Contributing to Eclipse: Understanding and Writing Plug-ins

Kai-Uwe Mätzel, Tom Eicher
Eclipse Committers
IBM Research, Zurich, Switzerland
kai-uwe_maetzel, tom_eicher @ch.ibm.com
Tutorial Outline

- **Understanding the Contribution Cycle**
  - Introducing the technical side of Eclipse
  - Using Eclipse to explore Eclipse
  - Becoming an Extender
  - Becoming a Publisher
  - Becoming an Enabler
  - Closing the circle

**Bonus**
- Improving as Extender/Enabler
- RCP: Getting away from integrated software development tools
Eclipse

- Is more than a Java IDE
- Has an open, extensible architecture
- Is built out of layers of plug-ins
Platform vs. Extensible Application

- Eclipse is a platform with a small runtime kernel
Platform Implications

- Everybody can contribute plug-ins
  - Every programmer can be a tool smith

- Creating opportunities for further extension makes it possible for the tool smith to benefit from the work of others

- “In many ways Eclipse is the Emacs for the 21st century.”

- It has to be easy to install and manage plug-ins
Eclipse Involvements

- Committers
- Enablers
- Publishers
- Extenders
- Configurers
- Users
Eclipse Involvements

- Users
  - Users of Eclipse

- Configurers
  - Adapt Eclipse to their personal needs by choosing and installing plugins and customizing them in anticipated ways

- Extenders
  - Providers of extensions to existing extension points

- Publishers
  - Extenders who make their extensions available using the Eclipse mechanisms

- Enablers
  - Providers of extension points others provide extensions for
The Contribution Circle

- Users
- Enablers
- Publishers
- Extenders
- Configurers
Tutorial Outline

- Understanding the Contribution Cycle
  - Introducing the technical side of Eclipse
- Using Eclipse to explore Eclipse
- Becoming an Extender
- Becoming a Publisher
- Becoming an Enabler
- Closing the circle

Bonus
- Improving as Extender/Enabler
- RCP: Getting away from integrated software development tools
Introduction

- The Eclipse plug-in architecture
- Plug-in activation
- The Eclipse platform
Eclipse Plug-in Architecture

- **Plug-in – set of contributions**
  - Smallest unit of Eclipse functionality
  - Big example: HTML editor
  - Small example: Action to create zip files

- **Extension point** - named entity for collecting contributions
  - Example: extension point for workbench preference UI

- **Extension** - a contribution
  - Example: specific HTML editor preferences
Contribution Rule

➢ Everything is a contribution.
Eclipse Plug-in Architecture

- Each plug-in
  - Contributes to 1 or more extension points
  - Optionally declares new extension points
  - Depends on a set of other plug-ins
  - Optionally contains Java code libraries and other files
  - May export Java-based APIs for downstream plug-ins
  - Lives in its own plug-in subdirectory

- Theoretically unbound number of plug-ins

→ Platform mechanisms must scale up to large numbers of plug-ins
Eclipse Plug-in Architecture

- Allow for loading on demand by separating declaration and implementation:
  - Declaration of plug-in contributions
    - Describes plug-in functionality
    - Describes UI elements to present plug-in functionality
    - Used by the platform to render parts of the plug-in’s UI
    - Specifies implementation classes
  - Implementation of plug-in contributions
    - Implemented in Java and provided as Java archive
    - Is loaded on demand when
      - An extension provided by the plug-in is called
      - Declared UI elements are triggered by the user
Tip of the Iceberg

- Startup time: $O(#\text{used plug-ins})$, not $O(#\text{installed plug-ins})$
Eclipse Plug-in Architecture

- Plug-in details spelled out in the **plug-in manifest**
  - Manifest declares contributions
  - Code implements contributions and provides API
  - `plugin.xml` file in root of plug-in subdirectory
Plug-in Manifest

```xml
<plugin
    id="org.eclipse.iconexplorer"
    version="1.0.0"
    name="Icon Explorer Plug-in"
    class="org.eclipse.iconexplorer.internal.IconExplorerPlugin" />
<requires>
    <import plugin="org.eclipse.core.resources" version="3.0.0" />
    <import plugin="org.eclipse.ui" version="3.0.0" />
</requires>
<runtime>
    <library name="iconexplorer.jar"/>
</runtime>
<extension
    point="org.eclipse.ui.preferencepages">
    <page
        id="org.eclipse.iconexplorer.preferences"
        icon="icons/knob.gif"
        title="Icon Exploration"
        class="org.eclipse.iconexplorer.internal.PreferencePage" />
</extension>
<extension-point
    name="Image Filters"
    id="imagefilters"/>
</plugin>
```

