

Programmierpraktikum Haskell

Sommersemester 2004

emerge2 — a package management system

Abgabetermin: 18. Juli 2004

Ziel des vorliegenden Projekts ist die Erstellung eines einfachen Systems zur Verwaltung von Softwarepaketen. Das zu erstellende System ist im wesentlichen eine Re-Implementierung von Gentoos „portage system“. [Gentoo ist eine beliebte Linux Distribution.]

1 Einführung

In diesem Abschnitt wird Gentoos „portage system“ kurz vorgestellt. Zur Zeit (Mai 2004) unterstützt Portage fast 7000 verschiedene Softwarepakete (im folgenden kurz Pakete genannt). Dazu gehören vollständige Benutzungsoberflächen wie Gnome oder KDE, Office Suites, FTP-, Mail- und Webserver, Übersetzer für Dutzende von Programmiersprachen, T_EX, Spiele und vieles andere mehr. Zur besseren Übersicht sind die Pakete in insgesamt 107 verschiedene Kategorien eingeteilt. Der Glasgow Haskell Compiler zum Beispiel befindet sich in der Kategorie `dev-lang`.

Um ein Paket auf einem Rechner zu installieren, verwendet man den Befehl `emerge`. Nehmen wir an, dass wir `detex` benötigen, einen einfachen Filter, um T_EX-Steuersequenzen aus Textdateien zu entfernen.

```
basilisk root # emerge --pretend detex
These are the packages that I would merge, in order:
[ebuild N    ] dev-tex/detex-2.7
```

Die Option `--pretend` instruiert Portage, das Paket nicht unmittelbar zu installieren, sondern zunächst nur die dazu notwendigen Aktionen mitzuteilen. So erfahren wir, dass sich `detex` in der Kategorie `dev-tex` befindet, dass die aktuelle Version 2.7 ist, und dass das Paket noch nicht installiert ist (N). Ist das Paket bereits vorhanden, wie im folgenden Beispiel,

```
basilisk root # emerge --pretend ghc
These are the packages that I would merge, in order:
[ebuild R    ] dev-lang/ghc-6.2.1
```

wird dies mit einem R signalisiert (ließen wir `--pretend` weg, so würde das Paket noch einmal installiert: `re-emerge`).

Ein Paket kann von vielen anderen Paketen abhängen. Falls alle oder einige der benötigten Pakete nicht vorhanden sind, müssen diese zunächst installiert werden.

```
basilisk root # emerge --pretend kdenlive
These are the packages that I would merge, in order:
[ebuild N    ] media-gfx/graphviz-1.10
[ebuild N    ] dev-util/valgrind-2.0.0
[ebuild N    ] dev-util/calltree-0.9.6
[ebuild N    ] kde-base/kdesdk-3.2.2
```

```

[ebuild N ] sys-libs/libraw1394-0.9.0
[ebuild N ] media-libs/libdv-0.99-r1
[ebuild N ] dev-cpp/libxmlpp-0.27.0
[ebuild N ] media-sound/esound-0.2.34
[ebuild N ] gnome-base/gnome-libs-1.4.2
[ebuild N ] media-libs/gdk-pixbuf-0.22.0
[ebuild N ] sys-libs/libavc1394-0.4.1
[ebuild N ] media-libs/libsndfile-1.0.5
[ebuild N ] media-video/piave-0.2.4
[ebuild N ] media-video/kdenlive-0.2.4

```

In diesem Beispiel werden vor der gewünschten Installation von `kdenlive`, einem Videobearbeitungsprogramm, zunächst 13 andere Pakete eingespielt. Die Abhängigkeiten kann man sich auch als Baum anzeigen lassen (dabei sind Kinder von Eltern abhängig).

```

basilisk root # emerge --pretend --tree kdenlive
These are the packages that I would merge, in reverse order:
[ebuild N ] media-video/kdenlive-0.2.4
[ebuild N ] media-video/piave-0.2.4
[ebuild N ] media-libs/libsndfile-1.0.5
[ebuild N ] sys-libs/libavc1394-0.4.1
[ebuild N ] media-libs/gdk-pixbuf-0.22.0
[ebuild N ] gnome-base/gnome-libs-1.4.2
[ebuild N ] media-sound/esound-0.2.34
[ebuild N ] dev-cpp/libxmlpp-0.27.0
[ebuild N ] media-libs/libdv-0.99-r1
[ebuild N ] sys-libs/libraw1394-0.9.0
[ebuild N ] kde-base/kdesdk-3.2.2
[ebuild N ] dev-util/calltree-0.9.6
[ebuild N ] dev-util/valgrind-2.0.0
[ebuild N ] media-gfx/graphviz-1.10

