Implementierung kleiner Datenbanken unter UNIX II

WS 1999/2000

Andreas Franz Borchert
Universität Ulm

4. Februar 2000

Einführung

- Larry Wall begann 1987 mit der Entwicklung von Perl aus Frust über die Unzulänglichkeiten von *awk*.

- Perl ist wahlweise eine Abkürzung von "Practical Extraction and Report Language" oder von "Pathologically Eclectic Rubbish Lister".

- Larry Wall wurde schon vorher bekannt als Autor von *patch* und *rn* (ein Stammvater diverser News-Reader).

- Große Popularität und Verbreitung innerhalb der UNIX-Gemeinde fand perl4 (letzte Version: 4.036)

- Ende 1994 kam Perl in der Version 5.000 heraus, die nicht nur viele Erweiterungen mit sich brachte (beliebige Datenstrukturen, Modulkonzepte, OO-Features), sondern auch signifikante Aufräumarbeiten, die zur Vereinfachung der Grammatik führten.

- Die zur Zeit aktuelle stabile Version ist 5.005_03. Die aktuelle Entwicklung von Perl in Richtung 5.006 erfolgt von einem Team, dessen Zusammensetzung sich häufig ändert. Larry Wall scheint die Entwicklung im Augenblick aus gewisser Distanz zu verfolgen – an vorderster Front ist er im Augenblick nicht mehr. Darüberhinaus beteiligt sich die gesamte Netzgemeinde mit zahllosen Modulen und Ports an dem nahezu grenzenlosen Wachstum von Perl.

Nachteile der Shells

- Shell-Skripte sind praktisch, solange sie sehr kurz sind, jedoch ist es bei längeren Shell-Skripten vollkommen unmöglich, sie lesbar zu halten. Damit wird eine spätere Wartung außerordentlich erschwert.

- Shell-Skripte enthalten eine Vielzahl von Kommandos, die nicht in der Shell eingebaut sind, sondern in separaten Prozessen ausgeführt werden müssen. Dies betrifft auch einfache Arithmetik- oder String-Operationen (zumindest in der Bourne-Shell):

```
i=1
while [ $i -lt 10 ]
do ...
   i=‘expr $i + 1‘
done
```

  Entsprechend sind Shell-Skripte furchtbar langsam und ressourcenfressend.

- Durch die Vielzahl externer Kommandos sind Shell-Skripte fast immer völlig inportabel – sogar bei verschiedenen UNIX-Releases des gleichen Herstellers. Man denke nur an den Übergang von SunOS 4.1.x zu Solaris 2.x

- “Verwende Perl. Shell will man koennen, dann aber nicht verwenden.”
  Kristian Köhntopp, de.comp.os.unix.misc

Nachteile von *awk*

- *awk* ist wundervoll einfach, wenn eine Datei auf einfache Weise einmal bearbeitet wird, um dann z.B. einen Report zu erstellen.

- Was vom "Standardmodell" von *awk* abweicht, ist nur umständlich zu formulieren und scheitert u.U. auch an Restriktionen – ältere *awk*-Versionen ließen z.B. nur 10 offene Dateien (insgesamt) zu.

- *awk* unterstützt nur einfache Prozeduren, aber keine Techniken, die für größere Projekte erforderlich sind.

Vorteile von Perl

- Perl und Unmengen zugehöriger Module sind freie Software, die der GNU General Public License oder alternativ der Artistic License von Larry Wall unterliegen.

- Perl läuft auf einer Vielzahl von Plattformen und ist portabler als andere – insbesondere portabler als C und erst recht portabler als andere UNIX-Werkzeuge.

- Perl bietet viele Strukturierungsmechanismen, die für große Projekte notwendig sind:

  - Modularisierung

  - OO-Techniken

  - Beliebige Datenstrukturen

  - Zahlreiche Überprüfungen, die nicht nur über die anderer Skript-Programmiersprachen hinausgehen, sondern insbesondere auch über die von C und C++.

Vorteile von Perl

- Perl vereinigt fast den gesamten UNIX-Werkzeugkasten in zumeist sehr vertrauter Form. Dazu gehören insbesondere
  - Bourne-Shell
  - awk
  - sed
  - tr
  - sort
- Perl bietet reguläre Ausdrücke in ansonsten unerreichter Luxus-Fassung an (aufwärtskompatibel zu den sonst unter UNIX gewohnten regulären Ausdrücken) – wenngleich sie nicht mehr im strengen Sinne regulär sind.
- Der Zugriff auf alle Systemaufrufe ist auf direkte (und weitgehend portable Weise) möglich. Dazu gehören z.B. auch alle gängigen Operationen für UNIX-Sockets.

Vorteile von Perl

- Perl-Module, zu denen auch beliebiger C-Code gehören kann, können dynamisch hinzugeladen werden.

- Perl ist um Größenordnungen schneller als alle anderen UNIX-Tools. Nur allerneueste Versionen von *awk* (z.B. *mawk*) können teilweise in einfachen Fällen mithalten. Das hat folgende Ursachen:

  - Perl übersetzt zuerst das gesamte Skript mitsamt allen importierten Modulen in einen internen Byte-Code, der anschließend sehr effizient ausgeführt werden kann.

  - Durch die hohe Integration von Perl müssen kaum fremde Prozesse gestartet werden.

  - Perl besitzt eine hervorragende Speicherverwaltung, so ziemlich die schnellste bekannte Implementierung für reguläre Ausdrücke und nahezu perfekte Repräsentierungen für die wichtigen Datenstrukturen (Listen und assoziative Arrays).

Nachteile von Perl

- Es ist kein Problem, in Perl Programme zu schreiben, deren Unlesbarkeit deutlich die von allerschlimmsten Shell-Skripten oder *awk*-Programmen übersteigt. Entsprechend gibt es regelmäßig den "Obsfuscated Perl Contest".

- Da Perl mehr oder weniger den gesamten UNIX-Werkzeugkasten integriert und zusätzlich zahllose Sprachfeatures besitzt, hat es einen enormen Sprachumfang, der insbesondere zu Beginn kaum zu überblicken ist.

- Im Augenblick gehört Perl nur in seltenen Fällen zur Standardauslieferung eines UNIX-Systems. Da Perl glücklicherweise aber frei ist, kann es problemlos Bestandteil eines kommerziellen Software-Pakets sein, das an Kunden weitergegeben wird.

Alternativen zu Perl

- Die Unzulänglichkeiten älterer Skript-Programmiersprachen ließen eine Reihe von neueren Ansätzen gedeihen. Perl ist hier nicht der einzige Vertreter.

- Python und S-Lang sind zwei relativ gut strukturierte und moderne Vertreter, die insbesondere (bislang) keine so chaotisch-historische Entwicklung wie Perl genommen haben.

- Für die Anhänger von Lisp gibt es interessante Scheme-Implementierungen und natürlich Emacs, der im Zweifelsfall sowieso alles kann :-)

- Tcl ist gut als Shell innerhalb einer Anwendung (analog zu den gewohnten UNIX-Shells) – jedoch katastrophal schlecht in der Effizienz und bei dynamischen Datenstrukturen (obwohl sich diesbezüglich in der letzten Zeit auch einiges getan hat).

- Allen Alternativen ist gemeinsam, daß sie weder die Effizienz, noch den Umfang an durchaus sinnvollen Operationen haben. Ganz zu schweigen von der inzwischen sehr umfangreichen Perl-Bibliothek.

Dokumentation zu Perl

perldata	Datenstrukturen von Perl
perlsyn	Syntax von Perl
perlop	Operatoren und Vorränge
perlre	Reguläre Ausdrücke in Perl
perlrun	Aufruf von Perl auf der Kommandozeile
perlfunc	Überblick aller eingebauten Funktionen in Perl (ist sehr umfangreich)
perlvar	Vordefinierte Variablen
perlsub	Prozeduren in Perl
perlmod	Module in Perl
perlref	Zeiger in Perl
perlobj	Objekte in Perl

- Perl kommt zusammen mit einer sehr guten und ausführlichen Referenz-Dokumentation. Die Dokumentation gibt es als Manualseiten, die aber auch in HTML verwandelt werden kann (sehr empfehlenswert). Inzwischen gibt es die Dokumentation auch im PDF-Format.

- Es gibt zahlreiche weitere Manualseiten zu Perl, die u.a. in tutorieller Form auf dynamische Datenstrukturen und OO-Techniken eingehen.

- Bei uns gibt es diese Seiten unter
  `http://www.mathematik.uni-ulm.de/help/perl5/doc/perl.html`
  im Web.

Bücher zu Perl

- *Programming Perl*, Larry Wall, Randal L. Schwartz, et al. O'Reilly, 2nd Edition September 1996, ISBN: 1-56592-149-6
  Dies ist das Standardwerk zu Perl, das sich auf Perl 5.003 bezieht. Die Unterschiede zu 5.005 sind im Rahmen dieser Vorlesung nicht von größerer Bedeutung.
  Wegen des Titelbilds ist es auch als "camel book" bekannt und das Kamel sind seitdem das Wappentier von Perl :-)

- *Learning Perl*, Randal L. Schwartz, O'Reilly, 2nd Edition July 1997, ISBN: 1-56592-284-0
  Sehr gute Einführung zu Perl, die auf 5.004 basiert.
  Wegen dem Titelbild ist es als "llama book" bekannt.

- *Advanced Perl Programming*, Sriram Srinivasan, O'Reilly, 1st Edition August 1997, ISBN: 1-56592-220-4
  Dieses Buch geht ein auf die Verbindung von Perl mit Datenbanken, graphischen Benutzeroberflächen und Netzwerkdiensten.

- *Effective Perl Programming*, Joseph Hall, Randal L. Schwartz, Addison-Wesley, 1st Edition 1998, ISBN: 0-201-41975-0
  Sehr gutes Tutorial zur besseren Programmierung in Perl und zu den neueren Sprachkonstrukten in Perl.

- *Perl Cookbook*, Tom Christiansen & Nathan Torkington, O'Reilly, 1st Edition August 1998, ISBN: 1-56592-243-3
  Ist mittlerweile eines der Standardwerke, das zeigt, wie zahllose immer wieder vorkommende Probleme in Perl elegant und korrekt gelöst werden können. Seinen Ursprung fand das Buch in den tutoriellen Beiträgen von Tom Christiansen zur mitgelieferten Perl-Dokumentation, die jedoch auf ein Nachschlagewerk im Umfange von über 700 Seiten ausgedehnt wurden.

Noch mehr Bücher

- *Mastering Regular Expressions*, Jeffrey E.F. Friedl, O'Reilly, 1st Edition January 1997, ISBN: 1-56592-257-3
  Ein Meisterwerk über reguläre Ausdrücke bei UNIX-Werkzeugen, wobei der Autor insbesondere auf Perl eingeht. Das Buch zählt nicht nur die Möglichkeiten auf, sondern erklärt die genaue Funktionsweise und hilft, reguläre Ausdrücke effizienter zu gestalten.

- *Learning Perl/Tk*, Nancy Walsh, O'Reilly, 1st Edition January 1999, ISBN: 1-56592-314-6
  Ein sehr ausführliches Werk über Perl/Tk, das allerdings weniger tutoriell organisiert ist, sondern sich mehr in der Weise eines Referenz-Manuals durch die Tk-Widgets und anderen Themen durchbewegt.

- *Web Client Programming with Perl*, Clinton Wong, O'Reilly, 1st Edition March 1997, ISBN: 1-56592-214-X
  Dieses Buch geht ein auf HTTP, die Socket-Schnittstelle von Perl und die sehr beliebte LWP-Bibliothek von Perl.

- *MySQL & mSQL*, Randy Jay Yarger, George Reese & Tim King, O'Reilly, 1st Edition July 1999, ISBN: 1-56592-434-7
  Ein ausführliches Werk zu MySQL, das auch auf Perl in Verbindung mit MySQL eingeht (Kapitel 10 und 21).

- Weitere Buchhinweise gibt es unter
  `http://reference.perl.com/query.cgi?books` im WWW.

Periodika und Webseiten zu Perl

- Es gibt eine Zeitschrift nur für Perl: *The Perl Journal*. Mehr dazu im WWW unter
  `http://www.tpj.com/`

- In der Zeitschrift *UNIX Review* gibt es eine regelmäßige Perl-Kolumne von Randal Schwartz. Sie wird auch im Web unter `http://www.stonehenge.com/merlyn/UnixReview/` veröffentlicht.

- Eine weitere Perl-Kolumne gibt es in der Zeitschrift *Web Techniques* ebenfalls von Randal Schwartz. Im Web unter `http://www.stonehenge.com/merlyn/WebTechniques/` zu finden.

- Im WWW gibt es natürlich einige interessante Seiten:

  - Perl-Home-Page: `http://www.perl.com/`

  - Perl Oasis: `http://www.oasis.leo.org/perl/`

  - Meine Seite: `http://www.mathematik.uni-ulm.de/help/perl5/`

Erste Schritte

first.pl

```
#!/usr/local/bin/perl -w

print "Hello world!\n";
```

- Wie bei anderen Skript-Programmiersprachen auch sorgt die erste Zeile (unter UNIX) dafür, daß der richtige Interpreter gestartet wird.

- Da Perl typischerweise nicht vom UNIX-Hersteller mitgeliefert wird, ist `/usr/local/bin/perl` die wahrscheinlichste Lokation. Allerdings gibt es beim Perl automatische Installierungsmechanismen, die von selbst dafür sorgen können, daß dieser Pfad stimmt.

- Die Option `-w` ist von besonderer Bedeutung, da sie all die vielen Warnungen von Perl einschaltet (die leider nicht per Voreinstellung zum Tragen kommen). Da fast jede Warnung ein ernsthafter Fehlerhinweis ist, sollte man so gut wie nie Perl ohne diese Option verwenden.

- Die eingebaute `print`-Anweisung ermöglicht Ausgaben (per Voreinstellung nach `STDOUT`).

Skalare Variablen

scalar.pl

```
#!/usr/local/bin/perl -w

$greeting = "Hallo Perl-Fans!";
print "$greeting\n";
```

- Vor dem Namen einer skalaren Variable wird immer ein `$` angegeben. Dies geschieht unabhängig davon, ob die Variable benutzt wird oder ob ihr etwas zugewiesen wird.

- Analog zu vielen anderen Skript-Programmiersprachen wird hier auf Variablendeklarationen verzichtet. Allerdings ergibt sich aus der Benutzung *immer* der jeweilige Typ. Hier legt der `$` fest, daß es sich um eine skalare Variable handelt.

- Analog zu C ist ein Semikolon am Ende einer Anweisung anzugeben. Im Gegensatz zu *awk* und der Bourne-Shell hat der Zeilentrenner keine Bedeutung als Trenner zwischen Anweisungen.

- Genauso wie in der Bourne-Shell können innerhalb von Zeichenketten, die in doppelte Anführungszeichen eingeschlossen sind, Variablen angegeben werden, deren Wert dann eingefügt wird.

- Analog zu ANSI-C werden innerhalb von Zeichenketten verschiedene Sonderzeichen wie z.B. `\n` unterstützt.

Eingabe

hello.pl

```
#!/usr/local/bin/perl -w

print "Name: ";
$name = <STDIN>;
print "Sei gegruesst, $name";
```

- Das Einlesen erfolgt mit dem <>-Operator, bei dem eine Dateiverbindung angegeben werden kann (hier: `STDIN`).

- Bei der Zuweisung an eine skalare Variable wird dabei genau eine Zeile eingelesen inklusive dem Zeilentrenner (was der immer sein mag).

- Da in `$name` bereits der Zeilentrenner enthalten ist, brauchen wir ihn nicht mehr bei der `print`-Anweisung mit anzugeben.

Ein paar Operationen

sum.pl

```
#!/usr/local/bin/perl -w

print "a = "; chomp($a = <STDIN>);
print "b = "; chomp($b = <STDIN>);
print "a + b = ", $a + $b, "\n";
```

- Die zum Standardumfang von Perl gehörende Prozedur `chomp` eliminiert den Zeilentrenner am Ende der übergebenen Zeichenkette.

- In Perl werden Parameter (anders als in C) via *call by reference* (soweit wie möglich) übergeben. Auf diese Weise kann `chomp` die übergebenen Variablen problemlos modifizieren.

- Anders als in C ist das Resultat eines Zuweisungsoperators wieder ein *lvalue*, d.h. er kann auf der linken Seite einer Zuweisung stehen oder hier von `chomp` weiter bearbeitet werden.

- Analog zu *awk* unterscheidet Perl bei skalaren Variablen nicht zwischen Zeichenketten und Gleitkommazahlen. Je nach Bedarf wird die Repräsentierung jeweils angepaßt.

- Bei `print` können durch Kommata getrennt eine Reihe von Argumenten angegeben werden, die alle hintereinander auszugeben sind.

Vorränge

Soweit Operatoren von ANSI-C übernommen worden sind, wurden deren Vorränge untereinander beibehalten:

left	terms and list operators (leftward)
left	`->`
nonassoc	`++ --`
right	`**`
right	`! ~ \` and unary `+` and `-`
left	`=~ !~`
left	`* / % x`
left	`+ - .`
left	`<< >>`
nonassoc	named unary operators
nonassoc	`< > <= >= lt gt le ge`
nonassoc	`== != <=> eq ne cmp`
left	`&`
left	`| ^`
left	`&&`
left	`||`
nonassoc	`..`
right	`?:`
right	`= += -= *=` etc.
left	`, =>`
nonassoc	list operators (rightward)
right	`not`
left	`and`
left	`or xor`

Hinweis: Die Vorrang-Tabelle wurde wörtlich aus der *perlop*-Manualseite übernommen.

Ein erster Filter

ulm.pl

```
#!/usr/local/bin/perl -w
while (<>) {
   print if /Ulm/;
}
```

- Wenn beim Einlese-Operator <> keine Dateiverbindung angegeben worden ist, wird entweder die Standardeingabe (`STDIN`) verwendet oder, falls Parameter angegeben worden sind, nacheinander alle angegebenen Dateien zum Lesen eröffnet.

- Wenn der Operator <> ganz alleine in einer Schleifen-Kontrollbedingung steht, wird das Ergebnis nicht weggeworfen, sondern implizit der Variablen `$_` zugewiesen.

- Beim Erreichen des Eingabe-Endes liefert <> als Resultat `undef`. Erst dies wird von `while` als FALSE interpretiert (ganz im Gegensatz zu einer leeren Zeile).

- Obwohl innerhalb der `while`-Schleife nur eine einzige Anweisung steht, sind die geschweiften Klammern notwendig (im Gegensatz zu C, wo sie nur bei mehreren Anweisungen zwingend sind).

- Die `print`-Operation gibt implizit die Variable `$_` aus, wenn keine Parameter angegeben sind.

- Hinter einem Ausdruck kann eine `if`-Bedingung folgen, die dazu führt, daß der vorstehende Ausdruck nur dann bewertet wird, wenn die Bedingung wahr ist.

- Ein regulärer Ausdruck ohne Bezug bezieht sich implizit auf die Variable `$_`.

Kurzübersicht der Kontrollstrukturen

expr1 `if` *expr2*
expr1 `unless` *expr2*
expr1 `while` *expr2*
expr1 `until` *expr2*
expr1 `foreach` *expr2*

`if (`*expr*`) {` *stmts* `}`
`[ elsif (`*expr*`) {` *stmts* `} ...]`
`[ else {` *stmts* `} ]`
`unless (`*expr*`) {` *stmts* `}`
`[ else {` *stmts* `} ]`

`[` *label* `] while (`*expr*`) {` *stmts* `} [ continue {` *stmts* `} ]`
`[` *label* `] until (`*expr*`) {` *stmts* `} [ continue {` *stmts* `} ]`
`[` *label* `] for ( [` *expr* `] ; [` *expr* `] ; [` *expr* `] ) {` *stmts* `}`
`[` *label* `] foreach [` *var* `] (` *list* `) {` *stmts* `}`
`[` *label* `] {` *stmts* `} [ continue {` *stmts* `} ]`

`goto [` *label* `]`
`last [` *label* `]`
`next [` *label* `]`
`redo [` *label* `]`

`do {` *stmts* `} while` *expr* `do {` *stmts* `} until` *expr*

Diese Tabelle wurde in etwas geänderter Form aus der Perl-Referenz von Johan Vromans übernommen.

Dateien eröffnen

addresses.pl

```
#!/usr/local/bin/perl -w

$addressbook = "addressbook";
open(BOOK, "<$addressbook") ||
   die "Cannot open $addressbook\n";
while (<BOOK>) {
   chomp;
   ($name, $address) = split /:/;
   printf "%-20s | %s\n", $name, $address;
}
close(BOOK);
```

- Der erste Parameter von `open` ist der Name der Dateiverbindung. Per Konvention werden die immer in Großbuchstaben gesetzt.

- Der zweite Parameter orientiert sich an der Syntax der Bourne-Shell. Hier wird mit < konkret die Datei `addressbook` zum Lesen eröffnet.

- Wenn `open` fehlschlägt, wird `die` aufgerufen, das nach Ausgabe der angegebenen Meldung die Ausführung des Programms beendet (mit einem Exit-Code von 2).

- Genauso wie im letzten Beispiel wird jeweils eine Zeile innerhalb der `while`-Bedingung eingelesen und an `$_` zugewiesen.

- Wenn `chomp` ohne Parameter aufgerufen wird, entfernt es den Zeilentrenner am Ende von `$_`.

Dateien eröffnen

```
juliet$ cat addressbook
Andreas Borchert:Universitaet Ulm, 89069 Ulm
Larry Wall:O'Reilly, ...
juliet$ addresses.pl
Andreas Borchert     | Universitaet Ulm, 89069 Ulm
Larry Wall           | O'Reilly, ...
juliet$
```

- `split` erhält als ersten Parameter einen regulären Ausdruck, der als Feldtrenner interpretiert wird. Fehlt der zweite Parameter, so zerlegt `split` implizit `$_`.

- `split` liefert eine Liste von Feldern zurück, die wiederum an eine Liste von Variablen zugewiesen werden kann (dazu in Kürze mehr).

- Hier landet jedenfalls das erste Feld in `$name` und das zweite Feld in `$address`. Sollte es weitere Felder gegeben haben, fallen sie unter den Tisch (das funktioniert hier also anders als die `read`-Anweisung in der Bourne-Shell).

- Alternativ zu `print` gibt es auch `printf`, das in gewohnter Weise operiert. Zu beachten ist nur, daß `%*` im Format nicht zulässig ist – stattdessen kann die jeweilige Variable, die die Breite enthält, auf gewohnte Weise interpoliert werden (also etwa `%$breite`).

Formate

fmtaddresses.pl

```
#!/usr/local/bin/perl -w

$addressbook = "addressbook";
open(BOOK, "<$addressbook") ||
   die "Cannot open $addressbook\n";
while (<BOOK>) {
   chomp;
   ($name, $address) = split /:/;
   write;
}
close(BOOK);

format STDOUT_TOP =
Name                    | Adresse
------------------------+------------------------
.

format STDOUT =
@<<<<<<<<<<<<<<<<<<<<<< | @<<<<<<<<<<<<<<<<<<<<<<
$name,                    $address
.
```

- Mit den Formaten können noch eleganter als mit `printf` Tabellen und Listings generiert werden. Sehr elegant ist dabei die Unterstützung der Seitenköpfe (wobei sich einstellen läßt, wieviel Zeilen auf eine Seite gehen).

- Die Formate bestehen aus beliebigem Text, der abgesehen von den Platzhaltern, die mit `@` beginnen, genauso übernommen wird.

Formate

```
juliet$ cat addressbook
Andreas Borchert:Universitaet Ulm, 89069 Ulm
Larry Wall:O'Reilly, ...
juliet$ fmtaddresses.pl
Name                   | Adresse
-----------------------+------------------------
Andreas Borchert       | Universitaet Ulm, 89069
Larry Wall             | O'Reilly, ...
juliet$
```

- Unterhalb einer Zeile, in der Platzhalter vorkommen, ist jeweils eine (durch Kommata getrennte) Liste von Variablen anzugeben, die die Platzhalter füllen sollen.

- Ein Format wird jeweils mit einem Punkt als einziges Zeichen einer Zeile abgeschlossen.

- Bei `write` werden die zu der angesprochenen Dateiverbindung (hier `STDOUT`) gehörenden Formate (hier `STDOUT_TOP` und `STDOUT`) aktiv.

- Wenn eine der auszugebenden Zeichenketten größer als der Platzhalter ist, wird entsprechend abgeschnitten.

Listen

- Listen haben eine sehr große Bedeutung in Perl, da

  - sie außerordentlich effizient realisiert sind (vergleichbar mit den Arrays in konventionellen Programmiersprachen),

  - sehr häufig das Ergebnis von Operationen oder Funktionen in Form einer Liste vorliegt und

  - auch Parameter in Form von Listen übergeben werden.

- Im Gegensatz zu *awk* ist es in Perl nicht nötig, assoziative Arrays für diesen Zweck zu verwenden.

Eine erste Liste

friends1.pl

```
#!/usr/local/bin/perl -w

@friends = ("Eva", "Werner", "Andreas", "Martin", "Franz");
for ($index = 0; $index <= $#friends; ++ $index) {
   print "Gruesse bitte $friends[$index] von mir!\n";
}
```

- Dem Variablennamen einer Liste geht immer ein `@` voraus, es sei denn, durch einen nachfolgenden Index wird genau ein Element herausgegriffen.

- Die Elemente von Listen (und assoziativen Arrays) sind immer Skalare. Kompliziertere Datenstrukturen werden nachher möglich auf Basis von Zeigern, die ebenfalls Skalare sind.

- Innerhalb von runden Klammern können durch Kommata getrennt beliebig viele Skalare und Listen angegeben werden, die zu einer einzigen flachen Liste werden.

- Listen können einander zugewiesen werden. Es werden dann wirklich alle Elemente kopiert und der ursprüngliche Inhalt der Zielliste geht verloren.

- Indexbereiche beginnen mit 0 und sind nach oben hin dynamisch offen. Der derzeitig höchste belegte Index von `@friends` ist `$#friends` – im Beispiel 4.

- Mit `$friends[$index]` wird entsprechend dem Index auf ein einzelnes Element zugegriffen. Zugriffe jenseits des aktuellen Indexbereiches sind zulässig und liefern `undef`.

Das Durchlaufen einer Liste

friends2.pl

```
#!/usr/local/bin/perl -w

@friends = ("Eva", "Werner", "Andreas", "Martin", "Franz");
foreach $friend (@friends) {
   print "Gruesse bitte $friend von mir!\n";
}
```

- Die Schlüsselworte `for` und `foreach` sind äquivalent. Typischerweise wird `for` für den C-Stil verwendet, während `foreach` bei der expliziten Angabe einer zu durchlaufenden Liste bevorzugt wird.

- Wenn keine Schleifenvariable angegeben wird, dann wird implizit `$_` verwendet.

- Innerhalb der Klammern kann alles stehen, was auf die eine oder andere Weise eine Liste ergibt:

friends3.pl

```
#!/usr/local/bin/perl -w

@friends = ("Eva", "Werner", "Andreas", "Martin", "Franz");
foreach $friend (@friends, "Manuela", @friends) {
   print "Gruesse bitte $friend von mir!\n";
}
```

Kontext

Manche Ausdrücke können unterschiedlich bewertet werden, je nachdem, ob sie in einem skalaren Kontext oder in einem Listen-Kontext stehen. Kontext bezieht sich hier jeweils auf den anderen Operanden (z.B. die linke Seite einer Zuweisung).

- Der Wert einer Listenvariablen ist im Listen-Kontext die gesamte Liste,

- wenn jedoch eine Listenvariable im skalaren Zusammenhang verwendet wird, so wird die Länge der Liste als Wert verwendet. Entsprechend kann die Anzahl der Freunde ausgegeben werden:

```
print "Sie haben ", @friends + 0, " Freunde\n";
```

  Das Addieren der Null erzwingt hier einen skalaren Kontext, so daß wir die Länge der Liste erhalten und nicht all die vielen Elemente der Liste. Dies geht allerdings auch etwas lesbarer und weniger trickreich mit Hilfe des `scalar`-Operators, der einen skalaren Kontext erzwingt:

```
print "Sie haben ", scalar @friends, " Freunde\n";
```

- Ein Listenkonstruktor liefert im skalaren Kontext das letzte Element der Liste.

- Bei Funktionen oder Operatoren, die potentiell eine Liste zurückliefern können, gibt es keine allgemeine Regel, was in Abhängigkeit des Kontexts zurückgeliefert wird.

Kontext

Ein weiteres Beispiel ist der Einlese-Operator <>. In den bisherigen Beispielen wurde jeweils eine Zeile eingelesen, da der Operator jeweils in einem skalaren Kontext stand:

```
$line = <STDIN>;
```

Alternativ können auch alle Zeilen auf einmal eingelesen werden:

```
@lines = <STDIN>;
```

Die erste Zeile ist dann unter `$lines[0]` erreichbar, die zweite unter `$lines[1]` usw.

Interpolation von Listen

- Listen können nur skalare Werte enthalten. Listen von Listen sind damit nicht möglich.

- Wenn dennoch Listenkonstruktoren ineinander verschachtelt werden, kommt es zur Interpolation, d.h. alle Elemente werden unabhängig von ihrer Verschachtelung alle hintereinander in einer Liste repräsentiert.

- Somit entspricht jeweils die rechte Seite der linken:

```
(1, (2, 3, (4, 5), 6), 7)   (1, 2, 3, 4, 5, 6, 7)
((), ((), ()))              ()
```

- Wenn tatsächlich verschachtelte Datenstrukturen benötigt werden, geht dies über Zeiger (die auch als skalare Werte gelten). Dies wird später vorgestellt.

Kommandozeile

echo.pl

```
#!/usr/local/bin/perl -w

print join(" ", @ARGV), "\n";
```

- Die vordefinierte Liste `@ARGV` enthält alle Parameter aus der Kommandozeile.

- Nicht in `@ARGV` enthalten ist der Kommandoname, der stattdessen in `$0` zu finden ist.

- `join` faßt beliebig viele Listenmitglieder zu einer Zeichenkette zusammen, wobei der erste Parameter als Feldtrenner verwendet wird.

- Es geht aber noch kürzer, da auch Listen in Zeichenketten direkt eingebettet werden dürfen:

  ```
  print "@ARGV\n";
  ```

  Als Feldtrenner wird hier die Variable `$"` verwendet, die per Voreinstellung genau ein Leerzeichen hat.

- Zu beachten ist, daß bei

  ```
  print @ARGV, "\n";
  ```

  keine Feldtrenner implizit eingefügt werden.

Listen im Überblick

`$var`	eine einfache skalare Variable
`@list`	eine Liste
`$#list`	höchster belegter Index der Liste
`$list[$index]`	ein Element der Liste
`@list[$i..$j]`	eine Teilliste
`push(@list, $var)`	am Ende der Liste etwas anhängen
`push(@list, @elems)`	viele Elemente auf einmal anhängen
`pop(@list)`	das letzte Element wegnehmen und zurückliefern
`unshift(@list, $var)`	an den Anfang etwas einfügen
`unshift(@list, @elems)`	viele Elemente auf einmal am Anfang einfügen
`shift(@list)`	das erste Element wegnehmen und zurückliefern

Assoziative Arrays

lookup.pl

```
#!/usr/local/bin/perl -w

$addressbook = "addressbook";
open(BOOK, "<$addressbook") ||
   die "Cannot open $addressbook\n";
%address = ();
while(<BOOK>) {
   chomp;
   ($name, $address) = split /:/;
   $address{$name} = $address;
}
close(BOOK);

while (defined($name = <STDIN>)) {
   chomp($name);
   if (exists $address{$name}) {
      print $address{$name}, "\n";
   } else {
      print "Unbekannt!\n";
   }
}
```

- Assoziative Arrays akzeptieren als Index beliebige skalare Werte (Zeiger sind jedoch nur mit Vorsicht zu verwenden).
- Dem Variablennamen eines assoziativen Arrays geht (wenn nicht ein einzelnes Element indiziert wird) ein % voraus.
- Der Index wird in geschweifte Klammern gefaßt.

`defined` und `exists`

```
juliet$ (echo 'Larry Wall';
> echo 'Franz Schweiggert') | lookup.pl
O'Reilly, ...
Unbekannt!
juliet$
```

- Perl kann immer genau zwischen wohldefinierten und undefinierten Werten unterscheiden (`undef`).
- Bei assoziativen Arrays wird sogar unterschieden, ob
  - ein Index existiert und
  - der zugehörige Wert definiert ist.

  Es ist also möglich, daß ein Index existiert und der zugehörige Wert undefiniert ist.
- Mit den Operatoren `defined` und `exists` kann das entsprechend überprüft werden:
  - `defined $var` (ist der Wert von `$var` definiert?)
  - `exists $array{$index}` (ist `$index` für `%array` definiert?)
- Der Zugriff auf undefinierte Variablen führt (bei eingeschalteter Option `-w`) zu entsprechenden Warnungen.

Assoziative Arrays und Listen

Assoziative Arrays und Listen können einander zugewiesen werden:

```
%phones = ("Hans", 3156, "Gertrud", 5467, "Martin", 2464);
```

Hier wird ein Paar aus der Liste jeweils als Index und zugehöriger Wert betrachtet. Bei einer ungeraden Anzahl von Listenelementen ist der letzte Wert undefiniert (`undef`).

Es läßt sich auch etwas lesbarer aufschreiben mit der zum Komma äquivalenten Notation =>:

```
%phones = (
   "Hans" => 3156,
   "Gertrud" => 5467,
   "Martin" => 2464,
);
```

Umgekehrt geht es natürlich auch:

```
@list = %phones;
```

Die Reihenfolge der Paare ist dann natürlich völlig undefiniert.

Das Durchlaufen assoziativer Arrays

sortaddr.pl

```
#!/usr/local/bin/perl -w

$addressbook = "addressbook";
open(BOOK, "<$addressbook") ||
   die "Cannot open $addressbook\n";
while (<BOOK>) {
   chomp;
   ($name, $address) = split /:/;
   $address{$name} = $address;
}
close(BOOK);

foreach $name (sort keys %address) {
   printf "%-20s | %s\n", $name, $address{$name};
}
```

- Mit dem Operator `keys` werden alle Schlüssel eines assoziativen Arrays (in undefinierter Reihenfolge) als Liste extrahiert. Alternativ gibt es auch `values`, das alle Werte zurückliefert.

- Der Operator `sort` sortiert eine Liste und liefert sie wieder als neue(!) Liste zurück. Das Sortierkriterium läßt sich beliebig bestimmen – dazu später mehr.

- Wenn die Reihenfolge keine Rolle spielt, ist `each` effizienter zum Durchlaufen:

  ```
  while (($name, $address) = each %address) { ... }
  ```

Assoziative Arrays im Überblick

`%array`	das gesamte assoziative Array
`$array{$index}`	ein Element des Arrays
`@array{@list}`	`($array{$list[0]}, $array{$list[1]}, ...)`
`keys %array`	alle Schlüssel eines Arrays
`values %array`	alle Werte eines Arrays
`exists $array{$index}`	gibt es den Schlüssel?
`delete $array{$index}`	einen Eintrag im Array löschen

Prozeduren

```
sub add {
   my($a, $b) = @_;
   return $a + $b;
}
```

- `add` erhält zwei Parameter, addiert sie und liefert die Summe wieder zurück.
- Ein Aufruf von `add` könnte etwa folgendermaßen aussehen:

  ```
  $c = add(17, $x);
  ```

- Grundsätzlich gibt es keine formale Parameterliste – stattdessen werden alle Parameter als eine einzige Liste betrachtet, die über `@_` innerhalb der Prozedur zugänglich ist.
- Die Parameterübergabe erfolgt via *call by reference*. Wenn einer der übergebenen Parameter kein *lvalue* ist, wird implizit eine Kopie angelegt.
- Mit Hilfe von `my` können lokale Variablen deklariert werden.
- Durch die Zuweisung von `@_` an die beiden lokalen Variablen erhalten wir die normalerweise bevorzugte Semantik von *call by value*.
- So wäre es auch kürzer gegangen, da ohne `return` der Wert des zuletzt bewerteten Ausdrucks verwendet wird:

  ```
  sub add { $_[0] + $_[1] }
  ```

Listen als Parameter

```
sub add {
   my(@values) = @_;
   my($sum) = 0;
   foreach $value (@values) {
      $sum += $value;
   }
   return $sum;
}
```

- Diese Version von `add` liefert die Summe aller übergebenen Parameter.

- Ohne formale Parameterliste gibt es auch keine Beschränkung der Anzahl der Parameter. Entsprechend einfach ist es, beliebig viele Parameter zu akzeptieren.

- Wenn beim Aufruf mehrere Listen angegeben worden sind, verschmelzen sie alle zu einer einzigen flachen Liste.

- Es ist also nicht sinnvoll, mehrere getrennte Listen zu erwarten:

  ```
  sub multilist { my(@list1, @list2) = @_; ... }
  ```

  Hier würde `@list1` alle Parameter "aufsaugen" und `@list2` würde in jedem Fall zur leeren Liste.

- Wenn sowohl skalare Parameter als auch eine Liste übergeben werden sollen, dann müssen die skalaren Parameter zuerst kommen, damit es noch möglich ist, die Parameter auseinanderzuhalten:

  ```
  sub multipar { my($p1, $p2, @list) = @_; ... }
  ```

Kontext eines Prozeduraufrufs

- Die `return`-Anweisung wird in dem Kontext des Prozeduraufrufs ausgewertet.

- Mit Hilfe der Funktion `wantarray` ist es möglich, in Abhängigkeit des Kontexts des Prozeduraufrufs zu operieren. `wantarray` liefert

`undef`	falls der Return-Wert überhaupt nicht beachtet wird,
`0`	bei einem skalaren Kontext und
`1`	bei einem Listenkontext.

Anordnung von Prozeduren

- Da Perl zunächst generell die gesamte Quelle durchliest, bevor mit der Ausführung begonnen wird, dürfen Prozeduren nach ihren Aufrufen stehen, ohne daß vorher eine Deklaration notwendig ist.

- Es ist bei Perl üblicher Stil, daß zuerst das Hauptprogramm kommt und dann erst die Prozeduren. Ganz am Ende stehen dann typischerweise die Formate.

- Prozeduren dürfen (genauso wie in Pascal oder Modula-2) ineinander geschachtelt werden. Die Sichtbarkeitsbereiche der Prozeduren und mit `my` deklarierten Variablen ergeben sich dann entsprechend.

Sortierkriterien

sortaddr2.pl

```
#!/usr/local/bin/perl -w

# Anfang analog zu sortaddr.pl

foreach $name (sort byLastName keys %address) {
   printf "%-20s | %s\n", $name, $address{$name};
}

sub byLastName {
   my ($left, $right) = ($a, $b);
   $left =~ s/.*\s//; $right =~ s/.*\s//;
   return $left cmp $right || $a cmp $b;
}
```

- Bei `sort` kann noch vor der Liste der zu sortierenden Elemente ein Sortierkriterium in Form einer speziellen Prozedur eingeschoben werden.

- Sortierkriterien erhalten genau zwei Parameter namens `$a` und `$b` (wie sonst auch via *call by reference*).

- Das Sortierkriterium liefert analog zur C-Funktion `strcmp` entweder einen Wert < 0, $= 0$ oder > 0 zurück – je nachdem, ob `$a` kleiner, gleich oder größer als `$b` ist.

- Perl hat zwei Sätze von Vergleichsoperatoren: Der aus C gewohnte Satz vergleicht numerisch, die Operatoren `lt`, `le`, `eq`, `ne`, `ge` und `gt` beziehen sich auf Zeichenketten.

- Die speziellen Operatoren `<=>` und `cmp` liefern einen Wert analog zu `strcmp` und sind damit ideal für Sortierkriterien.

Sortierkriterien

sortaddr2.pl

```
sub byLastName {
   my ($left, $right) = ($a, $b);
   $left =~ s/.*\s//; $right =~ s/.*\s//;
   return
      $left cmp $right ||    # nach dem Nachnamen und
      $a cmp $b;             # dann nach dem Vornamen
}
```

- Bei zwei Personen mit dem gleichen Nachnamen wäre bei dem vorherigen Sortierkriterium die Reihenfolge undefiniert.

- Mit dem ||-Operator können mehrere Sortierkriterien kaskadiert werden. Wegen der *short circuit evaluation* von || wird die Bewertung der Kette beendet, wenn das erste Kettenglied einen Wert ungleich 0 zurückliefert.

Reguläre Ausdrücke

- Reguläre Ausdrücke können in gewohnter Weise in Schrägstriche gefaßt werden und sind weitgehend aufwärtskompatibel zu den anderen UNIX-Werkzeugen.

- Alternativ kann hinter dem `m//`-Operator ein anderes Begrenzungszeichen angegeben werden, wobei bei einem der 4 offenen Klammerzeichen das entsprechende Gegenstück erwartet wird (z.B.: `m{^/}`).

- Der Muster-Operator `=~` erlaubt das Testen eines regulären Ausdrucks gegen eine Zeichenkette:

```
$filename =~ /^\.\.?$/ # . oder ..?
```

- Mit dem Operator `!~` steht die Negation von `=~` zur Verfügung:

```
die "Absolute path required" if $filename !~ m{^/};
```

- Der `s///`-Operator erlaubt Textersetzungen analog zu *sed(1)* und *ex(1)*:

```
($cmdname = $0) =~ s{.*/}{}; # verwende den Basisnamen
```

Ein Nachbau von *grep*

grep.pl

```
#!/usr/local/bin/perl -w

($cmdname = $0) =~ s{.*/}{};
die "Usage: $cmdname pattern [file...]\n" if @ARGV == 0;
$pattern = shift @ARGV;

while(<>) {
   print if /$pattern/;
}
```

- Genauso wie in Zeichenketten können auch innerhalb von regulären Ausdrücken Variablen interpoliert werden.

- Der <>-Operator ohne angegebene Dateiverbindung bezieht sich jeweils auf die übriggebliebenen Elemente von `@ARGV`. Somit kann hier mit dem `shift` das erste Argument weggenommen werden und der Rest der Argumente dem <> zur weiteren Bearbeitung überlassen werden.

Parsieren einfacher Textdateien

mountpoints.pl

```
#!/usr/local/bin/perl -w
# Ausgabe einer sortierten Liste aller Mountpoints
# aus der fstab

if (!open(FSTAB, "/etc/fstab") &&
      !open(FSTAB, "/etc/vfstab")) {
   die "Unable to locate fstab file!\n";
}
@mountpoints = ();
while (<FSTAB>) {
   next if /^#/;     # Auskommentierte ...
   next if /^\s*$/;  # und leere Zeilen ueberspringen
   chomp;
   $mountpoint = (split /\s+/)[2];
   next unless $mountpoint =~ m{^/};
   push(@mountpoints, $mountpoint);
}
close(FSTAB);

print join("\n", sort @mountpoints), "\n";
```

- Mit `next` kann (analog zu `continue` in C) die nächste Iteration einer Schleife initiiert werden. (Analog gibt es `last`, das `break` in C entspricht).
- `unless` entspricht `if !`.
- `\s` in regulären Ausdrücken steht für Leerzeichen, Tabs, Zeilentrenner usw.
- Auch ein Ausdruck, der eine Liste liefert, kann direkt indiziert werden, wenn er eingeklammert wird.

Flexibler Textersatz

translate.pl

```
#!/usr/local/bin/perl -w

($cmdname = $0) =~ s{.*/}{};
die "Usage: $cmdname dictionary [file...]"
   unless @ARGV >= 1;
$dict = shift @ARGV;

%dict = ();
open(DICT, "<$dict") ||
   die "$cmdname: unable to open $dict\n";
while(<DICT>) {
   next if /^#/;
   chomp;
   ($orig, $replace) = split /:/;
   $dict{$orig} = $replace;
}
close(DICT);

while (<>) {
   s{\w+}{
      defined $dict{$&}?
         $dict{$&}
      :
         $&
   }xge;
   print;
}
```

- Dieses Skript ersetzt aufgrund einer Übersetzungstabelle alle Worte, die es in der Eingabe findet. Dabei werden unbekannte Worte unverändert gelassen.

Flexibler Textersatz

translate.pl

```
s{\w+}{
   defined $dict{$&}?
      $dict{$&}
   :
      $&
}xge;
```

- Ähnlich wie bei *sed(1)* oder *ex(1)* können einige Optionen am Ende eines regulären Ausdrucks (bzw. eines `s///`-Operators) stehen.

- Hier kommen folgende Optionen zum Tragen:

`g`	alle Ersetzungen werden durchgeführt, nicht nur einmal
`e`	der Textersatz darf ein beliebiger Perl-Ausdruck sein
`x`	erweiterte Syntax: Leerzeichen können frei eingestreut werden

- `\w` steht für ein beliebiges alphanumerisches Zeichen oder den Unterstrich (_) und ist somit zur Worterkennung geeignet.

- In der Variablen `$&` ist der soeben erkannte Text zu finden.

- Genauso wie in C gibt es den `?:`-Operator.

Modifikatoren im Überblick

`e`	Betrachte die rechte Seite bei einer Ersetzung als Ausdruck.
`g`	Alle Vorkommen des Musters sind global zu ersetzen.
`i`	Groß- und Kleinschreibung ist nicht signifikant.
`m`	`^` und `$` beziehen sich nicht auf den absoluten Anfang und das absolute Ende des Textes, sondern auf Zeilenanfang und Zeilenende eines potentiell mehrzeiligen Textes.
`o`	Übersetze das Muster nur ein einziges Mal.
`s`	Der Text ist als einzige Zeile zu betrachten ungeachtet der darin enthaltenen Zeilentrenner, insbesondere trifft der Punkt auch für den Zeilentrenner zu.
`x`	Innerhalb des regulären Ausdrucks sind Leerzeichen und Kommentare zulässig.

