Schreiben eigener R-Pakete

Dr. Matthias Kohl
Lehrstuhl für Stochastik

19. Juni 2008

Inhaltsverzeichnis

1 Einleitung

Wir beschreiben in diesem Dokument, wie man in R eigene Pakete erstellen kann und was dabei zu beachten ist. Dieses Dokument basiert auf dem Manual "Writing R Extensions" [3] - R Version 2.7.0 (2008-04-22). Ähnliche und z.T. zusätzliche Informationen finden Sie auch im Abschnitt 8.2 von [4], wobei dieser Abschnitt jedoch auf einer älteren R Version basiert und sich inzwischen verschiedene Kleinigkeiten verändert haben.

Um das Erstellen von Paketen zu vereinfachen, steht die Funktion `package.skeleton` aus dem Paket *utils* zur Verfügung. Zum Einstieg wollen wir dies an einem einfachen Beispiel demonstrieren. Wir haben verschiedene Dateien mit R-Code (Funktionen, S3-Methoden, S4-Klassen und Methoden) in einem Verzeichnis `RFunktionen` abgespeichert und wollen nun aus diesen Dateien ein Paket mit dem Namen *RKursPackage* erzeugen. Hierfür starten wir R und verwenden das Verzeichnis `RFunktionen` als Arbeitsverzeichnis; vgl. `setwd`. Anschließend rufen wir folgenden R-Code auf und erzeugen damit eine Verzeichnisstruktur, welche einem R Paket mit NAMESPACE entspricht.

```
R > fnames <- list.files()
R > package.skeleton(name = "RKursPackage", ## Name des Paketes
+                    namespace = TRUE,  ## mit einem NAMESPACE?
+                    path = "../", ## Verzeichnis in dem das Paket liegen soll
+                    code_files = fnames) ## Dateien mit R-Code
```

Alternativ zu einem Vektor mit Dateinamen kann man auch ein `environment` angeben, welches die R Objekte enthält, aus denen das Paket erzeugt werden soll bzw. eine explizite Liste (Argument: `list`) mit den R Objekten angeben.

Was leider nicht zu funktionieren scheint (zumindest nicht unter R version 2.7.1 RC (2008-06-16 r45948) und R version 2.8.0 Under development (unstable) (2008-06-18 r45948)) – Bug? – ist, dass die R-Dateien, die das Paket ausmachen sollen, in unterschiedlichen Verzeichnissen liegen und man bei `code_files` nicht nur Dateinamen, sondern vollständige Pfade eingeben kann.

Im vorliegenden Fall legt die Funktion `package.skeleton` im Verzeichnis `../` (d.h., oberhalb des aktuellen Arbeitsverzeichnisses) einen Ordner mit Namen `RKursPackage` an, der die notwendigen Ordner und Dateien für das R-Paket enthält. Es wird außerdem eine Datei `Read-and-delete-me` mit folgendem Inhalt erzeugt:

```
* Edit the help file skeletons in 'man', possibly combining help files
  for multiple functions.
* Edit the exports in 'NAMESPACE', and add necessary imports.
* Put any C/C++/Fortran code in 'src'.
* If you have compiled code, add a useDynLib() directive to
  'NAMESPACE'.
* Run R CMD build to build the package tarball.
* Run R CMD check to check the package tarball.

Read "Writing R Extensions" for more information.
```

Wir wollen im Folgenden nun genauer auf die Einzelheiten beim Erzeugen von Paketen eingehen. R bietet auch die Möglichkeit ein Bündel von Paketen zusammen ein einem sog. bundle weiterzugeben. Hierauf wollen wir aber nicht weiter eingehen, sondern verweisen lediglich auf Abschnitt 1.1.4 von [3].

