JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

PortBrowser

A user interface for the BSD ports system

Handed in by
Böhm Igor

Supervised by
Dipl.-Ing. Albrecht Wöß

Institute for System Software
Johannes Kepler University Linz

Linz, April 2, 2005

Abstract

BSD operating systems offer a very powerful and flexible framework for installing third party packages, called the BSD ports system. This framework provides a complete environment coupled with a rich set of commands, which is able to build almost any kind of software from source code and package it. Thus any piece of software which has been incorporated into the BSD ports system, can easily be installed, deleted or upgraded.

The PortBrowser, a graphical front end for the BSD ports system, is a light weight tool which offers a simple, portable, and secure environment for the most frequent tasks provided by the ports system. It is a very useful tool for novice as well as experienced users, since it allows for easy browsing through the BSD ports tree, and offers search facilities to a certain extent.

This paper covers the basics and some details of various flavors of BSD ports tree implementations, followed by a description of the PortBrowser project. Implementation details as well as the GUI design of the PortBrowser are covered and described in an easy to understand manner.

Table of Contents

Chapter 1

Introduction

1.1 Structure of this Paper

In order to understand how the `PortBrowser` works and some of the design decisions which have been made during the development process, it is necessary to understand the underlying `BSD`[1] ports system.

But even before the introduction of the `BSD` ports system starts, a coarse overview of various other state of the art `UNIX` software management tools is given in chapter 1. This overview helps to understand the strengths and weaknesses of the `BSD` ports system compared to other solutions.

In the subsequent chapter the basics and concepts of the `BSD` ports system is described, followed by the description of the most important commands which are necessary for every days work with the ports system. Most of the descriptions apply to all `BSD` flavors, but also the advantages and disadvantages of specific ports tree extensions which are only present in certain `BSD` flavors, are covered.

Chapter 3 gives an introduction to the graphical user interface of the `PortBrowser` coupled with operating guidelines and followed by an overview of some implementation details like the class hierarchy, used data structures, safe execution of privileged commands and graphical programming.

The last chapter summarizes the results of this thesis and the practical part of this project and gives a short outlook for various planned improvements for future `PortBrowser` releases.

[1]`BSD` - Berkeley Software Distribution is the name of the `UNIX` derivative distributed by the University of California, Berkeley starting in the 1970s.

1.2 UNIX Software Management Tools

In the early days of UNIX, installing a program from source code was often a very tedious and error prone act. Every systems administrator had to understand the platform a program had been written for, and its differences from their platform, before they had a hope of porting a chunk of code [4]. Sometimes it was even necessary to rewrite parts of a program from scratch because of basic assumptions which didn't hold on a specific system.

Thus a variety of package management tools evolved on many UNIX like operating systems, which make the task of installing and or removing third party software a lot more transparent for the user. Basically the idea behind these tools is that the user should not have to worry about things like dependency requirements, build and configure tools and many more things which may be necessary during the installation and or removal process of software packages.

In order to be able to compare the BSD ports system with other approaches, it is necessary to take a look at popular package management tools like *RPM*, which stands for RedHat Package Manager, or Debians package maintenance system *dpkg*. A more detailed comparison explaining conceptual differences between the BSD ports system and other approaches will be given in the subsequent chapter.

The *RPM* Package Manager and Debians *dpkg* are both powerful command line driven package management systems capable of installing, uninstalling, verifying, querying, and updating computer software packages. Each software package consists of an archive of files along with information about the package like its version, a description, and the like. There is also a related API, permitting advanced developers to bypass 'shelling out' to a command line, and to manage such transactions from within a native coding language. Both *RPM* and *dpkg* have been designed for binary package management.

There are also a variety of graphical front ends for *RPM* like YaST[2], and Synaptic for *dpkg*. The primary goal of these graphical front ends is to simplify the task of selecting, searching, updating, installing and deleting software packages for the user. The design of the PortBrowser has been driven by similar goals but with a special emphasis on simplicity, usability and portability amongst various BSD flavors.

[2]YaST - Yet Another Setup Toolkit is an operating system setup and configuration tool that is featured in the SUSE Linux distribution.

Chapter 2

BSD Ports System

The BSD ports collection offers a simple solution for users and administrators, which enables them to transparently install various third party software applications. Such third party software applications are referred to as "ports", where a "port" is simply a set of instructions usually combined with patches for compiling a piece of software.

2.1 BSD Ports Collection Basics

Basically all that is needed in order to be able to install a port is to find the directory in which all the port specific information is located, and execute the command `make`[1] `install`. This command triggers many actions like downloading the source code from the network, confirming the checksums, uncompressing, patching, building and installing the chosen port.

Usually the most difficult part for many users is to find a port which provides a certain functionality. Since all the information is kept in plain text files, classical tools like `grep`[2] and `find`[3] can be used to search for certain ports. In order to speed up and to simplify the search process, the most important parts of every port are aggregated into one `INDEX` file.

[1]`make` - is a computer program that automates the compilation of programs whose files are dependent on each other.
[2]`grep` - file pattern searcher.
[3]`find` - walks a file hierarchy locating files based on some user-specified criteria.

2.2 BSD Ports Infrastructure

The ports tree master `Makefile` fragment `bsd.port.mk`, contains all the standard routines which is used by the ports tree. Since the various `BSD` flavors have adapted their ports system collections to their specific needs, the ports tree layout is not the same on every xBSD system. `OpenBSD` will serve as an example to describe the most important parts of the `BSD` ports infrastructure.

A port itself is fairly simple. It contains a `Makefile` which holds all the necessary information that is needed in order to build the downloaded source code, like the version of `make` which should be used, which compiler is needed, or dependency requirements which need to be satisfied before the port can be installed. The `distinfo` file contains `distfile` checksums, where the term `distfile` simply denotes a file of source code. The `DESCR` file contains a longer description of the port, usually including an URL for further information. The `PLIST` file contains a list of all the files a port installs and which are independent of whether the architecture supports shared libraries or not. If a port provides shared libraries, the file `PFRAG.shared` exists and lists the extra files which need to be installed for architectures which support them. The `PFRAG.noshared` file lists only files being additionally installed on architectures without shared library support.

2.2.1 Targets

In order to get a better overview of what happens when one goes to a port directory and types `make install`, it is necessary to take a look at the targets which control individual ports:

- `fetch`: Fetch all of the files needed to build a port.
- `checksum`: Verify that the fetched `distfile` matches the one the port was tested against.
- `depends`: Install any dependencies of the current port.
- `extract`: Expand the `distfile` into a working directory.
- `patch`: Apply any patches that are necessary for the port.
- `configure`: Configure the port.
- `build`: Build the port.

- `fake`: Pretend to install the port under a subdirectory of the work directory where the work directory is the directory where all port activity occurs. Apart from the actual port, the work directory may also hold all kinds of cookies that checkpoint the port's build.
- `package`: Create a binary package from the fake installation. This binary package can be used to install the port on several machines using `pkg_add` (see next section for more information about the package tools).
- `install`: Install the resulting package.

There are also other targets which can be called but they do not run during the normal install process and usually deal with cleaning up, reinstalling or printing extended information about ports like build or run dependencies which need to be satisfied.

The lock infrastructure which has been implemented in the `BSD` ports tree allows for concurrent builds of several ports at the same time. The locking protocol follows a big-lock model where each top level target in a port directory will require the corresponding lock, complete its job and then release its lock. This means that while one concurrent install process is in the `fetch` target stage, another concurrent install process can be in the `build` stage and so on. Thus this feature is really handy for bulk package building because it speeds up the process by utilizing the available resources much more efficiently.

2.2.2 Flavors and Multi packages

The `OpenBSD` ports tree comes with two orthogonal mechanisms [1] called Flavors and Multi packages. Because of these mechanisms the user can select specific options provided by a given port. A good example of the Flavor mechanism would be the text editor port of `vim`[4]. By looking at the `FLAVORS` variable in the `Makefile` which is located in the ports respective directory, one will see that there are 9 different Flavors which can be specified. Thus the user has the option to compile `vim` with `gtk`[5] support or without `X11` support and so on. To avoid confusion with other packages or flavors, the package name will be extended with a dash-separated list of the selected flavors.

