

Introductory Talk

Programing with Paludis

Danny van Dyk
<kugelfang@gentoo.org>

Gentoo

February 25th, 2007

Outline

Overview

- Libraries and Clients

Paludis Namespace Paradigm

- Code Conventions

- Environments

- Package Database

- Repositories

- Handling Dependency Strings

- Accessing Package Metadata

The Ruby API

Example

Lead-out

Paludis

- ▶ Is a package manager for use with Gentoo and Gentoo-based distributions that allows multiple repositories
- ▶ Is prepared to support more than just ebuild repositories, e.g. CRAN, Ruby gems.
- ▶ Is written in C++ and Bash.
- ▶ Has a modular structure that separates (console/GUI/web) clients from backend libraries.

Outline

Overview

Libraries and Clients

Paludis Namespace Paradigm

Code Conventions

Environments

Package Database

Repositories

Handling Dependency Strings

Accessing Package Metadata

The Ruby API

Example

Lead-out

Libraries

Paludis consists of several libraries. Depending on what your code uses you may need to link against one or more of the below libraries.

`libpaludis` contains common code to interface environments and repositories.

`libpaludisqa` contains QA checks.

`libpaludisargs` contains common code to handle commandline arguments.

`libpaludisdeplist` contains code to build the dependency list for a given set of install targets.

Clients

The following clients are part of Paludis as of version 0.20 and can be used as a deeper reference on how to interact with the libraries.

`paludis` Main client that handles querying, installation, uninstallation and updating of packages.

`adjutrix` Helper client, for both power users and Gentoo developers.

`qualudis` QA utility intended for Gentoo developers that currently implements most of `repoman`'s and some additional checks.

`gtkpaludis` Proof of concept X-Windows client using GTK.

`contrarius` crossdev-alike client.

`inquisitio` A search client for the package database, employing various matching algorithms over various criteria.

Outline

Overview

Libraries and Clients

Paludis Namespace Paradigm

Code Conventions

Environments

Package Database

Repositories

Handling Dependency Strings

Accessing Package Metadata

The Ruby API

Example

Lead-out

Code Conventions

- ▶ All types and classes suitable for clients, effectively the Paludis API, are part of the namespace `paludis`. All future references to its classes will be marked like `Environment`.
- ▶ Class names are always capitalized, e.g. `PackageDatabaseEntry`.
- ▶ Member elements are lowercase with underscores, i.e. `dl_upgrade_as_needed` for an `enum` member or `begin_set_keywords()`
- ▶ `enum` members are prefixed with a unique abbreviation identifying the `enum`'s typename, e.g. in the above example the `dl` identifies it to be one of `DepList`'s options.

Outline

Overview

Libraries and Clients

Paludis Namespace Paradigm

Code Conventions

Environments

Package Database

Repositories

Handling Dependency Strings

Accessing Package Metadata

The Ruby API

Example

Lead-out

Environments

What does Environment mean to Paludis?

- ▶ An **Environment** instance is the first point of contact for any non-trivial Paludis client.
- ▶ It handles configuration setup, non-repository-based metadata and various misc. functions.
- ▶ This *may* include user-defined masks, unmaskers and keywording.

Environment and Descendants

Environment is an abstract base class; its usable descendants are named **FooEnvironment**, where Foo can take the following values:

Default Environment with configuration described by `/etc/paludis/` and `$HOME/.paludis`.

DefaultEnvironment is a singleton; its only instance can be obtained using the static `get_instance()` method.

NoConfig For where user configuration is irrelevant and ignored. Used for clients like `adjutrix` and `qualudis` (via the `libpaludisqa-private` subclass **QAEnvironment**).

Test Only used internally by the unit tests.

Common Members

- ▶ `query_use(UseFlagName &, PackageDatabaseEntry *)`
is used to query the boolean status of a given useflag for a specific package/version tuple. The second argument can be 0 to query useflag's default status.
- ▶ `known_use_expand_names(UseFlagName &, ↪PackageDatabaseEntry *)`
returns a collection of known suffixes to a given USE_EXPAND-prefix. As above, the second argument can be 0 to query globally.
- ▶ `accept_{keyword,eapi,license(...)}`
returns true if the used `Environment` will accept packages featuring the given KEYWORD, EAPI or LICENSE.

Outline

Overview

Libraries and Clients

Paludis Namespace Paradigm

Code Conventions

Environments

Package Database

Repositories

Handling Dependency Strings

Accessing Package Metadata

The Ruby API

Example

Lead-out

PackageDatabase

At the heart of each subclass of `Environment` lies `PackageDatabase`. This class supports

- ▶ iterating over all repositories via `begin_repositories()` and `end_repositories()`.
- ▶ fetching repositories via their names:
`fetch_repository(RepositoryName &)`
- ▶ rule-based querying for packages and more, see next slide.