2 Paket-Struktur

Das Paket *RKursPackage* entsteht aus allen Informationen, welche sich in dem Verzeichnis mit dem gleichen Namen befinden. Dabei muss jedoch eine gewisse, fixe Struktur beachtet werden. Das Verzeichnis enthält bzw. kann die folgenden Dateien enthalten:

- `DESCRIPTION`: Beschreibung des Paketes; siehe Abschnitt 3.
- `INDEX`: Diese Datei wird i.d.R. nicht explizit angegeben, sondern wird bei der Erzeugung des R Paketes automatisch generiert. Sie enthält eine Beschreibung der (interessanten) R Objekte des Paketes in Form von Name des Objekts und Beschreibung.
- `NAMESPACE`: Enthält die Namen der R Objekte, die vom Paket exportiert werden sollen sowie die Namen von R Objekten, die aus anderen Paketen bzw. Paket name spaces importiert werden sollen. Weitere Einzelheiten finden sich in Abschnitt 4
- `configure`: Falls es nötig ist, dass vor der Installation des Paketes gewisse systemabhängige Konfigurationen durchgeführt werden müssen, so kann man hierfür die Datei `configure` mit einem entsprechenden (shell) Skript in das Paket integrieren. Dieses Skript wird dann vor der Installation des Paketes ausgeführt. Für weitere Einzelheiten siehe Abschnitt 1.2 in [3]. Die Initialisierung kann aber in gewissem Umfang auch mittels R-Dateien erreicht werden; vgl. Abschnitt 1.6.3 in [3].
- `cleanup`: Diese Datei dient dazu im Anschluss an die Installation aufzuräumen. Sie enthält also z.B. ein (shell) Skript, welche alle Dateien, die mittels `configure` angelegt wurden, wieder entfernt. Für weitere Einzelheiten siehe Abschnitt 1.2 in [3]. Das Aufräumen kann aber in gewissem Umfang auch mittels R-Dateien erreicht werden; vgl. Abschnitt 1.6.3 in [3].
- `LICENSE`: Enthält die Lizenzbestimmungen zum Paket. Im Fall einer GNU public license sollte darauf verzichtet werden bzw. lediglich ein Hinweis auf die Linzenzbestimmungen von R eingefügt werden.
- `LICENCE`: alternativ zu `LICENSE`.
- `COPYING`: alternativ zu `LICENSE`.
- `NEWS`: (seit R 2.7.0) Enthält ergänzende Neuigkeiten zum Paket. Welche Dinge haben sich im Vergleich zur alten Version verändert. Wird diese Datei konsequent

gepflegt, erhält man eine knappen Überblick über die Historie eines Paketes. Generell sollte die Struktur den Vorgaben des GNU Projektes (vgl. http://www.gnu.org/prep/standards/standards.html#Documentation) entsprechen.

Dateien mit anderen Namen wie etwa `README` oder `ChangeLog` werden von R ignoriert, können aber für den Benutzer des Paketes durchaus von Interesse sein. Neben diesen Dateien enthält bzw. kann der Ordner `RKursPackage` die folgenden Unterordner enthalten:

- `R`: In diesem Verzeichnis finden sich ausschließlich Dateien mit R Code. Die Dateinamen müssen mit einem ASCII-Buchstaben oder Ziffer starten und die Datei muss eine der Endungen `.R`, `.S`, `.q`, `.r` oder `.s` haben. Es wird empfohlen die Endung `.R` zu verwenden. Generell müssen die Dateien R Code enthalten, der mit `source` eingelesen werden kann.
  Es gibt zudem zwei Ausnahmen, was dieses Verzeichnis betrifft. Es kann zum Einen eine Data `sysdata.rda` enthalten, welches ein gespeichertes Image von R Objekten enthält, die nicht direkt vom Benutzer des Paketes verwendet werden. Zum Anderen kann es Dateien mit der Endung `.in` enthalten, welche mit Hilfe des `configure` Skripts in geeignete Dateien umgewandelt werden. Eine Initialisierung und ein Aufräumen kann aber in gewissem Umfang auch mittels R-Dateien erreicht werden; vgl. Abschnitt 1.6.3 in [3]. Darüber hinaus kann dieses Verzeichnis die Unterverzeichnisse `unix` und `windows` besitzen, in denen für das jeweilige Betriebssystem angepasste Dateien enthalten sind. Weitere Einzelheiten finden sich im Abschnitt 1.1.3 von [3].