The Multi packages mechanism is used when a smaller package can be broken down into several smaller components, referred to as subpackages. Again

[4] `vim` - vi clone with many additional features.
[5] `gtk+` - Gimp Toolkit. A multi-platform toolkit for creating graphical user interfaces.

a good example for the Multi packages mechanism would be the gtk+2 port. By looking at the MULTI_PACKAGES variable defined in the Makefile, it becomes obvious that the gtk+2 port is broken down into two packages where the default installs the multi-platform graphical toolkit, and the defined Multipackage -docs creates and installs the documentation for gtk+2. If a port is "subpackaged", in addition to the main package, each subpackage will have a corresponding description in the the DESCR-subpackage file.

Port installation example using Flavors and Multi packages

```
$ cd /usr/ports/editors/vim
$ env FLAVOR="gtk2 huge" SUBPACKAGE="-lang" make install
```

Together, Flavors and Multi packages account for OpenBSDs ports tree being somewhat smaller than the other BSDs, as they allow one single port directory to build lots of distinct packages.

FreeBSD offers a similar but less flexible mechanism through the OPTIONS directive which can be defined in a ports Makefile. This mechanism also provides hooks that the port author can use to control which configuration of the corresponding port should be built. The FreeBSD approach is similar to OpenBSDs Flavors, even though it must be mentioned that the OpenBSD project offers a more fine grained and cleaner implementation which makes it possible to really exploit all the options certain ports are able to provide in a very compact and optimal way.

2.2.3 Port Flavors and Combinatorics

This section covers a problem which occurs when there are many ports with various flavors in the ports tree. It basically explains why it is not feasible at the moment to display complete information about a port and its flavors in the PortBrowser. Ways to work around this problem will be discussed in the last chapter.

Even though there is much power and flexibility available with the Flavor and Multi packages concept, some drawbacks must be taken into account when trying to aggregate ports tree information into one large INDEX file. Suppose that for each flavored port, there should be an entry in the INDEX file which lists the port without flavors, and there should also be entries for all possible combinations of flavors for that specific port. This would indeed be a very nice feature, since it would be possible to quickly skim through all flavor combinations a port provides, without having to look in the ports Makefile for that information. Let's use the vim port again as an example to demonstrate the problem:

- **Question:** The `vim` port has 9 flavors. How many combinations of flavors are possible?

There is a very similar problem in the theory of sets, which arises when one tries to find the amount of distinct subsets of an n-element set. Actually, the problem of finding out how many distinct subsets of an n-element set exist, is exactly the same problem as with our combinations of flavors. Thus if we find a generation formula for the amount of distinct subsets within an n-element set, we have found a solution for the original problem. So there are 2^n distinct subsets of an n-element set, including the empty set as well as the set itself. The big problem is that this grows exponentially, thus imposing a limitation on the amount of ports which should be aggregated in the `INDEX` file:

- **Answer:** $2^9 = 512$ entries in the aggregated `INDEX` file would be needed in order to register all possible combinations of flavors which the `vim` port provides.

So if there are a couple of ports with several flavors, the representation of those ports with all of their possible flavor combinations would quickly exceed the total amount of available unflavored ports in the whole ports tree.

Now there is also the case where certain combinations of flavors are not feasible. An example for this case would be the combination of the `no_x11` flavor with the `gtk` flavor of the `vim` port, since it is not possible to build vim with `gtk` support without `X11` libraries. So even though this case reduces the amount of combinations of flavors for the `vim` port, it increases the complexity on the algorithm which tries to find all combinations of valid flavors.

The `OpenBSD` ports system implements a solution for this problem, where the common case has been made fast, and the rare case involves additional work, but still offers the same functionality. If there exists a common and frequently used combination of flavors for a port, this combination is defined as the default by the port maintainer, and also gets registered in the aggregated `INDEX` file. Thus there are at most two versions of a port registered in an `INDEX` file, the unflavored version, and the version including the default flavor of a port. For all the other combinations the user is advised to look into the corresponding ports `Makefile`.

2.3 Package Tools

Packages are the binary equivalent of ports, which have been described in the previous sections. Usually a small collection of pre-compiled packages is available for most common architectures. A compiled port becomes a package that can be registered into the system using the `pkg_add` command.

2.3.1 `pkg_add`

The `pkg_add` command is used to install packages. Such packages contain pre-compiled applications from the ports tree and usually can be found on ftp mirrors or official `BSD` distribution CDs. Each package name may be specified as a file name, which normally consists of the package name itself plus the ".tgz" suffix, or an URL pointing to FTP, HTTP or SCP locations. This implies that installing a package is as easy as typing `pkg_add pkgname.tgz`. It is usually very useful to also set the `PKG_PATH` environment variable since it is evaluated by `pkg_add`. Thus `pkg_add` can resolve and fetch all dependent packages from the correct place and one does not always need to specify the complete path to the package repository:

`pkg_add` installation example

```
$ export PKG_PATH=ftp://ftp.openbsd.org/pub/OpenBSD/3.6/packages/i386/
$ pkg_add vim-6.3.13-no_x11.tgz
```

In order to understand how `pkg_add` works, it is necessary to look at the steps which it walks through. First of all, the "packing information" of a package is extracted into a temporary directory so that `pkg_add` can perform the following checks:

- check if the package is already recorded as installed
- check whether the package conflicts with an already installed package
- check if the architecture for which the package was compiled matches
- check if all dependencies have been resolved
- check for collisions with installed file names

After those checks have been performed and no conflicts have been found, install scripts which may have been hooked into the "packing information" are executed. Then the package contents, except the "packing information", are extracted to their final locations. After the installation is complete, a copy of the packing list, deinstall script, description and display files are copied into `/var/db/pkg/<pkg-name>` for subsequent possible use by `pkg_delete`.

2.3.2 `pkg_info`

The `pkg_info` utility is used for displaying information on software packages, which may still be packed up or already installed on a system. There are many command line parameters which make it possible to adjust the amount of information `pkg_info` should display. The following example outputs only the one line comment field for a package:

`pkg_info` example

```
$ pkg_info -c pb-browser
Information for pb-browser-0.3
Comment:
Graphical Ports System front end
```

As mentioned before, there are many more parameters which can be passed to `pkg_info`, but it is not the purpose of this document to list all of the options and the interested reader can find a description of all the parameters in the respective `OpenBSD` manual pages.

2.3.3 `pkg_delete`

The `pkg_delete` command is used for deleting previously installed software package distributions. In order to specify the package which should be deleted it is enough to provide the package name itself, or as a filename which consists of the package name followed by the ".tgz" suffix, or as a full pathname like `/var/db/pkg/pkgname`, so that shell wild cards can be used.

Before `pkg_delete` can remove a package, it has to check whether the package is required by other installed packages not mentioned in the list of packages to remove. If that is the case, `pkg_delete` will list those dependent packages and refuse to remove the package.

`pkg_delete` example

```
$ pkg_delete /var/db/pkg/gnome-session-2.6.2
```

`pkg_delete` also checksums files before it deletes them so if a file has been edited by a user and thus the checksum changed, it will normally notify the user and not remove the changed file.

2.4 Upgrading Ports and Packages

Being able to upgrade ports and packages on a production system is an important requirement for many system administrators and users. Even though the various `BSD` flavors implement this functionality, it is interesting to look at the different approaches they have taken.

2.4.1 `FreeBSD` approach

The `FreeBSD` project provides the `portupgrade` tool in order to untangle the port upgrade mess [5] which occurs when one has many packages installed.

The tools included in `portupgrade` implement some new features into the `FreeBSD` ports system. First there are `pkgdb` and `portdb`, which build databases to index the contents of `/var/db/pkg`, the place where all installed packages are recorded, and the ports tree itself. This is done in order to accelerate searching and manipulating this information. The `pkgdb` and `portdb` tools even rewrite files located in `/var/db/pkg` to maintain consistency. Finally, there are wrappers for the various `pkg_*` tools which handle both, rewriting the plain text records located in `/var/db/pkg` and the indexed databases. The following are some of the most important wrapper tools which `portupgrade` offers:

- `portinstall`: helps to install new ports in a handy way.
- `portcvsweb`: a tool to instantly browse a history via `CVSweb`.
- `portversion`: a tool to compare installed packages with the ones in the ports tree.
- `portsclean`: a tool to clean ports and packages garbage like unreferenced distfiles, working directories and old shared libraries.
- `pkg_deinstall`: a package deinstaller with wild cards and dependency recursion support.
- `pkg_fetch`: a remote package fetcher.

