Developing Rich Clients with Eclipse 4.x RCP

Kai Tödter, Siemens AG

Tom Schindl, BestSolution.at

Who is Kai?

- Software Architect/Engineer at Siemens Corporate Technology
- Eclipse RCP expert and OSGi enthusiast
- Open Source advocate
- Committer at e4 and Platform UI
- E-mail: kai.toedter@siemens.com
- Twitter: twitter.com/kaitoedter
- Blog: toedter.com/blog

Who is Tom?

- CEO BestSolution Systemhaus GmbH
- Committer
  - e4
  - Platform UI
  - EMF
- Project Lead
  - Nebula
  - UFaceKit

Outline

- Eclipse 4.x RCP Overview
- Creating a “Hello, World” RCP 4.0 application
- Workbench model
- Toolbar, menu, parts, commands, and handlers
- Dependency injection
- Services

e4 Objectives

Picture from
http://www.sxc.hu/photo/1081630

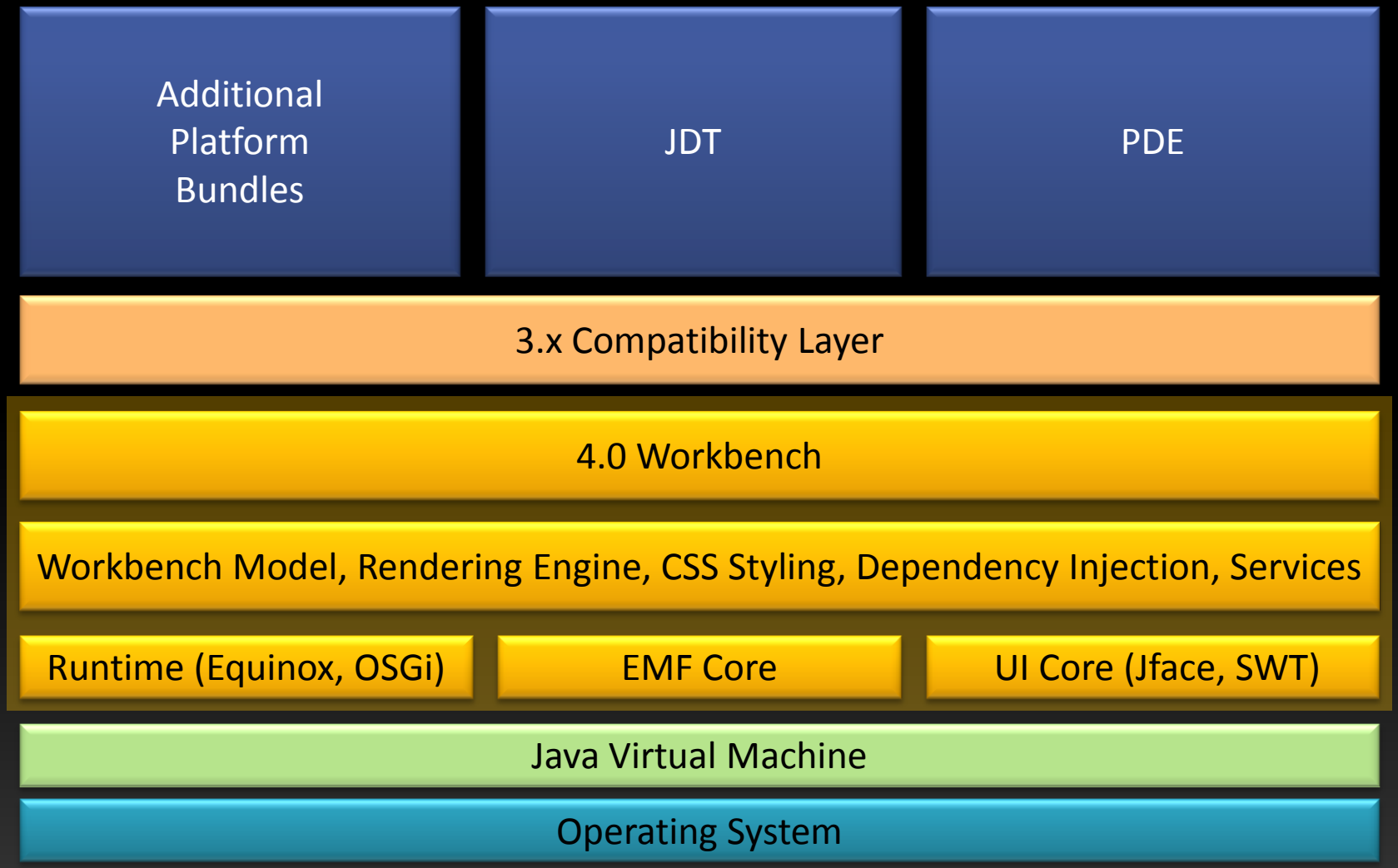
e4 Objectives

- Make it easier to write plug-ins
- Allow better control over the look of Eclipse based products
- Provide a uniform, pervasive platform across computing environments (Web / RIA, Desktop, Server, Cloud, Embedded)
- Increase diversity of contributors to the platform
- Maintain backward compatibility for API-clean clients

What's new in Eclipse 4.0?

- Workbench Model (based on EMF)
- Workbench objects are mostly POJOs
- Dependency Injection
- CSS Styling
- Rendering Engine
- Services (aka "the 20 things")
- …

Eclipse RCP 4.x Architecture



Outline

- Eclipse 4.0 RCP Overview
- Creating a “Hello, World” RCP 4.0 application
- Workbench model
- Toolbar, menu, parts, commands, and handlers
- Dependency injection
- Services
- Look & Feel customization with CSS
- Rendering Engine

Starting Eclipse 4.x Development

There are two options:

- Eclipse 4.x SDK +
  - Eclipse 4.x tooling
- Eclipse 3.x SDK +
  - Eclipse 4.x tooling
  - Eclipse 4.x RCP target platform

For this tutorial we choose
Eclipse 4.x SDK + Eclipse 4.x tooling

Eclipse 4.x SDK

Java - org.eclipse.e4.demo.contacts/src/org/eclipse/e4/demo/contacts/views/DetailComposite.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access | Java

Package Explorer

- org.eclipse.e4.demo.contacts
  - JRE System Library [J2SE-1.5]
  - Plug-in Dependencies
  - src
    - org.eclipse.e4.demo.contacts
      - BundleActivatorImpl.java
    - org.eclipse.e4.demo.contacts.databinding
    - org.eclipse.e4.demo.contacts.handlers
    - org.eclipse.e4.demo.contacts.model
    - org.eclipse.e4.demo.contacts.model.internal
    - org.eclipse.e4.demo.contacts.processors
    - org.eclipse.e4.demo.contacts.views
      - ContactViewerComparator.java
      - DetailComposite.java
      - DetailsView.java
      - ListView.java
  - css
    - blue-gradient.css
    - blue-gradient.png
    - bright-gradient.css
    - bright-gradient.png
    - dark-gradient.css
    - dark-gradient.png
  - icons
  - images
  - META-INF
  - vcards
  - Application.e4xmi
  - build.properties
  - contacts.product
  - plugin_customization.ini
  - plugin.xml
  - splash.bmp
- org.eclipse.e4.demo.contacts.mvp

contacts.product | dark-gradient.css | DetailComposite.java

```
* Copyright (c) 2009 Siemens AG and others.

package org.eclipse.e4.demo.contacts.views;

import java.net.URL;

public class DetailComposite extends Composite {

    private final MDirtyable dirtyable;
    private Contact originalContact;
    private Contact clonedContact;
    private boolean commitChanges = false;

    private Label imageLabel;

    private Image dummyPortrait;
    private boolean generalGroup;

    private final DataBindingContext dbc;
    private final WritableValue contactValue = new WritableValue();
    private final IObservableValue scaledImage;

    @Inject
    public DetailComposite(MDirtyable dirtyable, final Composite parent) {
        super(parent, SWT.NONE);
        this.dirtyable = dirtyable;

        parent.getShell().setBackgroundMode(SWT.INHERIT_DEFAULT);

        dbc = new DataBindingContext();

        URL url = FileLocator.find(Platform
                .getBundle("org.eclipse.e4.demo.contacts"), new Path(
                "images/dummy.png"), null);
        ImageDescriptor imageDescriptor = ImageDescriptor.createFromURL(url);
        if (imageDescriptor != null) {
            dummyPortrait = imageDescriptor.createImage();
        }

        final GridLayout layout = new GridLayout(1, false);
        layout.verticalSpacing = 5;
        setLayout(layout);
```

Outline

- org.eclipse.e4.demo.contacts.views
- import declarations
- DetailComposite
  - dirtyable : MDirtyable
  - originalContact : Contact
  - clonedContact : Contact
  - commitChanges : boolean
  - imageLabel : Label
  - dummyPortrait : Image
  - generalGroup : boolean
  - dbc : DataBindingContext
  - contactValue : WritableValue
  - scaledImage : IObservableValue
  - DetailComposite(MDirtyable, Composite)
    - new ComputedValue() {...}
    - new DisposeListener() {...}
  - setDirty(boolean) : void
  - checkEmptyString(Object) : boolean
  - createSeparator(Composite, String) : void
  - createVerticalSpace(Composite) : void
  - createComposite(Composite) : Composite
  - createText(Composite, String, String) : Text
    - new ModifyListener() {...}
  - getOriginalContact() : Contact
  - getModifiedContact() : Contact
  - setTextEnabled(Composite, boolean) : void
  - update(Contact) : void