- `man`: Dieses Verzeichnis enthält die Dokumentationsdateien für alle R Objekte, die für den Benutzer des Paketes zur Verfügung gestellt werden. Der Name der Datei muss mit einem ASCII-Buchstaben oder Ziffer starten und die Datei muss entweder die Endung `.Rd` (default) oder `.rd` haben. Objekte, die nicht für den Nutzer gedacht sind, sollten einer Datei `pkg-internal.Rd` dokumentiert werden bzw. in einem name space (siehe Abschnitt 4) versteckt werden. Weitere Einzelheiten finden Sie in Abschnitt 5 bzw. in den Abschnitten 1.1.3 und 2 von [3].

- `src`: Die Sourcen und Header-Dateien von kompiliertem Code und evtl. einem `Makevars` oder `Makefile` müssen in diesem Verzeichnis abgelegt werden. Es gibt Unterstützung für C, C++, FORTRAN 77, Fortran 9x, Objective C and Objective C++ mit entsprechenden Dateiendungen `.c`, `.cc` oder `.cpp` oder `.C`, `.f`, `.f90` oder `.f95`, `.m` und `.mm` oder `.M`. Für die Header-Dateien sollte einheitlich die Endung `.h` verwendet werden. Weitere Einzelheiten finden Sie in den Abschnitten 1.1.3 und 1.2 von [3].

- `data`: In diesem Ordern befinden sich zusätzlich Datenfiles, welche das Paket zum Laden mit `data` bereitstellt. Aktuell werden drei verschiedene Typen von Dateien unterstützt - reiner R-Code (`.R` bzw. `.r`), Tabellen (`.tab`, `.txt` oder `.csv`) sowie gespeicherte Images (`.Rdata` bzw. `.rda`). Falls man Images verwenden möchte, sollte man diese mittels der Funktion `save` unter Verwendung der Option `compress`

`= TRUE` generieren. Wird R-Code zur Verfügung gestellt, so sollte dieser ohne die Funktionalität des Paketes lauffähig sein, so dass die Daten auch verwendet werden können ohne, dass das Paket geladen werden muss. Bei vielen bzw. großen Datenfiles kann die Installation beschleunigt werden, indem man zusätzlich eine Datei `datalist` in diesem Verzeichnis ablegt, welche für jeden Datensatz eine Zeile Dokumentation enthält. Für weitere Einzelheiten siehe auch Abschnitt 1.1.3 von [3].

- `demo`: Dieses Verzeichnis it für R Skripten (`.R` bzw. `.r`), welche die Funktionalität des Paketes demonstrieren. Neben diesen Dateien muss das Verzeichnis eine Datei `00Index` enthalten, welche den Namen und die Beschreibung für jedes Demo enthält (jeweils eine Zeile). Für weitere Einzelheiten siehe auch Abschnitt 1.1.3 von [3].

- `tests`: In diesem Verzeichnis kann Paket spezifischer Test-Code abgelegt werden. Möglich sind die Dateiendungen `.R` und `.Rin`, wobei die Datei `.Rin` ein entsprechendes File `.R` erzeugt, welches den Test-Code enthält. Die Ergebnisse der Tests werden in eine Datei mit gleichem Namen und Endung `.Rout` abgelegt. Falls es eine Datei `.Rout.save` gibt, so werden die Ergebnisse in `.Rout` mit den Ergebnissen in `.Rout.save` abgegelichen und Unterschiede gemeldet.

- `inst`: Der Inhalt von diesem Verzeichnist wird rekursiv in das Installationsverzeichnis des R Paketes kopiert. Die Namen der Unterverzeichnisse von `inst` sollten sich von den von R verwendeten und erzeugten Verzeichnissen `R`, `data`, `demo`, `exec`, `libs`, `man`, `help`, `html`, `latex`, `R-ex`, `chtml` und `Meta` unterscheiden. Mit Ausnahme der bereits erwähnten Files INDEX, LICENSE/LICENCE, COPYING und NEWS werden Dateien, die sich im Hauptverzeichnis des Paketes befinden nicht mit installiert. D.h., sollte man für den Nutzer weitere Dateien zur Verfügung stellen wollen, so sollte man diese im Verzeichnis `inst` ablegen. Insbesondere eine Datei CITATION, welche dann mit Hilfe der Funktion `citation` aufgerufen werden kann.

