

Chapter 6

The Development Process

What Good Are Releases?

Most software users do not want to know what changes were made in the source code of the various releases of a particular product. Mainly, they want to get the software up and running right away and not worry about it until the next release comes out (at which point they can decide whether to upgrade or wait). In previous chapters, we said that free software projects are in a state of continuous release—the official, numbered versions occasionally announced by the developers are nothing more than snapshots of particular moments in a project's lifetime. Although that's accurate from a developer's point of view, users don't usually think of the process in this way. Compiling and installing from unstable development sources can be quite a hassle, and most people are understandably reluctant to deal with it. Instead, they'll get the software by downloading one of the "officially blessed" releases.

Developers therefore realize that their periodic releases need to be stable and relatively unthreatening to the user. Unfortunately, this runs counter to what most developers want. In a developer's dream world, every single user would update and reinstall the very latest software version every morning, use it all day long, and report every bug they find, and even send bug fixes.

From a developer's point of view, stability is not the point—finding the bugs and fixing them is the point. The user base would, in effect, function as a continuously running QA (Quality Assurance) department, exercising all aspects of the software and reporting all failures and unexpected behaviors in detail. In a user's

ideal world, by contrast, developers would organize legions of testers to try out every new version extensively before release, finding and fixing as many bugs as possible, and the release would always ship with no known bugs. (Of course, this is a goal that developers would also like to achieve, in the long run.)

In other words, everybody wants the bugs to be found and eliminated, but hardly anyone wants to go through the pain of finding them. Bug discovery can be painful for the discoverer—maybe you first noticed the bug when it caused a valuable file to disappear unexpectedly or sent a sensitive email to the wrong recipient. The developers welcome the news of your tragedy, because it allows them to fix a latent problem in the code. In a sense, they want you to encounter bugs—and the more the better. They'll probably even thank you in the log message for the bug fix (which is a small consolation for your missing file or insulted boss).

This conflict of interests between developer needs and user needs is not entirely resolvable. Users expect formal releases to be safe and as bug-free as possible. Developers have compromised to a degree, often imposing a feature freeze and a period of intense testing right before a major release. This accommodates users who don't want to deal with unstable software and, perhaps equally importantly, imposes a healthy discipline on developers, who otherwise might have no or little pressure to create stable software. Nevertheless, developers often regard the release process as an annoyance—a distraction from coding and improving the software.

Users have to make some adjustments, too: Instead of receiving sympathy when they encounter a bug, they're expected to file bug reports, which involves extra time in addition to the time lost by the bug itself. Users are also expected to understand that no release is completely bug-free and that developers rely on them to participate in the bug-finding process.

This chapter concentrates on the developer side of this process, although there are glimpses of the users' point of view as well. Because this material is directed toward people managing software projects, you might want to skip it entirely if you're just using CVS to manage text documents or other non–development-related files. We assume that you are familiar with the jargon of software development, the concepts of configuration and build scripts, and the Unix **make** program.

The release procedures described here are most appropriate for medium- to large-sized projects, although you might choose to apply some of the techniques to releases of smaller programs. In general, the more complex your project and the more users depend on it, the stricter your release procedures should be.

Starting the Release Process

A new release doesn't happen overnight. It is the culmination of a long process, usually starting with a discussion among the developers about whether it's time to release a new version.

Usually, there are two primary motivations behind a release:

- The developers have fixed many bugs since the last release.
- The developers have added significant new features.

A third reason, sometimes openly acknowledged and sometimes not, is public relations. If it's been a long time since the last release, people begin to wonder whether the software is still under active development. A release is a newsworthy event, a way of reminding the world that work on the software is continuing as usual. Publicity is, we suppose, an understandable motivation, but you shouldn't let it be the sole—or even the main—reason behind a release. If you start doing releases for attention, "release inflation" is the result. The perceived value of each new version decreases, and people start taking all of your releases less seriously. It is best to release for substantial reasons, such as bug fixes and new features.

