Eclipse RCP Development

Virtual Class

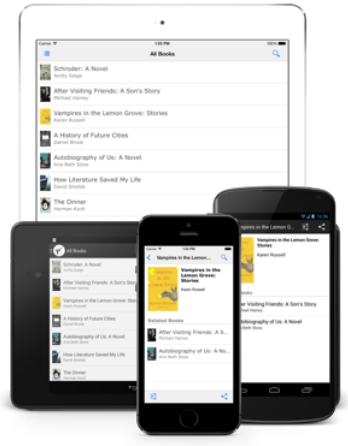
December 9–10, 2014

Elias Volanakis

About Elias

Elias Volanakis (elias@eclipsesource.com)

- Technical Lead, EclipseSource
- Based in Hamburg, Germany
- 11+ years of Eclipse Development
- Eclipse RAP committer
- developer.eclipsesource.com/tabris
- Workshops, seminars, consulting



EclipseSource

- Leadership and committer roles in many key Eclipse projects: Equinox p2, RAP, RTP, EPP, EMFStore, EMF Client Platform ...
- Provides services and solutions for Eclipse adopters

Training by EclipseSource

- To learn more about various Eclipse topics, check our courses:
  - Eclipse RCP Advanced Topics (3 Days)
  - Introduction to Eclipse 4 (2 Days)
  - Eclipse Modeling Basics (2 Days)
  - RAP for Java Developers (4 Days)
  - RAP for RCP Developers (2 Days)
  - Equinox and OSGi (3 Days)
- See http://eclipsesource.com/en/services/training/

Note: We are offering these trainings online too!

Training philosophy

- Theory and examples go together
- This course talks about basic concepts and patterns
- Almost everything is learned by practice: "learning by doing"
  - Extensive sample solution code to compare with
- Approximately 50% theory, 50% practice
- Learn to work effectively with the Eclipse IDE (shortcuts, tips & tricks)
- **There are no stupid questions!**

Who is everybody?

- Java experience?
- GUI programming with SWT or other Widget toolkits?
- Experience with Eclipse as IDE?
- Experience with Eclipse as a platform for rich client applications?
- Your goals for this course
- Other technical background?

Outline

Day 1

- Introduction to RCP
- Applications & Workbench
- Views & Perspectives
- JFace Viewers
- Actions

Day 2

- Editors
- Branding
- Packaging
- SWT Widgets
- SWT Layouts
- Commands

Introduction to Eclipse RCP

What is Eclipse RCP?

- **Eclipse** is an open source community of people building Java based tools and frameworks
- Best known output is the Eclipse Java IDE
- Underneath the IDE is a generic tooling platform
  - Supports a wide range of different language IDEs
  - Tools for data manipulation and reporting build on it
- Under the tooling platform is the **Eclipse Rich Client Platform**
  - A generic platform for running applications

Why Eclipse RCP?

- Robust **Component** Model
  - Plug-ins offer i.e. versioning and shared installation
- Rich **middleware and infrastructure** eases application development
  - i.e. flexible and scalable UI, extensible application, help support, context-sensitive help, network updates etc.
- **Native user experience** with a rich, comfortable UI that integrates into the Operation System
- Easy **portability** with Java and SWT
- Rich clients can run **disconnected**
- Eclipse offers a really good **development tooling support**

Eclipse RCP Over the Years

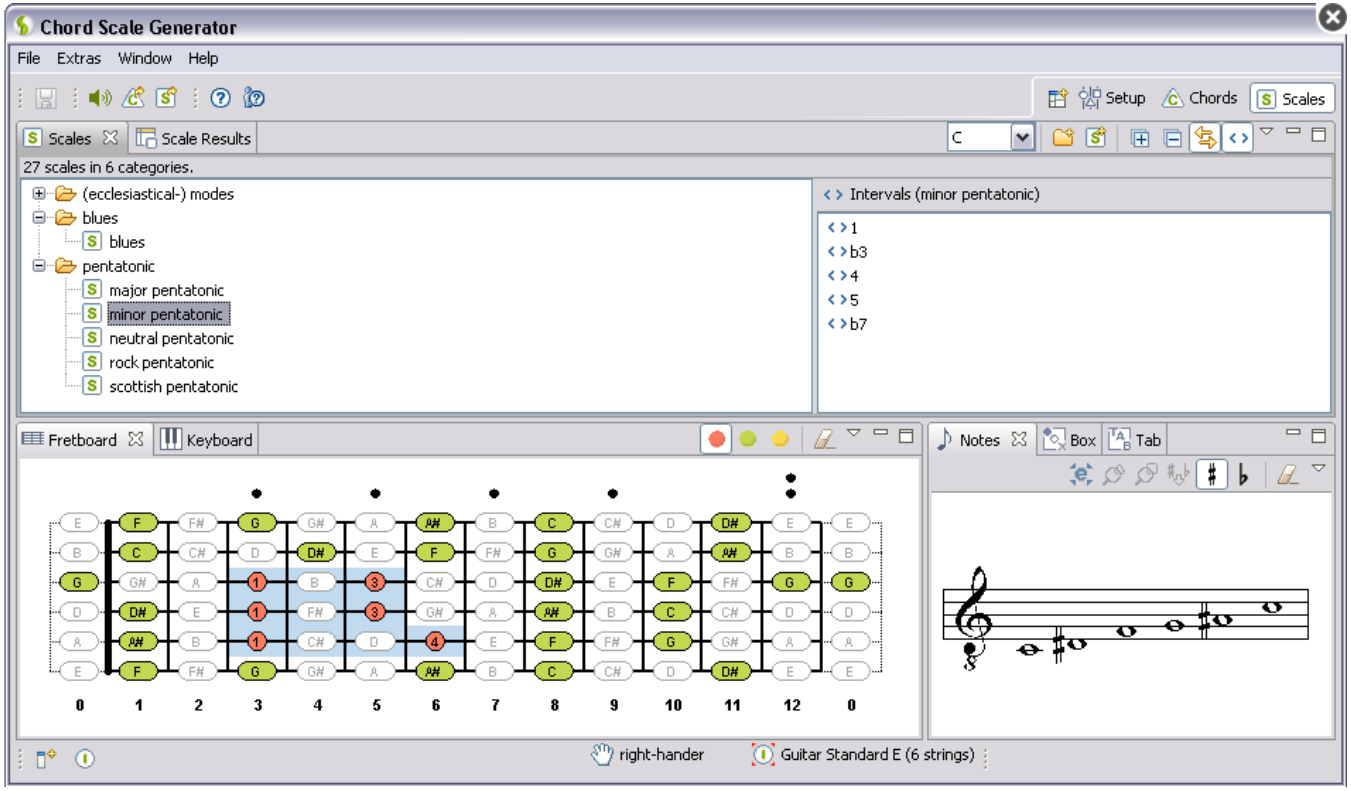
- Originally Eclipse was just a tooling platform with an especially good Java IDE on top of it
- With Eclipse 2.1, Eclipse RCP started as hacker activity
  - Good tooling made it attractive to write more generic applications
  - Still many assumptions made this a challenge
- Eclipse 3.0 was the first version to come with an explicit RCP
  - The IDE-related interdependencies where eliminated
  - OSGi-based runtime enabled dynamic plug-in installation, removal and update
- Commercial applications began to emerge
  - IBM introduced its Workplace™ products
  - NASA uses Eclipse RCP for managing, modeling and analysing space missions

Examples

- Many Open-Source and Commercial applications have been build on Eclipse RCP
- A selection
  - Chord Scale Generator - Chord Scale Generator, award winner as best commercial RCP application 2009
  - NeTS - Railway track planning, nominated for best commercial RCP application 2009
  - My Tourbook - Visualize GPS data, Open-Source
  - ZDT - Help with learning chinese, Open-Source
  - NASA JPL - Maestro Mission Control (Mars Rover), proprietary RCP app
- More: http://www.eclipse.org/community/rcp.php

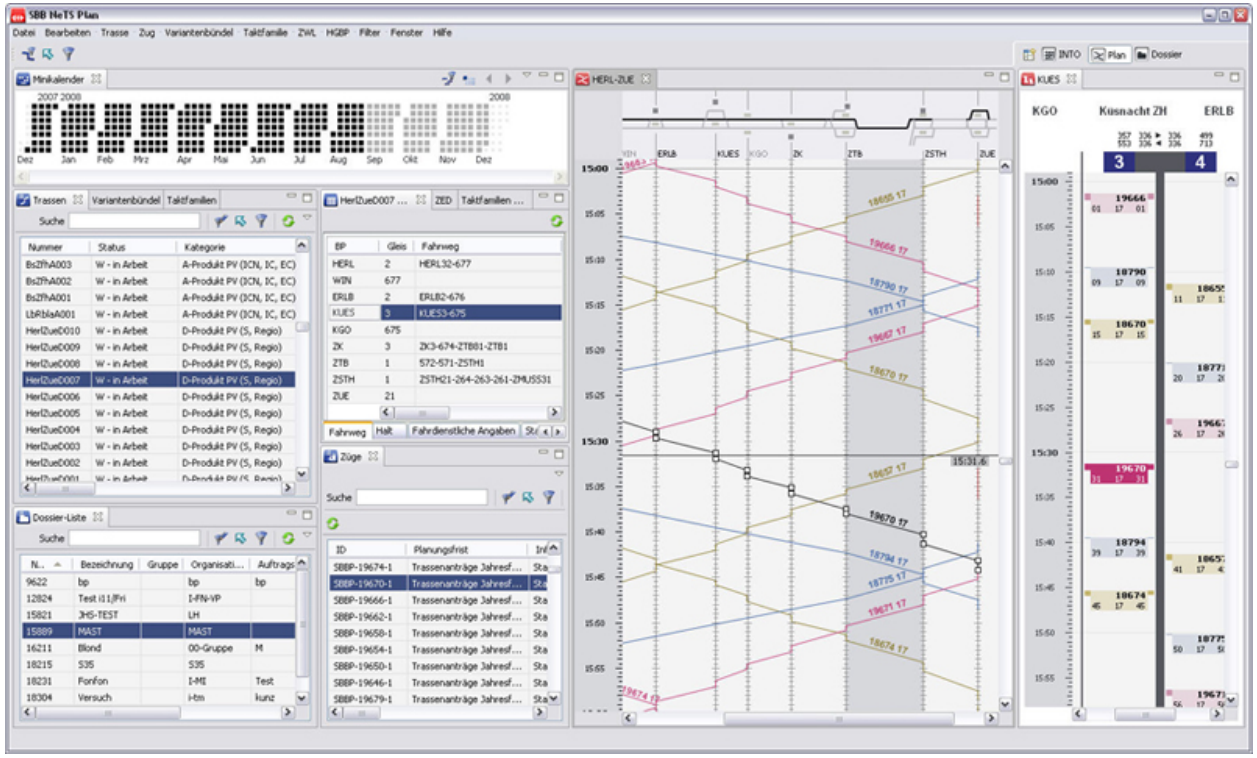
Chord Scale Generator

- Chord Scale Generator is an educational software for plucked and stringed instruments
  - Supports all stringed instruments with 2 to 12 strings
- helps you to find all possible fingerings for a certain chord



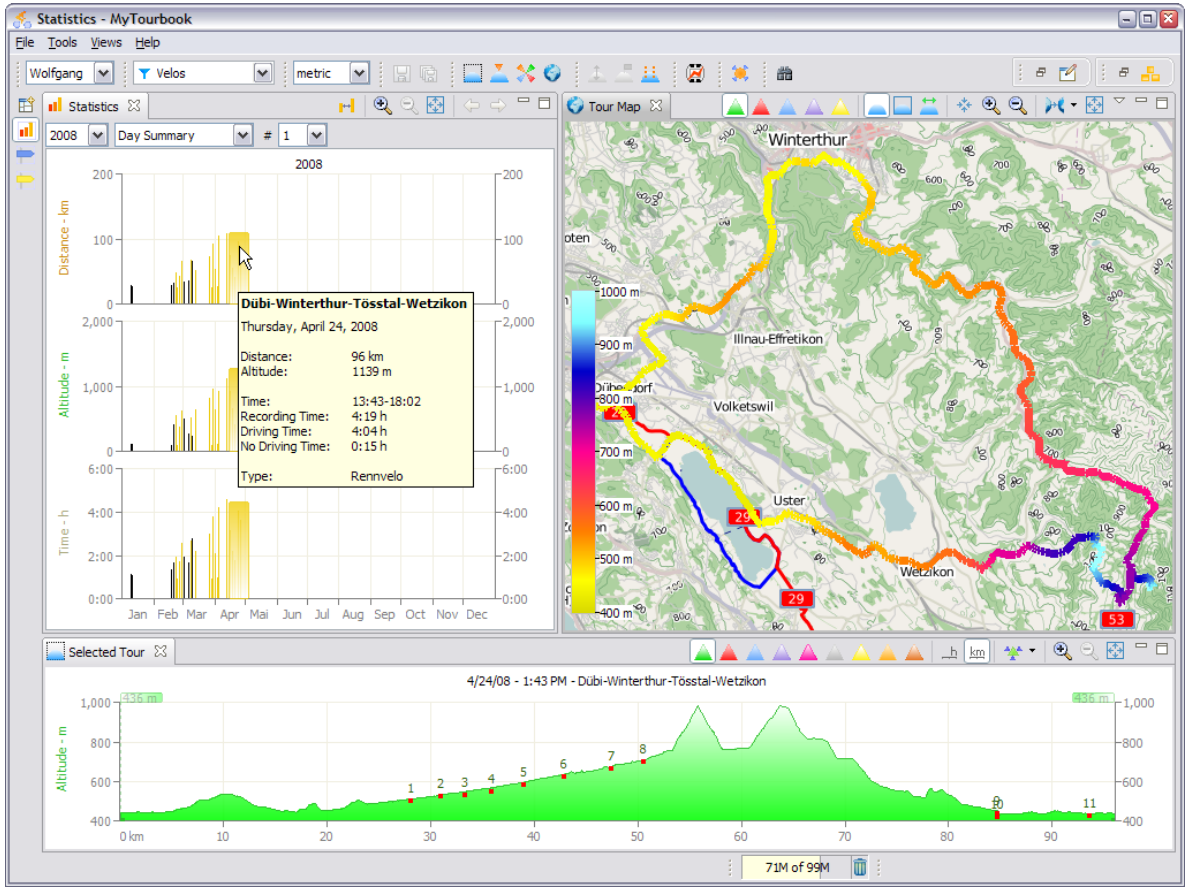
NeTS

- Track Management System for planning tracks and trains
- Table-based and graphical tools for the planning phase
- Supports the planning phase and the collaboration between different experts



My Tourbook

- MyTourbook visualizes and analyzes tours which are recorded by GPS devices
- Tours can be displayed in a map
- Summarizes the data and shows the performance by day/ week/month/year

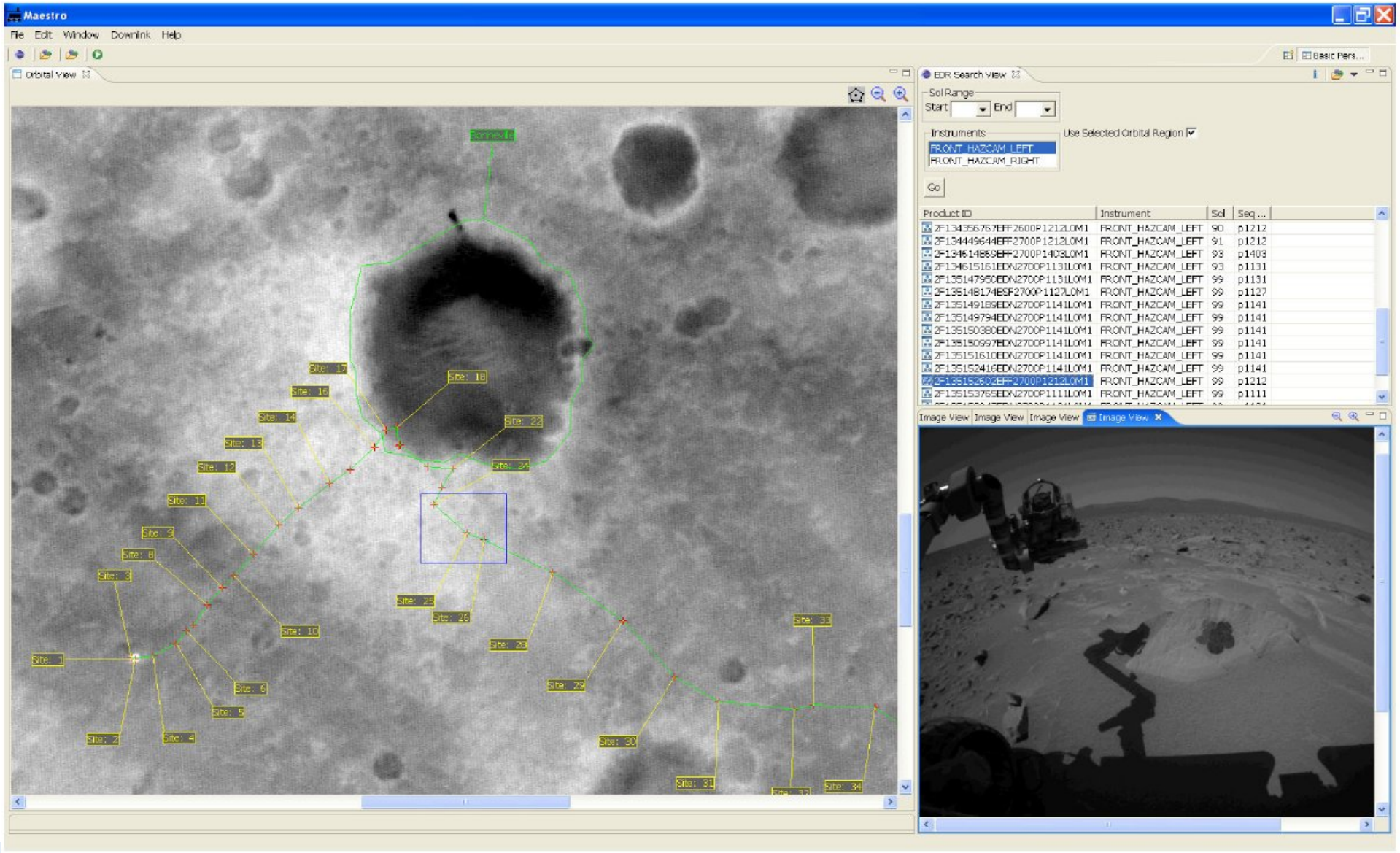


ZDT (Zhongwen Development Tool)

- ZDT is multi-platform application aimed at helping people learn Chinese
- It features
  - a built-in Chinese-English dictionary (based on the CEDICT project),
  - an annotation tool for obtaining popup translations of Chinese text,
  - an integrated flashcard system.

- A sound plug-in is available, which allows hearing characters pronounced by a native Chinese speaker in any part of the application.

NASA JPL - Maestro Mission Control

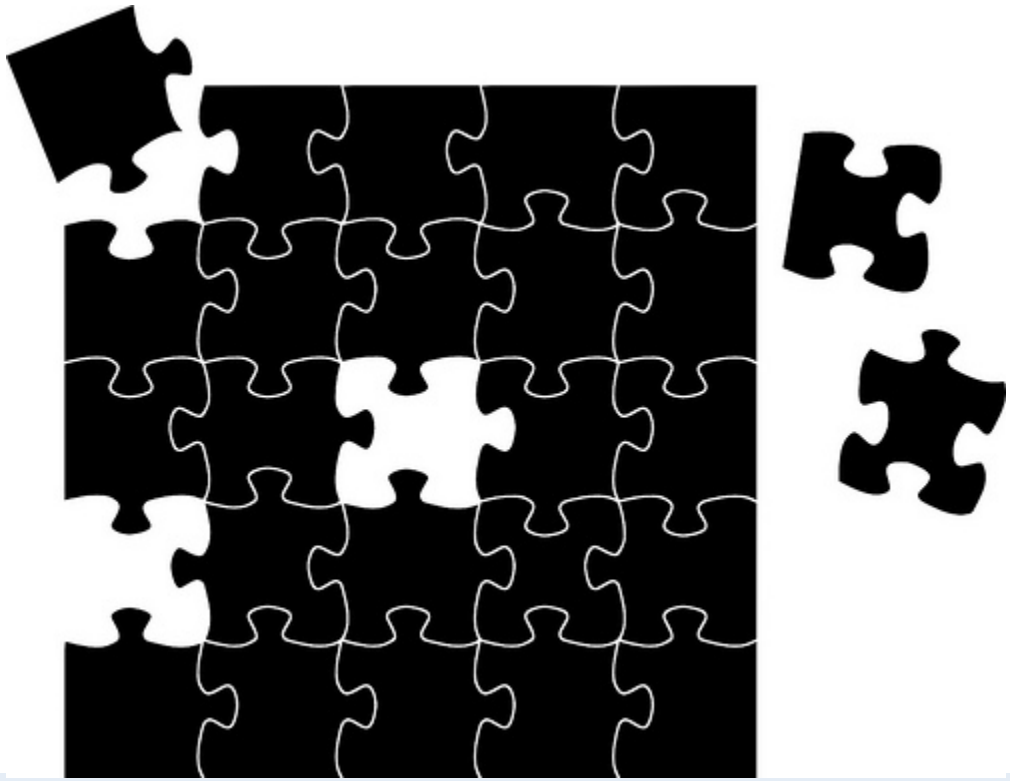


Eclipse RCP Concepts

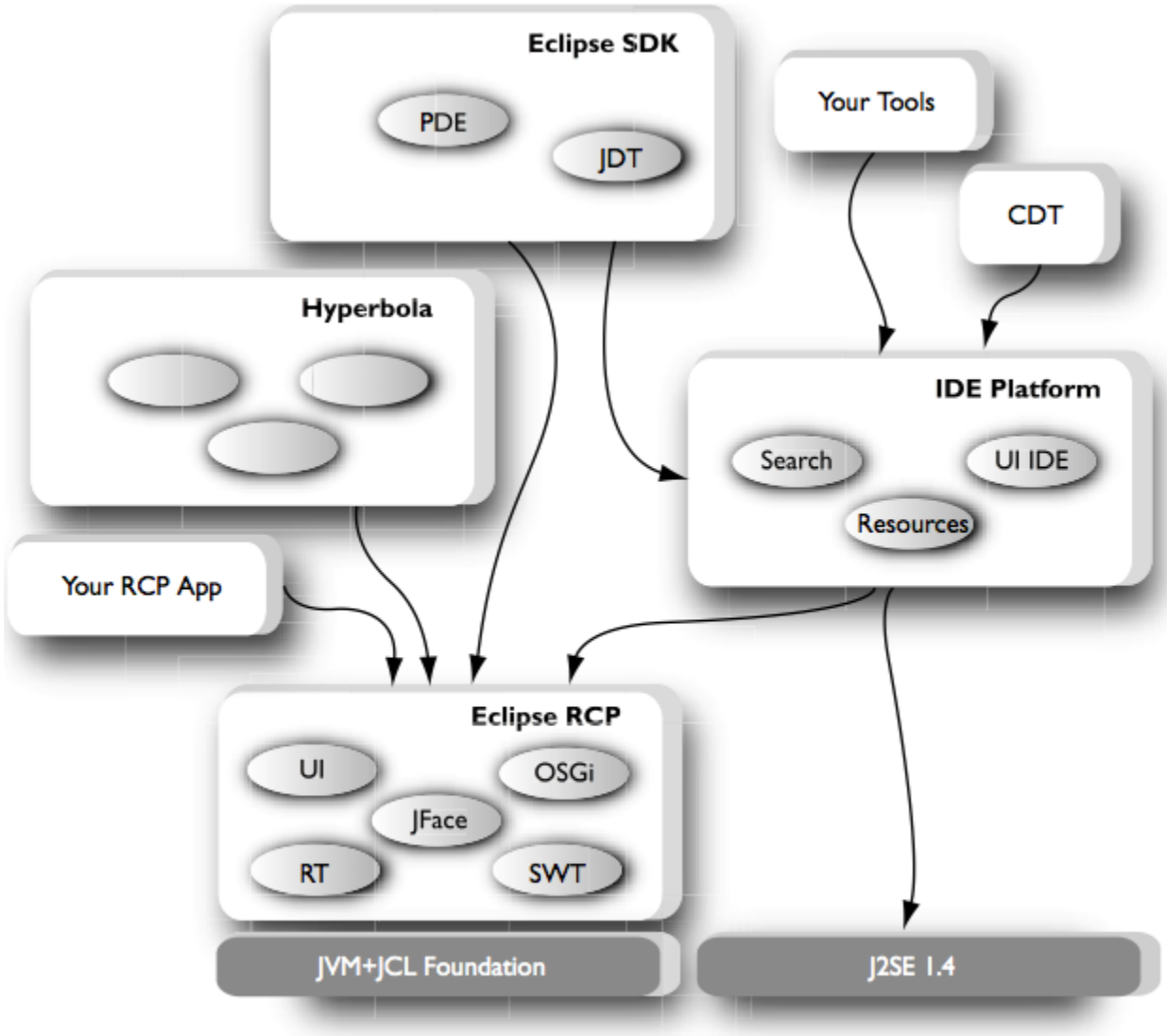
- Eclipse contains quite a number of different concepts
- A few of them are essential to the *Eclipse-ness*
- Let's have a look at Eclipse from 10.000ft for a start
- Then have a look at RCP from 1.000ft
- And then we look around

RCP is community of plug-ins

- In RCP everything is a plug-in
- An application is: your plugins and the RCP runtime
- You are free to slice and dice them
- RCP Development is about plug-ins



10.000ft Architecture

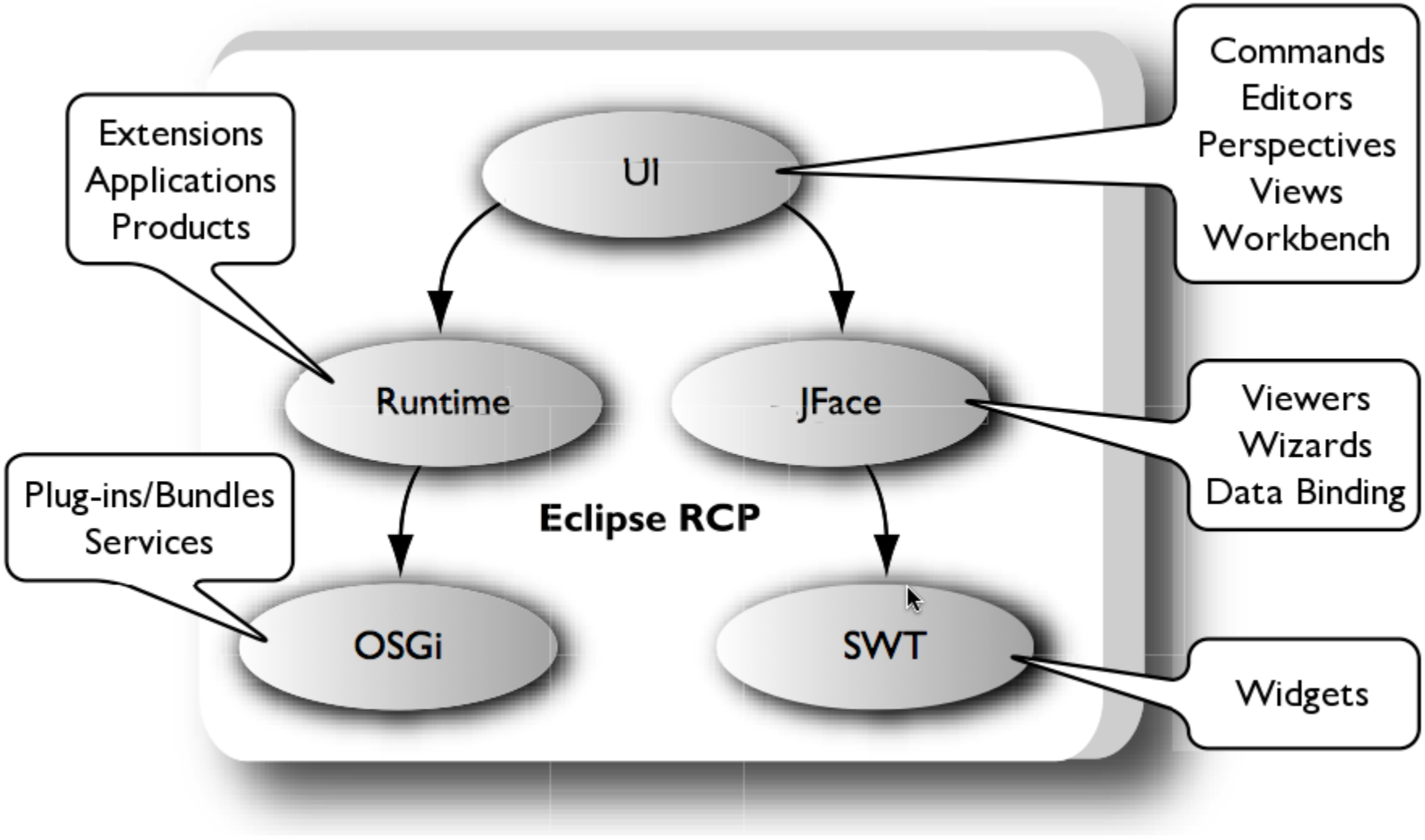


Notes

- The Eclipse projects release tons of plug-ins
- The grouping abstraction is needed to hide detail
- From that view point, the Eclipse IDE itself is just another RCP Application
- RCP is another platform like a basic OS or the Java JRE itself

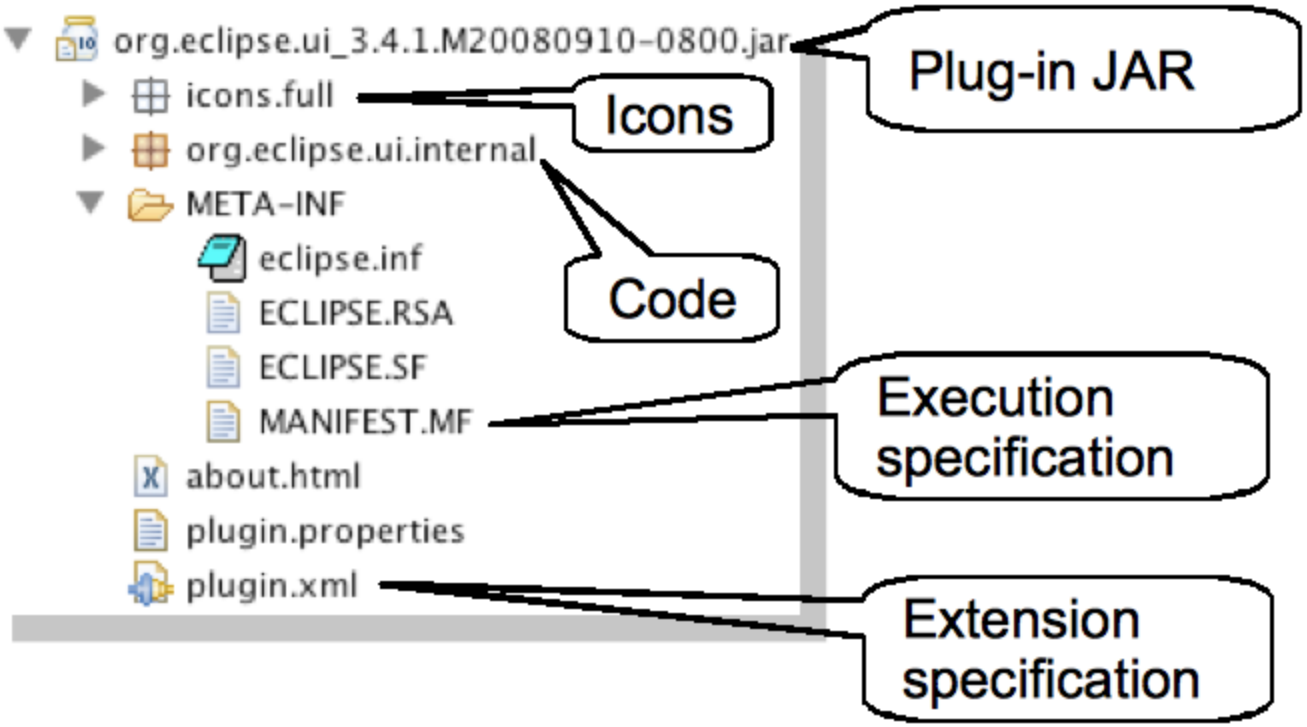
Note: The Eclipse RCP requires only Foundation Java classes. Foundation is a J2ME standard class set typically meant for embedded or smaller environments. See http://java.sun.com/products/foundation for more details. If you are careful to use only a Foundation-supported API, then you can ship Eclipse-based applications on a Java Runtime that is only about 6MB rather than the 40MB J2SE 1.4 JRE.

1.000ft Architecture



100ft: Inside plug-ins

- On the disk the plug-in is a jar file
- As a JAR it contains a `MANIFEST.MF`
- Plug-ins can contain ready-only content
  - Code
  - Images, Web pages, Documentation, ...

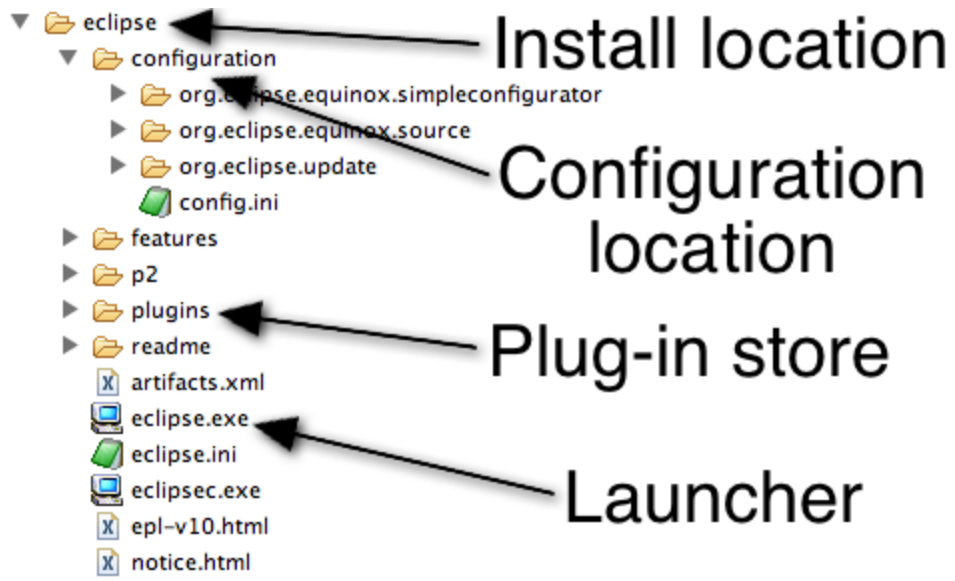


- `plugin.xml` contains extension and extension point declarations
  - Historically, the information that is now in `MANIFEST.MF` was stored there

Plug-ins can also be stored in directories if they need this physical layout. However, the preferred way is to store them as jar files.

Putting a System Together

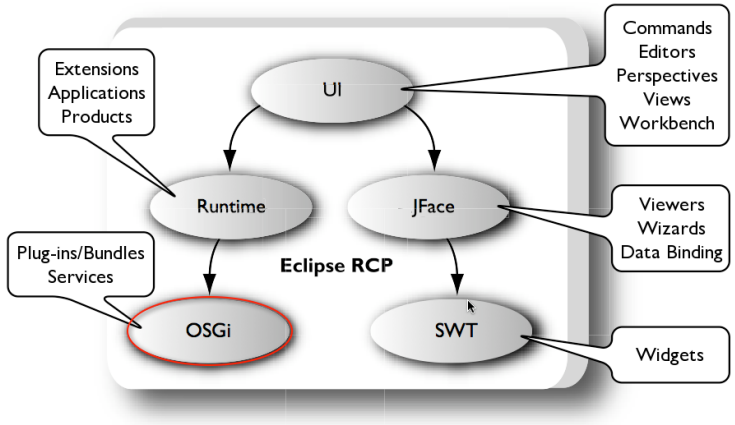
- This is a typical SDK Installation
- Everything below the install location makes up the application
- The plug-in store is a directory containing plug-ins as jar files or folders



- The configuration location contains config files to
  - describe which plug-ins are to be installed into the Runtime
  - Plug-in settings and preferences or cached data

OSGi and Eclipse

- OSGi specifies a framework for defining, composing and executing bundles
- *Plug-in* and *Bundle* today mean the same thing!
- A Bundle is defined in it's `MANIFEST.MF`




```
Bundle-Name: Eclipse UI
Bundle-SymbolicName: org.eclipse.ui; singleton:=true
Bundle-Version: 3.4.1
Bundle-Activator: org.eclipse.ui.internal.UIPlugin
Require-Bundle: org.eclipse.core.runtime,
 org.eclipse.swt;visibility:=reexport,
 org.eclipse.jface;visibility:=reexport,
 org.eclipse.ui.workbench;visibility:=reexport,
 org.eclipse.core.expressions
Export-Package: org.eclipse.ui.internal;x-internal:=true
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0,J2SE-1.3
```

Plug-in id and version

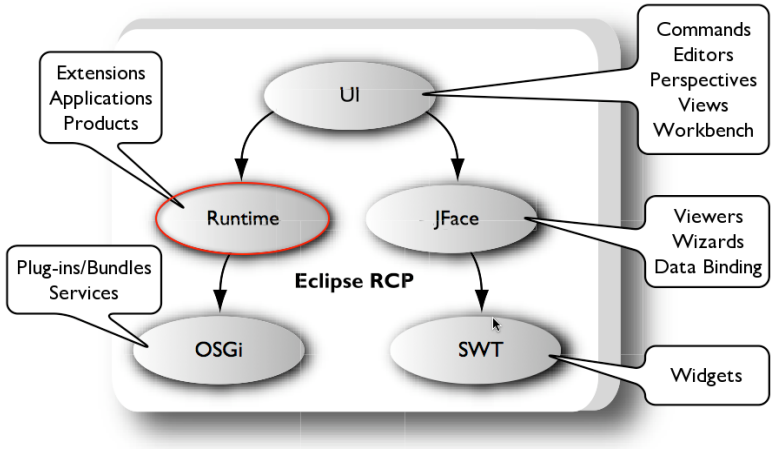
Plug-in Activator

Prerequisite Plug-ins

Exported Code

Eclipse Runtime

OSGi provides the plug-in model. Other runtime mechanisms are still provided by Eclipse



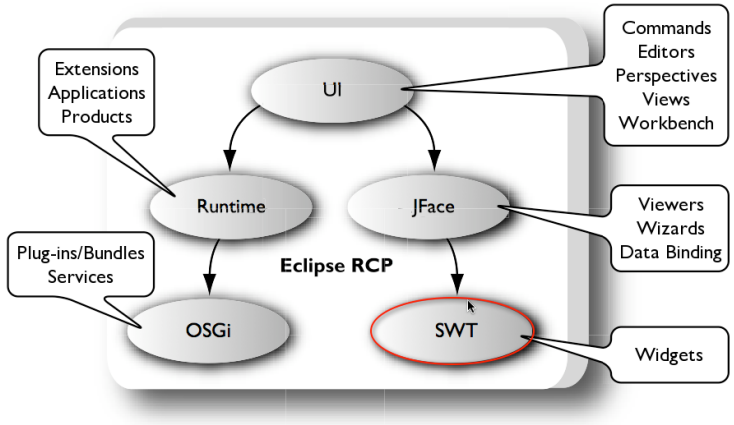
- The *Application* is the Eclipse equivalent to the `main()` method in Java programs
  - There can be many applications in a runtime
  - The application is chosen on startup
- The particular look of the application is configured with a *Product*
- The *Extension Registry* combines functionality provided by separate bundles

Extension Registry

- The *Extension Registry* provides a framework for combining functionality from different bundles.
- Plug-ins allow contributions by declaring an *extension point*.
- Plug-ins add contributions by declaring an *extension*.
- These are defined in the plugin.xml

SWT

- The Standard Widget Toolkit (SWT) provides standard UI controls:
  - Buttons, text fields, lists, menus, fonts, colors, ...
- Consistent and portable Java API
- Uses native widgets underneath
  - i.e. how widgets look depends on the platform

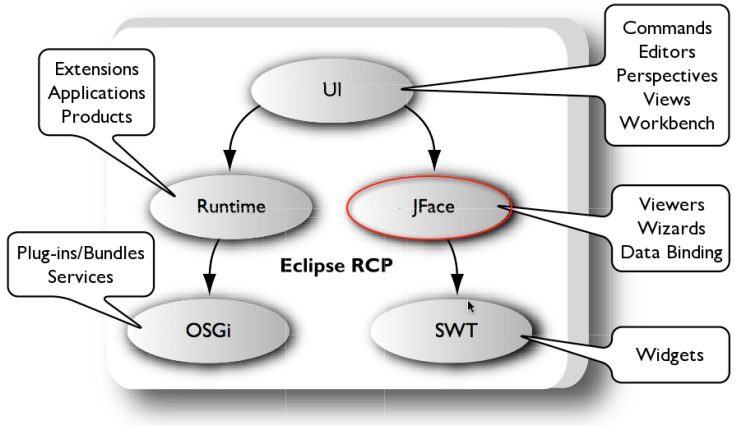


Plus:

- Can be used as a standalone library

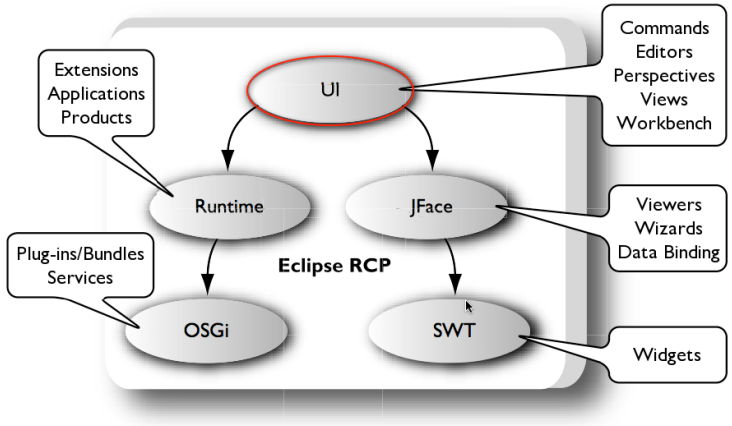
JFace

- High-level abstraction UI toolkit
- Handles common UI programming tasks
- Window system-independent
- Designed to work with SWT without hiding it
- Image and font registries, text support, dialogs, preferences frameworks, wizards, progress reporting, ...