- `exec`: In diesem Verzeichnis können ausführbare Dateien abgelegt werden, welche das Paket benötigt. Typischerweise sind dies Skripten für Interpreter wie etwa shell, Perl oder Tcl. Achtung: Dies wird bisher nur von wenigen Paketen verwendet und muss daher noch als nicht ausgereift angesehen werden!

- `po`: Dient zur Internationalisierung von R und enthält die Übersetzungen von R und C Fehlermeldungen. Weitere Einzelheiten finden sich im Abschnitt 1.9 von [3].

Im Minimalfall enthält der Paketordner die DESCRIPTION Datei und die Unterverzeichnisse `R` und `man` bzw. im Fall eines Datenpaketes `data` und `man`. Ausnahmen sind Frontend- bzw. Translation-Pakete; vgl. Abschnitt 1.10 von [3].

3 DESCRIPTION Datei

Die DESCRIPTION Datei ist eine einfache ASCII-(Text)-Datei mit einer vorgegebenen Struktur. Im Fall des sehr umfangreichen Paketes *lattice* hat diese folgenden Inhalt.

```
R > packageDescription(pkg = "lattice")


Package: lattice
Version: 0.17-8
Date: 2008/05/16
Priority: recommended
Title: Lattice Graphics
Author: Deepayan Sarkar <deepayan.sarkar@r-project.org>
Maintainer: Deepayan Sarkar <deepayan.sarkar@r-project.org>
Description: Implementation of Trellis Graphics. See ?Lattice for a
        brief introduction
Depends: R (>= 2.5.0)
Suggests: grid
Imports: grid, grDevices, graphics, stats, utils
Enhances: chron
LazyLoad: yes
LazyData: yes
License: GPL (>= 2)
Packaged: Fri May 16 17:32:47 2008; dsarkar
Built: R 2.7.1; i686-pc-linux-gnu; 2008-06-18 15:55:58; unix

-- File: /home/btm722/RTOP/R-2-7-branch/library/lattice/Meta/package.rds
```

Die Bedeutung und Notwendigkeit der einzelnen Felder wollen wir im Folgenden genauer beschreiben. Die folgenden Felder sind zwingend erforderlich:

- `Package`: Name des Paketes. Erlaubt sind Buchstaben, Zahlen und ".". Das erste Zeichen muss ein Buchstabe sein.
- `Version`: Version des Paketes. Dabei handelt es sich um mindestens zwei ganze Zahlen, welche mit "." oder "-" voneindander getrennt sind; z.B., 1.0, 2.1, 0-2, etc.. Die kanonische Form ist x.y-z mit ganzen Zahlen x, y und z.
- `License`: Angabe der Lizenz in standardisierter Form. Möglichkeiten sind:
  1. die Standardabkürzungen:
     GPL-2, GPL-3, LGPL-2, LGPL-2.1, LGPL-3, AGPL-3, Artistic-1.0, Artistic-2.0
     vgl. http://www.r-project.org/Licenses/.
  2. Abkürzungen von freien oder offenen Softwarelizenzen möglicherweise versehen mit einer Einschränkung an die Version. Für weitere Einzelheiten verweisen wir auf Abschnitt 1.1.1 von [3]
  3. Die Zeichenketten `file LICENSE` bzw. `file LICENCE`, welche auf die entsprechende Lizenzdatei hinweisen sollen.

4. Die Zeichenkette `Unlimited`, welche bedeutet, dass es keine Einschränkungen außer durch gültige Gesetze gibt.

- `Description`: Die genaue Beschreibung des Paketes, z.B. in der Form mehrerer, vollständiger Sätze.
- `Title`: Kurz-Titel des Paketes, idealer Weise weniger als 65 Zeichen.
- `Author`: Name des Autors des Paketes.
- `Maintainer`: Der Name des Verantwortlichen (ein Verantwortlicher!) für das Paket mit einer gültigen E-Mail Adresse in spitzen Klammern; z.B. <Matthias.Kohl@uni-bayreuth.de>.

