Software Design for Installability

Steve Simmons
Inland Sea

Abstract

In classic software design, little or no consideration is given to the issue of installing the resulting package on the customer machine(s). This is complicated by the rich variety of administrative policies and styles used in UNIX[1] installations. This paper will not attempt to prescribe a specific method for software installation. Instead, it will focus on (a) issues which make a package installable into highly customized sites such that the package can be installed with minimal disruption to both the site and the package and (b) the reprogramming of system configurations in a style designed to minimize the impact of that reprogramming.

Introduction

At first glance, the issues of software quality and software installability are almost completely distinct. There are high-quality software products which are trivial to install, and high-quality software products which are a nightmare. There are small, off-the-cuff programs on the net which are trivially easy to install, and others which give ones nightmare.

After fifteen years as both software developer and system administrator, I have found a number of features of software design which affect the installability of the end product. These are not large features, and during software development they usually seem to be trivial. This paper will attempt to point out what makes software installable and why one particular choice might be superior to another. It will also describe some of the system checks and precautions which a good install script will do.

There are standards efforts under way for software installation [Archer93], but these focus much more on the mechanics of installation rather than the qualities which make a package installable. The resulting standard will be an improvement over the current wild mishmash, but it will not in and of itself improve the quality of the installations. It will simply make them more predictable.

As a part of this, we will occasionally discuss a theoretical word processing package **WhizzyWimp**[2]. All of WhizzyWimps installation features will be based on features of existing software packages. Some of these are taken from good examples found in the real world, others are taken from fixes needed to install recalcitrant packages. The guilty will remain nameless.

Much of this paper will look at the issues of what makes for good installability. Once this is complete, we will turn to how those issues affect the development of the program itself.

What Is Installability

To be installable, a package must meet many criteria. Some are related directly to installation, while others cover the reconfiguration and use of the package after installation.

During the installation process, a well-designed package

[1] UNIX is a trademark of X/Open.

[2] What You See Is What I Mostly Programmed

- respects the existing technical and political policies of the the site;
- provides installation methods appropriate for both the naive end user and the sophisticated large-site manager;
- is resilient in the face of partial installation (that is, a failed installation can be recognized as such and backed out);
- makes minimal impact on the configuration of the machines on which it runs;
- does not attempt to enforce change of style or environment on the end users or systems managers;
- is resilient in the face of system customization;
- provides a clear indication of what has been installed and where;
- respects file permissions;
- can be upgraded in place without affecting the previous installation;
- permits the simultaneous installation and use of multiple versions;
- gives good error messages in the case of both internal and external failure;
- respects the security of the installation;
- provides good documentation for the system manager.

After installation, a good package

- is uninstallable;
- respects the security of the installation;
- provides good documentation for the system manager.

In summary, a good package has minimum impact on the rest of the system when installed. In actual practice, the total changes can be reduced to the amount of disk space consumed and the appearance of a single file in the location of a sites standard executables.

These are not cast in stone; there may be reasons to violate one or another of the above that are package-related. For example, numerous word processors have one-way conversion of old file formats to new formats. This should not, however, mean that users of the old form must upgrade if there is no need for sharing files.

The benefits increased installability are many. The most important for the supplier are:

- Ease of installation makes "test runs" and product samples much more acceptable. If a product is difficult to install, the likelihood of the installation completing successfully and the customer actually trying out the product are increased.
- Respect for the site (policies, uninstall, resilience) means reduced cost of ownership. This increases the value of the software to the end user and makes future sales more likely.
- Easier installation means greater likeliness of correct installation, reducing the cost of support.
- Many of the features which make a piece of software easier to install also make it easier to verify correctness at a later date; further reducing the cost of support.

In addition, the professional programmer is concerned with the quality of *all* work, not simply the source code. When building a package, the installation and cost of ownership are as important an issue as correct performance. The professional should be as concerned about these items as about user interface or any other developmental issue.[3]

Localizing Changes With A Shell Script

By making all system changes in a single area, one can minimize the effect of a change on a given system. This has several additional benefits, which we will discuss as well.

The ideal package consists of a single publicly installed executable which references an area owned only by that package. In that way the system is minimally impacted by the additional software, and the chance of packages interfering with each other is reduced to almost zero.

