RPM packaging "the right way"

Jos Vos **<jos@xos.nl>**

X/OS Experts in Open Systems BV
Kruislaan 419
1098 VA Amsterdam
The Netherlands

March 28, 2003

1 Introduction

Software package management covers a wide range of aspects. For system administrators, it provides a method for controlled installation, upgrade, removal, and consistency checks of software components. During all those operations, the package system keeps track of the status of installed files and maintains inter-package dependencies.

For software developers and maintainers, package management provides a structured and trackable method for generating ready-to-install binary packages. Using package management, it is possible to define an exact recipe for generating binary packages starting from the original (pristine) source code. This recipe may include references to the original source code, patches, additional scripts, commands for compiling and installing the software, and a detailed list of package contents. You can also include implicit and explicit dependencies to other packages.

Several different package management systems exist for UNIX and UNIX-like operating systems. In the Linux-world, two advanced package management systems exist: the Debian package management system (dpkg) and the Red Hat Package Manager (rpm). The rest of this paper will focus on the details of rpm, that is used by many Linux distributions, including Red Hat Linux, SuSE Linux, and Mandrake. The style of using rpm and writing rpm packages in some aspects depends on the target Linux distribition.[1] Here, we will focus on writing packages for Red Hat Linux.

This paper is not a tutorial on rpm packaging, nor is it a manual showing you all features of rpm. Instead, it tries to give you some guidelines, hints, and tricks for building "good" and maintainable rpm packages, although what is good or bad remains a matter of taste.

[1]The Linux Standard Base project includes guidelines for writing distribution-independent packages.

2 Getting started

2.1 Sample spec file

A simple spec file for a imaginary package called **mypkg** looks like:

```
Name: mypkg
Version: 6.0
Release: XOS.1
Source: ftp://ftp.xos.nl/pub/demo/mypkg-6.0.tar.bz2
Patch0: mypkg-6.0-config.patch
Patch1: mypkg-6.0-hotfix.patch

URL: http://www.xos.nl/
Packager: X/OS Experts in Open Systems BV <info@xos.nl>
BuildRoot: %{_tmppath}/rpm-buildroot-%{name}-%{version}-%{release}
Prefix: %{_prefix}

Summary: Demo package for rpm
License: GPL
Group: Development/Tools

%description
The mypkg package is only a demo package for rpm.

%prep
%setup -q
%patch0 -p1
%patch1 -p1

%build
./configure --prefix=/usr
make

%install
make DESTDIR=%{buildroot} install

%clean
rm -rf %{buildroot}

%files
%defattr(-,root,root)
%doc README COPYING
%{_bindir}/*
%{_mandir}/man1/*
%dir %{_sysconfdir}/mypkg
%config(noreplace) %{_sysconfdir}/mypkg/*.conf

%changelog
* Thu Mar 06 2003 Jos Vos <jos@xos.nl> 6.0-XOS.1
- Sample changelog entry.
```

In the next sections, we will look at spec files in more detail and explore more complex examples. If you plan to package your own software, it is a good idea to create your own template and use that as a base for new packages, in order to give your rpm's a uniform style. When you modify existing spec files of Red Hat Linux or other distributions, mark your own changes with easily identifiable comment lines, so that porting your changes to newer versions of the package will be easier.

2.2 Build environment

The default rpm build environment on Red Hat Linux systems is **/usr/src/redhat**. You probably want to create a personal build environment in your home directory. This can be realized by creating a file **$HOME/.rpmmacros** with the following contents:

```
%_topdir        /home/xos/rpm
```

The top directory should contain subdirectories named **SPECS**, **SOURCES**, **BUILD**, **SRPMS**, and **RPMS**. Architecture-specific subdirectories of **RPMS** are automatically created when binary packages are built. The **$HOME/.rpmmacros** file can be extended to support GPG package signing:

```
%_signature     gpg
%_gpg_path      /home/xos/.gnupg
%_gpg_name      X/OS Experts in Open Systems BV <info@xos.nl>
```

In this case, rpm uses the key with the specified name for signing packages, using the keyring in the standard GnuPG directory. You should also import your public key into rpm's own keyring, in order to let rpm check signatures of built packages on other systems. This can be done with the **rpm --import** command.

