

Eclipse RCP Everywhere

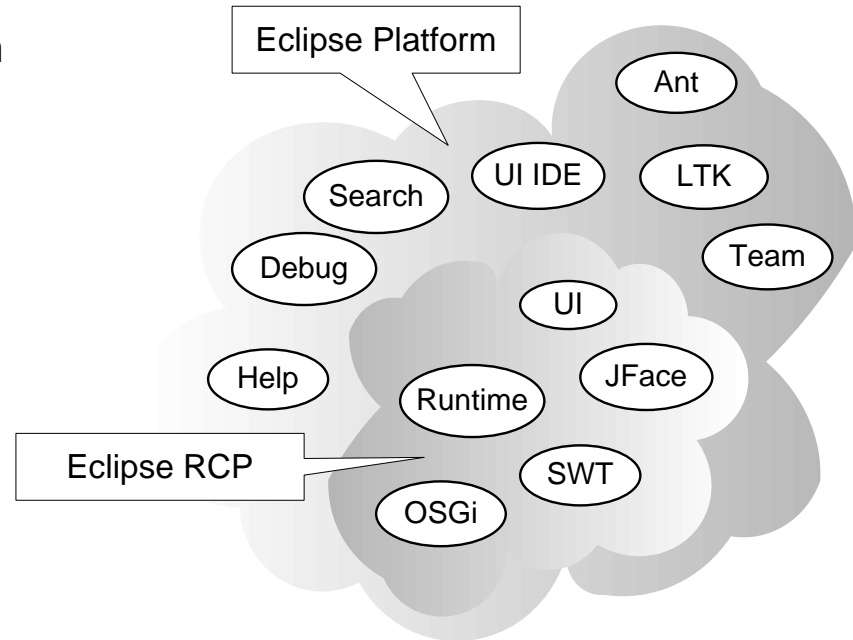
Jeff McAffer and Jean-Michel Lemieux
IBM Rational Software
Ottawa, Canada
Eclipse Platform Committers

**Authors of the upcoming book
“Eclipse Rich Client Platform - Designing, Coding, and
Packaging Java Applications” (Addison-Wesley)
Availability: Summer 2005.**