Even though with all of these tools the ports and packages upgrade is possible, it is not an elegant and simple to use approach. The main problem is that the `FreeBSD` ports system has not been designed with an upgrade feature in mind. Thus almost all of the base `pkg_*` tools need to be replaced and enhanced by wrapper tools, which means duplicating code and making it hard for the user to grasp what he or she should actually use in order to get a functional ports and packages system. Another problem with `portupgrade` is, that it is a port itself, and thus is not as thoroughly maintained as the standard `pkg_*` tools which are in the base system.

2.4.2 `OpenBSD` approach

The `OpenBSD` project is in the process of enhancing their ports framework to provide the upgrade functionality in the base system for the 3.7 release. Since one of the really complicated parts of upgrading software is to also upgrade packages with all of their dependencies to other packages and to account for shared libraries which many ports use, some new variables have been introduced in order to specify which shared libraries a port needs. This means that packages which are built from ports now, include all the information needed for the upgrade process.

A new option `-r` has been introduced to `pkg_add` to allow the replacement of existing packages. The code tries to take every precaution to make sure the update can proceed before removing the old package and adding the new one, and it also handles shared libraries correctly.

`pkg_add` is only able to upgrade and replace single packages, thus `pkg_update` will be introduced to the `pkg_*` toolchain. `pkg_update` will be able to perform global updates of the system:

- `pkg_update` will be able to sort through individual replacement operations to find an "optimal" order for replacing packages,
- `pkg_update` will tell the user that he or she actually needs to replace some packages and install some new ones in some rare cases and helps them to do so.
- `pkg_update` will also have some look up capabilities in order to figure out what old package it replaces.

The big advantage of the `OpenBSD` approach is that the upgrade functionality is implemented in the base ports system and not provided as an add-on package. Thus the code goes through a constant and thorough audit process and there is no need for wrapper tools which only add up to the overall complexity of the ports system.

2.4.3 `NetBSD` approach

"Port" is the term used by the `FreeBSD` and `OpenBSD` community for what the `NetBSD` community calls a package. In `NetBSD` terminology, the term "port" refers to an architecture like `i386`, `macppc` or `sparc64` et cetera. The `NetBSD` package framework which is also known by the term `pkgsrc`, includes the upgrade functionality of binary packages in the base system. The `pkg_add` command has an update switch which basically does the following:

- If the package that's being installed is already installed, either in the same or a different version, an update is performed. If the command line switch is specified twice, then any dependent packages that are too old will also be updated to fulfill the dependency.

The package framework also supports the `update` and `replace` targets. The `update` target updates the installation of the current packages and all dependent packages which are installed on the system. The `replace` target updates the installation of the current package but does not replace dependent packages.

2.5 BSD Ports System Advantages

By looking at the various concepts and approaches present in the `BSD` ports system, its strengths and advantages when compared to other package management tools like `RPM` or `dpkg`, become obvious.

The `BSD` ports system was originally designed around the concept of building software from source, with the ability to make and install binary packages as an afterthought [2], while many other packaging systems like *RPM* or *dpkg* and such were designed around the concept of installing a binary package, with building from source as an afterthought.

So the big advantage of building from source is flexibility. While a binary package "just installs", ports have been designed to cover the full range of bits and pieces of installing software like encoding, tracking and installing dependencies, packaging, installing and deinstalling software, applying platform specific changes, compile time configuration options and much more.

Again, a *RPM* is just a binary package and in order to be able to automatically resolve and install dependencies higher level tools are needed and since it's binary, one has to deal with library versioning conflicts, missing compile time options, or any other of the limitations which occur when one does not build the package on his or her own system.

Chapter 3

Port Browser

3.1 Graphical User Interface

The GUI of the `PortBrowser` has been designed to be as simple as possible. The tree view on the left hand side represents the ports tree infrastructure. Ports are listed based on the category they belong to. There are some cases where more than one category is specified for a port, but since a port can only reside in one directory in the actual ports tree, it will only be assigned to one category in the the tree view.

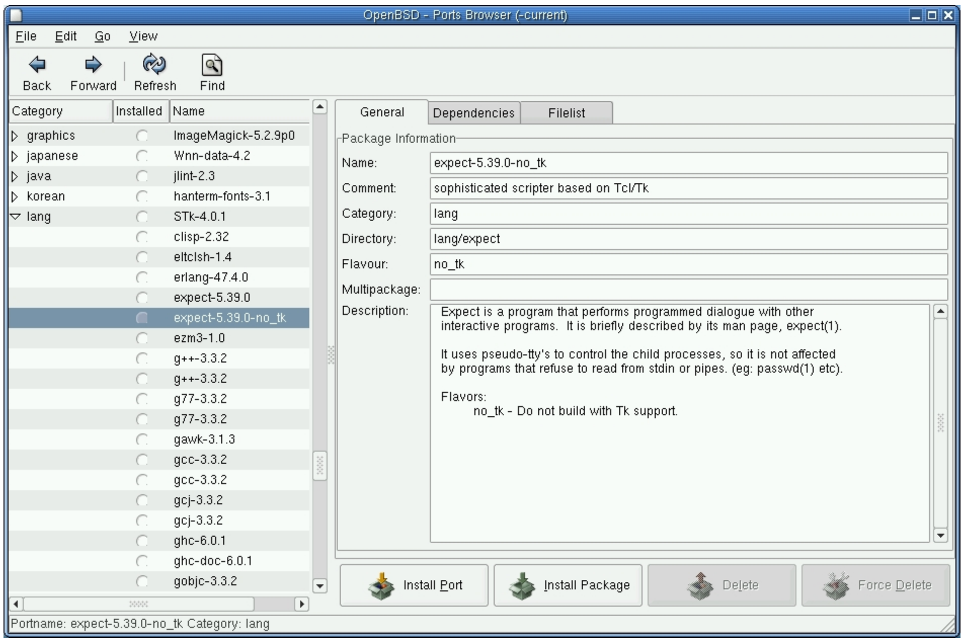
Because of this representation of the ports tree, it is very easy to navigate around, thus turning the process of searching, installing or deleting of various ports in an easy task. This has actually been one of the main goals of this project - providing a simple front end for frequent ports operations. In order to get a better understanding of the functionality which is offered by the `PortBrowser`, the next sections will describe the most important graphical interfaces and their usage.

3.1.1 Port Description View

The `Port Description View`, as illustrated in figure 3.1, is based on a mixture of information found in the `pkg/DESCR` file and the `INDEX` file, which contains aggregated information about all ports. Basically the following port specific values will be displayed:

- Port name combined with the version number.
- Comment used for the port.

- Category the port belongs to. If a port may be part of multiple categories, only the first category will be displayed.
- Relative directory location in the ports tree where the port resides in.
- If a port is *Flavored* or *Multipackaged* the appropriate information is displayed. There may be more variants of Flavors and Multi packages present in the ports respective `Makefile`, but only those which are present in the aggregated `INDEX` file will be displayed.
- The extended description which is found in `pkg/DESCR` is also displayed.