2.3 Macros

The rpm system provides you with many predefined macros and the ability to add custom macros. Use them whenever you can, as it makes packages more portable and easy to maintain. The command **rpm --showrc** shows you the current rpm settings, including the predefined macros. You can define your own macros in a spec file with the **%define** statement:

```
%define mymacro myvalue
```

In some cases you might want to reuse the same spec file for different instances of the package. In that case, you can define your own macro on the rpm command line at the moment you generate a package:

```
rpmbuild -ba --define "mymacro myvalue" mypkg.spec
```

Use this feature with extreme care, as the resulting source rpm file does not reflect the value of your dynamically set macro and is thus not the ultimate reference for regenerating the binary rpm file, thereby nullifying one of the major advantages of rpm.

Macros are much more powerful than just substituting values, as they can also be used in conditional expressions in the spec file (compare this with the **#ifdef** feature of the C preprocessor):

```
%define krb5_support 1

%if %{krb5_support}
```

```
...
%else
...
%endif
```

In this example, you can easily switch between a package with or without Kerberos support by setting the value of the **%krb5_support** macro. This is especially useful in case one spec file is used as a base for more versions of a distribution, where some features are not available in all versions. Note that in that case you should also make a difference in release numbers (see below).

3 Package headers

3.1 Identification

The spec file for a package starts with several key-value pairs, containing various information. The **Name** and **Version** fields contain the name and version of the rpm package. In most cases, packages have lowercase names, but sometimes, when the package is identified by a specific combination of upper- and lowercase characters (i.e., **ImageMagick** or **LPRng**), this is also used for the rpm package name.

Sometimes the original version number of a package contains a dash. As a version number of a rpm package should not contain a dash, this has to be changed, e.g., by using an underscore instead.

The **Release** field is defined by the rpm packager. As a package is uniquely identified by the 3-tuple (name, version, release), the release number is usually reset to 1 for every new version of a package. When you make your own packages, you have to decide what convention to use. Original packages in Red Hat Linux in general use plain numbers, but you can also add your own identification to make a distinction between Red Hat packages and your own packages. At X/OS we use the convention to number our own packages XOS.1, XOS.2, ..., while our versions of a Red Hat package with release (say) 6 are numbered 6.XOS.1, 6.XOS.2, ... As with version numbers, release numbers may not contain dashes.

The **Group** field is a tree-style category, indicating to what group of software a package belongs. It is used in some tools, of which the Red Hat Linux installer is the most important example. The best way for choosing the group for a package is to look in which group a similar standard Red Hat Linux package belongs. Another choice could be to make a local group (or set of groups) and put all local packages in one of those groups.

3.2 Source and patch files

The **Source** and **Patch** fields contain either URL's or plain filenames of the source and patch files. The last part is used as the filename in case an URL is specified. It is very important to include an URL if the file can be found somewhere on the Internet. Although rpm itself only looks at the plain filename, the URL is a great help for maintaining the rpm package and makes it easier to find new versions. In some cases, especially with non-free software, there are no URL's pointing directly to the software, for example because the download URL is dynamically created after filling in a form. In those cases, it is a good idea to include references to this form in a comment line, to make future upgrades easy to find.

Any number of source and patch files may be specified, using the field names **`Source0`**, **`Source1`**, ..., **`Patch0`**, **`Patch1`**, ..., whereby omitting the number is the same as using number 0. These files may be referred to as **`%SOURCE0`**, **`%SOURCE1`**, ..., respectively **`%PATCH0`**, ..., in the rest of the spec file. The numbers do not need to be consecutive, so for large and complex packages these fields can be numbered according to a logical structure. Red Hat's kernel package contains patches up to **`Patch11030`**, while it (luckily...) does not include 11031 patches.

An useful feature of rpm is the ability to specify that certain sources and/or patches should not be included in the source rpm (src.rpm) file. This is done with the **`NoSource`** and **`NoPatch`** fields, containing a comma-separated list of source- or patch-numbers to be excluded.[2] This is used for non-free software, that may not be distributed. In that case, the nosrc.rpm file, with everything except the excluded sources and patches, may still be distributed so that everyone can build their own binary packages for internal use in case they have (legal) access to the software.[3]

3.3 Architectures, preconditions and requirements

In case your package only contains architecture-independent files, like with PHP or Python packages, you should specify **`BuildArch: noarch`** in the header section. In that event, a generic noarch.rpm file is created, instead of an architecture-specific package, like i386.rpm.