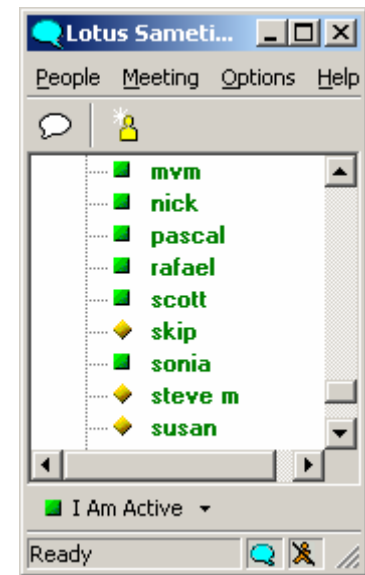
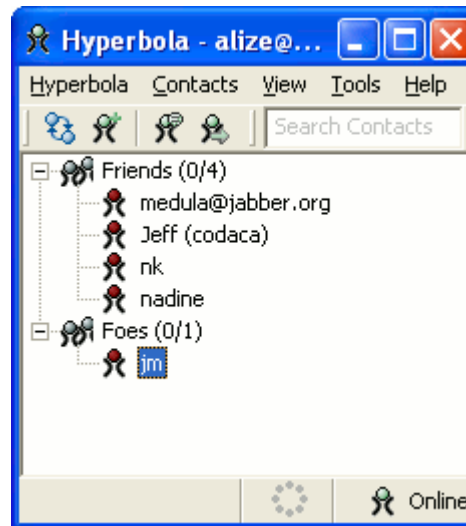
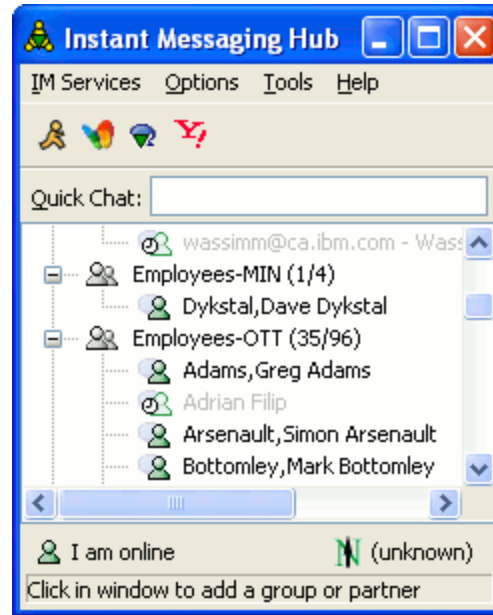
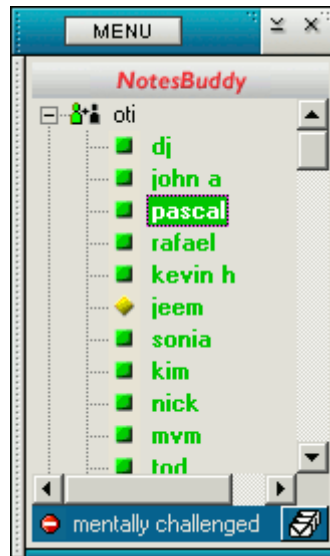
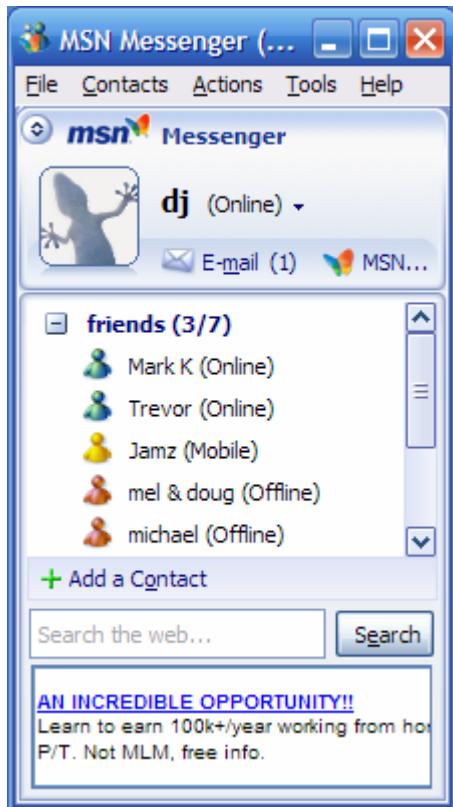
What is RCP?

While the Eclipse platform is designed to serve as an open tools platform, it is architected so that its components can be used to build just about any client application. The minimal set of plug-ins needed to build a rich client application is collectively known as the **Rich Client Platform**.

- A factoring of the full Platform
- There are other factorings
- Middleware



Can you tell the difference?



Hyperbola

- Hyperbola: Chat client product written for the RCP book
- Scenario: You've sold Hyperbola to a hospital
 - Facilitate communication between outpatient registration and admitting
 - Used to use e-mail
 - Moving to instant messaging
- Good News: They love it!

RCP Everywhere

- Bad News: They have new requirements – they want Hyperbola everywhere!
 - 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

Don't Panic! You've been to this talk!

