

Smart Package Manager The Complete Users Guide

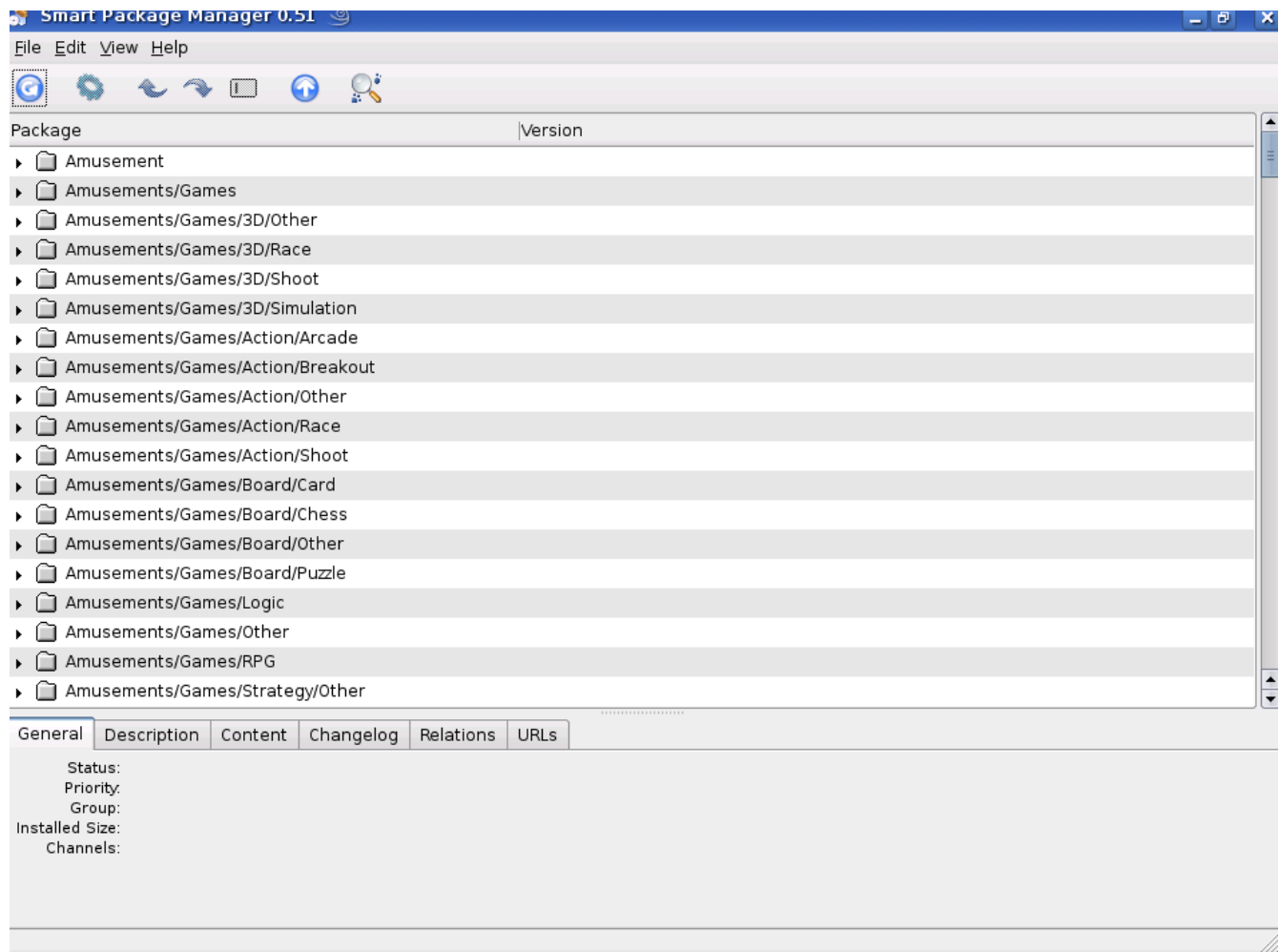


Table of Contents

Who is Smart for	4
The case for Smart.....	5
Understanding package management.....	12
Understanding packages.....	12
Repositories.....	13
What is metadata?.....	13
Features.....	14
Available Interfaces.....	17
Before we begin.....	18
Considerations.....	18
Dependencies.....	19
Installing Smart.....	20
Installing using an existing package manager.....	20
Installing the smart binary with out a package manager.....	20
Installing from source.....	20
Using Smart.....	21
Supported Sources.....	21
Setup Commands.....	25
Action Commands.....	25
Query Commands.....	25
Configuration.....	25
Packaging for Smart.....	29
Sample specs.....	29
Coding for smart.....	29
The smart API.....	29
Understanding Python.....	29
Bash.....	29
C.....	29
Writing patches.....	29
Troubleshooting Smart.....	29
Error messages.....	29
Strace reports?.....	29
How to interpret those cryptic error messages.....	29
Where to get help.....	29

Introduction

The [Smart Package Manager home page](#) says this:

The **Smart Package Manager** project has the ambitious objective of creating smart and portable algorithms for solving adequately the problem of managing software upgrading and installation. This tool works in all major distributions, and will bring notable advantages over native tools currently in use (APT, APT-RPM, YUM, URPMI, etc).

Notice that this project is **not** a magical bridge between every distribution in the planet. Instead, this is a software offering **better package management** for these distributions, even when working with **their own packages**. Using multiple package managers at the same time (like rpm and dpkg) **is possible**, even though not the software goal at this moment.

From *The Free On-line Dictionary of Computing*:

- [smart](#)
 - 1. <[programming](#)> Said of a program that does the *{Right Thing}* in a wide variety of complicated circumstances. (...)