With WhizzyWimp, it turns out we have a package which consists of a number of separate executables. There is WhizzyWimp itself, the spelling checker, the PostScript[4] translation package, the index generator, etc, etc. All of these need to be installed in some area where WhizzyWimp can find them, but we want to avoid installing them in the public areas.

What we do is make WhizzyWimp itself begin with a shell script which references a configuration file. This particular model is based on the C News install [Collyer87], but it may well predate such use. The WhizzyWimp script is shown in Figure 1. Note that 90% of this file is error checking. Nothing is ever used before its existence is checked, and decent error messages are produced. It's a reasonable example of defensive shell script writing, with an overriding effort to describe the error condition rather than simply letting (non-)execution take its course.

Note the copious use of double quotes and curly braces. The double quotes in the `if` tests insure that uninitialized variables (whether from oversight or typographical errors) will not cause the test to abort. The curly brackets are defensive programming so that future modifications are less likely to affect the resilience of the script. The double quotes in the bodies of the error messages will capture errors from uninitialized variables and from variables which contain spaces or other whitespace.

There are other methods which can be used to test definition of shell variables. In the C News configuration we see constructs like

```
NEWSCTL=${NEWSCTL-/usr/lib/news}
```

While this is both correct and portable, it requires the reader to know shell variable construction rules fairly intimately. The `if` construction is immediately obvious to anyone with the slightest experience in programming, and hence preferable.

The most critical portion is the line

```
. ${WHIZZYWIMPCONFIG}
```

This is the invocation of the configuration script itself. By using the **.** directive, we can make changes in the local shell script environment. As we add more and more programs to the WhizzyWimp package, each of them refers to this configure script rather than embedding the definitions individually in the programs. When a system change is needed, a single change to the configuration file updates all programs immediately.

The configuration file show in Figure 2 follows similar paranoid principles. Note that it also error checks for the existence of every file and produces not simply an error message about what is missing, but a message about whether it is the standard or a locally defined alternate which is missing.

The configuration file is also executable on its own, so it may be tested without having to invoke the actual application.

---

[3] And packages which are easy to install and maintain often result in gifts of beer at conferences such as this.

[4] PostScript is a trademark of Adobe, Inc.

```
#!/bin/sh
DEFAULT_WW=/usr/local/lib/WhizzyWimp/Version3.1/ww.config
if [ "${WHIZZYWIMPCONFIG}" != "" ] ; then
      ERRMSG="Your sites locally defined"
else
      WHIZZYWIMPCONFIG=${DEFAULT_WW}
      ERRMSG="The standard"
fi
if [ ! -r "${WHIZZYWIMPCONFIG}" ] ; then
      cat << EOF
$ERRMSG master definition file for WhizzyWimp,
"${WHIZZYWIMPCONFIG}", seems to be missing.

Please let your system manager know.  WhizzyWimp
cannot run until the file is available.
EOF
      exit
fi
WW_EXEC=${WHIZZYWIMP_HOME}/ww
if [ -x "${WW_EXEC}" ] ; then
      exec "${WW_EXEC}" $@
else
      cat << EOF
The WhizzyWimp executable, "${WW_EXEC}", seems to
be missing.  Please let your system manager know.
WhizzyWimp cannot run until the file is available.
EOF
fi
```

Figure 1

```
#!/bin/sh
DEFAULT_WHIZZYWIMP_TOP=/usr/local/lib/WhizzyWimp
if [ "${WHIZZYWIMP_TOP}" = "" ] ; then
      WHIZZYWIMP_TOP="${DEFAULT_WHIZZYWIMP_TOP}"
      ERR_MSG="The default"
else
      ERR_MSG="Your locally defined"
fi
if [ -d "${WHIZZYWIMP_TOP}" ] ; then
      :
else
      cat << EOF
$ERR_MSG master directory for WhizzyWimp, "${WHIZZYWIMP_TOP}",
cannot be found.  WhizzyWimp cannot run without that directory
and its contents.  Please inform your system manager.
EOF
      exit
fi
DEFAULT_WHIZZYWIMP_VERSION="Version3.3"
if [ "${WHIZZYWIMP_VERSION}" = "" ] ; then
      WHIZZYWIMP_VERSION="${DEFAULT_WHIZZYWIMP_VERSION}"
fi
WHIZZYWIMP_HOME="${WHIZZYWIMP_TOP}/${WHIZZYWIMP_VERSION}"
if [ -d "${WHIZZYWIMP_HOME}" ] ; then
      :
else
      cat << EOF
$ERR_MSG master directory for WhizzyWimp ${WHIZZYWIMP_VERSION},
"${WHIZZYWIMP_HOME}", cannot be found.
WhizzyWimp cannot run without that directory and its contents.
Please inform your system manager.
EOF
        exit
fi
```