Once a consensus has been reached that it's time to release, the developers decide on a release manager to guide the process. The release manager's job is not just a matter of packaging up the software, placing it on the appropriate download sites, and making an announcement. The manager must first coordinate testing, possibly enforce a code freeze (by carefully reviewing, and occasionally reverting, changes to the source code during the period leading up to the release), and in general be responsible for seeing that the new version is stable. For small projects, the release manager and the maintainer are one and the same person. The maintainer might simply not want the hassle of managing the release or might recognize that an independent, third-party judgment is needed to distinguish between unnecessary, destabilizing changes and legitimate pre-release bug fixes.

Avoiding the "Code Cram" Effect

If the release manager is not the regular maintainer, the maintainer should very publicly confer a hefty amount of decision-making authority on the release manager for the duration of the release process. For the manager, the most important asset is to be able to say "no," because when contributing developers realize that a new release is imminent, they tend to rush to finish up their current works-in-progress for inclusion in the next release. This, of course, is exactly what the release manager doesn't want or need. Hastily completed changes, checked in at the last minute with insufficient time for testing, are guaranteed to produce bugs that will be discovered by users after the release. In fact, often the very best time to put a new feature into the code is immediately *after* a release. It can then be tested for the maximum possible amount of time before the next release.

The author of this edition has seen this happening many a times with his own OpenSource project "openMosix". In the life-cycle of the project we have various release managers. We release a new version every 2-3 months and very often, the best releases are the ones when a long time transpired between the last commit to the CVS tree and the final release to the public. In order to make more people use a particular snapshot of a tree before releasing it officially, it helps to create a subsection in the project's website with daily snapshots, which are basically pre-compiled binaries of a particular CVS tree. These users will be able to test the software under a vastly more varied environment than the developers will ever be able to and new bugs will surface before the release.

Naturally, it's very hard to persuade developers to hold off. If they have a change 90 percent ready, they'll be tempted to finish the last 10 percent quickly and commit. They know that if the new code gets into the release, it will have many more testers than otherwise. However, users don't want to function as early testers and are unpleasantly surprised to encounter bugs that should have been caught before the release went out.

Unfortunately, the release manager can't simply refuse all code changes. The whole point of the release preparation process is to weed out as many bugs as possible, so code changes can't be avoided entirely. The manager must exercise judgment about which ones should be permitted and which ones should wait. Balancing the priorities of improvement versus stability is very tricky, and we can't really offer any firm guidelines about how to do it—it really depends on the nature of the software and of the proposed changes. In general, a change should probably be permitted if it fixes a known bug, is small, and seems unlikely to cause any dangerous new problems. However, if there is doubt, conservatism should rule. If a change disrupts an area of the code known to be currently stable and it's difficult for the release manager to tell at first glance whether the change has any problems, it's probably best deferred until after the release.

Freezing

The release manager's decisions will be best received if they appear to be part of a unified and consistent policy, rather than as case-by-case judgments. The policy most often used is called a *freeze* (because it cools down the code to a very slow rate of change for a while). Until now, we have been using the phrase "code freeze" as a generic name for all freezes; however, the term *freeze* can mean different things to different people. Here's a summary of the different uses of the term and what people mean by them:

- *Feature freeze*—No significant new functionality is added to the program, but bug fixes are permitted. Minor improvements are allowed as long as they are isolated and (theoretically) cannot destabilize other code. A "minor improvement" means a trivial change that doesn't involve new code paths or dependencies (for example, changing an error message or adding a **-help** option).

- *Code freeze*—No changes are to be made to the code except those absolutely necessary to fix known bugs. Even these may be deferred if the bug is minor and the best available fix involves changes that could have repercussions elsewhere.

Many people use *code freeze* synonymously with *feature freeze*. The difference between the two terms is fuzzy enough that we usually prefer to use these alternate terms instead:

- *Soft freeze*—This is similar to a feature freeze, but the decisive factor for permitting changes is how complex and destabilizing the changes are, rather than whether they implement a new feature or fix a bug. This essentially means, "Don't do anything big." The release manager has to review each submission carefully and ask, "Will the program be likely to cause someone a problem if this change is allowed?" The answer is largely a matter of opinion and subjective judgment, but the decision should be biased in favor of the safe, conservative course until the release is finished.

- *Hard freeze*—All code changes are discouraged. Only those that fix known bugs are permitted. Even with bug fixes, the release manager may chose to hold off if the bug in question is old and familiar to users, if its fix is complex, or if it is likely to have unexpected consequences. A hard freeze usually follows a soft freeze and is the last freeze before the actual release. If a bug is found, the code is warmed up to a soft freeze temporarily for the fix, then hard frozen again so it can go through a complete testing cycle without any changes.