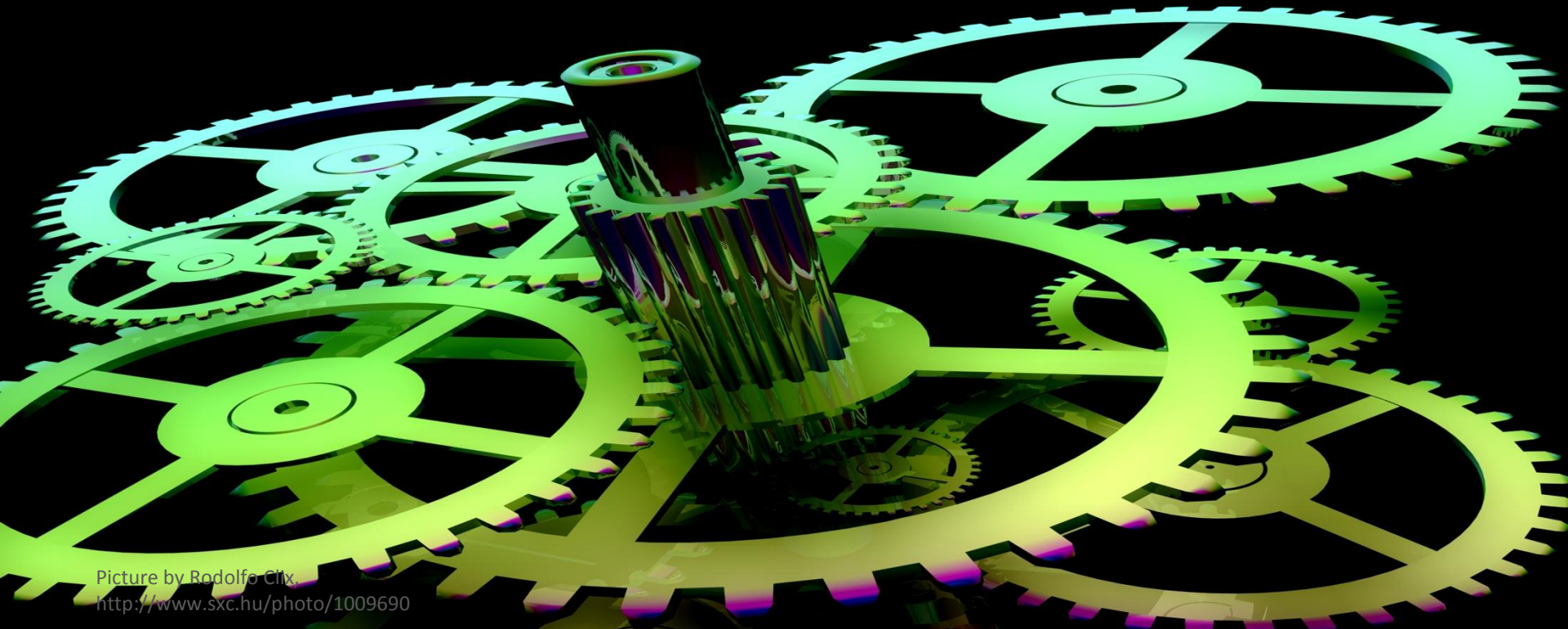
Problems | @ Javadoc | Declaration

Description	Resource	Path	Location	Type
Warnings (100 of 201 items)				

Writable | Smart Insert | 2 : 1

Install the e4 Tooling

- Select Help/Install new Software…
- Work with: e4 0.11 IBuild Updates
- Install E4 Tools/Eclipse e4 Tools (Incubation)



Picture by Rodolfo Clix
http://www.sxc.hu/photo/1009690

Getting Started

- Eclipse 4.0 tooling provides a new wizard to generate an e4 based RCP application
- We start with generation such a “Hello, e4” application
- Later we will create our tutorial application e4 Contacts manually

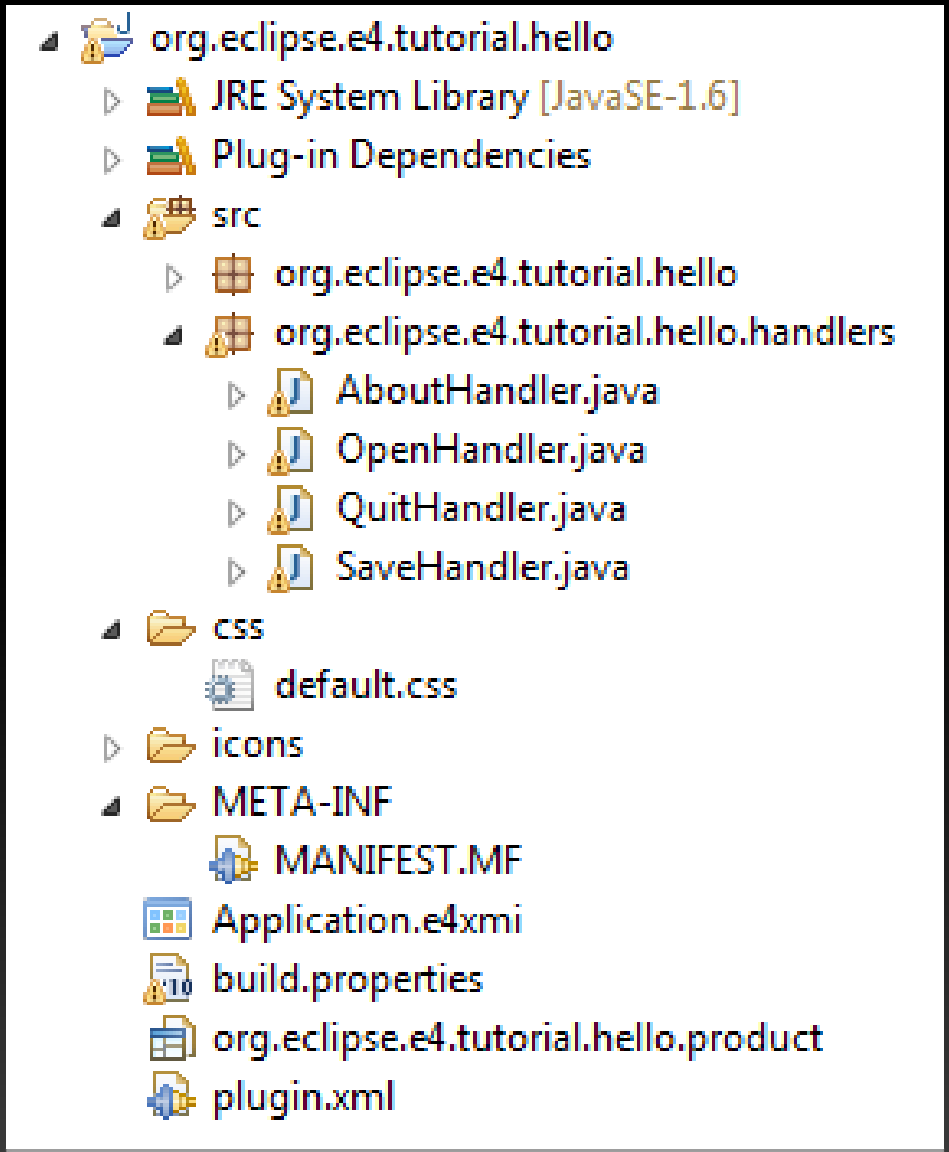
Lab: Generate “Hello, e4”

- Start the SDK
- Select New/Other…/e4/e4 Application Project
- Name the project org.eclipse.e4.tutorial.hello
- Click “Next”
- Change property Name to Hello e4
- Change property Provider to your name
- Click “Finish”

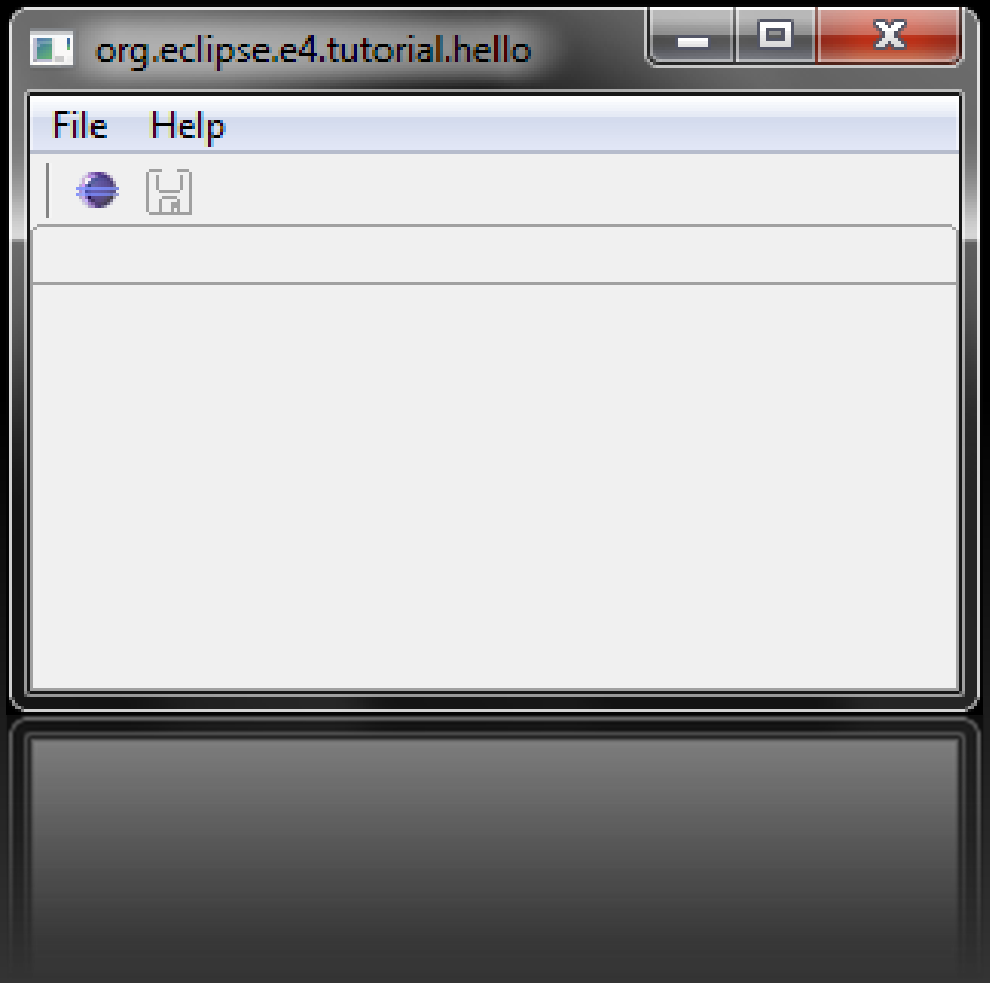
Lab: Launch “Hello, e4”