Category	Installed	Name
graphics		ImageMagick-5.2.9p0
japanese		Wnn-data-4.2
java		jlint-2.3
korean		hanterm-fonts-3.1
lang		STk-4.0.1
		clisp-2.32
		eltclsh-1.4
		erlang-47.4.0
		expect-5.39.0
		expect-5.39.0-no_tk
		ezm3-1.0
		g++-3.3.2
		g++-3.3.2
		g77-3.3.2
		g77-3.3.2
		gawk-3.1.3
		gcc-3.3.2
		gcc-3.3.2
		gcj-3.3.2
		gcj-3.3.2
		ghc-6.0.1
		ghc-doc-6.0.1
		gobjc-3.3.2

Name:	expect-5.39.0-no_tk
Comment:	sophisticated scripter based on Tcl/Tk
Category:	lang
Directory:	lang/expect
Flavour:	no_tk
Multipackage:	

Description:

```
Expect is a program that performs programmed dialogue with other
interactive programs.  It is briefly described by its man page, expect(1).

It uses pseudo-tty's to control the child processes, so it is not affected
by programs that refuse to read from stdin or pipes. (eg: passwd(1) etc).

Flavors:
        no_tk - Do not build with Tk support.
```

Portname: expect-5.39.0-no_tk Category: lang

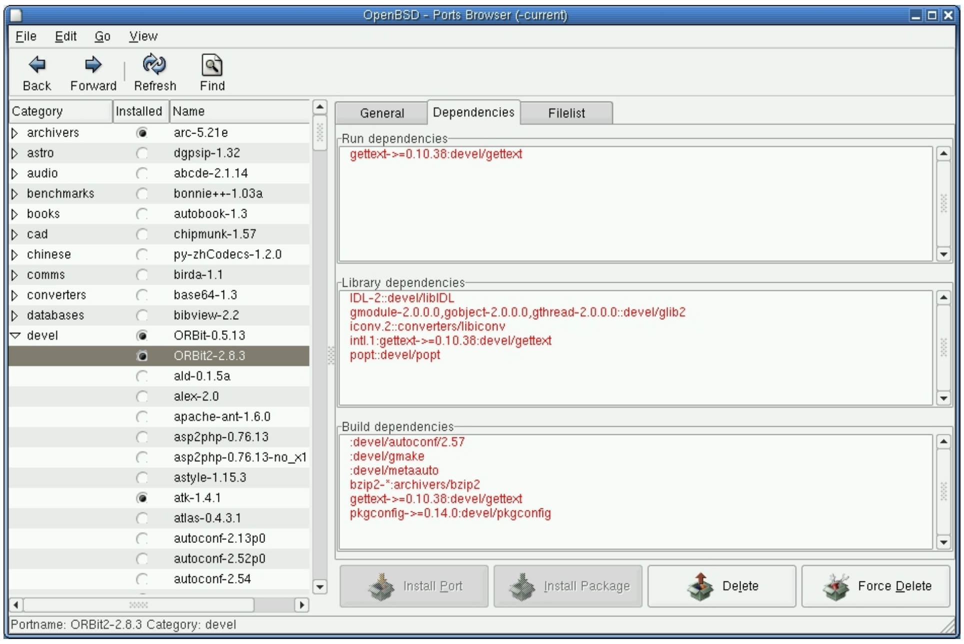
Figure 3.1: Port Browser - Port description view.

3.1.2 Port Dependency View

The `Port Dependency View`, as illustrated in figure 3.2, displays the possible port dependencies which are extracted from the `INDEX` file. The values are split into the following dependency categories:

- `Run Dependencies` include a specification of ports this port needs installed to be functional.

- `Build Dependencies` include a list of other ports the current port needs to build correctly.
- `Library Dependencies` specify libraries this port depends upon.



Category	Installed	Name
archivers	●	arc-5.21e
astro	○	dgpsip-1.32
audio	○	abcde-2.1.14
benchmarks	○	bonnie++-1.03a
books	○	autobook-1.3
cad	○	chipmunk-1.57
chinese	○	py-zhCodecs-1.2.0
comms	○	birda-1.1
converters	○	base64-1.3
databases	○	bibview-2.2
devel	●	ORBit-0.5.13
	●	ORBit2-2.8.3
	○	ald-0.1.5a
	○	alex-2.0
	○	apache-ant-1.6.0
	○	asp2php-0.76.13
	○	asp2php-0.76.13-no_x1
	○	astyle-1.15.3
	●	atk-1.4.1
	○	atlas-0.4.3.1
	○	autoconf-2.13p0
	○	autoconf-2.52p0
	○	autoconf-2.54

Run dependencies

```
gettext->=0.10.38:devel/gettext
```

Library dependencies

```
IDL-2::devel/libIDL
gmodule-2.0.0.0,gobject-2.0.0.0,gthread-2.0.0.0::devel/glib2
iconv.2::converters/libiconv
intl.1:gettext->=0.10.38:devel/gettext
popt::devel/popt
```

Build dependencies

```
:devel/autoconf/2.57
:devel/gmake
:devel/metaauto
bzip2-*:archivers/bzip2
gettext->=0.10.38:devel/gettext
pkgconfig->=0.14.0:devel/pkgconfig
```

Portname: ORBit2-2.8.3 Category: devel

Figure 3.2: Port Browser - Port dependency view.

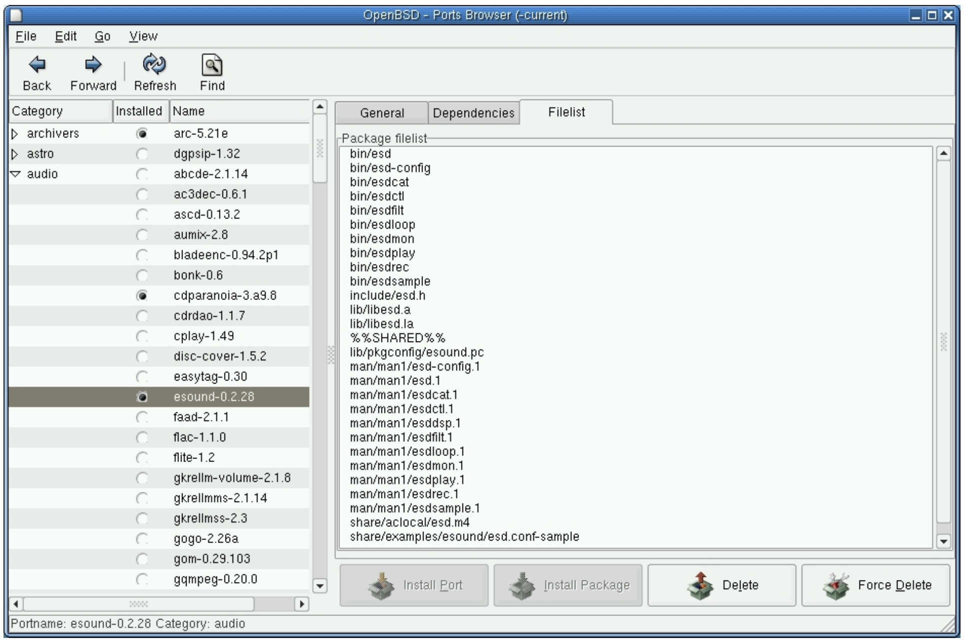
3.1.3 Port Packing List View

The `Port Packing List View`, as illustrated in figure 3.3, displays all files which are listed in `pkg/PLIST` and belong to a specific port like executables, libraries, manual pages, header files, documentation et cetera. Note that shared libraries are not yet displayed in this particular view, and thus the file `pkg/PFRAG.shared` is not evaluated by the `PortBrowser`, although this will be implemented in the next upcoming releases.

3.1.4 Port Search Window

The `PortBrowser` offers some basic search functionality. The search window, as illustrated in figure 3.4, offers the user the possibility to search for phrases in the following fields:

- Package Name Field



Category	Installed	Name
▷ archivers	●	arc-5.21e
▷ astro	○	dgpsip-1.32
▽ audio	○	abcde-2.1.14
	○	ac3dec-0.6.1
	○	ascd-0.13.2
	○	aumix-2.8
	○	bladeenc-0.94.2p1
	○	bonk-0.6
	●	cdparanoia-3.a9.8
	○	cdrdao-1.1.7
	○	cplay-1.49
	○	disc-cover-1.5.2
	○	easytag-0.30
	●	esound-0.2.28
	○	faad-2.1.1
	○	flac-1.1.0
	○	flite-1.2
	○	gkrellm-volume-2.1.8
	○	gkrellmms-2.1.14
	○	gkrellmss-2.3
	○	gogo-2.26a
	○	gom-0.29.103
	○	gqmpeg-0.20.0

Package filelist

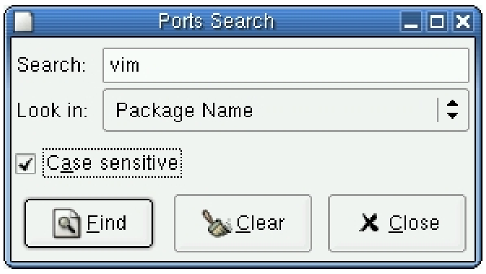
```
bin/esd
bin/esd-config
bin/esdcat
bin/esdctl
bin/esdfilt
bin/esdloop
bin/esdmon
bin/esdplay
bin/esdrec
bin/esdsample
include/esd.h
lib/libesd.a
lib/libesd.la
%%SHARED%%
lib/pkgconfig/esound.pc
man/man1/esd-config.1
man/man1/esd.1
man/man1/esdcat.1
man/man1/esdctl.1
man/man1/esddsp.1
man/man1/esdfilt.1
man/man1/esdloop.1
man/man1/esdmon.1
man/man1/esdplay.1
man/man1/esdrec.1
man/man1/esdsample.1
share/aclocal/esd.m4
share/examples/esound/esd.conf-sample
```

Portname: esound-0.2.28 Category: audio

Figure 3.3: Port Browser - Ports packing list.