The rpm system automatically generates dependency information at the moment a binary package is created. This behaviour can be disabled by specifying **`AutoReqProv: no`**. Furthermore, additional dependency information can be included with the **`Provides`** and **`Requires`** header fields. The automatic dependency mechanism can also be changed. In the default setup, the scripts **`/usr/lib/rpm/find-provides`** and **`/usr/lib/rpm/find-requires`** are used to calculate the dependency information. But if you redefine the **`%__find_requires`** and/or **`%__find_provides`** macros, a custom script can be used:

```
%define __find_provides %{_builddir}/%{name}-%{version}/my-find-provides
%define __find_requires %{_builddir}/%{name}-%{version}/my-find-requires
```

The custom scripts have to be specified with a full pathname. In case a script is included as a source file, it can be referred to as **`%SOURCE2`**, for example.

[2]Recent versions of rpm contain a bug, so that only one number is accepted in these fields. You then need to include the **`NoSource`** and **`NoPatch`** fields once for every source/patch that you want to exclude.

[3]Some licensing conditions may even prohibit repackaging software for internal use.

4 Unpacking the source

The preparation phase, defined by the section of the spec file that starts with the **%prep** keyword, is used for unpacking the source code and for applying local patches. A simple example is:

```
%setup -q
%patch0 -p1
%patch1 -p1
%patch2 -p1
```

The **%setup** statement unpacks **%SOURCE0** in a quiet (**-q**) way. The **%setup** macro interprets the source filename and can handle different formats and compression methods. This is sufficient for most Linux software, which is distributed as **.tar.gz** or **.tar.bz2** files.

Rpm by default assumes that the extracted source tree has a top directory named **%{name}-%{version}**. If this directory is named differently, the alternative name can be specified with **-n**. And in the unusual case that the source archive does not contain a top directory at all, the option **-c** can be specified, that creates a top directory with the standard name before extracting the source archive. Finally, the **-T** option prohibits **%setup** from extracting **%SOURCE0**, so this command just creates an empty top directory:

```
%setup -T -c
```

It is also possible to extract more source archives at once, for example with the following command:

```
%setup -q -a 1 -a 2
```

This extracts **%SOURCE0**, goes to the just created top directory, and then extracts **%SOURCE1** and **%SOURCE2**.

The **%patch** commands are all applied in the top directory, so for patches that include those top directory in the pathnames of the patched files, the **-p1** flag should be used. The number behind **%patch** refers to the patch number to be used and omitting this number refers to **%PATCH0**.

5 Compiling

The build phase, starting with **%build**, contains the recipe for compiling the package. The current working directory at the beginning of the build phase is the top directory of the package's source tree, for example **mypkg-6.0**. A typical example for this phase is:

```
./configure --prefix=/usr --mandir=/usr/share/man
make
```

Like in the **%prep** phase, several macros can be used to make the build recipe easier to write and maintain. The first one is **%configure**. This macro is used as an abbreviation for calling **./configure** with several standard arguments, like those shown above (see the output of **rpm --showrc** for all the arguments). Note that you can not use this if you want to create a package that has its files in another place than the standard system directories.

Even handier than **%configure** is the fact that several standard paths used for compilation and installation are available as macros. A few examples are listed here:

%_bindir	**/usr/bin**
%_libdir	**/usr/lib**
%_sbindir	**/usr/sbin**
%_sysconfdir	**/etc**
%_initrddir	**/etc/rc.d/init.d**
%_datadir	**/usr/share**
%_mandir	**/usr/share/man**

Note that some macros are defined in terms of other macros, but for the sake of simplicity, the final results are shown in the above table. It is a good idea to define your own macros, in case you need some pathnames frequently in the spec file:

```
%define mylibdir %{_libdir}/mypkg
```

You can then use **%mylibdir** in all scripts and also in the **%files** section, and change it very easily in future versions of the package.

If software does not need to be build, i.e. when it contains only PHP or Perl scripts or when packaging binary software, the build recipe can be left empty.

