eclipseCON 2006

Rich Client Application Development

A Tutorial based on the Eclipse Rich Client Platform (eclipsercp.org)

Jean-Michel Lemieux and Jeff McAffer
IBM Rational Software

Introduction

15 minutes

Who are we?

- Jean-Michel Lemieux
- Jeff McAffer

Who are you?

- Eclipse usage?
- Eclipse plug-in development?
- Building RCP application today?
- What domains?
- Care to share?

What are we doing here?

- Learn how to use the tools to create an RCP application
- Learn how to write RCP applications
- Learn how to brand and package RCP applications

Introduction

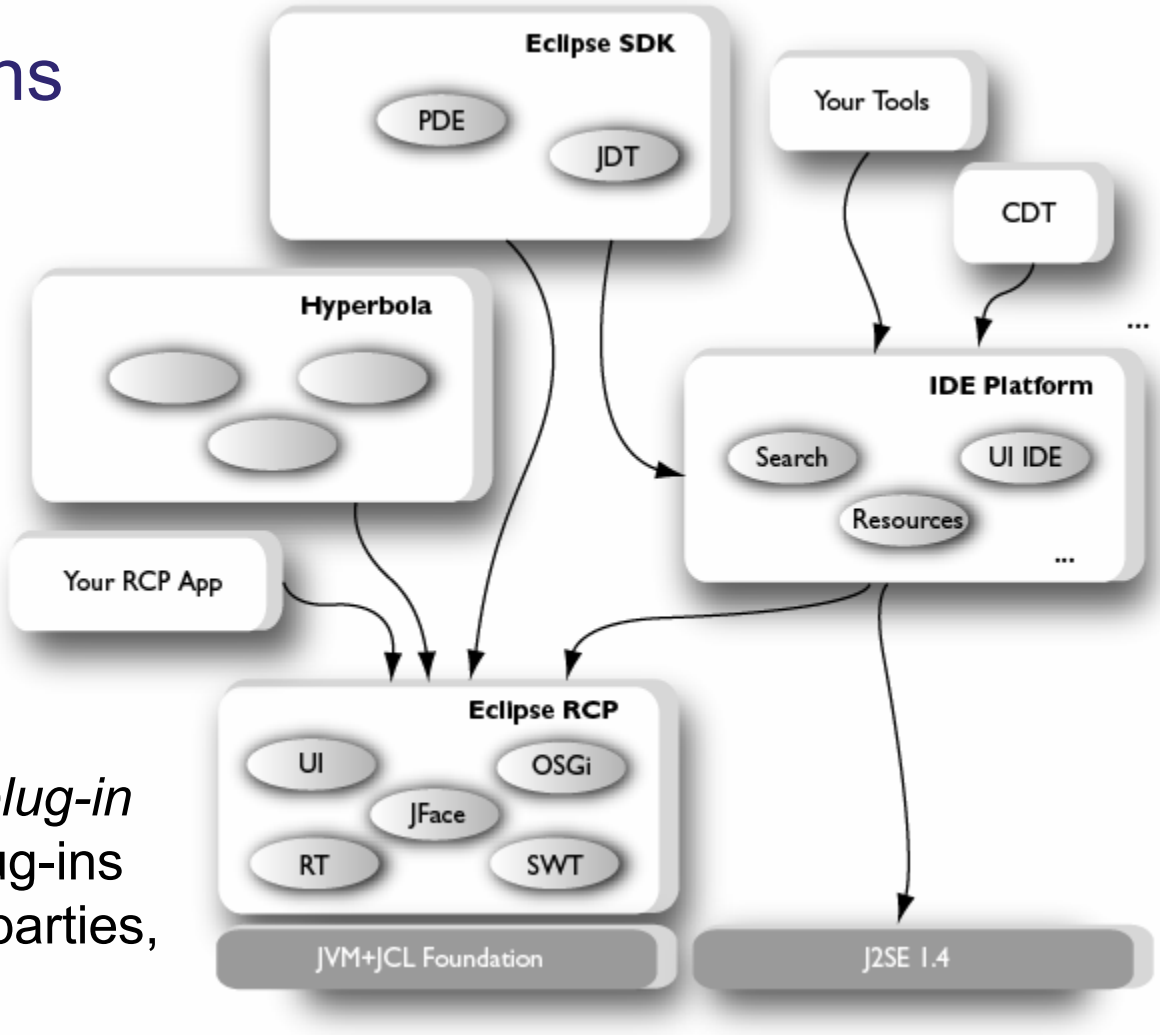
- Writing RCP applications is different than just writing plug-ins. You have the opportunity to define more of the look and feel, the branding, and other fundamental elements of Eclipse which are not exposed to plug-in developers.
- Traditional Eclipse plug-in development is focused on plugging into an IDE. RCP development expands the boundaries of your application and pushes you to think about frameworks of your own and how others will integrate into your product.
- RCP architects are involved in more than just writing the app: architecture (how many plug-ins, layering), branding, building (releng, multiple platforms), deployment (how to get it to customers).

One minute sales pitch

How would you explain RCP to your boss in 8 bullets?

- Components
- Middleware and infrastructure
- Native user experience
- Portability
- Install and Update
- Disconnected Operation
- Development tooling support
- Component libraries

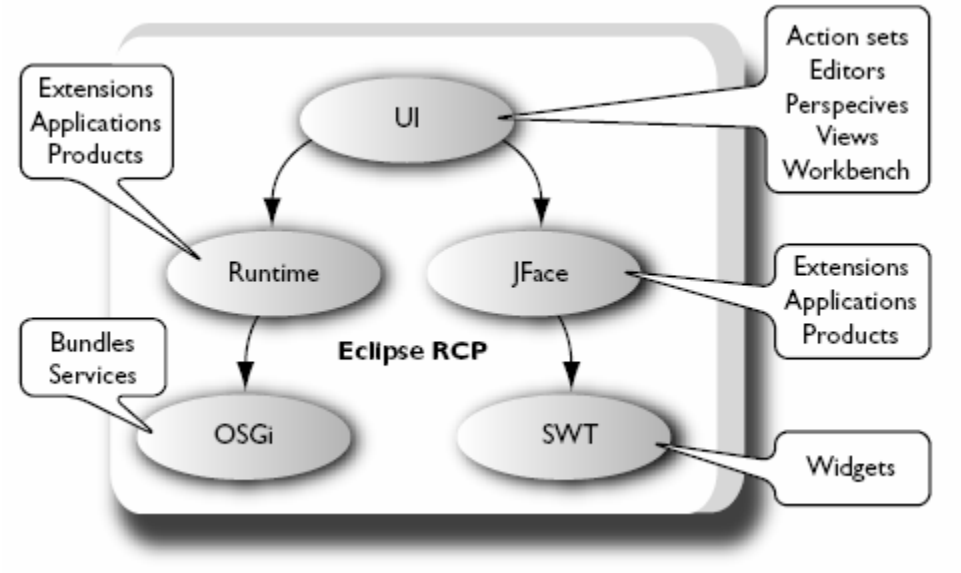
Community of plug-ins



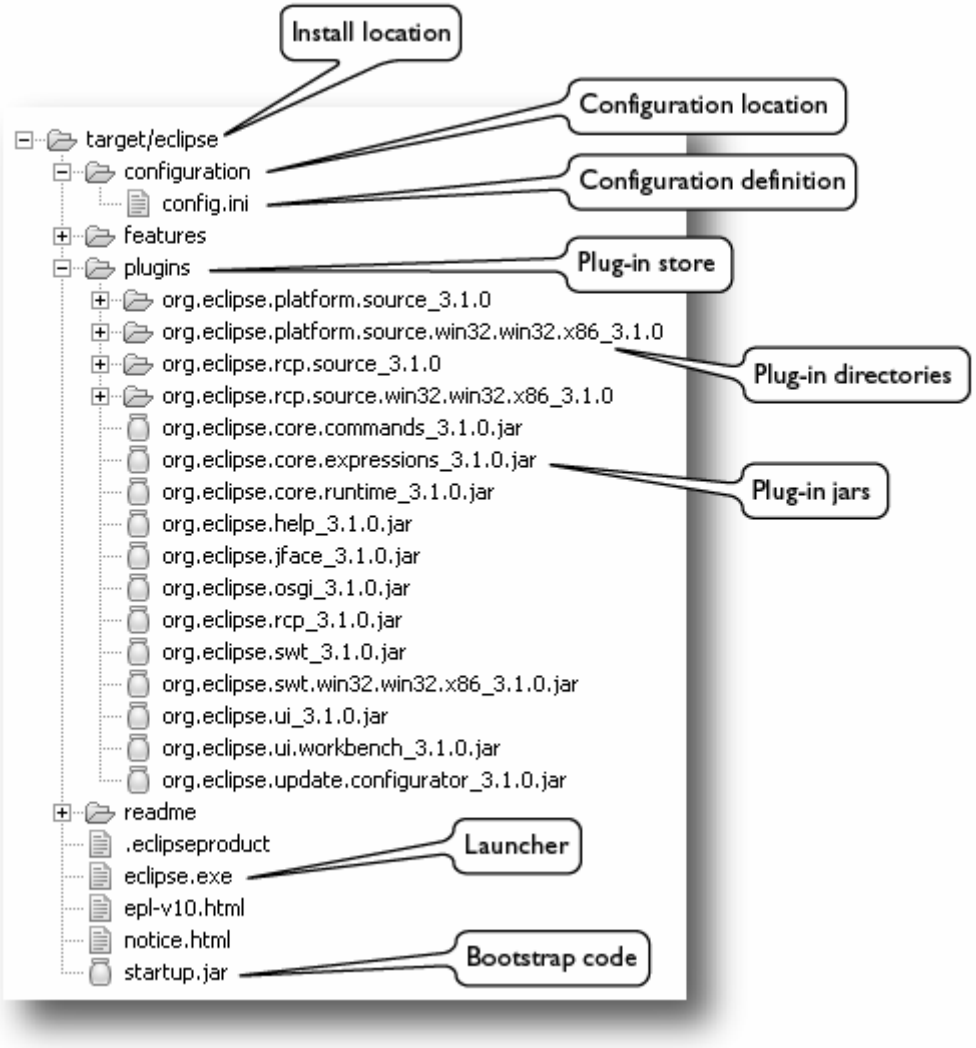
- Basic unit of function is a *plug-in*
- RCP developers collect plug-ins from the Eclipse base, 3rd parties, and develop their own

Eclipse RCP Base

- You are in fact free to slice and dice the RCP itself or any other plug-in set to suit your needs as long as the relevant plug-in interdependencies are satisfied. In this book, we focus on *RCP applications* as applications that use the full RCP plug-in set.



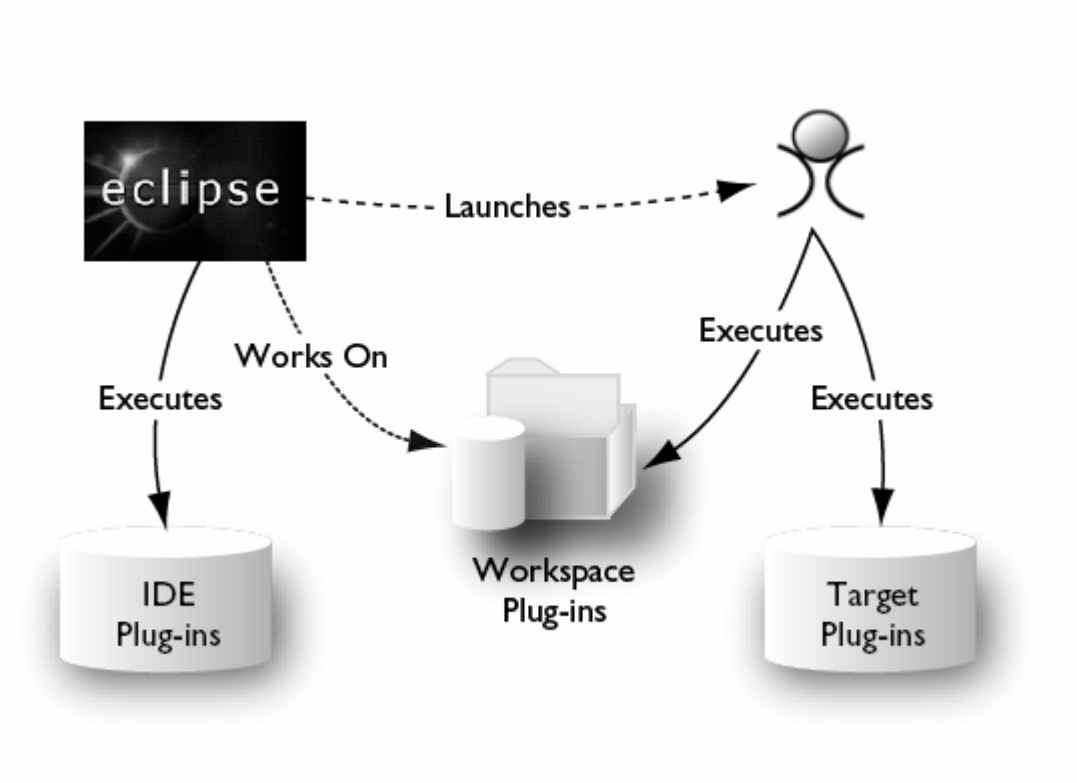
What does it look like?



Getting Started

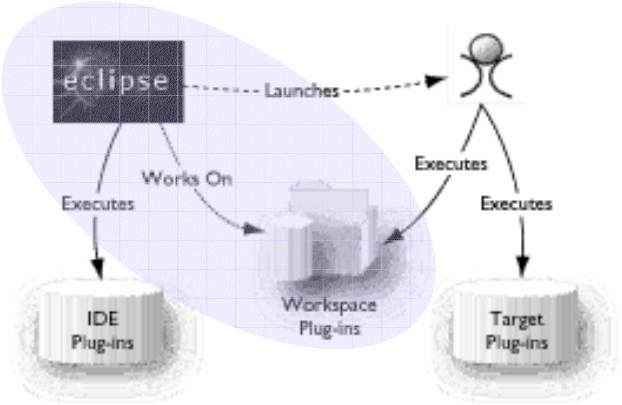
15 minutes

Eclipse Development Workflow

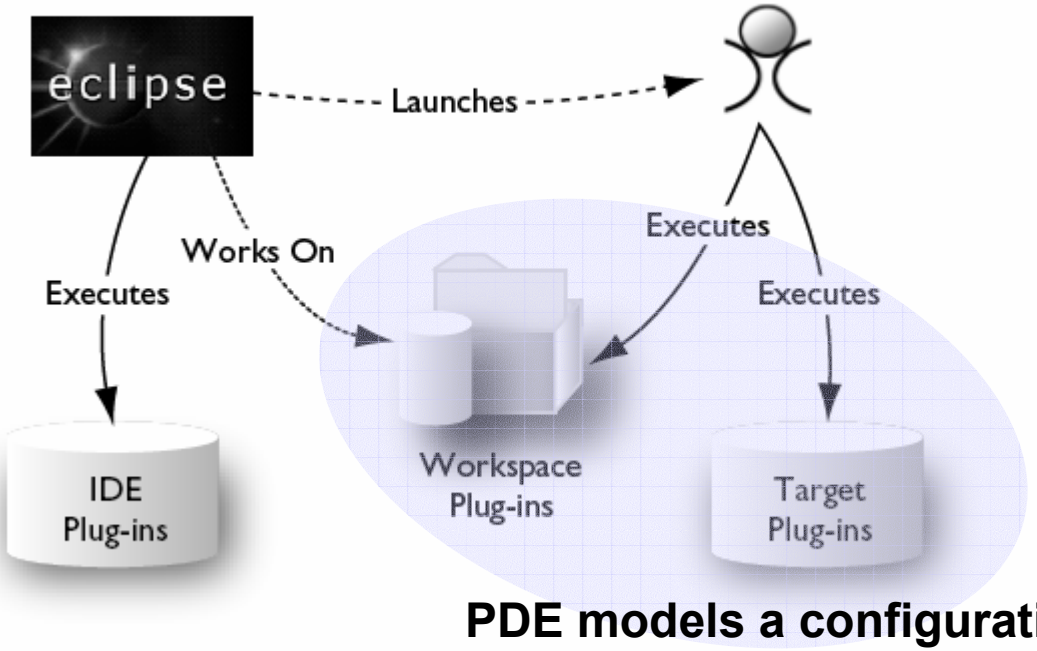


IDE Installation

- Using Eclipse 3.1 SDK for this tutorial
  - **eclipse-SDK-3.1-win32.zip**
- Install at wherever you like
  - We use c:\ide
- Start with an empty workspace
  - We use c:\workspace



What is a “Target Platform”?