Who is Smart for

The Smart Package Manager is for those who seek better package management in various Linux distributions. Each distribution has their default package manager. In Debian and Debian variants, such as Ubuntu, apt is commonly used, with a graphical interface called synaptic. In Slackware, the default package manager is pkgtools. Slackware has a port of apt called slapt. In rpm based distributions, such as Redhat, Fedora, openSUSE, Mandriva, and PCLinuxOS (to name a few), they all have their own package manager. In Fedora, yum is common place. In Mandriva, urpmi is the default tool, with rpmDrake being the graphical front end. In openSUSE YaST/zypper is the default tool. Zypper is the background tool for YaST.

Each of these package managers have their short comings. There is no perfect package manager. But there are some that are better than others. When looking at package managers, there are a few things you must consider. I will discuss some of these in more detail later on, but for now, I'll just touch on them.

Some considerations could be; what has you dissatisfied about your current package manager? In some cases, people are upset at the speed of their current package manager. In other cases, it's how the package manager handles dependencies.

Whatever your reason, smart package manager typically out performs the other package managers in dependency resolution and in speed. For more information on how smart package manager compares to some other package managers, read on **The case for Smart**.

The case for Smart

Study Cases

In this section will be described real cases showing Smart behavior in comparison with other tools, or handling unusual situations.

Notice that Smart was not tuned to work in any of these cases, and the reason it works is because handling unusual situations was the initial project goal.

Case 1 - APT

This case happened in a real world environment where a weakness in the algorithm used by APT (which is the same used in APT-RPM) turned a simple operation into a problem of obscure results. Smart Package Manager was used in the same environment to show its results.

The problem starts when an installation of xscreensaver is tried:

```
[root@damien:/root] apt-get install xscreensaver
Reading Package Lists... Done
Building Dependency Tree... Done
Some packages could not be installed. This may mean that you have
requested an impossible situation or if you are using the unstable
distribution that some required packages have not yet been created
or been moved out of Incoming.
```

Since you only requested a single operation it is extremely likely that the package is simply not installable and a bug report against that package should be filed.

The following information may help to resolve the situation:

The following packages have unmet dependencies:

```
xscreensaver: Depends: libglade-2.0.so.0
                  Depends: libxml2.so.2
```

E: Broken packages

The error shown makes the user believe that libglade-2.0.so.0 and libxml2.so.2 are not available. That's not the case:

```
[root@damien:/root] apt-get install libxml2
Reading Package Lists... Done
Building Dependency Tree... Done
Some packages could not be installed. This may mean that you have
requested an impossible situation or if you are using the unstable
distribution that some required packages have not yet been created
or been moved out of Incoming.
```

Since you only requested a single operation it is extremely likely that

the package is simply not installable and a bug report against that package should be filed.
The following information may help to resolve the situation:

The following packages have unmet dependencies:
libxml2: Depends: glibc-iconv but it is not going to be installed
E: Broken packages

Another misleading error message. Let's go further:

```
[root@damien:/root] apt-get install glibc-iconv
Reading Package Lists... Done
Building Dependency Tree... Done
Some packages could not be installed. This may mean that you have
requested an impossible situation or if you are using the unstable
distribution that some required packages have not yet been created
or been moved out of Incoming.
```

Since you only requested a single operation it is extremely likely that the package is simply not installable and a bug report against that package should be filed.
The following information may help to resolve the situation:

The following packages have unmet dependencies:
glibc-iconv: Depends: glibc-gconvdata (= 2.3.3) but 1:2.3.2-586_1cl is to be installed
E: Broken packages

Version 2.3.3 is needed, but 1:2.3.2-586_1cl is to be installed. This message is mostly correct. The only problem is, "1:2.3.2-586_1cl" is already installed:

```
[root@damien:/root] apt-cache policy glibc-gconvdata
glibc-gconvdata:
  Installed: 1:2.3.2-586_1cl
  Candidate: 1:2.3.2-586_1cl
  Version Table:
*** 1:2.3.2-586_1cl 0
      100 RPM Database
  0:2.3.3-69473cl 0
      500 file: conectiva/all pkglist
```

The problem was found. A package from another repository (586_1cl shows it's not native, in that specific case) has a higher epoch than the one available in the usual repository. This clearly shows that the APT algorithm marks a single version as candidate, and when this is not the wanted version for some operation, the whole operation is compromised.

When testing Smart Package Manager in the same environment, the expected result is obtained:

```

[root@damien:/root] smart install xscreensaver
Updating cache... ##### [100%]

Computing transaction...

Downgrading packages (1):
  glibc-gconvdata-0:2.3.3-69473cl.i386

Installed packages (4):
  glibc-iconv-0:2.3.3-69473cl.i386
  libglade2-2.4.0-68154cl.i386
  libxml2-2:2.6.13-67598cl.i386
  xscreensaver-4.15-69825cl.i386

Confirm changes (Y/n)?

Done.

```

Case 2 - APT & YUM

This is another real case, and is being reproduced in a controlled environment for tests with YUM, APT-RPM, and Smart.

The issue is, a package named A requires package BCD explicitly, and RPM detects implicit dependencies between A and libB, libC, and libD. Package BCD provides libB, libC, and libD, but additionally there is a package B providing libB, a package C providing libC, and a package D providing libD.

In other words, there's a package A which requires four different symbols, and one of these symbols is provided by a single package BCD, which happens to provide all symbols needed by A. There are also packages B, C, and D, that provide some of the symbols required by A, but can't satisfy all dependencies without BCD.

The expected behavior for an operation asking to install A is obviously selecting BCD to satisfy A's dependencies, on the other hand, YUM and APT fail to deliver that as a guaranteed operation, as is shown below.