6 Installing

6.1 Buildroot support

After the build phase, the installation of the software is done in the section starting with the **%install** keyword. The install recipe is usually the hardest part to write. The main reason for this is that the software needs to be installed in a temporary directory, the so-called buildroot. Note that rpm does not require this, but not using a buildroot environment for installation simply means completely messing up the build system when generating packages, which is highly undesirable and unmanageable.

There are a few ways to get a software package installed in a buildroot:

- The most popular method is to use the **%makeinstall** macro. Similar to **%configure**, this macro is an abbreviation for **make install** with extra arguments to redefine the paths in the Makefile, like **prefix**, **bindir**, and **includedir**, so that they are prefixed with **%buildroot**. Using **%makeinstall** assumes that the Makefile uses a standard set of make macros for installation paths.

- The second method is to manually specify an extra parameter (make variable) for installation in a different target root:

  ```
  make install DESTDIR=%buildroot
  ```

  To specify an alternative installation root directory, many packages support the **DESTDIR** variable in their makefiles, In some cases a similar variable with another name is available, like **ROOTDIR** or **TARGETDIR**.

Although the above examples look simple, often reality is a bit more difficult. For some packages the above examples will not work, either because the Makefile does not conform to the usual format or because the package has no provisions for installation in a different root. The install can be done “manually” if the package is simple. A complete **%install** recipe would then look like:

```
rm -rf %{buildroot}
mkdir -p %{buildroot}%{_bindir}
install -m 0755 src/mytool %{buildroot}%{_bindir}/
mkdir -p %{buildroot}%{_mandir}/man1
install -m 0755 man/mytool.1 %{buildroot}%{_mandir}/man1/
```

Most software is not that small and so it is often easier to add a patch to support a **DESTDIR** make variable by prefixing all target directories in install commands with **$(DESTDIR)** and inserting an initial **DESTDIR=** in the top of the Makefile to define its default value. And if you send the patch to the software author/maintainer, it might be included in the next release, eliminating the need to maintain it yourself.

If it is unclear what happens when installing , do a **make -n install** and see what commands are executed.

6.2 Creating directories

First of all, always remove an existing installation target directory:

```
rm -rf %{buildroot}
```

If you skip this step and the directory exists, it may be filled with undefined data, possibly from a previous build (or build attempt).

Sometimes you may need to create the installation directories yourself, especially when the software does not support alternative install directories by default.

You can either do this by including separate commands in the build recipe or by patching the Makefile adding **mkdir -p** commands.

If a directory must be created that belongs to this package (and thus is included), better use **install** instead of **mkdir -p**:

```
install -m 0755 -d %{buildroot}%{_libdir}/mypkg
```

This enforces the mode of the created directories, something that you do want to control as part of your package.

6.3 Customizing installation

In case you want to add your own files to the package, like an initialization script or a PAM file, you can install them directly from the **SOURCES** directory:

```
mkdir -p %{buildroot}%{_initrddir}
install -m 0755 %SOURCE3 %{buildroot}%{_initrddir}/mypkg

mkdir -p %{buildroot}%{_sysconfdir}/pam.d
install -m 0644 %SOURCE4 %{buildroot}%{_sysconfdir}/pam.d/mypkg
```

Sometimes packages do not install files at the correct place in the directory tree to conform to the Filesystem Hierarchy Standard. Often this is easy to fix, but some packages rely on having their subdirectories and files in their own subtree. You can often overcome this by moving parts of that tree to other places and add symlinks.

6.4 Non-root builds

Most of the time you can build your package as a non-root user, which is strongly recommended. Some packages, however, explicitly change file permissions and ownerships during installation. This usually happens when the package installs setuid- or setgid-programs. In those cases, it is usually better to eliminate those changes by patching the Makefile and explicitly specify file permissions and ownerships in the **`%files`** section of the spec file (see below).