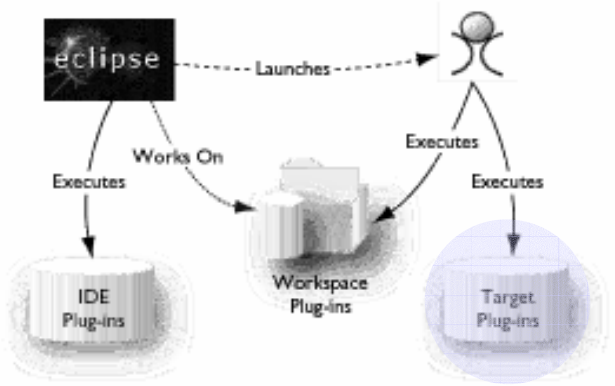


PDE models a configuration

Configuration = Workspace + Target

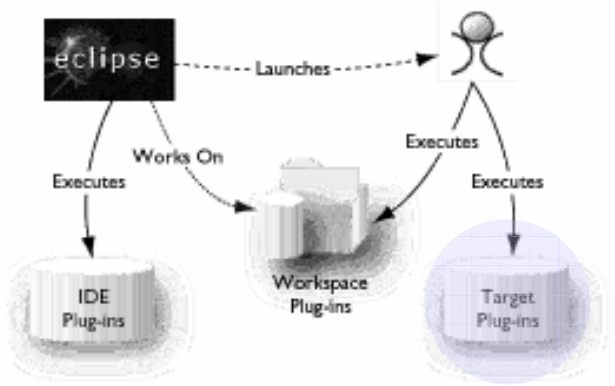
Why Separate the Target and IDE?

- Cross development
- Develop for different versions of Eclipse
  - Use leading edge 3.2M5 to develop product based on stable 3.1
- Have many different targets
- Update IDE and target independently
  - Add new tooling function to the IDE
  - Add new function to target that may not work in IDE
- Some hassles but lots of power/flexibility

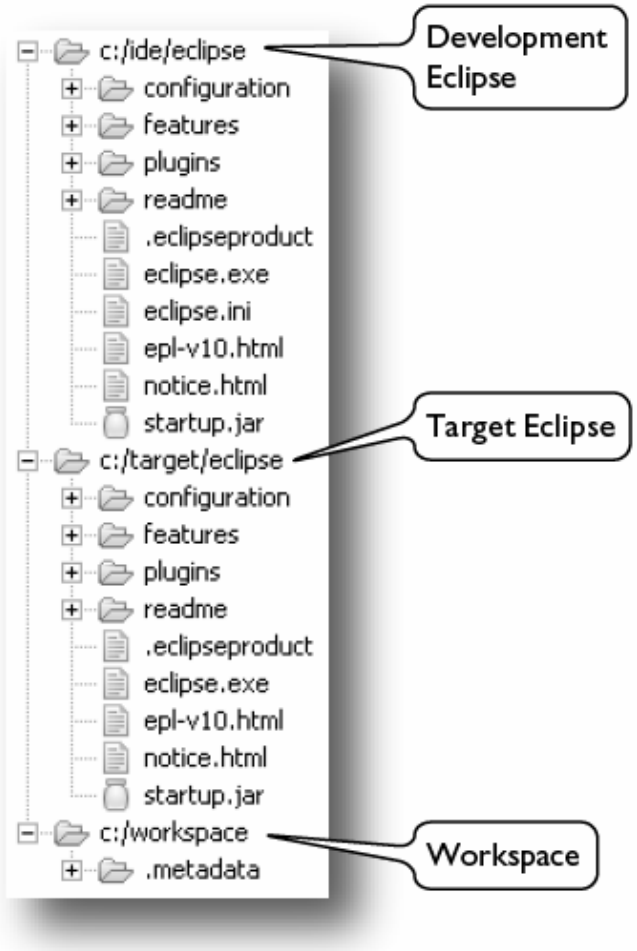


Target Installation

- Using Eclipse 3.1 RCP **SDK** for this tutorial
  - **eclipse-RCP-SDK-3.1-win32.zip**
- Install at wherever you like
  - System defaults to use IDE itself as target!
  - We use c:\target
- See Target Platform location
  - Window > Preferences > Plug-in Development > Target Platform

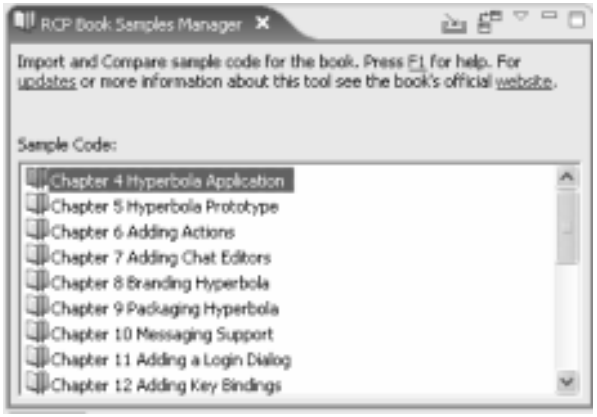


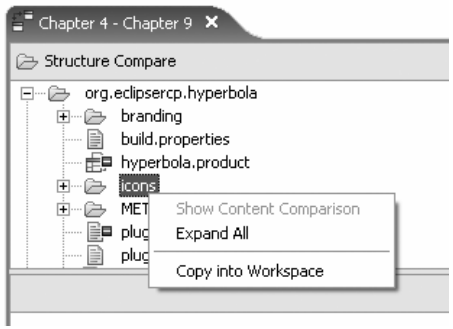
Checkpoint


c:/ide/eclipse
configuration
features
plugins
readme
.eclipseproduct
eclipse.exe
eclipse.ini
epl-v10.html
notice.html
startup.jar
c:/target/eclipse
configuration
features
plugins
readme
.eclipseproduct
eclipse.exe
epl-v10.html
notice.html
startup.jar
c:/workspace
.metadata
Development Eclipse
Target Eclipse
Workspace

Sample Manager

- Tool for loading and comparing various stages of the sample application in this tutorial (not part of Eclipse itself)
- Lists ***final code*** for each stage
  - Load stage N-1 to start work on stage N
  - Compare as you go to correct errors, get large code chunks or data files (e.g., icons), see what's next
- Loading new stage removes content of old





Installing the Samples Manager

1. **Help > Software Updates > Find and Install…**
2. **RCP Book > Samples Manager**.
3. **New Local Site**
4. Identify the location of the “updates” directory from the CD
5. Choose “org.eclipsercp.book.tools.feature”
6. Continue through wizard
7. No need to restart, just **Apply Changes Now**

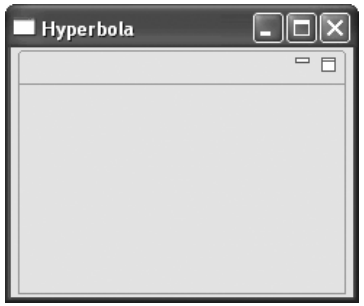
After installing use **RCP Book > Samples Manager**

The Hello Hyperbola RCP Application

1. **File > New > Project…**
2. Plug-in Project
3. Enter “org.eclipsercp.hyperbola”
4. Uncheck the **Plug-in Class** option
5. Select the **Yes** radio button in the **Rich Client Application** area of the page
6. On the RCP **Templates** page choose the **Hello RCP** template
7. Enter “Hyperbola” for the **Application Window Title**

Launching Hyperbola

1. Open Plug-in editor by double-clicking on Hyperbola's
   - META-INF/MANIFEST.MF
   - plugin.xml
2. Use the links in the **Testing** section of the **Overview** page
3. Click on the **Launch an Eclipse application** link
4. Hyperbola starts and looks like this



Exercise: Get setup and run Hyperbola

- Install IDE
- Install target
- Configure target into IDE
- Install Sample Manager
- Create Hyperbola shell application
- Run Hyperbola

10 minutes

Tour of the Code

20 minutes

Application

```
org.eclipsercp.hyperbola/Application
public class Application implements IPlatformRunnable {
   public Object run(Object args) throws Exception {
      Display display = PlatformUI.createDisplay();
      try {
         int returnCode = PlatformUI.createAndRunWorkbench(
            display, new ApplicationWorkbenchAdvisor());
         if (returnCode == PlatformUI.RETURN_RESTART) {
            return IPlatformRunnable.EXIT_RESTART;
         return IPlatformRunnable.EXIT_OK;
      } finally {
         display.dispose();
      }
   }
}
```

Must implement

run() can do anything you want

Start UI; headless apps don't need one

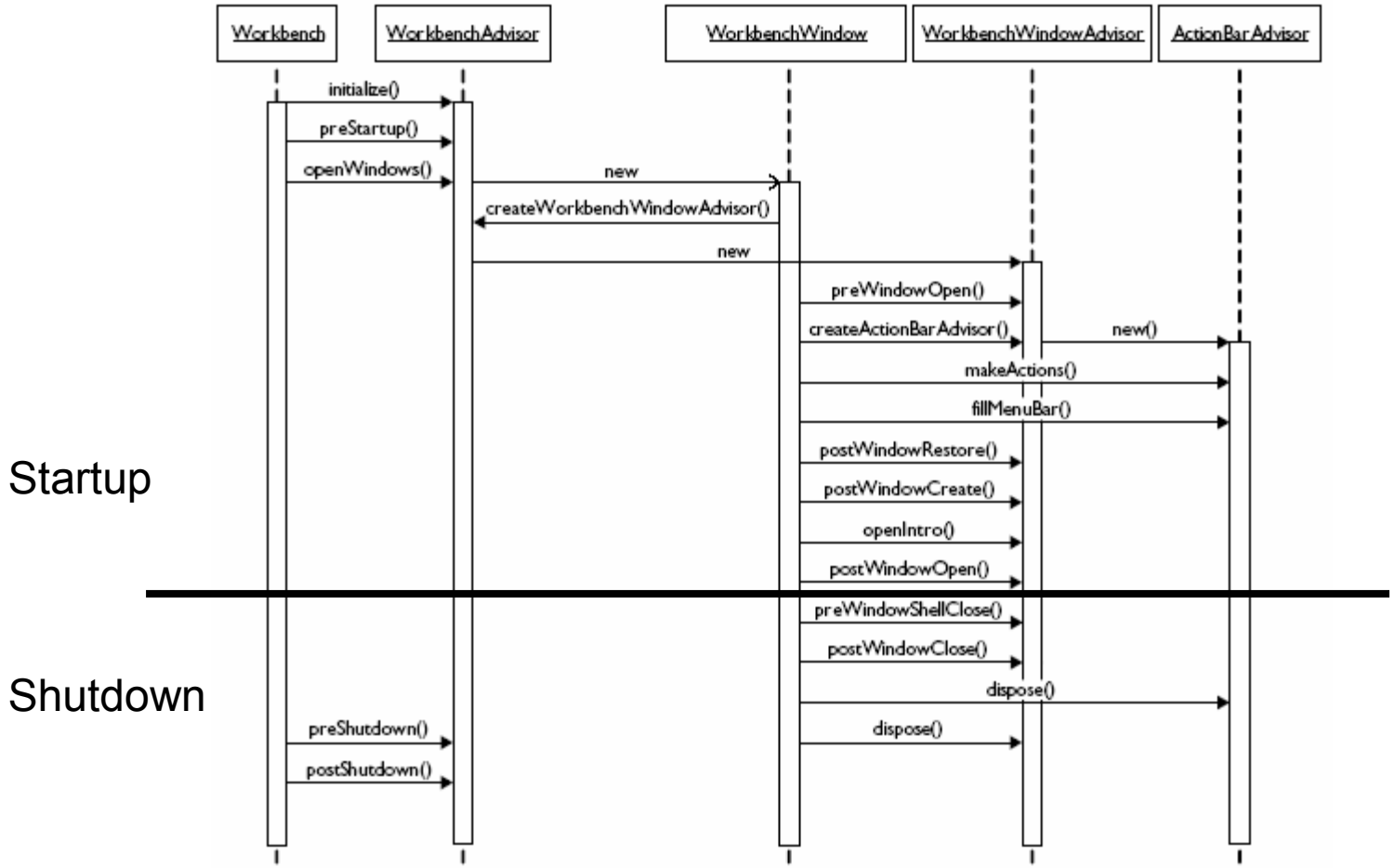
Application

run() doesn't have to start the Workbench.

smack.testing/Application

```
public Object run(Object args) throws Exception {
  try {
    XMPPConnection con = new XMPPConnection("eclipsercp.org");
    con.login("reader", "secret",
    Long.toString(System.currentTimeMillis()));
    Chat chat = con.createChat("eliza@eclipsercp.org");
    chat.sendMessage("Hi There!");
    Message message = chat.nextMessage(5000);
    System.out.println("Returned message: "
        + (message == null ? "<timed out>" : message.getBody()));
  } catch (XMPPException e) {
    e.printStackTrace();
  }
  return IPlatformRunnable.EXIT_OK;
}
```

Workbench Timeline



WorkbenchAdvisor

Must extend

```
org.eclipsercp.hyperbola/ApplicationWorkbenchAdvisor
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {
   public WorkbenchWindowAdvisor createWorkbenchWindowAdvisor(
         IWorkbenchWindowConfigurer configurer) {
      return new ApplicationWorkbenchWindowAdvisor(configurer);
   }
   public String getInitialWindowPerspectiveId() {
      return “org.eclipsercp.hyperbola.perspective”;
   }
}
```

WindowAdvisor defines look of all windows

Id of perspective extension to use for new windows

Perspective

org.eclipsercp.hyperbola/Perspective

```
public class Perspective implements IPerspectiveFactory {
   public void createInitialLayout(IPageLayout layout) {
   }
}
```

Empty implementation

WorkbenchAdvisor

Must extend

```
org.eclipsercp.hyperbola/ApplicationWorkbenchWindowAdvisor
public class ApplicationWorkbenchWindowAdvisor extends WorkbenchWindowAdvisor {
    public ApplicationWorkbenchWindowAdvisor(
            IWorkbenchWindowConfigurer configurer) {
        super(configurer);
    }
    public ActionBarAdvisor createActionBarAdvisor(
            IActionBarConfigurer configurer) {
        return new ApplicationActionBarAdvisor(configurer);
    }
    public void preWindowOpen() {
        IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
        configurer.setInitialSize(new Point(250, 350));
        configurer.setShowCoolBar(false);
        configurer.setShowStatusLine(false);
        configurer.setTitle("Hyperbola");
    }
}
```

ActionBarAdvisor defines the toolbars, menus, …

Control the general look of the window

ActionBarAdvisor

Must extend

```
org.eclipsercp.hyperbola/ApplicationActionBarAdvisor
public class ApplicationActionBarAdvisor extends ActionBarAdvisor {
   public ApplicationActionBarAdvisor(IActionBarConfigurer configurer) {
      super(configurer);
   }

   protected void makeActions(IWorkbenchWindow window) {
      …
   }
   protected void fillMenuBar(IMenuManager menuBar) {
      …
   }
}
```