First, let's see how YUM deals with the problem:

```

[root@burma ~]% yum install A
Setting up Install Process
Setting up Repo: localpub
 repomd.xml          100% |=====| 951 B    00:00
Reading repository metadata in from local files
localpub : ##### 5/5
Resolving Dependencies
--> Populating transaction set with selected packages. Please wait.
---> Downloading header for A to pack into transaction set.
A-1.0-1cl.i386.rpm   100% |=====| 1.0 kB    00:00
---> Package A.i386 0:1.0-1cl set to be installed
--> Running transaction check

```

```

--> Processing Dependency: libD for package: A
--> Processing Dependency: libC for package: A
--> Processing Dependency: libB for package: A
--> Processing Dependency: BCD for package: A
--> Restarting Dependency Resolution with new changes.
--> Populating transaction set with selected packages. Please wait.
--> Downloading header for D to pack into transaction set.
D-1.0-1cl.i386.rpm      100% |=====| 1.0 kB    00:00
--> Package D.i386 0:1.0-1cl set to be installed
--> Downloading header for C to pack into transaction set.
C-1.0-1cl.i386.rpm      100% |=====| 1.0 kB    00:00
--> Package C.i386 0:1.0-1cl set to be installed
--> Downloading header for B to pack into transaction set.
B-1.0-1cl.i386.rpm      100% |=====| 1.0 kB    00:00
--> Package B.i386 0:1.0-1cl set to be installed
--> Downloading header for BCD to pack into transaction set.
BCD-1.0-1cl.i386.rpm    100% |=====| 1.0 kB    00:00
--> Package BCD.i386 0:1.0-1cl set to be installed
--> Running transaction check

```

Dependencies Resolved

Transaction Listing:

```
Install: A.i386 0:1.0-1cl
```

Performing the following to resolve dependencies:

```
Install: B.i386 0:1.0-1cl
Install: BCD.i386 0:1.0-1cl
Install: C.i386 0:1.0-1cl
Install: D.i386 0:1.0-1cl
```

Is this ok [y/N]:

YUM selected all packages for installation, even though BCD alone would satisfy A's dependencies.

Let's see how APT deals with that:

```

[root@burma ~]% apt-get install A
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  B BCD
The following NEW packages will be installed:
  A B BCD
0 upgraded, 3 newly installed, 0 removed and 0 not upgraded.
Need to get 0B/4055B of archives.
After unpacking 0B of additional disk space will be used.
Do you want to continue? [Y/n] n

```

As a coincidence, APT did a better job, and selected only B and BCD to satisfy A's dependency, which is still not right.

Now, let's see how Smart would solve the problem:

```
[root@burma ~]% smart install A
Updating cache... ##### [100%]

Computing transaction...

Installed packages (2):
  A-1.0-1cl@i386      BCD-1.0-1cl@i386

2.7kb of package files are needed.

Confirm changes (Y/n)?
```

Smart correctly selected only BCD, since it's necessary anyway, and solves all dependencies.

Case 3 - APT & YUM

That's another interesting case which was tested with APT-RPM and YUM.

In this case, there's a package A version 1.0 installed in the system, and there are two versions available for upgrading: 1.5 and 2.0. Version 1.5 may be installed without problems, but version 2.0 has a dependency on B, which is not available anywhere.

In this case, the best possibility is upgrading to 1.5, since upgrading to 2.0 is not an option.

Let's see how APT reacts to this situation:

```
[root@burma ~]% apt-get upgrade A
Reading Package Lists... Done
Building Dependency Tree... Done
The following packages have been kept back
  A
0 upgraded, 0 newly installed, 0 removed and 1 not upgraded.
```

APT seems to refuse to upgrade A, even though version 1.5 might be installed without problems.

What happens when forcing APT to install A:

```
[root@burma ~]% apt-get install A
Reading Package Lists... Done
Building Dependency Tree... Done
Some packages could not be installed. This may mean that you have
requested an impossible situation or if you are using the unstable
distribution that some required packages have not yet been created
or been moved out of Incoming.
```

Since you only requested a single operation it is extremely likely that the package is simply not installable and a bug report against that package should be filed.
The following information may help to resolve the situation:

The following packages have unmet dependencies:
A: Depends: B but it is not installable
E: Broken packages

It really refuses to install the newest version, and doesn't consider the possibility of using version 1.5.

Now, let's see how YUM would handle it:

```
[root@burma ~]% yum update
Setting up Update Process
Setting up Repo: localpub
repomd.xml          100% |=====| 951 B    00:00
Reading repository metadata in from local files
localpub : ##### 3/3
Resolving Dependencies
--> Populating transaction set with selected packages. Please wait.
---> Package A.i386 0:2.0-1cl set to be updated
--> Running transaction check

Dependencies Resolved
Transaction Listing:
  Update: A.i386 0:2.0-1cl
Is this ok [y/N]: y
Downloading Packages:
Running Transaction Test
Finished Transaction Test
Transaction Test Succeeded
Running Transaction
Updating: A 100 % done 1/2

Updated: A.i386 0:2.0-1cl
Complete!
```

It succeeded installing version 2.0, and failed to report a problem, as shown below:

```
[root@burma ~]% rpm -V A
Unsatisfied dependencies for A-2.0-1cl: B
```

Now, let's see how Smart would behave in the same situation:

```
[root@burma ~]% ./smart.py upgrade
Loading cache...
```

```
Updating cache... ##### [100%]
Computing transaction...
Upgrading packages (1):
  A-1.5-1cl@i386
1.3kb of package files are needed.
Confirm changes (Y/n)?
```

Smart correctly selects the intermediate version 1.5, which is the only viable possibility given the current options.

