Migration Path for Extensible Launch Options

Michael Rennie

Eclipse Committer, Debug Team

IBM Canada, Rational Group

Introduction

Starting from 3.3 milestone there are significant changes in the way things are launched. Most of the changes are not transparent to contributors directly, but will be outlined along with the external changes directly affecting contributors. Along with the changes themselves, examples of the changes and the suggested migration path for each of the changes is discussed.

Overview

For the past few releases, launching in Eclipse has been strictly mode-based (run, debug, profile, etc). This paradigm works well for most cases where contributors want to add in their own launch configuration types, tab groups, launch delegates, etc. Where it falls short though is if a contributor only wants to add an additional tab to an existing tab group, or if they want to use a different launch delegate if certain criteria are met.

To add an additional tab traditionally would mean creating your own tab group duplicating all of the existing tabs of that group then adding your new tab, alternatively, to use a different launch delegate could mean listening to launch notifications and manually substituting your delegate in place of another or making modifications to the current command line.

Neither of the aforementioned scenarios are simple or robust. That is where extensible launch options come in.

1. The Players

All of the changes to the launching framework to enable extensible launch options occur in org.eclipse.debug.core and org.eclipse.debug.ui. The changes to the respective plugins are internal implementations and extension point changes (as explicitly outline in the last section *Changes Listing*).

All of the aforementioned changes, both transparent and internal are explained in the following sections, beginning with the changes and additions of extension points, contrasting examples of the changes and the suggested migration path for each. Following that the transparent implementation changes are discussed.

2. Launch Configuration Types Extension Point

The launch configuration types extension point has typically been the way to contribute both a launch configuration type and a launch delegate. In the past this has led to two distinct listings of launch delegates being maintained:

1. Launch delegates acquired from launch configuration type extension points
2. Launch delegates acquired from launch delegate extension points

Creating, maintaining, and working with multiple listings of delegates has been problematic, both for computational reasons (traversing many lists) and maintaining relationships between them (resolving duplicates, who launches what, etc).

The idea, for the launch configuration type extension point, was to abstract the notion of a launch delegate away from it and move in the image for the type (more on this in the *Launch Configuration Type Images* section). Removing the launch delegate from its associated type allows us to treat every launch delegate as though it is contributed (eliminating one of the listings of delegates), while also creating increased manageability of launch delegates (searching, acquiring, duplicate resolution).

Existing (pre-3.3)

The existing launch configuration types extension point allows everything to be added in one place, to form a 'default' launch delegate for the contributed type. See the following example of the Java Application launch configuration type.

```
<extension point="org.eclipse.debug.core.launchConfigurationTypes">
  <launchConfigurationType
    id="org.eclipse.jdt.launching.localJavaApplication"
    delegate="org.eclipse.jdt.launching.JavaLaunchDelegate"
    modes="run, debug"
    name="%localJavaApplication"
    sourceLocatorId="org.eclipse.jdt.launching.sourceLocator.JavaSourceLookupDirector"
    sourcePathComputerId="org.eclipse.jdt.launching.sourceLookup.javaSourcePathComputer"
    migrationDelegate="org.eclipse.jdt.internal.launching.JavaMigrationDelegate">
  </launchConfigurationType>
</extension>
```

Changes in 3.3

To effectively separate out the launch delegate from the launch configuration type contribution we deprecated the attributes: `delegate` and `modes.` The reason for the deprecations is that these attributes can be set in the launch delegates extension point when contributing a launch delegate (more on this in the following section). Furthermore as other parties have already contributed launch configuration types we had to maintain backwards compatibility.

The new extension point example of the Java Application contribution (deprecated items left out) is now trivial.

```
<extension point="org.eclipse.debug.core.launchConfigurationTypes">
  <launchConfigurationType
    id="org.eclipse.jdt.launching.localJavaApplication"
    migrationDelegate="org.eclipse.jdt.internal.launching.JavaMigrationDelegate"
    sourceLocatorId="org.eclipse.jdt.launching.sourceLocator.JavaSourceLookupDirector"
    sourcePathComputerId="org.eclipse.jdt.launching.sourceLookup.javaSourcePathComputer"
    name="%localJavaApplication"
    icon="$nl$/icons/full/etool16/java_app.gif">
  </launchConfigurationType>
</extension>
```

The noticeable changes to the contribution are the omission of the deprecated items mentioned above and the addition of the `icon` attribute. It should be noted though that the `icon` attribute is contribution relative, meaning the icon must be provided in the plugin that contributes the type definition.

