Introduction

This article is a detailed guide to the UML Profile Authoring support in the IBM® Rational® Software Architect, IBM® Rational® Systems Developer, and IBM® Rational® Software Modeler (Version 7.0 and later) products. It covers both the UI support and programmatic support (for users extending the tool itself). It mainly targets profile authors, but it also offers some insight into profile usage.

Part 1 of this series showed you how to create or import UML profiles. You also learned about built-in support for OCL and Java™, as well as how to apply a profile.

This article, Part 2 of this series, will address change management, deployment, localization, programmatic support, and best practices.

The specifically referenced versions (in chronological order as they were released) are 7.0, 7.0.0.1, 7.0.0.2, 7.0.0.3, 7.0.0.4, 7.0.0.5 (Rational Software Architect only) and 7.0.5 (Rational Software Modeler and Rational Systems Developer). This article clearly indicates where there is a difference in behavior between versions.
Change management

It is important to understand the change management aspects of working with profiles. When an older version of a profile is applied to a model, for instance, the migration process (to the newer version) uses the versioning mechanism built into the profile itself. That mechanism is completely independent of whatever source control system you are using. Of course, the existence of a source control system introduces even more complexity when you merge two versions of the profile, or two versions of a model using two different versions of the profile.

Profile versioning

Each profile file contains one or more versions of the profile. A new version is created every time the profile is saved, and older versions are not removed. It may help for you to imagine that a version is a runtime component of the profile, with links to its UML definition. When the profile is applied to a model, it is really the latest version (the runtime component) that is linked with the model. The profile's UML definition is linked to the model indirectly through the version.

The profile is validated when it is saved. If there are any errors found, then the profile is saved, but a profile version is not created. You (as the profile author) must fix all errors in order for the profile version to be generated. Warnings are also reported, but they do not prevent the profile version from being created. A common example of an error in a profile is a stereotype or class attribute with no type.

Only the latest version completely and accurately reflects the UML definition of the profile. All other versions only hold enough information to enable the smooth transition of a model -- with that version applied to it -- to the newest version of the profile. This is assuming that there were no incompatible changes made between the versions (more on this topic in the Compatibility rules section).

To see the current (latest) version number of the profile, select it in the Project Explorer and navigate to the General tab in the Properties view. It has a read-only Version field. The information can also be found in the Profile Editor, in the Versions table on the General tab.

To compare it with the version that is actually applied to the model, select the model in the Project Explorer and navigate to the Profiles tab in the Properties view. One of the rows will represent the profile of interest (if it is actually applied), and there is a Version column in the table.
Releasing a profile

Releasing a profile creates a special version of the profile. To invoke it, select the profile in the Project Explorer, right-click it, and click Release. You could also do so using the Profile Editor: select the latest version in the Versions table on the General tab and click the Release button. This launches the Release Profile dialog shown in Figure 1, where you can specify the release label that will be associated with the released version. The release label is completely arbitrary: it can be any string you wish to enter.

Figure 1. Release Profile dialog

![Release Profile dialog](image)

All released versions of the profile are listed in the Releases field on the General tab in the Properties view. All profile versions (including the released ones) are listed in the Versions table, on the Overview tab of the Profile Editor.

If the most recent version is released, then the Release Label field displays the label associated with it. If the profile has not ever been released, or if it has been modified since the last release, then the Release Label field will indicate that.

In addition to marking the latest non-released profile version as released (and associating it with the release label), all other non-released versions are removed. Consequently, non-released versions of the profile should be applied only to test models. The models with intermediate profile versions applied will be able to migrate to the released version, but they will lose data: applied stereotypes and values of stereotype contributed properties. The release process uses the latest non-released version as the release version unless the profile is in the modified state: in this case the profile is first saved, thus creating a new non-released version, and then that version is released.

There is a way to safely move data from models with the non-released profile version applied: before releasing the profile, all such models are migrated to the latest non-released version of the profile. The profile is then released without making any other modifications.

Releasing a profile also establishes a check-point: the tool will enforce compatibility rules when you make changes to a profile that has been released.

Compatibility rules

Certain changes made to the profile result in the loss of data during profile migration. The following actions will cause compatibility problems:
• Rename an element that existed in the previous version
• Delete an element that existed in the previous version
• Change the type of a Stereotype or Class attribute that existed in the previous version

The tool prevents you from making such changes, but only in regard to released versions.

For example, if you create a profile with one stereotype (s1) and release it, then add another stereotype (s2) to the profile, the tool would let you delete (or rename) s2 but not s1.