```

Hat man sich schließlich entschlossen, ein Paket zu installieren, läßt man die Option `--pretend` weg. Dann lädt Portage automatisch die Quellen (Gentoo verwendet in der Regel keine Binärpakete), überprüft deren Integrität, übersetzt diese und installiert abschließend die erzeugten Programme, die Dokumentation usw.

```

basilisk root # emerge detex
>>> emerge (1 of 1) dev-tex/detex-2.7 to /
>>> Downloading ftp://sunsite.informatik.rwth-aachen.de/gentoo/distfiles/detex-2.7.tar
>>> md5 src_uri ;-) detex-2.7.tar
>>> Unpacking source...
>>> Unpacking detex-2.7.tar to /var/tmp/portage/detex-2.7/work
>>> Source unpacked.
sed -f states.sed detex.l > xxx.l
flex xxx.l
rm -f xxx.l
mv lex.yy.c detex.c
gcc -march=pentium3 -O3 -pipe -c -o detex.o detex.c
gcc -march=pentium3 -O3 -pipe -o detex detex.o -lfl
>>> Install detex-2.7 into /var/tmp/portage/detex-2.7/image/ category dev-tex
>>> Completed installing into /var/tmp/portage/detex-2.7/image/
>>> Merging dev-tex/detex-2.7 to /
--- /usr/
--- /usr/bin/

```

```

>>> /usr/bin/detex
--- /usr/share/
--- /usr/share/doc/
>>> /usr/share/doc/detex-2.7/
>>> /usr/share/doc/detex-2.7/README.gz
--- /usr/share/man/
--- /usr/share/man/man1/
>>> /usr/share/man/man1/detex.1l.gz
* Caching service dependencies...
>>> dev-tex/detex-2.7 merged.
>>> Recording dev-tex/detex in "world" favorites file...
>>> clean: No packages selected for removal.
>>> Auto-cleaning packages ...
>>> No outdated packages were found on your system.
* GNU info directory index is up-to-date.

```

2 Hintergrund

Im folgenden nehmen wir Gentoo's „portage system“ etwas genauer unter die Lupe. Nicht alle vorgestellten Aspekte sind für das durchzuführende Projekt relevant. Trotzdem ist es sicherlich hilfreich, wenn man einen Überblick über die Funktionsweise von Portage hat.

2.1 Ebuilds

Das Herz von Portage bilden die sogenannten Ebuild-Skripte (im folgenden kurz Skripte genannt). Für jedes Paket gibt es (mindestens) ein Skript, das alle Informationen, die für die Installation notwendig sind, bündelt. Ein Skript bezieht sich immer auf eine bestimmte Paketversion. Gibt es von einem Paket mehrere Versionen, so gibt es entsprechend auch mehrere Skripte. Abbildung 1 zeigt das Skript für GNUs „stream editor“ `sed`. Das Skript enthält unter anderem eine kurze Beschreibung des Pakets, die Adresse der Quellen, die Lizenz, und Instruktionen zur Übersetzung und Installation des Pakets. Am Anfang des Skripts werden verschiedene Variablen definiert: die Variable `KEYWORDS` enthält eine Liste aller Architekturen, auf denen das Paket läuft; die Variable `DEPEND` listet Abhängigkeiten von anderen Paketen auf; die Variable `IUSE` zählt für das Paket relevante USE-Flags auf. Was es mit diesen Variablen auf sich hat, beleuchten wir in den nächsten Abschnitten genauer.

