# Extending MQ Explorer

**Mark Taylor**

***marke_taylor@uk.ibm.com***

***IBM Hursley***

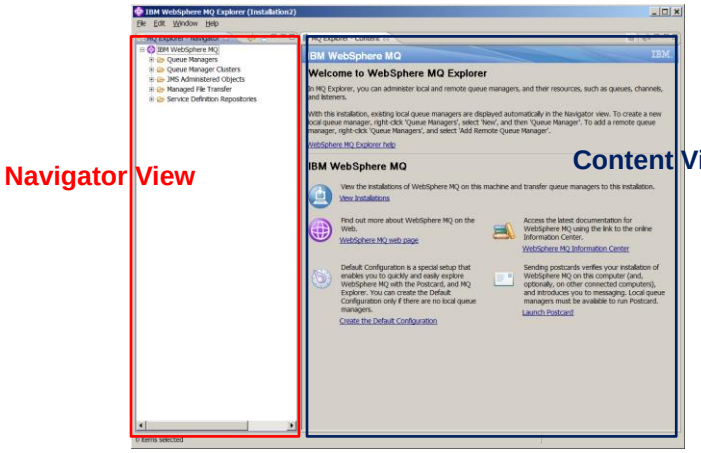
*Capitalware's MQ Technical Conference v2.0.1.3*

## Introduction

- I am going to show how to extend the MQ Explorer with new function
  - There will be code
  - But not too much
- My experience … SupportPac MS0P now contains 11 plugins providing
  - Event formatting
  - Activity monitoring
  - CSV exports for table
  - Connection wizard
  - Message Manager
  - Topic Viewer
  - Traceroute
  - Remote Admin
- See http://www.youtube.com/playlist?list=PLEE594DC49986AB67
  - Search for MS0P

## What is MQ Explorer (MQX)

- An configuration tool based on Eclipse
- Basic function includes configuration of MQ and JMS resources



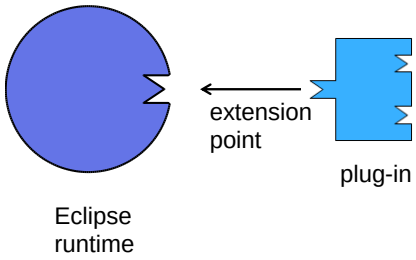



## Why Eclipse

- Eclipse is an open-source platform originally designed for application development tooling
- Provides core technologies which can then be extended with plug-ins
  - Giving common look-and-feel
- Many products and solutions use Eclipse-provided components
  - Then build unique value on top
- Enables cross-platform products without native coding
  - The most visible pieces (GUI widgets) use platform-specific implementations
  - Abstract interface (SWT) hides the details

## Plug-ins

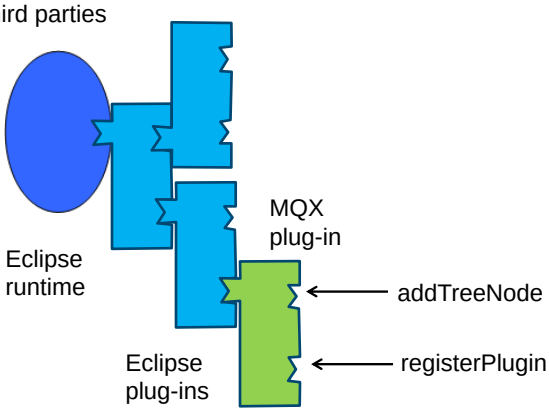
- Eclipse runtime is a small core program
- Provides extension points
- Plug-ins use those extension points
- Runtime dynamically discovers the plug-ins






## MQX implementation

- A set of plug-ins
- Which in turn provide further extension points
- Simplifying the addition of function
  - Either by other product-provided plug-ins
  - Or by third parties



## MQX packaging

- Packaging has changed over time
- Originally shipped only as part of the full MQ product
- Now also downloadable as SupportPac MS0T
  - Identical code to that in the product
  - But does not contain JNI libraries so cannot see any local queue managers
  - "Client-only" approach
  - Embeds the MQ Java client
- Originally included a full Java development environment
  - Though not visible by default
  - One of the standard Eclipse packages
- Now includes only the pieces needed for MQX to run
  - Runs as "RCP" application
  - But can be imported into other Eclipse environments
    - CICS Explorer, IDE etc
- Examples here will show V7.1/V7.5 formats

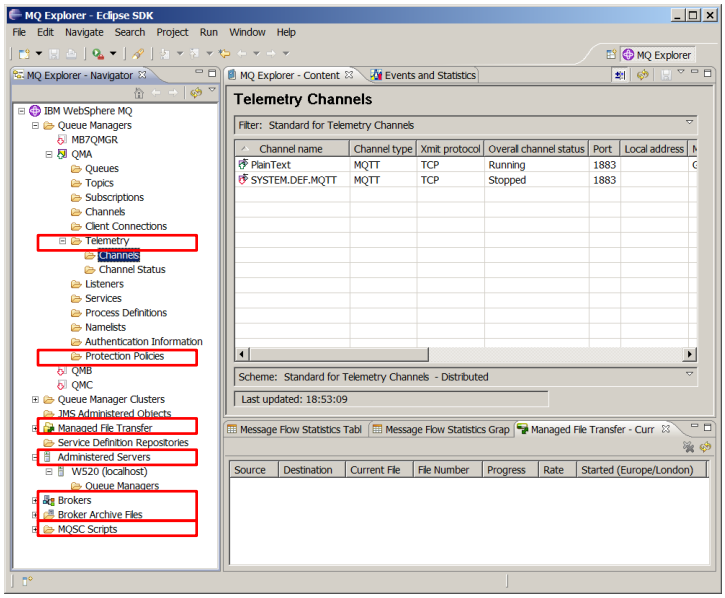


## Writing plug-ins

- An MQ plug-in exploits MQ-specific extension points
  - Making it easy to access defined queue managers
- But also any other Eclipse extension point