The compatibility rules are implemented through a live constraint: from the Preferences > Window menu, click Model Validation > Constraints. Next, click either UML 2.1 > Modeling > Profiles or UML 2.1 > Model Quality Dimensions > Compatibility. Finally, select the Profile Compatibility Restrictions constraint.

If you must make a change that would be incompatible with a released version, for whatever reason, then there is a back door of temporarily disabling this constraint. Of course, you should only do this if it is absolutely necessary. However, be warned that in this case it is most likely that there will be data loss during profile migration.

Migration

Migration upgrades the applied profile version in the model. It upgrades the model’s links, from the version that was previously applied to it, to the newest version of the profile. The applied stereotypes and values for stereotype-contributed properties are migrated to conform to the new UML definition of the profile.

If there were incompatible changes between the versions, some of the data will be lost. For example, if a stereotype (that has been applied to some elements in the model) has been deleted from the profile, then all of those stereotype applications will be lost too. The same will happen to all properties that the stereotype has contributed.

If the profile file does not contain the version that was previously applied to the model at all, then only the link to the applied profile version is updated. All applied stereotypes and values of stereotype-contributed properties are lost.

Auto migration

Automatic migration happens when a model is opened: as shown in Figure 2, if the system detects that an older version of the profile is applied, it will prompt you to perform the migration. Of course, you may decline.
However, if the profile file does not have the previously applied version anymore, the migration will result in removal of the applied stereotypes. When such a model is saved, you are prompted (see Figure 3), about preserving the unrecognized content (which is essentially the lost stereotype applications). If the profile file is missing altogether, then the tool does not offer the migration; you are simply notified about the missing profile.

Figure 3. Saving a model with unrecognized content

Both the behavior of opening and saving the model with unrecognized content can be personalized through the preference settings (click Window > Preferences, then navigate to the Modeling page). Your options are Always, Never, and Prompt. The default behavior is to prompt.

Manual migration

You can manually migrate an applied profile at any time from the Properties view (for the selected package) or from the Model Editor. To perform the migration, select the package (model) to which
the profile is applied in the Project Explorer and navigate to the **Profiles** tab in the Properties view, as shown in Figure 4.

The table lists all applied profiles, and any profile that is out of sync is clearly marked as such. The **Migrate Profile** button is used to perform actual migration. The same result could be accomplished through the Model Editor: navigate to the **Details** tab, and you will see the **Migrate** button in the **Applied Profiles** section.

**Figure 4. Manual profile migration**

![Manual Profile Migration](image)

**Compressing a profile**

It is possible to purge intermediate (non-released) versions without releasing the profile. To do so, select the profile in the Project Explorer, right-click it and click **Compress**. This deletes all non-released versions except the last one. If the profile is dirty, then it will first create a non-released version, then save it and keep both the version it created and the last non-released version before that one.

The same result could be accomplished through the Profile Editor: navigate to the **Overview** tab, and the **Compress** button is in the **Versions** section.

This action is useful when there is a prolonged development of the profile. Because a new version is created every time that the profile is saved, its size can grow quickly, which can significantly affect performance (both in terms of memory usage and speed).

**Deployment**

There are two ways to deploy profiles: file system-based deployment, and plug-in deployment. In either case, it is highly recommended to have a path-map pointing to the location of the profile. Although not mandatory, it is certainly the easiest way to manage the deployment. If you do not use a path-map, then the system will store a file path (relative, if possible) in the model file where the profile is applied. That might be OK in some cases, but definitely leads to problems when the models and profiles are shared, especially if the profile location is not fixed.
Also, it is highly recommended to keep profiles in a different location from the models where they are applied, although it is not forbidden that they share a location. The tool will store a relative path in models to reference such profiles, even if there is a path-map defined for that location.

**File system deployment**

The profile can be deployed anywhere on the file system: it does not necessarily have to be in the workspace. Of course, you have to keep profiles in your workspace for authoring.

However, it is advantageous if the location containing the profiles is pointed to by a path-map. Path-maps can be used only in conjunction with linked resources (this is an Eclipse concept: please see the Resources section for more details). Essentially, a path-map is assigned to a linked resource representing a folder.

To create a path-map, click **Window > Preferences**, and then click **Modeling > Pathmaps**. The result is shown in Figure 5.

