>> Gentoo Linux Entwickler HOWTO

[Bitte Kapitel auswählen]

1. Der Portage Verzeichnisbaum

1.1 Einführung

Der Portage Verzeichnisbaum ist normalerweise unter /usr/portage zu finden und ist hierarchisch strukturiert bestehend aus Kategorien gefolgt von Paket-Verzeichnissen. Hier ein Beispiel: Die util-linux-2.11g.ebuild Datei ist unter /usr/portage/sys-apps/util-linux zu finden. In diesem Verzeichnis können sich mehrere verschiedene Versionen neben dem util-linux-2.11g.ebuild befinden. Alle unterschiedelichen Pakete teilen sich unabhängig von ihrer Version dasselbe kategorie/paket Verzeichnis in /usr/portage.

1.2 Den Portage Verzeichnisbaum mit CVS auschecken

Falls CVS noch unbekannt für Sie ist, gibt es das CVS Tutorial. Der Portage Verzeichnisbaum befindet sich im gentoo-x86 Paket des Gento Linux Verzeichnisbaumes. Um das ganze (doch recht grosse) Paket auszuchecken, sollte mit Hilfe obiger Anleitung CVS eingerichtet werden und dann der gentoo-x86 Verzeichnisbaum ausgecheckt werden.

1.3 Was gehört nicht in den Portage Verzeichnisbaum?

Befor ein ebuild geschrieben wird, sollte in bugs.gentoo.org nachgeschaut werden, ob sich nicht jemand anderes bereits die Mühe gemacht hat, das ebuild zu schreiben, welches aber noch nicht in den offiziellem Portage Verzeichnisbaum übernommen wurde. Dabei sollte wie folgt vorgegangen werden: Auf der bugs.gentoo.org "query" auswählen, als Produkt "Gentoo Linux" markieren, als Component "ebuild" auswählen. Im Suchfeld dann den Namen des ebuilds eingeben, sowie als Status NEW, ASSIGNED, REOPENED und (wichtig!) RESOLVED markieren, dann die Suchabfrage starten.

Generell sollte im Portage Verzeichnis nur die .ebuild Dateien und sehr kleine zugehörige Dateien, wie zum Beispiel Patches oder Beispielskonfigurationen gespeichert werden. Diese Dateien sollten immer unter /usr/portage/kategorie/paket/files abgespeichert werden, um das kategorie/paket Verzeichnis nicht zu unübersichtlich machen. Generell ist es für Entwickler keine gute Idee Binärdateien (Nicht-ASCII-Dateien) im CVS abzulegen. Wenn dies dennoch notwenig sein sollte (zum Beispiel eine PNG Grafik aus welchem Grund auch immer), dann sollte sie mit der -kb option in das CVS eingebracht werden:

Befehlsauflistung 1: Code Auflistung 1

```
# cvs add -kb myphoto.png
```

Die -kb option teilt CVS mit, dass myphoto.png eine Binärdatei ist und einer besonderen Behandlung bedarf. Zum Beispiel das Zusammenführen von zwei verschiedenen Binärdateien wird somit nicht erlaubt, was hier auch naheliegend ist. Wenn wir schon beim Zusammenführen sind, alle Patches die dem Portage Verzeichnisbaum hinzugefügt werden, sollten nicht komprimiert sein. Dies erlaubt dann CVS Veränderungen einzubringen und die Entwickler von dabei möglicherweise entstehenden Konfliken korrekt zu warnen.

Nochmals zur Erinnerung: Die Pakete, die in CVS als stable eingecheckt werden, solten wirklich ohne Einschränkungen funktionieren. Es sollte sichergestellt werden, dass ein guter Satz an Standardeinstellungen gewählt wurde, der auf der Mehrheit aller Systeme laufen wird und somit die Benutzer zufrieden stellt. Wenn das Paket Probleme bereitet, oder Sie nicht sicher sind, wie es zum Laufen gebracht werden kann, hilft sicher ein Blick auf andere Distributionen, welche meistens ihre eigene Fassung des Paketes besitzen. Gute Beispiele sind hier Mandrake oder Debian.

Beim Einchecken der ebuilds im CVS sollten alle Entwickler repoman commit statt cvs commit verwenden. Vor Ausführung des commit sollte mit **lintool** das Digest, der Changelog und das Ebuild selbst verifiziert werden.

