CHAPTER 3

Model Editing with EMF.Edit

In the previous chapter we saw how EMF can take a model definition and produce a good, easily customizable Java implementation for it. Well, that's just the beginning. Once you decide to use EMF to model your application, you can then use the EMF.Edit framework to build very functional viewers and editors for the model. You can generate an editor that will display and edit (that is, copy, paste, drag-and-drop, and so on) instances of your model using standard JFace viewers and a property sheet, all with unlimited undo/redo. Alternatively, you can use the reflective support in EMF.Edit to do the same kinds of editing reflectively, even with a dynamic EMF model that you didn't generate.

You may be thinking that this is beginning to sound like one of those infomercials: If you buy EMF in the next 24 hours, we'll throw in the free viewers, drag-and-drop, and a bonus icon directory. Well, maybe it does, but the bottom line is that EMF.Edit is simply exploiting the information that is available in the model, along with the mechanisms provided in the EMF core, to provide more and higher-level function. The free function being offered comes naturally from the fact that we have a model, so you can rest assured that there is no "free juicer" about to be offered.

3.1 Displaying and Editing EMF Models

Let's return to the simple purchase order model we looked at in Chapter 2. Recall that it consisted of two simple classes with a containment association between them (Figure 3.1):

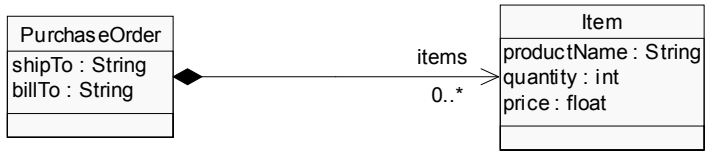

Figure 3.1 Purchase order model with containment association.

In the previous chapter we saw how the base framework of EMF gave you the ability to generate Java implementations for the model classes `PurchaseOrder` and `Item`, as well as other supporting classes. We also saw how using the generated classes along with the framework classes, `Resource` and `ResourceSet`, you could persist a purchase order instance, and its items. For example, assuming you used the framework default (XMI) serializer, a purchase order with two items could be serialized something like this:

```
<simplepo:PurchaseOrder xmi:version="2.0"
                        xmlns:xmi="http://www.omg.org/XMI"
                        xmlns:simplepo="http:///simplepo.ecore"
                        billTo="123 Maple Street">
  <items productName="Apples" quantity="12" price="0.5"/>
  <items productName="Oranges" quantity="24" price="0.3"/>
</simplepo:PurchaseOrder>
```

Let's assume you save this into a file, *My.po*, somewhere in your Eclipse workspace. The next thing you might like to do is display and edit it using a purchase order editor launched in the Eclipse workbench. You could then, for example, display the containment structure in a tree view and edit the attributes in a property sheet as shown in Figure 3.2. To really leverage the power of Eclipse, you would want to "integrate" the purchase order implementation with the Eclipse desktop this way.

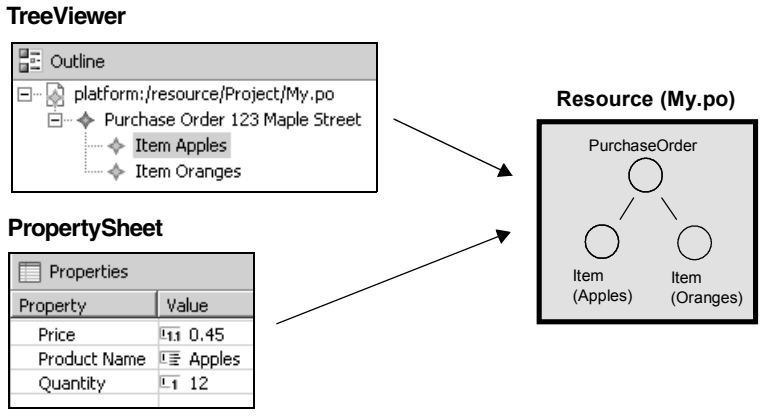

Figure 3.2 Outline and Properties views of a purchase order.

So, what would it take to display a model in a UI like this? To understand that, we first need to understand how Eclipse viewers work in general. The following section gives a brief overview of the Eclipse UI framework's viewer classes, property sheet, and action mechanism. If you already know how they work, you might want to skip it and go straight to Section 3.1.2 where we start to look at what EMF.Edit provides, to help you use the Eclipse framework to display and edit EMF models.

3.1.1 Eclipse UI Basics

Included in JFace, a part of the Eclipse user interface framework, is a set of reusable viewer classes (for example, `TreeViewer`) for displaying structured models. Instead of querying the model objects directly, the JFace viewers use a content provider to navigate the content and a label provider to retrieve the label text and icons for the objects being displayed. Each viewer class uses a content provider that implements a specific provider interface. For example, a `TreeViewer` uses a content provider that implements the interface `ITreeContentProvider`, as shown in Figure 3.3.

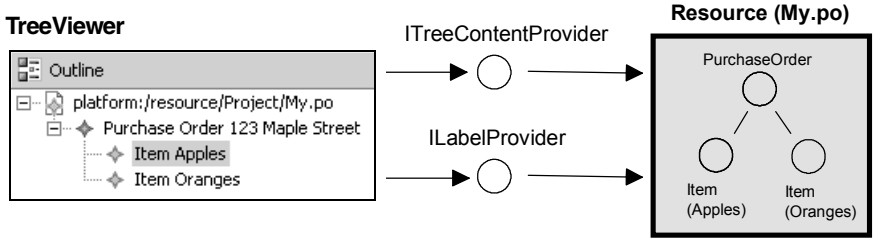

Figure 3.3 JFace viewer access to a model instance.

To display our purchase order resource in a `TreeViewer`, we start by providing the root object (the Resource object, "My.po", in this example) to the viewer. The viewer will respond by calling the `getText()` and `getImage()` methods on its label provider to retrieve the image and text, respectively. Next, the `TreeViewer` will call the `getChildren()` method on its content provider to retrieve the next level of objects to display in the tree (only the single purchase order in our example). This process of retrieving the text, icon, and children will then repeat for the rest of the tree.

In addition to class `TreeViewer`, JFace also includes `TableViewer` and `ListViewer` classes, which work the same way. They use a different content provider interface, `IStructuredContentProvider`, to retrieve their content. However, the `ITreeContentProvider` interface actually extends from

`IStructuredContentProvider`, so any tree content provider implementation class can also conveniently be used to support the other viewers as well.