- Die Modifikatoren `e`, `g` und `o` sind nur bei dem `s///`-Operator zulässig.

- Die Modifikatoren werden in beliebiger Reihenfolge direkt hinter dem Muster (oder der rechten Seite bei einer Ersetzung) ohne zwischenliegende Leerzeichen angegeben.

- Die Modifikatoren `i`, `m`, `s` und `x` können auch in den regulären Ausdruck selbst übernommen werden mit `(?m)`. Das ist praktisch für Variablen, die ein Muster zusammen mit so einem Modifikator tragen sollen:

```
$pattern = "(?i)uni-ulm.de";
# ...
if (/$pattern/) { # ...
```

Metazeichen in regulären Ausdrücken

\	Nimmt dem folgenden nicht-alphanumerischen Zeichen die Sonderfunktion.
^	Anfang der Zeile oder des Textes (siehe `m`-Modifikator).
.	Beliebiges Zeichen außer dem Zeilentrenner (siehe jedoch `s`-Modifikator).
$	Zeilentrenner oder Ende des Textes (siehe `m`-Modifikator).
|	Alternation.
()	Gruppierung.
[]	Zeichenbereiche.

- Dies entspricht dem Umfang von *egrep(1)*.
- Im Gegensatz zu *vi(1)* gelten die Klammern als Metasymbole.
- Es empfiehlt sich, alle nicht-alphanumerischen Zeichen, die sich selbst repräsentieren sollen, durch ein \ zu schützen.
- \ gefolgt von einem alphanumerischen Zeichen nimmt nicht dem alphanumerischen Zeichen seine (nicht vorhandene) Sonderbedeutung, sondern – ganz im Gegenteil – verleiht eine Sonderbedeutung (siehe noch folgende Übersicht).

Quantifikatoren in regulären Ausdrücken

`*`	0 bis beliebig oft.
`+`	Mindestens einmal, ansonsten beliebig oft.
`?`	Optional: 0 oder 1 mal.
`{n}`	Genau n mal.
`{n,}`	Mindestens n mal.
`{n,m}`	Zwischen n und m mal.

- Normalerweise sind reguläre Ausdrücke "gierig", das heißt sie versuchen, soviel Text wie möglich zu erfassen. Wenn dies nicht gewünscht wird, dann kann hinter jedem Quantifikator noch ein ? angegeben werden, um eine minimale Erfassung zu erreichen.

- Beispiel: Das Muster `m{/\*.*?\*/}s` erfaßt genau einen Kommentar in C im Gegensatz zu dem Muster `m{/\*.*\*/}s`, das mehrere Kommentare auf einmal mitsamt dem dazwischenliegenden Text verschlingen kann.

Sonderzeichen in regulären Ausdrücken

`\t`	Tabulator.
`\n`	Zeilentrenner.
`\r`	Wagenrücklauf (CR).
`\f`	Papiervorschub (FF).
`\a`	Alarmglocke (BEL).
`\e`	Fluchtzeichen (ESC).
`\033`	Oktaldarstellung eines Zeichens.
`\x1b`	Hexdarstellung eines Zeichens.
`\cx`	Control-X.
`\l`	Folgendes Zeichen in einen Kleinbuchstaben verwandeln.
`\u`	Folgendes Zeichen in einen Großbuchstaben verwandeln.
`\L`	Bis zu `\E` alles in Kleinbuchstaben verwandeln.
`\U`	Bis zu `\E` alles in Großbuchstaben verwandeln.
`\Q`	Bis zu `\E` allen Metazeichen die Sonderbedeutung nehmen.

Kurzformen in regulären Ausdrücken

`\w`	Erfaßt alle alphanumerischen Zeichen einschließlich dem Unterstrich _.
`\W`	Erfaßt alle Zeichen, die `\w` nicht erfaßt.
`\s`	Erfaßt alle Leerzeichen.
`\S`	Erfaßt alle Zeichen, die nicht zu den Leerzeichen gehören.
`\d`	Alle Ziffern.
`\D`	Alle Zeichen, die nicht zu den Ziffern gehören.

- Mit `\w` wird noch kein Wort, sondern nur ein Zeichen eines Wortes erfaßt. Jedoch läßt sich ersteres leicht mit `\w+` erreichen.

Bestimmte Punkte in regulären Ausdrücken

`\b`	Steht für eine Wortkante.
`\B`	Steht für einen Punkt, der keine Wortkante ist.
`\A`	Absoluter Anfang des Textes.
`\Z`	Absolutes Ende des Textes oder vor dem Zeilentrenner am absoluten Ende des Textes.
`\z`	Absolutes Ende des Textes.
`\G`	Ende des zuvor erfaßten Textes (nur sinnvoll in Verbindung mit dem `g`-Modifikator).

- Diese Ausdrücke erfassen keinen Text, sondern nur bestimmte Punkte innerhalb des Textes.

Variablen, die von regulären Ausdrücken abhängen

`$&`	Der gesamte durch den letzten regulären Ausdruck erfaßte Text.
`$‘`	Text vor dem erfaßten Text.
`$’`	Text hinter dem erfaßten Text.
`$1`	Der Text, der durch das erste Klammernpaar im regulären Ausdruck erfaßt worden ist. Analog gibt es `$2` usw., wobei es keine Begrenzung der Anzahl der Klammernpaare gibt (`$37` ist auch in Ordnung).
`$+`	Text, der durch das letzte effektive Klammernpaar erfaßt worden ist.

- All diese Variablen behalten ihren Wert bis zur Bewertung des nächsten regulären Ausdrucks innerhalb des umgebenden Blocks (oder `eval`).

- Wenn ein Klammernpaar mehrfach Teile eines Textes erfaßt (wegen einem Quantifikator, der Wiederholungen zuläßt), dann ist in der zugehörigen Variable nur der zuletzt erfaßte Text zu finden.

- Bei verschachtelten Klammernpaaren zählt das äußere Klammernpaar zuerst.

- `(?:regexp)` bietet im Vergleich zu `(regexp)` nur die reine klammernde Funktion an, ohne daß es als Klammernpaar im obigen Sinne zählt.

Der m//-Operator im Listenkontext

```
($wday, $month, $mday,
      $hour, $minute, $second, $timezone, $year) =
   m{
      (\w+)                    # Mon, Tue, ...  --> $wday
      \s
      (\w+)                    # Jan, Feb, ...  --> $month
      \s+
      (\d{1,2})                # 1 .. 31        --> $mday
      \s+
      (\d{2}):(\d{2}):(\d{2}) # 23:27:41       --> $hour...
      \s
      (\w+(?:\s\w+)*?)         # MET DST or GMT --> $timezone
      \s
      (\d{4})                  # 1998           --> $year
   }x;
```

- Während der m//-Operator im skalaren Kontext nur zurückliefert, ob der reguläre Ausdruck zutrifft, werden im Listenkontext $1, $2 usw. als Liste zurückgeliefert.

- Im skalaren Kontext wird überhaupt erst keine Liste erzeugt und entsprechend entweder 0 oder 1 zurückgeliefert und nicht etwa die Länge der Liste, die im Listenkontext zurückgeliefert worden wäre.

- split kann eine sehr gute Alternative sein – insbesondere, wenn die Anzahl der herauszupickenden Felder nicht bekannt ist.

Rückwärtsverweise

double.pl

```
#!/usr/local/bin/perl -w

while(<>) {
   print if /^(\w+)\1$/;
}
```

- Rückwärtsverweise gehören zu den Techniken, die den formalen Rahmen regulärer Ausdrücke sprengen (d.h. die damit beschriebenen Sprachen sind nicht mehr notwendigerweise regulär). Dennoch sind sie beliebt und werden beispielsweise von *vi(1)*, *egrep(1)* und natürlich Perl unterstützt.

- Ein Rückwärtsverweis bezieht sich auf bereits erkannten Text, der durch ein Klammerpaar eingegrenzt ist. Der Verweis trifft zu, wenn der bereits erkannte Text an der Stelle des Verweises erneut vorkommt.

- `\1` ist der Rückwärtsverweis auf das erste Klammerpaar, `\2` auf das zweite usw. Mehrstellige Verweise sind zulässig – führende Nullen jedoch nicht (das wäre in Konflikt zu den Sonderzeichen).

- Rückwärtsverweise sollten nicht auf der rechten Seite des `s///`-Operators stehen – dort sind `$1`, `$2` usw. zu verwenden.

- Auf `/usr/dict/words` angewandt, liefert das Beispiel beriberi, coco, couscous, dodo, gogo, ii, juju, lulu, murmur, papa, tartar, testes, tete und tutu.

Voraus- und Rückschau bei regulären Ausdrücken

```
@words = split /(?=[A-Z])/;
```

- Mit `(?=regexp)` findet eine Vorausschau statt, ohne daß dabei Text erfaßt wird (*zero-width lookahead*).
- Obiges Beispiel zerlegt Namen wie `"ThisIsALongName"` in `("This", "Is", "A", "Long", "Name")`.
- Bei `(?!regexp)` handelt es sich um eine negierte Vorausschau: Das Muster trifft zu, wenn nichts folgt, auf das `regexp` zutrifft.
- Mit `(?<=regexp)` ist eine Rückschau möglich, wobei jedoch `regexp` nur eine begrenzte Länge erfassen darf.
- Die entsprechende negierte Form ist `(?<!regexp)`.

Arbeitsweise bei regulären Ausdrücken

- Hinweis: Folgende Zusammenstellung folgt in abgekürzter Form den Erläuterungen über *The rules of regular expression matching* aus *Programming Perl* von Larry Wall (Seite 60 ff).

- Perl verwendet einen nicht-deterministischen endlichen Automaten (NFA) in Verbindung mit Backtracking, um zu untersuchen, ob (und wenn ja, wo) ein Muster für einen Text zutrifft:

- **Regel 1**: Der Automat versucht, das Muster soweit links wie möglich im Text unter Beachtung von Regel 2 zu finden.

- **Regel 2**: Das Muster wird als Menge von Alternativen betrachtet (wovon mindestens eine vorliegt), die genau dann zutrifft, wenn irgendeine der Alternativen unter Beachtung der Regel 3 zutrifft. Die Alternativen werden in der vorgegebenen Reihenfolge durchprobiert und die Untersuchung endet beim ersten Erfolg.

Arbeitsweise bei regulären Ausdrücken

- **Regel 3**: Eine Alternative trifft dann zu, wenn jedes Teil in der sequentiellen Reihenfolge entsprechend den Regeln 4 und 5 zutrifft. Ein Teil ist entweder Zusicherung bezüglich eines Punktes im Text (z.B. `\b`, siehe Regel 4) oder ein Atom mitsamt einem optionalen Quantifikator (z.B. `x*`, siehe Regel 5).

  Teile, die auf unterschiedliche Weise zutreffen können, werden in hierarchischer Reihenfolge durchprobiert. So erhält bei `x*y*` zuerst `x*` eine Möglichkeit und dann werden alle Varianten von `y*` durchprobiert, bis es klappt. Klappt es nicht, geht es bei `x*` mit der nächsten Variante weiter.

- **Regel 4**: Wenn eine Zusicherung bezüglich eines Punktes im Text nicht zutrifft, dann geht es bei Regel 3 entsprechend der hierarchischen Reihenfolge bei den vorangehenden Teilen der aktuellen Alternative weiter.

Arbeitsweise bei regulären Ausdrücken

- **Regel 5**: Ein quantifiziertes Atom trifft genau dann zu, wenn das Atom (entsprechend Regel 6) in einer Quantität zutrifft, die vom Quantifikator zugelassen ist. Fehlt der Quantifikator, so wird `{1}` angenommen.

  Wenn verschiedene Quantitäten zugelassen sind, dann wird bei Quantifikatoren ohne ein abschließendes `?` zuerst die maximale Anzahl probiert ("greedy matching") und bei Quantifikatoren mit einem zusätzlichem `?` beginnt die Untersuchung mit der geringstmöglichen Anzahl.

  Bei "gierigen" Quantifikatoren beginnt die Untersuchung natürlich nicht bei dem theoretischen Maximum (das bei ∞ liegen könnte) – stattdessen wird beim konkreten Text untersucht, wie häufig hintereinander das Atom maximal zutreffen kann.

- **Regel 6**: Atome sind entweder Gruppierungen (also in `(...)` eingeschlossene reguläre Ausdrücke), Voraus- oder Rückschauungen, Sonderzeichen, Kurzformen, Zeichenbereiche, Rückwärtsverweise oder Zeichen, die für sich selbst stehen. Bei Gruppierungen und Vorausschauungen wird entsprechend Regel 2 verfahren, bei allen anderen läßt sich unmittelbar überprüfen, ob sie zutreffen oder nicht.

Neue Techniken bei regulären Ausdrücken

Mit Perl in der Version 5.005 sind einige Varianten hinzugekommen:

- Der Backtracking-Algorithmus (insbesondere die Regeln 3 und 5) können zu einem exponentiell anwachsenden Aufwand führen, der bei längeren Texten unzumutbar ist. Hierfür kann die Konstruktion `(?>regexp)` recht nützlich sein, die `regexp` an der aktuellen Stelle im Text untersucht, als ob kein umgebender regulärer Ausdruck existieren würde. Backtracking kommt entsprechend nicht zum Zuge.

- Mit `(?(condition)yes-pattern|no-pattern)` bzw. `(?(condition)yes-pattern)` können reguläre Ausdrücke in Abhängigkeit von einer Bedingung eingefügt werden, die entweder

  - eine natürliche Zahl n ist und dann zutrifft, falls das Muster der n-ten Klammer zutraf oder

  - eine Zusicherung ist, die keinen Text erfaßt (Vor- oder Rückschau, bestimmte Punkte).

- Mit `(?{ code })` können Perl-Ausdrücke eingebettet werden, die immer zutreffen. Da Sichtbarkeitsbereiche entsprechend der Backtracking-Ebenen gezogen werden, kann auf diese Weise beispielsweise ermittelt werden, welchen Wert ein bestimmter Quantifikator angenommen hat.

Verzeichnisse lesen

dir.pl

```
#!/usr/local/bin/perl -w

($cmdname = $0) =~ s{.*/}{};
push(@ARGV, ".") unless @ARGV > 0;
foreach $dir (@ARGV) {
   opendir(DIR, $dir) ||
      die "$cmdname: unable to open $dir\n";
   @filenames = ();
   while (defined($filename = readdir(DIR))) {
      next if $filename =~ /^\.\.?$/;
      push(@filenames, $filename);
   }
   closedir(DIR);
   print "$dir:\n";
   foreach $filename (sort @filenames) {
      print "\t$filename\n";
   }
}
```

- Dieses Skript eröffnet jedes angegebene Verzeichnis (bzw. das aktuelle, wenn keines genannt worden ist) und gibt die darin enthaltenen Dateinamen in alphabetischer Reihenfolge aus.

- Verzeichnisverbindungen sind zwar keine Dateiverbindungen (im Sinne von Perl) – sind in der Verwendung aber ähnlich.

- Mit `opendir` kann ein Verzeichnis eröffnet werden, `readdir` liefert die darin enthaltenen Dateinamen in undefinierter Reihenfolge und `closedir` schließt eine Verzeichnisverbindung.

Rekursives Durchwandern des Dateibaums

find.pl

```
#!/usr/local/bin/perl -w
# Schmalspurfassung von ‘‘find’’, bei der ein Dateiname
# rekursiv ab dem aktuellen Verzeichnis gesucht wird.

$wanted = $ARGV[0]; # gesuchter Dateiname
find(".", $wanted); # im aktuellen Verzeichnis starten

sub subdirs { # liefere alle Unterverzeichnisse zurueck
   my ($path) = @_;
   my ($file, @dirs) = ();
   return () unless opendir(DIR, $path);
   while (defined($file = readdir(DIR))) {
      next if $file =~ /^\.\.?$/;
      push(@dirs, "$path/$file") if -d "$path/$file";
   }
   closedir(DIR);
   return @dirs;
}

sub find {
   my ($path, $wanted) = @_;
   print "$path/$wanted\n" if -f "$path/$wanted";
   foreach $dir (subdirs($path)) {
      find($dir, $wanted);
   }
}
```

- Dies ist eine sehr einfache Fassung von *find*, die ab dem aktuellen Verzeichnis nach allen Dateien sucht, die den angegebenen Namen tragen.

Dateitest-Operatoren

Sowohl Dateinamen als auch Dateiverbindungen sind als Operanden zulässig ($_ wird per Voreinstellung verwendet):

`-r -w -x`	Datei ist lesbar/schreibbar/ausführbar für die effektive UID/GID
`-R -W -X`	Datei ist lesbar/schreibbar/ausführbar für die reelle UID/GID
`-o -O`	die Datei gehört der effektiven/reellen UID
`-e -z`	die Datei existiert / hat die Länge 0
`-s`	die Datei existiert mit einer Länge > 0, der Operator liefert die Länge zurück
`-f -d`	eine normale Datei / ein Verzeichnis
`-l -S -p`	die Datei ist ein symbolischer Verweis / ein Socket / eine benannte Pipeline (FIFO)
`-b -c`	die Datei ist ein Block- / Zeichen-Gerät
`-u -g -k`	setuid/setgid/sticky Bit ist gesetzt
`-t`	die Dateiverbindung ist mit einem Terminal verbunden
`-T -B`	Textdatei / Binärdatei – beides ist wahr für leere Dateien
`-M -A -C`	liefert die Modifizierungs- / Zugriffs- / Inode-Änderungszeit in Tagen im Vergleich zu dem Zeitpunkt, als das Skript startete

Die Tabelle wurde (in übersetzter Form) der Perl 5 Desktop Reference von Johan Vromans entnommen.

Globs und `stat`

tobecompiled.pl

```
#!/usr/local/bin/perl -w

@tobecompiled = ();
foreach $source (glob("*.c")) {
   ($object = $source) =~ s/\.c$/.o/;
   if (! -f $object || newer($source, $object)) {
      push(@tobecompiled, $source);
   }
}
if (@tobecompiled > 0) {
   print "To be compiled: ", join(" ", @tobecompiled), "\n";
} else {
   print "All objects are up-to-date!\n";
}

sub newer { # return true if file1 is newer than file2
   my ($file1, $file2) = @_;
   return (stat($file1))[9] > (stat($file2))[9];
}
```

- Dieses Skript stellt (analog zu *make*) fest, welche C-Quellen frisch zu übersetzen sind aufgrund der vorhandenen Objekte und der Zeitstempel.

- `glob` akzeptiert als Parameter ein Dateimuster der Bourne-Shell und liefert alle gefundenen Dateinamen als Liste zurück.

- `stat` liefert eine 13-elementige Liste mit den Werten aus der entsprechenden Struktur zurück: (0: dev, 1: ino, 2: mode, 3: nlink, 4: uid, 5: gid, 6: rdev, 7: size, 8: atime, 9: mtime, 10: ctime, 11: blksize, 12: blocks).

Zeiger

- Einfache Listen und assoziative Arrays sind für viele Probleme unzureichend, bei denen kompliziertere Datenstrukturen notwendig sind.

- Bei Listen und assoziativen Arrays ist in Perl die Restriktion, daß Elemente nur Skalare sein können. Somit wird z.B. eine Liste von Listen oder ein assoziatives Array mit Listen nicht direkt unterstützt.

- Stattdessen unterstützt Perl sichere Zeiger, die zwar nicht typgebunden sind, aber trotzdem immer wissen, worauf sie zeigen.

- Natürlich kann Speicher dynamisch beschafft werden (dafür gibt es sogar mehrere Möglichkeiten).

- Eine Speicherfreigabe gibt es nicht – stattdessen kümmert sich darum eine Garbage-Collection, die auf Basis von Referenz-Zählern arbeitet. Bei zyklischen Datenstrukturen sind somit Speicherlecks denkbar.

- Da Zeiger immer wohldefiniert sind (entweder `undef` oder auf ein lebendes Objekt zeigend, dessen Typ Perl bekannt ist), gibt es keine Probleme mit Zeigern, die "in den Wald zeigen".

- Die Notation in Zusammenhang mit Zeigern ist zunächst sehr gewöhnungsbedürftig, da sie kaum Ähnlichkeiten mit anderen Programmiersprachen aufweist. Aber man kann sich daran gewöhnen...

- Nach wie vor läßt sich jeweils der Typ bzw. die Datenstruktur direkt dem Ausdruck ablesen.

Eine Liste von Listen

sortaddr3.pl

```
#!/usr/local/bin/perl -w

$addressbook = "addressbook";
open(BOOK, "<$addressbook") ||
   die "Cannot open $addressbook\n";
@addresses = ();
while (<BOOK>) {
   chomp;
   ($name, $address) = split /:/;
   ($lastname = $name) =~ s/.*\s//;
   push(@addresses, [$lastname, $name, $address]);
}
close(BOOK);

foreach $address (sort { $a->[0] cmp $b->[0] } @addresses) {
   printf "%-20s | %s\n", $address->[1], $address->[2];
}
```

- Wenn eine Liste in `[...]` statt in `(...)` eingeklammert ist, wird ein Zeiger auf die neu im Speicher angelegte Liste zurückgeliefert.

- In der `foreach`-Schleife ist `$address` jeweils ein Zeiger auf eine Liste mit drei Elementen (für Nachname, ganzer Name und Adresse). Mit `->` wird der Zeiger dereferenziert und ein Zugriff auf eines der Elemente ist möglich.

Eine Liste von Listen

sortaddr4.pl

```
#!/usr/local/bin/perl -w

$addressbook = "addressbook";
open(BOOK, "<$addressbook") ||
   die "Cannot open $addressbook\n";
@addresses = ();
while (<BOOK>) {
   chomp;
   ($name, $address) = split /:/;
   ($lastname = $name) =~ s/.*\s//;
   push(@addresses, [$lastname, $name, $address]);
}
close(BOOK);

foreach $address (sort { $a->[0] cmp $b->[0] } @addresses) {
   ($lastname, $name, $address) = @{$address};
   printf "%-20s | %s\n", $name, $address;
}
```

- Wenn nicht ein einzelnes Element hinter dem Zeiger selektiert werden soll, sondern die gesamte Liste, dann geht dies mit `@{$address}`.

- Alternativ wäre hier auch `@$address` akzeptiert worden – dennoch ist die andere Variante lesbarer und in komplizierteren Fällen ohne Mehrdeutigkeiten.

Speicherallokation mit `my`

sortaddr5.pl

```
#!/usr/local/bin/perl -w

$addressbook = "addressbook";
open(BOOK, "<$addressbook") ||
   die "Cannot open $addressbook\n";
@addresses = ();
while (<BOOK>) {
   chomp;
   my(@address) = /(.*)\s(\S+):(.*)/;
   push(@addresses, \@address);
}
close(BOOK);

foreach $address (sort { $a->[1] cmp $b->[1] } @addresses) {
   ($firstname, $lastname, $address) = @{$address};
   printf "%-20s | %s\n", "$firstname $lastname", $address;
}
```

- `my` erzeugt nicht lokale Variablen im herkömmlichen Sinn, sondern legt dynamisch Speicher an (analog zu `malloc`).
- In der `while`-Schleife wird somit `@address` jedesmal frisch angelegt.
- Der \-Operator ist der Adreß-Operator analog zu `&` in C.

Eine Liste von assoziativen Arrays

sortaddr6.pl

```
#!/usr/local/bin/perl -w

$addressbook = "addressbook";
open(BOOK, "<$addressbook") ||
   die "Cannot open $addressbook\n";
@addresses = ();
while (<BOOK>) {
   chomp;
   ($name, $address) = split /:/;
   ($lastname = $name) =~ s/.*\s//;
   push(@addresses, {
      lastname => $lastname,
      name => $name,
      address => $address
   });
}
close(BOOK);

foreach $address (sort byLastName @addresses) {
   printf "%-20s | %s\n",
      $address->{name}, $address->{address};
}

sub byLastName { $a->{lastname} cmp $b->{lastname} }
```

Eine Liste von assoziativen Arrays

sortaddr6.pl

```
push(@addresses, {
   lastname => $lastname,
   name => $name,
   address => $address
});
```

- Genauso wie mit `[...]` bei Listen können assoziative Arrays in `{...}` eingeklammert werden, um einen entsprechenden Zeiger auf eine neu allokierte Datenstruktur zu erhalten.

- Assoziative Arrays dienen in Perl häufig als Records (bzw. als `struct`). Die Komponentennamen werden dabei als Schlüssel verwendet.

- Wenn der Index eines assoziativen Arrays den Regeln für Identifier entspricht (`[A-Za-z_][A-Za-z_0-9]*`), kann auf eine Quotierung vor dem `=>` oder innerhalb der `{...}` verzichtet werden.

```
foreach $address (sort byLastName @addresses) {
   printf "%-20s | %s\n",
      $address->{name}, $address->{address};
}

sub byLastName { $a->{lastname} cmp $b->{lastname} }
```

- Genauso wie zuvor kann nach einer Dereferenzierung mit `->` ein Index für ein assoziatives Array in `{...}` angegeben werden.

Assoziative Arrays von assoziativen Arrays

scanpasswd.pl

```
#!/usr/local/bin/perl -w

$passwd = "</etc/passwd";
open(PASSWD, $passwd) ||
   die "Unable to read from $passwd\n";
while(<PASSWD>) {
   chomp;
   ($login, $passwd, $uid, $gid,
      $name, $home, $shell) = split /:/;
   $passwd_by_uid{$uid} =
   $passwd_by_login{$login} = {
      login => $login, passwd => $passwd,
      uid => $uid, gid => $gid,
      name => $name, home => $home, shell => $shell
   };
}
close(PASSWD);

sub getpwent_by_uid { return $passwd_by_uid{$_[0]}; }
sub getpwent_by_login { return $passwd_by_login{$_[0]}; }
```

- Genauso wie bei Listen können auch bei assoziativen Arrays die Elemente Zeiger auf beliebige Datenstrukturen sein.

Zeiger auf Funktionen

cmdline.pl

```
#!/usr/local/bin/perl -w

%cmd = ('sum' => \&sum, 'quit' => \&quit);
while (print(": "), defined($line = <STDIN>)) {
   ($cmdname, @fields) = split /\s+/, $line;
   next unless defined $cmdname;
   if (defined $cmd{$cmdname}) {
      &{$cmd{$cmdname}}(@fields);
   } else {
      print "Unknown command!\n";
   }
}

sub sum {
   my $sum = 0; foreach my $val (@_) { $sum += $val };
   print $sum, "\n";
}

sub quit { exit 0 }
```

- Wenn ein Zeiger von einer benannten Funktion gewünscht wird, darf das &-Symbol vor dem Funktionsnamen nicht vergessen werden.

- Das &-Symbol ist dann auch beim indirekten Aufruf durch den Zeiger notwendig.

Anonyme Funktionen

cmdline2.pl

```
#!/usr/local/bin/perl -w

%cmd = (
   'sum' => sub {
      my $sum = 0; foreach my $val (@_) { $sum += $val };
      print $sum, "\n";
   },
   'quit' => sub { exit 0 },
);
while (print(": "), defined($line = <STDIN>)) {
   ($cmdname, @fields) = split /\s+/, $line;
   next unless defined $cmdname;
   if (defined $cmd{$cmdname}) {
      &{$cmd{$cmdname}}(@fields);
   } else {
      print "Unknown command!\n";
   }
}
```

- `sub { ... }` liefert einen Zeiger auf eine anonyme (also unbenannte) Funktion.
- Das erlaubt eine sehr elegante Einbettung von Funktionen in Datenstrukturen.

Closures

counters.pl

```
#!/usr/local/bin/perl -w

%counter = ();
while (<>) {
   chomp;
   next unless ($name) = /^(\S+)$/;
   $counter{$name} = new_counter()
      unless defined $counter{$name};
   print &{$counter{$name}}(), "\n";
}

sub new_counter {
   my $counter = 0;
   return sub { return ++ $counter };
}
```

- Jedesmal, wenn `sub { ... }` bewertet wird, entsteht eine neue Inkarnation. Der Programmtext ist zwar konstant, nicht jedoch die jeweilige Umgebung.

- Auch wenn die Umgebung (zu der die umgebenden, mit **my** deklarierten Variablen gehören) längst verschwunden zu sein scheint, bleibt sie für die in ihr entstandenen anonymen Prozeduren erhalten.

- Die anonymen Prozeduren können dann auf den ererbten Variablen operieren.

- Diese Technik wurde durch Lisp bekannt unter dem Stichwort “closures”, d.h. anonyme Prozeduren behalten ihre Hülle (oder Umgebung).

Closures

counters2.pl

```
#!/usr/local/bin/perl -w

%counter = ();
while (<>) {
   chomp;
   next unless ($sign, $name) = /^([+-])(\S+)$/;
   $counter{$name} = new_counter()
      unless defined $counter{$name};
   print &{$counter{$name}{$sign}}(), "\n";
}

sub new_counter {
   my $counter = 0;
   return {
      '+' => sub { return ++ $counter },
      '-' => sub { return -- $counter },
   };
}
```

- Mehrere Inkarnationen anonymer Prozeduren können sich eine Umgebung teilen, wenn sie alle der gleichen Umgebung entstammen.

Zeiger im Überblick

`\$var`	Zeiger auf eine skalare Variable
`${$ptr}`	Dereferenzierung eines Zeigers auf eine skalare Variable
`\@list`	Zeiger auf eine Liste
`[$elem1, $elem2, ...]`	Zeiger auf eine neu erzeugte Liste
`@{$ptr}`	Dereferenzierung eines Listen-Zeigers
`$ptr->[$index]`	Zugriff auf ein Listenelement durch einen Zeiger
`$ptr->[$i][$j]`	`$ptr->[$i]->[$j]`
`$list[$i][$j]`	`$list[$i]->[$j]`
`\($elem1, $elem2, ...)`	`(\$elem1, \$elem2, ...)`
`\%hash`	Zeiger auf ein assoziatives Array
`%{$ptr}`	Dereferenzierung eines Zeigers auf ein assoziatives Array
`{$key1 => $val1, ...}`	Zeiger auf ein neu erzeugtes assoziatives Array
`$ptr->{$index}`	Zugriff auf ein Element eines assoziativen Arrays über einen Zeiger
`\&func`	Zeiger auf eine benannte Funktion
`$coderef = sub { ... }`	Zeiger auf eine anonyme Funktion
`&{$coderef}($p1, ...)`	Aufruf einer Funktion durch einen Zeiger

Module

Passwd.pm

```
package Passwd;

$passwd = "</etc/passwd";
open(PASSWD, $passwd) ||
   die "Unable to read from $passwd\n";
while(<PASSWD>) {
   chomp;
   ($login, $passwd, $uid, $gid,
      $name, $home, $shell) = split /:/;
   $passwd_by_uid{$uid} =
   $passwd_by_login{$login} = {
      login => $login, passwd => $passwd,
      uid => $uid, gid => $gid,
      name => $name, home => $home, shell => $shell
   };
}
close(PASSWD);

sub getpwent_by_uid { return $passwd_by_uid{$_[0]}; }
sub getpwent_by_login { return $passwd_by_login{$_[0]}; }

1; # Beim Laden muss TRUE zurueckgeliefert werden
```

- Jedes Modul hat seinen eigenen Namensraum und entsprechend gibt es keine globale Variablen – abgesehen von verschiedenen Standard-Variablen von Perl (`$_` zum Beispiel).
- Die Dateiendung für Perl-Module ist zwingend `.pm`.
- Modulnamen sollten mit einem Großbuchstaben beginnen, um Probleme und Kollisionen zu vermeiden.

Die Benutzung von Modulen

testpasswd.pl

```
#!/usr/local/bin/perl -w

require Passwd;

($cmdname = $0) =~ s{.*/}{};
die "Usage: $cmdname login\n" unless @ARGV == 1;
$pwentry = Passwd::getpwent_by_login($ARGV[0]);
if (defined $pwentry) {
   print "Name:         $pwentry->{name}\n";
   print "UID:          $pwentry->{uid}\n";
   print "GID:          $pwentry->{gid}\n";
} else {
   print "$ARGV[0] ist unbekannt!\n";
}
```

- Mit `require` wird ein Modul importiert. Dadurch wird nicht der aktuelle Namensraum beeinflußt.

- Module werden in allen Verzeichnissen gesucht, die in `@INC` enthalten sind. Dazu gehören alle Bibliotheks-Verzeichnisse von Perl und das aktuelle Verzeichnis. Mit der Environment-Variablen `PERL5LIB` oder entsprechenden Angaben auf der Kommandozeile (`-Idir`) können weitere Verzeichnisse für Module angegeben werden. Alternativ kann natürlich `@INC` explizit manipuliert werden.

- Alle Prozedur- oder Variablennamen aus einem so importierten Modul müssen mit dem Modulnamen voll qualifiziert werden.

- Es gibt in Perl keinen (einfachen) Schutz gegen die Verwendung von Internas eines Moduls. Per Konvention beginnen private Namen mit einem _.

Module, die automatisch Namen exportieren

Passwd2.pm

```
package Passwd2;

require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(getpwent_by_uid getpwent_by_login);

$passwd = "</etc/passwd";
open(PASSWD, $passwd) ||
   die "Unable to read from $passwd\n";
while(<PASSWD>) {
   chomp;
   ($login, $passwd, $uid, $gid,
      $name, $home, $shell) = split /:/;
   $passwd_by_uid{$uid} =
   $passwd_by_login{$login} = {
      login => $login, passwd => $passwd,
      uid => $uid, gid => $gid,
      name => $name, home => $home, shell => $shell
   };
}
close(PASSWD);

sub getpwent_by_uid { return $passwd_by_uid{$_[0]}; }
sub getpwent_by_login { return $passwd_by_login{$_[0]}; }

1;
```

Module, die automatisch Namen exportieren

Passwd2.pm

```
package Passwd2;

require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(getpwent_by_uid getpwent_by_login);
```

- Wenn ein Modul eine Erweiterung des Modules `Exporter` ist, lassen sich bestimmte ausgewählte Namen automatisch in den Namensraum des Importierers übernehmen.
- Mit `require Exporter` wird zunächst das entsprechende Modul importiert.
- Die spezielle Liste `@ISA` legt Erweiterungsbeziehungen fest.
- In der Liste `@EXPORT` stehen alle Namen drin, die automatisch im Namensraum des Importierers auftauchen sollen.
- `qw(name1 name2 name3)` ist eine Kurzform für `('name1', 'name2', 'name3')`.

Die `use`-Anweisung

testpasswd2.pl

```
#!/usr/local/bin/perl -w

use Passwd2;

($cmdname = $0) =~ s{.*/}{};
die "Usage: $cmdname login\n" unless @ARGV == 1;
$pwentry = getpwent_by_login($ARGV[0]);
if (defined $pwentry) {
   print "Name:         $pwentry->{name}\n";
   print "UID:          $pwentry->{uid}\n";
   print "GID:          $pwentry->{gid}\n";
} else {
   print "$ARGV[0] ist unbekannt!\n";
}
```

- Wenn nicht nur das Modul zur Verfügung stehen soll, sondern auch Namen in den eigenen Namensraum zu übernehmen sind, wird `use` anstatt von `require` verwendet.
- Entsprechend läßt sich nachher `getpwent_by_login` ohne weitere Qualifizierung verwenden.
- Von dieser Technik sollte nur sehr zurückhaltend Gebrauch gemacht werden, um Namenskollisionen zu vermeiden.
- Es gibt Techniken zur partiellen Übernahme von Namen, auf die hier nicht weiter eingegangen wird. Mehr dazu in der *perlmod*-Manualseite und ein sehr umfangreiches Beispiel dafür ist das bei Perl mitgelieferte Modul `POSIX`.

Exportlisten ohne Flutung

Passwd3.pm

```
package Passwd3;

require Exporter;
@ISA = qw(Exporter);
@EXPORT_OK = qw(getpwent_by_uid getpwent_by_login);
```

testpasswd3.pl

```
#!/usr/local/bin/perl -w

use Passwd3 qw(getpwent_by_login);
```

- `@EXPORT_OK` führt im Gegensatz zu `@EXPORT` nicht zu einer Flutung des Namensraumes des importierenden Moduls.
- Analog zu Modula-2 können dann bei `use` selektive Importlisten angegeben werden.
- Diese Technik ist der Flutung vorzuziehen, da sie stabil gegen Änderungen ist und genauer dokumentiert, was von wo benutzt wird. Natürlich ist die qualifizierte Benutzung externer Namen auch eine gut lesbare Alternative.

Mehr Überprüfungen

```
use strict;
```

- Die Angabe von `-w` auf der Kommandozeile von Perl ist sehr hilfreich. Dennoch bleiben eine Reihe obskurer (Fehl-)Interpretationen oder Techniken übrig, deren versehentliche Verwendung ausgeschlossen werden sollte.

- Hierfür gibt es das Pragma-Modul **`strict`**, von dem insbesondere bei größeren modularisierten (und damit nicht mehr sehr leicht überschaubaren) Projekten Gebrauch gemacht werden sollte.

- Im Gegensatz zu `-w` erstreckt sich die Wirkungsweise von `use strict` nicht auf das gesamte Programm, sondern nur auf den importierenden Block.

- Zur Zeit gibt es drei konkrete Überprüfungen bei **`strict`**, die auch separat spezifiziert werden können:

`use strict qw(refs);`	Keine symbolische Zeiger.
`use strict qw(vars);`	Nur Verwendung lokalisierter oder qualifizierter Variablen.
`use strict qw(subs);`	Restriktionen bei der Verwendung von ungeschützten Namen (*barewords*).

Vermeidung symbolischer Zeiger

```
use strict qw(refs);

$var = "Hallo";

$ptr = \$var; print ${$ptr} # zulaessig
$ptr = "var"; print ${$ptr} # nicht (mehr) zulaessig
```

- Symbolische Zeiger sind ganz normale Textvariablen, deren Inhalt beim Zugriff als Variablenname interpretiert wird.

- Diese Technik ist recht abenteuerlich, da die Bewertung vom jeweils lokalen Kontext abhängt – schließlich kann mit `my` oder `local` der Variablenname überdefiniert worden sein (oder wir befinden uns in einem anderen Modul).

- Von der Verwendung symbolischer Zeiger ist deswegen abzuraten und sie werden normalerweise nur versehentlich verwendet. Dies wird durch diese Variante von `strict` zuverlässig verhindert.

Mehr Ordnung bei Variablennamen

```
package XXX;

use strict qw(vars);
require Exporter;
@XXX::ISA = qw(Exporter);
@XXX::EXPORT = qw(...);

$var1 = 17;     # nicht (mehr) zulaessig,
                # falls $var1 noch nicht deklariert worden ist
local $var2;    # ist ebenfalls nicht (mehr) zulaessig
my $var;        # ist zulaessig
$XXX::var = 1; # qualifizierte Verwendung ist zulaessig
```

- Bei der mit `use strict qw(vars);` erzwungenen strikten Ordnung bei Variablennamen sind nur noch qualifizierte oder mit `my` lokal deklarierte Variablennamen zulässig.

- Dies erhöht gerade bei umfangreicheren Projekten nicht nur die Lesbarkeit (durch die erzwungenen Deklarierungen oder die qualifizierte Schreibweise), sondern vermeidet (in gegenüber `-w` vermehrten Maße) Probleme mit vertippten Variablennamen oder Fallen durch dynamische Sichtbereiche (bei `lokal`).

Einschränkungen in der Verwendung ungeschützter Namen

```
use strict qw(subs);

print STDOUT I, like, Perl, "\n"; # nicht (mehr) zulaessig
$array{key} = "value"; # ist noch zugelassen
%array = (key => 'value'); # ebenfalls noch zugelassen
```

- Ungeschützte Namen (in der Perl-Dokumentation *barewords* genannt) sind Namen ohne ein einleitendes Symbol wie `$`, `@`, `%` oder `&` bzw. alphanumerische Zeichenketten, die nicht eingefaßt sind (z.B. in `"..."`, `'...'` oder `qq(...)`).

- Grundsätzlich (d.h. mit Ausnahmen) teilen sich ungeschützte Namen den Namensraum mit den (zahlreich vorhandenen) Schlüsselwörtern von Perl. Die Gefahr liegt hier insbesondere in den Schlüsselwörtern, die noch kommen können.

- Prozedurnamen dürfen weiterhin ohne führendes `&` verwendet werden, wenn sie entweder vorher deklariert worden sind oder durch einen nachfolgenden Listenkonstruktor als solcher zu erkennen sind.

- Indizes für assoziative Arrays sind ebenfalls ausgenommen (das schließt die Verwendung ungeschützter Namen vor `=>` ein).

Ausnahmenbehandlungen

Passwd4.pm

```
use Carp;

# ...

sub pwinit {
   my $passwd = shift;
   $passwd = "/etc/passwd" unless defined $passwd;
   open(PASSWD, $passwd) ||
      croak "Unable to read from $passwd";
   # ...
}
```

- Bibliotheksmodule schieben die Schuld gerne auf den Aufrufer und verwenden daher gerne **croak** aus dem Modul **Carp** anstatt **die**.

testpasswd4.pl

```
use Passwd4 qw(getpwent_by_login pwinit);

(my($cmdname) = $0) =~ s{.*/}{};
die "Usage: $cmdname login [passwd]\n"
   if @ARGV == 0 || @ARGV > 2;
my $login = shift;
pwinit(shift) if @ARGV > 0;
my $pwentry = getpwent_by_login($login);
```

```
theseus$ testpasswd4.pl root /no/such/file
Unable to read from /no/such/file at testpasswd4.pl line 9
theseus$
```

Ausnahmenbehandlungen

testpasswd5.pl

```
if (@ARGV > 0) {
   my $passwd = shift;
   eval {
      pwinit($passwd);
   };
   if ($@) {
      warn "Unable to open $passwd. Trying default.\n";
      pwinit("/etc/passwd");
   }
}
```

- Mit **eval** können beliebige Ausführungsfehler abgefangen werden. Geht irgendetwas schief, wird die zugehörige Fehlermeldung in `$@` abgelegt.

- Damit ist es möglich, andere Fehlermeldungen zu erzeugen oder Ersatzstrategien zu verfolgen.

```
theseus$ testpasswd5.pl root /no/such/file
Unable to open /no/such/file. Trying default.
Name:         Super-User
UID:          0
GID:          1
theseus$
```

Ausnahmenbehandlungen im Überblick

`die $msg`	Abbruch mit der angegebenen Fehlermeldung
`warn $msg`	Fehlermeldung ohne Abbruch
`warn $msg if $^W`	Fehlermeldung ohne Abbruch, falls **-w** angegeben wurde
`use Carp; croak $msg`	Analog zu **die**, jedoch beziehen sich Modulangabe und Zeilennummer auf den Aufrufer
`use Carp; carp $msg`	Analog zu **warn**, jedoch beziehen sich Modulangabe und Zeilennummer auf den Aufrufer
`use Carp; confess $msg`	Analog zu **croak**, jedoch wird ein vollständiger Stack-Trace ausgegeben.
`$SIG{__DIE__} = sub { ... }`	Reaktion statt Abbruch bei **die**
`$SIG{__WARN__} = sub { ... }`	Reaktion statt Ausgabe bei **warn**
`eval { ... }; if ($@) { ... }`	Bau einer Schutzmauer mit **eval**

Objekte

- Objekte sind in Perl immer Referenzen auf eine beliebige Datenstruktur.
- Typischerweise wird als dahinterliegende Datenstruktur ein assoziatives Array verwendet, dessen Schlüssel durch die Komponentennamen vorgegeben sind.
- Jedes Objekt ist mit einem Modul verbunden. Die Verbindung wird durch die `bless`-Operation hergestellt.
- Wird eine Methode auf so einem Objekt aufgerufen (Syntax kommt gleich), wird zunächst in dem zugehörigen Modul nach der zugehörigen Prozedur gesucht und anschließend rekursiv in den Modulen, wovon das erste eine Erweiterung ist.
- Im Konfliktfall (mehrfache Implementierung einer Methode) entscheidet die Reihenfolge in `@ISA`.
- Der erste Parameter aller Methoden ist das Objekt selbst. Die weiteren Parameter sind dahinter zu finden.
- Module, die Objekt-Klassen repräsentieren, exportieren normalerweise überhaupt keine Namen, sondern sind nur über die Objekt-Schnittstelle erreichbar.

Ein Modul als Objekt-Klasse

Counter.pm

```
package Counter;

require Exporter;
@ISA = qw(Exporter);

sub new {
   my ($package, $startvalue, $increment) = @_;
   $startvalue = 0 unless defined $startvalue;
   $increment = 1 unless defined $increment;
   my ($self) = bless {
      value => $startvalue, incr => $increment}, $package;
   return $self;
}

sub inc {
   my ($self) = @_;
   $self->{value} += $self->{incr};
   return $self;
}

sub dec {
   my ($self) = @_;
   $self->{value} -= $self->{incr};
   return $self;
}

sub val {
   my ($self) = @_;
   return $self->{value};
}

1;
```

Objekt-Konstruktoren

Counter.pm

```
sub new {
   my ($package, $startvalue, $increment) = @_;
   $startvalue = 0 unless defined $startvalue;
   $increment = 1 unless defined $increment;
   my ($self) = bless {
      value => $startvalue, incr => $increment}, $package;
   return $self;
}
```

- Objekt-Konstruktoren werden per Konvention `new` genannt und erhalten als Parameter zunächst einen Zeiger auf ihr Modul und anschließend die weiteren Parameter, die explizit übergeben worden sind.

- `bless` erhält als ersten Parameter eine Referenz, die das Objekt darstellen soll und als zweiten Parameter das Modul, dem es zugeordnet ist.

- Zwar wäre es auch möglich, direkt den eigenen Modulnamen als zweiten Parameter bei `bless` anzugeben – dann wären aber erweiternde Module gezwungen, `new` überzudefinieren. Deswegen wird per Konvention immer der erste Parameter des Konstruktors dafür genommen, der eben auch ein abgeleitetes Modul sein kann.

