Designing Integrated High Quality Linux Applications

Avi Alkalay

avi at br.ibm.com
avi at unix.sh

Senior IT and Software Architect :: OpenSource/Linux Solutions
IBM Linux Impact Team :: ibm.com/linux

v1.9.9, 2002−04−20

Revision History

Revision 1.9.9	20 Apr 2002	Revised by: avi
Included other document locations		
Revision 1.98	14 Apr 2002	Revised by: avi
Title changed from "Creating" to "Designing"		
Revision 1.97	09 Apr 2002	Revised by: avi
Converted to XML 4.1.2, and started to use real XSLT. Spell checked the english version.		
Revision 1.96	23 Mar 2002	Revised by: avi
Better HTML style sheets.		
Revision 1.95	17 Mar 2002	Revised by: avi
Last chapter: One Body, Many Souls. Created appendix. Still have to translate some words here and there.		
Revision 1.9	16 Mar 2002	Revised by: avi
Added universal software table with FHS		
Revision 1.7	16 Mar 2002	Revised by: avi
Everything is now translated except some words.		
Revision 1.3	27 Feb 2002	Revised by: avi
Translated and reviewed the most important section of the article: The /opt and /usr/local section.		
Revision 1.2	23 Feb 2002	Revised by: avi
English translation at 65%. Doing some corrections to potuguese version also.		
Revision 1.1	17 Feb 2002	Revised by: avi
Started english translation.		
Revision 1.0	16 Feb 2002	Revised by: avi
First final version of proposed skeleton.		

Revision 0.9.6	16 Feb 2002	Revised by: avi
Finished Plugin chapter.		
Revision 0.9.5	15 Feb 2002	Revised by: avi
Finished chapter about boot and subsystems.		
Revision 0.9.4	14 Feb 2002	Revised by: avi
Finished chapter describing the boot process		
Revision 0.9.3	08 Feb 2002	Revised by: avi
Text and style updates		
Revision 0.9.2	07 Feb 2002	Revised by: avi
Text updates		
Revision 0.9	06 Feb 2002	Revised by: avi
First translation to DocBook		

Table of Contents

1. Introduction

Linux is becoming more and more popular, and many Software vendors are porting their products from other platformas. This document (article) tries to clarify some issues and give tips on how to create Linux applications highly integrated to the Operating System, security and easy of use.

The examples run on Red Hat Linux, and should be compatible with other distributions based on Red Hat (Conectiva, Turbolinux, Caldera, PLD, Mandrake, etc).

2. User Friendly: Guaranteed Success

The *user−friendly* concept is missassociated with a good GUI (graphical user interface). In fact, it is much more than that. In systems like Linux (with more server−like characteristics), the user measures how easy a Software is, mainly in the installation and initial configuration. He can even forget how easy were to install and use a certain product, but it will never forget that a Software package has a complex configuration and installation process. A migration or new installation allways will be a nightmare, making the user avoid it.

2.1. Embrace the *Install−and−Use* Paradigm

Imagine you'll install that expansive product your company bought from ACME, and realized you'll have to do the following:

1. To have a manual that shows the installation process step−by−step. We know that a manual is the last thing the user reads
2. Read some README files
3. Uncompress huge files in your disk (after downloading them from net our CD), to create the installation environment
4. Read more README files that appeared in the installation environment
5. Comprehend that the installation requires you to execute in a special way some provided script (the inconvenient `./install.sh`)
6. Uncomfortably answer some questions that the script does, like target directory, user for the installation, etc. To make it worse, it frequently happens in a terminal that has a missconfigured backspace
7. After the installation, configure some environment variables in your profile, like `$PATH`, `$LIBPATH`, `$ACMEPROGRAM_DATA_DIR`, `$ACMEPROGRAM_BIN_DIR`, etc
8. Edit OS files to include the presence of the new product (e.g. `/etc/inetd.conf`, `/etc/inittab`)
9. And the worse: Change security permissions of OS directories and files to let the product run OK

Sounds familiar? Who never faced this sad situation, that inducts the user to make mistakes? If your products' installation process sound like *Uncompress−Copy−Configure−ConfigureMore−Use*, like this one, you have a problem, and the user won't like it.

Users like to feel that your Product integrates well with the OS. You should not demand that the OS adapt himself to your Product (changing environment variables, etc). It must let the user ***Install−and−Use***.

The *Install−And−Use* glory is easily achieved using a 3 ingredients receipt:

1. Understanding the Four Universal Parts of Any Software
2. Understanding how they are related to Linux's directory hierarchy
3. Aggressively use a package system, for process automation and leverage first items. In our case is RPM.

We'll discuss here what are these ingredients and how to implement them.

---

3. The Four Universal Parts of Any Software

The file set of any Application Software, graphical, server−side, commercial, open/free, monolithic etc, has allways four universal parts:

1st :: The Software on its own: the body

The executables, libraries, static−data files, examples, manuals and documentation, etc. Regular users must have read−only access to these files. They are changed only when the system administrator makes an upgrade in this Software.