Sollte im Fall eines Feldes eine Zeile zur Beschreibung nicht genügen, so beginnt man eine neue Zeile und startet diese entweder mit einem Leer- oder Tabulatorzeichen. Die folgenden Felder sind optional, können aber durchaus nötig sein, damit ein Paket fehlerfrei installiert werden kann:

- `Date`: Datum der Veröffentlichung des Paketes. Es wird empfohlen, das Format yyyy-mm-dd zu verwenden.
- `Depends`: Diese Feld enthält eine durch Komma getrennte Liste aller Pakete, welche man für das vorliegende Paket benötigt. Neben dem Paketnamen ist es möglich zusätzlich in Klammern (kein Leerzeichen vor der Klammer!) ein Version mit einem Vergleichsoperator ("<=" oder ">=") anzugeben; z.B. distr(>= 1.9) (Leerzeichen zwischen <= und 1.9!), falls das Paket mindestens die Version 1.9 des Paketes distr benötigt. Neben Paketen und deren Versionen kann man auch Anforderungen an die R Version stellen; z.B. R(>= 2.6.0), falls das Paket mindestens die R Version 2.6.0 erfordert. Andere Erfordernisse des Paketes sollte man im Feld `SystemRequirements` oder aber in einem README File angeben.
- `Imports`: Hier werden die Pakete aufgelistet, von denen der name space importiert wird, die aber nicht "attached" werden müssen. Name spaces, auf die mittels "::" oder ":::" zugegriffen wird, müssen hier oder in `Suggests` bzw. `Enhances` angegeben werden. Wichtig ist dieses Feld vorallem im Zusammenhang mit Paketen, die S4-Klassen und Methoden verwenden.
- `Suggests`: Dieses Feld besitzt dieselbe Syntax wie das Feld `Depends` und führt solche Pakete auf, die nicht notwendigerweise benötigt werden. Das sind z.B. Pakete, die nur in einem Beispiel oder einer Vignette auftauchen oder im Funktionskörper von Funktionen geladen werden.
- `Enhances`: Hier können Pakete aufgeführt werden, die durch das vorliegende Paket erweitert werden; z.B., indem Methoden für Klassen zur Verfügung gestellt werden.
- `URL`: Hier kann eine Liste (getrennt durch Leerzeichen oder Kommata) von URLs angegeben werden; z.B. Homepage des Autors oder Homepage des Paketes.

- `Priority`: Im Fall der base und recommended Pakete ist dieses Feld auf `base` bzw. `recommended` gesetzt. Diese Prioritäten dürfen von anderen Paketen nicht verwendet werden!

- `Collate`: Mit diesem Feld – bzw. den systemspezifischen Varianten davon, also z.B. `Collate.windows` – kann man spezifizieren, in welcher Reihenfolge die R Dateien des Paketes aneinandergehängt werden sollen. Die Dateien werden hierzu als eine mit Leerzeichen getrennte Liste aufgeführt. Wichtig: Hier müssen alle (!) Dateien mit R Code aufgeführt werden, wobei Dateien in Unterverzeichnisse durch Angabe des Pfades genau zu spezifizieren sind; vgl. auch Abschnitt 1.1.1 von [3].

- `LazyLoad`: Erlaubt sind `yes`, `no`, `true` und `false`. Es wird damit gesteuert, ob der "lazy-loading" Mechanismus für R Objekte verwendet werden soll.
  Achtung: Falls das vorliegende Paket das *methods* Paket verwendet (d.h., S4-Klassen und Methoden), so sollte man hier `yes` oder `true` angeben!

- `LazyData`: Erlaubt sind `yes`, `no`, `true` und `false`. Es wird damit gesteuert, ob der "lazy-loading" Mechanismus für Datensätze verwendet werden soll.

- `ZipData`: Erlaubt sind `yes` und `no`. Damit wird gesteuert, ob unter Windows das Verzeichnis `data` gezippt werden darf oder nicht. Man sollte hier `no` setzen, falls das Paket mit einem gezippten Datenverzeichnis nicht korrekt funktioniert.

- `Encoding`: Sollte die DESCRIPTION Datei nicht vollständig ASCII sein, so muss in diesem Feld das "Encoding" (z.B., latin1, latin2 oder UTF8) angegeben werden. Achtung: Man sollte dieses Feld nur angeben, wenn man es wirklich benötigt, da dies sonst die Portabilität des Paketes einschränkt! Weitere Einzelheiten finden sich im Abschnitt 1.7 von [3].