- Konstruktoren sollten entweder `undef` (im Falle eines Mißerfolges) oder das neu geschaffene Objekt zurückliefern.

Methoden

Counter.pm

```
sub inc {
   my ($self) = @_;
   $self->{value} += $self->{incr};
   return $self;
}
```

- Methoden erhalten als ersten Parameter das Objekt, worauf sie operieren sollen.
- In den weiteren Parameter ist das enthalten, was ansonsten übergeben worden ist.

Die Benutzung von Klassen

testcounter.pl

```
#!/usr/local/bin/perl -w

use Counter;

$counter = new Counter @ARGV;
while (defined($ch = getc(STDIN))) {
   if ($ch eq "+") {
      $counter->inc;
   } elsif ($ch eq "-") {
      $counter->dec;
   }
}
print $counter->val, "\n";
```

- Mit `use Counter` wird die Klasse importiert, wobei hier der eigene Namensraum unverändert bleibt (abgesehen von `Counter`).
- `require Counter` hätte auch funktioniert – dennoch ist `use Counter` hier vorzuziehen. Dazu später mehr.
- Die Syntax von Methodenaufrufen ist entweder
  `method object parameters` oder
  `object->method parameters`
- Ein Modul ist ein Objekt, das mit sich selbst verknüpft ist.
- Die Operation `new` kann entsprechend hier via
  `new Counter @ARGV` oder
  via `Counter->new(@ARGV)` aufgerufen werden.

Dateiverbindungen

- Dateiverbindungen und Verzeichnisverbindungen haben jeweils kein spezielles Symbol, das ihnen syntaktisch vorausgeht.

- Grundsätzlich gilt, daß jede Art von Variablen ihren eigenen Namensraum hat. Somit stört es nicht, wenn z.B. Dateiverbindungen, Skalare, Listen und Prozeduren den gleichen Namen tragen.

- Entsprechend werden die Namen von Dateiverbindungen (und Verzeichnisverbindungen) nur jeweils in einem bestimmten Kontext erkannt.

- Das hat auch zur Folge, daß es nicht so einfach ist, Dateiverbindungen in eine Liste aufzunehmen.

- Genauso wenig ist es auf einfache Weise möglich, Dateiverbindungen als Parameter zu übergeben (insbesondere über Modulgrenzen hinweg) oder als lokale Variable zu definieren.

Type-Globs

- Wenn einem Variablennamen (unabhängig vom Typ) ein * vorausgeht, bezieht sich das auf *alle* Variablen, die diesen Namen tragen.

- Das heißt zum Beispiel, daß `*var` gleichzeitig für `$var`, `@var`, `%var` und die anderen Variablen mit dem Namen `var` des aktuellen Moduls steht, die durch kein spezielles Symbol gekennzeichnet werden.

- Auf diese Weise sind Eingriffe in die Symboltabelle möglich, die dann für die Lokalität von Dateiverbindungsvariablen sorgen können.

- Mit Hilfe von Symboltabellenzeigern (d.h. Zeiger auf Type-Globs) ist eine Übergabe als Parameter in fremde Module möglich.

- Diese Lösung ist weder elegant noch lesbar. Deswegen wird hier auf eine detaillierte Beschreibung verzichtet.

Skalare als Dateiverbindungen

```
# irgendwann zu Beginn
$unique_fh = "fh0000";

# spaeter bei der Eroeffnung einer Datei
$fh = $unique_fh ++;
open($fh, ...) || ...;
while(<$fh>) {
   ...
}
```

- Statt einer Variablen kann auch ein Skalar angegeben werden – in diesem Fall wird der Wert als Variablenname verwendet.
- Entsprechend muß durch einen geeigneten Zähler auf die Eindeutigkeit geachtet werden.
- Es ist in Perl möglich, beliebige alphanumerische Zeichenketten zu inkrementieren (so ergibt z.B. `'aZ'` inkrementiert `'bA'`).
- Hierdurch wird keine Lokalität im engeren Sinne hergestellt.
- Verbleibendes Problem: Dateiverbindungen können auf diese Weise nicht über Modulgrenzen hinweg übergeben werden, da der Name entsprechend des Werts der skalaren Variable jeweils im lokalen Namensraum gesucht wird.

Dateiverbindungen aus dem Modul `IO::File`

addresses3.pl

```
#!/usr/local/bin/perl -w

use strict;
use IO::File;

my $addressbook = "addressbook";
my $book = new IO::File $addressbook;
die "Cannot open $addressbook: $!\n" unless defined $book;
while (defined($_ = $book->getline)) {
   chomp;
   my($name, $address) = split /:/;
   printf "%-20s | %s\n", $name, $address;
}
$book->close;
```

- Das Modul `IO::File` offeriert eine elegante Lösung, die alle genannten Probleme löst.
- Der Konstruktor von `IO::File` liefert ein Objekt zurück, das einerseits eine Reihe von Methoden unterstützt aber auch in gewohnter Weise (z.B. innerhalb von <...>) verwendet werden kann.
- Dateiverbindungen, die `IO::File` kreiert hat, werden automatisch geschlossen, wenn sie der Garbage-Collection anheimfallen.
- Aus implementierungstechnischen Gründen ist es nicht möglich, eine Erweiterung des Moduls `IO::File` zu definieren.

Verzeichnisverbindungen aus dem Modul `IO::Dir`

dir2.pl

```
#!/usr/local/bin/perl -w

use strict;
use IO::Dir;

(my($cmdname) = $0) =~ s{.*/}{};
push(@ARGV, ".") unless @ARGV > 0;
foreach my $dir (@ARGV) {
   my $dh = new IO::Dir $dir;
   die "$cmdname: unable to open $dir: $!\n"
      unless defined $dh;
   my @filenames = (); my $filename;
   while (defined($filename = $dh->read)) {
      next if $filename =~ /^\.\.?$/;
      push(@filenames, $filename);
   }
   $dh->close;
   print "$dir:\n";
   foreach my $filename (sort @filenames) {
      print "\t$filename\n";
   }
}
```

- Das Modul `IO::Dir` liefert analog zu `IO::File` die Lösung für Dateiverbindungen.
- Mit der Methode `read` läßt sich jeweils der nächste Dateiname holen.

Beispiel: Kryptographische Hash-Funktionen

md5.pl

```
use Digest::MD2;
use Digest::MD5;
use Digest::SHA1;
use IO::File;
use strict;

my %digests = (
    'md2' => sub { return new Digest::MD2 },
    'md5' => sub { return new Digest::MD5 },
    'sha1' => sub { return new Digest::SHA1 },
);
```

- In der Perl-Bibliothek gibt es die Module `Digest::MD5`, `Digest::SHA-1` und `Digest::MD2` zur Berechnung der gleichnamigen kryptographischen Hash-Funktionen.

- Hier wird ein Skript verwendet, das in Abhängigkeit des verwendeten Namens jeweils ein Objekt aus dem zugehörigen Modul kreiert. Da alle Objekte die gleiche Schnittstelle haben, ist der Rest des Programms unabhängig von der Wahl der Hash-Funktion.

Beispiel: Kryptographische Hash-Funktionen

md5.pl

```
if (@ARGV == 0) {
   process(\*STDIN);
} else {
   foreach my $file (@ARGV) {
      my $in = new IO::File $file;
      die "$cmdname: unable to open $file: $!\n"
         unless defined $in;
      process($in, (@ARGV > 1? $file: undef));
      $in->close;
   }
}

sub process {
   my ($in, $filename) = @_;
   my $context = &{$digests{$cmdname}}();
   $context->addfile($in);
   print $filename, ": " if defined $filename;
   my $hex = $context->hexdigest;
   $hex =~ tr/a-f/A-F/;
   print $hex, "\n";
}
```

- `\*STDIN` ist ein Zeiger auf einen Type-Glob und die einzige Methode, STDIN als Parameter zu übergeben.

- Die Syntax des **tr**-Operators lehnt sich an dem s-Operator an. Die Arbeitsweise entspricht dem tr-Kommando, d.h. die Zeichen aus der geordneten ersten Menge werden durch die entsprechenden Zeichen der zweiten Menge ersetzt.

Alternative Implementierungen für standardmäßige Datenstrukturen

- Mit Hilfe der `tie`-Operation ist es möglich, alle internen Operationen für ein assoziatives Array oder für skalare Variablen überzudefinieren.

- `tie` ist auch für Listen möglich – seit Perl 5.005 sind alle Operationen überdefinierbar, ältere Versionen unterstützen nur einen Teil davon.

- Ähnlich ist die Situation für Dateiverbindungen: Die wichtigsten Operationen sind überdefinierbar einschließlich `print`, `printf`, `readline`, `getc`, `read` und natürlich *destroy*.

- Hauptsächlich wird dieses Feature bei assoziativen Arrays eingesetzt. So ist es zum Beispiel möglich, diverse DBM-Formate in Form eines assoziativen Arrays zu bearbeiten. Hierfür werden auch standardmäßig eine Reihe von Modulen mit ausgeliefert.

- Bei Skalaren besteht die Möglichkeit, besondere Aktionen bei Veränderungen durchzuführen. Als Beispiel ist z.B. eine Variable `$nice` denkbar, deren Veränderung zu entsprechenden Systemaufrufen führt, die den Nice-Level verändern.

Assoziative Arrays mit einer alternativen Implementierung

Folgende Operationen müssen von einem Modul definiert werden, das als Implementierung eines assoziativen Arrays dienen soll:

`TIEHASH` *classname*, *list*	Dient als Konstruktor und wird implizit von der `tie`-Operation aufgerufen. Genauso wie bei `new` wird als 1. Parameter ein Zeiger auf das Modul übergeben, gefolgt von implementierungsspezifischen weiteren Parametern. Das neu kreierte Objekt ist zurückzuliefern (oder `undef`).
`DESTROY` *this*	Diese Methode wird aufgerufen, wenn das assoziative Array von dem Garbage-Collector aufgeräumt wird.
`FETCH` *this*, *key*	Dies ist die Implementierung eines normalen Lese-Zugriffes, bei dem ein Schlüssel gegeben ist. Der entsprechende Wert (bzw. `undef`) ist dann zurückzuliefern.
`STORE` *this*, *key*, *value*	Ein neuer Wert ist für einen bestimmten Schlüssel einzutragen.
`DELETE` *this*, *key*	Analog zu `delete` ist das Element mit dem gegebenen Schlüssel aus dem assoziativen Array zu entfernen.
`CLEAR` *this*	Alle Elemente sind aus dem assoziativen Array zu entfernen. Dies geschieht, wenn z.B. das assoziative Array auf der linken Seite einer Zuweisung steht.

Assoziative Arrays mit einer alternativen Implementierung

`EXISTS` *this*, *key*	Entspricht dem `exists`-Operator und liefert zurück, ob es den Schlüssel gibt.
`FIRSTKEY` *this*	`FIRSTKEY` und `NEXTKEY` werden vom `each`-Operator verwendet. `FIRSTKEY` hat dabei den ersten Schlüssel zurückzuliefern und ...
`NEXTKEY` *this*, *lastkey*	... `NEXTKEY` alle weiteren.

Folgendes Beispiel aus der `perlfunc`-Manualseite illustriert die Verwendung des `tie`-Operators:

```
# print out history file offsets
use NDBM_File;
tie(%HIST, NDBM_File, '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

`NDBM_File` ist ein Modul, das assoziative Arrays auf Basis von ndbm-Dateien implementiert.

DBM-Dateien

- Ken Thompson entwickelte für UNIX Edition VII ein Hash-Verfahren, das ein Dateipaar verwendete (für Schlüssel und die eigentlichen Daten). Die zugehörige Bibliothek wurde unter dem Namen DBM bekannt.

- Analog zur originalen Implementierung von DBM entstanden später eine Reihe ähnlicher Bibliotheken, die allesamt von Perl unterstützt werden:

`ODBM_File`	Schnittstelle zum Uralt-DBM-Bibliothek, falls sie lokal noch existiert (nicht frei verfügbar).
`NDBM_File`	Im Rahmen von BSD verbreitete Fassung von DBM (nicht frei verfügbar).
`SDBM_File`	Freier Nachbau von ndbm, basiert auf dem dynamischen Hash-Verfahren von P.-A. Larson und wurde von Ozan Yigit 1990 entwickelt. Nachteile: Langsam und auf jeweils 1kb begrenzte Datensätze. Vorteil: Wird mit Perl mitgeliefert.
`GDBM_File`	GNU-Variante einer DBM-Bibliothek von Philip A. Nelson, die ebenfalls 1990 entstanden ist. Im Gegensatz zu SDBM unterstützt sie beliebig große Datensätze. Nachteil: Diese Bibliothek wird nicht zusammen mit Perl geliefert.
`DB_File`	Sleepycat Software's Berkeley DB: Unterstützt alternativ B^*-Bäume, dynamische Hash-Tabellen und durchnumerierte Datensätze fester oder variabler Länge. Nachteil: Nicht bei der Perl-Distribution dabei, kann aber unter `CPAN:src/misc/db-2.7.7.tar.gz` organisiert werden.

Beispiel: mkfiledb und findfile

mkfiledb.pl

```
#!/usr/local/bin/perl -w

use Fcntl;
use IO::Dir;
use DB_File;
use strict;

my $home = $ENV{'HOME'};
my $database = "$home/.filedb";
my %db;
tie(%db, 'DB_File', $database, O_RDWR|O_CREAT|O_TRUNC,
    0600, $DB_HASH);
chdir($home) && descend(".");
untie %db;
```

- Problem: Suche aller Dateien eines vorgegebenen Namens im eigenen Dateibaum. Da der ziemlich umfangreich sein kann, ist es sinnvoll (z.B. nachts per Cron-Job) eine entsprechende Datenbank anzulegen (**mkfiledb**), die dann anschließend sehr effizient abgefragt werden kann (**findfile**).

- Zur Implementierung wird hier Sleepycat Software's Berkeley DB verwendet.

- `$DB_HASH` selektiert die Variante einer dynamischen Hash-Tabelle.

Beispiel: mkfiledb und findfile

mkfiledb.pl

```
sub scan { # scan and return all sub directories
   my ($path) = @_;
   my ($file, @dirs) = ();
   my $dir = new IO::Dir $path;
   my $dbpath = $path; $dbpath =~ s{^\./}{};
   return () unless defined $dir;
   while (defined($file = $dir->read)) {
      next if $file =~ /^\.\.?$/;
      if (defined($db{$file})) {
         $db{$file} .= "\0" . $dbpath;
      } else {
         $db{$file} = $dbpath;
      }
      next if -l $file; # skip symbolic links
      push(@dirs, "$path/$file") if -d "$path/$file";
   }
   $dir->close;
   return @dirs;
}

sub descend {
   my ($path) = @_;
   foreach my $dir (scan($path)) {
      descend($dir);
   }
}
```

Beispiel: mkfiledb und findfile

findfile.pl

```
#!/usr/local/bin/perl -w

use Fcntl;
use IO::Dir;
use DB_File;
use strict;

my $cmdname = $0; $cmdname =~ s{.*/}{};
my $usage = "Usage: $cmdname filename\n";
die $usage unless @ARGV == 1;
my $filename = shift;
my $home = $ENV{'HOME'};
my $database = "$home/.filedb";
die "$cmdname: no database found, call mkfiledb first.\n"
   unless -f $database;

my %db;
tie(%db, 'DB_File', $database, O_RDONLY, 0600, $DB_HASH);
if (defined $db{$filename}) {
   my(@paths) = split /\0/, $db{$filename};
   foreach my $path (@paths) {
      print $path, "/", $filename, "\n";
   }
} else {
   die "$cmdname: $filename not found\n";
}
```

ASCII-Dateien als assoziative Arrays

FileDB.pm

```
package FileDB;

use Carp;
use IO::File;
use strict;

sub TIEHASH {
   my ($pack, $filename, $ifs) = @_;
   my $db = new IO::File $filename;
   croak("cannot open $filename: $!") unless defined $db;
   my $self = {
      db => {},
      modified => 0,
      filename => $filename,
      ifs => $ifs,
   };
   while (<$db>) {
      chomp;
      my ($key) = split /$ifs/;
      next unless $key;
      $self->{db}->{$key} = $_;
   }
   $db->close;
   return bless $self, $pack;
}
```

ASCII-Dateien als assoziative Arrays

FileDB.pm

```
sub STORE {
   $_[0]->{db}->{$_[1]} = $_[2];
   $_[0]->{modified} = 1;
}

sub FETCH {
   $_[0]->{db}->{$_[1]};
}

sub FIRSTKEY {
   my $a = scalar keys %{$_[0]->{db}}; each %{$_[0]->{db}};
}

sub NEXTKEY {
   each %{$_[0]->{db}};
}

sub EXISTS {
   exists $_[0]->{db}->{$_[1]};
}

sub DELETE {
   delete $_[0]->{db}->{$_[1]};
   $_[0]->{modified} = 1;
}

sub CLEAR {
   %{$_[0]->{db}} = ();
   $_[0]->{modified} = 1;
}
```

ASCII-Dateien als assoziative Arrays

FileDB.pm

```
sub DESTROY {
   my ($self) = @_;

   if ($self->{modified}) {
      my($filename) = $self->{filename};
      my($backup) = "$filename.backup";
      foreach my $file ($backup, $filename) {
         my $db = new IO::File ">$file";
         croak("cannot open $file for writing\n")
            unless defined $db;
         while (my($key, $val) = each(%{$self->{db}})) {
            print $db $val, "\n";
         }
         $db->close;
      }
      unlink($backup);
   }
}

1;
```

ASCII-Dateien als assoziative Arrays

addresses4.pl

```
#!/usr/local/bin/perl -w

use FileDB;
use strict;

my %addresses;
tie(%addresses, 'FileDB', "addressbook", ":");
foreach my $name (keys %addresses) {
   printf "%-20s | %s\n", $name, $addresses{$name};
}
```

- Das Modul `FileDB` wird bei uns gelegentlich als elegante Zugriffs-Methode für kleine Datenbanken in ASCII-Form verwendet.
- Bei den implementierungsspezifischen Parametern wird zuerst ein Dateiname genannt und anschließend der Feldtrenner.
- Die Implementierung notiert, ob irgendwann etwas geändert worden ist und sichert in Abhängigkeit davon am Ende (bei `DESTROY`) den neuen Stand in die gegebene Datei.
- Das Standard-Modul `Carp` offeriert die Möglichkeit, Fehlermeldungen unter Angabe des Aufrufers auszugeben. Dies ist in Bibliotheks-Modulen interessant, die die Schuld dem fremden Programmierer geben wollen und nicht sich selbst :-)

Eingebettete Dokumentation

hello-world.pl

```
#!/usr/local/bin/perl -w

=head1 NAME

hello-world -- print greeting

=head1 SYNOPSIS

hello-world

=head1 DESCRIPTION

hello-world just prints ``Hello world!'' and exits.

=cut

print "Hello world!\n";
```

- Innerhalb eines Perl-Programmtextes kann an beliebiger Stelle Text für die zugehörige Dokumentation im Pod-Format (*plain old documentation*) eingefügt werden.

- Textabschnitte für die Dokumentation beginnen mit einer =-Direktive am Anfang einer Zeile und enden mit `=cut` (ebenfalls am Anfang einer Zeile).

- Eine verwandte Technik ist die des *literate programming*, die insbesondere durch Donald Knuth bei TEX populär wurde. Jedoch steht bei *literate programming* mehr die Erläuterung des Programmtexts in Vordergrund als die Benutzung.

Eingebettete Dokumentation

```
oberon$ pod2text hello-world.pl
NAME
    hello-world -- print greeting

SYNOPSIS
    hello-world

DESCRIPTION
    hello-world just prints ``Hello world!'' and exits.

oberon$
```

- Das Pod-Format kann in zahlreiche Formate konvertiert werden einschließlich *troff(1)*-Manualseiten, HTML, LaTeX und einfachem ASCII-Text.

Pod-Format

- Text im Pod-Format wird Paragraph für Paragraph bearbeitet und konvertiert. Zwei Paragraphen werden durch eine oder mehrere Leerzeilen voneinander getrennt.

- Es gibt drei Varianten von Paragraphen:

  - Eine Pod-Direktive, die mit = am Anfang der ersten Zeile beginnt.

  - Eingerückter Text, der unverändert übernommen wird (sinnvoll für Programmtext).

  - Am Zeilenanfang beginnender Text, der im Blocksatz gesetzt wird.

- Generell sind die Formatiermöglichkeiten sehr beschränkt und orientieren sich an dem elementaren Bedarf für Manualseiten. Auf der anderen Seite ist es durch die Einfachheit leicht, neue oder alternative Konvertierungswerkzeuge zu schaffen.

Pod-Direktiven

`=head1`	Überschrift, die typischerweise im Manualstil zur Gliederung verwendet wird (NAME, SYNOPSIS, DESCRIPTION etc).
`=head2`	Untergeordnete Überschrift (typischerweise zur Untergliederung eines langen Textes bei DESCRIPTION).
`=over` n	Beginn einer Aufzählung. Manche Konvertierer benötigen dafür die Einrückweite n, die normalerweise auf 4 gesetzt wird.
`=item` $text$	Ein Punkt innerhalb einer Aufzählung. Viele Konvertierer gehen davon aus, daß die Angaben von $text$ innerhalb einer Aufzählung einheitlich sind: Entweder generell `*`, oder nur numerische Aufzählungen (`1.`, `2.` etc) oder Begriffe (z.B. Namen von Funktionen oder Methoden).
`=back`	Ende einer Aufzählung.
`=pod`	Anfang des Textes im Pod-Format.
`=cut`	Ende des Textes im Pod-Format.
`=for` $formatter$	Der Rest des Paragraphen wird nur von dem spezifizierten Formatierer (z.B. HTML) berücksichtigt und ansonsten ignoriert.

- Hinweis: Direktiven sind nur als solche erkennbar, wenn sie einen eigenen Paragraphen bilden. Das heißt, daß sie jeweils von Leerzeilen umgeben werden müssen.

Pod-Sequenzen

I<*text*>	*text* kursiv setzen (Variablen und Hervorhebungen).
B<*text*>	*text* in Fettschrift setzen (Optionen der Kommandozeile und Programmnamen)
S<*text*>	Verhindert Umbruch bei Leerzeichen innerhalb von *text*.
C<*code*>	Programmtext.
L<*name*>	Hypertext-Verweis auf eine andere Manualseite (z.B. `L<perlpod>`) oder auf eine bestimmte Stelle innerhalb einer anderen Manualseite (z.B. `L<perlfunc/print>`).
F<*file*>	Dateiname.
X<*entry*>	Hinzufügen eines Eintrags in den Index.
Z<>	Wird durch leeren Text ersetzt.
E<*escape*>	Einfügen des durch *escape* spezifizierten Sonderzeichens, z.B. `E<auml>`.

Perl und Datenbanken

- Perl bietet eine sehr große Vielfalt an Techniken und Möglichkeiten, auf Daten zuzugreifen. Desweiteren gibt es eine Reihe von Abstraktionstechniken oder vorhandener Schnittstellen.

- Die Kunst liegt darin, ein passendes abstraktes Modell für das eigene Problem zu finden, das durch geeignete Implementierungen und Schnittstellen in effizienter, verläßlicher und flexibler Weise auf bevorzugte Datenbank-Technologien abgebildet werden kann.

- Effizient heißt, daß die Verwendung von zwischenliegenden Abstraktionsebenen und Schnittstellen die Anwendung kaum gegenüber einer Lösung benachteiligen sollte, die direkt eine geeignete Datenbank benutzen würde.

- Verläßlich heißt u.a., daß

  - die Integritätsbedingungen des abstrakten Datenmodells nicht verletzt werden können,

  - Schutz gegen Abstürze oder andere Katastrophen existiert (z.B. durch Transaktionen oder automatisierte Backups) und

  - Fehlerzustände nicht verloren gehen und transparent durch alle Ebenen durchgereicht werden.

- Flexibel bedeutet, daß die im Augenblick geeigneteste Datenbanktechnik jederzeit durch eine andere ersetzt werden kann, ohne daß deswegen die Anwendung geändert werden muß.

Datenbank-Techniken bei Perl

- Daten, die in irgendeiner Form von einem Programm oder einer Person in ASCII-Form abgelegt worden sind. Beispiele: TEX-Quellen mit interessanten Tabellen, Webseiten mit Daten. Nachteile: Nur zur Datenerhebung geeignet, große Abhängigkeit und Empfindlichkeit bezüglich Änderungen des Ausgangsformats.

- Traditionelle ASCII-Dateien, bei denen jede Zeile einen Datensatz repräsentiert und bei denen Felder durch einen Feldtrenner getrennt werden (z.B. `/etc/passwd`). Nachteile: Ineffizient bei größeren Datenmengen, Zeilentrenner und Feldtrenner dürfen in Feldern nicht enthalten sein.

- DBM-Dateien bieten einen effizienten schlüsselbasierten Zugriff (analog zu assoziativen Arrays) zu Datensätzen, die aus einer beliebigen Zeichenfolge bestehen. Nachteile: Ältere Varianten haben Limitierungen bei der maximalen Datensatzlänge, keine vorgegebene Strukturierung der Datensätze.

- Einfache Tabellen, auf die über das Netz zugegriffen wird (z.B. NIS (auch bekannt als YP = *yellow pages*) und NIS+, Zugriff erfolgt hier über SunRPC). Nachteile analog zu denen von DBM-Dateien.

- Relationale Datenbanken mit Unterstützung von SQL. Nachteile: Teilweise erheblicher Overhead bei kleinen Datenmengen oder häufigen Verbindungsaufbauten, nicht alle Datentypen werden von Perl perfekt unterstützt.

- Mit Hilfe des Moduls `Data::Dumper` ist es möglich, beliebige (ggf. auch zirkuläre) Datenstrukturen von Perl in ASCII-Text zu konvertieren und mit `eval` sie wieder aufzubauen.

Abstraktionstechniken für Datenbanken

- Eine wichtige vorgegebene Abstraktion sind die assoziativen Arrays in Perl, die auf Basis von `tie` beliebig implementiert werden können. Es gibt hierfür auch einige Module wie z.B. `Tie::StdHash`, die dies in Standardsituationen erleichtern.

- Grundsätzlich ist es leicht möglich, selbst eine abstrakte Schnittstelle für Daten zu definieren. So liefert z.B. die Ulmer Table-Schnittstelle eine Abstraktion für einfache Tabellen mit benannten Feldern, wovon eine spezifizierte Teilmenge den Primärschlüssel bildet. Die einzelnen Felder dürfen dabei aus beliebig langen Zeichenfolgen bestehen.

- Vorhandene Standards können ebenfalls eine gute Grundlage für Abstraktionen bilden. Bestes Beispiel hierfür ist die Unterstützung von SQL durch das Perl-Modul `DBI`.

Standards bei Datenbanken

- Die Abfragesprache SQL ist der wichtigste Standard für relationale Datenbanken. Zuletzt wurde 1992 der für SQL-2 verabschiedet: ISO/IEC 9075:1992 und ANSI X3.135-1992.

- Zur Zeit werden eine Reihe von Erweiterungen von SQL standardisiert. Die Fertigstellung von SQL-3 wird demnächst erwartet.

- Zu SQL-3 gehört insbesondere die Komponente SQL CLI (Call Level Interface), die eine einheitliche Bibliotheksschnittstelle definiert. CLI wurde bereits 1992 im Rahmen der ODBC-Spezifikation (Microsoft Open Database Connectivity) veröffentlicht. Danach lag die weitere Entwicklung dieses Standards in den Händen der Open Group (früher X/Open), der 1995 als X/Open C451 veröffentlicht wurde (mit Schnittstellen für C und Cobol). Mehr Informationen dazu gibt es unter `http://www.jcc.com/SQLPages/jccs_sql.htm`.

- Darüber hinaus gehen noch die Standardisierungsbemühungen der Object Database Management Group (ODMG), deren aktueller Standard ODMG 2.0 folgende Komponenten einschließt (siehe auch `http://www.odmg.org/`):

  - Datenbanken: SQL (zur Zeit SQL-92)

  - Objekte: OMG (Object Management Group), die einen offenen Standard für Objekte in verteilten Systemen entwickelt, zu der insbesondere CORBA (Common Object Request Broker Architecture) und IDL (Interface Definition Language) gehören (siehe auch `http://www.omg.org/`).

  - Programmiersprachen: Konkrete Bibliotheken sind für C++, Smalltalk und Java vorgegeben.

DBI – generelle Schnittstelle für SQL-Datenbanken

- Schnittstellen für Perl und Datenbanken gab es schon recht früh. Recht populäre wurde z.B. schon Oraperl (Verbindung von Perl 4 mit Oracle) von Tim Bunce.

- Während jede C-Schnittstelle sich mit überschaubaren Aufwand auf Perl übertragen läßt, fehlte doch eine abstrakte und verallgemeinernde Schnittstelle, die die Abhängigkeiten von konkreten Datenbanken vermeidet und sich stattdessen auf den SQL-Standard abstützt.

- Kurz nachdem mit Perl 5 die notwendigen Techniken vorhanden waren, begann Tim Bunce (etwa ab 1995) mit der laufenden Entwicklung von DBI, einer generellen Schnittstelle für SQL-Datenbanken. Allerdings kam es zu häufigen Änderungen dieser Schnittstelle.

- Mittlerweile ist es ein gewisser stabiler Zustand erreicht und es gibt u.a. DBI-Implementierungen für ODBC (Standard, der von einigen Datenbanken unterstützt wird), Adabas, DB2, Empress, Fulcrum, Illustra, Informix, Ingres, mSQL, MySQL, Oracle, Progress, Solid, Sponge, Sybase, XBase und einige mehr.

Frei zugängliche SQL-Implementierungen

- Neben den bekannten kommerziellen Datenbanken, die SQL unterstützen (z.B. Oracle, Informix, DB2 und Sybase), gibt es auch zwei SQL-Implementierungen, die frei über das Netz mitsamt allen Quellen bezogen werden können: Msql (oder Mini SQL) und MySQL.

- Msql wird von Hughes Technologies in Australien entwickelt und war (meines Wissens) die erste SQL-Implementierung, die ihren Schwerpunkt auf die Schnelligkeit für kleinere und mittlere Anwendungen im Web-Bereich setzte.
  Mehr dazu unter `http://www.Hughes.com.au/`.

- MySQL stammt von TCX DataKonsult AB in Schweden und ist als Alternative mit sehr ähnlichen Zielen zu Msql zu sehen.
  Mehr dazu unter `http://www.tcx.se/`.

- Msql und MySQL unterstützen insbesondere Skriptprogrammiersprachen einschließlich Perl.

- Beide kommen mit sehr liberalen Lizenzen (bei Msql ist es für Universitäten, gemeinnützigen Organisationen und privates Studium frei und bei MySQL wird nur bei dem Verkauf von Software, die den MySQL-Server einschließt, Geld verlangt).

- Beide realisieren SQL nur partiell. So werden insbesondere Transaktionen und Konsistenzbedingungen nicht unterstützt (entsprechende Anweisungen werden schlicht ignoriert).

MySQL von TCX DataKonsult AB

- Alle folgenden Beispiele für DBI und SQL basieren auf MySQL.
- Zum Herunterziehen der aktuellen Version (für Ihr Linux-System zu Hause) empfiehlt sich der Mirror der FH Wolfenbüttel: `http://www.fh-wolfenbuettel.de/ftp/pub/database/mysql/`
- Für eine Reihe von Systemen (einschließlich Linux) gibt es fertig übersetzte Fassungen.
- Im Rahmen dieser Vorlesung wird die aktuelle stabile Version 3.22 eingesetzt. Sie ist auch auf der Theseus installiert (unter `/usr/local/mysql`).
- Normalerweise favorisiert MySQL einen Standardport und einen Datenbank-Server pro Maschine (dürfte auch der typische Einsatz sein). Es ist jedoch auch möglich, daß jede Gruppe einen eigenen MySQL-Server startet und ihn für alle Gruppenmitglieder öffnet.
- Hinweise dazu finden Sie auf den Webseiten zur Vorlesung.

Kurzeinführung in SQL

- Nach der Einführung des relationalen Datenbankmodells von Codd (1969 und 1970), gab es zunächst nur wenige experimente Implementierungen, die insbesondere keine umfassende Abfragesprache offerierten.

- Von großer Bedeutung für alle nachfolgenden relationalen Datenbanken und insbesondere SQL war das Forschungssystem System/R, das in der Zeit von 1974 bis 1979 beim IBM San Jose Research Laboratory entwickelt worden ist.

- SQL wurde später von kommerziellen Anbietern relationaler Datenbanken von System/R übernommen und danach auch zu einem formalen Standard:

1986	SQL	ISO/TC 97/SC 21/WG 3 N 117 und ANSI X3.135-1986
1992	SQL-2	ISO/IEC 9075:1992 und ANSI X3.135-1992
1999?	SQL-3	

- Ursprünglich war SQL nur eine reine Abfrage- und Datenmanipulationssprache. Es fehlten Kontrollstrukturen und Prozeduren, die normalerweise bei Programmiersprachen erwartet werden (*computational completeness*). Dies soll sich bei SQL-3 noch ändern.

- Normalerweise wird SQL entweder interaktiv verwendet (über einen sogenannten Monitor, eine Art Shell für eine Datenbank) oder innerhalb von anderen Programmiersprachen (Embedded SQL).

SQL vs MySQL

- MySQL orientiert sich an dem Standard von SQL-2 – bietet jedoch keine vollständige Implementierung.

- Nicht unterstützt werden Transaktionen, verschachtelte `select`-Anweisungen, `select into table`, Trigger, Fremdschlüssel und virtuelle Tabellen (Views).

- Bei einigen nicht unterstützten Konstrukten wird zwar die Syntax der entsprechenden Anweisungen akzeptiert, sie aber im weiteren ignoriert (z.B. bei der Spezifikation von Fremdschlüsseln und weiterer Indizes).

- Ferner unterstützt MySQL nicht das Rechtevergabesystem von SQL-2, sondern nur ein eigenes, das im Prinzip sehr viel eleganter ist, da es auf Tabellen und damit normalen Datenbankmechanismen beruht.

- Einige dieser Einschränkungen haben den Vorteil, daß MySQL deutlich schneller ist als traditionelle Datenbanken. Wirklich schmerzlich ist das Fehlen der verschachtelten `select`-Anweisungen – allerdings versprechen die Entwickler, diese Lücke sehr bald zu schließen. Allerdings steht dieses Versprechen schon seit mehreren Jahren ganz oben auf der Prioritätenliste...

SQL Übersicht (Auswahl)

Tabellenorientierte Anweisungen	
`alter table`	Änderungen der Definition einer Tabelle
`create table`	Erzeugen einer Tabelle
`show columns`	Anzeige der Definition einer Tabelle
`drop table`	Entfernen einer Tabelle mitsamt Inhalt
Tupelorientierte Anweisungen	
`delete`	Löschen von Tupeln
`insert`	Neueinfügen von Tupeln
`select`	Auswahl und Anzeige von Tupeln
`update`	Ändern von Tupeln
`replace`	Ersetzung von Tupeln
Sichten	
`create view`	Definition einer virtuellen Tabelle
`drop view`	Entfernen einer virtuellen Tabelle
Zugriffsbeschleunigung	
`create index`	Anlegen weiterer Indizes
`drop index`	Indizes entfernen
Vergabe von Zugriffsrechten	
`grant`	Zugriffsrechte vergeben
`revoke`	Zugriffsrechte zurücknehmen
Synchronisation	
`lock tables`	Eine Menge von Tabellen exklusiv reservieren
`unlock tables`	Freigabe gesperrter Tabellen
Transaktionen	
`start transaction`	Beginn einer Transaktion
`commit transaction`	Abschluß einer Transaktion
`rollback transaction`	Abbruch der Transaktion
`savepoint`	Definition von Zwischenpunkten, zu denen eine Rückkehr mit `rollback` möglich ist

Datentypen bei SQL (Auswahl)

`integer`	ganze Zahlen (32 Bit)
`float`	Gleitkommazahlen
`char(`n`)`	Zeichenkette mit der festen Länge $n \leq 255$
`varchar(`n`)`	Zeichenkette mit der maximalen Länge $n \leq 255$
`date`	Datum in dem Format `'YYYY-MM-DD'`
`time`	Uhrzeit im Format `'HH:MM:SS'`
`datetime`	Zeitstempel im Format `'YYYY-MM-DD HH:MM:SS'`
`text`	Text mit der maximalen Länge von 65535 Zeichen
`blob`	Binäre Zeichenfolge mit der maximalen Länge von 65535 Zeichen
`enum('value', ...)`	Zeichenkette, die nur einer der angegebenen Werte (einschließlich `null`) annehmen kann.
`set('value', ...)`	Menge aus den angegebenen Werten.

Repräsentierung von Daten

- Relationale Datenbanken bevorzugen aus Performance-Gründen Tabellen fester Länge. Deswegen ergibt sich mit Ausnahme von `text` und `blob` immer eine festgelegte Größe eines entsprechenden Datenfeldes.

- Typischerweise werden bei `text` und `blob` die eigentlichen Daten separat abgelegt. Bei dem entsprechenden Datenfeld verbleibt dann nur noch ein Zeiger.

- Aus diesem Grund unterliegen auch `text` und `blob` der Beschränkung, daß sie nicht indizierbar sind und insbesondere nicht Teil eines Primärschlüssels sein können.

- Nach außen hin ist normalerweise immer nur eine textuelle Repräsentierung der einzelnen Datentypen sichtbar. Dies kommt auch insbesondere Perl entgegen.

- Sofern nicht explizit ausgeschlossen, können alle Datenfelder den Wert `NULL` besitzen (entspricht dem `undef` in Perl).

Syntax von SQL

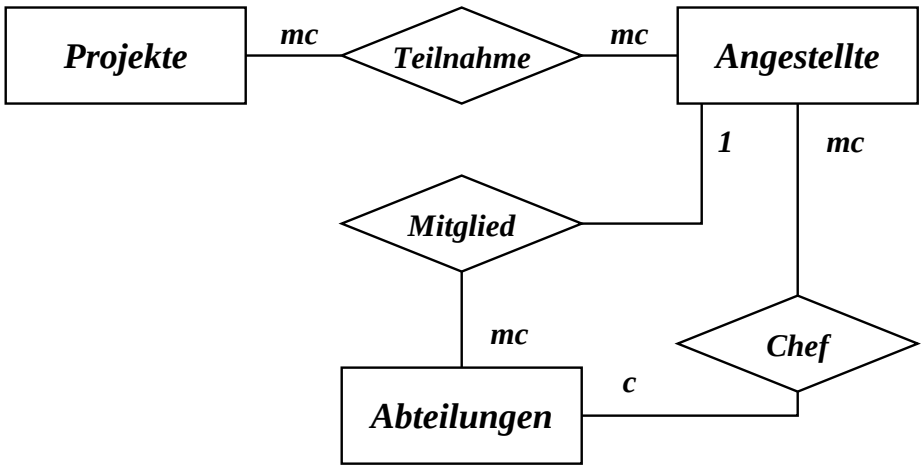
- Alle Schlüsselwörter von SQL werden unabhängig von der Verwendung von Klein- oder Großbuchstaben erkannt, d.h. sowohl `CREATE TABLE` als auch `create table` sind zulässig.

- Bei selbst definierten Namen für Tabellen und Spalten ist die Schreibweise jedoch signifikant.

- MySQL unterstützt mehrzeilige Kommentare analog zu C (`/* ... */`) und zusätzlich analog zu Perl `#` (bis zum Zeilenende). Die im Standard vorgegebene Variante analog zu Ada (beginnend mit `--` bis zum Ende der Zeile) wird aus guten Gründen nicht unterstützt.

- Zeichenketten können entweder in `'...'` oder `"..."` eingeschlossen werden. Innerhalb von `"..."` kann ein `"` durch `""` dargestellt werden (analog zu Pascal). Im übrigen werden einige `\`-Sequenzen akzeptiert einschließlich `\n`, `\t`, `\'`, `\"` und `\\`.

ER-Diagramm für die folgenden Beispiele



- Bei der Abbildung dieses ER-Diagramms in eine relationale Datenbank, wird (zur Einhaltung der dritten Normalform) eine weitere Tabelle benötigt, um die mc-mc-Beziehung zwischen Projekten und Angestellten zu unterstützen.

- Die Tabellen haben dann folgende Attribute (Primärschlüssel sind unterstrichen und die Fremdschlüssel fett gesetzt):

Angestellte	persid, name, **abtid**
Abteilungen	abtid, name, **chef**
Projekte	projektid, name
ProjektTeilnehmer	projektid, persid

Das Anlegen von Tabellen

```
CREATE TABLE table_name ( create_definition,... )

create_definition:
  column_name type [NOT NULL | NULL] [DEFAULT default_value]
                   [AUTO_INCREMENT] [ PRIMARY KEY ]
                   [reference_definition]
  or    PRIMARY KEY ( index_column_name,... )
  or    KEY [index_name] KEY( index_column_name,...)
  or    INDEX [index_name] ( index_column_name,...)
  or    UNIQUE [index_name] ( index_column_name,...)
  or    FOREIGN KEY index_name ( index_column_name,...)
           [reference_definition]
  or    CHECK (expr)

index_column_name:
        column_name [ (length) ]

reference_definition:
        REFERENCES table_name [( index_column_name,...)]
                   [ MATCH FULL | MATCH PARTIAL]
                   [ ON DELETE reference_option]
                   [ ON UPDATE reference_option]

reference_option:
        RESTRICT | CASCADE | SET NULL | NO ACTION |
        SET DEFAULT
```

- Diese und die folgenden Syntax-Angaben wurden der MySQL-Dokumentation entnommen.

Das Anlegen von Tabellen

```
CREATE TABLE Angestellte (
   persid VARCHAR(32) NOT NULL PRIMARY KEY, INDEX (persid),
   name VARCHAR(255) NOT NULL,
   abtid VARCHAR(32) NOT NULL REFERENCES Abteilungen
);

CREATE TABLE Abteilungen (
   abtid VARCHAR(32) NOT NULL PRIMARY KEY, INDEX (abtid),
   name VARCHAR(255) NOT NULL,
   chef VARCHAR(32) REFERENCES Angestellte
);

CREATE TABLE Projekte (
   projektid VARCHAR(32) NOT NULL PRIMARY KEY,
      INDEX (projektid),
   name VARCHAR(255) NOT NULL
);

CREATE TABLE ProjektTeilnehmer (
   projektid VARCHAR(32) NOT NULL REFERENCES Projekte,
      INDEX (projektid),
   persid VARCHAR(32) NOT NULL REFERENCES Angestellte,
      INDEX (persid),
   PRIMARY KEY (projektid, persid)
);
```

- Wenn der Primärschlüssel nur aus einem einzigen Attribut besteht, kann `PRIMARY KEY` direkt bei der Spezifikation des Attributs angegeben werden. Bei nicht-elementaren Primärschlüsseln ist eine extra Definition notwendig, die die beteiligten Attribute aufzählt (siehe `ProjektTeilnehmer`).

Das Anlegen von Tabellen

```
CREATE TABLE Angestellte (
   persid VARCHAR(32) NOT NULL PRIMARY KEY, INDEX (persid),
   name VARCHAR(255) NOT NULL,
   abtid VARCHAR(32) NOT NULL REFERENCES Abteilungen
);
```

- Bei Primärschlüsseln und Fremdschlüsseln mit einem Komplexitätsgrad von 1 ist darauf zu achten, daß `NOT NULL` angegeben wird.

- MySQL (und auch andere dem SQL-2 Standard entsprechende Datenbanken) achten darauf, daß die Primärschlüssel auch wirklich eindeutig bleiben. Wenn diese Eigenschaft auch für andere Attribute gewünscht wird, dann kann dies mit `UNIQUE` angegeben werden.

- Die Angaben von Beziehungen (`REFERENCES`), die zur Erhaltung und Überprüfung der Konsistenz dienen, werden von MySQL ignoriert. Dennoch sind sie alleine schon aus Gründen der Dokumentation sinnvoll.

- Indizes werden von MySQL unterstützt, wenn sie bei `CREATE TABLE` gewünscht werden oder sie später bei `ALTER TABLE` hinzugefügt (oder auch entfernt werden). Im Gegensatz dazu wird die Anweisung `CREATE INDEX` ignoriert.

Das Einfügen von Datensätzen

```
        INSERT INTO table [ (column_name,...) ]
           VALUES (expression,...)
or      INSERT INTO table [ (column_name,...) ] SELECT ...
```

- Mit der ersten Variante wird ein Datensatz eingefügt, während mit der zweiten viele Datensätze auf einmal generiert werden können.
- Normalerweise werden vollständige Tupel angegeben, aber es ist auch möglich, eine Teilmenge zu spezifizieren – der Rest wird dann (soweit zulässig!) auf `NULL` gesetzt.
- Die Reihenfolge der Werte entspricht genau der Reihenfolge der Attributdefinitionen der `CREATE TABLE`-Anweisung.