2nd :: Configuration Files: the soul

These are files that define how the Software will run, how to use the Content, security, performance etc. Without them, the Software on its own is usually useless.

Depending on your Software, specific privileged users may change these files, to make the Software behave as they want.

It is important to provide documentation about the configuration files.

3rd :: Content

Is what receives all the user attention. Is what the user delegated to be managed by your Product. Is what makes a user throw away your product and use the competitors', if it gets damaged.

Are the tables of a database system, the documents for a text editor, the images and HTML pages of a web−server, the servlets and EJBs of an Application Server, etc.

4th :: Logs, Dumps etc

Server Software use to generate access logs, trace files problem determination, temporary files etc. Other types of softwares also use this files, but it is less common.

It is the last class of file, but many times they are the most problem generator for a system administrator, because their volume can surpass even the content size. Due this fact, it is important for you to think in some methodology or facility for this issue, while you are in design time.

3.1. Practical Examples

Let's see how universal is this concept analyzing some types of softwares:

Table 1. Universality of 4 Parts

	Software on its Own	Configurations	Content	Logs, Dumps etc
Data Base Server	Binaries, libraries, documentations.	Files that define the directory of the data files. For this type of Software, the remaining configurations usually are in special tables inside the database.	Table files, index files, etc. This software use to have whole trees under the same directory. And many times they need several filesystems to guarantee performance. Their local in the system is defined by they Configurations.	For DBs, there are the backup, generated in a daily basis. And the logs are used by the DBA to define indexing strategy. His local on the system is also defined by the Configurations.
Text Processor	The same, templates, modular file format filters, etc	As a user–oriented Software, its configurations must be put in each user's `$HOME` directory, and are files that defines standard fonts and tabulation, etc.	The documents generated by the user, and they go some place in his `$HOME`	They show as temporary files that can be huge. User can define their location with a user–friendly dialog (that saves it in some Configuration file)
MP3 generator	Same, audio modular filters	Each user has a configuration file in his `$HOME`, and contains bitrate preferences etc	Similar to Text Editor	Similar to Text Editor
Web Server	Similar to Data Base			

		Files that define the Content directory, network and performance parameters, security, etc	Directories where the webmaster deposits his creativity. Again defined by the Configurations	Preciouses access logs, vital for Marketing Intelligence, that are generated in a location and format defined by Configurations
e−Mail Server	Similar to Database and Web−Server	Files that define how to access user database, mail routing rules, etc	The preciouses users mail boxes. Again defined by the Configurations	Mail transfer log, virus detection log, etc. Again defined by the Configurations

Pay attention that the Software on its Own contains all your product *business logic*, which could be useless if you hadn't a Configuration to define how to work with a data bundle, provided by the user. So, Configurations are what connects your product to the user.

We can use a metaphor about a Sculptor (business logic), that needs Bronze (content) and a Theme or Inspiration (configuration) from a Mecenas (user), to produce a beautiful work (content). He make annotations in his Journal (logs) about his day−by−day activities, to report to his Mecenas (user).

3.2. The Importance of Clear Separation Between Four Parts

OK, so let's be more practical. The fact is, if we correctly use the universal parts concept, we greatly improve the quality of our Product. We'll do that simply separating, encapsulating, each one of these parts in different system directories (having only different files for each part is not sufficient). There is a standard called FHS that defines the Linux directories for each part, and we'll discuss it later in Section 4.

By now let's see the value of this separation to the user:

1. He gains a clear vision about where is each part, specially his Configurations and Content, and he feels your Product as something completely under control. The clareza brings ease of use, security and confidence in your Product. And in practice it permits him manipulate each part independently
2. It is clear now that, for instance, when backing up, user action is needed only for Configurations and Content (the puritans will also backup some logs). The user don't have to care about Software on its Own, because it is safe, original, on the product CD, in his shelf.
3. For upgrades, the new package will overwrite only the business logic, leaving intact the user's precious Configurations and Content. Here is very important to keep old content and configuration compatible, or to provide some tools help migration of data
4. The logs being kept in a separate filesystem (obviously suggested in your documentation), avoids that their exaggerated growth interfere with the Content, or with the stability of the whole system
5. If your Software follows some directory standards, the user don't have to reconfigure his system or environment to use it. He will simply ***Install−and−Use***.

Let's make some exercise with separation using as example a system called MySoftware, in which the business logic is in Example 1 and the configuration is in Example 2.

Example 1. A Shell program referring an external configuration file

```
#!/bin/sh

###############################################################################
##
## /usr/bin/MySoftware
##
## Business logic of MyProgram system.
## Do not change nothing in this file. All configuration can be
## made on /etc/MySoftware.conf
##
## We'll not support any modifications made here.
##

# Default configuration file
CONF=/etc/MySoftware.conf  ❶

# Minimal content directories
MIN_CONTENT_PATH=/var/www:/var/MySoftware/www        ❷

if [ -r "$CONF"]; then
        . "$CONF"          ❸
fi

# All the content I'll serve are the "minimal" plus the ones provided
# by the user in the configuration file $CONF
CONTENT_PATH=$MIN_CONTENT_PATH:$CONF_CONTENT_PATH    ❹

.
.
.
```

❶ Definition of the configuration file name.