**Note:** The tool-defined pathmaps are displayed in read-only mode.

**Figure 5. Pathmaps**

{image}

First use the link on the top of the page (Linked Resources) to create a linked resource pointing to the desired folder (see Figure 6). Then exit the preference page, reopen it, and navigate again to the Pathmaps page. This step is necessary because the newly created linked resource will not show up in the list until the Preferences page is restarted.
Now on the Pathmap page, enable the linked resource as shown in Figure 7.

Figure 6. Create new linked resource

Now on the Pathmap page, enable the linked resource as shown in Figure 7.

Figure 7. Enabling linked resource as pathmap

If the model and the applied profiles are shared, then the same path-map has to be created on every system where they are used. The tool will not enforce the existence or validity of the pathmap: it is up to your particular team to ensure that the pathmaps exist and point to valid locations. If the pathmap does not exist or is not valid, the model simply will not open properly, because the tool will be unable to find the applied profile (please see the Change management section for details).

Please note that you, as the profile author, cannot ensure pathmap usage: it is really the profile user who has to set it up.
Plug-in deployment

Profiles can be deployed through plug-ins. The plug-in mechanism gives more control to the profile author, and simplifies the job of using and maintaining the profile for the profile user. On the other hand, it does require a bit more work and knowledge from the profile author. Basically, the profile is stored in a plug-in and registered through its plugin.xml file. The plug-in is then distributed to the profile users, who must install it in their IDE (integrated development environment) -- or in the runtime environment -- depending on where they need to use the profile or plug-in.

Profiles deployed this way are managed by the system. That means that you do not have to browse for the profile in the file system, but rather select it from the drop-down list (Deployed Profile) when applying profiles. You have an option to prevent the profile from showing up in this list (more on this later).

Another useful feature is that you can define the pathmap in the plugin.xml file as well. That way the profile user does not have to worry about profile locations and path-maps either.

The profiles deployed through this plug-in mechanism can also use Java as the language for profile constraints.

The complexity of the plug-in holding the profile really depends on you and the actual design. You might create the plug-in with the sole purpose of deploying the profile, or the profile could be added to an existing plug-in where it will be just one of many artifacts. There could be code accompanying the profile, providing runtime manipulation of models where the profile is applied (again, if required).

The easiest way to configure the profileâs plug-in deployment in Rational Software Developer or Rational Systems Developer (V7.0.5 or later) is to use the Generate Profile Tooling menu item on profile in Project Explorer. Even if no tooling is required and only a plug-in holder for the profile needs to be built, the wizard is still the easiest way to accomplish this: simply clear all of the tooling options and a properly configured plug-in holding the profile will be generated. The remainder of this section covers manual set-up for plug-in deployment. It is useful in following scenarios:

- The profile is being built in V7.0 or one of the 7.0.0.x versions so the Generate Profile Tooling menu item is not available.
- The plug-in already exists. Perhaps it already contains the profile as well. You need to make some manual adjustments and needs to understand the semantics and syntax involved.

Location

The actual physical location where the profile is stored in the plug-in is not dictated by the tool; it is up to you to determine the directory structure that is the best for the particular plug-in. The profile can be stored as high as the root directory of the plug-in, or in any subfolder. There can be one or many profiles. The folder can also contain files and folders other than profiles.

That being said, the tool uses the convention that tool-created profiles are stored in the profiles subfolder of the plug-in’s root. This is useful for the user-created profiles as well.
Note: This is a convention only. It is not required nor enforced.

The profile file should be included in the binary build, which is set in the build.properties file. You can set it manually in that file, or through the Plug-in editor (using the Plug-in editor is covered in a separate section). You can either specify each profile separately, or simply include the whole folder containing the profiles.

Listing 1 is an example of the content of the build.properties file (it assumes that the profiles are in the profiles folder).

Listing 1. build.properties file

```properties
bin.includes = plugin.xml, \
  plugin.properties, \
  <some file>, \
  profiles/, \
  <some file>
```

The row defining the profile location is in bold font.

Dependencies

The plug-in hosting the profile will need dependencies to the org.eclipse.gmf.runtime.emf.core and com.ibm.xtools.uml.msl plug-ins. If the profile is going to use Java-based constraints, it will also need a dependency to the org.eclipse.emf.validation plug-in.

Registering a pathmap

Regardless of the location in the plug-in where the profile is stored, there should be a pathmap defined for that location. The pathmap can be registered through the plugin.xml file of the plug-in.

Listing 2 shows an example.