Development vs. Stable Branches

Freeze policies are only guidelines, of course. Release managers are constantly called on to violate their own policy by making exceptions for special cases. However, if you make too many exceptions, you'll have to come up with justifications for all the cases for which you didn't make an exception.

One technique for sidestepping (at least partially) decisions about what to allow is to split the code into two branches: one for ongoing development and one for the stable release. Exactly when and where they split, and which one is on the main trunk, is up to the release manager. We recommend splitting off the release as a branch and leaving development on the trunk, so that developers can continue to work as they always have. As bugs are fixed on the release branch, the fixes should be merged back into the trunk (so those bugs will be absent from all future releases, too) and the release branch tagged as described in Chapter 4. When the release is made, the tip of that branch can be packaged and shipped as the new version. All activity on the branch will cease after any remaining changes have been merged into the trunk. When it's time for the next release, you can make a new branch, and the cycle starts again.

Having separate release and development branches does not obviate the need for a release manager, but it does mean that the draconian freeze policies need apply only on the release branch. Eager developers have a place to commit their changes without disturbing the code that's about to be released, rather than having to wait until the release is over before committing their changes.

The Two-Lane Approach

Some projects—most famously the Linux kernel—have taken this split approach to an extreme. They have a permanent stable branch and a permanent development branch. (Note that the Linux kernel is not using CVS to do this; we are not sure what mechanism they use to merge changes from one branch to the other.) When they decide it's time to make a new stable release, the development branch goes into a freeze—we think the Linux folks call it a *feature freeze*. When everything's been debugged, they take the frozen development branch, release it as the new stable branch, and start a new development branch immediately, even though the next release might be months or years away.

The important thing is that both branches are always available for download and installation by anyone at any time, whether or not there's a freeze or release planned. You choose which one to get based on your preferences: If you want safety, you go with the stable

version; if you want to help test new features (or perhaps you need some of the new features for some reason), you get the development version. The kernel team advertises loudly and clearly that the development kernels are less safe and that using them is riskier than sticking with the stable ones. Nevertheless, many people do use the unstable kernels, and the developers get a constant supply of free testers to find bugs for them.

This approach can work for any project sufficiently large that some users will try out the development branch, but it obviously involves some bureaucratic overhead. Life is always simpler when you minimize the number of branches active at any one time. With two permanent, parallel code lines, every change has to be considered in terms of which branch it's most appropriate for. If a change goes onto the stable branch, a cross-branch operation might be required to make sure it gets into the development code as well. We wouldn't recommend this approach unless the benefits of having extra testers outweigh the hassles of dealing with branch management.

Stability and Version Numbers

The Linux kernel also started a tradition that has since spread to other projects: They use even version numbers to refer to stable releases and odd numbers for experimental ones. The convention applies to only the minor version number—that is, the portion after the decimal point: Kernels 2.2 and 2.4 are stable, and 2.3 and 2.5 are unstable. Kernel 2.4, which has been out since early January 2001, is the stable release derived from the 2.3 development branch and will be the root of a new 2.5 development version. The CVS tree for 2.5 should be open to developers by the time this book is in your hands.

If you're going to use the two-lane approach of always having development and stable versions available, it's a good idea to follow this version number convention. Although the convention is not yet universal, more people have come to expect it, and many users might interpret your version numbers according to those expectations, whether that was your intention or not. You should state somewhere (on the project home page and in the documentation) that you're following the convention, so those who don't assume it's universal will know what to think, too.

Testing

In a perfect world, thousands of devoted testers would try out each release until no bugs remain. In the real world, testing is far too often a thankless task that developers usually end up doing themselves (and without time to be very thorough). However, having a team of dedicated people testing the code before the release greatly increases the odds of shipping stable software.

Getting and keeping this team is not easy. Testing is not, for most people, an inherently attractive task. It has none of the glory of code development, and it can be risky if you are testing with live data. However, if you can persuade people to do it, you'll be very glad. A good testing team will uncover problems you never even suspected.

