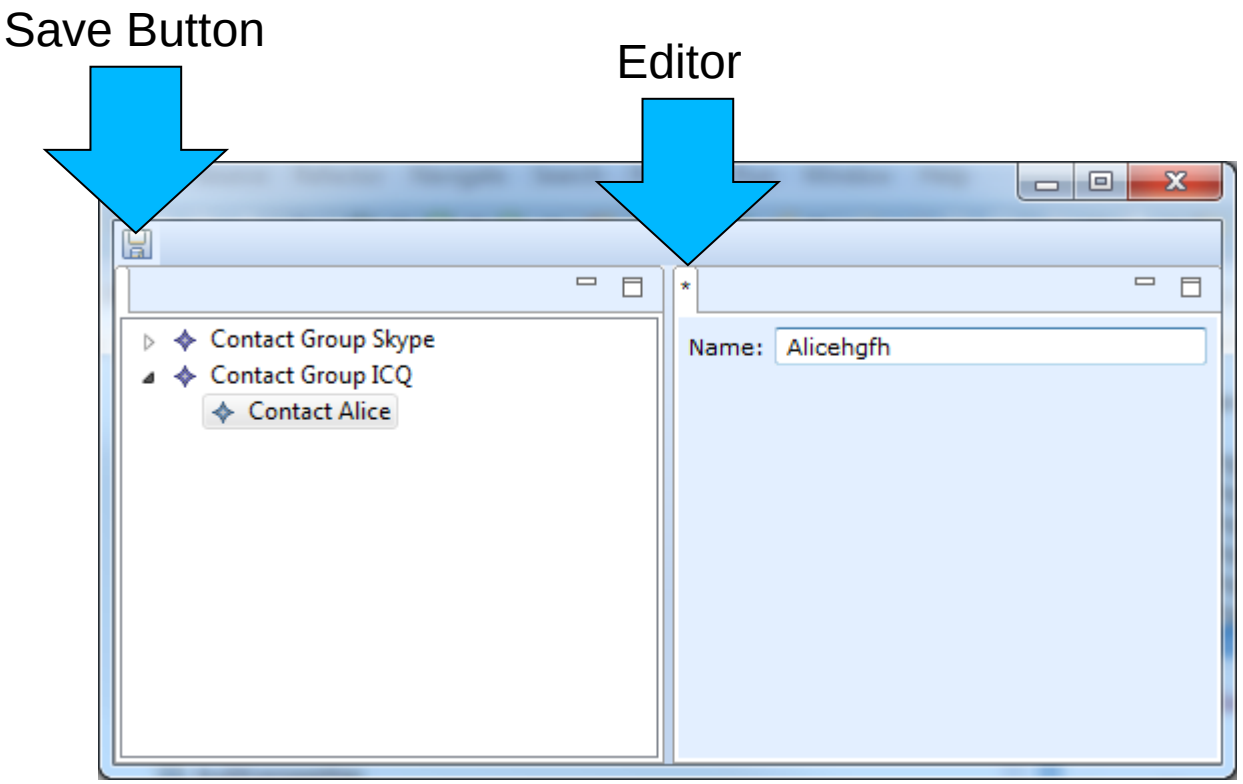
Outline

- The Application Model
- Implementing Views
- Selection Service

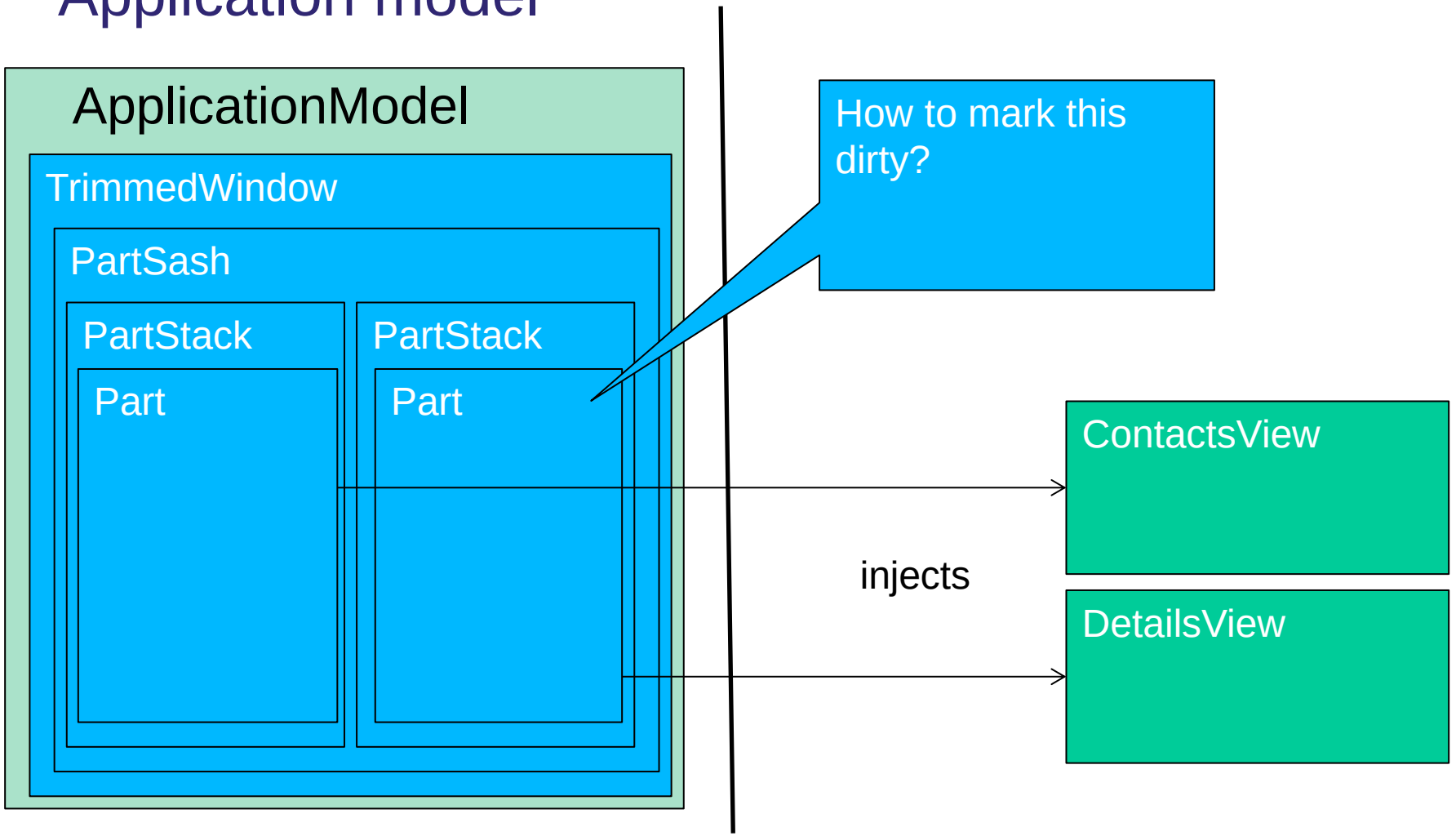
- Handlers, Commands and Items
- Editors

- Modularity
- Migration

Goal


Save Button
Editor
Contact Group Skype
Contact Group ICQ
Contact Alice
*
Name:
Alicehgfh