Listing 2. The pathmap element

```xml
<extension
  id="somePathmapExtensionId"
  name="somePathmapExtensionName"
  point="org.eclipse.gmf.runtime.emf.core.Pathmaps">
  <pathmap
    name="MY_SAMPLE_PROFILES"
    path="profiles">
  </pathmap>
</extension>
```

The extension point is org.eclipse.gmf.runtime.emf.core.Pathmaps. You can choose the extension's id and name. The pathmap element has two attributes: name and path. The name is going to be referenced elsewhere and should be unique. It should also indicate the owner and intent. The path is a relative path starting at the root of the plug-in.

The plugin.xml file can also define a pathmap pointing to a location in another plug-in. In that case, the pathmap element has another attribute: plugin, as shown in Listing 3. The value of the plugin attribute is the name of the referenced plug-in, without the version information.

Listing 3. A pathmap pointing to another plug-in

```xml
<extension
  id="somePathmapExtensionId"
  name="somePathmapExtensionName"
  point="org.eclipse.gmf.runtime.emf.core.Pathmaps">
  <pathmap
    plugin="someOtherPlugIn"
    name="MY_SAMPLE_PROFILES"
    path="profiles">
  </pathmap>
</extension>
```
Listing 3. The pathmap plugin attribute

```xml
<extension
   id="somePathmapExtensionId2"
   name="somePathmapExtensionName2"
   point="org.eclipse.gmf.runtime.emf.core.Pathmaps">
   <pathmap
      name="MY_SAMPLE_PROFILES_2"
      plugin="com.mycompany.myproduct.myplugin"
      path="profiles">
   </pathmap>
</extension>
```

Registering a profile

The profile itself is also registered through the plugin.xml file. Listing 4 shows an example using the pathmap **MY_SAMPLE_PROFILES** from the previous section, and three profiles (MyProfileA, MyProfileB, and MyProfileC) deployed in the profiles subfolder of the plug-in.

Listing 4. The plugin.xml file with three profiles

```xml
<extension
   name="someProfileExtensionName"
   point="com.ibm.xtools.uml.msl.UMLProfiles">

   <UMLProfile
      id="MyProfileA"
      name="%MyProfileA.name"
      path="profiles/MyProfileA.epx"
      required="false"
      visible="true">
   </UMLProfile>

   <UMLProfile
      id="MyProfileB"
      name="MyProfileB"
      path="profiles/MyProfileB.epx"
      required="false"
      visible="true">
   </UMLProfile>

   <UMLProfile
      id="MyProfileC"
      name="%MyProfileC.name"
      path="pathmap://MY_SAMPLE_PROFILES/MyProfileC.epx"
      required="false"
      visible="false">

      <LegacyProfileVersion
         description="%MyProfileC.3.description"
         migrationNotice="%MyProfileC.3.migration"
         version="http://schemas/MyProfileC/_UZU6IL2lEdyr3qYLLLwtaSA/3">
      </LegacyProfileVersion>
   </UMLProfile>

</extension>
```

The extension point is `com.ibm.xtools.uml.msl.UMLProfiles`. You can choose the extension's id and name. The `UMLProfile` element has the following attributes: `id`, `name`, `path`, `required`, and `visible`. 

The id should be unique, to ensure that the profile registration can be found programmatically. One could use a Java package convention for that, or something like <plug-in name>.<profile name>.

The name of the profile will be used for display purposes. For example, it is used in the list of deployed profiles when you apply profiles. The value is used as is unless it is preceded with a percentage sign (%). The percentage sign indicates that the name is localized, and will be retrieved from the plugin.properties file, using the remainder of the value as the key in the properties file (this is the standard Eclipse behavior for providing localized strings in the plugin.xml: read more about Eclipse in the Resources section). So, in this example the name for MyProfileB is defined in the plugin.xml file (MyProfileB) while the names for MyProfileA and MyProfileC are defined in the plugin.properties file using the specified keys (MyProfileA.name and MyProfileC.name, respectively).

You can specify the path in a number of different formats but it is advantageous to use the pathmap format. In this example, all three profiles use pathmaps, although it might not seem so at first glance. The paths for MyProfileA and MyProfileB are specified relative to the plug-in's root (profiles/MyProfileA.epx and profiles/MyProfileB.epx). However, the profiles folder is covered by the MY_SAMPLE_PROFILES pathmap, so the tool will automatically use the pathmap. The registration for MyProfileC is even more explicit: it states the pathmap directly (pathmap://MY_SAMPLE_PROFILES/MyProfileC.epx).

You should not use the required flag. If it is specified, the value should be set to false (which is the default anyway). There might be some extreme use cases where the use of this flag is absolutely necessary, but it should be done with extreme caution. Basically, if it is set to true, this flag tells the tool to apply that profile automatically to any model that does not have it already applied.