```
insert into Angestellte values
   ('1', 'Franz Schweiggert', 'SAI');
```

Die Selektion

```
SELECT [STRAIGHT_JOIN] [DISTINCT | ALL] select_expression,...
   [INTO OUTFILE 'file_name' ...]
   [ FROM table_references [WHERE where_definition]
      [GROUP BY column,...]
      [HAVING where_definition]
      [ ORDER BY column [ASC | DESC] ,..]
      [LIMIT [offset,] rows] [PROCEDURE procedure_name]
   ]
```

- `select_expression` besteht im einfachen Falle aus einer durch Kommata getrennten Aufzählung von Feldnamen oder aus *, falls alle Attribute gewünscht werden.

- Werden bei `FROM` mehrere Tabellen angegeben, so können Feldnamen mit dem Namen der Tabelle qualifiziert werden, um Eindeutigkeit herzustellen.

- Statt Feldnamen sind auch Konstanten oder auch kompliziertere Ausdrücke einschließlich eine Reihe interessanter Funktionen möglich.

- `where_definition` ist ein beliebiger Ausdruck mit einem Bool'schen Endresultat. Alle Datensätze bzw. alle Kombinationen von Datensätzen (wenn mehrere Tabellen angegeben sind), für die diese Bedingung zutrifft, werden selektiert und in der von `select_expression` spezifizierten Form zurückgeliefert.

Die Selektion

```
mysql> select * from Angestellte;
+--------+-------------------+-------+
| persid | name              | abtid |
+--------+-------------------+-------+
| 1      | Franz Schweiggert | SAI   |
| 2      | Andreas Borchert  | SAI   |
| 3      | Ingo Melzer       | SAI   |
+--------+-------------------+-------+
3 rows in set (0.01 sec)
```

- `*` ist eine Kurzform für alle Feldnamen der Tabelle `Angestellte`.

```
mysql> select name, abtid from Angestellte;
+-------------------+-------+
| name              | abtid |
+-------------------+-------+
| Franz Schweiggert | SAI   |
| Andreas Borchert  | SAI   |
| Ingo Melzer       | SAI   |
+-------------------+-------+
3 rows in set (0.01 sec)
```

- Natürlich können auch weniger Felder gewünscht werden und die Reihenfolge beliebig bestimmt werden.

Die Selektion

```
mysql> select * from Angestellte where persid >= 2;
+--------+------------------+-------+
| persid | name             | abtid |
+--------+------------------+-------+
| 2      | Andreas Borchert | SAI   |
| 3      | Ingo Melzer      | SAI   |
+--------+------------------+-------+
2 rows in set (0.02 sec)
```

- Bei `where` können beliebige Bedingungen zur Selektion angegeben werden.

```
mysql> select Angestellte.name, Angestellte.abtid
    ->    from Angestellte, Abteilungen
    ->    where Angestellte.persid = Abteilungen.chef;
+-------------------+----------+
| name              | abtid    |
+-------------------+----------+
| Franz Schweiggert | SAI      |
| Frank Stehling    | WiWi     |
| Eduard Wirsing    | Mathe II |
+-------------------+----------+
3 rows in set (0.02 sec)
```

- Bei einer Verknüpfung von zwei (oder mehr Tabellen) wird das kartesische Produkt gebildet und dann entsprechend der `WHERE`-Bedingung selektiert. Dank der Optimierungstechniken relationaler Datenbanken ist dies weniger teuer als es sich anhört. Trotzdem ist es natürlich wichtig, daß für Fremdschlüsselzugriffe entsprechende Indizes vorhanden sind.

Die Selektion

```
mysql> select Angestellte.name, Projekte.name
    ->    from Angestellte, Projekte, ProjektTeilnehmer
    ->    where
    ->    Angestellte.persid = ProjektTeilnehmer.persid and
    ->    ProjektTeilnehmer.projektid = Projekte.projektid;
+-------------------+--------------------------------------+
| name              | name                                 |
+-------------------+--------------------------------------+
| Franz Schweiggert | Implementierung kleiner Datenbanken  |
| Andreas Borchert  | Implementierung kleiner Datenbanken  |
| Ingo Melzer       | Implementierung kleiner Datenbanken  |
| Andreas Borchert  | Ulmer Oberon-System                  |
+-------------------+--------------------------------------+
4 rows in set (0.04 sec)
```

- Diese Abfrage liefert alle Projekte mitsamt ihren Projektteilnehmern.
- Wegen der zwischenliegenden Tabelle `ProjektTeilnehmer` (zur Realisierung der mc-mc-Beziehung), ist dafür eine zweifache Tabellenverknüpfung (Join) notwendig.

Das Ändern von Datensätzen

```
UPDATE table
   SET column=expression,...
   [WHERE where_definition]
```

- Mit `UPDATE` ist es möglich, einzelne Attribute der mit `WHERE` selektierten Tupel zu modifizieren.
- Wenn `WHERE` weggelassen wird, bezieht sich die Operation auf alle Tupel der angegebenen Tabelle.
- Auf der rechten Seite einer Zuweisung bei `SET` kann der Name eines Attributs verwendet werden, der dann für den jeweils alten Wert steht.
- Primärschlüssel sollten nicht auf diese Weise aktualisiert werden, obwohl MySQL (und wohl auch andere relationale Datenbanken) dies akzeptieren. Besser wäre in so einem Fall die `REPLACE`-Anweisung...

```
mysql> update Angestellte
    ->    set abtid = 'WiWi'
    ->    where persid = '3';
Query OK, 1 row affected (0.10 sec)
```

Das Ersetzen von Datensätzen

```
        REPLACE INTO table [ (column_name,...) ]
           VALUES (expression,...)
or      REPLACE INTO table [ (column_name,...) ]
           SELECT ...
```

- `REPLACE` arbeitet analog zu `INSERT`, abgesehen davon, daß zuvor Datensätze mit den gleichen primären Schlüsseln (oder mit `UNIQUE` gekennzeichneten Feldern!) gelöscht werden.

```
mysql> replace into Angestellte values
    >     ('2', 'Kurt Nachfolger', 'SAI');
Query OK, 2 rows affected (0.10 sec)
```

Perl/DBI

- DBI bietet eine elegante Abstraktion für den Zugriff auf Datenbanken, die SQL unterstützen.

- Dennoch besitzt DBI selbst keine Abhängigkeiten von der Abfragesprache. Es wäre somit auch theoretisch möglich, Alternativen zu SQL zu wählen – aber dies wäre eine Verletzung der Abstraktionsgrenze.

- Folgendes gehört zum Umfang von DBI:

  - Operationen zum Aufbau und Abbau einer Datenbankverbindung. Prinzipiell können mehrere Datenbanken gleichzeitig aktiv benutzt werden.

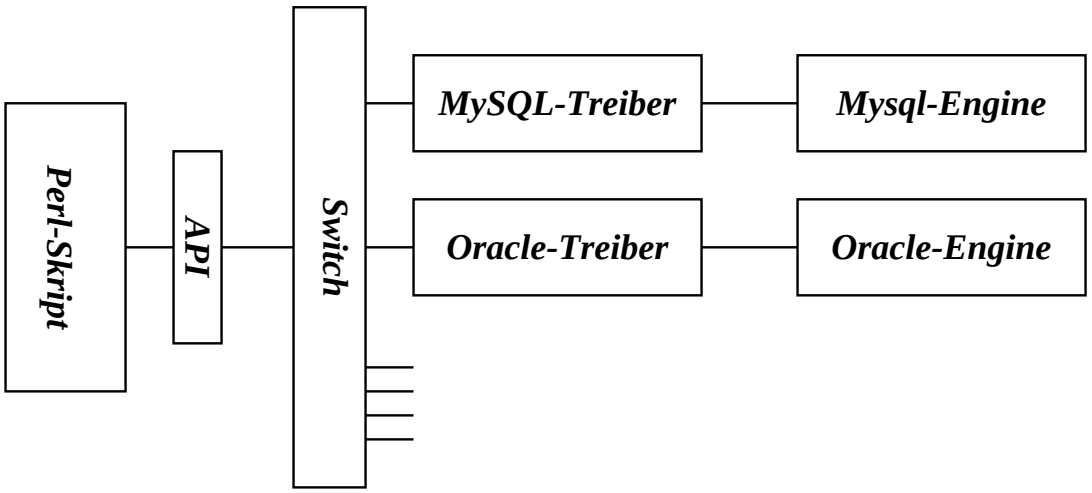
  - Operationen zum Transaktions-Management, wobei in der Voreinstellung (`AutoCommit`) jede erfolgreiche Operation sofort persistent wird.

  - Operationen zur Vorbereitung und Ausführung von SQL-Anweisungen und Mechanismen zum Transfer von Daten von und zur Datenbank.

- Alle folgenden Erläuterungen und Beispiele basieren auf der im Augenblick aktuellen Version 1.13.

- Detaillierte Dokumentation rund um DBI ist unter
  `http://www.mathematik.uni-ulm.de/help/perl5/dbi.html`
  zu finden.

Modulstruktur von DBI



- Die Abstraktion von DBI wird mit `use DBI` importiert. Normalerweise müssen keine weiteren Module für die Benutzung einer konkreten Datenbank importiert werden. Dies erledigt DBI selbst.

- Unterhalb von DBI gibt es einige zu DBI interne Module.

- Alle Implementierungen von DBI zur Unterstützung konkreter Datenbanken befinden sich unter DBD (*database driver*), zum Beispiel: DBD::mysql.

- Das Diagramm wurde (in leicht modifizierter Form) dem DBI-Manual entnommen.

Aufbau einer Verbindung

```
use DBI;

$db = DBI->connect($data_source, $username, $auth);
```

- `DBI->connect` erhält drei Parameter (ggf. auch mehr, siehe Dokumentation), die die gewünschte Datenbank identifizieren und den Benutzernamen mitsamt Passwort für diese Datenbank angeben.
- Die Zeichenkette, die die Datenbank identifiziert,
  - beginnt mit einem festgelegten Format, aus dem sich der Name des Moduls der konkreten Datenbank-Implementierung ableiten läßt, und besteht aus
  - weiteren Text, der datenbank-spezifisch beschreibt, wie sie zu erreichen ist.
- Beispiel: `"dbi:mysql:project:turing:17113"`
- DBI interessiert sich für den vorderen Teil (bis zum zweiten Doppelpunkt, im Beispiel `"dbi:mysql:"`) und sucht nach einem entsprechenden Modul unter `DBD`. Der verbliebene Teil (im Beispiel `"project:turing:17113"`) wird dann mitsamt den weiteren Parametern an den Konstruktor der Implementierung weitergeleitet.
- Wenn es mit dem Eröffnen der Verbindung zur Datenbank nicht klappt, dann ist `$db` undefiniert und in `$DBI::errstr` findet sich die Fehlermeldung.

Die Ausführung von SQL-Anweisungen

```
my $st = $db->prepare(q{
   CREATE TABLE Angestellte (
      persid VARCHAR(32) NOT NULL PRIMARY KEY,
         INDEX (persid),
      name VARCHAR(255) NOT NULL,
      abtid VARCHAR(32) NOT NULL REFERENCES Abteilungen
   )
});
$st->execute(); $st->finish();
```

- Mit `prepare` kann eine SQL-Anweisung zur Ausführung vorbereitet werden. Typischerweise wird sie dabei bereits syntaktisch analysiert und in eine interne Form überführt, die später mit dem dafür zurückgelieferten Objekt referenziert wird (*statement handle*).

- `q{...}` ist äquivalent zu `'...'` – hat aber den Vorteil, daß es sich über mehrere Zeilen erstrecken kann.

- Mit der Methode `execute` kann dann eine SQL-Anweisung zur Ausführung gebracht werden. Interessant ist hier, daß `execute` beliebig oft hintereinander angewandt werden darf, ohne daß bei den weiteren Ausführungen die SQL-Anweisung frisch parsiert werden muß.

- Mit `finish` werden alle Ressourcen freigegeben, die mit der internen Repräsentierung dieser Anweisung zusammenhingen. Wenn dies versäumt wird, kann es zu Warnungen kommen.

- Wenn eine Anweisung nur ein einziges Mal ausgeführt werden soll, geht es auch kürzer: `$db->do(q{...});`

Platzhalter in SQL-Anweisungen

```
my $st = $db->prepare(q{
   INSERT INTO Angestellte VALUES (?, ?, ?)
});
$st->execute(split /:/) while (<>);
$st->finish();
```

- Innerhalb dem Text einer SQL-Anweisung dürfen an Stellen, bei denen die Angabe einer Konstante zulässig wäre, Fragezeichen als Platzhalter eingefügt werden.

- Bei der Ausführung dieser Anweisung können dann die konkreten Werte in Form einer Liste übermittelt werden.

- Diese Methode hat zwei große Vorteile gegenüber der Triviallösung, bei der jedesmal neu eine Anweisung in Perl zusammengebastelt wird:

  - Sie ist deutlich schneller, da die SQL-Anweisung nur ein einziges Mal syntaktisch analysiert werden muß.

  - Es entfällt die nicht ganz triviale Aufgabe, die zu übergebenden Zeichenketten korrekt in Anführungszeichen zu setzen.

- Per Voreinstellung erfolgt die Übergabe von Platzhaltern mit dem Typ `VARCHAR`. Dies ist auch dann kein Problem, wenn für die entsprechenden Felder andere Datentypen (z.B. `INTEGER`) erwartet werden, da SQL genügend tolerant ist. Der Datentyp kann jedoch auch (siehe Dokumentation) explizit spezifiziert werden.

Fehlerbehandlung bei DBI

```
$db->{RaiseError} = 1;

# later ...

eval {
   $db->do($statement);
};
if ($@) {
   print "Fatal error: ", $db->errstr, "\n";
} else {
   print "OK\n";
}
```

- Bei der Voreinstellung muß bei jeder DBI-Operation überprüft werden, ob sie geklappt hat. Das ist mitunter umständlich.
- Mit dem Setzen der Variablen `RaiseError` (beim Datenbank-Objekt) auf `1` ist jeder Fehler fatal, d.h. das Programm würde normalerweise abgebrochen werden.
- Dies kann vermieden werden, indem die "gefährlichen" Anweisungen in `eval` eingebettet werden.
- Wenn nach `eval` in `$@` eine Fehlermeldung vorliegt, dann ist etwas schiefgegangen.
- Mit `$db->errstr` besteht die Möglichkeit, auf die Fehlermeldung von DBI zuzugreifen.

Werkzeug zum Füllen von Datenbanken

filldb.pl

```
#!/usr/local/bin/perl -w

use Getopt::Std;
use DBI;
use Mysql::Admin qw(dbi_connect);
use strict;

my $cmdname = $0; $cmdname =~ s{.*/}{};
my $usage = "Usage: $cmdname [-c] [-d delim] " .
            "dbdir table #rows {file}\n";
my %opts = (); getopts('cd:', \%opts);
my $delim = '\s+';
$delim = $opts{'d'} if defined($opts{'d'});
my $strip_comments = defined($opts{'c'});
die $usage unless @ARGV >= 3;
my $dbdir = shift;
my $table = shift;
my $rows = shift;

my $db = DBI->connect(dbi_connect($dbdir), "", "");
die "$cmdname: unable to connect to db: $DBI::errstr\n"
   unless defined $db;
$db->{RaiseError} = 1;
```

- Mit `Mysql::Admin` können sehr bequem MySQL-Datenbanken eröffnet werden, die nach unserem Schema kreiert worden sind.

Werkzeug zum Füllen von Datenbanken

filldb.pl

```
my $tuple = join(", ", ("?") x $rows);
my $st = $db->prepare("insert into $table values ($tuple)");
while(<>) {
   chomp;
   if ($strip_comments) {
      next if /^\s*#/;
      next if /^\s*$/;
   }
   my @tuple = split /$delim/;
   $st->execute(@tuple);
}

$db->disconnect;
```

- Mit dem `x`-Operator können Listen oder Zeichenketten vervielfältigt werden. `$tuple` enthält somit genau soviel Platzhalter, wie über `$rows` angegeben worden sind.

Das Abholen von Datensätzen

catdb.pl

```
#!/usr/local/bin/perl -w

use Getopt::Std;
use DBI;
use Mysql::Admin qw(dbi_connect);
use strict;

my $cmdname = $0; $cmdname =~ s{.*/}{};
my $usage = "Usage: $cmdname [-d delim] dbdir table\n";
my %opts = (); getopts('d:', \%opts);
my $delim = ' ';
$delim = $opts{'d'} if defined($opts{'d'});
die $usage unless @ARGV == 2;
my $dbdir = shift;
my $table = shift;

my $db = DBI->connect(dbi_connect($dbdir), "", "");
die "$cmdname: unable to connect to db: $DBI::errstr\n"
   unless defined $db;
$db->{RaiseError} = 1;

my $st = $db->prepare("select * from $table");
$st->execute();
my $record;
while (defined($record = $st->fetch())) {
   print join($delim, @{$record}), "\n";
}
$st->finish;

$db->disconnect;
```

Das Abholen von Datensätzen

catdb.pl

```
my $st = $db->prepare("select * from $table");
$st->execute();
my $record;
while (defined($record = $st->fetch())) {
   print join($delim, @{$record}), "\n";
}
$st->finish;
```

- Mit `fetch` kann jeweils ein Datensatz aus einem Resultat einer `SELECT`-Anweisung entgegengenommen werden. `fetch` liefert dabei einen Zeiger auf eine Liste zurück.

- Zu beachten ist, daß die Variablen der zurückgelieferten Liste bei jedem `fetch` aufs Neue überschrieben werden. Wenn der Datensatz länger benötigt wird, muß er in andere Datenstrukturen kopiert werden.

- Alternativ kann auch eine Liste mit `fetchrow_array` oder ein assoziatives Array mit benannten Feldern durch `fetchrow_hashref` zurückgeliefert werden. Dann entfällt das Problem mit dem Überschreiben.

- Ferner ist es möglich, die einzelnen Felder zurückgelieferter Tupel mit eigenen Variablen zu verbinden, die dann automatisch bei jeder `fetch`-Operation gesetzt werden. Details dazu in der Dokumentation von DBI.

Transaktionen

```
$db->{RaiseError} = 1;
$db->{AutoCommit} = 0;

$db->commit;
eval {
   # diverse Datenbank-Operationen
};
if ($@) {
   $db->rollback;
} else {
   $db->commit;
}
```

- In der Voreinstellung (`AutoCommit` ist eingeschaltet), wird jede SQL-Anweisung in einer separaten Transaktion abgehandelt, d.h. nach jeder erfolgreichen Anweisung wird der neue Stand permanent.

- Wenn der Umfang einer Transaktion größer sein soll und dies die Datenbank unterstützt (ist z.B. bei MySQL oder msql nicht der Fall), dann muß `AutoCommit` auf `0` gesetzt werden.

- Unabhängig von der Vorstellung der eigentlichen Datenbank ist aus der Sicht eines Benutzers der DBI-Schnittstelle *immer* eine Transaktion aktiv. Durch die `commit`-Operation wird somit die alte Transaktion beendet und gleichzeitig eine neue begonnen.

- Mit `rollback` kann eine Transaktion abgebrochen werden.

- Zwischenpunkte für `rollback` innerhalb einer Transaktion werden von DBI nicht unterstützt.

Metainformationen einer Datenbank

- Praktisch alle relationalen Datenbank bieten Metainformationen an bezüglich der vorhandenen Datenbanken, Tabellen, Felder (mitsamt Typen, Feldnamen und Schlüsseln).

- Metainformationen sind sehr nützlich bei der Entwicklung von Werkzeugen, die unabhängig von einer konkreten Anwendung operieren können sollen wie z.B. generelle Datenbank-Betrachter.

- In SQL-2 stehen diese Metainformationen standardisiert im Informationsschema zur Verfügung. So lassen sich beispielsweise mit
  `select TABLE_NAME from INFORMATION_SCHEMA.TABLES`
  die Namen aller zugänglichen Tabellen ermitteln. Leider unterstützt MySQL diese Tabellen nicht.

- DBI bietet einige Metainformationen an, die jedoch sehr dürftig sind.

- Viele DBI-Schnittstellen bzw. die dahinterliegenden Datenbanken bieten nicht-standardisierte Formen von Metainformationen an.

Metainformationen einer Datenbank

tables.pl

```
my @tables = ();
my $st = $db->prepare("show tables");
$st->execute;
my $record;
while (defined($record = $st->fetch())) {
   push(@tables, $record->[0]);
}
$st->finish;

foreach my $table (@tables) {
   $st = $db->prepare("show columns from $table");
   $st->execute;
   my @fields = (); my @pkey = ();
   while (defined($record = $st->fetchrow_hashref())) {
      if (defined($record->{'Key'}) &&
          $record->{'Key'} eq "PRI") {
         push(@pkey, $record->{'Field'});
      } else {
         push(@fields, $record->{'Field'});
      }
   }
   print "$table: ";
   print "primary key = (", join(", ", @pkey), "), "
      if @pkey > 0;
   print "other fields = (", join(", ", @fields), ")\n";
}
$st->finish;

$db->disconnect;
```

Metainformationen einer Datenbank

```
mysql> show columns from Abteilungen;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| abtid | varchar(32) |      | PRI |         |       |
| name  | varchar(255)|      |     |         |       |
| chef  | varchar(32) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

- In MySQL liefert
  - `SHOW TABLES` alle Tabellennamen und
  - `SHOW COLUMNS FROM` alle Spalten mitsamt Hinweisen, zu welchen Schlüsseln sie gehören.
- Diese Anweisungen gibt es nicht in SQL-2 – u.a. schon deswegen nicht, weil das Metadatenmodell von SQL-2 deutlich komplizierter ist.
- Beide Abfragen können genauso wie `SELECT`-Anweisungen mit `DBI` bearbeitet werden.

Metainformationen einer Datenbank

- Recht nützlich können noch folgende Attribute eines Abfragehenkels sein (möglicherweise erst nach dem ersten Aufruf von `execute` verfügbar):

`$st->{NUM_OF_FIELDS}`	Anzahl der Felder.
`$st->{NAME}`	Verweis auf eine Liste mit den Namen der einzelnen Felder.
`$st->{NULLABLE}`	Verweis auf eine Liste mit Boolean-Werten, die angeben, ob NULL-Werte bei dieser Spalte zulässig sind bzw. möglicherweise zu erwarten sind.
`$st->{TYPE}`	Verweis auf eine Liste mit ganzzahligen Werten, die jeweils den Typ einer Spalte repräsentieren.
`$st->{PRECISION}`	Verweis auf eine Liste mit der jeweiligen Feldlänge (ggf. in Abhängigkeit des Datentyps zu interpretieren).
`$st->{SCALE}`	Verweis auf eine Liste mit den entsprechenden Werten (normalerweise 0, kann bei numerischen Typen wie z.B. `DECIMAL(5,2)`, positiv sein).
`$st->{NUM_OF_PARAMS}`	Zahl der verwendeten Platzhalter.

Metainformationen einer Datenbank

tables2.pl

```
my @tables = ();
my $st = $db->table_info();
my $record;
while (defined($record = $st->fetchrow_hashref())) {
   push(@tables, $record->{TABLE_NAME});
}
$st->finish;

foreach my $table (@tables) {
   $st = $db->prepare("select * from $table");
   $st->execute;
   my @names = $st->{NAME};
   print "$table: ", join(", ", @{$st->{NAME}}), "\n";
   $st->finish;
}
```

- Die Ermittlung der Tabellennamen läßt sich auch abkürzen zu: `my @tables = $db->tables();`
- Die Angaben über die einzelnen Felder sind leider nicht ohne entsprechende Abfragen zu ermitteln.
- Der Primärschlüssel läßt sich über die Attribute des Abfrage-Henkels leider nicht ermitteln.

DBI im Überblick

`my $db = DBI->connect($data_source,` `  $username, $auth, %attr);`	Aufbau einer Verbindung

`use Mysql::Admin qw(dbi_connect);`
`my $db = DBI->connect(`
`  dbi_connect($dbdir), "", "");`
	Aufbau einer Verbindung zur "privaten" MySQL-Datenbank bei uns (in den Übungen)

`$db->{RaiseError} = 1;`	Alle DBI-Fehler zu fatalen Ausnahmen werden lassen.

`my $tuple = join(", ", ("?") x $rows);`
`my $st = $db->prepare(qq{`
`  insert into $table values ($tuple)`
`});`
`$st->execute(@tuple);`
	Einfügen von Datensätzen vorbereiten
	und durchführen

`my $st = $db->prepare(qq{`
`  select * from $table`
`}");`
`$record = $st->fetch()`
	Abfrage vorbereiten
	Einen Datensatz abholen

`$st->finish;`	Freigabe eines Abfragehenkels
`$db->disconnect;`	Verbindung zur Datenbank "sanft" schließen

Eine Abstraktion für Tabellen

projekte.pl

```
#!/usr/local/bin/perl -w

use TBI;
use strict;

my $ang = TBI->open("Angestellte");
my $teilnehmer = TBI->open("ProjektTeilnehmer");
my $projekte = TBI->open("Projekte");

foreach my $key ($teilnehmer->keys) {
   my %ang = $ang->fetch($key->{'persid'});
   my %projekt = $projekte->fetch($key->{'projektid'});
   printf "%-32s %s\n", $projekt{'name'}, $ang{'name'}, "\n";
}
```

- Für viele Anwendungen kommt eine Abstraktion entgegen, die zwischen der vollen Mächtigkeit relationaler Datenbanken auf der einen Seite und den schlichten assoziativen Arrays auf der anderen Seite angesiedelt ist.

- TBI bietet analog zu DBI eine abstrakte Schnittstelle für Tabellen mit einem Primärschlüssel (darf aus einem oder mehreren Feldern bestehen) und einer beliebigen Anzahl weiterer Felder.

- Im Vergleich zu assoziativen Arrays fällt die Problematik weg, einen Datensatz mit Hilfe eines Feldtrenners selbst zu zerlegen und zusammenzusetzen.

- TBI kann auf recht effiziente Weise auf relationale Datenbanken abgebildet werden – aber auch auf einfache ASCII- oder DBM-Dateien.

Eine Abstraktion für Tabellen

- Bei TBI hat jede Tabelle einen eindeutigen Namen innerhalb eines globalen Namensraums. Typischerweise sind alle Tabellennamen einer Applikation sofort sichtbar, wenngleich erst durch ein `open` eine Verbindung ggf. eröffnet wird.

- Eine Tabelle darf mehrfach eröffnet werden – in diesem Falle wird jeweils der gleiche Zeiger zurückgeliefert.

- Mit `TBI->tables()` ist es möglich, eine Liste der bekannten Tabellen zu erhalten.

- Wenn ein Modul namens `TBI_Scanner` zur Verfügung steht, wird es von TBI während der Initialisierungszeit benutzt, um das Tabellenverzeichnis zu Beginn zu erstellen.

- Eine Tabelle kann mit der Methode `close` geschlossen werden oder auch ganz einfach durch die Garbage Collection aufgeräumt werden.

Eine Abstraktion für Tabellen

projekte.pl

```
foreach my $key ($teilnehmer->keys) {
   my %ang = $ang->fetch($key->{'persid'});
   my %projekt = $projekte->fetch($key->{'projektid'});
   printf "%-32s %s\n", $projekt{'name'}, $ang{'name'}, "\n";
}
```

```
select Angestellte.name, Projekte.name
   from Angestellte, Projekte, ProjektTeilnehmer
   where Angestellte.persid = ProjektTeilnehmer.persid and
         ProjektTeilnehmer.projektid = Projekte.projektid;
```

- Das Beispiel auf Basis von TBI entspricht dem zweifachen Join der SQL-Anweisung.

- Die Methode `keys` liefert analog zu dem `keys`-Operator von Perl alle Schlüssel einer Tabelle. Besteht der Primärschlüssel aus mehr als einem Feld, so wird ein Zeiger auf ein assoziatives Array zurückgeliefert. Ansonsten eben der skalare Wert des einzigen Feldes des primären Schlüssels.

- `fetch` liefert für den (durch einen konkreten Primärschlüssel) selektierten Datensatz alle Felder in Form eines assoziativen Arrays.

- Prinzipiell wird durch TBI in Verbindung mit relationalen Datenbanken der Rechenaufwand höher, da wesentlich mehr SQL-Anweisungen ausgeführt werden müssen. Gewonnen wird jedoch eine sehr kompakte (und von DBI & SQL unabhängige) Schreibweise.

Eine Abstraktion für Tabellen

mitarbeiter.pl

```
#!/usr/local/bin/perl -w

use TBI;
use strict;

my $abteilung = shift @ARGV;
my $ang = TBI->open("Angestellte");

foreach my $key ($ang->select('abtid' => $abteilung)) {
   my %ang = $ang->fetch($key);
   print $ang{'name'}, "\n";
}
```

- Mit Hilfe assoziativer Arrays können leichtere Abfragen, bei denen eine Reihe von Feldern bestimmte Werte haben sollen, leicht formulieren.

- `select` liefert analog zu `keys` Primärschlüssel zurück.

Eine Abstraktion für Tabellen

neuerchef.pl

```
#!/usr/local/bin/perl -w

use TBI;
use strict;

my $abteilung = shift @ARGV;
my $chef = shift @ARGV;

my $ang = TBI->open("Angestellte");
my $abt = TBI->open("Abteilungen");
if ($abt->exists($abteilung) && $ang->exists($chef)) {
   $abt->modify($abteilung, 'chef' => $chef);
}
```

- `exists` liefert **TRUE** zurück, wenn es einen Datensatz mit dem gegebenen Schlüssel gibt.
- Mit `modify` können vorhandene Datensätze modifiziert werden.
- Dabei ist es insbesondere nicht notwendig, alle Datenfelder frisch anzugeben. Es müssen nur die geänderten in Form eines assoziativen Arrays übergeben werden.

Eine Abstraktion für Tabellen

neuesprojekt.pl

```
#!/usr/local/bin/perl -w

use TBI;
use strict;

my $projektid = shift @ARGV;
my $name = shift @ARGV;

my $projekte = TBI->open("Projekte");
$projekte->add($projektid, 'name' => $name);
```

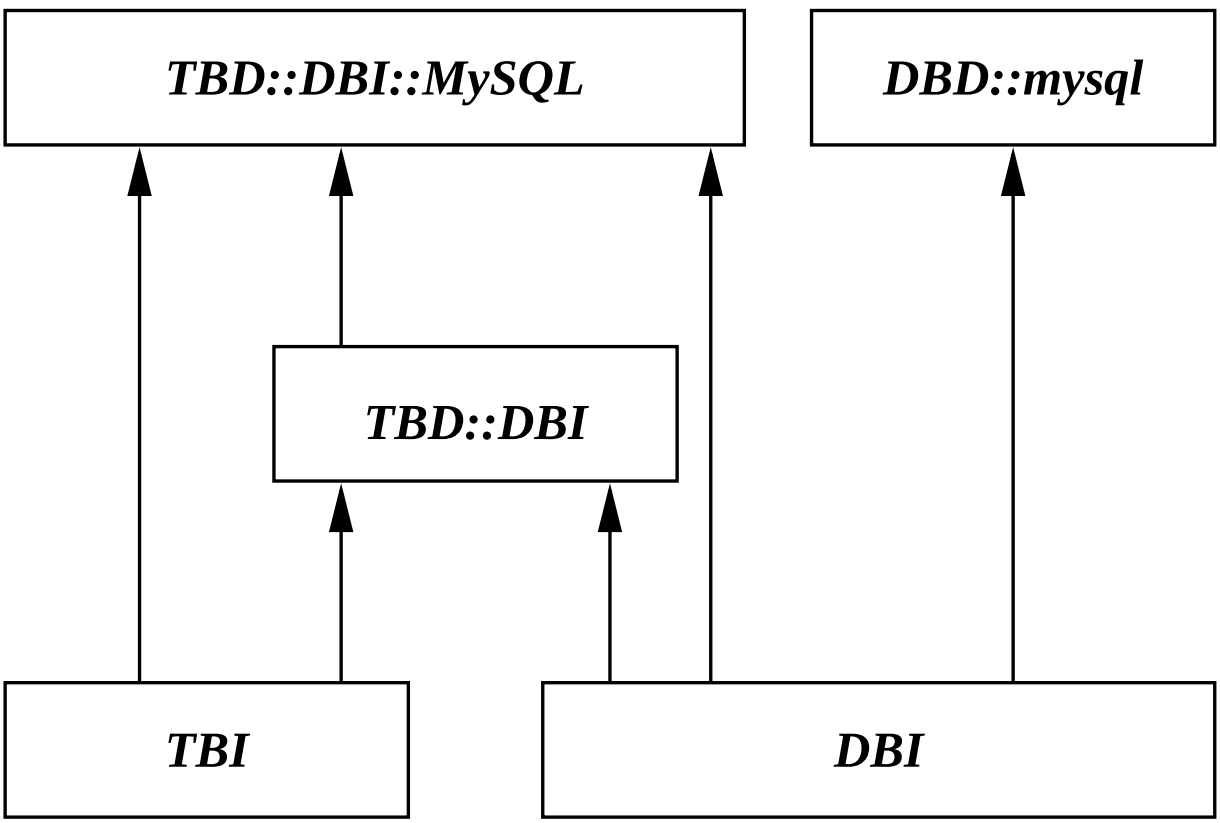
- Mit `add` können neue Datensätze hinzugefügt werden, solange ihr jeweiliger Primärschlüssel noch nicht in der Tabelle eingetragen ist.

- Nach dem Schlüssel (als skalarer Wert) kommen die restlichen Felder in Form eines assoziativen Arrays.

Übersicht von TBI

`$t = TBI->open($name)`	Eröffnen einer bekannten Tabelle
`TBI->tables`	Liste aller bekannten Tabellen
`TBI->exists($name)`	liefert **TRUE**, falls die benannte Tabelle existiert
`$t->name`	liefert den Namen der Tabelle zurück
`$t->fields`	geordnete Liste aller Feldnamen
`$t->keyfields`	geordnete Liste aller Feldnamen des Primärschlüssels
`$t->exists($key)`	liefert **TRUE**, falls ein Datensatz mit diesem Schlüssel existiert
`%fields = $t->fetch($key)`	Zugriff auf einen Datensatz über den Primärschlüssel
`$t->get($key, @fieldnames)`	liefert Liste der benannten Felder des ausgewählten Datensatzes
`$t->getfield($key, $fieldname)`	liefert ein einzelnes benanntes Feld des ausgewählten Datensatzes
`$t->keys`	liefert alle Schlüssel der Tabelle als Liste
`$t->select(%equations)`	liefert alle Schlüssel der Datensätze zurück, für die `%equations` zutrifft
`$t->add($key, %entry)`	Hinzufügen eines Datensatzes
`$t->delete($key)`	Löschen eines Datensatzes
`$t->modify($key, %changes)`	Änderung eines vorhandenen Datensatzes
`$t->close`	Schließen einer Tabelle (optional)

Modulstruktur von TBI



- Analog zu DBI gibt es die abstrakte Schnittstelle und eine Vielzahl von Implementierungen (hier im Beispiel nur für DBI mit dem Namen `TBD::DBI`).

- Wenn eine Implementierung (wie im Falle für DBI) auf einer anderen Abstraktion aufsetzt und diese nicht den vollen gewünschten Umfang bietet, bleibt nichts anderes übrig, als noch eine weitere Ebene anzusetzen (im Beispiel `TBD::DBI::MySQL`), die dies leistet.

Modulstruktur von TBI

- Konkret besteht bei DBI daß Problem, das es keine portable Ermittlung von Meta-Informationen über eine Datenbank gibt (Namen der Tabellen, Felder einer Tabelle, primärer Schlüssel einer Tabelle).

- Wenn längere Ableitungsketten (wie im Beispiel mit `TBI` → `TBD::DBI` → `TBD::DBI::MySQL`) vorkommen können, dann ist es sehr wichtig, daß bereits vom Basismodul (hier TBI) Spielregeln festgelegt werden,

  - wie neue Objekte in kooperativer Weise angelegt und initialisiert werden,

  - wie `DESTROY` (bei der Garbage-Collection) und ggf. explizite Schließ-Operationen abgearbeitet werden,

  - unter welchen Namen private Komponenten bei dem Objekt abgelegt werden dürfen und

  - wie ggf. Registrierungen abgeleiteter Module bei dem Basismodul ablaufen bzw. umgekehrt das Basismodul abgeleitete Module entdeckt.

Initialisierungs-Sequenzen

TBI.pm

```
sub new {
   my $package = shift;
   my $self = bless {'closed' => 0}, $package;
   $self->initialize1(@_);
   $self->initialize2();
   my $name = $self->name();
   if (defined($tables{$name})) {
      $tables{$name}->{'handle'} = $self;
   } else {
      $tables{$name} = {'handle' => $self};
   }
   return $self;
}
```

- Es ist üblich, den Konstruktor (also die Operation, die in Perl das Objekt anlegt und `bless` aufruft), von der eigentlichen Initialisierung zu trennen.

- Bei TBI sind zwei Initialisierungs-Sequenzen vorgesehen:

  - `initialize1` initialisiert alle (interessierten) Parteien in der Ableitungskette – beginnend mit dem Basismodul und abschließend mit dem am weitesten abgeleiteten Modul.

  - Die zweite Sequenz auf Basis von `initialize2` durchläuft gleichfalls alle interessierten Parteien, jedoch diesmal beginnend mit dem am weitesten abgeleiteten Modul und endend mit dem Basismodul.

Initialisierungs-Sequenzen

TBD/DBI.pm

```
sub initialize1 {
   my ($self, %attributes) = @_;

   $self->SUPER::initialize1(%attributes);

   # ...
}

sub initialize2 {
   my ($self) = @_;

   # ...

   $self->SUPER::initialize2();
}
```

- Das Durchlaufen der Kette erfolgt jeweils mit dem Aufruf der entsprechenden mit `SUPER` qualifizierten Methode.

- Je nachdem, ob der Aufruf am Anfang oder am Ende erfolgt, ergibt sich die Richtung, in der die Kette tatsächlich abgearbeitet wird.

- Das Arbeiten mit zwei Ketten wäre nicht notwendig, wenn es eine zu `SUPER` entgegengesetzte Variante gäbe – analog zu `inner` bei der OO-Programmiersprache Beta.

Registrierung bei TBI

TBI.pm

```
my %tables = ();

sub register {
   my ($self, $name, $package, $params) = @_;
   unless (defined($tables{$name})) {
      $tables{$name} = {};
   }
   $tables{$name}->{'package'} = $package;
   $tables{$name}->{'params'} = $params;
}
```

- In der globalen Variable `%tables` verwaltet `TBI` alle bekannten Tabellen:

handle	Zeiger auf das Tabellen-Objekt, falls es bereits eröffnet worden ist.
package	Das Modul, das für diese Tabelle zuständig ist.
params	Die Parameter, die für das Eröffnen der Tabelle notwendig sind.

- Die Methode `register` erlaubt es, Tabellen bekannt zu machen, ohne sie bereits eröffnen zu müssen (was nur unnötig Ressourcen kosten würde).

Eröffnen von Tabellen

TBI.pm

```
sub open {
   my ($package, $name) = @_;
   my $table = undef;
   if (defined($tables{$name})) {
      unless (defined($table =
                  $tables{$name}->{'handle'})) {
         $table = $tables{$name}->{'package'}->
                   new(%{$tables{$name}->{'params'}});
      }
   }
   return $table;
}
```

- Wenn eine Tabelle eröffnet werden soll, wird von der Basisabstraktion TBI überprüft, ob sie bereits offen ist (dann wird der vorhandene Verweis darauf zurückgeliefert) oder ob sie bekannt ist und daher frisch eröffnet werden kann.

- Bei einer Neu-Eröffnung wird der Konstruktor `new` verwendet, der dann die gesamte Initialisierungs-Sequenz zur Folge hat.

Eröffnen von Tabellen

TBD/DBI.pm

```
my $key = "$database/$login/$auth";

my $db;
if (defined($connections{$key})) {
   ++ $connections{$key}->{'refs'};
   $db = $connections{$key}->{'handle'};
} else {
   $db = DBI->connect($database, $login, $auth);
   croak "unable to connect to database $database"
      unless defined $db;
   $connections{$key} = {
      'handle' => $db,
      'refs' => 1,
   };
   $db->{'RaiseError'} = 1;
}
```

- `TBD::DBI` unterhält mit `%connections` ein assoziatives Array für alle offenen Datenbank-Verbindungen, wobei als Schlüssel die Parameter für `DBI->connect` dienen.

- Damit ist es möglich, daß sich mehrere Tabellen eine Datenbankverbindung teilen.

- Um zu entscheiden, wann die Datenbank-Verbindung wieder geschlossen werden kann, gibt es einen Zähler für die Anzahl der Nutzungen.

- Innerhalb von `initialize1` bei `TBD::DBI` wird dann entweder die vorhandene Verbindung übernommen oder eine neu eröffnet und in `%connections` eingetragen.

Eingebettete Initialisierungen

TBD/DBI/MySQL.pm

```
sub initialize2 {
   my ($self) = @_;

   my $st = $self->{'db'}->prepare(qq{
      show index from $self->{'table'}
   });
   $st->execute();
   my @keyfields = ();
   my $record;
   while (defined($record = $st->fetchrow_hashref())) {
      next unless $record->{'Key_name'} eq "PRIMARY";
      $keyfields[$record->{'Seq_in_index'}] =
         $record->{'Column_name'};
   }
   $st->finish();

   shift @keyfields; # Seq_in_index is running from 1
   croak "no primary key defined" unless @keyfields > 0;
   $self->{'keyfields'} = \@keyfields;

   $self->SUPER::initialize2();
}
```

- Durch die beiden Initialisierungs-Sequenzen über `initialize1` und `initialize2` wird die am meisten abgeleitete (und spezialisierte) Implementierung der Abstraktion genau in der Mitte des gesamten Initialisierungsvorgangs aktiv.

Eingebettete Initialisierungen

TBD/DBI.pm

```
sub initialize2 {
   my ($self) = @_;
   croak "Key fields not defined"
      unless ref($self->{'keyfields'});

   my @clauses = ();
   foreach my $keyfield (@{$self->{'keyfields'}}) {
      push(@clauses, "$keyfield = ?");
   }
   $self->{'byKey'} = "where " . join(" and ", @clauses);
   $self->{'select'} = "select * from $self->{'table'} ";
   $self->{'select_byKey'} = $self->{'db'}->prepare(
      $self->{'select'} . $self->{'byKey'});

   # ... more prepared statements

   $self->SUPER::initialize2();
}
```

- `TBD::DBI` und `TBD::DBI::MySQL` kooperieren dahingehend, daß
  - zunächst `TBD::DBI` mit `initialize1` die Datenbank eröffnet und noch die Feldnamen ermittelt (portabel möglich),
  - dann `TBD::DBI::MySQL` kann in `initialize2` (oder bei `initialize1` – das ist egal) die Namen der Felder des primären Schlüssels ermitteln (nicht portabel),
  - worauf `TBD::DBI` in `initialize2` weitere Initialisierungen durchführt, bei denen die Namen der Schlüsselfelder bekannt sein müssen.

Schließen von Tabellen

TBI.pm

```
sub close {
   my $self = shift;
   delete $tables{$self->name()}->{'handle'};
}

sub DESTROY {
   my $self = shift;
   unless ($self->{'closed'}) {
      $self->{'closed'} = 1;
      $self->close();
   };
}
```

- Wenn die Garbage-Collection ein Objekt aufräumen möchte, wird (sofern vorhanden) die Methode `DESTROY` aufgerufen, die es erlaubt, Aufräumungsarbeiten durchzuführen.

- Alternativ ist es auch vorgesehen, eine Tabelle explizit mit `close` zu schließen.

- Wenn eine implizite und eine explizite Schließmöglichkeit parallel existieren, dann muß sichergestellt werden, daß die Aufräumarbeiten nicht doppelt ausgeführt werden. Dies läßt sich gleich in der Basisabstraktion sicherstellen.

- Analog zu den Initialisierungs-Sequenzen ist bei `close` ebenfalls die gesamte Sequenz abzuarbeiten, d.h. alle beteiligten Parteien leisten zuerst ihre eigenen Aufräumarbeiten, um anschließend die entsprechende Methode bei der `SUPER`-Klasse aufzurufen.

Schließen von Tabellen

TBD/DBI.pm

```
sub close {
   my ($self) = @_;

   return unless defined $self->{'db'};

   foreach my $handle (qw(select_byKey select_keys
         insert delete)) {
      $self->{$handle}->finish;
   }

   undef $self->{'db'};
   my $key = $self->{'key'};
   if (--$connections{$key}->{'refs'} == 0) {
      $connections{$key}->{'handle'}->disconnect;
      delete $connections{$key};
   }
   $self->SUPER::close();
}
```

- Bei `TBD::DBI` ist beim Schließen die Verwaltung der Verbindungen zu aktualisieren.
- Wenn der Zähler auf 0 sinkt, dann wird die Verbindung zur Datenbank geschlossen und der entsprechende Eintrag aus `%connections` entfernt.

Generierung von SQL-Anweisungen

TBD/DBI.pm

```
sub fetch {
   my ($self, $key) = @_;

   my @keys = $self->order_keys($key);
   $self->{'select_byKey'}->execute(@keys);
   my $record = $self->{'select_byKey'}->fetch;
   return undef unless defined $record;

   my %fields = (); my $index = 0;
   foreach my $fieldname (@{$self->{'fields'}}) {
      $fields{$fieldname} = $record->[$index++];
   }
   return %fields;
}
```

- Soweit möglich und sinnvoll, wurden von `TBD::DBI::initialize2` alle benötigten SQL-Anweisungen bereits mit `DBI::prepare` vorbereitet, so daß das wiederholte Analysieren von immer gleichen Anweisungen vermieden wird.

- Die private Komponente `select_byKey` enthält eine `SELECT`-Anweisung, bei der alle Werte des Primärschlüssels vorgegeben werden (über Platzhalter), so daß auf diese Weise genau ein gewünschter Datensatz selektiert werden kann.

- `order_keys` ist eine interne Methode, die einen Schlüssel in eine geordnete Liste von Werten verwandelt, die bei `execute` für die Platzhalter übergeben werden kann.

Generierung von SQL-Anweisungen

TBD/DBI.pm

```
sub hash2tuple {
   my ($self, %hash) = @_;

   my @tuple = ();
   foreach my $fieldname (@{$self->{'fields'}}) {
      push(@tuple, $hash{$fieldname});
   }
   return @tuple;
}

sub add {
   my ($self, $key, %fields) = @_;
   if (ref($key)) {
      %fields = (%fields, %{$key});
   } else {
      $fields{$self->{'keyfields'}->[0]} = $key;
   }
   $self->{'insert'}->execute($self->hash2tuple(%fields));
}
```

- Die `INSERT`-Anweisung ist ebenfalls in `TBD::DBI::initialize2` vorbereitet worden, so daß nur noch die Liste mit den Platzhaltern vorzubereiten ist.