- Comment Field
- Description Field
- Library-, Build- and Run-Dependency Field
- All of the above

If the phrase has been found, the corresponding entry in the ports tree view is automatically selected. Support for regular expressions has not been added yet, but is planned for the future. It is also not possible to search for files or shared libraries in specific ports, but this will be supported in upcoming versions.



Search: vim

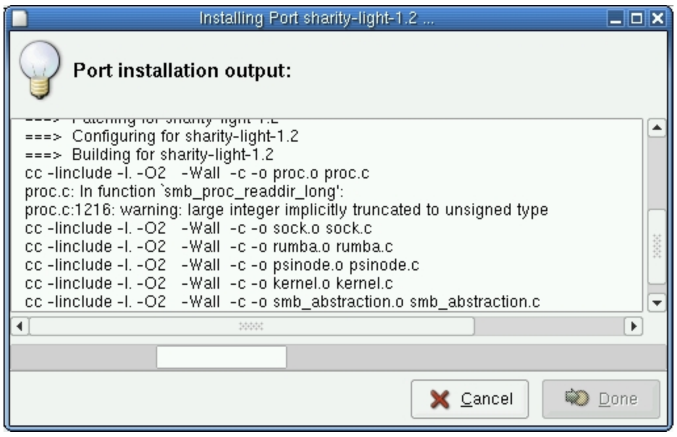
Look in: Package Name

✓ Case sensitive

Figure 3.4: Port Browser - Ports search window.

3.1.5 Port Operation Progress Window

Whenever the user starts to install or delete ports or packages, a window (see figure 3.5) which tracks the chosen process will pop up. The output of each port or package operation is obtained from a pseudo `tty` where all the necessary commands are executed. It is also possible to cancel the chosen operation which will basically kill the install or delete process and may result in an inconsistent or unclean state of the ports tree. The cancel process can be compared to pressing `CTRL+C` in a terminal while the install or delete process of a port or package is executing. After a port or package operation completes, the ports tree view is updated depending on whether a port has been installed or deleted.


Installing Port sharity-light-1.2 ...
Port installation output:
Cancel
Done

```
===>  Configuring for sharity-light-1.2
===>  Building for sharity-light-1.2
cc -Iinclude -I. -O2   -Wall  -c -o proc.o proc.c
proc.c: In function `smb_proc_readdir_long':
proc.c:1216: warning: large integer implicitly truncated to unsigned type
cc -Iinclude -I. -O2   -Wall  -c -o sock.o sock.c
cc -Iinclude -I. -O2   -Wall  -c -o rumba.o rumba.c
cc -Iinclude -I. -O2   -Wall  -c -o psinode.o psinode.c
cc -Iinclude -I. -O2   -Wall  -c -o kernel.o kernel.c
cc -Iinclude -I. -O2   -Wall  -c -o smb_abstraction.o smb_abstraction.c
```

Figure 3.5: Port Browser - Ports operation progress window.

3.2 Implementation

This section gives an overview of some implementation details and describes the rationale behind certain decisions. First of all, the reason for choosing `C` as programming language should be explained. Even though there are many high level languages like `JAVA`, `C++` or `C#`, the choice has been made in favor of the `C` programming language. Amongst many reasons for this decision, the most important ones were portability, the availability of a free compiler toolchain, and the availability of a multi-platform toolkit for creating graphical user interfaces.

Another important premise was to use all the functionality which is provided by the `xBSD` base system, and to only depend on a very minimal subset of add on software which can be installed through the ports system itself. Thus the `PortBrowser` only depends on one third party software package, namely `GTK+`. All the other tools like a `C` compiler, `lex`[1], `yacc`[2] and the `Make` utility are available with the `BSD` base system.

3.2.1 Class Hierarchy

The first prototypes of the `PortBrowser` used `C` structures to internally represent ports with all of the information that belongs to them. Even though this approach worked considerably well, another approach which uses the `GObject` object oriented framework provided by the `glib`, has been chosen. The `glib` is a general purpose utility library which includes support routines for `C` such as lists, trees, hashes, memory allocation, and many other things.

`GObject`, and its lower level type system `GType`, are used by `GTK`+ and most `GNOME`[3] libraries to provide:

- Object oriented `C` based APIs [3].
- Automatic transparent API bindings to other compiled or interpreted languages.

The reason why `GObject` was chosen for the `PortBrowser` is mainly because it provides an object oriented `C` based API and comes with many useful functions which can be inherited and used from base objects such as `GObject` itself. If not defined otherwise, `GObject` is the base class of every newly created object in the `GObject` object oriented framework.

[1]`lex` - fast lexical analyzer generator.
[2]`yacc` - An LALR parser generator.
[3]`GNOME` - GNU Network Object Model Environment.

Mainly functions for creating, unreferencing, setting and getting properties of an object are widely used in the PortBrowser, and replace custom allocation and de-allocation functions as well as self written access functions to structure variables. Another powerful feature which GObject provides is information hiding, which is very hard to do with C structures.