The visible flag indicates to the tool whether or not the profile should be in the list of deployed profiles when you apply profiles. The default is false. The value obviously depends on the intent: if the profile should be applied only programmatically, then the value should be false, otherwise it should be true.

The LegacyProfileVersion sub-element (in the example for the MyProfileC profile) registers version 3 as the legacy version of the profile. The legacy version is checked in the following scenarios:

- When you apply such a profile, you see a list of the profile's versions from which to choose to apply (as opposed to simply applying the latest version). The list consists of the latest version (and it is selected by default) and all registered legacy versions.
- When you apply a model with such a profile, and the tool detects that an older version of the profile is applied to it, the older profile version is checked against the list of legacy versions. If it is a legacy version then a) you are not prompted to migrate during model load and b) you are warned that it is legacy version if the user attempts to migrate it manually. Of course, you can still proceed and migrate.
The LegacyProfileVersion is available only in Rational Software Modeler or Rational Systems Developer V7.0.5 or later. The tool comes with one profile using this mechanism: the Ecore profile is registered with the legacy version corresponding to the version available in Rational Software Architect V7.0.0.x. Also, the LegacyProfileVersion is supported only for profiles without required stereotypes.

The plugin.xml file can also define a UML Profile located in another plug-in. In that case, the UMLProfile element has another attribute: bundle. The value of the bundle attribute is the name of the referenced plug-in, without the version information, as shown in Listing 5

Listing 5. The bundle attribute

```xml
<extension
   name="someProfileExtensionName2"
   point="com.ibm.xtools.uml.msl.UMLProfiles">

   <UMLProfile
      id="AnotherProfile"
      name="%AnotherProfile.name"
      path="profiles/AnotherProfile.epx"
      bundle=com.mycompany.myproduct.otherplugin
      required="false"
      visible="true">

   </UMLProfile>

</extension>
```

Using the New Plug-in Project wizard and Plug-in editor

The New Plug-in Project wizard and the Plug-in editor are Eclipse concepts, so for more details on this topic please see the Resources section. The only profile-specific information is the exact extensions for registering pathmaps and profiles, as described in the previous sections. However, for the sake of completeness, this section will provide a brief overview of how to create a plug-in project using a wizard, and then how to modify the plugin.xml file using the Plug-in editor.

To invoke the New Plug-in Project wizard, click File > New > Project and then select the Plug-in Project in the Plug-in Development group, as shown in Figure 8.

**Note:** To use the plug-in development capabilities, the Eclipse Plug-in Development capability has to be enabled. To do this:

1. Click Window > Preferences
2. Click Workbench > Capabilities
3. Click Eclipse Developer
4. Select the Eclipse Plug-in Development capability
The values for the fields in the wizard (Figure 9) depend on the intended functionality of the project. For example, if it is simply going to hold the profile, then you probably do not need Java support. If there is going to be some programmatic manipulation -- or if it is a plug-in that will have more artifacts than just profiles -- then you might need to create the Java project. It is up to you to determine the settings at this point.
The next step is to create a folder in the plug-in to hold profiles. If the naming convention is used, it would be the profiles folder of the plug-in's root directory. Of course, the next logical step is to either create the profile at that location or -- if the profile already exists in another project -- copy and move it.

To register a pathmap and the profile in the plugin.xml, you need to launch the Plug-in editor. Simply double-click the plugin.xml file and the editor will launch, providing plug-in details, as shown in Figure 10.
To work in the text mode, simply go to the plugin.xml and the build.properties tabs of the Plug-in editor. The syntax is the same as used in examples from the previous few sections. However, you may wish to take advantage of the more advanced features of the Plug-in editor.

The dependencies can be specified on the Dependencies tab. You can register the pathmap and the profile through the Extensions tab. Also, make sure on the Build tab that the folder containing the profile is included in the binary build.

Once the plug-in is ready for deployment, you can export it using Eclipse's plug-in exporter. To invoke the exporter shown in Figure 11:

1. Select the desired plug-in(s) in the Navigator view
2. Click File > Export
3. Select Plug-in Development > Deployable plug-ins and fragments entry from the list
Of course, the plug-in could be part of a bigger application that has its own way of packaging plug-ins.