Query

- ▶ All queries are executed using

```
query(Query &, QueryOrder),
```

returning a collection of `PackageDatabaseEntries` based on the rule of class `Query`.

- ▶ `QueryOrder` states how the returned `PackageDatabaseEntryCollection` shall be ordered; either sorted by versions or grouped by slots.
- ▶ All descendants of `Query` are part of the namespace `paludis::query`.

Query Classes

- ▶ As of paludis 0.20.0, these classes are implemented:
 - ▶ `Matches`, constructable from a `PackageDepSpec`.
 - ▶ `Package`, constructable from a `QualifiedPackageName`.
 - ▶ `NotMasked`
 - ▶ `RepositoryHasInstalledInterface`
 - ▶ `RepositoryHasInstallableInterface`
 - ▶ `RepositoryHasUninstallableInterface`
 - ▶ `InstalledAtRoot`, constructable from a `FSEntry`.
- ▶ Query rules can be merged using the operator `&`
- ▶ `query` does optimisation behind the scenes. This is why merged instances of `Query` should be used in all cases rather than doing multiple queries and intersecting their results manually.
- ▶ There is room for future extension of this list.

Query Classes - Examples

- ▶ Find all packages databases entries that match <app-admin/eselect-2:

```
query(Matches(PackageDepSpec("<app-admin/eselect-2")),  
↪ qo_grouped_by_slot)
```

- ▶ Find all packages installed to /my/chroot/:

```
query(InstalledAtRoot(FSEntry("/my/chroot")),  
↪ qo_order_by_versions)
```

- ▶ Find all installable packages in all repositories which are not masked:

```
query(RepositoryHasInstallableInterface() &  
NotMasked(),  
↪ qo_order_by_versions)
```

Outline

Overview

Libraries and Clients

Paludis Namespace Paradigm

Code Conventions

Environments

Package Database

Repositories

Handling Dependency Strings

Accessing Package Metadata

The Ruby API

Example

Lead-out

Repositories

In contrast to Portage, Paludis does **not** support overlays directly. It does support multiple full-fledged repositories though, as well as repositories that properly define their use of foreign `eClass`-directories and/or `profile`-directories.

Paludis repositories work like this:

- ▶ Each repository is an instance of a `Repository` subclass
- ▶ Each `Repository` subclass implements support for one particular package format

Repositories (Cont.)

- ▶ `FooRepository`'s code is not part of `libpaludis` but rather of `libpaludisfoorepository`.
- ▶ Repository instances are owned and can be fetched from `Environment::package_database` by various methods.
(See `PackageDatabase`)
- ▶ In general, repositories have varying capabilities which are handled by Paludis using repository interface classes.
- ▶ Most repository interfaces are only used by higher level interfaces (`Environment`, `Tasks`)

Common Methods

All subclasses of `Repository` must implement the following common methods

- ▶ `has_category_named(CategoryNamePart &)`

Returns true if a category of given name exists in this repository.

- ▶ `has_package_named(QualifiedPackageName) &`

Return true if a package of given name exists in this repository.

- ▶ `has_version(QualifiedPackageName &, VersionSpec &)`

Return true if the given package/version tuple exists in the repository.

- ▶ `category_names()`

Return a collection of all category names in this repository.

Common Methods (Cont.)

- ▶ `version_specs(QualifiedPackageName &)`

Returns a collection of `VersionSpec` for the given package.

- ▶ `has_version(QualifiedPackageName &, VersionSpec &)`

Returns true if the specified package exists, false otherwise.

- ▶ `version_metadata(QualifiedPackageName &, VersionSpec &)`

Returns an object of class `VersionMetadata` for the given package; discussed later on.

Interfaces

Any interface is a class of name `RepositoryFooInterface`. The following values are some of the more interesting ones Foo can take as of Paludis 0.20.

- ▶ Sets, provides

```
sets_list(),
```

which returns a collection of all sets in this repository, as well as

```
package_set(SetName &),
```

which returns the `DepSpecs` for the specified package set.

- ▶ Contents, provides

```
contents(QualifiedPackageName &, VersionSpec &),
```

which returns an object that lists all filesystem contents that the given package provides

Interfaces

- ▶ Installed, provides

```
installed_time(QualifiedPackageName &, VersionSpec &),
```

which returns the time that the given package was lastly installed.

- ▶ News, provides

```
update_news(),
```

a method to update the repository's list of unread news items. (see GLEP 42)

- ▶ EnvironmentVariable, provides a method to obtain an environmental variable's contents for a specified entry of the PackageDatabase.

