A guide to FOMOD scripts (in XML)

FOMOD scripts automate unpacking a mod through FOMM or the Nexus Mod Manager. They can display a menu to give choice over optional files, and make sure requirements (e.g. for DLC or other mods) are followed.

There are two languages in which you can a FOMOD script...

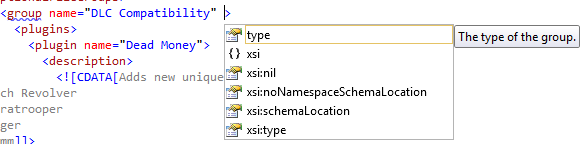
* C#: Writing a script in C# gives a lot of control over what the script does (in addition to unpacking mods, you can edit .ini files, and mods can be unpacked to a specific place in the load order) and the appearance of any menus. However, these scripts are more difficult to write. We won’t cover them here, but there is a nice guide to this sort of script here: <http://zumbs.wordpress.com/2009/11/08/fomm-and-fomods-for-dummies-3/>
* XML: Kaburke (the maker of FOMM) has made a simple and quite powerful system to set up a script with an XML file. This can do most things you would want, except editing .ini files and controlling load order. This guide is intended to explain how to make one.

To make this work, you need to have a ‘fomod’ folder in the root of the mod archive, and inside that, you need a file called ModuleConfig.xml. This can be done automatically inside FOMM if you make your FOMOD archive with the ‘Create FOMod’ button, or if you want to add a script to an existing FOMOD archive, using the ‘Create script’ button (both in the Package Manager).

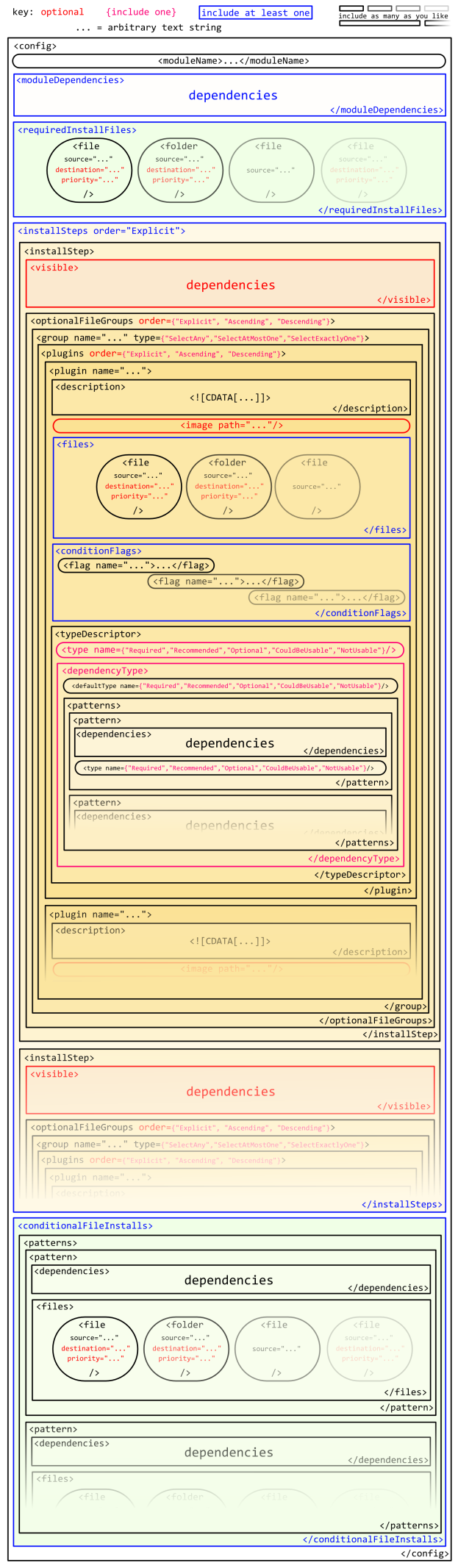
FOMM isn’t perfectly stable, especially the ‘Create FOMod’ button, so it is better to write your script in a text editor. If you include the line

<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://qconsulting.ca/fo3/ModConfig5.0.xsd">

at the beginning of your script, and edit it in Microsoft Visual Studio, you are given hints about what needs to/can be included at each point:

and incorrect syntax will be highlighted.

This guide is intended to go over the features you can use, and give some useful tricks. As I go through, I’ll build up the diagram on the next page which shows everything in a script...