Application model



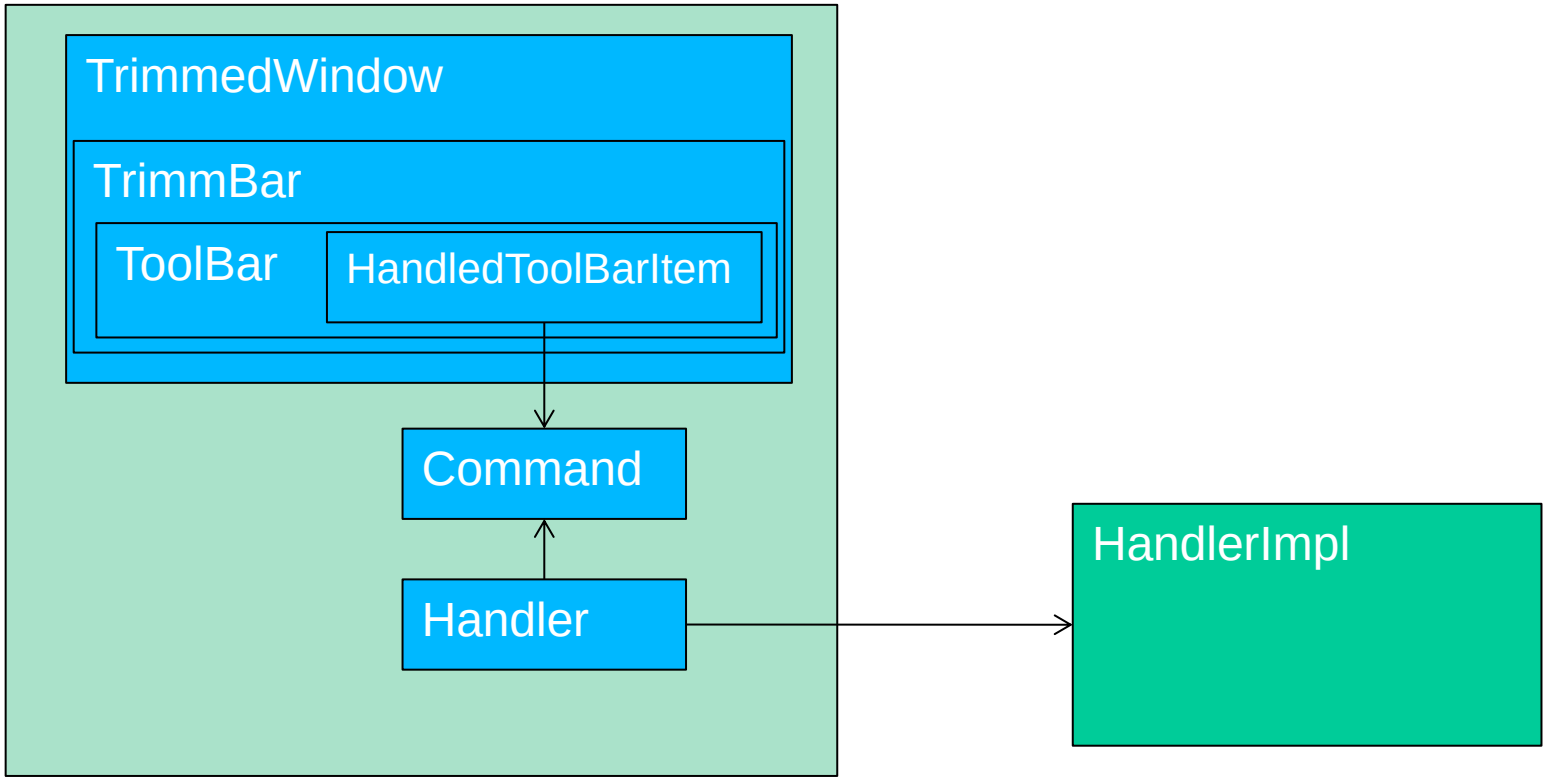
Editors in Eclipse 4

- 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

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



Handler

- Implementations are POJOs
- The can easily be tested
- Mark the method to be execute 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

Save Button in e4

- Active part can be injected can checked if 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()

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 the save method to write changes to the contact

- 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

DetailsView

```
@Inject
MDirtyable dirtyable;
```

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

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

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

        }
    });
}
```

```
@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);
}
```

http://goo.gl/Xn2Bp