2.2 Keywords

Nicht jedes Paket funktioniert auf jeder Rechnerarchitektur. Die Architekturen, auf denen das Paket stabil läuft, sind unter `KEYWORDS` aufgeführt. Befindet sich ein Paket im Teststadium, wird dem Namen der Architektur eine Tilde vorangesetzt; weiß man, dass das Paket auf einer bestimmten Architektur *nicht* läuft, wird dies mit einem Minuszeichen festgehalten.

```
KEYWORDS="~x86 -sparc"
```

2.3 USE flags

Einige Pakete verfügen über optionale Features, die über sogenannte USE flags ausgewählt werden können. GNUs „stream editor“ zum Beispiel bietet „native language support“. Dazu muss das USE flag `nls` gesetzt werden: entweder in der globalen Konfigurationsdatei `/etc/make.conf` oder beim Aufruf von `emerge` in der Kommandozeile

```
basilisk root # USE="nls" emerge sed
```

Auf diese Weise wird `sed` mit „native language support“ übersetzt.

```

DESCRIPTION="Super-useful stream editor"
SRC_URI="mirror://gnu/sed/${P}.tar.gz"
HOMEPAGE="http://www.gnu.org/software/sed/sed.html"

KEYWORDS="x86 amd64 ppc sparc alpha hppa ia64"
SLOT="0"
LICENSE="GPL-2"
IUSE="nls static build"

DEPEND="virtual/glibc
        nls? ( sys-devel/gettext )"

src_compile() {
    local myconf
    use nls \
        && myconf="${myconf} --enable-nls" \
        || myconf="${myconf} --disable-nls"
    econf ${myconf} || die "Configure failed"
    if [ -z 'use static' ] ; then
        emake || die "Shared build failed"
    else
        emake LDFLAGS=-static || die "Static build failed"
    fi
}

src_install() {
    into /
    dobin sed/sed
    if [ -z "use build" ]
    then
        einstall || die "Install failed"
        dodoc COPYING NEWS README* THANKS TODO AUTHORS BUGS ANNOUNCE ChangeLog
        docinto examples
        dodoc ${FILESDIR}/dos2unix ${FILESDIR}/unix2dos
    else
        dodir /usr/bin
    fi
    rm -f ${D}/usr/bin/sed
    dosym ../../bin/sed /usr/bin/sed
}

```

Abbildung 1: Ebuild für GNUs „stream editor“ sed.

```

PROVIDE="virtual/ghc"
DEPEND="virtual/ghc
  >=dev-lang/perl-5.6.1
  >=sys-devel/gcc-2.95.3
  >=sys-devel/make-3.79.1
  >=sys-apps/sed-3.02.80
  >=sys-devel/flex-2.5.4a
  >=dev-libs/gmp-4.1
  doc? ( >=app-text/openjade-1.3.1
    >=app-text/sgml-common-0.6.3
    =app-text/docbook-sgml-dtd-3.1-r1
    >=app-text/docbook-dsssl-stylesheets-1.64
    >=dev-haskell/haddock-0.6-r2
    tetex? ( >=app-text/tetex-1.0.7
      >=app-text/jadetex-3.12 ) )
  opengl? ( virtual/opengl virtual/glu virtual/glut )"
RDEPEND="virtual/glibc
  >=sys-devel/gcc-2.95.3
  >=dev-lang/perl-5.6.1
  >=dev-libs/gmp-4.1
  opengl? ( virtual/opengl virtual/glu virtual/glut )"

```

Abbildung 2: Abhängigkeiten des Glasgow Haskell Compilers.

2.4 Abhängigkeiten

Eines der wichtigsten Merkmale von Portage ist die automatische Bestimmung und Auflösung von Paketabhängigkeiten. Die Abhängigkeiten werden im jeweiligen Skript in den Variablen `DEPEND` (Abhängigkeiten zur Übersetzungszeit) und `RDEPEND` (Abhängigkeiten zur Laufzeit) festgehalten. GNUs „stream editor“ ist zum Beispiel von der C Bibliothek abhängig (`virtual/glibc`). Ist das `USE` flag `nls` gesetzt, besteht eine zusätzliche Abhängigkeit von GNUs `gettext` Programm (`nls? (sys-devel/gettext)`). Eine wesentlich längere Liste von Abhängigkeiten hat der Glasgow Haskell Compiler, siehe Abbildung 2. Wie man der Abbildung entnehmen kann, lassen sich auch Abhängigkeiten von bestimmten Paketversionen spezifizieren. So muss etwa GNUs C Compiler mindestens in der Version 2.95.3 vorliegen. Die genaue Syntax für Abhängigkeiten ist auf der „man page“ für `ebuild` beschrieben (`man 5 ebuild`).