- There's no requirement for an MQX plug-in to do anything with MQ
  - For example, the service definition repository is mostly independent
- But you will probably use the MQ Java classes to interact with MQ
  - Full MQI available
  - PCF classes

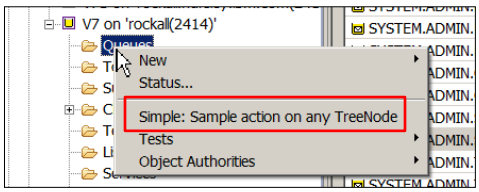
## MQX with extensions

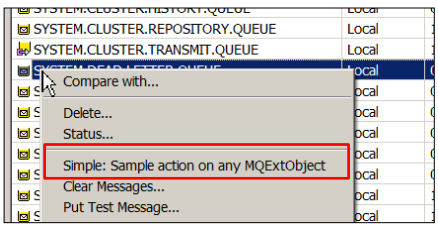


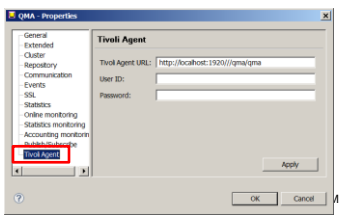



## What can you extend and modify

- In the navigator view:
  - Can add menu items to any tree node
  - Can add new nodes
- In the content view:
  - Can add menu items to items in table
- Can add new views
  - Arbitrary content
- Can inhibit some MQX operations
  - eg Block a queue manager from being deleted
- Can add tabs to object property windows
- Can add to import/export



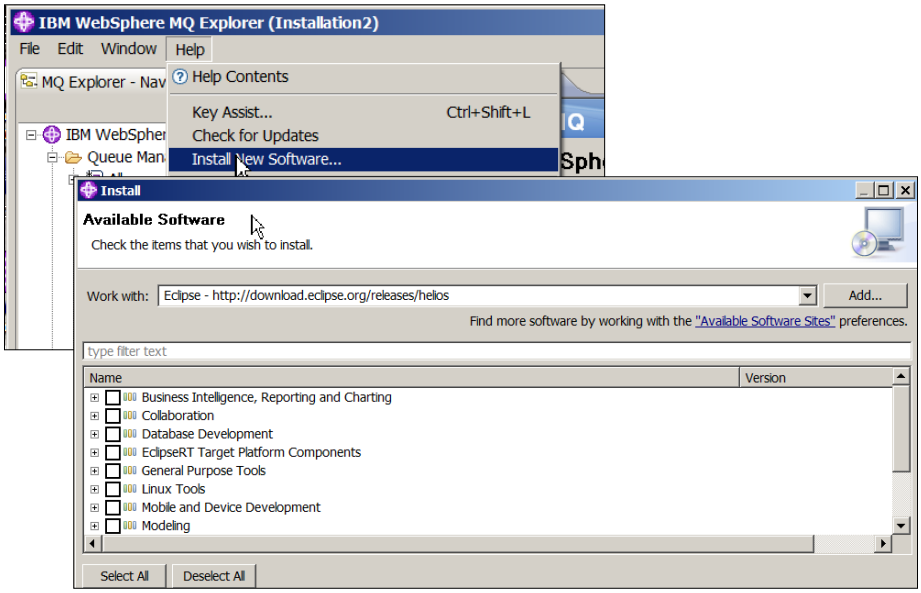
## What can you not (easily) extend or modify

- Cannot delete existing menu items
  - Or reorder them
- Cannot add to toolbars
- Cannot access object properties
  - Unless you use PCF commands
- Cannot add a new top-level tree node
- Cannot use existing text (translations) or icons

- The shipped core Eclipse plug-ins are limited scope
  - For example, do not have gef (graphical editors) or emf (model editors)
  - These can be added to MQX
  - Or you might require MQX to be run inside another Eclipse