❷ Definition of some static parameters.

❸ The configuration is readed from an external file, if exists.

❹ After reading the configuration file, all content directories –– user's + product's –– goes together in the `$CONTENT_PATH`, that will be used from now on.

Example 2. File containing only the configurations for MySoftware

```
###############################################################################
##
## /etc/MySoftware.conf
##
## Configuration parameters for MySoftware.
## Change as much as you want.
##

# Content directory.
# A ':' separated list of directories for your content.
# The directories /var/www and /var/MySofware are already there, so
# include here your special directories, if any.
CONF_CONTENT_PATH=/var/NewInstance:/var/NewInstance2  ❶

# Your e-mail address, for notifications.
EMAIL=john@mycompany.com  ❷

# Logs directory
LOG_DIR=/var/log/myInstance❸
```

❶❷❸
These are user defined parameters.

3.3. One Body, Many Souls

When I was a system administrator for IBM e−business Hosting Services, I was fascinated by Apache's flexibility letting us do things like this:

```
bash# /usr/sbin/httpd &
bash# /usr/sbin/httpd −f /etc/httpd/dom1.com.br.conf &
bash# /usr/sbin/httpd −f /etc/httpd/dom2.com.br.conf &
bash# /usr/sbin/httpd −f /etc/httpd/dom3.com.br.conf &
```

If we don't pass any parameter (like the first example), Apache loads its default, hardcoded configuration file from `/etc/httpd/conf/httpd.conf`. We built other configs, one for each customer, with a completely different structure, IP address, loaded modules, content directory, passwords, domains, log strategy etc.

This same concept is used by a text editor of a multiuser desktop (like Linux). When the code is loaded, it looks for a configuration file on the user's `$HOME`, and depending who invoked him (user A or B), it will appear differently because each user has its own personal configuration.

The obvious conclusion is that the Software's body (business logic) is pure e completely oriented by his manipulator's spirit (configuration). But the competitive advantage lays on how easy we switch from one spirit to another, like in Apache's example. It is very healthy to promote it to your user. You'll be letting him create intimacy, reliability, confort with your Product.

We used this approach with many different Softwares in that e−business Hosting time, and it was extremely usefull for maintenance etc. In a version migration we had total control over where were each of its parts, and upgraded and downgraded Software with no waste of time, with obvious success.

But there were some Products that refused to work this way. They had so many hardcoded parameters, that we couldn't see what divided the body from their spirit (or other parts). These Softwares were marked as *bad guys* and discarded/replaced as soon as possible.

We concluded that the *good guys* Softwares were intuitively blessed by their developer's four parts vision. And they made our life easyer. In fact, in that time we formulated this theory, that continues to prove itself.

Do you want to deploy *bad guy* or *good guy* Software?

4. Linux Directory Hierarchy: Oriented to the Software Parts

By now, all discussion are OS independent. On Linux, the Four Software Parts theory is expressed in his directory structure, which is classified and documented in the Filesystem Hierarchy Standard. The FHS is part of the LSB (Linux Standard Base), which makes him a good thing because all the industry is moving thowards it, and is a constant preoccupation to all distributions. FHS defines in which directories each peace of Apache, Samba, Mozilla, KDE *and your Software* must go, and you don't have any other reason to not use it while thinking in developing your Software, but I'll give you some more:

1. FHS is a standard, and we can't live without standards
2. This is the most basic OS organization, that are related to access levels and ***security***, where users intuitively find each type of file, etc
3. *Makes user's life easyer*

This last reason already justifies FHS adoption, so allways use the FHS !!!

More about FHS importance and sharing the same directory structure can be found in Red Hat website.

4.1. FHS Summary

So let's summarize what the FHS has to say about Linux directories:

Linux system directories

`/usr/bin`

Directory for the executables that are accessed by all users (everybody have this directory in their `$PATH`). The main files of your Software will probably be here. You should never create a subdirectory under this folder.

`/bin`

Like `/usr/bin`, but here you'll find only boot process vital executables, that are simple and small. Your Software (being high−level) probably doesn't have nothing to install here.

`/usr/sbin`

Like `/usr/bin`, but contains only the executables that must be accessed by the administrator (root user). Regular users should never have this directory in their `$PATH`. If your Software is a daemon, This is the directory for some of executables.

`/sbin`

Like `/usr/sbin`, but only for the boot process vital executables, and that will be accessed by sysadmin for some system maintaining. Commands like **fsck** (filesystem check), **init** (father of all processes), **ifconfig** (network configuration), **mount**, etc can be found here. It is the system's most vital directory.

/usr/lib

Contains dynamic libraries and support static files for the executables at `/usr/bin` and `/usr/sbin`. You can create a subdirectory like `/usr/lib/myproduct` to contain your helper files, or dynamic libraries that will be accessed only by your Software, without user intervention. A subdirectory here can be used as a container for plugins and extensions.