Recruiting and Retaining Testers

To recruit testers, you should first post announcements on the appropriate mailing lists and newsgroups. Stress that the release process has already started and that you're planning to have the new version out by a certain date. In our experience, people always respond more positively to tasks that have a definite lifespan and deadline. Once you've collected some volunteers, here are some tips to help them understand their role, and come back next time:

- *Make it easy for your volunteers to download and install the test releases.* This means making the releases conveniently accessible both via CVS and via nightly downloadable "snapshots." The snapshots should be packaged just as the final release will be so that your team can test the installation process as well as the actual software application. (See the section "Building, Installing, and Packaging" later in this chapter.)

- *Organize the testing.* If the program is particularly complex, you'll want to assign people to test particular areas or subsystems. Keep the assignment list posted in a public place so everyone knows who's accountable for what. Whether or not people are assigned particular responsibilities, you should regularly query them individually on their progress. (If you haven't heard from them for a while, it might mean that they've found no bugs, or it might mean that they haven't actually tried the code yet.) If people feel that you notice their efforts (or lack of same), they'll devote more time. However, if you just throw the code onto a download site and don't give the testing team any deadlines or expectations, they'll probably make testing the last item in their priority queue.

- *Be responsive.* Every bug report from a tester should be answered right away, even if it's only to say that the bug has already been found and fixed. If testers are sending in vague or unreproducible reports, you might need to train them on what to include in bug reports. Nonprogrammers often don't realize that the core of most good bug reports is the "reproduction recipe," which gives the developers a reliable way to make the bug happen on demand.

- *Remember to thank your testing team, by name, in a prominent place.* For example, the top-level README file in the CVS source distribution contains a list of testers. Many other software distributions contain equally prominent acknowledgements. Also, when writing the log message for a bug fix, you should mention the discoverer's name. That might seem unimportant, but people really appreciate it and often contribute more when their work is publicly noticed. Recording their contributions in the permanent history of the code is a great way to give recognition.

Automated Testing

Depending on what your program does, it might be possible to automate some of the testing. Humans must still test any user-interaction code by actually typing at the keyboard, clicking the mouse, and so on, to assess the program's responses. However, you can, in theory, automatically test any circumstance in which the code takes discrete inputs and produces predictable output.

The exact implementation of automated testing is, of course, totally dependent on the program, so we won't go into detail about it here. To see a particularly complicated (to be charitable) example of an automated test suite, take a look at the Bourne shell script src/sanity.sh in the CVS distribution. If you do decide to implement automated testing, you should probably set the test script to run nightly on a fresh version of the sources, especially during periods of release preparation, and possibly even have the script email results to all of the developers and testers.

Automated testing is certainly useful, but it can never be a complete substitute for human testing. Many so-called "bugs" turn out to be deficiencies in the program's documentation or nonintuitive behaviors that surprise the tester. An automated test suite does not detect these problems because the program's developers won't make the same sorts of "mistakes" that an outside tester would, even when the design of the software encourages the mistakes. More generally, the problem with automated testing is that the developers—who usually write the test suite—are far too familiar with the intended use of each feature to think of unexpected and creative ways in which it might be misused.

Thus, test suites tend to be very good at confirming that what worked yesterday still works today, but bad at finding problems no one ever considered. Historically, the CVS test suite (the one with which we are most familiar) has caught some bugs during release preparation, but certainly not all of them. Once the CVS test suite utterly failed to notice a major bug before the release went out—or rather, the developers never anticipated the bug and so did not include code in the test suite to look for it.

Building, Installing, and Packaging

Certain standard automated methods have evolved to handle the highly repetitive cycle of compilation, installation, and packaging of source code. By adhering to these standards, you can make life much easier for other developers and testers, who expect everything to work in the conventional way.

Building and Installing: **make** and **autoconf**

The standard revolves around the Unix program **make** and, to a lesser degree, around GNU **autoconf**. For years, the **make** program has been the standard method of compiling from source code in Unix. The **autoconf** program is a more recent system, introduced by the Free Software Foundation, for dealing with portability issues across Unix variants. Unfortunately, documenting these two systems is quite beyond the scope of this book, but we will give very brief introductions. You can read more about the GNU implementation of **make** at **www.gnu.org/software/make/make.html**. Its online manual is available at **www.gnu.org/manual/make/**. You can find information about **autoconf** at **www.gnu.org/software/autoconf/autoconf.html** and its manual at **www.gnu.org/manual/autoconf/**. We assume that you're acquainted with tar and GNU zip.