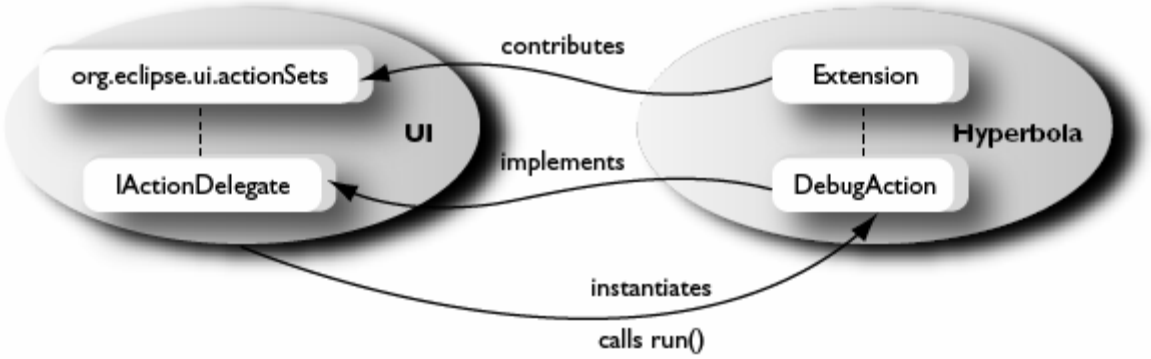
Create the various actions needed by this window

Add the actions to the menu bar

The Extension Registry

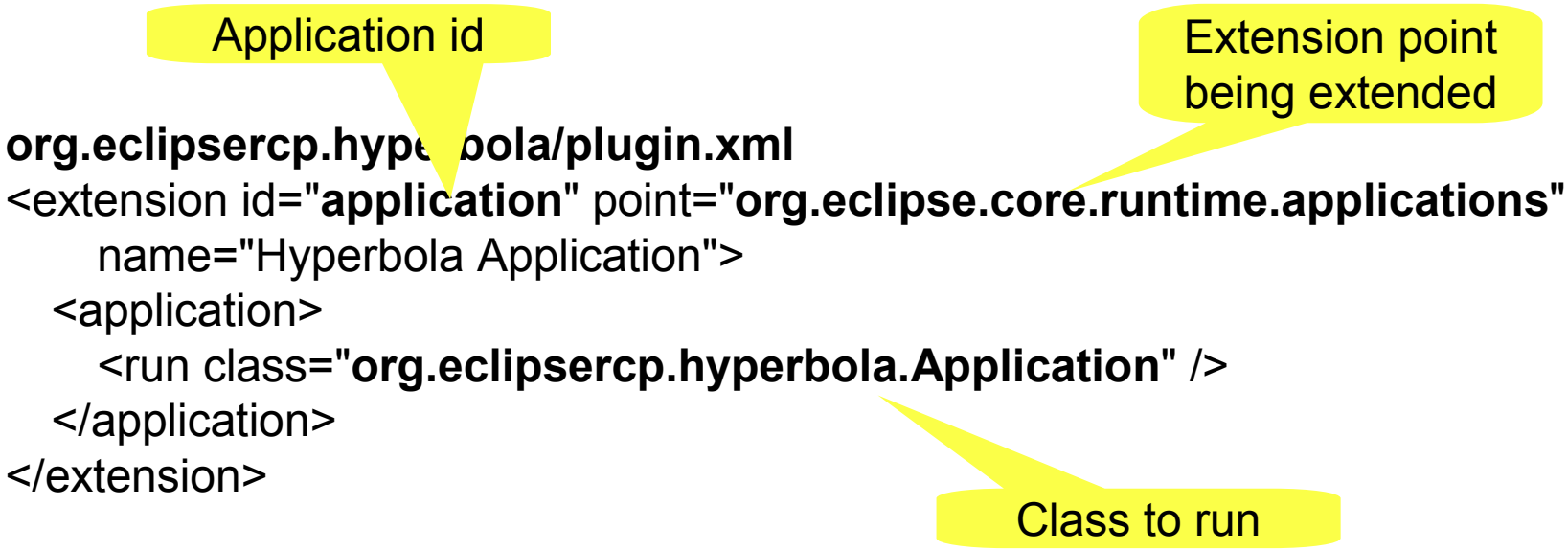
- *Extension Registry* : declarative relationships between plug-ins
- *Extension Point* : plug-ins open themselves for configuration/extension
- *Extension* : plug-in extends another by contributing an extension

"Plug-ins can contribute *actionSets* extensions that define actions with an id, a label, an icon, and a class that implements the interface *IActionDelegate*. The UI will present that label and icon to the user, and when the user clicks on the item, the UI will instantiate the given action class, cast it to *IActionDelegate*, and call its *run()* method."



Defining the application

The application is defined as an extension



Running the application

Full application id is plug-in id + extension id

- Run eclipse and pick the application

  eclipse -application **org.eclipsercp.hyperbola.application**

- Run the JRE and pick the application

  java -jar startup.jar -application **org.eclipsercp.hyperbola.application**

Perspectives, Actions, and Views

60 minutes

Perspectives, Actions, Views, and all that

At this stage, you've ween the skeleton for Hyperbola. You know how to run it, debug it, and are familiar with the basic classes that are part of all RCP applications.

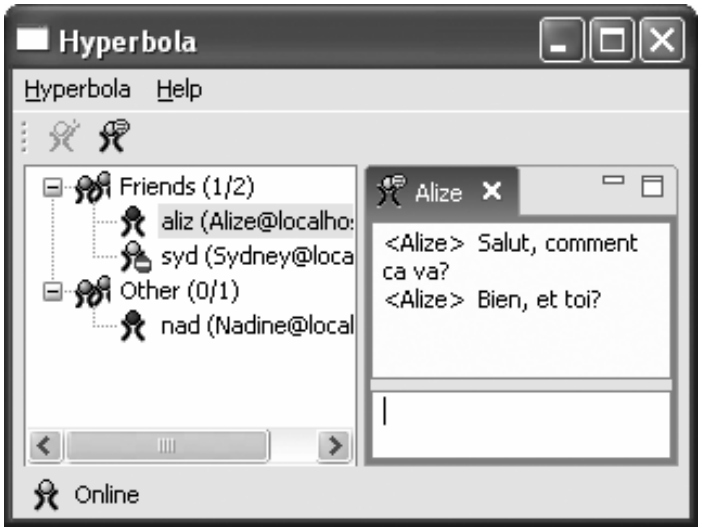
Next we will focus on Perspectives, Actions, and Views with the goal of understanding the basics of the Workbench RCP features.

Note: FOCUS on RCP things!!! Won't get into generic Workbench topics in the tutorial…

Perspectives, Actions, Views

In the end Hyperbola will look like this…

Some views, an editor, many menus with actions, …



The chat client domain is a good example for experimenting with UI aspects of RCP

Perspectives, Actions, Views

In Eclipse, users interact with applications through *views* and e*ditors*.

Perspectives are a mechanism for arranging views and editors and supporting scalable UIs. Put another way, views and editors contain the content for your application; perspectives allow those elements to be organized for users.

We already have a perspective, it's just empty.

Perspectives, Actions, Views

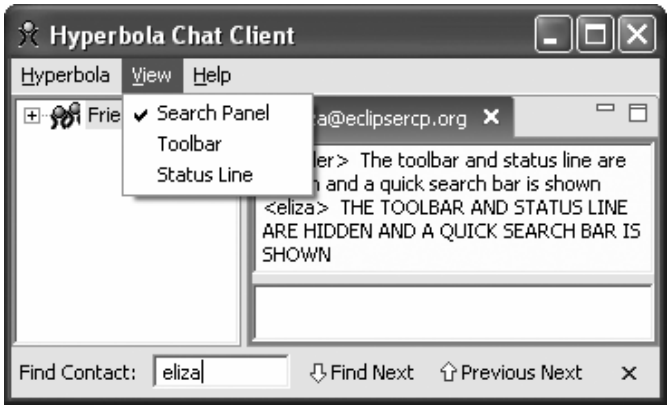
Ok, this is all great stuff but there is nothing RCP specific about it!

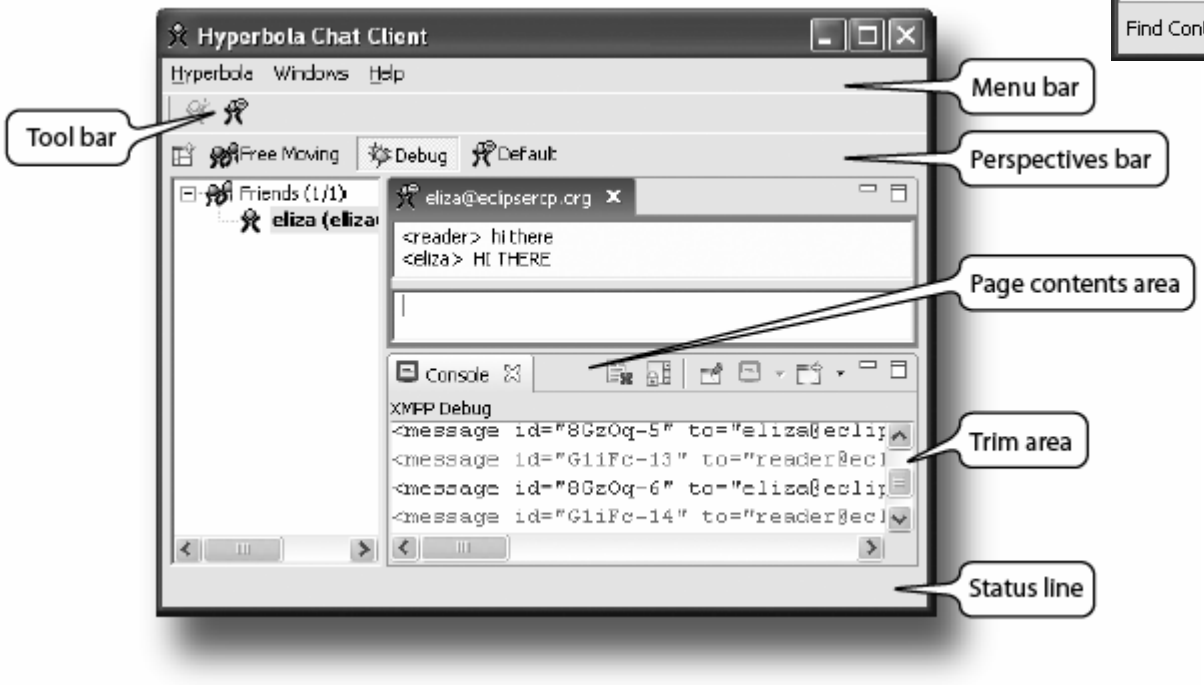
The difference with RCP applications is that you are in control of much more than you were as a simple plug-in into an existing product.

- Views can be created with properties such that they can't be moved, don't show the title bar, and can't be closed.
- The editor area is optional.
- Perspectives can be hidden from the user and you have full control how they are presented.

Perspectives, Actions, Views

You are in full control, but let's start with the basics first.

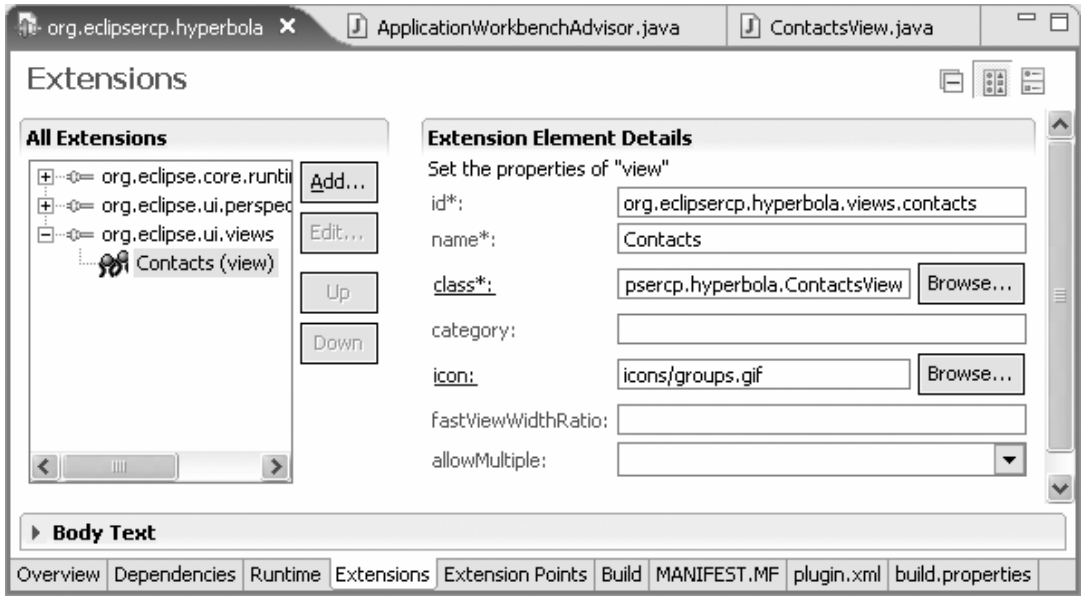




Adding the contacts view

- A view is added by contributing a view extension to the *org.eclipse.ui.views* extension point.
- Open the org.eclipsercp.hyperbola project's plugin.xml and go to the **Extensions** page. Click **Add…** and create an extension of type org.eclipse.ui.views.
- Right-click on the extension and add a view attribute using **New > view** from the context menu. When you click on the new view attribute, the details pane at the right shows the default values for the view.

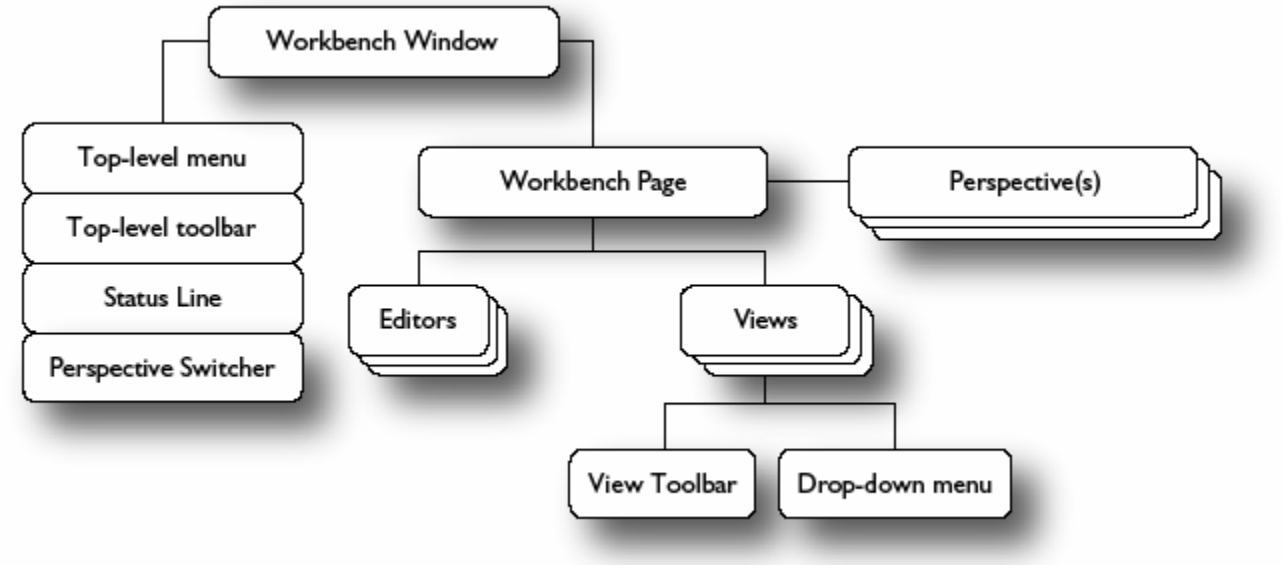
Adding the contacts view



You'll need a class to implement the view. Click on the **class** link to create a new class. When the new class wizard appears, most of the fields, including superclass ViewPart, are already filled in. All you have to do is type "ContactsView" for the class name. Click **Finish** and a skeleton view class is created and opened in an editor.

Adding the contacts view

All RCP applications must define at least one perspective; otherwise, there would be nothing to lay out the views. Think of a perspective as a set of layout hints for a window. Every IWorkbenchWindow has one page. The page owns its editor and view instances and uses the active perspective to decide its layout. The perspective details where, and whether or not, to show certain things, such as views, the editor area, and actions.