- `Type`: Zulässige Typen sind `Package` (default), `Frontend` und `Translation`. Für Einzelheiten siehe Abschnitt 1.10 von [3].

- `Packaged`: Wird automatisch erzeugt und soll nicht per Hand gesetzt werden.

- `Built`: Wird automatisch erzeugt und soll nicht per Hand gesetzt werden.

4 NAMESPACE

Das Konzept des Namensraums (name space) ist mittlerweile weit verbreitet; vgl. http://de.wikipedia.org/wiki/Namensraum bzw. http://en.wikipedia.org/wiki/Namespace_(computer_science).
R besitzt seit einigen Versionen auch ein sog. name space Management System. Dies ermöglicht es dem Entwickler eines Paketes genau anzugeben, welche Variablen seines Paketes exportiert werden sollen und damit für den Benutzer zur Verfügung stehen bzw. sichtbar sind und welche nicht. Außerdem kann er angeben, welche Variablen von einem anderen Paket importiert werden sollen.

Man erzeugt für ein R Paket einen Namensraum, indem man ein File mit dem Namen NAMESPACE im Hauptverzeichnis des Paketes anlegt. In dieser Datei finden sich die Anweisungen, welche Variablen importiert und welche exportiert werden sollen. Darüber hinaus wird das Laden gemeinsame genutzte Objekte (shared objects) aufgelistet und S3-Methoden sowie S4-Klassen und Methoden werden registriert.
Achtung: Auch wenn der Inhalt der NAMESPACE Datei eine gewisse Ähnlichkeit mit R Code hat, wird dieser jedoch nicht wie R Code verarbeitet. Es sind lediglich einfache bedingte Anweisungen (if Aussagen) implementiert.
Wie herkömmliche Pakete werden Pakete mit einem Namensraum durch den Aufruf von `library` geladen und zum Suchpfad hinzugefügt. Jedoch werden nur die exportierten Variablen in den hinzugefügten Rahmen (frame) eingefügt. Lädt man ein Paket, welches Variablen eines anderen Paketes importiert, so führt dies dazu, dass auch diese andere Paket geladen wird (falls es nicht bereits geladen war). Jedoch wird dieses Paket durch dieses implizite Laden nicht zum Suchpfad hinzugefügt.
Ein Namensraum ist versiegelt sobald er geladen ist; d.h., Importe und Exporte sowie Bindungen zwischen internen Variablen können nicht verändert werden. Der Grund für diese Versiegelung ist eine einfachere Implementation, es ermöglicht die Analyse von Code und erlaubt Kompilationswerkzeugen, die Definition von globalen Variablen in einem Funktionskörper genau zu identifizieren.
Achtung: Das Hinzufügen von einem Namensraum zu einem Paket verändert die Suchstrategie. Der Namensraum des Pakets kommt an erster Stelle, dann kommen die Imports, dann der *base* Namensraum und schließlich der normale Suchpfad.
Wir wollen hier nicht näher auf sog. “load hooks” eingehen. Für Einzelheiten verweisen wir auf Abschnitt 1.6.3 von [3] sowie auf die Hilfeseite für `.onLoad`.

4.1 Imports und Exports

Mittels der Angabe von `export` wird innerhalb der Namensraum-Datei angezeigt, welche Variablen exportiert werden. Es empfiehlt sich die Namen der Variablen in Anführungszeichen zu setzen. Eigentlich ist dies aber nur für reservierte Worte und nicht-standard Namen notwendig. Um die Variablen `X` und `Y` zu exportieren schreiben wir also

```
export("X", "Y")
```