- Double-Click org.e4-tutorial.hello.product
- Click on “Launch an Eclipse Application” in the Testing section
  - If the launch fails, don’t worry…
  - Open your org.e4-tutorial.hello.product launch configuration, select the Plug-ins tab and add all required plug-ins
  - Launch again, this should work!
- Play around with the application

The generated Project



The generated “Hello, e4” Application



Outline

- Eclipse 4.0 RCP Overview
- Creating a “Hello, World” RCP 4.0 application
- Workbench model
- Toolbar, menu, parts, commands, and handlers
- Dependency injection
- Services

Workbench Model

Picture by miamiamia,
http://www.sxc.hu/photo/1168590

Workbench Model

- Is using EMF (Eclipse Modeling Framework)
- Contains graphical elements, like
  - part stacks, parts, menu bar and toolbar
- Contains non-graphical element, like
  - commands, handlers, key bindings
- Can be modified at runtime
- Can be edited using tools like the e4 WorkbenchModel Editor

Differences with Eclipse 3.x

- No distinction between views and editors
  => Everything is a part
- No need of perspectives
  => Can be used if desired

The e4 WorkbenchModel Editor

org.e4-tutorial.hello | Application.e4xmi

- Application
  - Binding Context - In Dialog and Windows
    - Binding Context - In Windows
    - Binding Context - In Dialogs
  - Addons
  - Handlers
    - Handler - quitCommand
    - Handler - openCommand
    - Handler - saveCommand
    - Handler - aboutCommand
  - Part Descriptors
  - BindingTables
  - Commands
    - Command - quitCommand
    - Command - openCommand
    - Command - saveCommand
    - Command - aboutCommand
  - Command Categories
  - Windows
    - Trimmed Window - org.e4-tutorial.hello
      - Main Menu
        - Handlers
        - Windows
        - Controls
        - Shared Elements
      - TrimBars
    - Menu Contributions
    - Toolbar Contributions
    - Trim Contributions

Application

Id: org.eclipse.e4.ide.application

Root Context ☑

To Be Rendered ☑

Visible ☑

Variables [ ] Add

Up | Down | Remove

Binding Contexts [ ] Add

org.eclipse.ui.contexts.dialogAndWindow

Up | Down | Remove

Workbench Model

- The model is stored in an XMI file
  - Best practice is to name it Application.e4xmi
- The model is contributed through the extension point org.eclipse.core.runtime.products
- In the product tag there is a property
  - with name applicationXMI
  - And value org.eclipse.e4.tutorial.hello/Application.e4xmi

NLS / Externalized Texts

- Similar to 3.x
  - Use %<yourkey> in the model as a placeholder
  - Add „yourkey“ to the bundle‘s resource properties file (default OSGI-INF/l10n/bundle.properties)
- Translation is a pure decoration process that happens at rendering time
  - In future it will be possible to switch the UI language dynamically on the fly
- Use the model tooling to externalize Strings

Addons

- Allow to bring in code that needs access to the DI container e.g. to register code or register with the event bus
- Used to plug in into the event bus to modify the application model on certain events (e.g. double click on a TabItem)
- A standard e4 app needs at least 6 addons
  - CommandServiceAddon, CommandProcessingAddon
  - ContextServiceAddon, ContextProcessingAddon
  - BindingServiceAddon, BindingProcessingAddon

Lab: Manual Application Creation

- Create a Plug-in Project org.eclipse.e4.tutorial.contacts
- Add a Product Extension
- Create a minimal Workbench Model
- Create a Product Configuration
- Launch the application

Lab: Create a Plug-in Project

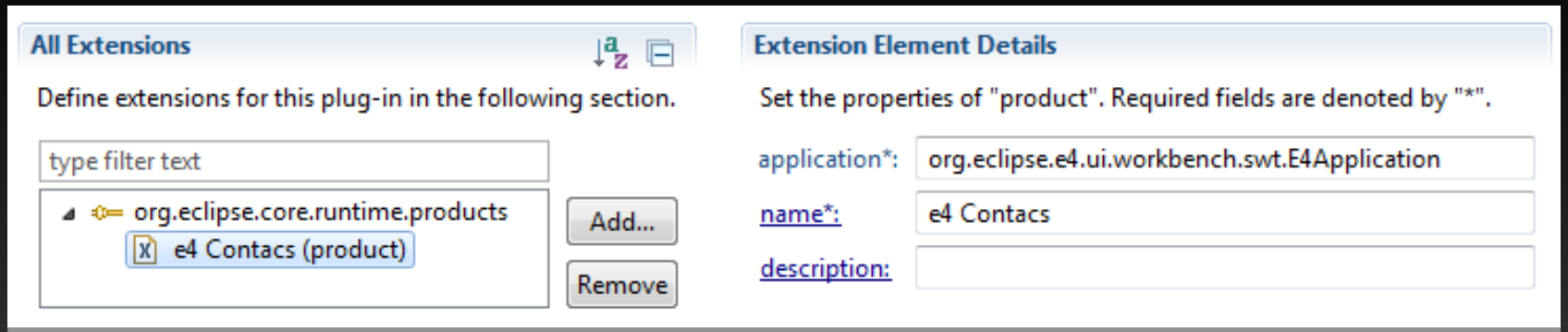
- On the first wizard page
  - Name the project org.eclipse.e4.tutorial.contacts
  - Select Equinox as target platform
- On the second wizard page
  - Change the Name property to e4 Contacts
  - Put your name as Provider
  - Uncheck Generate an Activator
- Click Finish
- Now your project is created

Lab: Create a Product Definition (1)

- Open the file META-INF/MANIFEST.MF
- Click on Extensions in the Overview tab
  - This makes the Extensions tab visible if hidden
- Add a dependency to org.eclipse.equinox.app
- Save (Press CTRL s)
- Select the Extensions tab and add the extension point org.eclipse.core.runtime.products
- Make sure to put product in the ID field

Lab: Create a Product Definition (2)

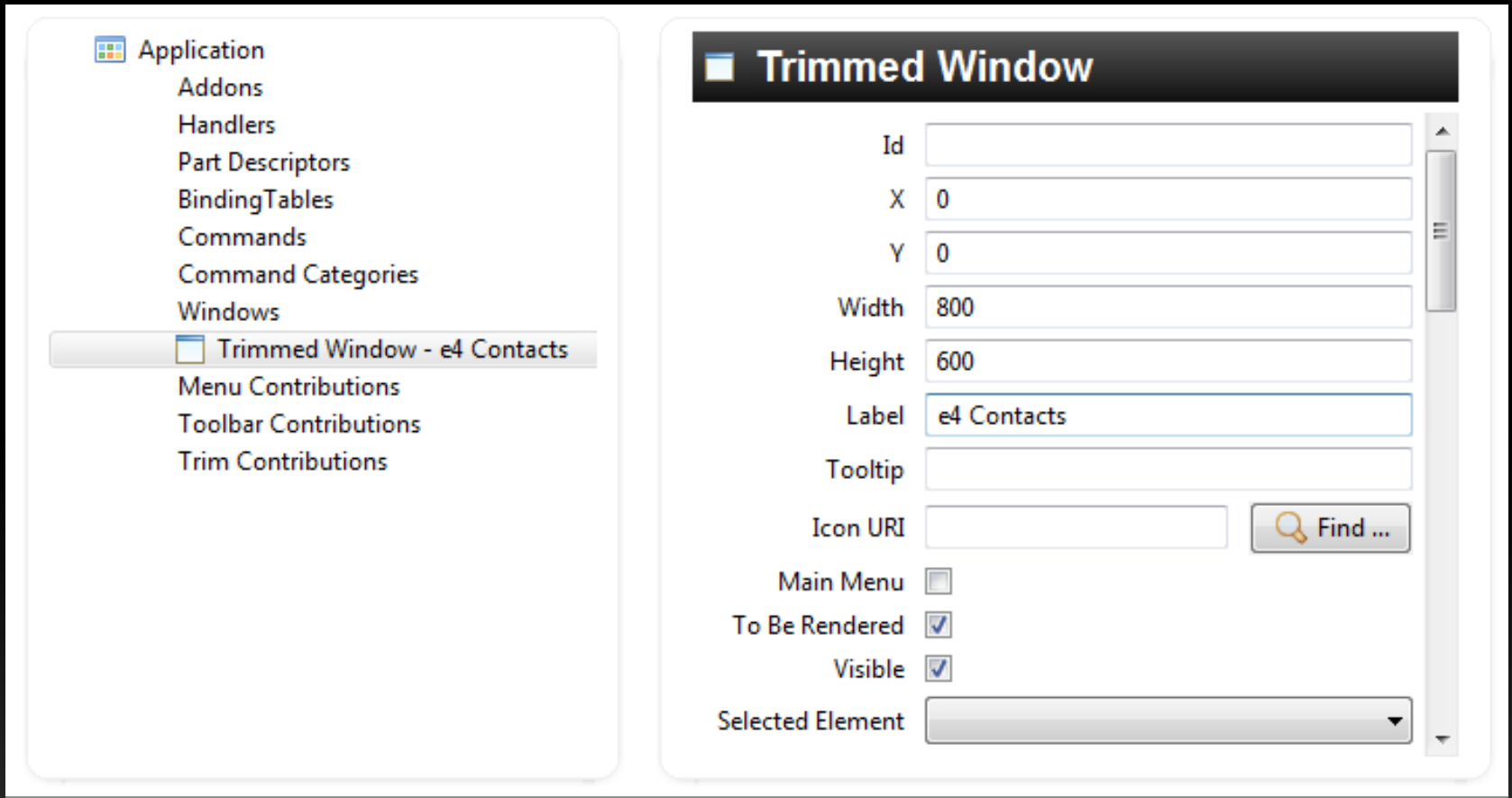
- Add a product to the …products extension
- In the application field, put org.eclipse.e4.ui.workbench.swt.E4Application
- Use e4 Contacts as product name