Adding to a perspective

org.eclipsercp.hyperbola/Perspective

```
public class Perspective implements IPerspectiveFactory {
  public void createInitialLayout(IPageLayout layout) {
    layout.setEditorAreaVisible(false);
    layout.addView(ContactsView.ID, IPageLayout.LEFT,
        1.0f, layout.getEditorArea());
  }
}
```

Making the view standalone

But you didn't want the close button.

org.eclipsercp.hyperbola/Perspective

```
public class Perspective implements IPerspectiveFactory {
  public void createInitialLayout(IPageLayout layout) {
    layout.setEditorAreaVisible(false);
    layout.addStandaloneView(ContactsView.ID, false,
        IPageLayout.LEFT, 1.0f, layout.getEditorArea());
  }
}
```

Perspective debugging hints

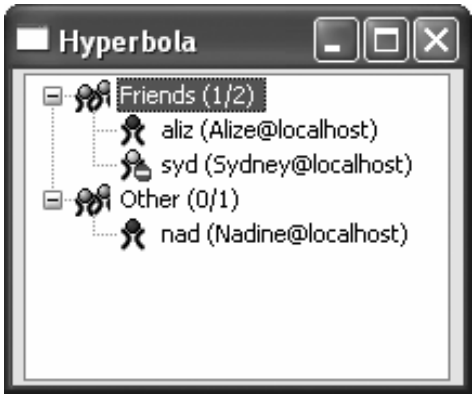
You may notice that changes you made in code to a perspective seem to be been ignored. Since we told the Workbench to save settings on shutdown, it saves the perspective layouts in the workspace location and on startup does not consult with the perspective factory at all. IPerspectiveFactory is only needed the first time a perspective is created.

To debug changes to a perspective factory, you must configure your launch configuration to clear the workspace area on each launch. Open the launch configuration dialog as shown earlier and check the option called **Clear workspace data before launching** and uncheck **Ask for confirmation before clearing**.

Exercise – Playing with perspectives

Start with Chapter 5 sample code, try and modify the code to do the following:

- Experiment with perspective layouts
  - Add the editor area
  - Add a moveable contact list
  - Add a fixed contacts list with a title but which is not closeable
- Advanced
  - Add two other empty views
  - Create different perspective layouts



Actions

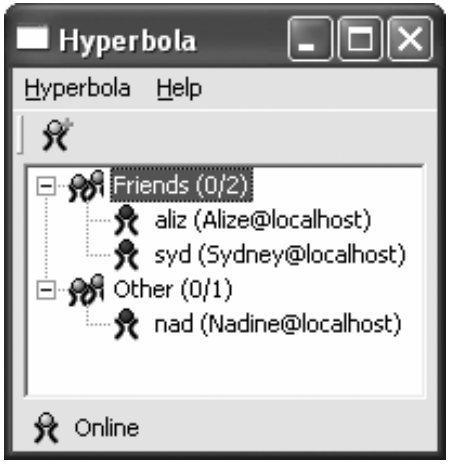
Actions are everywhere: in toolbars, top-level menus, context menus, status lines, trim area.

RCP applications have an important responsibility of designing the top-level action area structures.

org.eclipsercp.hyperbola/ApplicationWorkbenchWindowAdvisor

```
public void preWindowOpen() {
  IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
  configurer.setInitialSize(new Point(250, 350));
  configurer.setShowMenuBar(true);
  ...
}
```

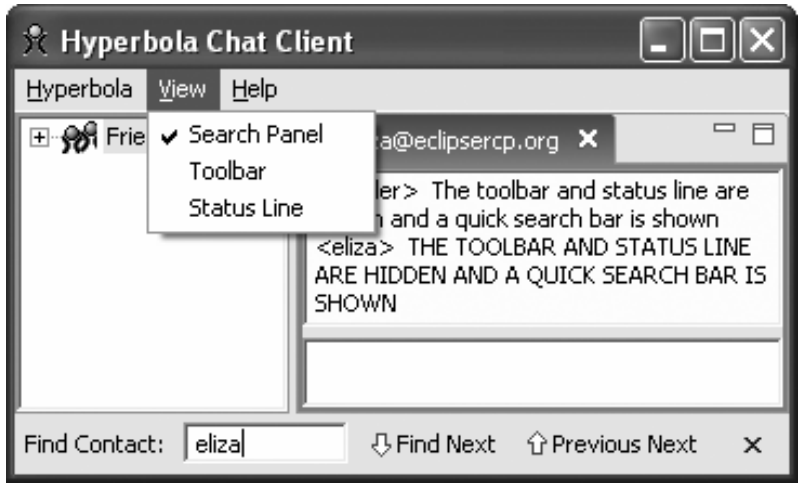
Advisor controls menu, toolbar, status line visibility.



Actions – Hide and seek

Although the advisor controls the initial visibility of the areas, it can't toggle them afterwards. If this is something you need, it is possible.

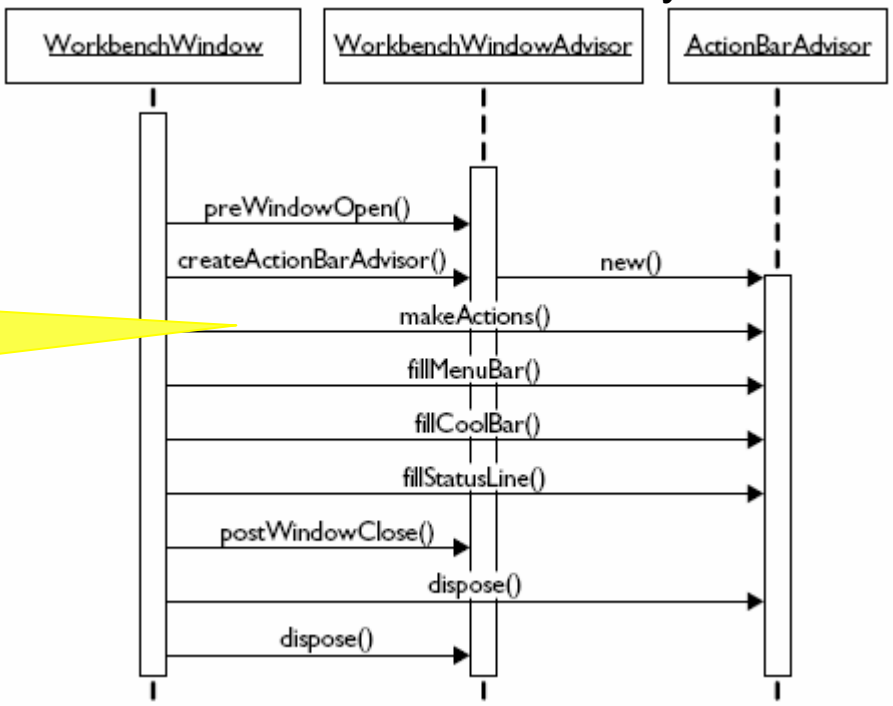
See ApplicationWorkbenchWindowAdvisor.createWindowContents()



Actions - Lifecycle

- ActionBarAdvisor is a dedicated advisor for creating and managing top level actions.
- The ActionBarAdvisor is separate from the WorkbenchWindowAdvisor since an application often has hundreds of actions — this more clearly separates the concerns.

Advisor.makeActions() is called *before* the WorkbenchWindow's controls are created.
Widgets can't be accessed.



Actions – Top-level menus

```
org.eclipsercp.hyperbola/ApplicationActionBarAdvisor
public class ApplicationActionBarAdvisor extends ActionBarAdvisor {
  private IWorkbenchAction exitAction;
  private IWorkbenchAction aboutAction;
  protected void makeActions(IWorkbenchWindow window) {
    exitAction = ActionFactory.QUIT.create(window);
    register(exitAction);
    aboutAction = ActionFactory.ABOUT.create(window);
    register(aboutAction);
  }
  protected void fillMenuBar(IMenuManager menuBar) {
    MenuManager hyperbolaMenu = new MenuManager(
      "&Hyperbola", "hyperbola");
    hyperbolaMenu.add(exitAction);
    MenuManager helpMenu = new MenuManager("&Help", "help");
    helpMenu.add(aboutAction);
    menuBar.add(hyperbolaMenu);
    menuBar.add(helpMenu);
  }
}
```

Create the actions once

Provide context to the actions

Register with Workbench for keybindings and lifecycle

Create the menu structure

Place the actions

Actions – Toolbars

Same idea as for menus, same action instances can be added to both the toolbar and the menu.

org.eclipsercp.hyperbola/ApplicationWorkbenchWindowAdvisor

```
public void preWindowOpen() {
  IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
  configurer.setShowCoolBar(true);
  ...
}
```

org.eclipsercp.hyperbola/ApplicationActionBarAdvisor

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

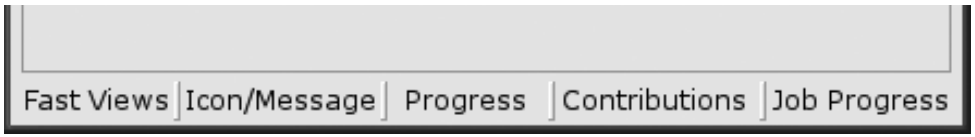
Actions – Status line is problematic

- Showing the status line is easy

org.eclipsercp.hyperbola/ApplicationWorkbenchWindowAdvisor

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

- Beware: Status line is a shared resource!
- Don't call IStatusLineManager.setMessage() in ActionBarAdvisor.fillStatusLine().



Actions – Status line

- You can make contributions IStatusLineManager.addContribution(). But they are placed in the middle of the status line.
- In 3.2, you can use trim contributions to allow more configurability of the status line.
- In Hyperbola since you are in full control, the simple solution is to use the message area.

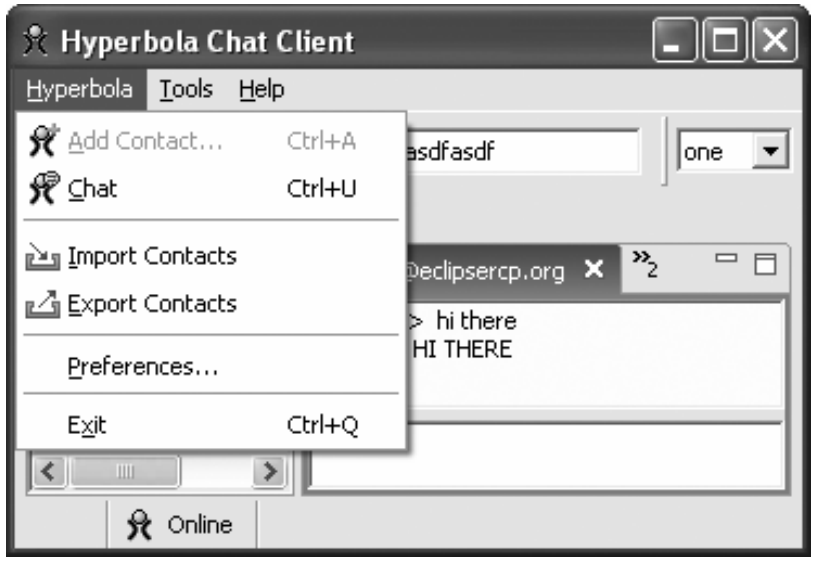
Actions – Standard actions

- Workbench defines a set of reusable actions. These actions are defined as inner classes of org.eclipse.ui.actions.ActionFactory and are instantiated and used like regular actions.

```
exitAction = ActionFactory.QUIT.create(window);
```

- Before implementing an action in your application, check to see if ActionFactory defines a related action such as:

  Save, Save all, Save As, Preferences, Properties, About, Close, Close All, Help, Import, Export



Actions – What about declarative actions?

- Workbench has many extension points for contributing actions.
- When should they be used instead of programatically adding them in the ActionBarAdvisor?
- In small RCP apps there is no need for declarative actions. Minimally the app needs to define the application structure and stable set of menus and toolbars.

Actions – Declarative action advantages

- They allow lazy loading of plug-ins by being shown in the UI without loading their associated plug-in. In large applications with many plugins, this is very important.
- Declarative actions can be associated with perspectives and easily allow dynamic reconfiguration of top-level menus and toolbars based on the active perspective.
- They allow users to configure top-level menus and toolbars via the perspective customization dialog (refer to the ActionFactory.EDIT_ACTION_SETS class for more details).
- Declarative action contributions can easily be filtered out of the application using *capabilities*.

Actions - Allowing declarative actions

- Even if you choose not to use declarative actions for your part of the application, you should design your application so that other plug-ins can extend its menus and toolbars.
- Add placeholders to top-level menus and toolbars.
- Placeholders are named entities that can be referenced from extension points.
- Hyperbola already has a named menu with the id “hyperbola”

org.eclipsercp.hyperbola/ApplicationActionBarAdvisor
MenuManager hyperbolaMenu = new MenuManager(“&Hyperbola”, “hyperbola”);

Actions - Allowing declarative actions

- The Workbench supplies a standard placeholder id that is used to mark the location in a contribution manager where contributions are added. The constant IWorkbenchActionConstants.MB_ADDITIONS (defined as “additions”).

org.eclipsercp.hyperbola/ApplicationActionBarAdvisor

```
protected void fillMenuBar(IMenuManager menuBar) {
// Top-level menu called Hyperbola with id ‘hyperbola’.
MenuManager hyperbolaMenu = new MenuManager(“&Hyperbola”,
  “hyperbola”);
hyperbolaMenu.add(chatAction);
// Placeholder within the ‘hyperbola’ menu called ‘additions’. This
// can be referenced as ‘hyperbola/additions’.
hyperbolaMenu.add(new
  Separator(IWorkbenchActionConstants.MB_ADDITIONS));
hyperbolaMenu.add(new Separator());
hyperbolaMenu.add(exitAction);
}
```

Menu path is “hyperbola/additions”

Actions - Allowing declarative actions

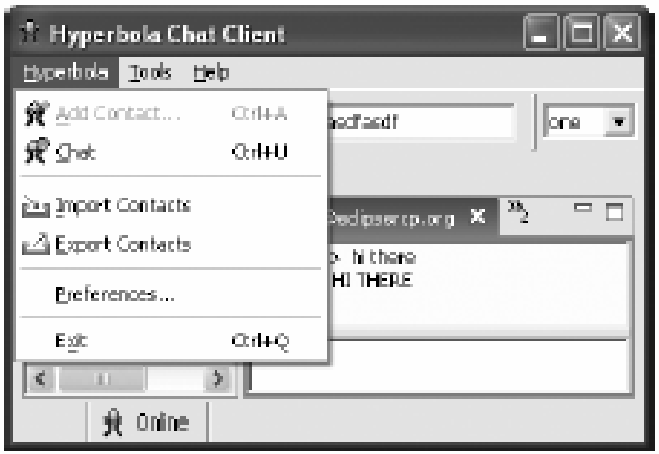
- Placeholders are added as either **Separators** or **GroupMarkers**. **Separators** surround all contributions by the appropriate separators, whereas contributions to a **GroupMarker** are added as-is, without any additional separators.
- To support action contributions to menus anywhere in your application, not just at the top level of menus, you must document the menu and group ids defined by your application.
- The Workbench provides a list of commonly used menu ids in IWorkbenchActionConstants.The IDE product uses these, but you are free to use your own identifiers instead.
- These ids effectively become API and so should be documented and maintained.

Actions - Declaring the actions

- Now that the placeholders are in place, let's declaratively add some action sets to Hyperbola.
- **org.eclipse.ui.actionSets**—This extension describes a set of menus and actions that is added to the top-level menu and toolbar. Action sets are enabled and disabled as a group.

org.eclipsercp.hyperbola/plugin.xml

```
<extension point="org.eclipse.ui.actionSets">
  <actionSet
    id="org.eclipsercp.hyperbola.actionSet1"
    label="Tools"
    visible="true">
  <action
    class="org.eclipsercp.hyperbola.actions.ExportContactsAction "
    icon="icons/export.gif"
    id="org.eclipsercp.hyperbola.exportContacts"
    label="&amp;Export Contacts"
    menubarPath="hyperbola/additions"
    style="push"/>
  </actionSet>
</extention>
```



Exercise: Perspectives, View, and Actions

Start with the code from Chapter 5 and add the following:

- Basics
  - Add the menu bar and toolbar. Add actions to both. Basic actions to exit, start a new chat, open another view.
  - Add status line contributions.
  - Add a task tray and actions to the task tray.
  - Compare with Chapter 6 code for hints
- Advanced – Start with Chapter 14 code.
  - Add declarative actions to Chapter 14 and compare with Chapter 17.
  - Add another perspective and add actions to switch between perspectives.
  - Add a action to open another window.