/lib

Like `/usr/lib` but contains dynamic libraries and support static files needed in the boot process. You'll never find an executable at `/bin` or `/sbin` that needs a library that is outside this directory. Kernel modules (device drivers) are under `/lib`.

/etc

Contains configuration files. If your Software uses several files, put them under a subfolder like `/etc/myproduct/`

/var

The name comes from "*variable*", because everything that is under this directory changes a frequently. Usually `/var` is mounted over a separate high–performance partition. In `/var/log` logfiles grow up. For web content we use `/var/www`, and so on.

/home

Contains the user's (real human beings) *home* directories. Your Software package should *never* install files here (in installation time). If *your* business logic requires a special UNIX user (not a human being) to be created, you should assign him a home directory under `/var` or other place *outside* `/home`. Please, never forget that.

/usr/share/doc, /usr/share/man

The "share" word is used because what is under `/usr/share` is platform independent, and can be shared among several machines across a network filesystem. Therefore this is the place for manuals, documentations, examples etc.

/usr/local, /opt

These are obsolete folders. When UNIX didn't have a package system (like RPM), sysadmins needed to separate an *optional* (or *local*) Software from the main OS. These were the directories used for that.

You may think is a bad idea to break your Software (as a whole) in many pieces, instead of keeping it all under a self–contained directory. But a package system (RPM) has a database that manages it all for you in a very professional way, taking care of configuration files, directories etc. And if you spread your Software using the FHS, beyond the user friendliness, you'll bring an intuitive way to the sysadmin configure it, and work better with performance and security.

4.2. Examples Using the FHS

Now that we know where each part of our software must be installed, lets review the Universal Parts Table applied to the FHS.

Table 2. Same Software, applying FHS

	Software on its Own	Configurations	Content	Logs, Dumps etc
Data Base Server	/usr/bin/, /usr/lib/, /usr/share/doc/mydb/, /usr/share/doc/mydb/examples/	/etc/mydb/	/var/db/instance1/, /var/db/instance2/, etc	/var/db/instance1/transactions/, /var/log/db/access-instance1.log, /var/log/db/access-instance2.log
Text Editor	/usr/bin/, /usr/lib/, /usr/lib/myeditor/plugins/, /usr/share/myeditor/templates/, /usr/share/doc/myeditor/	$HOME/.myeditor.conf	$HOME/Docs/	$HOME/.myeditor-tmp/
MP3 Generator	/usr/bin/, /usr/lib/, /usr/lib/mymp3/plugins/, /usr/share/doc/mymp3/	$HOME/.mymp3.conf	$HOME/Music/	$HOME/.mymp3-tmp/
Web Server	/usr/sbin/, /usr/bin/, /usr/lib/httpd-modules/, /usr/share/doc/httpd/, /usr/share/doc/httpd/examples/	/etc/httpd/, /etc/httpd/instance1/, /etc/httpd/instance2/	/var/www/, /var/www/instance1/, /var/www/instance2/	/var/logs/httpd/, /var/logs/httpd/instance1/, /var/logs/httpd/instance2/
E−Mail Server	/usr/sbin/, /usr/bin/, /usr/lib/, /usr/share/doc/mymail/	/etc/mail/, /etc/mailserver.cf	/var/mail/	/var/spool/mailqueue/, /var/logs/mail.log

4.3. Friendly Advice: Never use `/opt` or `/usr/local`

This is a very polemic subject, and that is why this is the most important section in this document. After almost ten years of UNIX experience I can beat any `/usr/local`−prone argument. I'll try to organize some ideas here.

It is very important to Linux evolution and popularization (especially in the desktop battlefield), that developers stop using this hell directories, and start using the FHS. After reading this section, if you still think this folders are good business, please drop me an e−mail.

Products that are *entirely* installed under *one* directory, uses the *self−contained* approach, that has several problems:

1. Forces the user to change environment variables like `$PATH` and `$LD_LIBRARY_PATH` to use your product easily.
2. Puts files in non−standard places, complicating system integration, and future installation of extensions to your product.

3. The sysadmin probably didn't prepared disk space in these partitions, generating problems in installation time.
4. It is an accepted approach only for pure graphical application, without the command line concept. This is why it were well accepted in Windows. But...
5. ...even using this approach, you can't avoid installing or changing files in standard locations to, for instance, make your icons appear in the user desktop.

This directories exists for historical reasons, and for compatibility with other UNICES, from the epoch we didn't have a package management system, and the sysadmin needed to segregate the products, to keep the control. And the last item shows us that even if you want, you'll not be able to segregate your product. You'll make it distant and non−practical to the user.

You don't have to be afraid of spreading your files according to FHS because RPM will keep an eye on them.

Many developers believe that the "self−contained" approach let them work with several versions of the same product, for testing purposes, or whatever. Yes, agree, with this or any good reason in the planet. But remember that a High Quality Software (or Commercial Grade Software) objective is to be practical for the final user, and not to be easy to their developers and testers. Invite yourself to visit an unexperienced (but potential customer) user and watch him installing your product.