### Plug-in Identification
- **Id**: `org.eclipse.iconexplorer`
- **Version**: `1.0.0`
- **Name**: `Icon Explorer Plug-in`
- **Class**: `org.eclipse.iconexplorer.internal.IconExplorerPlugin`

### Other Plug-ins Needed
- Import plugins:
  - `org.eclipse.core.resources` version `3.0.0`
  - `org.eclipse.ui` version `3.0.0`

### Location of Plug-in’s Code
- Library: `iconexplorer.jar`

### Declare Contribution This Plug-in Makes
- Page:
  - **Id**: `org.eclipse.iconexplorer.preferences`
  - **Icon**: `icons/knob.gif`
  - **Title**: `Icon Exploration`
  - **Class**: `org.eclipse.iconexplorer.internal.PreferencePage`

### Declare New Extension Point Open to Other Plug-ins
- **Name**: `Image Filters`
- **Id**: `imagefilters`
Plug-in Activation

1. Open dialog

2. Collect all contributions for org.eclipse.ui.preferencepages

3. Get extension contributing Java preference page

<extension point="org.eclipse.ui.preferencePages">
  <page name="Java">
    id="...JavaBasePreferencePage"
    class="...JavaBasePreferencePage"
  </page>
</extension>

4. Instantiate class JavaBasePreferencePage

Plug-in Registry

Java UI Plug-in
Lazy Loading Rule

- Contribution code is only loaded when it is needed.
Eclipse Platform

- Eclipse Platform is the common base
- Consists of several key components
Strata Rule

- Separate core functionality from UI functionality.
Workspace Component

- Project – Folder – Files termed resources
- Tools manipulate resources in workspace
- Meta data management:
  - Natures (e.g., Web, Java)
  - Properties (persistent, session)
  - Markers
- Incremental builders
  - Builders are passed resource deltas
  - Delta describes changes since last build
  - Enable coordinated analysis and transformation of thousands of files
  - Any number of builders per project
- Local history
Workbench Component

- SWT – generic low-level graphics and widget toolkit
- JFace – UI frameworks for common UI tasks
- Workbench – UI personality of Eclipse Platform
SWT

- A portable widget set
  - OS-independent API
  - Uses native widgets where available
  - Emulates widgets where unavailable
  - Simple, small, fast: “less is more”

- Supported platforms
  Win32, WinCE
  Linux/Motif, Solaris/Motif, AIX/Motif, HP-UX/Motif,…
  QNX/Photon, Linux/GTK,
  Mac OS X/Carbon
JFace

- UI framework built on top of SWT
  - Viewers
    - Model aware adapters for SWT widgets
    - Trees, tables, lists, styled text, ...
  - Dialog, Preference and Wizard frameworks
- Actions
  - Location-independent user commands
  - Contribute action to menu, tool bar, or status line
Workbench

- Defines common user interface paradigm
  - Workbench
  - Views – navigation support, properties
  - Editors – edit files, e.g. Java Editor
  - Perspectives – arrangement of views and editors

- Extended by contributing
  - Views, editors, preference pages, wizards, …
Tutorial Outline

- Understanding the Contribution Cycle
- Introducing the technical side of Eclipse
  ➢ Using Eclipse to explore Eclipse
- Becoming an Extender
- Becoming a Publisher
- Becoming an Enabler
- Closing the circle

Bonus
- Improving as Extender/Enabler
- RCP: Getting away from integrated software development tools
Explore Eclipse with Eclipse

- The Plug-in Development Environment
- Explore a running Eclipse installation
- Create a self-hosting workspace
- Verify the described Eclipse architecture
Plug-in Development Environment

- Extenders use PDE to implement plug-ins
- PDE = Plug-in development environment
- Specialized tools for developing Eclipse plug-ins
- Built atop Eclipse Platform and JDT
  - Implemented as Eclipse plug-ins
  - Using Eclipse Platform and JDT APIs and extension points
- Features
  - Specialized views to explore a running Eclipse installation
  - Specialized PDE editor for plug-in manifest files
  - Templates for new plug-ins
  - PDE runs and debugs another Eclipse workbench
Available and Activated Plug-ins