## Adding public plug-ins

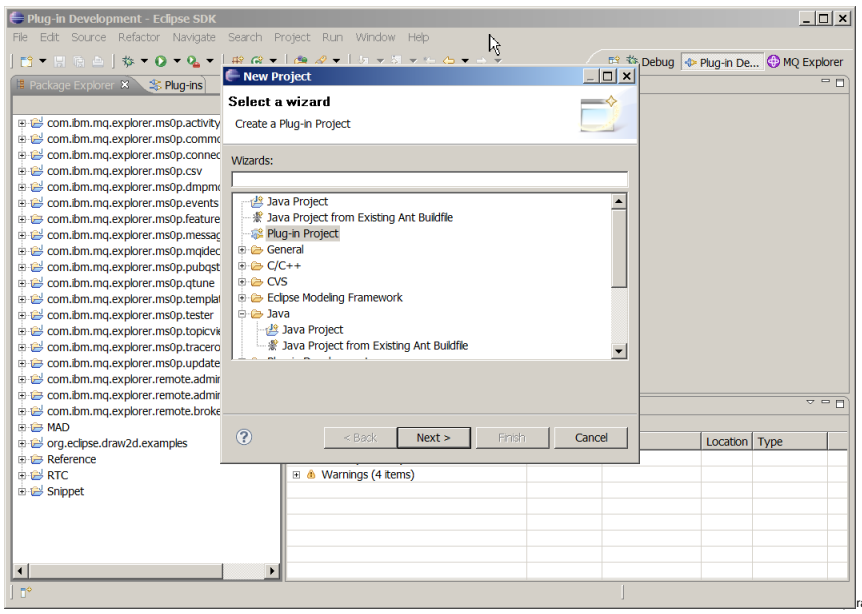


## Getting started

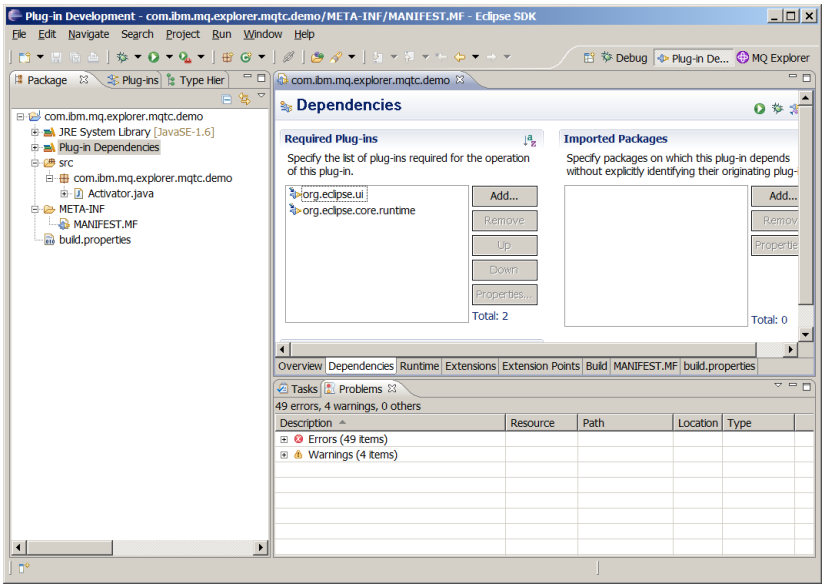
- To begin writing a plug-in, you need a development environment
- I was originally able to use the MQX-provided IDE
  - But that disappeared in V7.1
- My system now consists of a standard Eclipse 3.6.1 download
  - And then adding the MQX plug-ins to it
  - And any other plug-ins you care about (eg version control)
  - Lets me target V7.1 and later levels of MQX
- Open the plug-in development perspective and create a new project
  - This will create basic parts you will need for any Eclipse plug-in
    - Activator class and plugin.xml
  - An MQ plug-in has additional requirements



## Getting started



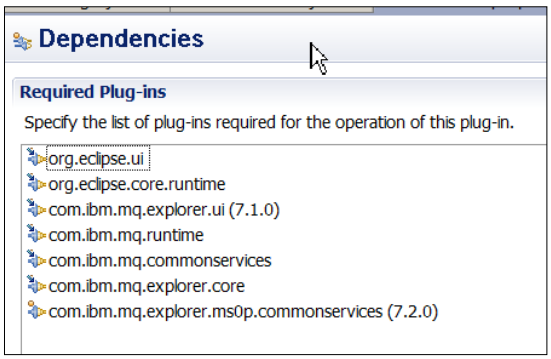
## Wizard-created template






## Dependencies

- The Manifest file shows which plug-ins you are dependent on
  - This file is edited in the same set of tabs with plugin.xml
- Can define version prereqs in here as well
  - Most MQ plug-ins will have a similar set of dependencies



## The MQX Extension Points

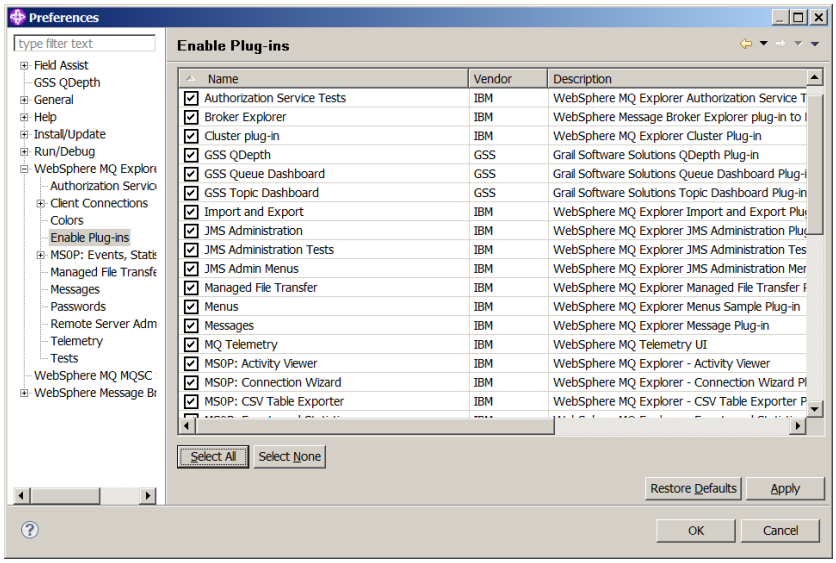
- There are several extension points
- The most important provided by MQX are
  - registerPlugin
  - addTreeNode
  - addContentPage
  - addImportExportSubcategory
- And the most commonly-used Eclipse extensions used in parallel are
  - popupMenus
  - preferencePages
- Define which extensions are being used, and any parameters, in plugin.xml
  - Eclipse knows the format of this file and can help fill it in



## registerPlugin extension point

- Registers a plug-in with MQX to receive event notifications
  - Your class **implements** IExplorerNotify
  - The plugin.xml stanza names the plug-in, names the class, gives descriptions
- Notifications include
  - MQX initalised or closing
  - Your plug-in is enabled or disabled
  - Queue manager added or removed
- Eclipse Java editor can create the class with empty methods
  - Showing TODO
- The MQX Preferences section “Enable Plug-ins” shows your plug-in
  - Disabling a plug-in in here will automatically remove its nodes and content pages

## Enabling MQ Plug-ins






## Plug-in lifecycle

- Plug-ins have a lifecycle
  - Enabled/disabled
  - Started/stopped
- Eclipse will load your plug-in
  - Activator is initialised (constructor), then “start”
  - Then MQX “enables” plug-in
  - MQX may not yet have completed its initialisation
    - So you may not be able to do anything yet
  - After “start” AND “enabled” AND “explorerInitialised”
    - THEN you can start working
- Asynchronous operations and parallelism means these may not always happen in the same order
- Similarly at shutdown, there are several opportunities to clean up
- One class could **extend** AbstractUIPlugin and **implement** IExplorerNotify
  - I prefer to have two classes