1.4 Allgemeine CVS Commit Regeln

Warnung
Achtung **lintool** hat einige Probleme. Es sollte stattdessen **repoman** verwendet werden.

- Vor einem commit immer repoman ausführen
- Bitte lintool vor dem commit ausführen
- Immer testen, ob package.mask in Ordnung ist, indem mit 'emerge --pretend glibc' mögliche Konflikte angezeigt werden
- Immer das ChangeLog vor dem commit aktualisieren!
- Um sicherzugehen, das Konflikte während dem CVS commit keine schwerwiegenden Auswirkungen auf den Portage Verzeichnisbaum auftreten, sollte immer vor dem commit des eigentlichen ebuilds, das aktualisierte package.mask eincheckt werden.
- Bitte sauber einchecken! Zuerst das aktualisierte package.mask, dann das ebuild, ChangeLog und die Lizenz in einem Zug, da ansonsten die Benutzer bei ihren Installationen Probleme bekommen.

1.5 Das files Verzeichnis

Wie vorher bereits beschrieben, existiert in jedem paket Unterverzeichnis das Verzeichnis files. Alle Patches, Konfigurationsdateien oder andere Dateien, die von diesem Paket gebraucht werden, gehören hier hinein. Selbst erstellte Patches sollten mit einem Versions-Namen versehen werden, wie zum beispiel meinpaket-1.0-gentoo.diff. Die Gentoo Erweiterung im Namen informiert die Benutzer, dass dieser Patch durch uns, den Gentoo Entwicklern, erstellt wurde und nicht aus einer Mailingliste oder sonst wo gezogen wurde. Auch hier sollten die diffs nicht komprimiert werden, da CVS mit Binärdateien nicht so gut umgehen kann.

Es sollte immer ein Suffix, wie z.b. mypkg-1.0 an das Ende jeder Datei gesetzt werden, die in das files Verzeicnis gehören, um die einzelnen Dateien unmissverständlich den einzelnen ebuild Skripten zuzuordnen können und die Unterschiede zwischen den einzelnen diffs sichtbar werden. Dies ist generell eine gute Sache :-) Natürlich kann, wenn genauere Unterscheidung benötigt wird, ein anderes Suffix verwendet werden.

Bei einer grösseren Anzahl von Dateien empfiehlt es sich ein Unterverzeichnis wie z.B. files/mypkg-1.0 anzulegen und die jeweiligen zugehörigen Dateien hier hineinzukopieren. Bei dieser Methode kann man natürlich auf das Suffix bei den einzelnen Dateien verzichten, da dies bereits aus dem Verzeichnisnamen hervorgeht. Das spart einiges an Arbeit.

2. Die ebuild Skripte

2.1 Einführung

Die ebuild Skripte bilden die Grundlage des gesamten Portage Systems. Sie enthalten die gesamte Information zum Herunterladen, Entpacken, Kompilieren und Installieren des Quellcodes, aber auch eventuelle Änderungen und Konfigurationen, die vor oder nach der Installation oder beim Entfernen durchgeführt werden. Während das meiste von Portage in Python geschrieben ist, sind die ebuilds als bash-Skripte gehalten, da dies das Aufrufen von Befehlszeilen wie auf der Kommandozeile ermöglicht. Eines der Hauptdesign-Prinzipien der ebuild Skripte ist, dass die darin aufgerufenen Befehle denen der Kommandozeile entsprechen, wie wenn der Benutzer das Paket manuell installieren würde. Aus diesem Grund ist die bash-Syntax eine gute Wahl.

Ebuild Skripte werden durch die *ebuild* und *emerge* Befehle ausgeführt. Man muss sich das *ebuild* Kommando als einfaches Handwerkzeug vorstellen. Es kann ein ebuild erstellen und installieren, aber mehr auch nicht. Es kontrolliert, ob Abhängigkeiten erfüllt sind, kann diese aber nicht selbständig auflösen. Auf der anderen Seite ist *emerge* als hochspezialisierte Zentrale für *ebuild*, welches die Möglichkeit besitzt, selbständig abhängige Pakete zu installieren, wenn gewünscht als "was wäre wenn..." dem Benutzer anzeigen, welche ebuilds eingeführt **würden** und vieles mehr. Allgemein sticht *emerge* den *ebuild* Befehl in allen Belangen aus, bis auf einen Punkt. Mit *ebuild* kann der Benutzer inkrementell alle unterschiedlichen Bereiche einer Paketinstallation (herunterladen, entpacken, kompilieren, installieren und einführen) Schritt für Schritt ausführen. Für Entwickler ist dies ein unverzichtbares Werkzeug zum debuggen, da nur so Probleme mit dem ebuild auf einen kleinen Bereich des Gesamtprozesses eingegrenzt werden können.