- Zunächst wird dazu der Schlüssel zu `%fields` hinzugefügt und dann das assoziative Array mit allen Feldinhalten in eine geordnete Liste konvertiert.

Generierung von SQL-Anweisungen

TBD/DBI.pm

```
sub keys {
   my $self = shift;

   my $st = $self->{'select_keys'};
   $st->execute();
   return $self->return_keys($st);
}

sub return_keys {
   my ($self, $st) = @_;

   my @keys = (); my $record;
   if (@{$self->{'keyfields'}} == 1) {
      while (defined($record = $st->fetch())) {
         push(@keys, $record->[0]);
      }
   } else {
      while (defined($record = $st->fetchrow_hashref())) {
         push(@keys, {%{$record}});
      }
   }
   return @keys;
}
```

- Bei `initialize2` wurde bereits die für `keys` immer konstante SQL-Anweisung vorbereitet und in der internen Komponente `select_keys` abgelegt.

Generierung von SQL-Anweisungen

TBD/DBI.pm

```
sub select {
   my ($self, %clauses) = @_;

   my @clauses = (); my @values = ();
   my ($field, $value);
   while (($field, $value) = each %clauses) {
      push(@clauses, "$field = ?");
      push(@values, $value);
   }
   my @keyfields = @{$self->{'keyfields'}};
   my $keyfields = join(", ", @keyfields);
   my $st = $self->{'db'}->prepare(
      "select $keyfields from $self->{'table'} " .
      " where " . join(" and ", @clauses)
   );
   $st->execute(@values);
   $st->finish;
   return $self->return_keys($st);
}
```

- Bei `select` hängt die SQL-Anweisung direkt von `%clauses` ab, so daß sie jedesmal neu erzeugt wird.

Sicherheitstips für Perl

- Bei jeder Anwendung, die für andere mit eigenen Privilegien läuft, muß streng auf Sicherheit geachtet werden.

- Das betrifft Skripte, die mit einem s-bit ausgestattet sind (das heißt sie laufen unter UNIX mit anderen Privilegien als denen des Aufrufers), und es gilt in ganz besonderem Maße für Netzdienste, da sie für jeden auf dem Internet einen möglichen Angriffspunkt darstellen.

- Neben den üblichen Techniken zur Vermeidung von Programmierfehlern ("-w" auf der Kommandozeile und `use strict`), gibt es hierfür noch eine weitere wertvolle Unterstützung: den "taint mode", der entweder automatisch eingeschaltet wird (bei Skripten mit s-bit) oder auch explizit gewünscht werden kann (Option "-T" auf der Kommandozeile).

- Aber dies allein reicht nicht, da es noch weitere Angriffspunkte gibt, die nicht automatisch verhindert werden können.

Typische Angriffspunkte bei Perl-Skripten

- Die Ausführung externer Kommandos (sei es durch `system` oder auch ganz einfach bei einer Pipeline mit `open`) ist grundsätzlich sehr gefährlich, wenn darauf Einfluß genommen werden kann:

  - Wenn bei auszuführenden Kommandos kein absoluter Pfadname angegeben wird, kann es "Uberraschungen" durch eine manipulierte Umgebungsvariable `PATH` geben oder durch vorgeschobene Kommandos eines Angreifers, die weiter vorne im Pfad vor dem eigentlich gewünschten Kommando liegen.

  - Sowohl bei `open` als auch bei `system` sind beliebige Shell-Metazeichen zugelassen. Wenn eine Einflüßmöglichkeit auf die Zeichenkette existiert, die der Shell übermittelt wird, ist es damit leicht möglich, ein Kommando des Angreifers einzufügen. Beliebt sind hierfür `‘evil command‘` und `;evil command`.

  - Die Shell trennt Kommandozeilen auf Basis der Umgebungsvariablen `IFS` auf. Wenn die z.B. auf den Schrägstrich gesetzt wird, dann wird aus einem absoluten Kommandonamen beispielsweise unerwartet das Kommando `usr`.

Typische Angriffspunkte bei Perl-Skripten

- Wenn Dateien zum Schreiben eröffnet werden, bestehen ebenfalls Risiken,
  - wenn der Dateiname in Abhängigkeit von Angaben eines Angreifers gewählt wird,
  - die zu kreierende Datei in einem vom Angreifer beeinflußbaren Verzeichnis liegt, oder
  - wenn der Dateiinhalt beeinflußt werden kann.
- Eine typische Falle kann hier die Mächtigkeit von `open` darstellen. Während `open(OUT, $filename)` recht harmlos aussieht, wird dies zum idealen Angriffspunkt, wenn beispielsweise `$filename` am Ende ein Pipeline-Zeichen erhalten kann oder wichtige Systemdateien (wie z.B. `/etc/passwd`) bedroht werden können.
- Selbst wenn auf `$filename` kein Einfluß ausgeübt werden kann, kann das Anlegen temporärer Dateien in öffentlichen Verzeichnissen (z.B. `/tmp`) zur tödlichen Falle werden, wenn dort der Angreifer zuvor einen symbolischen Verweis auf `/etc/passwd` für den zu erwartenden Dateinamen hinterließ.
- Natürlich sind auch viele weitere Systemaufrufe gefährlich wie z.B. `mkdir`, `chmod`, `chown` usw.

Typische Angriffspunkte bei Perl-Skripten

- Perl selbst ist zwar immun gegen den Hauptangriffspunkt bei in C geschriebenen Programmen auf Basis von Puffer-Überläufen, jedoch sind möglicherweise in C geschriebene Programme oder hinzugeladene Bibliotheken bedroht.

- Typische Fallen sind hier Umgebungsvariablen (z.B. `HOME`), die auf extrem lange Werte gesetzt werden und damit aufgerufene Shells zur Ausführung mit übergebenen Codes des Angreifers bringen können (diese Technik ist als *stack smashing* bekannt). Weitere Kandidaten sind die Umgebungsvariablen `USER` oder `LOGIN`, bei denen häufig naive Annahmen über deren maximale Länge gemacht wird.

- Weitere Probleme kann es mit (ansonsten durchgeprüften) Kommandozeilenargumente geben, die zu lang sind.

- Auch Eingaben, die über eine Pipeline an ein fremdes Programm erfolgen, können einen Angriffspunkt darstellen, wenn z.B. `gets` statt `fgets` auf der einlesenden Seite verwendet wird.

- Häufig werden solche Sicherheitsaspekte bei Hilfsprogrammen nicht beachtet, da sie alleine genommen kein Sicherheitsrisiko darstellen.

Generelle Abwehrtechniken

- Bei Skripten mit s-bit sind grundsätzlich alle Umgebungsvariablen mit äußerster Vorsicht zu behandeln, wenn irgendwelche Kommandos oder Pipelines gestartet werden. Insbesondere sind `PATH` und `IFS` neu zu setzen und alle anderen Umgebungsvariablen sollten in ihrer Länge begrenzt werden.

- Jedes `open` und jeder Aufruf von `system` sollten ganz genau daraufhin untersucht werden, inwieweit hier Abhängigkeiten von den Eingaben eines Angreifers vorliegen.

- Temporäre Dateien sollten in privaten Verzeichnissen angelegt werden, die für niemanden sonst zugänglich sind oder es sollte mit `O_EXCL` und `O_CREAT` gearbeitet werden (dies empfiehlt sich übrigens generell!).

- Generell sollte überprüft werden, welche Privilegien tatsächlich wie lange notwendig sind:

  - Es ist möglicherweise sinnvoll, den Prozeß (ggf. ab einem bestimmten Zeitpunkt) oder einen Kindprozeß unter einer chroot-Umgebung oder unter einer Benutzerberechtigung laufen zu lassen, die nur sehr wenig Rechte gibt.

  - Ein Prozeß, der mit der Außenwelt in Verbindung steht, kann unter sehr restriktiven Bedingungen laufen und alle gefährlichen Operationen an einen weiteren Prozeß mit normalen Privilegien delegieren. Diese Architektur wird z.B. von smtpd und smtpfwdd von Obtuse verwendet.

Taint-Modus von Perl

- Der Taint-Modus von Perl gibt eine Unterstützung bei der Ausführung des Perl-Skripts, die Benutzereingaben und andere Einflußmöglichkeiten verfolgt.

- Das Prinzip ist einfach: Alle Benutzereingaben und durch Benutzer einflußbaren Variablen sind kontaminiert. Wird eine Variable gesetzt in Abhängigkeit einer kontaminierten Variable, so wird sie selbst kontaminiert.

- Kontaminierte Variablen dürfen nicht verwendet werden, um bestimmte gefährliche Operationen durchzuführen.

- Der Modus wird durch "-T" auf der Kommandozeile oder implizit (durch das s-bit) eingeschaltet.

Kontaminierung von skalaren Variablen

- Folgendes wird von Anfang an als kontaminiert betrachtet:
  - alle Kommandozeilen-Argumente,
  - alle Umgebungs-Variablen,
  - Lokalitätsinformationen (*locale*),
  - Resultate einiger Systemaufrufe wie *readdir(2)* oder *readlink(2)*,
  - das *gecos*-Feld aus der Passwort-Datei (beim Zugriff über die `getpw*`-Prozeduren),
  - Resultate von `‘...‘` und
  - alle Eingaben (von Dateien, Netzwerkverbindungen oder anderen Kanälen).
- Eine Variable, die in Abhängigkeit einer kontaminierten Variable gesetzt wird, wird ebenfalls kontaminiert – selbst wenn die Operation garantiert ungefährlich ist (z.B. weil der ursprüngliche Wert nicht verändert wird).
- Die Kontaminierung ist nur mit einzelnen skalaren Werten verbunden und nicht mit größeren Datenstrukturen. So können bei einer Liste einige Elemente kontaminiert sein, während andere davon frei sind.

Dekontaminierung

```
#!/usr/local/bin/perl -Tw
# ...
my $filename = shift @ARGV; # tainted
unless ($filename =~ /^(\w+)$/) {
   die "Invalid filename: $filename\n";
}
$filename = $1; # no longer tainted
open(OUT, ">$filename") || # OK
   die "Unable to open $filename: $!\n"
# ...
```

- Mit Klammern erfaßte Teile eines regulären Ausdrucks werden als frei von Kontaminierung betrachtet – selbst wenn die Zeichenkette, die dem regulären Ausdruck zugeführt wurde, kontaminiert ist.

- Damit ist es möglich, Zeichenketten nach einer Überprüfung gegen einen regulären Ausdruck normal zu verwenden.

- Natürlich muß darauf geachtet werden, daß dies nicht versehentlich geschieht.

- Wenn ein regulärer Ausdruck nicht der Sicherheitsüberprüfung dient und erkannte Teile davon verwendet werden, sollten sie ggf. als kontaminiert gekennzeichnet werden:

```
if ($address =~ /(.*)@(.*)/) {
   use Taint qw(taint);
   $user = $1; $domain = $2;
   if (tainted $address) {
      taint $user; taint $domain;
   }
}
```

Sicherheitsrelevante Operationen

- Folgende Operationen sind nicht zugelassen, wenn sie von einer kontaminierten Variable abhängen:
  - Kommandos, die direkt oder indirekt eine Shell aufrufen,
  - Operationen, die Dateien oder Verzeichnisse modifizieren und
  - Operationen, die Prozesse beeinflussen.
- Während es somit nicht möglich ist, eine Datei mit einem kontaminierten Namen zum Schreiben zu eröffnen, würde dies sehr wohl lesenderweise gehen, obwohl dies auch ein Risiko darstellen kann (z.B. bei `/etc/shadow`).
- Insofern ist es sehr sinnvoll, selbst bei kritischen Operationen mit Hilfe des `Taint`-Moduls zu überprüfen, ob Abhängigkeiten vorliegen:
  ```
  croak "insecure operation" if tainted $var;
  ```
- Hingegen ist Perl sehr pingelig bei der Ausführung von Kommandos – das ist praktisch unmöglich, solange `$ENV{'PATH'}` nicht auf einen nicht-kontaminierten Wert gesetzt worden ist.

Abschließende Tips

- Betrachten Sie *niemals* Ihre Software als wirklich sicher. Dieser Zustand ist genauso schwer zu erreichen wie 100%-ig fehlerfreie Software.

- Viele Techniken, die der Fehlerreduzierung dienen, lassen sich auch auf die Sicherheitsproblematik übertragen. So sind Audits beispielsweise sehr sinnvoll.

- Grundsätzlich empfiehlt es sich immer, die aktuelle Literatur und insbesondere die aktuellen Sicherheitshinweise zu lesen:

  - `perlsec`-Manualseite.

  - Diverse News-Gruppen, z.B. comp.security.misc, comp.security.unix und de.comp.security.

  - DFN-CERT: `http://www.cert.dfn.de/`

  - The World Wide Web Security FAQ
    `http://www.w3.org/Security/Faq/www-security-faq.html`

  - BUGTRAQ-Mailing-Liste:
    `http://www.securityfocus.com/about/feedback/subscribe-bugtraq.html`
    Webseite: `http://www.SECURITYFOCUS.COM/`

  - Archiv von "Best of Security":
    `ftp://ftp.cyber.com.au/pub/archive/b-o-s/`

  - "Security Tips" bei der Dokumentation von Apache unter "www.apache.org".

  - "Web Security & Commerce" By Simson Garfinkel with Gene Spafford, 1st Edition June 1997,
    ISBN 1-56592-269-7

Netzdienste

- Wenn Dienste über das Netzwerk angeboten und in Anspruch genommen werden, ergeben sich viele Vorteile:

  - Der Dienst kann allen offenstehen und ein direkter Zugang zu dem Rechner, auf dem der Dienst angeboten wird, ist nicht notwendig.

  - Viele Parteien können in kooperativer Weise einen Dienst gleichzeitig nutzen.

  - Der Dienste-Anbieter hat weniger Last, da die Benutzerschnittstelle auf anderen Rechnern laufen kann.

- Andererseits bergen Netzdienste eine Reihe von Risiken und Problematiken:

  - Der Kreis derjenigen, die auf einen Netzdienst zugreifen können, ist möglicherweise ziemlich umfangreich (normalerweise das gesamte Internet).

  - Somit muß jeder Netzdienst Zugriffsberechtigungen einführen und überprüfen und kann sich dabei nicht wie traditionelle Applikationen auf die des Betriebssystems verlassen.

  - Dienste, die gleichzeitig von vielen genutzt werden können, haben vielerlei zusätzliche Konsistenz- und Synchronisierungsprobleme, für die nicht jede Art von Datenhaltung geeignet ist.

  - Netze bringen neue Arten von Ausfällen, wenn eine Netzwerkverbindung zusammenbricht oder es zu längeren "Hängern" kommt.

Netzwerke

- Es gibt eine unüberschaubare Vielfalt an Netzwerk-Hardware, Transport-Protokollen und Schnittstellen zu deren Benutzung. Einen exzellenten Überblick hierfür gibt das Standardwerk "Computer Networks" von Andrew S. Tanenbaum (Prentice Hall, Third Edition, 1996, ISBN 0-13-349945-6).

- Von praktischer Relevanz sind heute insbesondere die Transport-Protokolle des Internets (TCP/IP (verbindungsorientiert) und UDP (Vermittlung einzelner Pakete)).

- Hierfür gibt es im wesentlichen zwei Schnittstellen:

  - Berkeley Sockets, die mit BSD 4.2 im Jahr 1983 eingeführt worden sind und

  - TLI (*Transport Layer Interface*, auf Streams basierend), das mit UNIX System V Release 3.0 im Jahr 1987 verbreitet wurde.

- Durchgesetzt hat sich heute die Socket-Schnittstelle und sie ist auch diejenige, die von Perl bzw. den zugehörigen Bibliotheken unterstützt wird.

Berkeley Sockets

- Das Standardwerk über BSD 4.3 (*The Design and Implementation of the 4.3 BSD UNIX Operating System*, Samuel J. Leffler et al, Addison Wesley, 1989, ISBN 0-201-06196-1) nennt folgende Ziele:

  - **Transparenz**: Die Kommunikation zwischen zwei Prozessen soll nicht davon abhängen, ob sie auf dem gleichen Rechner laufen oder nicht.

  - **Effizienz**: Zu Zeiten von BSD 4.2 (also 1983) war dies ein außerordentlich wichtiges Kriterium wegen der damals noch sehr geringen Rechenleistung. Aus diesem Grund werden insbesondere keine weiteren System-Prozesse zur Kommunikation eingesetzt, obwohl dies zu mehr Flexibilität und Modularität hätte führen können.

  - **Kompatibilität**: Viele bestehende Applikationen und Bibliotheken wissen nichts von Netzwerken und sollen dennoch in einem verteilten Umfeld eingesetzt werden können. Dies wurde dadurch erreicht, daß nach einem erfolgten Verbindungsaufbau (der z.B. von einem anderen Prozeß durchgeführt werden kann) Ein- und Ausgabe in gewohnter Weise (wie bei Dateien, Pipelines oder Terminal-Verbindungen) erfolgen können.

Berkeley Sockets: Abstraktionen

Um den Anforderungen gerecht zu werden, wurden folgende Abstraktionen entwickelt:

- Die Abstraktion eines Kommunikationsbereiches macht es möglich, nicht nur TCP/IP zu unterstützen, sondern auch viele weitere Netzwerke (z.B. Appletalk, DECnet, IPX von Novell). Zu jedem Kommunikationsbereich gibt es unterschiedliche Namen (bzw. Adressen) für Kommunikationsendpunkte. Bei TCP/IP sind das die bekannten 32 Bit langen IP-Adressen (z.B. 134.60.166.1 für die Turing) kombiniert mit der Port-Nummer des einzelnen Dienstes (z.B. 25 für SMTP).

- Die Abstraktion eines Kommunikationsendpunktes (daher der Name "socket"), der mit der eines Dateideskriptors verbunden wird und über den eine bidirektionale Kommunikation möglich ist.

- Die Semantik einer Kommunikation.

Berkeley Sockets: Semantik

- Die Semantik einer Kommunikation umschließt bei jeder Verbindung eine Teilmenge der folgenden Punkte:

  1. Daten werden in der Reihenfolge empfangen, in der sie abgeschickt worden sind.

  2. Daten kommen nicht doppelt an.

  3. Daten werden zuverlässig übermittelt.

  4. Einzelne Pakete kommen in der originalen Form an (d.h. sie werden weder zerstückelt noch mit anderen Paketen kombiniert).

  5. Unterstützung für Nachrichten außerhalb des normalen Kommunikationsstromes (*out-of-band messages*).

  6. Verbindungs-orientierte Kommunikation (damit entfällt die Notwendigkeit, sich bei jedem Paket identifizieren zu müssen).

- Beispiele für unterstützte Varianten:

Name	1	2	3	4	5	6
`SOCK_STREAM`	⋆	⋆	⋆		⋆	⋆
`SOCK_DGRAM`				⋆		
`SOCK_SEQPACKET`	⋆	⋆	⋆	⋆	⋆	⋆
`SOCK_RDM`	⋆	⋆	⋆	⋆		

Berkeley Sockets: Verbindungsaufbau

`s = socket(domain, type, protocol)`
: Anlegen eines neuen Endpunktes: `type` legt die gewünschte Semantik fest innerhalb der Rahmen der Möglichkeiten des ausgewählten Kommunikationsbereiches `domain`. `protocol` kann zur Auswahl eines speziellen Protokolls genutzt werden (ist aber normalerweise 0 und überläßt dies der Implementierung).

`error = bind(s, addr, addrlen)`
: Verknüpfung eines Endpunktes mit einer Adresse `addr` (hängt von dem zuvor ausgewählten Kommunikationsbereich ab) von der Länge `addrlen`.

`error = listen(s, backlog)`
: Wird von einem Diensteanbieter aufgerufen, der auf eingehende Verbindungen wartet. `backlog` gibt die maximale Anzahl von Verbindungswünschen an, die das System zu einem Zeitpunkt akzeptiert.

`fd = accept(s, clientaddr, clientaddrlen)`
: Nachdem ein Verbindungswunsch eingetroffen ist und `listen` zurückkehrte, kann mit `accept` die Verbindung zum Klienten eröffnet werden. In der Variablen `clientaddr` wird die Adresse des Klienten abgelegt und `clientaddrlen` ist ein Zeiger auf die Größenangabe des Adressfeldes, das von `accept` aktualisiert wird.

`error = connect(s, serveraddr, serveraddrlen)`
: Hiermit kann ein Klient seinen Endpunkt mit dem eines Diensteanbieters verbinden.

Berkeley Sockets: Zeitansage

timeserver.c

```
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#define PORT 11011
int main () {
   struct sockaddr_in address, client_addr;
   size_t client_addr_len = sizeof client_addr;
   int sfd, fd; int optval = 1;
   bzero(&address, sizeof(struct sockaddr_in));
   address.sin_family = AF_INET;
   address.sin_port = htons(PORT);
   if ((sfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 ||
        setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
           &optval, sizeof optval) < 0 ||
        bind(sfd, (struct sockaddr *) &address,
             sizeof(struct sockaddr_in)) < 0 ||
        listen(sfd, SOMAXCONN) < 0) {
      perror("socket"); exit(1);
   }
   while ((fd = accept(sfd, (struct sockaddr *) &client_addr,
                       &client_addr_len)) >= 0) {
      char timebuf[32]; time_t clock; time(&clock);
      ctime_r(&clock, timebuf, sizeof timebuf);
      write(fd, timebuf, strlen(timebuf)); close(fd);
   }
}
```

Berkeley Sockets: Ein einfacher Klient

client.c

```
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <unistd.h>
#define PORT 11011
int main (int argc, char ** argv) {
   struct sockaddr_in addr; int fd;
   struct hostent * hp; char * hostname = argv[1];
   char buffer[BUFSIZ]; ssize_t nbytes;

   if ((hp = gethostbyname(hostname)) == NULL) {
      fprintf(stderr, "unknown host: %s\n", hostname);
      exit(1);
   }
   bzero(&addr, sizeof(struct sockaddr_in));
   addr.sin_family = AF_INET;
   bcopy(hp->h_addr, (void *) &addr.sin_addr, hp->h_length);
   addr.sin_port = htons(PORT);
   if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket"); exit(1);
   }
   if (connect(fd, (struct sockaddr *) &addr,
               sizeof addr) < 0) {
      perror("connect"); exit(1);
   }
   while((nbytes = read(fd, buffer, sizeof buffer)) > 0 &&
         write(1, buffer, nbytes) == nbytes);
   exit(0);
}
```

Berkeley Sockets: Hinweise

- Typischerweise wird (wie in dem vorangegangenen Beispiel) der Typ `SOCK_STREAM` bevorzugt. Zwar gibt es auch einige auf `SOCK_DGRAM` basierende Protokolle – jedoch hat die dadurch gewonnene Effizienz ihren Preis in dem zusätzlich erforderlichen Verwaltungsaufwand wegen der Unzuverlässigkeit.

- Im Gegensatz zu `SOCK_SEQPACKET` kommen bei `SOCK_STREAM` einzelne Pakete (also das, was mit einer `write`- oder `read`-Operation geschrieben bzw. gelesen wird) nicht in ihrer originalen Form an – die Implementierung ist frei, sie beliebig zu zerstückeln und zusammenzusetzen. Das erfordert auf der einlesenden Seite eine geschickte Pufferung.

- Wer fleißiger schreibt als der andere lesen kann (das kann auch an der Netzverbindung liegen), wird genauso schlafen gelegt wie eine Partei, die etwas lesen möchte, ohne daß im Augenblick etwas da ist.

- Wenn beide Parteien gleichzeitig lesen (oder genügend viel schreiben) gibt es dementsprechend einen Deadlock. Dies muß durch ein geeignetes Protokoll vermieden werden.

- Mit `select` (oder `poll`) ist es möglich, festzustellen, welche Verbindungen lese- oder schreibbereit sind. Außerdem gibt es einen nicht-blockierenden Modus, bei dem ein Prozeß dann einen Fehler zurückerhält, wenn er andernfalls blockiert worden wäre.

Socket-Schnittstelle bei Perl

timeserver1.pl

```
#!/usr/local/bin/perl -wT

use strict;
use Socket;

my $port = 11011;

socket(SERVER, AF_INET, SOCK_STREAM, 0) ||
   die "Unable to create socket: $!\n";
setsockopt(SERVER, SOL_SOCKET, SO_REUSEADDR, pack("l", 1)) ||
   die "setsockopt failed: $!\n";
bind(SERVER, sockaddr_in($port, INADDR_ANY)) ||
   die "bind failed: $!\n";
listen(SERVER, SOMAXCONN) ||
   die "listen failed: $!\n";
while (defined(my $peer = accept(CLIENT, SERVER))) {
   print CLIENT scalar localtime, "\n";
   close(CLIENT);
}
```

- Prinzipiell stehen auch in Perl alle Systemaufrufe zur Verfügung und die beiden C-Programme können mehr oder weniger direkt übernommen werden.

- Das Modul `Socket` stellt all die zugehörigen Konstanten zur Verfügung wie `AF_INET`, `SOCK_STREAM` usw.

- Mit `pack` ist es möglich, binäre Datenfelder zu erzeugen. So liefert `pack("l", 1)` beispielsweise eine `1` als binäres Feld vom Datentyp `long` in C (daher "l").

Socket-Schnittstelle bei Perl

timeserver2.pl

```
#!/usr/local/bin/perl -wT

use IO::Socket;
use strict;

my $socket = new IO::Socket::INET (
   LocalPort => 11011, Type => SOCK_STREAM,
   Listen => SOMAXCONN, Reuse => 1,
);
die "Unable to setup socket: $!\n" unless defined $socket;
while (defined(my $conn = $socket->accept)) {
   print $conn scalar localtime, "\n";
   $conn->close;
}
```

- `IO::Socket` von Graham Barr bietet eine komfortablere Schnittstelle für den normalen Umgang mit Sockets und unterstützt sowohl TCP/IP und UDP (`IO::Socket::INET`) als auch UNIX-Domain-Sockets (`IO::Socket::UNIX`).

- Der Konstruktor in diesem Beispiel umfaßt vier Systemaufrufe: *socket(2)*, *bind(2)*, *listen(2)* und *setsockopt(2)*.

- Der `Listen`-Parameter geht an den `backlog`-Parameter von *listen(2)*.

- *setsockopt(2)* wird hier benötigt, um den `Reuse`-Parameter zu berücksichtigen. Normalerweise läßt sich eine verwendete Portnummer auch nach Beendigung des Dienstes erst nach Ablauf einer Wartefrist wiederverwenden. Wenn `Reuse` auf `1` gesetzt wird, ist dies dann sofort wieder möglich – und dies ist die normalerweise bevorzugte Variante.

Socket-Schnittstelle bei Perl

client2.pl

```
#!/usr/local/bin/perl -w

use IO::Socket;
use strict;

my $hostname = shift @ARGV;

my $socket = new IO::Socket::INET (
   PeerAddr => $hostname, PeerPort => 11011,
   Type => SOCK_STREAM,
);
die "Unable to open connection: $!\n"
   unless defined $socket;

print while(<$socket>);
```

- Im Gegensatz zur Version in C entfällt hier die Notwendigkeit, einen Rechnernamen in eine IP-Adresse zu konvertieren.
- Im übrigen werden die Systemaufrufe *socket(2)* und *connect(2)* beide von dem Konstruktor übernommen.

Unterstützung paralleler Klienten-Zugriffe

- Das vorgestellte einfache Schema für einen Netzdienst in Perl läßt sich leider nicht ohne weiteres in den allgemeinen Fall überführen, bei dem wir beliebig viele Klienten parallel betreuen.

- Das Problem liegt darin, daß alle anderen Dialoge "hängen", wenn bei einem einzigen Klienten auf Eingabe oder die Möglichkeit, Rückmeldungen loszuwerden, gewartet wird.

- Zu diesem Problem gibt es bei Perl folgende Lösungen:

  - Für jeden Klienten wird ein neuer Prozeß gestartet. Nachteile: Nicht sehr effizient und Datenaustausch zwischen dem Hauptprozeß und den einzelnen Klienten-Prozessen ist nicht mehr trivial.

  - Grundsätzlich werden alle Systemaufrufe vermieden, die den Prozeß blockieren könnten. Ähnlich wie bei Tk gibt es dann eine zentrale Ereignis-Schleife, die für jedes Ereignis (z.B. "es gibt etwas von einem bestimmten Klienten zu lesen") einen zugehörigen Bearbeiter aufruft. Nachteile: Umständlicher Programmierstil und in Kombination mit bestimmten Bibliotheken (z.B. DBI/MySQL) nicht machbar.

  - Modernere UNIX-Versionen bieten Multi-Threading, d.h. mehrere parallel laufende Threads teilen sich einen Adreßraum. Nachteile: Unterstützung dafür bislang bei Perl nur experimentell und jede Menge Code ist noch nicht MT-safe.

Lösung mit mehreren Prozessen

motdserver.pl

```
#!/usr/local/bin/perl -Tw

use Getopt::Std;
use IO::Dir;
use IO::File;
use IO::Socket;
use strict;

my $cmdname = $0;
$cmdname =~ s{.*/}{};
my $usage = "Usage: $cmdname [-p port] messagedir\n";
my %opts = (); getopts('p:', \%opts);
my $port = 11014;
$port = $opts{'p'} if defined($opts{'p'});
die $usage unless @ARGV == 1;
my $dir = shift @ARGV;
die "$cmdname: no such directory: $dir\n" unless -d $dir;

tie my %dir, 'IO::Dir', $dir;
```

- Dieser einfache Netzdienst bietet die Möglichkeit, eine Reihe von Tagesmeldungen, die alle in einem gegebenen Verzeichnis liegen, abzurufen.
- Mit dem Modul `IO::Dir` ist es möglich, ein Verzeichnis als ein assoziatives Array zu betrachten.

Lösung mit mehreren Prozessen

motdserver.pl

```
my $socket = new IO::Socket::INET (
   LocalPort => $port, Type => SOCK_STREAM,
   Listen => SOMAXCONN, Reuse => 1,
);
die "Unable to setup socket: $!\n" unless defined $socket;

$SIG{'CHLD'} = sub { wait() };
```

- An das assoziative Array `%SIG` können Signalbearbeiter zugewiesen werden.
- Das Signal `CHLD` trifft ein, wenn ein Kindprozeß stirbt. Solange für einen terminierten Kindprozeß *wait(2)* nicht aufgerufen wird, belegt es einen Platz in der Prozeßtabelle (als sogenannter Zombie).
- Mit dem eingetragenen Bearbeiter `sub { wait() }` wird dafür gesorgt, daß dieses Schicksal für die Kindprozesse vermieden wird.

Lösung mit mehreren Prozessen

motdserver.pl

```
while (defined(my $handle = $socket->accept)) {
   my $pid;
   if (defined($pid = fork) && $pid == 0) {
      while ((defined(my $cmd = <$handle>))) {
         $cmd =~ s/\r?\n$//m;
         # process $cmd
      }
      $handle->close;
      exit(0);
   } else {
      $handle->close;
   }
}
```

- Für jeden neu hinzukommenden Klienten wird mit *fork(2)* ein neuer Prozeß erzeugt.
- `fork` liefert in Perl
  - `undef`, falls es nicht möglich war, einen neuen Prozeß zu kreieren,
  - 0 beim Kindprozeß und
  - die Prozeß-ID des Kindprozesses beim elterlichen Prozeß.
- Alle Daten und offenen Dateiverbindungen werden vererbt – jedoch nicht gemeinsam gehalten.
- Der Kindprozeß kann dann in Ruhe seinen Dialog durchführen, ohne die Dialoge mit den anderen Klienten zu behindern.

Lösung mit mehreren Prozessen

motdserver.pl

```
# process $cmd
if ($cmd =~ /^$/) {
   foreach my $entry (keys %dir) {
      next if $entry =~ /^\./;
      return_text($handle, $entry);
   }
   finish_text($handle);
} elsif ($cmd =~ /^[^.]/ && exists $dir{$cmd}) {
   my $in = new IO::File "$dir/$cmd";
   unless (defined($in)) {
      return_error($handle, "unable to read $cmd");
      next;
   }
   while(<$in>) {
      chomp;
      return_text($handle, $_);
   }
   finish_text($handle);
} else {
   return_error($handle, "unknown entry: $cmd");
}
```

- Bei einer leeren Zeile werden die Namen aller vorhandenen Dateien ausgegeben und
- bei der Angabe einer Datei wird der Inhalt übermittelt.

Lösung mit mehreren Prozessen

motdserver.pl

```
sub return_text {
   my ($handle, $text) = @_;
   print $handle "-" . $text . "\n";
}

sub finish_text {
   my ($handle) = @_;
   print $handle "OK\n";
}

sub return_error {
   my ($handle, $msg) = @_;
   print $handle "FAILED: $msg\n";
}
```

- Bei Netzwerk-Protokollen empfiehlt es sich,
  - klar Fehlermeldungen von regulären Ausgaben zu unterscheiden,
  - das Ende einer mehrzeiligen Antwort speziell zu markieren (analog zu "over" beim Funken) und
  - klar zwischen dem Fragenden (Klient) und dem Antwortenden (Diensteanbieter) zu unterscheiden, wobei jeder den anderen "ausreden" läßt, bevor etwas Neues gesagt wird.
- Diese Techniken erleichtern es zu verhindern, daß Klienten und Diensteanbieter sich gegenseitig blockieren.

Lösung mit einem Prozeß

```
turing$ telnet turing 11013
Trying 134.60.166.1...
Connected to turing.
Escape character is '^]'.
Your nickname, please.
Andreas
Andreas, you are welcome!
Andreas: has joined the channel.
Hallo zusammen, ist jemand da?
Andreas: Hallo zusammen, ist jemand da?
Robert: Hallo Andreas!
Wolfgang: Nett, Dich zu treffen, Andreas!
quit
Connection closed by foreign host.
turing$
```

- Eine (leider nicht sehr einfache) Lösung auf Basis nur eines Prozesses empfiehlt sich dann, wenn die einzelnen Dialoge mit den Klienten nicht unabhängig voneinander sind wie bei dem in diesem Beispiel vorgestellten Chat-Dämonen.

- Bei diesem Beispiel realisiert der Dämon einen Gesprächskanal, an dem beliebig viele Parteien gleichzeitig teilnehmen können. Jede Zeile, die jemand eintippt, wird an alle Teilnehmer zusammen mit dem Namen des Absenders weitergegeben.

Lösung mit einem Prozeß

chatserver.pl

```
#!/usr/local/bin/perl -Tw

use Getopt::Std;
use IO::Select;
use IO::Socket;
use strict;

my $cmdname = $0;
$cmdname =~ s{.*/}{};
my $usage = "Usage: $cmdname [-p port]\n";
my %opts = (); getopts('p:', \%opts);
my $port = 11013;
$port = $opts{'p'} if defined($opts{'p'});
die $usage unless @ARGV == 0;

my $socket = new IO::Socket::INET (
   LocalPort => $port, Type => SOCK_STREAM,
   Listen => SOMAXCONN, Reuse => 1,
);
die "Unable to setup socket: $!\n" unless defined $socket;
```

- Gestartet wird der Dämon nur ggf. mit der Angabe einer Portnummer, unter der sich die jeweilige Gesprächsrunde treffen soll.

Lösung mit einem Prozeß

chatserver.pl

```
my %handler = (
   $socket => {
      'read' => \&new_participant,
      'handle' => $socket,
   },
);
my $read = new IO::Select;
$read->add($socket);
my $write = new IO::Select;
my %nickname = ();

while (my @sets = IO::Select->select($read, $write)) {
   foreach my $handle (@{$sets[0]}) {
      &{$handler{$handle}->{'read'}}($handle);
   }
   foreach my $handle (@{$sets[1]}) {
      &{$handler{$handle}->{'write'}}($handle);
   }
}
```

- `%handler` ist die zentrale Datenstruktur, die für jede offene Verbindung den aktuellen Status hält.

- Das Modul `IO::Select` unterstützt den Systemaufruf *select(2)*, der den Prozeß blockiert, bis eines von vielen Ereignissen in Zusammenhang mit offenen Verbindungen eintritt (oder ein Zeitlimit verstreicht).

Lösung mit einem Prozeß

chatserver.pl

```
while (my @sets = IO::Select->select($read, $write)) {
   foreach my $handle (@{$sets[0]}) {
      &{$handler{$handle}->{'read'}}($handle);
   }
   foreach my $handle (@{$sets[1]}) {
      &{$handler{$handle}->{'write'}}($handle);
   }
}
```

- `$read` und `$write` sind zwei Objekte von `IO::Select`, die jeweils eine Menge von offenen Verbindungen repräsentieren.
- `IO::Select->select` akzeptiert bis zu vier Parameter:
  - Eine Menge von Verbindungen, bei denen auf Eingabe gewartet wird,
  - eine Menge von Verbindungen, bei denen darauf gewartet wird, daß blockierungsfrei geschrieben werden kann,
  - eine Menge von Verbindungen, bei denen auf das Eintreten von Sonderbedingungen gewartet wird (z.B. *out-of-band* Daten) und
  - ein Zeitlimit.
- `IO::Select->select` liefert dann eine Liste von Zeigern auf Listen mit Verbindungen zurück, für die die Ereignisse eingetreten sind.

Lösung mit einem Prozeß

chatserver.pl

```
sub new_participant {
   my $socket = shift;
   return unless defined (my $handle = $socket->accept);
   $read->add($handle);
   $write->add($handle);
   $handler{$handle} = {
      'read' => \&handle_read,
      'readline' => \&handle_line,
      'write' => \&handle_write,
      'ibuf' => "",
      'obuf' => ["Your nickname, please.\r\n"],
      'nickname' => "",
      'handle' => $handle,
   };
}
```

- `new_participant` wird von der zentralen Ereignis-Schleife aufgerufen als Bearbeiter für die zentrale Socket, wenn neue Verbindungen zum Chat-Dämon eröffnet werden.

- Mit `$read->add($handle)` wird `$handle` in die Menge `$read` aufgenommen.

- Folgende Komponenten gehören zu einer Verbindung in der Datenstruktur `%handler`:

`read`	Bearbeiter für Eingaben.
`readline`	Bearbeiter für vollständige Zeilen.
`write`	wird aufgerufen, wenn eine Zeile ausgegeben werden kann.
`ibuf`	Eingabepuffer.
`obuf`	Ausgabepuffer (Zeiger auf eine Liste mit einzelnen Zeilen).
`nickname`	Der Name des Teilnehmers.
`handle`	Die offene Netzverbindung.

Lösung mit einem Prozeß

chatserver.pl

```
sub handle_write {
   my $handle = shift;
   my $handler = $handler{$handle};
   my $line = shift @{$handler->{'obuf'}};
   $write->remove($handle) unless @{$handler->{'obuf'}} > 0;
   unless (defined(syswrite($handle, $line,
                            length($line)))) {
      quit($handle);
   }
}
```

- Eine offene Verbindung ist genau dann in der Menge `$write` mit dem zugehörigen Bearbeiter `handle_write` vertreten, wenn der Ausgabe-Puffer `obuf` nicht leer ist.

- Bei jedem Aufruf von `handle_write` wird genau eine Zeile ausgegeben. Hierfür wird `syswrite` verwendet, da dies direkt auf *write(2)* zurückgeht und ungepuffert ist.

- Wenn der Ausgabe-Puffer dabei geleert wird, dann wird die Verbindung aus der Menge `$write` entfernt.

- Dieser Lösung liegt die naive Annahme zugrunde, daß die Ausgabe einer Zeile immer blockierungsfrei klappen wird, obwohl nur ein Byte sicher geschrieben werden kann und die Länge der Zeilen nicht begrenzt ist.

- Besser wäre es, hier mit nicht-blockierender Ausgabe zu arbeiten – aber dann wäre auch die Eingabe nicht-blockierend, oder es müßte ständig hin- und hergeschaltet werden.

Lösung mit einem Prozeß

chatserver.pl

```
sub handle_read {
   my $handle = shift;
   my $buffer = "";
   if (defined(sysread($handle, $buffer, 8192))) {
      my $handler = $handler{$handle};
      $handler->{'ibuf'} .= $buffer;
      if ($handler->{'ibuf'} =~ /^(.*\n)(.*)/s) {
         if ($1 =~ /^\r?\n$/) {
            &{$handler->{'readline'}}($handle, "");
         } else {
            foreach my $line (split /\r?\n/, $1) {
               &{$handler->{'readline'}}($handle, $line);
            }
         }
         $handler->{'ibuf'} = $2;
      }
   } else {
      quit($handle);
   }
}
```

- `handle_read` arbeitet mit `sysread`, das direkt auf *read(2)* zurückgeht, um die internen Eingabepuffer von Perl zu umgehen (*select(2)* funktioniert dafür nicht).

- *read(2)* bietet ferner das Entgegenkommen, auch dann weniger zurückzuliefern (nämlich genau das, was im Betriebssystemskern zur Verfügung steht), wenn mehr gefordert wird (hier: 8192 Bytes).

Lösung mit einem Prozeß

chatserver.pl

```
my $handler = $handler{$handle};
$handler->{'ibuf'} .= $buffer;
if ($handler->{'ibuf'} =~ /^(.*\n)(.*)/s) {
   if ($1 =~ /^\r?\n$/) {
      &{$handler->{'readline'}}($handle, "");
   } else {
      foreach my $line (split /\r?\n/, $1) {
         &{$handler->{'readline'}}($handle, $line);
      }
   }
   $handler->{'ibuf'} = $2;
}
```

- Da die Eingabe beliebig fragmentiert sein kann, können unvollständige Zeilen eintreffen oder mehrere Zeilen auf einmal.

- Deswegen wird das Eingelesene (abgelegt in `$buffer`) zuerst an `ibuf` gehängt und anschließend wird überprüft, ob mindestens eine vollständige Zeile damit vorliegt. Wenn ja, werden alle vollständigen Zeilen einzelen an den Bearbeiter, der bei `readline` eingetragen ist, weitergeleitet.

Lösung mit einem Prozeß

chatserver.pl

```
sub handle_line {
   my ($handle, $line) = @_;
   my $handler = $handler{$handle};
   if ($line eq "quit" || $line eq "exit") {
      quit($handle);
   } elsif ($handler->{'nickname'} eq "") {
      $line =~ s/^\s+//;
      $line =~ s/\s+$//;
      if (defined($nickname{$line})) {
         send_msg($handle, "Nickname is in use. " .
                           "Try another one.");
      } elsif ($line =~ /^\w+$/) {
         $handler->{'nickname'} = $line;
         $nickname{$line} = $handler;
         send_msg($handle, "$line, you are welcome!");
         broadcast($line, "has joined the channel.");
      } elsif ($line =~ /^$/) {
         send_msg($handle, "Your nickname, please.");
      } else {
         send_msg($handle, "Nicknames should be " .
                           "alphanumeric, please.");
      }
   } else {
      broadcast($handler->{'nickname'}, $line);
   }
}
```

- Wenn die eingelesene Zeile kein Abschiedskommando ist und auch nicht der beim Anfangsdialog erwartete Name ist, dann wird er an alle Anwesenden mit `broadcast` weitergeleitet.

Lösung mit einem Prozeß

chatserver.pl

```
sub broadcast {
   my ($from, $line) = @_;
   foreach my $handle (keys %handler) {
      send_msg($handler{$handle}->{'handle'},
               "$from: $line");
   }
}

sub send_msg {
   my ($handle, $line) = @_;
   my $handler = $handler{$handle};
   return unless defined $handler->{'obuf'};
   $write->add($handle) if @{$handler->{'obuf'}} == 0;
   push(@{$handler->{'obuf'}}, "$line\r\n");
}
```

- Bei einer Ausgabe an Klienten wird zunächst nur der entsprechende Ausgabe-Puffer ergänzt.
- Dabei ist darauf zu achten, daß die entsprechende Verbindung in die Menge `$write` aufgenommen wird, falls der Ausgabe-Puffer vorher leer war.
- Wenn Zeiger als Schlüssel in einem assoziativen Array verwendet werden (wie hier `$handle` bei `%handler`), dann liefert `keys %handler` keine verwertbaren Zeiger (Restriktion von Perl).

Lösung mit einem Prozeß

chatserver.pl

```
sub quit {
   my $handle = shift;
   my $handler = $handler{$handle};
   $read->remove($handle);
   $write->remove($handle) if @{$handler->{'obuf'}} > 0;
   delete $handler{$handle};
   delete $nickname{$handler->{'nickname'}};
   $handle->close;
   broadcast($handler->{'nickname'}, "has left.")
      if $handler->{'nickname'} ne "";
}
```

- Wenn irgendwelche Fehler auftreten oder eine Verbindung beendet wird, dann räumt `quit` auf.

Webschnittstelle

- HTTP (Hypertext Transfer Protocol) ist die Basis des WWW (World Wide Web — oder auch kurz Web).

- HTTP ist seit 1989 in Benutzung und wurde zunächst von Tim Berners-Lee bei CERN (Europäisches Kernforschungszentrum in der Schweiz) entwickelt.

- In Dezember 1991 gab es eine erste öffentliche Demonstration auf der Hypertext'91-Konferenz in San Antonio, Texas.

- In größerem Rahmen bekannt wurde HTTP 1993 mit der ersten Dokumentation und dem ersten graphischen Klienten Mosaic (entwickelt u.a. von Marc Andreessen am NCSA, dem National Center for Supercomputer Applications, an der University of Illinois).

- Über HTTP können unter anderen Dokumente von einem Webserver abgerufen werden. Vielfach sind Dokumente in HTML (Hypertext Markup Language, wurde in Anlehnung an SGML ebenfalls in der ersten Fassung von Tim Berners-Lee entwickelt) – genauso können aber auch Bilder, PostScript-Texte oder einfache Texte zurückgeliefert werden.