UI Workbench

- Provides the UI model for an RCP application
- *Views*, *Editors* and *Commands* are combined into *Perspectives*
- Contribution-based extensibility
  - All Workbench elements can be added via extensions and programmatically



Conclusion

- Eclipse RCP is a mature and powerful platform
- It comes with a module concept for Java
- It provides a rich set of UI concepts to work with

About this course

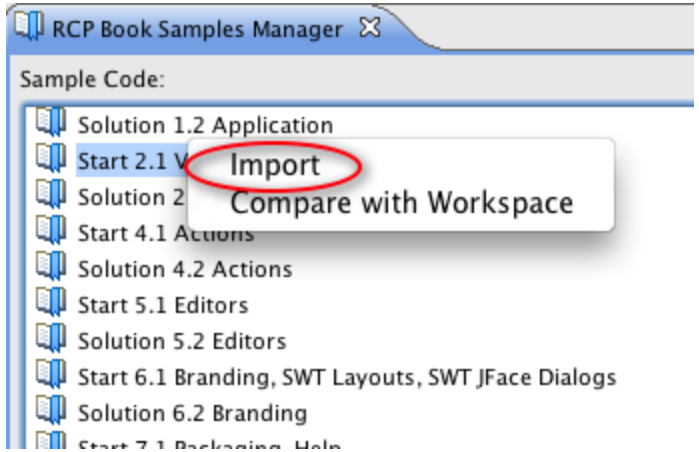
- This course was designed by the authors of the book *Eclipse Rich Client Platform*
- The course reuses the *Samples Manager* from the book to get the sample code into the workspace
- The code samples for this course are based on the Hyperbola chat application from the book
  - They were restructured into modular dependencies instead of a sequence
  - Each module provides a defined start and end

Installing the Samples

- Use the p2-based **Software Updates** mechanism to install the **Samples Manager**
- **Help > Install New Software... > Add... > Archive**
- Enter location of course_samples.zip and hit **OK**
- Uncheck **Contact all update sites during install ...**
- Install **RCP Samples**

Importing Sample Code

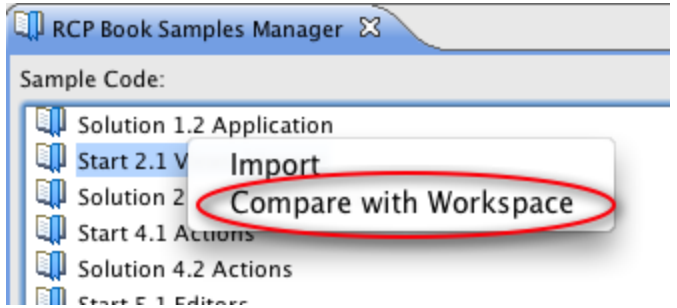
- After installation, the **Samples Manager** view can be opened by selecting **RCP Course** > **RCP Code Samples**
- It displays a list of sample codes
- Select **Import** to import the sample code into your workspace

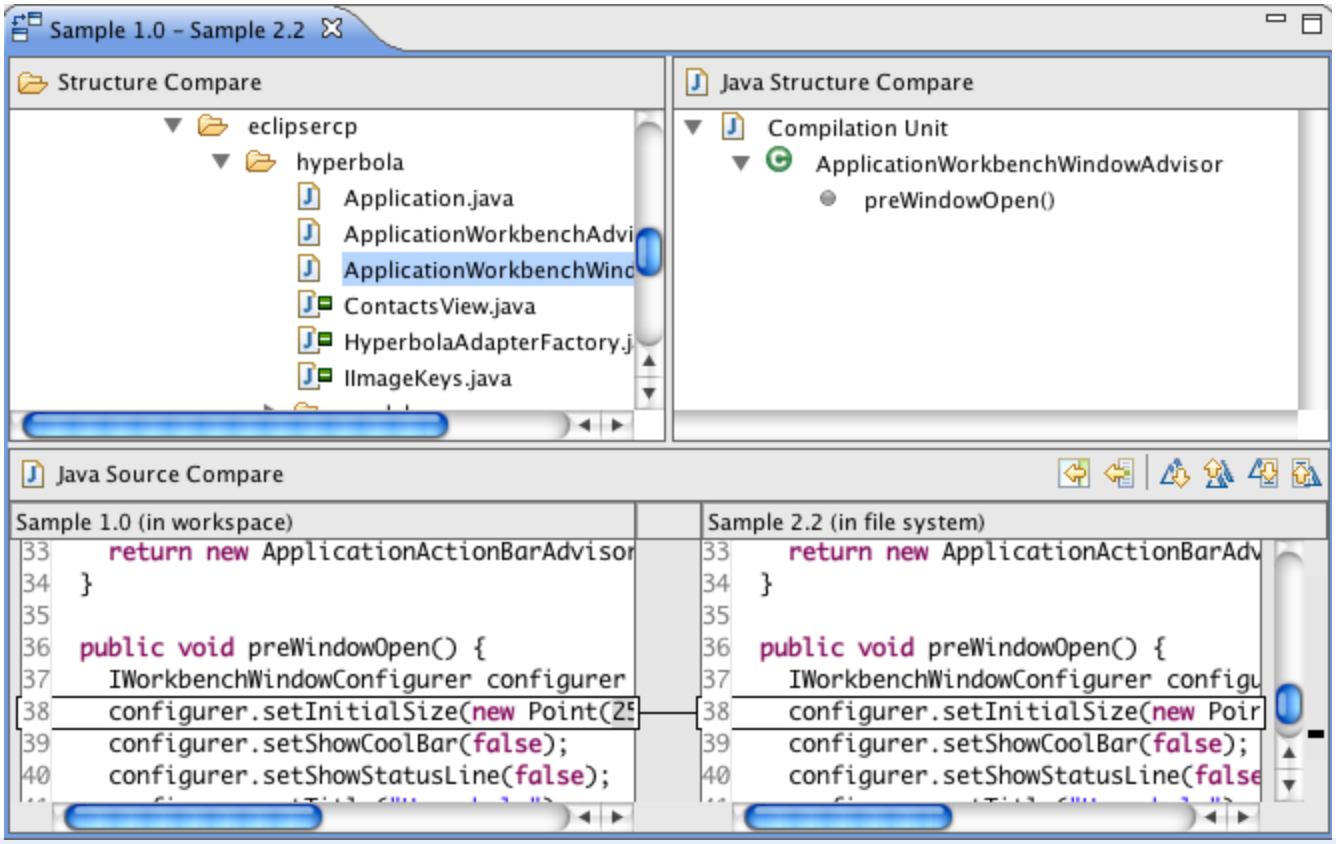


Beware: Importing a snapshot will replace your existing projects!

Finding differences

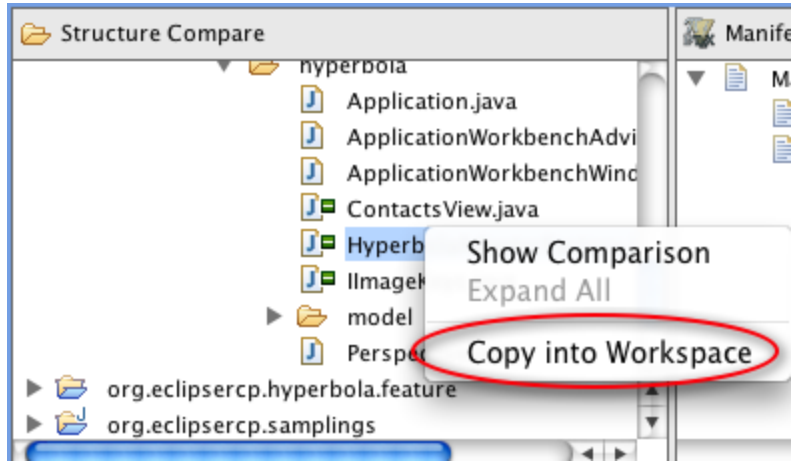
- Select **Compare with Workspace** to compare your workspace content with a specified module sample code
- This opens the **Compare Editor**

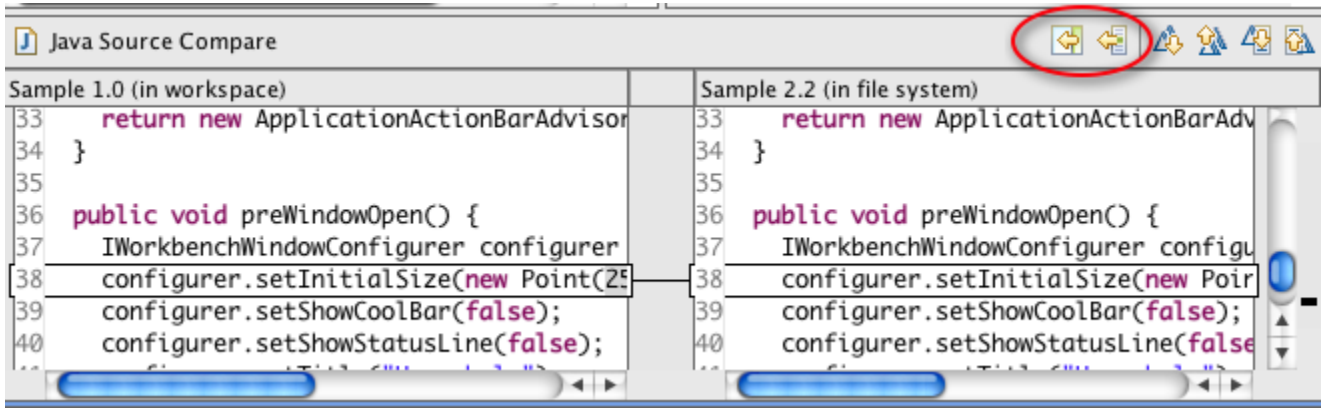




Importing differences

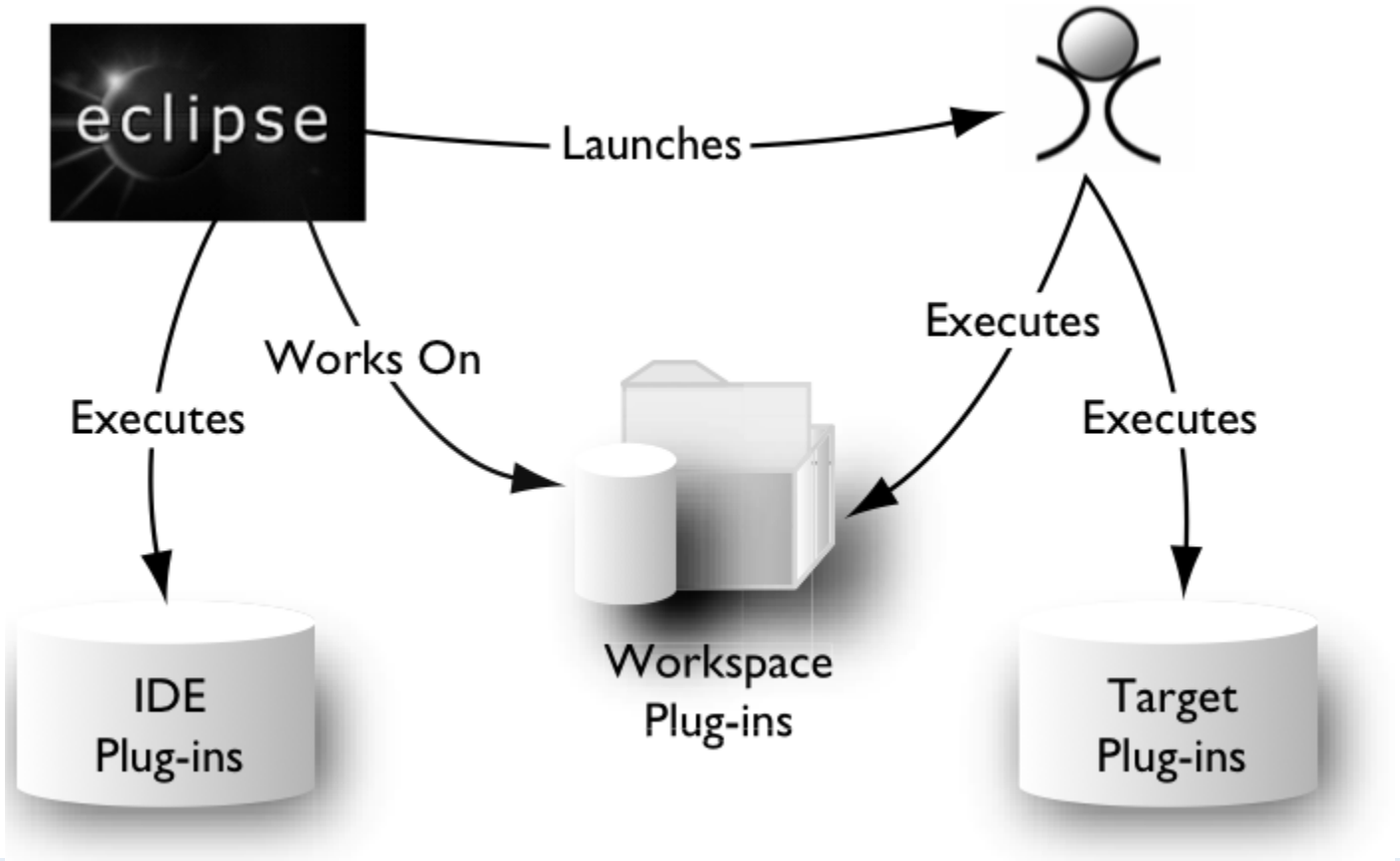
- The upper part of the **Compare Editor** shows files that differ
- By selecting **Copy into Workspace** you can copy whole files from the sample into the workspace
- The lower part of the **Compare Editor** shows differences in the selected file
- With the marked buttons you can copy a specified line or all lines into the workspace





Target Platform

- A *Target Platform* contains the binaries for compiling and running our application
  - By default this is the Eclipse IDE
  - RCP is a subset of the Eclipse IDE



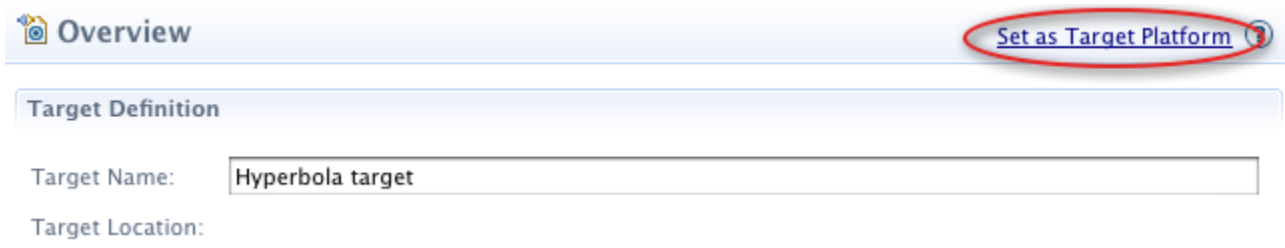
Custom Target Platform

- It is convenient to use a custom target platform for RCP projects
  - Independent from the IDE
  - Custom dependencies are easier managed
- Custom target platforms are often projects in the workspace and shared in the version control
- So-called **Target Definitions** are files that define a target platform

Use a target platform appropriate for your project!

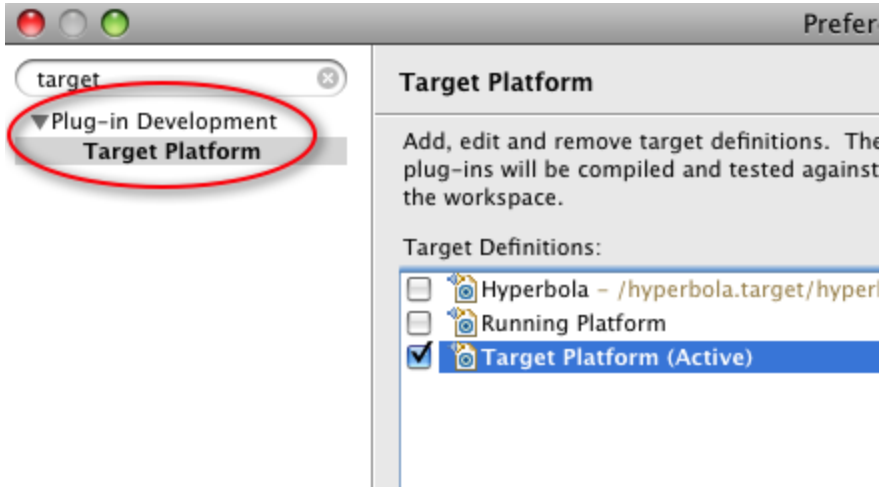
Setting the Target Platform

- Open the **RCP Samples Manager** view
- Right click > **Load Target**
- The new project contains a file `hyperbola.target`
- Open it, then select **Set as target platform**

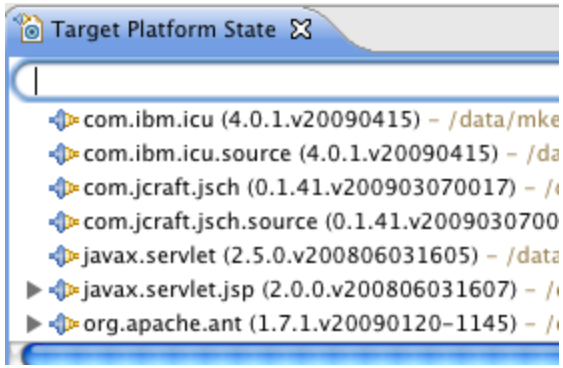


Checking the target platform

- You can look at your target platform settings in the preferences



- The view **Target Platform State** has information about the target platform



Target-platform handcrafted

- The directories in the hyperbola target are downloads from the eclipse.org homepage
- The Eclipse Project Downloads (http://download.eclipse.org/eclipse/downloads/) page provides an entry point for builds of different versions

Build Type	Build Name	Build Status
Latest Release	3.4.2	
3.5 Stream Stable Build	3.5M5	
3.5 Stream Integration Build	I20090313-0100	
3.5 Stream Nightly Build	N20090307-2000	
3.4.2 Stream Build		
Language Pack		

Latest Releases

Build Name	Build Status	Build Date
3.4.2		Wed, 11 Feb 2009 -- 17:00 (-0500)
3.4.1		Thu, 11 Sep 2008 -- 17:00 (-0400)
3.4		Tue, 17 Jun 2008 -- 20:00 (-0400)

3.5 Stream Stable Builds

Build Name	Build Status	Build Date
3.5M5		Mon, 2 Feb 2009 -- 15:35 (-0500)

Tasks

- Install the Samples Manager
- Import the target platform from the course materials
- Double-check with the target platform preferences that the target platform is used
  - The preferences should point to the project location of the target platform project
- If in doubt, ask the instructor

Applications & Workbench

Goals

- Understand basic Workbench concepts
- Identify the major players of an Eclipse RCP Application
- Create a Hello world application
- Run and debug RCP Applications

What is an Application?

Meaning

- The entry point for an Eclipse-based program
- Think of it as the `main`-method.

Purpose

- Start and configure the **Workbench**

IApplication

Application classes implement the interface `IApplication`

- `start()` is called on application start
  - Parameter `IApplicationContext` contains command-line arguments etc.
- `stop()` is called to force the application to shutdown
  - Called on forced exits, i.e. `kill`-commands.
  - Not called during a normal exit, i.e. last workbench window closed.

IApplication.start()

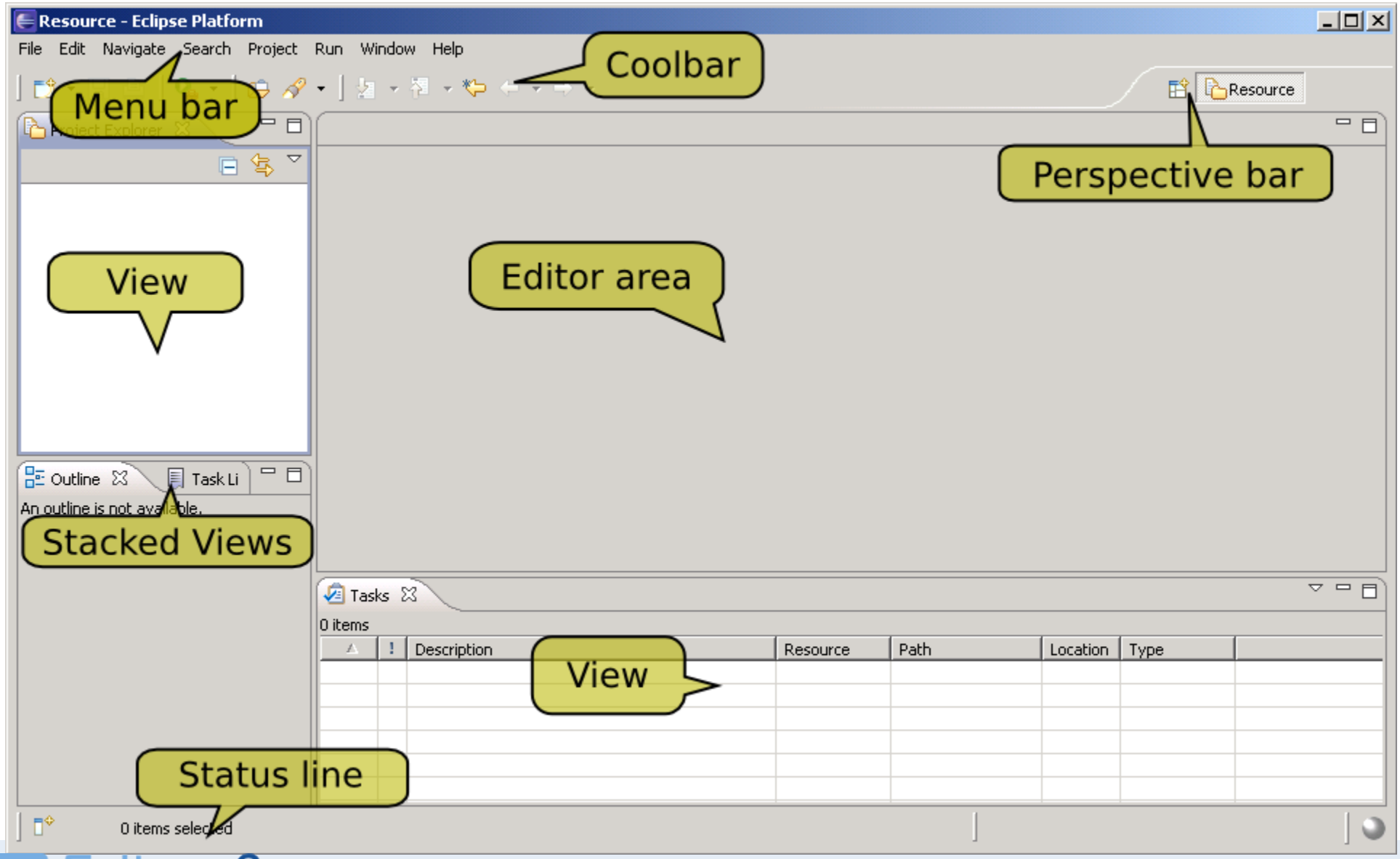
```
public Object start( IApplicationContext context )
                                        throws Exception {
   Display display = PlatformUI.createDisplay();
   try {
     int returnCode = PlatformUI.createAndRunWorkbench( display,
                                new ApplicationWorkbenchAdvisor() );
     if( returnCode == PlatformUI.RETURN_RESTART ) {
       return IApplication.EXIT_RESTART;
     }
     return IApplication.EXIT_OK;
   } finally {
     display.dispose();
   }
}
```

The application could do anything you want. There is no need to set up a graphical display or start a workbench.

Workbench

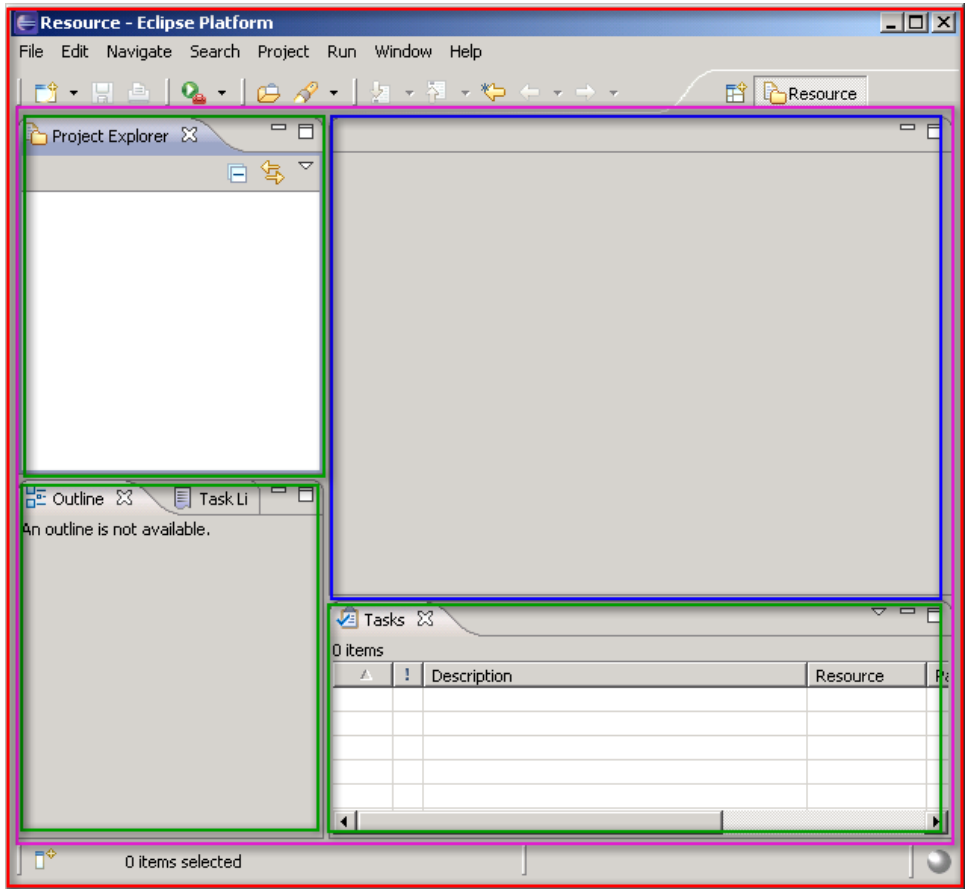
- RCP applications usually fire up a workbench using the code from the previous slide
- The basic look of the workbench is configured using
  - `WorkbenchAdvisor`
  - `WorkbenchWindowAdvisor`
  - `ActionBarAdvisor`
- Before we dive into that, let's have a look at the workbench structure

The Workbench Window



Workbench Structure

- Workbench (1)
  - Workbench windows (1-N)
    - Page (1)
      - Perspectives (1-N)
      - Views (0-N)
      - Editors (0-N)



Workbench API

- The workbench elements can be accessed in a hierarchical API
- Traversal up and down the tree is possible
- Entry point is `PlatformUI.getWorkbench()`

IWorkbench

- The workbench has no visual representation by itself
- It provides access to the elements that do influence visual representation
  - ProgressService
  - EditorRegistry
  - PerspectiveRegistry
  - SharedImages
  - see `IWorkbench#get*()`-methods
- It provides methods for controlling the workbench
  - `close()`
  - `restart()`
  - various `save()`-methods

IWorkbenchWindow

- There may be 1 to *n* workbench windows for a workbench

- Each workbench window contains a workbench page
  - A container for parts (views and editors)
  - Does not need to be programmed
- While programming an RCP application you may need to reference the WorkbenchPage:

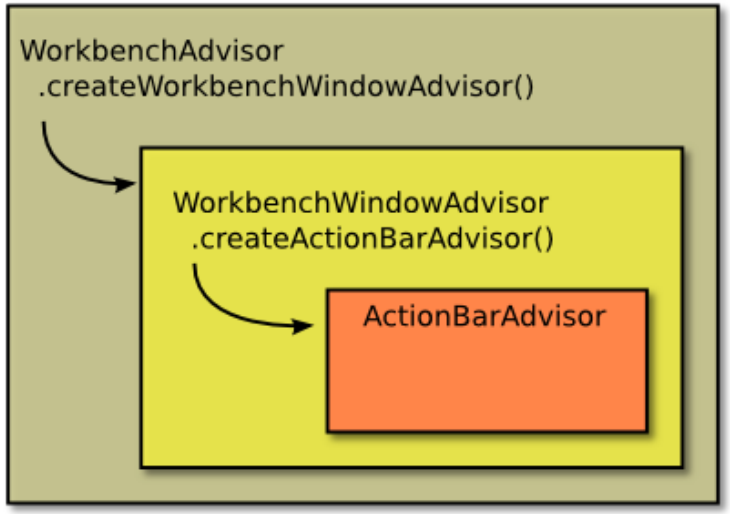
```
// within view / editor code:
IWorkbenchPage page = getSite().getPage();
// elsewhere:
IWorkbenchPage page = workbenchWindow.getActivePage();
```

Advisors

- Creation of the workbench is done by the framework upon calling

```
PlatformUI.createAndRunWorkbench( Display, WorkbenchAdvisor );
```

- The creation process is influenced by the `WorkbenchAdvisor` and subsequent advisors:



WorkbenchAdvisor

- The `WorkbenchAdvisor` advises mainly two things:
  - The initial perspective to be shown.
  - The `WorkbenchWindowAdvisor` to be used.

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {
  public String getInitialWindowPerspectiveId() {
    return "org.eclipsercp.hyperbola.perspective";
  }

  public WorkbenchWindowAdvisor createWorkbenchWindowAdvisor(
                              IWorkbenchWindowConfigurer configurer) {
    return new ApplicationWorkbenchWindowAdvisor(configurer);
  }
}
```

Perspective

- The initial perspective id is contributed by an *extension*
- The extension defines the id, a human readable name and a class
- The initial perspective id is used **only** when no saved workbench state exists (from previous runs)
- Details to follow in the next chapter

```
<extension point="org.eclipse.ui.perspectives">
   <perspective
         id="org.eclipsercp.hyperbola.perspective"
         name="Hyperbola Perspective"
         class="org.eclipsercp.hyperbola.Perspective">
   </perspective>
</extension>
```

```
public String getInitialWindowPerspectiveId() {
   return "org.eclipsercp.hyperbola.perspective";
}
```

WorkbenchWindowAdvisor

- A `WorkbenchWindowAdvisor` guides the UI rendering in the window
- Various methods are called during different life cycle steps
  - i.e. `preWindowOpen()` or `postWindowCreate()`

```
public void preWindowOpen() {
    IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
    configurer.setInitialSize( new Point( 250, 350 ) );
    configurer.setShowMenuBar( true );
    configurer.setShowCoolBar( true );
    configurer.setShowStatusLine( false );
    configurer.setTitle("Hyperbola");
}
```

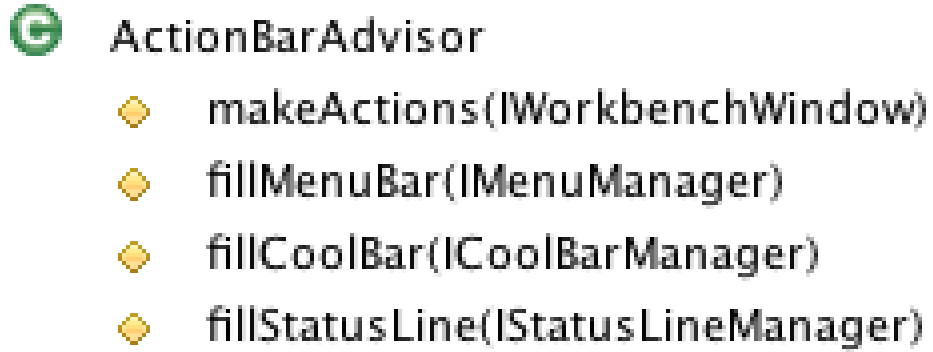
WorkbenchWindowAdvisor (cont.)

- The `WorkbenchWindowAdvisor` is asked for an `ActionBarAdvisor`