```
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();
    }

}
```

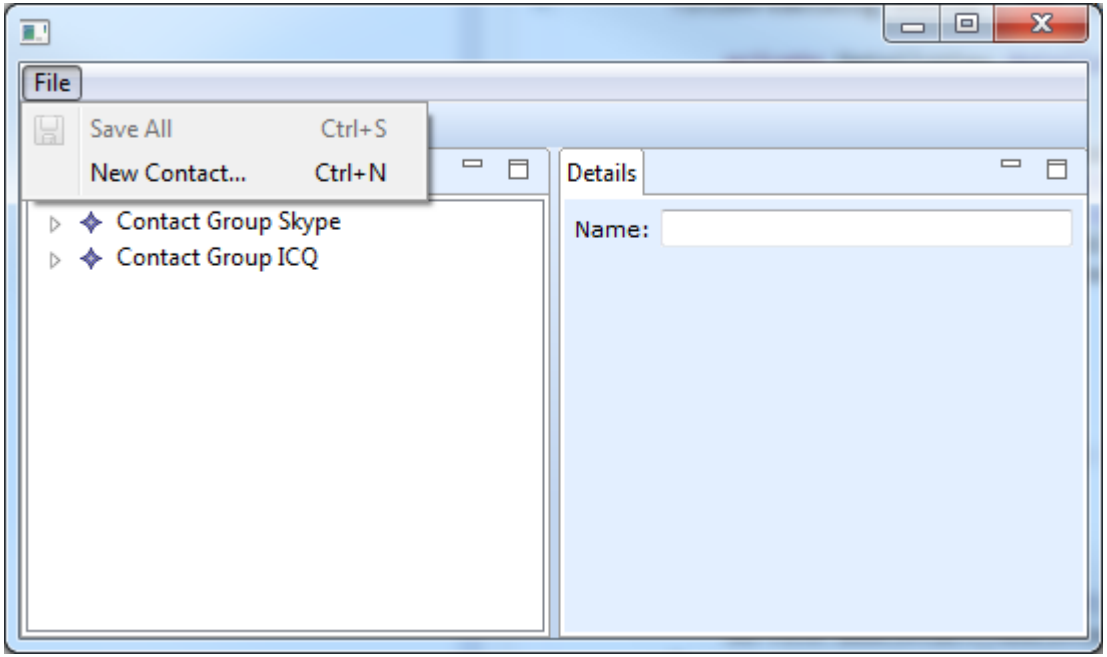
Outline

- The Application Model
- Implementing Views
- Selection Service

- Handlers, Commands and Items
- Editors

- Modularity