### Open-source Eclipse UML2 format

Rational Software Modeler and Rational Software Architect also support the open-source Eclipse UML2 format to a certain degree. The profiles in the UML2 format (the file naming convention is `<profile_name>.profile.uml`) can be directly applied to models in all three tools (*.emx). You can use both file system deployment and plug-in deployment. Rational Software Modeler and Rational Software Architect also provide export and import functionality to and from the uml2 format.

For more details on the open source UML2 format, please see the Resources section.

### Export

To invoke the Export wizard, click File > Export > Other > UML 2.1 Model. The wizard -- shown in Figure 12 -- will create a `*.profile.uml` file in the specified location.

The important fields are **Recreate IDs** (should be turned off if you plan to maintain the profile in Rational Software Architect and re-export it later) and **Export applied profiles** (should be turned on if there any Rational Software Architect profiles applied to your profile).
**Note:** The ability to browse in a workspace is available only in Rational Software Modeler or Rational Systems Developer V7.0.5 and later; in older versions, you have to browse in the file system.

**Figure 12. Export wizard**

The import functionality is very similar to the export: it just works in the opposite direction. To invoke the Import wizard, click **File > Import > Other > UML 2.1 Model**. The wizard will create a *.profile.epx file in the specified location.

**Localization**

Profiles use the standard Eclipse mechanism for localization purposes. To localize the profile, you need to provide the localization file(s) in the same location as the profile file itself. The naming convention for the localization file is the same as Eclipse's:

<profile file name>[_<locale>].properties

For example, if you create the profile *MyProfile.epx*, the name of the default localization file would be *MyProfile.properties*. The name of the localization file in the French locale (as an example) would be *MyProfile_fr.properties*. 
Since there are strict rules on naming various profile artifacts, you can use the localization mechanism to provide more suitable display names. Those names could be stored in the default localization file, even if there is no plan to provide actual localization.

The following strings in a profile are localizable:

- Names of named elements (profile, stereotypes, classes, attributes, enumerations, and enumeration literals)

**Note:** the constraint names are not localizable

- Stereotype categories
- Constraint messages

The localization file consists of the ID-value pairs, using standard Java format. The ID for the name of a named element is the fully qualified name of that element, with the ::: (two colons) sequences replaced by double underscores (__). Any space characters should be escaped using the forward slash (/) character. The IDs for stereotype categories and constraint messages are those strings themselves.

Figure 13 shows an example.

**Figure 13. Example profile structure for localization**

The stereotype s1 has a category MyCategory. Its constraint has a message ID MyMsgId. Also, please note the space in the name of the enumeration literal (L 1). Listing 6 shows the contents of a corresponding localization file:
Listing 6. Localization file

```
#Profile: MyProfileA
#Fri May 27 13:03:55 EDT 2005
MyProfileA__E1=E1
MyProfileA__S1_e1=e1
MyProfileA__E1_L 1=L 1
MyProfileA__S1=S 1
MyCategory=My Category
MyProfileA__E1_L2=L2
MyProfileA=My Profile A
MyMsgId=My Message
```

Rational Software Architect can pre-populate the localization file. The action is invoked from the pop-up menu when the profile is selected (the **Localize** menu item). The pre-populated file will have proper IDs, and the values will be the same as the actual strings.

The action does not handle constraint message IDs: they will not be included in the pre-populated file.

**Programmatic support**

This section is intended for situations where you plan to provide some sort of (additional) programmatic manipulation in the model where the profile is applied, or even to apply the profile programatically. It does not address programatically creating and modifying profiles themselves; if you plan to do that, please refer to the open source documentation in the **Resources** section.

Rational Software Architect mostly does not publicly expose its API for profile and stereotype support. However, there are a few classes that are exposed that would make life easier when you are authoring profiles. Of course, since the tool is based on the open source Eclipse UML2 project, the entire open source API is freely available. The open source API has its own documentation, but this section will outline the main interfaces that could be useful to you.

To perform the programmatic manipulation, you have to deploy the code through a plug-in. The simplest structure would be to deploy both the profile(s) and the code in the same plug-in. Of course, the profile should still be registered using pathmaps for its location.

To use these APIs, the plug-in will need dependencies on various other plug-ins.

**Rational Software Architect Specific APIs: UMLModeler**

The **UMLModeler** class can be used to obtain the profile. It is provided by the `com.ibm.xtools.modeler.ui` plug-in.