Understanding package management

In Linux there are a variety of package managers. Each of them with their own goals, features and benefits. The basic purpose of a package manager is to install or upgrade packages efficiently and cleanly. A package manager should correctly handle dependencies. Preferably, with little to no interaction from the user.

Dependency resolution is the primary function of a package manager. Without this basic function, you might as well download the packages, and install them in proper order yourself. In fact, Linux didn't always have package managers. So back then, you did have to download and install the packages in the correct order.

In the summer of 1994 Red Hat released RPP which was the precursor to RPM. Also in the summer of 1994 Ian Murdock, the founder of Debian began working on dpkg. But while you can download a package to your hard drive, and use RPM or dpkg to install it, if you don't have all the dependencies met, RPM or dpkg will fail, complaining about dependencies. To this end, package managers such as yum, apt, synaptic, YaST, urpmi/rpmDrake, and a host of others have been developed. These were front ends designed to enhance rpm or dpkg or the other system package managers. They provided enhancements such as a graphical interface, and allowing users to add repositories. By adding repositories, this greatly improved the dependency resolution ability since the dependencies could be fetched from the repositories.

The next thing that played a part in dependency resolution was the algorithm. The algorithm is what allows the package manager to calculate the dependencies. This is another major factor in package managers, as the algorithm is what makes the package manager correctly resolve dependencies. There has been a lot of advancements made in the algorithms for the corresponding package managers.

Understanding packages

All programs start as packaged source code. This is even true in Windows. Then it is then packaged into a more user friendly package. In Windows you recognize it by an .exe in which the InstallShield unpacks and installs, and registers the software with the Windows registry. Once you understand this, the concept of packaging in Linux isn't so different.

In Gentoo, they use e-builds. These e-builds point to source packages that work with a package manager called portage. In Debian, the packages are done as .deb, in which they use dpkg, which is quite similar to rpm. Then we have the rpm distributions (Redhat, Fedora, openSUSE, Mandriva, etc...), which use rpm. All package managers do is take the source, package it in a binary executable. This is how you receive it. Then for you to use it, you use the package manager which extracts it, installs it, and registers it with the system.

In every package, there are things called dependencies. A dependency is

something that a package relies on to make it work. In most cases, it would be another package. Also, in packages, there are obsoletes. This is what is outdated by installing or upgrading to this package.

Repositories

So what are these things called repositories? In the reference in which we are talking about, repositories are web sites that contain packages. Often, but not always, these repositories contain a file or directory with metadata. Every distribution has a repository. Usually more than one.

What is metadata?

From the Free On-line Dictionary of Computing:

[meta-data](#)

<data> /me't*-day` t*/, or combinations of /may'-/ or (Commonwealth) /mee'-/; /-dah` t*/ (Or "meta data") Data about [data](#). In [data processing](#), meta-data is definitional data that provides information about or documentation of other data managed within an application or environment.

For example, meta-data would document data about [data elements](#) or [attributes](#), (name, size, data type, etc) and data about [records](#) or [data structures](#) (length, fields, columns, etc) and data about data (where it is located, how it is associated, ownership, etc.). Meta-data may include descriptive information about the context, quality and condition, or characteristics of the data.

So what does all that mean? Meta-data is data about data. In the case of packages, repositories and package managers, meta-data is what provides information from the repositories to the package managers about the packages on the repositories.

Meta-data contains the product name, the description of the product, the version of the product, dependencies, and obsoletes. This is then read by the package manager, and the job of the package manager to handle this correctly.

Features

Modular

Smart has been developed with modularity and flexibility in mind. It's completely backend-based, and package-manager-agnostic. Support is currently implemented for *RPM*, *DPKG*, and *Slackware* package management systems, and porting it to new systems should be very easy.

Smart Transactions

That's one of the most interesting aspects of Smart Package Manager, and the one who has motivated calling it *smart*. Computing transactions respecting the relations involved in the package management world may become an unpleasant task when thousands of packages and relations are being considered, or even when just a few complex relations turn the most obvious choice into the unwanted one.

While other softwares try to find a possible solution to satisfy the relations involved in some user-requested operation, and sometimes even fail to do so, Smart goes beyond it. In the kernel of Smart Package Manager lives an algorithm that will not only find a solution, if one is available, but will find the best solution.

This is done by quickly weighting every possible solution with a pluggable policy, which redefines the term "best" depending on the operation goal (install, remove, upgrade, etc).

This behavior has many interesting consequences. In upgrades, for instance, while precedence is given to newer versions, intermediate versions may get selected if they bring a better global result for the system. Packages may even be reinstalled, if different packages with the same name-version pair have different relations, and the one not installed is considered a better option.

Another important goal achieved with the transaction algorithm is that, even though it is able to check and fix relations in the whole system, it will work even when there are broken relations in installed packages. Only relations related to the operation being made are checked for correctness.

Channels

Channels are the way Smart becomes aware about external repositories of information. Many different channel types are supported, depending on the backend and kind of information desired:

- APT-DEB Repository
- APT-RPM Repository
- DPKG Installed Packages
- Mirror Information

- Red Carpet Channel
- RPM Directory
- RPM Header List
- RPM MetaData (YUM)
- RPM Installed Packages
- Slackware Repository
- Slackware Installed Packages
- URPMI Repository

Priority Handling

Priorities are a powerful way to easily handle integration of multiple channels and explicit user setups regarding preferred package versions.

Basically, packages with higher priorities are considered a better option to be installed in the system, even when package versions state otherwise. Priorities may be individually assigned to all packages in given channels, to all packages with given names, and to packages with given names inside given channels.