- Plug-in name
- Activation flag
- Declared extension points
- Declared extensions
- Required plug-ins
- Provided Java archives
Self-hosting Workspace

- PDE allows to import the plug-ins of a running Eclipse installation into an Eclipse workspace.
Self-hosting Workspace

- Plug-ins correspond to Java projects

- Source projects “projects you are working on”
  - Consist of plug-in manifest, source
  - Source can be changed and compiled

- Binary projects “projects you are browsing only”
  - Consist of plug-in manifest, plug-in jar, source jar
  - Source can be inspected
  - Must not be compiled
  - Small foot print

- Project’s build class path is derived from the required plug-ins
Verify Eclipse Architecture

PDE Search

PDE Manifest Editor

Hyperlink to referenced Java files
Tutorial Outline

- Understanding the Contribution Cycle
- Introducing the technical side of Eclipse
- Using Eclipse to explore Eclipse
  ➢ Becoming an Extender
- Becoming a Publisher
- Becoming an Enabler
- Closing the circle

Bonus
- Improving as Extender/Enabler
- RCP: Getting away from integrated software development tools
The First Step in the Contribution Circle

- Icon Explorer example
- Plug-in for viewing the standard Eclipse icons
Becoming an Extender

- Create the Icon Explorer plug-in
- Declare and sketch the implementation of the Icon Explorer View
- Create a PDE launch configuration
- Run the Icon Explorer
- Finish implementation of the Icon Explorer View
Icon Explorer Plug-in

- PDE defines the notion of Plug-in projects
- Plug-in projects are (Java) projects with all the information to play the role of an Eclipse plug-in
- PDE provides Plug-in project creation wizards
Explicit API Rule

➢ Separate the API of your plug-in from its internals.

- Package naming convention
  - Packages containing internal types
    - `<plugin id>`.internal.*
    - `org.eclipse.iconexplorer.internal.*`
  - API packages
    - `<plugin id>).*`
    - `org.eclipse.iconexplorer.*`
Sharing Rule

- Add, don’t replace.

- Contributions do not
  - Override existing behavior
  - Remove or replace existing components
  - Harm existing or future contributions
Contributing a View

View implementation details:

- Extension point is `org.eclipse.ui.views`
- Interface `IViewPart` defines extension’s protocol
- Default implementation is `ViewPart`

```xml
<extension point="org.eclipse.ui.views">
  <view
    id="org.eclipse.iconexplorer.iconview"
    name="Icon Explorer View"
    class="org.eclipse.iconexplorer.internal.IconViewPart"
    icon="icons/iconexplorerview.gif"/>
</extension>
```
Conformance Rule

- Contributions must conform to expected interfaces.
Implementing a View

A view part has to implement two methods:

```java
void createPartControl(Composite parent)
```
- Method to create the view’s widget hierarchy
- Called by workbench to make a view visible

```java
void setFocus()
```
- Called by workbench to set focus to a widget when view receives focus

Often an additional method is added:

```java
void setInput(DomainType input)
```
- Sets the domain element the view is working on
- Called by client code
Viewer Details

- Adapt domain model to a widget
  - Accessing the contents/structure
  - Rendering domain objects

Diagram:
- TreeViewer
  - setContentProvider()
  - setLabelProvider()
- ILabelProvider
  - getImage(Object)
  - getText(Object)
- ITreeContentProvider
  - getChildren(Object)
  - getParent(Object)
  - inputChanged(Object)
- Adapter
- Domain
- Adaptee

Tree
- Widget
public void createPartControl(Composite parent) {
    int flags = SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL;
    fViewer = new TableViewer(parent, flags);
    fViewer.setContentProvider(new ContentProvider());
    fViewer.setLabelProvider(new LabelProvider());
    fViewer.setInput(new Object());
}

Icon Explorer View
PDE Launch Configuration

- PDE provides the launch configuration type “Run-time Workbench”
- Allows to specify plug-in search path and tracing options
- No plug-in deployment required
Relevance Rule

➢ Only contribute if you can successfully operate.