Now that we understand how viewers are populated from a model, let's look at how a property sheet, which populates the Properties view, works. A property sheet gets the properties of an object by first calling the `getPropertySource()` method on its associated `IPropertySourceProvider`. Figure 3.4 shows how the property source provider produces an `IPropertySource` corresponding to a specific model object, for example, the "Apples" Item. Next, the property sheet calls the `getPropertyDescriptors()` method on the property source to get a list of `IPropertyDescriptors` for the object's properties ("Price", "Product Name", and "Quantity"). The `IPropertyDescriptor` interface is then used by the property sheet to display and edit the properties.

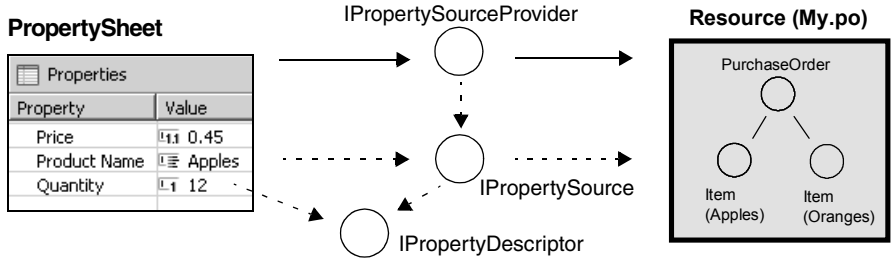

Figure 3.4 Eclipse property sheet.

There is one more important part of the Eclipse UI framework that we should look at: the action mechanism. Actions implement the `IAction` interface and represent the commands that can be run from menu items or toolbar buttons. When a menu item or toolbar button is selected, the framework invokes the associated action by calling its `run()` method. Actions also include methods to retrieve, among other things, the label and icon. For example, the `getText()` method is called by the framework to get the text for a menu item, when it is showing. The `getImageDescriptor()` method is used to get the icon to display both on menu items and toolbar buttons.

An action bar contributor is used to create and manage the actions for an editor. A subclass of `EditorActionBarContributor` is used to contribute the actions on behalf of its associated `EditorPart`. For example, to add a **New Purchase Order** item to the menu bar for a purchase order editor, `POEditor`, we would create an action bar contributor subclass, `POActionBarContributor` for example, and override the `contributeToMenu()` method to add (contribute) the new action. `POActionBarContributor` would be associated with `POEditor` in the workbench "registered editor" extension in the editor plug-in's manifest file.

3.1.2 EMF.Edit Support

Now that we know how views and actions work in Eclipse, we can look at how EMF.Edit helps you implement a UI for a model based on EMF.

To implement a tree viewer, like the one shown in the previous section, we need an implementation of the `ITreeContentProvider` interface that is capable of returning the children of EMF model objects. We also need an implementation of `ILabelProvider` to return a suitable text string for the label, usually based on one of the Ecore attributes. For a property sheet, we need a way of producing a set of `IPropertyDescriptors` for the (subset of) Ecore attributes and references that should be properties. EMF.Edit supports two ways of implementing these things, one using the reflective `EObject` API, and the other using generated classes.

The reflective approach consults the core model at runtime to provide a "best guess" implementation. For example, it implements the `getChildren()` method by returning the contained objects, `eContents()`, of the parent. For a label, it first tries to find a "name" attribute (case insensitive) in the class or failing that, one that includes the string "name" (for example, "productName"). For our simple purchase order example, the reflective implementation will produce pretty much what we want.

The second approach is to use the EMF.Edit generator to generate an implementation. The generated approach will, by default, produce the same behavior as the reflective implementation. This is not surprising, since the code is generated from the same core model that the reflective implementation uses at runtime. With the generated approach, however, you have an opportunity to influence some of the choices before the code is generated. For example, you could pick the "quantity" attribute for the label feature instead of default "productName".

Generating the implementation classes also results in a much more easily customizable solution. Just like the EMF model classes described in Chapter 2, you can modify and regenerate the EMF.Edit classes any way you like. The generated classes provide convenient override points for the most common types of customizations, unlike the reflective implementation, which would require a monolithic override with lots of `instanceof` checks.

EMF.Edit also includes support for Eclipse actions and model modification in general. Changing the state of an EMF model object, by running an action or using a property descriptor, is implemented (in EMF.Edit) by delegating to a command [5]. EMF.Edit includes generic implementations of a number of common commands, as well as framework support for customizing their behavior or for implementing your own specialized commands.

As shown in Figure 3.5, EMF.Edit is the bridge between the Eclipse UI framework and the EMF core framework.

Figure 3.5 EMF.Edit connects the Eclipse UI and EMF core frameworks.

A large amount of EMF.Edit function is actually independent of the UI. To support reuse of the UI-independent parts, the EMF.Edit framework is divided into two separate plug-ins:

1. **org.eclipse.emf.edit** is the low-level UI-independent portion.
2. **org.eclipse.emf.edit.ui** contains the Eclipse UI-dependent implementation classes.

Most of the real editing work is done in the UI-independent plug-in. The UI part handles the actual connection to the display, with an implementation tied to the Eclipse UI framework. As we'll see in the following two sections, the bulk of the work is actually done by delegating to two very important UI-independent mechanisms: item providers and commands.

3.2 Item Providers

Item providers are the single most important objects in EMF.Edit; they are used to adapt model objects so the model object can provide all of the interfaces that it needs to be viewed or edited. If you think back to Chapter 2, where we saw how EMF adapters can be used as both behavioral extensions and as change observers, you can see how adapters would be just right for implementing item providers. As behavioral extensions, they can adapt the model objects to implement whatever interfaces the editors and views need, and at the same time, as observers, they will be notified of state changes which they can then pass on to listening views.

Although item providers are usually EMF adapters, they are not always. An item provider that is "providing" for an EMF object will be an adapter, but other item providers may represent non-modeled objects, mixed in to a view with modeled items. This is an important feature of the EMF.Edit framework. It has been carefully designed to allow you to create views on EMF models that may be structurally different from the model itself (that is, views that suppress objects or include additional, non-modeled, objects). We'll look at this issue in Chapter 14, but for now you should simply think of item providers as adapters on EMF objects, but keep in mind that the framework is actually more flexible.