The example in Listing 7 obtains the MyProfileA profile from the location specified by the MY_SAMPLE_PROFILES path-map. If the profile is not already loaded, this code forces loading (the Boolean parameter of the **getResource** method): assuming that the path provided in the URI constructor is right.
Listing 7. UMLModeler class

```java
import java.util.List;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.uml2.uml.Profile;
import com.ibm.xtools.modeler.ui.UMLModeler;
...
...
Profile profile;
URI uri = URI.createURI("pathmap://MY_SAMPLE_PROFILES/MyProfileA.epx");
Resource resource = UMLModeler.getEditingDomain().getResourceSet()
    .getResource(uri, true);
if (resource != null) {
    List contents = resource.getContents();
    if (contents.size() == 1 & contents.get(0) instanceof Profile) {
        profile = (Profile) contents.get(0);
    }
}
```

Please note that the `URI` class is from the `org.eclipse.emf.common.util` package.

Open Source API ## Eclipse EMF

Profile constraints implemented as Java classes have to extend the `org.eclipse.emf.validation.AbstractModelConstraint` class. It is provided by the `org.eclipse.emf.validation` plug-in.

The method to override is:

```java
public IStatus validate(IValidationContext ctx)
```

The argument (`ctx`) can be used to create the status object to return. The `IValidationContext` interface defines the `createFailureStatus` and `createSuccessStatus` methods.

Open Source API: Eclipse UML2

For the full documentation for the open source UML2 API please see the Resources section.

The following sections correspond to main interfaces that might be of interest when you are dealing programmatically with profiles. Each section then lists the main relevant methods. This section is not intended to be exhaustive documentation for the open source API, but rather to point you in the right direction in terms of the needed APIs.

All of the classes and interfaces listed in this section are provided by the `org.eclipse.uml2.uml` plug-in.
Listing 8. org.eclipse.uml2.uml.Package

```java
EList<EObject> applyProfile(Profile profile);
ProfileApplication createProfileApplication();
EList<Profile> getAllAppliedProfiles();
Profile getAppliedProfile(String qualifiedName);
Profile getAppliedProfile(String qualifiedName, boolean recurse);
EList<ProfileApplication> getAllProfileApplications();
ProfileApplication getProfileApplication(Profile profile);
ProfileApplication getProfileApplication(Profile profile, boolean recurse);
boolean isProfileApplied(Profile profile);
EList<EObject> unapplyProfile(Profile profile);
```

Listing 9. org.eclipse.uml2.uml.Element

```java
boolean addKeyword(String keyword);
EObject applyStereotype(Stereotype stereotype);
Stereotype getApplicableStereotype(String qualifiedName);
EList<Stereotype> getApplicableStereotypes();
Stereotype getAppliedStereotype(String qualifiedName);
EList<Stereotype> getAppliedStereotypes();
Stereotype getAppliedSubstereotype(Stereotype stereotype, String qualifiedName);
EList<Stereotype> getAppliedSubstereotypes(Stereotype stereotype);
EList<String> getKeywords();
Stereotype getRequiredStereotype(String qualifiedName);
EList<Stereotype> getRequiredStereotypes();
EObject getStereotypeApplication(Stereotype stereotype);
EList<EObject> getStereotypeApplications();
Object getValue(Stereotype stereotype, String propertyName);
boolean hasKeyword(String keyword);
boolean hasValue(Stereotype stereotype, String propertyName);
boolean isStereotypeApplicable(Stereotype stereotype);
boolean isStereotypeApplied(Stereotype stereotype);
boolean isStereotypeRequired(Stereotype stereotype);
boolean removeKeyword(String keyword);
void setValue(Stereotype stereotype, String propertyName, Object newValue);
EObject unapplyStereotype(Stereotype stereotype);
```

**Note:** This listing also contains keyword-related methods. Keywords are not really a profile concept, but they are often used as a lightweight method to mark up elements instead of stereotypes.

Listing 10. org.eclipse.uml2.uml.Profile

```java
EObject create(Classifier classifier);
ElementImport createMetaclassReference(PackageableElement importedElement);
PackageImport createMetamodelReference(org.eclipse.uml2.uml.Package importedPackage);
Stereotype createOwnedStereotype(String name);
Stereotype createOwnedStereotype(String name, boolean isAbstract);
EPackage define();
EPackage getDefinition();
ENamedElement getDefinition(NamedElement namedElement);
ElementImport getMetaclassReference(PackageableElement importedElement);
EList<ElementImport> getMetaclassReferences();
PackageImport getMetamodelReference(org.eclipse.uml2.uml.Package importedPackage);
EList<PackageImport> getMetamodelReferences();
EList<Extension> getOwnedExtensions(boolean requiredOnly);
Stereotype getOwnedStereotype(String name);
EList<Stereotype> getOwnedStereotypes();
EObject getReferencedMetaclass();
EList<org.eclipse.uml2.uml.Class> getReferencedMetaclasses();
EObject getReferencedMetamodels();
```