2.2 Die Benennung von ebuild Dateien

Jeder Ebuild Dateiname besteht aus vier Abschnitten:

Der erste Abschnitt ist der Paketname, der nur aus Kleinbuchstaben, den Zahlen 0-9 und dem Bindestrich ('-') enthalten darf. Beispiele sind: *util-linux*, *sysklogd* und *glibc*.

Der zweite Abschnitt ist die Versionsnummer des Paketes, welche normalerweise mit der Versionsnummer des Quellcode Tarballs übereinstimmt. Diese besteht normalerweise aus zwei oder drei Nummern, die durch Punkte von einander getrennt sind, wie zum Beispiel *1.2* oder *4.5.2* (sehr

lange, durch Punkte voneinander getrennte Nummern werden auch unterstützt) und darf von einen einfachen Buchstaben auf die letzte Zahl gefolgt werden, zum Beispiel: *1.4b* oder *2.6h*. Die Paketnummer wird mit der Versionsnummer durch einen Bindestrich verbunden: *foo-1.0*, *bar-2.4.6*, etc.

Wichtig
Wenn Sie mit dem Gedanken spielen, einen angehängten Buchstaben in Ihrer Versionsnummer zu verwenden, sollten Sie darüber im klaren sein, dass dieser Buchstabe **nicht** dazu verwendet werden sollte, einen eventuellen Alpha oder Beta Status anzuzeigen, da alphas und betas als **prereleases** zählen, angehängte Buchstaben jedoch als **neuere Versionen**. Das ist eine grosse Unterscheidung, da Portage die Versionsnummer eines ebuilds dazu verwendet, herauszufinden, ob es neuer oder älter ist als ein Paket mit gleichem Namen aus derselben Kategorie. Es ist sehr wichtig, dass Versionsnummern glaubwürdig die Version eines Paketes repräsentieren, damit Portage seine Abhägigkeits-Prüfungen korrekt ausführt..

Der dritte Abschnitt (optional) enthält eine spezielle Suffix, entweder *_alpha*, *_beta*, *_pre* oder *_rc*. Alle diese Suffixe werden durch eine Nummer ergänzt, zum Beispiel *linux-2.4.0_pre10*. Bei identischen Versionsummern geht Portage davon aus, dass *_alpha* älter als *_beta* ist, *_beta* älter als *_pre* und *_pre* äter als *_rc* ist.

Notiz
Ein *_rc* Paket ist älter als ein Paket ohne vorausgehende Suffixe mit Unterstrich (z.B. *linux-2.4.0*) und *linux-2.4.0* ist älter als ein Paket mit einem Buchstaben Präfix, hier *linux-2.4.0b*. Nochmal: Diese Versionsinformationen sind wichtig, da Portage es zur Bestimmung heranzieht, ob ein Paket oder ebuild älter oder jünger ist, als ein Paket aus der selben Kategorie mit gleichem Namen.

Der vierte Abschnitt (ebenfalls optional) ist die Gentoo Linux spetzifische **Revisionsnummer**, welche durch *-r#* angegeben wird, wobei *#* ein Integer (Zahlenwert) ist, als Beispiel: *package-4.5.3-r3*. Die Revisionsnummer ist unabhängig von der Versionsnummer des Quellpaketes und kann dazu genutzt werden, die Benutzer davon in Kennntnis zu setzen, dass es sich um ein neues bzw. erweiterte Revision eines ebuilds handelt.

Wenn grössere Veränderungen an einem bestehenden ebuild vorgenommen werden, sollte die Datei vorher kopiert und mit einer um 1 erhöhten Revisionsnummer abgespeichert werden. Die ersten Fassungen besitzen normalerweise keine Revisionsnummer, z.B. package-4.5.3, da sie von Portage so betrachtet werden, als ob sie eine Revisionsnummer von Null haben. Das bedeutet für die Zählweise: *1.0* (erste Version), *1.0-r1*, *1.0-r2*, und so weiter.

Und natürlich gehen wir davon aus, dass der **fünfte** Abschnitt der ebuild Bezeichnung nicht vergessen wird, es ist die *.ebuild* Endung.