```
ApplicationWorkbenchWindowAdvisor.java
 public ActionBarAdvisor createActionBarAdvisor(
                              IActionBarConfigurer configurer)
 {
   return new ApplicationActionBarAdvisor(configurer);
 }
```

ActionBarAdvisor

- The `ActionBarAdvisor` provides hooks for creating and positioning actions in the workbench window
- Subclasses override appropriate methods for configuring the workbench window



Using PDE

- The Plug-in Development Environment (PDE) provides tools for creating, testing, debugging and building Eclipse artifacts:
  - Plug-ins
  - Features
  - Fragments
  - Products
  - ...
- It adds various tools into the IDE, such as specialized Wizards, Editors, Views, help content and more
- PDE is part of the Eclipse Classic download

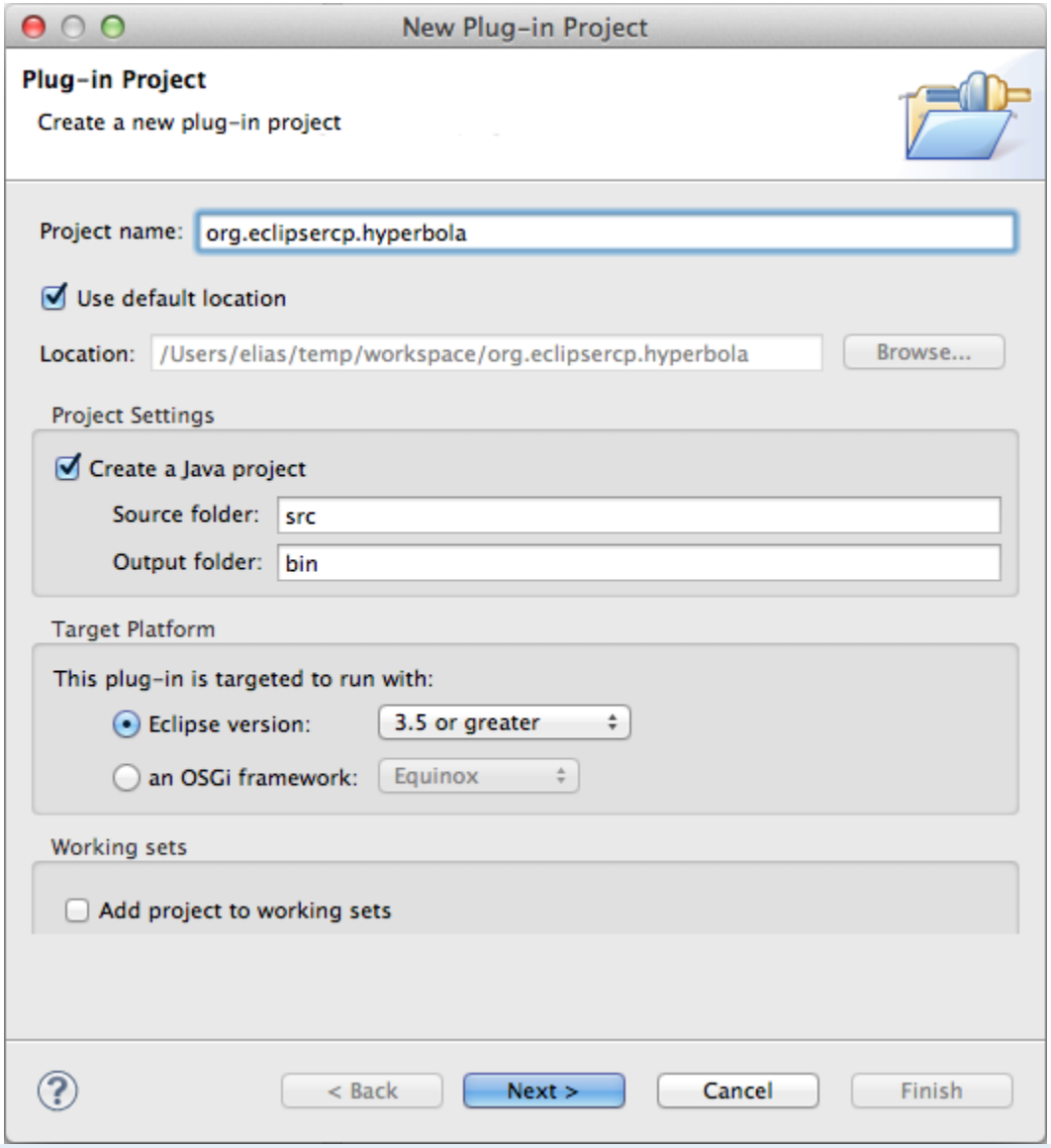
Creating a New Plug-in Project

- To create a new Plug-in use the *New Plug-in Project* Wizard.
  - Go to **File > New > Project...**.
  - Choose **Plug-in Project** and click **Next**.
  - Fill out the **Project Name** and click **Next**.
  - Decide to create an RCP application or regular plug-in and click **Next**.
  - Select an appropriate template and click **Finish**

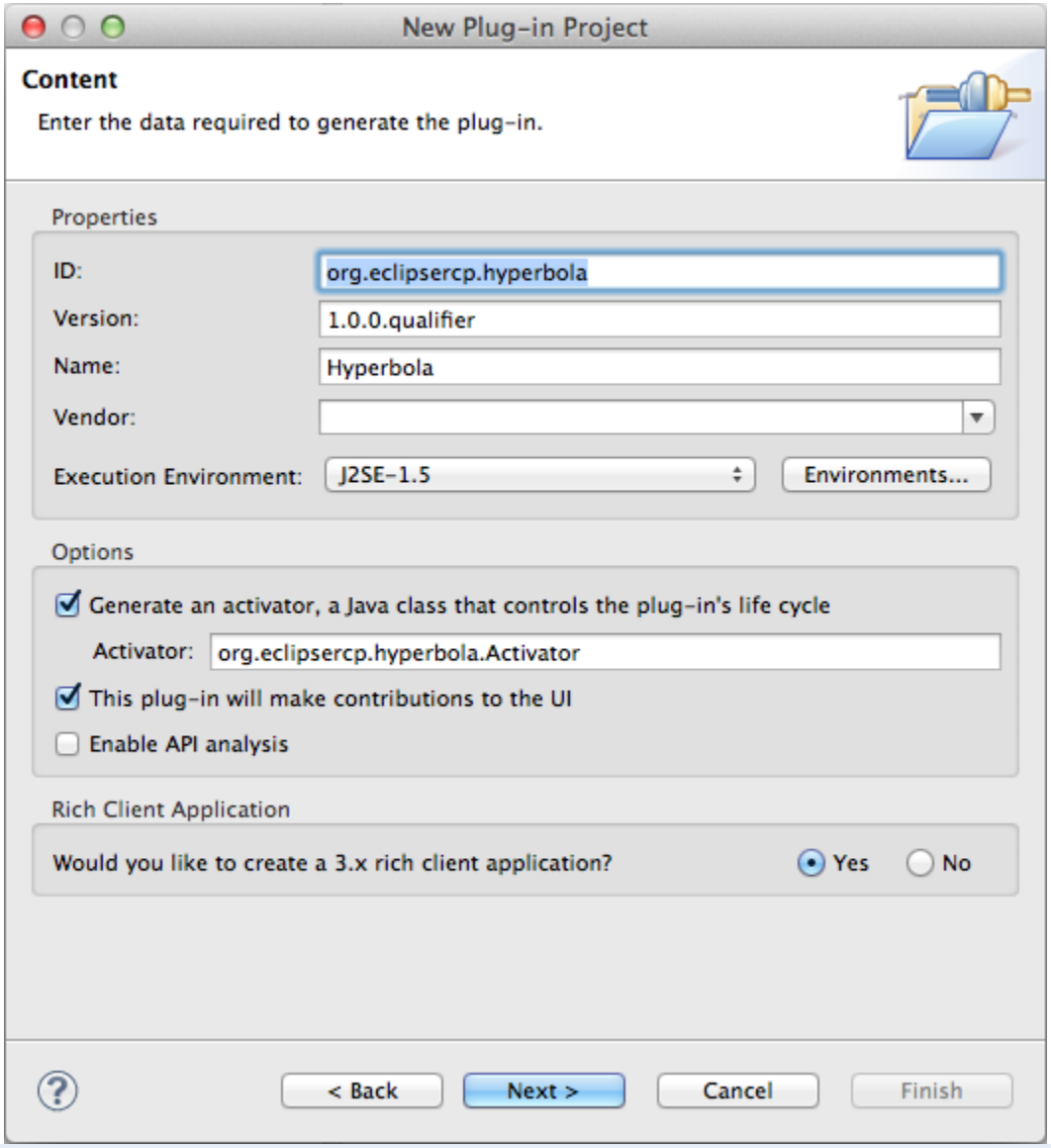
Plug-in Naming Conventions:

- The project name should be the same as the plug-in id
- Plug-in ids must be unique
- Create plug-in ids by following *Java namespace* naming conventions
  - Pick a url you own and flip the segments
  - append your project name
  - append the role of the plug-in in the project

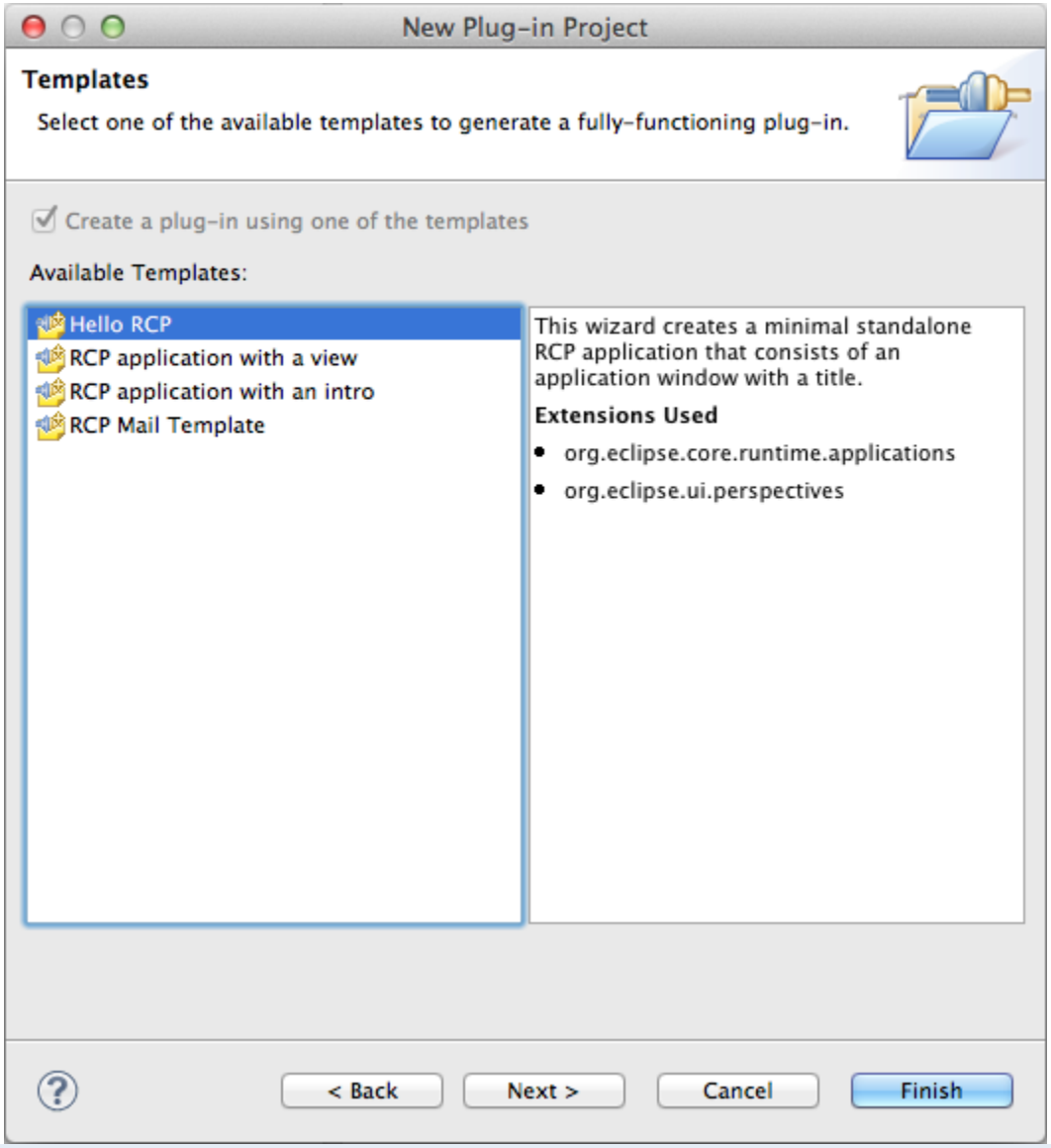
New Plug-in Wizard



New Plug-in Wizard

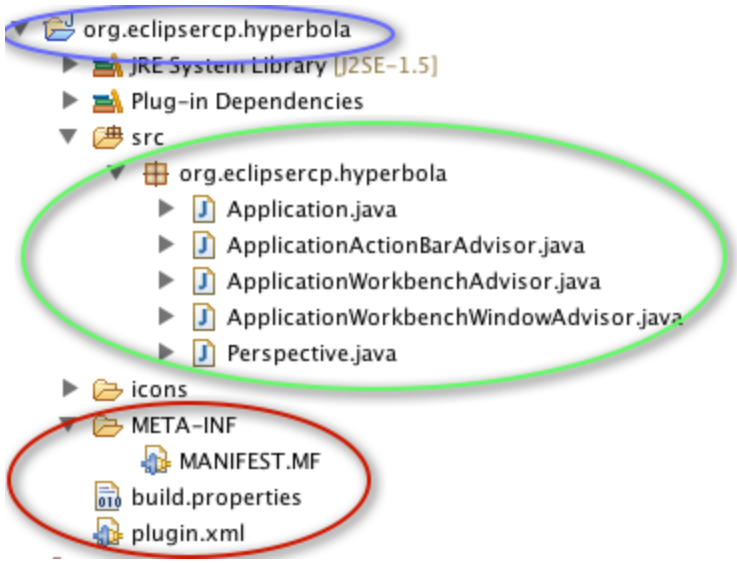


New Plug-in Wizard



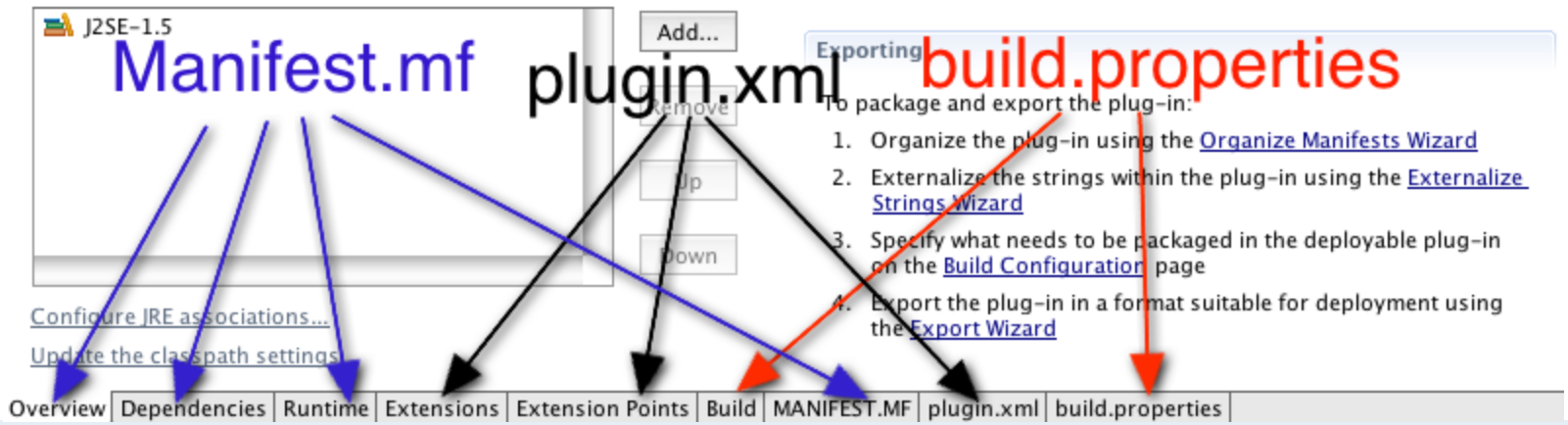
Created files

- The template creates
  - A Java project,
  - Java source files and
  - Plug-in manifest files



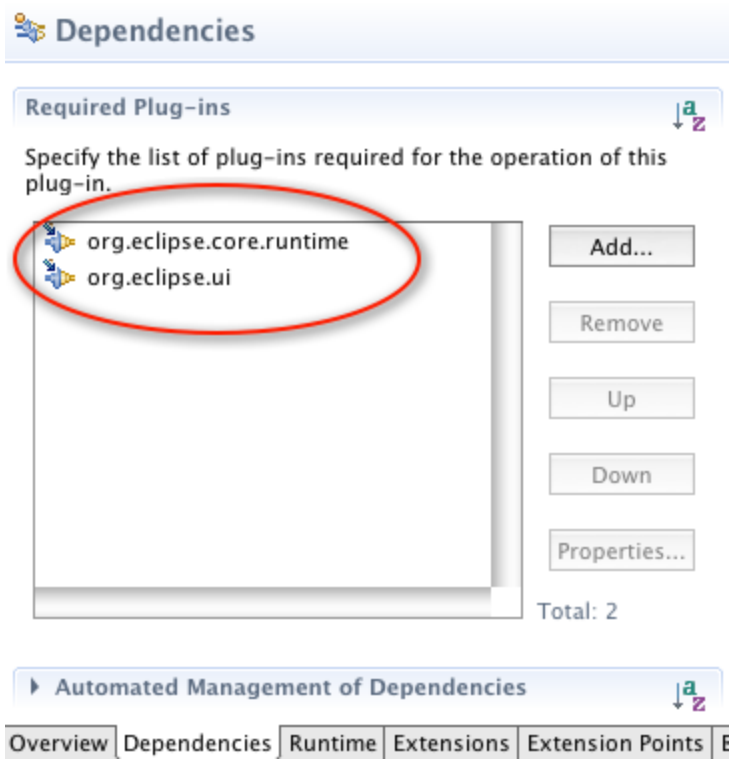
The Manifest Editor

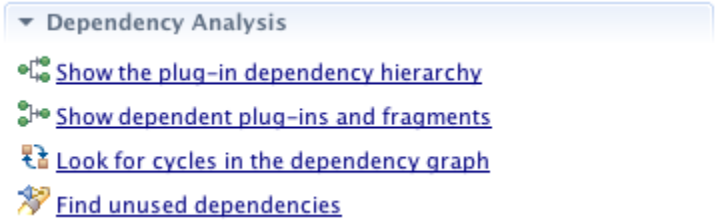
- The manifest files are edited in the *Plug-in Manifest Editor*
- The editor provides multiple tabs for specific aspects of the plug-in, i.e.
  - Provider, ID and versions
  - Dependency management
  - Extensions
  - Build properties



Plug-in Dependencies

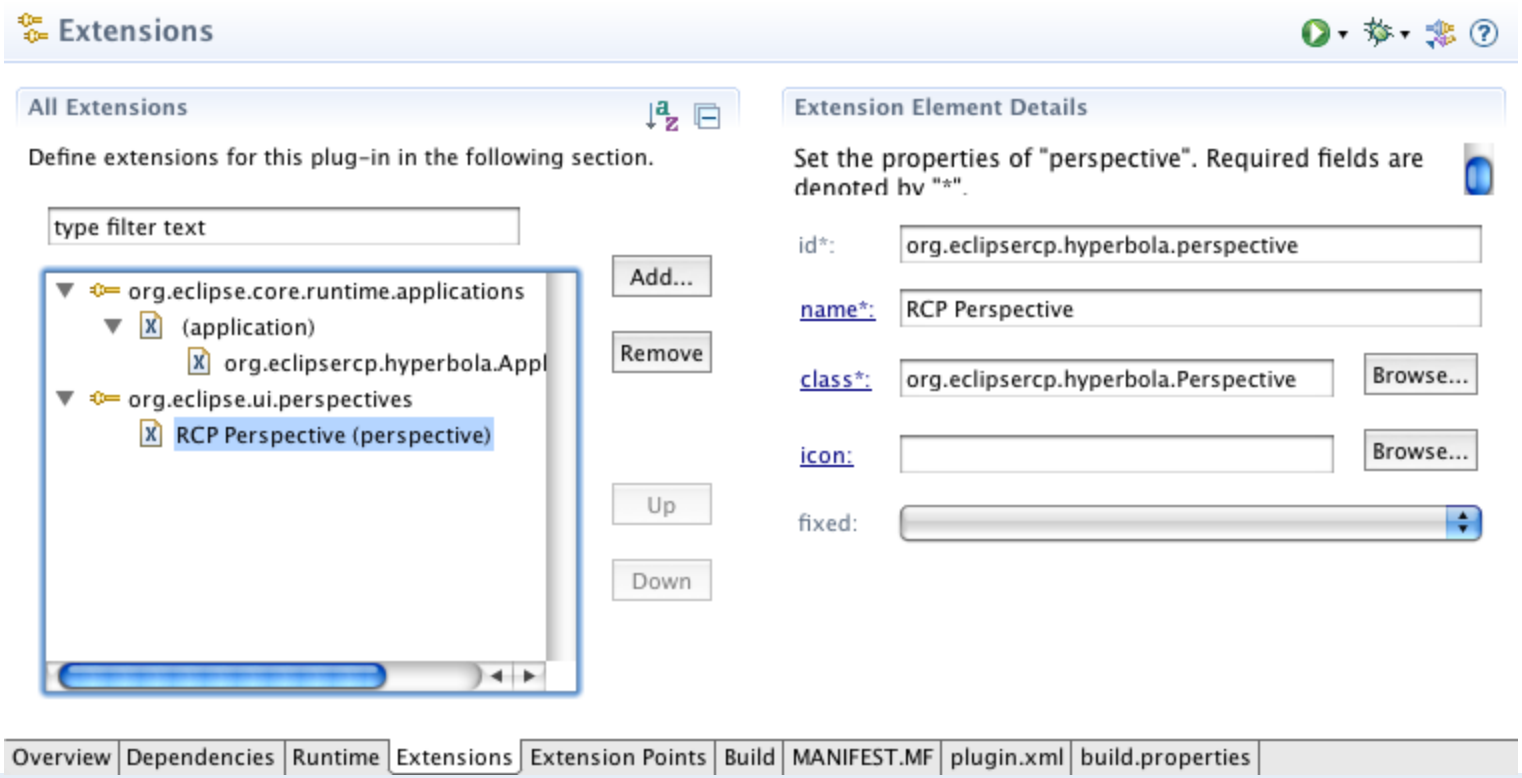
- The dependencies define which plug-ins we depend on
- Only classes in the plug-ins we depend on are visible to us
- Dependency analysis allows us to browse the dependency hierarchy in both directories





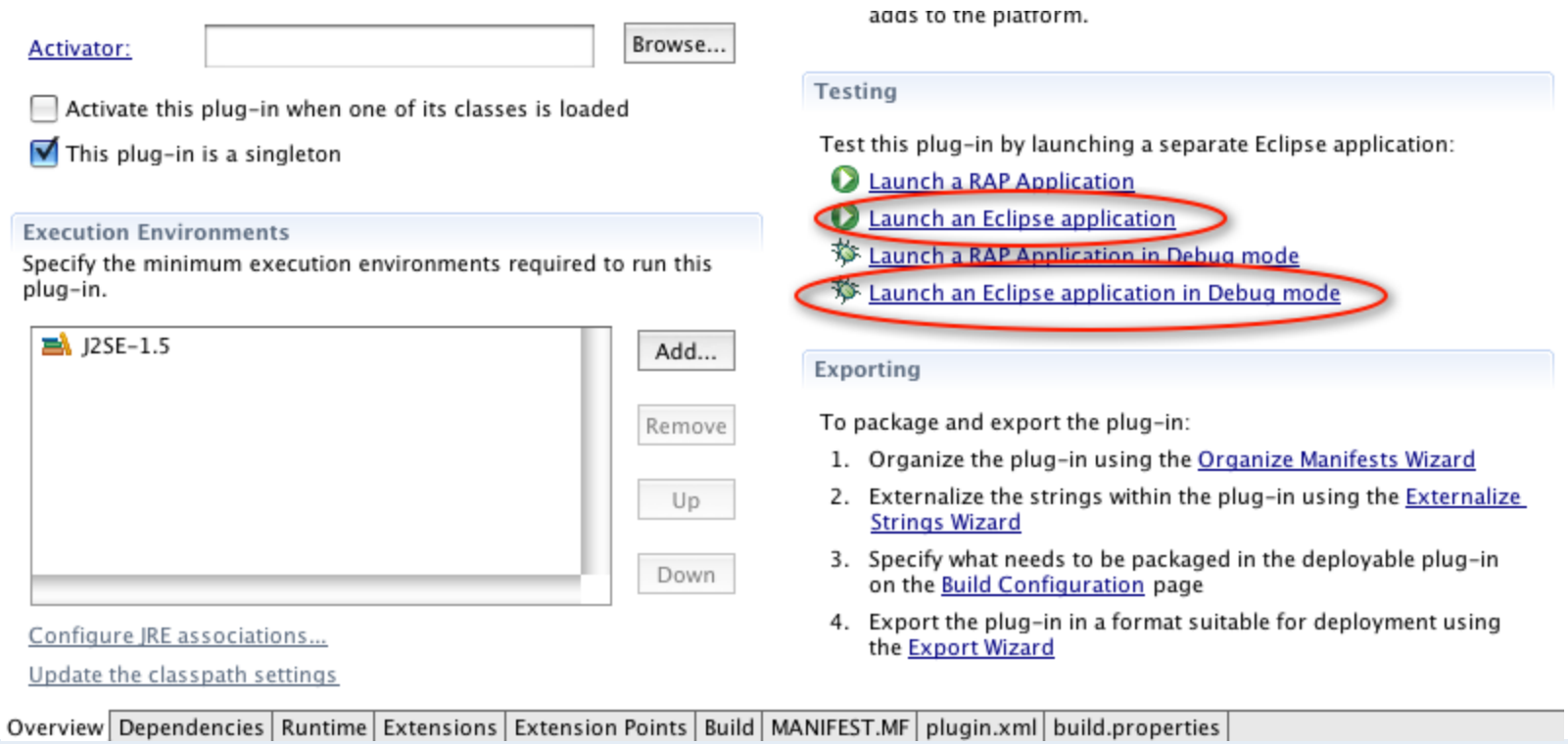
Extensions

- Two extensions were already generated for us
  - The application
  - The initial perspective

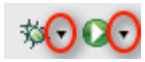


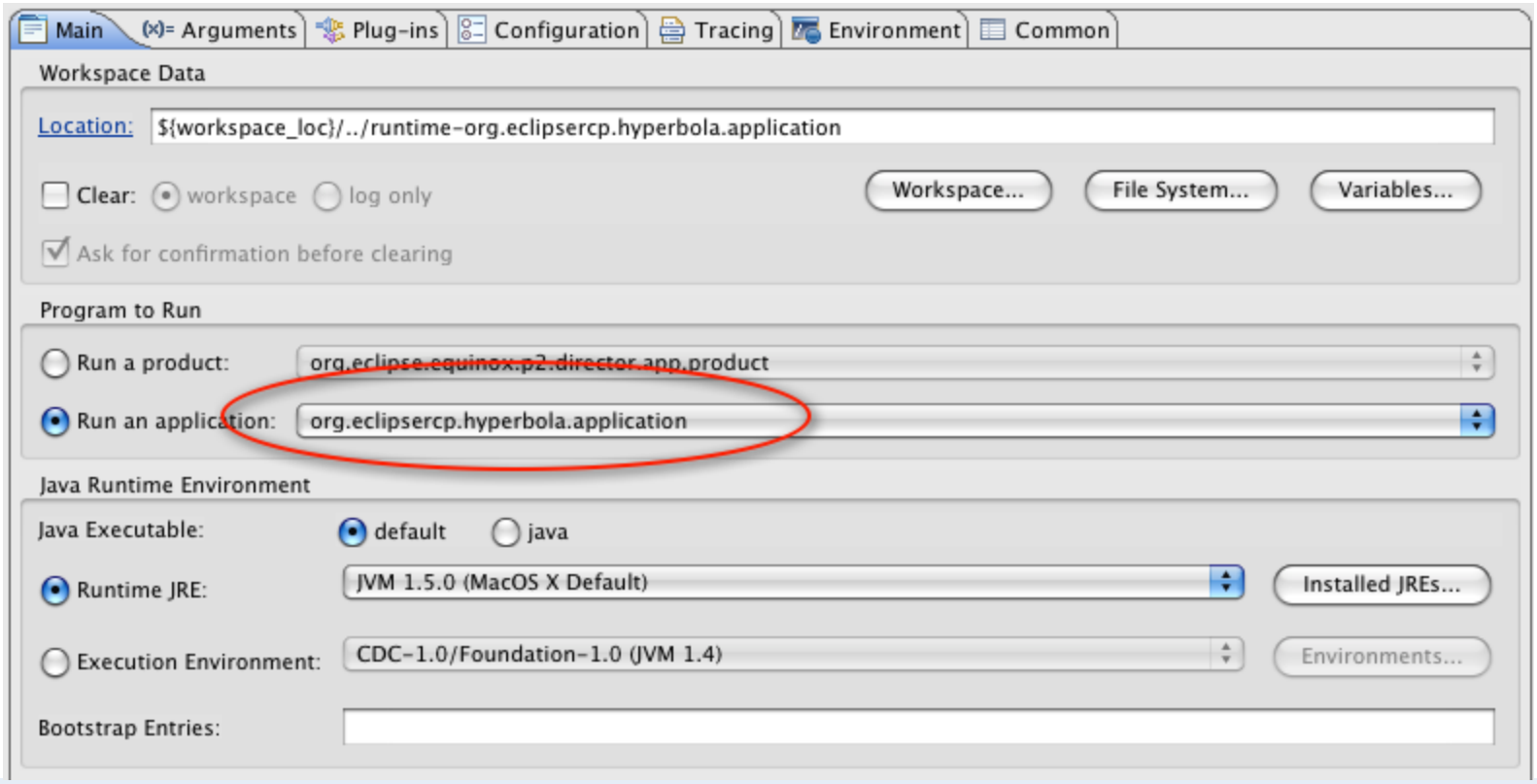
Running and Debugging

- The *Testing* section in the *Overview* tab provides possibilities to launch the application



Launch configurations

- *Launch configurations* allows to configure the launched application
  - Select **Run configuration...** from the **Run** or **Debug** menu 



Tasks

- Create a plug-in with the New Plug-in Wizard
  - Name your project **org.eclipsercp.hyperbola**
  - Select the option **Yes** next to "**Would you like to create a rich client application?**"
  - Use the **Hello RCP** template to create a small RCP application
- Make yourself familiar with the generated plug-in metadata and Java classes
- Launch the Application in **run** and **debug** mode using the links in the plug-in manifest editor
  - Set breakpoints in the advisor classes to see when they are called

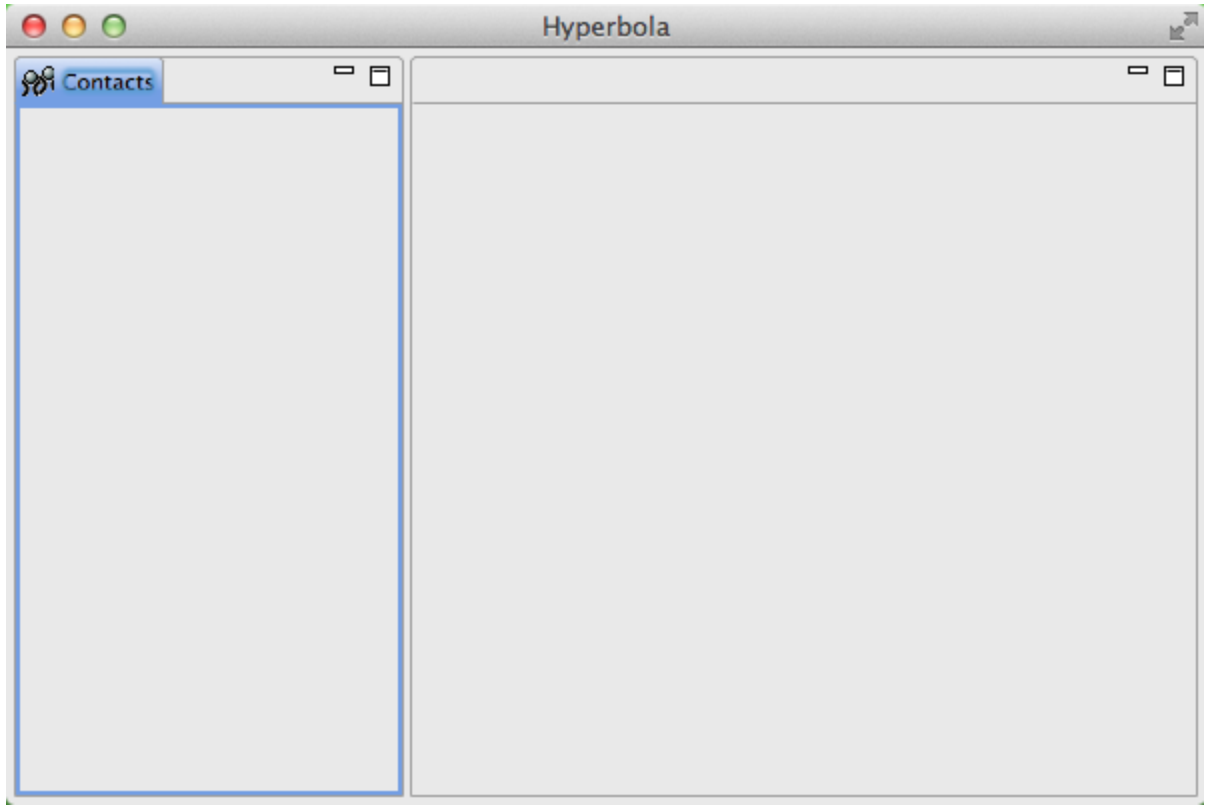
Solution

- Start with an empty workbench
- If unsure how to proceed, compare to **1.1 Applications**

Pointers

- **Platform Plug-in Developer Guide > Programmer's Guide > Platform architecture**
- **Platform Plug-in Developer Guide > Programmer's Guide > Simple plug-in example**

Views & Perspectives

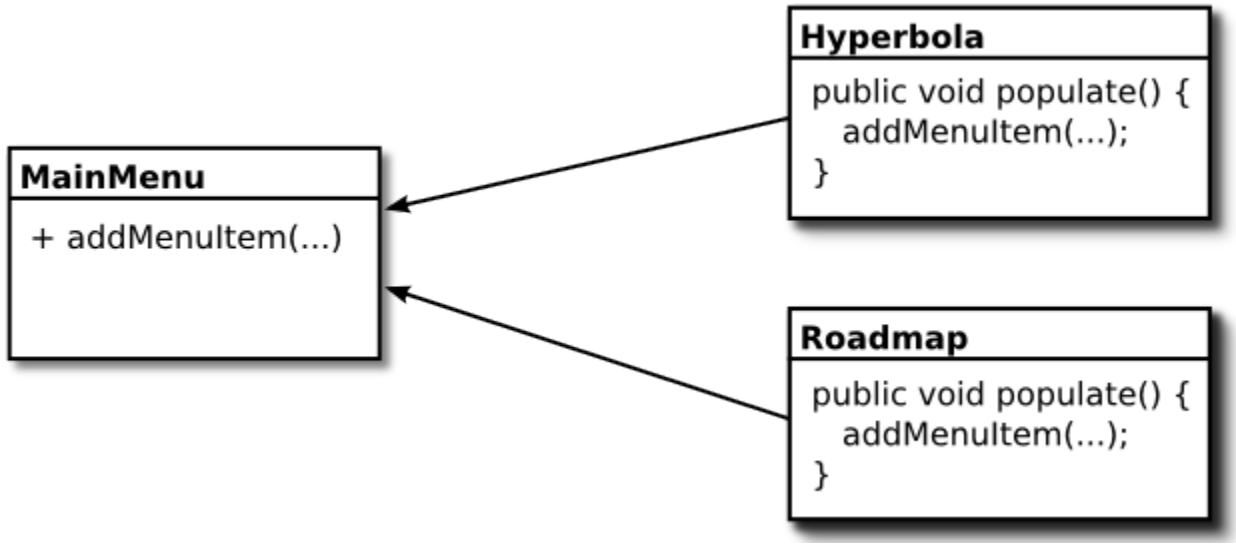


Goals

- Get to know the concept of *Extensions*
- Learn more about Views and Perspectives
- Learn how to contribute a View
- Learn how to contribute a Perspective

Extensions

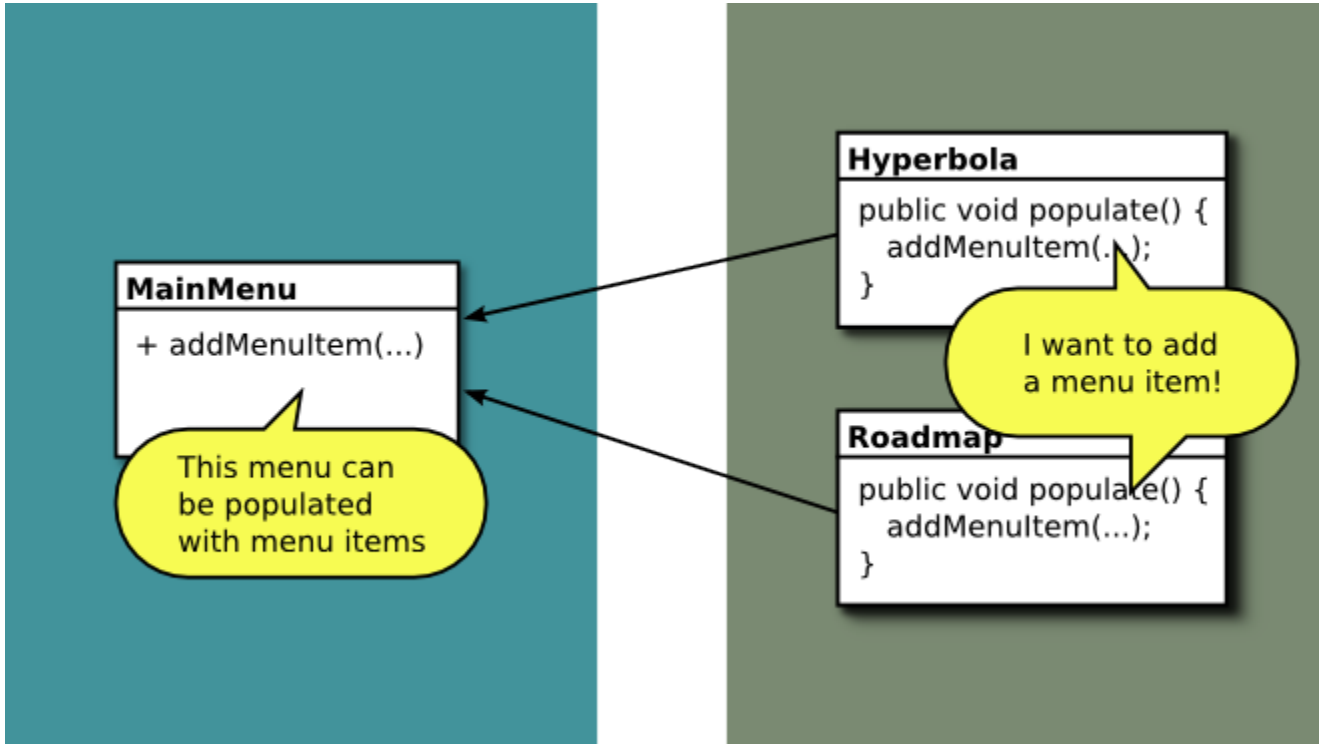
- Imagine an extensible application that wants to allow menu entries to be added from various application parts



- This extensible application has to solve a few problems:
  - how do to find out which parts want to add to the menu?
  - how to call `populate()` to add the entries?
  - what happens if parts of the application are installed, uninstalled?

Extension Registry

- The *Extension Registry* is infrastructure that brings together plug-ins
  - *Extension Point* Provider offers a possibility
  - *Extension* Contributor hooks itself in
- Declarative format with XML



Extensions and Extension Points I

Plug-in org.eclipse.ui

META-INF/MANIFEST.MF

```
...
```

plugin.xml

```
<extension-point id="actionSets"
                 schema="schema/actionSets.exsd"/>
```

schema/actionSets.exsd

```
<?xml version='1.0' encoding='UTF-8'?>
<schema targetNamespace="org.eclipse.ui"
        xmlns="http://www.w3.org/2001/XMLSchema">
   ...
   <element name="actionSet">
     <sequence>
        <element ref="action" />
     </sequence>
   </element>
   ...
</schema>
```

MenuCreator

```
Platform.getExtensionRegistry()
  .getConfigurationElementsFor(
  "org.eclipse.ui.actionSets" );
```

Plug-in org.eclipsercp.hyperbola

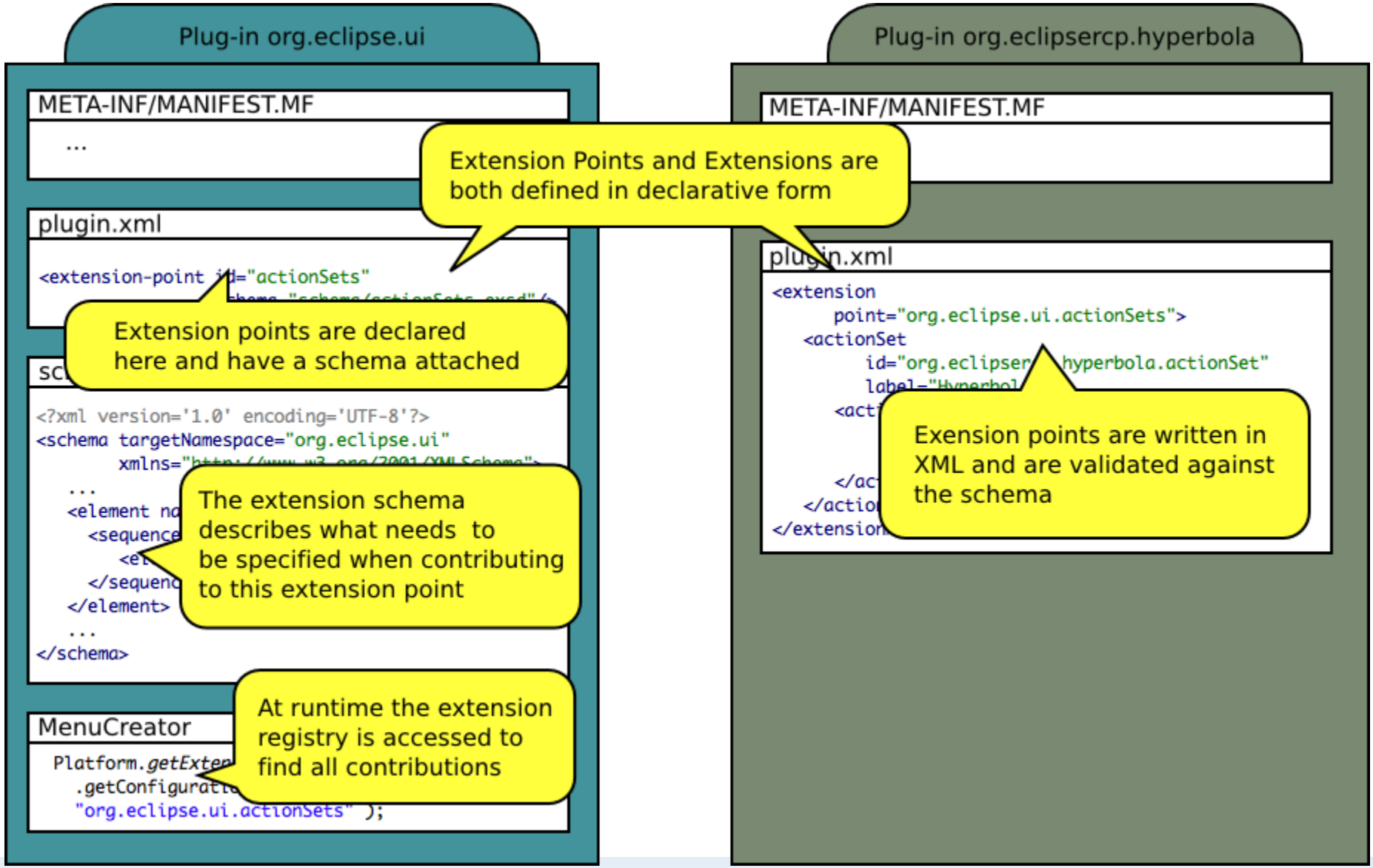
META-INF/MANIFEST.MF