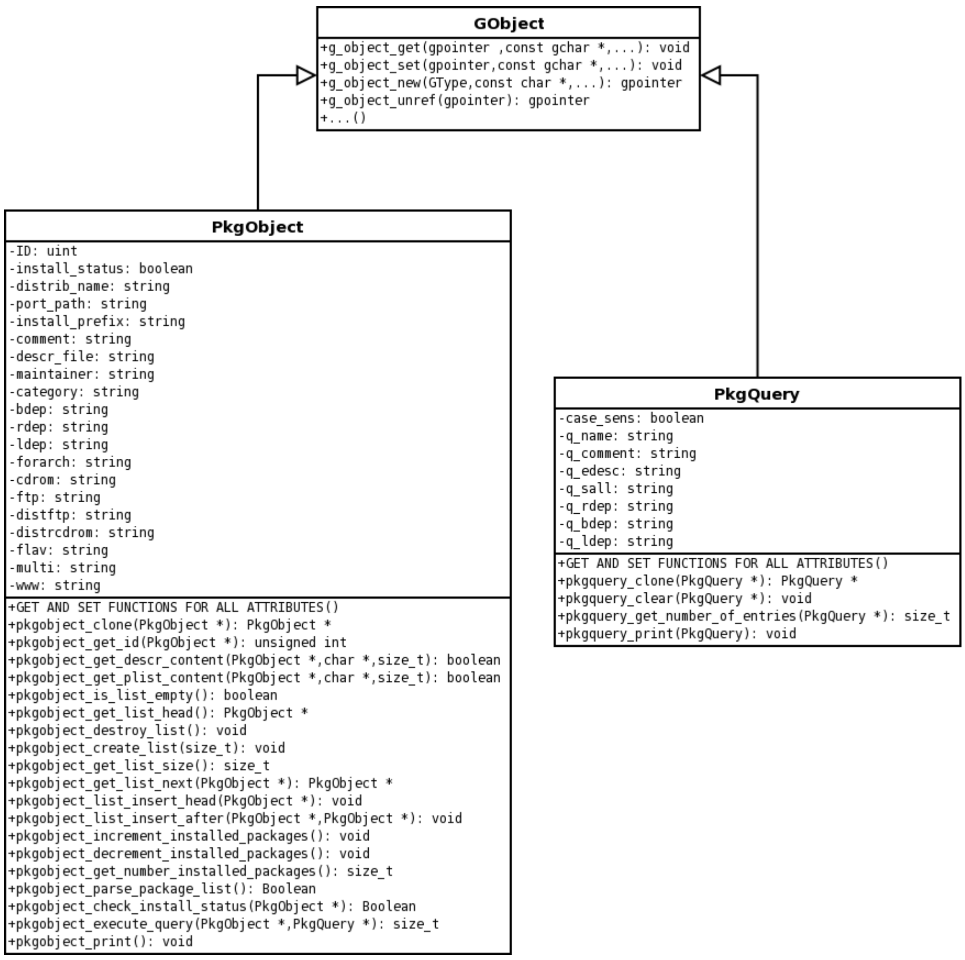

GObject
+g_object_get(gpointer ,const gchar *,...): void
+g_object_set(gpointer,const gchar *,...): void
+g_object_new(GType,const char *,...): gpointer
+g_object_unref(gpointer): gpointer
+...()
PkgObject
-ID: uint
-install_status: boolean
-distrib_name: string
-port_path: string
-install_prefix: string
-comment: string
-descr_file: string
-maintainer: string
-category: string
-bdep: string
-rdep: string
-ldep: string
-forarch: string
-cdrom: string
-ftp: string
-distftp: string
-distrcdrom: string
-flav: string
-multi: string
-www: string
+GET AND SET FUNCTIONS FOR ALL ATTRIBUTES()
+pkgobject_clone(PkgObject *): PkgObject *
+pkgobject_get_id(PkgObject *): unsigned int
+pkgobject_get_descr_content(PkgObject *,char *,size_t): boolean
+pkgobject_get_plist_content(PkgObject *,char *,size_t): boolean
+pkgobject_is_list_empty(): boolean
+pkgobject_get_list_head(): PkgObject *
+pkgobject_destroy_list(): void
+pkgobject_create_list(size_t): void
+pkgobject_get_list_size(): size_t
+pkgobject_get_list_next(PkgObject *): PkgObject *
+pkgobject_list_insert_head(PkgObject *): void
+pkgobject_list_insert_after(PkgObject *,PkgObject *): void
+pkgobject_increment_installed_packages(): void
+pkgobject_decrement_installed_packages(): void
+pkgobject_get_number_installed_packages(): size_t
+pkgobject_parse_package_list(): Boolean
+pkgobject_check_install_status(PkgObject *): Boolean
+pkgobject_execute_query(PkgObject *,PkgQuery *): size_t
+pkgobject_print(): void
PkgQuery
-case_sens: boolean
-q_name: string
-q_comment: string
-q_edesc: string
-q_sall: string
-q_rdep: string
-q_bdep: string
-q_ldep: string
+GET AND SET FUNCTIONS FOR ALL ATTRIBUTES()
+pkgquery_clone(PkgQuery *): PkgQuery *
+pkgquery_clear(PkgQuery *): void
+pkgquery_get_number_of_entries(PkgQuery *): size_t
+pkgquery_print(PkgQuery): void

Figure 3.6: PortBrowser - Object Hierarchy.

The PortBrowser implements two classes, PkgObject and PkgQuery, which are derived from GObject. Figure 3.6 shows a class diagram displaying the most important functions provided by those two classes. Almost all of the specific names of *getter* and *setter* functions have been left out, since that would only cause confusion and bloat the diagram. Even though class diagrams already give many clues regarding the purpose of classes, some more information about the usage and purpose of these two classes will be provided in the following sections.

`PkgQuery`

The `PkgQuery` class holds all the information which is necessary in order to execute search queries. It provides *getter*, *setter* and *clear* functions in order to fill, extract, or clear the corresponding search phrases. There are also functions which provide information about how many search phrases a `PkgQuery` object includes.

`PkgObject`

The `PkgObject` class holds all the information about a port which gets extracted from the aggregated `INDEX` file. It also provides functions for managing double linked lists of `PkgObjects`. A search function which takes a `PkgQuery` object and a `PkgObject` as a parameter is also implemented. It returns the amount of successful hits in the `PkgObject` which meet the specified query criteria in the `PkgQuery` object.

In order to account for installed packages, there are functions which determine, the install status and modify a counter which keeps up with the amount of installed packages on the system.

Since some information about a port like the `PLIST`, or the extended description is not available in the aggregated `INDEX` file, `PkgObject` provides functions which transparently extract the necessary information out of the corresponding files.

3.2.2 Data Structures

All `PkgObjects` are linked together into a double linked list which gets sorted based on port categories and port names. Instead of using the `glib` double linked list structures and functions, a set of standard macros which are provided by the `BSD` base system are used. Since the `C` preprocessor expands these macros into fast pointer operations, they are much faster than and almost as easy to use as their `glib` counterparts.

The data structure behind the tree view is a `GtkTreeStore`. A `GtkTreeStore` is a tree like data structure that can be used with the `GtkTreeView`[4]. This data structure only holds information about the unique id of each port, the category a port falls into, the port name and the install status of a port.

Since the directory tree organization of the ports system naturally resembles a tree like data structure, it is planned to replace the double linked list which holds all `PkgObjects` with a tree like data structure.

[4]`GtkTreeStore` - A widget for displaying both trees and lists.

3.2.3 Parsing of Aggregated Ports Tree Information

At the current date, `OpenBSD` offers approximately 3.000 ports and `FreeBSD` offers over 10.000 ports. Thus it is not feasible to scan the ports directory tree for this large amount of ports in order to be able to display all of them.

Because of the vast amount of available ports, the most important information about each port is aggregated in an `INDEX` file, which is usually located in `/usr/ports/INDEX`. Basically the `INDEX` file consists of entries separated by a "|", where one line represents the aggregated information of one port. Parsing this information as efficient as possible is important, since the user shouldn't have to wait very long for the `PortBrowser` application to start up.

The first prototypes of the `PortBrowser` came with a hand written parser for the `INDEX` file. The first version of the hand written parser used many high level functions provided by the `glib` library. All it did was to create simple structures of parsed ports, and display them in a tree view. The implementation was very basic but hard to read and modify.

The second version of the hand written parser only used `libc` functions, and thus was independent of any third party framework. This version of the parser was very fast, since a lot of the functionality was implemented by using pointer arithmetic's. Although this version of the parser did not deal with all the special cases which arise on various `BSD` flavors, and only generated simple data structures from the information it parsed, the drawbacks from this approach were quite obvious. 350 lines of unreadable code which was running very fast though. This was again not satisfying since unreadable code is hard to port, debug, clean up, and generally something that should be avoided.

The current state of the implementation uses `lex` and `yacc` in order to parse the information in the `INDEX` file. Since the grammar specification for `yacc` is very simple, only 24 lines of code are needed for it, it makes the implementation of `BSD` specific port tree extensions like `OpenBSD`s Flavors and Multi packages concept easier to handle. Also changes to the `INDEX` file format can be adapted quickly and easily with this approach.

3.2.4 Safe Execution of Privileged Commands

Since many of the operations which the `PortBrowser` provides need elevated privileges, a concept of safely executing privileged commands must be established. The easiest approach would be to execute the `PortBrowser` application with super user privileges, but in general it is not good to have applications which require `root` privileges throughout their life cycle for various reasons:

- Application bugs may cause fatal consequences since the process has super user privileges.
- Everyone who would like to use the application would need to know the root password.
- There is no way to provide fine grained access controls.
- ...

The goal is to limit the risk of those extra privileges being compromised in the event of an attack [7]. Thus the `PortBrowser` will refuse to start if it is executed with super user privileges. Since the problem of safely executing privileged commands is not `PortBrowser` specific, there are already various applications available for `UNIX` which accomplish this task. Probably the best known applications are `su` and `sudo`, which are the ones utilized by `PortBrowser`:

- The `su` - *substitute user* - command is used to assume the login shell of another user without logging out.
- The `sudo` - *superuser do* - command allows a permitted user to execute a command as the superuser or another user, as specified in a configuration file. If the invoking user is root or if the target user is the same as the invoking user, no password is required. Otherwise, `sudo` requires that users authenticate themselves with a password by default. Once a user has been authenticated, a time stamp is updated and the user may then use sudo without a password for a short period of time.