Let's start by looking at how most free software is compiled and installed these days. You unpack the package as follows:

```
yarkon$ gunzip somesoft-1.2.tar.gz | tar xf -

yarkon$ ls
somesoft-1.2.tar.gz   somesoft-1.2/
```

Go into the top level of its source tree and configure it:

```
yarkon$ cd somesoft-1.2
yarkon$ ./configure
...
```

Next, compile it:

```
yarkon$ make
...
```

If you are installing on operating systems that are not GNU-based, such as AIX, HP-UX, Solaris, and similar systems, make sure to use a GNU **make** for open source programs, while keeping the local, non-GNU **make** program preinstalled with the OS. The reason for this is that some of those OS functions (such as kernel recompiling) require the proprietary **make** to work correctly. It is best to keep a **gmake** (for the GNU version) and leave the other **make** as it is.

Finally, to install it, do:

```
yarkon# make install
...
```

Let's examine those steps in (almost) reverse order. Typing

```
yarkon# make
```

at the command prompt invokes the **make** program, which in turn looks for a Makefile file in the current directory. The Makefile specifies how the program is to be compiled. It usually also contains specifications on how to install the program once it's compiled, uninstall it, and clean up the source tree after compilation, among other things. Running

```
yarkon# make install
```

tells **make** to find a rule (that is, a specification) named "install" in the Makefile, and do whatever the rule says to do—in this case, copy the newly built executables to the appropriate system location. (Actually, if binaries haven't yet been built, **make install** first goes

through the steps that a plain **make** invocation would have gone through and then installs the compiled software. Some people prefer the two-step process, however, just in case anything goes wrong during the compilation.)

The previous command

```
yarkon$ ./configure
```

ran a script named "configure," located in the top level of the source tree. That script reads in a meta-Makefile, named "Makefile.in," performs various textual substitutions in a platform-dependent way, and writes out the result as "Makefile" (without the .in suffix). Once you have the Makefile, of course, you can run **make** to compile the program.

However, that's not the whole story. The configure script itself was originally produced (by the program's distributors) from a file named "configure.in," which defines various parameters that tell configure what to look for, and the command to do that is **autoconf**:

```
yarkon$ ls configure.in
configure.in
yarkon$ ls configure
configure: no such file or directory
yarkon$ autoconf
yarkon$ ls configure
configure
yarkon$
```

In summary, **autoconf** is a system for producing a portable configure script that can be run on any platform to produce a *nonportable* Makefile. This Makefile can compile the software only on the platform on which configure was run. (This is a bit simplified, but it will do for our purposes.)

It is sometimes difficult to understand which of these files should be stored in the project repository and which should be generated locally, as needed. Obviously, the configure.in file should be in the repository, because it is not derived from anything higher up. However, should the configure script (which is derived from configure.in) be kept in the repository?

Your first instinct might be that it should not. Because the configure script can be generated on demand from configure.in by running **autoconf**, you might think it could only cause confusion to store it in the repository as well. However, if it is not distributed, anyone who wants to run configure (that is, anyone who wants to compile the program) must have **autoconf** installed on their system—and although **autoconf** is not uncommon, it is not as common as, say, the Bourne shell and the **make** program are.

Therefore, it's probably better to keep both configure.in and configure under revision control. You simply must make sure that whenever a change is made to configure.in, configure is regenerated and checked in as well. Changes to configure.in are fairly rare, so this works out okay in practice.

Meanwhile, the configure script itself does not depend on the presence of **autoconf**—it needs only the Bourne shell and various standard Unix utilities, which every version of Unix has. Therefore, anyone can run configure to generate a Makefile, given Makefile.in. Although it's vital that Makefile.in be kept under revision control because it cannot be derived from anything else, it's not necessary to store Makefile as well. In fact, because the Makefile is platform-dependent, storing it is not a good idea.

You might not need to autoconfiscate your program (yes, that's the official GNU word for incorporating a package into the **autoconf** system) at all if it's small and has fairly simple portability requirements. However, as your program grows in complexity, you will find that an increasing proportion of the code is devoted to dealing with portability issues. Although **autoconf** might seem daunting at first, it is the standard way to deal with portability problems these days. (To be perfectly fair, **autoconf** is probably as simple as it can be, given the inherent complexity and multidimensional nature of the problem it's trying to solve.)