```
...
```

plugin.xml

```
<extension
      point="org.eclipse.ui.actionSets">
   <actionSet
         id="org.eclipsercp.hyperbola.actionSet"
         label="Hyperbola">
      <action
            id="org.eclipsercp.hyperbola.addContact"
            label="Add Contact"
            class="org.eclipsercp...AddContact">
      </action>
   </actionSet>
</extension>
```

Extensions and Extension Points II

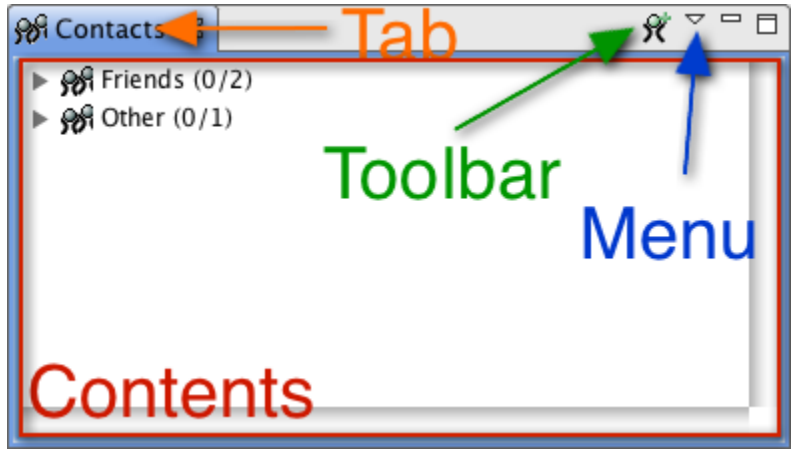


Workbench Extension Points

- There is a huge amount of extension points that can be used to plug into the workbench or more advanced parts of Eclipse
- Some examples for extension points:
  - `org.eclipse.ui.perspectives`
  - `org.eclipse.ui.views`
  - `org.eclipse.ui.editors`
  - `org.eclipse.ui.commands`
  - `org.eclipse.ui.menus`
- More detailed extension point references:
  - Help > Platform Plug-in Developer Guide > Reference > Extension Points Reference
  - Plug-in Manifest Editor > Extension Points
  - RCP Book chapter 15.5

Views

- Views and Editors make up the workbench page content

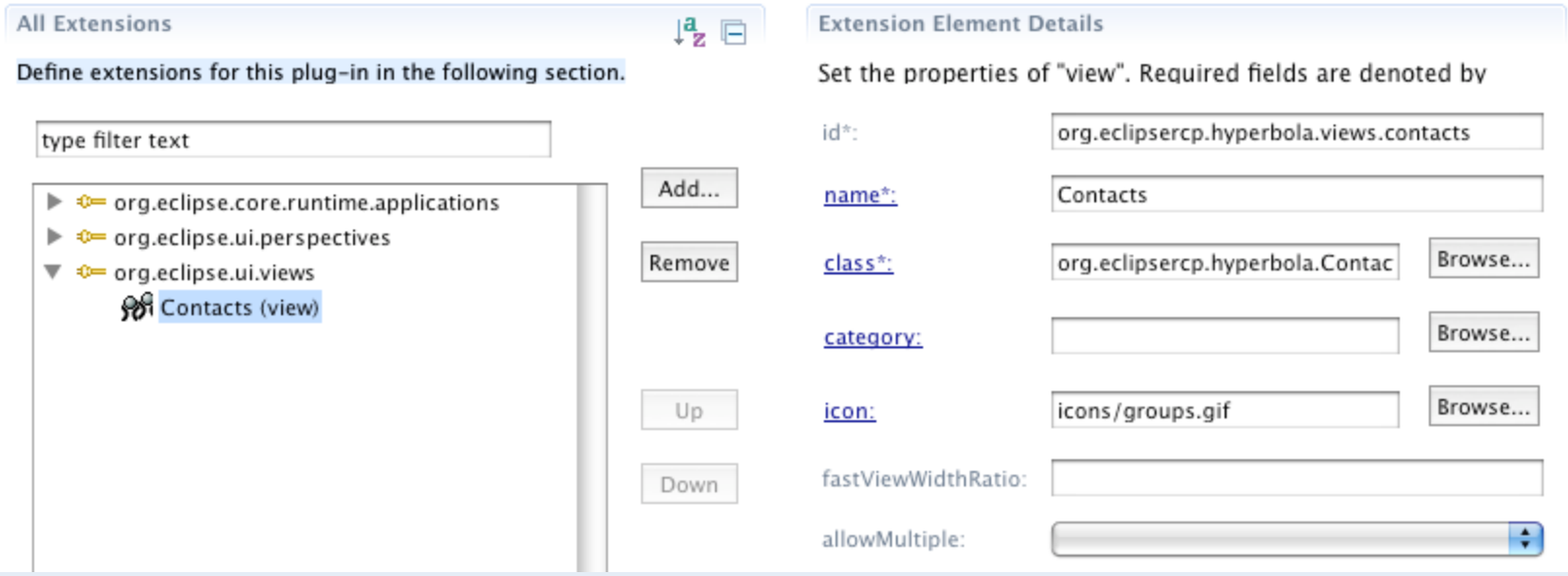


- Views can have
  - A tab with a name
  - A toolbar
  - A menu
  - View Contents
- Views can be
  - Resized, moved, maximized or minimized
  - Standalone -- without menu and toolbar
  - Fast Views

Adding a View

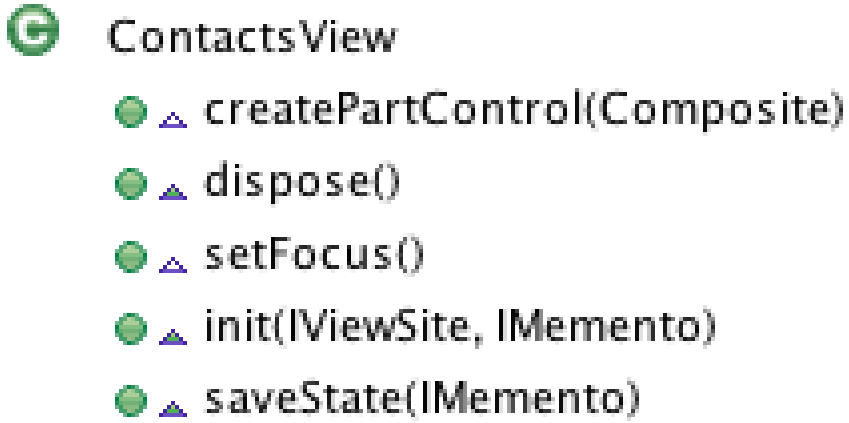
- To add a view extend `org.eclipse.ui.views`

```
<extension point="org.eclipse.ui.views">
   <view id="org.eclipsercp.hyperbola.views.contacts"
         icon="icons/groups.gif"
         class="org.eclipsercp.hyperbola.ContactsView"
         name="Contacts"/>
</extension>
```

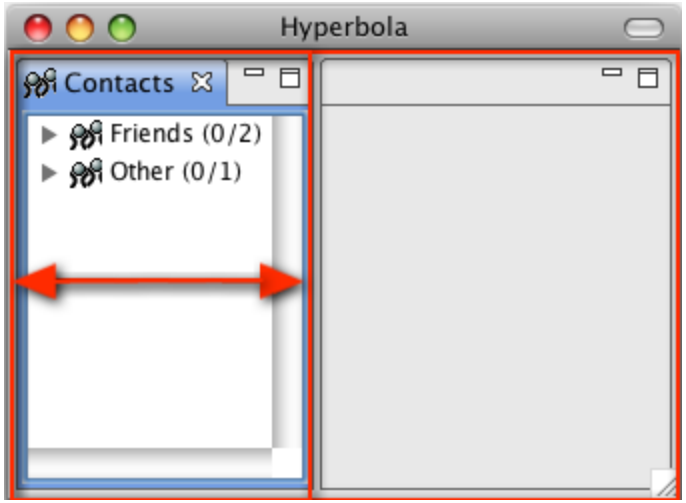


ViewPart Implementation

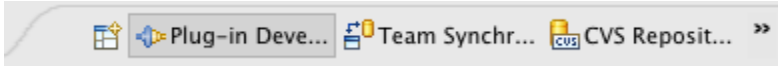
- The attribute `class` provides a fully qualified class name
- The view must implement `org.eclipse.ui.IViewPart`
  - Subclass `org.eclipse.ui.part.ViewPart` instead
  - `createPartControl(Composite)` allows the view to create controls
  - Remove listeners in `dispose()`
  - Save and restore state in `init(IViewSite, IMemento)` and `saveState(IMemento)`



Perspectives



- Perspectives define the arrangement of a set of views/editor that are displayed simultaneously
- Toolbar items and menu entries can be attached to a perspective
- Perspectives can be fixed (unchangeable)
- Perspectives can be switched by the user (or programmatically)

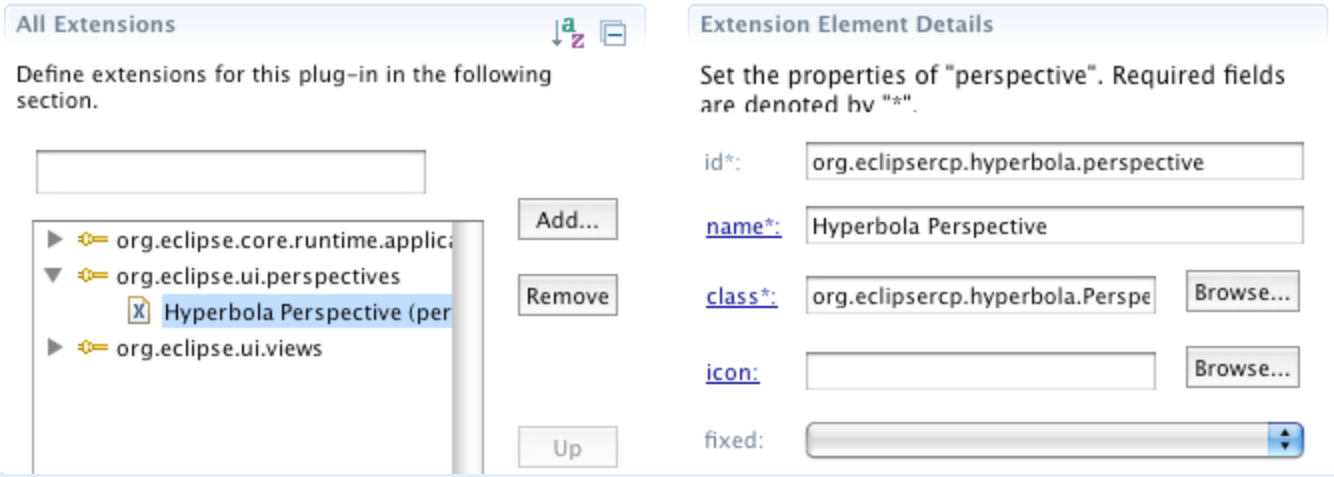


- Perspectives are used to combine a set of views and actions that make up a common workflow

Adding a Perspective

- Provide a perspective by defining an extension to
  - `org.eclipse.ui.perspectives`

```
<extension point="org.eclipse.ui.perspectives">
   <perspective
         id="org.eclipsercp.hyperbola.perspective"
         name="Hyperbola Perspective"
         class="org.eclipsercp.hyperbola.Perspective">
   </perspective>
</extension>
```



Placing Views in a Perspective

- `IPageLayout` provides methods for adding views to a perspective
- The editor area is not placed itself but used as orientation

Perspective.java

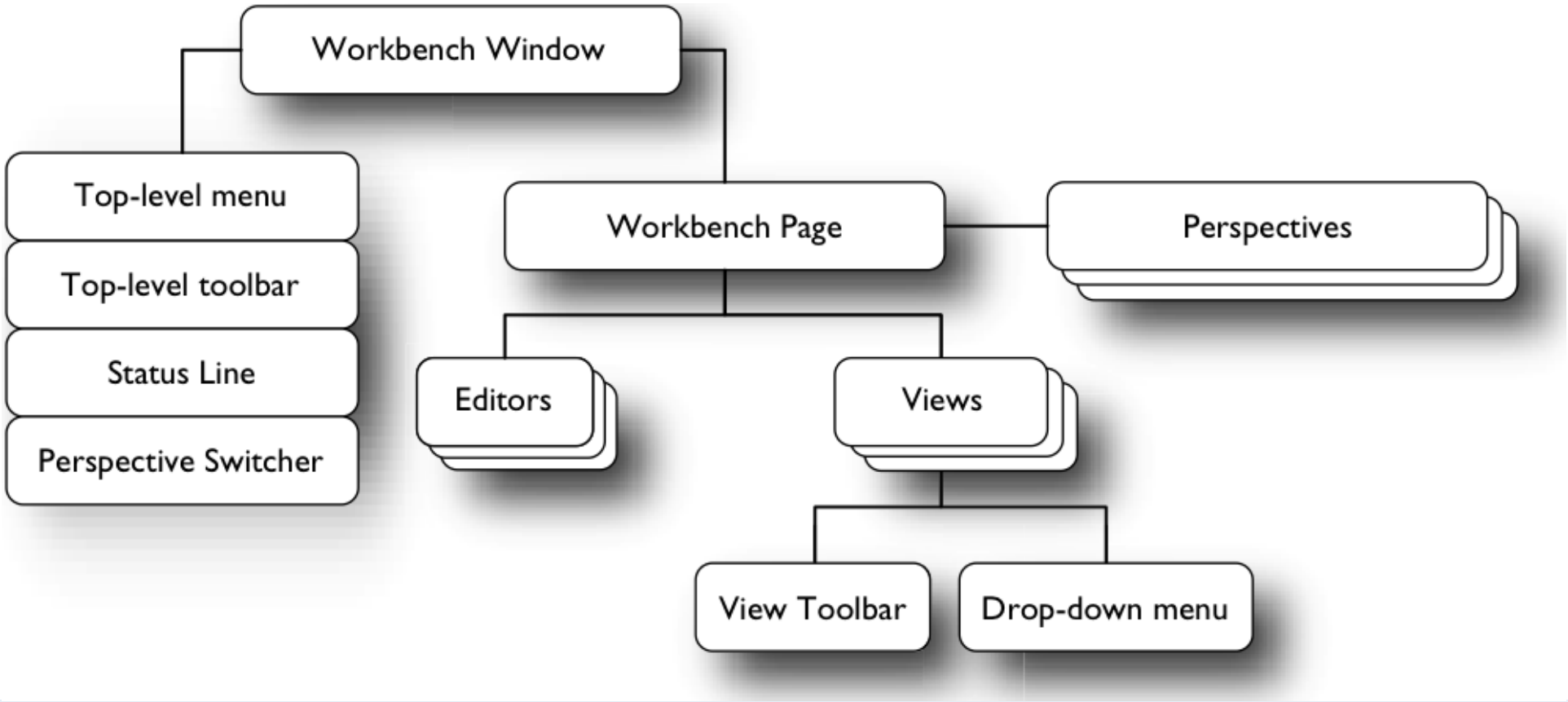
```
public class Perspective implements IPerspectiveFactory {

  public void createInitialLayout(IPageLayout layout) {
    layout.addView(ContactsView.ID, IPageLayout.LEFT, 0.33f,
      layout.getEditorArea());
    IViewLayout contactsView = layout.getViewLayout(ContactsView.ID);
    contactsView.setCloseable(false);
  }

}
```

IPageLayout API

- Use `layout.addStandaloneView( ... )` to
  - make the view unmovable
  - make the view uncloseable
  - hide the title (optional)
- Use `layout.setFixed( ... )` to
  - make all views in the perspective unmovable
  - make all views in the perspective uncloseable
  - **Important:** call this before anything else!
- Use `layout.createFolder( ... )` to get an `IFolderLayout` that stacks views

Perspectives, Workbench Pages and Workbench Windows

- Every workbench window has one workbench page
- The workbench page contents are layouted by a perspective



Saving and restoring state

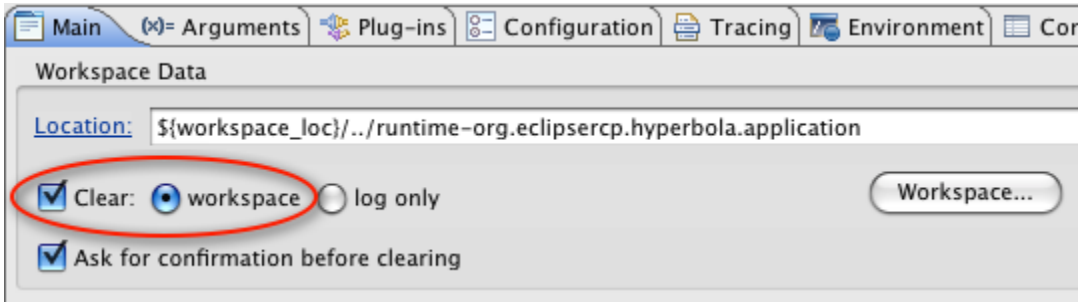
- State of workbench layout can be saved automatically

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {
  ...
  public void initialize( IWorkbenchConfigurer configurer ) {
    configurer.setSaveAndRestore( true );
  }
}
```

- To save and restore state in a view, use the `IMemento` given in
  - `IViewPart.init( IViewSite, IMemento )` and
  - `IViewPart.saveState( IMemento )`

Clear Workspace Data

- Perspective settings are saved in the workspace metadata directory
- Saved settings override code that advises settings
- To see changes in your code, clear the workspace data
  - In your run configuration, select **clear workspace**



Tasks

- Create a view `ContactsView`
  - Use the Plug-in Manifest Editor to create the extension
  - Integrate it in the perspective
  - optional: import and use icons/groups.gif
- Play with the possibilities to place the view relative to the editor area
  - Make the view non-closeable
- Increase the initial window size

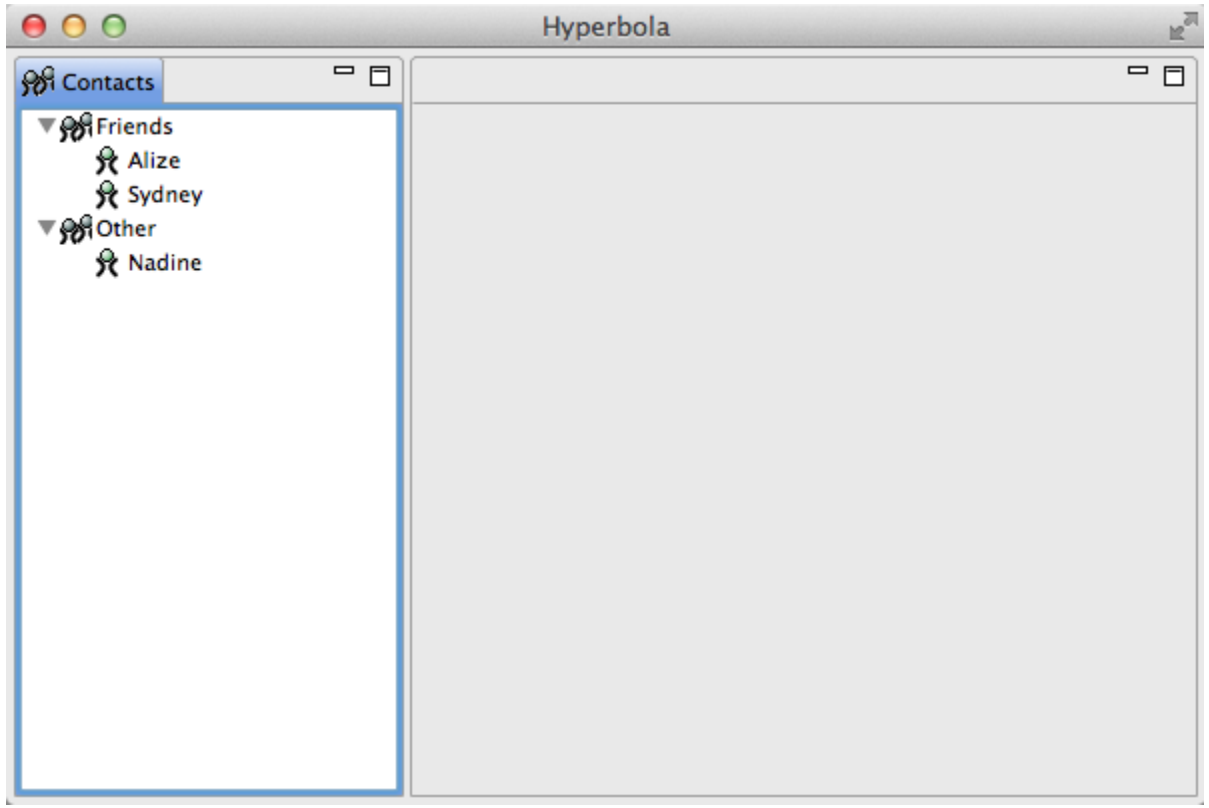
Solution

- If in doubt how to start, use **1.1 Applications**
- If unsure how to proceed, compare to **2.1 Views**

Pointers

- **Platform Plug-in Developer Guide > Programmer's Guide > Plugging into the workbench**

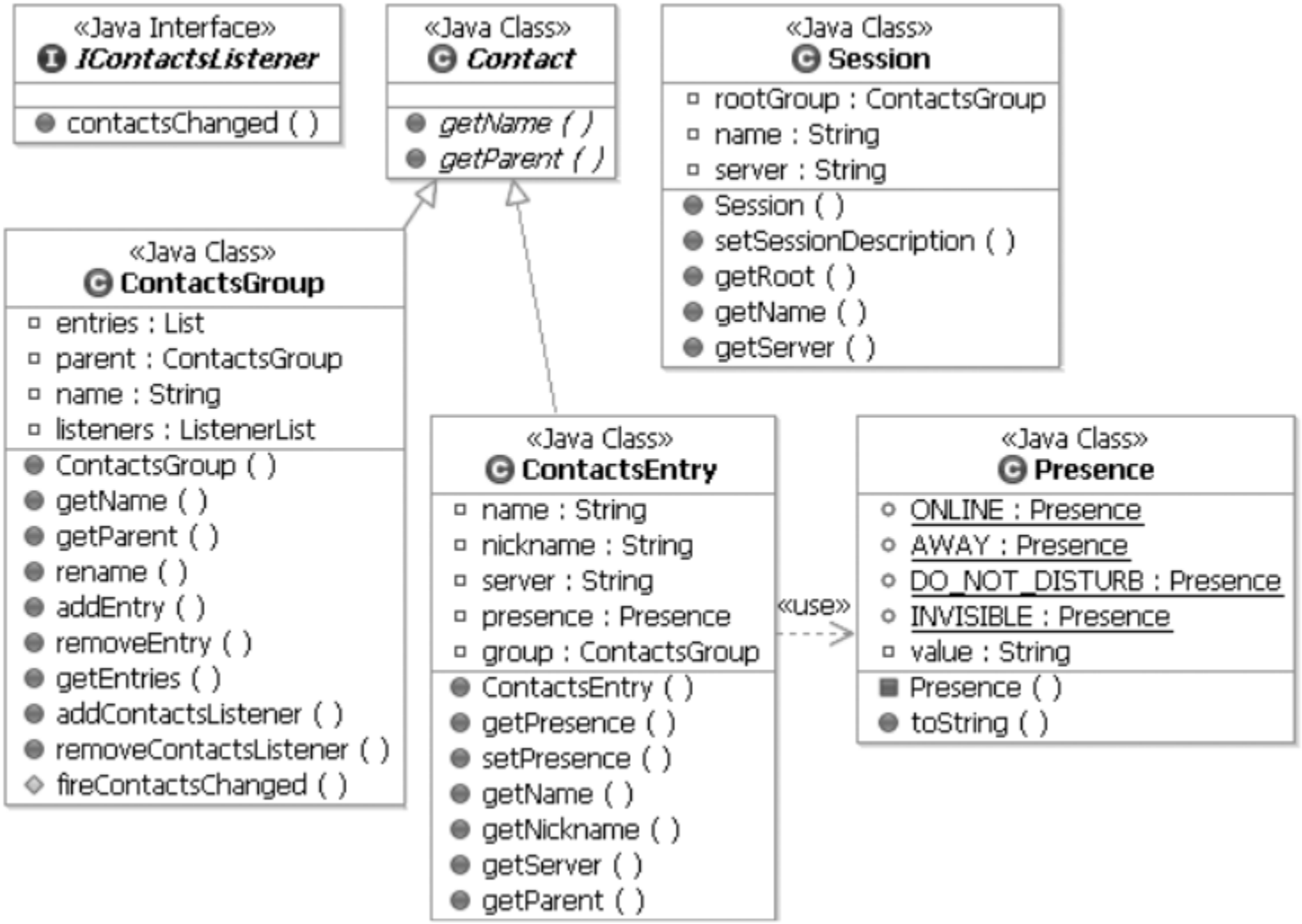
JFace Viewers



Goals

- View from previous chapter is still empty
- We want to fill it with contacts in a tree-like structure
- We use
  - A JFace `TreeViewer`
  - A content provider to populate the tree with elements
  - A label provider to display labels and images

Hyperbola Model



JFace Viewers

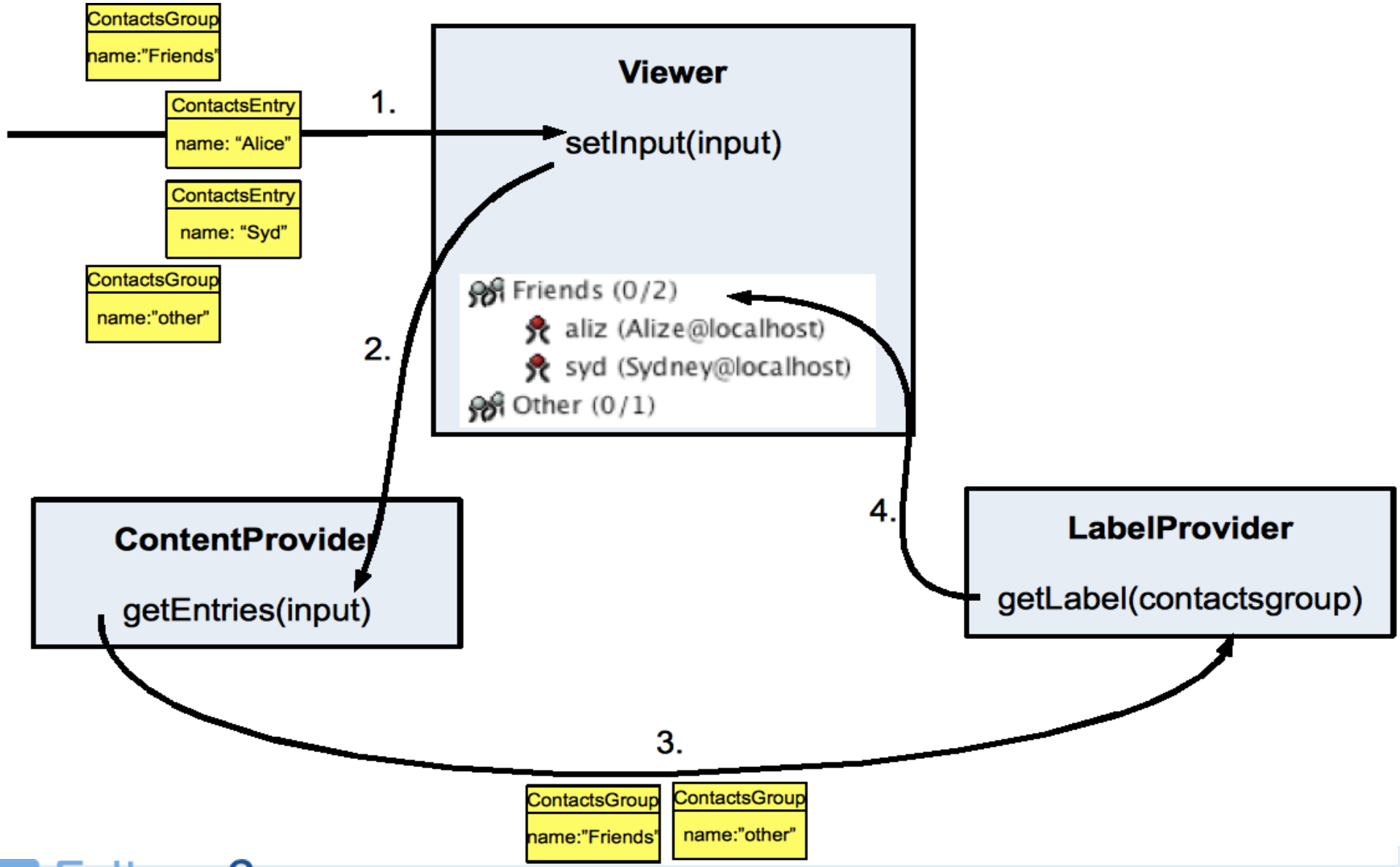
- From a users POV, lists, tables and trees have much in common:
  - Population with objects
  - Selection
  - Sorting
  - Filtering
  - Object update
- SWT does not handle these abstractions
- JFace is a higher abstraction level and does provide these abstractions in the form of **Viewers**

Viewer types

JFace provides viewers for SWT widgets that display collections

- `List` → `org.eclipse.jface.viewers.ListViewer`
- `Tree` → `org.eclipse.jface.viewers.TreeViewer`
- `Table` → `org.eclipse.jface.viewers.TableViewer`
- `Combo` → `org.eclipse.jface.viewers.ComboViewer`

Viewer Concepts



View creation

- `IWorkbenchPart.createPartControl(Composite)` is called for view creation
- This code creates and populates the view:

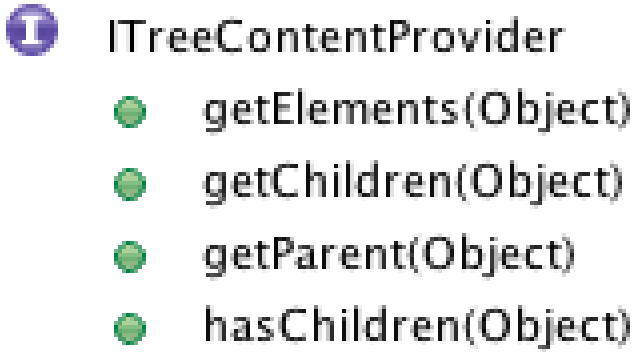
```
org.eclipsercp.hyperbola.ContactsView
public void createPartControl(Composite parent) {
   initializeSession(); // temporary tweak to build a fake model
   treeViewer = new TreeViewer(parent,
                               SWT.BORDER | SWT.MULTI | SWT.V_SCROLL);
   treeViewer.setContentProvider(new HyperbolaContentProvider());
   treeViewer.setLabelProvider(new HyperbolaLabelProvider());
   treeViewer.setInput(session.getRoot());
}
```

- `setInput(...)` triggers the display of the elements

ContentProvider types

- For lists and tables, an `IStructuredContentProvider` is required
  - Tip: use `ArrayContentProvider` which works for arrays and collections.

- For trees, a `ITreeContentProvider` is required
  - Provides methods for navigating tree-structured data



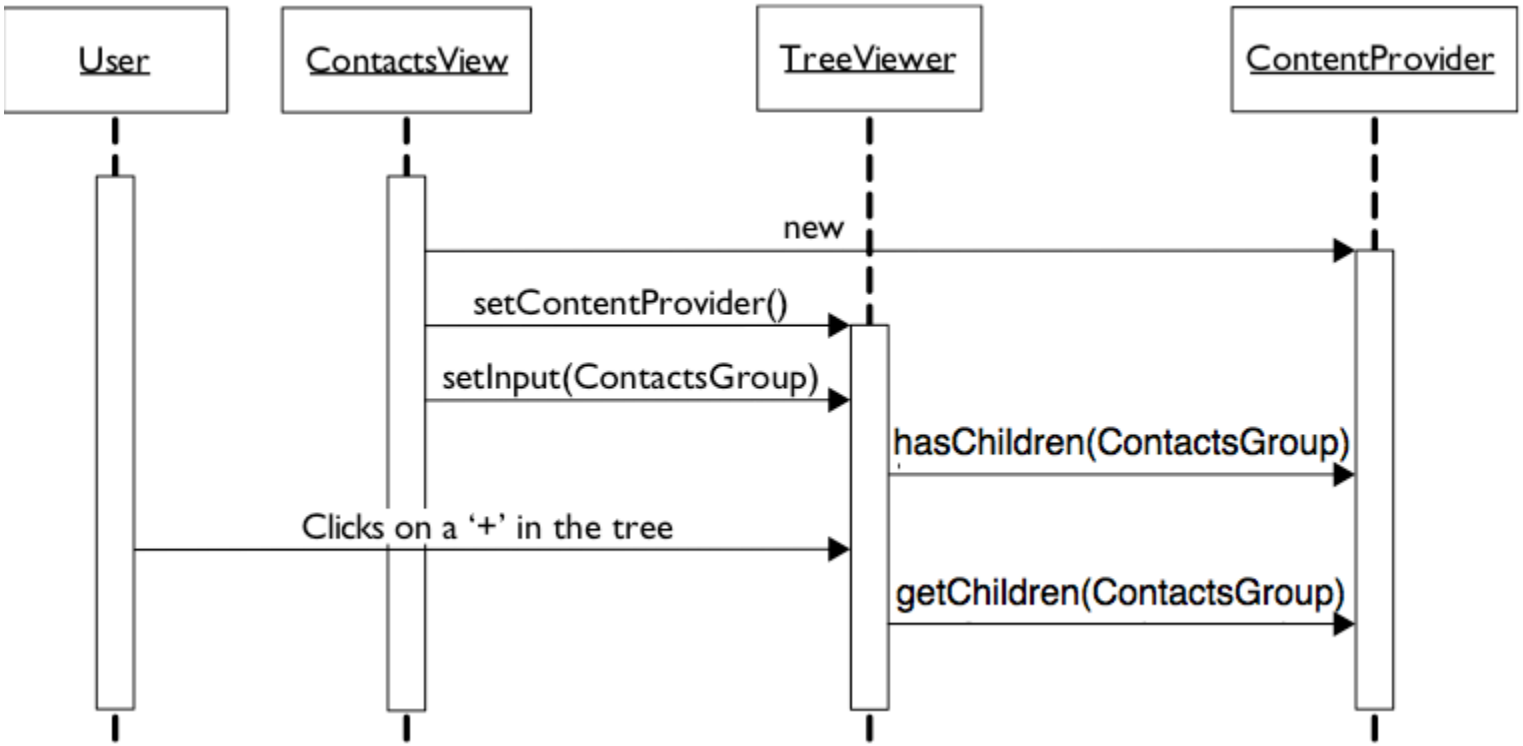
HyperbolaContentProvider and the input

- `getElements(Object)` is called once with the input
- `getElements(Object)` returns the root elements
- `inputChanged(...)` is called if the viewer input changes (typically to add/remove listeners)

```
org.eclipsercp.hyperbola.HyperbolaContentProvider
public Object[] getElements(Object inputElement) {
  ContactsGroup root = (ContactsGroup) inputElement;
  return root.getEntries();
}

public void inputChanged(Viewer viewer,
                         Object oldInput, Object newInput) {
  // do nothing here
}
```

HyperbolaContentProvider and children

- Each result from `getElements(...)` is passed to `hasChildren(...)`
- `hasChildren(...)` determines if **+** is shown in the tree
- Expanding an element will call `getChildren(...)`



HyperbolaContentProvider and children (cont.)

org.eclipsercp.hyperbola.HyperbolaContentProvider

```
public boolean hasChildren(Object element) {
  if (element instanceof ContactsGroup) {
    ContactsGroup group = (ContactsGroup) element;
    return group.getEntries().length > 0;
  }
  return false;
}

public Object[] getChildren(Object parentElement) {
  if (parentElement instanceof ContactsGroup) {
    ContactsGroup group = (ContactsGroup) parentElement;
    return group.getEntries();
  }
  return new Object[0];
}
```

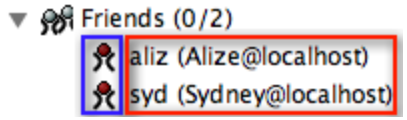
HyperbolaContentProvider and parents

- Some operations require to calculate the path of a child element to the root
  - i.e. `TreeViewer.reveal(...)`
- `getParent(...)` is required for those operations

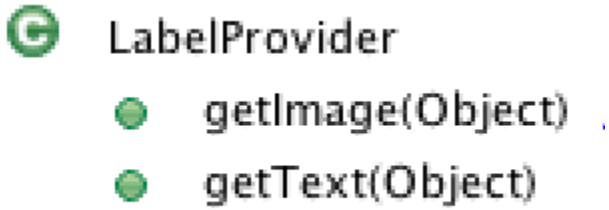
```
org.eclipsercp.hyperbola.HyperbolaContentProvider
public Object getParent(Object element) {
  Contact contact = (Contact) element;
  return contact.getParent();
}
```

The LabelProvider

- The LabelProvider presents the elements on the screen
  - Image
  - Label



- A label provider must implement `IBaseLabelProvider`
- Extend the convenience class `LabelProvider` and override
  - `getImage(Object)` and
  - `getText(Object)`



HyperbolaLabelProvider.getText(...)

- `getText(Object)` returns the text for each element or child of the content provider:

```
org.eclipsercp.hyperbola.HyperbolaLabelProvider
public String getText(Object element) {
  Contact contact = (Contact) element;
  return contact.getName();
}
```

HyperbolaLabelProvider.getImage(...)

- `getImage(Object)` returns an image for each element or child of the content provider
  - return `null` for no image
- This example uses cached images from the plugin's image registry:

```
org.eclipsercp.hyperbola.HyperbolaLabelProvider
public Image getImage(Object element) {
  boolean isGroup = element instanceof ContactsGroup;
  String key = isGroup ? IImageKeys.GROUP : IImageKeys.ONLINE;
  return Activator.getDefault().getImageRegistry().get(key);
}
```

Adding Images

- Images for viewers, actions, etc. should be 16x16-pixel GIF or PNG with transparent background
- Images are typically located in a folder `icons` below the plug-in root
- A class provides the paths relative to the plug-in as constants:

```
org.eclipsercp.hyperbola.IImageKeys
public interface IImageKeys {
  public static final String GROUP = "icons/groups.gif";
  public static final String ONLINE = "icons/online.gif";
  ...
}
```

Image representations

- Images are represented as `Image` or `ImageDescriptor`
- **`Image`**
  - Object containing graphical data
  - Uses system resources
  - Must be disposed
- **`ImageDescriptor`**
  - Lightweight representation
  - Knows where the image is
  - Can create `Image` objects

Creating Image objects

- Use this helper, to create an `Image` from an `ImageDescriptor`
  - works for any plug-in, regardless of format (.jar or folder)

```
ImageDescriptor descr = AbstractUIPlugin.imageDescriptorFromPlugin(
  Activator.PLUGIN_ID, "/icons/online.gif");
Image image = descr.createImage();
```

- Each call to `descr.createImage()` returns a new image!
- Each image has to be freed up, via `image.dispose()`!

Cached images with ImageRegistry

- `ImageRegistry` caches images and disposes them when the plug-in is stopped.
- To use this:
  - let your activator extend `AbstractUIPlugin`
  - override `initializeImageRegistry(...)` to register images

```
public class Activator extends AbstractUIPlugin {

  public static final String PLUGIN_ID = "org.eclipsercp.hyperbola";

  protected void initializeImageRegistry(ImageRegistry reg) {
    reg.put(IImageKeys.GROUP,
            imageDescriptorFromPlugin(PLUGIN_ID, IImageKeys.GROUP));
  }

}
```

Cached images with ImageRegistry (cont.)

To retrieve a shared Image from the image registry:

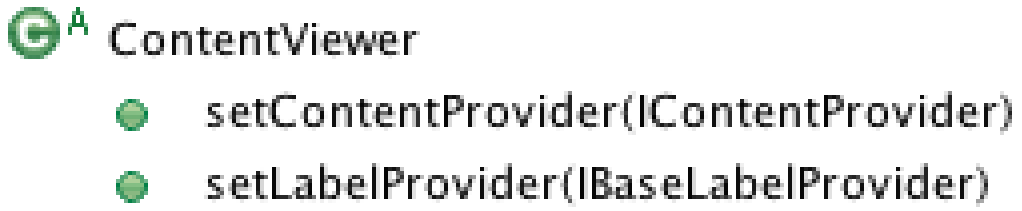
```
Activator.getDefault().getImageRegistry()
  .get(IImageKeys.GROUP);
```

To get an ImageDescriptor:

```
Activator.getDefault().getImageRegistry()
  .getDescriptor(IImageKeys.GROUP);
```

Viewer API

- There are different content and label provider specializations available for different tasks
  - This is not reflected in the methods of `ContentViewer` and subclasses



- General rules for provider types:
  - To fill rows: `IStructuredContentProvider`
  - To provide a tree structure: `ITreeContentProvider`
  - To display multiple columns for an element (in trees or tables): `ITableLabelProvider`

Example: Table and TableViewer

- Design the table as SWT widget

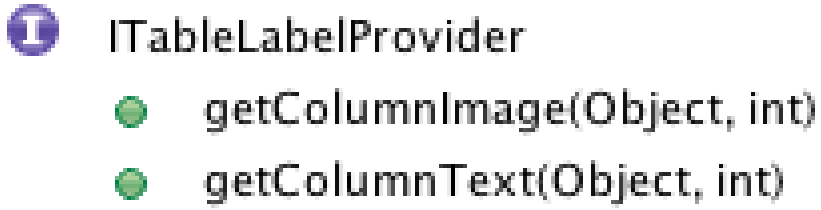
```
Table t = new Table(parent, SWT.MULTI | SWT.FULL_SELECTION);
TableColumn col1 = new TableColumn(t, SWT.RIGHT);
col1.setText("Col 1");
TableColumn col2 = new TableColumn(t, SWT.LEFT);
col2.setText("Col 2");
```

- Layouting a table is easiest done with the TableColumnLayout

```
TableColumnLayout tcl = new TableColumnLayout();
tcl.setColumnData(col1, new ColumnWeightData(2));
tcl.setColumnData(col2, new ColumnWeightData(1));
parent.setLayout(tcl); // yes, really at parent
```

Example: Table and TableViewer (cont.)

- The viewer does only provide the elements and display the cell contents
- It's important that the label provider is a `ITableLabelProvider`




```
TableViewer tv = new TableViewer(t);
tv.setContentProvider( myStructuredContentProvider);
tv.setLabelProvider(myTableLableProvider);
tv.setInput(input);
```

Viewers ease working with the table/tree/list *content*. They do not influence the *layout* of the actual table/tree/list widget. This still is the responsibility of SWT.