Demo

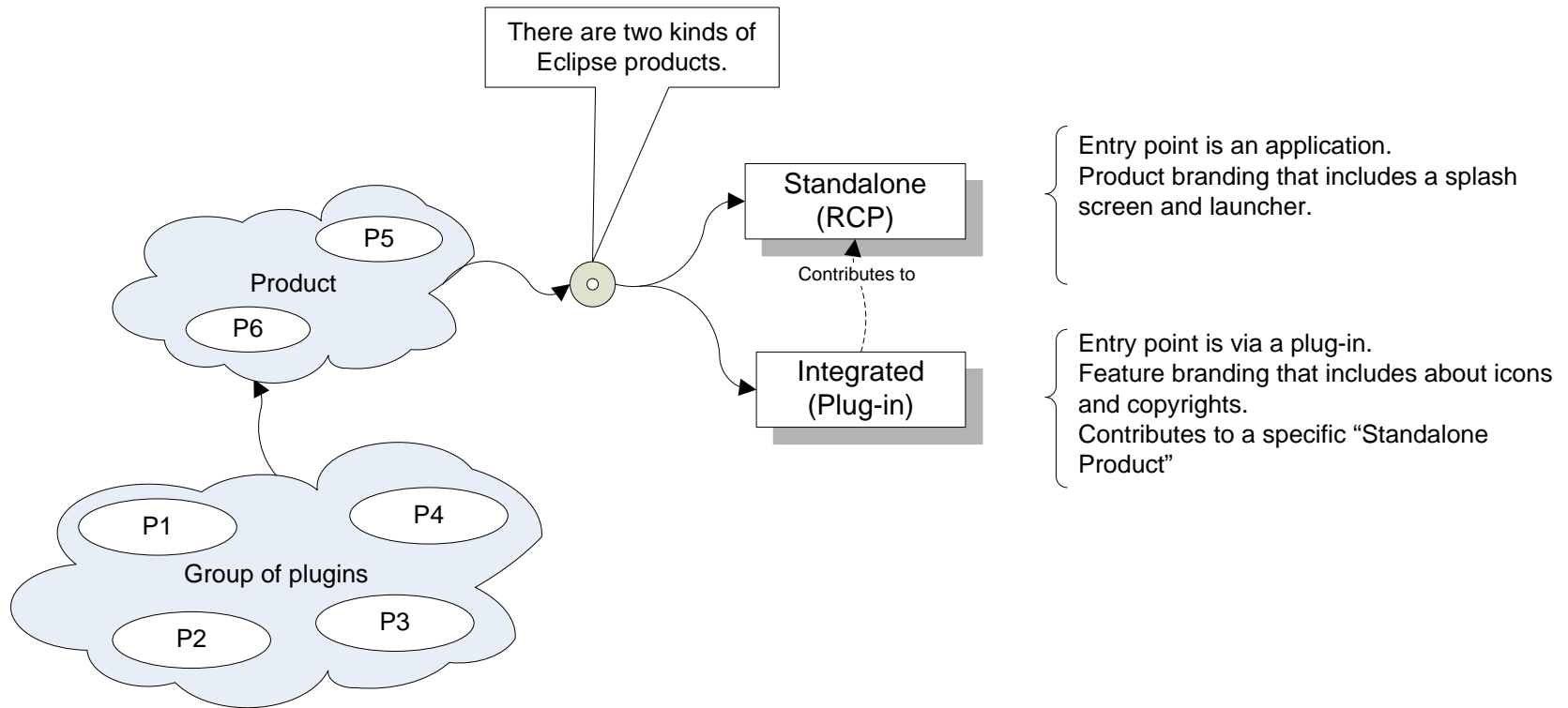
- Hyperbola in different configurations
- Real application needs
 - Looks good, task tray, functional
 - 3rd party library integration (smack)
- Canonical RCP application
 - Extend with debug feature
 - Swing integration
 - Kiosk
- Developers want to install into IDE
 - Deploy to update site and install into IDE

Hyperbola on a PDA



Some rules to follow...

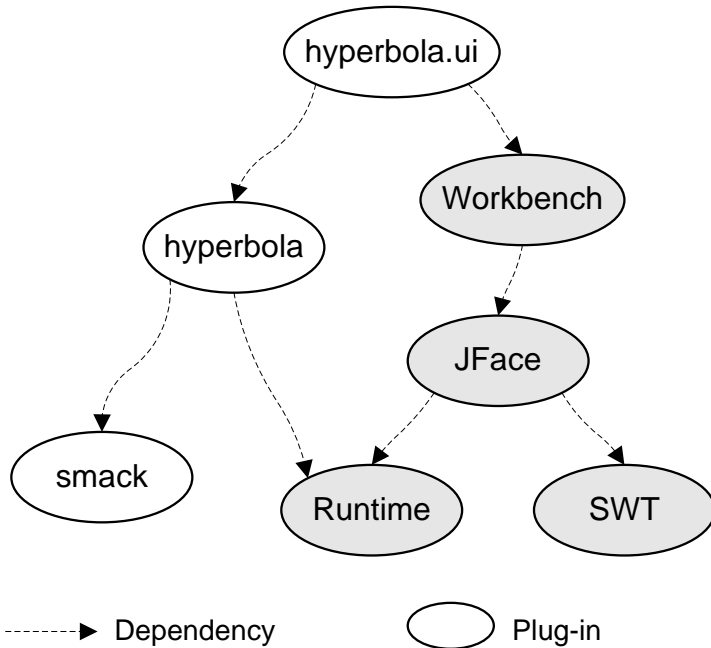
- Componentize ruthlessly
- Simplify structure
- Manage dependencies