2.5 Aktualisierung

Die beim Aufruf von `emerge` angegebenen Pakete merkt sich Portage als persönliche Favoriten (`/var/cache/edb/world`). So lässt sich mit zwei Befehlen stets das gesamte System aktualisieren.

```

basilisk root # emerge sync
>>> starting rsync with rsync://rsync.informatik.rwth-aachen.de/gentoo-portage...
...
>>> Updating Portage cache... ...done!
basilisk root # emerge --pretend --update world
These are the packages that I would merge, in order:
[ebuild U ] sys-apps/texinfo-4.6 [4.5]
[ebuild N ] sys-kernel/development-sources-2.6.7_rc1
[ebuild U ] net-misc/keychain-2.3.0 [2.2.2]
[ebuild UD] net-misc/rsync-2.6.0 [2.6.2-r2]
[ebuild U ] sys-devel/autoconf-2.59-r3 [2.58-r1]

```

Die Ebuild-Skripte sind lokal auf jedem Gentoo Rechner abgelegt (`/usr/portage/`). Mit `emerge sync` werden diese auf den neuesten Stand gebracht, das heißt, das lokale Portage Repository wird mit dem globalen Repository (bzw. mit einem der gespiegelten Repositories) über das Internet synchronisiert.

Mit `emerge --update world` lassen sich anschließend alle Pakete aus der Liste der Favoriten aktualisieren: existiert von einem Paket eine neue Version, so wird diese eingespielt. Natürlich können auch gezielt einzelne Pakete aktualisiert werden.

```
basilisk root # emerge --update kde
```

2.6 Literatur

Weiterführende Informationen zu Gentoo und Portage finden sich hier:

- <http://www.gentoo.org/> — Die Homepage von Gentoo
- <http://www.gentoo.org/doc/en/gentoo-howto.xml> — das Ebuild HOWTO
- `man emerge` — man page zum `emerge` Kommando
- `man ebuild` und `man 5 ebuild` — man page zum `ebuild` Kommando und zum Format der ebuild-Skripte

3 Projektbeschreibung

In diesem Projekt soll die wesentliche Funktionalität von Portage (`emerge`) nachgebildet werden: Installation, Aktualisierung und Deinstallation von Paketen. Ferner sollte das System, Auskunft über installierte Pakete geben können, erklären, warum bestimmte Pakete installiert sind, nicht mehr benötigte Pakete auf Wunsch entfernen und an Hand von Stichworten nach passenden Paketen suchen.

Es ist natürlich nicht notwendig, dass die Installation oder Deinstallation tatsächlich vollzogen wird (dazu müsste man über ein laufendes Gentoo System verfügen). Es genügt, über die Aktionen Buch zu führen bzw. wenn man über eine Gentoo Installation verfügt, die Aktionen an `emerge` oder `ebuild` zu delegieren.

4 Einige softwaretechnologische Hinweise

Die Gruppeneinteilung nehmen die Teilnehmer nach eingehender Absprache selbst untereinander vor. Hierbei muss auf eine möglichst gerechte Aufgabenverteilung bezüglich des Entwurfs, der Implementierung und des Testens geachtet werden.

In diesem Praktikum sollen insbesondere wesentliche Schritte eingeübt werden, die bei der Entwicklung von Software-Produkten von Bedeutung sind. Deshalb werden im folgenden einige Prinzipien und Methoden der Software-Entwicklung vorgestellt. Für unser Projekt schlagen wir folgendes Phasenmodell vor:

Planungsphase. Diese umfasst eine vorläufige umgangssprachliche Produktbeschreibung, die in Teilen im ersten Abschnitt vorgegeben ist.