With custom priority setups, it becomes possible to avoid unwanted upgrades, force downgrades, select packages in given channels as preferential, and other kinds of interesting setups.

Autobalancing Mirror System

Smart offers a very flexible mirror support. Mirrors are URLs that supposedly provide the same contents as are available in other URLs, named origins. There is no internal restriction on the kind of information which is mirrored. Once an origin URL is provided, and one or more mirror URLs are provided, these mirrors will be considered for any file which is going to be fetched from an URL starting with the origin URL.

Mirror precedence is dynamically computed based on the history of downloads of all mirrors available for a given origin URL (including the origin site itself). The fastest mirrors and with less errors are chosen. When errors occur, the next mirror in the queue is tried.

For instance, if a mirror `http://mirror.url/path/` is provided for the origin `ftp://origin.url/other/path/`, and a file in `ftp://origin.url/other/path/subpath/somefile` is going to be fetched, the mirror will be considered for being used, and the URL `http://mirror.url/path/subpath/somefile` will be used if the mirror is chosen. Notice that strings are compared and replaced without any pre-processing, so that it's possible to use different schemes (ftp, http, etc) in mirror entries, and even URLs ending in prefixes of directory entries.

Downloading Mechanism

Smart has a fast parallel downloading mechanism, allowing multiple connections to be used for one or more sites. The mechanism supports:

- Resuming

- Timestamp checking
- Parallel uncompression
- Autodetection of FTP user limit
- Cached file validation

and more.

At that moment, the following schemes are natively supported:

- file
- ftp
- http
- https
- scp

Additionally, the following schemes are supported when pycurl is available:

- ftps
- telnet
- dict
- ldap

Removable Media Support

Smart Package Manager implements builtin support for removable media (CDROMs, DVDs, etc) in most of the supported channel types. The following features are currently implemented:

- Mountpoint autodetection
- Support for multiple simultaneous media drives
- Medias may be inserted in any order
- Installed system is guaranteed to maintain correct relations between media changes
- Remote removable media support using any of the supported schemes (ftp, http, scp, etc)

Available Interfaces

Command line interface

Command line interface, with several useful subcommands: update, install, reinstall, upgrade, remove, check, fix, download, search, and more.

Shell interface

Shell interface, with command and argument completion, making it easy to perform multiple operations quickly using a local or remote terminal.

Graphic interface

Graphic interface, offering the friendliness of visual user interaction.

Command line interface with graphic feedback

Command line interface with graphic feedback, allowing one to integrate the power of command line with graphic environments.

Before we begin

As I have mentioned before, there are a few reasons why one might consider smart. One being dissatisfaction with their current package manager, and the other being curiosity.

Whatever the reason, you need to consider why you are doing this, what you hope to get out of this, and how will it benefit you.

I was speaking with a guy not to long ago, and bragging about smart. He finally decided to download smart and install it. However, upon running the graphical interface for smart, he quickly changed his mind. He preferred synaptic. Now the graphical interface for smart is quite plain and simplistic. This offers advantages, mainly that it is straight forward and easy to navigate. However, it lacks the eye candy of other package managers like synaptic. So if your reason is eye candy, stop right here.

On the other hand, if you are dissatisfied due to performance reasons, either dependency resolution, or speed issues, then smart may be the choice. When I talk about smart, I don't talk about it's graphical interface. Graphical interfaces are nice, sure, but usability, stability, and performance are what really matter. It is in these areas that smart simply rocks. But then, even this can be subjective. Applications do not always perform the same way on other computers, and there are various factors involved.

Considerations

So why should I use smart? Smart offers many features that vastly improve package management. The developer of smart, has spent much time researching and implementing features. Gustavo Niemeyer, the lead developer of smart, spoke at length with the people of Slackware. Michael Vogt, is a developer at Ubuntu, and used to be a developer for Debian. Michael Vogt developed synaptic, and is co-developer in smart. Gustavo Niemeyer spoke at length with Jeff Johnson, who used to maintain rpm, and created a fork of rpm, known as rpm-5. But none of this really matters a lot . All it does, is show that Gustavo did his homework, which we expect a developer to do.

So, again, why use smart? As you read in the features section of this book, smart offers several features that put smart ahead of other package managers. One, for example is parallel downloading. Smart will download several packages at once, to speed up the process. By default, it is limited to five downloads at a time. You can of course change that. Those on dial-up, may wish to restrict that to just one at a time. Those with higher bandwidth may want to increase the

number. Part of smart's algorithm is to re-order the packages. This helps in better dependency resolution. There are many other factors in the algorithm, and I'll get into that later. Unlike YaST, where YaST downloads a package, and then installs it, smart downloads all the packages, and then installs it from the cache. This also helps in saving time.

Smart also does mirror balancing, in which if you have mirrors set up, and a repository goes down, while downloading, then smart can switch over to the other mirror with out any problems. Smart also, again when set up, can determine which mirror offers the best performance, and will automatically choose that.

[Psyco](#), I know, what a weird name, is an enhancer for smart. It helps speed up smart, but is also resource intensive. You can disable psyco. Also, psyco is not available for 64 bit architecture.

Dependencies

As I mentioned previously, all packages have dependencies. The trick is knowing the dependencies, and knowing how to find them. Often times, the distributions package management system offers a way to find out the dependencies. I have also listed, for your convenience a table with the dependencies according to the distribution. I deliberately did not include version numbers, and only package names. Version numbers change all the time. Package names rarely change. This way it makes it easy to identify whether or not you meet the dependencies at a glance.