7 Pre/post-install/uninstall scripts

7.1 Do's and don't's for scripts

The most often misused (or abused) part of rpm is the ability to define scripts that run on the system where the package is installed. A few important things that should be kept in mind when writing such scripts:

- The scripts should not be interactive. Unlike Debian's dpkg, that provides a way to interact and configure a package at installation time, rpm is not designed as such and the scripts should run unattended.
- The scripts should not produce any output, so redirect any possible output (both stdout and stderr) of commands used in the scripts to **`/dev/null`**.
- The scripts should be very robust and must check every precondition. To illustrate this requirement: when a line is added or removed to a system-wide configuration file, the script should first check if this has already been done.
- Minimize the number of utilities used in your scripts. All used utilities (or the packages owning them) must be listed in the **`Prereq`** field of the package header. Very basic utilties are assumed to be present and do not need to be listed.

If we ignore trigger scripts for now (more about those special scripts later), there are four types of scripts that can be defined for the (un)installation phases.

7.2 Pre-install scripts

Only a few packages need a pre-install script, the spec file section that starts with **`%pre`**. Only a few percent of the Red Hat Linux spec files include such a script, but for additional packages (mosly application software) it is needed even less often. This type of script is for example needed when a package requires an own user and/or group, because it contains files or directories owned by that user. The **useradd** and **groupadd** utilities, used to create a users and groups, are in the context of pre-install scripts usually called with explicit parameters for uid and gid numbers, although this is not required.

7.3 Post-install scripts

A post-install script, starting with **%post**, is used more often. All packages including shared libraries should include the command **/sbin/ldconfig**. Furthermore, all packages that include a service start/stop script that has to be registered with **/sbin/chkconfig** need to do this in the post-install script. Finally, a typical example of how included documentation in the info format is handled:

```
if [ -f %{_infodir}/mypkg.info.gz ]; then
    /sbin/install-info %{_infodir}/mypkg.info.gz %{_infodir}/dir
fi
```

This piece of code also illustrates how robust the scripts should be. One may wonder why the test for the **info.gz** file is done, as the package is assumed to contain it. Well, rpm allows a system administrator to install a package without installing files marked as documentation (the very seldomly used **--excludedocs** flag). In that case the info files are not installed, but the post-install script is always executed. Many Red Hat Linux post-install scripts do not test for this, and thus fail when the documentation files are not installed.

7.4 Pre-uninstall scripts

The pre-uninstall script, the **%preun** section in the spec file, is run just before the package is uninstalled. At this point a service has to be stopped and unregistered:

```
if [ $1 -eq 0 ]; then
    /sbin/service mydaemon stop > /dev/null 2>&1
    /sbin/chkconfig --del mydaemon
fi
```

Note the check on **$1**: this is the number of installed versions of the package after the uninstall has been completed. In this case, the service actions will only be done if the pre-uninstall script is not run during a package upgrade. Another example, showing how info files can be handled:

```
if [ $1 -eq 0 ]; then
    if [ -f %{_infodir}/mypkg.info.gz ]; then
        /sbin/install-info --delete \
            %{_infodir}/mypkg.info.gz %{_infodir}/dir
fi
```

This is the reverse action of the post-install script.

7.5 Post-uninstall scripts

The post-uninstall script is started by **`%postun`** in the spec file. Packages that include shared libraries should at least contain the command **`/sbin/ldconfig`**. Packages that have created their own user or group in the pre-install script must remove them in the post-uninstall script.

As post-uninstall scripts are also run after a package has been upgraded, the following piece of code is used to conditionally restart a service after the upgrade:

```
if [ $1 -ge 1 ]; then
    /sbin/service mydaemon condrestart > /dev/null 2>&1
fi
```

Like with pre-uninstall scripts, the script’s argument is the number of installed versions of the package after uninstalling it, so the conditional restart is not done when the package is removed (**`rpm -e`**).

7.6 Non-shell scripts

By default, all the scripts are executed using **`/bin/sh`** as command interpreter. Optionally, scripts can run using an alternative command interpreter, like in this example

```
%post -p /usr/bin/perl
```

Using other interpreters then **`/bin/sh`** is discouraged, as it is usually unnecessary and causes extra dependencies. There is one frequently used exception:

```
%post -p /sbin/ldconfig
```

Here **`/sbin/ldconfig`** is executed directly, without first starting a shell. In such cases, the script needs to be left empty, so it can only be used if this is the only command needed in the script.