Tasks

- Copy these files from **3.1 JFace Viewers** into your workspace:
  - the org.eclipsercp.hyperbola.model package
  - the /icons folder
  - the method ContactsView.initializeSession()
- Create a tree viewer in the Contacts View
- Create a content provider and implement its methods
- Create a label provider and implement its methods
  - Let it extend `LabelProvider`
  - Use the Activators image registry for accessing images

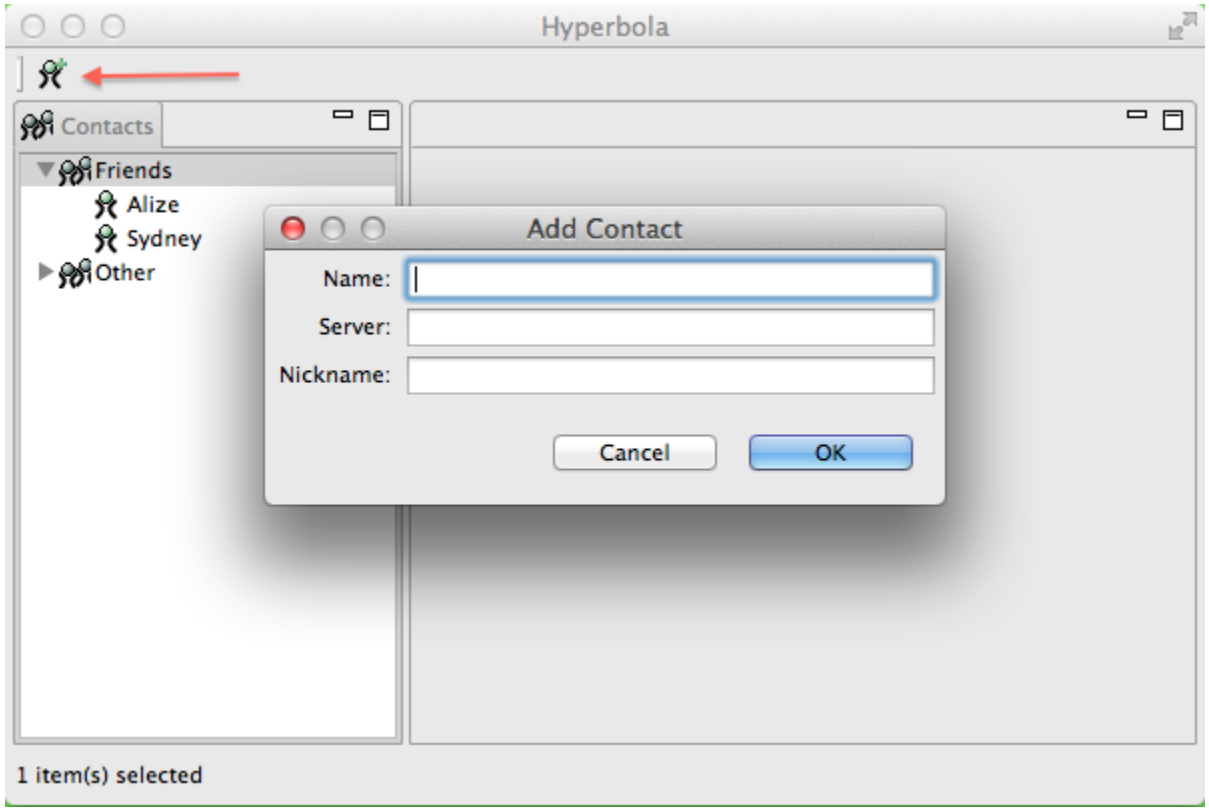
Solution

- If in doubt how to start, use **2.1 Views**
- If unsure how to proceed, compare to **3.1 JFace Viewers**

Pointers

- RCP Book, Chapters 16, 18, 19
- More on SWT: http://www.eclipse.org/swt/, look for snippets
- Building a table viewer: http://www.eclipse.org/articles/Article-Table-viewer/table_viewer.html
- Eclipse Help > Platform Plug-in Developer Guide > Programmer's Guide > Standard Widget Toolkit
- Eclipse Help > Platform Plug-in Developer Guide > Programmer's Guide > JFace UI Framework
- An article about adapters and adapter factories: http://www.eclipse.org/articles/article.php?file=Article-Adapters/index.html
- Another article about adapters and adapter factories: http://www.eclipse.org/resources/resource.php?id=407

Actions



Goals

- Place standard actions in the menu
- Create custom actions and show them in the menu and toolbar
- Use the selection service to react to selections
- Use the status line

Actions

- An *Action* is a visible element that allows users to initiate a unit of work
- An action can be triggered through:
  - a menu
  - toolbar
  - a key sequence

Enabling the MenuBar and CoolBar

- The method `WorkbenchWindowAdvisor.preWindowOpen()` controls which `WorkbenchWindow` elements are visible

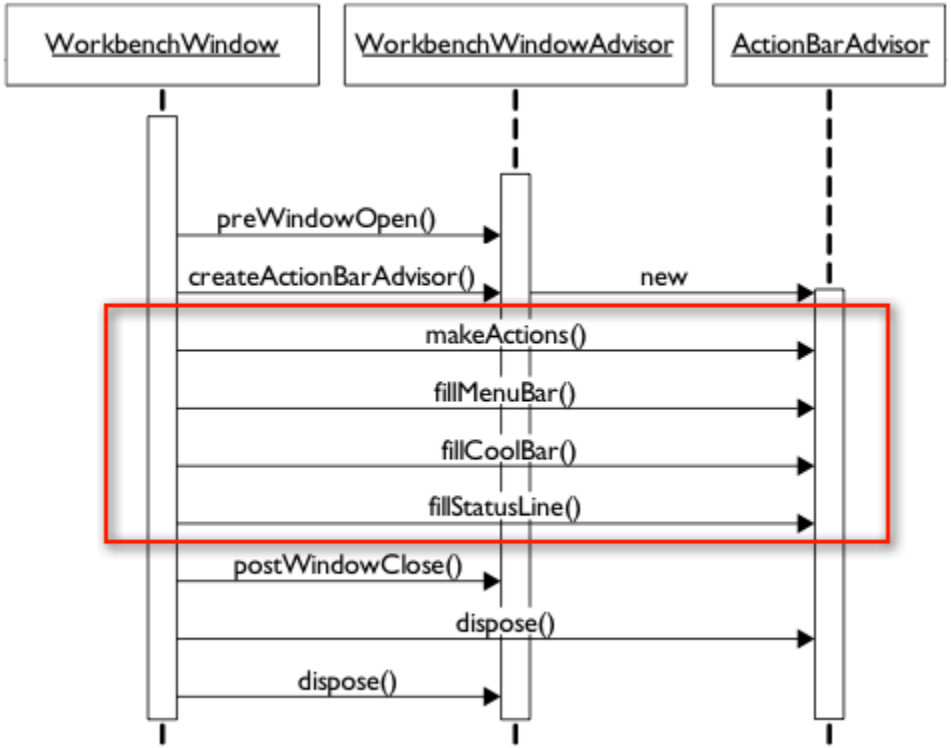
```
org.eclipsercp.hyperbola.ApplicationWorkbenchWindowAdvisor
 public void preWindowOpen() {
   IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
   ...
   configurer.setShowMenuBar( true );
   configurer.setShowCoolBar( true );
   ...
 }
```

- Menu structure is important.
  - Most users browse the top-level menu to get to know the application
- The toolbar should contain the most frequently used actions

Creating the Top-Level Menu

- The `MenuBar` is only shown when it contains menus
- The menus are created in the `ActionBarAdvisor`

- Actions are created before the menus
  - The method `makeActions()` is called before `fill...()`
  - Advisor creates the action once
  - Advisor reuses the action in different places



Creating standard actions

- Standard actions are created by the `ActionFactory`
- They are preconfigured with an icon, name and id
- Actions must be registered with the workbench
  - Enables disposal when the workbench window is closed
  - Enables key bindings

```
org.eclipsercp.hyperbola.ApplicationActionBarAdvisor
 protected void makeActions( IWorkbenchWindow window ) {
   exitAction = ActionFactory.QUIT.create( window );
   register( exitAction );
   aboutAction = ActionFactory.ABOUT.create( window );
   register( aboutAction );
 }
```

Placing actions

- `fillMenuBar(...)` gets the root menu manager as a parameter
- Each menu is created with a `new MenuManager(...)`
  - Constructor parameters are Strings
  - The 1st is shown in the UI
  - The 2nd creates a logical path (`menubarPath`)
- `IMenuManager.add(...)` adds Actions or submenus to a menu

```
org.eclipsercp.hyperbola.ApplicationActionBarAdvisor
 protected void fillMenuBar( IMenuManager menuBar ) {
   MenuManager fileMenu = new MenuManager( "&File", "file" );
   fileMenu.add( exitAction );
   menuBar.add( fileMenu );

   MenuManager helpMenu = new MenuManager( "&Help", "help" );
   helpMenu.add( aboutAction );
   menuBar.add( helpMenu );
 }
```

More on Menu Managers

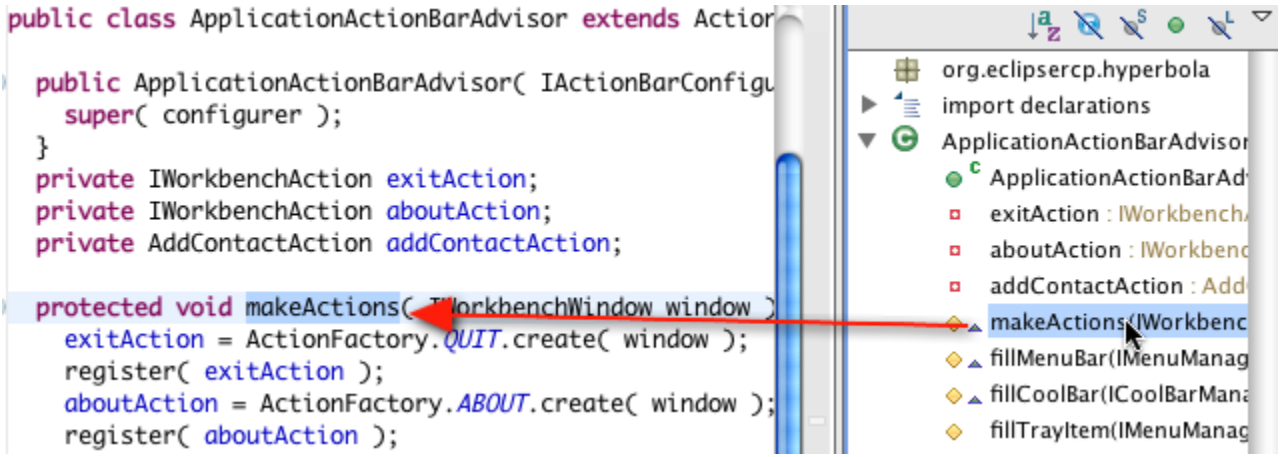
- A menu manager is responsible for
  - Keeping track of actions and sub-menus
  - Creating logical structures of actions with `GroupMarker` (and `Separator`)
- The strings for Menu and GroupMarker form the `menubarPath`
  - used in the plugin.xml to add actions to a predefined place in a menu

```
org.eclipsercp.hyperbola.ApplicationActionBarAdvisor
 protected void fillMenuBar( IMenuManager menuBar ) {
   MenuManager fileMenu = new MenuManager( "&File", "file" );
   fileMenu.add( new GroupMarker("additions") );
   ...
 }
```

Selection Service

- The workbench window knows the current selection
  - There is only one selection at a time
- Everybody can add selection listeners to the selection service
- Every page has one selection provider
  - The current selection is the selection of the active part

Selection listeners must be removed when it's associated UI is disposed. Not removing listeners is a common mistake that leads to memory leaks and huge error log files.



Creating a custom Action

- Some actions are enabled all the time, i.e. **Hyperbola** > **Exit**
- Other actions are enabled only in certain workbench states
  - i.e. based on current selection

- Such an action will
  - add a selection listener to be notified of selection events
  - react on the selection and remember it
  - work with the selected element(s) in `run()`
  - remove the selection listener when it is disposed

Being an Action

- Before anything can happen, the action has to integrate itself
  - Set an ID for identification within the workbench window (see `ActionBarAdvisor.register(IAction)`)
  - Set the name and tooltip for identification by the user
  - Set an image

```
org.eclipsercp.hyperbola.AddContactAction
 public AddContactAction( IWorkbenchWindow window ) {
   this.window = window;
   setId( AddContactAction.class.getName() );
   setText( "&Add Contact..." );
   setToolTipText( "Add a contact to your contacts list." );
   setImageDescriptor(
       AbstractUIPlugin.imageDescriptorFromPlugin(
           "org.eclipsercp.hyperbola",
           "icons/add_contact.gif" ) );
}
```

Life cycle of selection listeners

- The action implements the interface `ISelectionListener` and adds itself to the `ISelectionService` in the constructor
- `IWorkbenchAction` adds a `dispose()` method that is called at an appropriate time

```
org.eclipsercp.hyperbola.AddContactAction
public class AddContactAction extends Action
 implements ISelectionListener, IWorkbenchAction {

 public AddContactAction( IWorkbenchWindow window ) {
   ...
   window.getSelectionService().addSelectionListener( this );
 }

 public void dispose() {
   window.getSelectionService().removeSelectionListener( this );
 }
 ...
```

Reacting to selections

- The `ISelectionListener` gets selections via `selectionChanged(...)`
- Enablement is based on the type of the selected element

```
org.eclipsercp.hyperbola.AddContactAction
 public void selectionChanged( IWorkbenchPart part,
                               ISelection selection ) {
   // Selection containing ContactsGroup elements
   if( selection instanceof IStructuredSelection ) {
     sSelection = ( IStructuredSelection )selection;
     setEnabled( sSelection.size() == 1
                 && sSelection.getFirstElement()
                    instanceof ContactsGroup );
   } else {
     // Other selections
     setEnabled( false );
   }
 }
```

Why does this work?

- The action could bind itself just to the view that it is interested in
- Instead, it asked the workbench to be notified about *all* selections
  - This makes the action independent from the view
  - It's the selection that it's interested in, not the view
- The selection must be generated someplace
- JFace viewers are selection providers

```
org.eclipsercp.hyperbola.ContactsView
public class ContactsView extends ViewPart {

 public void createPartControl( Composite parent ) {
   treeViewer = new TreeViewer( parent, SWT.BORDER | SWT.MULTI  );
   getSite().setSelectionProvider( treeViewer );
   ...
 }
```

Action.run()

- When everything is set up, the user can click the action
- Open a dialog and evaluate it's results

```
org.eclipsercp.hyperbola.AddContactAction
 public void run() {
   AddContactDialog dialog = new AddContactDialog( window.getShell() );
   int code = dialog.open();
   if( code == Window.OK ) {
     ContactsGroup group
       = ( ContactsGroup ) sSelection.getFirstElement();
     ContactsEntry entry = new ContactsEntry( group,
                                              dialog.getName(),
                                              dialog.getNickname(),
                                              dialog.getServer() );
     group.addEntry( entry );
   }
```

The aspects of the action

- The elements for successful integration
  - Be an action with ID, name, image and `run()`
  - Work with the selections
  - Remove added listeners

Adding the custom action

- Adding the custom action is similar to adding the standard actions:

```
org.eclipsercp.hyperbola.ApplicationActionBarAdvisor
public class ApplicationActionBarAdvisor extends ActionBarAdvisor {
 ...
 protected void makeActions( IWorkbenchWindow window ) {
   exitAction = ActionFactory.QUIT.create( window );
   register( exitAction );
   ...
   addContactAction = new AddContactAction( window );
   register( addContactAction );
 }
```

- Update `fillMenuBar(...)` and `fillCoolBar(...)` respectively

Customizable Toolbars

- The Eclipse toolbar is implemented in terms of an SWT `CoolBar`
- `ToolBarManager`s are added to the `ICoolBarManager`
  - They show up as separate `CoolItem` groups
  - Can be positioned separately by the user

Separators

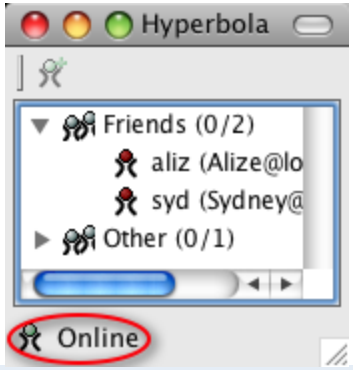
- A toolbar itself can have separators
  - The Strings in separators form the `toolbarPath`
  - used in the plugin.xml to add actions to a predefined place in a toolbar
  - you can also use `GroupMarker`

```
org.eclipsercp.hyperbola.ApplicationActionBarAdvisor
 protected void fillCoolBar( ICoolBarManager coolBar ) {
   IToolBarManager toolbar = new ToolBarManager();
   toolbar.add( addContactAction );
   toolbar.add( new Separator("additions") );
   coolBar.add( toolbar );
   // coolBar.add(new ToolBarContributionItem(toolbar, "hyperbola"));
 }
```

Showing the Status Line

- The status line can be customized
- It must be enabled in `WorkbenchWindowAdvisor.preWindowOpen()`

```
org.eclipsercp.hyperbola.ApplicationWorkbenchWindowAdvisor
 public void preWindowOpen() {
   IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
   ...
   configurer.setShowStatusLine( true );
   ...
 }
```



Adding to the Status Line

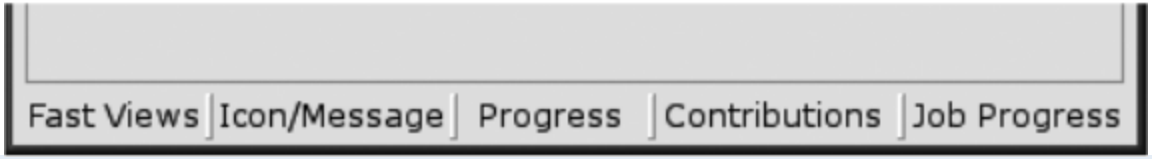
- The status line can be accessed via the `IStatusLineManager`
  - Accessible via `getViewSite()` or `getEditorSite()`

ContactsView.java

```
public void createPartControl(Composite parent) {
  ...
  treeViewer.addSelectionChangedListener(new ISelectionChangedListener()
  {
    public void selectionChanged(SelectionChangedEvent event) {
      IStatusLineManager statusLine
        = getViewSite().getActionBars().getStatusLineManager();
      IStructuredSelection sSelection
        = (IStructuredSelection) event.getSelection();
      if (sSelection.isEmpty()) {
        statusLine.setMessage("");
      } else {
        statusLine.setMessage(sSelection.size() + " item(s) selected");
      }
  }});
```

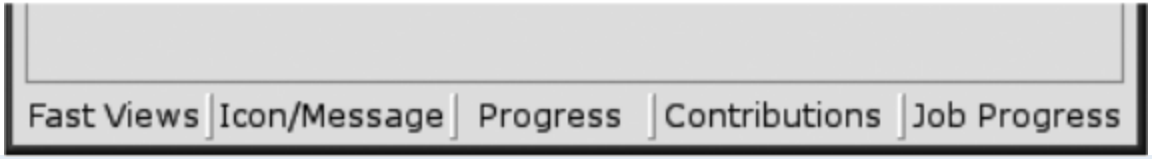
Status line areas

Fast Views	Fast views are removed from the window, can be shown quickly
Icon/ Message	Just an icon and a message
Progress	shows modal progress
Contributions	Reserved for Workbench advisor and active part contributions
Jobs progress	Progress area for jobs, see `IWorkbenchWindowConfigurer .setShowProgressIndicator(...)`



Status line areas (cont.)

- The status line is a shared resource
- Any plug-in can contribute to the status line
- Additions can be made using `IStatusLineManager.add(IContributionItem)`



Declarative Actions

- Actions can be added declarative by contributing to
  - `org.eclipse.ui.actionSets`
  - `org.eclipse.ui.viewActions`
  - `org.eclipse.ui.editorActions`
  - `org.eclipse.ui.popupMenus`
- These extension points require specialized subclasses of `ActionDelegate`
- The *Command Framework* has a more flexible way to contribute items declaratively

Tasks (1/2)

- Add a top-level menu and toolbar
- Add the standard **File** > **Exit** and **Help** > **About** actions
  - Add **AddContactDialog.java** from the solution to your workspace
- Create a custom action to add a contact and place it in the toolbar and menu
  - Extend the class `Action` adding the interfaces `ISelectionListener` and `IWorkbenchAction`
  - Add a constructor with an `IWorkbenchWindow` argument. Initialize the id, text and image for the action. Add the action to the selection service
  - Implement `dispose()`: remove the action from the selection service
  - Implement `selectionChanged()`: enable the action if one `ContactsGroup` element is selected
  - Implement `run()`: open the dialog and add the new entry to the group

Tasks (2/2)

- Make the TreeViewer a selection provider and refresh the tree when the model is modified
  - Add the following code to `ContactsView.createPartControl(...)`:

```
getSite().setSelectionProvider(treeViewer);
session.getRoot().addContactsListener(new IContactsListener() {
 public void contactsChanged(ContactsGroup contacts,
   ContactsEntry entry) {
  treeViewer.refresh(contacts);
 }
});
```

- Optional: enable the status line and display a message on it

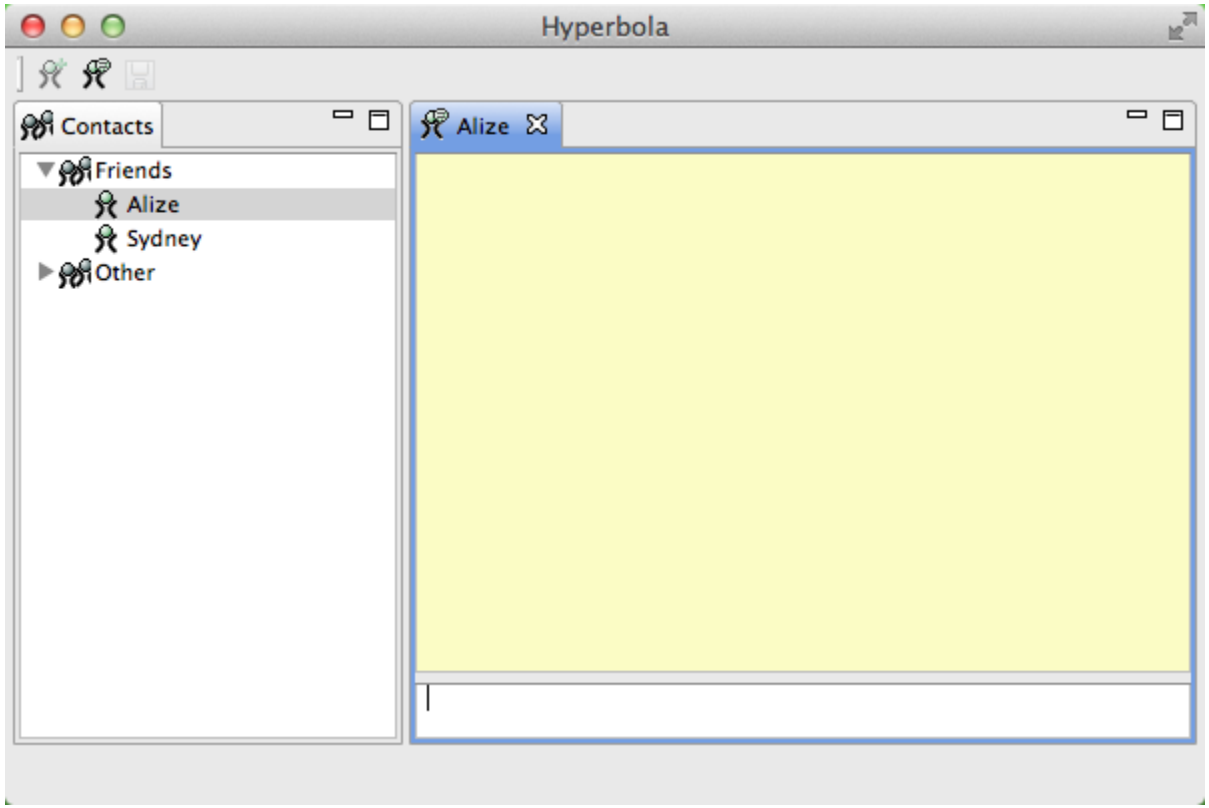
Solution

- If in doubt how to start, use **3.1 JFace Viewers**
- If unsure how to proceed, compare to **4.1 Actions**

Pointers

- Actions, RCP Book, Chapter 17
- Adding Contributions to the Status Line, RCP Book, Section 17.7
- Action definition and placement, RCP Book Chapter 23

Editors



Goals

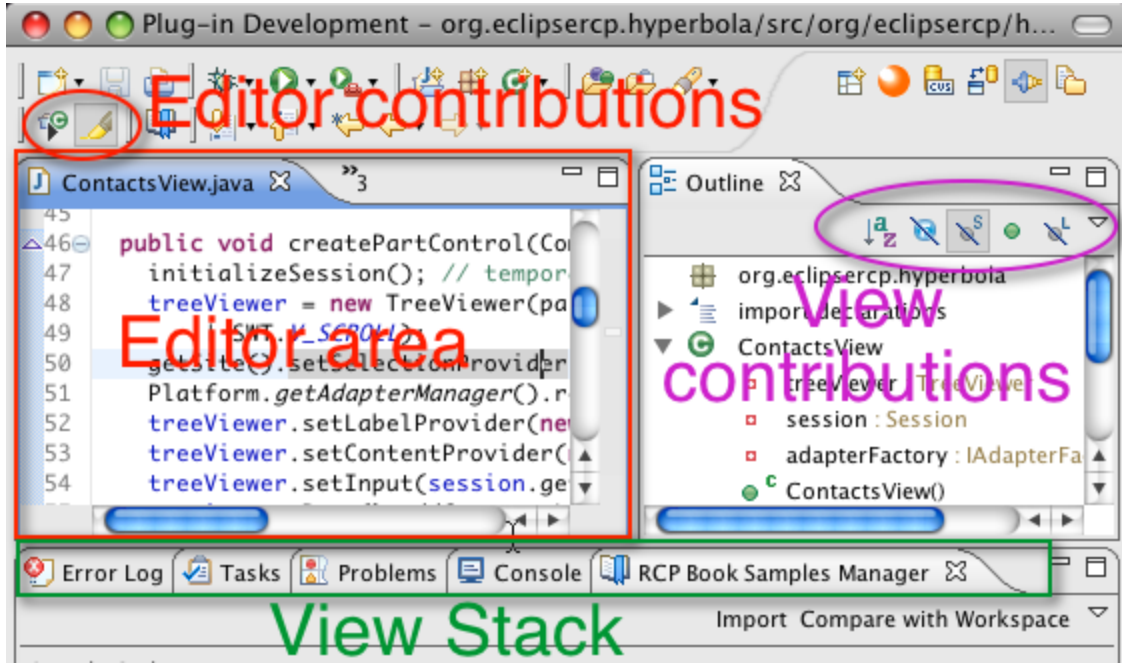
- Differences between Views and Editors
- Learn how to contribute editors
- Learn how to implement editor input and editors
- Find out how to open an editor

Editors in the Workbench

- An editor manipulates domain specific data in a way that is specialized for that data type
- Plug-ins can contribute editors to the workbench
- Editors can use any UI pattern they prefer:
  - Text (e.g. Java editor)
  - Forms (e.g. Plug-in Manifest editor)
  - Lines and Boxes (e.g. UML Editors)

Views and Editors

- Editors are shared between perspectives in the same window.
- Editors and views cannot be mixed in the same stack.
- Views can be detached from a Workbench window.



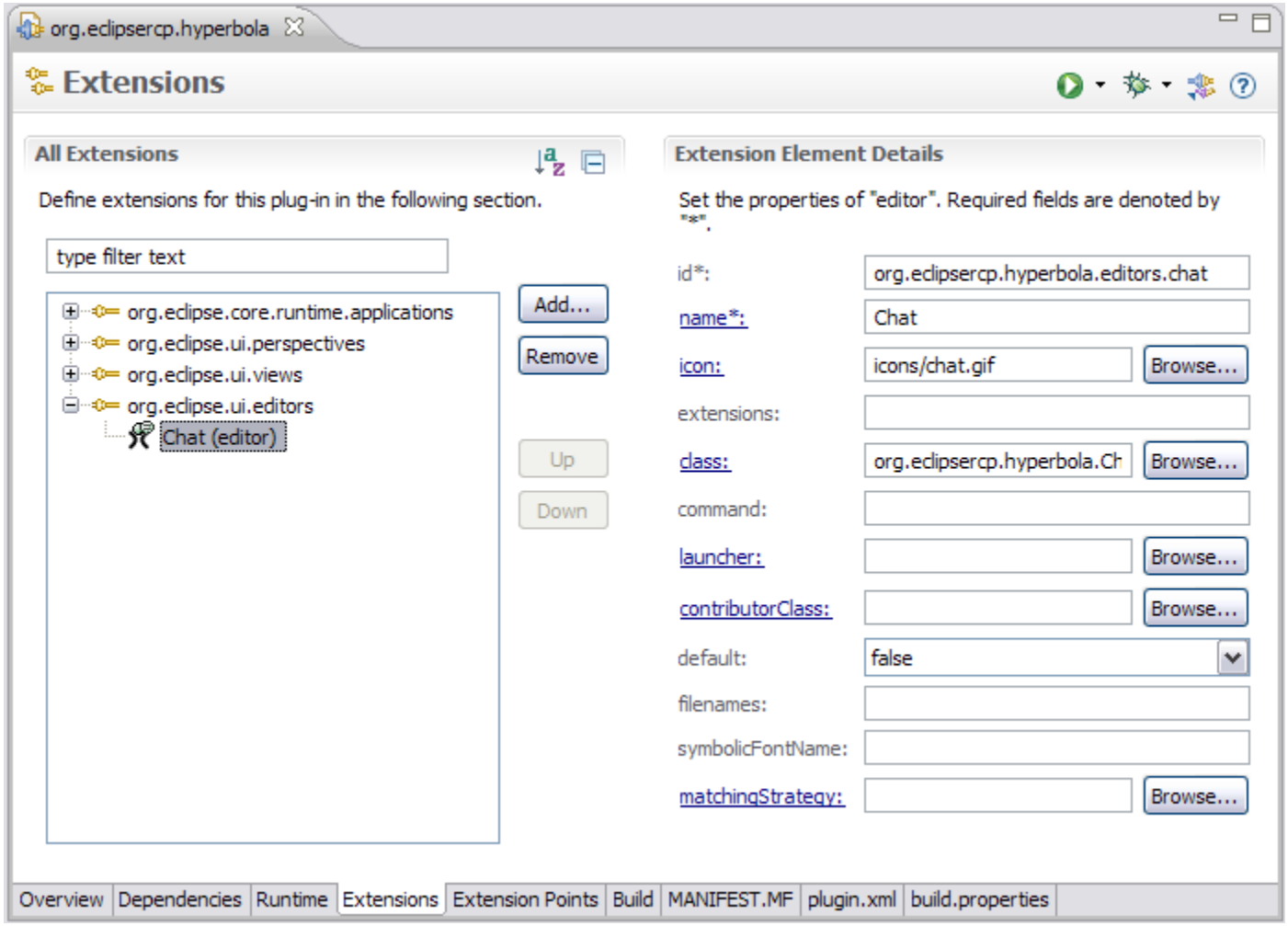
- Views can be shown without a title.
- Editors add contributions to the main toolbar and menu whereas views add contributions to their local toolbar and menu.
- It is possible to ask for the active editor even if the editor does not have focus.

Adding an Editor

- To add an editor extend `org.eclipse.ui.editors`:

```
<extension point="org.eclipse.ui.editors">
 <editor class="org.eclipsercp.hyperbola.ChatEditor"
   default="false"
   icon="icons/chat.gif"
   id="org.eclipsercp.hyperbola.editors.chat"
   name="Chat"/>
</extension>
```

Adding an Editor (cont.)



Implementing an Editor

- By contract, an editor must implement `IEditorPart`
  - Use convenience implementation `EditorPart`
- Editor controls are created in `createPartControl(Composite)`
- Subclasses of `EditorPart` must implement `init(...)`

ChatEditor.java

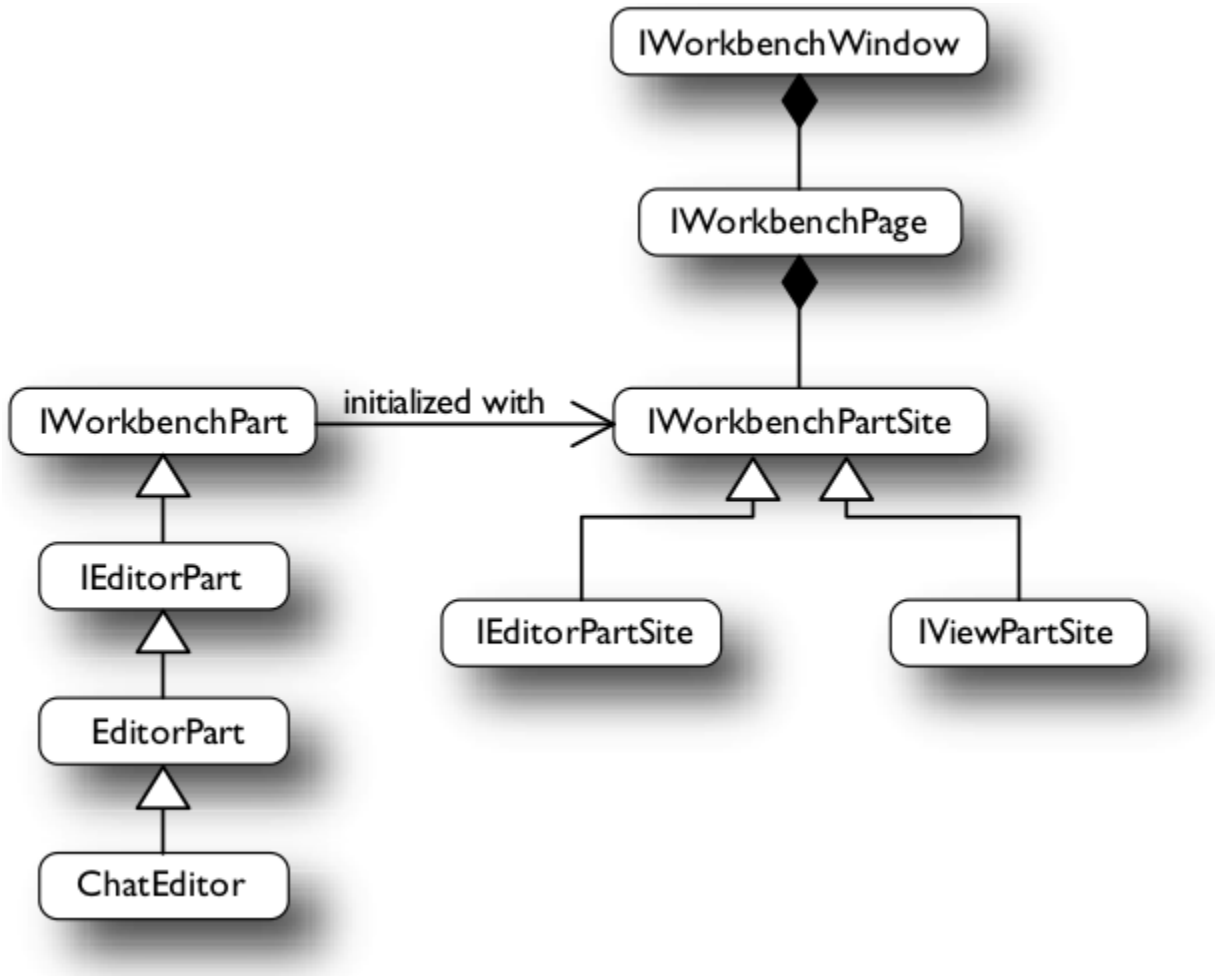
```
public void init( IEditorSite site, IEditorInput input )
            throws PartInitException  {
  setSite( site );
  setInput( input );
  setPartName( input.getName() );
}
```

The life of an editor

The general lifecycle of an editor is:

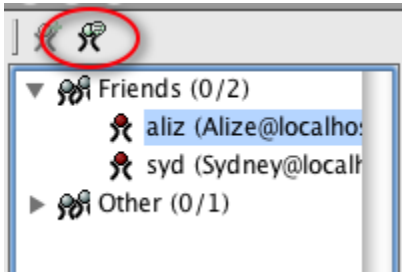
- The Workbench instantiates the editor, creates an editor site, then calls `EditorPart.init(IEditorSite, IEditorInput)`.
- When the editor is made visible, the method `EditorPart.createPartControl(Composite)` is called to create the editor's widgets.
- Once the editor is created, the method `EditorPart.setFocus()` is called.
  - This is also called every time the editor is focused by the user
- When the editor is closed and the contents need to be saved the method `EditorPart.doSave(IProgressMonitor)` is called.
- At the end of the editor's lifecycle, the method `EditorPart.dispose()` is called.

The ChatEditor in context



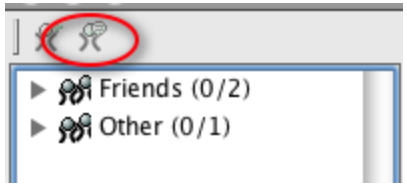
The ChatAction

- Editors are not opened automatically in a perspective, but rather as the direct result of a user action.
- We need a ChatAction that initiates a chat based on the contact selected by the user.
  - The ChatAction is a selection listener that opens the current selection if the user clicks on the action



The ChatAction / Selection listener

- It only makes sense to use the ChatAction when a contact is selected.

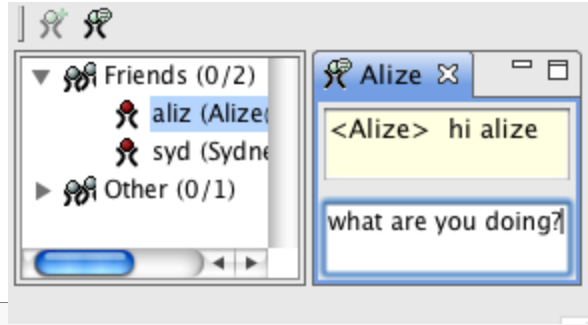


ChatAction.java

```
public void selectionChanged( IWorkbenchPart part,
                              ISelection selection ) {
  if( selection instanceof IStructuredSelection ) {
    sSelection = ( IStructuredSelection )selection;
    setEnabled( sSelection.size() == 1
        && sSelection.getFirstElement() instanceof ContactsEntry );
  } else {
    // Other selections
    setEnabled( false );
  }
}
```

The ChatAction.run()

- The `run()` method creates an input for the selected user and asks the Workbench Page to open a chat editor.



ChatAction.java

```
public void run() {
   ContactsEntry entry
     = ( ContactsEntry )sSelection.getFirstElement();;
   IWorkbenchPage page = window.getActivePage();
   ChatEditorInput input = new ChatEditorInput( entry.getName() );
   try {
     page.openEditor( input, ChatEditor.ID );
   } catch( PartInitException exc ) {
     Bundle bundle = Platform.getBundle( "org.eclipsercp.hyperbola" );
     Platform.getLog( bundle ).log( exc.getStatus() );
   }
 }
```

- The `IWorkbenchPage` has more methods to control editors, such as `saveAll()` or `close()`

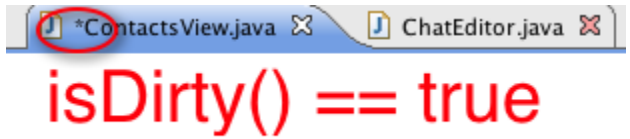
Editor Input

- The editor input is the **lightweight** model for the editor.
  - initialization data for the editor
  - identification information for the Workbench, telling it which editors are open.
- Editor inputs must implement `IEditorInput`
- Typically, an editor input also has methods to access the domain model object(s)

Dirty state

The chat editor does not need to be saved. Other editors might.

- An editor may inform the workbench that some of it's data has been changed
- This can be done by firing a property change event for the constant `IEditorPart.PROP_DIRTY`
  - `firePropertyChange(PROP_DIRTY)`
- The workbench will call `isDirty()` to query the actual dirty state



Dirty state (cont.)

ChatEditor.java

```
public void doSave(IProgressMonitor monitor) {
  // Save to file, etc., here.
  isDirty = false;
  firePropertyChange(PROP_DIRTY);
}

private void sendMessage() {
  ...
  // Mark editor as 'dirty' when sending a message
  isDirty = true;
  firePropertyChange(PROP_DIRTY);
}

public boolean isDirty() {
  return isDirty;
}
```

Dirty state (cont).

- To 'save' an editor, we will need the `ActionFactory.SAVE` action
  - The action is enabled / disabled automatically
  - When the action is clicked, the editor's `doSave(...)` method will be called

ApplicationActionBarAdvisor.java

```
protected void makeActions(IWorkbenchWindow window) {
  ...
  saveAction = ActionFactory.SAVE.create(window);
  register(saveAction);
}

protected void fillCoolBar(ICoolBarManager coolBar) {
  ToolBarManager toolbar = new ToolBarManager();
  ...
  toolbar.add(saveAction);
  ...
}
```

Tasks (1/3)

- Create a custom ChatAction:
  - **Hint:** copy and modify AddContactAction
  - Change the id, text, etc. in the constructor
  - Modify `selectionChanged(...)` to enable the action when one `ContactsEntry` is selected
  - Ignore the `run()` method for now
- Prepare the workbench for the editor to be shown:
  - add the ChatAction to the Hyperbola menu and toolbar
  - optional: increase the default size of the Hyperbola window
- Create an editor input
  - The editor input should store the name of the selected ContactEntry
  - change `getName()` and `getToolTipText()` to return the name
  - optional: implement `equals()` and `hashCode()`. Two inputs with the same name should be considered equal

Tasks (2/3)

- Add a chat editor to the plugin.xml
  - use the `org.eclipse.ui.editors` extension
  - fill out the id, name, icon and class attributes
- Create the editor class
  - in `init(...)` set the site, input and part name
  - add the code for `createPartControl(...)` plus helper methods from the solution
- Modify ChatAction.run() to open the editor
- Try your code out in the workbench
  - What happens if you select the ChatAction twice? Why?

