*Package Definitions*

*Basic Syntax*

As we learned in this chapter, a package is a Modelica entity that allows us to organize definitions (including definitions of other packages). The syntax definition of apackage has a lot in common with other Modelica definitions. The general syntax for a package is:

**package** **PackageName** "Description of package"

*// A package can contain other definitions or variables with the*

*// constant qualifier.*

**end** PackageName;

A package definition can be prefixed by the encapsulated qualifier. We’ll discuss that more when we examine Modelica’s [*Lookup Rules*](http://book.xogeny.com/components/packages/lookup/#lookup-rules).

Packages can also be nested, *e.g.,*

**package** **OuterPackage** "A package that wraps a nested package"

*// Anything contained in OuterPackage*

**package** **NestedPackage** "A nested package"

*// Things defined inside NestedPackage*

**end** NestedPackage;

**end** OuterPackage;

In fact, nesting of packages is very common and allows us to represent complex taxonomies.

*Directory Storage*

Although it is possible to build an entire library of Modelica definitions in a single file as a series of nested packages, this is undesirable for at least two reasons. The first is that the resulting file would be quite hard to read based on its length and the degree of indenting that would be required. The second is that from the standpoint of version control, it is much better to break things into smaller files to help avoid any merge conflicts.

*Stored in a Single File*

There are several ways that Modelica source code can be mapped to a file system. The simplest way is to store everything in a file. Such a file should have a .mo suffix. Such a file might contain only a single model definition or it might contain a deeply nested hierarchy of packages or anything in between.

*Stored as a Directory*

As we already discussed, storing everything in one file is usually not a good idea. The alternative is to map Modelica definitions into a directory structure. A package can be stored as a directory by creating a directory **with the same name as the package**. Then, inside that directory, there must be a file called package.mo that stores the definition of the package, **but not any nested definitions**. The nested definitions can be stored either as single files (as described above) or as directories representing packages (as described in this paragraph). The following diagram attempts to visualize a sample directory layout:

/RootPackage # Top-level package stored as a directory

package.mo # Indicates this directory is a package

NestedPackageAsFile.mo # Definitions stored in one file

/NestedPackageAsDir # Nested package stored as a directory

package.mo # Indicates this directory is a package

The package.mo file associated with the package named RootPackage would look something like this:

**within**;

**package** **RootPackage**

*// only annotations can be stored in a package.mo*

**end** RootPackage;

There are two important things to note here. First, the within clause should be present, but empty. This indicates that this package is not contained in any other packages. In addition, the definitions of NestedPackageAsFile and NestedPackageAsDir are not (and cannot be) present. Those must be stored outside thepackage.mo file.

Similarly, the package.mo file associated with the NestedPackageAsDir package would look like this:

**within** RootPackage;

**package** **NestedPackageAsDir**

*// only annotations can be stored in a package.mo*

**end** NestedPackageAsDir;

Again, there should be no definitions contained in this package, only annotations. The within clause is slightly different, reflecting the fact thatNestedPackageAsDir belongs to the RootPackage package.

Finally, the NestedPackageAsFile.mo file would look something like this:

**within** RootPackage;

**package** **NestedPackageAsFile**

*// The following can be stored here including:*

*// \* constants*

*// \* nested definitions*

*// \* annotations*

**end** NestedPackageAsFile;

The within clause is the same as for the NestedPackageAsDir package definition, but since we are storing this package as a single file, nested definitions for constants, models, packages, functions, *etc.* are allowed here as well.