If you have a business requirement to let the user work with several versions of your Product simultaneously (or any other reason), make a relocatable package, which is described in the Maximum RPM book. Be also aware about the implications of using this feature, described in the same book.

Nao e a toa that distributions like Red Hat and derivatives allways use FHS, instead of `/opt` or `/usr/local`. Read what Red Hat says about this subject, and think about it.

The Makefiles of an OpenSource Software that is portable to other UNICES must have the standard installation in `/usr/local` for compatibility reasons. But must also give the option, and induct the packager, to create the package using FHS specifications.

5. Provide Architecture for Extensions and Plugins

You'll probably let other Software vendors plug extensions to your product. Since you are the author of the initial Software, is your responsability to organize it in such a way that the user can simply install the extension RPM and use it, without forcing him modify any configuration file. Is is again the famous *Install−and−Use* that guarantees ease−of−use.

Well, and extension is nothing more that some files in a ***right format*** (DLLs that implements the API your Software defined), put in the ***right folders*** (directories your Software looks for extensions). We can see many applications requesting the user to change configuration files to "declare" the presence of a new plugin.

The most important thing to consider in your plugin architecture is *to not share files between plugins and your Software*. You should provide an architecture where plugins will fully install and uninstall themselves by simply putting and removing files in specific directories, documented in you Software. Good candidates are `/usr/lib/myproduct/plugins` as the plugins directory, and `/etc/myproduct/plugins` as the plugins configuration files directory. Your Software and plugins must be sufficient intelligent to know how to find files, specially configurations, in these directories. Using this approach, no post−install procedures is required from the user, and from the plugin provider.

5.1. Abstracting About Plugins

I would like to close this subject inviting the reader a se abstratir and think about any Software can be treated as an extension to the lower level Software. In the same way a third party plugin is an extension to your Software, your Software is also an extension to the OS (lower level). This is where all the Integration (from the title of this document) magic lives. So we can apply all the ease−of−use concepts we discussed before to the plugin architecture design of your Software.

6. Allways Provide RPM Packages of Your Softwares

This is extremely important for many reasons:

1. Ease−of−use. This is allways the primordial motivation.
2. Automates some tasks that must be made before and after the installation of your Software. Again bringing ease−of−use.
3. Intelligently manages configuration files, documentation etc, providing more control in an upgrade
4. Manages interdependencies with other packages and versions, guaranteeing good functionality.
5. Lets you distribute Software with your company's digital signature, and makes integrity checks (MD5) in each file, guaranteeing precedence, and reporting unwanted file modification.
6. Provides tools to let interact with your graphic installer.

But a good package is not only put together your files in a RPM. FHS must be followed, configuration and documentation files must be marked as is, and pre− and post−install scripts must be robust, to not let them damage the system (remember that installation processes is done by root).

Know well RPM because it can bring much power and facilities to you and your user. There are a lot of documentation available about RPM on the Internet:

- The book Maximum RPM, also available on−line and in printable PostScript format.
- RPM−HOWTO which is smaller and more straight−forward.
- www.rpm.org

6.1. Software Package Modularization

You should give user the option to install only the part of your Software he wants. Imagine your Software has a client part and a server part, and both use files and libraries in common. You should break them in 3 RPMs. For instance, lets say the name of your product is *MyDB*, so you'll provide the packages:

1. `MyDB-common-1.0-3.i386.rpm`
2. `MyDB-server-1.0-3.i386.rpm`
3. `MyDB-client-1.0-3.i386.rpm`

and last 2 packages depends on the first. If the user is installing a client profile, he will use:

1. `MyDB-common-1.0-3.i386.rpm`
2. `MyDB-client-1.0-3.i386.rpm`

If he is installing a server profile:

1. `MyDB-common-1.0-3.i386.rpm`
2. `MyDB-server-1.0-3.i386.rpm`

This approach will help the user save disk space, and be aware of how your Software is organized.

---

7. Security: The Omnipresent Concept

From a very general perspective, security is synonym of order, conscience. And insecure is everything that makes a system stop without the user wish. So besides open network ports, or weak cryptography (that are beyond the scope of this document), applications that inducts the user to use it only as root, or make him change files in inappropriate places, is considered insecure. We can say the same for the apps that fills a filesystem that is vital to the OS.

Many standards appeared from good practices discussed and developed in conjunction for a long time. So you should know and use them when you'll package your software, because they are key for you to achieve a good organization (security) level.

8. Graphical User Interface

Everybody loves graphical interfaces. Many times they make our life easyer, and this way helps to popularize a Software, because the learning curve get smaller. But for the everyday use, a command with many options and a good manual becomes much more practical, making scripts easy, remote access, etc. So the suggestion is, whenever is possible, to provide both interfaces: graphical for the beginners, and the powerfull command line for the expert.

8.1. KDE, GNOME, Java or Motif?

Better then a simple graphical interface is a consistent integrated desktop. And the desktops today in Linuxland are KDE and GNOME. Try to allways use one of them, or both.

KDE is the most outstanding, offering a true consistent desktop, flexible, with an extremely elegant architecture, using components, intercommunication, performance etc. It is constantly evolving, and is developed in C++. Its applications have an familiar integrated look−and−feel, is light and mature.