Their name, item provider, stems from the fact that they "provide" functions on behalf of individual editable model "items" (objects). As you'll see in the following sections, the EMF.Edit framework implements a delegation scheme whereby most functions involving model objects are ultimately implemented in their associated item provider. Consequently, item providers need to perform four major roles:

1. Implement content and label provider functions
2. Provide a property source (property descriptors) for EMF objects
3. Act as a command factory for commands on their associated model objects
4. Forward EMF model change notifications on to viewers

A given item provider can implement all of these functions or just a subset, depending on what editing functions are actually required. Most commonly, however, item providers will simply implement them all by subclassing the very functional EMF.Edit base class, `ItemProviderAdapter`. It implements most of the function generically, so a subclass (which may be generated, as we'll see in Section 3.4.1) only needs to implement a few methods to complete the job. EMF.Edit also provides a full function subclass, `ReflectiveItemProvider`, that implements all of the roles using the reflective `EObject` API. We'll talk about these and other implementation issues in Section 3.2.5, but first, the next four sections will describe each of the roles of an item provider.

3.2.1 Content and Label Item Providers

The first role of an item provider is to support the implementation of content and label providers for the viewers. In Section 3.1.1, we saw how Eclipse viewers use a content provider (for example, `ITreeContentProvider`) and a label provider (for example, `ILabelProvider`) to get the information they need from the model. EMF.Edit provides generic content and label provider implementation classes, `AdapterFactoryContentProvider` and `AdapterFactoryLabelProvider`, that delegate their implementation to item providers as shown in Figure 3.6. Both of these classes are constructed with an adapter factory (`POItemProviderAdapterFactory` in this example), which, like any other EMF adapter factory, serves to create and locate EMF adapters of a specific type (item providers for the purchase order model, in this case).

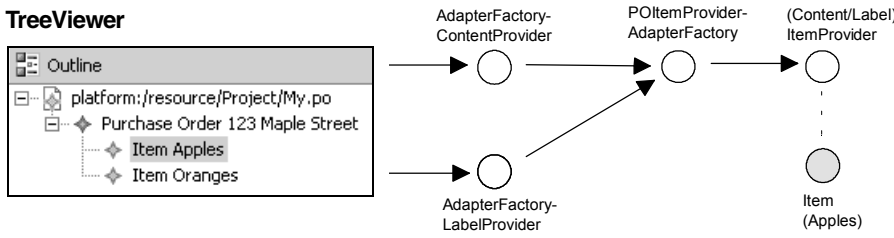

Figure 3.6 Content and label provider role of an item provider.

To service a request like `ITreeContentProvider.getChildren()`, for example, the `AdapterFactoryContentProvider` first calls `adapt()` on the `ItemProviderAdapterFactory`, which will create or return the `ItemProvider` (adapter) for the specified object. It then simply delegates to the `getChildren()` method of a corresponding item provider interface, `ITreeItemContentProvider`. The `getChildren`() method in `AdapterFactoryContentProvider` looks something like this:

```
public Object[] getChildren(Object object) {
  ITreeItemContentProvider adapter = (ITreeItemContentProvider)
    adapterFactory.adapt(object, ITreeItemContentProvider.class);
  return adapter.getChildren(object).toArray();
}
```

This same pattern is used for all of the content provider methods, and also by the `AdapterFactoryLabelProvider` to implement `ILabelProvider` methods (like `getText()`, for example). The adapter factory content and label providers do nothing more than simply delegate JFace provider methods to corresponding EMF content and label item provider mixin interfaces; there are four of them in total:

1. `ITreeItemContentProvider` is used to support content providers for `TreeViewer`s.
2. `IStructuredItemContentProvider` is used to support content providers for other structured viewers, such as `ListViewer`s and `TableViewer`s.
3. `ITableItemLabelProvider` is used to support label providers for `TableViewer`s.
4. `IItemLabelProvider` is used to support label providers for other structured viewers.

Notice the similarity of these interface names to those in JFace, only with the word "Item" added. The EMF.Edit interfaces are in fact very similar to the corresponding JFace ones. The main reason for having the parallel set of inter-

faces is to avoid any dependencies on JFace. Although item providers are primarily used to implement Eclipse (JFace-based) UIs, they are completely UI independent. In addition to their use in support of the JFace implementation classes, `AdapterFactoryContentProvider` and `AdapterFactoryLabelProvider`, they can also be used to implement views for other UI libraries (for example, Swing), or to implement non-UI, command-based utilities for EMF models.

3.2.2 Item Property Source

The second major role of an item provider is to act as a property source for the property sheet. Recall that the Eclipse `PropertySheet` uses an `IPropertySourceProvider` to request an `IPropertySource` for the object whose properties it wants to display and edit. In EMF.Edit, the `AdapterFactoryContentProvider` also implements the `IPropertySourceProvider` interface and is used to provide a property source to the property sheet, as shown in Figure 3.7.

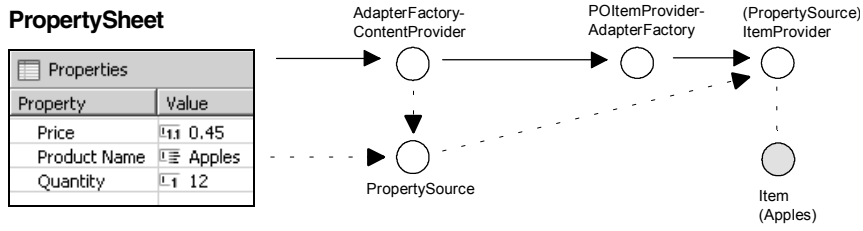

Figure 3.7 Property source role of an item provider.

Following the same pattern that we saw for `ITreeContentProvider` in the previous section, the `AdapterFactoryContentProvider` uses the adapter factory to locate an item provider, only this time one that implements the EMF.Edit item provider mixin interface `IItemPropertySource`. Again, just like the content and label provider interface, this EMF.Edit interface, `IItemPropertySource`, is very similar to its corresponding Eclipse interface, `IPropertySource`, only UI independent. Another EMF.Edit helper class, `PropertySource`, implements the actual `IPropertySource` interface needed by the property sheet. The `AdapterFactoryContentProvider` creates an instance of this class as a wrapper for the selected item provider (that is, the `IItemPropertySource`) and then returns it to the `PropertySheet`.

This same wrapping pattern is used again when the property sheet calls the `IPropertySource.getPropertyDescriptors()` method. Class `PropertySource` services the request by delegating to the `getPropertyDescriptors()`

method on the item provider, which returns a set of `IItemPropertyDescriptors`. `PropertySource` then constructs an `IPropertyDescriptor` wrapper class, `PropertyDescriptor`, for the item property descriptor and then returns it to the property sheet. The complete picture is shown in Figure 3.8.