It looks complicated, but this contains every part of a script.

# Important XML rules

XML files consist of tags in <angle brackets>. Inside a tag, you have a name (such as installSteps) and other options (such as order=”Explicit”). Every tag must be *closed* by a /, either like this:  
<tag>blah blah blah</tag>  
or like this:  
<tag parameter="whatever"/>

# General parts of a script

## Files

The same syntax is used at several points to unpack files, so this section will describe it.

Entries in the list of files can take two forms. A <file/> is a single file, and if it’s an .esp or .esm file it will be automatically activated. A <folder/> is a folder with all the files inside it.

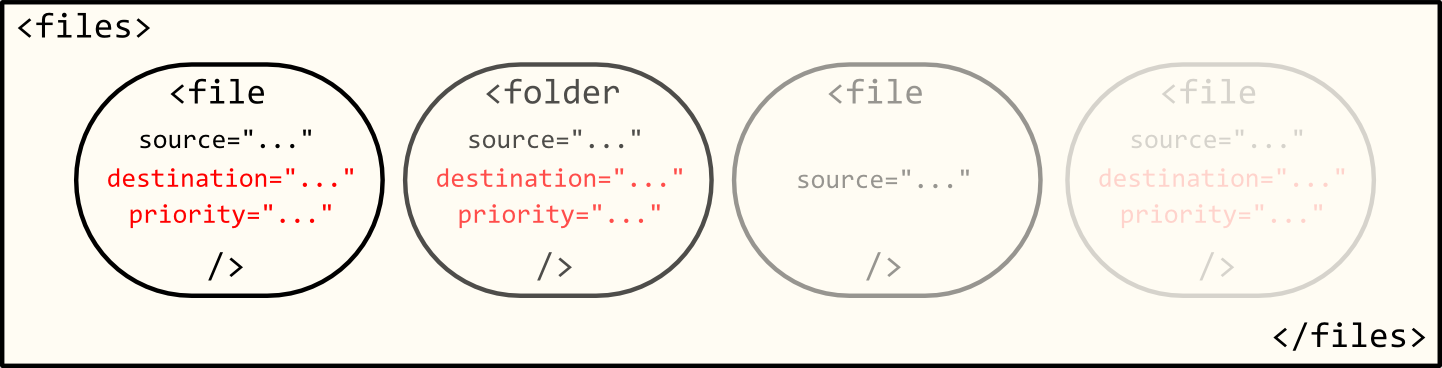
Both must have a source="...", where the ... is the path to the file from the root of the FOMOD/archive.

Both can optionally have a destination="..." where the ... is the path to unpack it to inside the Data folder.

Finally, both can have a priority="#" where # is an integer. Higher priority numbers are unpacked after lower priority numbers.

For example, if you wanted to unpack an esp directly, you might have  
<file source="example.esp"/>  
and if you wanted to unpack a folder from ‘optional/Meshes’ to ‘Meshes’ and have it overwrite whatever is at priority 1 or 0, you could use  
<folder source="optional/Meshes" destination="Meshes" priority="2"/>

Often you might want to take everything inside a folder and drop it directly in the Data folder:  
<folder source="path/Data" destination="" priority="1"/>



## Dependencies

Dependencies can be used to determine whether a menu option can be selected, whether a page of the script can be seen, and whether files are installed in the last section. They can be made quite complicated by nesting different <dependencies> inside each other.

Inside each <dependencies>, there will be a list of dependencies which can be true or false (e.g. “Player has file *x* in their load order”, “Flag *y* has been set to *z*”). Depending on whether (all/any) of them are true, the <dependencies> is treated as true or false. (All/any) depends on the value of operator in the <dependencies>, which defaults to "And" (all of them must be true) but can be set to "Or" (at least one must be true).

There are quite a few types of dependency available, but not all of them work as intended. For example, you can’t set an NVSE version dependency.

The main ones which do work are…

<fileDependency file="A" state="B"/>  
where A is the name of an esp or esm file, and B can be ‘Active’ (true if file is activated), ‘Inactive’ (true if file is present, but not activated) or ‘Missing’ (true if file is not present). So, for example,  
<dependencies operator="Or">  
 <fileDependency file="HonestHearts.esm" state="Missing"/>  
 <fileDependency file="HonestHearts.esm" state="Inactive"/>  
</dependencies>  
will be true if the player doesn’t have Honest Hearts activated.

<flagDependency flag="A" value="B"/>  
which is true if the flag A is set to B (see the next section).

