# A new binary alternatives system for Linux/Unix

Hans-Georg Eßer
LinuxUser, Editor-in-chief
*h.g.esser@linux-user.de*

September 22, 2003

**Abstract**

This paper presents a new concept for binary (program) alternatives that will aid Linux/Unix users in defining generic default programs for every task that allows multiple programs (such as: editor, mail client, web browser, etc.). While hiding the range of available programs and defaulting to a chosen editor, whatever editor launch command was used, it will still be possible to launch all installed programs.

## 1 Analysis of the current situation

Today Linux users are faced with a huge number of applications for most interesting tasks; e. g. there are lots of editors, mail clients, word processors, web browsers, etc.

Especially newbie Linux users find it disturbing to have to search through a list of menu items in the start menu in order to find a program for a given task. It is not simplifying the problem that the relevant menu entries often show the program name only: how would one deduce from an entry named "Kate" that this entry will launch an editor?

A user who knows what specific program he wants to start and happens to know the binary name as well, has no problem to launch this program by issuing a command on the shell. However, many applications start helper applications, and those might default to a program which is not the tool of choice of the user. In most cases it is possible to change the default behavior (even though occasionally it is not), but the procedures to change the default program vary as much as did the different GUI styles of X Window programs before the advent of standardized desktop environments such as KDE and GNOME.

### 1.1 Default programs set through environment variables

The standard Unix way to influence program behavior, including the setup of standard helper applications, is setting an environment variable that is evaluated in order to determine the default application for a given task; the most widely used example is that of the *EDITOR* variable.

This used to be a convenient way to solve the problem when Unix was primarily a text mode operating system and X Window was mostly a tool to display several *xterm* windows on one screen.

Further examples for these program definitions are *PAGER* (typically pointing to `more` or `less`) which is used by e. g. `man` and *SHELL*.

In a way, the variables *PATH*, *MANPATH*, *INFOPATH*, and *LD_LIBRARY_PATH* have similar functions, in that they define a search order: when you type "vi" and you have two different versions `/bin/vi` and `/usr/local/bin/vi`, it will depend on the order in the *PATH* which version is going to be executed.

Usefulness of these variables is obviously restricted to applications that make use of them.

## 1.2 Debian alternatives

The Debian GNU/Linux distribution maintains a directory `/etc/alternatives/` which holds symbolic links to binaries in their standard places, e. g. there is a link `pager` pointing to `/usr/bin/less`.

The tool `update-alternatives`(1) can be used to display and change information about known alternative programs, e. g.

```
# /usr/sbin/update-alternatives --display pager
pager - status is auto.
 link currently points to /usr/bin/less
/bin/more - priority 50
/usr/bin/less - priority 77
 slave pager.1.gz: /usr/share/man/man1/less.1.gz
/usr/bin/w3m - priority 25
 slave pager.1.gz: /usr/share/man/man1/w3m.1.gz
Current 'best' version is /usr/bin/less.
```

By giving programs priorities it is not necessary to define a standard application; instead the program with the highest priority is chosen.

## 1.3 Red Hat's "BlueCurve" desktop

With their 8.0 version Red Hat introduced the new BlueCurve desktop that makes KDE and GNOME look alike, cf. [Fio03, Tay03]. Another main feature is how they changed the start menu entries: Instead of giving the applications' names (as is the standard), menu entries are descriptive, e. g. *System tools/CD writer* instead of *System tools/XCDroast*.

Both changes shall aid new users in finding their ways around the Linux desktop without being confused by program names which do not always describe the program functions: A user new to Linux will not guess that "Konqueror" and "Nautilus" are file managers.

While standardizing menu entries and making them more self-explanatory is a useful step in helping new users (though Red Hat was criticized much for altering KDE and GNOME), it does not solve the problem of the wide range of helper applications launched by programs.

## 2 A new approach

In the OS News article [Sch03], Adam Scheinberg describes a collection of changes to Linux systems that would be needed in order to make Linux more accessible to new users. This paper is mainly motivated by the following suggestion from the article:

> Perhaps the way to get system-wide default is to have a given directory, say, `/system/commands`, that appears to be the equivalent of `/usr/local/bin` or `/usr/bin` – that contains the executables from the command line. Except, in our distro, the real files are kept in `/system/bin`. `/system/commands` is full of aliases. Then, when you change your default browser through our control panel – all the known browser commands – `konqueror`, `mozilla`, `opera`, `galeon` ... they all change to aliases of your selected browser.