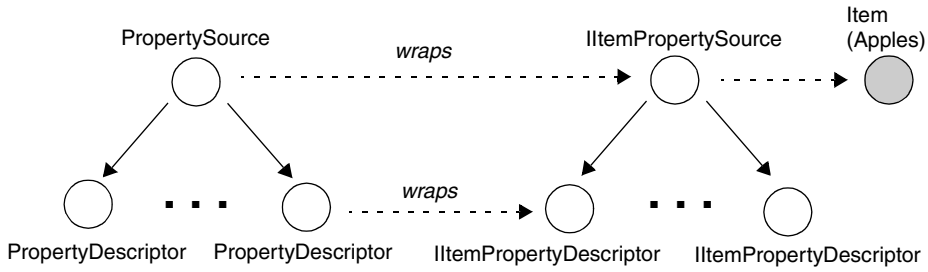

Figure 3.8 `PropertySource` and `PropertyDescriptor` delegation.

With this arrangement, property descriptor calls are now delegated to their "Item" equivalents. For example, if a property value is changed in the property sheet, the `PropertySource.setPropertyValue()` method will be called. The property source will then simply delegate to the `setPropertyValue()` method on the `ItemPropertyDescriptor`, which will make the actual model change.

If you're thinking that all this creating of wrapper objects seems fairly messy, you're right, but that's the price we have to pay to keep the item providers UI independent. The good news is that the property descriptors are in fact the worst, and the last, of this double-object pattern.

3.2.3 Command Factory

Item providers act as the factory for commands involving their adapted objects. In this role, item providers play a critical part in the EMF.Edit command framework, which we'll look at in Section 3.3. For now, let's just say that EMF.Edit provides all the mechanisms for modifying EMF objects in an undoable way, including a full set of generic commands. The framework makes it easy to tune the command behavior for specific models by delegating their creation to item providers.

Similar to the way the eclipse UI framework uses "provider" interfaces (for example, `ITreeContentProvider`) to access the model, the EMF.Edit command framework also has an interface, `EditingDomain`, which it uses for the same purpose. Also, just like the content and label providers, a delegating implementation class, `AdapterFactoryEditingDomain`, is used to implement it. As shown in Figure 3.9, `AdapterFactoryEditingDomain` works the same

as the other `AdapterFactory` implementation classes, only it delegates its methods to an item provider supporting the editing domain item provider mixin interface `IEditingDomainItemProvider`. We'll look at editing domains and the role of item providers in their implementation in Section 3.3.3.

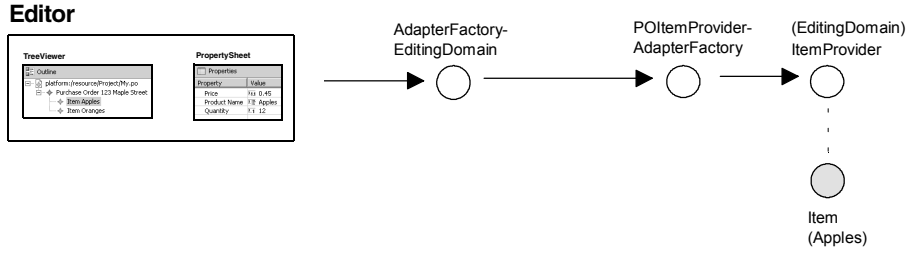

Figure 3.9 Command creation role of an item provider.

3.2.4 Change Notification

Being a standard EMF adapter, an `ItemProvider` will be notified, with a call to its `notifyChanged()` method, whenever an object that it is adapting changes state. The `ItemProvider`'s responsibility as an observer is to optionally filter uninteresting events and then to pass the remaining ones on to the central change notifier for the model, usually the `ItemProviderAdapterFactory`. The `ItemProviderAdapterFactory` implements the EMF.Edit interface `IChangeNotifier`, which allows views and other interested parties to register as listeners of the model as a whole. The design is illustrated in Figure 3.10.

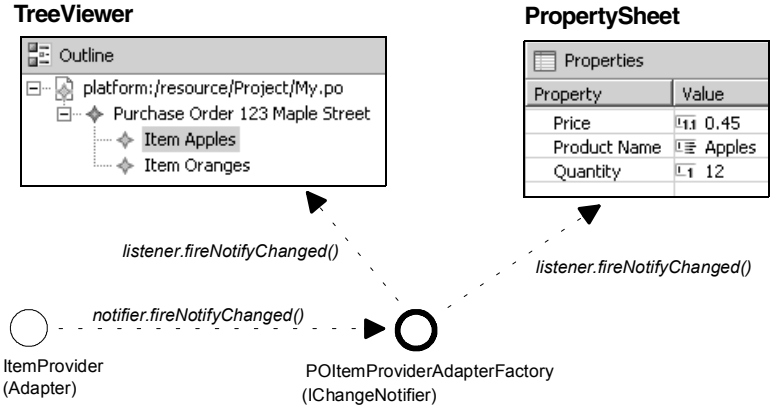

Figure 3.10 Notification flow from an `ItemProvider` to the views.

The diagram shows the flow of notification after a purchase order `Item` changes state in some way (for example, the **productName** attribute was just set to "Apples"). In its `notifyChanged()` method, the `ItemItemProvider` (that is the `ItemProvider` for the model class `Item`) passes on the change notification by calling the `fireNotifyChanged()` method on the `IChangeNotifier` (the `POItemProviderAdapterFactory` in our example), which in turn calls `fireNotifyChanged()` on all its registered listeners. In our example, the listeners are a `TreeViewer` and the `PropertySheet`, which would now be updated to reflect the model change.

That's all there is to it, but there is one more thing worth mentioning. When notifying a JFace viewer, as in this example, the viewer itself is not the actual listener. Instead, the content provider associated with the viewer registers on behalf of its viewer. As shown in Figure 3.11, it responds to the `fireNotifyChanged()` call by calling the appropriate update method(s) on the viewer. This is another function that is handled automatically for you by the EMF.Edit class `AdapterFactoryContentProvider`. It knows how to most efficiently update all of the standard JFace viewers.