Es kann natürlich vorkommen, dass ein Paket über sehr viele Variablen verfügt. In diesem Fall kann man auch reguläre Ausdrücke zusammen mit `exportPattern` für den Export verwenden. So exportiert

```
exportPattern("^[^\\.]")
```

alle Variablen, die nicht mit “.” beginnen.
Ein Paket mit einem Namensraum importiert automatisch den *base* Namensraum. Mit Hilfe von `import` kann man alle exportierten Variablen eines angegebenen Paketes importieren. So führt

```
import("stats", "graphics")
```

zum Beispiel dazu, dass alle Variablen aus den Paketen *stats* und *graphics* importiert werden. Benötigt man hingegen nur spezielle Variablen so kann man stattdessen `importFrom` verwenden. Mit

```
importFrom("graphics", "plot")
```

importiert man also lediglich die Funktion `plot` aus dem Paket *graphics*.
Man kann auch Variablen exportieren, die von einem anderen Namensraum importiert wurden. Solche einzelnen Importe kann man umgehen, indem man eine Variable im Code mit `::` (falls exportiert) bzw. `:::` (falls nicht exportiert) vollständig referenziert. Dieser Ansatz ist aber üblicherweise nicht zu empfehlen.

4.2 Registrierung von S3 Methoden

Das Dispatching von S3-Methoden kann fehlschlagen, falls eine Methode in einem Paket definiert wurde, welches lediglich importiert aber nicht zum Suchpfad hinzugefügt wurde. Um sicherzustellen, dass diese Methoden zur Verfügung stehen, sollte das Paket, welches die Methode definiert, dafür sorgen, dass die generische Funktion importiert wird und dass die konkrete Methode mittels `S3method` registriert wird. Angenommen wir implementieren eine neue `print` Methode für eine Klasse *neueKlasse*, so kann diese neue Methode mittels

```
S3method("print", "neueKlasse")
```

registriert werden und steht dann für das Dispatching zur Verfügung. Die Funktion `print.neueKlasse` muss dabei nicht selber exportiert werden. Im Fall von `print`, welche zu *base* gehört, muss die generische Funktion nicht explizit importiert werden, da der Namensraum von *base* automatisch importiert wird. Dieser Mechanismus ist speziell für die Verwendung mit generischen Funktionen gedacht, welche in einem Namensraum definiert sind. Alle Methoden für eine generische Funktion, die in einem Paket ohne Namensraum definiert ist, sollten exportiert werden und das Paket, welches die Methoden definiert und exportiert sollte zum Suchpfad hinzugefügt werden, damit die Methoden gefunden werden.

4.3 Registrierung von S4 Klassen und Methoden

Man beachte, dass bei einem Paket, welches S4-Klassen und Methoden verwendet, das Paket *methods* im Feld `Depends` der Datei DESCRIPTION angegeben ist.
Formale S4-Klassen müssen mit Hilfe von `exportClasses` registriert werden. Mit

```
exportClasses("neueS4Klasse")
```

exportiert man die S4-Klasse *neueS4Klasse*. Analog verhält es sich mit S4-Methoden. Sollte die zugehörige generische Funktion aus einem anderen Paket (außer *base*) stammen, so muss diese zusätzlich importiert werden. Generell verhält es sich so, dass beim Export von S4-Methoden auch die zugehörige generische Funktion exportiert wird und

umgekehrt werden mit dem Export der generischen Funktion auch die S4-Methoden exportiert. Für diesen Fall, dass sowohl eine generische Funktion also auch entsprechende Methoden definiert wurden, lautet die Empfehlung `exportMethods` und nicht `export` zu verwenden; d.h.,

```
exportMethods("neueGenerischeFunktion")
```

Umgekehrt ist es nötig S4-Klassen und Methoden aus anderen Paketen, die man verwenden möchte zu importieren. Dies geschieht über `importClassesFrom` bzw. `importMethodsFrom`. Durch die Verwendung von `importMethodsFrom` werden automatisch auch die zugehörigen generischen Funktionen importiert. Im Fall, dass der vollständige Namensraum eines Paketes importiert wird, werden automatisch auch die S4-Klassen dieses Namensraums mit importiert. Es werden auch die S4-Methoden mit importiert. Um aber sicherzugehen, dass dies fehlerfrei funktioniert – insbesondere, falls man Methoden aus mehreren Namensräumen importiert – empfiehlt es sich Methoden explizit zu importieren.

5 Rd Dateien

Vergleiche Abschnitt 8.2.6 in [4] sowie Abschnitt 2 in [3].