## Flags

Flags store a string of text, which starts empty and can be set to some other option by a menu option. Flags can be used to follow which options a player has chosen. Flags are set inside <conditionFlags>  
using  
<flag name="abc">xyz</flag>  
which sets a flag *abc* to the value *xyz*.

An example use of flags is in a mod which has a DLC patch, an optional file, and a DLC patch for the optional file. If the player chooses to install the DLC patch, a flag is set to “On”, e.g.  
<flag name="deadmoney">On</flag>  
and if they choose the optional file, another flag is set, e.g.  
<flag name="marvelousplugin">On</flag>  
Then, later on in conditionalInstallFiles (which we’ll cover later), the DLC optional-file patch is given these dependencies:  
<dependencies operator="And">  
 <flagDependency flag="marvelousplugin" value="On"/>  
 <flagDependency flag="deadmoney" value="On"/>  
</dependencies>  
The operator="And" isn’t actually needed, since dependencies work that way by default, but it can be helpful to keep track.

If you want to act only when a flag has *not* been set, use an empty string in value:  
<flagDependency flag="AVeryFineFlag" value=""/>

# Shape of the script

There are up to four main bits of a script.

If the entire mod has requirements to allow it to be activated, these use a  
<moduleDependencies>...</moduleDependencies> section.

If there are files which must always be installed, regardless of other choices on the menu, then put them in the <requiredInstallFiles>...</requiredInstallFiles> section.

If you want a menu to appear, this is controlled by the <installSteps order="Explicit">... </installSteps> section. Menu options can be used to unpack files or set flags.

The third section, <conditionalFileInstalls>...</conditionalFileInstalls>, is used to unpack files controlled only by dependencies.

None of these sections are required (though you should have at least one if you want to do anything…)

## Install Steps

There are three different levels to consider here.

Each <installStep name="DLCs and WME"> is a page of the menu. As long as you have <installSteps order="Explicit"> at the beginning of the section, they will appear in the menu in the same order as in the XML file.

Inside each <installStep> you have an <optionalFileGroups> section, and optionally a <visible> section. <visible> acts exactly like <dependencies>, and if it’s present, the page of the menu will only appear if it’s true.

Inside <optionalFileGroups>...</optionalFileGroups>, you have a list of <group>s. Each <group> is a small heading on the menu collecting some options under it. <optionalFileGroups> displays the groups in alphabetical order by default, but you can use  
<optionalFileGroups order="Explicit"> or <optionalFileGroups order="Descending"> to display the list of groups either in the order they appear in the XML file, or reverse-alphabetical order.

Each group must have a name (the text on the menu) and a type, like this:  
<group name="some text" type="SelectAny">  
The type is a rule about which options can be selected. It needs to be one of these:

* "SelectAny"
* "SelectExactlyOne"
* "SelectAtMostOne"
* "SelectAtLeastOne"
* "SelectAll"

Within each group, there is a <plugins> section, containing a list of menu options in the group. This can have an order in the same way as <optionalFileGroups>.

Each entry in the menu is described by a <plugin>, which must have a name to go on the menu:  
<plugin name="XYZ">...</plugin>.

Inside that, you first need a description.  
<description>  
 <![CDATA[Here is the text which appears when you click on this option.]]>  
</description>  
The description is shown on the right side of the menu when clicking on this option.

Next, you can include an image, which shows under the description, if you want:  
<image path="fomod/xyz.jpg"/>  
Images can be placed anywhere in the FOMOD archive, but it’s usually easiest to put them in the fomod folder. I’ve used .png and .jpg files; it may be possible to use other formats.

Now you either provide a list of files to unpack in a <files> section, set a list of flags inside a <conditionFlags> section, or both.

The final part of the <plugin> is the <typeDescriptor>. This determines the type of menu entry – I’ll say what that means in a moment. You can use either a direct <type>, e.g.  
<type name="Recommended"/>  
or a <dependencyType>, which sets the type using the dependencies described above.

The type puts some rules on how the menu option is selected. The available types (names) are:

* "NotUsable": The option shouldn’t be selected, and if you select it, FOMM will pop up a warning that this option is “not usable with your enabled mods” and that enabling it “may result in game instability”.
* "CouldBeUsable": In principle: the option can be selected, but FOMM should inform you that you must activate some other files in your load order for it to work. In practice, however, this type behaves exactly the same as "NotUsable".
* "Optional": The option can be freely selected or unselected.
* "Recommended": The option is selected automatically. If you disable it, FOMM will give you a warning that doing so “may result in game instability”.
* "Required": The option is selected automatically and can’t be unselected. If you try to disable it, FOMM will not disable it, and will show a warning that “This option is required”.