GNOME also brings the integrated desktop proposal, but it is far from the maturity and ease−of−use of KDE. From the other side, is very well supported by the comunity, and good improvements are appearing.

Motif isn't an integrated desktop. It is a widgets library (button, scrollbar etc), plus a window−manager. It was born commercial, is mature and popular in commercial applications. But is considered obsolete in front of KDE and GNOME, that integrates the desktop. Motif source code was opened by the OpenGroup and because that was renamed to OpenMotif.

Java is being used more and more for graphical interfaces, specially in server Software, where the graphics are only helpers to configuration and administration.

8.2. Web Interface: Access from Anywhere

Nowadays every desktop has a browser, and if your Product is a server application, the Web Interface is the right choice, because it lets a user administer it from anywhere. But keep in mind the security and organization of your CGIs, because they use to be front doors for crackers. Web interface (CGI) is completely different programming paradigm. Try to understand it conceptually first, starting from "how a web−server works", "what is a URL", etc, to get on this without compromising your Product's security.

8.3. Wizards and Graphical Installers

Specially for a commercial Product, your Software must provide a graphical installer. Believe me, they are impressive in a demonstration, and CIOs love them.

More then just installation, a wizard helps in the initial configuration of your Product, collects info like activation key etc, and shows the developer license.

A wizard should not do more than this:

1. Ask which modules to install, experienced by the user as checkboxes.
2. Get the necessary info to build an initial configuration (the soul) for the Software.
3. Install the selected modules, that are in fact RPM files. Each checkbox must represent one or more RPMs, because each RPM is a indivisible (atomic) portion of a Software.
4. After RPMs installation, change the configuration (soul) files (marked this way in the RPMs), or create some content, based on the data the user gave to the wizard.

So the wizard hides the RPM installation and writes initial personalization. RPM is still responsable for putting all your software files in the correct places. This role should never be of your installer. Think that an experienced user (there are a lot of them in the Linux world) should be able to reproduce your Product installation without the graphical help, using only RPM commands. In fact, in big data centers, where people make mass installations, a graphical installer only disturbs.

RPM provides tools that help your graphical installer interact with them, like installation percentage viewer. Documentation for use are allways in the RPM manual (**man rpm**) and in the Maximum RPM book.

9. Starting Your Software Automatically on Boot

The way Linux starts (and stops) all its subsystems is very simple and modular. Lets you define initialization order, runlevels etc

9.1. From BIOS to Subsystems

Lets review what happens when we boot Linux:

1. The BIOS or a bootloader (lilo, zlilo, grub, etc) loads Linux Kernel from disk to memory, with some parameters defined in the bootloader configuration. We can see this process watching the dots that appear in the screen. Kernel file stays in the `/boot` directory, and is accessed only at this moment.
2. In memory, Kernel code starts to run, detecting a series of vital devices, disk partitions etc.
3. On of the last things Kernel does is to mount the `/` (root) filesystem, that obrigatoriamente must contain the `/etc`, `/sbin`, `/bin` and `/lib` directories.
4. Immediately behind, calls the program called **init** (`/sbin/init`) and passes the control to him.
5. The **init** command will read his configuration file (`/etc/inittab`) which defines the system ***runlevel***, and some Shell scripts to be run.
6. These scripts will continue the setup of system's minimal infrastructure, mounting other filesystems (according to `/etc/fstab`), activating swap space (virtual memory), etc.
7. The last step, and most interesting for you, is the execution of the special script called `/etc/rc.d/rc`, which initializes the subsystems according to a directory structure under `/etc/rc.d`. The name *rc* comes from *run commands*.

9.2. Runlevels

The runlevels mechanism lets Linux initialize itself in different ways. And also lets us change from one profile (runlevel) to another without rebooting.

The default runlevel is defined in `/etc/inittab` with a line like this:

Example 3. Default runlevel (3, in this case) line in `/etc/inittab`

```
id:3:initdefault:
```

Runlevels are numbers from 0 to 6 and each one of them is used following this standard:

0

Halts the system. Turning to this runlevel, all subsystems are softly deactivated before the shutdown. Don't use it in the **initdefault** line of `/etc/inittab`.

1

Mono−user mode. Only vital subsystems are initialized because it is used for system maintenance. No user authentication (login) is required in this runlevel. A command line is directly returned to the user.

3, 2

3 is used when a system is in full production. Take it as the runlevel your software will run. 2 is historical and is like 3, but without NFS.

4

Not used. You can define it as you want, but is uncommon.

5

Like 3 plus a graphical login. It is ideal for a desktop workstation. Use 3 if the machine will be used as a server, for security and performance reasons.

6

Like runlevel 0, but after complete stop, the machine is rebooted. Don't use it in the **initdefault** line of `/etc/inittab`.