## addTreeNode extension point

- Allows plug-ins to contribute tree nodes to the Navigator View
- Plug-in specifies a class implementing ITreeNodeFactory
  - Responsible for creating and adding tree nodes
- The Navigator view extends the Eclipse CommonNavigator class
  - So there may be helpful methods available
  - For example to select a node
- One parameter to the extension point is the plug-in ID
  - This must match the name used when the plug-in was registered



## addContentPage extension point

- Allows plug-ins to contribute pages to the Content View
- Plug-in specifies a class implementing IContentPageFactory
  - Responsible for creating content pages
- Tree nodes know which content page should be shown when selected
- So in plugin.xml

```
<extension point="com.ibm.mq.explorer.ui.addcontentpage">
<contentPage
contentPageId="com.ibm.mq.explorer.ra.servers.folder.content"
pluginId="com.ibm.mq.explorer.ra"
class="com.ibm.mq.explorer.ra.content.AdminContentPageFactory"
name="RA Content"/>
</extension>
```

- And in RATreeNode.java

```
public String getContentPageId() {
 return "com.ibm.mq.explorer.ra.servers.folder.content";
}
```

## popupMenus extension point

- Eclipse extension point
  - But mediated by MQX
- Adds menu items to objects in tables or nodes in tree
- Menu visibility can depend on state
  - e.g. only connected Queue Managers
- The class you write has a selectionChanged() method
  - When an item is selected either in the content or navigator view
  - Table items are an MQExtObject class
  - Tree nodes return that object via the getObject method
- Can easily derive which queue manager is associated with an object
  - For example, if your menu is selected for a queue
- Then the run() method is invoked



## popupMenus extension point

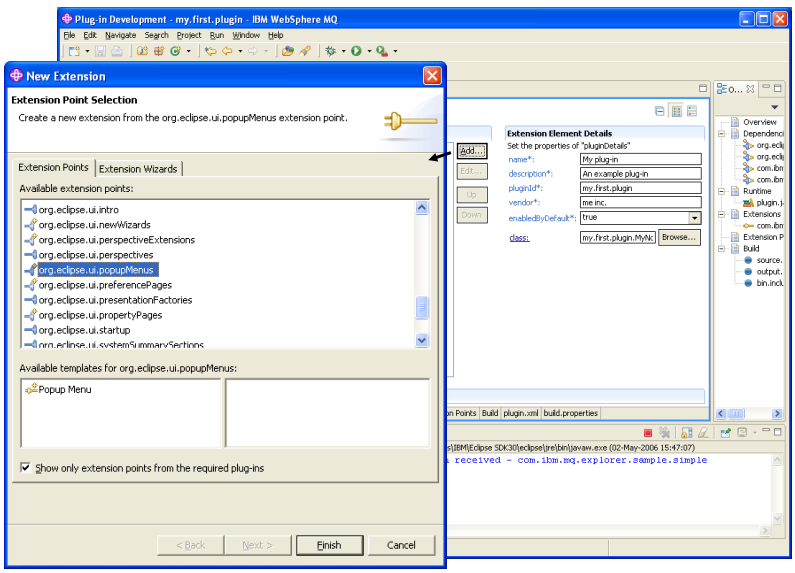
- Getting the order of your menu items correct can be challenging
  - Eclipse seems to add them in reverse order to how they appear in the XML
  - menubarPath will usually be set to “additions”
- Adding submenus can be done but it is not obvious
- Cannot have one entry to add an item for both an object and a tree node
  - For example, adding same menu to the qmgr in the Navigator and in the Content table
  - Same class can be referenced but needs to clauses in plugin.xml

## Menu definition in plugin.xml fragment

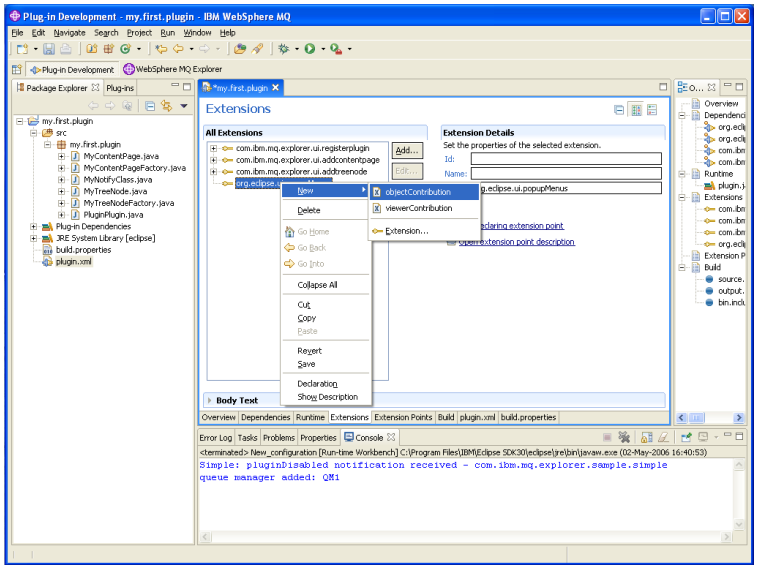
```
<extension point="org.eclipse.ui.popupMenus">
  <objectContribution adaptable="false"
     objectClass="com.ibm.mq.explorer.ui.extensions.TreeNode"
     id="com.ibm.mq.explorer.remote.admin.popup.GotoQMgr">
     <visibility>
       <and>
         <objectState value="com.ibm.mq.explorer.treenode.qm."
                      name="TreeNodePrefix"/>
         <objectState value="com.ibm.mq.explorer.remote.admin"
                      name="PluginEnabled"/>
       </and>
     </visibility>
     <action
        class="com.ibm.mq.explorer.ra.menuactions.MyActions"
        enablesFor="1"
        id="com.ibm.mq.explorer.ra.action.GotoQMgr"
        label="Select Server"/>
  </objectContribution>
</extension>
```