Lab: Minimal Workbench Model

- Select menu New/Other…/e4/Model/New Application Model
- Put /org.eclipse.e4.tutorial.contacts as Container
- Click Finish, then the Application.e4xmi will be opened in the e4 WorkbenchModel editor
- In the editor, click Window in the left area and select Trimmed Window in the right area
- Click on the icon next to Trimmed Window
- Set Width to 800 ,Height to 600 and Label to e4 Contacts

Lab: Workbench Model Editor



Lab: Resulting Application.e4xmi

```
<?xml version="1.0" encoding="ASCII"?>
<application:Application xmi:version= "2.0"
   xmlns:xmi= "http://www.omg.org/XMI"
   xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
   xmlns:application= "http://www.eclipse.org/ui/2010/UIModel/application"
   xmlns:basic= "http://www.eclipse.org/ui/2010/UIModel/application/ui/basic"
   xmi:id= "_PXE0EJ-qEd-iBNJWVQ-d9Q"
   elementId= "org.eclipse.e4.tutorial.contacts.application" >
   <children
      xsi:type= "basic:TrimmedWindow"
      xmi:id= "_eB6zsJ-xEd-iBNJWVQ-d9Q"
      label= "e4 Contacts"
      width= "800"
      height= "600" />
</application:Application>
```

Lab: Create a Product Configuration (1)

- Create a product configuration
  - Name it contacts
  - Choose Use an existing product: org.eclipse.e4.tutorial.contacts.product
  - Click Finish
- Add dependencies
  - org.eclipse.equinox.ds
  - org.eclipse.equinox.event
  - org.eclipse.e4.ui.workbench.renderers.swt
- Click Add Required Plug-ins and save

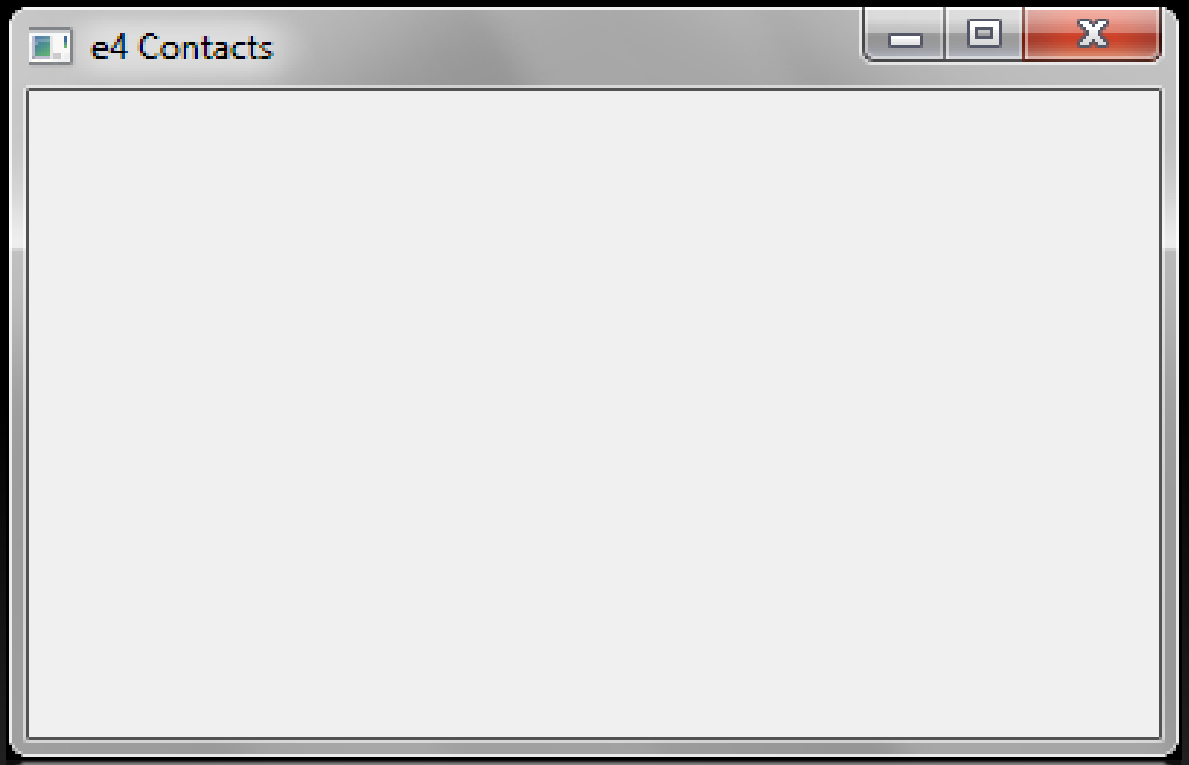
Lab: Create a Product Configuration (2)

- Open plugin.xml in the manifest editor
- Add property to the e4 Contacts product
  - name appName
  - Value e4 Contacts
- Add property to the e4 Contacts product
  - name applicationXMI
  - value org.eclipse.e4.tutorial.contacts/Application.e4xmi

Lab: Resulting plugin.xml

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

Lab: Launch empty e4 Contacts App



Outline

- Eclipse 4.0 RCP Overview
- Creating a “Hello, World” RCP 4.0 application
- Workbench model
- Toolbar, menu, parts, commands, and handlers
- Dependency injection
- Services

Commands and Handlers

Picture from
http://www.sxc.hu/photo/1005737

Command

- Is an abstraction for a generic action
  - Like save, open, etc.
- Has no implementation of a behavior
  - This is done by handlers
- Can be used in toolbars or menus
  - The same command can be customized with specific UI elements, like icon, text, etc.
- Have properties: id, name, description, category and tag
- Can have a key binding

Handler

- Provides an implementation of a behavior
- Can be bound to a Command
- Is a POJO!
- Uses Annotations
  - to declare methods to be executed: @Execute
  - to check, if it can be executed: @CanExecute
- Gets its dependencies injected (DI)

A simple Exit Handler

```
public class ExitHandler {
    @Execute
    public void execute(IWorkbench workbench) {
        workbench.close();
    }
}
```

Injected at Runtime

Lab: Create a Handler

- Add two new dependencies in your project
  - org.eclipse.e4.ui.workbench
  - org.eclipse.e4.core.di
- Create the class ExitHandler in the package org.eclipse.e4.tutorial.contacts.handlers
- Use the @Execute annotation
- Use the interface IWorkbench in the execute method, it will be injected automatically

Lab: Add a Menu

- In the e4 WorkbenchModel editor add a Main Menu to the Trimmed Window
- Give it the id menu:org.eclipse.ui.main.menu
- Expand the Main Menu and select Children
- Add a Menu with Label File
- Add a Direct MenuItem to the Menu
- Choose your ExitHandler the for the Class URI and put Exit (Direct) in the Label field
- Save and launch the e4 contacts product

e4 Contacts with Menu

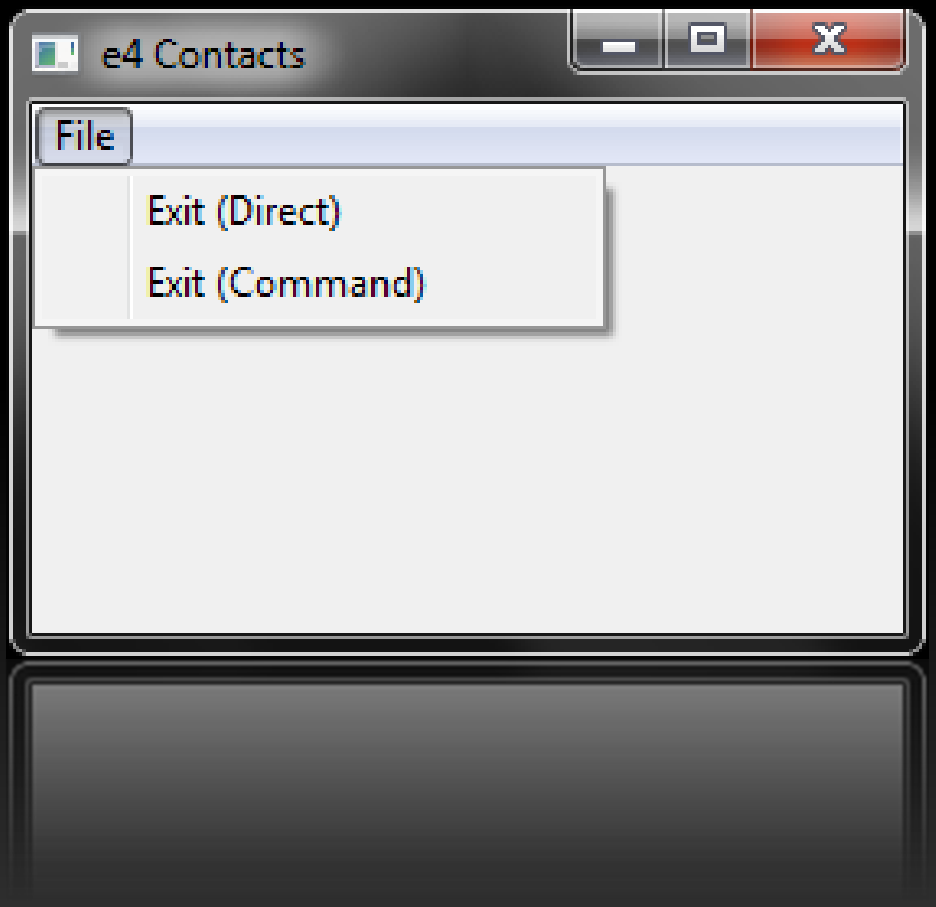
Lab: Use a Command (1)