- Contribute as precise as possible
- Your are not the only extender
Tutorial Outline

- Understanding the Contribution Cycle
- Introducing the technical side of Eclipse
- Using Eclipse to explore Eclipse
- Becoming an Extender
  - Becoming a Publisher
- Becoming an Enabler
- Closing the circle

Bonus
- Improving as Extender/Enabler
- RCP: Getting away from integrated software development tools
Becoming a Publisher

- Deploy a plug-in
- Sidebar: Fragments
- Create a feature for your plug-ins/fragments
- Create an update site
Deploying a Plug-in

- **Development time:**
  - The plug-in code isn’t packaged as a JAR
  - Executed in a special development mode by PDE
    - Faster turn-around

- **Deploy:**
  - Package plug-in code as JARs
  - Deployed plug-in can be installed into a run-time Eclipse
Deployment Steps

- Remove unused plug-in dependencies
Deployment Steps

- Externalize all user visible text
  - Use JDT compiler to find non-externalized strings in Java code
  - Externalize the UI text inside declaration code

```xml
<plugin
  id="org.eclipse.iconexplorer"
  name="%pluginName"
  provider-name="%providerName" ...
```

```properties
pluginName=Icon Explorer
providerName=eclipse.org
```
Deployment Rule

➢ Execute all deployment steps.

- Allows Extenders to provide translations of your plug-ins
  → see Plug-in Fragments
Plug-in Fragments

- **Plug-in fragments** hold some of the plug-in’s files
  - Separately installable

- Each fragment has separate subdirectory
  - Separate manifest file: `fragment.xml`

- Logical plug-in = base plug-in + fragments

- Plug-in fragments used for
  - Isolation of OS dependencies
  - Internalization – fragments hold translations

- PDE supports Plug-in fragment projects
Features

- **Features** group plug-ins and fragments into installable chunks
  - An Eclipse configuration is a set of installed features
  - Feature manifest file: `feature.xml`

- Plug-ins and features bear version identifiers
  - major . minor . service
  - Multiple versions may co-exist on disk

- PDE supports Feature projects
Features

- Feature describes
  - Contained plug-ins, fragments, and their versions
  - Pre-requisite plug-ins needed by the feature to be installable
  - Copyright, license notice, required data archives

```
<feature
   id="org.eclipse.iconexplorer.feature" version="1.0.0">
   <requires>
     <import plugin="org.eclipse.ui"/>
   </requires>
   <plugin
     id="org.eclipse.iconexplorer"
     download-size="0"
     install-size="0"
     fragment="false"
     version="1.0.0"/>
</feature>
```
License Rule

- License your feature.

- Without license a feature can not be installed with the Eclipse Install and Update Manager
Update Sites

- Features are downloadable from update sites
  - Using Eclipse Install and Update Manager
  - Obtain and install new features
  - Obtain and install updates to existing features
Update Sites

- An update-site
  - Is any URL addressable location
  - Contains zips for the features and plug-ins
  - Version information encoded in the zip name
  - Contents described by a `site.xml` file

- PDE supports Update site projects

```xml
<site>
  <feature>
    <url>features/org.eclipse.iconexplorer_1.0.0.jar</url>
    <category name="default"/>
  </feature>
  <category-def name="default" label="Default">
    <description>Icon Explorer Plugins</description>
  </category-def>
</site>
```
Tutorial Outline

- Understanding the Contribution Cycle
- Introducing the technical side of Eclipse
- Using Eclipse to explore Eclipse
- Becoming an Extender
- Becoming a Publisher
  - Becoming an Enabler
- Closing the circle

Bonus
- Improving as Extender/Enabler
- RCP: Getting away from integrated software development tools
Becoming an Enabler

- Identify the scope of contributions to your plug-in
- Declare the extension points
- Implement the extension points
- Consume the new extension points
- Publish the extension points
Extending the Icon Explorer

- Scope for contributions
  - Different presentations of the Eclipse standard icons, e.g., grayed out, disabled, etc.
Invitation Rule

➢ Whenever possible let others contribute to your contributions.
Declaring an Extension Point

- Define an extension point in the manifest file

```xml
<extension-point id="imagefilters" name="Image Filters"/>
```

- Define an extension point schema

- Define an extension interface

```java
package org.eclipse.iconexplorer;

public interface IImageFilter {
    Image filter(Image image);
}
```
Revelation Rule