An implementation of this approach would have to move all binaries from directories in the search path, add a new standard executable directory with placeholders for all moved binaries which would be links to default programs.

This approach has some design flaws:

- When a program is referred to by a full path name, e. g. `/usr/X11R6/bin/xemacs`, moving that file to a different location will break the calling application.
- Changing a standard application will require all links in `/system/commands` to be altered in order to reflect the change.
- There is no central configuration file or database that contains a list of classes of programs and the standard program for each subset of "similar" programs.

In this paper we present an approach that picks up Scheinberg's idea, but fixes these three flaws and allows for user-friendly and logical configuration of the program alternatives and defaults.

### 2.1 Linux Alternatives framework

Our new framework, called "Linux Alternatives framework" consists of five parts:

- a concept of "program classes". As an example, an "editor" class would consist of a list of programs and attributes of each of the programs. These attributes will be things such as: is this program an X or a terminal application, how are they invoked (command line arguments), etc.,
- a description of possible operations (modifications) on these classes and its elements, together with a specification of a command-line tool that shall handle these modifications,

- a suggestion for a file system structure that allows for the implementation of these ideas,
- a specification for configuration files,
- and finally a simple but working implementation.

### 2.1.1 Program classes

Programs which fulfill the same task, are grouped in program classes. That way, all the editors would belong to an "editor" class. Every member of the class can have special attributes (properties); such attributes can be flags (such as "is an X Window application" and "is a console application"), original path name (e. g. `/bin/vi`) for later restoration, or special command line options to be provided for starting as X or console application (e. g. `xemacs` understands the option "-nw" that makes it run as a console application).

A formal representation of an "editor" class could be like this:

```
class Editor {
  comment "Text editor"
  defaultX Emacs
  defaultC Vi
  member Emacs {
    comment "Eight megabytes almost continuously swapping"
    path /usr/X11R6/bin/emacs
    isX true
    isC true
    optX ""
    optC "-nw"
  }
  member Vi {
    comment "Visual Editor"
    path /bin/vi
    isX false
    isC true
    optC ""
  }
  member KEdit {
    comment "KDE Editor"
    path /opt/kde3/bin/kedit
    isX true
    isC false
    optX ""
  }
}
```

In the intended new framework, trying to execute any of these three editors from X Window would launch Emacs, while trying the same on a console (or

in a terminal window with no *DISPLAY* set) would start Vi. It is irrelevant whether the program call included the full path name or not.

### 2.1.2 Class operations

Program classes can be created, modified and removed. The creation of a class requires an identifier and an optional comment:

```
lxa-create  ( id class_id, string comment )
```

Several types of modification of an existing class are possible: a new member can be added, an existing one removed, the default X and console program set and unset (being unset means that no replacements take place), and the comment and class name can be changed:

```
lxa-add       ( id class_id, id member_id,  path member_path,
                string member_comment,      boolean isX, isC,
                string  optX, optC )
lxa-remove    ( id class_id, id member_id )
lxa-rename    ( id class_id, id new_id )
lxa-comment   ( id class_id, string new_comment )
lxa-defaultX  ( id class_id, id member_id )
lxa-defaultC  ( id class_id, id member_id )
lxa-nodefaultX ( id class_id )
lxa-nodefaultC ( id class_id )
```

A proposed command line call of "lxa-create", "lxa-add", and "lxa-defaultC" might look like this:

```
# lxa create Editor -c "Text editors"
# lxa add Editor Vi -p /bin/vi -c "Visual Editor" -C
# lxa add Editor xemacs -c "X Emacs" -X -C
# lxa defaultC Editor Vi
```

(Using options instead of a fixed ordered list of parameters allows for omitting parameters that are not required. The second "add" call adds `xemacs` and finds the path on its own; member name and binary name are identical here. However, it makes sense to explicitly give the path in case there are two identically named binaries in different directories.)

### 2.1.3 Class activation

For the convenience of allowing different users different setups it makes sense to introduce a notion of *activiation state*: A class can be active or inactive, and this state can be set globally (in the class definition) as well as locally (in a user configuration file).

For the global settings, the class description is augmented with an *active* keyword:

```
class Editor {
  comment "Text editor"
  active true
  defaultX Emacs
  defaultC Vi
  member Emacs {
  ...
```

The `lxa` tool will then allow two new commands to work on classes:

```
# lxa activate class
# lxa deactivate class
```

When called this way by the system administrator, changes will be made to the class configuration file: This will be a global change of settings.

