A cute introduction to Debtags

Author: Enrico Zini
Contact: enrico@debian.org

Revision: 1.0
Date: 2005-05-31

Copyright: GNU GPL v2 or later.

Abstract

The Debian archive is getting larger and larger, and the software more and more diverse and complex. Organising software in the archive is difficult, and the existing section system, designed to cope with a much smaller number of packages, is no longer sufficient. The goal of Debtags is to provide a working alternative for categorising software that can cope with our numbers.

The core idea of Debtags is to adapt the technique of Faceted Classification to be used for our packages. Faceted Classification is a 70-years-old library science technique which is being rediscovered and loved by modern Information Architects.

Debags attaches categories (we call them *tags*) to packages, creating a new set of useful structured metadata that can be used to implement more advanced ways of presenting, searching, maintaining and navigating the package archive.

Example uses of Debtags include searching for software, browsing the archive, and filtering out unwanted groups of packages.

The Debtags effort needs to face three major problems:

1. Creating a suitable vocabulary of categories.
2. Categorizing the vast array of packages.
3. Having applications make use of Debtags data.

All three issues are being actively addressed with good results:

- Debtags has already acquired a large set of tags, even if the set is in continuous need of refining;
- a large part of our package archive has been at least partially categorised, and there is a tool called debtags-edit that every developer and user can use to categorise the packages they know best;
- a new library called libapt-front is being developed as a smart front-end to `libapt` which can also access other data sources, such as Debtags, popularity contest (popcon) results, debram metadata and more.

This paper gives a broad technical overview of the Debtags project, its theoretical foundations, and the tools available for it now. The paper also offers some practical tutorials on how to do all sort of nice Debtags tricks.

Contents

Introduction 2
- Lots and lots of packages 2
- Archive sections 3

Debtags theoretical foundations 3
- Classifying software 4
- Faceted Classification 5
- Some use case examples 7
- The design of Debtags 7
  - Facets 7
  - Tags 8
  - The Vocabulary 8
  - The tag database 9

Using debtags 10
- debtags 10
- debtags-edit 12
  - Running debtags-edit 13
  - Searching packages 13
  - Tagging packages 13
- packagesearch 14
- packagebrowser 15
- debram 15

Where to go from here 15
- Contributing to the categorization 15
  - Using debtags-edit 15
  - Using debtags 17
  - Using tagcolledit 18
  - Automated tagging 19
  - Adopting a tag or a facet 19
- Contributing to the vocabulary 19
- If you are a package maintainer 20
- Integrating Debtags in other applications 20
  - libdebtags1-dev, python-debtags, libdebtags-perl 20
  - libapt-front 22
- User interface issues 22
  - Effective View Navigation 22

Conclusions 23

Bibliography 23

Introduction

Lots and lots of packages

As of May 24, 2005, my Debian unstable system counts 16769 different binary packages, and I feel very powerful.

I can access at least three complete office suites, various painting programs (of which at least one is designed for kids and one for world-famous digital effect movie studios), software for massively parallel quantum chemistry, GIS geographical tools, dental office management software, and, my favourite, a tool to have a cute cow read my presentations:

```
 _________________________________
/ a tool to have a cute cow read \
\ my presentations:              /
 ---------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

As a Debian user, I am very powerful. However, there is a simple question that I'm having a hard time answering:

```
What do you want to install in your computer?

[ ] 3dchess      3D chess for X11
[ ] 3ddesktop    "Three-dimensional" desktop switcher
[ ] 44bsd-rdist  4.4BSD rdist.
[ ] 6tunnel      TCP proxy for non-IPv6 applications
    [... 16765 more options follow ...]
```

And another one:

```
What do you want to remove from your system to make up some space?