- In the e4 WorkbenchModel editor select Application/Commands
- Add a Command with
  - Id contacts.exit
  - Name Exit (Command)
- In the e4 WorkbenchModel editor select Application/Handlers
- Add a Handler and use contacts.exit as Command

Lab: Use a Command (2)

- In the e4 WorkbenchModel editor select Trimmed Window/Main Menu/ Children/Menu/Children
- Add a HandledMenuItem to the Menu
- Use contacts.exit as Command
- Save and launch the e4 contacts product

e4 Contacts with two Menu Items



The Domain Model (1)

- We use two domain model interfaces
- An IContact
  - Has attributes like firstName, lastName, email
- An IContactsRepository
  - Provides methods for getting all contacts, adding and removing a contact
- We use OSGi Declarative Services (DS)
  - Loose coupling
  - Implementation can be changed at runtime
  - DI in Workbench Model objects works out of the box

The Domain Model (2)

- Bundle org.eclipse.e4.tutorial.contacts.model contains the two interfaces
- Bundle org.eclipse.e4.tutorial.contacts.model.simple contains a simple implementation and the OSGi declarative service

IContact

```
public interface IContact {
    String getFirstName();
    void setFirstName(String firstName);
    String getLastName();
    void setLastName(String lastName);
    String getEmail();
    void setEmail(String email);

    void addPropertyChangeListener(
                    PropertyChangeListener listener);
    void removePropertyChangeListener(
                    PropertyChangeListener listener);
}
```

Needed for Beans Databinding

IContactsRepository

```
public interface IContactsRepository {
    IObservableList getAllContacts();
    void addContact(IContact contact);
    void removeContact(IContact contact);
}
```

IObservableList can be used with databinding

OSGi Declarative Service

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component
  xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  name="org.eclipse.e4.tutorial.contacts.model.simple.contactsrepository">
  <implementation
     class="org.eclipse.e4.tutorial.contacts.model.simple.ContactsRepository"/>
  <service>
    <provide
     interface="org.eclipse.e4.tutorial.contacts.model.IContactsRepository"/>
  </service>
</scr:component>
```

Parts

Picture from http://www.sxc.hu/photo/1269461

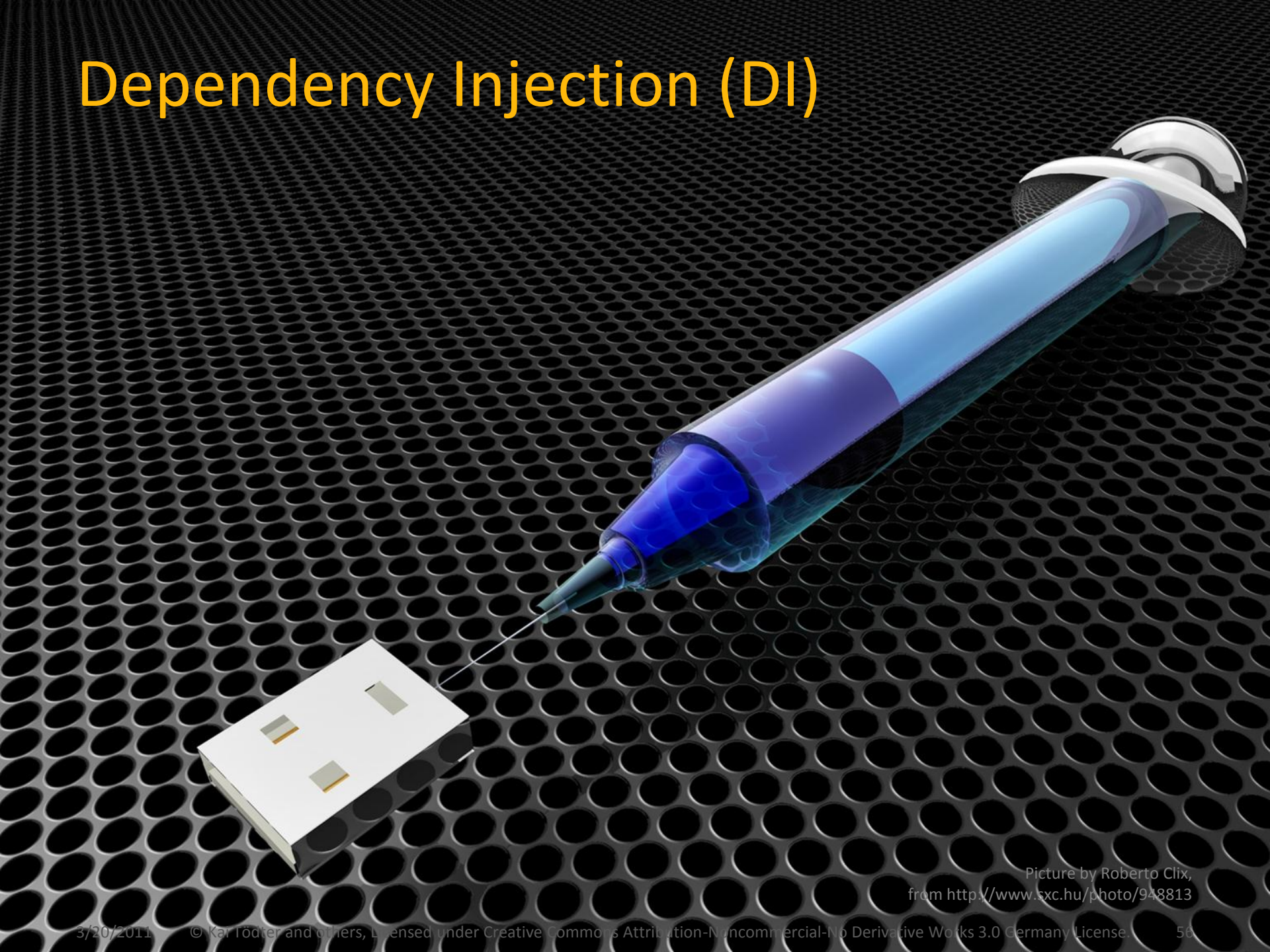
Part

- A Part is a POJO!
- There is no distinction between ViewPart and EditorPart like in Eclipse 3.x
- A Part gets his depended objects through DI
- A Part can use life cycle annotations for its methods
- The UI within a part is implemented with SWT and JFace

Outline

- Eclipse 4.0 RCP Overview
- Creating a “Hello, World” RCP 4.0 application
- Workbench model
- Toolbar, menu, parts, commands, and handlers
- Dependency injection
- Services

Dependency Injection (DI)



Picture by Roberto Clix,
from http://www.sxc.hu/photo/948813

e4 Dependency Injection (1)

- JSR 330 compatible injection implementation
  - @javax.inject.Inject
    - Field, Constructor and Method injection
  - @javax.inject.Named
    - A named qualifier to the context object
    - Default is fully qualified class name of the injected type

e4 Dependency Injection (2)

- e4 specific annotations
  - @Optional
    - Marks a parameter or attribute as optional. If there is no object to be injected at runtime, null is injected instead but the workflow continues
  - @PostConstruct
    - Is invoked after all injections are done and the object is created
  - @PreDestroy
    - Is invoked before the object will be destroyed by the DI container. This is good for cleaning up…

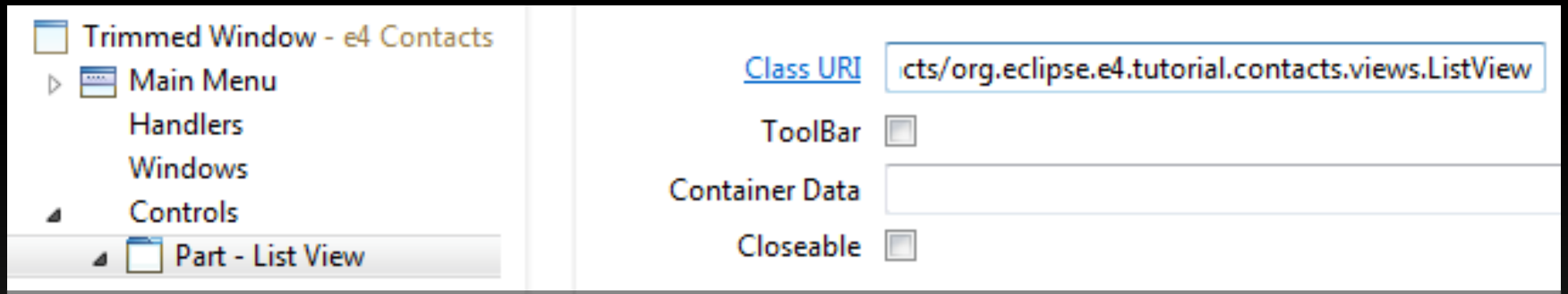
The List View Part