8 Trigger scripts

All scripts described in the previous section only run at the moment the package itself is installed or removed, which is usually good enough. In some situations, however, a package needs to perform actions that depend on the (un)installation of other packages. For this type of actions, trigger scripts are designed. An example:

```
%triggerin -- setup
grep -q '^myport' /etc/services ||
    echo -e "myport\t\t999/tcp\t\t\t\t# My Port" >> /etc/services
```

This script will be executed every time the **`setup`** package is installed (note that an upgrade also implies an install). In this particular example, the advantages compared to using a post-install script are:

- It is not required that the **setup** package is installed before this package is installed, to ensure that the **/etc/services** file exists at the moment the script is executed.
- As the script is also executed after an upgrade (or re-install) of the **setup** package, it is ensured that the action is not undone because the **/etc/services** file was replaced during an upgrade.

Besides trigger-install scripts, there also exist trigger-uninstall scripts, starting with the **%triggerun** keyword. They are executed every time another package is removed. Practice shows that there are very few cases that need trigger-uninstall scripts.

Like a normal script, a trigger script gets as first parameter the number of versions of the current package installed after the (un)install action has been completed. In addition to this, a second parameter is given, representing the number of installed versions of the package the trigger script is referring to. In case of a trigger-uninstall script, this makes it possible to see if the uninstall is part of an upgrade or if the package is completely removed.

9 Package files

9.1 File specification format

The **%files** section of the spec file defines the actual contents of the package. The list of files and directories to include can be specified using shell-style wildcards, a feature that often makes the file list very short (and easy to maintain).

When specifying a directory (or an expression that matches a directory), the whole subtree starting at that directory is included. You should use the **%dir** directive to let only the directory itself be included, not its contents.

For more complex packages, especially if you want to split the installed files into several subpackages, you might want to consider generating the contents of the **%files** section(s) in temporary files during the **%install** phase. You can refer to such a file, say **RPM-FILES**, by specifying it at the start of the **%files** section:

```
%files -f RPM-FILES
```

The specified file is assumed to be in the top of the package’s source directory. If you use this feature, you can still specify additional files explicitly.

For files that do not physically need to be included in the package, you can use the **%ghost** tag. Those files are not included in the binary rpm package, but exist in the rpm database and are also removed together with the package. This feature can for example be useful for logfiles.

9.2 Determining the correct fileset

Be sure to include all files and directories that belong to the package. Especially package-specific directories, like **/usr/lib/mypkg**, are often omitted. Because rpm creates all parent directories at installation time, similar to what **cpio -d** does, this problem might not be detected immediately. Newer versions of rpm produce a warning when files in the buildroot are not included in the generated binary package(s), but this rule does not apply to directories.

On the other hand, you should not include non-package-specific files or directories. Remember that specifying a directory includes the directory and everything below. So, do not specify **%{_mandir}** or **%{_mandir}/***, but use **%{_mandir}/*/*** instead. The latter only matches the manual page files, not the system directories.

If you are dealing with a large package and do not immediately have a good overview of all the installed files, then (temporarily) add the following statement to the **%install** section

```
find %buildroot -print | sort > /tmp/install.log
```

and run **rpmbuild -bi**.

9.3 File attributes

All files should have the correct ownerships and permissions. This is usually done by starting the **%files** section with **%defattr(-,root,root)** and explicitly override non-root-owned files in the file list itself. As all file modes are taken over from the installed buildroot, you have to be sure to install them properly or override the mode explicitly per file with the **%attr** tag. As discussed earlier, the latter is recommended for setuid/setguid programs.

Mark all configuration files that belong to the package with **%config**. If you want to be sure that the file is never replaced when a package is being upgraded, use **%config(noreplace)** instead.

For some files, it makes no sense to let rpm verify certain attributes, like its md5sum, size, or modification time. In that case, you can use the **%verify** tag to narrow down the verification attributes, like **%verify(not md5 size mtime)**.

Language-specific files can be marked as such with **%lang**, like in **%lang(en)**. These files will only be installed if the corresponding language is chosen at package installation time. For standard language files, there is a more easy methode. At the end of the **%install** phase, you can use:

```
%find_lang %{name}
```

This creates a file **%{name}.lang** with all language-specific files found in standard directories like **/usr/share/locale**, prefixed with the proper **%lang()** tag. This file than has to be specified with **%files -f** and the language-specific files should not be listed anymore in the **%files** section itself.