Suggested Migration Path

The idea for these changes is to migrate away from using the launch configuration type extension point to describe your launch delegate, but instead providing your launch delegate as its own contribution.

Simply stated, to migrate away from declaring your launch delegate in the launch configuration type extension, do not provide information in the deprecated attributes, instead contribute a launch delegate as described in the next section.

3. Launch Delegates Extension Point

In the pre-3.3 world of launching, contributors could provided a launch configuration type with a default delegate (co-located in the same extension point), and other parties could contribute a launch delegate of their own for that type via the launch delegates extension point. This led to duplication of delegates in some cases, problems comparing 'default' delegates against 'contributed' ones to find which should be used, and in some cases complete inconsistency between delegates and associated UI (see bug 137057). To combat this problem in 3.3 we are considering all delegates as one in the same, the are no more 'default' and 'contributed' delegates, they are all just 'delegates'.

Existing (pre-3.3)

The existing description of the launch delegates extension point is sufficient as is, containing all of the attributes required to effectively describe any one launch delegate for any one launch configuration type. However, in order to provide users the ability to select a default launch delegate in the face of duplicates, and to allow launch delegates to be provisionally applied based on specific criteria some attributes needed to be added.

See the following example of a contributed launch delegate for the Java Application launch configuration type:

```
<extension point="org.eclipse.debug.core.launchDelegates">
  <launchDelegate
    delegate="org.eclipse.jdt.launching.JavaLaunchDelegate"
    id="org.eclipse.jdt.launching.localJavaApplicationDelegate"
    modes="run, debug"
    name="%localJavaApplication"
    sourceLocatorId="org.eclipse.jdt.launching.sourceLocator.JavaSourceLookupDirector"
    sourcePathComputerId="org.eclipse.jdt.launching.sourceLookup.javaSourcePathComputer"
    type="org.eclipse.jdt.launching.localJavaApplication">
  </launchDelegate>
</extension>
```

Changes in 3.3

With the need to resolve duplicate delegates (possibly involving user interaction) there needed to be a human readable name for the delegate, much the same way a launch configuration type defines a name. To facilitate this the attribute `name` was added to be used in preference pages and the like when a user must select a default launch delegate from a listing of duplicates.

Along with the name attribute, we have also added in a new sub-element for launch delegates, which is the `modeCombination` element. This element is used to explicitly describe a combination of modes that this delegate supports, each of which is described as a comma separated list of registered launch modes.

Consider the following example:

> We have a delegate that can launch java applications in debug mode and we also have a delegate that can launch java applications in debug mode *and* with code inspection. Both delegates reference the same launch configuration type, but when code inspection is enabled, we want to use our contributed delegate, not the default one.

If we consider that 'enabling' code coverage is the same as debugging but with the additional option of inspection, we can effectively retarget the debugging action to the appropriate delegate based on the mode combinations supplied.

Consider the following example where we contribute a launch delegate for the Java Application type and retarget to the Java Applet launch delegate for any of the set of modes [“run”, “debug”, “debug, coverage”, “run, coverage”].

```
<extension point="org.eclipse.debug.core.launchDelegates">
  <launchDelegate
    delegate="org.eclipse.jdt.internal.launching.JavaAppletLaunchConfigurationDelegate"
    id="org.foo.testing"
    modes="run, debug"
    name="Foo Launch Delegate"
    sourceLocatorId="org.eclipse.jdt.launching.sourceLocator.JavaSourceLookupDirector"
    sourcePathComputerId="org.eclipse.jdt.launching.sourceLookup.javaSourcePathComputer"
    type="org.eclipse.jdt.launching.localJavaApplication">
  <modeCombination
    modes="debug, coverage"
    modes="run, coverage">
  </modeCombination>
  </launchDelegate>
</extension>
```

The only tricky part of the new mode combinations is that each entry in the traditional `modes` attribute, will be separated per comma and added to the set of supported modes as individual entries. However, each of the `modes` attributes of the `modeCombination` sub-element will be treated as a set comprised of **all** of the modes of the comma separated list.