```
public class ListView {

  @Inject
  public ListView(
    Composite parent,
    IContactsRepository contactsRepository) {
      // create UI ...
  }
}
```

Add a Part to the Workbench Model

- Add a Part to the Controls section
- Use a POJO like the List View as Class URI



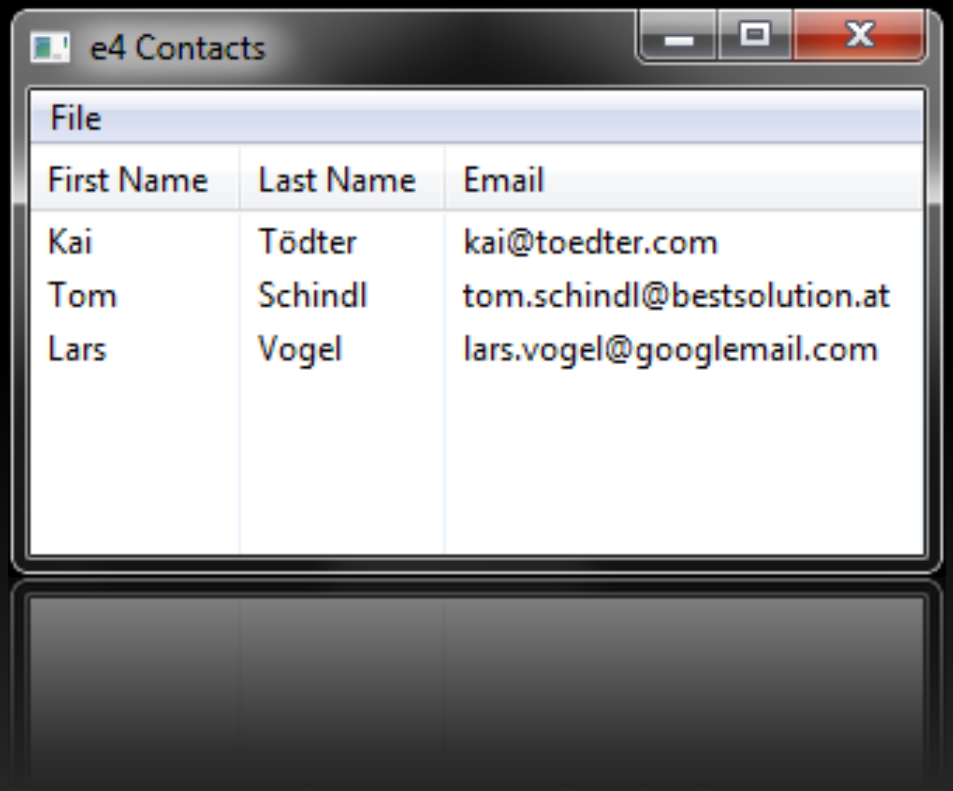
Lab: Add a List View (1)

- Import the projects
  - org.eclipse.e4.tutorial.contacts.model
  - org.eclipse.e4.tutorial.contacts.model.simple
- In your project org.eclipse.e4.tutorial.contacts
  - Add dependencies
    - org.eclipse.swt
    - org.eclipse.jface
    - org.eclipse.jface.databinding
    - javax.inject
    - org.eclipse.core.databinding
    - org.eclipse.core.databinding.beans
    - org.eclipse.e4.tutorial.contacts.model
  - Create a class org.eclipse.e4.tutorial.contacts.views.ListView

Lab: Add a List View (2)

- Copy the snippet "ListView.txt" into the body of your List View class
- Add your List View as Part to the Workbench Model
- Save all, then open your product configuration and add all required plug-ins in the dependencies
- Launch the application

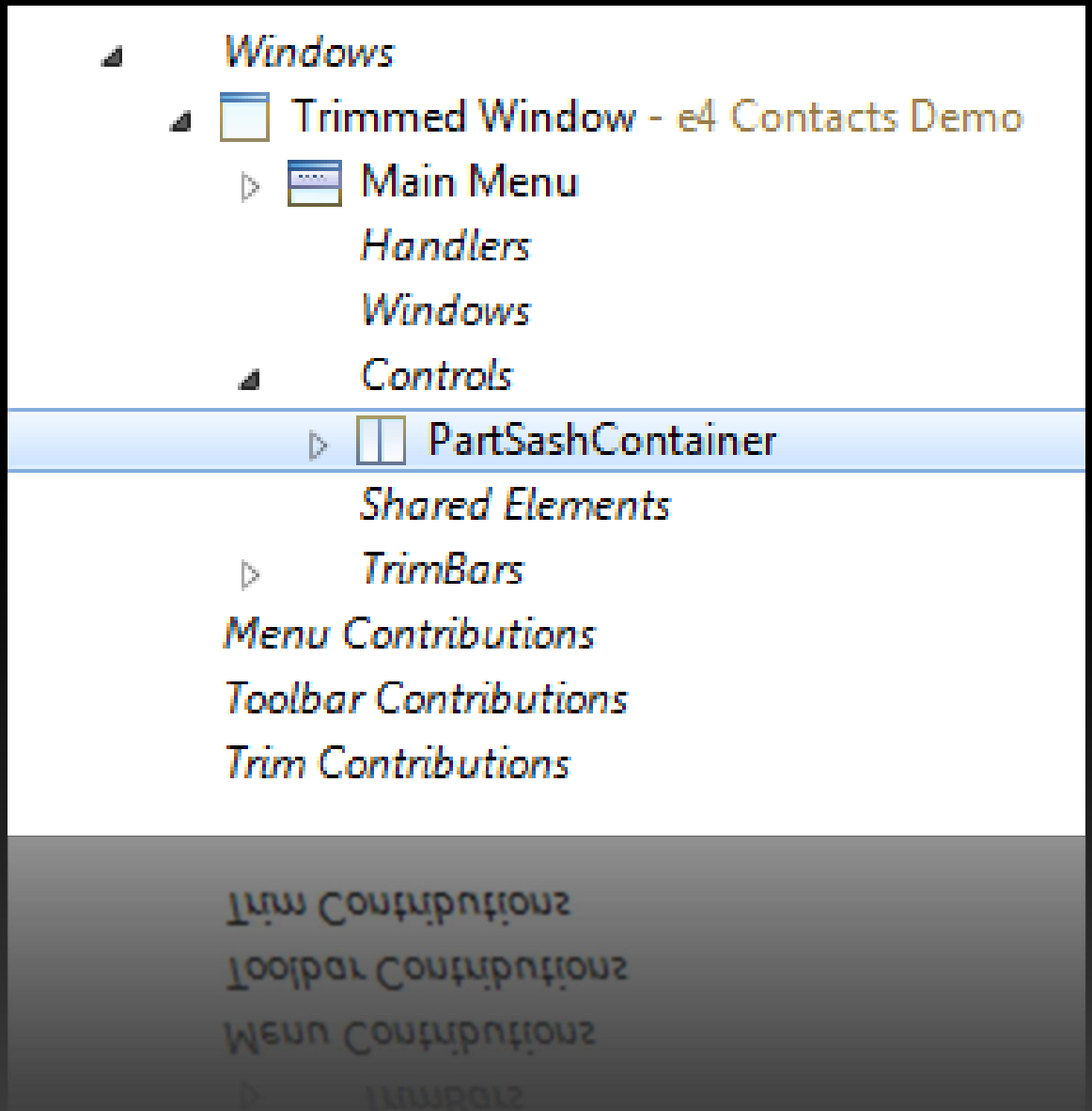
e4 Contacts with List View



Part Stacks and Sashes (1)

- A typical RCP application has more than one Part
- To use a Sash and a Part Stack, just add a PartSashContainer to the Trimmed Window and a Part Stack to the PartSashContainer
- You can use Drag & Drop to move your part into the Part Stack

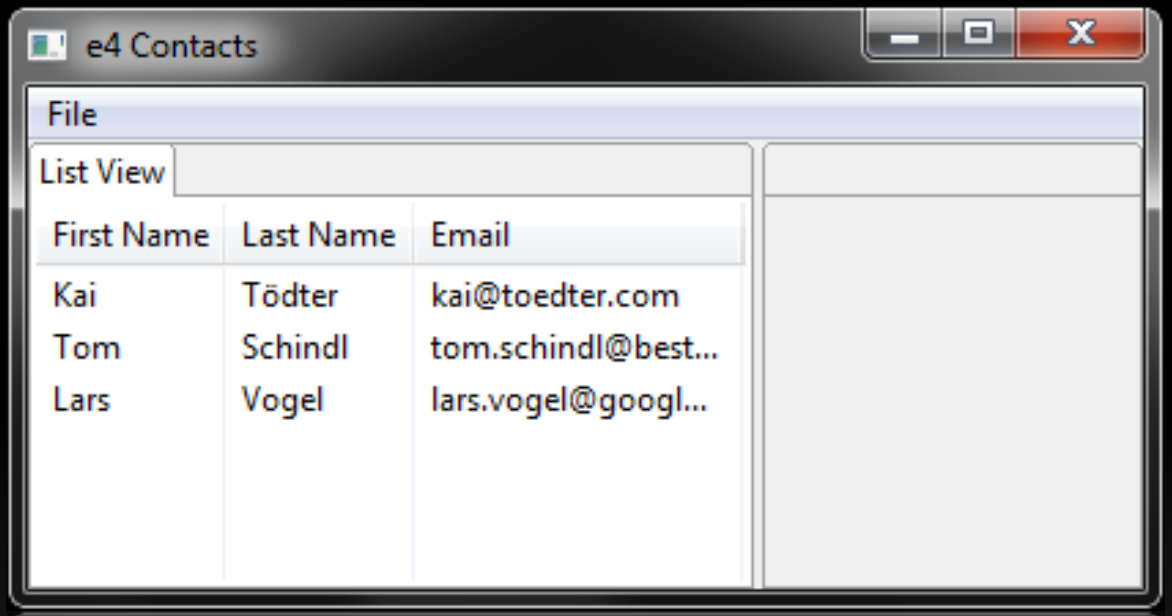
Part Stacks and Sashes in the Editor



Lab: Add Sash and Part Stacks

- Add a PartSashContainer to the Trimmed Window
  - Use Horizontal orientation
- Add two Part Stacks to the Sash
- Move your List View Part to the first Part Stack
- Give the second Part Stack the id org.eclipse.e4.tutorial.contacts.partstacks.second
  - We need that later in the tutorial

e4 Contacts with Part Stacks and Sash



Workbench Model Contributions

- The Workbench Model is dynamic
- Other bundles can make all kinds of contributions
- Contributors just need to know the id of the element they want to contribute to
- There are two kind of contributions
  - Static fragments
  - Dynamic processors

Model Fragments

- In this tutorial, a new bundle wants to contribute a part to the second part stack
- This bundle has to contribute to the extension point org.eclipse.e4.workbench.model
- The extension refers to a new XMI model fragment, stored in the file xmi/fragment.e4xmi

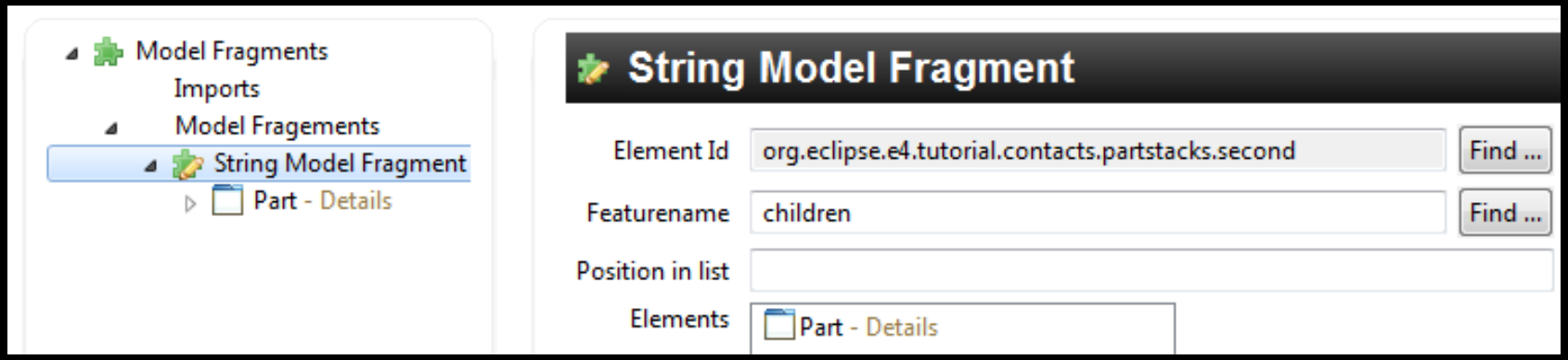
Model Fragment Extension

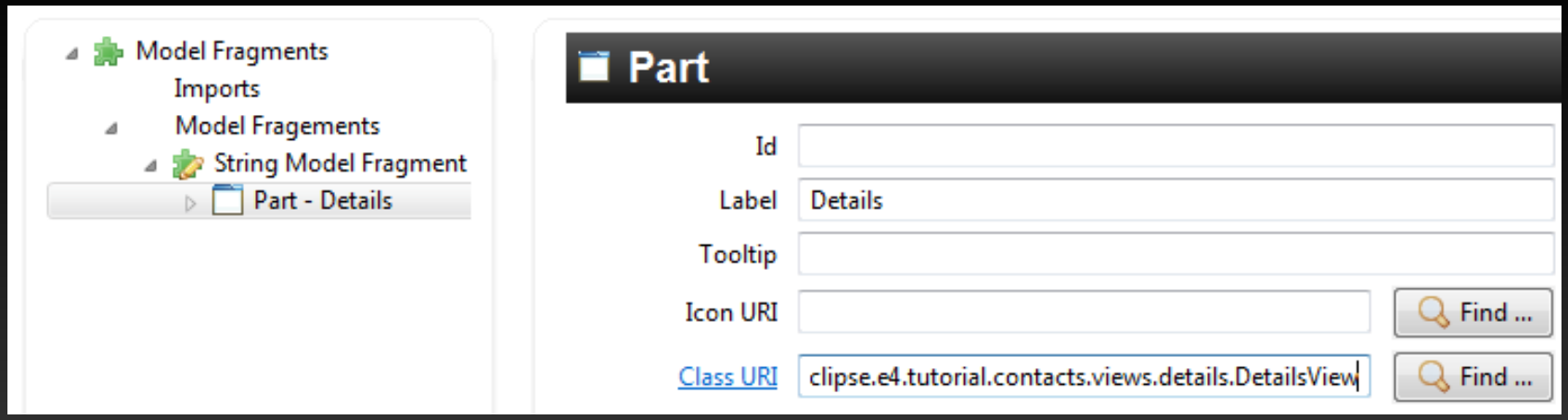
```
<extension
      id="org.eclipse.e4.tutorial.contacts.views.details.fragment"
      point="org.eclipse.e4.workbench.model">
    <fragment
        uri="xmi/fragment.e4xmi">
    </fragment>
</extension>
```

fragment.e4xmi