## Adding menu items

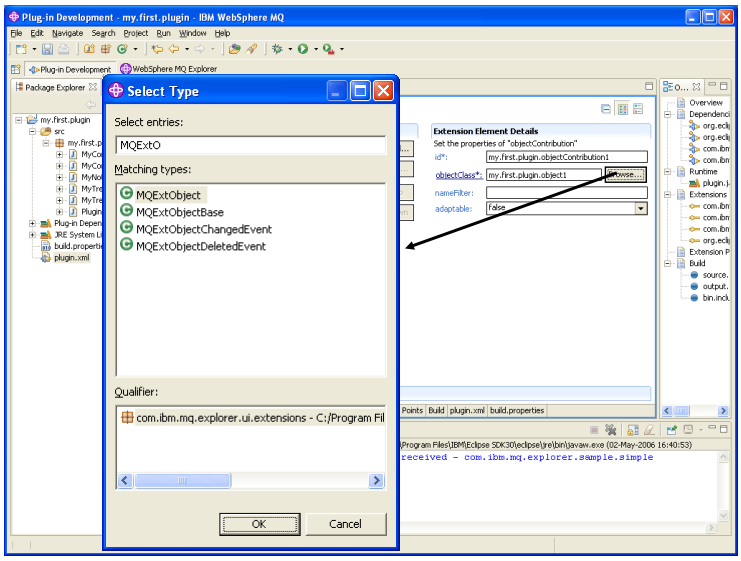


## Adding popup menu details

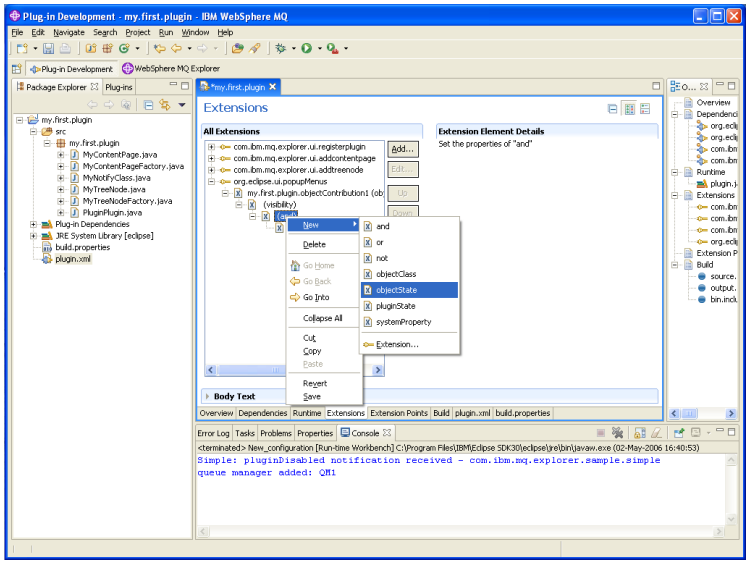





## Selecting the object

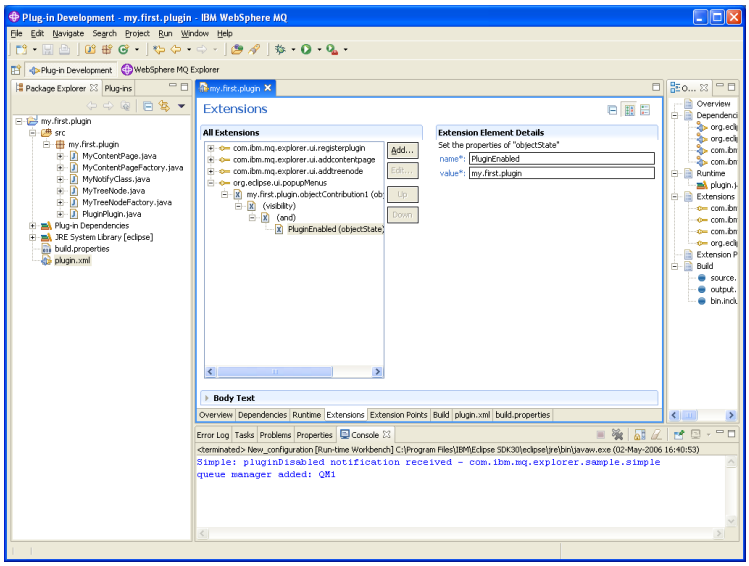


## Adding visibility filters

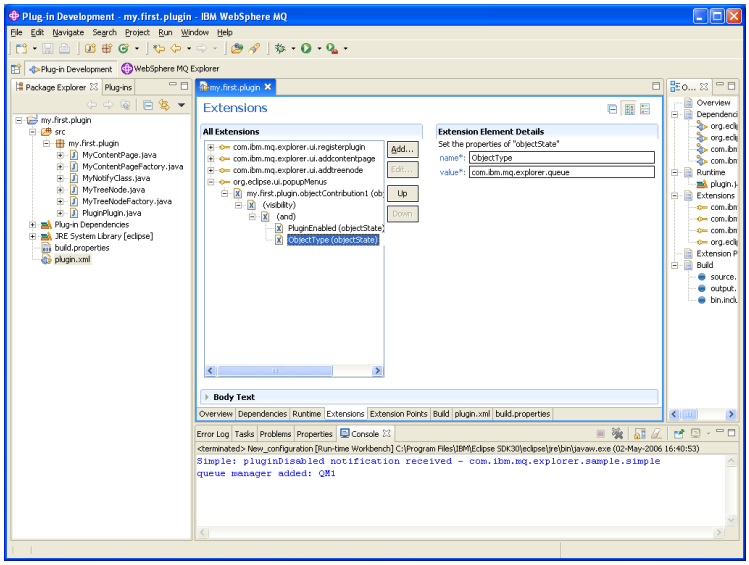





## Checking that our plug-in is enabled

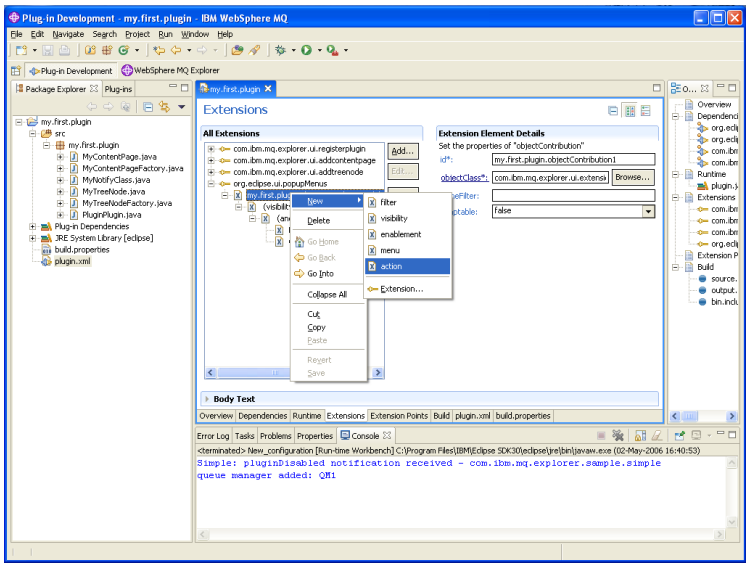


## Checking that the object is a queue

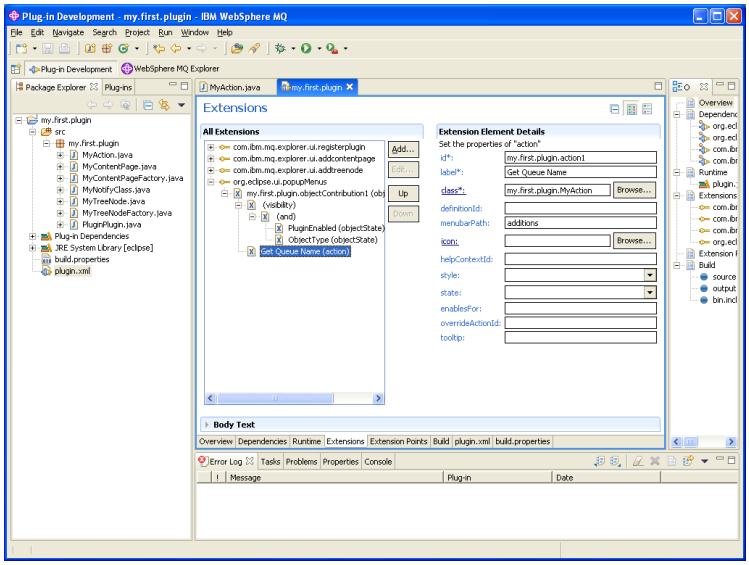





## Creating the action

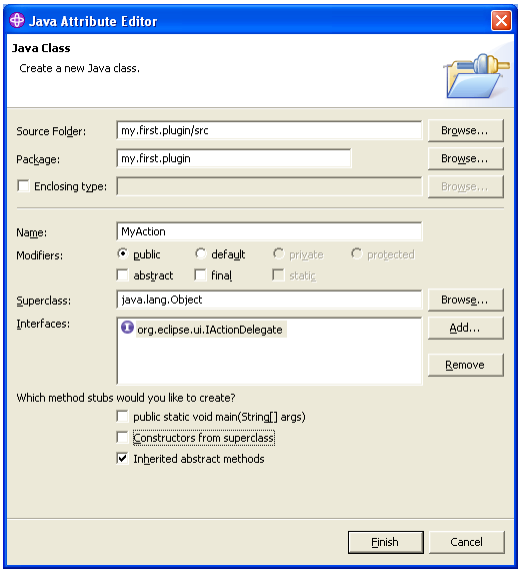


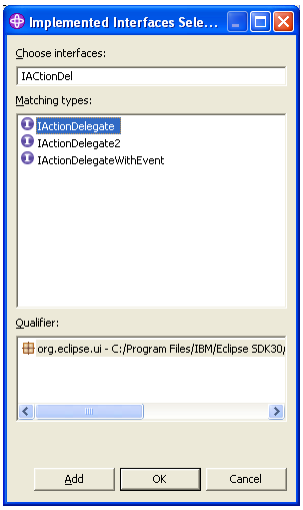
## Setting action details



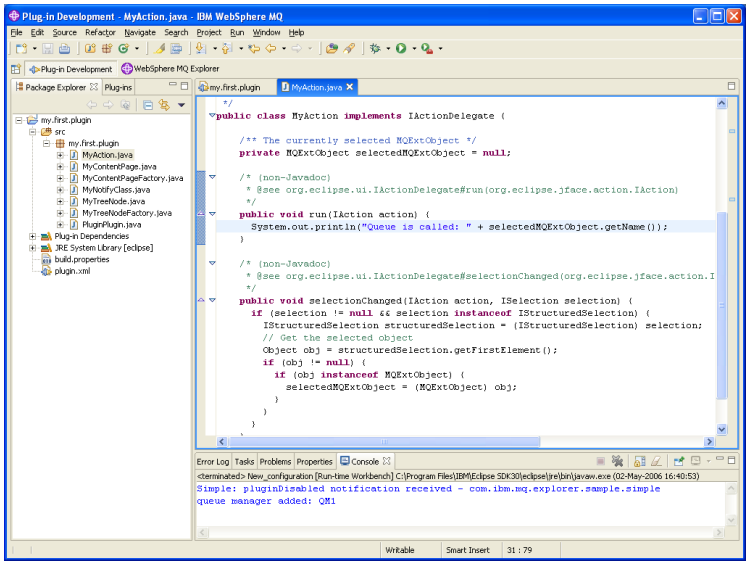



## Creating the menu action class



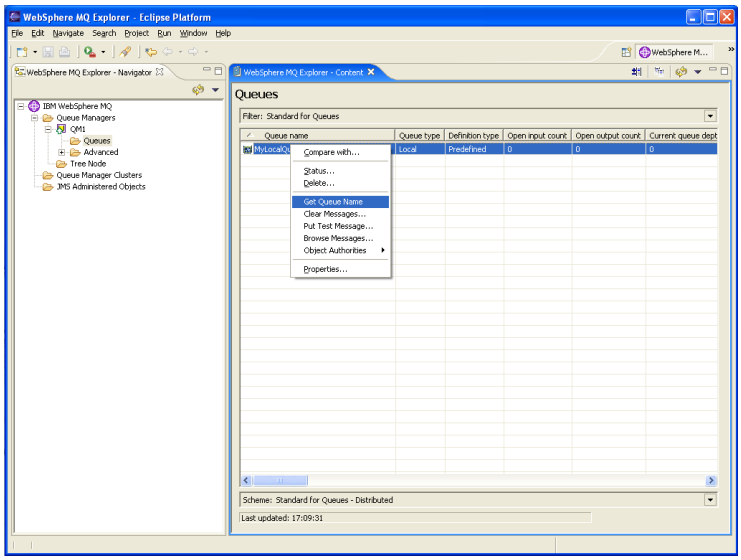



## Simple action implementation






## Test out the new menu item



## Nesting preference pages

- Adding preference pages under the main MQX panel
  - And then having further sub-pages

```
<extension point="org.eclipse.ui.preferencePages">
   <page class="...ms0p.events.eclipse.ViewPreferencePageRoot"
         category="com.ibm.mq.explorer.prefmain"
         name="MS0P: Events, Statistics, and more"
         id="…ms0p.events.eclipse.ViewPreferencePageRoot">
   </page>
   <page
     class="...ms0p.events.eclipse.ViewPreferencePageEvents"
     category="...ms0p.events.eclipse.ViewPreferencePageRoot"
     name="Events and Statistics"
     id="...ms0p.events.eclipse.ViewPreferencePageEvents">
   </page>
</extension>
```



## Working with a queue manager

- Assume that your plug-in has had its action invoked for a queue
- Your selectionChanged() method uses getParent() until it finds the qmgr