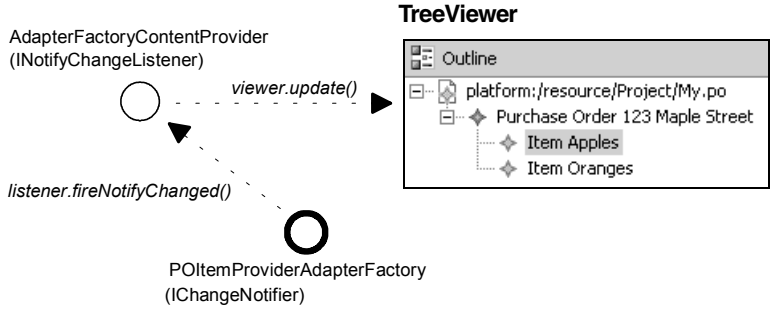

Figure 3.11 `AdapterFactoryContentProvider` updating its associated viewer.

3.2.5 Item Provider Implementation Classes

Now that we understand the various roles that item providers play, the question of how to implement them comes up. The EMF.Edit framework provides lots of flexibility as far as this is concerned, ranging from using a single generic (reflective) item provider to having generated type-specific item providers for every class in the model.

Reflective Item Provider

In Chapter 2, we showed how Java code does not necessarily need to be generated for an EMF model. If you choose to not generate implementation

classes for your model, you can still create and manipulate instances of your classes using dynamic EMF and the reflective `EObject` API. The same is true for the editing support in EMF.Edit.

You can decide to generate or to use a reflective implementation for your item providers. In fact, you can choose to use reflective item providers for some of your classes, and generated ones for others. The important thing to consider is how much customization you plan to do.

Class `ReflectiveItemProvider` is the EMF.Edit generic item provider implementation. As you might imagine, it implements essentially the same behavior as generated item providers with default generator settings. Using it is simple, but not easily customizable. Any customization you might want to do will involve `instanceof` checks. If you use typed item provider classes whose inheritance hierarchy mirrors the model's instead, you will be able to specialize the implementation in a clean, object-oriented way.

Typed Item Providers

When using the typed item providers, there are two possible patterns:

1. **The Stateful pattern**, where every object in a model (instance) has a one-to-one correspondence with its item provider. Each item provider has a target pointer to the one-and-only object that it is adapting. This pattern doubles the number of objects in the application, and therefore only makes sense in applications where the instances are needed to carry additional state. That's why we call it the Stateful pattern.
2. **The Singleton pattern** avoids most of the extra objects. With this pattern, we share a single item provider adapter for all objects of the same type. In this case, like the reflective case, each item provider has many targets.

To allow more than one object to share an item provider (that is, for the Singleton or reflective cases), an extra argument (the object) is passed to every item provider interface method. For example, the `getChildren()` method in the `ITreeItemProvider` interface looks like this:

```
public Collection getChildren(Object object);
```

In the Stateful case, this object will always be the same as the adapter's target, so a Stateful item provider implementation could choose to ignore this argument and use the adapter target to access the model object instead. However, doing the opposite (using the argument instead of the target) is a better approach since it leaves you the option of making the item provider a Singleton in the future, without having to change it.

3.3 Command Framework

In the previous sections, we've seen how EMF models can be viewed using content and label providers, but we haven't talked much about how to change, or edit, a model. Another very important feature of the EMF.Edit framework is its support for command-based editing of an EMF model, including fully automatic undo and redo.

The command framework in EMF is divided into two parts, the common command framework and the EMF.Edit commands. The common framework defines basic command interfaces and provides some implementation classes like a basic command stack, a compound command for composing commands from other commands, and other convenient command implementations. The commands in the common framework are very general and can be used independently of EMF.Edit. In fact, they don't even depend on EMF modeled objects (that is, `EObjects`); they're completely independent of EMF.

The EMF.Edit Commands, on the other hand, are implementation classes specifically for editing `EObjects`. EMF.Edit includes a full set of generic command implementation classes that, using the reflective `EObject` API, provide support for setting attributes, adding and removing references, copying objects, and other kinds of modifications to a model. In the next two sections we'll look at some of the most interesting commands provided by the framework. Then, in Section 3.3.3, we'll look at the rest of the infrastructure in the command framework, and how it allows you to easily customize a command's behavior for your specific model.

3.3.1 Common Command Framework

The common command framework includes the basic interfaces and implementation classes with which model-change commands can be built and executed. Although the main use of them is in EMF.Edit, they are completely general purpose (that is, they work with `java.lang.Objects` as opposed to `EObjects`) and therefore can be used with any model, EMF or not. The framework consists of the following classes and interfaces.

Command

Interface `Command` is the base interface that all commands must implement. As you might expect, it includes methods like `execute()`, `undo()`, and `redo()` to cause and reverse the command's effect. A command is tested for executability by calling the `canExecute()` method, which is often used to control the enablement of actions bound to the command. Not all commands can be undone; sometimes it is just too hard to maintain all the information needed

to reverse the state changes caused by a command. The method `canUndo()` is used to check for undoability of a command. Returning `false` from `canUndo()` indicates that the `undo()` and `redo()` methods are unimplemented.

The command interface includes a few more methods, two of which are particularly interesting: `getResult()` and `getAffectedObjects()`. Implementing these two methods is optional, but they can be quite useful. The `getResult()` method is used by a command implementation to return what should be considered the result of its execution. This allows one to implement compound commands where the result of one command can be the input of another. For example, if a generic object copy command returns the copy as its result, then a copy-and-paste command could simply be composed from the copy command and a generic add command; the add command's input would be the copy command's result.

The second method, `getAffectedObjects()`, is used to return the objects that have been changed during the last `execute()`, `undo()`, or `redo()` call. The EMF.Edit UI framework uses the affected objects to control the UI selection to highlight the effect of the command. The `getAffectedObjects()` method often returns the same thing as `getResult()`, but not always.

AbstractCommand

Class `AbstractCommand` is a convenient partial implementation of the `Command` interface that most commands extend from. It's a small class that doesn't really do a lot; non-trivial commands need to override most of the methods anyway. However, it does provide an important implementation of the `canExecute()` method, which calls out to another protected method, `prepare()`, like this:

```
public boolean canExecute() {
  if (!isPrepared) {
    isExecutable = prepare();
    isPrepared = true;
  }
  return isExecutable;
}
```

Notice that the `prepare()` method will be called only once, regardless of how often `canExecute()` is called. This is particularly significant if the enablement checking of the actual command (subclass) involves a lot of computation. With this design, all a subclass needs to do differently is override and put the enablement checking code in the `prepare()` method, instead of `canExecute()`.