[1563 options follow]
```

If I search the package archive, I get:

- apt-cache search web browser: 197 results.
- apt-cache search text editor: 170 results.
- apt-cache search image editor: 22 results, but Gimp is NOT among them.

Now, I profoundly believe that having choice is really good: what we need is not to reduce the size of the archive, but to create some way to make it easier for people to find what they need.

What does it mean, "make it easier"? Here is a nice measure: "in front of a list with more than about 7 plus or minus 2 items, our brain goes banana" [ZEN] [MILLER].

Archive sections

The way we are currently organizing packages is by grouping them in sections: Debian main has about 16197 binary packages divided in 33 different sections. However, this means an average of 490 packages per section, which is too many.

In fact the problem is even worse than this, because some sections are more lightly subscribed than others. After all, the more packages a section has, the more likely that the one package you want is found therein. By this metric, the Debtags team calculates that the typical Debian package is found in a section of 785 binaries. Such a number approaches the size of *the entire archive* at the time when the section system was first introduced.

Increasing the number of sections would not help too much: to have an average of 20 packages per section we would need 800 different sections: but then, how does one choose among 800 sections?

Besides the numbers, current software is getting more and more complex: for example, which section should a full-featured web browser such as Mozilla be put in? `net`? `web`? `mail`?

Sections were fine when Debian was smaller and simpler. Now there is a bug (#144046: Sections are not finely grained) asking for something better.

That bug has been assigned to Debtags. And Debtags is going to close it.

Debtags theoretical foundations

The original idea of Debtags' was just to allow more than one section per package. That would have allowed us to categorize Mozilla under `net`, `web` and `mail` simultaneously, allowing the user to find it under whichever classification they had in mind.

That is how we started, and that is how we got stuck. What do we mean with `net`? Some possibilities are:

- The package has a program that can use information not stored in the local computer;
- The package has code which invokes the socket() system call;
- The package analyses firewall logs;
- The package is useful to configure and maintain a network;
- All of the above.

All of these are probably valid interpretations. And then, why shouldn't all of Gnome belonging to `net`, since the 'N' in GNOME means 'Network'?

Classifying packages is a bigger problem than it seems at first sight. Luckily I suspected that someone has been thinking about it already, and I did some research.

It turned out that there is a whole new science calling "Information Architecture" whose goal is to sort out the Information Mess of the Information Age.

It also turned out that our problem had been solved somewhere in 1933, but they forgot to leave me a note.

Classifying software

Figure 1: Shiyali Ramamrita Ranganathan

Once upon a time in India, a mathematician and librarian called Shiyali Ramamrita Ranganathan started a secret project to innovate software categorization for Debian. It was about 1931, and he did not want to use the words "Debian" and "software" just yet, so he disguised all of his work under words such as "library", "books" and "library science" [STECKEL].

Ranganathan is famous for his five laws of library science:

1. Books are for use.
2. Every reader his or her book.
3. Every book its reader.
4. Save the time of the reader.
5. The Library is a growing organism.

If we do a simple word substitution, we obtain the perfect rules for classifying Debian packages:

1. Software is for use.

2. Every user his or her software.
3. Every software its user.
4. Save the time of the user.
5. Debian is a growing organism.

These rules look simple, yet they are important and deep: rule number 1 tells us that software is there to be used, which means that people need to know about it and find it. This is the main reason for organizing packages: to allow people to use them.

Rules number 2 and 3 tell us that not every software is appropriate for every user, and users need to be able not only to find software, but also to find the software which is appropriate for them.

Rule number 4 tells us that the research must be quick and efficient, and the user should not spend a lot of time trying to search through long lists of packages, or learning a complicated search system.

Rule number 5 says that Debian keeps growing and evolving, and any system we design will have to cope with that.

This is exactly the purpose of Debtags. Codified by Ranganathan in 1931.

Faceted Classification

Ranganathan has been working all his life to solve the problem of classifying books in big libraries. To do so, he designed a system that he called *Faceted Classification* or *Colon Classification*:

> A faceted classification [is a way of classification that] uses clearly defined, mutually exclusive, and collectively exhaustive aspects, properties, or characteristics of a class or specific subject.
>
> *Wynar, Bohdan S. "Introduction to cataloging and classification". 8th edition. p. 320*

In other words, with faceted classification we have more than one set of categories: one for each *aspect* of our packages. Every different aspect of an object is described using a separate *set of categories*, called "*facet*".

Ranganathan has also given suggestions on how to identify the facets to categorise. He suggested that there are 5 fundamental kinds of facets, which are sometimes addressed as "PMEST" because of their initials:

Personality what the object is primarily *about*. This is considered the "main facet."

Matter the *material* of the object.

Energy the *processes* or *activities* that take place in relation to the object.

Space *where* the object happens or exists.

Time *when* the object occurs.

If at first sight this seems unrelated to software, remember that Ranganathan was talking about books, and we need to do some mapping. For example, pixelcharmer has been mapping the PMEST to blog entries:

- Personality = topic

- Matter = form
- Energy = process
- Space = I don't believe I've used this facet, or perhaps it's the permalink of the post itself? Yes, let's go with that for now.
- Time = date

Let's follow her steps and map the PMEST to Debian packages:

Personality What the package is primarily *about*. Answers the question *what is it?*

Matter The "material" that constitutes the package. Answers the question *what is it made of?*

Energy The processes or activities that take place in relation to the object. Answers the question *what do I do with it?*

Space Where the object happens or exists. Answers the question *where do I use this?*

Time When the object occurs. I still haven't found a suitable mapping for this: it might just be that Debian packages exist a bit outside of time, and that would explain how come Sarge took so much to release.

In Debtags theoretical foundations I gave an example of many different ways to interpret the section `net`. We can try to look at some of them again using the PMEST:

- "The package has a program that can use information not stored in the local computer". Is storing such information the main purpose of the program? Then the program's *personality* would need to be categorised as related to the network. Else, the program's *energy* would be categorised as network-related instead.
- The package has code which invokes the socket() system call. This is about the *matter* of the package: the technology it uses, the functions it invokes.
- The package analyses firewall logs. This would mean that the *personality*, or the *energy* of the program is related to the network, but *not its matter*: if we look at how the package is made, we mainly see a text parser.

The PMEST is useful in that it gives us more mental structure to understand the properties of an object and to resolve the ambiguities in its categorisation.

Some use case examples

Let's see how this could help, with three examples:

- User A wants to find a simple software to manage their collection of audio CDs, which integrates nicely with his or her Gnome desktop;
- Developer B wants to write a software to manage a collection of audio CDs, and is looking for code and examples in existing applications and libraries;
- System administrator C works for a radio broadcaster and wants to set up a mail interface that allows people to search the radio collection of audio CDs; he or she is looking for the right existing tools to get it done.

Now, let's look better at what they want:

- User A is interested in integration with Gnome, not implementation details.
- Developer B mainly cares about implementation details, to find good examples to build from.
- System administrator C is only interested in commandline applications (no matter how they are implemented) or alternatively perl libraries.

User A will probably start searching through Gnome software. User B will probably start searching among software implemented in C. User C will probably start searching for commandline tools. And they will probably all look for software to manage an audio CD collection.

These three people are looking at the package archive from different angles.

Luckily, Faceted Classification does just that: it categorizes the items from different aspects. If we do things right, there will be something like a 'Desktop' facet, an 'Implemented-in' facet, a 'Kind-of-interface' facet and a 'Purpose' facet, each of these facets categorised with their sets of tags; and everyone will be able to find what they are looking for.

The design of Debtags

The key concepts of Debtags are *Facets*, *Tags*, the *Vocabulary* and the *Tag Database*. Let's consider them one by one.

Facets

Facets are the aspects of our packages that we choose to categorise. Example of facets in Debtags are `implemented-in`, `culture`, `interface`, `media`.

Each facet defines an aspect of packages we look at. For each facet there is a set of categories that we use to describe what we see when we look at that aspect of a package.

Faceted categorization is like saying, "If I look at what is the use of this package, I see *editing*". "If I look at what is its user interface, I see *a text-mode user interface*".

Facets are "If I look at X"; tags are "I see X".

The list of facets currently categorised by Debtags is available in The Vocabulary.

Tags

The *tag* is a Debtags category. Debtags uses different sets of categories, that we call tags, to categorise different facets of software. We use the word tags instead of categories because it's shorter, and we talk about tags a lot.

In Debtags, we represent tags with a short English name, prefixed by the name of the facet it categorises. For example, when we look at what media types a package can work with and we categorise "The Gimp", we use the tag `media::rasterimage`.

The Vocabulary

The Vocabulary is where we store the list of facets and tags that we use for Debian packages.

Having a common vocabulary is important so that we all refer to the same property with the same name. Other benefits of the common vocabulary include the abilities to present selections of facets and tags in a user interface, to store meta-information about them (such as a short and long description, possibly in many languages), and to validate classification information.

The Debtags vocabulary is the file that delineates the facets and lists the respective tags that can be used to categorize Debian packages. The vocabulary file contains whatever needs to be stored about facets and tags. At the moment this means:

- a list of facets and their tags;
- description of facets and tags;
- editorial information;
- comments.

This information is stored in a format similar to the usual Debian package records. For example:

```
Facet: implemented-in
Description: Language that the package is implemented in
Nature: matter
Status: complete

Tag: implemented-in::c++
Description: C++
```

This defines a facet called `implemented-in` and a tag `c++` inside it.

The main location of the vocabulary is in the subversion repository at `svn://svn.debian.org/debtags/vo` from there, it makes its way to the default Debtags tag source and then, by means of `debtags update`, it is stored locally in `/var/lib/debtags/vocabulary`. It is also indexed by `debtags update` to allow applications to quickly lookup entries inside it.

There are currently 32 facets defined, containing a total of 440 tags. To see the facets along with their descriptions and other data, you can run `debtags update` and then use `grep-dctrl`:

```
grep-dctrl -FFacet -r . /var/lib/debtags/vocabulary
```

A tutorial on sending patches to the vocabulary can be found at http://debtags.alioth.debian.org/vocabulary.html

The tag database

A tag is assigned to a package by entering the appropriate information in the Tag Database.

Designing this database is one of the most tricky and important implementation details of Debtags, as it is the way to share the tags and make them useful.

An obvious place for storing the tags is in the control file of the packages, together with the other package metadata. This possibility has been discarded, however, for a couple of good reasons:

- The aggregated package database is already big enough, and there are no clear short term ways of addressing this problem

- "Debian is a living organism": tags can change fairly often, because the classification is refined, a new tag is introduced or a whole new facet of packages is classified, and we cannot afford a new package upload or ftpmaster intervention every time such a change happens.

We chose instead to store tag data in a central tag repository, located at http://debian.vitavonni.de/packagebrowser, from which it can be downloaded in the user's computer by means of the `debtags update` command, just like the package database can be downloaded by means of `apt-get update`.

This makes it possible to store and maintain the categorization data without causing an added burden to the Debian infrastructure. It also allows to have different *tag sources* merged together. Check this out because it's cool!

Let's make an example with a fictitious Debian-Circus CDD (Custom Debian Distribution) that chooses to categorise packages also from the aspect ("facet") of their possible use in a circus. To do so, they want to have in Debtags a `circus` facet, categorised with tags such as `travel-info`, `installation-procedures`, `inventory`, `artist-management`, `show-management`, `propaganda`, `accounting`.

To realise this, Debian-Circus can create their own piece of Vocabulary and Tag Database and make it available to be downloaded as a *tag source*. `debtags` is able to download data from various tag sources (listed in `/etc/debtags/sources.list`) and merge them together.

Different tag sources can be merged easily when facets differ, and they complete each other's view of packages by shedding light on more aspects of them.

Now, Debian-Circus provides categorization along the `circus` facet, and Debian provides categorization for the `suite` and `media` facets. Any circus sysadmin can now merge this information together and look for `circus::propaganda` software for `suite::gnome` that can work with `media::vectorimage`.

Isn't it exciting?

The main location of the database is inside Erich Schubert's packagebrowser at http://debian.vitavonni.de/packagebrowser. From there, it makes its way to the default Debtags tag source and then, by means of `debtags update`, it is merged with other tag sources and stored locally in `/var/lib/debtags/package-tags`.

The database is stored locally in an easily parsable format that can also be understood by tools such as `tagcoll` and `tagcolledit`. It is also indexed by `debtags update` to allow applications to look entries up quickly in it.

The individual user can modify the Debtags database locally. Such local modifications are stored as *tag patches* in the file `~/.debtags/patch` in the user's home directory. A tag patch is a special patch format that can represent a change to a tag database, and can also be applied to future versions of the database, so that your changes are preserved across `debtags update` invocations.

Another useful feature of the tag patches is that you can send them to the central database, which will integrate them with the rest of the data. You can do this by using debtags-edit (it has a "*File/Mail changes to central database*" feature) or by using the command `debtags send`.

Using debtags

Debtags is not only categorization data, but also a suite of tools that allow to work with the data:

debtags Commandline interface to libdebtags functions and Debtags administration tool.

debtags-edit GUI application to search and tag packages.

tagcoll Commandline tool to manipulate generic collections of tagged items.

tagcoll-edit GUI application to perform mass-editing of collections of tagged items.

libtagcoll1 Library providing generic functions to manipulate collections of tagged items .

libdebtags1 Main library with Debtags functions, and wrappers to many languages.

Erich Schubert's packagebrowser Central tag archive, browsable and editable on-line.

autodebtags Experimental tool to perform some automatic and heuristic tagging tasks.

The suite is quite large and diverse, and it allows to do many different kinds of analyses and manipulations of the tagged data.

If you are new to Debtags and you are looking for some function, chances are that it is already implemented somewhere: if you cannot find it, don't hesitate to ask in the debtags-devel mailinglist.

debtags

These are some example uses of `debtags`:

`debtags update` Updates the local Debtags information.

`debtags stats` Prints some statistics about Debtags.

`debtags show mutt` Like `apt-cache show mutt`, but also shows the mutt's Debtags tags.

`debtags grep 'use::editing && media::rasterimage'` Shows all packages that allow to edit raster images. You can use full arbitrarily complex boolean expressions such as `(use::playing && ! use::recording) && media::audio && (interface::commandline || interface::text-mode)`.