- Reveal the API a little at a time.
Extension Points

- **Plug-in A**
  - Declares extension point P (`org.eclipse.iconexplorer.imagefilters`)
  - Declares interface I (`org.eclipse.iconexplorer.IImageFilter`) for P

- **Plug-in B**
  - Implements interface I with its own class C (`GrayFilter`)
  - Contributes class C to extension point P

- **Plug-in A** instantiates C and calls its I methods (`filter(Image)`)
Diversity Rule

- Extension points accept multiple extensions.
  - Support user arbitration when there are conflicting contributions
  - Allow for additive behavior
Extension Point Implementation

- Change code (LabelProvider) to use extension interface (IImageFilter)

```java
public Image getColumnImage(Object element, int column) {
    Image image = getImage(element);
    return filterImage(image);
}

private Image filterImage(Image image) {
    Iterator e = getImageFilters().iterator();
    while (e.hasNext()) {
        IImageFilter filter = (IImageFilter) e.next();
        image = filter.filter(image);
    }
    return image;
}
```
Extension Point Implementation

- Load the defined extensions on demand from the plug-in registry
  → Lazy Loading Rule

```java
private List fImageFilters;

private List getImageFilters() {
    if (fImageFilters == null)
        fImageFilters = computeImageFilters();
    return fImageFilters;
}

private List computeImageFilters() {
    List filters = new ArrayList();
    loadImageFilters(filters);
    return filters;
}
```
Loading Extensions

```java
private void loadImageFilters(List filters) {
    IPluginRegistry r = Platform.getPluginRegistry();
    String pluginID = "org.eclipse.iconexplorer";
    String extensionPointID = "imageFilters";
    IExtensionPoint p = r.getExtensionPoint(pluginID, extensionPointID);
    IConfigurationElement[] c = p.getConfigurationElements();
    if (c != null) {
        for (int i = 0; i < c.length; i++) {
            IImageFilter filter = null;
            try {
                filter = (IImageFilter) c[i].createExecutableExtension("class");
                if (filter != null) filters.add(filter);
            } catch (CoreException x) {
                // handle exception
            }
        }
    }
}
```

See extension point declaration
Safe Platform Rule

As the provider of an extension point, you must protect yourself against misbehavior on the part of Extenders.
Protecting Icon Explorer

```java
private Image filterImage(Image image) {
    Iterator e = getImageFilters().iterator();
    while (e.hasNext()) {
        final IImageFilter filter = (IImageFilter) e.next();
        final Image input = image;
        final Image[] output = new Image[1];
        ISafeRunnable runnable = new ISafeRunnable() {
            public void run() throws Exception {
                output[0] = filter.filter(input);
            }
            public void handleException(Throwable x) {
                log(x);
                output[0] = null;
            }
        };
        Platform.run(runnable);
        if (output[0] != null) {
            image = output[0];
        }
    }
    return image;
}
```
Responsibility Rule

- Clearly identify your plug-in as the source of problems.
Extension Point Consumption

- Define the extension

```xml
<extension point="org.eclipse.iconexplorer.imageFilters">
  <filter
    name="Gray Filter"
    class="org.eclipse.iconexplorer.internal.GrayFilter"/>
</extension>
```

- Implement the extension interface

```java
public class GrayFilter implements IImageFilter {
  public Image filter(Image image) {
  }
}
```
Fair Play Rule

➢ All clients play by the same rules, even me.
Program to API Contract Rule

- In your contributions check and program to the Eclipse API contract.
Explicit Extension Rule

- Declare and describe explicitly how your contribution can be extended.

→ PDE Help, not covered in this tutorial
API Stability Rule

- Once you have invited others to contribute, try hard to keep your API stable.

  - API stability “work arounds”
    - Deprecate and forward
    - Start over in a new package
    - Extension interfaces

  - Preserve binary compatibility
Tutorial Outline

- Understanding the Contribution Cycle
- Introducing the technical side of Eclipse
- Using Eclipse to explore Eclipse
- Becoming an Extender
- Becoming a Publisher
- Becoming an Enabler
  - Closing the circle

Bonus

- Improving as Extender/Enabler
- RCP: Getting away from integrated software development tools
Closing the Circle

- Prepare reworked Icon Explorer for deployment
- Rebuild Icon Explorer feature
- Update Icon Explorer update site
  - Increase version number
Tutorial Outline

- Understanding the Contribution Cycle
- Introducing the technical side of Eclipse
- Using Eclipse to explore Eclipse
- Becoming an Extender
- Becoming a Publisher
- Becoming an Enabler
- Closing the circle

Bonus
- Improving as Extender/Enabler
- RCP: Getting away from integrated software development tools
Improving as Extender/Enabler

- Create test plug-in
- Identify properties that should be tested
- Identify testable properties
- Restructure code to match the two sets
- Implement tests
- Run tests
Code Separation Rule

- Separate testing code from plug-in code.