Outline

Overview

Libraries and Clients

Paludis Namespace Paradigm

Code Conventions

Environments

Package Database

Repositories

Handling Dependency Strings

Accessing Package Metadata

The Ruby API

Example

Lead-out

Common Patterns

- ▶ Paludis uses common code to handle dependency string of types `SRC_URI`, `xDEPENDS` and `LICENSE`.
- ▶ Dependency strings can be cut down to reoccurring structures that are labeled *atoms*.
- ▶ Each atom is encapsulated by an instance class `FooDepSpec`. Meaningful values of `Foo` will be explained later on.
- ▶ All atom classes share the abstract base class `DepSpec`

Package Dependency Specs

Dependency atoms like `>=app-admin/eselect-1.0.5:0::gentoo` are encapsulated by the class `PackageDepSpec`.

- ▶ Constructable from `std::string`.
- ▶ The described package **must** be specified using a `QualifiedPackageName`; that means names of the form `category/package-name`. `QPN` is an instantiation of `Validated`.
- ▶ Provided methods are

```
package()  
version_requirements_ptr()  
slot_ptr()  
repository_ptr()  
use_requirements_ptr()
```

Composite Dependency Specs

- ▶ `CompositeDepSpec` and descendants provide a pair of functions,
`begin()`
`end()`
that allow iterating over all constituent `DepSpecs` (children).

There are 3 important subclasses of `CompositeDepSpec`

- ▶ `AllDepSpec`
- ▶ `UseDepSpec`
- ▶ `AnyDepSpec`

All-of and Any-of Dependency Spec

- ▶ Dependency strings like `(Spec1 Spec2)` are encapsulated by the class `AllDepSpec`.
- ▶ Strings like `|| (Spec1 Spec2)` are encapsulated by `AnyDepSpec`.
- ▶ There is no further difference between these two classes and their base class `CompositeDepSpec` beyond the class name. It is up to the program/programmer to discriminate between them.

Useflag Driven Dependency Specs

The class `UseDepSpec` encapsulates dependency strings of form `[!]useflag? (Spec1 Spec2)`.

Additionally to `CompositeDepSpec`'s members it features these methods:

- ▶ `flag()`
which returns the `UseFlagName` of the atom at hand,
- ▶ `inverse()`
which returns true if the meaning of the `UseFlagName` is inverted using a prepended exclamation mark, and false otherwise.

Working with Specs

- ▶ The need for manual parsing of `FOODEPEND` strings should not arise. You can access the atoms via the `foo_depend()` methods in `VersionMetadata` and the use of its parser function (will be discussed later on).
- ▶ There are GoF-based visitor methods for dealing with this kind of spec hierachies. Discussing them is beyond the cope of this talk, but classes like `LicenseDisplay` (part of `libpaludisoutput`) should give a fair example.
- ▶ If there is further interest then join `#paludis` for support.

Outline

Overview

Libraries and Clients

Paludis Namespace Paradigm

Code Conventions

Environments

Package Database

Repositories

Handling Dependency Strings

Accessing Package Metadata

The Ruby API

Example

Lead-out

VersionMetadata

- ▶ [VersionMetadata](#) is the source for all package related metadata stored in the package database. It is obtainable through the repository that holds a given [PackageDatabaseEntry](#).

- ▶ As the version metadata is highly dependent on the repository at hand, [VersionMetadata](#) has a capabilities driven interface system - similar to repository interfaces.

VersionMetadata (Cont.)

`VersionMetadata` has some basic members, that means members which do not belong to any interface. These are listed below

- ▶ `slot`
- ▶ `description`
- ▶ `homepage`
- ▶ `eapi`

Apart from `slot` which is of type `SlotName`, all of the above members are of type `std::string`

VersionMetadata - Interfaces

`VersionMetadata` interface classes are labeled `VersionMetadataFooInterface` where `Foo` can take one of the following values:

- ▶ Ebuild, needed by the "normal" PortageRepository
- ▶ Ebin, needed by PortageRepository's upcoming support for binary packages.
- ▶ CRAN, needed by CRANRepository
- ▶ Deps
- ▶ Origins
- ▶ Virtual
- ▶ License

For any `FooInterface` there is a `foo_interface` method that returns either `this` or `0`. Discussion of some of the above interfaces follows.

VersionMetadata - Ebuild

The following ebuild metadata is given as `std::string` as part of the `VersionMetadataEbuildInterface`:

- ▶ `provide_string`
- ▶ `src_uri`
- ▶ `restrict_string`
- ▶ `keywords`
- ▶ `eclass_keywords`
- ▶ `iuse`
- ▶ `inherited`

Except for `provide_string` and `src_uri` all of these members can be tokenised on whitespaces. If you want to parse the former two then talk to us over in `#paludis`.