Tasks (3/3)

Optional:

- Add the `ActionFactory.SAVE` action to your toolbar
- Modify the editor's `doSave(...)` method:
  - clear the dirty flag
  - fire the PROP_DIRTY event
- Implement the `isDirty()` method
- Try your code out in the workbench

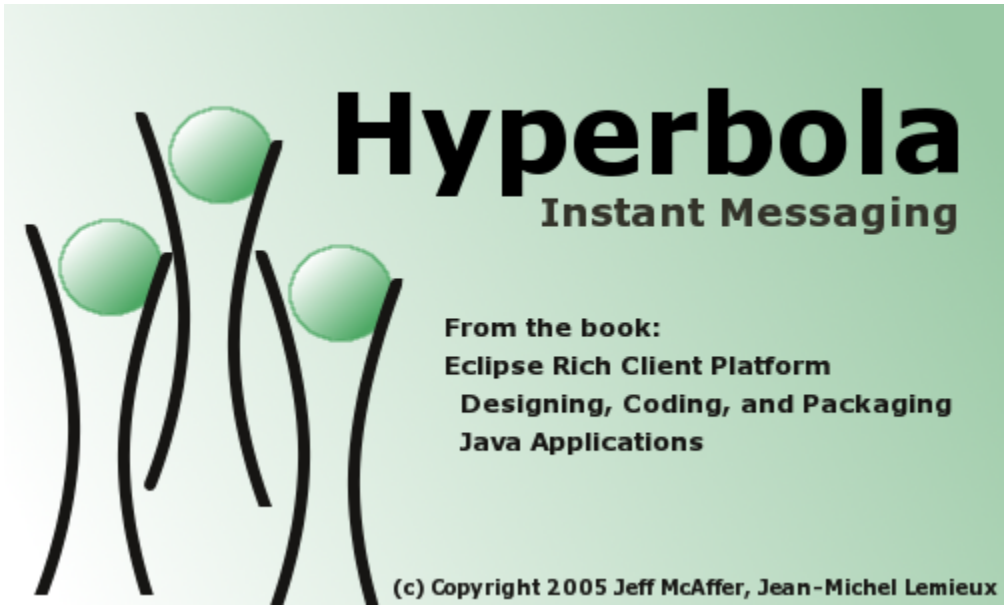
Solution

- If in doubt how to start, use **4.1 Actions**
- If unsure how to proceed, compare to **5.1 Editors**

Pointers

- Creating multiple instance views; see RCP Book Section 16.2.1
- Drag and drop into the editor area; see RCP Book Section 16.4
- Customizing the look and feel of views and editors; refer to RCP Book Chapter 19
- Eclipse User Interface Guidelines
  - http://eclipse.org/articles/Article-UI-Guidelines/Contents.html#Editors
- Building an Eclipse Text Editor with JFace Text
  - http://realsolve.co.uk/site/tech/jface-text.php

Product Branding



Goals

- Learn about *Product Configurations*
- Become familiar with the product editor
- Find out how to customize the launcher, splash, window image and about dialog

Branding

- The Hyperbola application looks nice now
- Still, it's not integrated well
  - It's known as **SWT** to the windowing system
  - The icons in the windowing system are meaningless
  - No splash screen
  - No branded launcher
- *Branding* the application means taking care of these things

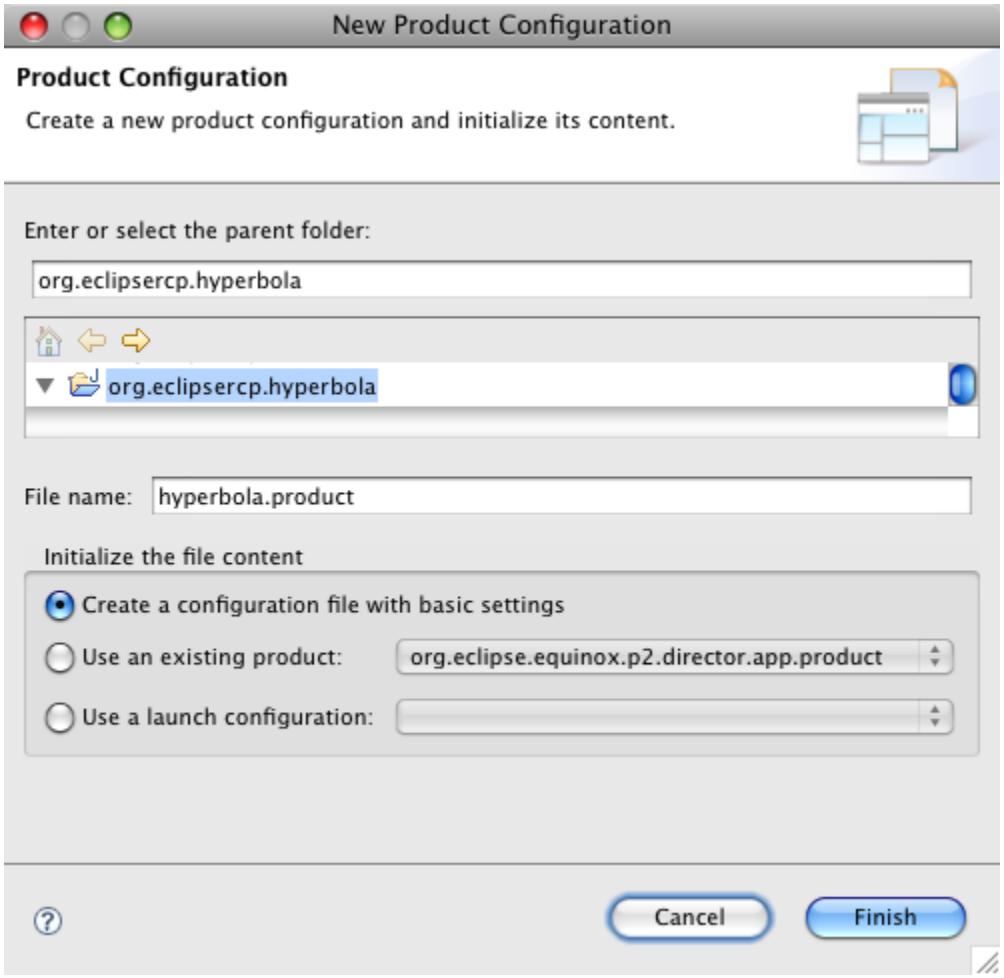
Product Configuration

A *Product Configuration* gathers all the information about

- Splash Screens
- Launcher Icons
- Window images
- About text
- Plug-ins or Features that make up the product
- ...

Creating a Product Configuration

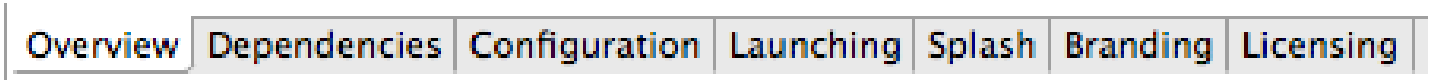
- From the **New Wizards**, select the **Product Configuration** wizard



You may choose a working launch configuration as initial file content.

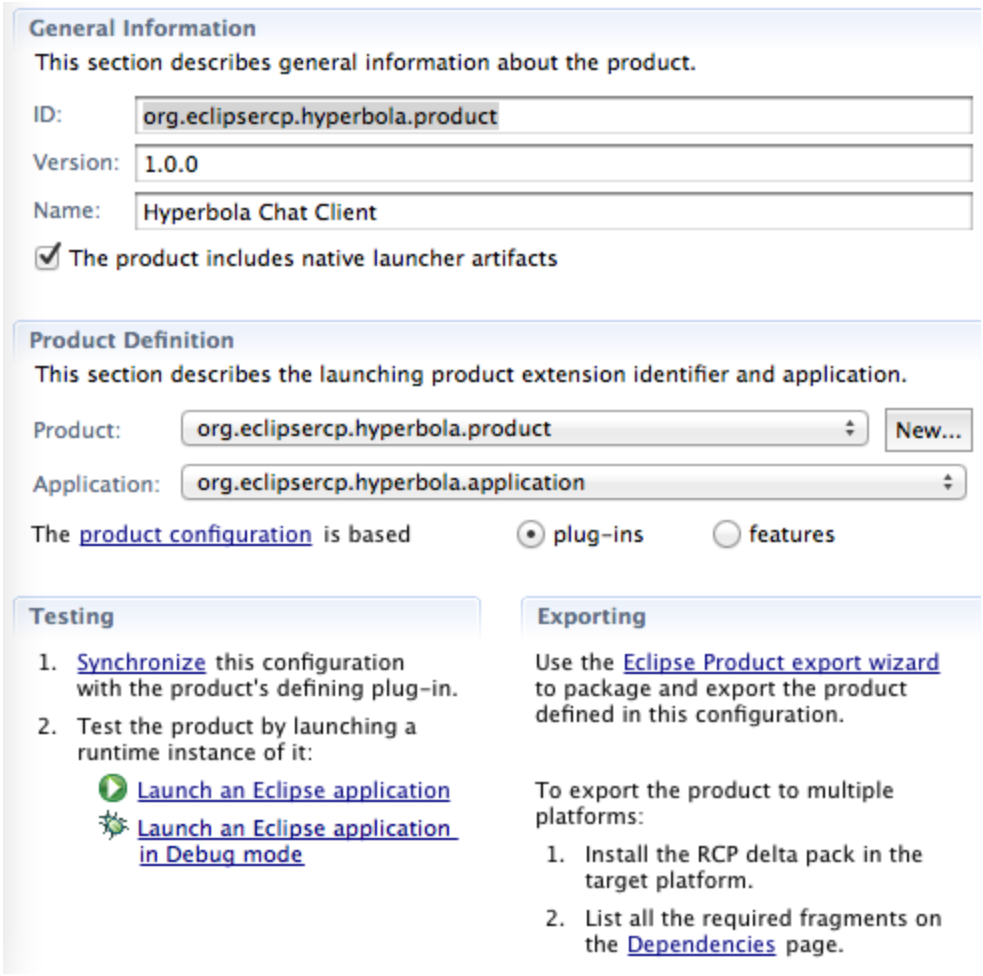
Product Configuration Editor

- The Product Configuration Editor gathers all the branding information together



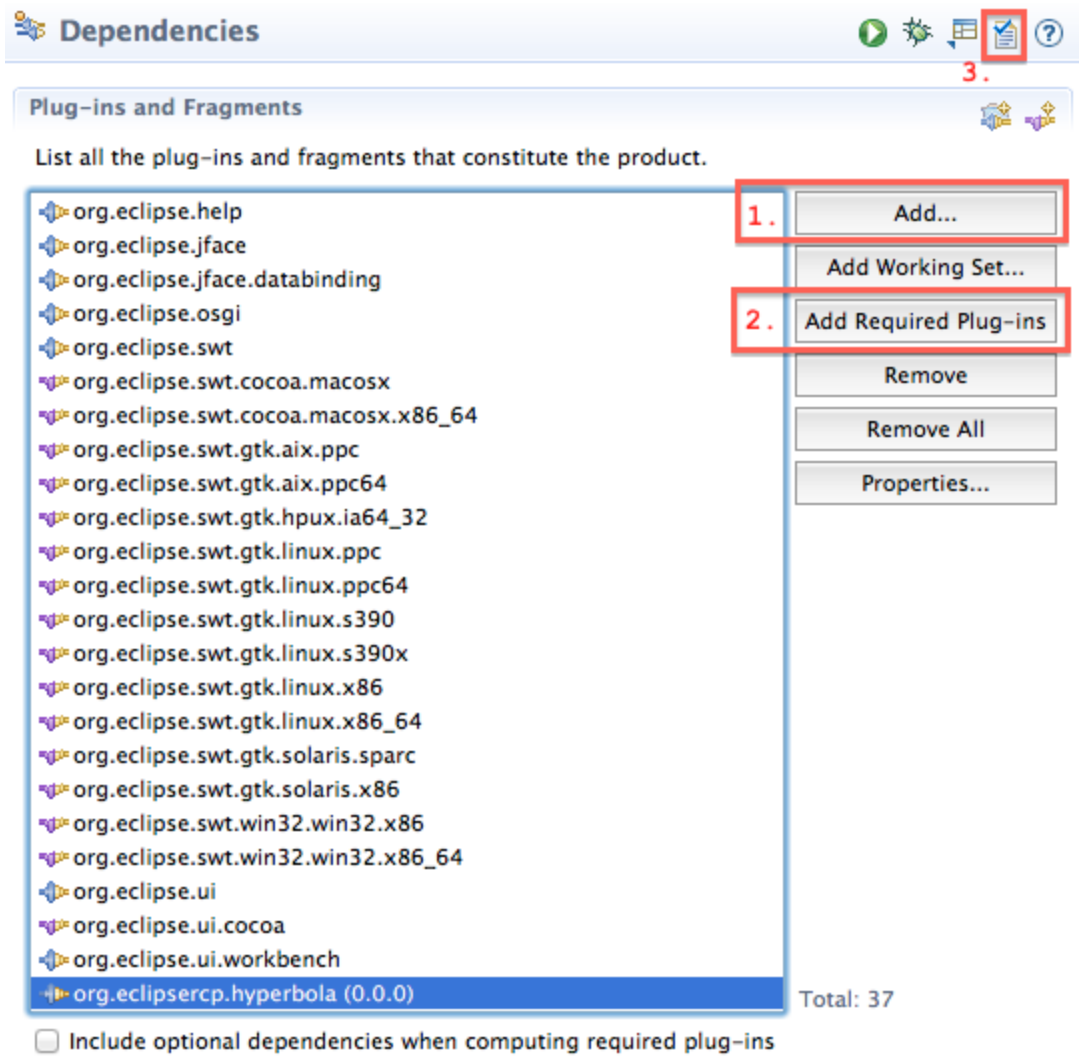
Overview	Title, ID, Application
Dependencies	Which plug-ins / features are part of the product
Configuration	Start levels and configuration properties
Launching	Launcher Name, Launcher Icon, JRE, Arguments
Splash	Splash image, possibly interactive
Branding	Window images, About Dialog, Welcome Page

Overview



- The **Product ID** is prefixed with the Plug-in ID.
- `configurer.setTitle()` in the WorkbenchWindowAdvisor overrides the title from field **Name**
- The product configuration editor gathers information from all kinds of files together.
- To make sure your changes are reflected, hit **Synchronize**!

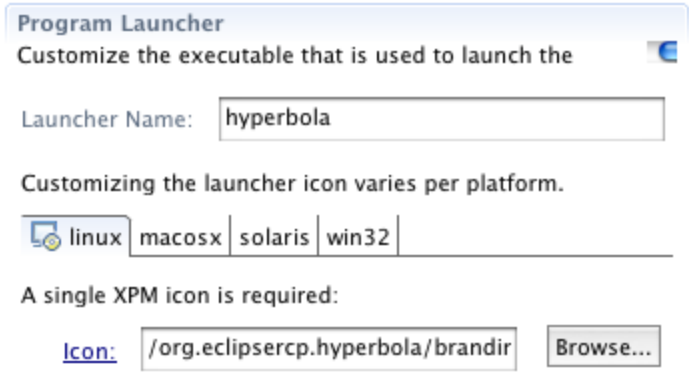
Dependencies



- List all plug-ins / fragments / features that are part of your product
- Use **Add** and pick your hyperbola plug-in
- Use **Add Required Plug-ins** to add dependencies
- Use the **Validate** button (top right) to check your configuration

Custom Launcher

- Telling the users to start `eclipse.exe` for launching *Hyperbola* is not appropriate
- Launcher name is specified in the **Launching** tab

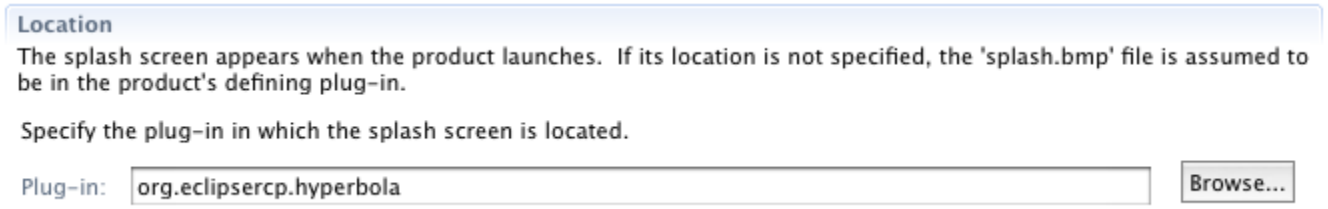


- Different icon sets can be specified according to the target platform

The launcher name is platform independent and should not include a `.exe`. The Product Build will take care of a file extension that fits the platform.

Splash Screen

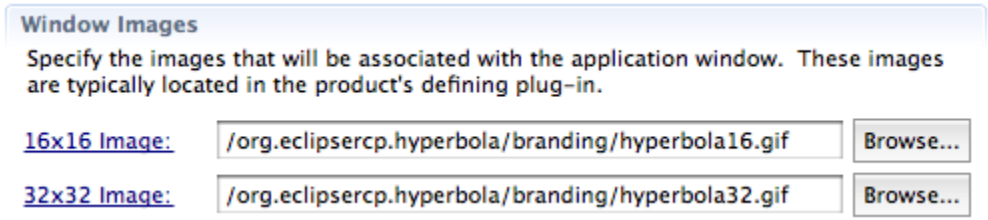
- The splash screen is the first visible part of the application
- The actual splash screen file name must be `splash.bmp`
  - You should specify the plug-in that contains `splash.bmp`
  - The standard choice is the plug-in defining the product



Splash screens are useful for applications that take a few seconds to start up.

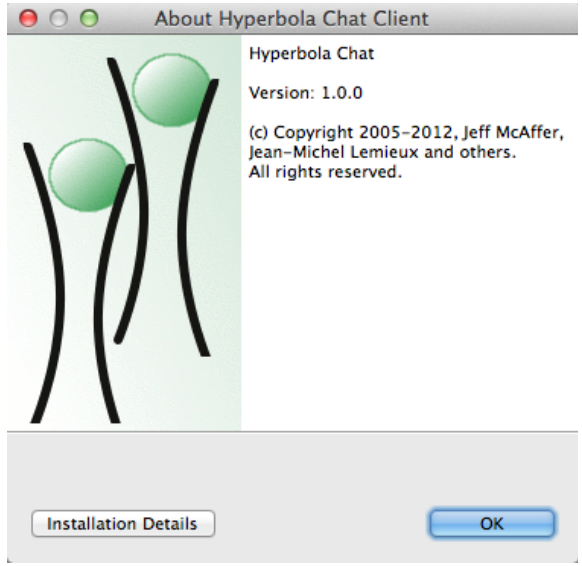
Window Images

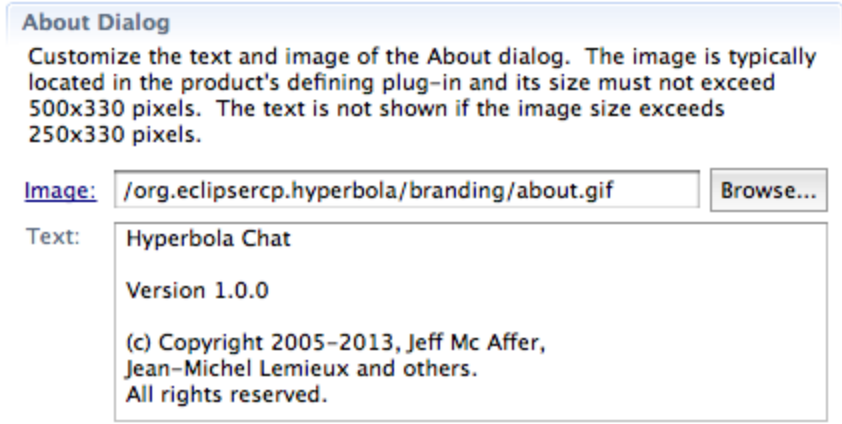
- The Window images are shown in the Task list of the windowing system
- Configured in the **Branding** tab
- Images should be stored in the plug-in that hosts the product configuration



About Dialog

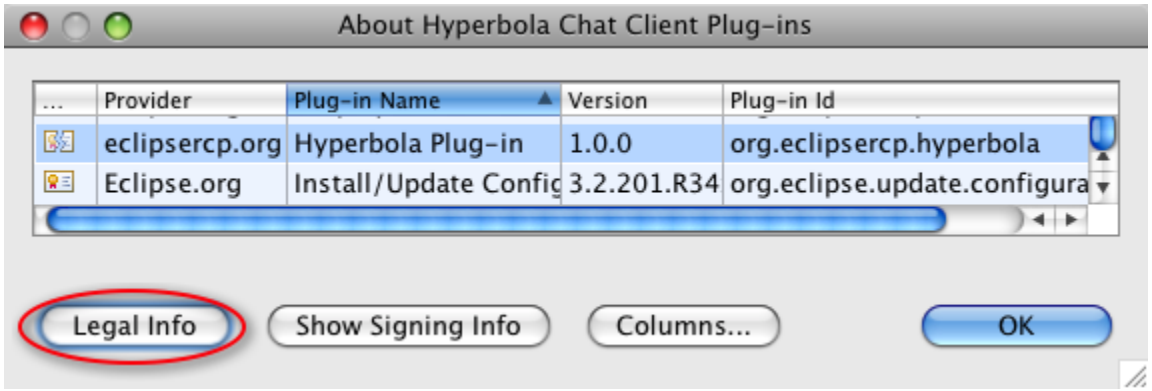
- Standard about dialogs in most applications contain:
  - Product version
  - License
  - Copyright Information
- Eclipse's About Dialog shows configuration details that aid with troubleshooting
- The About Dialog is configured in the **Branding** tab





Legal Info

- The button **Plug-in Details** in the About Dialog opens a list of installed plug-ins
- Selecting the **Legal Info** for a plug-in shows the `about.html` file of that plug-in



The HTML file could contain anything, including references. Additional pages must be in a directory called `about_files`. This is required because Eclipse extracts the files from the jar files. Browsers cannot look into jar files.

Tasks

- Copy the icons from the sample manager into your plug-in
- Add branding to your plug-in:
  - Customize your launcher (will be verified in the next chapter)
  - Add a splash screen
  - Add window images
  - Configure the About Dialog
  - Add an `about.html` and open it via the **Legal Info** button

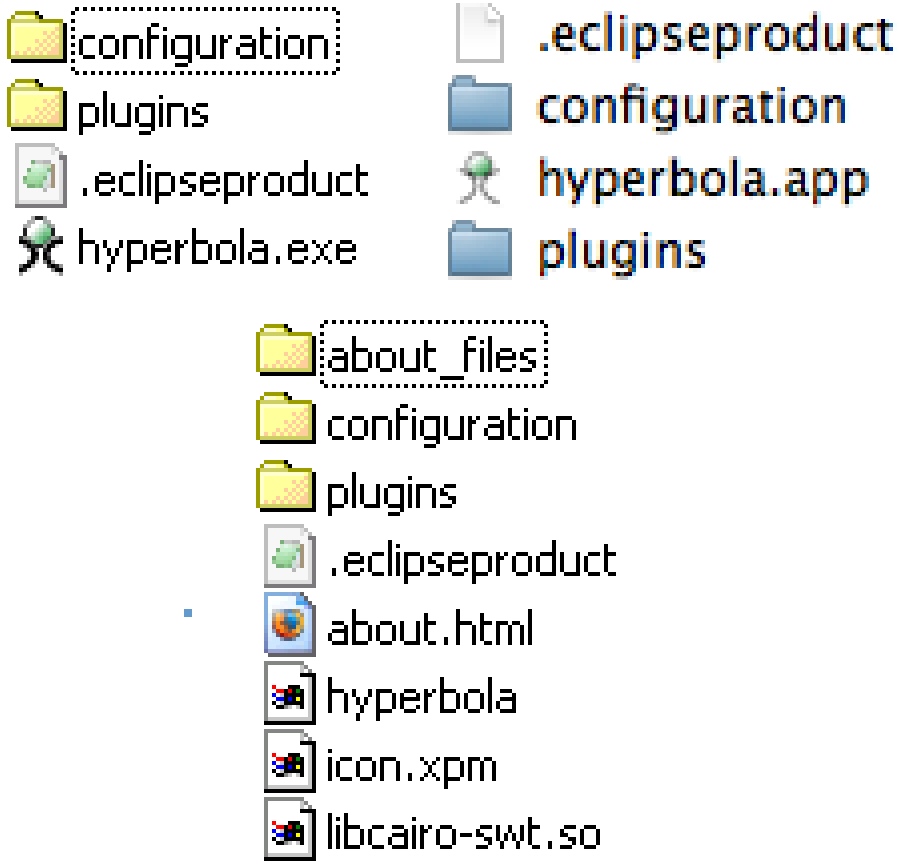
Solution

- If in doubt how to start, use **5.1 Editors**
- If unsure how to proceed, compare to **Solution 6.1 Branding**

Pointers

- Eclipse Corners Article Branding Your Application
- **Help > Platform Plug-in Developer Guide > Programmer's Guide > Packaging and delivering Eclipse based products**

Product Packaging



Goals

- Create a ready-to-run Hyperbola application
- Create applications for different platforms
- Learn about *Features*

Product Export

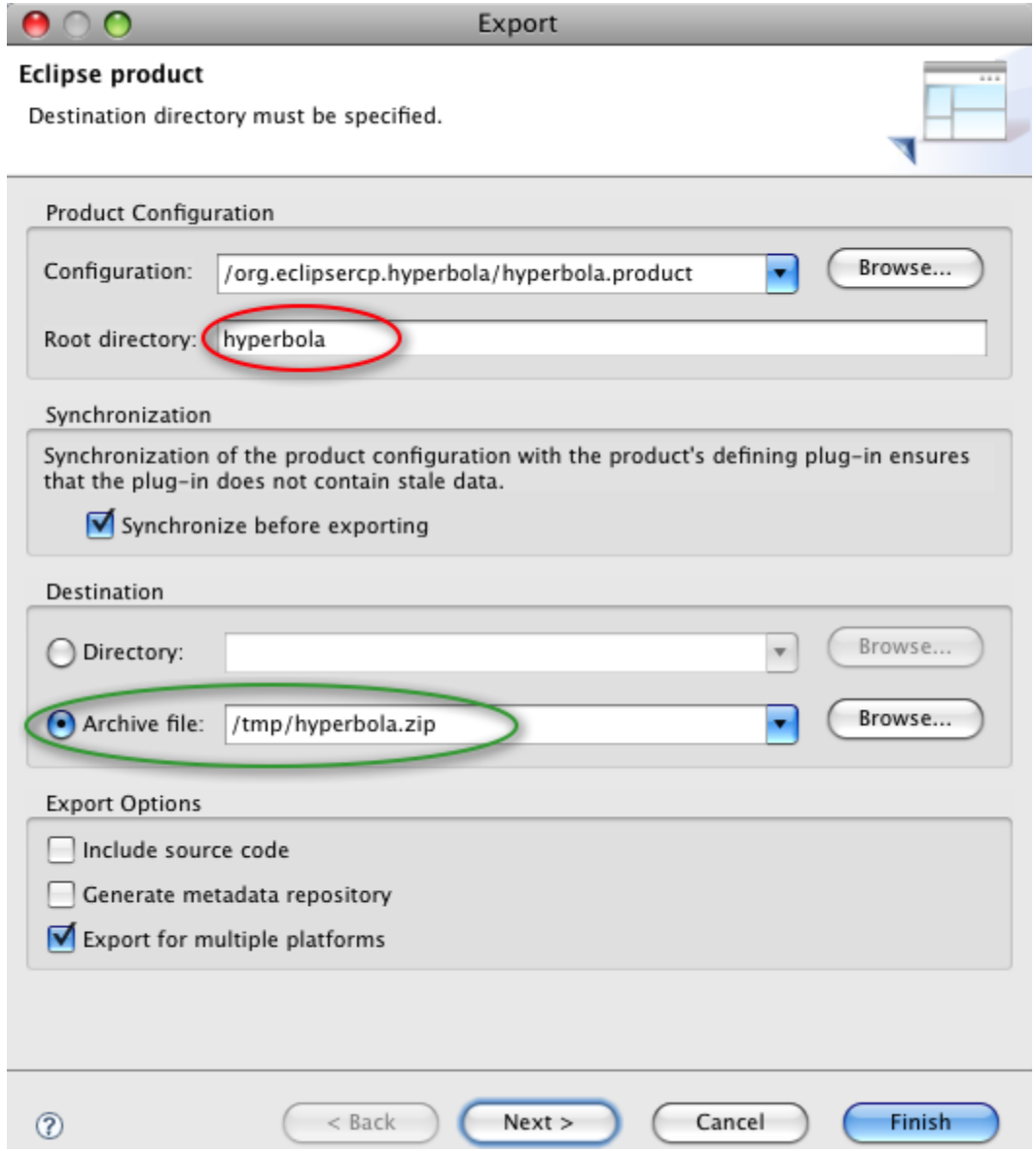
- PDE can export products as ready-to-run applications
- The product file was already used in branding
- During the export
  - Workspace plug-ins are built
  - Target plug-ins are copied

Export

- Open the `hyperbola.product` file
- In the section **Exporting**, click the Link to the Product export wizard

Export Wizard

- The root directory is the name of the application's directory
- Prefer export as archive file
  - PDE build does not purge directories
  - Archive files prevent mixing artifacts from multiple builds

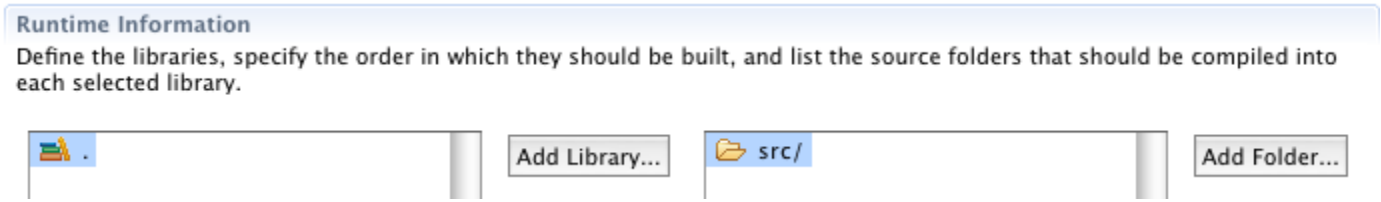


Running the stand-alone application

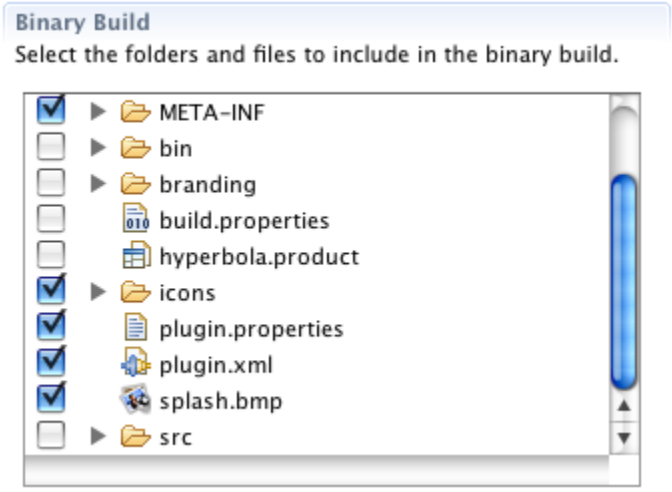
- When the export is done, the archive file `hyperbola.zip` contains a fully branded Hyperbola
- It contains a Eclipse-application like folder structure with a platform-specific executable
- Run the executable and enjoy the completed RCP product

Defining Plug-in contents

- Each workspace plug-in project has a `build.properties` file
- Section **Runtime Information** defines which source folders are built

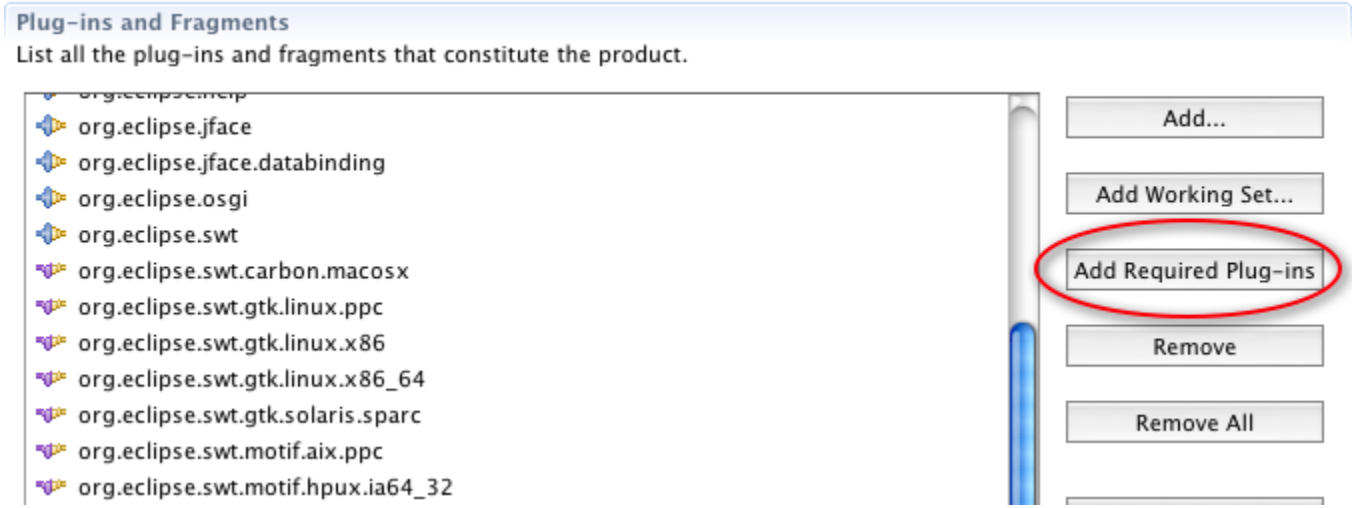


- Section **Binary Build** defines which files go into the deployable



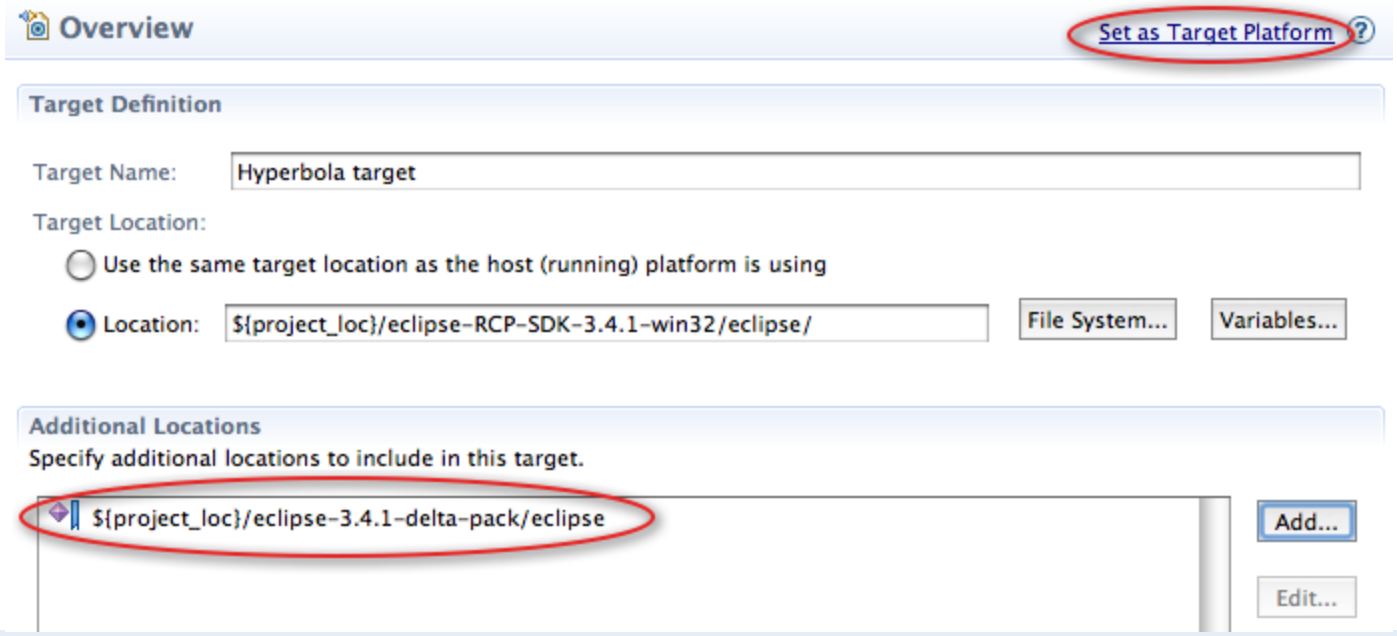
Delta Pack

- Eclipse supports a number of different platforms
- The Delta Pack `eclipse-RCP-3.X-delta-pack.zip` from the Eclipse Project's Download Site includes the fragments for other platforms
- The **Configuration** page of the product must be adjusted to include the SWT fragments for the other platforms

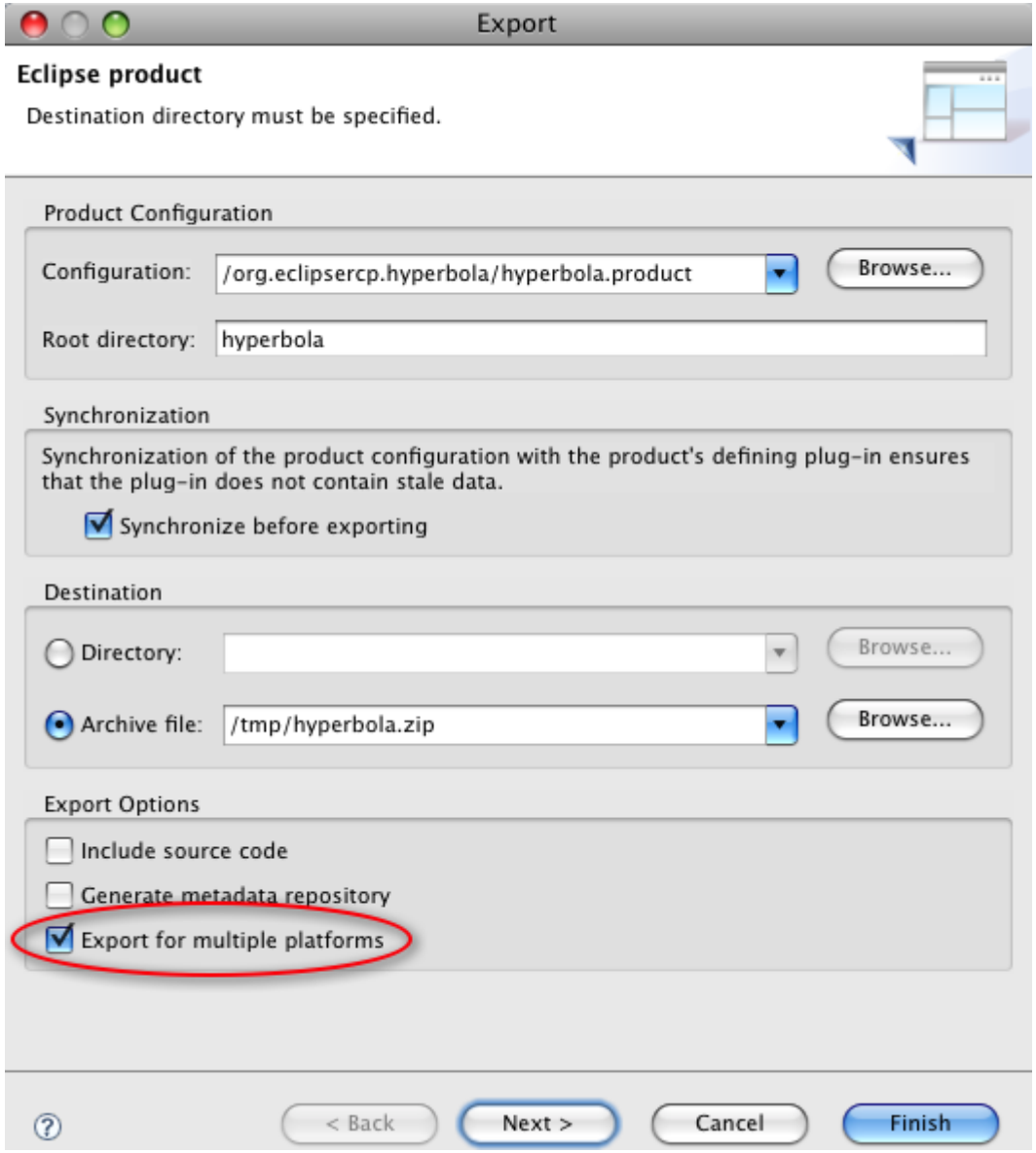


Delta Pack in the Target Platform

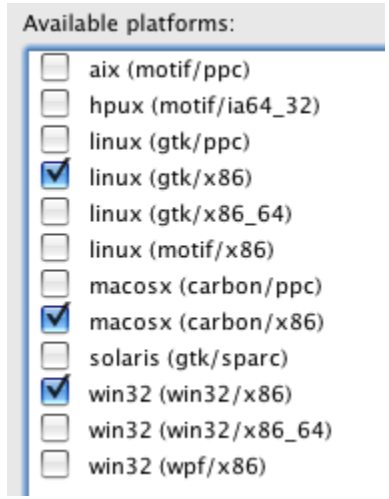
- The initial target platform contains only the RCP SDK plug-ins for windows
- The delta pack is provided as folder in the project location
- To include it in the target platform, add it as **Additional Location**
- Make sure to save the editor content before hitting **Set as Target Platform** again