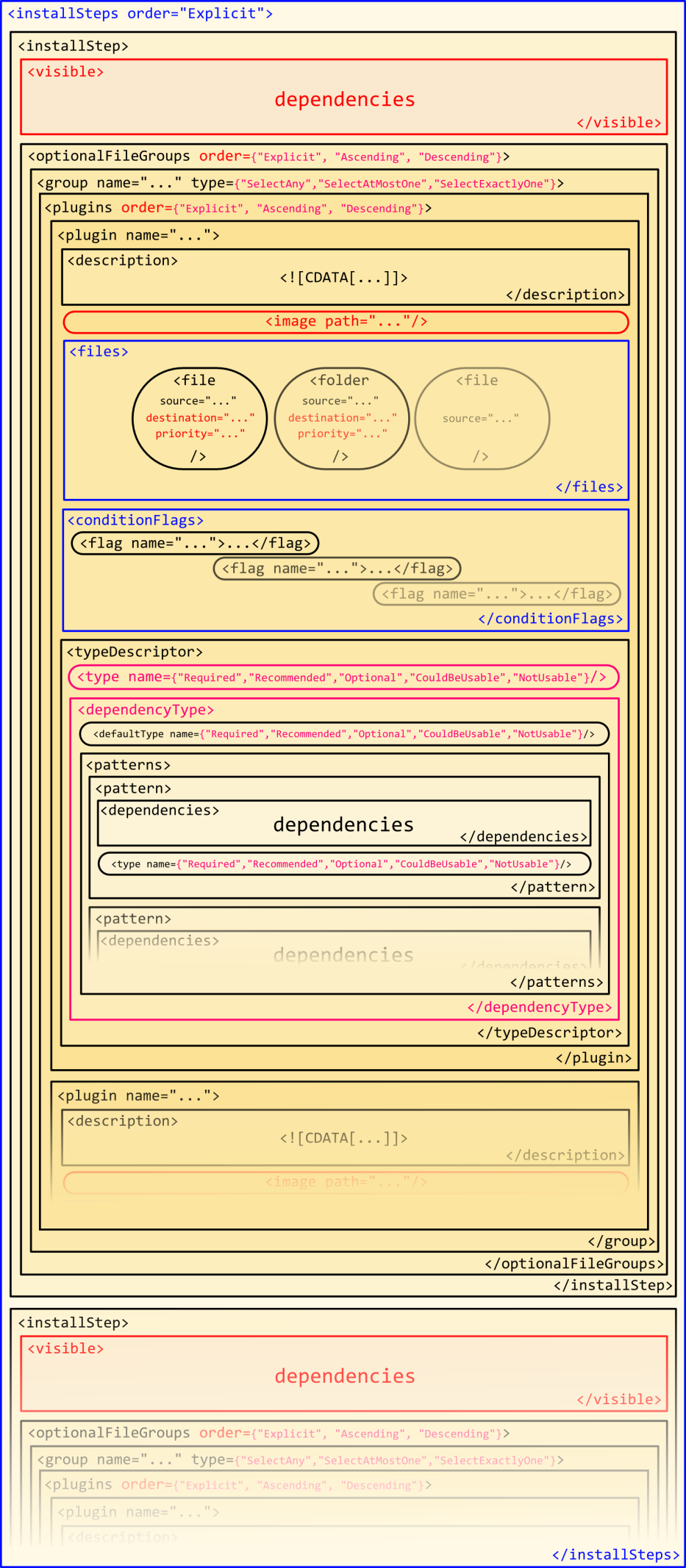
A <dependencyType> has two parts: <defaultType name="..."/> sets the default, and then a <patterns> section changes the type based on dependencies.

In the patterns section, each <pattern> contains a normal <dependencies> section and the corresponding <type name="..."/>.

An example of the use of this sort of type is for a mod with a patch for Honest Hearts. That plugin might have something like…  
<typeDescriptor>  
 <dependencyType>  
 <defaultType name="NotUsable"/>  
 <patterns>  
 <pattern>  
 <dependencies>  
 <fileDependency file="HonestHearts.esm" state="Active"/>  
 </dependencies>  
 <type name="Recommended"/>  
 </pattern>  
 <pattern>  
 <dependencies>  
 <fileDependency file="HonestHearts.esm" state="Inactive"/>  
 </dependencies>  
 <type name="CouldBeUsable"/>  
 </pattern>  
 </patterns>  
 </dependencyType>  
</typeDescriptor>  
This would show an option for the Honest Hearts patch on the menu, and warn the player if they’re trying to use the patch without Honest Hearts, or trying to disable the patch with Honest Hearts.