Mandriva/PCLinux OS	Fedora/Enterprise Linux	Debian/Ubuntu	SUSE
GLIBC	GLIBC	GLIBC	GLIBC
rpm-lib	rpm-lib	debhelper	rpm
python-rpm	python-abi	python2.4	rpm-python
python-base	usermode	python2.4-curl	python
gtk2	python	python2.4-gtk2	zlib
python-psyco	rpm	dpatch	popt
usermode-console-only	pygtk	python2.4-dev	gtk2
python	lsb-release		python-gtk
	pygtk2.0		bzip2

- Make sure you have these packages before installing smart. If you want the smart gui to run, then make sure that you have a current gtk2 and python-gtk/pygtk.

Most of these packages are part of a basic rpm distribution installation. GLIBC is standard in all distributions.

Installing Smart

If you're looking for prebuilt packages for your distribution, these links may help you:

- [Packages for Mandriva Cooker](#), by Mandriva. Also check in Contrib
- Packages Red Hat, RHEL, CentOS and Fedora, can be found [Atrpms](#) or [DAG](#)
- [Packages for Debian](#), by Michael Vogt
- [Packages for CCux Linux](#), by CCux Linux
- [Packages for SuSE Linux](#), by Pascal Bleser for openSUSE 10.2 and prior.
- [Packages for SuSE Linux](#), by OpenSUSE BuildService
- [Packages for MacOS X](#), by Jeff Johnson
- [Packages for Fox Desktop](#), by Fox Linux
- [Packages for PCLinuxOS](#), by Torbjörn Turpeinen (Thac)
- [Packages for Slamd64](#)

Installing using an existing package manager

Using a package manager is by far the easiest way to install smart. In most cases, smart is included in your distributions repositories, so it's just a matter of searching for smart using your existing package manager, and then selecting smart. The package manager should handle the dependencies and install smart for you.

Installing the smart binary with out a package manager

The first step is to obviously download the appropriate binary package for your particular distribution. Then use either rpm or dpkg or pkgtools to install the binary. If you have met and fulfilled all the dependencies, then smart will be installed just fine. If you have not met or fulfilled the dependencies, then the default package manager (rpm, dpkg, or pkgtools) will fail and tell you what is needed. Please see the above chapter on **Dependencies**. You will save yourself a lot of time by making sure the dependencies are met prior to installation.

Installing from source

To install from source, unpack package and cd into smart directory and execute as root:

```
python setup.py build
```

- (where x is the version number).

You don't have to install ksmarray unless you want it in the tray.

Using Smart

So you now have smart installed. Now what? Before you do anything else, first understand that a fresh install of smart means the config file and the cache file are blank. Nothing has been set up yet. Now if you have previously been using apt, then it's simple. Your first command in smart would be **smart update**. When coming from apt, smart uses a plugin called aptsync.py that automatically imports your apt list into smart. How very convenient for apt users. A similar plugin was made available for yum users as well. So the same is true if you are coming from yum. A similar plugin is being made for openSUSE users to import from zypper/YaST, but as of this writing, is not yet available.

So what do you do if you are not one of the lucky people to have this plugin? Well, you get to set up the channels first. Channels, in smart, are repository lists. In most package managers it's called repositories. Smart calls it channels. Just like channels on a television. Setting up the channels is your first step. With out this, all smart will do is read your package managers database, so that you can see the software that you have already installed. There are a couple ways you can set up your channels in smart. One is using the command line. The other is with the graphical interface.

Supported Sources





The following channel types are available:

apt-deb	APT-DEB Repository
apt-rpm	APT-RPM Repository
deb-dir	DEB Directory
deb-sys	DPKG Installed Packages
mirrors	Mirror Information
red-carpet	Red Carpet Channel
rpm-dir	RPM Directory
rpm-hdl	RPM Header List
rpm-md	RPM MetaData
rpm-sys	RPM Installed Packages
slack-site	Slackware Repository
slack-sys	Slackware Installed Packages
up2date-mirrors	Mirror Information (up2date format)
urpmi	URPMI Repository
yast2	YaST2 Repository




OK, so I provided a nice list of channel types. But that doesn't mean much now does it? You might be familiar with one or two, and more than likely, that's it. Besides, even if you know one or two channel types, that doesn't mean you know

how to set them up in smart. In the next chapter **Setup Commands** I will go in to how to add, remove and edit channels, and many more commands to help you set up smart. In the meantime, I will go through each of the channel types, and demonstrate what to look for.

The **apt-deb** channel, or repository is obviously for Debian and the various Debian off shoots like Ubuntu, Kubuntu, Xubuntu, Edubuntu and so on. I could go on listing the various Debian variants, but you can find that here: <http://www.debian.org/misc/children-distros>. So what does an apt-deb repository look like? Using Kubuntu as an example; we can see that this repository contains a few things.

	ARCHIVE	21-May-2007 12:27	1.6K
	RSYNC	21-May-2007 12:29	197
	dists/	21-May-2007 12:22	-
	pool/	21-May-2007 12:23	

Now it's the "dists/" directory that is important. If you follow the path by clicking on the "dists/" folder, you can follow it through till you get to the metadata that I mentioned above.