Diagram:

- Icon Explorer Test
  - uses
  - Icon Explorer
Create Test Plug-in

```xml
<plugin
  id="org.eclipse.iconexplorer.tests"
  version="1.0.0"
  name="Icon Explorer Tests"
  class="org.eclipse.iconexplorer.tests.IconExplorerTestPlugin">
  <requires>
    <import plugin="org.eclipse.ui" version="3.0.0"/>
    <import plugin="org.junit" version="3.0.0"/>
    <import plugin="org.eclipse.iconexplorer"/>
  </requires>
  <runtime>
    <library name="iconexplorer-tests.jar"/>
  </runtime>
</plugin>
```
Test Coverage Rule

- Cover both, the Extender and the Enabler behavior of your plug-ins.
Extender Test Case

- Is the Icon Explorer contribution available?
- Can the Icon Explorer be instantiated?

```java
public class IconExplorerTest extends TestCase {

    public void testViewCreationAndActivation() {
        IWorkbenchPage page = getActivePage();
        IconViewPart createdPart = createViewPart(page);
        assertNotNull(createdPart);
        IWorkbenchPart part = page.getActivePart();
        assertEquals(createdPart, part);
    }

    ...
}
```
private IWorkbenchPage getActivePage() {
    IWorkbench workbench= PlatformUI.getWorkbench();
    IWorkbenchWindow window= workbench.getActiveWorkbenchWindow();
    return window.getActivePage();
}

private IconViewPart createViewPart(IWorkbenchPage page) {
    try {
        IWorkbenchPart part= page.showView("org.eclipse.iconexplorer.iconview");
        if (part instanceof IconViewPart)
            return (IconViewPart) part;
    } catch (PartInitException x) {
        fail();
    }
    return null;
}
Running Plug-in Tests

- PDE provides the launch configuration type “JUnit Plug-in Test”
- Allows to specify plug-in search path and tracing options
- Allows to specify tests to be executed
- No plug-in deployment required
Running Plug-in Tests

- Allows to specify application to be executed
- Overall: Extension of “Run-time Workbench” launch configuration type
Enabler Test Case

- Does Icon Explorer adhere to the Safe Platform rule?

```java
public class IconExplorerTest extends TestCase {
    ... 
    public void testSafePlatformRule() {
        IWorkbenchPage page = getActivePage();
        IconViewPart part = createViewPart(page);
        part.setImageFilters(createNPEFilters());
        try {
            part.refresh();
        } catch (NullPointerException x) {
            fail();
        }
    }
    
    private IImageFilter[] createNPEFilters() {
        IImageFilter filter = new IImageFilter() {
            public Image filter(Image image) {
                throw new NullPointerException();
            }
        };
        return new IImageFilter[] { filter }; 
    }
```
Enabler Test Case

- Change IconViewPart to provide setImageFilters

```java
public class IconViewPart extends ViewPart {
    ...
    public void setImageFilters(IImageFilter[] filters) {
        if (fViewer == null) return;
        IBaseLabelProvider provider= fViewerCustomLabelProvider();
        if (provider instanceof LabelProvider) {
            LabelProvider labelProvider= (LabelProvider) provider;
            labelProvider.setFilters(filters);
        }
    }
}

public class LabelProvider ... {
    ...
    public void setFilters(IImageFilter[] filters) {
        fImageFilters= Arrays.asList(filters);
    }
}
```
Tutorial Outline

- Understanding the Contribution Cycle
- Introducing the technical side of Eclipse
- Using Eclipse to explore Eclipse
- Becoming an Extender
- Becoming a Publisher
- Becoming an Enabler
- Closing the circle

Bonus
- Improving as Extender/Enabler
  - RCP: Getting away from integrated software development tools
The Icon Explorer Application