Definitionsphase. In dieser Phase werden ein Produktmodell und eine Anforderungsliste entworfen, in der präzise und widerspruchsfrei festgehalten ist, was das zu erstellende Software-Produkt aus der Benutzersicht zu leisten hat. Dabei werden allerdings keine Entscheidungen, die die spätere Realisierung des Programms betreffen, vorweggenommen.

Entwurfsphase. Hier wird eine Konzeption entwickelt, aus der ersichtlich ist, wie das zu erstellende Produkt aussieht. Aus dem Produktmodell heraus wird die Architektur des Systems entworfen; die einzelnen Komponenten (Module) werden hinsichtlich ihres Funktionsumfanges und ihren Beziehungen (Modulschnittstellen) untereinander beschrieben.

Implementierungsphase. In dieser Phase wird ein ausführbares Programm als Realisierung der in der Definitions- und Entwurfsphase beschriebenen Modelle erstellt. Abhängig von den Schnittstellenspezifikationen werden die einzelnen Module entworfen, kodiert und getestet, bevor sie in das Endprodukt integriert werden.

Testphase. In einer Abnahme-Testserie wird das Produkt auf die Einhaltung der definierten funktionellen Anforderungen überprüft.

Im Rahmen dieses Projekts verzichten wir auf einige Aspekte, die aber bei größeren Projekten eine Rolle spielen, wie die Erstellung eines Pflichtenheftes, eines Benutzerhandbuchs oder gar eines Installationshandbuchs. Es reicht aus, die modulare Aufteilung des Projekts zu beschreiben, die Modulschnittstellen darzulegen, sowie eine ausführliche Kommentierung der implementierten Funktionen unter Nutzung der Literate-Skripten vorzunehmen. Hilfreich sind Hinweise für den Benutzer, wie sie das Programm aufzurufen hat und wie sie die Funktionen des Programms aktivieren kann.

4.1 Definitionsphase

In der Definitionsphase wird ein Gedankenmodell entworfen, welches auf Vollständigkeit und Konsistenz geprüft werden muss. Das Prinzip der schrittweisen Verfeinerung bildet dafür die Grundlage. Sinnvoll ist die Erstellung von Datenflußdiagrammen, die in geeigneter Weise hierarchisch gegliedert werden können. Hieraus lassen sich dann funktionelle Anforderungen an das Produkt ableiten, wobei die Wirkung jeder einzelnen Funktion und die Angabe des Ergebnisses ihrer Anwendung in verbaler Form beschrieben wird.

Die Kommunikationspfade zwischen dem Produkt und dem Benutzer werden in der Benutzerschnittstelle festgelegt. Die Charakterisierung umfasst die Art und Weise, wie und mit welchen Ein-/Ausgabegeräten die Kommunikation mit dem Benutzer abläuft, z.B. über Menüs, Kommandosprachen, Abfragesequenzen, Bildschirmmasken usw.

Man beachte, dass in der umgangssprachlichen Produktbeschreibung, die den Ausgangspunkt für die Definitionsphase bildet, oftmals Dinge nur ungenau oder unvollständig formuliert sind (z.B. sollen bei der Deinstallation von Paketen auch alle abhängigen Pakete deinstalliert werden) bzw. überhaupt nicht angesprochen werden (z.B. was hat es mit der Kategorie `virtual` auf sich).

4.2 Entwurfsphase

Der Entwurf der Systemarchitektur umfasst drei Schritte:

Modularisierung. Die Zerlegung der Gesamtaufgabe in Module basiert auf den Prinzipien der funktionalen Abstraktion und der Datenabstraktion. Eine funktionale Abstraktion stellt eine Operationsform dar, in der von konkreten Beispieloperationen auf allgemeine Funktionsformen verallgemeinert wird. Datenabstraktionen führen zur Erstellung von abstrakten Datentypen. Bei einem abstrakten Datentyp wird nicht mehr ein einzelnes Datenobjekt, sondern eine ganze Klasse von Datenobjekten samt verfügbaren Operationen definiert.