6 Vignetten

Zusätzlich zu den Hilfeseiten in Form der `.Rd` Dateien ist es möglich, Dokumentation in beliebigen Formaten in einem Paket zur Verfügung zu stellen. Aufgrund von Portierbarkeit empfiehlt es sich aber das PDF Format zu wählen. Standardmäßig sollten diese Dokument im Unterordner `inst/doc` abgelegt werden, welcher bei der Installation des Paketes zum Unterordner `doc` wird.
Ein Spezialfall sind Dokumente im Sweave format [1], welche auch Vignetten genannt werden. Sweave erlaubt die Integration von LaTeX und R Code und ist im Paket *utils* enthalten, welches Bestandteil der Grundinstallation von R ist. Vignetten, die sich im Verzeichnis `inst/doc` befinden, werden beim Überprüfen eines Paketes mitüberprüft und es werden automatisch PDF Dokumente daraus erzeugt, welche bei der Installation des Paketes zur Verfügung stehen.
Durch Angabe von

```
\VignetteIndexEntry
```

in den Sweave-Dateien – am besten innerhalb eines LaTeX-Kommentars – wird bei der Installation automatisch ein Index über die Vignetten angelegt.
Für weitere Einzelheiten verweisen wir auf Abschnitt 8.2.6(j) von [4], Abschnitt 1.4 von [3] sowie auf die Hilfeseite zu `Sweave`.

7 Einreichen bei CRAN

Bei CRAN (Comprehensive R Archive Network) handelt sich um ein Netzwerk von www-Seiten, welche die R Distributionen und allen beigesteuerten Code enthalten. Insbesondere natürlich die R Pakete.
Vor der Übermittlung eines Paketes sollte man die folgenden Schritte durchführen, um sicherzustellen, dass das Paket vollständig ist und sich korrekt installieren läßt.

1. Führe `R CMD build NameMeinesPaketes` aus. Damit wird eine `.tar.gz` Datei erzeugt, welche die Sourcen enthält.
2. Führe `R CMD check NameMeinesPaketes` aus. Durch diesen Aufruf werden verschiedene Kontrollen durchgeführt; vgl. Abschnitt 1.3 von [3]
3. Führe `R CMD INSTALL NameMeinesPaketes` aus, welches das Paket installiert.

Diese Schritte sollten ohne Warnung oder Fehlermeldung durchlaufen. Nur dann wird das Paket auf CRAN akzeptiert. Ist dies der Fall, so lädt man die `.tar.gz` Datei auf den incoming-Bereich von CRAN, indem man sich mit "anonymous" als Benutzername und E-Mail Adresse als Passwort anmeldet bei `ftp://CRAN.R-project.org/incoming/`. Nachdem dies erfolgt ist, verbleibt als letzter Schritt, noch eine Nachricht an `mailto:CRAN@R-project.org` zu senden, in der man mitteilt, dass man ein neues Paket auf den incoming-Bereich von CRAN übermittelt hat. Die Administratoren von CRAN werden daraufhin die obigen Schritte nochmals für das Paket wiederholen und falls es keine Warnungen oder Fehler gibt, das Paket in den offiziellen Bereich von CRAN (`http://cran.r-project.org/web/packages/index.html`) überspielen. Bei Warnungen oder Fehlern wird man entsprechend zur Nachbesserung aufgefordert.

8 Schreiben eigener Pakete unter Windows

Vergleiche Abschnitt 8.2.12 von [4].

Literatur

[1] Friedrich Leisch (2002). Dynamic generation of statistical reports using literate data analysis. In W. Haerdle and B. Roenz, editors, Compstat 2002 - Proceedings in Computational Statistics, pages 575-580. Physika Verlag, Heidelberg, Germany. ISBN 3-7908-1517-9. 11

[2] R Development Core Team (2008). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.R-project.org`

[3] R Development Core Team (2008). Writing R Extensions. ISBN 3-900051-11-9 URL `http://cran.r-project.org/doc/manuals/R-exts.html` 2, 3, 4, 5, 6, 8, 9, 11, 12

[4] Peter Ruckdeschel and Matthias Kohl (2006). R/S-Plus für Einsteiger und Fortgeschrittene - ein Kurs über zwei Semester. http://www.stamats.de/RKurs.pdf 2, 11, 12