An interesting one is this: `!culture::* || culture::italian`: it shows all packages that are either not locale-specific or that are specific to my locale. This can be used in package managers to filter out packages for locales not used by the system.

`debtags tagshow use::editing` Shows the vocabulary information about the tag `use::editing`.

`debtags tagsearch edit` Shows all the tags whose information (such as the name or the description) matches the string "edit".

debtags cat Outputs the entire tag database, including local modifications. It is useful to provide data to other packages such as `tagcoll` or `tagcolledit`.

debtags todo Prints a list of the packages installed in the system that are not yet tagged.

debtags todoreport Generates a report of packages that probably need to be tagged better.

debtags-edit

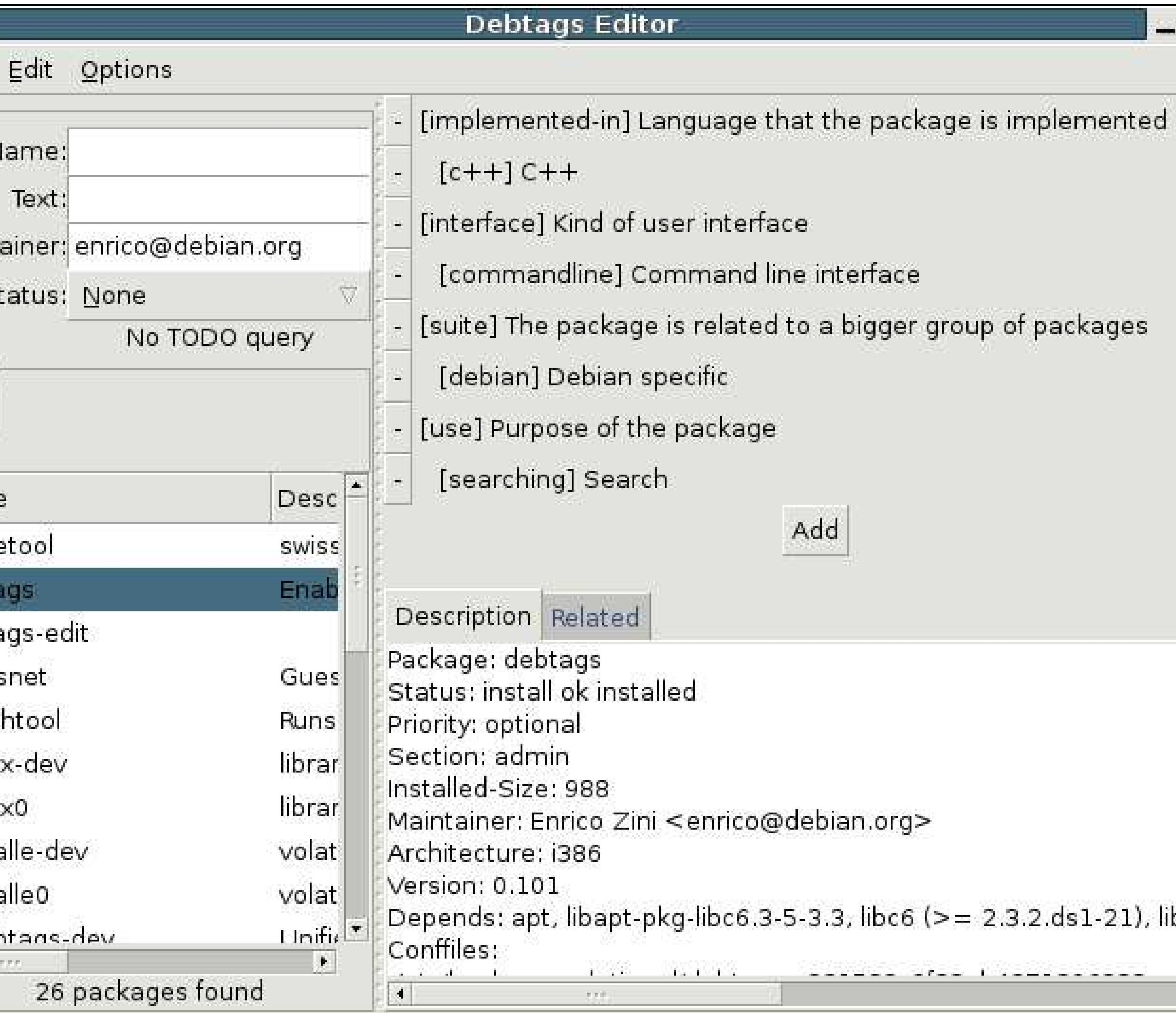

Figure 2: Screenshot of the main window of `debtags-edit`.

`debtags-edit` is a graphical interface for searching and tagging packages.

Since I'm always busy experimenting with new features, the interface is quite rough and I'll have to explain it a moment.

Running debtags-edit

Try running `debtags-edit`. If it does not work, it might be because you have not yet run `debtags update` to download the Debtags data: do it now and run `debtags-edit` again.

Searching packages

On the left-hand side of the application window there is a *Filter* area which allows you to look for packages: you play with the controls, and a list of matching packages appears in the bottom left part of the screen. A notable part of the filter area is the *Tags* area, which lets you add tags to the filter.

Try pushing the *Add* button of the tags area and add something like "Use/Editing": the list will display all packages which can edit something.

Now push "Add" again: the list of facets and tags has narrowed down to only the available possibilities, and if you look inside the "Media" facet, you'll see a list of the possible kinds of media that can be edited using Debian packages.

> **Note**
>
> If this example does not work for you, check what is in the "Maintainer:" field: there might be your e-mail there. That is because `debtags-edit` wants to kindly suggest you to tag the packages you maintain[1]. For the purpose of this example, you can clear the "Maintainer:" field and try again. After I tell you how to tag packages, please put your address back there ;)

Using the filter can search packages by name, do a full-text search à-la `apt-cache search`, search by maintainer, installed status or tags. This makes debtags-edit an interesting tool already, but let's see more.

Tagging packages

You can now click on a package name in the bottom-left part of the application: that will display the package in the main display on the right side. At the bottom part you will see all the package data, very much like the output of `apt-cache show`, while in the top part you will see the tags of the package. If the package you chose has no tags, try looking for `debtags`: that will be nicely tagged.

Now try adding a new tag to `debtags`: try adding "Games/Toys". Push the other "Add" button below the tag list in the top-right part of the window, and navigate the complete facet list until you find the "Games" facet: inside it you will find "Toys". Select it: the tag has been added, and you made your first step into Debtags tagging!

Now try doing "File/Save", then go to a terminal and type `debtags show debtags`: you will notice that your new games::toys tag is already there. Cool, isn't it?

What happened is that `debtags-edit` saved your modifications in `~/.debtags/patch`, and all Debtags-aware applications read that file at startup. Have a look at the file: it's

[1] If `debtags-edit` cannot find the `DEBEMAIL` environment variable, it will instead leave the "Maintainer:" field empty and preselect "Status: Installed", kindly suggesting you to check if the packages you know and use everyday are tagged well enough.

just a patch with the change you made, and its format is such that it can be applied to any future version of the tag database. Isn't it smart? I'm so proud of it!

Now go back to `debtags-edit` and look in the "File" menu: there's a "*Mail changes to central archive*" option. If you click on it, it will mail the contents of your `~/.debtags/patch` to the central archive, and the next day your contribute will be available to everyone when they will do `debtags update`.

What else do you need to contribute to the categorization? Only one more thing maybe: subscribe to the debtags-devel mailing list.

packagesearch

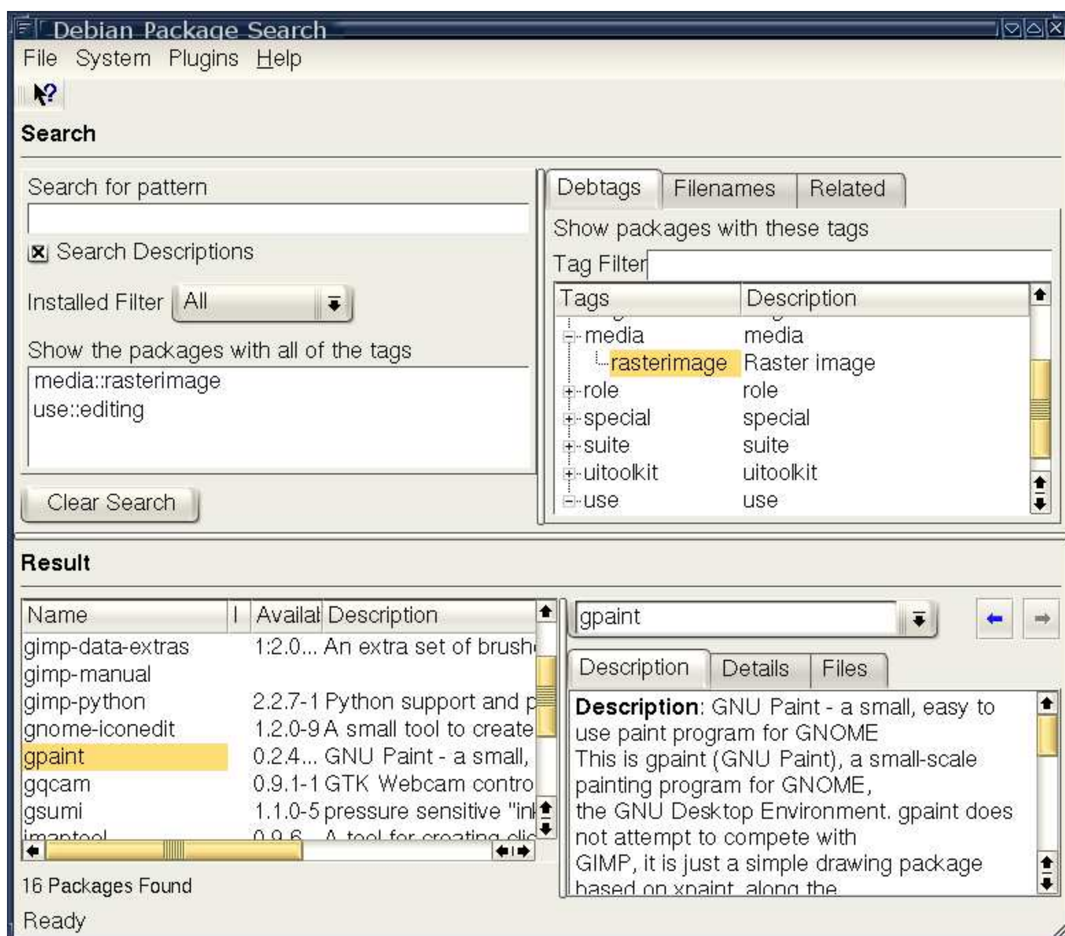

Figure 3: Screenshot of the main window of `packagesearch`.

packagesearch is a package search program written by Benjamin Mesing. Among its many features, it is able to make use of Debtags data.

To use packagesearch, you set search parameters in the top part of the application, and it displays the results in the bottom. The Debtags search is on the right side, on a tab named "Debtags".

Try running packagesearch, then select the Debtags tag "Use/Editing": you will see all the packages that can edit something. Then select "Media/Raster Image" and you will see all raster image editors in Debian that are known to Debtags.

Compared to debtags-edit, packagesearch does not allow to change the categorization, but it features more search functionalities, and it also integrates `apt-file` searches.

packagebrowser

Lastly, the central tag archive has a web interface (the *packagebrowser*, written by Erich Schubert) that can be used to navigate the Debtags data and edit them online. You can find it at http://debian.vitavonni.de/packagebrowser/.

One way of working with the packagebrowser is to help with the not yet tagged packages, following the link at the top of the main page.

Another way is to work on packages is from the Debtags TODO report, in which all package names directly link to their edit page.

> **Note**
>
> The packagebrowser edit page lists all available tags and is very long. However, modern browsers have very quick and convenient search functions such as the "/" function in Firefox. Using those functions it becomes quite fast and easy to find what you need in the list of tags to add.

debram