2.3 Aufbau einer ebuild Datei

1. Setzen der Variablen:

Der Anfang jeder ebuild Datei besteht aus einigen Variablen, die wie folgt gesetzt werden können:

P	Der Name und die Versionsnummer des Paketes, dies muss normalerweise nicht gesetzt werden, da es von Portage durch den Dateinamen bestimmt wird
A	Der Name ohne Pfadangabe von dem Haupt-Quellcode des Paketes
S	Das Quellcode-Verzeichnis für das Paket, im Normalfall${WORKDIR}/${P}
DESCRIPTION	Eine kurze Beschreibung des Paketes in einem Satz
SRC_URI	sind die URIs für jede Quellcode-Datei in diesem Paket, welche durch Leerzeichen getrennt werden. Das Erste ist normalerweise so etwas wie: **ftp://ftp.company.com/pub/somepackage/${A}**
HOMEPAGE	ist die Homepage des Pakets
DEPEND	build Abhängigkeiten, siehe dazu den Abschnitt Package Dependencies
RDEPEND	runtime Abhängigkeiten. mehr dazu im Abschnitt Package Dependencies

2. ebuild Funktionen

Es gibt verschiedene Funktionen, die ein den ebuild Dateien definiert werden können, um den Prozess der Zusammenstellung und Installation des Paketes steuern zu können.

pkg_setup	mit dieser Funktion können vorbereitende Schritte durchgeführt werden. Dazu gehört die Kontrolle des Benutzerkontos oder die Kontrolle nach einer existierenden Konfigurationsdatei. Diese Funktion muss den Wert 0 zurückgeben, damit das ebuild fortgesetzt werden kann.
pkg_nofetch	Informiert den Benutzer über Aktionen oder Downloads, die er selbst ausführen muss (z.B. aus Lizenzierungsgründen).

src_unpack	Mit dieser Funktion kann der Quellcode entpackt werden und autoconf/automake/etc. ausgeführt werden. Standardmässig wird das Paket in *${A}* entpackt. Das Standard-Startverzeichnis ist *${WORKDIR}*.
src_compile	Damit wird das Paket konfiguriert und kompiliert. Das Standard-Startverzeichnis ist *${S}*.
src_install	Mit dieser Funktion wird das Paket in *${D}* installiert. Benutzt das Paket automake, kann dies mit *make DESTDIR=${D} install* vereinfacht werden. **Es sollte sichergestellt werden, dass die Installation aller Dateien in *${D}* als root ausgeführt wird!**
pkg_preinst	Die Befehle in dieser Funktion werden vor dem mergen der Dateien ausgeführt.
pkg_postinst	Die Befehle in dieser Funktion werden nach dem mergen der Dateien ausgeführt.
pkg_prerm	Die Befehle in dieser Funktion werden vor dem Entfernen der Dateien ausgeführt.
pkg_postrm	Die Befehle in dieser Funktion werden nach dem Entfernen der Dateien ausgeführt.
pkg_config	Mit dieser Funktion kann die initiale Konfiguration des Paketes nach der Installation ausgeführt werden. Alle Pfade in dieser Funktion sollen mit dem Prefix ${ROOT} versehen werden. Diese Funktion wird **nur** ausgeführt, wenn der Benutzer folgenden Befehl ausführt: *ebuild /var/db/pkg/${CATEGORY}/${PF}/${PF}.ebuild config*.

Folgende Funktionen können ebenfalls im ebuild verwendet werden:

use	Kontrolle ob, eine oder mehere gegebenen USE-Flags gesetzt sind. Ist dies der Fall, wird die Funktion die USE-Flag zurückgeben. Zur Kontrolle der Existenz einer solchen Flag kann *[-z "`use foobar`"]* ausgeführt werden.
has_version	Gibt 1 zurück, wenn auf dem System die geforderte version installiert ist. Als Beispiel: *has_version >=glibc-2.3.0*.
best_version	Gibt **category/package-version** des jeweiligen Pakets zurück. Beispiel: *best_version x11-libs/gtk+extra*.
use_with	Diese Funktion prüft ob eine USE-Flag gesetzt wurde und gibt enstprechend "--with-foobar" oder "--without-foobar" zurück. Bei einem Argument ist dieses sowohl der USE-Flag und der with(out) String. Andernfalls ist das erste Argument der USE-Flag und das zweite Argument der with(out) String. Beispiel: *use_with truetype freetype*
use_enable	Hat die gleiche Funktion wie use_with, gibt jedoch "--enable foobar" oder "--disable foobar" zurück.
check_KV	Prüft ob Portage die Kernel-Versionsnummer erkennt. Ist dies nicht der Fall wird eine Fehlermeldung ausgegeben und das ebuild abgebrochen. Wenn im Script die Versionsnummer geprüft werden soll, kann *${KV}* verwendet werden, welche automatisch durch Portage definiert wird.
keepdir	Erstellt ein .keep Datei im gegebenen Verzeichnis, so dass dies nicht automatisch von Portage wieder entfernt werden kann.
econf	Führt *./configure*mit den notwendigen Pfadänderungen (Prefix,host, mandir, infodir, datadir, sysconfdir, localstatedir) durch. Optionale Argumente für *./configure*können gesetzt werden.
einstall	Führt ein *make install* durch, sodass Portage weiss, wohin die Dateien installiert werden.
die	Bricht den aktuellen Prozess ab. Das übergebene Argument wird dem Benutzer als Nachricht dargestellt.
einfo	Kann verwendet werden, um den Benutzer über wichtige Dinge zu informieren. Das Argument das an *einfo* übergeben wird, ist die Nachricht, die der Benutzer erhält.

2.4 Regeln zum Schreiben eines ebuilds

Da ebuilds in Wirklichkeit nur aus einfachen Shell Skripten bestehen, sollte der zu bearbeitende Editor auf diesen Modus eingestellt werden. Korrektes Einrücken sollte beachtet werden, nur Tab-Sprünge und keine Leerzeichen. Am besten den Editor so einstellen, dass die Tabulatorsprünge alle vier Leerzeichen gesetzt sind. Die Klammern um die Umgebungsvariablen sollten nicht vergessen werden; d.h. *${P}* anstatt nur *$P*.

Längere Zeilen sollten mit '\' umgebrochen werden:

Befehlsauflistung 2: Code Auflistung 2.1

```
./configure \
--prefix=/usr || die "configure failed"
```

Für weitere Details kann man einen Blick in **skel.ebuild** werfen, dies befindet sich in /usr/portage.

Bei der Benutzung von Vim kann folgender Code am Ende der .vimrc Datei eingetragen werden, damit sind automatisch alle Einstellungen für Gentoo-Dokumente gesetzt.

Befehlsauflistung 3: Code Auflistung 2.2

```
if (getcwd() =~ 'gentoo-x86\|gentoo-src\|portage')
set tabstop=4 shiftwidth=4 noexpandtab
endif
```

Bei der Benutzung von emacs kann in das .emacsrc (GNU Emacs) oder in die init.el (XEmacs) eingetragen werden.

Befehlsauflistung 4: Code Auflistung 2.3

```
(defun ebuild-mode ()
(shell-script-mode)
  (sh-set-shell "bash")
  (make-local-variable 'tab-width)
  (setq tab-width 4))
(setq auto-mode-alist (cons '("\\.ebuild\\'" . ebuild-mode) auto-mode-alist))
(setq auto-mode-alist (cons '("\\.eclass\\'" . ebuild-mode) auto-mode-alist))
```

2.5 Die USE Variablen

Die USE-Variablen dienen dem Zweck, Portage global zu konfigurieren und automatisch bestimmte Kompilierungsoptionen an- oder abzuschalten. Hier ist ein Beispiel. Wir gehen nun davon aus, dass Sie ein GNOME-Fan sind, und jedes ebuild, welches eine optionale GNOME Unterstützung anbietet, soll dies automatisch miteinkompilieren. In diesem Fall fügen wir *gnome* in die USE variable von **/etc/make.conf** hinzu und Portage berücksichgt dies bei jeder Kompilierung eines Paketes mit GNOME Unterstützung. Im umgekehrten Fall, wenn optinale GNOME Funktionen nicht gewünscht sind, editiert man /etc/make.conf und stellt sicher, dass *gnome* in der USE Variable nicht gesetzt ist. Gentoo Linux besitzt eine überwältigende Anzahl von USE Optionen, die es Ihnen erlauben, das System genau so zu konfigurieren, wie Sie es haben möchten.