	Packages	27-Sep-2007 00:24	795K
	Packages.bz2	27-Sep-2007 00:24	101K
	Packages.gz	27-Sep-2007 00:24	153K

OK, so here, you can see three files with the name Packages. Two of them have an extension that indicates they are compressed. The bz2 is the highest compression. Now if you click on Packages, you will get to see just how the metadata is done for apt.

```
Package: arts
Priority: optional
Section: sound
Installed-Size: 36
Maintainer: Debian Qt/KDE Maintainers <debian-qt-kde@lists.debian.org>
Architecture: all
Version: 1.5.7-0ubuntu1~feisty1
Depends: libartsc0 (>= 1.5.7-0ubuntu1~feisty1), libarts1c2a (>=
1.5.7-0ubuntu1~feisty1)
Filename: pool/arts/arts_1.5.7-0ubuntu1~feisty1_all.deb
Size: 5710
MD5sum: d1f9cd3e467076bd936340fe13659047
SHA1: 3a1726979ecb857e35fd467c0e7be47909eac27e
SHA256: 05fd5606ff45b0baca53a80664f0e96529000ab7dfc9e1d4506679560b2f6014
Description: sound system from the official KDE release
 KDE (the K Desktop Environment) is a powerful Open Source graphical
 desktop environment for Unix workstations. It combines ease of use,
 contemporary functionality, and outstanding graphical design with the
 technological superiority of the Unix operating system.
```

This metapackage includes the complete aRts sound system, without development packages. aRts is the core sound system for KDE.

Now, that is just one package. You can see it gives the name of the package, priority, section (group), installed size, the maintainer, architecture, version, the dependencies, filename, size, the md5sum (which is very important), SHA1 (a variant of the md5sum), SHA256 (another variant of md5sum, but stricter), and the description.

So for an **apt-deb** channel, this <http://kubuntu.org/packages/kde-latest/> would be the url you would enter for the channel.

Next up is, **apt-rpm**. Apt-rpm is a port of apt made for rpm distributions. Naturally, the url format is similar, but not quite the same. Apt-rpm uses pkglint.core the extension after pkglint, can be anything. This <ftp://ftp.uninett.no/pub/linux/apt/fedora/4/i386/base> give you an example of what I mean. Now, as in apt-deb, the base url is <ftp://ftp.uninett.no/pub/linux/apt/>. As of Fedora 5, they changed from pkglint to rpmmd (rpm metadata). For the release.core, which is a variant of the pkglint on the repository, let's take a look at that metadata.

```
Archive: stable
Component: core
Version: 4
Origin: Fedora
Label: Fedora Core
Architecture: i386
```

So here, they are telling you the product, that this particular directory is the core, the version, the origin, lable, and architecture. The pkglint is a binary file in which apt or smart, or whatever package manager will parse the information.

The **deb-dir** is a directory on the local system that you can select. Say for example, you store your deb packages in ~/Downloads/deb/. Then that would be your deb-dir.

Deby-sys is automatically added to smart when you first run it. This is a core channel, and reads from the dpkg database.

Mirrors are other repositories containing the same information. This is a good thing to have as sometimes a repository will get over loaded, or when a repository goes down. When mirrors are enabled, smart will automatically select the best mirror, and if a mirror has problems, smart will switch to another mirror, if one is available.

Red-carpet channels are rarely used. Ximian developed Red-carpet, and Novell bought out Ximian. Red-carpet from the command line used rug. Open-carper was the open source version of red-carpet. Open-carpet.org is no longer online. An example of a red-carpet channel would be <ftp://ftp.icm.edu.pl/packages/ximian/ximian-evolution-beta/mandrake-91-i586/>. If you notice, down at the bottom of the repository, is a file called packageinfo.xml.gz. This file contains the metadata.


Rpm-dir is similar to the deb-dir I described above. Rpm-dir is just a directory on the local system that contains rpm's. Since smart stores the downloaded packages in /var/lib/smart/packages, I created this directory as my rpm-dir (this could also be done for deb-dir).

Rpm-hdl is a header list repository. In fact hdl stands for header list. A header list can be from a CD or DVD or a repository. Header lists can be used for almost any sort of repository. I'll use SuSE as an example here. Here is an example of a header list: <ftp://ftp.suse.com/pub/suse/i386/9.3/suse/directory.yast>. The metadata there is the directory.yast. Now the base url <ftp://ftp.suse.com/pub/suse/i386/9.3/suse/>, which contains the directory.yast.

In **rpm-md**, the md stands for metadata. This is becoming ever popular among rpm distributions. Yum uses it, later versions of apt4rpm use it, YaST2 uses it, and Zypper uses it. For Fedora Core 8 and example of rpm-md can be found here

<http://ftp.belnet.be/packages/dries.ulyssis.org/fedora/linux/8/i386/dries/RPMS/>.

Notice that at the top is a directory called "repopdata". In "repopdata/" You will find several files:

 filelists.xml.gz	542K	25-Feb-2008 23:40
 other.xml.gz 250K		25-Feb-2008 23:40
 primary.xml.gz 647K		25-Feb-2008 23:40
 repopmd.xml 951		25-Feb-2008 23:40

These files contain the metadata information for rpm-md. Now openSUSE has added an xslt which styles the xml information. Another nice feature of this is, you can add

<http://ftp.belnet.be/packages/dries.ulyssis.org/fedora/linux/8/i386/dries/RPMS/repopdata/> to your RSS news reader.

Just like in deb-sys, **rpm-sys** is the same thing. It is automatically set up on the first run, and reads from the rpm database.

Slack-site is obviously a Slackware repository. An example of a slack-site is <ftp://ftp.slackbuilds.org/12.0>.

Slack-sys is just like rpm-sys and deb-sys. It is automatically added upon first run, and reads from pkgtools database.






With **up2date-mirrors**, these are mirrors of other repositories in up2date format. Keep in mind, that up2date uses the yum format. Which again, an example would be like this

<http://download.fedora.redhat.com/pub/fedora/linux/core/6/i386/os/>

With **urpmi**, the repositories are a bit different. As shown here; <ftp://ftp.easynet.fr/plf/mandriva/2008.0/free/release/binary/i586/> you'll see that you have a lot of rpm's. What makes this a repository, is the file called hdlst.cz.