  - An instance of the MQQmgrExtObject class
- Call getMQQueueManager() returns a connection to this queue manager
  - So you do not need to worry about client configurations etc
  - This connection is for your private use, so disconnect when done

```
MQQueueManager qMgr = null;
if (selectedMQQmgrExtObject != null) {
  // This is a new connection, not a ref to the existing one
  qMgr = selectedMQQmgrExtObject.getMQQueueManager();

  … do some work with the connection …
}
try {
  if (qMgr != null)
    qMgr.disconnect();
} catch (Exception e) { // ignore exceptions
}
```

## Tips, Tricks and Techniques (1)

- I found the Java to be “easy”, but getting the plugin.xml right was harder
  - Find another plug-in that seems to hook similar ways to what you want
  - The XML is accessible to review, even when the rest of the source is not
- Some values need to be the same in both plugin.xml and the Java code
  - No easy way to do that with a single definition so use good naming conventions
- Insert plenty of trace and debug in the code
  - I use a trivial wrapper to put things to stdout so they appear in the console when launched as child of the development environment
- Do long-running work in background threads
  - GUI can only be updated in main thread
  - SWT will throw exception if called from wrong thread



## Tips, Tricks and Techniques (2) - Java reflection

- There are lots of classes and methods in Eclipse … not always easy to work out what to use
- Java reflection classes and methods can be very useful
  - Even when there are no guarantees of ongoing compatibility