VersionMetadata - Deps

`VersionMetadataDepsInterface` holds the following metadata:

- ▶ `build_depend`
- ▶ `run_depend`
- ▶ `post_depend`
- ▶ `suggest_depend`

All these members are of type `std::tr1::shared_ptr<const CompositeDepSpec >`. Parsing was automatically done using the method `parser`. This way one always obtains a tree of `DepSpecs`, no matter what the dependency syntax may be.

Ruby vs. C++

- ▶ Ruby API corresponds to the C++ API apart from Ruby-isms.
- ▶ The Ruby interface aims to be natural to Ruby programmers and behave like a Ruby library, rather than being an exact translation.
- ▶ Example for this is the lack of STL iterators; instead arrays are returned or the usage of blocks is suggested.
- ▶ Another example is the use of predicate methods where appropriate.
- ▶ Not every class or method is currently available; support for more classes and methods can be added as needed. Some things (for example, defining new Environment subclasses) will likely never be available through this interface.
- ▶ There are applications that may never be implemented using the Ruby API, especially applications that make use of Paludis tasks.

Diggin' the VDB

Let's now write a new Paludis client that searches all installed packages' `DEPEND` strings for `UseDepSpecs` which contain the `gtk-useflag` and then print all the `PackageDepSpecs` to `stdout`.

We start by setting the proper `LogLevel` and get a collection of all installed packages:

```
Log::get_instance()->set_log_level(ll_silent);
```

```
std::tr1::shared_ptr<  
    const PackageDatabaseEntryCollection>  
packages(  
    DefaultEnvironment::get_instance()  
    ->package_database()->query(  
        query::RepositoryHasInstalledInterface(),  
        qo_order_by_version));
```

Now that we have a `PackageDatabaseEntry` for each installed Package let's iterate over them and check if the package's `VersionMetadata` actually has a `VersionMetadataDepsInterface`.

```
for (paludis::PackageDatabaseEntryCollection::Iterator
     p(packages->begin()), p_end(packages->end()) ;
     p != p_end ; ++p)
{
    std::tr1::shared_ptr<const Repository> repo(
        DefaultEnvironment::get_instance()->package_database()
        ->fetch_repository(
            p->repository));

    std::tr1::shared_ptr<const VersionMetadata> vm(
        repo->version_metadata(p->name, p->version));

    VersionMetadataDepsInterface * deps = vm->deps_interface;
    if (! deps)
        continue;
}
```

Diggin' the VDB

Now, print out the package name and hand over the traversal of `DepSpecs` to an instance of a visitor based class `GTKPrinter`.

```
std::cout << "*" << stringify(*p) << std::endl;
```

```
GTKPrinter printer;  
deps->build_depend()->accept(&printer);  
std::cout << std::endl;
```

GTKPrinter is a subclass of `DepSpecVisitorTypes::ConstVisitor`; that means it needs to provide the `visit` methods for all of the leaf-like subclasses of `DepSpec`. We will only populate some of them:

```
void visit(const paludis::UseDepSpec * spec)
{
    if (spec->flag() == paludis::UseFlagName("gtk"))
        _inside_gtk_spec = true;
    std::for_each(spec->begin(), spec->end(), paludis::accept_
        _inside_gtk_spec = false;
}
```

We will copy just the line

```
std::for_each(spec->begin(), spec->end(), paludis::accept_vis
```

for all the other functions handling subclasses of `CompositeDepSpec`.

Lastly, we make the handler for `PackageDepSpec` print out the spec to stdout if `_inside_use_spec` is true:

```
if (_inside_gtk_spec)
    cout << " " << stringify(*spec) << endl;
```

Diggin' the VDB

The complete example can be downloaded from

<http://dev.gentoo.org/~kugelfang/gtk-cond-deps.cxx>

The file should be compiled using

```
g++ -Wall -lpaludis -lpaludisdefaultenvironment  
↪ gtk-codn-deps.cxx -I /usr/include/paludis/
```

You will obviously need to have Paludis 0.20 installed and configured.

Recommended Literature

- ▶ Programming with Paludis (0.20)
<http://paludis.pioto.org/programmingwithpaludis.html>
- ▶ Paludis C++ Core API Documentation (0.20)
<http://paludis.pioto.org/doxygen/html>
- ▶ Paludis Ruby API Documentation (0.20)
<http://paludis.pioto.org/ruby>
- ▶ Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley
- ▶ Proper C++ documentation, preferably TCppPL and EffCpp.

Thanks

Thanks for valuable input and contribution on this talk go to

- ▶ Ciaran McCreesh
- ▶ Stephen P. Bennett
- ▶ Richard Brown, for insight in the Ruby API
- ▶ Tobias Scherbaum, for convincing me to give an example
- ▶ Marius Mauch, for the idea behind the VDB example