CommandStack

Interface `CommandStack` defines the interface for executing and maintaining commands in an undoable stack. Like `Command` itself, it includes methods to `execute()`, `undo()`, and `redo()` a command, only here it also maintains the command on the stack. The `canUndo()` and `canRedo()` method can be used to determine if there are any commands on the stack to undo or redo, respectively. Other methods are provided to access the command at the top of the stack or the next command to undo or redo, to flush the stack, and to add change listeners (interface `CommandStackListener`) to the stack.

BasicCommandStack

Class `BasicCommandStack` is a basic implementation of the `CommandStack` interface. It is fully functional and can be used, as is, as the command stack for an EMF.Edit-based editor. One interesting observation relates to non-undoable commands. It can be seen when looking at the implementation of the `canUndo()` method in `BasicCommandStack`:

```
public boolean canUndo() {
  return top != -1 && ((Command)commandList.get(top)).canUndo();
}
```

Notice that the command stack will not only return `false` from the `canUndo()` method when there are no commands to undo (that is, `top == -1`), but also if the last executed command cannot be undone (that is, `(...commandList.get(top)).canUndo()` returns `false`). This is a very important observation in that it implies that if a non-undoable command is executed, the entire stack of commands before it will no longer be undoable either. The undo list is effectively wiped out at this point, so it's important to consider this before executing a non-undoable command on the command stack.

CompoundCommand

Class `CompoundCommand` is probably the single most commonly used command in the framework. It's a very useful class that allows you to build higher-level commands by composing them from other more basic commands. The `execute()` method simply calls `execute()`, in order, on each of the commands it's composed from; `canExecute()` returns `true` if all the commands can execute; and so on.

In addition to delegating its implementation to the composed commands, `CompoundCommand` provides a few useful convenience methods. For example, the `appendAndExecute()` method can be used to append a command and

immediately execute it. This is a particularly good way of recording a set of executed commands, which can later be undone by simply calling undo on the compound command. This technique is often used to conditionally execute commands in the `execute()` method of another command. Chapter 14 includes an example of this.

Another useful convenience method is `unwrap()`, which returns the underlying (composed) command if there is only one, or itself (that is, `this`) otherwise. It allows you to optimize away compound commands that only have one command under them.

Other Common Commands

In addition to the classes we've just described, there are a few more common command implementation classes that are less commonly used but nevertheless quite handy. We won't describe them here, but Chapter 15 includes an overview of the whole set of common command classes.

3.3.2 EMF.Edit Commands

EMF.Edit includes a set of generic commands for modifying EMF models. Its commands extend from and build on the interfaces defined in the common command component but impose a dependency on the Ecore model; they use the reflective `EObject` API to operate on EMF modeled objects. The following basic commands are provided:

1. `SetCommand` sets the value of an attribute or reference on an `EObject`.
2. `AddCommand` adds one or more objects to a multiplicity-many feature of an `EObject`.
3. `RemoveCommand` removes one or more objects from a multiplicity-many feature of an `EObject`.
4. `MoveCommand` moves an object within a multiplicity-many feature of an `EObject`.
5. `ReplaceCommand` replaces an object in a multiplicity-many feature of an `EObject`.
6. `CopyCommand` performs a deep copy of one or more `EObjects`.

These commands work on any EMF model, and their implementations fully support undo and redo. All of these commands, except `CopyCommand`, are primitive commands that simply perform their function when called. A `CopyCommand`, however, is actually composed from instances of two other special-purpose primitive commands, `CreateCopyCommand` and `InitializeCopyCommand`, which create and initialize a shallow copy object,

respectively. The `CopyCommand` works by building up `CompoundCommand`s composed of `CreateCopyCommand`s and `InitializeCopyCommand`s for the individual objects that need to be copied. It then invokes the compound commands to perform the deep copy. The reason for this approach is to allow easy customization of any part of the copy operation.

In addition to the basic commands, the EMF.Edit command framework includes some higher-level commands that are built using the basic commands along with some of the classes from the common command component:

1. `CreateChildCommand` allows you to create a new object and add it to a feature of an `EObject`. It uses an `AddCommand` or `SetCommand` to add the child, depending on whether the feature is multiplicity-many or not.
2. `CutToClipboardCommand` invokes a `RemoveCommand`, and then saves the removed object on the clipboard.
3. `CopyToClipboardCommand` simply saves a pointer on the clipboard; it doesn't actually change the model.
4. `PasteFromClipboardCommand` uses a `CopyCommand` to copy the object on the clipboard and then an `AddCommand` to add the copy to the target location.
5. `DragAndDropCommand` uses `CopyCommand`, `RemoveCommand`, and `AddCommand` to implement standard drag-and-drop operations.

That's it for the predefined commands, but there are a couple of other interesting features of the EMF.Edit command package that we should point out. The first thing has to do with overrideability of the commands.

AbstractOverrideableCommand

Most of the generic commands extend from an EMF.Edit abstract base class, `AbstractOverrideableCommand`, which is a subclass of the common `AbstractCommand`. The EMF.Edit base class adds the ability to attach another command to override it (via delegation). For example, the `execute()` method looks like this:

```
public final void execute() {
  if (overrideCommand != null)
    overrideCommand.execute();
  else
    doExecute();
}
```

If an `overrideCommand` is attached, the `execute()` method is delegated to it, otherwise the `doExecute()` method is called. This pattern is used for all of the `Command` methods.

You may be wondering why this is needed, given that you can always override a command simply by subclassing it. The key word is "you." The EMF.Edit framework expects you to use ordinary subclassing to customize the generic commands with any model-specific specializations you may need. At the same time, the framework wants to reserve an orthogonal dimension of overrideability for itself.

The implication of this is that if you want to subclass an EMF.Edit overrideable command, you need to override the `doExecute()` method instead of `execute()`, `doUndo()` instead of `undo()`, and so on. Other than that, you typically shouldn't need to concern yourself with the `OverrideableCommand` mechanism.

CommandParameter and Static Create Methods

Another special feature of the EMF.Edit command package has to do with the creation of commands. We already mentioned in Section 3.2.3 that EMF.Edit commands are created using an `EditingDomain`, which, in turn, delegates to item providers. The `EditingDomain` interface contains (among other things) a command factory method, `createCommand()`, that looks like this:

```
Command createCommand(Class commandClass,
                      CommandParameter commandParameter);
```

Notice that class `CommandParameter` is used to pass the command arguments in a generic way. To use this method to create a command you would first need to create a `CommandParameter` object, set the command's parameters into it, and then call the create method, passing it the command class (for example, `SetCommand.class`) and the parameters.