- My CSV plug-in adds an item to the toolbar of most MQX tables
  - And then discovers the content of those tables
- There is no convenient mechanism in MQX to do any of that
- So the CSV code works first by
  - Hooking documented Eclipse methods to find all existing windows and parts
  - And asking to be notified if new windows and parts are created
  - If the MQ content view part exists or is created then add to the toolbar
- Then, it uses Java reflection to recursively decode the contents of the part
  - For each field in the current part, look at its type, and if it seems promising, look at that subtype’s fields
  - Until it finds a Table widget
  - At first, done under the debugger by hand to see if it worked at all

## Tips, Tricks and Techniques (3) –Templates and skeletons

- After writing a few plug-ins, a skeleton of core function simplifies the next
  - More function than the wizard-generated outlines
  - Gets basic operations running soon after your initial idea
- A template will deal with the lifecycle flows
  - Allocate and cleanup resources at the right times
  - Debug
  - A single menu item that can be invoked
  - Handy utilities for formatting or verification etc
- Eclipse wizards can help initial steps
  - But copy/paste is soon faster
- I'm now starting to move utility methods out of individual MS0P plug-ins
  - And into a single "internal" plug-in
  - So there aren't multiple similar copies
- Consider having simple standalone programs to drive dialog panels
  - To help with constant redesign of SWT layouts
  - Much faster than relaunching the whole Eclipse and getting to the right place