Notiz
Wenn eine USE Variable deaktiviert ist (hier im Beispiel *gnome*) wirk sich dies nur auf die **optinalen** Kompilierungsoptionen aus. Wenn Sie jedoch ein ebuild *emergen*, welches auf ein GNOME Paket angewiesen ist, wird GNOME automatisch als eine Abhängigkeit hiervon mitinstalliert, falls es nicht bereits erfolgt ist. Das ist auch der Grund, weshalb man vor dem eigentlichen *emerge* ein *emerge --pretend* ausführen sollte. In so einem Fall wissen Sie bereits vorher Bescheid, was alles auf Ihrem System installiert wird.

In Ihren eigenen ebuilds kann mit Hilfe der USE Variable geprüft werden, ob die entsprechende Variable vom Besitzer der Installation bereits gesetzt ist. Das *use* Kommando gibt den Namen jeder Variable zurück, die USE und seiner Befehlszeile präsent ist. Der Befehl wird normalerweise so angewendet:

if ["`use X`"]; then commands; fi

USE Variablen können auch dazu verwendet werden, bestimmte Abhängigkeiten zu setzen. In unserem Beispiel wollen wir ein bestimmtes Paket nur dann als erforderlich markieren, wenn eine bestimmte USE Variable gesetzt ist. Dies kann mit folgender Syntax durchgeführt werden: *variable? (mycat/mypackage-1.0-r1)* in der DEPEND Zeile Ihres ebuilds. In diesem Fall wird *mycat/mypackage-1.0-r1* nur dann erforderlich, wenn die entsprechende Variable in den USE vorhanden ist. Umgekehrt ist es auch möglich, festzulegen, welche weitere Abhängigkeit verwendet werden soll, wenn eine USE-Flag **nicht** gesetzt ist: *variable? (mycat/mypackage-1.0-r1) : (othercat/otherpackage-1.0-r5)*. In diesem Fall wird dann das Paket *othercat/otherpackage-1.0-r5* anstatt von *mycat/mypackage-1.0-r1* installiert. Soll ein Abhängigkeit nur dann installiert werden, wenn die Variable nicht vorhanden ist, verwendet man *!variable? (mycat/mypackage-1.0-r1)*. Es sollte sichergestellt werden, dass nur die oben genannte Syntax verwendet wird und nicht die Bash eigene "ifs". Bash Bedingungen stören Portages Abhängigkeits-Cache, sodass die Verwendung dieser zu einem funktionsunfähigem ebuild führen wird.

Hier noch ein wichtiger Tipp zur Verwendung von *USE*. In dem meisten Fällen besitzt das Paket bereits ein *./configure* Skript um das Paket zu konfigurieren. Optionale Bestandteile werden durch Übergabe bestimmer Argumente zur Laufzeit mitkompiliert. Zur Übergabe der USE-Flags an das Konfikurationsskript geht man am besten wie folgt vor: Zunächst sollte herausgefunden werden, ob eine bestimmte *./configure* Option standardmässig ein- oder abgeschaltet ist.

Befehlsauflistung 5: Code Auflistung 2.4

```
DEPEND="X? ( >=x11-base/xfree-4.3 )
        mysql? ( >=dev-db/mysql-3.23.49 )
        apache2? ( >=net-www/apache-2 ) : ( =net-www/apache-1.* )"

src_compile() {
        local myconf
        use X || myconf="--disable-x11"
        use mysql || myconf="${myconf} --disable-mysql"
```

```
    ./configure ${myconf} --prefix=/usr --host=${CHOST} || die
    emake || die
}
```

Im obigen Beispiel prüfen wir, ob die X und mysql USE Variablen abgeschaltet sind. Der Abschnitt *use X ||* kontrolliert, ob X in der USE Variable vorhanden ist. Ist dies nicht der Fall, so übergibt er das Argument *myconf="--disable-x11"* an das Konfigurationsskript. Es ist nicht notwendig X11 und mysql explizit zu aktivieren, da diese standardmässig bereits eingeschaltet sind. Wenn jedoch eine bestimmte Option standardmässig abgeschaltet ist kann man wie folgt vorgehen:

Befehlsauflistung 6: Code Auflistung 2.5

```
DEPEND="X? ( >=x11-base/xfree-4.3 )
        mysql? ( >=dev-db/mysql-3.23.49 )"

src_compile() {
    local myconf
    use X && myconf="--enable-x11"
    use mysql && myconf="${myconf} --enable-mysql"

    ./configure ${myconf} --prefix=/usr --host=${CHOST} || die
    emake || die
}
```

In diesem Beispiel werden X11 und MySQL-Unterstützung nur dann explizit aktiviert, wenn die entsprechende USE Variable vorhanden ist. Der Abschnitt *use mysql &&* prüft, ob mysql in der USE-Flag gesetzt ist, und übergibt dann *myconf="${myconf} --enable-mysql"*.