Before and After

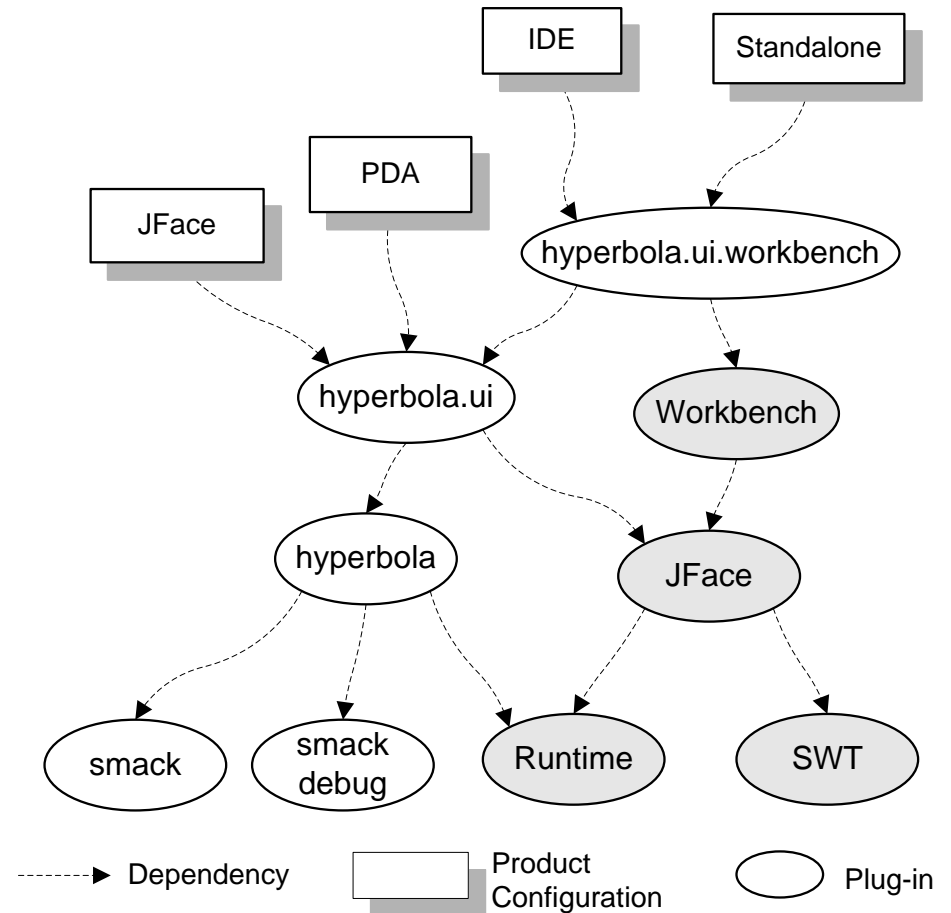
Before:

- Obvious RCP implementation
- Traditional Core/UI split



After:

- Several Product Configurations
- Refactored plug-in structure



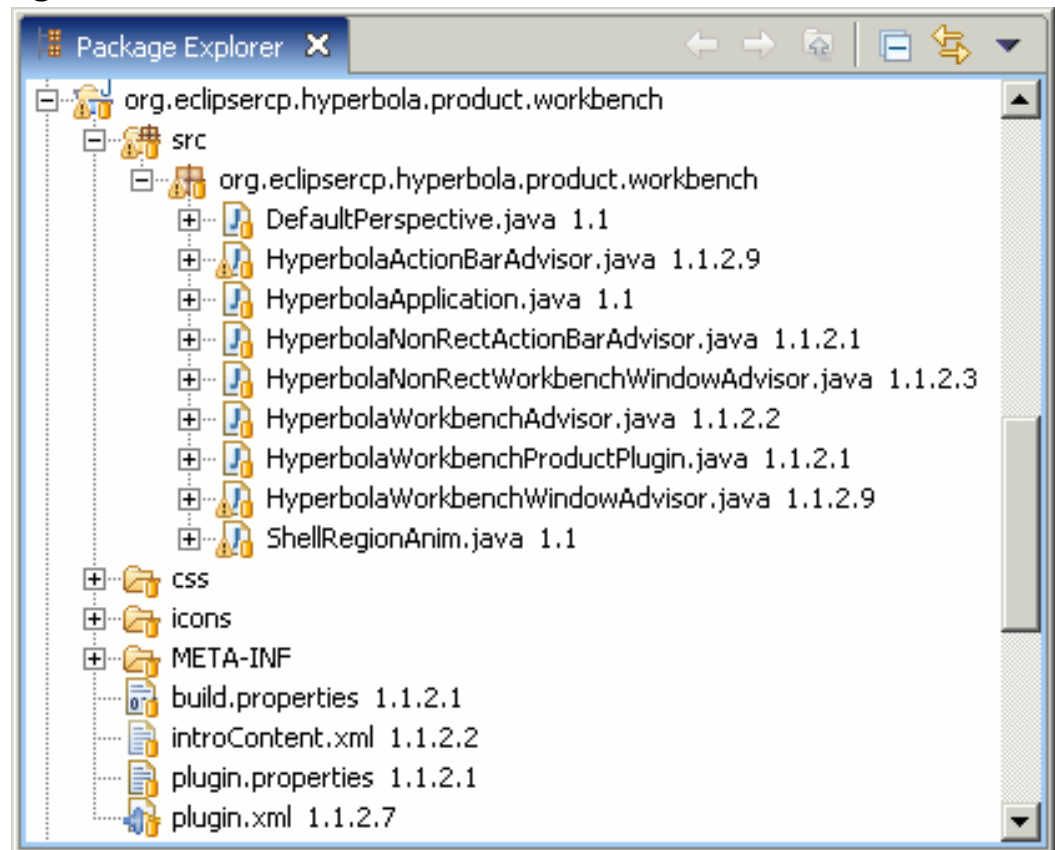
Why so many features and products?

- 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

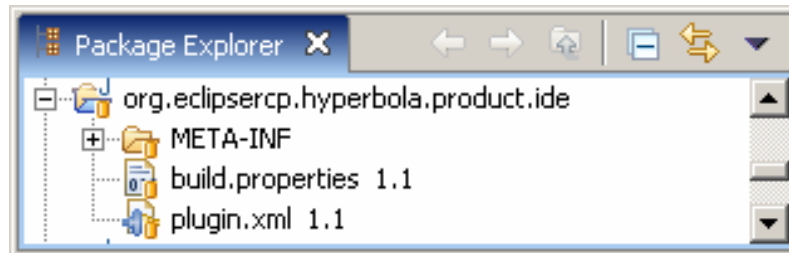
Workbench configuration

- Most classes are advisors
- Actual code is in framework plug-ins
- Advisors
 - Build the windows
 - Provide branding
 - Place the actions
- Adds product specific content
 - Branding
 - Intro content



IDE Configuration

- Integrated product (does not contribute application or product extensions)
- No code, all static mark-up!
- All actions reused from base workbench plug-in
 - Action sets include re-targetable actions that are plugged-in
- Plugin.xml contributes
 - Action sets
 - Views
 - Perspectives
 - Preference pages
 - Wizards



Demo

- Guided tour of our workspace
 - Plug-in structure
 - Feature structure
 - Product definitions

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

Images and Icons

- Product configurations typically use a base set of icons and images
- Factor common images into common plug-ins and share them
 - Just like you do with code
- Expose an image registry