You can switch from one runlevel to another using the **telinit** command. And you can see the current runlevel and the last one with the **runlevel** command. See bellow how we switched from runlevel 3 to 5.

```
bash# runlevel
N 3
bash# telinit 5
bash# runlevel
3 5
bash#
```

9.3. The Subsystems

Subsystems examples are a web–server, data base server, OS network layer etc. We'll not consider a user oriented application (like a text editor) as a subsystem.

Linux provides an elegant and modular way to organize the subsystems initialization. An important fact to think is about subsystems interdependencies. For instance, it makes no sense to start a web–server before basic networking subsystem is active.

Subsystems are organized under the `/etc/init.d` and `/etc/rc.d/rcN.d` directories:

`/etc/init.d`

All installed Subsystems put in this directory a control program, which is a script that follows a simple standard described bellow. This is a simplified listing of this directory:

Example 4. Subsystems installed in `/etc/init.d`

```
bash:/etc/init.d# ls -l
-rwxr-xr-x  1 root  root   9284 Aug 13  2001 functions
-rwxr-xr-x  1 root  root   4984 Sep  5 00:18 halt
-rwxr-xr-x  1 root  root   5528 Nov  5 09:44 firewall
-rwxr-xr-x  1 root  root   1277 Sep  5 21:09 keytable
-rwxr-xr-x  1 root  root    487 Jan 30  2001 killall
-rwxr-xr-x  1 root  root   7958 Aug 15 17:20 network
-rwxr-xr-x  1 root  root   1490 Sep  5 07:54 ntpd
-rwxr-xr-x  1 root  root   2295 Jan 30  2001 rawdevices
-rwxr-xr-x  1 root  root   1830 Aug 31 09:29 httpd
-rwxr-xr-x  1 root  root   1311 Aug 15 14:18 syslog
```

`/etc/rc.d/rcN.d` (`N` is the runlevel indicator)

These directories must contain only special symbolic links to the scripts in `/etc/init.d`. This is how it looks:

Example 5. `/etc/rc3.d` listing

```
bash:/etc/rc3.d# ls -l
lrwxrwxrwx  1 root  root     18 Jan 14 11:59 K92firewall -> ../init.d/firewall
lrwxrwxrwx  1 root  root     17 Jan 14 11:59 S10network -> ../init.d/network
lrwxrwxrwx  1 root  root     16 Jan 14 11:59 S12syslog -> ../init.d/syslog
lrwxrwxrwx  1 root  root     18 Jan 14 11:59 S17keytable -> ../init.d/keytable
lrwxrwxrwx  1 root  root     20 Jan 14 11:59 S56rawdevices -> ../init.d/rawdevices
lrwxrwxrwx  1 root  root     16 Jan 14 11:59 S56xinetd -> ../init.d/xinetd
lrwxrwxrwx  1 root  root     18 Jan 14 11:59 S75httpd -> ../init.d/httpd
lrwxrwxrwx  1 root  root     11 Jan 13 21:45 S99local -> ../rc.local
```

Pay attention that all link names has a prefix starting with letter ***K*** (from Kill, to deactivate) or ***S*** (from Start, to activate), and a 2 digit number that defines the boot activation priority. In our example we have HTTPd (priority 75) starting after the Network (priority 10) subsystem. And the Firewalling subsystem will be deactivated (***K***) in this runlevel.

So ***to make your Software start automatically in the boot process, it must be a subsystem***, and we'll see how to do it in the following section.

9.4. Turning Your Software Into a Subsystem

Your Software's files will spread across the filesystems, but you'll want to provide a simple and consistent interface to let the user at least start and stop it. Subsystems architecture promotes this ease–of–use, also providing a way (non obrigatoria) to be automatically started on system initialization. You just have to create your `/etc/init.d` script following a standard to make it functional.

Example 6. Skeleton of a Subsystem control program in `/etc/init.d`

```
#!/bin/sh
#
# /etc/init.d/mysystem
# Subsystem file for "MySystem" server
#
# chkconfig: 2345 95 05    ❶
# description: MySystem server daemon
```

```
#
# processname: MySystem
# config: /etc/MySystem/mySystem.conf
# config: /etc/sysconfig/mySystem
# pidfile: /var/run/MySystem.pid

# source function library
. /etc/rc.d/init.d/functions

# pull in sysconfig settings
[ -f /etc/sysconfig/mySystem ] && . /etc/sysconfig/mySystem        ❷

RETVAL=0
prog="MySystem"
.
❸
.

start() {     ❹
        echo -n $"Starting $prog:"
        .
❺       .
        .
        RETVAL=$?
        [ "$RETVAL" = 0 ] && touch /var/lock/subsys/$prog
        echo
}

stop() {      ❻
        echo -n $"Stopping $prog:"
        .
❼       .
        .
        killproc $prog -TERM
        RETVAL=$?
        [ "$RETVAL" = 0 ] && rm -f /var/lock/subsys/$prog
        echo
}

reload() {    ❽
        echo -n $"Reloading $prog:"
        killproc $prog -HUP
        RETVAL=$?
        echo
}

case "$1" in ❾
        start)
                start
                ;;
        stop)
                stop
                ;;
        restart)
                stop
                start
                ;;
        reload)
                reload
                ;;
        condrestart)
                if [ -f /var/lock/subsys/$prog ] ; then
```

```
                        stop
                        # avoid race
                        sleep 3
                        start
                fi
                ;;
        status)
                status $prog
                RETVAL=$?
                ;;
        *)    ⑩
                echo $"Usage: $0 {start|stop|restart|reload|condrestart|status}"
                RETVAL=1
esac
exit $RETVAL
```

❶
: Although these are comments, they are used by **chkconfig** command and must be present. This particular line defines that on runlevels 2,3,4 and 5, this subsystem will be activated with priority 95 (one of the lasts), and deactivated with priority 05 (one of the firsts).