Getting away from being part of an IDE

1. Create a plug-in
   ▪ Already done

2. Contribute a custom Eclipse application

3. Configure the workbench window
   ▪ Enable/disable workbench features
   ▪ Define the initial perspective
   ▪ Fill the menu and toolbar

4. Launch the workbench with your custom application
Define the Application

...even the entry point into the Eclipse platform is an extension point...

```xml
<extension id="application"
    point="org.eclipse.core.runtime.applications">
    <application>
        <run class="org.eclipse.iconexplorer.internal.Application"/>
    </application>
</extension>
```

```
IPlatformRunnable

run(Object args)
```

```
Application
```
Inside the Application

- Run the workbench with an advisor

```java
public class Application implements IPlatformRunnable {
    public Object run(Object args) throws Exception {
        WorkbenchAdvisor wa = new MinimalAdvisor();
        Display d = PlatformUI.createDisplay();
        return new Integer(PlatformUI.createAndRunWorkbench(d, wa));
    }
}
```
The Workbench Advisor

- **WorkbenchAdvisor**
  - A strategy object to configure a workbench window
  - Provides hook methods called at strategic points during the workbench life cycle
  - Defines the initial perspective

```
PlatformUI
createAndRun()=>

<<parameter>>

WorkbenchAdvisor

preStartup()
postStartup()
prewindowOpen()
postWindowOpen()
preshutdown()
postShutedown()
fillActionBars()
getInitialPerspective()
```
A Minimal WorkbenchWindow

```java
public class MinimalAdvisor extends WorkbenchAdvisor {

    public void preWindowOpen(IWorkbenchWindowConfigurer configurer) {
        super.preWindowOpen(configurer);
        configurer.setShowCoolBar(false);
        configurer.setShowMenuBar(false);
        configurer.setTitle("Icon Explorer");
    }

    public String getInitialWindowPerspectiveId() {
        return "org.eclipse.iconexplorer.perspective";
    }
}
```
Anatomy of WorkbenchWindow

Created with the Java Spider for Eclipse
www.javaspider.org
Defining the Perspective Layout

```xml
<extension point="org.eclipse.ui.perspectives">
  <perspective
    name="Icon Explorer Perspective"
    class="org.eclipse.iconexplorer.internal.PerspectiveFactory"
    id="org.eclipse.iconexplorer.perspective">
  </perspective>
</extension>
```

```java
public class PerspectiveFactory implements IPerspectiveFactory {
  public void createInitialLayout(IPageLayout layout) {
    String editors = layout.getEditorArea();
    layout.addView("org.eclipse.iconexplorer.iconview",
                  IPageLayout.TOP, 1.00f, editors);
    layout.setEditorAreaVisible(false);
  }
}
```
Launch the Application

- Inside Eclipse
  - Define “Run-time Workbench” launch configuration
  - Specify contributed application as program argument
  - Specify minimal plug-in search path

- Outside Eclipse

```
java -cp startup.jar org.eclipse.core.launcher.Main
   -application org.eclipse.iconexplorer.application
```
Applications

- Inherit the technical characteristics of Eclipse
  - Cross platform
  - Sophisticated, secure component system
  - Rich UI widget set based on native widgets
  - Rich UI framework

- Application metaphor
  - Wizards, preferences, properties
  - Perspectives, views, editors, workbench
  - Help system
  - Update system

- Application extension and integration
Applications

- Inherit the non-technical characteristics of Eclipse
  - Open source code base
  - Eclipse house keeping rules
  - Eclipse contribution cycle
Summary

- All functionality is provided by plug-ins and fragments
  - Includes all aspects of Eclipse Platform itself

- Contributions are made via extension points
  - Extensions are created lazily

- Plug-ins and fragments are packaged into separately installable features
  - Downloadable

- PDE and JDT turn Eclipse into the development environment to develop and test Eclipse plug-ins
Summary

3.0 “Eclipse is a general application platform”
2.0 “Eclipse is a general tooling platform”
1.0 “Eclipse is a Java IDE”
References

- [www.eclipse.org](http://www.eclipse.org)
- [www.eclipse.org](http://www.eclipse.org) > projects > The Eclipse Project > Platform > UI > Proposals > Rich Client Platform
- Gamma, Beck: Contributing to Eclipse – Principles, Patterns, and Plug-ins, Addison-Wesley, 2004
- [www.awprofessional.com/series/eclipse](http://www.awprofessional.com/series/eclipse)