Rather then making clients go through all that, the EMF.Edit command framework uses a convention of providing static convenience `create()` methods in every command class. Using the static method, you can create and execute a `SetCommand` like this:

```
Command cmd = SetCommand.create(ed, object, feature, value);
```

The static `create()` method will, in turn, create the `CommandParameter` object and call the `createCommand()` method on the `EditingDomain` for you.

3.3.3 EditingDomain

Similar to the way that content and label providers are used to manage viewer access to a model, the EMF.Edit framework uses an editing domain to manage

an editor's command-based modification of a model. It does this by providing three main functions:

1. Creating commands, optionally deducing some of their arguments
2. Managing the command undo stack
3. Providing convenient access to the set of EMF resources being edited

In Section 3.2.3 we saw that EMF.Edit's editing domain implementation class, `AdapterFactoryEditingDomain`, implements the first function, command creation, by delegating to an item provider. The second and third functions are handled by maintaining the editor's `CommandStack` and `ResourceSet`, respectively. The three roles are illustrated in Figure 3.12.

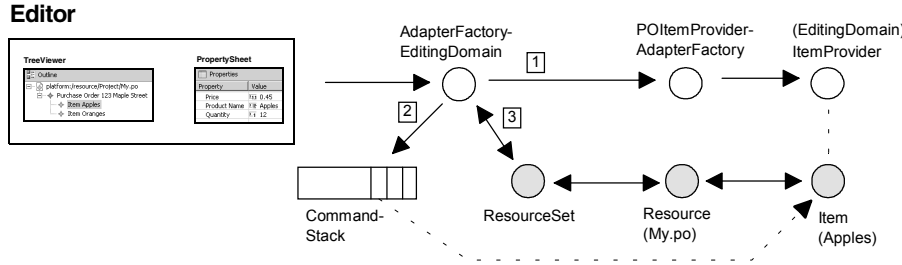

Figure 3.12 The roles of an `EditingDomain`.

Creating Commands

To understand how an editing domain handles its command creation role, let's walk through a simple example. Assume we want to remove one of the items from our purchase order model, *My.po*. The `RemoveCommand`, like most EMF.Edit commands, includes several static `create()` methods. Two of them look like this:

```
public static Command create(EditingDomain domain,
                             Object owner,
                             Object feature,
                             Object value) { ...

public static Command create(EditingDomain domain,
                             Object value) { ...
```

Notice that the first one takes three arguments in addition to the `EditingDomain`: the `owner` and `feature` being removed from, and the `value` being removed. For our example we would call it like this:

```
Command cmd = RemoveCommand.create(editingDomain,
                                   aPurchaseOrder,
                                   poPackage.getPurchaseOrder_Items(),
                                   aItem);
```

In this example, we provided all the information needed to create the `RemoveCommand`. This would not be the case if we used the second `create()` method instead:

```
Command cmd = RemoveCommand.create(editingDomain, aItem);
```

Notice that here we simply pass the item to be removed, `aItem`, without specifying where to remove it from. This is actually the way an EMF.Edit-based editor implements a **Delete** action; it simply passes the selected object, or objects, to the `RemoveCommand`'s `create()` method. The editing domain is given the added responsibility of filling in the missing arguments. Let's follow through with our example.

As we described in the previous section, the static `create()` methods simply package up their arguments in a `CommandParameter` and then call `createCommand()` on an editing domain. The `AdapterFactoryEditingDomain` simply delegates `createCommand()` to an item provider using the familiar adapter factory delegation pattern:

```
public Command createCommand(Class commandClass,
                             CommandParameter commandParameter) {
  Object owner = ... // get the owner object for the command
  IEditingDomainItemProvider adapter =
    (IEditingDomainItemProvider)
      adapterFactory.adapt(owner, IEditingDomainItemProvider.class);
  return adapter.createCommand(owner, this, commandClass,
                               commandParameter);
}
```

Notice that the `createCommand()` method uses the `owner` object to access the item provider to delegate to (that is, for the object used in the `adapterFactory.adapt()` call). If we used the four-argument `create()` method, then the `owner` is known (`aPurchaseOrder`). However, if we used the two-argument method, then the editing domain will need to compute it.

If you look at the actual implementation, the way `AdapterFactoryEditingDomain` finds the `owner` looks more complicated than it is. This is because it's designed to handle, among other things, removing collections of objects at once. For example, a user might select multiple items from more than one purchase order, and then invoke a **Delete** action. The editing domain handles this case by computing all the owners involved, and then creating a `CompoundCommand` containing a `RemoveCommand` for each owner.

For any given object (to be removed), the owner is computed by calling the `getParent()` method on its item provider, another method in the `IEditingDomainItemProvider` interface.[1] The effect of this is that the method `createCommand()` is finally called on the item provider of the purchase order (`aPurchaseOrder`).

In Section 3.2, we mentioned that most item providers that are also EMF adapters extend from the EMF.Edit convenience base class, `ItemProviderAdapter`, which provides a default implementation of many methods. Included among these is an implementation of `createCommand()` that handles all the standard commands provided by the EMF.Edit framework. For our example, the item provider will first deduce the final argument, the feature (**items**), and then simply return a new `RemoveCommand` constructed with all four arguments. In Chapter 14, we'll look in detail at how this works and how you can easily override the generic commands with your own model-specific customizations.

Maintaining the Command Stack

The command stack plays a key role in an EMF.Edit-based editor. If a single command stack is used pervasively for all changes to the model, editors can also use it to enable the **Save** action (that is, only enable it when the stack is not empty), and to both enable and execute the **Undo** and **Redo** actions.

In general, commands are created and executed in the same place. Since the editing domain serves as the command factory, it would also be an excellent holder for the command stack. Having created the command, the editing domain can then be used to access the command stack to execute it.

So, the only thing that we need to ensure is that the editing domain is available everywhere in the code that needs to change the model (that is, execute commands). For editor actions, the editing domain is readily available from the editor. For property descriptors, the editing domain needs to be located using the resource set, which brings us to the third role of the editing domain.

Accessing the ResourceSet

In its third role, the adapter factory provides convenience methods to load and save resources, as well as convenient access to the resource set. However, the real reason it provides these friendly services is so it can create a special

1. Actually, there is a `getParent()` method with the same signature in the `ITreeItemContentProvider` interface, which works out fine since we usually have the same item provider implementing both interfaces.

`ResourceSet` that knows about it—one that implements the interface `IEditingDomainProvider`. This is indicated in Figure 3.12 by the arrow on both ends of the line between the editing domain and resource set.