```
<?xml version="1.0" encoding="ASCII"?>
<fragment:ModelFragments xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:basic="http://www.eclipse.org/ui/2010/UIModel/application/ui/basic"
    xmlns:fragment="http://www.eclipse.org/ui/2010/UIModel/fragment"
    xmi:id="_glJTgHijEd-QLpUkCzu7EA">
  <fragments xsi:type="fragment:StringModelFragment" xmi:id="188321c0-f6a1-
    4502-8073-f8d9c746dfd8" featurename="children"
    parentElementId="org.eclipse.e4.tutorial.contacts.partstacks.second">
    <elements xsi:type="basic:Part"
    xmi:id="89a10afd-b11c-497e-b89f-956997b74293"
    contributionURI="platform:/plugin/org.eclipse.e4.tutorial.contacts.views.detai
    ls/org.eclipse.e4.tutorial.contacts.views.details.DetailsView" label="Details"/>
  </fragments>
</fragment:ModelFragments>
```

fragment.e4xmi in Model Editor





Lab: New Part as Model Fragment (1)

- Create a new bundle org.eclipse.e4.tutorial.contacts.views.details
- Add the following dependencies:
  - javax.inject
  - org.eclipse.swt
  - org.eclipse.core.databinding
  - org.eclipse.core.databinding.beans
  - org.eclipse.jface.databinding
  - org.eclipse.e4.tutorial.contacts.model
  - org.eclipse.e4.core.di
  - org.eclipse.e4.ui.services

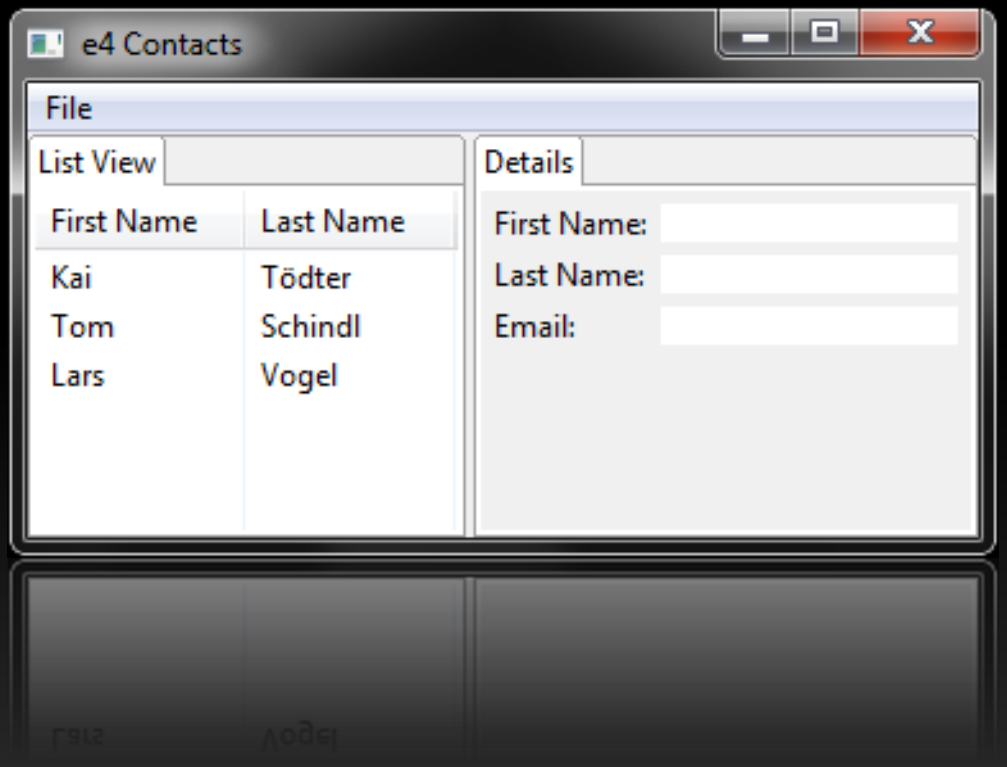
Lab: New Part as Model Fragment (2)

- Copy the file DetailsView.java into the src folder of the new project
- Create a new folder xmi in the new project
- Create a New Model Fragment in the xmi folder
- Add the DetailsView as Part to a new String Model Fragment with id org.eclipse.e4.tutorial.contacts.partstacks.second

Lab: New Part as Model Fragment (3)

- Create the extension for org.eclipse.e4.workbench.model
  - Tip: Uncheck “Show only extensions…”
- Remove email from your List View
- Add the bundle org.eclipse.e4.tutorial.contacts.views.details to your run configuration
- Save and launch

e4 Contacts with new Model Fragment



Outline

- Eclipse 4.0 RCP Overview
- Creating a “Hello, World” RCP 4.0 application
- Workbench model
- Toolbar, menu, parts, commands, and handlers
- Dependency injection
- Services

Services

Picture from
http://www.sxc.hu/photo/157966

e4 Services (aka “The 20 Things”)

e4 provides access to many services.

These services are divided in sections:

- Core
- User Interface
- Advanced
- Domain-Specific

Most of these services are injected using DI, see also:

http://wiki.eclipse.org/E4/Eclipse_Application_Services

e4 Services

- Editor lifecycle
- Receiving input
- Selection
- Standard dialogs
- Persisting UI state
- Logging
- Interface to help system
- Menu contributions
- Authentication
- Authorization
- Long-running operations
- Progress reporting
- Error handling
- Navigation model
- Resource management
- Status line
- Drag and drop
- Undo/Redo
- Accessing preferences

The above list is not complete…

Example: Selection Provider