On the other hand, non-`root` users can activate and deactivate classes on their own. A configuration file `~/.lxarc` will hold information about active and inactive classes. The default behavior is to use the global settings which can be overwritten by the user. Only after a user has issued an `lxa activate` or `lxa deactivate` command, will an entry be made to his local configuration file. To give up the local control over a class and return to the settings made by the administrator, the command

```
$ lxa useglobal class
```

can be used. This is not an option for the administrator.

If `root` wants to change settings for the `root` account but not globally, he must create the file `/root/.lxarc` manually and enter activity or inactivity statuses himself, the syntax in this file (as also in the users' `~/.lxarc` files) is simple:

```
Class:  active|inactive
```

### 2.1.4 File system structure

In order to allow for compatibility with all Linux distributions and other Unix operating systems, a new directory `/usr/lxa` will be used to hold all program data; only configuration files will be held in `/etc/lxa`.

- `/usr/lxa/bin`: This directory holds the two files `lxa`, `lxa-choose`. `lxa` is the configuration tool which is used for creation, modification, and deletion of program classes.
- `/usr/lxa/alternatives`: This is the target directory where all executables are moved on class inclusion. After setting up an editor class, binaries `vi`, `emacs`, `xemacs`, `kedit`, `nedit`, `jedit`, `xedit`, `kate`, ... will be found here.
- All files moved to `/usr/lxa/alternative` will be replaced with same-named symbolic links to `/usr/lxa/bin/lxa-choose`.

- `/etc/lxa`: This directory contains two configuration files, a general Linux Alternatives configuration `lxa.conf` and the class database `classes.conf`.

When adding a program to a class, it will be removed. For safety purposes a copy will be kept in a subdirectory named `.lxa` of the original directory. Thus, completely removing Linux Alternatives from a machine will be as simple as removing the `/usr/lxa` and `/etc/lxa` hierarchy and copying all `.lxa/*` files to their parent directories.

Note that all progam files are still available in the `/usr/lxa/alternatives` directory which should not be included in the `PATH` variable.

```
# lxa info emacs
emacs=/usr/X11R6/bin/xemacs : class Editor [Text Editors]:
  c-C-  Vi        Visual Editor
        /usr/lxa/alternatives/vi      (/bin/vi)
  cx--  Emacs     Eight megabytes almost continuously swapping
        /usr/lxa/alternatives/emacs   (/usr/X11R6/bin/emacs)
  cx--  XEmacs    X Emacs
        /usr/lxa/alternatives/xemacs  (/usr/X11R6/bin/xemacs)
```

## 2.2 Quick and dirty implementation

As a proof of concept we have created a Python-based implementation that handles all features described above. In most cases this implementation should be sufficient, but if many binaries are going to be replaced and called often (e. g. from scripts), a reimplementation in C or C++ and use of an optimized lookup scheme would give improvements in execution time.

The package will be available from `http://lxa.hgesser.com/` from September 15, 2003.

### 2.2.1 How it works

The technology behind the implementation is simple. When a program is called which belongs to one of the LXA classes, in fact `/usr/lxa/lxa-choose` will be started. By evaluating the variable *0* this script will know which program name was supplied on the command line. It will look up this name in the class database, evaluate *DISPLAY* to see if an X Window display is available, and then decide on this and the class information which program in `/usr/lxa/alternatives/` to start. Command line arguments from the original call are supplied to the real binary, and additional arguments are added if these were defined in the member definition (*optX* and *optC*).

# 3 Integration with Linux distributions

To allow for easy integration with current Linux distribution types, there must be a way to combine package installation and removal with respective operations

on the class definitions. RPM and Debian based distributions can handle this through post-install scripts which alter the class file after successful installation of a new package or removal of one.

If the execution of post-install and pre-uninstall scripts is impossible or not wanted, it is possible to go another way: Instead of writing all class information to a configuration file `/etc/lxa/classes.conf`, a directory `/etc/lxa/classes/` can be used which holds separate files each of which defines one program and its membership in a class. Then each of these files will be part of a software package: Installing and removing such a package will include creating and removing a class membership definition.

For source based packages (e. g. those with the standard `configure`, `make`, `make install` procedures) class configuration can be part of the install step, or makefiles could be extended to have a *classsetup* target.

# 4 Problems and solutions

Four issues with this concept have been identified so far. They are minor inconveniences which can be overcome if required.

## 4.1 Breaking integrity checks of package management tools

Modern Linux distributions use package management tools that can perform integrity checks on installed packages. Obviously removing a binary that belongs to an RPM or Debian package and replacing it with a symbolic link will break the integrity of the installed package.