The very first versions of `PortBrowser` only supported the `su` command. But since it is necessary to know the root password in order to execute commands with `root` privileges through `su`, and there is no real fine grained way to define privileges, support for `sudo` has been added and can be activated by the `-s` command line switch. `Sudo` definitely is the best solution since it allows a system administrator to give certain users or groups of users the ability to run some, or all commands as root or another user while logging the commands and arguments.

It should be quite obvious now which tools the `PortBrowser` utilizes in order to execute commands, but the way this is implemented is still unclear. The next section deals with this topic by showing the basic concepts which are necessary in order to achieve our goal of safely executing privileged commands.

Execution of `su` and `sudo` in Pseudo Terminals

Before we take a look at how things are implemented in the `PortBrowser`, a general overview of `pseudo terminals` is given. The term `pseudo terminal` implies that it looks like a terminal to an application program, but it's not a real terminal [6]. Figure 3.7 shows a typical arrangement of the processes involved when a pseudo terminal is being used. The key points in this figure are the following:

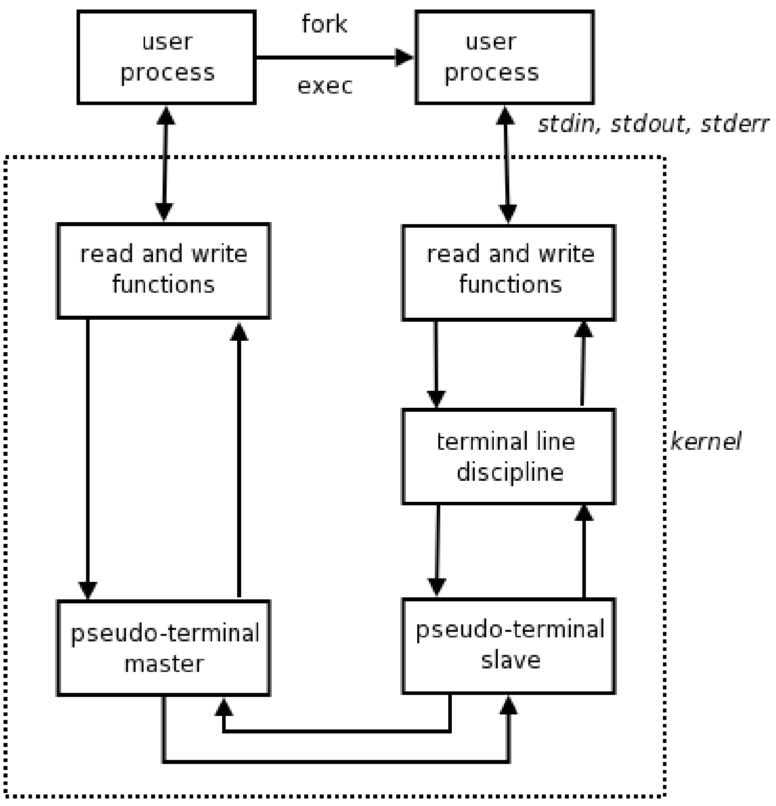


Figure 3.7: Typical arrangement of processes using a pseudo terminal.

1. Normally a process opens the pseudo terminal master and then calls `fork` in order to create a new process (child process). The child establishes a new session, opens the corresponding pseudo terminal slave, duplicates it to be standard input, standard output, and standard error, and then calls `exec` to execute a file. The pseudo terminal slave becomes the controlling terminal for the child process.

2. It appears to the user process above the slave that its standard input, standard output and standard error are a terminal device. It can issue almost all terminal `I/O` functions on these descriptors.

3. Anything written to the master appears as input to the slave and vice versa. Indeed all the input to the slave comes from the user process above the pseudo terminal master. This looks like a stream pipe but with the terminal line discipline module above the slave there exist additional capabilities over a plain pipe.

Since the concept of a pseudo terminal should be obvious by now, it is possible to take a look at how things are implemented in the `PortBrowser`. In order to be able to execute a command in a pseudo-tty, the function `forkpty()`, which combines the three functions `openpty()`[5], `fork()` and `login_tty()`[6] to create a new process operating in a pseudo-tty, must be called. So after `forkpty()` has been called successfully, the child process can execute a privileged command and the parent process set's up the appropriate file descriptors so that it can read the output of the child process and provide input for the child process through the pseudo-terminal master. In our case the only input the child process may ask for is a pass phrase, since we only execute `su` or `sudo` within a pseudo-tty. The actual commands which manipulate ports and packages are passed as parameters to `sudo` and `su`.

In order to monitor the execution of ports and packages commands, it is only necessary to read from the master `stdin` file descriptor which is connected to the slave `stdout` file descriptor. In order to provide transparent access to the output of the slave file descriptor, a thread safe non blocking function that provides access to a buffer which stores a certain amount of output produced by the slave, has been implemented. This function is mainly used by the operation progress window (see figure 3.5) to track the output of an executed operation and to display it to the user.

This concept of executing commands is really flexible since it doesn't depend on the underlying package management system. As long as that system provides commands which enable us to manipulate packages and ports, it is possible to port the `PortBrowser` to such a system. All that needs to be changed in order to get the `PortBrowser` to interact with a different package management system would be the commands which are passed as parameters to `sudo` and `su`. This again drastically decreases porting efforts to other ports or package management systems.

[5]`openpty()` - This function finds an available pseudo-tty and returns file descriptors for the master and slave.

[6]`login_tty()` - This function prepares for a login on a specified tty which may be a real tty device, or the slave of a pseudo-tty.

3.2.5 Graphical Programming with `GTK+`

`GTK+` is a powerful and platform independent toolkit intended for creating graphical user interfaces. It has initially been developed as a `widget`[7] set for the `GIMP`[8] and has grown extensively ever since. Today it is being deployed by a large number of applications such as the `GNOME` desktop project.

The following components which are plugged and used together in a very modular fashion, make up what is referred to as the `GTK+` toolkit:

- `GTK+` - Provides a complete and object oriented hierarchy of widgets.
- `GDK` - `GTK+` Drawing Kit. Thin layer between `GTK+` and the windowing system (e.g. `X11`) which handles the actual graphics drawing and event handling.
- `Pango` - Powerful library for rendering internationalized texts.
- `GdkPixbuf` - Image loading library.
- `ATK` - Accessibility Toolkit library providing a set of interfaces for accessibility. By supporting the ATK interfaces, an application or toolkit can be used with such tools as screen readers, magnifiers, and alternative input devices.
- `GObject` - Library and framework which provides object-oriented programming for the C programming language.
- `Glib` - General purpose utility library which includes support routines for `C` such as lists, trees, hashes, memory allocation, and many other things.

This thesis will only deal with the `GTK+` component, explaining its basics, major concepts and advantages. But first of all, it is necessary to understand the concept of *widgets* in `GTK+`.

`Widget` Concept

Widgets are like containers and can contain widgets on their part again. Thus widgets are responsible for the graphical layout of a program [8]. The most obvious container is definitely a *window*, which contains all widgets of a program. There are of course other containers like *Box*- or *Table* widgets,

[7]`widget` - Graphical component, that a computer user interacts with, such as a window or a text box.

[8]`GIMP` - GNU Image Manipulation Program.

and again those widgets are able to include other widgets which are then arranged together. So by boxing and packing widgets into each other, the application or dialog window gets its specific shape. The process of putting a widget into another widget is called *packing*, and a widget which has been packed into another widget is called its *child* widget. See figure 3.8 which displays a small part of the `GTK+` class hierarchy.