Ein erster Versuch mit HTTP

```
thales$ telnet www.mathematik.uni-ulm.de 80
Trying 134.60.66.5...
Connected to thales.mathematik.uni-ulm.de.
Escape character is '^]'.
GET /
<HTML>
[...]
<HEAD>
<TITLE>Fakult&auml;t f&uuml;r Mathematik
und Wirtschaftswissenschaften</TITLE>
</HEAD>
<BODY>
[... zahlreiche Zeilen herausgenommen ...]
<HR>
<ADDRESS><A HREF="/fak/admin/">Web-Administration</A>,
zuletzt aktualisiert am 02. Dezember 1999</ADDRESS>
[...]
</BODY>
</HTML>
Connection closed by foreign host.
thales$
```

- Die Idee des HTTP-Protokolls ist recht einfach: Mit **GET** wird ein (zum Webserver lokaler) Pfad einer Ressource (typischerweise eine Datei) angegeben, worauf der Webserver im Erfolgsfalle sofort mit dem Inhalt antwortet.

- Nach einer Anforderung und der Übermittlung der Antwort wird die TCP/IP-Verbindung geschlossen.

- Dies entspricht dem HTTP-Protokoll in der Version 0.9. Problem: Welcher Art ist der Text, der zurückgeschickt wird?

HTTP mit Attributen

```
thales$ telnet www.mathematik.uni-ulm.de 80
Trying 134.60.66.5...
Connected to thales.mathematik.uni-ulm.de.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Sun, 09 Jan 2000 16:24:14 GMT
Server: Apache/1.3.0 (Unix)
Vary: accept-language
Last-Modified: Thu, 02 Dec 1999 19:20:50 GMT
ETag: "15311d-10a3-3846c692"
Accept-Ranges: bytes
Content-Length: 4259
Connection: close
Content-Type: text/html
Content-Language: de
Expires: Sun, 09 Jan 2000 16:24:14 GMT

<HTML>
[... weiterer HTML-Text ...]
</HTML>
Connection closed by foreign host.
thales$
```

- Bei HTTP 1.0 (und späteren Versionen) wird bei der Anfrage nicht nur die HTTP-Anweisung (hier **GET**) übermittelt, sondern es besteht auch noch die Möglichkeit, Attribute (insbesondere zur Spezifikation von Präferenzen) zu übermitteln. Beendet wird dies durch eine Leerzeile.

- Die Antwort besteht dann ebenfalls aus Attributen, (wobei hier insbesondere *Content-Type* relevant ist), die von dem eigentlichen Inhalt wiederum durch eine Leerzeile getrennt ist.

HTTP-Anfragen

- Neben **GET** gibt es eine Reihe weiterer Anfragen bei HTTP (siehe RFC 1945):

GET	Abfrage ohne Seiteneffekte
HEAD	Äquivalent zu **GET**, jedoch darf kein Textinhalt zurückgeliefert werden (nur die Attribute)
POST	Übermittlung von Daten an eine Ressource (Seiteneffekte sind zulässig)

- Darüber hinaus können folgende Anfragen unterstützt werden, um über HTTP das Angebot eines Webservers zu ändern:

PUT	Installierung einer neuen Ressource
DELETE	Entfernung einer Ressource

Adressen von Ressourcen im Internet

- Mit den RFCs 1630 und 1738 (aus dem Jahr 1994) wurde eine erweiterbare Form der Adressierung von Ressourcen, genannt URIs (Uniform Resource Indicators), im standardisiert.

- URIs bestehen aus zwei Teilen: Dem Schema (z.B. "http") und einem schema-spezifischen Teil, die durch einen Doppelpunkt voneinander getrennt werden.

- URLs (Uniform Resource Locators) sind URIs, die die vollständige Zugriffsinformation spezifizieren (Rechneradresse, Portnummer, zu übergebende Parameter).

- In vielen Fällen folgen die schema-spezifischen Teile folgendem Muster:

  `//<user>:<password>@<host>:<port>/<url-path>`

user	Benutzername (optional)
password	Passwort (optional)
host	Rechnername oder IP-Adresse
port	Port-Nummer (optional)
url-path	Pfadname (optional)

- Beim HTTP-Protokoll wird nur der Pfadname bei **GET** übergeben (Benutzername und Passwort sind hier in der URL nicht zulässig).

- Sonderzeichen sind in URLs durch ein "von der einer Hexdarstellung des Zeichens, zu kodieren.

Web-Adressen

- Web-Adressen folgen laut RFC 1738 dem Schema
  `http://<host>:<port>/<path>?<searchpart>`
- Wie beim allgemeinen Muster ist die Port-Nummer optional (Voreinstellung ist 80).
- Optional können hinter dem Pfad (also nach Spezifikation der Ressource) Parameter übergeben werden – z.B. Begriffe für eine Suchmaschine.
- Bei Verwendung von mehreren benannten Parametern sind diese in der Form `<parameter>=<value>` anzugeben, wobei "`&`" als Trenner verwendet wird.

Dynamisch generierte Webseiten

- Idee: Statt den statischen Inhalt einer Datei zurückzuliefern, könnte ein Webserver ein Programm die Antwort auf eine Anfrage generieren lassen.

- Problem: Wie werden die URL und die übrigen Parameter an das Programm übermittelt?

- Beim CGI (Common Gateway Interface) wird alles notwendige über Umgebungsvariablen (und bei Verwendung von **POST**) zusätzlich über die Standardeingabe übermittelt.

- Typischerweise wird dann die Ausgabe des Programmes vom Webserver übernommen und in eine vollständige Rückantwort im Rahmen des HTTP-Protokolls eingebettet.

- Alternativ kann auch Programmtext in zurückzuliefernde Dateien eingebettet werden, der vom Webserver selbst interpretiert wird (SSI, PHP und viele andere in HTML eingebette Sprachen sind recht populär).

Ein CGI-Testprogramm

testcgi.pl

```
#!/usr/local/bin/perl -Tw

print "Content-Type: text/html\n\n";
print "<HTML><BODY><H1>Testseite</H1>\n<PRE>\n";
foreach my $param (keys %ENV) {
   print "$param = ", $ENV{$param}, "\n";
}
print "</PRE>\n";

my $nofbytes = $ENV{'CONTENT_LENGTH'};
if (defined $nofbytes) {
   my $input;
   read STDIN, $input, $nofbytes;
   print "<P>Got following input:<P>\n<PRE>\n";
   print $input;
   print "</PRE>\n";
}
print "\n</HTML>\n";
```

- Wenn die Umgebungsvariable **CONTENT-LENGTH** gesetzt ist, wurden per **POST** Daten übermittelt, die über die Standardeingabe zur Verfügung stehen.

Test mit GET

```
thales$ telnet thales 80
Trying 134.60.66.5...
Connected to thales.
Escape character is '^]'.
GET /cgi-bin/testcgi.pl?foo=bar&blubber=blaeh HTTP/1.0

HTTP/1.1 200 OK
Date: Sun, 09 Jan 2000 23:00:15 GMT
Server: Apache/1.3.0 (Unix)
Connection: close
Content-Type: text/html

<HTML><BODY><H1>Testseite</H1>
<PRE>
REMOTE_PORT = 45880
SERVER_SOFTWARE = Apache/1.3.0 (Unix)
GATEWAY_INTERFACE = CGI/1.1
DOCUMENT_ROOT = /www/thales/www/htdocs
SCRIPT_NAME = /cgi-bin/testcgi.pl
SCRIPT_FILENAME = /www/thales/www/cgi-bin/testcgi.pl
REMOTE_ADDR = 134.60.66.5
SERVER_NAME = www.mathematik.uni-ulm.de
REQUEST_URI = /cgi-bin/testcgi.pl?foo=bar&blubber=blaeh
SERVER_PROTOCOL = HTTP/1.0
REQUEST_METHOD = GET
SERVER_PORT = 80
QUERY_STRING = foo=bar&blubber=blaeh
PATH = /usr/sbin:/usr/bin
TZ = MET
SERVER_ADMIN = borchert@mathematik.uni-ulm.de
</PRE>

</HTML>
Connection closed by foreign host.
thales$
```

Test mit POST

- Zuerst wird die Anfrage vollständig übermittelt: Nach der Angabe von **POST**, der URL und der zugehörigen Attribute kommt der durch eine Leerzeile getrennte Text. Die Länge des zu übermittelten Textes muß mit dem Attribut `Content-Length` zuvor spezifiziert werden.

```
thales$ telnet thales 80
Trying 134.60.66.5...
Connected to thales.
Escape character is '^]'.
POST /cgi-bin/testcgi.pl HTTP/1.0
Content-Type: text/plain
Content-Length: 21

Dies ist eine Demo!
```

- Die Antwort gleicht dem vorherigen Test, abgesehen davon, daß `testcgi.pl` nun auch den übermittelten Text auswertet.

Test mit POST

```
HTTP/1.1 200 OK
Date: Sun, 09 Jan 2000 23:20:41 GMT
Server: Apache/1.3.0 (Unix)
Connection: close
Content-Type: text/html

<HTML><BODY><H1>Testseite</H1>
<PRE>
REMOTE_PORT = 46300
SERVER_SOFTWARE = Apache/1.3.0 (Unix)
GATEWAY_INTERFACE = CGI/1.1
DOCUMENT_ROOT = /www/thales/www/htdocs
SCRIPT_NAME = /cgi-bin/testcgi.pl
SCRIPT_FILENAME = /www/thales/www/cgi-bin/testcgi.pl
REMOTE_ADDR = 134.60.66.5
SERVER_NAME = www.mathematik.uni-ulm.de
REQUEST_URI = /cgi-bin/testcgi.pl
SERVER_PROTOCOL = HTTP/1.0
REQUEST_METHOD = POST
SERVER_PORT = 80
QUERY_STRING =
CONTENT_TYPE = text/plain
CONTENT_LENGTH = 21
PATH = /usr/sbin:/usr/bin
TZ = MET
SERVER_ADMIN = borchert@mathematik.uni-ulm.de
</PRE>
<P>Got following input:<P>
<PRE>
Dies ist eine Demo!
</PRE>

</HTML>
```

Formulare in HTML

form.html

```
<HTML>
<HEAD><TITLE>Formulartest</TITLE><HEAD>
<BODY><H1>Formulartest</H1>
<FORM METHOD="POST" ACTION="/cgi-bin/testcgi.pl">
<INPUT TYPE=HIDDEN NAME="foo" VALUE="bar">
<INPUT TYPE=TEXT SIZE="60" NAME="blubber" VALUE="blaeh">
<INPUT TYPE=SUBMIT VALUE="Abschicken">
</FORM>
</BODY>
</HTML>
```

- HTML unterstützt Formulare mit einer reichen Auswahl von möglichen Eingabe-Varianten, bei denen Parameternamen mit Parameterwerten verknüpft werden, die (mit Ausnahme vom Typ HIDDEN) vom Benutzer editierbar sind.
- Als Übergabemethode ist sowohl **GET** als auch **POST** zulässig.
- Wenn **POST** verwendet wird, wird als `Content-Type` normalerweise `application/x-www-form-urlencoded` verwendet, bei der die gleiche Kodierung wie bei URLs zum Einsatz kommt.
- Die folgende Seite zeigt die in ASCII konvertierte Ausgabe von testcgi.pl, wobei überlange Zeilen gekürzt wurden.

Formulare in HTML

```
Testseite

SERVER_SOFTWARE = Apache/1.3.0 (Unix)
GATEWAY_INTERFACE = CGI/1.1
DOCUMENT_ROOT = /www/thales/www/htdocs
REMOTE_ADDR = 134.60.8.88
SERVER_PROTOCOL = HTTP/1.0
REQUEST_METHOD = POST
HTTP_REFERER = http://www.mathematik.uni-ulm.de/[...]
QUERY_STRING =
HTTP_USER_AGENT = Mozilla/4.04 [en] ([...])
PATH = /usr/sbin:/usr/bin
TZ = MET
HTTP_CONNECTION = Keep-Alive
HTTP_ACCEPT = image/gif, image/x-xbitmap, [...]
REMOTE_PORT = 36154
HTTP_ACCEPT_LANGUAGE = en
SCRIPT_NAME = /cgi-bin/testcgi.pl
SCRIPT_FILENAME = /www/thales/www/cgi-bin/testcgi.pl
SERVER_NAME = www.mathematik.uni-ulm.de
REQUEST_URI = /cgi-bin/testcgi.pl
HTTP_ACCEPT_CHARSET = iso-8859-1,*,utf-8
SERVER_PORT = 80
CONTENT_TYPE = application/x-www-form-urlencoded
CONTENT_LENGTH = 21
HTTP_HOST = www.mathematik.uni-ulm.de
SERVER_ADMIN = borchert@mathematik.uni-ulm.de

Got following input:

foo=bar&blubber=blaeh
```

Das CGI-Modul

- Das Modul `CGI` von Lincoln Stein ist relativ verbreitet, wird auf allen Plattformen von Perl unterstützt und ist recht populär.

- Das Modul setzt sich zum Ziel, daß die Generierung eines Web-Formulars und die Bearbeitung desselben in einem Skript erfolgt. Das stellt sicher, daß beides nicht getrennt gepflegt werden muß.

- Typischerweise werden eine ganze Reihe zusammenhängender dynamischer Seiten durch ein einziges Skript erzeugt, das in Abhängigkeit von den Parametern und ggf. einem Sitzungs-Zustand die Art der zu erzeugenden Seite auswählt.

- Es gibt auch Alternativen, z.B. `CGI_Lite` von Shishir Gundavaram.

Such-Index für E-Mail-Adressen

- Wir haben zur Zeit knapp 2000 Benutzer auf den Suns der Mathematik registriert und setzen werktäglich ca. 8.000 E-Mails um. Entsprechend gibt es Bedarf nach einer effizienten Suchmöglichkeit für E-Mail-Adressen.

- Die Suche soll über den natürlichen Namen (oder Teile davon) oder den Benutzernamen möglich sein.

- Die dafür notwendigen Daten sind bei uns in zwei NIS-Tabellen: passwd (Benutzername, natürlicher Name) und mail.aliases: (E-Mail-Adresse, Weiterleitung).

- So sehen beispielsweise meine Einträge aus: In passwd:
  `borchert:x:120:200:Andreas Borchert:/home/thales/borchert:/bin/sh`
  In mail.aliases:
  `Andreas.Borchert: borchert`
  `borchert: borchert@thales`

- Beide Tabellen zusammen belegen etwa ein Viertel-Megabyte und müssten jeweils zum vollständigen Durchsuchen über das Netzwerk bezogen werden.

- Da dies zu aufwendig ist, lohnt es sich, spezielle Indizes (auf Basis von DBM-Dateien) lokal regelmäßig zu erzeugen, um eine effiziente Suche zu ermöglichen.

- Die folgende Lösung ist seit mehreren Jahren unter
  `http://www.mathematik.uni-ulm.de/cgi-bin/email`
  im Einsatz.

Such-Index für E-Mail-Adressen

- Aus den beiden NIS-Tabellen passwd und mail.aliases läßt sich eine Tabelle erzeugen mit jeweils einer gültigen E-Mail-Adresse und dem zugehörigen vollen Namen (soweit bekannt – es gibt auch E-Mail-Adressen bei uns ohne einen zugehörigen Zugang).

- Daraus werden zwei DBM-Dateien (auf Basis der Berkeley DB) erzeugt:

`byFullName.db`	indiziert nach dem vollen Namen, als Wert jeweils die zugehörige E-Mail-Adresse
`byKey.db`	indiziert nach Namensteilen (Vornamen, Nachnamen), als Wert die Liste der zugehörigen vollen Namen

- Das CGI-Skript überprüft dann, ob ein vorgegebener Suchbegriff entweder ein bekannter Namensteil (Tabelle `byKey.db`) oder vollständiger Name (Tabelle `byFullName.db`) ist oder einen bekannten Namensteil enthält und liefert dann die Liste der zugeörigen E-Mail-Adressen (mit Hilfe der Tabelle `byFullName.db`) zurück.

Erzeugung der Indizes

createmaildb.pl

```
#!/usr/local/bin/perl -w

use DB_File;
use IO::File;
use Fcntl;
use strict;

my $sep = ":"; # separator in keyword index

my $cmdname = $0; $cmdname =~ s{.*/}{};
my $usage = "Usage: $cmdname\n";
die $usage unless @ARGV == 0;

my %pwnames = (); # set by load_passwd, used by load_aliases
my @aliases = (load_passwd(), load_aliases());

my %fullname;
tie(%fullname, 'DB_File', "byFullName.db",
   O_RDWR|O_CREAT|O_TRUNC, 0660, $DB_HASH);
foreach my $alias (@aliases) {
   $fullname{$alias->{fullname}} = $alias->{email};
}
untie %fullname;
```

- Sowohl `load_passwd()` als auch `load_aliases()` liefern Einträge mit jeweils einem vollen Namen (Komponente *fullname*) und der zugehörigen E-Mail-Adresse (Komponente *email*).

- Die Tabelle `byFullname.db` läßt sich direkt daraus generieren.

Erzeugung der Indizes

createmaildb.pl

```
my %entries = ();
foreach my $alias (@aliases) {
   add($alias->{email}, $alias->{fullname});
   foreach my $name (split /\s/, $alias->{fullname}) {
      add($name, $alias->{fullname});
      if (lc($name) ne $name) {
         add(lc($name), $alias->{fullname});
      }
   }
}

my %keyword;
tie(%keyword, 'DB_File', "byKey.db",
   O_RDWR|O_CREAT|O_TRUNC, 0660, $DB_HASH);
foreach my $key (keys %entries) {
   eval {
      $keyword{$key} = join($sep, keys %{$entries{$key}});
   };
   print "Error: $@\n" if $@;
}
untie %keyword;

sub add {
   my ($key, $fullname) = @_;
   ${$entries{$key}}{$fullname} = 1;
}
```

- Da bei populären Namen Listen von zugehörigen vollen Namen ziemlich lange werden können, wurde die entsprechende Zeile, die solche Einträge in die DBM-Datei einfügt, durch ein `eval` geschützt. Bei einer Berkeley DB müßte es normalerweise klappen – aber bei anderen DBM-Dateien nicht unbedingt.

Erzeugung der Indizes

createmaildb.pl

```
sub load_passwd {
   my @aliases = ();
   my $passwd = new IO::File "ypcat passwd |";
   die "$cmdname: unable to process passwd table: $!"
      unless defined $passwd;
   while (<$passwd>) {
      my @entry = split/:/;
      next unless $entry[2] >= 100;
      my $email = $entry[0];
      my @name = split(/\s+/, $entry[4]);
      next unless @name >= 2;
      my $fullname = join(" ", @name);
      push(@aliases, {
         'fullname' => $fullname,
         'email' => $email,
      });
      $pwnames{$email} = $fullname;
   }
   $passwd->close;
   return @aliases;
}
```

- `load_passwd` wertet die passwd-Tabelle aus. Im Feld 0 steht der Benutzername, der bei uns immer eine gültige E-Mail-Adresse ist, und in Feld 4 ist der vollständige natürliche Name zu finden (jedoch mit Umlauten in Ersatzdarstellung).

Erzeugung der Indizes

createmaildb.pl

```
sub load_aliases {
   my @aliases = ();
   my $aliases = new IO::File "ypcat -k mail.aliases |";
   die "$cmdname: unable to process mail.aliases table: $!"
      unless defined $aliases;
   while(<$aliases>) {
      chomp;
      s/\s*,\s*/,/g;
      my ($key, $list) = split /\s+/;
      next unless $key =~ /\./;
      my @list = split(/,/, $list);
      next unless @list == 1;
      next if defined $pwnames{$list};
      my $name = $key; $name =~ s/\./ /g;
      $name =~ s/(^|[ -])([a-z])/$1\u$2/g;
      my @name = split(/ /, $name);
      my $fullname = join(" ", @name);
      push(@aliases, {
         'fullname' => $fullname, 'email' => $key,
      });
   }
   $aliases->close;
   return @aliases;
}
```

- `load_aliases` wertet die Tabelle mail.aliases aus. Im Feld 0 steht die E-Mail-Adresse, im Feld 1 die Adresse, an die entsprechende E-Mails weiterzuleiten sind. Gesucht sind hier Alias-Namen, die einem vollem Namen entsprechen (also mit Punkten separiert sind).

Das suchende CGI-Skript

email.pl

```
#!/usr/local/bin/perl -Tw
#-------------------------------------------------
# cgi-bin-Skript fuer die Suche nach E-Mail-Adressen
# afb 4/97
#-------------------------------------------------

use CGI;
use DB_File;
use Fcntl;
use strict;

my $dbdir = "/www/thales/www/src/email";
my $domain = "\@mathematik.uni-ulm.de";
my $admin = "postmaster$domain";

my $hinweise = <<'ENDE_DER_HINWEISE';
Dieser Zugang erlaubt die Suche nach E-Mail-Adressen von
Benutzern der Rechner der <A HREF="/">Fakult&auml;t
f&uuml;r Mathematik und Wirtschaftswissenschaften</A>. Zu
den Benutzern geh&ouml;ren alle Fakult&auml;tsmitglieder
einschlie&szlig;lich der Studenten und G&auml;ste. Bitte
geben Sie den Vor- und/oder Nachnamen ein.
<P>
ENDE_DER_HINWEISE
```

- CGI-Skripte sollten aus Sicherheitsgründen immer im Tainted-Mode (Option `T`) laufen und mit Warnungen und unter Verwendung von `use strict` operieren.

- Die beiden Indizes leben bei uns auf der Thales unter `$dbdir`.

Das suchende CGI-Skript

email.pl

```
my $p = new CGI;

if ($p->param && $p->param('keys')) {
   # Suche durchfuehren und Ergebnis ausgeben
} else {
   print_header("Suche nach einer E-Mail-Adresse");
   print $hinweise;
}
print_form();
print "<HR>\n";
print qq(<ADDRESS><A HREF="/">Fakult&auml;t f&uuml;r
Mathematik und Wirtschaftswissenschaften der
Universit&auml;t Ulm</A>,
<A HREF="mailto:$admin">$admin</A></ADDRESS>\n);
print $p->end_html;
```

- `new CGI` liefert ein Objekt, über das die CGI-Methoden zur Verfügung stehen.
- `$p->param` liefert eine Liste der Parameternamen aus der URL oder dem ausgefüllten Formular. Wenn noch nichts vorliegt, ist diese Liste leer.
- `$p->param('keys')` liefert den Wert des Parameter `keys`. Dies ist der einzige Parameter in diesem Beispiel.
- In jedem Falle wird ein Formular mit `print_form` erzeugt.

Das suchende CGI-Skript

email.pl

```
sub print_header {
   my $title = shift;
   print $p->header;
   print $p->start_html(
      -Title => $title,
      -Author => $admin,
   );
   print $p->h1($title);
}

sub print_form {
   print $p->start_form;
   print $p->textfield(-name => 'keys');
   print $p->submit(-name => 'Suchen');
   print $p->end_form;
}
```

- Das CGI-Modul liefert eine Vielzahl von Operationen zur Generierung des HTTP-Kopfes (Methode `header`) und von HTML.
- Formulare werden mit `start_form` und `end_form` eingegrenzt, innerhalb dessen können eine Reihe von Eingabefelder generiert werden (hier nur ein `textfield`) und ein Knopf zum Abschicken sollte auch dabei sein (`submit`).
- Mit `-name => 'keys'` wird der Parametername eines Eingabefeldes bestimmt. Per Voreinstellung ist dieses Feld leer oder, falls bereits Parameterwerte vorliegen, wird es mit dem alten Wert gefüllt.
- Auf diese Weise können Benutzer solcher CGI-Skripte Tippfehler korrigieren, ohne den Weg zurück suchen zu müssen.

Das suchende CGI-Skript

email.pl

```
my %fullname; my %keyword;
tie(%fullname, 'DB_File', "$dbdir/byFullName.db", O_RDONLY);
tie(%keyword, 'DB_File', "$dbdir/byKey.db", O_RDONLY);

my $key = $p->param('keys');
$key =~ s/,//g;
$key =~ s/\s+/ /g;
$key =~ s/^\s+//;
$key =~ s/\s+$//;
$key = umlaute2asc($key);
my @names = ();
if (defined $keyword{$key}) {
   @names = split /:/, $keyword{$key};
} elsif (defined $fullname{$key}) {
   @names = ($key);
} else {
   # Suche nach Namen, die eine Obermenge der
   # uebergebenen Namenskomponenten darstellen
}
```

- Wenn der Parameter `keys` vorliegt, werden beide DBM-Dateien mit `tie` eröffnet.
- Nachdem überflüssige Leerzeichen und Kommata entfernt und Umlaute in Ersatznotation konvertiert worden sind, wird untersucht, ob der ganze Suchbegriff als
  - Namensteil oder
  - vollständiger Name oder
  - Namens-Teilmenge eines vollständigen Namens

bekannt ist.

Das suchende CGI-Skript

email.pl

```
my @keys = ();
foreach my $key (split /\s/, $key) {
   if (defined $keyword{$key}) {
      push(@keys, $key);
   } else {
      @keys = (); last;
   }
}
if (@keys == 0) {
   @names = ();
} else {
   my %names = ();
   foreach my $key (@keys) {
      foreach my $name (split /:/, $keyword{$key}) {
         if (defined $names{$name}) {
            $names{$name} ++;
         } else {
            $names{$name} = 1;
         }
      }
   }
   @names = ();
   foreach my $name (keys %names) {
      push(@names, $name) if $names{$name} == @keys;
   }
}
```

- Dieser Programmtext filtert alle vollständigen Namen heraus, die eine Obermenge der angegebenen Namensteile bilden.

Das suchende CGI-Skript

email.pl

```
if (@names > 1) {
   print_header("Gefundene E-Mail-Adressen");
   print "Folgende E-Mail-Adressen wurden gefunden:\n<P>\n";
   print "<TABLE BORDER=3>\n";
   foreach my $name (sort byLastName @names) {
      my $address = $fullname{$name} . $domain;
      print "<TR><TD>$name</TD><TD>";
      print qq(<A HREF="mailto:$address">$address</A>);
      print "</TD></TR>\n";
   }
   print "</TABLE>\n<HR>\n";
} elsif (@names > 0) {
   print_header("Gefundene E-Mail-Adresse");
   print "Folgende E-Mail-Adresse wurde gefunden:\n<P>\n";
   my $name = $names[0];
   my $address = $fullname{$name} . $domain;
   print "<TABLE>\n";
   print "<TR><TD>Name:</TD><TD>$name</TD></TR>\n";
   print "<TR><TD>Adresse:</TD>";
   print qq(<TD><A HREF="mailto:$address">$address</A></TD>);
   print "</TR></TABLE><HR>\n";
} else {
   print_header("Leider nichts gefunden");
   print "Bitte probieren Sie es erneut:\n<P>\n";
}
```

- In `@names` hinterließ das suchende Programmteil die Liste der gefundenen vollständigen Namen, die unter Verwendung von `%fullname` in E-Mail-Adressen abgebildet werden können.

CGI im Überblick

`use CGI;`	CGI importieren
`my $cgi = new CGI;`	CGI-Objekt erzeugen
`my @params = $cgi->param();`	liefert die Liste der Parameternamen
`my $value = $cgi->param($name);`	den Wert eines Parameters abfragen
`my $url = $cgi->url();`	eigene URL ermitteln
`print $cgi->header;`	HTTP-Kopf ausgeben
`print $cgi->start_html($title);`	HTML-Kopf mitsamt Titel ausgeben
`print $cgi->h1($title);`	Überschrift ausgeben
`print $cgi->hr;`	Querstrich ausgeben
`print $cgi->end_html;`	HTML abschließen

CGI im Überblick

`print $cgi->start_form;`	Formular beginnen
`print $cgi->textfield(-name => $name);`	Textfeld ausgeben
`print $cgi->password_field(-name => $name);`	Passwortfeld
`print $cgi->popup_menu(-name => $name, -values => \@values);`	Popup-Menu
`print $cgi->scrolling_list(-name => $name, -values => \@values, -multiple => 'true');`	Liste mit Scroll-Bar
`print $cgi->checkbox_group(-name => $name, -values => \@values, -rows => 2, -columns => 2);`	Tabelle mit ankreuzbaren Feldern
`print $cgi->checkbox(-name => $name)`	Einzelnes ankreuzbares Feld
`print $cgi->radio_group(-name => $name, -values => \@values, -rows => 2, -columns => 2);`	Genau eines ist immer angekreuzt
`print $cgi->hidden(-name => $name, -default => $value);`	Nicht darzustellendes Feld
`print $cgi->submit(-name => $name);`	Knopf zum Abschicken generieren
`print $cgi->end_form;`	Formular beenden

Ein virtuelles Kaufhaus

- Als ein etwas umfangreicheres Beispiel fuer web-basierte Anwendungen dient im folgenden ein kleines virtuelles Kaufhaus.

- Das Beispiel demonstriert,

  - wie bei HTTP der Zustand einer Sitzung verwaltet werden kann (ohne Verwendung von Cookies) und

  - wie größere Web-Anwendungen so strukturiert werden können, so daß sie leicht erweiterbar sind.

- Alle zugehörigen Quellen und weiteren Dateien liegen öffentlich unter `/home/theseus/vshop`.

- Tests sind möglich über die URL
  `http://theseus.mathematik.uni-ulm.de:12345/cgi-bin/vshop`

Anforderungskatalog

- Es gibt eine Reihe von Produkten in diversen, hierarchisch geordneten Kategorien, die entsprechend dargeboten werden sollen.

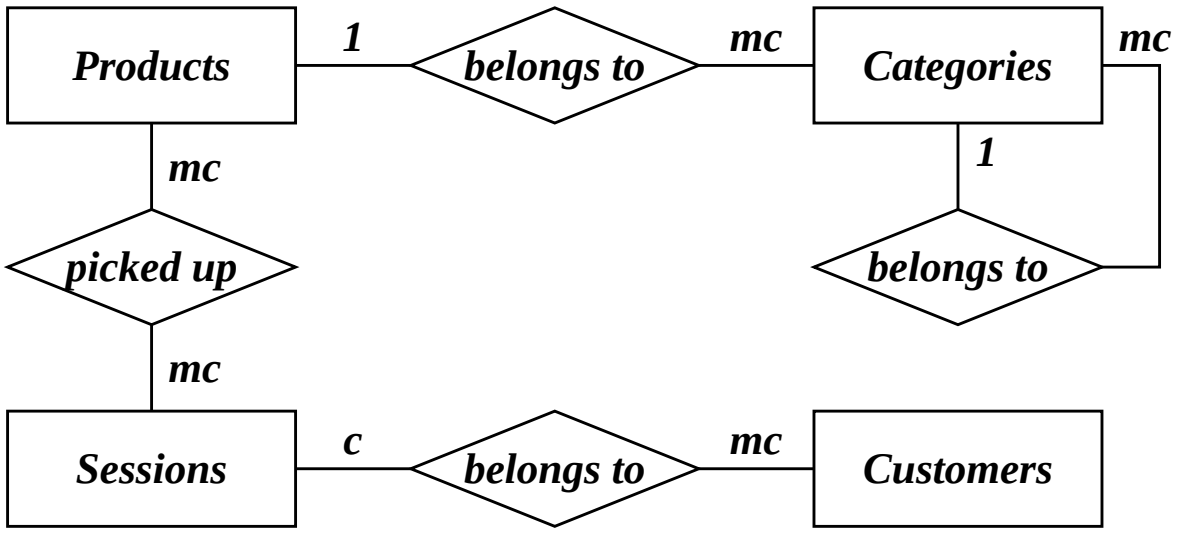
- Angezeigte Produkte sollen jederzeit in einen virtuellen Einkaufswagen übernommen werden können.

- Der Inhalt des Einkaufswagens soll frei änderbar sein.

- Kunden können sich mit Benutzername und Passwort registrieren bzw. später wieder anmelden. Zur Kundeninformation gehört die Lieferadresse und eine elektronische Kontaktadresse.

- Die Anmeldung bzw. Registrierung kann bis zum Zeitpunkt der Bestellung verschoben werden – die Verwendung eines Einkaufswagens soll schon vorher möglich sein.

ER-Diagramm



- Es gibt genau eine Wurzelkategorie, die auf sich selbst verweist.

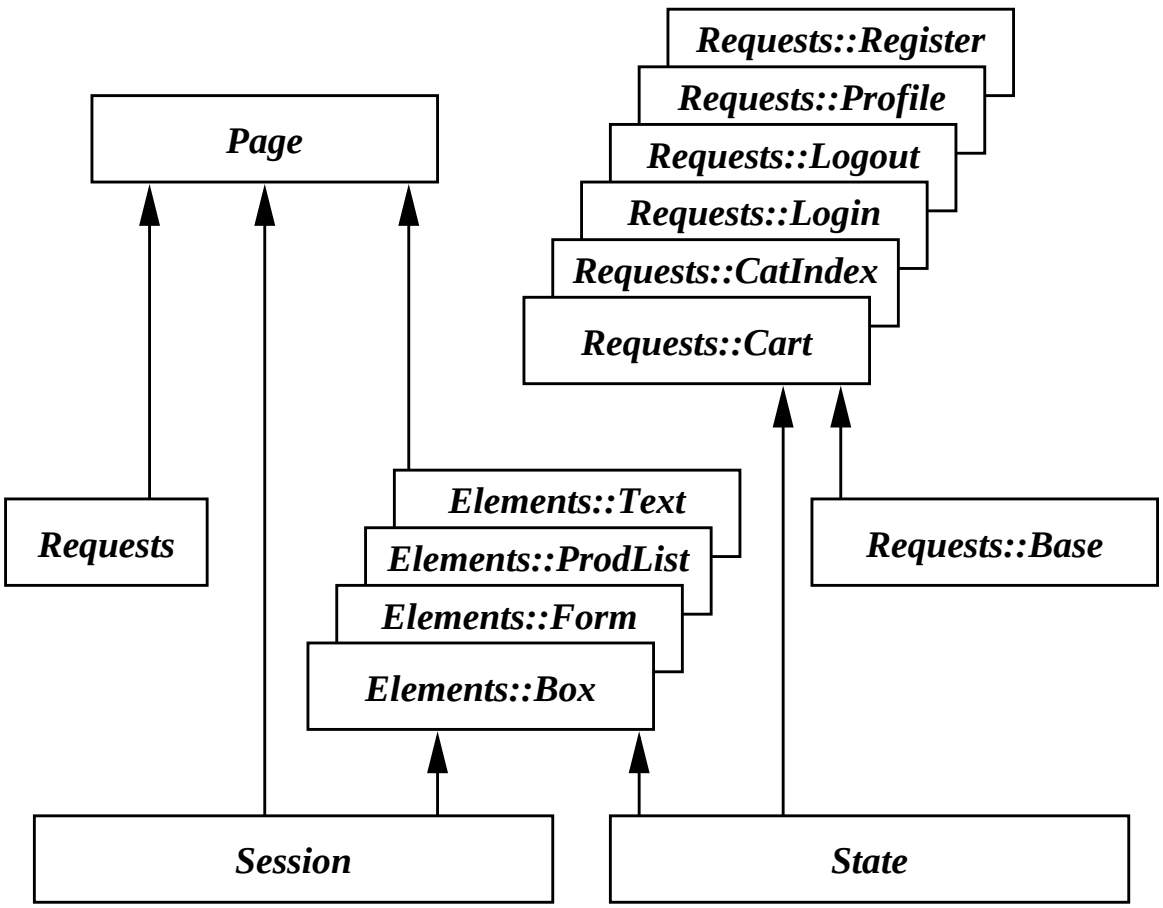
- Der Inhalt von Einkaufswagen ergibt sich durch die "picked up"-Beziehung, wobei ein zusätzliches Attribut für die Quantität benötigt wird.

- Eine Sitzung ist erst dann mit einem Kunden verbunden, wenn eine Registrierung oder eine erfolgreiche Anmeldung vorliegt.

SQL-Tabellen

```
CREATE TABLE Products (
   prodid VARCHAR(32) NOT NULL PRIMARY KEY, INDEX (prodid),
   catid VARCHAR(32) NOT NULL REFERENCES Categories,
   description VARCHAR(255) NOT NULL
);

CREATE TABLE Categories (
   catid VARCHAR(32) NOT NULL PRIMARY KEY, INDEX (catid),
   title VARCHAR(64) NOT NULL,
   parent VARCHAR(32) NOT NULL REFERENCES Categories
);

CREATE TABLE Customers (
   login VARCHAR(32) NOT NULL PRIMARY KEY, INDEX (login),
   name VARCHAR(255) NOT NULL,
   password VARCHAR(32) NOT NULL,
   email VARCHAR(255),
   address VARCHAR(255) NOT NULL
);

CREATE TABLE Sessions (
   sessionid VARCHAR(32) NOT NULL PRIMARY KEY,
      INDEX (sessionid),
   login VARCHAR(32) REFERENCES Customer,
   start DATETIME NOT NULL,
   lastuse DATETIME NOT NULL
);

CREATE TABLE Carts (
   sessionid VARCHAR(32) NOT NULL REFERENCES Session,
   prodid VARCHAR(32) NOT NULL REFERENCES Products,
   quantity INTEGER,
   PRIMARY KEY (sessionid, prodid)
);
```

Modulhierarchie



Modulhierarchie

- *Session* verwaltet den Sitzungszustand, der mit der Datenbank in Verbindung steht.
- *State* dient der Rettung von Zustandsvariablen, die nicht in der Datenbank verwaltet werden.
- Die *Elements*-Module dienen der flexiblen Generierung von HTML-Seiten – ein wenig analog zu graphischen Toolkits.
- *Requests* "entdeckt" die übrigen *Requests*-Module und stellt sie zur Verfügung.
- Die *Requests*-Module realisieren jeweils einzelne Funktionalitäten oder Masken der Web-Anwendung. Sie werden nicht explizit importiert, sondern allein durch ihre Existenz aktiv.
- *Page* koordiniert den Zusammenbau einer Webseite.

Sitzungs-Nummer

- Sofort, wenn die erste Seite abgerufen wird, kommt es zu der Vergabe einer Sitzungsnummer.
- Die Sitzungsnummer wird bei allen Formularen und Verweisen weitervererbt.
- Sobald eine Registrierung oder Anmeldung erfolgt, wird die Sitzungsnummer mit einem Benutzernamen verknüpft.
- Die Verknüpfung bleibt dann bestehen. Somit sind dann weitere Authorisierungen nicht notwendig, da die Sitzungsnummer (möglicherweise befristet) als hinreichendes Authorisierungsinstrument betrachtet wird.
- Sitzungsnummern sollten daher nicht erratbar sein.
- Bei einer expliziten Abmeldung werden in der Datenbank alle sitzungsbezogenen Daten gelöscht und die Sitzungsnummer ungültig.
- Das Modul *Session* kümmert sich um die Sitzungsnummer und die damit verbundenen Informationen in der Datenbank.

Weitere Zustandsvariablen

- Gelegentlich ist es sinnvoll, ein Teil des Zustands nicht in der Datenbank zu verwalten, sondern nur über Parameter zu halten, die durch Formulare oder Verweise vererbt werden.

- Anwendung findet dies zum Beispiel in der Navigation, um z.B. zu einem Ort im Produktkatalog wieder leichter zurückzufinden.

- Das Modul *State* koordiniert die Vererbung solcher Parameter und vermeidet damit Abhängigkeiten der *Requests*-Module untereinander.

Methoden der *Requests*-Module

`init`	wird von *Requests* nach der "Entdeckung" aufgerufen, um das Modul zu initialisieren und dient als Konstruktor
`reg`	wird von *Page* aufgerufen, um Gelegenheit zur Registrierung von Zustandsvariablen zu geben
`process`	wird aufgerufen, wenn ein Formular ausgefüllt worden ist, das von dem gleichen Modul generiert worden ist
`default`	nur eine Methode aller *Requests*-Module sollte 1 zurückliefern – sie erhält damit die Zuständigkeit der Hauptseite
`takeover`	sollte 1 zurückliefern, wenn dieses Modul den Hauptteil der Seite gestalten möchte
`main`	wird von *Page* aufgerufen, um das Gestaltungselement für den Hauptteil der zurückzuliefernden Seite zu generieren
`nav`	kann ein Gestaltungselement für die Navigationsleiste zurückliefern

Vorgehensweise von *Page*

- Zunächst wird mit Hilfe von *Session* der Sitzungskontext hergestellt.
- Bei allen *Requests*-Modulen wird die Methode `reg` aufgerufen.
- Wenn ein Formular ausgefüllt worden ist, wird die Methode `process` bei dem *Request*-Modul aufgerufen, das es generiert hat.
- Der Hauptteil der Seite wird unter Verwendung der Methode `main` generiert – entweder von dem Modul, daß das ausgefüllte Formular bearbeitet hat (falls es bei `takeover` **TRUE** liefert) oder von dem Modul, das sich bei `default` mit **TRUE** meldet.
- Abschließend erhalten alle *Requests*-Module die Gelegenheit, sich beim Bau der Navigationsleiste zu beteiligen.

Gestaltungselemente

- Die *Elements*-Module bieten alle jeweils

  - einen Konstruktor an, um ein entsprechendes Element zu kreieren,

  - diverse Operationen, um das Element zu erweitern und

  - eine Methode namens `gen`, die HTML-Text auf die mitgegebene Dateiverbindung ausgibt.

- Durch die Trennung von Aufbau und Generierung können Webseiten in beliebiger Reihenfolge Schritt für Schritt aufgebaut werden, ohne die physische Textreihenfolge zu berücksichtigen.

- Zur Verfügung stehen:

`Elements::Box`	vertikaler oder horizontaler Zusammenbau beliebiger Elemente, die bei Bedarf allesamt gerahmt werden können
`Elements::Form`	Formulare und Knöpfe
`Elements::ProdList`	Produktlisten mit zugehörigen Knöpfen
`Elements::Text`	einfacher HTML-Text

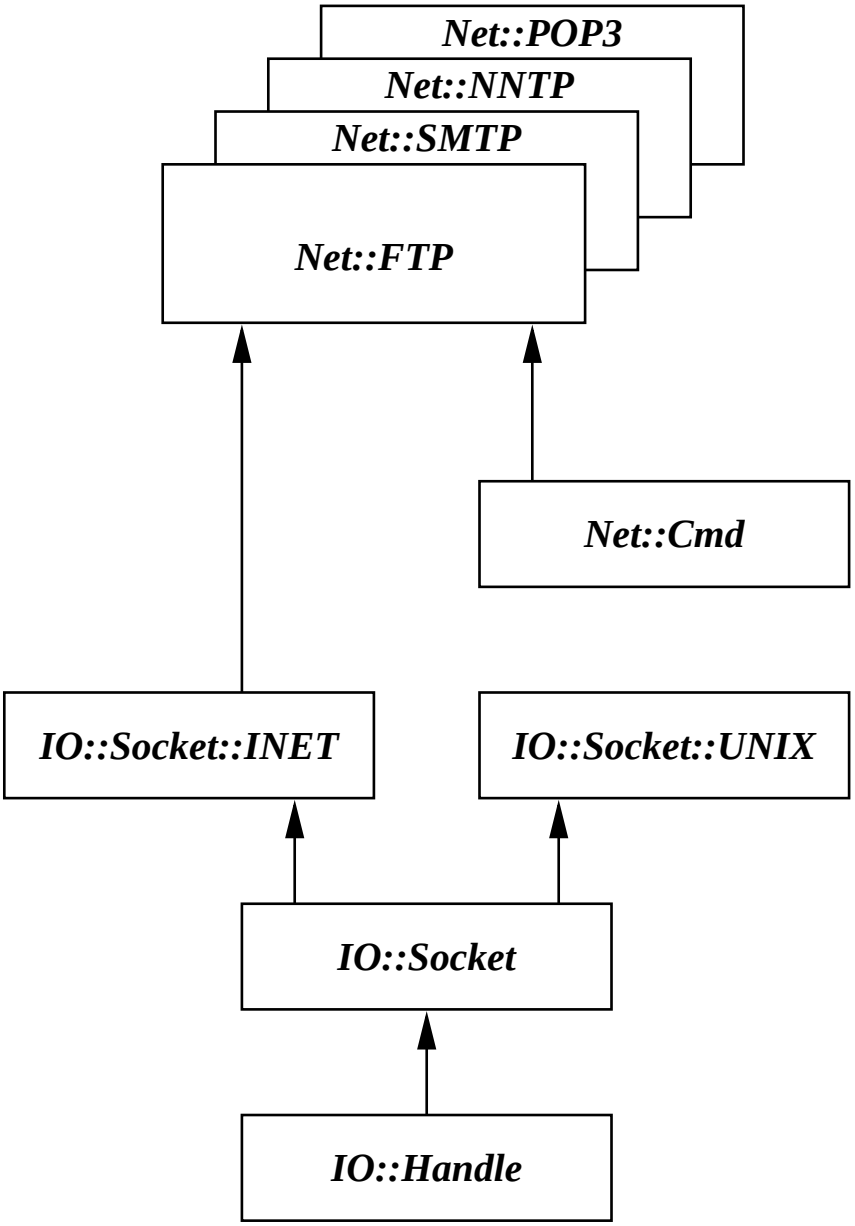
Schnittstellen für Netzdienste

- Für eine Reihe etablierter Netzdienste auf dem Internet gibt es fertige Schnittstellen in Perl.

- Die Module in der Net-Hierarchie offerieren individuelle objektorientierte Schnittstellen für eine Vielzahl von Diensten.