*Ordering for Directories*

When all definitions are stored within a single file, the order they appear in the file indicates the order they should appear when visualized (*e.g.,* in a package browser). But when they are stored on the file system, there is no implied ordering. For this reason, an optional package.order file can be included alongside the package.mo file to specify an ordering. The file is simply a list of the names for nested entities, one per line. So, for example, if we wanted to impose an ordering on this sample package structure, the file system would be populated as follows:

/RootPackage # Top-level package stored as a directory

package.mo # Indicates this directory is a package

package.order # Specifies an ordering for this package

NestedPackageAsFile.mo # Definitions stored in one file

/NestedPackageAsDir # Nested package stored as a directory

package.mo # Indicates this directory is a package

package.order # Specifies an ordering for this package

In the absence of a package.order file, a Modelica tool would probably simply sort packages alphabetically. But if we wanted to order the contents of theRootPackage in reverse alphabetical order, the package.order file in the RootPackage directory would look like this:

NestedPackageAsFile

NestedPackageAsDir

This would specify to the Modelica tool that NestedPackageAsFile should come before NestedPackageAsDir.

*Versioning*

*MODELICAPATH*

Most Modelica tools allow the user to open a file either by specifying the full path name of the file or by using a file selection dialog to open it. But it can be tedious to find and load lots of different files each time you use a tool. For this reason, the Modelica specification defines a special environment variable called MODELICAPATHthat the user can use to specify the location of the source code they want the tool to be able to automatically locate.

The MODELICAPATH environment variable should contain a list of directories to search. On Windows, that list should be separated by a ; and under Unix it should be separated by a :. When the Modelica compiler comes across a package it has not already loaded, it will search the directories listed by the MODELICAPATHenvironment variable looking for either a matching file or directory. For example, if the MODELICAPATH was defined as (assuming Unix conventions):

/home/mtiller/Dir1:/home/mtiller/Dir2

and the compiler was looking for a package called MyLib, it would first look in /home/mtiller/Dir1 for either a package named MyLib.mo (stored as a single file) or a directory named MyLib that contained a package.mo file that defined a package named MyLib. If neither of those could be found, it would then search the/home/mtiller/Dir2 directory (for the same things).

*modelica:// URLs*

In many cases, it is useful to include non-Modelica files along with a Modelica package. These non-Modelica files might contain data, scripts, images, etc. We call these non-Modelica files “resources”. Now that we’ve covered how Modelica definitions are mapped to a file system, we can introduce an extremely useful feature in Modelica which is the use of URLs to refer to the location of these resources.

For example, when we discussed [*External Functions*](http://book.xogeny.com/behavior/functions/func_annos/#ext-functions), we introduced several annotations that specified the location of such resources. Specifically, theIncludeDirectory and LibraryDirectory annotations specified where the Modelica compiler should look for include and library files, respectively. As was briefly mentioned then, the default values for these annotations started with modelica:://LibraryName/Resources. Such a URL allows us to define the location of resources **relative to a given Modelica definition on the file system**. Let us revisit the directory structure we discussed earlier, but with some resource files added:

/RootPackage # Top-level package stored as a directory

package.mo # Indicates this directory is a package

package.order # Specifies an ordering for this package

NestedPackageAsFile.mo # Definitions stored in one file

/NestedPackageAsDir # Nested package stored as a directory

package.mo # Indicates this directory is a package

package.order # Specifies an ordering for this package

datafile.mat # Data specific to this package

/Resources # Resources are stored here by convention

package.mo # Indicates Resources is a package

logo.jpg # An image file

If we have a model that needs the data contained in NestedPackageAsDir, we can use the following URL to reference it:

modelica://RootPackage/NestedPackageAsDir/datafile.mat

Such a URL starts with modelica://. This is our way of indicating that the resource being referenced is with respect to a Modelica model and not, for example, something to be fetched over the network. The // is then followed by the fully qualified name of a Modelica definition except that each component is separated by a/ instead of a .. The Modelica compiler will interpret this as the name of the directory that contains that definition. Finally, the last element in the URL names the file to be used.

As another example, if we wished to reference the logo.jpg file in the Resources package, we would use the following URL:

modelica://RootPackage/Resources/logo.jpg

It is a common convention to store resources related to a library in a nested package named Resources (hence the default values for IncludeDirectory andLibraryDirectory).