```
@Inject
private ESelectionService selectionService;

...

tableViewer
  .addSelectionChangedListener(new ISelectionChangedListener() {
    public void selectionChanged(SelectionChangedEvent event) {
      IStructuredSelection selection =
        (IStructuredSelection) event.getSelection();
        selectionService.setSelection(selection.getFirstElement());
    }
});
```

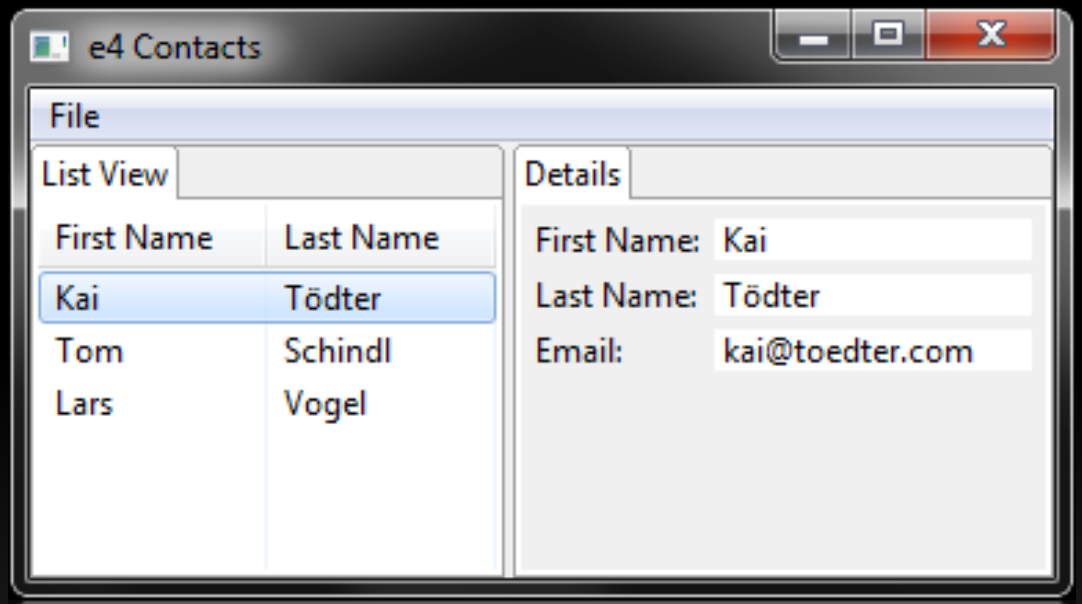
Example: Selection User

```
@Inject
public void setSelection(
    @Optional
    @Named(IServiceConstants.ACTIVE_SELECTION)
    IContact contact) {
        if (contact != null) {
        ...
```

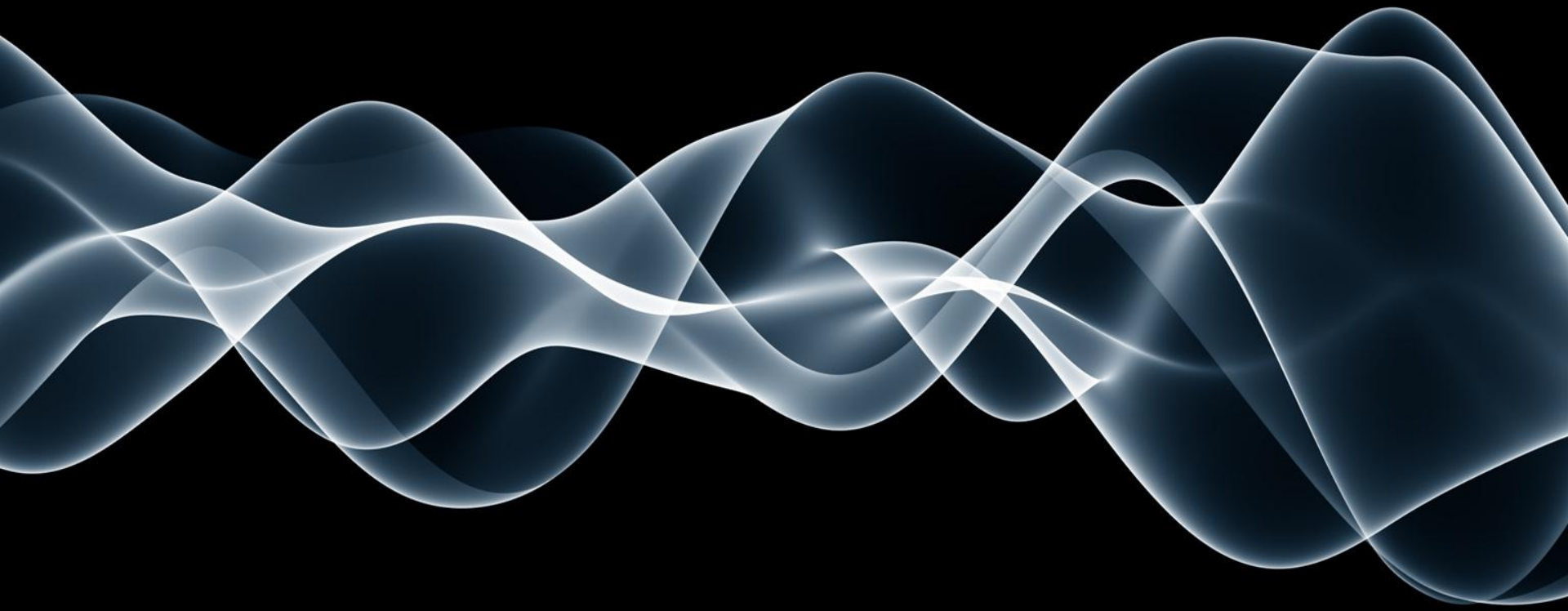
Lab: Add Selection

- React on selections in the ListView and propagate them to the selection service
- Make the DetailsView reacting on the active selection
- Launch and check if the selection mechanism is working properly

e4 Contacts with Selection



Rendering



Picture from
http://www.sxc.hu/photo/1263022

Workbench Model and Rendering

- The Workbench model has no dependencies to a specific UI toolkit
- During startup, the app context is asks for an IPresentationEngine service
- The default is an SWT based presentation engine
- The presentation engine asks a registered RendererFactory for Renderers

- => As a proof of concept, It would be possible to implement a Swing based presentation engine

Tasks of the Renderer

- Manages lifecycle of the UI element
  - Creation
  - Model to widget binding
  - Rendering
  - Disposal

RendererFactory Example

```
public class WorkbenchRendererFactory implements IRendererFactory {

public AbstractPartRenderer getRenderer(MUIElement uiElement,
                                        Object parent) {
    if (uiElement instanceof MPart) {
        if (contributedPartRenderer == null) {
            contributedPartRenderer = new ContributedPartRenderer();
            initRenderer(contributedPartRenderer);
        }
        return contributedPartRenderer;
    }

//...
```

Multiple Renderers

One model element (e.g. a Part Stack) could have different renderers

Custom Renderer Factories

```
public class RendererFactory extends WorkbenchRendererFactory {

    @Override
    public AbstractPartRenderer getRenderer(MUIElement uiElement,
                                            Object parent) {

        if (uiElement instanceof MPartStack && usePShelfRenderer() ) {
            if( stackRenderer == null ) {
                stackRenderer = new PShelfStackRenderer();
                initRenderer(stackRenderer);
            }
            return stackRenderer;
        }

        return super.getRenderer(uiElement, parent);
    }
}
```

RendererFactory Registration

- Add a property to your product extension
- name = "rendererFactoryUri"
- value = "<URI to your class>"
  - E.g. "platform:/plugin/org.eclipse.e4.tutorial.contacts.renderer/org.eclipse.e4.tutorial.contacts.renderer.RendererFactory"

Lab: Use a custom RendererFactory

- Create a new class RendererFactory that extends WorkbenchRendererFactory
- Implement getRenderer(MUIElement uiElement, Object parent)
- Print out the class of the MUIElement
- Return super.getRenderer()…
- Register the RendererFactory as property in your product
- Save and launch

Discussion

Picture from
http://www.sxc.hu/photo/922004

License & Acknowledgements

- This work is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 Germany License
  - See http://creativecommons.org/licenses/by-nc-nd/3.0/de/deed.en_US

- Parts of the material are based on work of Tom Schindl (http://www.bestsolution.at) and Lars Vogel (www.vogella.de)

Picture Index

Many thanks to the authors of the following pictures:

- Slide “e4 Objectives”: http://www.sxc.hu/photo/1081630
- Slide “Install the e4 Tooling”: http://www.sxc.hu/photo/1009690
- Slide “Workbench Model”: http://www.sxc.hu/photo/1168590
- Slide “Commands and Handlers: http://www.sxc.hu/photo/1005737
- Slide “Parts”: http://www.sxc.hu/photo/1269461
- Slide “Dependency Injection (DI)”: http://www.sxc.hu/photo/948813
- Slide “Services”: http://www.sxc.hu/photo/157966
- Slide “UI Styling”: http://www.sxc.hu/photo/1089931
- Slide “Dynamic Theme Switching”: http://www.sxc.hu/photo/823108
- Slide “Rendering”: http://www.sxc.hu/photo/1263022
- Slide “Discussion”: http://www.sxc.hu/photo/922004