Listing 11. org.eclipse.uml2.uml.Stereotype

```java
Extension createExtension(org.eclipse.uml2.uml.Class metaclass, boolean isRequired);
Image createIcon();
EList<org.eclipse.uml2.uml.Class> getAllExtendedMetaclasses();
EClass getDefinition();
EList<org.eclipse.uml2.uml.Class> getExtendedMetaclasses();
EList<Image> getIcons();
Profile getProfile();
```

Listing 12. org.eclipse.uml2.uml.util.UMLUtil

```java
public static Element getBaseElement(EObject stereotypeApplication);
public static Profile getProfile(EPackage definition);
public static Profile getProfile(EPackage definition, EObject context);
public static Stereotype getStereotype(EObject stereotypeApplication);
```

Best Practices

The following is a list of best practices:

- Keep profiles and models in separate locations. If possible at all, keep them on a separate change management life cycle.
- Always use pathmaps, regardless of the deployment type (file system versus plug-in based).
- Profiles should be considered tool extensions, not modeling artifacts.
- Keep the profile size in check. Every time you save a profile, a new version is created in the file. Opening a profile file with 10 versions does much more than opening a profile with one version or a model. The number of versions in the profile can significantly affect the performance of the application. Use the **Compress** option to reduce the size of the profile. You can also release the profile from time to time, but be aware of the consequences: it creates a checkpoint that prevents certain changes, that version cannot be deleted, and so on. There are more details on this topic in the **Change management** section.
- Do not apply a non-released profile to a production model. Non-released profiles should be applied only to test models to test the profile itself. Once the profile is released, all non-released versions are deleted (except the last one which gets marked as released), resulting in the loss of data in models that were applied during migration.
- Do not give a non-released profile to your customers, unless you make it clear that such a profile can only be used for demonstration or test purposes, and that the migration to the official (released) version will result in the loss of data.
- When developing a plug-in profile, create a pathmap with the same name pointing to the same location in both the development (IDE) and the runtime (target) environment. Preferably, the pathmap should be defined as a user pathmap (through the Preferences) in the development environment and in the plugin.xml file of the plug-in for the runtime environment. That way, the models can be easily shared between the two environments.
- In Rational Software Modeler or Rational Systems Developer, you should use the **Generate Profile Tooling** option to create plug-in if you are deploying your profile that way.

Known issues

The following is a list of profile-specific known issues for the product versions specified in this article. It does not list other general issues in Rational Software Modeler and Rational Software
Architect, although they could (and often do) affect profiles as well. For those, please consult the release notes of the particular Rational Software Modeler and Rational Software Architect version that you are using.

- There could be migration problems between non-released versions.
- Multiple generalization relationships between the same elements (for example, stereotype $s_1$ specializing stereotype $s_2$ twice) would prevent the profile from being loaded in migration scenarios: the V7.0.5 profile loaded in V7.0.0.4 or V7.0.0.5. To address that issue, remove the extraneous generalization relationship, save and compress the profile to remove older broken profile versions. If the profile was released while it had such generalizations, then it will have to be edited by hand or using an EMF editor to remove the extraneous generalization from the released profile version.
- The tool will let you pick any file as a stereotype icon or image file. It is your (the profile author's) responsibility to ensure that it is indeed a valid and supported image file.

**Final word**

In this series, you have learned how to create and deploy UML profiles in Rational Software Architect.

UML profiles enable you to extend the UML meta-model for different purposes. In essence, that means the tool's behavior is extended as well. Rational Software Architect offers extensive UI to build and use profiles. They can range from simple profiles providing some additional presentation functionality to the complex profiles deployed through plug-ins, and with accompanying code taking advantage of the profile's meta-model extensions.

**Acknowledgments**

The author wishes to thank Christian Damus, Daniel Leroux, Michael Hanner, and Wayne Diu.
Related topics

• Subscribe to an IBM developerWorks newsletter.
• Download the trial version of IBM Rational Software Architect V7.
• Download IBM product evaluation versions.
• The official UML site: More information on the UML 2.0 specification can be found at the official UML here.
• Eclipse site: Go here for general information about Eclipse.
• Eclipse UML2 project: Visit here for information on the open source Eclipse UML2 project.

© Copyright IBM Corporation 2008
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)