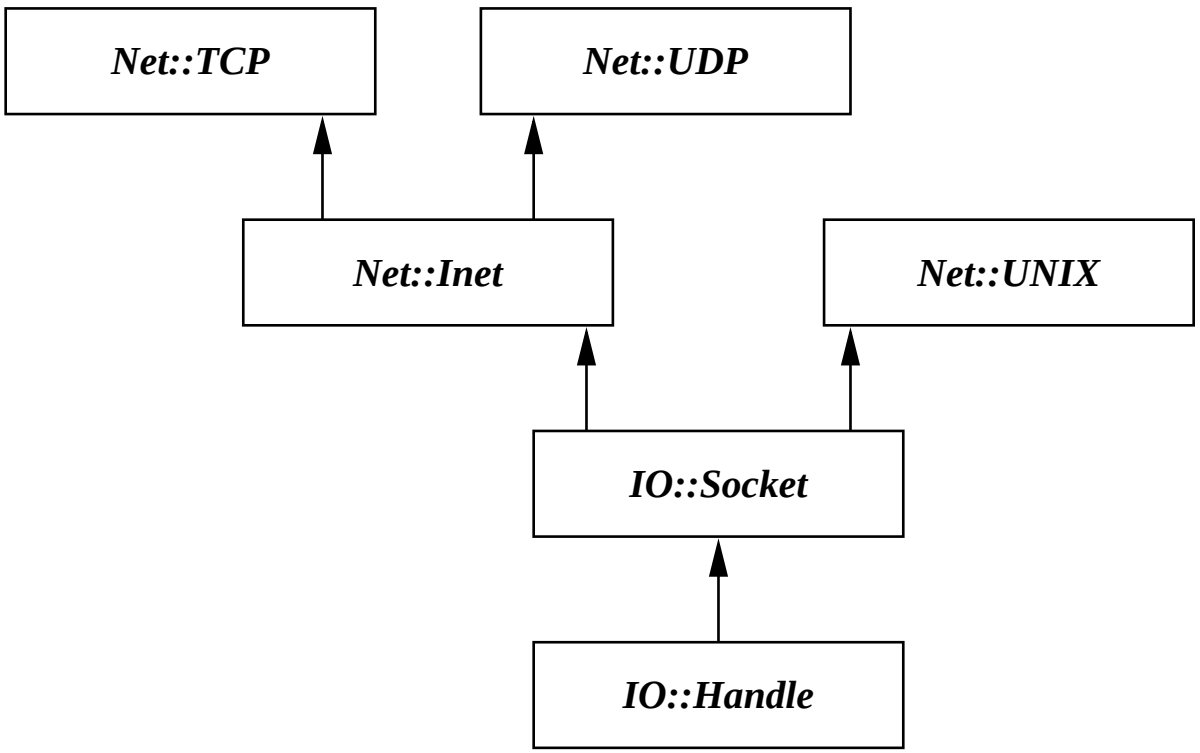
- `LWP` bietet in Verbindung einer Reihe weiterer Module eine abstrakte Schnittstelle, die sich an das Web-Modell bzw. das HTTP-Protokoll anlehnt: Zuerst wird ein Anfrage-Objekt erzeugt, dies einem Agent übermittelt, der dann ein Antwort-Objekt zurückliefert.

Die Net-Hierarchie von Graham Barr



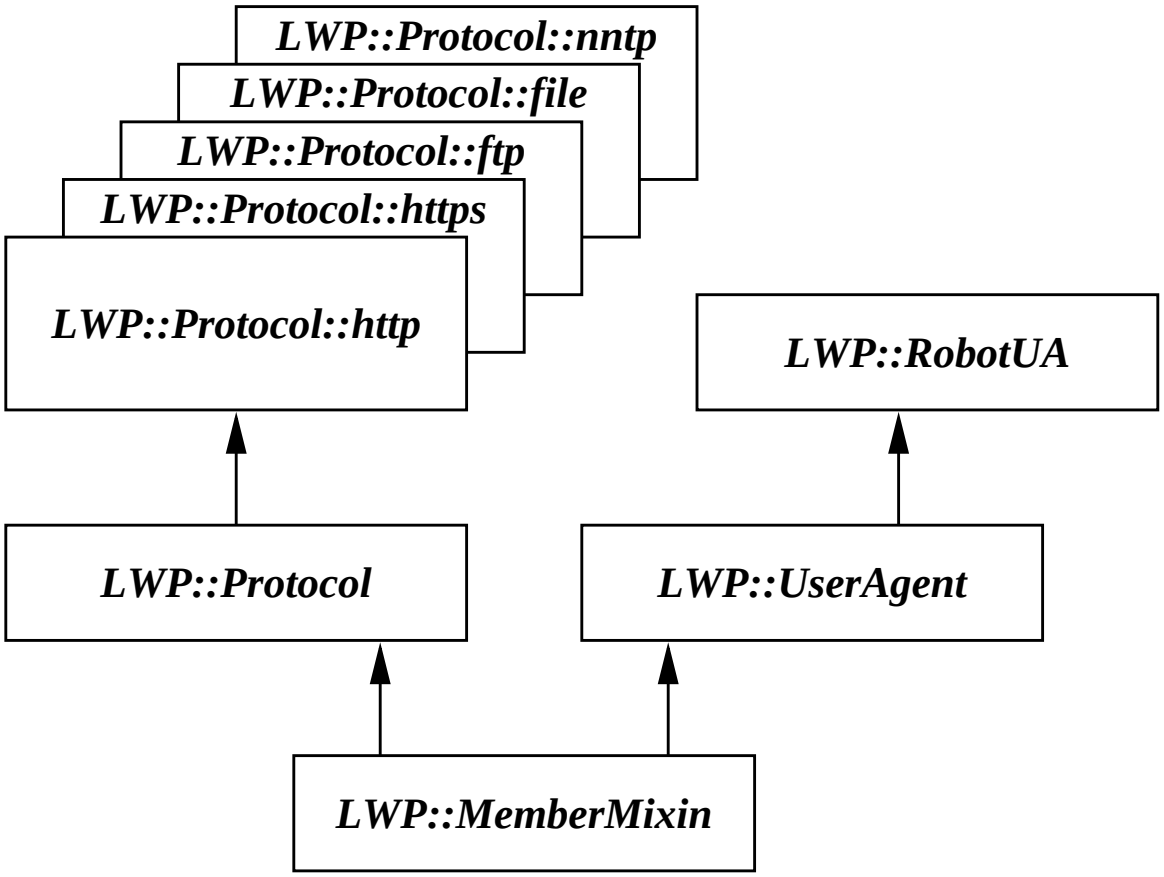
- Graham Barr hat auf Basis von `IO::Socket` und einer allgemeinen Kommandoschnittstelle `Net::Cmd` zahlreiche Internet-Dienste mit Modulen direkt unterstützt.

Die Net-Hierarchie von Spider Boardman



- Spider Boardman hat die Socket-Schnittstelle von Graham Barr erweitert und – um Konflikte zu vermeiden – die `IO::Socket`-Hierarchie neu implementiert.

Die LWP-Hierarchie von Gisle Aas



- Gisle Aas nahm die libwww-Bibliothek zum Vorbild und entwickelte eine Perl-Variante davon (LWP = libwww/Perl).

Auswertung eines cgi-Skriptes

fetchemail.pl

```
#!/usr/local/bin/perl -w
use HTTP::Request;
use LWP::UserAgent;
use URI::Escape;
use strict;

my $url = 'http://www.mathematik.uni-ulm.de/cgi-bin/email';
my $VERSION = 1.0;
my $cmdname = $0; $cmdname =~ s{.*/}{};
my $usage = "Usage: $cmdname name";
die $usage unless @ARGV > 0;
my $keywords = uri_escape(join(" ", @ARGV));
my $ua = new LWP::UserAgent;
$ua->agent("$cmdname/$VERSION " . $ua->agent);
my $request = new HTTP::Request POST => $url;
$request->content_type('application/x-www-form-urlencoded');
$request->content("keys=$keywords");
my $response = $ua->request($request);
unless ($response->is_success) {
   die "$cmdname: unable to retrieve $url\n";
}
my @lines = split /\n/, $response->content;
my $name;
foreach my $line (@lines) {
   if ($line =~ m{<TD>Name:</TD><TD>(.*)</TD>}) {
      $name = $1;
   } elsif (defined($name) &&
         $line =~ m{<TD><A HREF="mailto:(.*?)">}) {
      printf "%-32s %s\n", $name, $1;
   } elsif ($line =~
         m{<TD>(.*?)</TD><TD><A HREF="mailto:(.*?)">}) {
      printf "%-32s %s\n", $1, $2;
   }
}
```

Literaturhinweise

Allgemeine Einführungen und Hintergründe zur Webschnittstelle:

- Andrew S. Tanenbaum, *Computer Networks*, Third Edition, Prentice Hall, Abschnitt 7.6: *The World Wide Web*

- W. Richard Stevens, *TCP/IP Illustrated, Volume 3*, Kapitel 13: *HTTP: Hypertext Transfer Protocol*

- Cricket Liu, Jerry Peek, Russ Jones, Bryan Buus, Adrian Nye, *Managing Internet Information Services*, O'Reilly (Erstausgabe von 1994 ist mittlerweile veraltet, jedoch historisch interessant)

Zu Perl und Webschnittstellen:

- Lincoln Stein, *CGI Programming*, The Perl Journal, Volume 1, Issue 1, 1996
(Beginn einer Artikelserie über CGI im Perl Journal)

- Lincoln Stein, *Official Guide to Programming with CGI.pm*, Wiley
(Ein Tutorial für diejenigen, die Manualseiten nicht mögen)

- Clinton Wong, *Web Client Programming*, O'Reilly
(Wie können Perl-Skripte auf Informationen im Web zugreifen oder mit interaktiven Seiten agieren)

Perl/Tk

- Tk fand als eine sehr gelungene Bibliothek für eine graphische Benutzeroberfläche auch sehr weitreichendes Interesse jenseits von Tcl bei anderen Programmiersprachen wie Scheme, Python, Guile und insbesondere Perl.

- John Ousterhout hat Tk so speziell für Tcl entwickelt, daß entsprechende Portierungen für andere Skript-Programmiersprachen nicht sehr einfach waren. Malcom Beattie, der den ersten Versuch für Perl startete, übernahm deswegen des Tcl-Interpreter mit.

- Später versuchte Nick Ing-Simmons es mehr mit einem generellen Ansatz: Er bereinigte Tk von allen Abhängigkeiten zu Tcl und schuf eine sprach-unabhängige Schnittstelle. Dies nennt sich **pTk** (*portable* Tk). Die heutige Perl-Schnittstelle zu Tk basiert auf pTk.

- Anders als bei unserer Tcl/Tk-Installation wird bei Perl kein separater Interpreter für Perl/Tk benötigt. Stattdessen wird Tk über das Modul `Tk` dynamisch hinzugeladen.

- Mittlerweile gibt es Perl/Tk auch auf zahlreichen Nicht-UNIX-Plattformen einschließlich MacOS und diversen Windows-Variationen.

Erste Schritte mit Perl/Tk

tkhello.pl

```
#!/usr/local/bin/perl -w

use strict;
use Tk;

my $main = new MainWindow;
my $label = $main->Label(-text => 'Hello, world!');
my $button = $main->Button(
   '-text' => 'Quit',
   '-command' => sub { exit },
);
$label->pack; $button->pack;
MainLoop;
```

- Mit `use Tk` werden alle zu Tk gehörenden Module importiert einschließlich aller Widgets wie beispielsweise `MainWindow`. Ferner wird auch der eigene Namensraum mit einigen Namen wie beispielsweise `MainLoop` geflutet.

- Per Konvention wird `use strict` verwendet, um die Disziplinen zu erzwingen, die das Leben in einer modularisierten Welt erleichtern.

- `$main` ist ein Objekt von dem Typ `MainWindow`, das mit `my` nur lokal sichtbar angelegt wird (dies ist insbesondere auch wegen `use strict` notwendig).

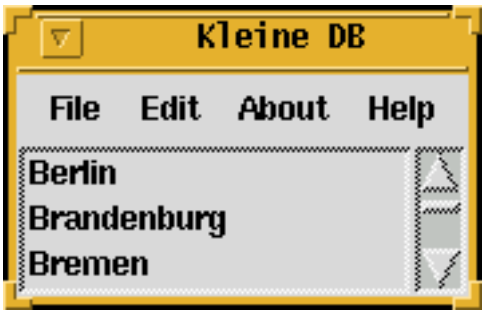
Erste Schritte mit Perl/Tk

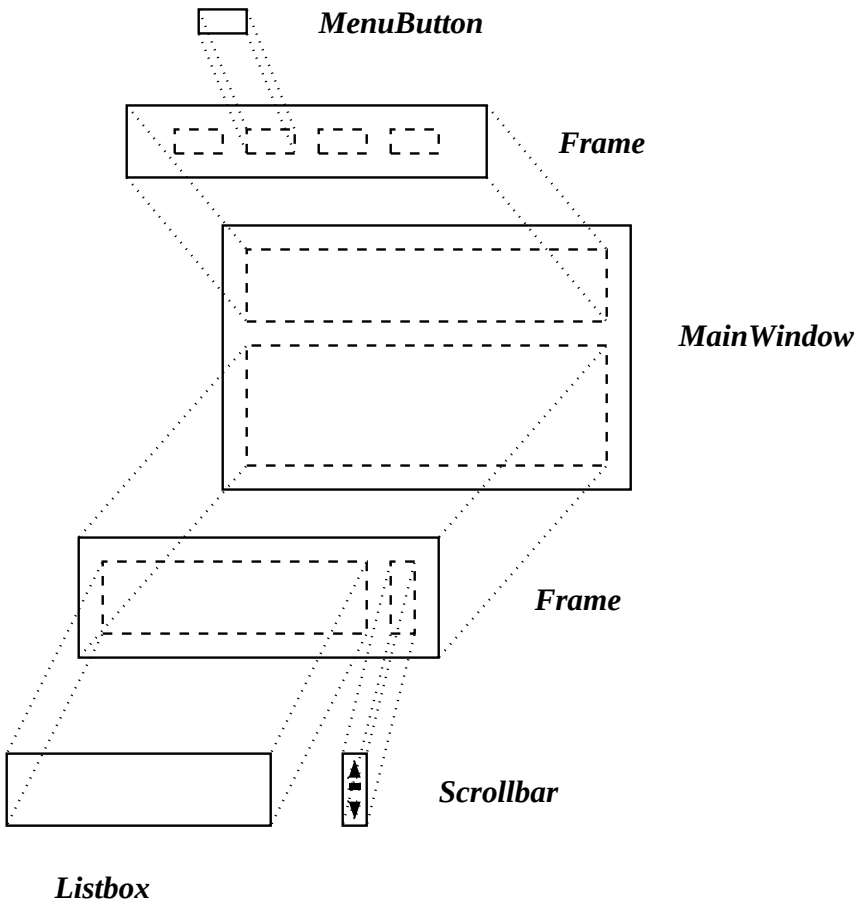
tkhello.pl

```
#!/usr/local/bin/perl -w

use strict;
use Tk;

my $main = new MainWindow;
my $label = $main->Label(-text => 'Hello, world!');
my $button = $main->Button(
    '-text' => 'Quit',
    '-command' => sub { exit },
);
$label->pack; $button->pack;
MainLoop;
```

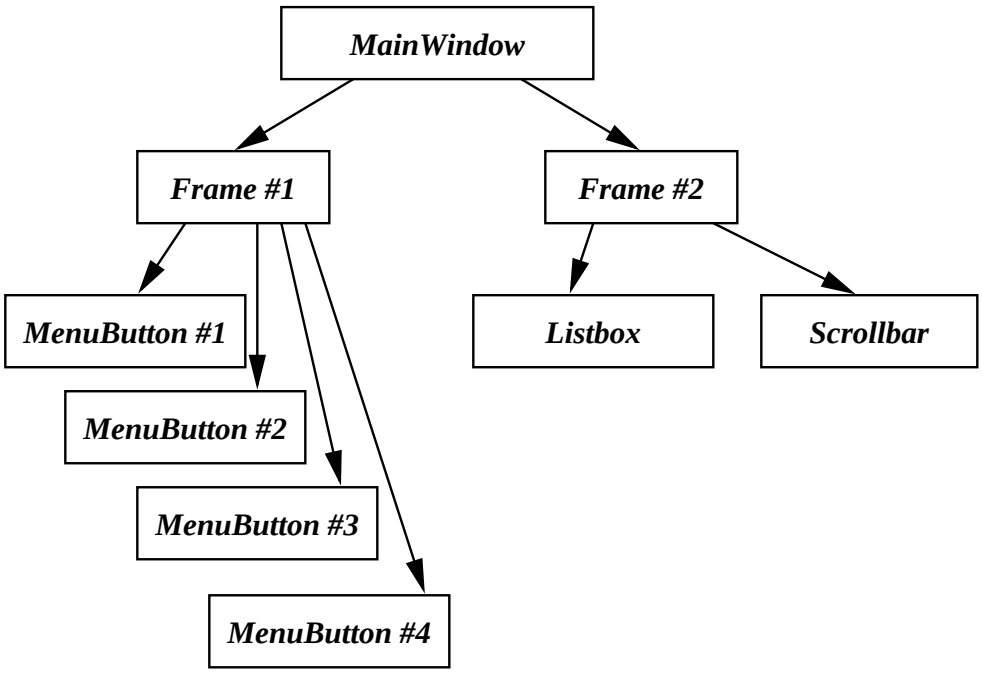
- `Label` und `Button` sind Methoden von `$main`, die entsprechende Widgets kreieren.
- Die Konstruktions-Parameter von Widgets werden als assoziative Arrays übergeben.
- Mit der Methode `pack` wird das jeweilige Objekt an den entsprechenden Geometrie-Manager übergeben.
- `MainLoop` ist die zentrale Schleife, die alle Ereignisse abarbeitet.

Graphischer Aufbau einer Anwendung





Graphischer Aufbau einer Anwendung



- Die graphische Benutzeroberfläche einer Anwendung besteht aus einer Vielzahl von Komponenten (den sogenannten *widgets*), die hierarchisch organisiert und zusammengebaut werden.

- Das Problem liegt in der Findung (und Nutzung) geeigneter Abstraktionen zum Zusammenfügen einer Widget-Hierarchie.

Geometrie-Manager bei Tk

- Tk offeriert drei verschiedene Abstraktionen (die sich Geometrie-Manager nennen) zum Zusammenfügen:

pack	Die Positionierung von Widgets erfolgt jeweils innerhalb des übergeordneten Widgets im noch verbliebenen "Hohlraum" (*cavity model*).
place	Hierbei wird die Position und Größe eines Widgets präzise in Relation zum übergeordneten Widget spezifiziert.
form	Verallgemeinerung, die die Fähigkeiten von **pack** und **place** miteinander vereinigt.
grid	Hier erfolgt die Positionierung innerhalb eines Gitters, das innerhalb des übergeordneten Widgets angelegt worden ist.

- Verschiedene Teile einer Anwendung können unterschiedliche Geometrie-Manager verwenden.

- Die folgenden Beispiele verwenden alle **pack**.

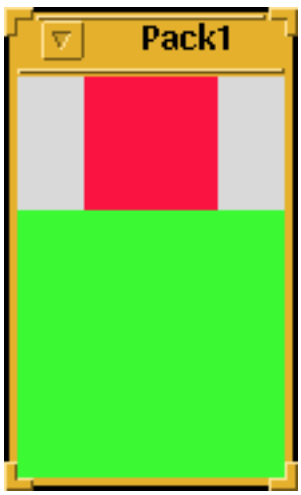
Geometrie-Manager pack

- Das übergeordnete Widget stellt einen rechteckigen Raum zur Verfügung, der möglicherweise bereits teilweise belegt ist.
- Wenn die Methode **pack** aufgerufen wird, erfolgt die Plazierung innerhalb des verbliebenen Hohlraumes.
- Die gewünschte Seite kann dabei angegeben werden: Beispielsweise mit `-side => 'top'` (Positionierung an der Oberseite des Hohlraumes).
- Wenn die Größe des übergeordneten Widgets nicht fest vorgegeben ist, dann ergibt sie sich implizit aus der Größe der eingefügten Widgets.

Geometrie-Manager pack

pack1.pl

```
#!/usr/local/bin/perl -w

use Tk;
use strict;

my $top = new MainWindow;
my $frame1 = $top->Frame(-width => 50, -height => 50,
   -bg => 'red');
my $frame2 = $top->Frame(-width => 100, -height => 100,
   -bg => 'green');
$frame1->pack(-side => 'top');
$frame2->pack(-side => 'top');
MainLoop;
```



Geometrie-Manager pack

pack2.pl

```
#!/usr/local/bin/perl -w

use Tk;
use strict;

my $top = new MainWindow;
my $frame1 = $top->Frame(-width => 50, -height => 50,
   -bg => 'red');
my $frame2 = $top->Frame(-width => 100, -height => 100,
   -bg => 'green');
$frame1->pack(-side => 'left');
$frame2->pack(-side => 'left');
MainLoop;
```

Geometrie-Manager pack

pack3.pl

```
#!/usr/local/bin/perl -w

use Tk;
use strict;

my $top = new MainWindow;
my $frame1 = $top->Frame(-width => 50, -height => 50,
   -bg => 'red');
my $frame2 = $top->Frame(-width => 100, -height => 100,
   -bg => 'green');
my $frame3 = $top->Frame(-width => 75, -height => 75,
   -bg => 'blue');
$frame1->pack(-side => 'left'); # links ist zu
$frame2->pack(-side => 'right'); # rechts ist zu
$frame3->pack(-side => 'left'); # kommt in die Mitte
MainLoop;
```

Geometrie-Manager pack

pack4.pl

```
#!/usr/local/bin/perl -w

use Tk;
use strict;

my @colors = qw(black white);
my $top = new MainWindow;
foreach my $i (1..8) {
   my $row = $top->Frame->pack(-side => 'top');
   foreach my $j (1..8) {
      my $box = $row->Frame(-width => 10, -height => 10,
         -bg => $colors[($i + $j) % 2]);
      $box->pack(-side => 'left');
   }
}
MainLoop;
```

- Es empfiehlt sich, innerhalb eines übergeordneten Widgets (typischerweise Frames) nur eine Packungsrichtung zu verwenden – sonst wird es unübersichtlich.

- Um kompliziertere Aufbauten zu erreichen, werden Frames ineinander verschachtelt.

Geometrie-Manager pack

- Die Option `-fill` bei **pack** gibt einem Widget die Möglichkeit, ungenutzten Raum (jedoch nicht den verbleibenden Hohlraum) zu nutzen. Das Widget wird also u.U. größer als zunächst vorgesehen:

`-fill => 'none'`	Größe bleibt konstant unabhängig von benachbarten Freiräumen (Voreinstellung).
`-fill => 'x'`	Eine Expansion findet nur in x-Richtung statt (primär sinnvoll bei `-side => 'top'` oder `-side => 'bottom'`).
`-fill => 'y'`	Expansion in y-Richtung möglich (primär sinnvoll bei `-side => 'left'` oder `-side => 'right'`).
`-fill => 'both'`	Expansion in beide Richtungen möglich (wovon zunächst nur eine genutzt wird).

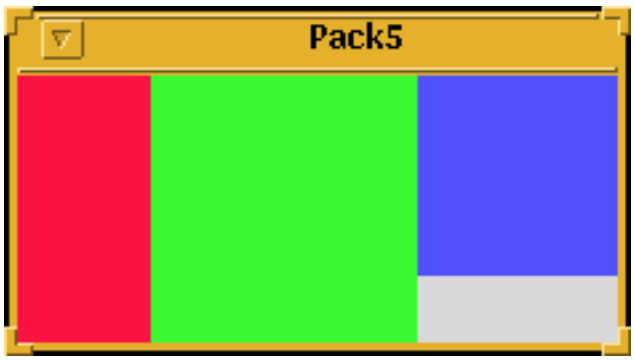
- Wird das Hauptfenster in der Größe verändert, kann dies zur dynamischen Vergrößerung von Widgets führen.

- Damit das klappt, müssen ggf. auch alle in der Hierarchie dazwischenliegenden Widgets mit der entsprechenden **fill**-Option gepackt worden sein.

Geometrie-Manager pack

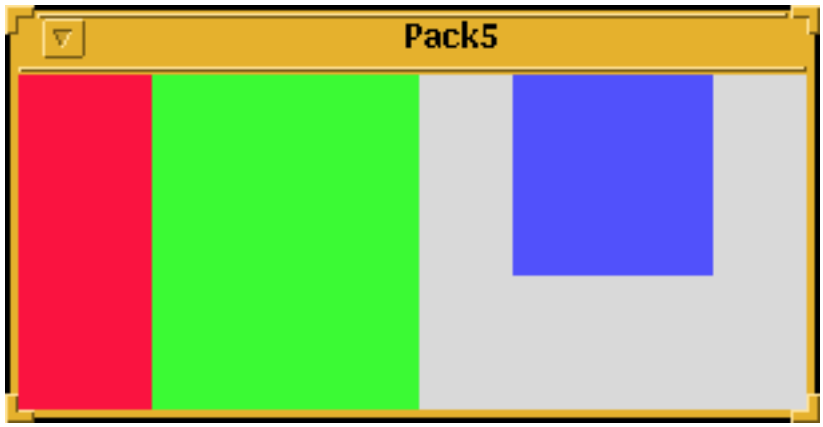
pack5.pl

```
#!/usr/local/bin/perl -w

use Tk;
use strict;

my $top = new MainWindow;
my $frame1 = $top->Frame(-width => 50, -height => 50,
   -bg => 'red');
my $frame2 = $top->Frame(-width => 100, -height => 100,
   -bg => 'green');
my $frame3 = $top->Frame(-width => 75, -height => 75,
   -bg => 'blue');
$frame1->pack(-side => 'left', -fill => 'y');
$frame2->pack(-side => 'left', -fill => 'y');
$frame3->pack(-side => 'top', -fill => 'y');
MainLoop;
```



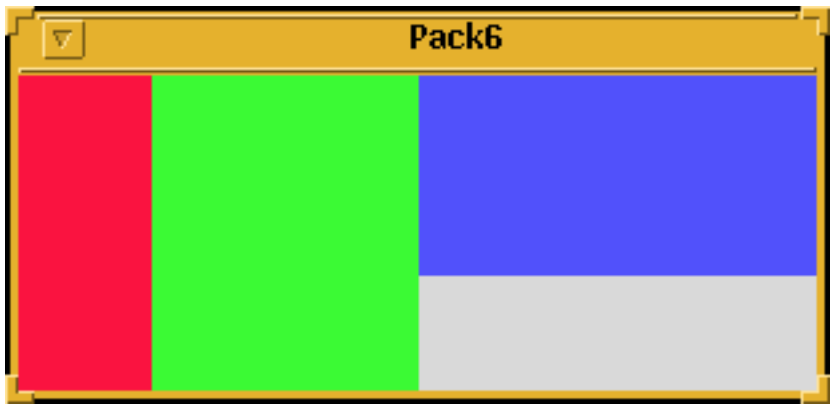
- `$frame3` expandiert nicht in den Hohlraum (wegen `-side => 'top'`).

Geometrie-Manager pack



- Hier wurde das Fenster etwas vergrößert, wodurch sich auch `$frame1` und `$frame2` mit vergrößerten.

- Wenn `$frame3` mit der Option `-fill => 'x'` kreiert worden wäre, würde ein vergrößertes Fenster so aussehen:



Geometrie-Manager pack

pack7.pl

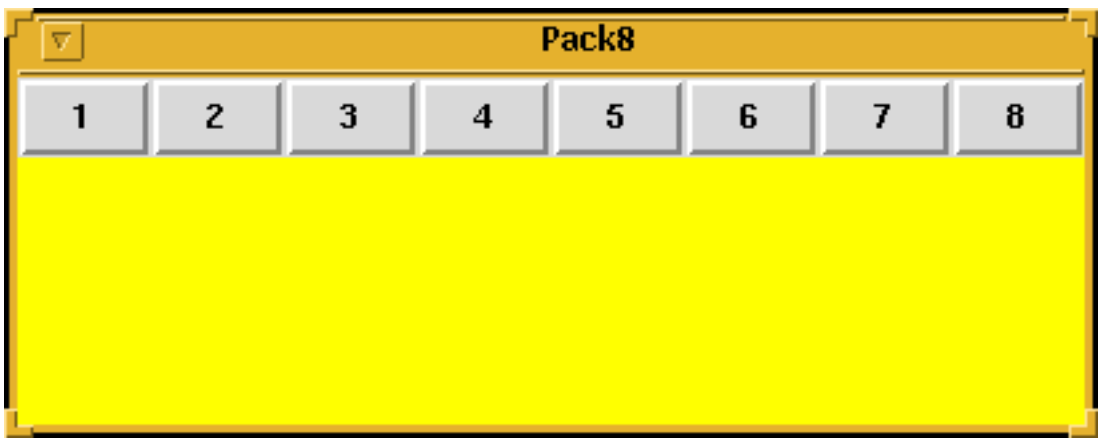
```
#!/usr/local/bin/perl -w

use Tk;
use strict;

my $top = new MainWindow;
my $frame1 = $top->Frame(-width => 50, -height => 50,
   -bg => 'red');
my $frame2 = $top->Frame(-width => 100, -height => 100,
   -bg => 'green');
my $frame3 = $top->Frame(-width => 75, -height => 75,
   -bg => 'blue');
$frame1->pack(-side => 'left', -fill => 'y');
$frame2->pack(-side => 'left', -fill => 'y');
$frame3->pack(-side => 'top', -fill => 'y', -expand => 1);
MainLoop;
```

- Die jeweilige **fill**-Option läßt sich mit `-expand => 1` noch verstärken, indem auch verbleibende Hohlräume mit verkonsumiert werden können.

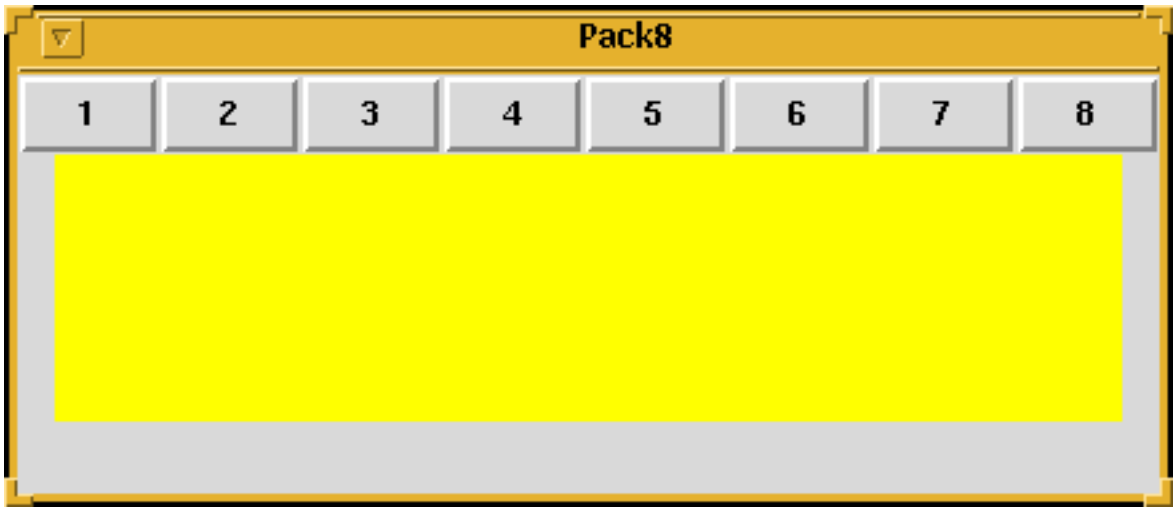
Geometrie-Manager pack

pack8.pl

```
#!/usr/local/bin/perl -w

use Tk;
use strict;

my $top = new MainWindow;
my $buttons = $top->Frame;
foreach my $i (1..8) {
   my $button = $buttons->Button(-text => $i);
   $button->pack(-side => 'left',
      -fill => 'x', -expand => 1);
}
my $frame = $top->Frame(-width => 400, -height => 100,
   -bg => 'yellow');
$buttons->pack(-side => 'top', -fill => 'x');
$frame->pack(-side => 'top');
MainLoop;
```



Geometrie-Manager pack



- So sieht das Fenster nach einer Vergrößerung aus.
- Es lohnt sich also, konsequent über die Ausnutzung freien Raumes nachzudenken.
- Wenn jedoch mehrere Widgets per `-expand => 1` den gleichen Hohlraum vereinnahmen möchten, dann findet eine gleichmäßige Aufteilung statt.
- Somit ist es normalerweise sinnvoll, eine Button-Leiste in x-Richtung expandieren zu lassen – die y-Richtung sollte jedoch den Widgets zur Expandierung überlassen werden, die sie auch mit Inhalt füllen können (insbesondere solche, die mit Scrollbars ausgestattet sind).

Ereignisse

bind.pl

```
#!/usr/local/bin/perl -w

use Tk;
use strict;

my $top = new MainWindow;
my $frame = $top->Frame(-width => 50, -height => 50,
   -bg => 'yellow');
$frame->pack;
$frame->bind('<ButtonPress>',
   sub { print "Ouch, that hurts!\n" });
MainLoop;
```

- Alle Aktionen des Benutzers (Bewegen der Maus, Benutzung der Maustasten oder der Tastatur) führen zu einem Strom einzelner Ereignisse.

- Aus der Mausposition oder der Fokus-Zugehörigkeit zum Zeitpunkt der Entstehung eines Ereignisses und vorheriger Interessensbekundungen (z.B. durch die Methode **bind**) ergibt sich eine Menge von Parteien, denen das Ereignis mitzuteilen ist.

- Eine "Mitteilung" erfolgt in Perl/Tk durch den Aufruf einer (typischerweise anonymen) Prozedur und (bei Bedarf) einer Reihe von Parametern.

- Interessant sind die Spezifikationen von Ereignissen, die Möglichkeiten der Interessensbekundungen und die Reihenfolge der Mitteilungen.

Spezifikation von Ereignissen

- Ereignis-Spezifikationen sehen folgendes allgemeines Schema vor:
  `<modifier-modifier-type-detail>`

- Zu den Modifiern zählen `Control`, `Shift`, `Button1` (1. Button der Maus), `Alt`, `Meta` und auch `Double` (für Doppelklicks). Auf die Angabe von Modifiern kann ganz verzichtet werden (dann spielen sie keine Rolle) oder es können ein oder zwei angegeben werden. In letzterem Falle müssen dann beide Modifier gleichzeitig aktiv sein.

- Zu den Typen gehören `ButtonPress`, `ButtonRelease`, `KeyPress`, `KeyRelease`, `Enter` und `Release` (neben vielen weiteren).

- Beim Detail kann (z.B. bei `KeyPress`) der Name einer weiteren Taste angegeben werden (oder schlicht der Buchstabe) oder (z.B. bei `ButtonPress`) die Nummer der Maustaste.

- Die Angaben können weitgehend gekürzt werden, solange noch eindeutig ist, was gewünscht wird. Statt
  `<Control-KeyPress-U>` ist auch `<Control-U>` zulässig.

- Ereignis-Spezifikationen operieren wie Muster, die eingehende Ereignisse filtern. So trifft beispielsweise `<KeyPress>` für alle gedrückten Tasten zu.

- Die Namen der Tasten hängen natürlich von dem lokalen Window-System ab und sind daher nicht in jedem Falle uneingeschraenkt portabel (wie auch Tastaturen sehr unterschiedlich aussehen können).

Interessensbekundungen

- Eine Interessensbekundung für Ereignisse besteht aus einem Kontext und einer Ereignisspezifikation.
- Mögliche Kontexte sind
  - ein einzelnes Widget,
  - alle Widgets eines bestimmten Typs (z.B. alle Buttons),
  - alle Widgets, die zu einem Toplevel-Fenster gehören und
  - die Gesamtheit aller Widgets.
- Der Kontext wird als erster Parameter bei **bind** angegeben – wenn er fehlt, wird nur ein Bezug zu dem vorgegebenen Widget-Objekt hergestellt. Es kann ein Klassenname (z.B. `'Tk::Button'`) angegeben werden, das Widget eines Toplevel-Fensters oder schlicht `'all'`.

Interessensbekundungen

- Wenn für ein Ereignis mehrere Interessensbekundigungen vorliegen, dann werden sie in einer definierten Reihenfolge abgearbeitet. Per Voreinstellung gilt folgende Reihenfolge:

  - typgebundene Bearbeiter,

  - für das einzelne Widget registrierte Bearbeiter,

  - Bearbeiter des Toplevel-Fensters und schließlich

  - generelle Bearbeiter.

- Die Reihenfolge kann (mit **bindtags**) für jedes Widget verändert (und sogar noch erweitert) werden.

Ereignis-Bearbeiter

keypress.pl

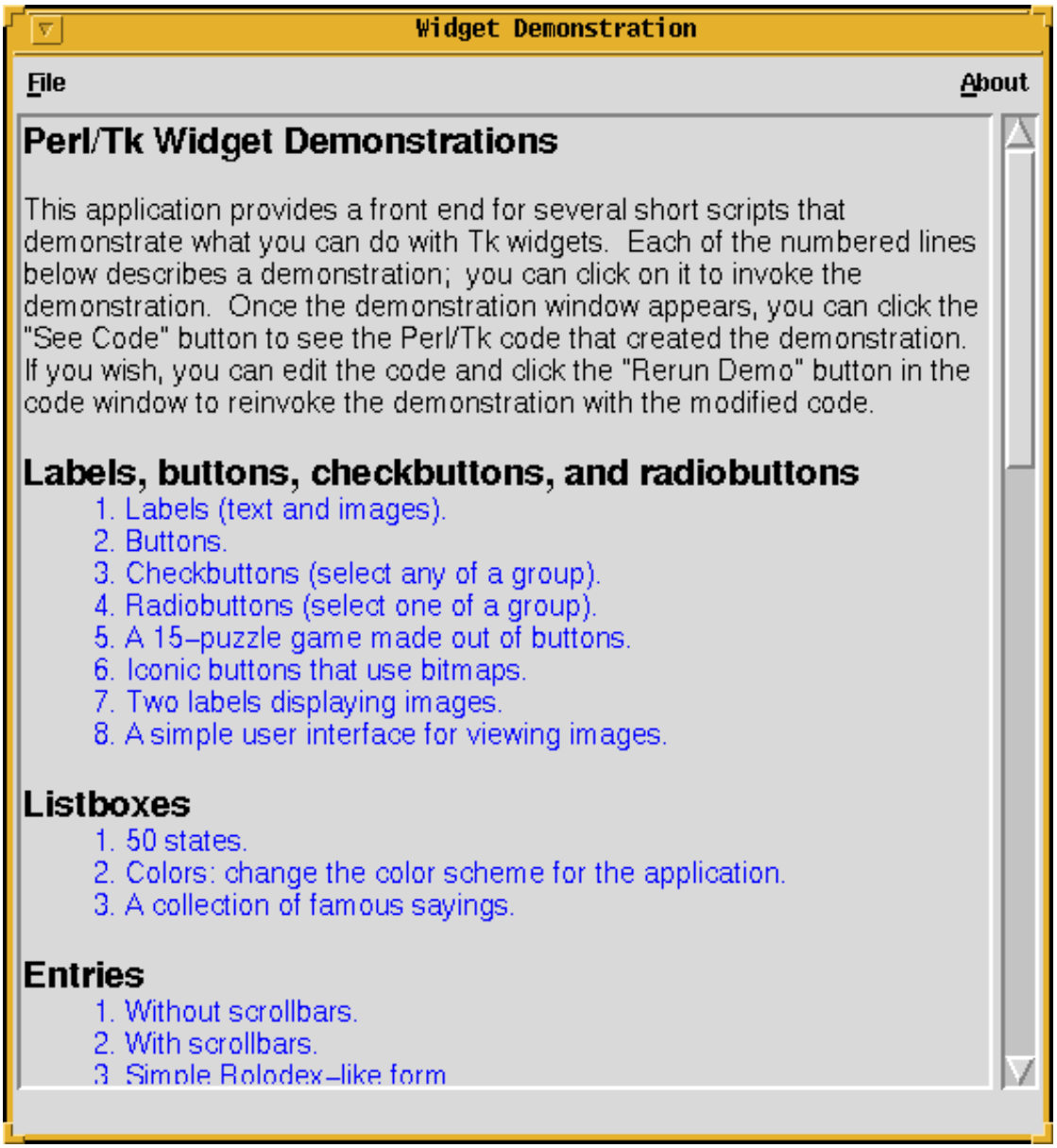
```
#!/usr/local/bin/perl -w

use Tk;
use strict;

my $top = new MainWindow;
my $frame = $top->Frame(-width => 50, -height => 50,
   -bg => 'red');
$frame->bind('<KeyPress>', [
   sub {
      my ($self, $keycode) = @_;
      printf "keycode: %s\n", $keycode;
   }, Ev('k')]);
$frame->pack; $frame->focus;
MainLoop;
```

- Bei der Methode **bind** können Ereignis-Bearbeiter in einer Vielzahl von Varianten angegeben werden:

`... => \&subname`	Zeiger auf vorhandene Prozedur
`... => sub { ... }`	Anonyme Prozedur
`... => 'methodname'`	Methode des zugehörigen Widgets
`... => [\&subname, ...]`	Mit Parameterliste
`... => [sub { ... }, ...]`	
`... => ['methodname', ...]`	

- Parameterlisten sind überflüssig, solange nicht Interesse an speziellen Ereignisparametern besteht, die über von **Ev** erzeugte Callback-Objekte zugänglich sind.

Widgets in Perl/Tk



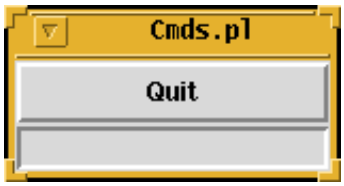
- Das bei Perl/Tk beiliegende Kommando `widget` demonstriert alle zur Verfügung stehenden Widgets in Perl/Tk mitsamt zugehörigem beispielhaften Code.

Buttons & Entries

cmds.pl

```
#!/usr/local/bin/perl -w

use Tk;
use strict;

my $command = "";    # text variable of entry
my @buttons = ();    # list of command buttons

my $main = new MainWindow;
my $quit = $main->Button(
   '-text' => "Quit",
   '-command' => sub { exit },
);
$quit->pack('-fill' => 'x');
my $entry = $main->Entry('-textvariable' => \$command);
$entry->bind('<Control-u>', sub { $command = "" });
$entry->bind('<Return>',
   sub { add_button($command); system($command); });
$entry->pack('-fill' => 'x');
MainLoop;

sub add_button {
   my $cmd = shift;
   my $button = $main->Button(
      '-text' => $cmd,
      '-command' => sub { system($cmd) },
   );
   $button->pack('-fill' => 'x');
   push(@buttons, $button);
   if (@buttons > 5) {
      shift(@buttons)->destroy;
   }
}
```

Entries



cmds.pl

```
my $entry = $main->Entry('-textvariable' => \$command);
$entry->bind('<Control-u>', sub { $command = "" });
$entry->bind('<Return>',
   sub { add_button($command); system($command); });
$entry->pack('-fill' => 'x');
```

- Alle bekannten Parameter bei Tcl/Tk wie z.B. `textvariable` sind auch in Perl/Tk bekannt.
- Statt dem Namen einer Variable wird bei Perl/Tk ein Zeiger auf eine Variable übergeben. Damit werden die Probleme symbolischer Referenzen vermieden.
- `bind` ist eine Methode aller Widgets und keine Prozedur im konventionellen Sinne.
- Die Spezifikation von Ereignismustern ist gegenüber Tcl/Tk unverändert. Sie müssen jedoch als Zeichenkette angegeben werden, da die <...> sonst im Konflikt zu dem Einlese-Operator wären.

Buttons

cmds.pl

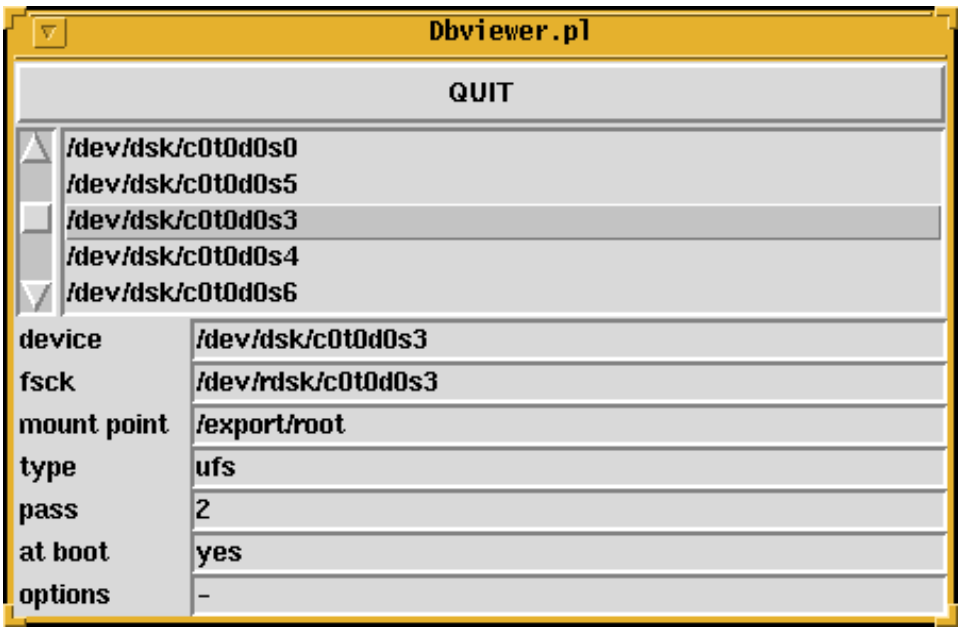
```
sub add_button {
   my $cmd = shift;
   my $button = $main->Button(
      '-text' => $cmd,
      '-command' => sub { system($cmd) },
   );
   $button->pack('-fill' => 'x');
   push(@buttons, $button);
   if (@buttons > 5) {
      shift(@buttons)->destroy;
   }
}
```

- `$cmd` innerhalb von `sub { system($cmd) }` bezieht auf die lokale Variable `$cmd` der jeweiligen Instanz von `add_button`, die die anonyme Prozedur kreiert hat.