Figure 2

All subsidiary programs and files which WhizzyWimp requires should be installed in `${WHIZZYWIMP_HOME}`. The structure under this tree can be as complex as desired. As long as all shell scripts begin by invoking the `${WHIZZYWIMPCONFIG}` file and all child processes are started from there, the tree can be placed anywhere. Any time system installation requires moving the tree, the system manager can do so without fear of breaking WhizzyWimp.

Since the invocation of WhizzyWimp manipulates the environment, it might be tempting to modify the users `PATH` in the configuration file. Don't! This can result in ugly and subtle errors. WhizzyWimp executables, both the master and any slaves, should reference the environment variables to build paths for any subsidiary executable they invoke. This will avoid problems of name collision.

This method has the additional benefit of allowing installation of multiple versions of WhizzyWimp. Each version gets its own tree under the master WhizzyWimp tree. Individual users can simply put the appropriate version number in their environment, and the right thing will happen. This allows the system manager to install a new version of WhizzyWimp without disturbing old versions. At some future date, the manager decides the new version is stable and deletes the old version or makes the directory inaccessible. Users of the old version will get a useful error message when this occurs. This also simplifies uninstallability, which we'll return to later.

We have reduced the number of changes visible to the average user to one – the installation of the master WhizzyWimp script in some (any!) normally searched public area. We have not had to modify a users `PATH` or `.cshrc` or `.login` files – always a risky proposition anyway, as we have no reliable way of predicting what shell a user uses or the method by which it is invoked on a given system.

This method permits the sophisticated administrator to take very direct control of the installation location of WhizzyWimp. The default installation method will use the default directories as indicated, the expert method will permit the local administrator to make major changes without affecting the basic run style of WhizzyWimp.

One complicating factor of this method is that it requires the install script build the shell script on the fly. This is a long-solved problem. The reader is referred to the C-News or INN source [Collyer87,Salz92], or any Cygnus install package for examples. Two tools of particular note are the Cygnus configuration utility [Pixley92,Pixley93,Cygnus93] and Larry Wall's Meta-Config [Wall].

When using shell scripts for configuration, one wants to avoid needless repetition of configuration. Since WhizzyWimp has a stand alone spelling utility, we would like a reliable method of determining if the configuration has already been done.

The simplest method is to create an environment variable which is used exclusively for that purpose and check it at the beginning of the configuration file or before doing the invocation of it. Thus one would change the beginning of the config file to

```
WHIZZY_WIMP_CONFIGURED=0
```

and a check for `WHIZZY_WIMP_CONFIGURED` would be done before invoking the config file.

Modification of User Setup Files

In most situations, modifications to user setup files are both unneeded and dangerous. This tends to be the area where most sites do extensive customization, and it is the area where conflicts are most likely to occur[5].

The already described method of a standard configuration file with a conditional invocation applies equally well to personal initialization files like `.cshrc`, `.login`, `.profile`, etc. Given the configuration method we have shown above, it should not be necessary. But if it is needed, do it right.

Modification of Existing System Files

We have already shown that it is not necessary to modify users personal setup files. Unfortunately, sometimes a package requires changes system boot files or configuration files. This can lead to some interesting problems. Curiously, the better-managed the site, the more likely the problem. Consider the following two real-world examples.

At the Industrial Technology Institute, we carefully placed all system configuration files under control of RCS. This has the same benefits for system management that it does for source code control. It also was a time bomb for one particular package which added two lines to

---

[5] One CAD/CAM package which will remain unnamed requires over 300 lines of addition to `.cshrc` and `.login` files and includes multiple(!) lines like `set path = ( a b c )`, utterly destroying the users carefully constructed `path`. Worse, the installing site consisted of `ksh` users. Worst, the lines to be added contained several bugs.

/etc/services and a ‘paragraph’ to /etc/rc.local. When the system manager went to modify one of those files, he checked out new copies from the RCS archive, made the changes, and installed the modified file, deleting the changes the install had made. Bringing back the install-time changes required restoring files from tape – not a favorite task of any administrator – and then adding them into the RCS archive. The problem occurred several times with several packages before it was diagnosed; now the managers routinely do rcsdiff even on what appear to be pristine files before checking out new copies for modification.