20-30 minutes

Branding

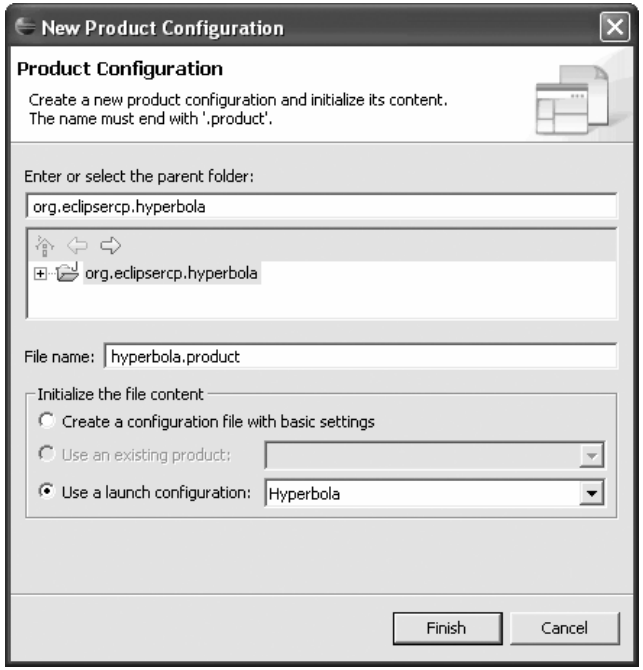
30 minutes

What is Branding?

- One of the big key differences when developing for RCP
- Visual cues that differentiate your product from others
  - Title bar label
  - Window icons
  - Desktop icon
  - Splash screen
  - Program (launcher) name
  - About dialog information
- A *product* wrapper on an *application*

Defining a Product

- Products are composed of the following files
  - Product configuration (.product) file
  - Product extension (in plugin.xml)
  - config.ini
  - launcher
- Define a Product Configuration file
  - **File > New > Other… > Plug-in Development > Product Configuration**
  - Use a launch configuration

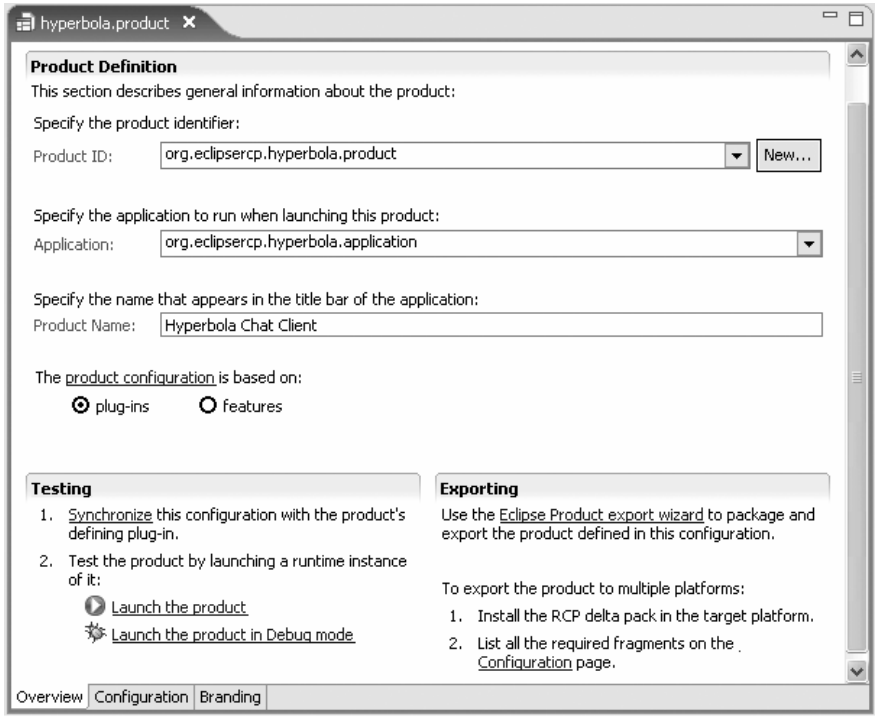


Filling out the Product

- Define a product extension

  org.eclipsercp.hyperbola.product
- Identify application
- Set product name

- Products list plug-ins or features
  - Listing plug-ins is easy
  - Listing features scales better

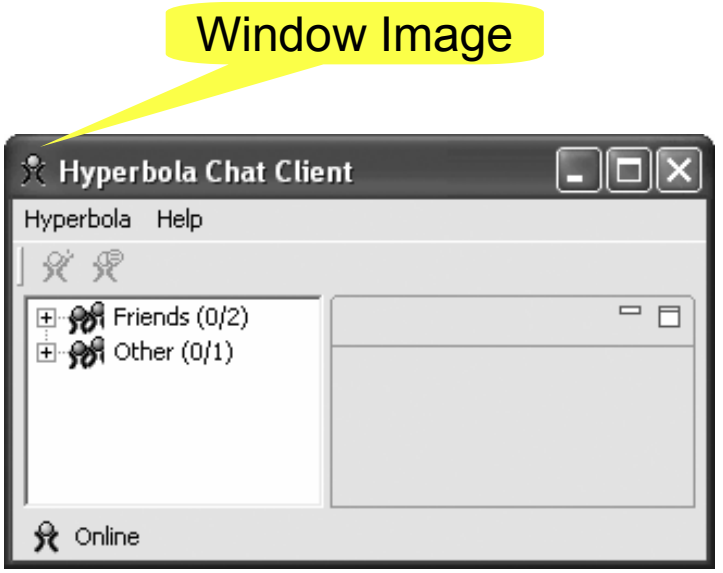


- Plug-in list automatically populated from launch configuration
- Product id stored in config.ini at build-time

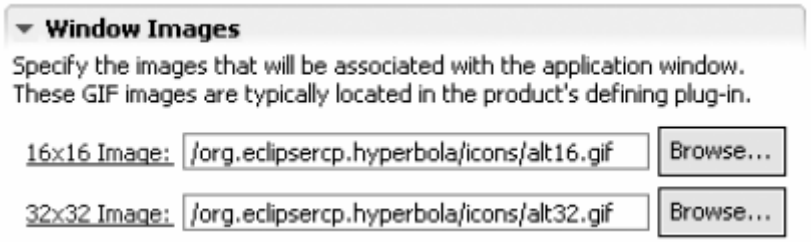
  eclipse.product=org.eclipsercp.hyperbola.product

Branding – Window Images

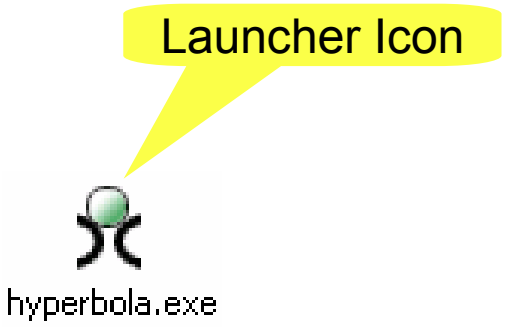
- Window images appear
  - in the top-left corner of windows
  - Task bar on Windows
  - …



- Actually can be a list of images
- Add images on the branding page of the product editor
- Image locations stored in product extension in plugin.xml

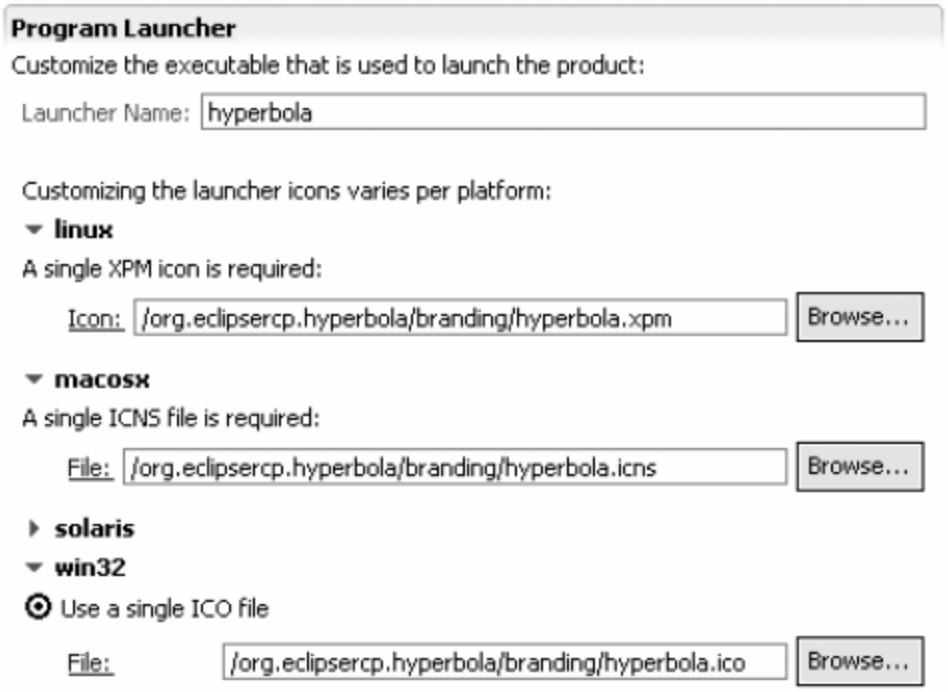


Branding – Launcher



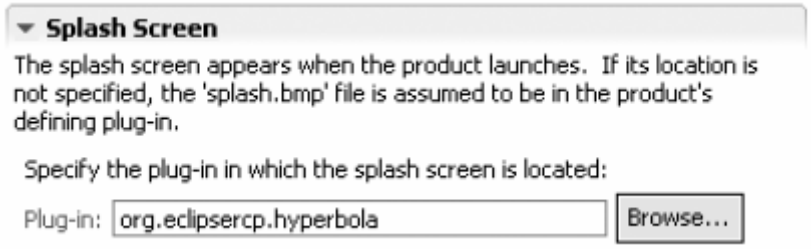
- From your user's point of view the launcher is "your application"
  - It's the first thing your users see

- Brand launcher with
  - name
  - icon(s)
- Platform-dependent images
- Supply all resolutions on Windows
- Images are used at build-time



Branding – Splash Screen

- Hopefully users don't see this for long ☺
  - Applications can be big
  - Running over networks,
  - …
- Nonetheless important for
  - Usability
  - Product image
  - Legal issues



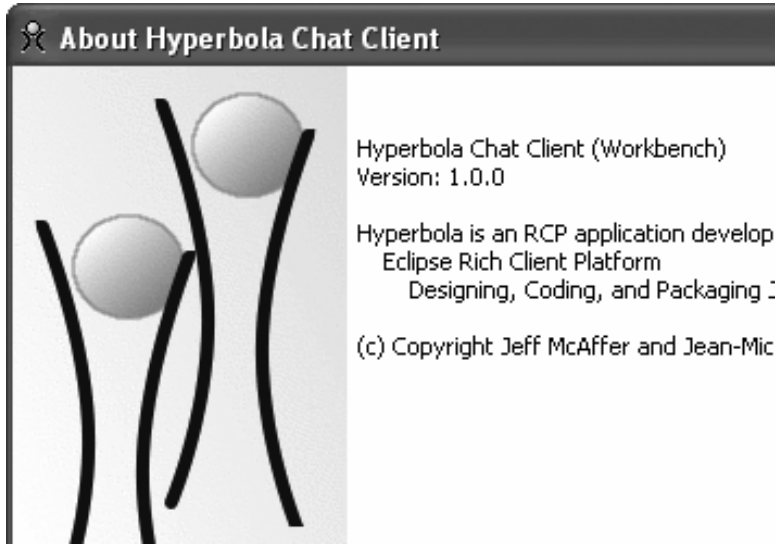
- Splash supplied in a plug-in identified in the config.ini

  osgi.splashPath=platform:/base/plugins/org.eclipsercp.hyperbola
- Fragments can be used to deliver locale-specific splashes

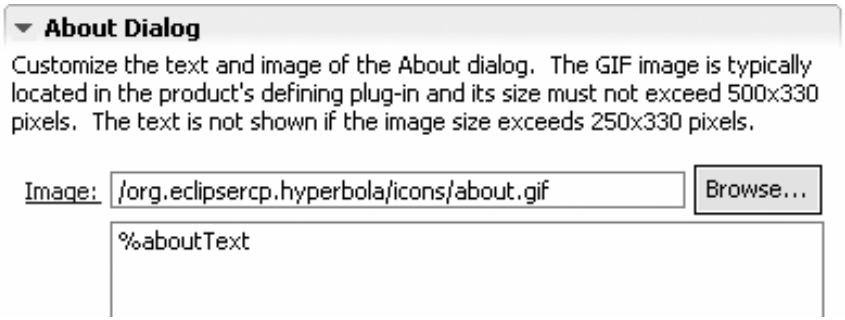
Branding – About information

- About dialog shows attribution, version, …
- Gateway to detailed info about the app

- About Image and text stored in product extension in plugin.xml
- Localized text stored in properties files/fragments



org.eclipsercp.hyperbola/plugin.properties
aboutText=\n\nHyperbola Chat Client (Workbench)\n\
Version: 1.0.0\n\n\
Hyperbola is an RCP application developed for the book \n\
\tEclipse Rich Client Platform \n\
\t\tDesigning, Coding, and Packaging Java Applications \n\
\n\
(c) Copyright Jeff McAffer and Jean-Michel Lemieux 2005. \
All Rights Reserved.\n



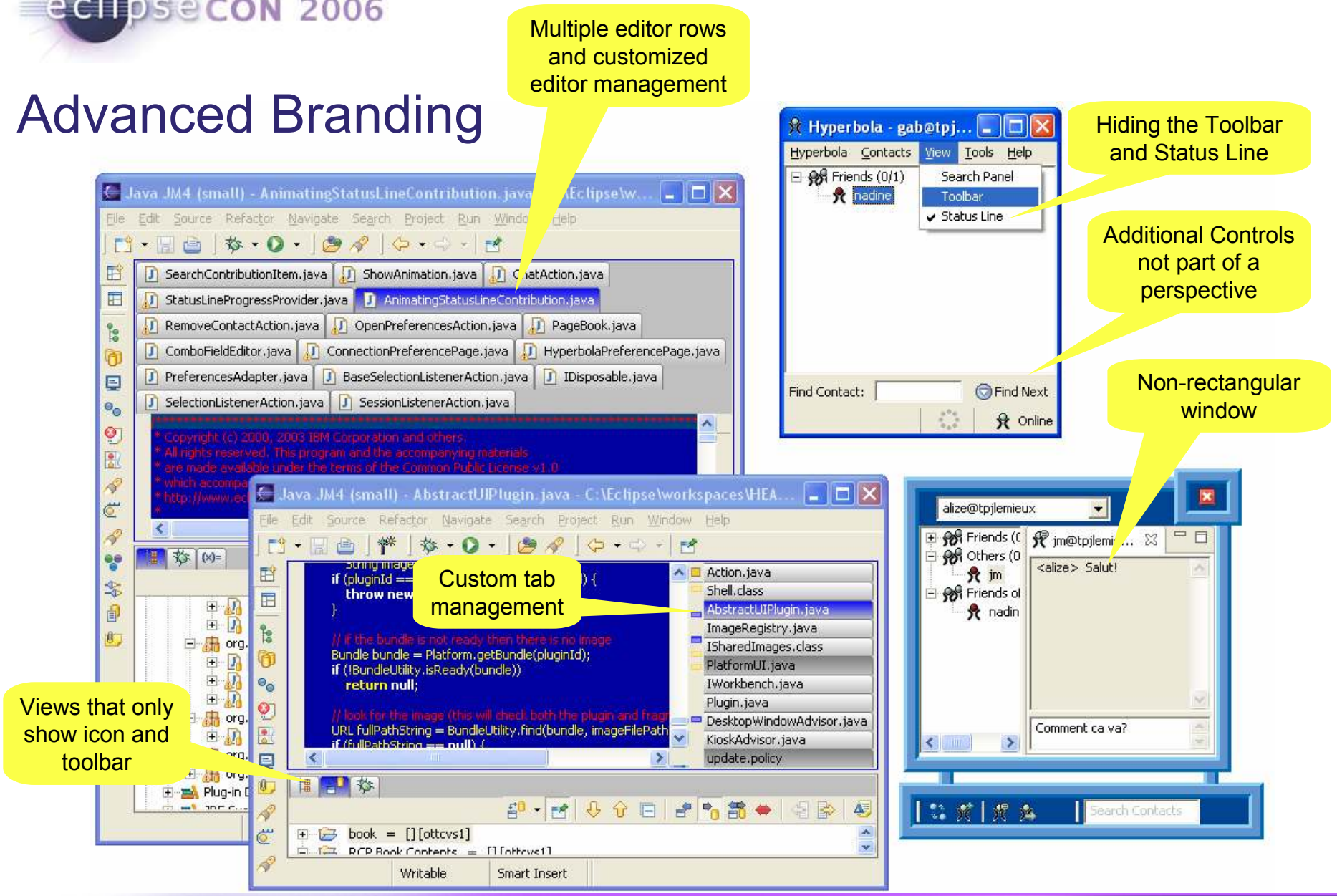
Advanced Branding

"I showed an RCP application to my boss the other day and he said that it was nice but it still looked like Eclipse."
– over heard in a meeting

If you don't like it – change it!