Der Vorteil der Abstraktion liegt in der Reduzierung der Komplexität der Aufgabenstellung. Zudem wird eine hohe Kontextunabhängigkeit, eine geringe Fehleranfälligkeit bei Änderungen und ein hoher Grad der Wiederverwendbarkeit erreicht. Das Ergebnis der Modularisierung ist eine Menge von Modulen zusammen mit einer Liste der gewünschten Operationen, die jedes einzelne Modul enthalten soll.

Festlegen von Operationen. In diesem Schritt werden die Modulbeschreibungen um die Beschreibungen der einzelnen Operationen erweitert, jede Operation erhält eine Liste der gewünschten Parameter.

Bestimmen der Benutzt-Relation. Ziel dieses Schrittes ist die Dokumentation, welches Modul auf welche Operationen anderer Module zugreift. Die Einbettung eines Moduls in seinen Kontext erfolgt über die von einem Modul exportierten Operationen, die Kontextabhängigkeit

hingegen ist durch die importierten Operationen bestimmt. Um einen schnellen Überblick über die Modulabhängigkeiten zu erlangen, wird die Benutzt-Relation in einem Diagramm veranschaulicht.

Haskell erlaubt eine rechnergestützte Schnittstellen-Spezifikation mit sogenannten Platzhalter-Typen (placeholder types). Die Module enthalten zunächst nur „unbesetzte“ Typen (z.B. `data Graph ohne rechte Seite`), die aus der Phase der Datenabstraktion hervorgegangen sind. Diese Typen können für die Spezifikation der Funktionen, die ein Modul anbietet, herangezogen werden. Die Funktionen werden dabei einfach durch `f = undefined` definiert. Auf diese Weise kann der Haskell-Compiler zur Konsistenzüberprüfung einer Schnittstellenbeschreibung herangezogen werden. Erst in einer späteren Entwicklungsphase werden die Platzhalterttypen durch ein Typsynonym, einen algebraischen oder einen abstrakten Datentyp ersetzt.

4.3 Implementierungsphase

Die Implementierung gliedert sich in mehrere Phasen:

1. Der Entwurf eines Moduls wird unabhängig von der Bearbeitung anderer Module durchgeführt und bestimmt die Struktur der modulinternen Realisierung, wozu modulinterne Typdefinitionen und die Angaben für die „versteckten“ Operationen gehören.
2. Die Kodierung eines Moduls beinhaltet die Umsetzung von Algorithmen in die Notation der Programmiersprache.
3. Der Test eines Moduls dient der Feststellung der funktionalen Korrektheit der von einem Modul bereitgestellten Funktionen. Bei Fehlern werden die Schritte „Modulentwurf“ und „Modulkodierung“ erneut durchlaufen.

Es hat sich erwiesen, dass die Qualität der erstellten Software größer ist, wenn mehrere Teilaufgaben von mehreren Teilnehmern nach dem Rotationsprinzip vergeben werden.

	Aufgabe 1	Aufgabe 2	Aufgabe 3
Entwurf	Mitglied A	Mitglied B	Mitglied C
Kodierung	Mitglied C	Mitglied A	Mitglied B
Test	Mitglied B	Mitglied C	Mitglied A

In der anschließenden Integration werden die einzelnen Module zusammengesetzt und auf ihre Ausführbarkeit überprüft.

4.4 Testphase

Das fertiggestellte Produkt muss geeigneten Testläufen unterzogen werden. Zu beachten ist dabei, dass aufgrund der begrenzten Auswahl von Testdaten aus fehlerlosen Testläufen nicht auf die vollständige Korrektheit eines Programms geschlossen werden kann.

Die Vorbereitung für einen Test umfasst die Auswahl der Eingaben, die Ermittlung der Sollergebnisse und die Erstellung einer geeigneten Testumgebung, d.h. die Vorbereitung des Programms für den Test.

Als Eingaben sollten nicht nur Normalwerte, sondern auch Grenzwerte und Fehlerwerte verwendet werden, um das Verhalten des Programms unter möglichst vielen Bedingungen überprüfen zu können.