A similar problem occurred at sites which run file integrity checking packages for security or rdist [Cooper92] for distributed file management. In the first case, the integrity package begin reporting security violations on the rc files as soon as the package was installed. Recovery involved rebuilding the integrity database. In the second, the rdist update undid the install changes. Unlike the situation at ITI, the changed file was not backed up and hence not recoverable. A new install of the product was required.

The situation is further complicated when multiple packages are installed in quick succession. Package A modifies a file and renames it to file.bak. Package B modifies the same file and renames it to file.bak. The original file is now lost.

There is some simple methods for dealing with this. The install package should come with checksums for all system files for which it intends to modify. (Of course, this assumes the install has been checked in advance on all systems for which you are selling it.) A pristine system should be obtained and checksums computed for all files which will be modified. At install time, those files should be checksummed. If there is no match, it’s extremely likely that either the file has changed or the install is being done on a new type of system. In either case, the install should halt and report the problem. *Blindly modifying already changed files is never acceptable.*

In addition, the scope of the changes should be strongly restricted. Adding ports or daemons to inetd.conf, services, and other simple tables is fairly simple and hard to do wrong. The various rc files are much more complex, and are discussed in a separate section below.

File permissions should be checked before making changes. RCS and other similar systems make the controlled file unwritable. Running as root, most packages simply ignore writability and blast away. *Check first!* If inetd.conf is permitted 444, the system manager is telling you something.

RC Files

An error in an rc file can make an system unbootable. In spite of this danger, a number of installation methods are quite cavalier about how they modify one or more of the rc files.

The failures largely fall into two groups: bugs in shell commands inserted into the rc file, and excessive changes to the file[6].

System V rc files avoid the latter by isolating the changes into a directory of rc subfiles which are executed in an easily-specified order. The master rc file is careful about how it executes those files, and thus is resilient in the face of failure.

Berkeley-based systems are not so lucky. A number of changes have been proposed [Nieusma,Romig91,Simmons91], the reader should consult [Romig91] in particular for good suggestions. While the install package cannot make wholesale changes to the rc files, whatever changes are made should be done in accordance with the suggestions for good quality rc files.

In general, do not take the existing style or methods you see in the vendor-supplied rc as examples of how to do things. The quality of standard rc files is low.

[6]Although I have seen one install which simply mangled the rc file.

For the installer, the choice is a somewhat simpler one. Use the System V style of a number of small, individual rc files which are invoked by one of the master rc files. In the master, make a minimal change as shown in Figure 3.

Once again, we have defensive shell programming with checking for a file of the correct type. Some particular points to note:

- The use of mixed quotes in the first echo will reveal such programming errors as getting variable names wrong or putting spaces or newlines in the variable.
- Forcing all output (including errors) to /dev/console ensures that error messages get where they belong.
- Forcing the execution of the rc file into the background ensures that any errors in it will not affect the parent rc script.
- The check for the existence of the product-specific rc file ensures that the product and its rc file can be removed from the system without requiring modification of the master rc file.
- The entry both begins and ends with a comment such that it's scope can easily be determined by the administrator. Good administrators do this with all their modifications to rc files; the install programs should do the same.

This last point, appropriate comments with begin and end markers, also applies to any other system tables which are modified. However, one should not assume those markers will still exist at uninstall time. One uninstall script looked for the begin marker, found it, and deleted everything from there to the end marker or end of file, whichever came first.

Diskless, Dataless and Diskful Nodes

Many installation scripts assume the hardware or software installed resides on the same machine as the install media or the execution of the install script. This is often not the case. It is not at all unusual for a systems /usr partition to be NFS-mounted read-only from one system, for /usr/local to be read-only from another, and for the installation media to be attached to a third system. These become formidable problems to which there is no general solution. The best one can do is design the package in such a way that installation can be performed in a series of steps

```
# Beginning of added material for WhizzyWimp
#
WW_RC_FILE="<name of file here>"
if [ -x "${WW_RC_FILE}" ] ; then
        "${WW_RC_FILE}" > /dev/console 2>&1 &
else
      echo 'WhizzyWimp startup file "'${WW_RC_FILE}'" not found or' \
            > /dev/console 2>&1
      echo 'not executable.  Boot continues.' \
            > /dev/console 2>&1
fi
#
# End of added material for WhizzyWimp.
```

Figure 3

appropriate to the various environments.