- Presentations API
  - Controls the look and feel of views and editors
- WorkbenchWindowAdvisor customizations (RCP applications only)
  - Override createWindowContents()

Advanced Branding



Advanced Branding

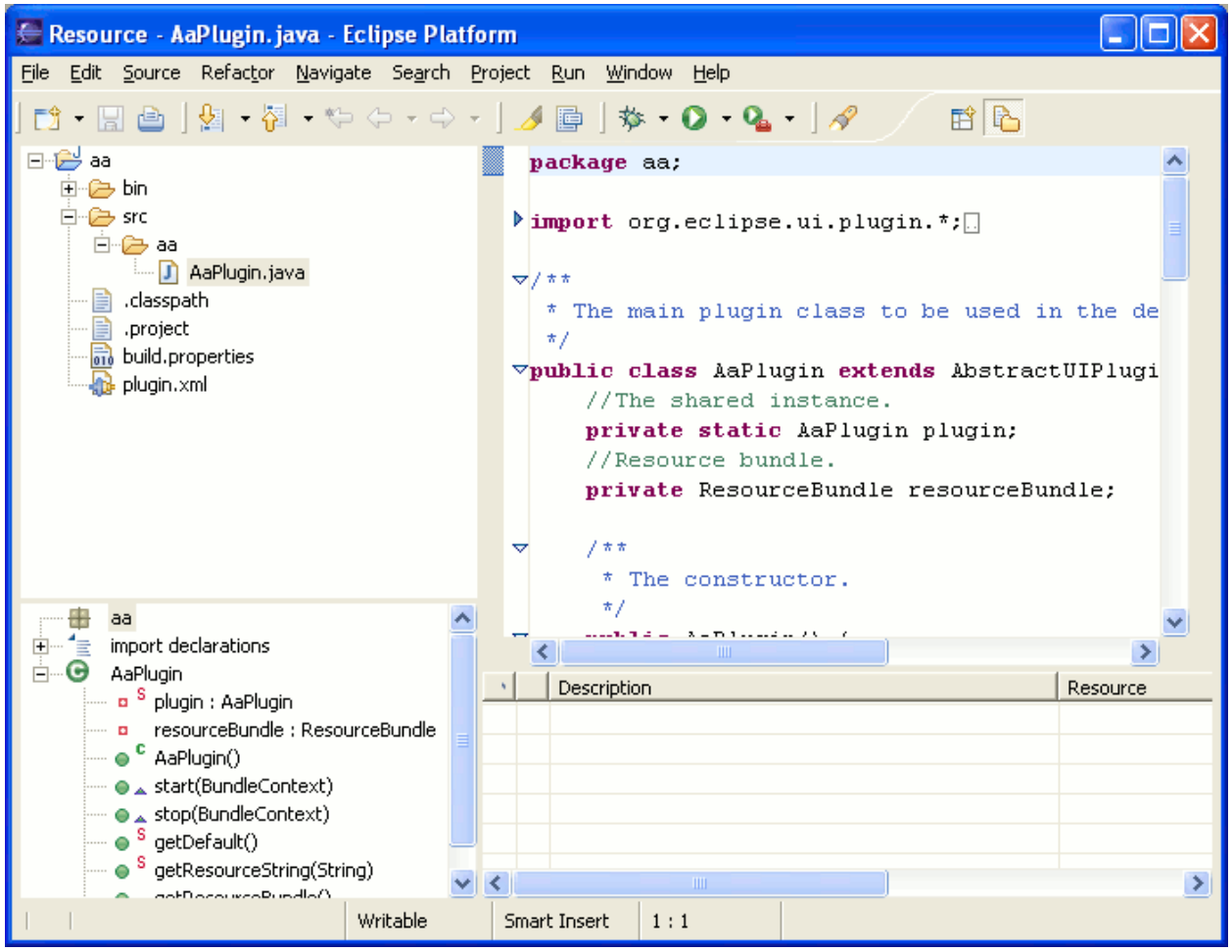
- *Presentation* = A look and feel extension for Workbench parts.
- *Part* = a view or editor.
- Presentations manage the trim and layout of areas that contain one or more parts – called *part stacks*.

- Widget factories, not skinning
  - Don’t just paint widgets. Supply the widgets themselves.
  - Don’t lose that native feelin’
- Benefits
  - Maximum flexibility
  - No custom widgets needed

Advanced Branding

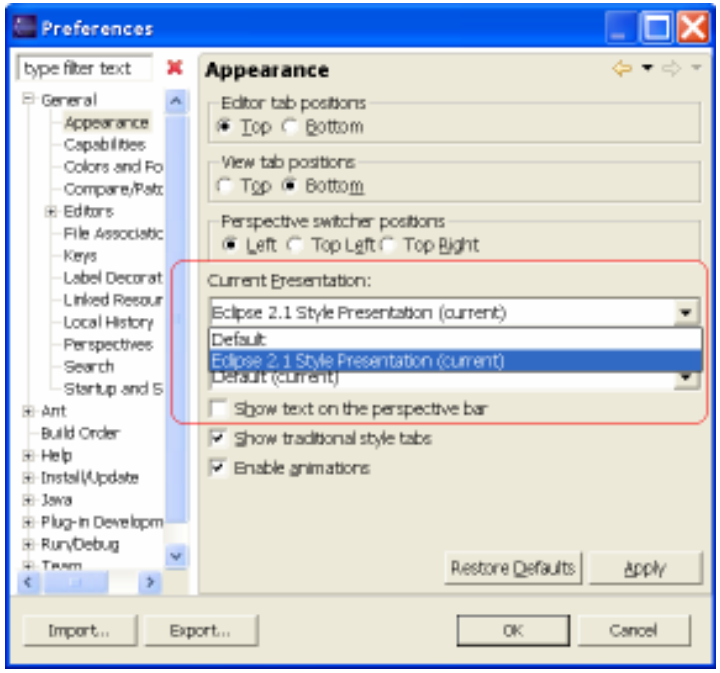
- Create the trim for part stacks
  - Tabs
  - Title
  - System menu
  - Close/Minimize/Maximize buttons
  - Borders
- Control layout and visibility
  - Parts
  - Toolbars
  - View menus
  - Drag/drop regions

Advanced Branding – Removing the presentation

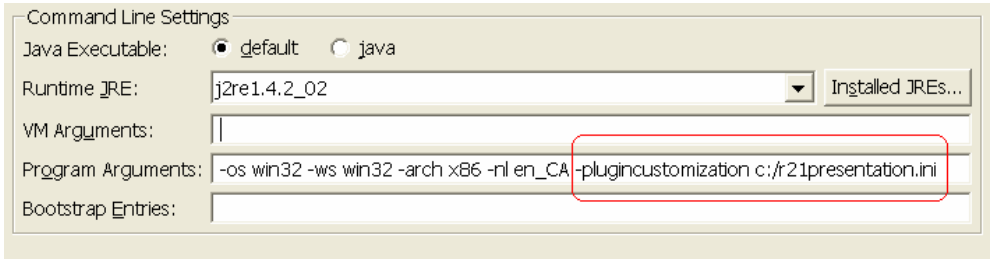


Advanced Branding

From the IDE preference page



Using a preference



Preference customization file

```
# R21 presentation settings: copy these values to your product's
# plugin_customization.ini file before starting eclipse

# use the R2.1 style
org.eclipse.ui/presentationFactoryId=org.eclipse.ui.internal.
     r21presentationFactory
```

You need to restart the Workbench – kind of ☺

- When you change the presentation in the preference dialog, you will be asked to restart the Workbench. Instead, open a new window. The new presentation will be used for all new windows.

Advanced Branding

If you want to change how the Workbench creates these controls you have two choices:

- Change the visibility of some of the controls by using the **IWorkbenchWindowConfigurer.setShow*()** methods. These control the initial visibility only and cannot hide/show the controls once the window is opened.

- Change the visibility and layout of the entire workbench window by overriding **WorkbenchAdvisor.createWindowContents()**. This moves responsibility for creating all of the window's controls to the advisor.

Advanced Branding

The IWorkbenchWindowConfigurer provides methods to create the common workbench controls and can be called from within the createWindowContents() method. The controls returned from the control creation methods are un-typed and are provided uniquely for placement within the layout defined by the advisor. They are not meant to be configured or downcasted.

```
public void createWindowContents(IWorkbenchWindowConfigurer
     configurer, final Shell shell) {
  Menu menu = configurer.createMenuBar();
  shell.setMenuBar(menu);
  FormLayout layout = new FormLayout();
  layout.marginWidth = 0;
  layout.marginHeight = 0;
  shell.setLayout(layout);
  toolbar = configurer.createCoolBarControl(shell);
  page = configurer. createPageComposite(shell);
  statusline = configurer.createStatusLine(shell);
```

Using a FormLayout to allow toggling the visibility of controls in the workbench window.

Advanced Branding

Another common technique for making an application stand out is to forfeit the platform's standard window look and provide your own custom shaped window with curves, see through areas, and a funky design.

- Hire a graphic designer.
- Use image mask to create SWT Regions to define Shell shape.
- WorkbenchWindowAdvisor.preWindowCreate() ensures Shell style set to SWT.NO_TRIM before the Shell is created.
- Once the window is created in postWindowCreate(), add a paint listener to the Shell to draw the border for the shaped window and set the region of the Shell.
- In createWindowContents() create the controls for the window.

```
public void preWindowOpen() {
  getWindowConfigurer().setShellStyle(
    SWT.NO_TRIM | SWT.ON_TOP | SWT.NO_BACKGROUND);
}

shell.setRegion(currentRegion);
```

Branding

There are many options for branding, but nothing comes for free. Here's a quick summary of your options:

- Splash, Icons – easy
- Help, About – easy
- Window Contents – moderate
- Custom Presentation – hard

Packaging

20 minutes

What is *packaging*?

- Transformation
  Development form ⇒ Deployment form
- For example
  Workspace projects ⇒ Built JARs

- Produces
  - Stand-alone, runnable application in directory or archive
  - Update site population
  - JNLP/WebStart deployment

- Installers (e.g., InstallShield) not directly supported

Mapping Development onto Deployment

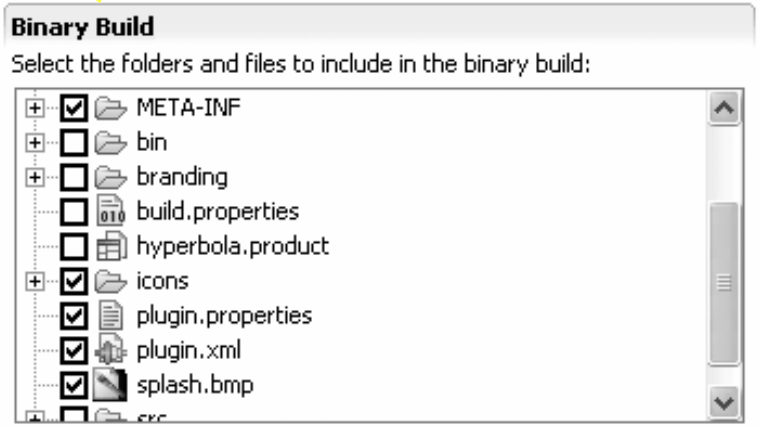
- Main task is identifying development time artifacts to include/exclude from deployed form for each plug-in/feature
- Define this information in the **build.properties** file

Source for '.'

Development resources to include in deployment

org.ec psercp.hyperbol uild.properties

```
source.. = src/
bin.includes =
    plugin.xml, META-INF/,\
    plugin.properties,\
    .,\
    splash.bmp,\
    icons/
```