- Migration

Optional Task



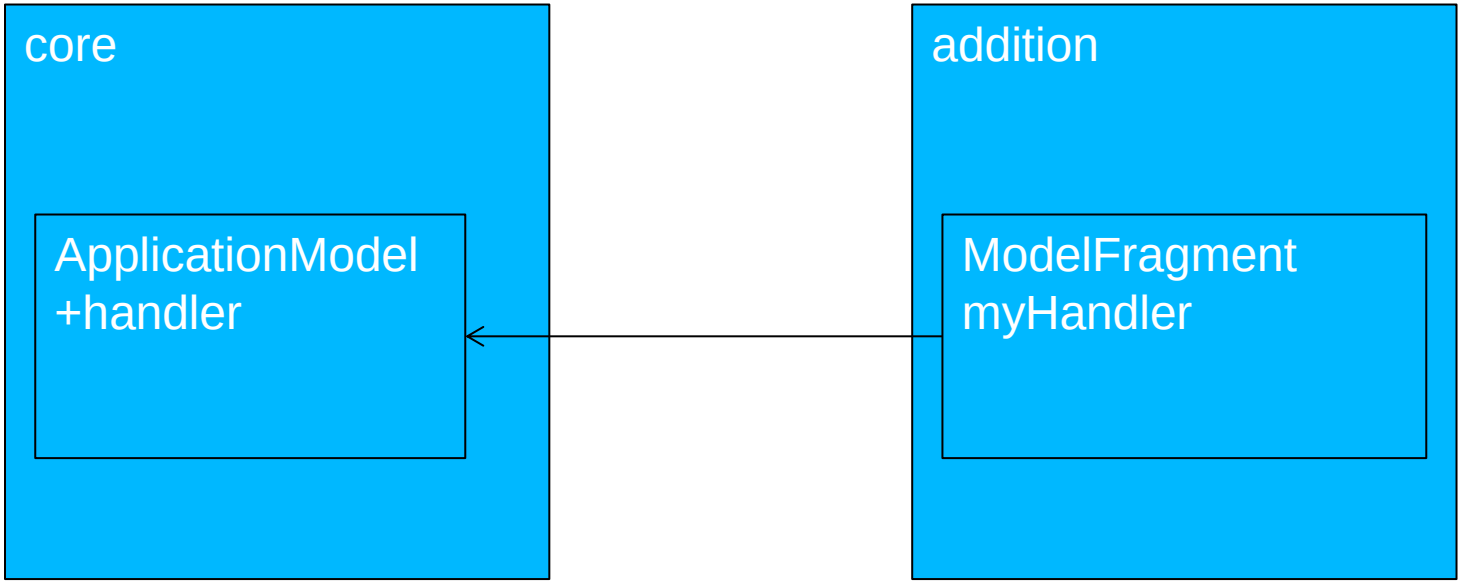
Dependency Injection Details

- 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

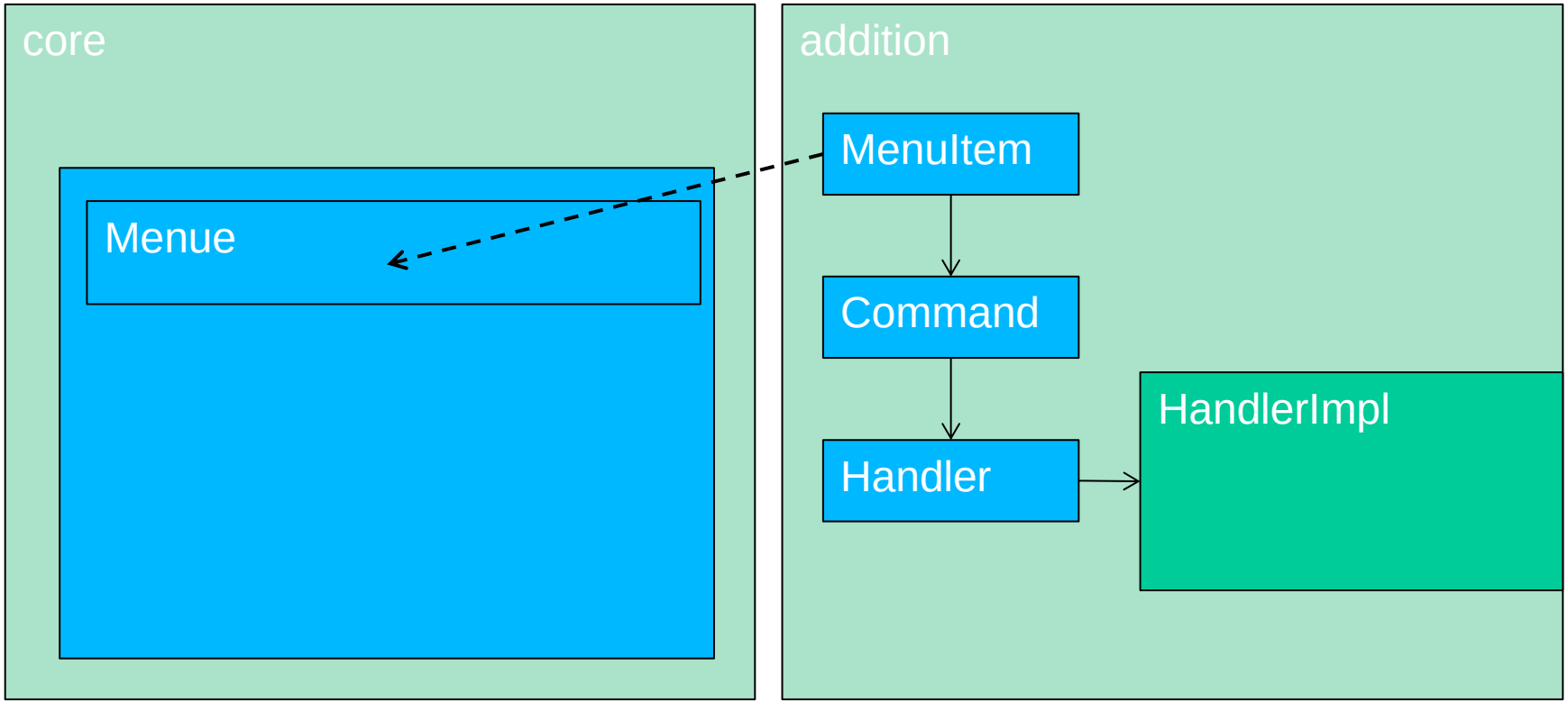
- Injection can be manually triggered: ContextInjectionFactory.make(DetailsView.class, context);
- Elements marked with @Creatable are automatically created

Add contributions from other plugins

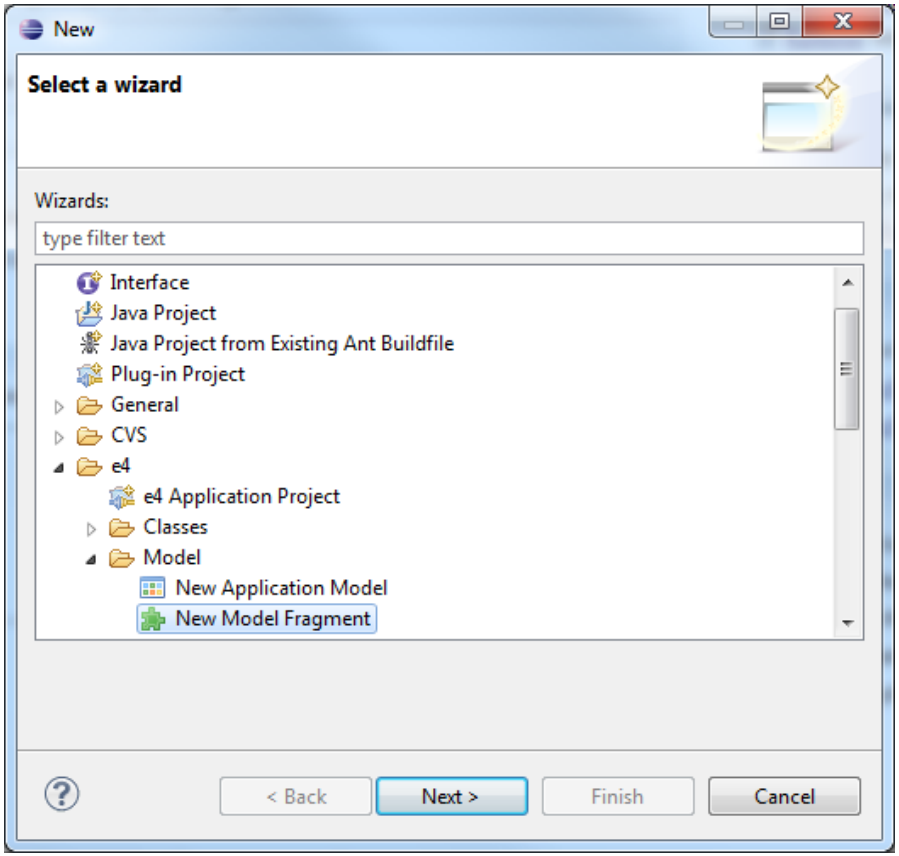
- Contributions can be added from plugins, e.g. to deploy an optional feature