Extracting data from remote media should be possible without requiring remote `root` access. An un-privileged account should be able to extract file sets using `tar`, `cpio`, or whatever is appropriate. The files containing the extracted sets can then be made read-accessible across the network to the install program.

There are four primary areas where files may need to be installed:

- The root partition. This is always writable on the local station by the local `root`.[7]
- The `/usr` partition. This is often mounted read-only from a master server, and requires root access on the master server.
- The `/usr/local` structure (which may not actually be local). This is sometimes local and sometimes read-only from another server.
- The install directory for most executables. This may be either local or remote.

Modifying each of these disk areas may require accessing the install media from four different systems, sometimes with and without `root` access. The only way to successfully deal with the problem is to break the install process internally into four steps which can be performed individually by the appropriate system managers if needed. Thus there would typically a separate set in the install media for each of the areas the install modifies.

Fortunately, installing device drivers onto diskless dataless workstations is an extreme case. On the other hand, it doesn't have to happen often to prevent sales of the particular hardware or software. The install software should be designed to detect which (if any) of these situations apply and generate the appropriate messages. A relatively small amount of effort results in an install which seems no different in the standard case but handles the extremes well.

Documentation Of Changes

A good installation will include complete documentation of all changes to the system. This documentation should cover files added, files changed, and preserved files.

Many packages include a `Manifest` file which lists all files installed. This is useful but should go much further. The manifest file should list all files, their ownerships and group memberships, permission modes, and a checksum of all files which should be invariant. Ideally there would also be a script included which the administrator could run at any time to determine if any files have changed and how, and note missing and added files. A number of existing installation methods (Digital Equipment, UNIX V, proposed POSIX standard) come quite close.

When system files are changed, three particular items should be documented. First, the reason for the change should be noted. This should be done in a `ReadMe.InstallNotes` file. The file should also include comments describing the manifest, the uninstall method, and the next two items.

Second, a copy of the original unmodified file should be kept. Great caution should be used here! It is not sufficient to copy `/etc/rc.local` to <*some-dir*>`/rc.local`, as subsequent reinstallation will wipe out the saved copy. The copied file should be named `rc.local.YYMMDD.HHMMSS` where `YYMMDD.HHMMSS` is the full date as printed by the standard UNIX command `date '+%y%m%d.%T'`. This particular choice ensures that an `ls` of the directory will show all the files in age order. The truly paranoid will put the century as well.

Finally, a patch should be generated which will undo the change. As subsequent package installs and system modifications are made the patch will no longer work in it's literal form.

[7] Some configurations literally rebuild the root partition at every boot. In this case installation of software which requires modification of files in `/etc` is best left to the manual intervention of the site managers.

However, the patch file can easily be modified to change the line number and can become a tool to allow the experienced administrator to either undo the changes made or re-apply the changes if lost.

The typical inexperienced administrator or end user will never need or see the `ReadMe` files. Their purpose is not for day to day use, but for dealing with errors. The experienced administrator will look for it immediately if a problem is found; and the inexperienced will eventually stumble over it.

Dealing with multiple simultaneous installations

A properly designed software package should permit the simultaneous installation of multiple versions. This is often temporarily necessary for the conversion period between versions, and sometimes is needed indefinitely for the support of legacy systems. One of WhizzyWimps inspirations is a classic example of this. A read-only version of WhizzyWimp is available to be packaged with other software. This allows the package vendor to include very nice on-line documentation, but leads to interesting complications. The package vendor may support products on systems (eg, Sun 3s running SunOS 3.5) where up-to-date versions of WhizzyWimp will no longer run. This is fairly easy to deal with at the end user site, but a nightmare for the package vendor.

The use of the WhizzyWimp version settings in the configuration files is one way to handle the problem. The package vendor has multiple versions of WhizzyWimp installed, each neatly isolated into directories named after the version. Most users do not define a WhizzyWimp version number in their environment, and hence always get the latest version.

Another form of multiple simultaneous installation comes from having multiple processor and/or operating system types in a network. Here there are two methods, both of which work well. In both cases, one first isolates the executables for given OS/processor into appropriately named directories.