Exporting for Other Platforms

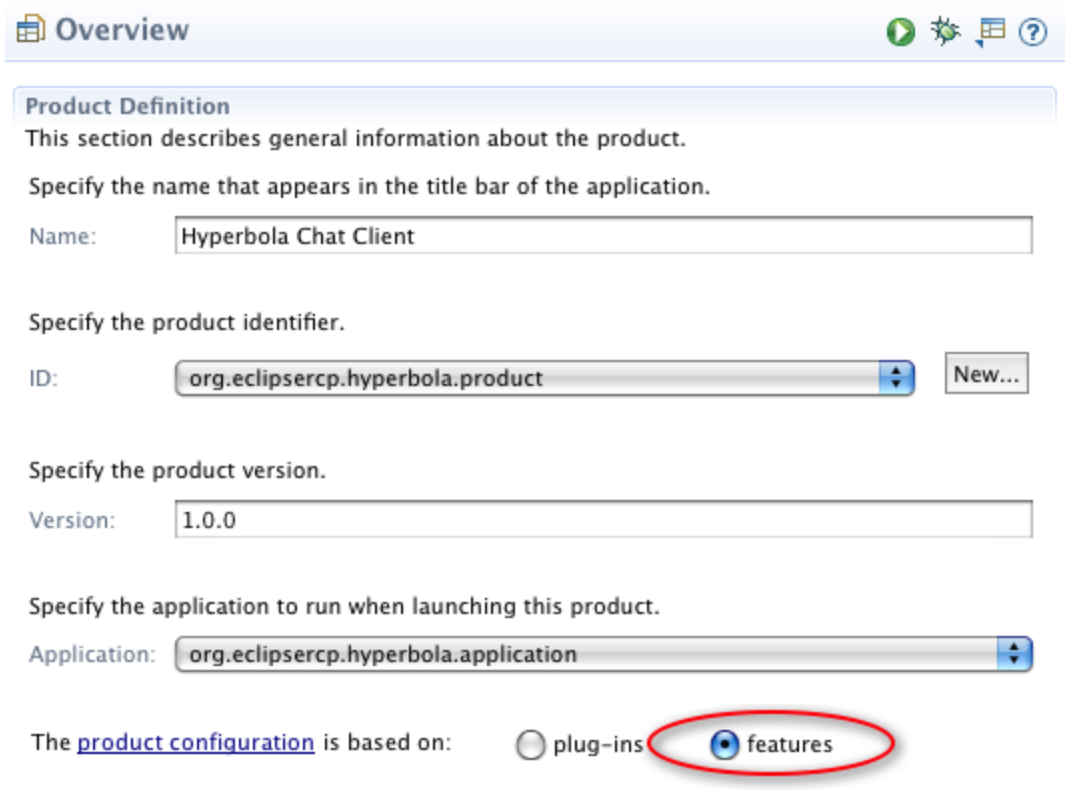


- Due to the Delta Pack the check box **Export for multiple platforms** appears
- This enables selection of other platforms on the next page



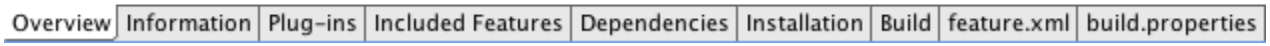
Packaging with Features

- As alternative to being plug-in based, Products can be feature based
- Features are metadata that list plug-ins and deployment requirements



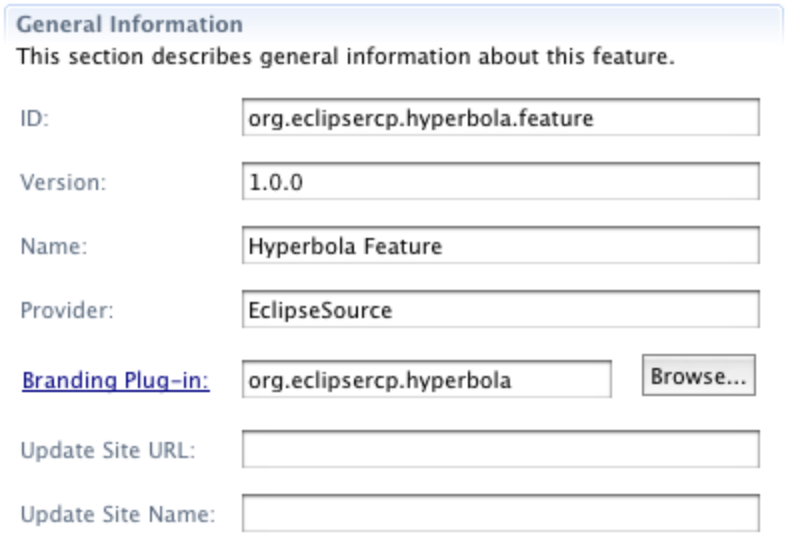
Features

- A feature is a grouping of plug-ins described in the feature manifest file (`feature.xml`)
- A feature is metadata that contains information for building and updating a group of plug-ins
- Headless product builds can only be feature based
  - When exporting from the Eclipse IDE, they can be plug-in based
- Create features using the **New Feature Project** wizard
- Features are edited in a feature manifest editor



Feature Manifest Editor Overview

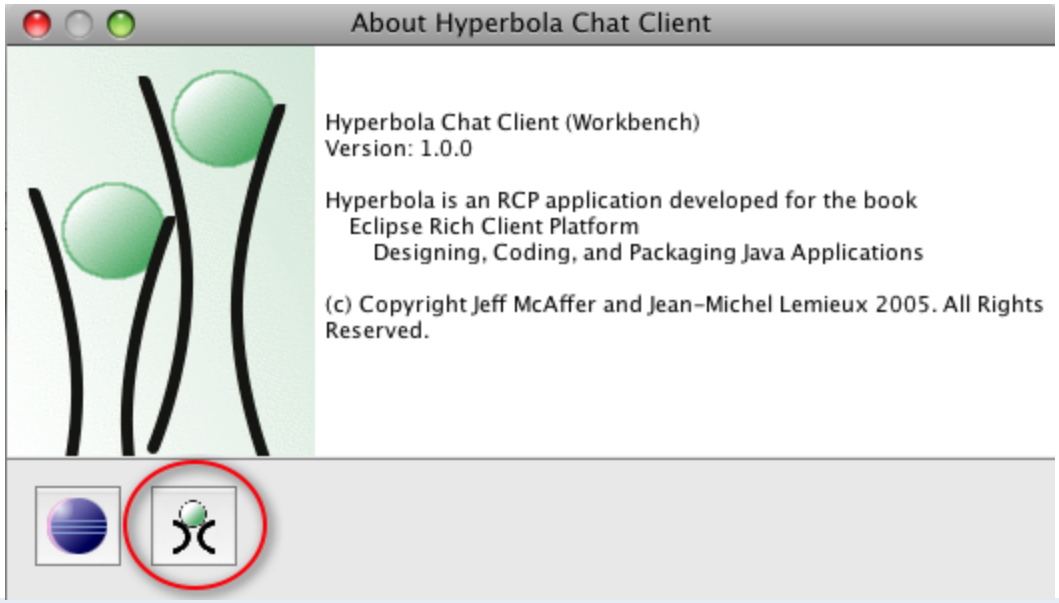
- In the overview page, the identity and version of the feature are configured
- The branding plug-in can contribute items to the about dialog (see next page)
- If an update site is given, the Eclipse update mechanism knows where to check for updates



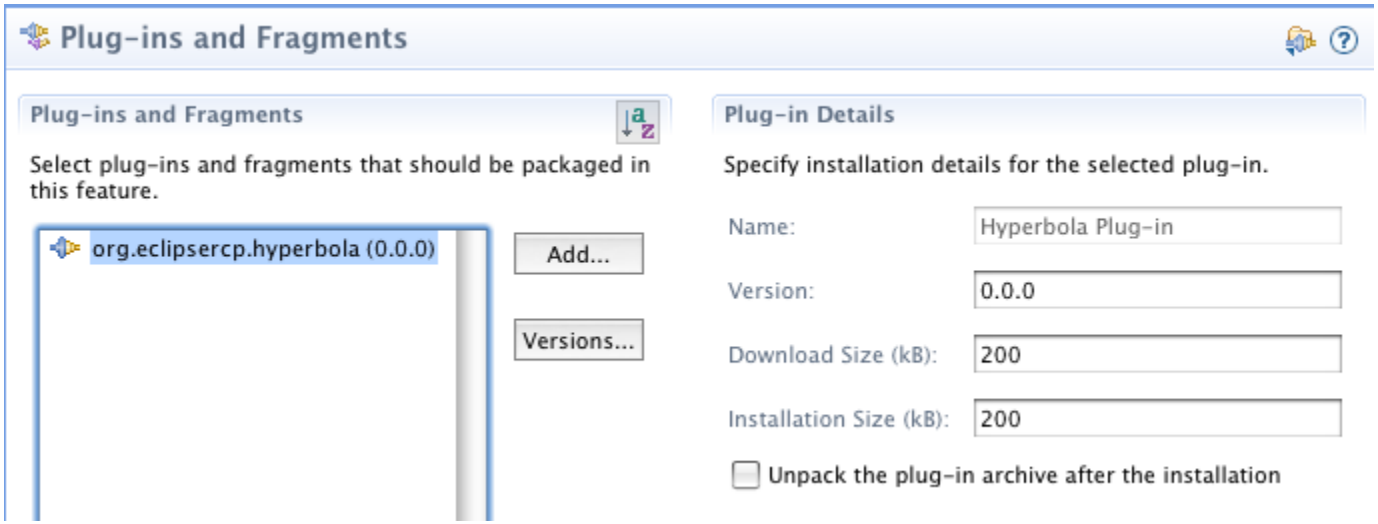
Feature Branding

- A Feature's **Branding Plug-in** is configured on the Overview page in the Feature Manifest Editor
- A branding plug-in contains a `about.ini` that contributes to the about dialog

```
aboutText=Blurb for the "About Dialog"

featureImage=icons/alt32.gif

appName=Hyperbola Chat Client
```



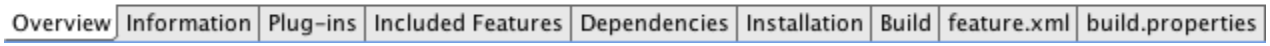
Plug-ins in a feature



- The tab **Plug-ins** in the Feature Manifest Editor lists all plug-ins and fragments that this feature contains
- Plug-in Details specify details about the selected plug-in
  - Version *0.0.0* is a placeholder for the version available at build time
  - The checkbox **Unpack the plug-in ...** decides whether the plug-in is installed as JAR file or as directory

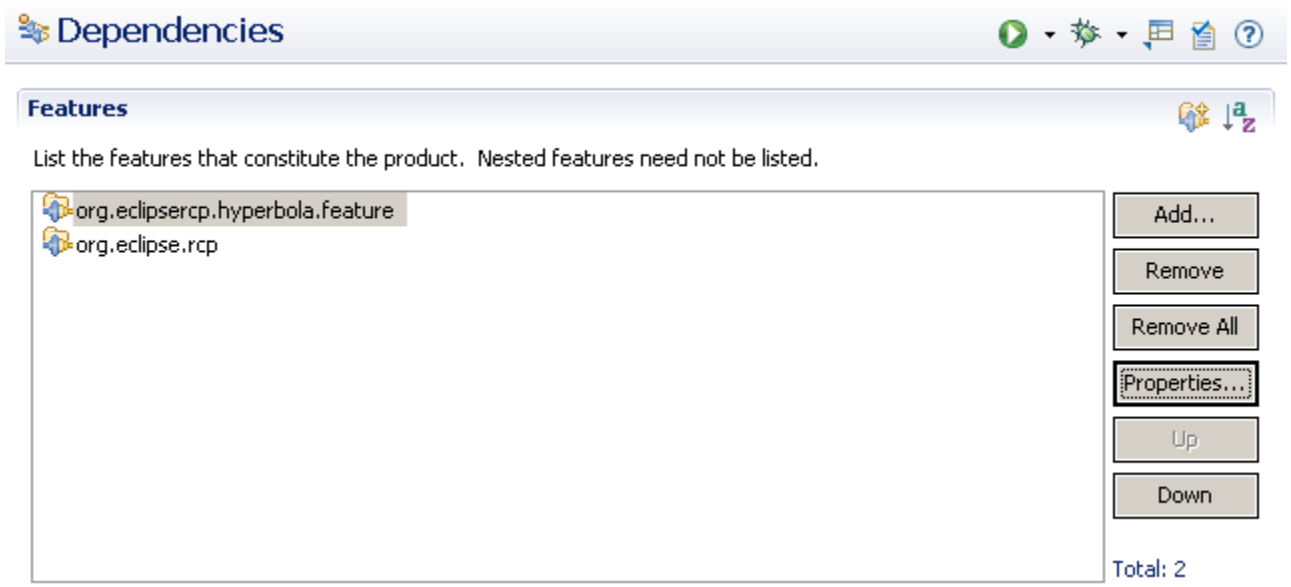
Other Feature Sections

- **Included Features** specifies feature dependencies
  - The included features always come with a feature when building or installing without being listed separately
- **Installation** defines requirements of the target host machine, i.e. platform
- **Build** defines which files to include in the build
  - This is the same mechanism as for plug-ins



Feature Based Product

- Feature requirements cannot be computed automatically
- Obviously the Hyperbola feature must be included
- Additionally, the RCP feature is included, it contains the dependencies of Hyperbola



Note: Specifying feature versions obliges you to update your product after each feature update.

Tasks

- Add the delta pack to your target platform
- Export your product and start it
  - Decide yourself if you want a plug-in or feature based build
- Make sure all the images are included in the exported product
- Try to find the delta pack download on eclipse.org

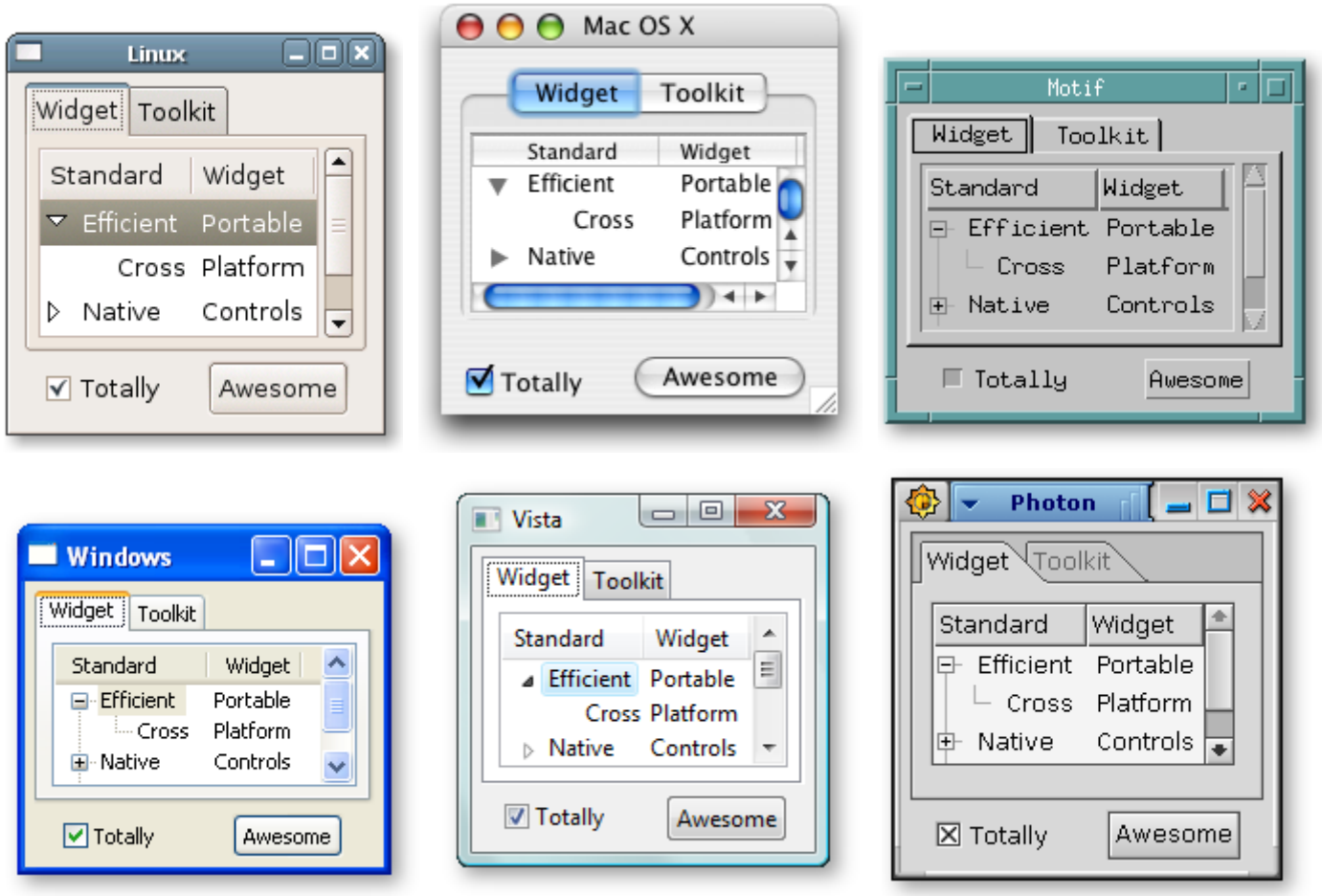
Solution

- If in doubt how to start, use **6.1 Branding**
- If unsure how to proceed, compare to
  - **Solution 7.1 Packaging** for a plug-in based product
  - **Solution 7.2 Packaging with features** for a feature based product

Pointers

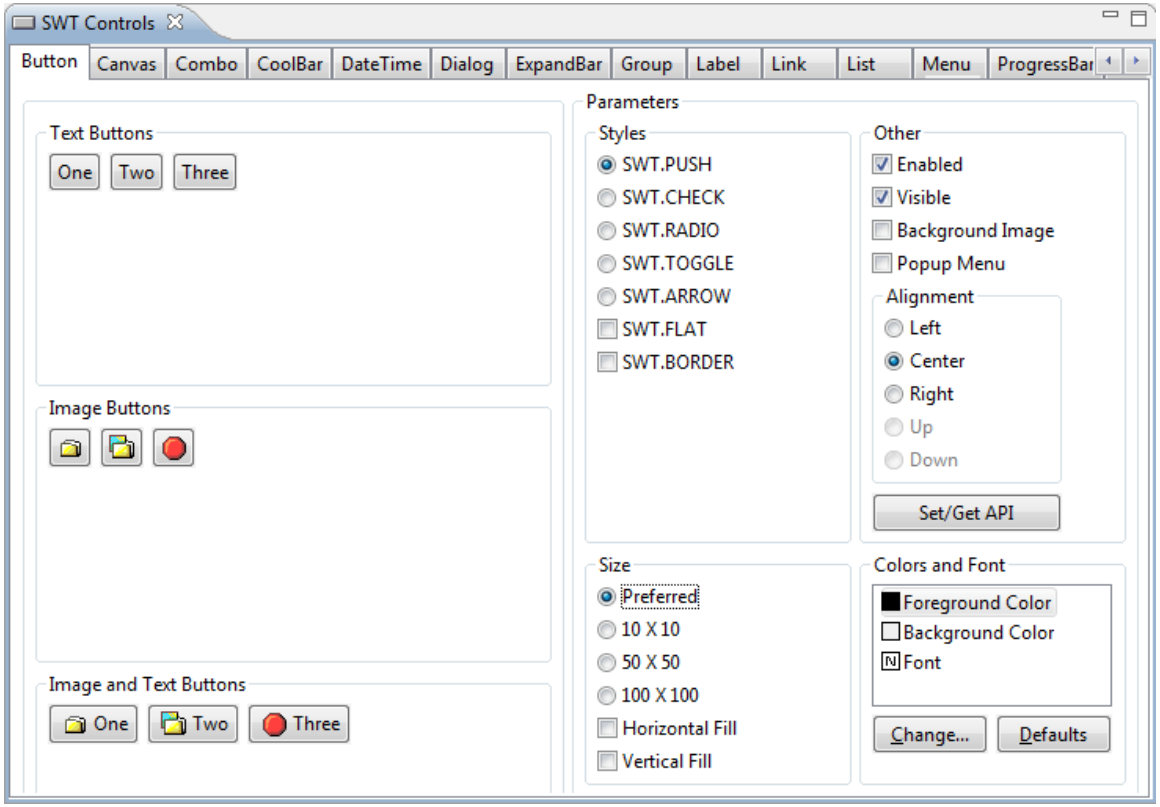
- Eclipse Foundation Downloads
- Latest Eclipse Project Builds
- **Help > Plug-in Development Environment Guide > Tools > Export Wizards**
- **Help > Platform Plug-in Developer Guide > Programmer's Guide > Packaging and delivering Eclipse based products**

SWT Widgets



Optional Task

- Extract swt_examples.zip to eclipse\drop-ins
- Restart Eclipse
- Open the SWT Controls View
  - via Window > Show View > Other... > SWT Examples > SWT Controls



Standard Widget Toolkit

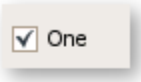
- SWT provides access to native operating system widgets using a consistent 100% Java API
  - SWT uses MFC under Windows, GTK under Linux and Cocoa under Mac OS X using an 1:1 mapping of JNI Methods to native code
- API provides the minimal common denominator for all supported platforms, some missing OS functionality is emulated
- SWT provides only a very raw, low level abstraction
  - model-based components can be found in JFace which provides a higher level of abstraction

Widgets Examples 1

- SWT supports many native widgets:

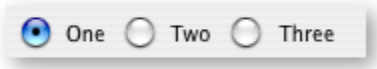
Browser
javadoc - snippets

Button (SWT.ARROW)
javadoc - snippets



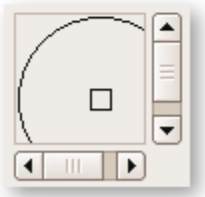
Button (SWT.CHECK)
javadoc - snippets

Button (SWT.PUSH)
javadoc - snippets

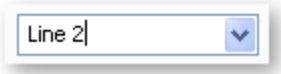


Button (SWT.RADIO)
javadoc - snippets

Button (SWT.TOGGLE)
javadoc - snippets



Canvas
javadoc - snippets

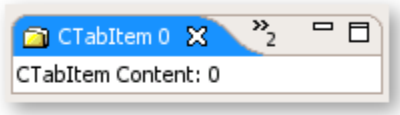


Combo
javadoc - snippets

Composite
javadoc - snippets

Widgets Examples 2

CoolBar
javadoc - snippets



CTabFolder
javadoc - snippets

DateTime
javadoc - snippets

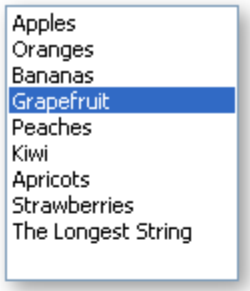
ExpandBar
javadoc - snippets

Group
javadoc

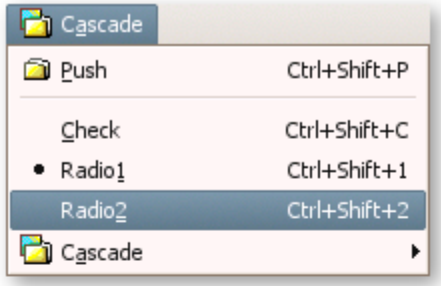
Label
javadoc - snippets

Widgets Examples 3

Link
javadoc - snippets

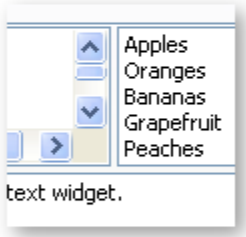


List
javadoc - snippets



Menu
javadoc - snippets

ProgressBar
javadoc - snippets



Sash
javadoc - snippets

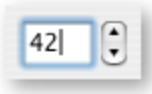
ScrolledComposite
javadoc - snippets

Shell
javadoc - snippets

Slider
javadoc - snippets

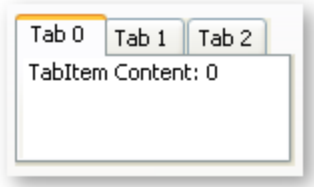
Scale
javadoc - snippets

Widgets Examples 4



Spinner
javadoc - snippets

StyledText
javadoc - snippets



TabFolder
javadoc - snippets

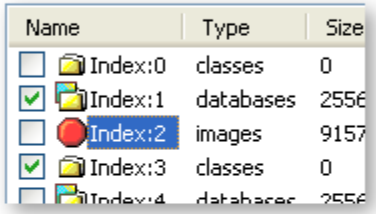
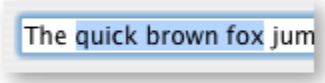
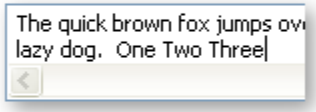


Table
javadoc - snippets

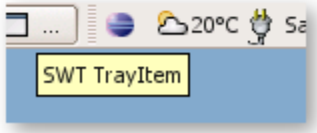


Text (`SWT.SINGLE`)
javadoc - snippets

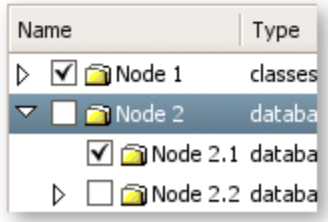


Text (`SWT.MULTI`)
javadoc - snippets

ToolBar
javadoc - snippets



Tray
javadoc - snippets



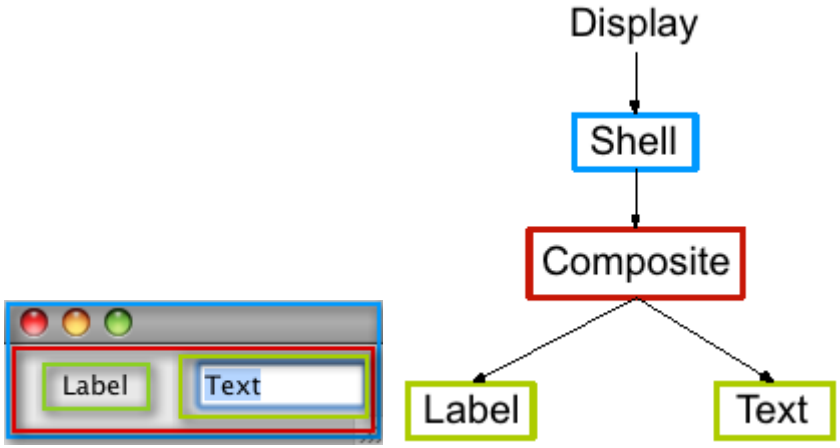
Tree
javadoc - snippets

Pros and cons

- Advantages of SWT
  - Consistent native look & feel on each platform
  - Really good performance
  - Easy to program
- Criticism of SWT
  - Requires native library
  - Platform dependent odds-and-ends - it is highly recommended to test on each platform you want to support with your application
  - Low level of abstraction
- There are endless discussions which UI framework is better - SWT or Swing. Long story short: Both are great UI frameworks. For developing Eclipse RCP applications, you have to use SWT.
  - SWT can host Swing components (SWT/AWT bridge)

Widget Hierarchy

- All widgets are arranged in a tree-like structure
- All Windows, dialogs, tooltips etc are `Shell`s
- `Composite` is a container in which children can be placed
- Every widget has exactly one widget as parent
  - Except the top-level shell with the display as parent



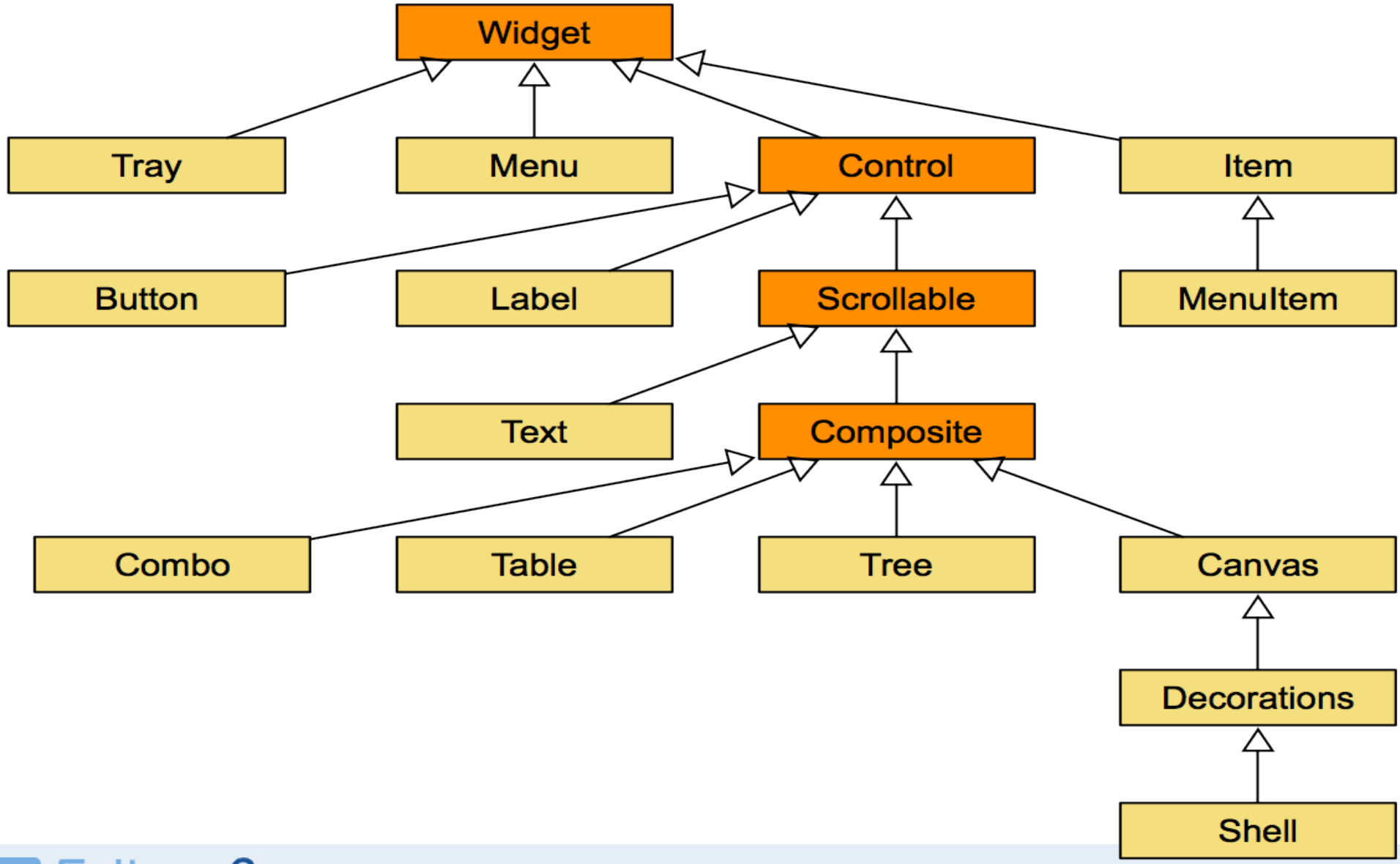
Instantiating SWT Widgets

- SWT Controls generally have a two-argument constructor
- First argument is the parent composite (or subclass)
- Second argument is a style constant from `org.eclipse.swt.SWT`
  - See API of the widget which style constants apply to it
  - Multiple constants are concatenated with a logical OR

```
Label label = new Label( parent, SWT.WRAP | SWT.BORDER );
```

Because the parent is always passed in the constructor, there is no `add...()` method for adding child widgets on containers as in Swing.

SWT Classes



Widget

- `Widget` is the superclass of all SWT widgets
- A widget...
  - Can be created
  - receives notifications when events occur
  - must be disposed
- Most of SWT GUI element classes **cannot be extended** outside the SWT implementation
  - They are not final because they are subclassed within `org.eclipse.swt`

Control

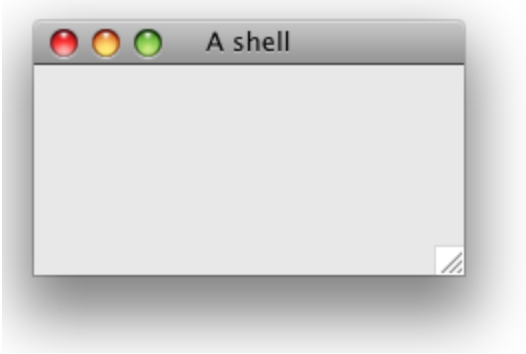
- The `Control` is a subclass of `Widget` and acts as a base class for all "windowed" UI classes
- A `Control`...
  - knows its size and position
  - can be enabled or disabled
  - can be shown or hidden (marked as visible or invisible)
  - can receive keyboard focus
  - can handle focus, mouse, keyboard, size and paint events

Composite

- `Composite` is a special `Control` that contains other `Control`s
- `Composite` manages it's child elements
  - When a composites `dispose()` method is called, all it's children will be disposed
  - Supports layouts
  - `Composite` subclasses: `Group`, `Canvas`, `Shell`

Shell

- `Shell` is a special subclass of `Composite` and represents the "window" concept of the underlying graphical system
- There can be primary (top level) and secondary windows (main window vs. dialog window)



- For a GUI at least one top level `Shell` (window) is required
- In Eclipse RCP the top level `Shell` is created by the framework
  - See `PlatformUI.createAndRunWorkbench(...)` in the application class

Display

- Each SWT program has to create a `Display`
- In SWT programs, a `Display` is created in the first line of code
- In RCP applications, the `Display` is created by the application class
- A `Display` connects your program to the windowing system of the underlying platform
- In most cases only one instance of SWT's `Display` is needed per application
  - Some platforms allow only the creation of one instance

Event loop

- All SWT events are processed synchronously by the event loop

```
while ( !shell.isDisposed() ) {
  if( !display.readAndDispatch() ) {
    display.sleep();
  }
}
```

- This code must be run within the same thread that created the display
- All events are dispatched synchronously by the event loop
  - No event is processed before the current event handler has processed its code
  - Do not process long running operations in the event handler!

Event loop responsibilities

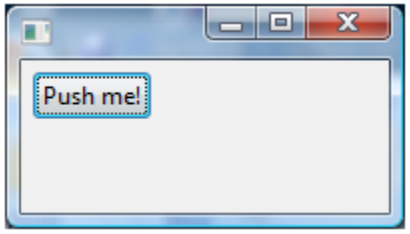
- The event loop receives different events of the underlying windowing system
- It converts the events
- It passes the events to appropriate listeners within the application
- In RCP the event loop is created and run automatically by the framework
  - See `PlatformUI.createAndRunWorkbench(...)` in the application class

Listeners

- Typical SWT listeners which can be attached to a control are:
  - `SelectionListener`
  - `KeyListener`
  - `MouseListener`
  - `ModifyListener`
  - ... `see org.eclipse.swt.events` package ...
- For most listeners appropriate adapter-classes with empty implementations of required methods exist, e.g.:
  - `MouseListener` → `MouseAdapter`
  - `SelectionListener` → `SelectionAdapter`

Listener Example - SelectionAdapter

```
Button button = new Button( shell, SWT.PUSH );
button.setText( "Push me!" );
button.addSelectionListener( new SelectionAdapter() {
  public void widgetSelected( SelectionEvent e ) {
    System.out.println( "Thanks!" );
  }
});
```



Hello world SWT

```
public static void main( String[] args ) {
  Display display = new Display();
  Shell shell = new Shell( display );
  shell.setSize( 200, 200 );
  shell.open();
  shell.setText( "Hello World!" );
  while( !shell.isDisposed() ) {
    if( !display.readAndDispatch() ) {
      display.sleep();
    }
  }
  display.dispose();
}
```

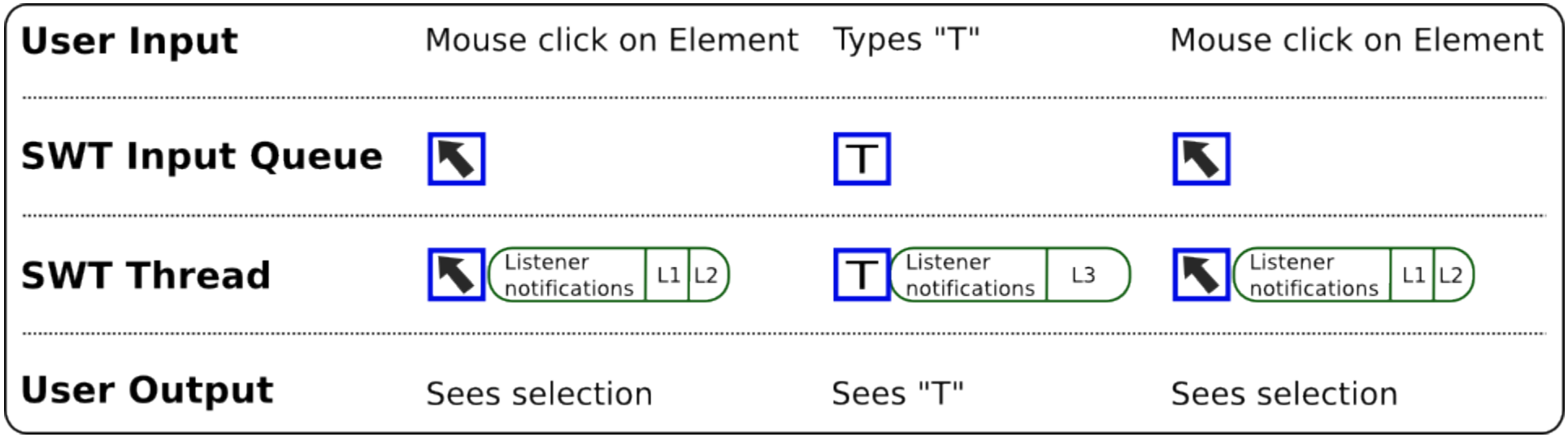
UI thread

- The thread that runs the event loop is the *user-interface thread* for it's `Display`
- SWT methods can only be invoked from this thread. Otherwise `SWTException` "Invalid Thread Access" will be thrown
- To synchronize with the UI Thread from other threads use:

```
Display.syncExec( Runnable ); // blocks until parameter was run
Display.asyncExec( Runnable ); // returns immediately
```

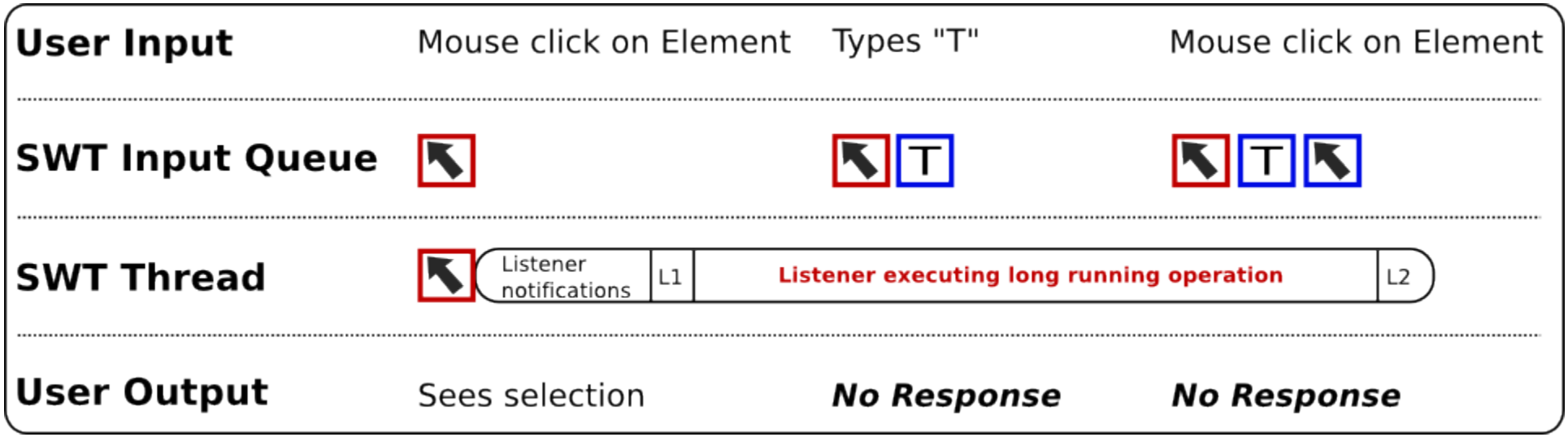
UI Queue

- The UI Input queue contains all user input and runnables from `(a)syncExec(...)` calls
- The SWT Thread works on one item at a time
  - Listener notifications are done synchronously



Long Running operations

- Execution of long running operations in the UI thread blocks the User Interface



- Users will complain:
  - "Application hangs"
  - "Java is slow"

Handling long running operations

For long running operations:

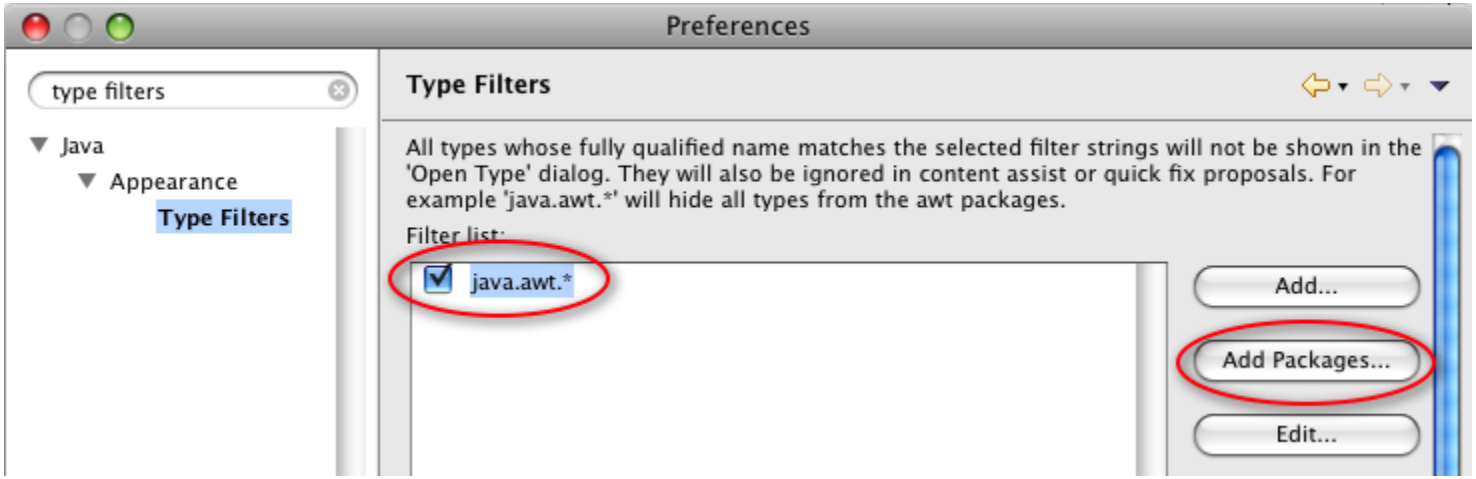
1. Immediately get out of the UI thread with
   - a Job
   - a ProgressMonitorDialog
   - a new Thread
2. When the operation is done, give user feedback inside `Display.(a)syncExec( Runnable );`

SWT/AWT bridge

- Swing (AWT) and SWT elements can be placed within the same widget hierarchy
- Class `org.eclipse.swt.awt.SWT_AWT` provides factory methods for interaction
- Typical integration problems:
  - Flicker during scrolling
  - Proper focus handling
  - Tab traversal
  - Modal dialogs
  - ... see article in Pointers ...