If the packaging tool (`rpm` or `dpkg`) was aware of Linux Alternatives, it could query the Alternatives database, find the replaced file and check it instead of the original path.

Further, removing a package without previously removing its binaries from all class definitions can break the Alternatives system. The original binary will still be there (since it was moved to `/usr/lxa/alternatives/`). Now if a user tries to call the removed program, this should lead to the normal error message since in removing the package, the link from the binary's old position to `lxa-choose` will have been removed, too.

However if the removed package was defined as standard application for a class, any attempt to start a *different* program in the same class may do one of the following two things:

- Since the binary is still in the `/usr/lxa/alternatives/` folder, the program is going to start. That means that it will still be available even though the packages was removed – which is not the desired behavior.
- If the program starts but cannot find required files which were part of the removed package, it may break and give an unreasonable error message.

Thus the administrator should make sure that before or while removing a package all its binaries are removed from any class memberships as well.

### 4.2 Ignorance of the PATH

Most Linux distributions use different values for the *PATH* variable for `root` and regular users, e. g. removing the `.../sbin/` directories from the path of regular users, thus primitively blocking attempts to call `fdisk` as non-`root`. Typically programs which are out of the regular users' path lack rights needed for execution, e. g. the +x bit for "others".

Joining privileged and non-privileged commands in one class could lead to the effect that the chosen application cannot be executed by regular users. However, that is a question of defining the classes properly. For example, putting `fdisk` and some (imaginary) command line tool for displaying disk settings in one class would be a wrong attempt.

`fdisk` and `cfdisk`, on the other hand, could be joined in one class, which will still cause problems if `/usr` sits in a partition of its own and is not mounted on start-up due to a problem. Adding vitally important programs to classes thus is not recommended. (Note that any program will still be available in the `.lxa/` subdirectory; thus a moved `/sbin/fdisk` can still be executed as `/sbin/.lxa/fdisk` in case of a problem.)

### 4.3 Redundancy of menu entries

Implementing this concept leads to the situation that standard start menus hold several entries for e. g. editors which will all launch the same editor, thus making the menu redundant. However when a menu clearly states "Xemacs", "Nedit", and so on, it is not the expected behavior that they should all start the same program.

To remove the problem, after any change to the classes, the menu system should be updated, too, then pointing to the real programs in `/usr/lxa/alternatives/`.

In case of the KDE desktop, it is sufficient to search the `/opt/kde3/share/applnk/` hierarchy (which might me located elsewhere, e. g. in `/usr/share/applnk`), grep for "Exec=..." lines and exchange the paths. The same holds for GNOME which uses the same syntax for its menu entries. All other window managers use plain text files that can be modified accordingly.

### 4.4 Unique names required

In the current implementation, binaries are moved to `/usr/lxa/alternatives/`. If two binaries have the same name (since e. g. two different `vi` versions are available) one will overwrite the other if both are added to a class. However it is possible to change this by checking and renaming.

## 5 Future extensions

Currently the only information about a program used in Linux Alternatives is whether it is an X or console application. A possible extension is to describe requirements, e. g. demanding that the user work locally rather than via an `ssh`

or other network connection. That way an X application may not be chosen even though X forwarding was enabled. (Credits to my colleague *Mirko Dölle* for this suggestion.)

Instead of defining standard X and terminal applications, programs could be given priorities, as does the Debian Alternatives system (see section 1.2). Then after checking whether an X or terminal application is needed, the program that fulfills this requirement and has the highest priority would be chosen.

A graphical frontend might be nice, at least one that allows non-`root` users to activate and deactivate classes and view the class definitions.

# 6 Summary

We have presented a system for definition of program classes that allows to choose standard tools out of groups of similar applications both for X Window and the console. A simple but working implementation is available, and minor problems of this approach as well as possible extensions have been addressed.

# References


[Fio03] Marco Fioretti. Hooray for Bluecurve. *Linux Review / Linux Journal*, 2003. `http://www.linuxjournal.com/article.php?sid=6476`.

[LXA03] LXA Linux Alternatives Project. Project web page. 2003. `http://lxa.hgesser.com/`.

[Sch03] Adam Scheinberg. If I Had My Own Distro. *OS News*, 2003. `http://www.osnews.com/story.php?news_id=3431`.

[Tay03] Owen Taylor. Configuring the Red Hat Linux Desktop. 2003. `http://people.redhat.com/otaylor/rh-desktop.html`.