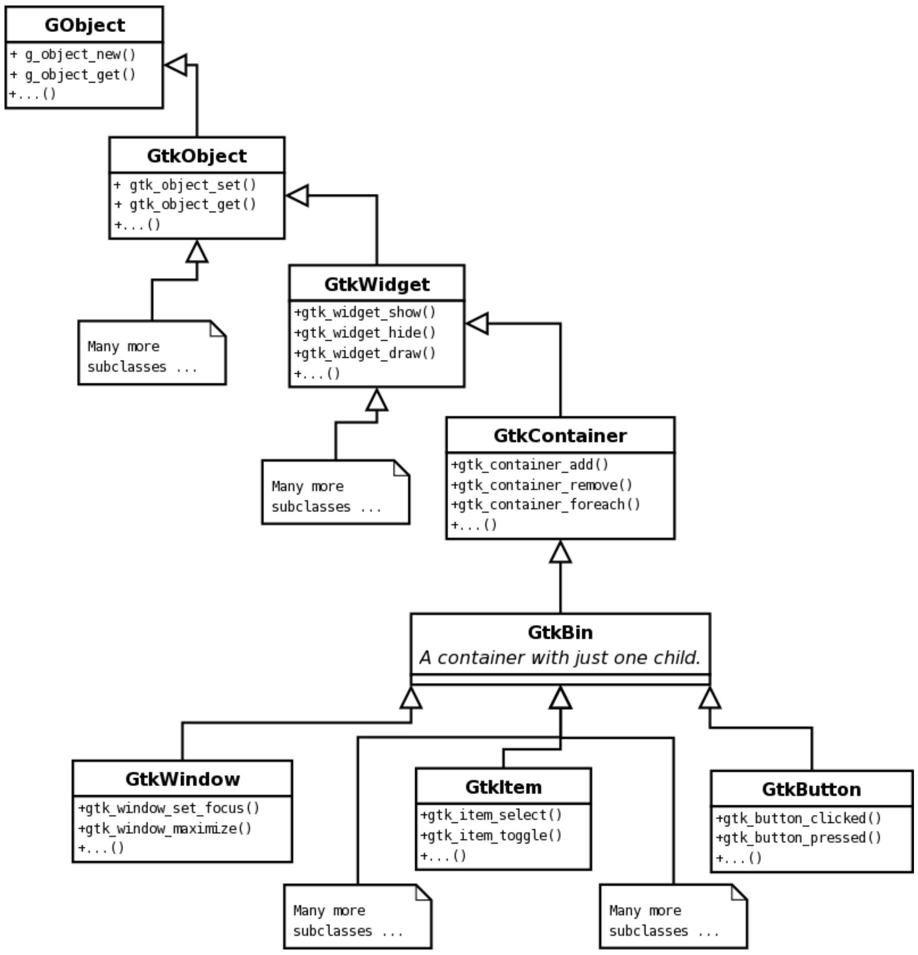


Figure 3.8: Example of a small part of the `GTK+` class hierarchy. Only very few functions are displayed to avoid unnecessary complexity in the class diagram.

`GtkWidget`, the base class of all visible control elements, offers a large amount of methods, properties and signals. Functions for creating, destroying, activating, deactivating, hiding or showing a widget are only a few, out of a vast variety. It is not the purpose of this thesis to provide a detailed reference of `GTK+` functions, thus the interested reader is advised to visit `http://www.gtk.org` for further and more detailed documentation of the

GTK+ API.

Finally an overview of most commonly used widgets which are available with GTK+ should demonstrate the completeness of the toolkit:

- `Windows` - top level and dialog.
- `Containers` - vertical and horizontal.
- `Buttons`, `Labels`, `ComboBoxes`, `Menus`.
- `ScrollBars`, `ProgressBars`
- `TreeViews`, `ListBoxes`
- `Panes` - horizontal and vertical.
- `Notebooks`, `Tabs`
- `TextViews`, `Tables`
- ...

GTK+ Advantages

Even though some advantages of GTK+ have already been covered in previous sections, it is necessary to complete them and present the rationale behind the choice of GTK+ as the GUI toolkit for the `PortBrowser`:

- GTK+ provides a complete widget set.
- The UI is very scalable and fast since it is implemented in C.
- There is support for themes, thus the look of feel of GTK+ applications can easily be changed at runtime.
- Stock widgets are easily extensible with custom widgets.
- Full internationalization is available.
- Used in a variety of products ranging from embedded applications to a complete desktop environment like GNOME.
- Clean design following the MVC[9] paradigm.

[9]MVC - Model View Controller. A software architecture that separates an application's data model, user interface, and control logic into three distinct components.

While evaluating the appropriate GUI toolkit for this project, many of the named factors played an important role. In spite of all these facts, the main reason why `GTK+` has been chosen for this project was the clean and easy to use API which resembles a sophisticated design.

Chapter 4

Summary

This thesis presented various aspects and many details about the PortBrowser project. The BSD ports system, which resembles the basic foundation for the PortBrowser, as well as various BSD specific extensions and limitations have been covered.

An introduction to the simple graphical user interface coupled with screenshots demonstrates which information is displayed and how the interface should be used.

Various implementation details are covered in order to describe the most important structures and concepts which have been incorporated into the PortBrowser. Many design decisions have been presented and backed up by supportive arguments in order to make them more transparent.

The practical part of this project, which consisted of shaping an initial idea into a fully-fledged project followed by the actual implementation, was accompanied with various positive but also some negative experiences and aspects.

A negative aspect most certainly was the fact that no real software engineering process took place at the beginning. There certainly were many ideas, but there was no clear specification of what should be implemented and what not. Even though it would have been hard to provide a halfway accurate specification, because at the beginning of the project I was not really acquainted with the programming language, the BSD ports internals and GTK+, it would have been enough to at least specify what the PortBrowser should accomplish as far as functionality is concerned. Since there has been no accurate specification, the implementation procedure was a little bit unsystematic.

The eagerness to write secure code was most certainly very positive for the project, since it requires special care and a lot of research in order to find out about secure state of the art implementation techniques for certain problems. Producing prototypes during the initial phases of the project also resulted in a better overall outcome of the project, because it was possible to learn more about the programming language and the toolkits which have been used during the prototyping phase.

Shortly after the first testing releases of the `PortBrowser`, a stable version has been produced which fixed a couple of small bugs encountered on non `i386` architectures. Afterwards some porting efforts have been made in order to support the `FreeBSD` ports system. These porting efforts have been quite successful. Unfortunately there was not enough time to implement some `FreeBSD` specific features which would have made the `PortBrowser` a really handy tool for `FreeBSD` too.

After posting to the appropriate lists and informing the public[1] about this project, the `PortBrowser` got imported into the `OpenBSD` and `FreeBSD` ports system and is now available with upcoming releases of those two `BSD` flavors.

4.1 Future Work

There are many improvement ideas for this project which will hopefully be implemented soon. The goal of the ongoing project is to provide a useful tool for frequent ports operations and the following future plans for improvement should ensure that:

- Provide a more generic `su` and `sudo` wrapper with better error handling capabilities.
- Replace the list structure which holds all `PkgObjects` with a tree structure. The main advantage of this solution is that deeper package hierarchies can be mapped correctly.
- Improve the tree-view which displays all ports and make it possible to sort it based on various categories.
- Make it possible to install or remove several ports at once by providing the functionality to make multiple selections.
- Display all possible ports and flavors for a port by parsing the appropriate information in the `Makefile`.
- Constant code review and clean up is also a task which should occur on a regular basis.

[1] `OpenBSD` Journal - http://www.undeadly.org/cgi?action=article&sid=20041107194608

List of Figures

Bibliography

[1] Espie, M. OpenBSD porting information - Important differences from other BSD projects, 2005. http://www.openbsd.org/porting/diffs.html.

[2] Fuller, M. D. The Ports System, 2005. http://www.over-yonder.net/~fullermd/rants/bsd4linux/bsd4linux4.php.

[3] Lacage, M. The Glib Object system, 2004. http://www.le-hacker.org/papers/gobject/.

[4] Lucas, M. BSD Ports Collection Basics, 2000. http://www.onlamp.com/pub/a/bsd/2000/12/21/Big_Scary_Daemons.html.

[5] Lucas, M. Cleaning Up Ports, 2001. http://www.onlamp.com/pub/a/bsd/2001/11/29/Big_Scary_Daemons.html.

[6] Stevens, W. R. *Advanced Programming in the UNIX©Environment*. Addison-Wesley, Indianapolis, 2004.

[7] Viega, J., and Messier, M. *Secure Programming Cookbook for C and C++*. O'Reilly, California, 2003.

[8] Warkus, M. *GNOME 2.0 - Das Entwicklerhandbuch*. Galileo Computing, Bonn, 2002.