- Dies gilt auch, wenn `add_button` jeweils längst beendet ist. `$cmd` bleibt in den jeweiligen Inkarnationen jeweils so lange leben, bis der letzte Verweis darauf verschwindet (z.B. wenn der Button eliminiert wird).

- Mit der Methode `destroy` wird ein Widget eliminiert. Der dadurch entstehende freie Platz (falls es zu sehen gewesen ist) wird auf die bekannte Weise von dem jeweiligen Geometry-Manager frisch verteilt. In diesem Beispiel rutschen dabei alle darunter liegenden Buttons nach oben.

Betrachter für kleine Datenbanken



- *dbviewer* ist eine kleine Applikation mit einer Listbox zur Auswahl eines Datensatzes und einer Reihe von Feldern zur Anzeige des selektierten Datensatzes.

Betrachter für kleine Datenbanken

dbviewer.pl

```
#!/usr/local/bin/perl -w

use IO::File;
use Getopt::Std;
use Tk;
use strict;

my $cmdname = $0; $cmdname =~ s{.*/}{};
my $usage = "Usage: $cmdname [-c] [-d delim] " .
            "[-t title_column] dbfile {fieldname}\n";
my %opts = (); getopts('cd:t:', \%opts);
my $delim = '\s+';
$delim = $opts{'d'} if defined($opts{'d'});
my $strip_comments = defined($opts{'c'});
my $tcol = 0;
$tcol = $opts{'t'} - 1 if defined($opts{'t'});
die $usage unless @ARGV > 0;
my $dbfile = shift;
my @fieldnames = @ARGV;
```

- Das Modul `Getopt::Std` definiert u.a. die Funktion `getopts`, mit der relativ einfach Optionen abgearbeitet werden können.
- Die erste Zeichenkette bei `getopts` definiert die bekannten einbuchstabigen Optionen. Wenn ein `:` hinter einem Optionsbuchstaben angegeben wird, gehört jeweils ein Argument dazu.
- `getopts` legt dann die Optionen im übergebenen assoziativen Array ab.

Betrachter für kleine Datenbanken

- Erwartet wird der Name einer Datei, in der die kleine Datenbank enthalten ist und eine Liste von Feldnamen.

- Zusätzlich werden noch folgende Optionen unterstützt:

`-c`	Kommentare, die mit `#` beginnen, sind zu entfernen.
`-d delim`	Feldtrenner.
`-t title_column`	Spalte (ab 1 zählend), in der der in der Listbox anzuzeigende Titel eines Datensatzes steht.

- Das Kommando zu dem Schnappschuß auf der vorletzten Folie:

```
dbviewer.pl -c /etc/vfstab device fsck 'mount point' \
   type pass 'at boot' options
```

Betrachter für kleine Datenbanken

dbviewer.pl

```
sub scan_db {
   my ($filename, $delim, $tcol, $strip_comments) = @_;
   my $fh = new IO::File $filename;
   die "$cmdname: $filename: $!\n" unless defined $fh;
   my @db = (); my $maxcols = 0;
   my @titles = (); my $maxtitlelen = 0; my $recno = 1;
   while (<$fh>) {
      chomp;
      next if /^\s*$/;
      next if $strip_comments && /^#/;
      my @fields = split $delim;
      $maxcols = @fields if @fields > $maxcols;
      my $title = "Record #" . $recno ++;
      $title = $fields[$tcol]
         if defined $fields[$tcol] && $fields[$tcol] ne "";
      my $length = length($title);
      $maxtitlelen = $length if $length > $maxtitlelen;
      push(@titles, $title);
      push(@db, \@fields);
   }
   $fh->close;
   return (\@db, \@titles, $maxcols, $maxtitlelen);
}
```

- `scan_db` liest die Datenbank ein und liefert die vorgefundenen Datensätze zusammen mit einigen weiteren Angaben zurück.

Betrachter für kleine Datenbanken

dbviewer.pl

```
my ($db, $titles, $maxcols, $maxtitlelen) =
   scan_db($dbfile, $delim, $tcol, $strip_comments);
my $maxfieldnamelen = 0;
foreach my $col (1 .. $maxcols) {
   $fieldnames[$col - 1] = sprintf("%02d", $col)
      unless defined $fieldnames[$col - 1];
   my $length = length($fieldnames[$col - 1]);
   $maxfieldnamelen = $length if $length > $maxfieldnamelen;
}

my @fields = ();
setup_widgets();
MainLoop;
```

- Nachdem die Daten eingelesen sind, werden die Feldnamen bestimmt (sofern nicht bereits vorgegeben) und die Oberfläche aufgebaut.

- In der Liste `@fields` wird nachher der jeweils selektierte Datensatz abgelegt.

Betrachter für kleine Datenbanken

dbviewer.pl

```
sub setup_widgets {
   my $main = new MainWindow;
   my $quit = $main->Button(
      '-text' => 'QUIT', '-command' => sub { exit });
   $quit->pack('-fill' => 'x');
   my $listbox = $main->Scrolled('Listbox' =>
      '-scrollbars' => 'w', '-width' => $maxtitlelen,
      '-height' => @{$db} > 5? 5: scalar @{$db},
   );
   $listbox->insert('end', @{$titles});
   $listbox->bind('<Button-1>',
      sub { select_record($listbox->index('anchor')) });
   $listbox->pack('-fill' => 'both', '-expand' => 1);
   foreach my $col (0..$maxcols-1) {
      my $frame = $main->Frame();
      my $label = $frame->Label(
         '-text' => $fieldnames[$col],
         '-width' => $maxfieldnamelen,
         '-anchor' => 'w',
      );
      $label->pack('-side' => 'left');
      my $field = $frame->Label(
         '-textvariable' => \$fields[$col],
         '-relief' => 'sunken',
         '-width' => 50, '-anchor' => 'w',
      );
      $field->pack(
         '-side' => 'left', '-fill' => 'x', '-expand' => 1);
      $frame->pack('-fill' => 'x');
   }
}
```

Listboxes

dbviewer.pl

```
my $listbox = $main->Scrolled('Listbox' =>
   '-scrollbars' => 'w', '-width' => $maxtitlelen,
   '-height' => @{$db} > 5? 5: scalar @{$db},
);
$listbox->insert('end', @{$titles});
$listbox->bind('<Button-1>',
   sub { select_record($listbox->index('anchor')) });
$listbox->pack('-fill' => 'both', '-expand' => 1);
```

- Mit der Konstruktor-Methode `Scrolled` ist es relativ einfach möglich, ein Widget zusammen mit der zugehörigen Scrollbar zu kreieren. Der erste Parameter ist dabei der Name der gewünschten Widget-Klasse (hier `'Listbox'`).

- Mit `'-scrollbars' => 'w'` kann die Lage der Scrollbar spezifiziert werden. Es können natürlich sowohl eine horizontale und auch eine vertikale Scrollbar gleichzeitig gewünscht werden (ist sogar Voreinstellung).

- Mit der Methode `insert` kann eine Liste von Einträgen an einen gegebenen Punkt (hier: `'end'`) hinzugefügt werden.

- Wenn ein Eintrag aus der Listbox selektiert wird, soll die Prozedur `select_record` mit dem Index des Eintrags (ab 0 zählend) aufgerufen werden.

- Die Methode `index` liefert diesen Index für einen angegebenen Punkt (hier: `'anchor'`, das für den Anfang des selektierten Bereiches steht).

Labels

dbviewer.pl

```
foreach my $col (0..$maxcols-1) {
   my $frame = $main->Frame();
   my $label = $frame->Label(
      '-text' => $fieldnames[$col],
      '-width' => $maxfieldnamelen,
      '-anchor' => 'w',
   );
   $label->pack('-side' => 'left');
   my $field = $frame->Label(
      '-textvariable' => \$fields[$col],
      '-relief' => 'sunken',
      '-width' => 50, '-anchor' => 'w',
   );
   $field->pack(
      '-side' => 'left', '-fill' => 'x', '-expand' => 1);
   $frame->pack('-fill' => 'x');
}
```

- Diese Schleife legt für alle Spalten jeweils zwei Labels an: Eines für den Namen des Feldes und eines für den Feldinhalt des selektierten Datensatzes.
- Um jeweils zwei Labels auf einer Zeile unterzubringen, werden Frames benötigt.
- Mit `'-anchor' => 'w'` wird der angezeigte Text linksbündig dargestellt.
- Die Feldinhalte des selektierten Datensatzes ergeben sich aus der Liste `@fields`.

Labels

dbviewer.pl

```
sub select_record {
   my ($index) = @_;
   my @record = @{$db->[$index]};
   foreach my $col (0..$maxcols-1) {
      if (defined($record[$col])) {
         $fields[$col] = $record[$col];
      } else {
         $fields[$col] = "";
      }
   }
}
```

- `select_record` erhält als Parameter die Nummer des selektierten Datensatzes (ab 0 zählend) und kopiert die gewünschten Felder in die Liste `@fields`. Dies hat zur Folge, daß sie in den entsprechenden Labels angezeigt werden.

Betrachter für wachsende Dateien

- Analog zu `tail -f logfile` soll `logviewer.pl` eine wachsende Datei darstellen. Dies ist beispielsweise sinnvoll für Logdateien und Dateien, in die die Ausgabe von Übersetzungs-Jobs fließt.

Betrachter für wachsende Dateien

logviewer.pl

```
#!/usr/local/bin/perl -w

use IO::File;
use Tk;
use Tk::ROText;
use strict;

my $cmdname = $0; $cmdname =~ s{.*/}{};
my $usage = "Usage: $cmdname logfile\n";
die $usage unless @ARGV == 1;
my $logfile = shift @ARGV;
die "$cmdname: Unknown file: $logfile\n"
   unless -f $logfile;
my ($log, $pid) = spawn_pipe("tail -f $logfile");
die "$cmdname: Unable to open $logfile: $!\n"
   unless defined $log;
```

- Auf der Kommandozeile erwartet `logviewer.pl` die Angabe genau eines Namens der darzustellenden wachsenden Datei.
- Die Aufgabe, das Wachstum der Datei zu verfolgen, wird an `tail -f` delegiert. `logviewer.pl` liest dann nur noch von einer Pipeline ein.

Betrachter für wachsende Dateien

logviewer.pl

```
sub spawn_pipe {
   my $cmd = shift;
   my $pipe = new IO::File;
   my $pid;
   if (defined($pid = open($pipe, "-|"))) {
      if ($pid > 0) {
         return ($pipe, $pid);
      } else {
         exec($cmd) || exit 1;
      }
   } else {
      return undef;
   }
}

sub close_pipe {
   my ($pipe, $pid) = @_;
   kill(15, $pid);
   $pipe->close;
}
```

- Das Verfolgen des Wachstums einer Datei wird `tail -f` überlassen.
- Um Hänger am Ende zu vermeiden, muß der `tail`-Prozeß vor dem Schließen der Pipeline abgeschossen werden. Sonst würde `waitpid` (aufgerufen von `$pipe->close`) solange hängen, bis der `tail`-Prozeß beendet ist.

Betrachter für wachsende Dateien

logviewer.pl

```
my $pipe = new IO::File;
my $pid;
if (defined($pid = open($pipe, "-|"))) {
   if ($pid > 0) {
      return ($pipe, $pid);
   } else {
      exec($cmd) || exit 1;
   }
} else {
   return undef;
}
```

- `$pipe = new FileHandle;` legt einen neue Dateiverbindung an, ohne sie konkret zu eröffnen.
- Im Vergleich zu `open($pipe, "$cmd |")` wird bei `open($pipe, "-|")` nur eine Pipeline angelegt und `fork` ausgeführt – jedoch kein `exec`.
- Entsprechend liefert diese Variante von `open` den Return-Wert von `fork` zurück:

`0`	Abgeleiteter Prozeß, führt nachher `exec` aus.
`> 0`	Elterlicher Prozeß.
`undef`	`fork` ging schief.

- Mehr zu dieser Technik gibt es auf der Manualseite `perlipc` unter der Rubrik "Safe Pipe Opens".

Betrachter für wachsende Dateien

logviewer.pl

```
sub close_pipe {
   my ($pipe, $pid) = @_;
   kill(15, $pid);
   $pipe->close;
}
```

- Da wir durch die spezielle Eröffnungstechnik der Pipeline im Besitz der Prozess-ID des `tail`-Prozesses sind, können wir ihn abschiessen (Signal `SIGTERM`), bevor wir die Pipeline schließen.

- Das Schließen einer Pipeline führt nicht nur zu einer `close`-Operation, sondern schließt auch ein `waitpid` ein, so daß kein herrenloser Prozeß zurückbleibt.

- Wenn kein `kill` erfolgen würde, dann käme es zu einem Hänger des elterlichen Prozesses, bis der `tail`-Prozess auf andere Weise beendet wird.

- Dies wäre normalerweise erst dann der Fall, wenn die Datei weiter wächst und die darauf erfolgende Schreiboperation von `tail` zu dem Eintreffen eines `SIGPIPE`-Signals führen würde, weil die Pipeline bereits geschlossen ist.

Textfenster

logviewer.pl

```
my $main = new MainWindow;
$main->configure('-title' => $logfile);

my $logwin = $main->Scrolled('ROText' =>
   '-borderwidth' => 3, '-relief' => 'groove',
   '-scrollbars' => 'e',
   '-height' => '20', '-width' => 80,
);
$logwin->fileevent($log, 'readable',
   sub { update_log($logwin, $log) });
```

- Widgets für Textfenster sind `Text` und `ROText`, wobei letzteres von dem Modul `Tk::ROText` exportiert wird. `Text` unterstützt frei editierbare Texte, während `ROText` alle Editieroperationen unterbindet.

- Neben den Ereignissen in Bezug auf die Benutzeroberfläche ist es auch möglich, auf Ereignisse bei IO-Verbindungen und auf zeitliche Ereignisse zu reagieren. (All dies läßt sich in einem `select`- oder `poll`-Systemaufruf regeln).

- Die Methode `fileevent` ruft die beim dritten Parameter angegebene Prozedur für jedes beim zweiten Parameter spezifizierte Ereignis für die beim ersten Parameter angegebene IO-Verbindung auf. Dies wird solange wiederholt, bis dies explizit rückgängig gemacht wird (z.B. bei dem dritten Parameter `undef` angeben).

Textfenster

logviewer.pl

```
sub update_log {
   my ($logwin, $log) = @_;
   my $line = <$log>;
   if (defined $line) {
      $logwin->insert('end', $line);
      $logwin->see('end');
   } else {
      $logwin->fileevent($log, 'readable', undef);
   }
}
```

- `update_log` wird immer dann aufgerufen, wenn es etwas (frisch!) zum Einlesen gibt. Die Lösung ist hier naiv und nimmt an, daß es sich jeweils um eine Zeile handelt. Besser wäre hier ein nicht-blockierendes Lesen unter der Verwendung von `sysread`.

- Wenn das Einlesen erfolgreich war, wird die Zeile an das Ende des Textfensters `$logwin` angefügt. Statt `'end'` wären auch beliebige andere Positionierungen denkbar.

- Mit `$logwin->see('end')` wird sichergestellt, daß das Ende zu sehen ist, falls der gesamte Text nicht mehr vollständig dargestellt werden kann.

- Sollte das Ende der Eingabe erreicht sein (z.B. weil `tail` von jemanden abgeschossen wird), dann ist darauf zu achten, daß die Bearbeitung dieser Ereignisse rückgängig gemacht wird – sonst hätten wir eine CPU-fressende Dauerschleife, da das Ereignis sofort wieder eintreffen würde.

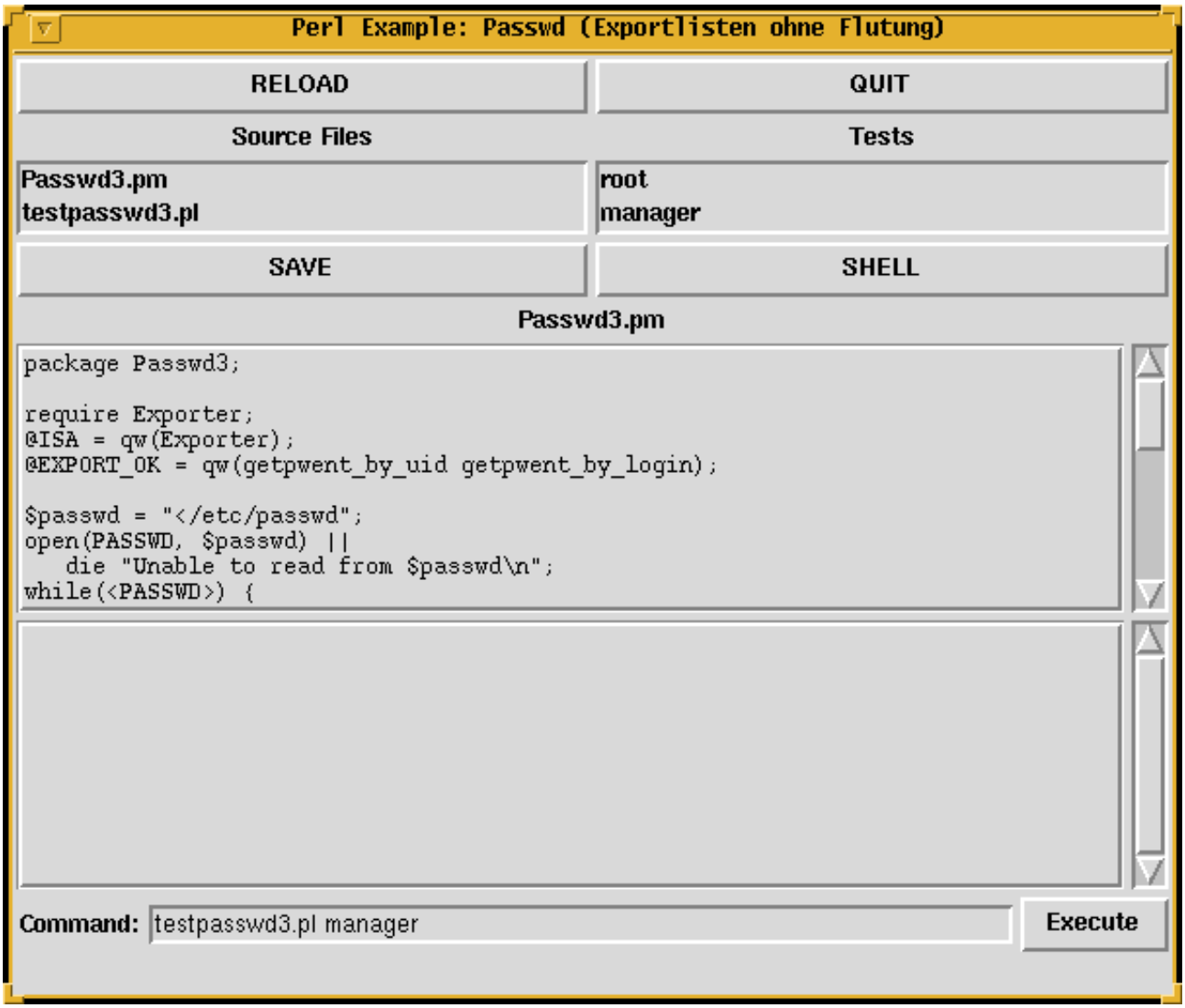
Leiste mit Buttons

logviewer.pl

```
my $frame = $main->Frame();
my @buttons = (
   ['QUIT', sub { close_pipe($log, $pid); exit 0 }],
   ['CUT',  sub { $logwin->delete('1.0', 'end') }],
);
foreach my $button (@buttons) {
   my $b = $frame->Button(
      '-text' => $button->[0],
      '-command' => $button->[1],
   );
   $b->pack('-side' => 'left',
      '-fill' => 'x', '-expand' => 1);
}
$frame->pack('-fill' => 'x');

$logwin->pack('-fill' => 'both', '-expand' => 1);

MainLoop;
```

- Buttons und ggf. auch andere Widgets, die in größeren Quantitäten auftreten, werden am geschicktesten in Abhängigkeit einer kompakten Datenstruktur angelegt.

Eine Umgebung für Perl-Beispiele



Eine Umgebung für Perl-Beispiele

passwd3.plex

```
name Passwd (Exportlisten ohne Flutung)
dir ftp://ftp.mathematik.uni-ulm.de/.../zb/passwd3
load Passwd3.pm
load testpasswd3.pl
test root testpasswd3.pl root
test manager testpasswd3.pl manager
```

- `run_plex` soll es auf einfache Weise ermöglichen, mit vorbereiteten Perl-Beispielen zu experimentieren.

- Es wird hierfür ein Skript benötigt, das angibt, von wo die dafür benötigten Dateien zu beziehen sind und ferner einige Testbeispiele vorgibt.

- Das Skript kann direkt an der Kommandozeile angegeben werden oder auch indirekt von einer Webseite referenziert werden. Mit Hilfe entsprechender MIME-Typen und der Eintragung von `run_plex` bei `mailcap` ist es möglich, diese Applikation direkt von einem Web-Browser aus aufzurufen.

- Im folgenden werden einige interessante Ausschnitte von `run_plex` gezeigt.

Die Bearbeitung eines Plex-Skriptes

runplex.pl

```
my $name;
my $dir;
my @files = ();
my @tests = ();

my %cmds = (
   'dir' => sub { $dir = $_[0] },
   'load' => sub { push(@files, $_[0]) },
   'name' => sub { $name = join(" ", @_) },
   'test' => sub { push(@tests,
                    {'tag' => shift @_, 'cmd' => [@_]}) },
);
```

- Es ist sehr typisch in Perl, statt einer `switch`-Anweisung wie sie von C her bekannt ist, ein assoziatives Array zu verwenden, das den einzelnen Fällen (typischerweise anonyme) Prozeduren zuweist.

- Hier werden den Prozeduren alle hinter dem Kommandonamen aufgeführten Parameter übergeben.

- In `@files` werden die zu ladenden Dateien notiert, in `@tests` die vorgegebenen Testfälle.

Die Bearbeitung eines Plex-Skriptes

runplex.pl

```
while(<$fh>) {
   chomp;
   next if /^#/;
   my @cmd = split /\s+/;
   die "$cmdname: Unknown command in $script: $cmd[0]\n"
      unless defined $cmds{$cmd[0]};
   &{$cmds{$cmd[0]}}(@cmd[1..$#cmd]);
}
$fh->close;
```

- Kommentare und leere Zeilen werden überlesen.
- Alles andere wird als Kommando betrachtet (auf einer Zeile, Leerzeichen dienen als Trenner).

Die Packerei

runplex.pl

```
my $top = new MainWindow;
my $title = "Perl Example";
$title .= ": $name" if defined $name;
$top->title($title);

my $bf = $top->Frame();
$bf->pack('side' => 'top', @fillx);
my @buttons = (
   ['RELOAD' => sub { load($dir, @files) }],
   ['QUIT' => sub { exit }],
);
foreach my $button (@buttons) {
   my $b = $bf->Button(
      '-text' => $button->[0],
      '-command' => $button->[1],
      '-width' => 6,
   );
   $b->pack('side' => 'left', @expandx);
}
```

- Gepackt wird von oben nach unten und innerhalb von Frames von links nach rechts.
- Bei Leiste mit Buttons empfiehlt es sich, allen die gleiche Weite mitzugeben.
- `@fillx` steht für `'fill' => 'x'` und `@expandx` ist eine Kurzform für `'fill' => 'x', 'expand' => 'y'`.

Die Listboxen

runplex.pl

```
my $ff = $top->Frame();
$ff->pack('-side' => 'top', @fillxy);
my ($ff_left, $left_lb) =
   create_listbox($ff, "Source Files", @files);
my ($ff_right, $right_lb) =
   create_listbox($ff, "Tests", @tags);
foreach my $frame (($ff_left, $ff_right)) {
   $frame->pack('-side' => 'left', @expandx);
}

$left_lb->bind('<Button-1>', \&select_file);
$right_lb->bind('<Button-1>', \&select_test);
```

- Die beiden Listboxen dienen zur Auswahl zu betrachtender Dateien oder vorgegebener Testfälle.

- Angenehm ist hier die voreingestellte Reihenfolge der "binding tags". Nachdem die für die Listboxen zuständige Klasse die Selektion bei dem Drücken der linken Maustaste durchgeführt hat, können wir diese abfragen und die entsprechende Auswahl treffen in den für die konkreten Listbox-Widgets gegebenen Verknüpfungen mit `select_file` bzw. `select_test`.

Die Listboxen

runplex.pl

```
sub create_listbox {
   my ($frame, $title, @entries) = @_;

   my $inner_frame = $frame->Frame();
   my $label = $inner_frame->Label(
      '-text' => $title,
      '-width' => $listbox_width,
   );
   $label->pack('-side' => 'top', @fillx);
   my $listbox = $inner_frame->Listbox(
      '-width' => $listbox_width,
      '-height' => $listbox_height,
   );
   $listbox->insert('end', @entries);
   $listbox->pack('-side' => 'top', @expandx);
   return ($inner_frame, $listbox);
}
```

- `$listbox_width` und `$listbox_height` wurden entsprechend der vorgegebenen Angaben festgelegt. Das erspart etwaige Scrollbars.

Auswahl einer Datei

runplex.pl

```
sub select_file {
   my $file = $left_lb->get('anchor');
   $textwin->delete('1.0', 'end');
   return unless defined $file;
   $fname = $file;
   my $fh = new IO::File $file;
   unless (defined($fh)) {
      msg("unable to open $file"); return;
   }
   while (<$fh>) {
      $textwin->insert('end', $_);
   }
   $fh->close;
   $edited_file = $file;
}
```

- Mit `get('anchor')` erhalten wir (den Beginn) der aktuellen Selektion.
- Nachdem mit `delete('1.0', 'end')` das gesamte Textfenster gelöscht worden ist, wird es mit dem Inhalt der ausgewählten Datei frisch gefüllt.
- Die Variable `$edited_file` bestimmt die Überschrift des Editierfensters.

Das Editierfenster

runplex.pl

```
my $fname = "";
my $twlabel = $top->Label('-textvariable' => \$fname);
$twlabel->pack('-side' => 'top', @fillx);
my $textwin = $top->Scrolled('Text' =>
   '-borderwidth' => 3, '-relief' => 'groove',
   '-scrollbars' => 'e',
   '-height' => '10', '-width' => 80
);
$textwin->pack('-side' => 'top', @expandxy);
```

- Das Editierfenster präsentiert eine der geladenen Dateien, die in der zugehörigen Listbox ausgewählt worden sind.
- Es ist das einzige Fenster, das auch in y-Richtung expandiert. Somit ist es relativ einfach, dieses Fenster zu vergrößern.

Die Ausführung von Kommandos

runplex.pl

```
my $cf = $top->Frame();
$cf->pack('-side' => 'top', @fillx);
$cf->Label('-text' => "Command:")->pack('-side' => 'left');
my $command = "";
my $cmd_entry = $cf->Entry('-textvariable' => \$command);
$cmd_entry->bind('<Control-u>', sub { $command = "" });
$cmd_entry->bind('<Return>', \&execute_command);
$cmd_entry->pack('-side' => 'left', @expandx);
$cf->Button(
   '-text' => "Execute",
   '-command' => \&execute_command,
)->pack('-side' => 'left');
```

- Die Kommandozeile ist mit der Variablen `$command` verbunden.
- Wenn bei der rechten Listbox einer der vorgegebenen Testfälle selektiert wird, dann wird `$command` entsprechend gesetzt.
- Mit dem Drücken von Return oder dem `Execute`-Button kann das Kommando ausgeführt werden.

Die Ausführung von Kommandos

runplex.pl

```
sub execute_command {
   return unless defined($command) && $command ne "";
   logline("*** execution of $command ***\n");
   msg("starting execution of $command");
   my $fh = new IO::File "$command 2>&1 |";
   while (<$fh>) {
      logline($_);
   }
   $fh->close;
   msg("");
}

sub logline {
   my $line = shift;
   $execwin->insert('end', $line);
   $execwin->see('end');
   $top->idletasks;
}
```

- `execute_command` eröffnet eine Pipe zu dem auszuführenden Kommando und gibt jede Zeile an `logline()` weiter.

- `logline` fügt jeweils eine Zeile zu dem Fenster, in dem die Ausgaben des auszuführenden Kommandos protokolliert werden.

- Während der Ausführung von `execute_command` können keine Ereignisse von X abgearbeitet werden. Auch die Aktualisierungen bleiben unberücksichtigt, wenn die zentrale Ereignisschleife nicht kurzfristig Gelegenheit erhält, sich darum zu kümmern. Dies geschieht mit `$top->idletasks`.

Das Herunterladen von Dateien

runplex.pl

```
sub load {
   my ($dir, @files) = @_;
   msg("Loading source files...");
   my $basedir = $ENV{'HOME'} . "/.$cmdname";
   unless (-d $basedir) {
      msg("Creating $basedir...");
      mkdir($basedir, 0777) ||
         die "$cmdname: Unable to create $basedir\n";
   }
   my $tmpdir = "$basedir/$$";
   if (-d $tmpdir) {
      my $suffix = "a";
      while (-d "$tmpdir$suffix") {
         msg("$tmpdir$suffix");
         ++ $suffix;
      }
      $tmpdir .= $suffix;
   }
   msg("Creating $tmpdir...");
   mkdir($tmpdir, 0777) ||
      die "$cmdname: Unable to create $tmpdir\n";
   chdir($tmpdir) ||
      die "$cmdname: Unable to chdir to $tmpdir\n";

   msg("Fetching files from $dir");
   if ($dir =~ m{^ftp://(.*?)/(.*)$}) {
      my ($server, $dir) = ($1, $2);
      ftp_load($server, $dir, @files) ||
         die "$cmdname: Unable to load files\n";
   } else {
      die "$cmdname: unknown dir: $dir\n";
   }
}
```

Das Herunterladen von Dateien

- Für jedes herunterzuladende Beispiel wird ein Verzeichnis unter `~/.run_plex` angelegt.

- `mkdir` gehört zu den eingebauten Prozeduren von Perl (wie soviele andere Systemaufrufe auch). Der zweite Parameter gibt die Zugriffsrechte an.

- `$$` steht für die Prozess-ID (analog zur Konvention der Bourne-Shell). Es ist eine weitere der vielen eingebauten Variablen (siehe `perlvar`-Manualseite).

Herunterladen von Dateien via FTP

runplex.pl

```
sub ftp_load {
   my ($server, $dir, @files) = @_;
   my $ftp = Net::FTP->new($server);
   unless (defined($ftp)) {
      msg("Unable to setup FTP connection to $server");
      return 0;
   }
   unless ($ftp->login("anonymous", "$cmdname@")) {
      msg("Unable to login on FTP server $server");
      $ftp->quit;
      return 0;
   }
   unless ($ftp->cwd($dir)) {
      msg("Unable to cd $dir on FTP server $server");
      $ftp->quit;
      return 0;
   }
   foreach my $file (@files) {
      msg("Loading $file...");
      unless ($ftp->get($file)) {
         msg("Loading $file... FAILED"); $ftp->quit;
         return 0;
      }
      chmod(0755, $file) if $file =~ /\.pl$/;
      msg("Loading $file... OK");
   }
   msg("Loading finished successfully");
   $ftp->quit;
   return 1;
}
```

Herunterladen von Dateien via FTP

runplex.pl

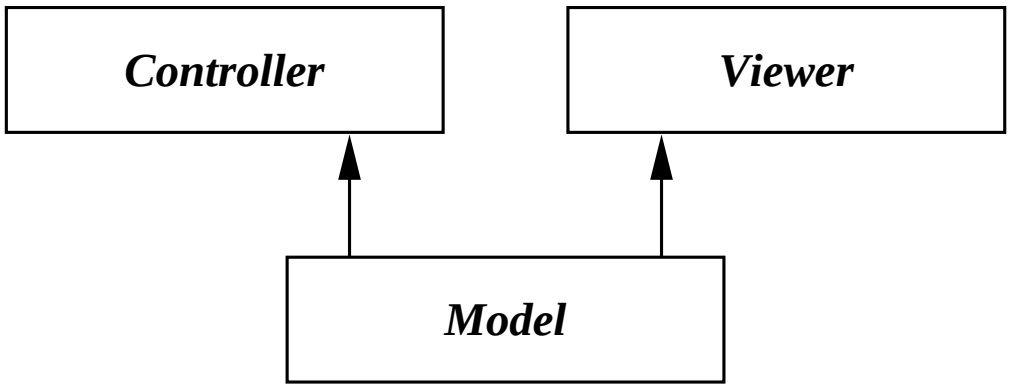
```
my $ftp = Net::FTP->new($server);
```

- Das Modul `Net::FTP` gehört zu einer Familie von Modulen unter `Net`, das elegante Schnittstellen zu einzelnen Internet-Diensten anbietet.

- Bei der Methode `login` sind Benutzername und Passwort anzugeben. Bei öffentlichen FTP-Servern ist ein Zugang mit `anonymous` (oder `ftp`) und einer E-Mail-Adresse zulässig (die allerdings nicht stimmen muß :-).

- Mit der Methode `cwd` ist ein Wechsel des Verzeichnisses möglich und mit `get` können Dateien heruntergeladen werden. Durch die Methode `quit` wird die Verbindung beendet.

Models, Viewers und Controllers



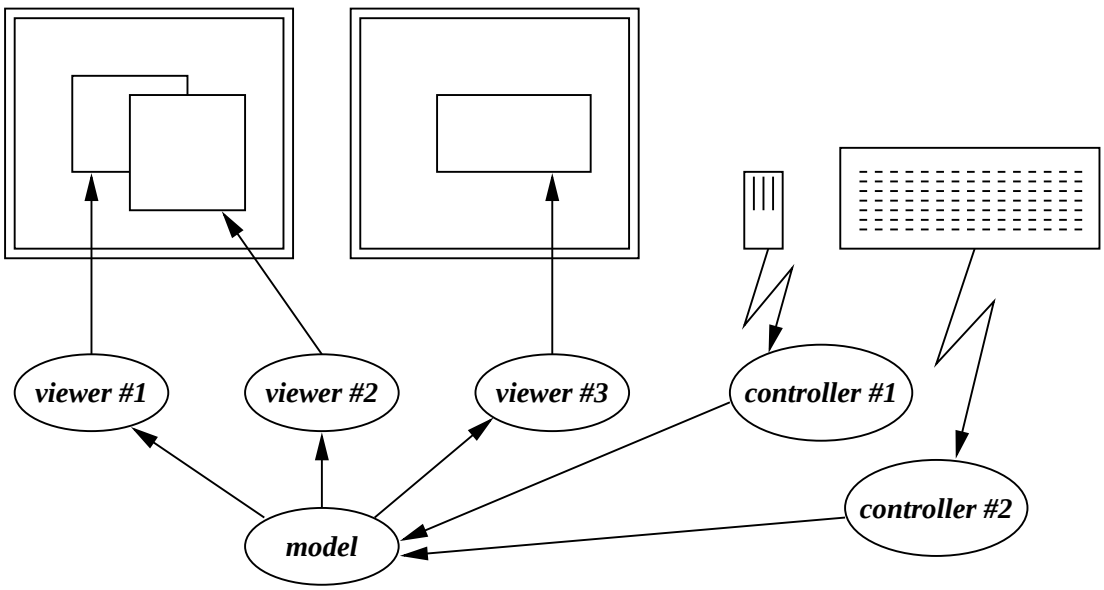
- Die Einteilung in Models, Viewers und Controllers im Zusammenhang mit graphischen Benutzeroberflächen wurde von Smalltalk-80 eingeführt. Die Vorteile dieses Ansatzes liegen in der damit gewonnen Flexibilität und der leichteren Wiederverwendbarkeit.

- Model-Klassen repräsentieren ein abstraktes Modell, das der jeweiligen Applikation zugrunde liegt. Das Modell ist im einfachen Fall ein reines Datenmodell oder bietet im allgemeinen Fall eine Reihe von Abfragemethoden und Operationen, die den Zustand verändern können.

- Viewer-Klassen sind in der Lage, bestimmte abstrakte Modelle graphisch in einer Benutzeroberfläche zu repräsentieren. Relevant ist hier, daß ein konkretes Modell gleichzeitig durch eine Vielzahl von konkreten Instanzen von Viewer-Klassen dargestellt werden kann.

- Controller-Klassen können Ereignisse von einer graphischen Benutzeroberfläche (Interaktionen) oder externen Quellen in Operationen auf dem Modell konvertieren.

Informationsflüsse



- Eingehende Ereignisse werden von den Controller-Objekten erfaßt und in Form von Operationen an das Modell-Objekt weitergeleitet.

- Alle Viewer-Objekte, die sich zuvor bei dem Modell-Objekt registriert haben, werden über alle Statusänderungen informiert.

- Bei einem verteilten System können nicht nur beliebig viele Viewer- und Controller-Objekte beteiligt sein, sondern auch mehr als ein Rechner und eine graphische Benutzeroberfläche partizipieren.

MVC-Modell bei Widgets

labels.pl

```
my $main = new MainWindow;
my $textvariable = "Hi :-)";
foreach (1 .. 5) {
   $main->Label('-textvariable' => \$textvariable)->pack;
}
$main->Entry('-textvariable' => \$textvariable)->pack;
MainLoop;
```

- Selbst bei den allereinfachsten Widgets gibt es ein dahinterliegendes abstraktes Modell.
- So gehört zu einem Label das abstrakte Modell einer Zeichenkette, das von dem Label (im Rahmen seiner Möglichkeiten) dargestellt wird.
- Im Beispiel ist die Abstraktion einer skalaren Variablen von Perl das Modell, die Labels sind allesamt Viewer und der Entry operiert simultan als Viewer und Controller.
- Leider hört bei komplizierteren Widgets (Textfenster beispielsweise) die Unterstützung bei Tk auf.

MVC-Modell bei Applikationen

- Genauso wie sich die graphische Benutzeroberfläche einer Applikation aus vielen sich zusammensetzenden Einzelkomponenten besteht, kann das MVC-Modell beginnend mit den elementaren Widgets Ebene um Ebene bis hin zur Abstraktionsebene der Applikation hochgezogen werden.

- Im Gesamtbild gibt es dann drei Hierarchien (Models, Viewers und Controllers), die jeweils in ihrer Ebene zueinander in Beziehung stehen und bei denen von Ebene zu Ebene eine Aggregation stattfindet.

MVC-Klassen in Perl

texts.pl

```
#!/usr/local/bin/perl -w

use strict;
use Tk;
use MVC::Model::Text;
use MVC::Viewer::ROText;

my $main = new MainWindow;
my $model = new MVC::Model::Text "some\nlines\nof\ntext\n";

foreach (1..3) {
   my $viewer = $main->Scrolled('ROTextViewer',
      '-model' => $model,
      '-width' => 20,
      '-height' => 5);
   $viewer->pack;
}
my $line = "";
my $entry = $main->Entry('-textvariable' => \$line)->pack;
$entry->bind('<Return>',
   sub {
      $model->append($line . "\n");
      $line = "";
   });

MainLoop;
```

- Natürlich kann die MVC-Strukturierung auch in Perl/Tk übernommen werden, wenn einmal die Grundmechanismen dafür geschaffen worden sind.

MVC::Model

MVC/Model.pm

```
package MVC::Model;

require Exporter;
use strict;

@MVC::Model::ISA = qw(Exporter);


sub new {
   my ($package, @options) = @_;
   my ($self) = bless {}, $package;
   $self->init(@options);
   return $self;
}

sub init {
   my ($self) = @_;
   $self->{viewers} = {};
}
```

- `MVC::Model` dient als Basisabstraktion für alle Modelle.
- Typischerweise werden die Initialisierungsarbeiten außerhalb des Konstruktors von einer speziellen Methode durchgeführt. In diesem Fall von `init`. Von `MVC::Model` abgeleitete Klassen dürfen zwar `init` überdefinieren – müssen jedoch sicherstellen, daß auch `init` für die Basisklassen aufgerufen wird.
- In `$self->{viewers}` verwaltet `MVC::Model` die Menge der registrierten Viewer.

MVC::Model

MVC/Model.pm

```
sub register {
   my ($self, $viewer) = @_;
   $self->{viewers}->{$viewer} = $viewer;
}

sub deregister {
   my ($self, $viewer) = @_;
   delete $self->{viewers}->{$viewer};
}

sub update {
   my ($self) = @_;
   foreach my $viewer (values %{$self->{viewers}}) {
      $viewer->update($self);
   }
}

1;
```

- Neu hinzukommene Viewer können sich mit `register` registrieren und dies später mit `deregister` rückgängig machen.

- Klassen, die von `MVC::Model` abgeleitet sind, müssen `update` aufrufen, wenn sich der Zustand in einer von außen sichtbaren Weise verändert hat.

- Alle von `MVC::Viewer` abgeleiteten Klassen benötigen eine Methode `update`, die für die Synchronisation des aktuellen Zustands eines Modells mit der Präsentierung verantwortlich ist.

MVC::Viewer

MVC/Viewer.pm

```
package MVC::Viewer;

require Exporter;
@ISA = qw(Exporter);

use Carp;
use strict;

sub new {
   my ($package, @options) = @_;
   my ($self) = bless {}, $package;
   $self->init(@options);
   return $self;
}

sub init {
   my ($self, $model) = @_;
   $self->{model} = $model;
   $model->register($self) if defined $model;
}
```

- `MVC::Viewer` ist die Basisklasse für alle Klassen, die ein Modell präsentieren.

MVC::Viewer

MVC/Viewer.pm

```
sub attach {
   my ($self, $model) = @_;
   $self->{model}->deregister($self)
      if defined $self->{model};
   $self->{model} = $model;
   $model->register($self) if defined $model;
}

sub detach {
   my ($self) = @_;
   $self->{model}->deregister($self)
      if defined $self->{model};
   $self->{model} = undef;
}

sub update {
   croak("update has not been declared by derived class");
}

1;
```

- Mit `attach` und `detach` ist es möglich, die Verbindung zwischen einem Viewer und einem konkreten Model zu ändern bzw. aufzulösen.

- Virtuelle Methoden, die überdefiniert werden müssen, sollten in der Basisklasse mit der Generierung eines Fehlers (über `croak`) vordefiniert werden.

MVC::Model::Text

MVC/Model/Text.pm

```
package MVC::Model::Text;

use MVC::Model;
use strict;

@MVC::Model::Text::ISA = qw(MVC::Model);

sub init {
   my ($self, $init) = @_;
   $self->SUPER::init();
   $init = "" unless defined $init;
   $self->{text} = $init;
}

sub get {
   my ($self) = @_;
   return $self->{text};
}
```

- `MVC::Model::Text` ist ein (sehr einfaches) Modell für Text (Zeichenfolge, die über beliebig viele Zeilen gehen kann).
- Das Durchlaufen der Initialisierungskette für ein neues Objekt durch alle beteiligten Klassen wird durch `$self->SUPER::init()` sichergestellt. `SUPER` ist ein Verweis auf die übergeordnete Klasse.

MVC::Model::Text

MVC/Model/Text.pm

```
sub set {
   my ($self, $text) = @_;
   $text = "" unless defined $text;
   if ($text ne $self->{text}) {
      $self->{text} = $text;
      $self->update();
   }
}

sub append {
   my ($self, $text) = @_;
   if (defined ($text) && $text ne "") {
      $self->{text} .= $text;
      $self->update();
   }
}

1;
```

- Mit `set` oder `append` ist es möglich, den aktuellen Zustand des Text-Modells sichtbar zu verändern. Entsprechend ist es notwendig, die Methode `update` jeweils anschließend aufzurufen, um Änderungen an die Viewer weiterzugeben.

MVC::Viewer::ROText

MVC::Viewer::ROText.pm

```
package MVC::Viewer::ROText;

use Carp;
use Tk;
use Tk::ROText;
use MVC::Viewer;
use strict;

@MVC::Viewer::ROText::ISA =
   qw(Tk::ROText MVC::Viewer);

Construct Tk::Widget 'ROTextViewer';

sub init {
   my ($self, @options) = @_;
   $self->SUPER::init(@options);
}
```

- Die Klasse `MVC::Viewer::ROText` ist eine Ableitung sowohl von dem Widget `Tk::ROText` (repräsentierendes Widget) als auch der Basisklasse für Viewer `MVC::Viewer`.

- Mit `Construct Tk::Widget 'ROTextViewer'` wird dafür gesorgt, daß `ROTextViewer` als Konstruktor-Methode für alle Widgets eingetragen wird, die andere aufnehmen können (`MainWindow`, `TopLevel`, `Frame` usw).

- Da `Tk::ROText` innerhalb `@ISA` zuerst genannt wird, hat der Konstruktor (Methode `new`) von `Tk::Widget` Priorität vor dem von `MVC::Viewer`.

MVC::Viewer::ROText

MVC::Viewer::ROText.pm

```
sub InitObject {
   my ($self, $args) = @_;
   my $model = undef;
   if (defined($args->{'-model'})) {
      $model = $args->{'-model'};
      delete $args->{'-model'};
   }
   $self->SUPER::InitObject($args);
   $self->init($model);
   $self->update($model);
}

sub update {
   my ($self, $model) = @_;
   $self->delete('1.0', 'end');
   $self->insert('end', $model->get());
   $self->see('end');
}

1;
```

- Bei den von `Tk::Widget` abgeleiteten Modulen geht die Initialisierungskette nicht über die Methode `init`, sondern über `InitObject`. Dies ist ein glücklicher Umstand, da es sonst bei *multiple inheritance* nicht ganz einfach ist, alle beteiligten Klassen an der Initialisierung teilhaben zu lassen.