In the words of its developer Thaddeus H. Black, Debram was `a primitive attempt to treat the same Debian package problem Debtags more properly solves.` Today, Thaddeus has enthusiastically joined Debtags development and is now in the process of merging the large body of Debram data into Debtags. Unable to attend Debconf5, Thaddeus has asked me to recommend to Debram users that they migrate directly to Debtags at their convenience. Post-sarge, Debian Maintainers need not tag their packages in Debram at all; standard Debtags tagging suffices.

Where to go from here

Contributing to the categorization

> *Tag your packages, you will*
>
> (Master Yoda)

Using debtags-edit

These are some examples on how to use debtags-edit to do some more serious categorization work. Most of them are ways of identifying packages that need some work: once you find them in debtags-edit, you can directly work on them

1. Fire up debtags-edit and set the filter to display all installed packages with the tag `special::not-yet-tagged`. Now look at the results, pick something you know and add tags to it. After you are satisfied with its categorization, remove the not-yet-tagged tags and submit.

2. Click on "Edit/TODO dialog": a dialog will open with various TODO options:

   Empty tagset lists all packages with no tags at all. Those are usually not reported, because the central repository should automatically add `special::not-yet-tagged` tags to them; however this is sometimes not the case, especially for local packages or packages that have recently been added in the archive, so this option is available to spot such packages.

   Uitoolkit and not interface lists all packages which have `uitoolkit` tags but not `interface` tags. This might be a normal situation, especially for software libraries, but for normal applications it is probably an anomaly worth checking.

   Uitoolkit and not implemented-in lists all packages which have `uitoolkit` tags but not `implemented-in` tags. This might be normal for documentation about user interface toolkits, however it is an anomaly for applications and libraries, and it's worth checking as well.

   Missing Role Ideally every package should have a kind of role in the distribution. However, there are lots of packages which do not yet have a `role::*` tag. Since the tags in the Role facet are still under discussion, this function is good to bring up food for ideas.

   Specials The options starting with "Specials" allow to see packages whose tag sets are special cases compared to all the other packages with similar tags. How this is computed is a bit complex, but this is proving to be very useful to highlight some corner cases that lead to extra reasoning and improvements of the vocabulary.

3. Click on "Edit/Facet dialog" to activate an experimental new feature of debtags-edit to find even more corner cases.

   This will pop up a dialog with two columns: the *Has* column and the *And not* column. Both columns list facets.

   If you click on a facet on the "Has" column, the "And not" column will list all the other facets, together with a number. The number represents how many packages have the "Has" facet, but do not have that "And not" facet. The "And not" facets are sorted by increasing values of this number.

   Sometimes all the "And not" facets have similar numbers, but sometimes there are facets with numbers that are definitely smaller than the others. Those facets are very likely to have something in common with the current facet in the "Has" column, since a lot of packages have tags from both facets.

   For instance, it happens at the time of writing that, when selecting the "Game" facet on the left side, the corresponding "Use" facet reports a count which is one order of magnitude less than the others. This is because of the existence of `use::gaming`, which is normally found in most, if not all, `game::*` packages.

What are then those packages that have `game::*` but not `use::gaming`? Let's see them: click on "Use" on the right column, then click ok. That will set `game:: && !use::` as the base for the package list, and further filtering will be based on them. You can now see what are those packages, and if you want you can check which of them are installed in your computer.

With `game::*`, it turns out that there is a reason: there are games (many of which are `game::toys`) that are not useful for gaming. Some examples are cappuccino, cowsay, polygen and fortune. However, one could realise that these toys have some other use before gaming.

Maybe a new "use::*something*" tag is needed?

The Facet Dialog has helped finding out.

Using debtags

debtags can be useful for tagging in two ways: both to give you hints on what needs to be worked on, and as a low-level tool you can use if you want to automate tagging tasks.

There are three features of debtags you can use to get hints on what to do:

`debtags todo`: `debtags todo` prints a list of packages you have installed which are tagged with the `special::not-yet-tagged` tags. No need for further explanations: tag them!

`debtags todoreport`: this prints a text report about packages that need some work to do. It includes a range of simple to very complex checks to find out problematic areas of the categorization, and the report that it outputs contains explanations about the reasons behind every group of packages. The report also includes packages that are not installed.

The output of `debtags todoreport` is periodically processed with rst2html and published at http://debtags.alioth.debian.org/todoreport.html

`debtags related`: `debtags related` shows you what packages are similar (tagging-wise) to a given one. Try running `debtags related cdcd`: you will see a list of other CD players. You can use the `-d` switch to increase the *distance* of the search, that is, to include more packages that are "a bit less similar".

You can use this function to check if the list of related packages is what you would expect: if not, you can check the tags of the missing or extra packages to see if there is anything wrong.

You can also use debtags as a low-level backend if you want to experiment with scripted or automated tagging ideas:

`debtags tagsearch` and `debtags tagshow`: They allow you to search the tag vocabulary and show information about a tag.

`debtags tag ls`, `debtags tag add` and `debtags tag rm`: These commands are intended to be run from scripts, and they list the tags attached to a package and allow you to add or remove tags to it. Try this:

```
debtags tag ls debtags
debtags tag add debtags game::toys
debtags tag ls debtags
debtags tag rm debtags game::toys
debtags tag ls debtags
```

Using tagcolledit

`tagcolledit` is a generic tool to do mass-editing of tags. It is not specific for Debtags, so it does not know about packages, but compared to debtags-edit it can make it easy to do large-scale operations like renaming of tags.

Let's run `tagcolledit`:

```
# Get a version of the tag database including the local patch
debtags cat > /tmp/tags
tagcolledit /tmp/tags
```

The interface of `tagcolledit` is unfortunately very rough, just like debtags-edit, and it needs some explanation. It is divided in two panels, like Midnight Commander, and every panel works in the same way: it has a filter, and a list of matching items (that are packages in our case).

Now try doing this: in the left panel, select the `uitookit::qt` tag. In the right panel, select the `implemented-in::c++` and the `interface::x11`. Now, QT is a C++ library, so one would expect most of the QT applications to have an X11 interface and to be implemented in C++. With `tagcolledit` it's very easy to have a look at the left panel, select groups of packages that need fixing (the list supports multiple selection), then right click on the selection and choose "*Copy to other panel*". This will add the tags `implemented-in::c++` and `interface::x11` to the selected packages, and they will show up in the other panel as well.

You can do many other tricks with `tagcolledit`: merge or intersect tag sets, add a tag to a group of items or even to all of them (right click on the tag in the filter: you will see the "*Add to all*" and "*Remove from all*" functions).

It is however easy to get carried away and make mistakes, such as adding the `use::gaming` tag to all the `games::toys`, to find out later that packages such as fortune or cowsay cannot really be used to play games.

Once you are finished with `tagcolledit`, you can save the tag database and quit. If you are happy with your changes you will want to integrate them in the main tag database: first create a patch:

```
debtags mkpatch /tmp/tags > /tmp/tags.patch
```

This will create a tag patch with the changes you have made with `tagcolledit`. You can then add your patch to the local tag patch:

```
cat /tmp/tags.patch >> ~/.debtags/patch
```

And see your changes in all the Debtags-aware applications.

Then you can submit it, with either debtags-edit or `debtags submit`.

You can even submit it directly:

```
debtags submit /tmp/tags.patch
```

Automated tagging

While generally there is a need for a smart brain to do good categorization, there are some tagging tasks that can be automated. For example, it's good enough to assume that all packages that depend on `libgtk2.0` will need a `uitoolkit::gtk` tag, while all packages that depend on `libstdc++6` will probably need an `implemented-in::c++` tag.

These ideas have brought to creating autodebtag, which is a Perl script that does all kinds of automated reasoning that I and other people in the debtags-devel list conceived so far. Then it prints out a tag patch that can be evaluated, tested and submitted to the central tag database.

autodebtag has also some experimental code to try and bring categorization data from the debram database into the Debtags world.

Benjamin Mesing is also experimenting with setting up a smart Bayesian engine to infer new tags based on what tags are there now. This has very promising and exciting possibilities: there is a prototype available in the autodebtag subversion repository, and it needs more people to play with it.

Adopting a tag or a facet

If you regularly try out all image manipulation software, if you are a Gnome expert, if you are addicted to role-playing games, you might consider *adopting a tag*.

Adopting a tag means that you take responsibility for checking from time to time that all packages that should have your tag actually have it. It is a very important task, because it introduces a level of reliability and guaranteed accurateness into some parts of the tag database.

For example, Stefano Zacchiroli is an Ocaml guru, and is taking care of the `implemented-in::ocaml` tag. You will notice that all Ocaml applications have the `implemented-in::ocaml` tag, and thanks to Stefano you can rely on that tag to be accurate. We need more people to do what he is doing.

To adopt a tag, just send an e-mail to the debtags-devel list, introduce yourself and tell what you want to adopt. If it's already taken, you can team up with the others and share the brainwork.

Contributing to the vocabulary

The vocabulary is another area needing help. Debian spans a very wide and diverse range of areas, and knowing about all of them is nearly impossible.

There are two ways of having a good vocabulary to cover all of Debian's interests:

1. Wait until I learn about everything in Debian, reach enlightenment and open a Debian monastery in the Alps, or

2. Create a group of people with knowledge on different fields to work on the vocabulary together.

So far, we have been working mainly on 1. This was because there was poor documentation on Debtags and its technical foundations. Now there is this paper, and it's time to switch to the second strategy.

If you feel like you could help (and you do not need to be a Debian Developer, but a Debian user with a lot of experience in some field), you can read the tutorial

at http://debtags.alioth.debian.org/vocabulary.html and join the debtags-devel mailing list.

You can also make local experiments with the vocabulary: the Debtags FAQ has an entry with a small tutorial on how to do it. Someone is trying to do it to provide specific sets of tags for their Custom Debian Distribution, but it's also a good way to experiment with ideas for new facets or tags.

If you are a package maintainer

If you are a package maintainer, please run debtags-edit and have a look at your packages. You are probably the best person in Debian to assign tags such as `implemented-in::*`, `interface::*` and `uitoolkit::*`, and it's going to be very easy for you to do it.

Also remember that you can use more than one tag from the same facet, as your package may contain more than one program, or it may contain programs with multiple behaviours: for example, mutt or aptitude have uses both as `interface::text-mode` and `interface::command-line` programs.

Integrating Debtags in other applications

Now that you know everything about Debtags, you may want to make use of the Debtags data in your applications, or play with Debtags using code instead of existing tools.

If that is the case, there is a selection of libraries and modules available for you.

libdebtags1-dev, python-debtags, libdebtags-perl

If you look inside the code of debtags and debtags-edit, you will notice that most of what they do related to tags is calling functions from the `libdebtags1` or `libtagcoll1` libraries. `libtagcoll1` provides generic manipulation functions for tagged stuff, while `libdebtags1` uses `libtagcoll1` to implement all Debtags functionality.

If you like programming in C++, install `libdebtags1-dev`; from there, you can either look at debtags and debtags-edit source code for examples (debtags is particularly straightforward) and use `/usr/share/doc/libdebtags1-dev/html/index.html` for the doxygen documentation.

The documentation still has missing parts. If you have questions, no matter how silly they could seem, please ask them freely in debtags-devel list: I have promised to myself that I will turn every answer I give into more documentation of `libdebtags1-dev`.

If you instead like coding in Perl or Python rather than C++, you can install either `libdebtags-perl` or `python-debtags`: they are swig-generated bindings to `libdebtags1`.

Again, for whatever documentation or examples that are missing, ask in debtags-devel, and every answer will also contribute to improve the documentation and examples in the packages.

This is an example Python code that plays with Debtags a bit (please teach me how to use iterators properly):

```
import Debtags

# Instantiate the simple Debtags class
```

```
dt = Debtags.DebtagsSimple(0)

# Get some information from the vocabulary
voc = dt.vocabulary()
print "Facets:"
for a in voc.getFacets().getIterable():
   print "%s: %s %s" % (a.name(), a.field("Status"), a.field("Nature"))