For the first case, can add a test to the configuration file to dynamically determine which OS and processor is in use and and create the appropriate pathname. into particular directories. This complicates the configuration file somewhat, but is amenable to simple analysis and test.

The other is more subtle, and is taken from the FrameMaker installation. In this method, all scripts are symbolic links to a script called `.wrapper` The `.wrapper` script dynamically determines OS and processor type, and invokes an executable from the appropriate directory. The name of the executable is taken from `$0` in the environment, almost always the same as the symbolic link to `.wrapper`.

As shipped by Frame, the `.wrapper` script is functional for FrameMaker but is not general enough to be used by most other packages. While it could be extended, there seems to be no significant benefit to using it over the simple configuration script method.

Uninstallability

For a product to be uninstallable, there must be careful tracking of the location of all installed files and all modifications made to other system files. Here is where the manifest files, dated backup files and patch files come into their own.

The uninstall script should *not* blindly copy back the modified system files, nor should it ignore them. It should apply the patch file to a copy of the saved copy (thereby reversing the changes) and compare the result to the current installed copy. If they differ, there have been subsequent changes to the system file. These changes would be lost if the old versions of the system file were simply copied back. Instead, a message should be generated by the uninstall script stating

[8] The Frame `.wrapper` script cannot be reproduced here due to copyright restrictions.

that the files are restored to their pre-install version, and directing the uninstaller to the directory containing the patch and install `ReadMe` file.

Another caution for uninstallation is the presence of multiple versions. If one is simply removing a now-outdated version, it is not a good idea to delete the master shell script or the socket numbers from `/etc/services`.

Note, however, that is is perfectly safe to delete the executables and library files in the directory and the specialized `rc` file which is invoked by the master `rc` file. The modification done to the master `rc` file checks for the existence of the specialized `rc` file, prints an appropriate message, and continues with system boot.

Inevitably, someone will eventually notice the message and investigate. For ease in that investigation, the uninstall process should write a message in the directory with the `ReadMe` file informing future administrators of "pending" changes to the files.

System Security

The security issues which have already been discussed focused on the issues of modifying files in a distributed network. Unfortunately these are the easy issues.

Many packages require the use of a dedicated ID for file ownership and set user ID programs. This is a perfectly acceptable practice, but great caution should be used in the implementation.

One cannot rely on a given user login id or UID number to be available on any system. Name conflicts[9] are inevitable and can happen on systems of any size. If your product is popular enough, you will eventually find one. A number of large sites are already facing exhaustion of UID space [Doster90] (as few as 32765 on some systems), so choosing a fixed UID number will inevitably involve a collision as well.

The only general solution is to allow the product login id to be modified by the site manager at install time and modified again later if needed. This modification would be made in the system install files The running executable would then do the appropriate `getpwnam(3)` calls rather than depending on compiled-in parameters of any type.

With a user id defined for the package, one can do a great many useful things with `suid` programs which will not compromise system security. Unfortunately these require careful programming. A full discussion of this is outside of the scope of this paper. [Simmons90] examines many of these issues; a careful examination of the C News or INN source will yield useful examples. The `lpr` system is a classic bad example.

Many packages attempt to work around the issue by installing suid root programs. This should *never* be done on a wholesale basis. As system administrators become and sites in general become more concerned about security issues sites will simply refuse to install such programs as unacceptable risks.

Other packages take a different tack – they make all files globally writable, then depend on internal equivalents of access control lists to manage data integrity. Typically such packages have been ported from non-UNIX systems to UNIX and the developers were either ignorant of how UNIX file and suid systems worked or were not given sufficient time to do the correct work[10].

---

[9] Around 1982 the Los Angeles phone book contained a listing for the Ingres family, and the last name Root has caused interesting issues on some systems.

[10] There is one popular mainframe accounting package which has been ported to UNIX which does exactly this. It has now been two years since the problem was reported and they have failed to fix it.

Adding IDs to a system is a surprising difficult task. At a minimum the installation script should ask before doing it; it is far preferable to have the administrator set up the id first.

As a final note on IDs, everything which has been said about user ids applies to group ids as well.

Some administrators will not blindly execute *any* script as root. Make your scripts straightforward, whenever possible using standard utilities.