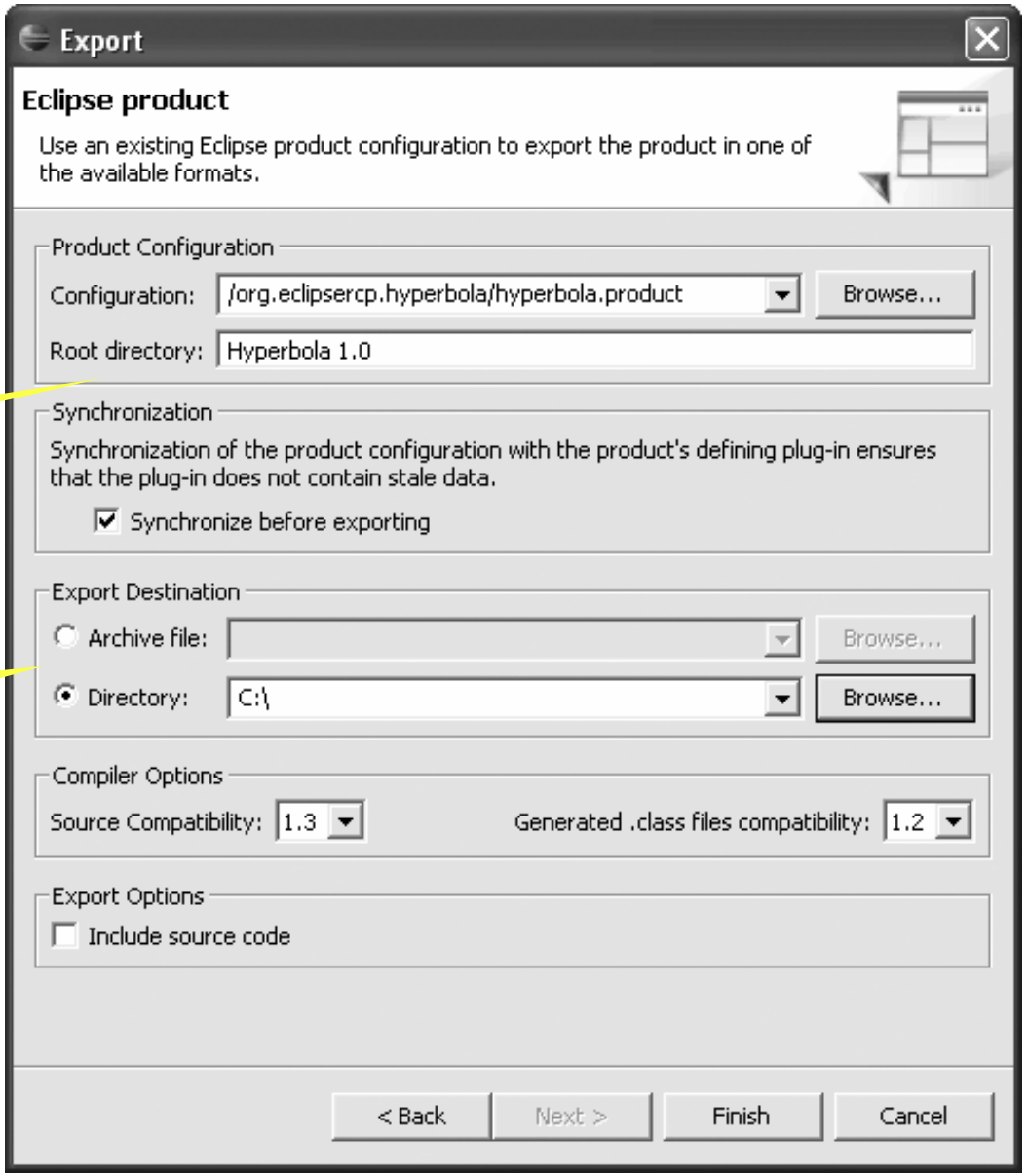


Exporting the Product

- **Product Export wizard** link on Overview page

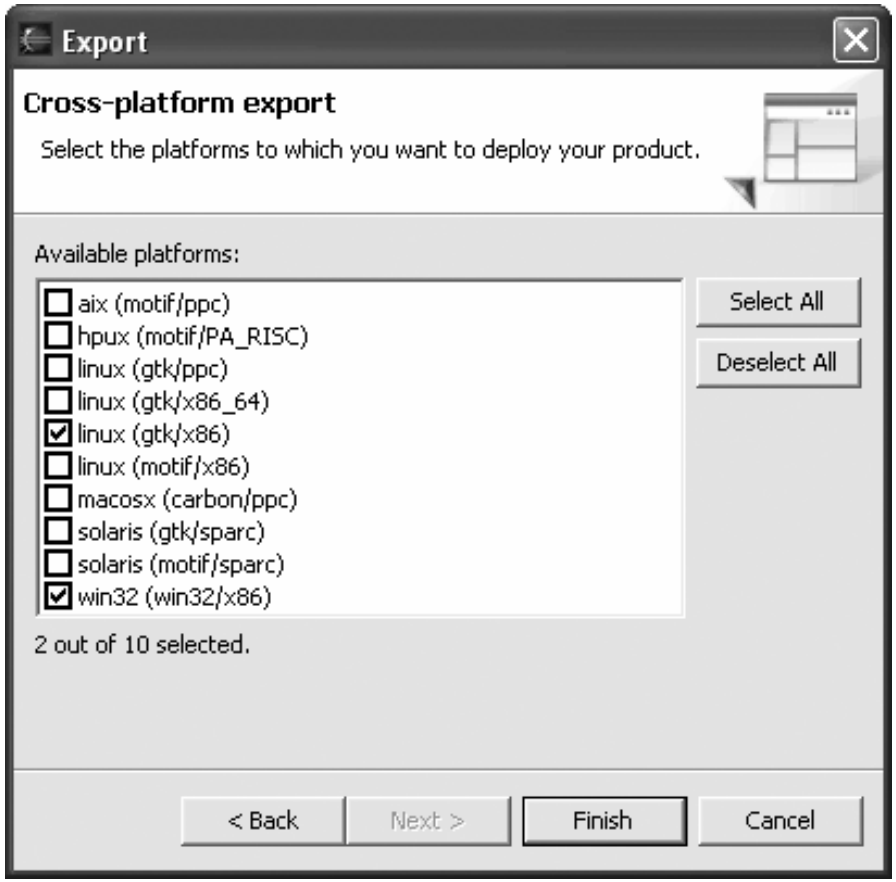
Root directory to put in output archive or directory

Output destination and form



Cross-platform Exporting

- Typical target has the current platform's fragments
  - org.eclipse.swt.**win32.win32.x86**
- Delta pack includes fragments for all platforms
  eclipse-RCP-3.1-delta-pack.zip
- Export for many platforms at once



Cross-platform exporting steps

- Unzip delta pack over target directory, **Reload** the target
- Add all needed platform-specific fragments to the product configuration plug-in list
- Use **Export product** wizard and check **Export for multiple platforms** and then click **Next** for **Cross-platform export**
- Choose platforms and export
- Output appears in appropriately named archives or directories

Exercise: Branding and Packaging Hyperbola

- Start with Chapter 7 sample, work towards Chapter 9
- Brand Hyperbola
  - Get branding images etc by comparing with Chapter 8 sample
- Package Hyperbola
  - Export and run
  - Confirm branding is correct
- Bonus: Export Hyperbola for several different platforms

20 minutes

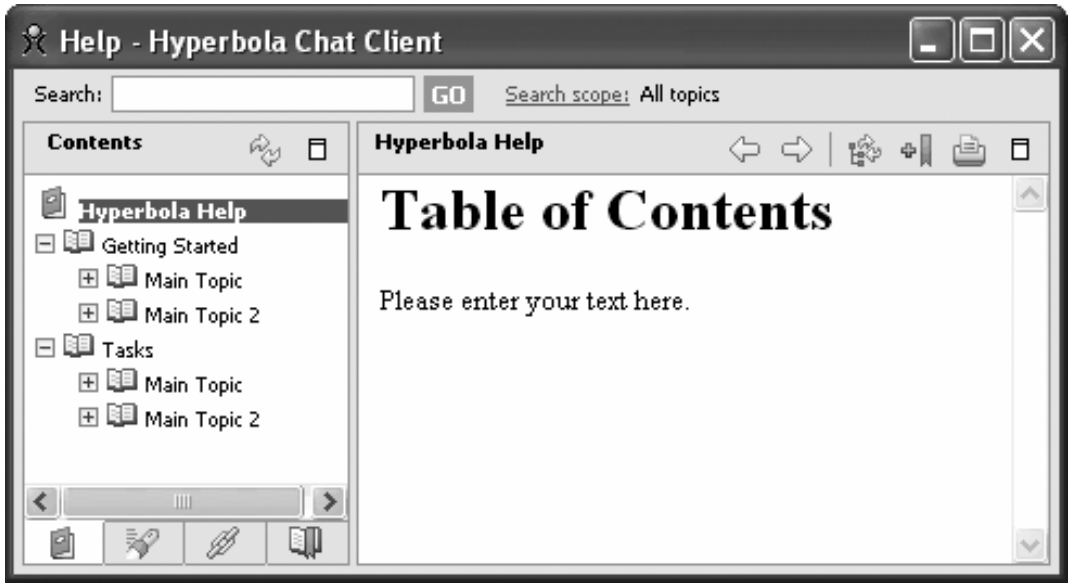
Adding Function: Help

20 minutes

What is the Help system?

- Browser driven help system
- Fully Integrated (e.g., supports F1 help)
- Context-sensitive, dynamic help
- Extensible weaving of content into books

- Other “user assistance” mechanisms such as Cheat Sheets, Welcome Pages, Intro, … also available



Getting Help

- Help comes as part of the IDE Platform and SDK
- Copy the following Help plug-ins from
  <ide>/eclipse/plug-ins into <target>/eclipse/plugins
  1. org.apache.lucene
  2. org.eclipse.help.appserver
  3. org.eclipse.help.base
  4. org.eclipse.help.ui
  5. org.eclipse.help.webapp
  6. org.eclipse.tomcat
  7. org.eclipse.ui.forms (prerequisite)
- Reload the Target Platform (**PDE preferences > Target Platform**)

Adding Help to Hyperbola

- Add the new plug-ins to the Hyperbola product configuration
- Create the action

org.eclipsercp.hyperbola/ApplicationActionBarAdvisor

```
makeActions() {
    …
    helpAction = ActionFactory.HELP_CONTENTS.create(window);
    register(helpAction);
```

- Place the action

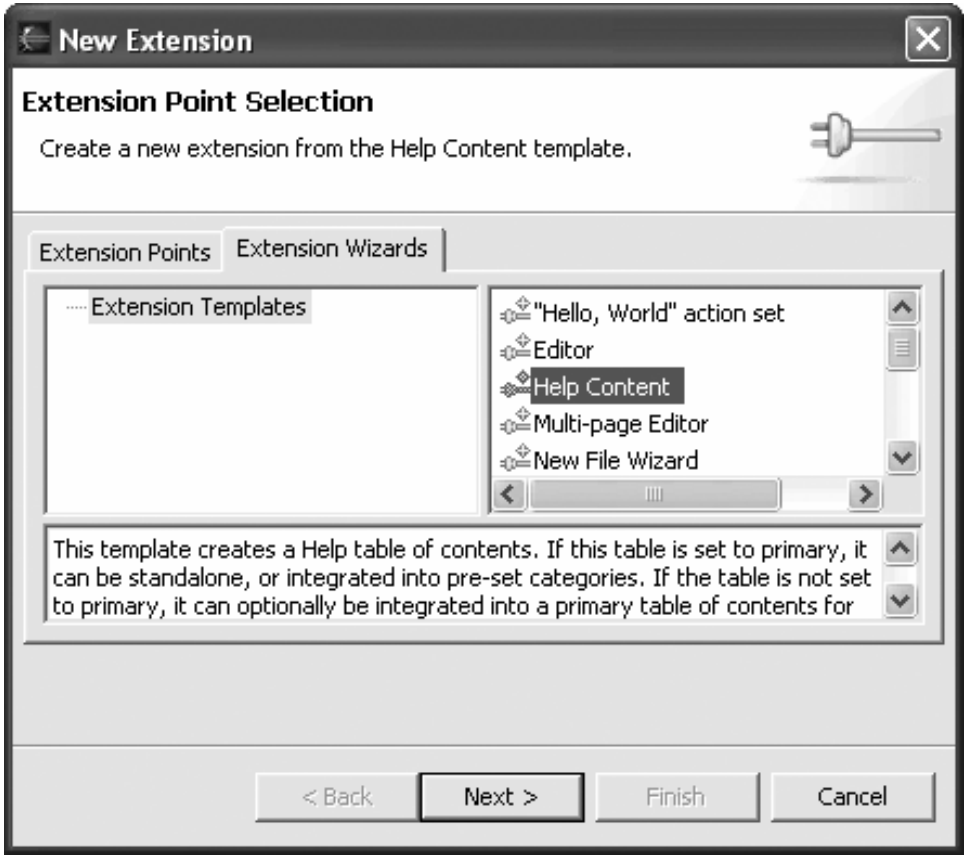
org.eclipsercp.hyperbola/ApplicationActionBarAdvisor

```
makeActions() {
    …
    MenuManager helpMenu = new MenuManager(“&Help”,”help”);
    helpMenu.add(helpAction);
    helpMenu.add(aboutAction);
```


Adding Help Content

- Contributed via extensions
- Use the **Help Content** extension wizard!
- Set label and choose categories
- Change content as desired
- Run Hyperbola and select
  Help > Help Contents

- Don't forget to add generated content to build.properties



Exercise: Add Help to Hyperbola

- Start with Chapter 12 and work towards Chapter 13
- Follow steps outlined here
- Compare to see examples of dynamic help

10 minutes

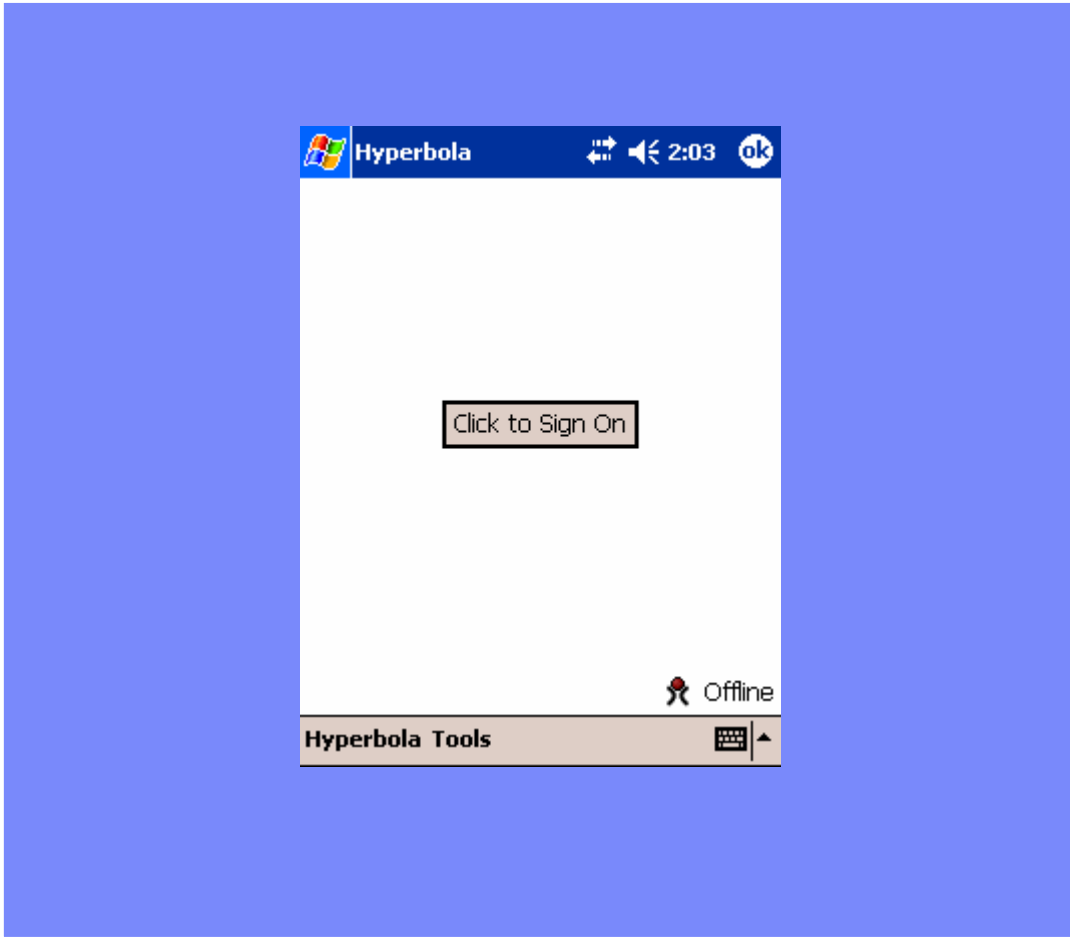
RCP Everywhere

30 minutes

RCP Everywhere

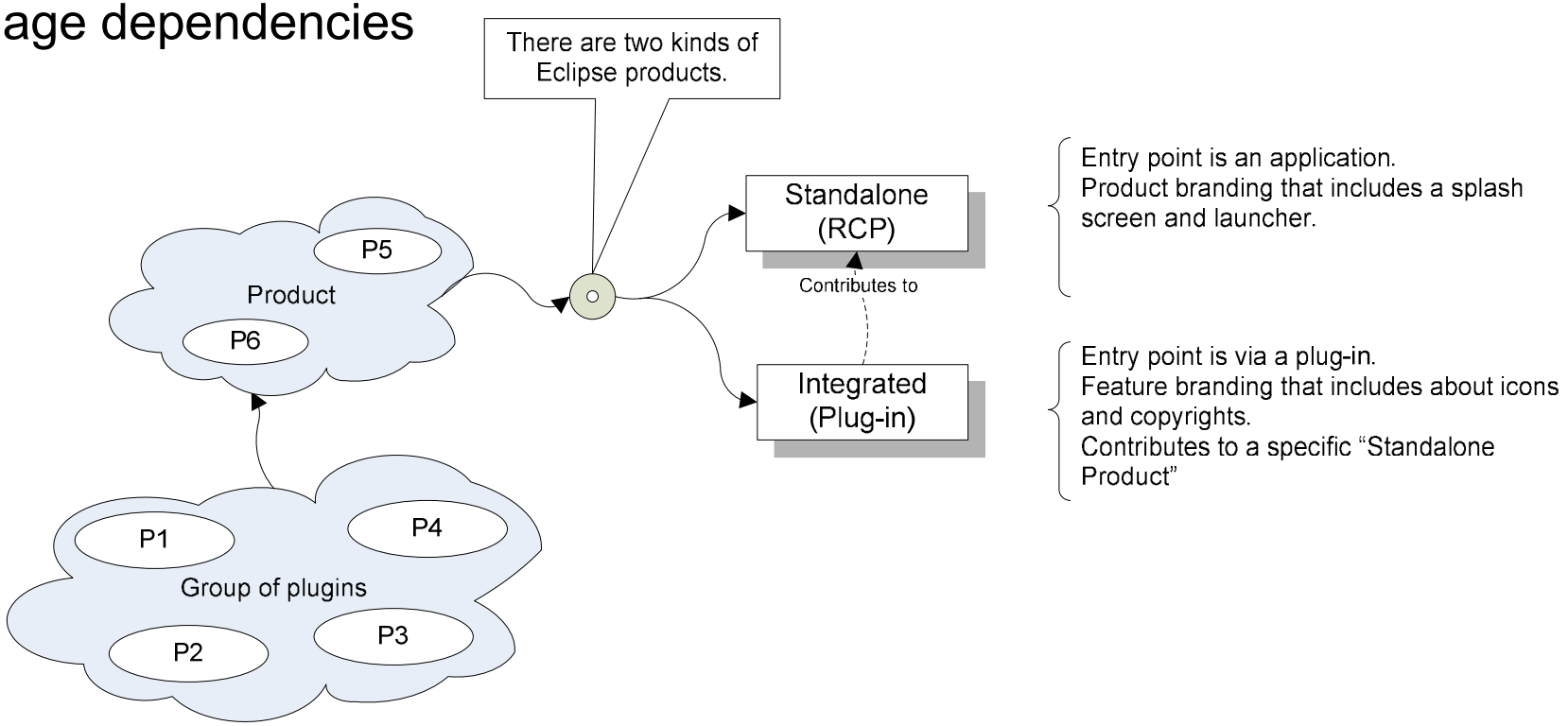
- Imagine Hyperbola was to be used in a hospital
  - Doctor to doctor from PDAs while making rounds
  - Patient to Patient/Staff using kiosks in patient rooms, ER, waiting rooms, …
  - Administration, management, researchers and lab technicians using standalone desktop application
  - IT department developer's integrated with their Eclipse IDE