10 Special cases

10.1 Non-free software

Commercial, non-free software often comes with a binary installer. In general, there is no way to see in advance what the installer does other then running it and try to find the installed files back (which is usually not that hard). With some tricks, like the **strings** command, it is often possible to manipulate the behaviour of the installer a bit by setting environment variables. For example, for a package showing a license with **more** it was enough to set **PAGER=cat** to avoid the paging.

If installers ask questions, use here-input to answer them. In that case it is also easy to prefix installation directories with **%buildroot**. In some cases, the installer reads directly from the terminal and you need an **expect** script to guide the installer. But it can be even worse: Linux software exists that has a graphical-only installer, that does not allow you to write a workaround to do an unattended installation. The last thing you can do then is writing clear instructions to stdout in the **%install** phase and let the package builder click on some buttons and fill in some fields.

When repackaging binary software, you probably want to set the following header fields:

```
ExclusiveOS: Linux
ExclusiveArch: i386
```

This specifies that the package should not be built on non-Linux and/or non-i386-based systems.

10.2 Python software

Python software often includes support for the Python Distribution Utilities (distutils), that makes building rpm packages very easy:

```
%build
%{__python} setup.py build

%install
%{__python} setup.py install --root=%{buildroot} --record=RPM-FILES

%files -f RPM-FILES
%defattr(-,root,root)
```

In the ideal case, the above example is enough for building a distutils-compliant Python package.

10.3 Perl modules

Another example is the packaging of Perl modules. The following recipe is a generic template[4] for Perl module spec files:

```
%build
%{__perl} Makefile.PL
make

%install
rm -rf %{buildroot}

eval `%{__perl} '-V:installarchlib'`
mkdir -p %{buildroot}/$installarchlib
make PREFIX=%{buildroot}%{_prefix} install

eval `%{__perl} '-V:sitearchexp'`
find %{buildroot}${sitearchexp}/* -type d -print |\
    sed "s|^%{buildroot}|%dir |" > RPM-FILES
find %{buildroot}/* -type f ! -name perllocal.pod -print |\
    sed -e "s|^%{buildroot}||" \
        -e "s|%{_mandir}\(.*\)|%{_mandir}\1*|" >> RPM-FILES

%files -f RPM-FILES
%defattr(-,root,root)
```

Note that this example will not universally work for all Perl modules, as some modules require extra parameters for building or have a different installation method. Furthermore, many Perl modules depend on other modules, so you need to specify explicit dependencies.

11 Finally...

Several more things are worth mentioning, such as:

- Split the package into subpackages whenever appropriate. Reasons for splitting large pieces of software into subpackages are:
  - For libraries, the software is usually split into a base package with shared libraries used by other programs and a development package (suffix **`-devel`**) with include files and static libraries. The latter does not need to be installed on non-development systems.
  - Software that includes both client and server side network services are often split into a base package, including the libraries for client software, and a server package (suffix **`-server`**) with the actual server (daemon).
  - Software that includes both a graphical and a non-graphical front-end might be split into a base package and a **`-x11`** subpackage. This makes the base package installable on non-graphical (server) systems, where X11-related dependencies usually can not be solved.

[4]This template can be used on Red Hat Linux 7.3, but it needs some modifications for newer releases.

- In case your package needs an init script, be sure it is compliant with the Red Hat Linux scripts and choose the correct start/stop numbers in the **`chkconfig`** comment line. In general, the first parameter of that line should be a dash, indicating that the service is not enabled by default at the moment the service is registered with **`chkconfig`**.
- Always use a proper build environment. If you generate a package in an unclean enviroment, like with wrong versions of certain libraries, the result is undefined.

12 Conclusion

Building rpm packages is pretty easy, building rpm packages in "the right way" is not easy, especially because a definition for "the right way" is partly a matter of personal taste and experience. Even the official Red Hat Linux spec files make different choices for the same problem at different places, depending on the person that maintains the package. If you want (or need) to write your own spec files, try to develop an organization-wide (or personal) set of guidelines, to ensure a consistent, distribution-compliant set of packages.