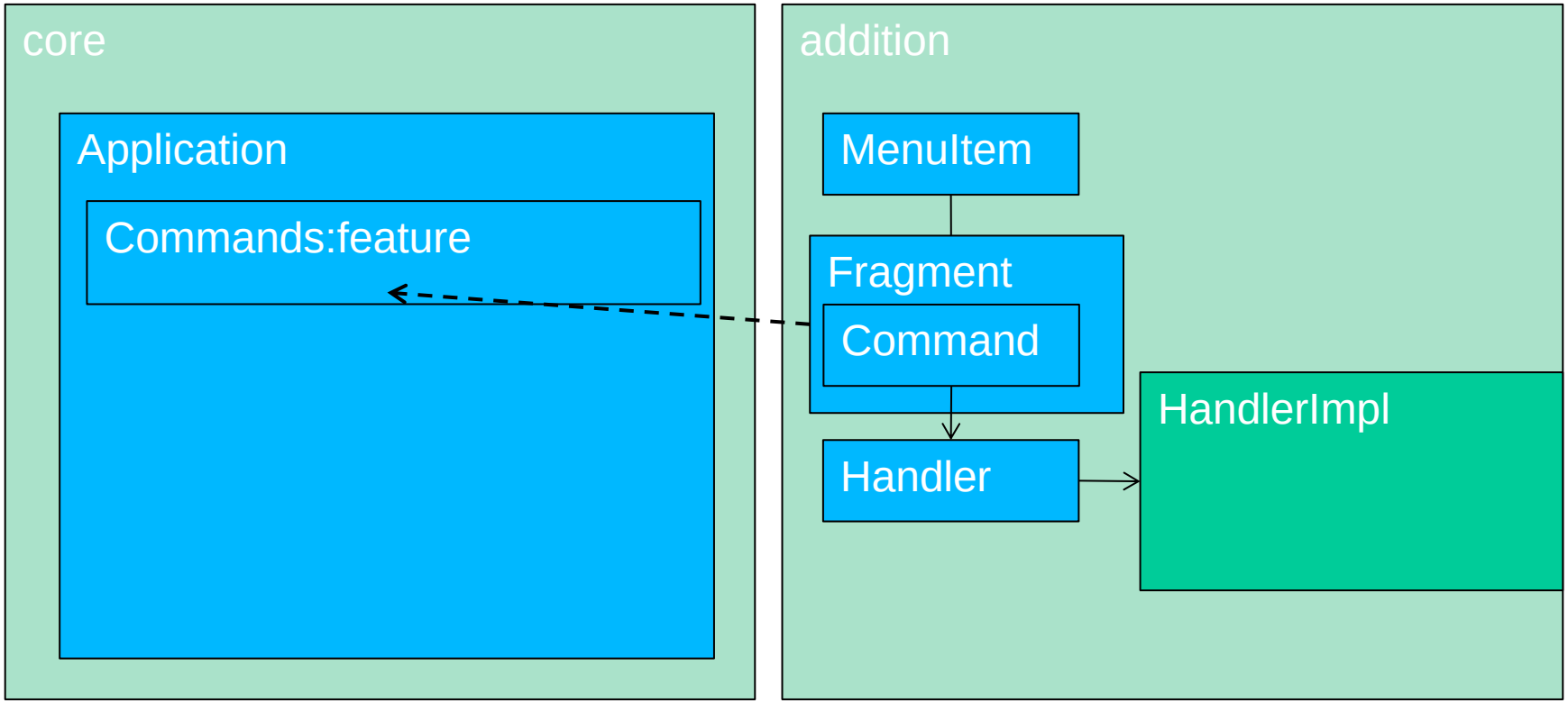
Extend the Application Model



Fragments contain new pieces of the model

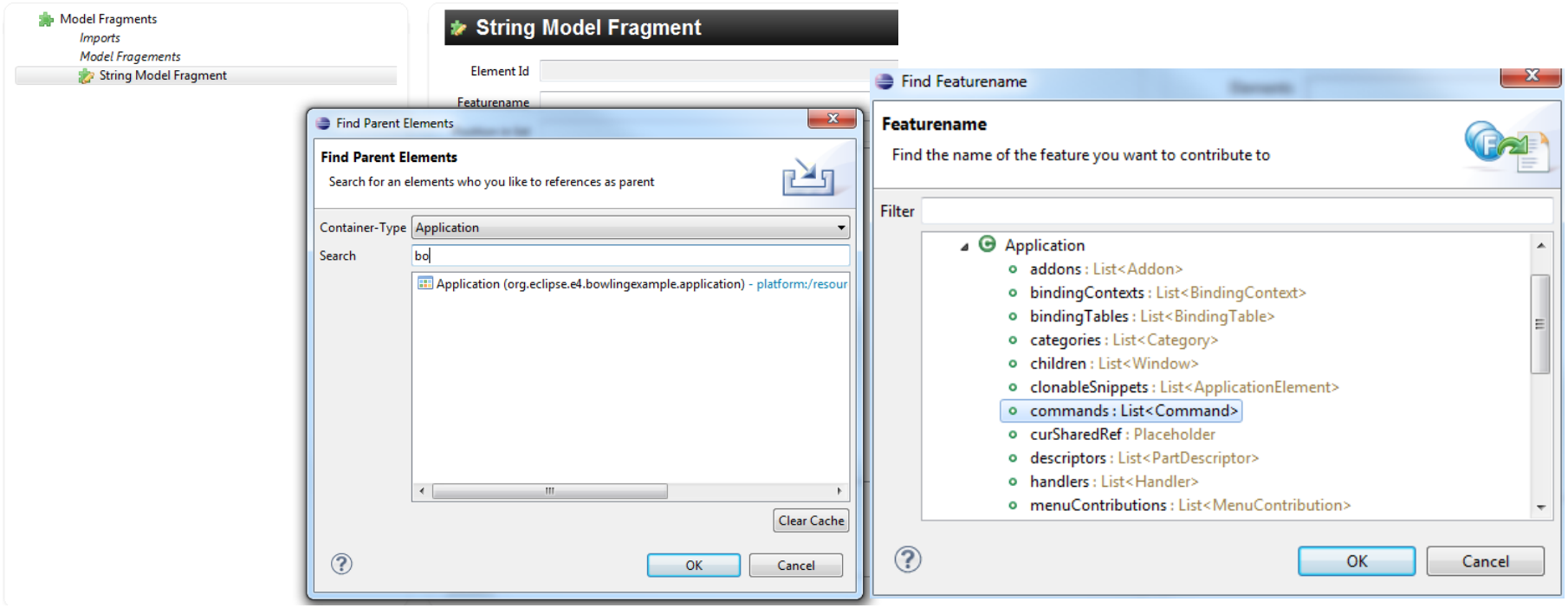


Extend the Application Model



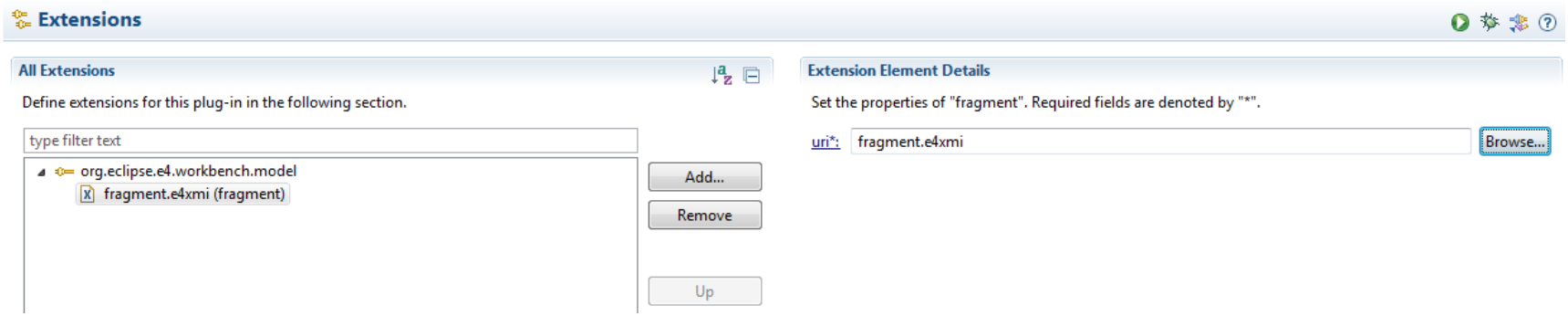