Since a resource is aware of its resource set, and an object is aware of its resource, with this arrangement we can now find the editing domain for any model object. This is important, in that it allows commands to be executed on model objects from anywhere in the code. For example, when a property sheet change is made, the `ItemPropertyDescriptor setPropertyValue()` method will locate the editing domain from the object being changed, like this:

```
EditingDomain editingDomain = getEditingDomain(object);
```

Once it has access to the editing domain, it can then create the command and, since it also now has access to the command stack, execute it:

```
editingDomain.getCommandStack().execute(
  SetCommand.create(editingDomain, object, feature, value));
```

3.4 Generating EMF.Edit Code

In Chapter 2, we saw how EMF lets you take a model definition in any of several forms and generate Java implementation code for it. Given the same model definition, we can also use the EMF.Edit code generation support to generate item providers and other classes needed to edit the model. The EMF.Edit code generator is not a separate tool; it's just another feature of the model generator. As you'll see in Chapter 4, after generating your model, you can generate the EMF.Edit parts via the **Generate Edit Code** and **Generate Editor Code** menu items.

When they are needed to hold generated code, the EMF generator will create new projects. We have seen that, by default, model code is generated into the existing project that contains the core and generator models. However, this is not the case for generated EMF.Edit code. As described in Section 3.1.2, the EMF.Edit framework is divided into two separate plug-ins: the UI independent part and the Eclipse UI-dependent part. By default, the code generated by EMF.Edit follows this same pattern. **Generate Edit Code** will generate a plug-in containing the UI independent editing support classes, while **Generate Editor Code** will generate the rest into a separate plug-in that also depends on the Eclipse UI. You can, however, override this and force the generator to put everything into a single plug-in, if that's what you want.

3.4.1 Edit Generation

Invoking **Generate Edit Code** in the EMF generator will generate a complete plug-in containing the UI independent portion of a model editor. It produces the following:

1. A set of typed item provider classes, one for each class in the model.
2. An item provider adapter factory class that creates the generated item providers. It extends from the model-generated adapter factory base class described in Chapter 2.
3. A `Plugin` class that includes methods for locating the plug-in's resource strings and icons.
4. A plug-in manifest file, *plugin.xml*, specifying the required dependencies.
5. A property file, *plugin.properties*, containing the externalized strings needed by the generated classes and the framework.
6. A directory of icons, one for each model class.

The most important of these is the set of item provider implementation classes. As described in Section 3.2.5, there are two possible item provider patterns: Stateful or Singleton. The generator gives you the option on a class-by-class basis to generate a Stateful, a Singleton, or no item provider. The chosen pattern only affects the generated `create()` method in the adapter factory and not the generated item provider itself. The generated item providers are implemented using the pattern-neutral approach described in Section 3.2.5.

Choosing not to generate an item provider for a class would be appropriate if you plan to never display instances of it or if you don't need to customize it and can therefore use the EMF.Edit reflective item provider, class `ReflectiveItemProvider`, for it.

The generated item providers mix in all the interfaces needed for basic support of the standard viewers, commands, and the property sheet:

```
public class PurchaseOrderItemProvider extends ItemProviderAdapter
  implements
    IEditingDomainItemProvider,
    IStructuredItemContentProvider,
    ITreeItemContentProvider,
    IItemLabelProvider,
    IItemPropertySource { ...
```

It extends, either directly or indirectly, from the EMF.Edit item provider base class, `ItemProviderAdapter`.

If you look at a generated item provider class, you'll notice that most of the methods from the item provider interfaces are actually implemented in the

base class generically, or by calling out to a few simple methods that are implemented in the item provider subclasses. We'll cover these methods, along with all the EMF.Edit generator patterns, in detail in Chapter 10.

3.4.2 Editor Generation

Generate Editor Code is used to generate a fully functional editor plug-in that will allow you to view instances of the model using several common viewers and to add, remove, cut, copy, and paste model objects, or to modify the objects in a standard property sheet, all with full undo/redo support. The following is generated in the editor plug-in:

- An integrated Eclipse workbench editor
- A wizard for creating new model instance documents
- An action bar contributor that manages the popup menus, and toolbar and menu bar items
- A `Plugin` class that includes methods for locating the plug-in's resource strings and icons
- A plug-in manifest file, *plugin.xml*, that specifies the required dependencies and extensions of the editor, wizard, and action workbench extension points
- A property file, *plugin.properties*, containing the externalized strings needed by the generated classes and the framework
- A directory containing icons for the editor and model wizard

The generated editor is a very functional multipage editor. The Outline view displays the model file in a tree viewer. Each page of the editor is synchronized with it and demonstrates a different way of displaying the model. The following pages are generated by default:

- **Selection** shows a tree viewer similar to the one in the Outline view.
- **Parent** is an inverted tree showing the container path from the element selected in the Outline view back to the root.
- **List** shows a list viewer containing the children of the selection in the Outline view.
- **Tree** shows another tree viewer, only rooted at the current selection.
- **Table** shows a table viewer containing the children of the current selection.
- **TableTree** is the same, only using a table tree viewer.

The generated wizard allows you to create a new model instance document containing a single root object of one of the model's types. The generated default implementation provides a drop-down list of concrete classes in the model, from which the user selects an appropriate root.

3.4.3 Regenerating EMF.Edit Plug-Ins

When regenerating into existing projects, the EMF generator supports the same kind of merging for EMF.Edit code as it does for model code, which we described in Chapter 2. You can edit the generated classes to add methods and instance variables, or to modify the generated ones. As long as you remove the `@generated` tags from any generated methods that you change, your modifications will be preserved during the regeneration.

As shown in the last two sections, EMF.Edit also generates three types of non-Java content: property files, icons, and manifest files. Generated property files contain the translated text strings (resources) referenced by the generated code. You can manually add new resource strings or edit the generated ones and then later regenerate without losing your changes. Any newly generated strings will be added, but unused ones, whether initially generated or not, will never be removed. You will need to manually remove them.

Every icon generated by EMF.Edit is a uniquely colored version of a generic icon. The generated icons are expected to be replaced by properly designed model-specific ones, and therefore, the generator will never overwrite an existing icon.

The generator also never overwrites manifest files. There is no automatic merge support either, because it's rarely needed. If you've change a generated manifest file by hand (for example, to add a new extension point), and then later change the model in a way that affects the generated *plugin.xml*, then you should rename it (to *plugin.tmp* for example), run the generator to produce the new *plugin.xml*, and then manually merge your changes into the newly generated version.