Here’s a somewhat more complicated example with a patch for all four DLCs:  
<typeDescriptor>  
 <dependencyType>  
 <defaultType name="CouldBeUsable"/>  
 <patterns>  
 <pattern>  
 <dependencies operator="And">  
 <fileDependency file="DeadMoney.esm" state="Active"/>  
 <fileDependency file="HonestHearts.esm" state="Active"/>  
 <fileDependency file="OldWorldBlues.esm" state="Active"/>  
 <fileDependency file="LonesomeRoad.esm" state="Active"/>  
 </dependencies>  
 <type name="Recommended"/>  
 </pattern>  
 <pattern>  
 <dependencies operator="Or">  
 <fileDependency file="DeadMoney.esm" state="Missing"/>  
 <fileDependency file="HonestHearts.esm" state="Missing"/>  
 <fileDependency file="OldWorldBlues.esm" state="Missing"/>  
 <fileDependency file="LonesomeRoad.esm" state="Missing"/>  
 </dependencies>  
 <type name="NotUsable"/>  
 </pattern>  
 </patterns>  
 </dependencyType>  
</typeDescriptor>  
In both these examples, I’ve included the "CouldBeUsable" setting, but it’s not necessary since it behaves at the moment like "NotUsable".

On the next page is a diagram summarizing all the information.

Red things are optional.  
Select at least one from a set of blue things.  
Select only one from a set of pink things. Inside a tag, the set is indicated by {}.

## Conditional File Installs

This section works very similarly to a <dependencyType>. There is no default, however, and instead of a <type/>, each pattern has <files> just as with a <plugin>.

That may be a bit opaque, so here is an example:  
<conditionalFileInstalls>  
 <patterns>  
 <pattern>  
 <!--All DLC, no GRA version of Ironsights.-->  
 <dependencies operator="And">  
 <flagDependency flag="relevantflag" value="On"/>  
 <flagDependency flag="DLC" value="On"/>  
 <flagDependency flag="unwantedflag" value=""/>  
 </dependencies>  
 <files>  
 <file source="Options/xyz.esp" destination="xyz.esp"/>  
 </files>  
 </pattern>  
 </patterns>  
</conditionalFileInstalls>  
This will install xyz.esp if the flags *relevantflag* and *DLC* have been set to “On”, and *unwantedflag* hasn’t been set.

# Other FOMOD things

The above should explain how to make ModuleConfig.xml do almost everything it can, and contains a few examples of useful code for things like DLC patches.

A few other things need to go into a complete FOMOD. They are all contained in info.xml, and can be entered most easily using the ‘Create FOMod’ button in FOMM’s Package Manager. You should add the version of the FOMod (important, since FOMM will complain if it doesn’t match the page), a short description, the author’s name, and a screenshot which will appear in the top right of the Package Manager (clicking on the screenshot shows it at full size).

The ‘Create FOMod’ button can also be used to create a Premade FOMOD Pack. This takes a set of files and generates a FOMOD from them, so it can be used to turn somebody else’s mod into a FOMOD without distributing their files. This could also be used for updates – you could release a PFP which takes the old version of your mod and adds the new files to update it.

Once you’ve made a FOMOD with the ‘Create FOMod’ button, you need to take the files in order to distribute them (e.g. on New Vegas Nexus). To do this, go to wherever FOMM keeps its FOMODs – to find this, go into FOMM’s settings and look at the Fallout New Vegas tab; your mods will be in the ‘Mod Directory’. You can take your FOMOD and upload it, or you can turn it into a FOMod-ready archive to make things slightly easier.

A FOMOD is simply a compressed folder, and it could really be a zip, 7z or rar file. To find out which, go to the ‘FOMod Options’ tab of FOMM’s Package Manager and look at the ‘Format’ option. Rename your FOMod appropriately (to .7z, .rar or .zip) and it will be a FOMod-Ready Archive which can be added to FOMM using the ‘Add FOMod’ button, and work automatically in the Nexus Mod Manager – but because it is an archive, it can be installed or edited manually by people who prefer not to use FOMM or the NMM.