❷
: Besides your Software's own configuration, this script can also have a configuration file. The standard place for it is under `/etc/sysconfig` directory, and in our case we call it `mySystem`. This code line reads this configuration file.

❹❻❽
: Your script can have many functions, but it is obrigatorios the implementation of **start** and **stop** methods, because they are responsible for (de)activation of your Subsystem on boot. Other methods can be called from the command line, and you can define as much as you want.

❾
: After defining the script actions, the command line is analyzed and the requested method (action) is called.

❿
: If this script is executed without any parameter, it will return a help message like this:

```
bash# /etc/init.d/mysystem
Usage: mysystem {start|stop|restart|reload|condrestart|status}
```

❸❺❼
: Here you put your Software's specific command.

The **mysystem** subsystem methods you implemented will be called by users with the **service** command like this example:

Example 7. service command usage

```
bash# service mysystem start
Starting MySystem:                     [ OK ]
bash# service mysystem status
Subsysten MySystem is active with pid 1234
bash# service mysystem reload
Reloading MySystem:                    [ OK ]
bash# service mysystem stop
Stopping MySystem:                     [ OK ]
bash#
```

You don't have to worry about managing the symbolic links in `/etc/rc.d/rcN.d`. The **chkconfig** command makes it for you, based on the control comments defined in the beginning of your script.

Example 8. Using the chkconfig command

```
bash# chkconfig --add mysystem
bash# chkconfig --del mysystem
```

Read the **chkconfig** manual page to see what more it can do for you.

9.5. Packaging Your Boot Script

When you'll create the RPM, put your Subsystem script in `/etc/init.d` and ***do not include*** any `/etc/rc.d/rcN.d` link, because it is a user decision to make your subsystem automatic or not. If you include them and the user makes any change, the RPM file inventory will become inconsistent.

The symbolic links must be created and removed dynamically by the post−installation and pre−uninstallation process of your package, using the **chkconfig** command. This approach guarantees 100% package and filesystem consistency.

A. Red Hat, About the Filesystem Structure

This text was taken from The Official Red Hat Linux Reference Guide

Why Share a Common Structure?

An operating system's filesystem structure is its most basic level of organization. Almost all of the ways an operating system interacts with its users, applications, and security model are dependent upon the way it stores its files on a primary storage device (normally a hard disk drive). It is crucial for a variety of reasons that users, as well as programs at the time of installation and beyond, be able to refer to a common guideline to know where to read and write their binary, configuration, log, and other necessary files.

A filesystem can be seen in terms of two different logical categories of files:

1. Shareable vs. unsharable files
2. Variable vs. static files

Shareable files are those that can be accessed by various hosts; unsharable files are not available to any other hosts. Variable files can change at any time without system administrator intervention (whether active or passive); static files, such as documentation and binaries, do not change without an action from the system administrator or an agent that the system administrator has placed in motion to accomplish that task.

The reason for looking at files in this way has to do with the type of permissions given to the directory that holds them. The way in which the operating system and its users need to utilize the files determines the directory where those files should be placed, whether the directory is mounted read−only or read−write, and the level of access allowed on each file. The top level of this organization (*/ directory*)is crucial, as the access to the underlying directories can be restricted or security problems may manifest themselves if the top level is left disorganized (security=organization) or without a widely−utilized structure.

However, simply having a structure does not mean very much unless it is a standard. Competing structures can actually cause more problems than they fix. Because of this, Red Hat has chosen the most widely−used filesystem structure and extended it only slightly to accommodate special files used within Red Hat Linux.

B. About this Document

This document must be distributed under the terms of GNU Free Documentation License, which makes him sufficiently free. Everybody in invited to contribute to his content and ideas.

This document is published in the following locations:

- Main distribution [pt_BR] [XML Source]
- LinuxDoc, as a HOWTO [single page] [PDF]
- Linux and Main essay (24th March 2002)

It was written originally in brazilian portuguese, and then translated to english. SGML and the more−then−incredible DocBook was used, that made possible this document being distributed in other formats, found in website.

It got ready (potuguese+english) in mid march 2002. Everything changed after this epoch is cosmetics.

I wrote it to help commercial companies and OpenSource developers make plug−and−play, easy−to−use software for Linux, and this way improve Linux usability and popularity.

All concepts (from a high level perspective) described here, can be used in any UNIX flavor, or even other OS, like Windows. Maybe some day I'll write one of these for Windows....or Mac....