If you don't use **autoconf**, it's fine to just have a Makefile with hard-coded rules for compiling and installing the program (you can instruct users to edit the Makefile if they don't like the defaults). Most people just type

```
yarkon$ make install
```

and not worry about it. Your testers will be grateful that you're following the standard, so they don't have to remember any unusual commands to install the software.

We have even seen some projects include a dummy configure script in their distributions that outputs something like this when run:

```
yarkon$ configure
No need to run configure, just type 'make install'
yarkon$
```

However, most people know to go directly to the **make** step if they see no configure script.

Let CVS Help You with Packaging

Automating the build and install process is useful for general users, but developers (and often testers) will also want an automated way to package the software for release. Although there is only one "real" release, there will probably be many test releases leading up to it, and it's important to have a consistent procedure for producing them.

If you've set things up in CVS in a reasonable way, it should be possible to configure, build, and install directly from a working copy. The only thing you might have to do is to turn that working copy into a compressed tar file suitable for distribution. The contents of that tar file will be the same as the working copy tree, but without the administrative CVS subdirectories. You could just write a script to remove all those directories, but that would ruin your working copy. Anyway, CVS provides a command to check out a project as a simple tree (not as a working copy, so it will have no CVS subdirectories).

The command is **cvs export**, and it's similar to **checkout**, except that it demands a tag name or date. The following shows how to create a tag and then export based on that tag (because some of these commands take place outside a working copy, we'll assume that the **CVSROOT** environment variable is set):

```
yarkon$ cvs -q tag Release_1_0
T README.txt
T foo.gif
T hello.c
T a-subdir/whatever.c
T a-subdir/subsubdir/fish.c
T b-subdir/random.c
yarkon$ cd ..
yarkon$ cvs -q export -r Release_1_0 -d myproj-1.0 myproj
U myproj-1.0/README.txt
U myproj-1.0/foo.gif
U myproj-1.0/hello.c
U myproj-1.0/a-subdir/whatever.c
U myproj-1.0/a-subdir/subsubdir/fish.c
U myproj-1.0/b-subdir/random.c
yarkon$
```

The **-d myproj-1.0** makes the exported copy go into a directory with a different name from the working copy (you don't want to destroy the working copy, because there's probably still work to be done there). Once you have the myproj-1.0 directory, it's a simple matter to package it up:

```
yarkon$ tar cvf myproj-1.0.tar myproj-1.0
myproj-1.0/
myproj-1.0/README.txt
myproj-1.0/foo.gif
myproj-1.0/hello.c
myproj-1.0/a-subdir/
myproj-1.0/a-subdir/whatever.c
myproj-1.0/a-subdir/subsubdir/
myproj-1.0/a-subdir/subsubdir/fish.c
myproj-1.0/b-subdir/
myproj-1.0/b-subdir/random.c
yarkon$ gzip myproj-1.0.tar
yarkon$ ls -l myproj-1.0.tar.gz
-rw-r--r--   1 jrandom  users        1611 Aug  9 02:43 myproj-1.0.tar.gz
yarkon$
```

It's easy to come up with scripts or Makefile rules to automate this process. A typical method is to start the process by invoking

```
yarkon$ make dist
```

which goes through the preceding steps and deposits the package myproj-1.0.tar.gz in the top level of the working copy, presumably for removal to an appropriate publicly accessible location. (By the way, although we chose not to in the example, you can run **export** inside a working copy, as long as the exported directory won't overwrite any subdirectories of the working tree.)

During the prerelease testing phase, there's no need to create (or delete) lots of spurious tags just to placate **export**. If you've frozen the trunk and just want to export its tip for each prerelease, you should just pass **export** a date—specifically, the special date **now**:

```
yarkon$ cvs -q export -D now -d myproj-1.0-beta myproj
U myproj-1.0-beta/README.txt
U myproj-1.0-beta/foo.gif
U myproj-1.0-beta/hello.c
U myproj-1.0-beta/a-subdir/whatever.c
U myproj-1.0-beta/a-subdir/subsubdir/fish.c
U myproj-1.0-beta/b-subdir/random.c
yarkon$
```

We have found, however, that using **now** can sometimes miss very recent changes in the repository, due to slight clock differences between the repository machine and the working copy machine. So to be perfectly safe, pass **tomorrow** instead

```
yarkon$ cvs -q export -D tomorrow -d myproj-1.0-beta myproj
...
```

counterintuitive as that might seem!