This file is what holds the metadata.

With **yast2**, it is similar to rpm-hdl. Earlier, I mentioned how that rpm-hdl can be for a variety of repositories, and I showed how that directory.yast is one. With yast2, this file is place in another directory. A directory called media.1/. We see here; <http://download.opensuse.org/distribution/10.3/repo/oss/> that there is a directory.yast file right there in that directory. However, the one we want, is found <http://download.opensuse.org/distribution/10.3/repo/oss/media.1/>. You'll notice in this directory it contains the following.

 directory.yast	27-Sep-2007 18:51	41
 media	27-Sep-2007 16:48	42
 products	27-Sep-2007 16:48	25
 products.asc	27-Sep-2007 18:35	189
 products.key	26-Sep-2007 19:41	2.1K

This was an overview of the various formats supported by smart. I could write more on each, but then we'd have a book on each of them.

Setup Commands

Action Commands

Query Commands

Configuration

General

Options	Value	Description
log-level	ERROR, WARNING, INFO, DEBUG	how verbose smart messages should be
psyco	True, False	whether to use the Python optimizer or not
default-gui	gtk, qt	Set the default GUI (QT not yet implemented)
explain-	True, False	force --explain on all operations

changesets		
last-update		shows the time of the last successful update
text-hide-version	True, False	show if the changeset has less than 40 actions. If the changeset has more than 40 actions (install/remove), it's false, if it's less, it's true.
prefer-removable	True. False	Smart will use removable channels (e.g. CDROM) rather than fetching from a remote location, if both options are available

Locations

Options	Value	Description
data-dir	directory	root of Smart control directories
user-data-dir	directory	user-specific control directory
default-localmedia	directory	Set the directory where your media is

Cache

Options	Value	Description
disk-cache	True, False	whether to enable smart's metadata cache on disk
remove-packages	True, False	whether to remove downloaded packages after a successful transaction

Network

Options	Value	Description
http-proxy	url	use given proxy on HTTP downloads
ftp-proxy	url	use given proxy on FTP downloads
socket-timeout	seconds	timeout for download operations
keyserver	server	some server (e.g. wwwkeys.uk.pgp.net)

For a list of keyservers see [here](#).

Channel handling

Options	Value	Description
detectlocalchannels-maxdepth	some numeric value	max directory recursion level when adding local directory channels
force-channels	channel(s)	Channels that should always be considered as enabled, even when marked disabled
detect-sys-channels	True, False	Whether to detect local system channels when starting Smart
channel-sync-dir	directory	By default it's /etc/smart/channels (not distro specific), import channel information from files on that directory (the files are read only once, either upon creation or changing)
force-channel-sync	True, False	force channel sync from channel-sync-dir on every Smart initialization

RPM specific

Options	Value	Description
rpm-root	directory	what root directory to use to install the packages (rpm's --root option)
rpm-check-signatures	True, False	whether RPM package signatures should be verified before installing
rpm-justdb	True, False	only install RPM package information into the RPM database but don't install the package files (see rpm's --justdb option)
rpm-noconfigs	True, False	don't install files marked as configuration files
rpm-nodocs	True, False	don't install files marked as documentation files
rpm-excludedocs	True, False	same as rpm-nodocs
rpm-nomd5	True, False	don't verify the RPM integrity using its MD5 hash sum
rpm-noscripts	True, False	don't execute post-installation scripts that are in the RPM package

rpm-notriggers	True, False	don't execute trigger scripts that are in the RPM package
rpm-test	True, False	don't perform the package installation for real, just test if it would work (rpm's --test option)
rpm-force	True, False	Install anyway
rpm-repackage	True, False	Re-package the files before erasing. The previously installed package will be named according to the macro <code>%_repackage_name_fmt</code> and will be created in the directory named by the macro <code>%_repackage_dir</code> (default value is <code>/var/spool/repackage</code>)
rpm-log-level	emerg, alert, crit, err, warning, notice, info, debug	Set how much detail you want
rpm-allfiles	True, False	Installs or upgrades all the missingok (missingok indicates that the file need not exist on the installed machine.) files in the package
rpm-order	True, False	Reorder the packages for an install. The list of packages would normally be reordered to satisfy dependencies. (by default smart is set for true)

DEB specific

Options	Value	Description
deb-root	directory	What root directory to use to install the packages
deb-arch	pentium, i386, sparc64, sparc, powerpc, mipseb, mips, shel, sh, x86_64, amd64	Set architecture
deb-admindir [∞]	directory	Change default directories
deb-instmdir [∞]	directory	Change default directories

Slackware specific

Options	Value	Description
slack-root	directory	What root directory to use to install the

		packages
slack-packages-dir	directory	What directory to use to install the packages

Packaging for Smart

Sample specs

Coding for smart

The smart API

Understanding Python

Bash

C

Writing patches

Troubleshooting Smart

Error messages

Strace reports?

How to interpret those cryptic error messages

Where to get help

Mailing List

You may subscribe to the mailing list by:

- Using the [web interface](#),
- Sending a message to smart-subscribe@labix.org

Messages to the list should be sent to smart@labix.org (subscribers only).

Archives can be found at:

- [Current mailing list](#) (starting from September/05)
- [Old messages](#) (up to September/05)

Issue Tracker

Using the [issue tracker](#) you may:

- [Open a new issue](#)
- [View current issues](#).

The issue tracker has moved some time ago. The old issues are still at the following address:

- <http://smartpm.python-hosting.com/report/1>

Please, do not post new issues in the old tracker.

IRC

Talk with the developers and other Smart users at **#smart** on irc.freenode.net.