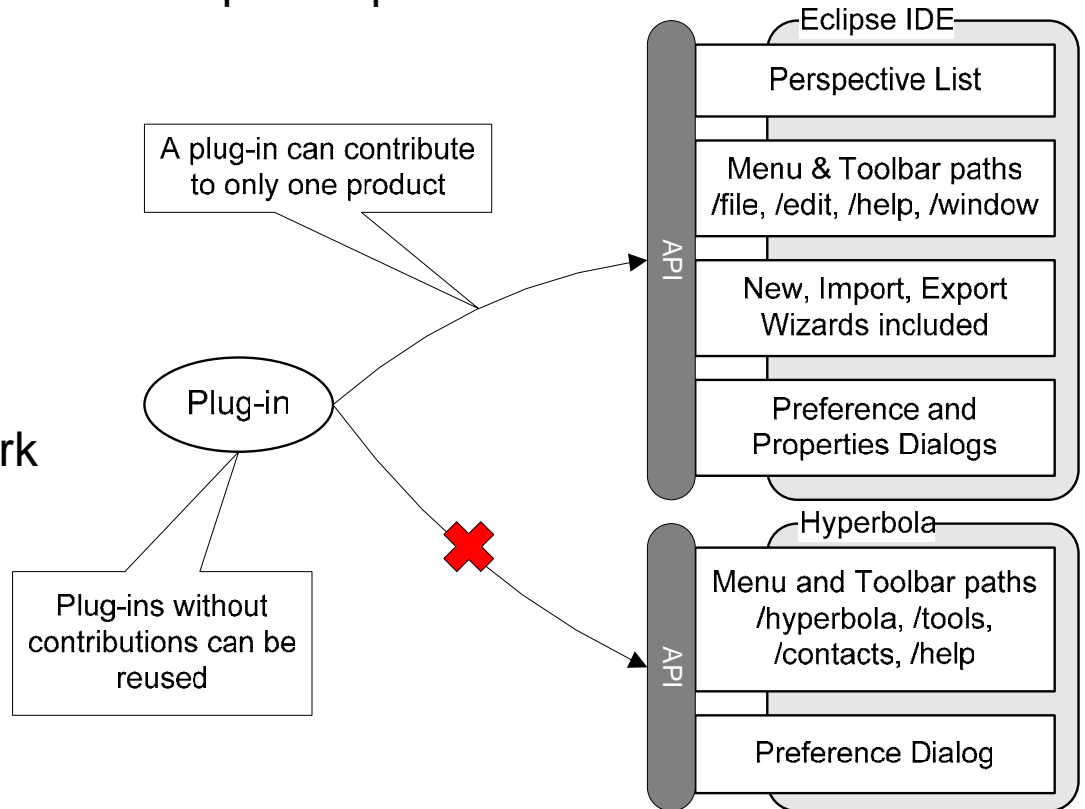
```
Hyperbola.getDefault().getSharedImages().getImage(IMG_ONLINE);
```

- Use URLs directly in the plugin.xml for sharing in extensions

```
platform:/plugin/org.eclipse.rcp.hyperbola.ui/icons/online.gif
```

Isolate Workbench Contributions

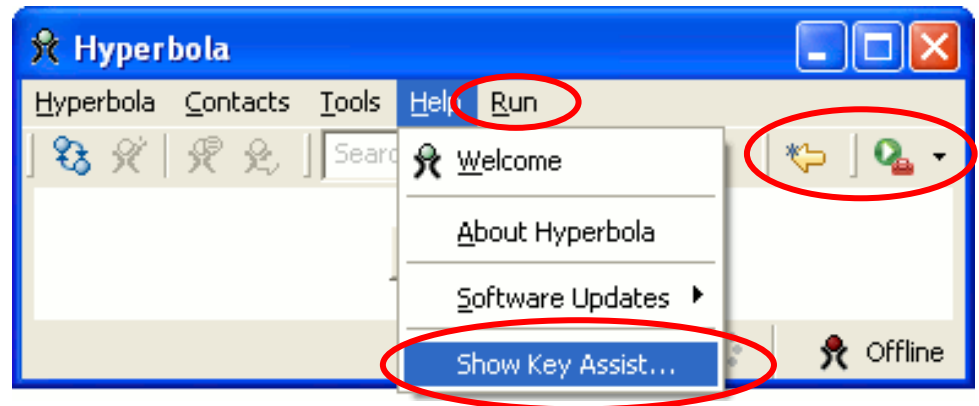
- Product configurations contribute to specific products
 - Many implicit APIs
 - Menu and toolbar paths
 - Predefined menus
 - Well known entry points
 - Show View
 - Open Perspective
- Increase reuse of framework plug-ins



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

Example: Re-targetable Actions

- Re-use via re-targetable actions
- Workbench product defines RosterView
 - Registers actions with the Workbench
- Standalone product adds re-targetable actions to the toolbar
 - The RosterView simply plugs in

```
IActionBars bars = getViewSite().getActionBars();  
addContactAction = new AddContactAction(shell, session);  
bars.setGlobalActionHandler(JabberActionFactory.ADD_CONTACT.getId(), addContactAction);
```

- Integrated IDE product adds an action set with re-targetable actions
 - No code is required and supports multiple RosterViews

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

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 is not just for tools anymore