For example: if all that was supplied were the modes run, debug in the traditional `modes` attribute, the set of supported modes for that delegate would be {{run}, {debug}}. If however all that was supplied was `modes` attributes of the `modeCombination` sub-element (as shown above), the the set of supported modes for this delegate would be {{debug, coverage}, {run, coverage}}. Finally, with both contributed the set of supported modes becomes {{run}, {debug}, {debug, coverage}, {run, coverage}}.

Suggested Migration Path

The migration path is fairly straightforward. Contribute a launch delegate similarly to pre-3.3, except now give a human readable name and provide launch modes combinations (as required)

NOTE:

It is very important to note that when contributing a launch delegate make sure that all of the attributes that are deprecated in the launch configuration types extension point are provided here, and that none of the attributes are spread between the two contributions. For example do not provide a `sourceLocatorId` in the launch configuration type extension and the rest in a launch delegate. Provide all of the information in ***one or the other***.

4. Launch Configuration Tabs Extension Point

Another one of the challenges of extensible launch options and being able to retarget delegates etc. is a good way to provide additional UI to the launch dialog to allow users to set options or configuration attributes that translate to options.

Consider the following scenario:

> You want to contribute a code inspection tab to the Java Application launch configuration tab group.

Traditionally you would have to replicate all of the existing Java Application launch tabs and add in the one (or more) extra ones that you wish to contribute in a new tab group contribution. To eliminate the need to create any extra contributions, the launch configuration tabs extension point allows ILaunchConfigurationTabs to be contributed to existing tab groups. Optionally as well the placement of the tab can be specified in relative terms. More specifically, you can specify that your tab can be placed in relation to another specific tab, denoted by is id. If the id of the tab you wish to place your new tab relative to does not exist or cannot be found, your tab is placed at the end of the tab grouping.

Existing (pre-3.3)

Currently to add new UI to the launch dialog you really only have two options:

1. Create your own launch configuration type and tab group
2. Replicate an existing tab group with your UI additions

Changes in 3.3

For 3.3 we have added the ability to to make UI contributions to the launch dialog via the launch configuration tabs extension point. Let us revisit the previous example about contributing to the Java Application tab group; in this case though we will add a tab that will allow the option 'fooapplet' to be set.

```
<extension point="org.eclipse.debug.ui.launchConfigurationTabs">
```

```
  <tab
    class="test.launch.options.FooTab"
    group="org.eclipse.jdt.debug.ui.launchConfigurationTabGroup.localJavaApplication"
    id="org.eclipse.jdt.debug.ui.fooTab"
    name="Foo Tab">
    <placement
      after="org.eclipse.jdt.debug.ui.javaMainTab">
    </placement>
  </tab>
</extension>
```

To add a tab to existing UI now only requires that one class be created, which is specified in the `class` attribute and must implement the ILaunchConfigurationTab interface. The `group` attribute is the unique id of the ILaunchConfigurationTabGroup that you wish to contribute this tab to, `id` is the unique id of the tab, and `name` is a human readable name for the tab (the text that appears on the tab itself in the dialog).

Suggested Migration Path

If you make launching UI contributions like those mentioned in the *Existing pre-3.3* section than the migration path is to contribute tabs via this extension point. To do so will require the removal of replicated tab groups and associated contributed launch configuration types, and to add the tabs that were previously defined in your tab group as new contributions via this extension point.

4. Launch Configuration Type Images Extension Point

This extension point allows an image to be specified for a given launch configuration type. Having the contribution of the image separated from the launch configuration type allowed UI (the image) to be separated from non-UI (if your launch configuration type was contributed in a plugin you consider non-UI).

Existing (pre-3.3)

Images could be contributed to specific launch configuration types external from the configuration type contribution.