Porting From Non-UNIX Systems

Many of the problems discussed can be laid at the feet of programs which have been ported from non-UNIX systems. These programs and their operation usually have many fundamental assumptions about how the operating system works. When the port effort starts, the developers and administrators take advantage of UNIXs customizability and make their UNIX systems look as much as possible like their original systems. This does minimize their own difficulties and seems to reduce the slope of the learning curve of a new OS. In reality, they have not gotten up the curve at all. They have merely delayed encountering it until the package is ready to install at a UNIX-savvy site. At that point the curve becomes a brick wall and the product falls apart.

The whole world is not UNIX, or VMS, or MVS. The techniques which have been discussed here are specific to UNIX, but their underlying principals are *general*. We are now ready to articulate them and how they relate to software development.

Localization of Product Information

A general interface for getting locations of files and the values of settable items should be developed. A generalized `getsetting()` function should be so than the high level interface is independent of the low-level implementation. Such an interface works equally well for UNIX environment variables, VMS logicals, data from flat files, or compiled-in tables. The underlying mechanism is unimportant. What is critical is an organized method of retrieving the data from a well-managed store rather than compiling values into multiple locations in a program.

Verification of Configuration

Once obtained, data should not be trusted until tested. Just as the configuration script checks to see if given directories exist, the initialization portion of the program should check the environment retrieved by `getsetting()` against the actual system environment. Far too many programs halt with the simple message "could not open library". This is inadequate. The error message should indicate which library, the name of the file, and the type of error (permission denied, missing file, etc).

Documentation

The documentation provided must be two-fold. First, standard documentation must be available for the administrator who is attempting to debug a possibly defective product installation. Without knowledge of what a proper installation is, the administrator can never be sure that the installation is correct.

Second, the installation process must provide some dynamic tracking of what it does and preserve that tracking in a reasonable location. Messages printed at install time are useful, but must be supplemented with logs, message files, and sometimes even recordings of the install process.

Conclusion

Two of the most important principles of good programming are information hiding and well-defined interfaces with disciplined use. These apply equally well to the installation of software. Follow them for the installation, and the resulting package will be improved.

Other Commentary

The previously referenced POSIX standard [Archer93] is a must. FTPable drafts are available from `dcdmjw.fnal.gov` in the directory `/posix/1387.2`. As of this writing, the 12th draft is now available.

An informal *Software Installation Workshop* was conducted by Paul Anderson at the 1992 Large Installation System Administration Conference. Notes from the workshop are in [Anderson93], and a mailing list formed as a result can be reached at `soft-managers-request@nas.nasa.gov`.

References

[Anderson93]: Paul Anderson, *Software Installation On Large Systems*, **;login:**, March/April 1993, Volume 18, No. 2.

[Archer93]: Barrie Archer, *Towards a POSIX Standard for Software Administration*, Proceedings of the Large Installation Systems Administration Conference, 1993.

[Collyer87]: Geoff Collyer and Henry Spencer, *News Need not be Slow*, Winter USENIX Conference, 1987.

[Cooper92]: Michael A. Cooper, Overhauling Rdist for the '90s, Proceedings of the Large Installation Systems Administration Conference, 1992.

[Cygnus93]: David MacKenzie, Roland McGrath, and Noah Friedman, **Autoconf: Generating Automatic Configuration Scripts**, Cygnus Support documentation.

[Doster90]: William A. Doster, Yew-Hong Leong, and Steven J Mattson, *Uniqname Overview*, Proceedings of the Large Installation Systems Administration Conference, 1990.

[Nieusma]: Posting of rewritten Sun `rc` files to `comp.sys.sun`.

[Pixley92]: K. Richard Pixley, **On Configuring Development Tools**, Cygnus Support documentation.

[Pixley93]: K. Richard Pixley, **Cygnus Configure**, Cygnus Support documentation.

[Romig91]: Steve Romig, *Some Useful Changes for Boot RC Files*, Proceedings of the Large Installation Systems Administration Conference, 1991.

[Salz92]: Rich Salz, *InterNetNews: Usenet transport for Internet sites*, Summer USENIX Conference, 1992.

[Simmons90]: Steve Simmons, *Life Without Root*, Proceedings of the Large Installation Systems Administration Conference, 1990.

[Simmons91]: Posting of rewritten Sun `rc` files to `comp.sys.sun` and Ultrix `rc` files to `comp.unix.ultrix`.

[Wall]: Larry Wall, *metaconfig(1)* manual page, *dist2* package, posted in `comp.sources.unix`, Volume 16, 1988.