# List the accessibility tags
fac = voc.getFacet("accessibility");
print "Tags in 'accessibility::':"
for a in fac.tags().getIterable():
   print a.name()

# Print the tags of ``debtags``
tags = dt.getTags("debtags")
for a in tags.getIterable():
   print a.name()

# List the image editors in Debian
tags = Debtags.TagSet()
tags.insert(voc.getTag("use::editing"))
tags.insert(voc.getTag("media::rasterimage"))
editors = dt.getPackages(tags)
print "Raster image editors:"
for a in editors.getIterable():
   print a.name()
```

This is the same example, written in Perl:

```
#!/usr/bin/perl -w
use strict;
use warnings;
use Debtags;

my $dt = Debtags::DebtagsSimple->new(0);
my $voc = $dt->vocabulary();

print "Facets: ",
  join(', ', map { $_->name() }
    @{$voc->getFacets()->getIterable()}), "\n";

my $fac = $voc->getFacet("accessibility");
print "Tags in 'accessibility::': ",
  join(', ', map { $_->name() }
    @{$fac->tags()->getIterable()}), "\n";

print "Tags for 'debtags': ",
  join(', ', map { $_->fullname() }
    @{$dt->getTags('debtags')->getIterable()}), "\n";
```

```
my $tags = Debtags::TagSet->new();
$tags->insert($voc->getTag('use::editing'));
$tags->insert($voc->getTag('media::rasterimage'));
my $editors = $dt->getPackages($tags);
print "Raster image editors: ",
  join(', ', map { $_->name() } @{$editors->getIterable()} ), "\n";
```

Note

I'm sorry for the need of that `getIterable` function, but swig does not properly wrap C++ `std::set` classes yet. This is hopefully going to change as swig evolves.

libapt-front

libapt-front is the next-generation wrapper for `libapt`. The goal of the project is having a library to base all package managers on, and to have it able to access different sources of metadata besides `libapt`. Including Debtags.

libapt-front is still under heavy development, and Debtags support is still not implemented. However, I have commit access to the libapt-front repository and I've recently been able to take `libdebtags1` where I wanted it to be in order to support libapt-front: expect exciting news from this corner of Alioth!

User interface issues

Once we have the data and the functions to access it, we can improve the way we navigate our package archive.

One area where more research is needed is finding good ways of navigating the package archive using Debtags. As I mentioned before, in front of a list with more than about 7 plus or minus 2 items, our brain goes banana [ZEN] [MILLER]. This sets a first goal for Debtags: to allow people to navigate the package archive in such a way that no list of unrelated choices should have more than 7 plus or minus 2 items.

Do you know of any existing and very good interfaces to navigate such a large archive using faceted categorization? I don't. I know of very good examples on a much smaller scale, but we seem to have a much more large, complex and heterogeneous dataset.

We need mockups and prototypes of interfaces offering an intuitive and efficient way of navigating through tags, keeping in mind those 7+/-2 and EVN requirements.

As you can see running debtags-edit and `tagcolledit`, I still haven't done much progress in this field, and I'm desperately in need of good ideas.

Effective View Navigation

Prof. G. W. Furnas, who usually writes good stuff, made a very nice research (check it out!) and found 4 properties that something should have to be easy to navigate. He called them the properties of *Effective View Navigation* or *EVN properties* [FURNAS]:

1. (EVT1) The number of outgoing links must be "small" compared to the size of the structure
2. (EVT2) The distance between pair of nodes in the structure must be "small" compared to the size of the structure

3. (VN1) Every node must contain some valid information (called residue) on how to reach any given other node
4. (VN2) The information associated to each outgoing link (called outlink-info) must be small.

I did some research on this idea related to categorization [TAGCOLL], and designed an algorithm that is able to present navigational choices of tags in a way that is compatible with the 4 EVN requirements. It is a good start, and recent debtags-edit are making use of it in the filter, to reduce the amount of facets when there are too many to display.

This algorithm has potential, but the results are still not perfect: you can now start the navigation with 19 facets instead of the 32 available, but that's still far beyond 7 plus or minus 2. The main cause of this is that tag data is still incomplete: the "TODO Specials" features of debtags-edit are there to expose those packages that are causing problems.

From my experience, however, I see that algorithm as a point of departure rather than a point of arrival, and more smart navigation ideas are waiting to come to life.

Conclusions

This paper is the first single comprehensive source of information about the Debtags project. It has covered the theoretical foundations, the tools existing at the moment, how to take advantage of Debtags and various ways of getting involved.

Since we would really like to get help, I'll repeat a short summary on how to get involved:

- help with tagging (see Contributing to the categorization)
- adopt a tag or facet (see Adopting a tag or a facet)
- help maintaining the Debtags webpages
- help maintaining the various Debtags Debian packages (but I have to warn that it's not an easy packaging task)
- maintain a language binding (if you know swig)
- help testing the code and writing example programs

If you are interested in helping, or just curious to see what's happening, join the debtags-devel mailing list: that's where people hang out, discussions happens and announcements are posted.

Bibliography

[STECKEL] Mike Steckel, *"Ranganathan for IAs"*, 2002, http://www.boxesandarrows.com/archives/ranganathan_for_ias.php

[ZEN] Enrico Zini, *"Zen and the art of Free Software: know your user, know yourself"*, 2004, http://people.debian.org/~enrico/talks/2004linuxtag/

[MILLER] George A. Miller, *"The magical number seven, plus or minus two: Some limits on our capacity for processing information"*, 1956, Psychological Review n.63 pp. 81-91, http://www.well.com/user/smalin/miller.html

[FURNAS] George W. Furnas, *"Effective View Navigation"*, 1997, Proceedings of CHI'97 pp. 367--374, ACM Conference on Human Factors in Computing Systems, Atlanta, http://www.si.umich.edu/~furnas/Papers/CHI97-EVN.2.pdf

[TAGCOLL] Enrico Zini, *"The tagged collection: an alternative way of organizing a collection of bookmark-like items and its integration with existing web browsers"*, 2001, http://svn.debian.org/wsvn/debtags/tagcoll/trunk/doc/