We won't describe the details of scripting the packaging process—There's More Than One Way To Do It, as the Unix motto goes. As long as the end result is a file with a name like myproj-1.0.tar.gz that unpacks into a directory named myproj-1.0/, you will be adhering to a widely recognized standard, and people will know what to do without having to look at your README file.

Releasing

If you've done all the preparation work right, releasing is a simple matter of putting the new version online for downloading and announcing it in the appropriate forums. If the program is a popular one, you should make sure that the primary download server is able to handle the load or that mirror sites are available if it can't.

Sometimes a prerelease—also known as a "beta" release—is done when the program appears to be approaching stability but still has a few kinks to be worked out. Beta releases are usually considered safe but not necessarily stable (as opposed to earlier "alpha" releases, which by convention are neither safe nor stable, and are meant only for the hardiest testers and early adopters). It is normal for beta releases to be distributed with the word "beta" somewhere in the version number, as in "myproj-1.0-beta." The fact that they have already been through a testing cycle makes them palatable to the general public, some percentage of whom usually downloads the beta version and starts using it. Once bug reports from the beta have slowed down to a trickle, it's normal to give it an official blessing and remove the "beta" from the name.

Telling the World about Changes

Most free software packages contain a file named NEWS at the top level, summarizing the changes from the previous release. Normally, the NEWS file is kept continuously up to date as new features are added, so there should be no need to edit it when the release process starts. Nevertheless, sometimes people forget to mention their changes, so it's a good idea to look it over before the release to see if anything important is missing.

The most recent portion of the NEWS file (the portion covering changes in the new release) is usually pasted into the release announcement that gets posted to mailing lists and newsgroups, so that people know before they download what improvements to expect. The format of NEWS files seems fairly standardized by now (see the one in the CVS distribution for a good example).

Recording the Release in CVS: Tags and Revision Numbers

You will definitely want to tag the CVS tree with some tag name, such as "Release_1_0", when the release is finally ready. This allows you to refer back to the released revisions later on—for example, when trying to reproduce bugs reported in the released version. Having a tag also gives people a way to retrieve or **diff** against a stable version of the software from their CVS working copies.

If you've incremented the major version number of the software, you might consider incrementing the major revision numbers of all the files in the project. Until now, all of the revision numbers we've used have been of the form "1.x." CVS provides a way to change the "1," but only in an upward direction (it is assumed that version numbers never move backward):

```
yarkon$ pwd
/home/jrandom/myproj
yarkon$ cvs commit -m "upping major version number" -r 2.0
 ...
```

This will add a new revision 2.0 to each file, and the contents of the new revision will be the same as those of the file's current highest revision (if any files in the working copy are not at the highest repository revision, the commit will complain and abort). Also, this command works only if all the files in the project are currently below revision 2.0 (but that's probably the case, because CVS has never crossed release 2.0 before).

Running a **commit** with the **-r** flag does have the unexpected consequence of setting a sticky tag "2.0" on everything in the working copy, making further commits impossible. Afterward, you'll have to run **cvs update -A** or check out a new working copy if you want to continue working on the project.

Incrementing the revision numbers this way is entirely optional. There's no actual need for any relationship between the program's version number and the revision numbers of its files. It's just a convenience for reminding developers on which major version they're working at any given time.

Finding Out More

The best source for current release practices is to participate in, or observe, someone else's release. In this chapter, we have tried to cover the process in some detail, but of course there's no substitute for the real thing. If you are a developer on other projects, watch carefully how those projects handle releases and make notes on what works and what doesn't. If you're managing your own release for the first time, just remember these main objectives:

- During the release preparation period, avoid changes to the code.
- Treat your testers well—very well.
- Fix all the bugs you can find.
- Once the software is released, make it easy for people to find out what's new and to get the software.

If everything you do is directed toward these goals, your release will go smoothly.