Eine kontnuierlich aktualisierte Liste aller USE Variablen fidnet mann hier.

3. Aufbau des Dateisystems

3.1 Einführung in FHS

Der Aufbau des Gentoo Linux Dateisystems orientiert sich am am FHS Standard, kurz für: **Filesystem Hierarchy Standard**. Eine kurze Beschreibung dieses Standards wir hier gegeben, eine komplette Spezifikation findet man unter http://www.pathname.com/fhs/ .

Notiz
Der /opt Breich wird in Abschnitt 3.12 der FHS Spezifikation beschrieben. Abschnitt 4.4 behandelt das /usr/X11R6 Verzeichnis. KDE und GNOME werden nicht spezifisch besprochen und sind in der aktuellen Fassung des FHS überhaupt nicht berücksichtigt.

3.2 Wohin mit meinem Paket in diesem Dateisystem?

Normalerweise wird bei der Verwendung von autoconf und automake das Paket standardmässig korrekt installiert, allerdings mit einigen wenigen Ausnahmen:

- Wenn das Programm nach /bin, /sbin, /usr/bin oder /usr/sbin installiert wird, sollten die dazugehörigen Man-Pages nach /usr/share/man kopiert werden. Dies kann meistens durch Übergabe des Arguments *./configure --mandir=/usr/share/man* im ebuild Skript bewerkstelligt werden.
- GNU Info Dateien gehören nach /usr/share/info, **auch wenn die Info-Dateien X11, GNOME oder KDE-Programme betreffen**. Nochmals: /usr/share/info ist der **einzige** offizielle Platz für GNU Info-Dateien! Leider instellieren die meisten ./configure Skripte die GNU Dateien nach /usr/info, sodass man mit Hilfe des *--infodir=/usr/share/info* Arguments nachhelfen muss.
- Dokumentationen werden in ein Unterverzeichnis von /usr/share/doc installiert. Dies sollte Namen, Version und Revisionsnummer des betreffenden Programmes beinhalten. Das gilt für alle Programme, GNOME, KDE, X11 wie auch der Konsole. Manchmal wird weitere Dokumentation für spezielle Zwecke in die /usr/shareHierarchy eingebunden.
- X11-spezifische Programme und Bibliotheken sollten immer nach /usr installiert werden und nicht direkt nach /usr/X11R6, welches ausschliesslich für das X Window System, Version 11 Release 6 vorbehalten ist. Dies ist wahrscheinlich einer genauere Interpretation der FHS Spezifikation, als es in anderen Distributionen üblich ist.
- GNOME und KDE Programme gehören ebenfalls nach /usr.

Wichtig
Manche Distributionen installieren KDE und GNOME nach /opt. Zur Zeit existiert kein Standard für die Installtion von Dateien der Desktop-Umgebungen. Im Interesse einer einfachen und konsistenten Handhabung haben wir uns entschieden, alle KDE und GNOME Pakete nach /usr zu installieren.

Generell sollte das ebuild seine Dateien in den /usr Verzeichnisbaum installieren. Manche Programme können mit oder ohne GNOME; KDE und X11 Bibliotheken kompiliert werden, was für verwirrung sorgt. Unsere Lösung, alles nach /usr zu installieren, verhindert doppelte Arbeit und unnötige Komplexität für die ebuild Entwickler. Der Pfad, in den die Programmdateien installiert werden, sollte nicht an das Vorhandensein oder Fehlen von USE Variablen geknüpft werden. Somit installieren die ebuilds im Portage Verzeichnisbaum in fast allen Fällen ihre Dateien nach /usr.

Notiz
Das /opt Verzeichnis in Gentoo Linux ist für binäre Pakete reserviert. Als Beispiele können hier mozilla-bin, acroread und realplayer angeführt werden. Die hierhin installierten Pakete benötigen in der Regel eine /etc/env.d/foo Markierungsdatei. Dies dient der Möglichkeit, Pfade und zusätzliche Variablen der Laufzeitumgebung hinzuzufügen können.