Extend the application model

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



http://goo.gl/Xn2Bp

Fragments need to registered

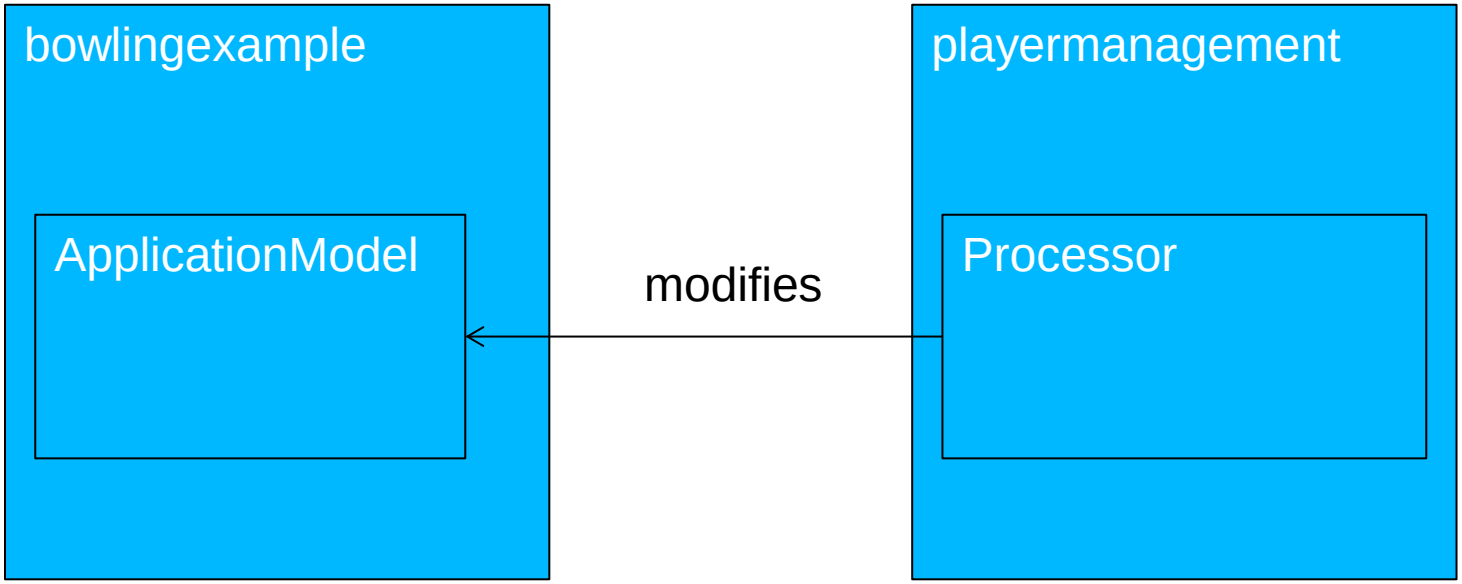


Additionally, the Application Model needs to be registered