RCP Everywhere: Hyperbola on a PocketPC



RCP Everywhere

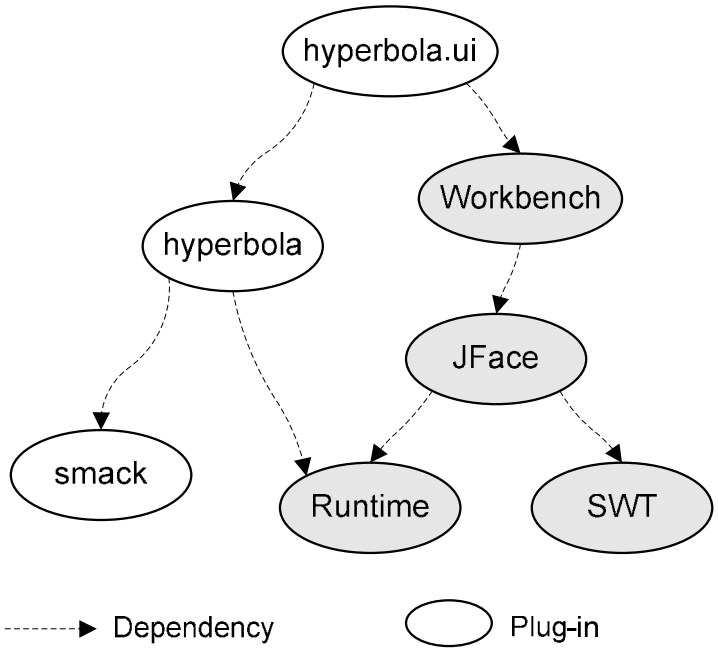
- Componentize ruthlessly
- Simplify structure
- Manage dependencies



RCP Everywhere

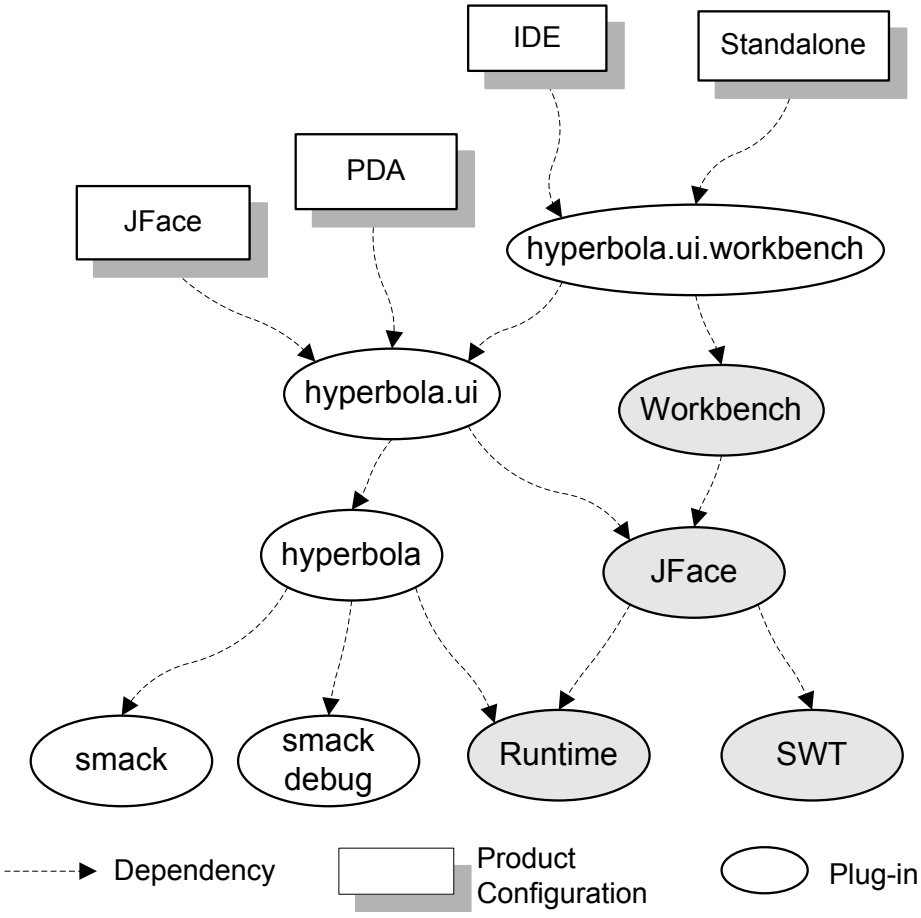
Before:

- Obvious RCP implementation
- Traditional Core/UI split



After:

- Several Product Configurations
- Refactored plug-in structure



Why so many features and products?

- The scenarios require different product configurations
- Product plug-in contributes and positions common function
- Product feature captures the plug-ins required for a configuration
  - Package configuration by exporting the feature
  - Package by exporting product definitions (.product files)
- Features facilitate use of Eclipse Update Manager
- Features enable the use of PDE for “releng” (automated) builds

Rule 1: Have a top-level feature and plug-in for every product configuration

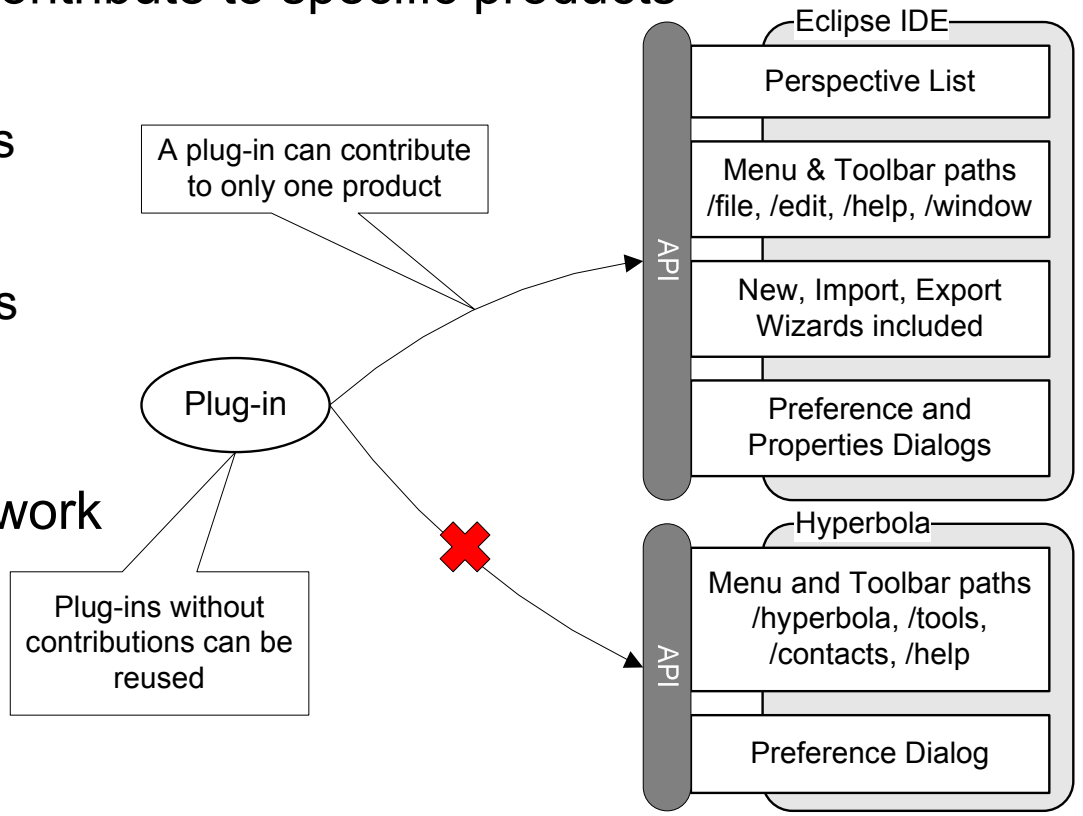
Minimize Dependencies

- Monolithic plug-ins don't scale
  - Worse, they can't be re-used in different applications
  - Develop your application as a set of loosely coupled plug-ins
- Eclipse SDK anti-patterns
  - Many plug-ins could be refactored – would break APIs.
  - Other plug-ins have hard dependencies on IDE or Resources plug-ins
- Minimize and layer dependencies
  - Workbench contributions
  - Actions
  - Views, editors, wizards, preferences, …
  - Data model

Rule 2: Minimize and layer plug-in dependencies

Isolate Workbench Contributions

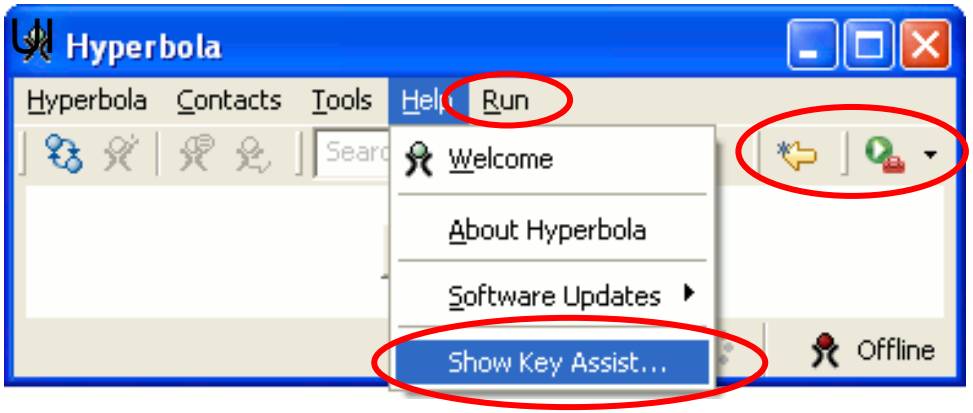
- Product configurations contribute to specific products
  - Many implicit APIs
  - Menu and toolbar paths
  - Predefined menus
  - Well known entry points
    - Show View
    - Open Perspective
- Increase reuse of framework plug-ins



Rule 3: Minimize Workbench contributions in framework plug-ins

What happens if you don't?

- Add full-featured text editing to Hyperbola
  - Use org.eclipse.ui.externaltools and org.eclipse.ui.editors – that's where all the funky text editing is!
  - Set up dependencies (this is usually a warning sign – drags in IDE)
  - Write code
  - When you run you see
    - `Invalid Menu Extension (Path is invalid): org.eclipse.ui.edit.text.gotoLastEditPosition`
  - Unexpected contributions in the UI



Views and Editors: UI Containers

- Build UI elements using only JFace primitives
  - Allows building applications without the Workbench
- Treat views and editors as containers
- Design components separate from container
- Use containers to plug together components as needed

- Example: Hyperbola ChatViewer
  - may show up in a view or an editor based on user preference
- RosterViewer and ChatViewer expose application's data model
- Decoupling pattern, such as Inversion of Control, used for composition

Rule 4: Decouple UI components from their containers

Optional Dependencies

- Layer dependencies **within** the same plug-in

```
<requires>
  <import plugin="org.eclipse.core.runtime"/>
  <import plugin="org.eclipse.emf.common" export="true"/>
  <import plugin="org.eclipse.core.resources" optional="true"/>
</requires>
```

- Example: core EMF plug-in
  - Resources dependent code gathered into the same package
  - Abstract data model accommodates both java.io.File and IResource
  - Build RCP and Eclipse IDE applications using the same core EMF plug-in
- Example: org.eclipse.ui.forms
  - 95% depends on SWT but Workbench needed for support multiple-editors
- Example: org.eclipse.help.base
  - Depends on Ant for JSP compilation
- Disadvantage: unused code can be shipped

Rule 5: Use optional dependencies for intra-plug-in layering

Building a platform

- The progression from a simple product to a platform involves pushing function down into framework plug-ins.
- The focus shifts from shipping one specific product to understanding what function is generic and how other products can be built around that function. That is the nature of the platform.
- Platform oath: APIs, Extension Points, Long term commitment

Identifying RCP friendly plug-ins

- We use the term friendly as a synonym of what we've also been calling framework plug-ins. These are plug-ins that are designed to work in any product.
- The simple answer is that if a plug-in manages its dependencies and uses optional dependencies appropriately, then it is a framework plug-in and is RCP friendly.
  - Open the plugin.xml for the plug-in you are examining and look at the set of plug-ins it requires. If it requires a product, for example the Eclipse IDE, then it is not a framework plug-in.
  - Next, look at its extensions for specific references to toolbar or menu paths, preference pages, or other contributions that place elements in the UI. In general, views and editors are acceptable since they do not appear in the UI unless they are explicitly placed.

RCP Everywhere - Conclusion

- Focus on your domain
- Ruthlessly componentize

- Eclipse RCP can produce high quality products in many configurations
- Eclipse RCP Everywhere rules are relatively simple
- Following the rules generates extreme flexibility
- Can be retrofitted but best if RCP Everywhere is an upfront goal

Eclipse 3.2

What's New in 3.2?

- Too much to list but here are some highlights…
- Improved Target management (e.g., Named Targets)
- Minimum Execution environment support (target plug-ins to J2ME etc)
- Many refinements in PDE tooling
- Refactored runtime plug-ins
- Integrated progress on startup
- Lots of new widgets and features in SWT
- Loads of User Assistance (e.g., Help, Cheat Sheets, …) improvements

Equinox

- Stand-alone implementation of OSGi framework
- Basis for all Eclipse systems
- Server-side Incubator
  - http://www.eclipse.org/equinox/incubator/server
- Extension registry outside Eclipse
  - http://www.eclipse.org/equinox/bundles/
- Many useful bundles
  - HTTP Service
  - Declarative services
  - Event Admin
  - Preferences

Wrap-up

Eclipse Rich Client Platform – The Book

- The book dedicated to RCP
- Tutorial and deep dive suitable for newbies and oldies

- http://eclipsercp.org
- ISBN 0-321-33461-2
- Available from Amazon.com etc. and in the EclipseCon bookstore ☺