Tips and Tricks: Ignoring packages

- When writing SWT Code, you want to import your SWT widgets from `org.eclipse.swt`
- Sometimes the code completion gets in the way with import suggestions from `java.awt` or `javax.swing`
- In the preferences, the type filters can be configured to suppress suggestions from specific packages



Tasks

- Use SWT standalone
  - Create a new Java project
  - Copy a SWT snippet from http://eclipse.org/swt/snippets into the source folder
  - Add the jar `org.eclipse.swt.win32.win32.x86.<version>.jar` from the target platform as library to the java build path
  - Run the snippet as Java application

(More specific tasks with SWT will come after SWT Layouts)

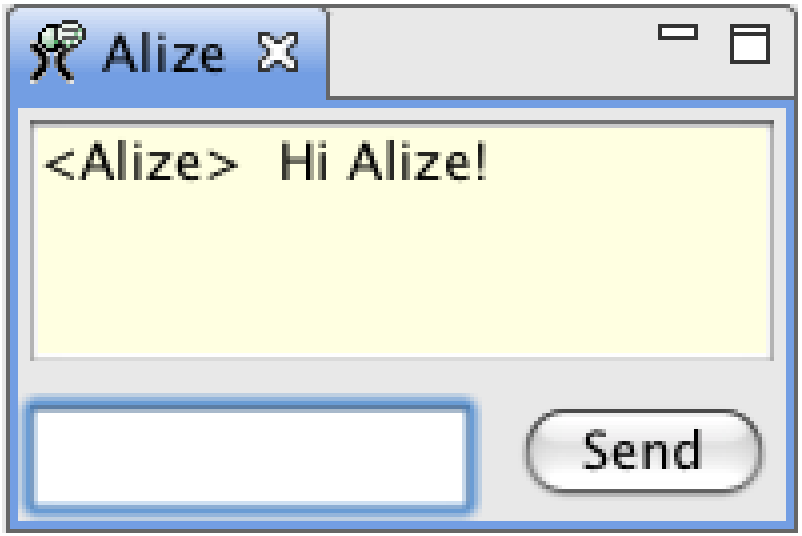
Solution

- Start with an empty project
- No solutions are provided for this non-RCP task

Pointers

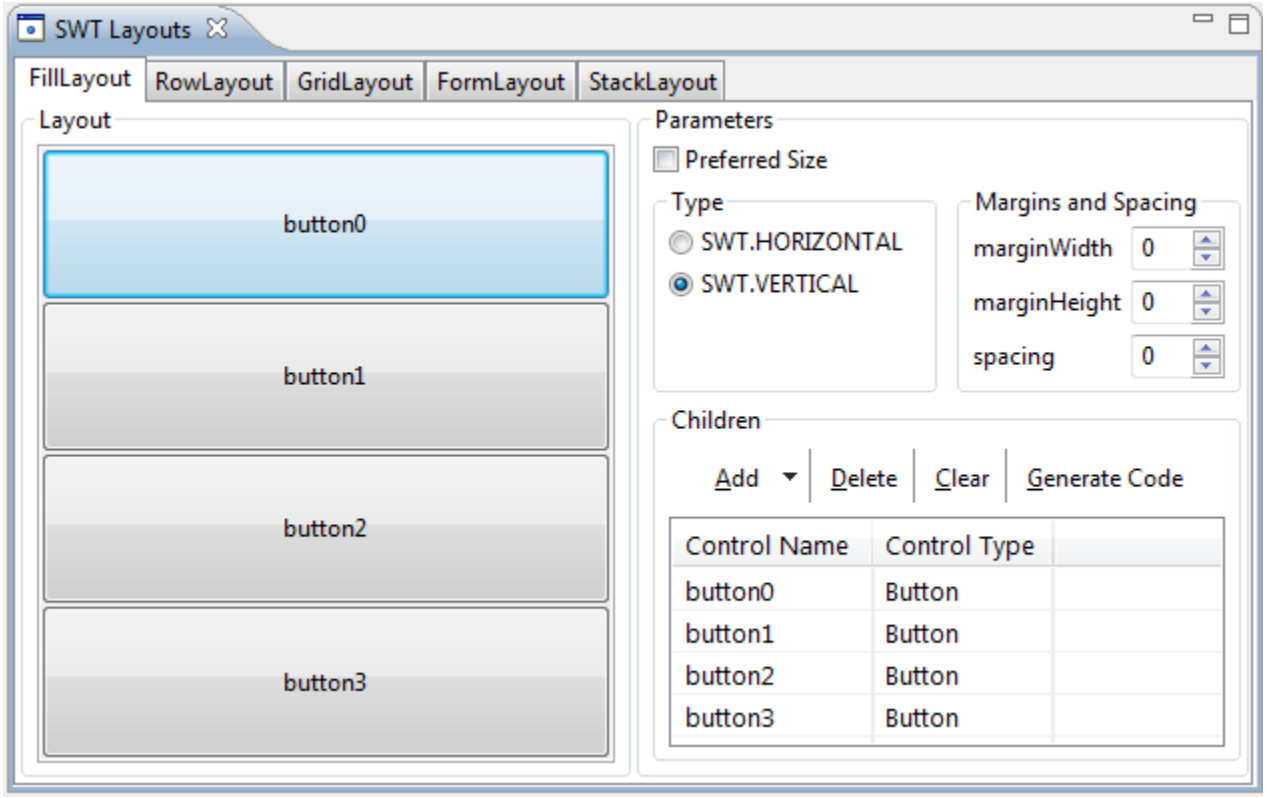
- SWT Homepage http://eclipse.org/swt
- SWT Snippets http://eclipse.org/swt/snippets
- Eclipse Nebula - more SWT widgets: http://eclipse.org/nebula
- Eclipse WindowBuilder: http://eclipse.org/windowbuilder
- Article on Swing/SWT Integration http://eclipse.org/articles/Article-Swing-SWT-Integration/index.html

SWT Layouts



Optional Task

- Install org.eclipse.sdk.examples-3.6.1.zip via p2
- Open the SWT Layouts View



SWT Layout Basics

- By default SWT does not set size or position of it's components
- Every new control has the size (0,0), so it is invisible
- Applications can define positions and sizes of controls when they are created or later (inside resize listener):
  - `Control.setSize(Point point)`
  - `Control.setSize(int x, int y)`
  - `Control.setBounds(Rectangle rect)`
  - `setBounds(int x, int y, int width, int height)`
- Alternatively, a Layout may be specified. An instance of the `Layout` class will be responsible for sizing and positioning controls.

SWT Layout Basics

- A layout controls the position and size of child elements in a `Composite`
  - `Composite.setLayout(Layout layout)`
- The size and positioning of a control can be defined by setting an object with layout data
  - `Control.setLayoutData(Object layoutData)`
- SWT layouts are similar to layouts in AWT / Swing
- It is possible to define custom layouts: Layout classes are subclasses of the abstract class `Layout`.

Layout instances may be shared - LayoutData instances must NOT be shared.

SWT Layouts

- SWT comes with five basic layouts:
  - `FillLayout` - distribute space equally
  - `RowLayout` - fill line-by-line or row-by-row
  - `GridLayout` - place in grid l-to-r, t-to-b
  - `FormLayout` - place relative to others
  - `StackLayout` - stack of "cards", one on top

FillLayout

- `FillLayout` distributes the available space equally
- `l.margin{Height,Width}` sets margins
- `l.spacing` sets space between cells


```
composite.setLayout(new FillLayout());
```


```
composite.setLayout(new FillLayout(SWT.VERTICAL));
```

RowLayout

- `RowLayout` places all controls in rows / columns - wraps by default
- is customized through fields of `RowLayout` and `RowData`


```
RowLayout layout
    = new RowLayout();
composite.setLayout(layout);
```

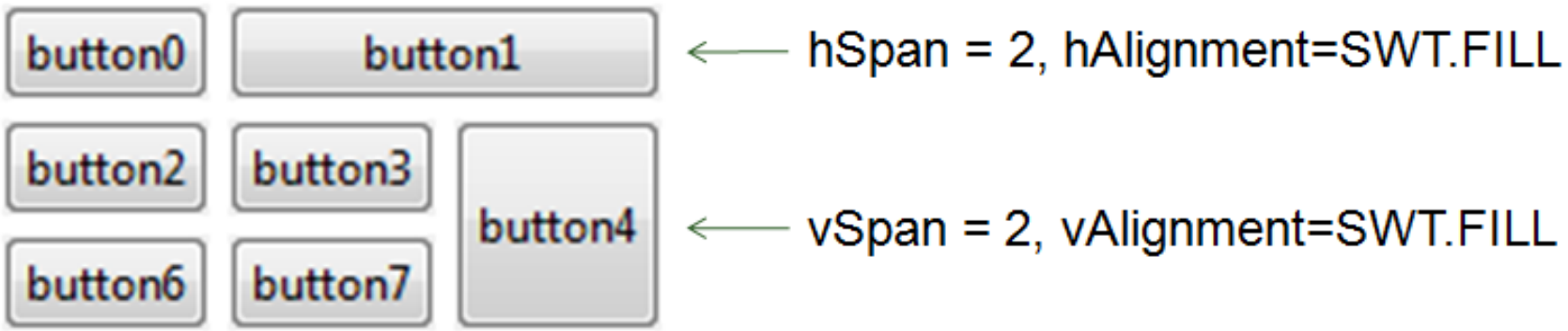

```
RowLayout layout = new RowLayout();
layout.wrap = false;
composite.setLayout(layout);
```


```
RowLayout layout
    = new RowLayout(SWT.VERTICAL);
composite.setLayout(layout);
```


```
RowLayout layout = new RowLayout(SWT.VERTICAL);
layout.wrap = false;
composite.setLayout(layout);
```

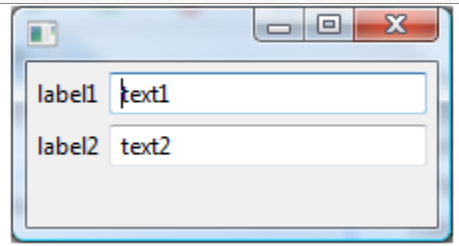
GridLayout

- `GridLayout` positions controls from left-to-right, top-to-bottom in a grid
- is customized through fields of `GridLayout` and `GridData`
- controls can span several rows or columns (see `GridData`)



GridLayout Example

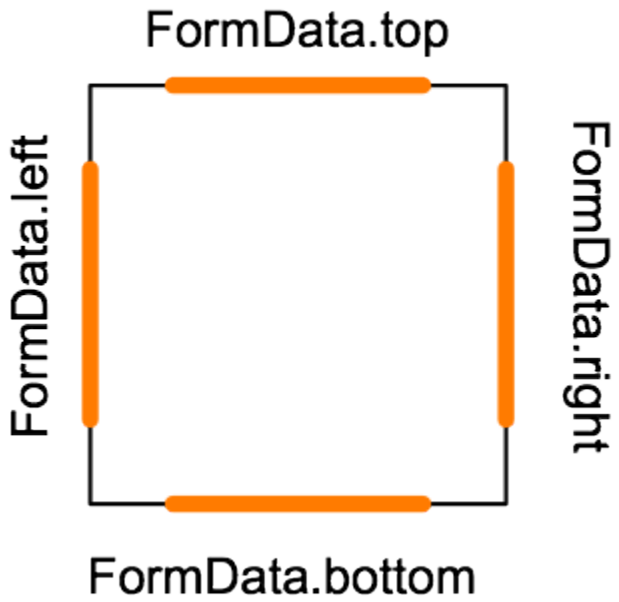
```
shell.setLayout(new GridLayout(2, false));

Label label1 = new Label(shell, SWT.NONE);
label1.setText("label1");
Text text1 = new Text(shell, SWT.BORDER);
text1.setText("text1");
text1.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true, false));

Label label2 = new Label(shell, SWT.NONE);
label2.setText("label2");
Text text2 = new Text(shell, SWT.BORDER);
text2.setText("text2");
text2.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true, false));
```



Pro Tip: `GridLayoutFactory` and `GridDataFactory` can simplify this!

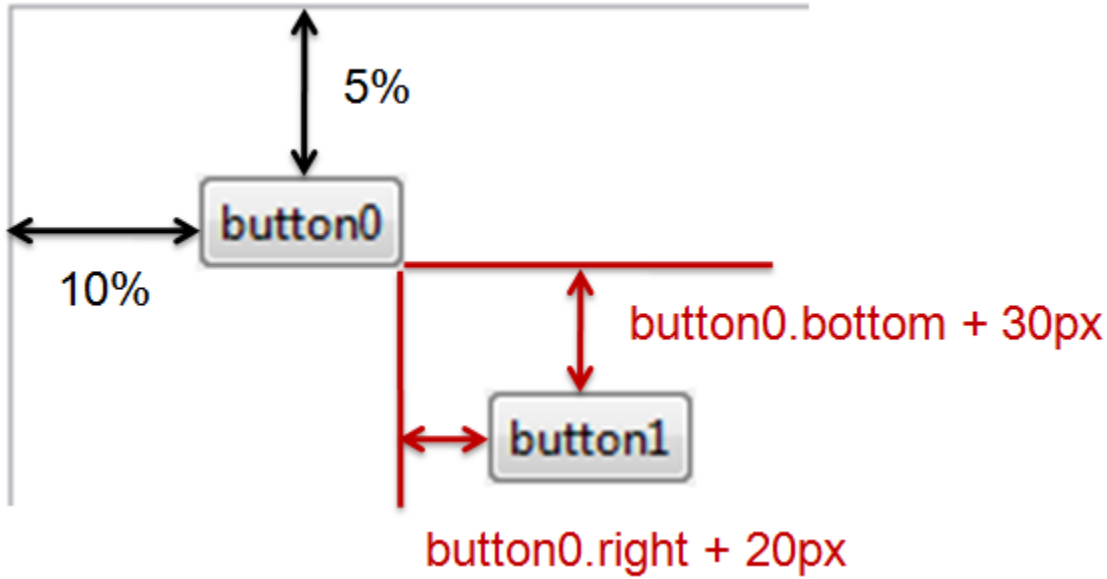
FormLayout

- `FormLayout` positions controls in relation to other controls or the parent
- The control receives a `FormData` object
- A `FormAttachment` can be attached to every side of a control's `FormData` and modifies the placement



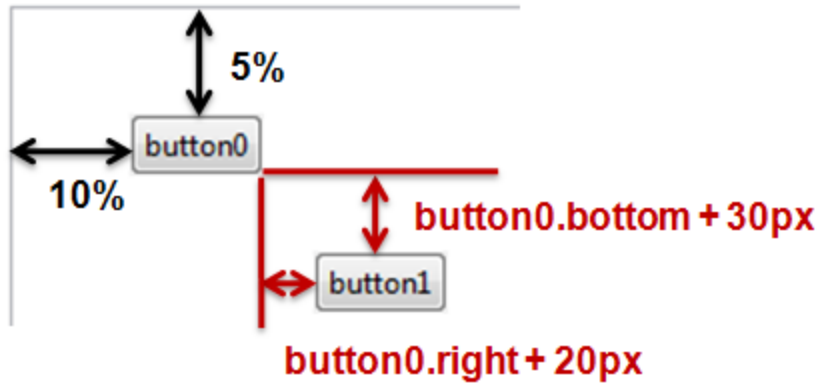
FormAttachment

- `FormAttachments can`
  - refer to a percentage of the parent's width / height
  - relate to the side of another control



FormLayout Example

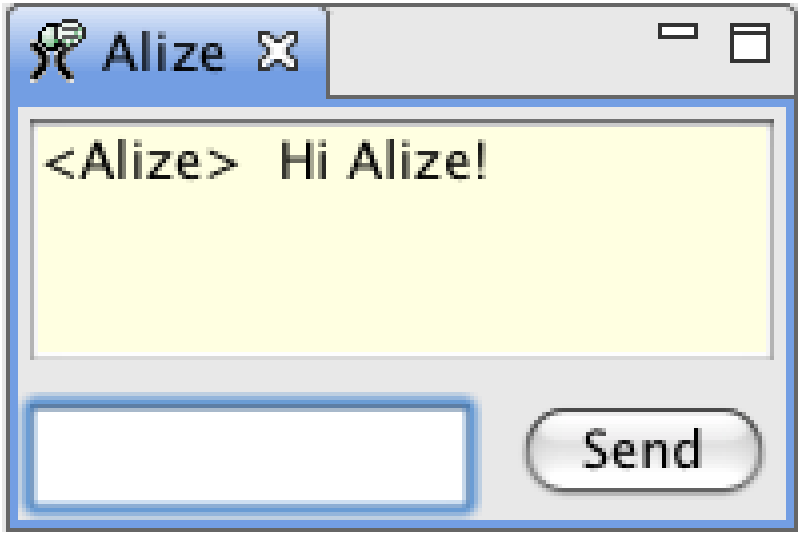
```
shell.setLayout(new FormLayout());
```




```
Button button0 = new Button(shell, SWT.PUSH);
button0.setText("button0");
FormData data = new FormData();
data.left = new FormAttachment(10, 0);
data.top = new FormAttachment(5, 0);
button0.setLayoutData(data);

Button button1 = new Button(shell, SWT.PUSH);
button1.setText("button1");
data = new FormData();
data.left = new FormAttachment(button0, 20);
data.top = new FormAttachment(button0, 30);
button1.setLayoutData(data);
```

Tasks

- Add a **Send** Button to the `ChatEditor`
  - Modify `createPartControl(...)` to change the layout as shown below
  - Attach a `SelectionListener` to the button
  - Clicking the button sends the message in the text field



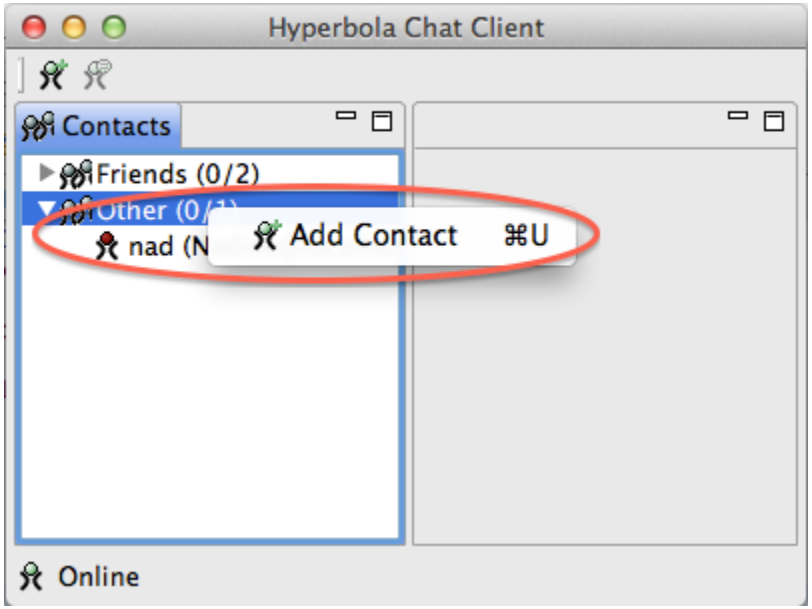
Solution

- If in doubt how to start, use **7.1 Packaging**
- If unsure how to proceed, compare to **9.1 SWT Layouts**

Pointers

- Eclipse WindowBuilder: http://eclipse.org/windowbuilder
- Understanding Layouts in SWT http://eclipse.org/articles/article.php?file=Article-Understanding-Layouts/index.html
- MigLayout: An advanced 3rd party Layout http://www.miglayout.com/

Commands



Commands

- A *Command* is the abstract idea of something to be executed
- The behavior of a Command is implemented within a *Handler*
  - alternatively Actions can be bound to a Command
- Commands can be mapped to a *Key Binding*
- Handlers and Key Bindings can be associated with a *Context*
- A *Context* is a certain state of the workbench

Commands philosophy

- The Command Framework introduces a cleaner separation between view and execution concerns
- Provides a declarative approach for activating and triggering code execution
- Easy binding to keystrokes

Why add additional complexity to actions? A good example is when you need to provide key bindings for standard actions such as **copy**, **cut**, and **paste**. The implementation of these operations depends on the context. A key binding can be defined for the command that applies regardless of the underlying implementation.

Contributing Commands

- To define a Command use `org.eclipse.ui.commands`
  - `command` - Defines the `Command`
    - `id` - the unique identifier String for this command
    - `name` - the UI String for this command
    - `category` - assigns this command to a category (optional)
    - `defaultHandler` - Defines the standard `Handler` for this command (optional)
  - `category` - Defines grouping for multiple commands (optional)

```
<extension point="org.eclipse.ui.commands">
   <command
        categoryId="org.eclipsercp.hyperbola.category"
        id="org.eclipsercp.hyperbola.addContact"
        name="Add Contact">
   </command>
</extension>
```

Command images

- To associate icon(s) with a command use
  - `org.eclipse.ui.commandImages`
    - `commandId` - the identifier of an already declared command
    - `icon` - icon shown when the command is enabled (16x16)
    - `disabledIcon` - optional icon shown when the command is disabled
    - `hoverIcon` - optional icon shown when hovering over the command

```
<extension point="org.eclipse.ui.commandImages">
   <image
        commandId="org.eclipsercp.hyperbola.addContact"
        icon="icons/add_contact.gif"/>
</extension>
```

Handlers

- Handlers define and execute the behavior of Commands
  - there can be only one active Handler per Command
  - a handler can be enabled or disabled
- They are declared by extending `org.eclipse.ui.handlers`
  - `commandId` - the identifier of an already declared command
  - `class` - specifies a the java class with executable code

```
<extension point="org.eclipse.ui.handlers">
  <handler
    commandId="org.eclipse.ui.edit.delete"
    class="org.eclipsercp.hyperbola.RemoveContactHandler">
  </handler>
</extension>
```

Handlers (cont.)

- The handler class has to implement `IHandler`
  - tip: extend the convenience class `AbstractHandler`
  - the `execute(...)` method will run when the user clicks on the command
  - use `HandlerUtil` to get the selection or interact with the workbench

```
public class RemoveContactHandler extends AbstractHandler {

  public Object execute(ExecutionEvent event) throws ExecutionException {
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (selection instanceof IStructuredSelection) {
      IStructuredSelection sSelection = (IStructuredSelection) selection;
      Object first = sSelection.getFirstElement();
      if (first instanceof ContactsEntry) {
        ContactsEntry entry = (ContactsEntry) first;
        entry.getParent().removeEntry(entry);
      }
    }
    return null;
  }
}
```

Handler Activation

- A command can have several Handlers
  - at most **one** active Handler for per command at any time
  - a command with no active handler is grayed out
- The `<activeWhen>` expression controls the activation state:

```
<handler
  class="org.eclipsercp.hyperbola.RemoveContactHandler"
  commandId="org.eclipse.ui.edit.delete">
  <activeWhen>
    <with variable="activePartId">
      <equals value="org.eclipsercp.hyperbola.views.contacts"/>
    </with>
  </activeWhen>
</handler>
```

Tip: Consult `ISources` for a list of available variables!

Actions as Handlers

- An action becomes a handler for a command by setting the *Action Definition ID* to the id of such command

org.eclipsercp.hyperbola/plugin.xml

```
<extension point="org.eclipse.ui.commands">
   <command id="org.eclipsercp.hyperbola.chat" name="Chat"/>
</extension>
```

org.eclipsercp.hyperbola/ChatAction

```
public class ChatAction extends Action...{

 public ChatAction( IWorkbenchWindow window ) {
   ...
   setActionDefinitionId( "org.eclipsercp.hyperbola.chat" );
   ...
 }
```

Actions as Handlers (cont.)

- To bind the action to the command it is necessary to call `ActionBarAdvisor.register(IAction)`

```
org.eclipsercp.hyperbola/ApplicationActionBarAdvisor
 protected void makeActions( IWorkbenchWindow window ) {
   chatAction = new ChatAction( window );
   register( chatAction );
 }
```

Standard Actions as Handlers

- Standard actions like **Quit**, **Copy** or **Paste** already are handlers for predefined commands
- The first argument to `WorkbenchCommandAction` is the ID of the command that's behind the action
- The predefined commands from the Eclipse Platform are listed in class `org.eclipse.ui.IWorkbenchCommandConstants`

```
org.eclipse.ui.actions/ActionFactory.QUIT
   public IWorkbenchAction create( IWorkbenchWindow window ) {
     ...
     WorkbenchCommandAction action = new WorkbenchCommandAction(
                                IWorkbenchCommandConstants.FILE_EXIT, window )
     ...
     return action;
   }
```

Key Bindings

- Key bindings are defined by contributing to
  - `org.eclipse.ui.bindings`
- A key binding associates a key sequence with a Command
- Those associations are specific to a *Context* and a *Scheme*

```
<extension point="org.eclipse.ui.bindings">
  <key
        sequence="M1+Q"
        commandId="org.eclipse.ui.file.exit"
        contextId="org.eclipse.ui.contexts.window"
        schemeId="org.eclipse.ui.defaultAcceleratorConfiguration"/>
</extension>
```

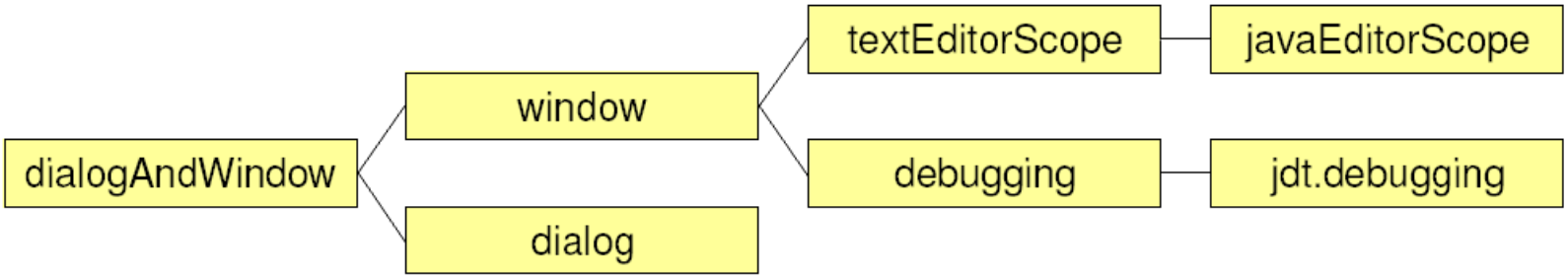
Key sequences

- Key *sequences* are sets of key *chords* separated by spaces
  - A key chord represents one or more keys held down at the same time
  - A chord can have zero or more modifier keys and exactly one other key
  - The keys in a chord are separated by the "+" character
- Keys are specified as uppercase ASCII characters
- Meta keys **M1** (CTRL/Command), **M2** (Shift), **M3** (Alt) to **M4** (CTRL on Mac) are used as a platform independent key description
- Examples of valid key sequences: "M1+C", "M2+M3+X T", "CTRL+SPACE"

For a list of other non-printable keys see **Help** > **Platform Plug-in Developer Guide** > **Reference** > **Extension Points Reference** > **org.eclipse.ui.bindings**

Contexts

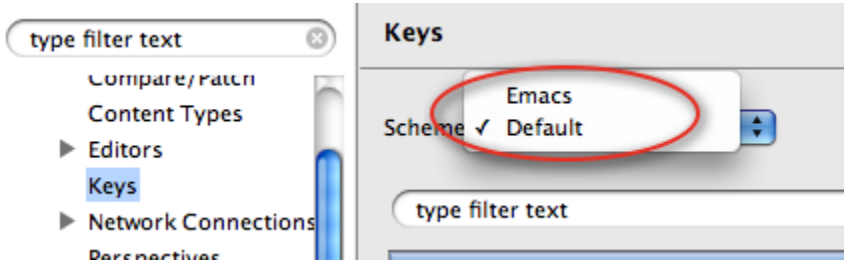
- A *Context* can be used to influence which key bindings are available to the user at any given moment
  - a context is an answer to the question "when?"
- Quite a number of contexts are already defined:



- For example, the key bindings available to a user while editing text might be different than those available while editing Java code or browsing packages in the Package Explorer

Schemes

- A *Scheme* is a set of key bindings
- There are two standard schemes available:
  - `org.eclipse.ui.defaultAcceleratorConfiguration`
    - the **default scheme** for an RCP application
    - use this one in your key bindings
  - `org.eclipse.ui.emacsAcceleratorConfiguration`
    - Emacs-like key bindings



- You can define your own scheme by extending
  - `org.eclipse.ui.bindings`

Adding commands to the UI

- Commands are placed in menus, pop-ups and toolbars (including the trim areas and status line) by extending
  - `org.eclipse.ui.menus`
- use `<menuContribution>` to add commands to the user interface
  - `locationURI` controls placement (see next slides)

Example – places a command into an existing menu:

```
<extension point="org.eclipse.ui.menus">
  <menuContribution
        locationURI="menu:hyperbola">
        <command commandId="org.eclipsercp.hyperbola.chat"/>
   </menuContribution>
</extension>
```

Location URIs

- The `locationURI` determines where commands are placed
- It has the format **`scheme:id`**
  - `scheme` can have one of these values:
    - `menu` – places commands into a menu
    - `toolbar` – places commands into a toolbar, trim area, status bar
    - `popup` – places commands into a pop-up menu
  - `id` is the unique identifier of an existing UI element (i.e. menu, toolbar, command, view part)

```
menu:hyperbola
menu:org.eclipse.ui.main.menu
menu:org.eclipsercp.hyperbola.views.contacts
toolbar:org.eclipse.ui.main.toolbar
toolbar:org.eclipsercp.hyperbola.views.contacts
popup:org.eclipse.ui.popup.any
popup:org.eclipsercp.hyperbola.views.contacts
```

Predefined Location URIs

- A list of predefined locationURI's:
  - menu:org.eclipse.ui.main.menu - the top-level menu
  - toolbar:org.eclipse.ui.main.toolbar - the top-level tool bar
  - toolbar:org.eclipse.ui.trim.command1 - the top left trim area
  - toolbar:org.eclipse.ui.trim.command2 - the top right trim area
  - toolbar:org.eclipse.ui.trim.vertical1 - the left vertical trim area
  - toolbar:org.eclipse.ui.trim.vertical2 - the right vertical trim area
  - toolbar:org.eclipse.ui.trim.status - the status line trim area
  - popup:org.eclipse.ui.popup.any - all pop-up menus

Location URIs - placement (cont.)

- To control placement more precisely the following can be added to `locationURI`:
  - append `?before=xyz.id` for placement before a certain contribution
  - append `?after=xyz.id` for placement after a certain contribution

```
<extension point="org.eclipse.ui.menus">
  <menuContribution
     locationURI="menu:hyperbola?before=additions">
    <command
          commandId="org.eclipsercp.hyperbola.chat"/>
  </menuContribution>
</extension>
```

Adding a new menu

- This adds a new menu to the menubar and places a command in it:

```
<extension point="org.eclipse.ui.menus">
  <menuContribution
        locationURI="menu:org.eclipse.ui.main.menu">
        <menu label="Edit">
          <command
              commandId="org.eclipse.ui.edit.copy"/>
    </menu>
  </menuContribution>
</extension>
```

Adding to the coolbar

- This adds a new toolbar to the coolbar and places a command in it:

```
<extension point="org.eclipse.ui.menus">
  <menuContribution
        locationURI="toolbar:org.eclipse.ui.main.toolbar">
        <toolbar
            id="org.eclipsercp.hyperbola.sampleToolbar">
            <command
                commandId="org.eclipsercp.hyperbola.chat"/>
        </toolbar>
   </menuContribution>
</extension>
```

Adding a pop-up menu

- A pop-up menu is created by a menu manager for a given control

- The pop-up menu has to be registered with the control
- To be available to contributions by the workbench, it has also to be registered with the site

ContactsView.java

```
private void createContextMenu(Viewer viewer) {
  MenuManager menuMgr = new MenuManager();
  menuMgr.add(new GroupMarker("additions"));
  Menu menu = menuMgr.createContextMenu(viewer.getControl());
  viewer.getControl().setMenu(menu);
  getSite().registerContextMenu(menuMgr, viewer);
}
```

Adding to a pop-up menu

- This places a command into **all** pop-up menus which have an `"additions"` marker
  - better: use `popup:org.eclipsercp.hyperbola.views.contacts` to target a pop-up menu in a **specific** view or editor

```
<extension point="org.eclipse.ui.menus">
 <menuContribution
     locationURI="popup:org.eclipse.ui.popup.any">
   <command commandId="org.eclipsercp.hyperbola.chat">
      <visibleWhen
            checkEnabled="true"/>
   </command>
 </menuContribution>
</extension>
```

Tasks

- Assign the key sequence M1+Q to the **Exit** command with
  - `org.eclipse.ui.bindings`
- Set the action definition ids for the "Chat" and "Add Contact" Actions
- Create two commands with these action definition ids using
  - `org.eclipse.ui.commands`
- Add key bindings to the new commands using
  - `org.eclipse.ui.bindings`
- Edit ContactsView.java to create a pop-up menu
  - add a `createContextMenu()` method (see slides)
  - call this method from the end of `createPartControl(...)`

Tasks (cont).

- Put both commands into the pop-up by using
  - `org.eclipse.ui.menus`
    - add a `menuContribution` element
      - set `locationId` set to `popup:id.of.your.contacts.view`
    - add two `command` elements to `menuContribution`
      - set `commandId` to match the id's of the commands you created before
- Optional – make the commands visible only when the appropriate type is selected:

```
<visibleWhen>
   <iterate>
      <instanceof
            value="org.eclipsercp.hyperbola.model.ContactsEntry">
            <!-- or ContactsGroup -->
      </instanceof>
   </iterate>
</visibleWhen>
```

Solution

- If in doubt how to start, use **12.1 OSGi Essentials**
- If unsure how to proceed, compare to **13.1 Commands**

Pointers

- UML Diagram explaining the command framework
- Differences between Actions and Commands
- Links:
  - http://wiki.eclipse.org/Platform_Command_Framework
  - http://wiki.eclipse.org/Menu_Contributions
  - http://wiki.eclipse.org/Menus_Extension_Mapping
  - http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/workbench.htm

Conclusion

Questions and Answers

- Do you have any further questions?

- Check out our blog: eclipsesource.com/blogs
- Follow us on Twitter: twitter.com/eclipsesource

Thanks

- Thanks for attending!

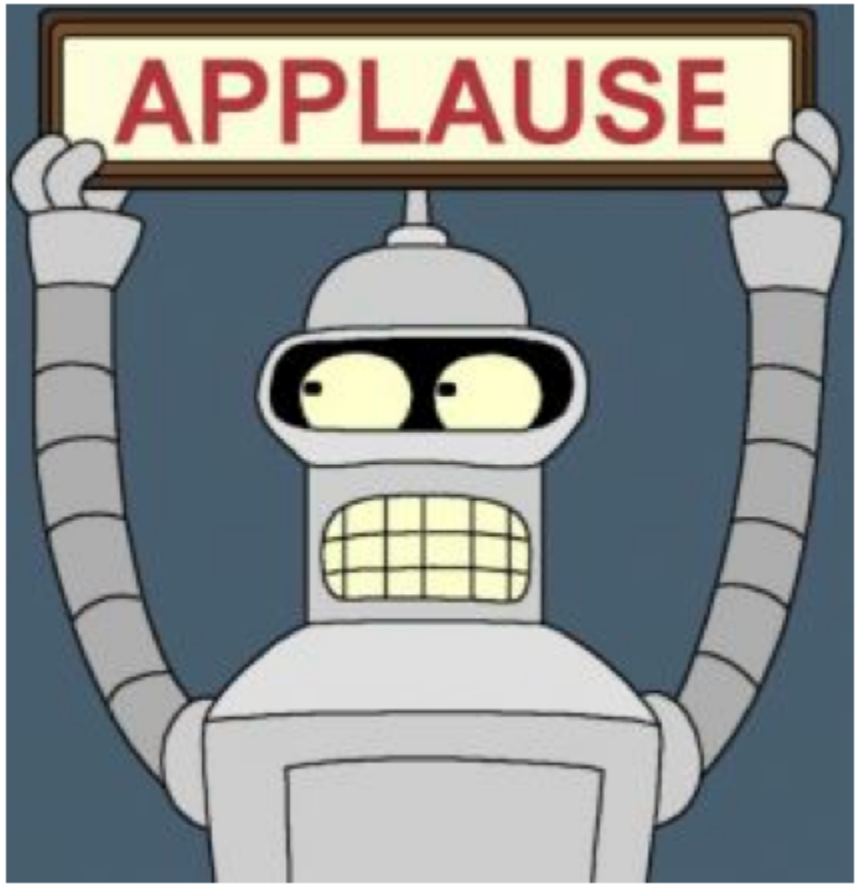


Image Credits

- jigsaw.jpg
  - http://www.flickr.com/photos/35577089@N00/6092851547/
  - by annie
  - Attribution 2.0 Generic (CC BY 2.0)
- power_sockets.jpg
  - http://www.flickr.com/photos/aigarius/5960629676/
  - by aigarius
  - Attribution 2.0 Generic (CC BY 2.0)
- russian_dolls.jpg
  - http://www.flickr.com/photos/marinashemesh/6770690477/
  - by Marina Shemesh
  - Attribution-ShareAlike 2.0 Generic (CC BY-SA 2.0)

Misc

- Powered by Prince XML (http://princexml.com/license/)