```
<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.eclipsercp.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“
- The 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 componentes
- 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 componentes
- Disadvantages:
  - Mix of technologies

Ways of mixing e4 with 3.x

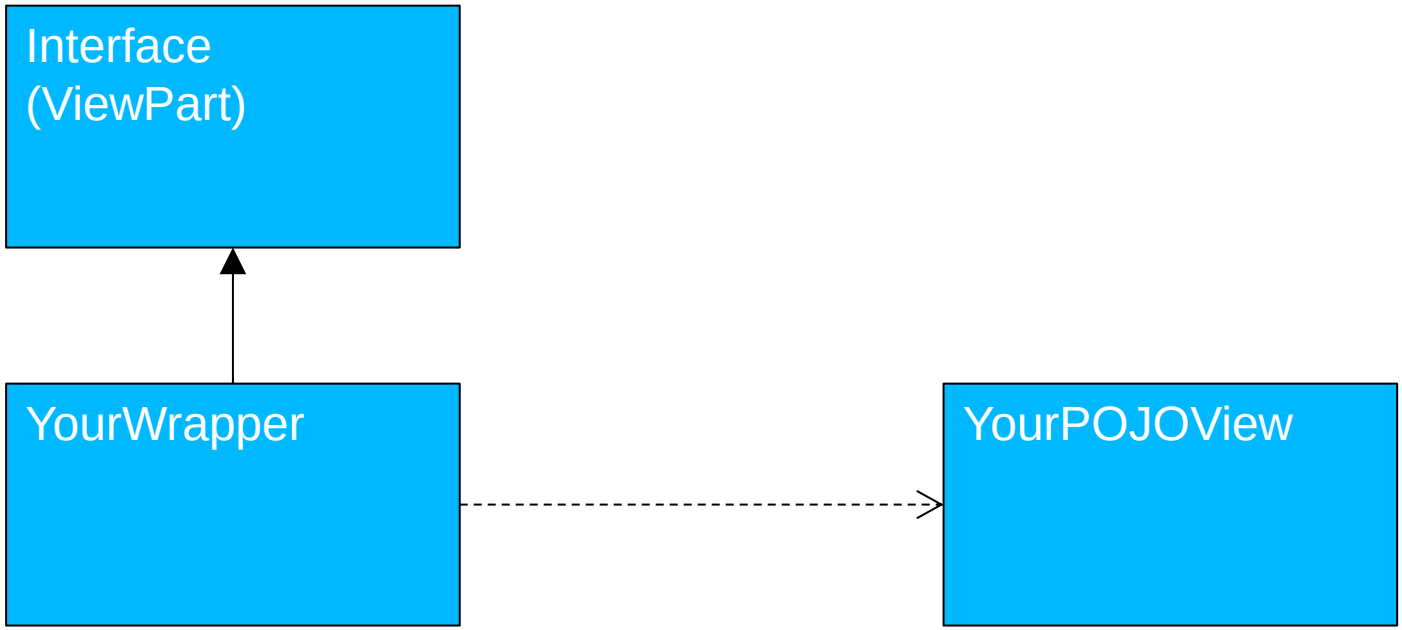
The compatibility layer mocks org.eclipse.ui and translates all calls and extensions to an Application Model in the background

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

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

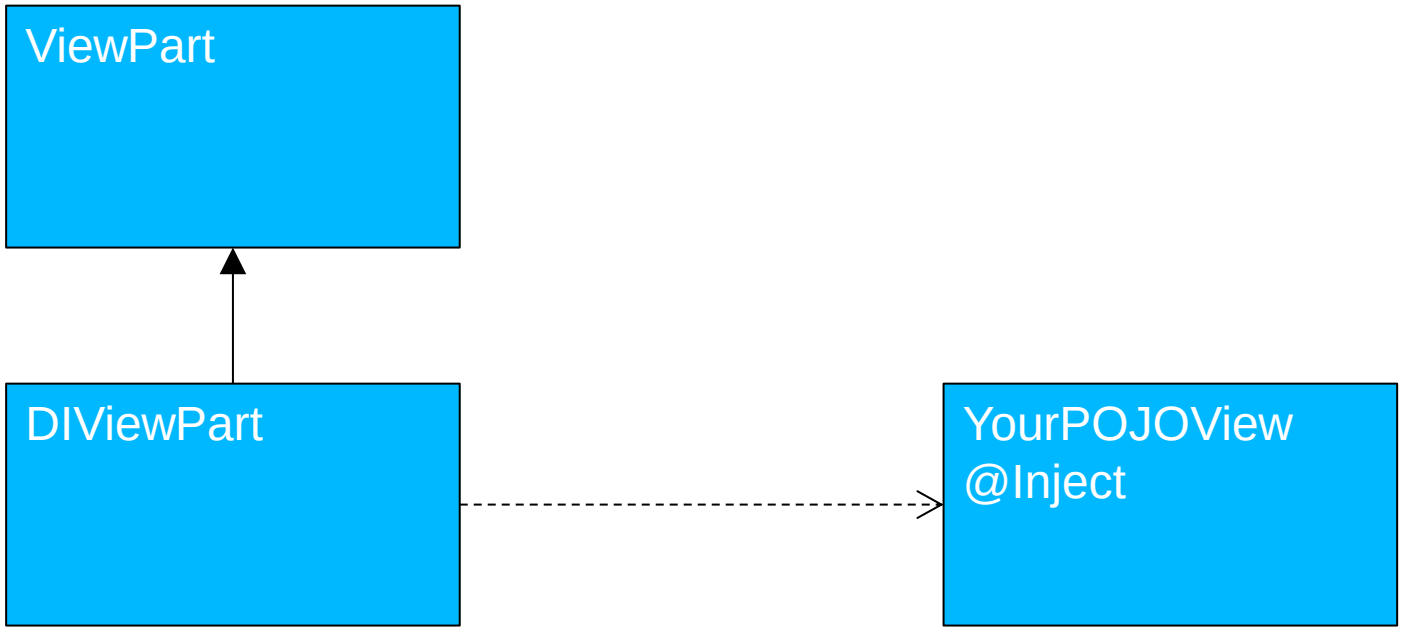
Single Sourcing

Keep your application independant from technologies



Single Sourcing

There are wrapper implementations supporting dependecy injection (e4Tools Project)



More Information

- E4 Wiki
- E4 Newsgroup
- Blogs
- Books
- eclipsesource.com/eclipse4tutorial

- Professional Training => eclipsesource.com
- Jonas.Helming@eclipsesource.com