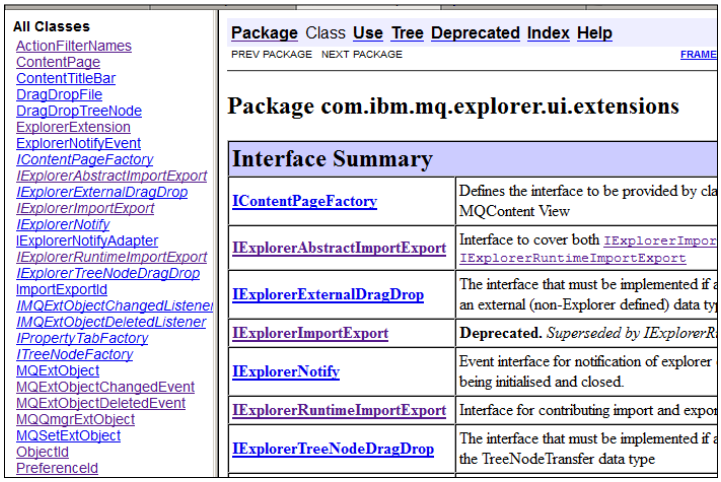


## Samples

- Samples provided with product
  - "simple" and "menus"
- These are in the plugins directory under your MQX installation
  - com.ibm.mq.explorer.sample.menus_<version>.jar
  - com.ibm.mq.explorer.sample.simple_<version>.jar
- The jar files can be expanded to show source
  - Java code
  - XML configuration

## MQX Documentation

- Javadoc provided for MQX external interfaces
- In a jar file in the plugins directory






## Eclipse documentation

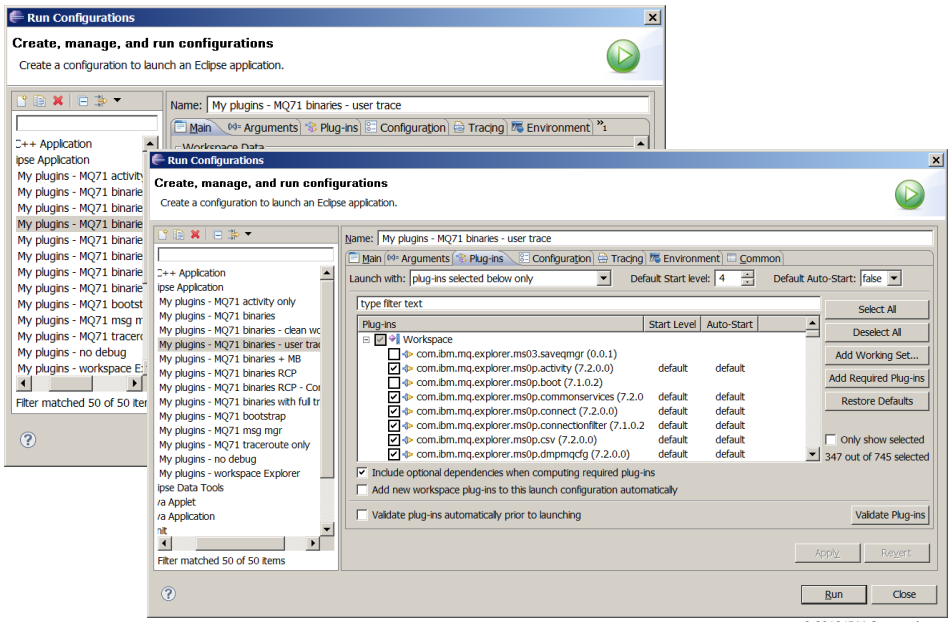
- Full set of Javadoc for Eclipse classes is in the SDK
- SWT help is at http://www.eclipse.org/swt/
  - Including samples and snippets
- Books:
  - I used "Java Developer's Guide to Eclipse" (D'Anjou, Fairbrother et al)
  - Plenty of newer ones now available
  - And specialised ones on EMF etc

## Testing

- Initial (unit) testing can be done from within the development environment
- "Run configurations"
  - What to launch
  - Special configurations
  - Which JRE
  - Which other plug-ins
  - Clean workspace?
- The same configurations are used for debug
  - For setting breakpoints, looking at variables etc
- But you then need to test with real installed MQX instances
  - I have seen examples of missing prereqs or bad version numbers
  - The prereq checking seems different from the IDE launch than runtime



## Run configurations



## Compiling, Building, Shipping

- Plug-ins can be delivered and installed in a variety of ways
- Each plug-in may be a subdirectory structure or a single jar
  - Compilation process must produce these
  - Eclipse integrates with ant tasks for building
- 3 installation mechanisms
  - Put the plug-in tree into the “dropins” directory
  - Put plug-ins in a private directory and point to it via a file in the “links” directory
    - Message Broker Explorer uses this
  - Package for use with the Eclipse “site” installer
    - More complex to build these, assembling individual plug-in jars and having a feature
    - But may be preferred, especially for centralised delivery
- One issue has been seen with Windows
  - When UAC is active, changes using any of these mechanisms can require running Explorer once “with admin authority”
  - Once processed, do not need admin authority



## Extract from ant build.xml

```
<property file="${basedir}\..\build_properties.xml"/>
<path id="classpath.refid">
       <fileset dir="${eclipsePluginsHome}">
                <include name="com.ibm.mq.**\**\*.jar" />
                <include name="com.ibm.mq.*.jar" />
                <include name="org.*\*.jar" />
                <include name="org.eclipse.*.jar" />
       </fileset>
       <fileset dir="${mqPluginsHome}">
                <include name="com.ibm.mq.**\**\*.jar" />
                <include name="com.ibm.mq.*.jar" />

       </fileset>
       <fileset dir="${rasHome}">
                <include name="**\*.jar" />
       </fileset>
</path>

<pathconvert targetos="windows"
       property="classpath" refid="classpath.refid"/>

<target name="classes" depends="clean">
<javac srcdir="${basedir}\src" executable="${javac}"
       destdir="${buildClassesPath}" fork="no"    optimize="true"
       debug="false">
       <classpath refid="classpath.refid" />
</javac>
</target>
```

## Troubleshooting

- Exceptions are not always visible
  - They are often propagated up to Eclipse and ignored
- But they may appear in the errorlog
- They will appear in the console when launched from within the development environment
- Normal MQ error logs and FFSTs will be created if you misuse MQ



## Summary

- Writing plug-ins is a good way to add value to MQ
- Without needing to write a load of framework code
- Can get straight into the real work