Consider the following example, where we contribute the image for the Java Application launch configuration type:

```
<extension point="org.eclipse.debug.ui.launchConfigurationTypeImages">
  <launchConfigurationTypeImage
    icon="$nl$/icons/full/etool16/java_app.gif"
    configTypeID="org.eclipse.jdt.launching.localJavaApplication"
    id="org.eclipse.jdt.debug.ui.launchConfigurationTypeImage.localJavaApplication">
  </launchConfigurationTypeImage>
</extension>
```

Changes in 3.3

This extension point has not been deprecated in 3.3, in favor of adding the image directly in the launch configuration type contribution. The reason for maintaining this extension point when you can provide the information in the type definition is because the `icon` attribute is contribution relative; meaning the icon must exist in the plugin where it is defined. In the case where a contributor has all UI in one plugin and all non-UI in another but would like launching capabilities *without UI*, they can define a launch type without having icons an any other UI related artifacts in the non-UI plugin.

Consider the following example, where we now have the ability to move the Java Application image contribution to the launch configuration type extension:

```
<extension point="org.eclipse.debug.core.launchConfigurationTypes">
  <launchConfigurationType
    id="org.eclipse.jdt.launching.localJavaApplication"
    migrationDelegate="org.eclipse.jdt.internal.launching.JavaMigrationDelegate"
    name="%localJavaApplication"
    icon="$nl$/icons/full/etool16/java_app.gif">
  </launchConfigurationType>
</extension>
```

Suggested Migration Path

Unless there are special circumstances that require the separation of UI from non-UI to the point where images are physically located in another plugin, then the image for your launch configuration type should be contributed with its associated type. To simplify, remove the launch configuration type images contribution and add the image via the launch configuration type extension.

6. Launch Manager

The launch manager has been enhanced with new functionality which provides access to both launch options and launch delegates.

Launch Delegates

1. The method LaunchManager.getLaunchDelegates() allows access to all of the currently contributed launch delegates. In 3.3 delegates can come from launch configuration type or launch delegate extension points, but in the end are both resolved to an ILaunchDelegate object. Returns an array of ILaunchDelegate objects.
2. The method LaunchManager.getLaunchDelegates(String typeid) will return an array of actual instantiated ILaunchConfigurationDelegate objects (if found) that apply to the specified configuration type. If no delegates are found null is returned.

7. Launch Configurations

With new UI contributions which possibly set options for launch delegate retargetting, the launching framework needs some way to persist and reuse specified settings. To do so, options can be saved to a launch configuration and reused on subsequent launches.

1. The method ILaunchConfiguration.getModeCombinations() was added, to allow access to any options currently set on the associated launch configuration. There is no setOptions() method on ILaunchConfiguration to support the transaction template of working with launch configurations.

2. The method ILaunchConfigurationWorkingCopy.setModes(String[] options) allows options to be set on a working copy of the current launch configuration. These changes can then be saved back to the original configuration following the transaction template for working with launch configurations.

8. Launch Configuration Types

1. The method ILaunchConfigurationType.getContributorName() allows access to the plugin that contributed the associated launch configuration type.
2. The method ILaunchConfigurationType.getImageDescriptorPath() allows access to the new icon attribute. Returns null if there is not one defined.
3. The method IlaunchConfigurationType.getDelegate(Set modes) should now be used to query the type for its associated delegate to use. If a suitable delegate cannot be found null is returned.

9. Proxy Classes and Helpers

Proxy Classes

With new extension points comes new IConfigurationElements. Working with configuration elements can be a bit of a hassle, so we provide proxy classes to work with them (basically a wrapper with getter methods).

1. In 3.3 we have coalesced the notion of launch delegates to be one description; a LaunchDelegate. This proxy allows us to wrap either a launch delegate or a launch configuration type contribution as simply 'a delegate'.
2. Allowing tabs to be contributed called for a proxy to describe the extension for a launch configuration tab; ILaunchConfigurationTabExtension. This proxy only wraps the extension point, it does not instantiate the specified tab class until explicitly asked for (to preserve the lazy loading paradigm). There is also an internal implementation called LaunchConfigurationTabExtension.

Helpers

1. A new interface IConfigurationElementConstants was added to debug core which contains common IConfigurationElement node names represented as constants.
2. LaunchConfigurationTabGroupWrapper was added to wrap an existing tap group with any contributed tabs.