Objektorientierte Programmierung mit Perl 5

Gesegnet

Michael Schilli

Perl ist schon seit geraumer Zeit in Version 5 erhältlich und bietet seit dieser Release auch die Vorzüge der objektorientierten Programmierung. Doch die revolutionären neuen Features finden außerhalb der mitgelieferten Online-Manual-Pages kaum Erwähnung – obwohl sie in Programmen wie dem Tk-Perl-Paket bereits eifrig genutzt werden.

Zur besseren Strukturierung umfangreicher Programmlogik gab es schon in Perl Version 4 die Methode der Modularisierung – Codepakete konnten in externe Dateien ausgelagert und mittels des *require*-Befehls zum Hauptprogramm hinzugebunden werden. Hierbei stellte das *package*-Konstrukt sicher, daß jedes Programmpaket seinen eigenen Namensraum für Variablen erhielt, ohne sich beispielsweise mit dem des Hauptprogramms zu überschneiden.

In Version 5 hat die Objektorientierung Einzug gehalten. Folgende Neuerungen bahnen den Weg:

- Definition von Klassen einschließlich Vererbung,
- Konstruktoren/Destruktoren für Objekte sowie
- Kapselung von Objektdaten durch Zugriff über Member-Funktionen.

Durch das hiermit mögliche höhere Abstraktionsniveau im Programmdesign läßt sich eines der Hauptziele objektorientierter Programmierung erreichen: die leichtere Wiederverwendbarkeit einmal geschriebenen Codes.

Über Referenzen zu Objekten

Doch vor die Früchte der Objekte hat Perl-Autor Larry Wall den steinigen Weg der Referenzen gesetzt. Perl 5 wartet nämlich mit einem völlig neuen Konzept auf: Es ist nun möglich, Referenzen auf verschiedene Konstrukte zu definieren, beispielsweise auf Variablen verschiedenen Typs (Skalare, Arrays und Hashs), aber auch auf Funktionen und sogar Codeblöcke (*anonymous subroutines*). So legt zum Beispiel *$scalarref = \$foo;* eine Referenz des Skalars *$foo* in der Variablen *$scalarref* ab. Anschließend ist der Zugriff auf den Wert von *$foo* sowohl direkt als auch über die Referenz *$scalarref* möglich, also ist *$bar = $foo;* äquivalent zu *$bar = $$scalarref;* und zu *$bar = ${$scalarref};*

Beim Dereferenzieren schließt Perl jedoch nie 'magisch' auf den zugrundeliegenden Datentyp, sondern folgert diesen aus dem Kontext. So setzt *$hashref = \%myhash;* eine Referenz auf eine Variable vom Typ Hash. Der Zugriff auf *%myhash* erfolgt nun entweder über das Original mit *$myhash{'key'} = 'value';* oder äquivalent per Dereferenzierung mittels *$$hashref{'key'} = 'value';* beziehungsweise *${$hashref}{'key'} = 'value';*. Ganz neu in Perl 5 ist zusätzlich *$hashref->{'key'} = 'value';* zulässig – als ein 'syntaktisches Zuckerl', wie es in den Manual-Pages [1] so schön heißt.

Soweit der kleine Exkurs in das Reich der Referenzen. Er war notwendig, um den ersten Schritt in die objektorientierte Welt von Perl 5 zu unternehmen: die Klassendefinition.

Zugriff auf Variablen mit *$this*

Eine Ernüchterung gleich vorweg: Eine Klasse ist nichts weiter als ein 'package', dessen Subroutinen auf Objektreferenzen zugreifen und damit Methoden im Sinne der objektorientierten Programmierung sind. Jede Subroutine erwartet als erstes Argu-

ment eine Referenz auf das zugehörige Objekt. Über diesen meist *$self* oder *$this* genannten Parameter (die Ähnlichkeit mit dem gleichnamigen Pointer in C++ ist beabsichtigt) können innerhalb der Subroutine dann die Instanzvariablen des Objektes angesprochen werden.

Eine erste Klassendefinition läßt sich einfach als

```
package Myclass;
# Es folgen die Methoden ...
sub .. { ...
 }
sub .. { ...
 }

1;
```

notieren. Dies definiert lediglich ein *package*, üblicherweise in einer eigenen Datei. Als Return-Wert liefert es getreu der Perl-Konvention eine 1. Noch fehlen dieser Klasse die Methoden, einschließlich des Konstruktors.

Vor der Instantiierung der ersten Klasse ist es notwendig, auf die Objektstruktur einzugehen. Perl stellt keinen Automatismus zur Erzeugung von Objekten bereit, sondern erfordert etwas 'Handarbeit', wobei folgende Überlegung weiterhilft:
Jedes Objekt besitzt
- einen Bezug zu seiner Klasse,
- damit einen Bezug zu den Namen aller Instanzvariablen und zu den Methoden und
- einen Satz von Werten für die Instanzvariablen.

Da Perl ein Objekt fest mit seiner Klasse verbindet, also einen Bezug zu einem *package* herstellt, ist lediglich ein Namensraum für die Instanzvariablen des Objekts zu schaffen. Dieser stellt sicher, daß die Variable *status* des Objekts *A* einen anderen Wert annehmen kann als die gleichnamige Variable des Objekts *B* derselben Klasse.

bless bindet Objekte an Klassen

Ein jedem Objekt zugeordneter eigener Hash implementiert diesen Namensraum. Seine *keys* sind die Variablennamen, seine *values* deren Werte. Dieser Hash ist an die Klasse gebunden und seine Referenz dient als Verweis auf das Objekt. Den Konstruktor für ein Objekt definiert das *package Myclass* folgendermaßen:

```
package Myclass;
# Konstruktor
sub new
{ bless {};
}

1;
```

Als erstes erzeugt *{}* die Referenz eines anonymen Hash, der den Namensraum der Instanzvariablen des neuen Objektes repräsentiert. Der Hash benötigt keinen Namen, da er später über die Objektreferenz eindeutig zugänglich ist. Er ist zum Zeitpunkt der Initialisierung leer, das heißt, er enthält keinerlei *key-value*-Paare: Für das Objekt sind noch keine Instanzvariablen definiert. Über seine *keys* werden später die Instanzvariablen angesprochen und mit Werten belegt – der Hash 'ist' sozusagen das Objekt. Die Funktion *bless* nimmt nun die Referenz des Hash und teilt ihm mit, daß er zur Klasse *Myclass* gehört. Anschließend gibt *bless* die Referenz auf den Hash zurück, und der Konstruktor *new* reicht sie weiter ans Hauptprogramm, welches sie für zukünftige Zugriffe auf das Objekt behält. Bis jetzt existiert nur die Referenz auf einen namenlosen Hash, der fest mit der Klasse *Myclass* verkettet ist. In anderen Worten: eine Objektreferenz. Der Aufruf des Konstruktors aus dem Hauptprogramm sieht nun folgendermaßen aus:

```
use Myclass;
$myobj = Myclass->new();
```

Nach dem Einbinden von *Myclass* liefert *Myclass->new()* eine Referenz auf das erzeugte Objekt.

Lauf' ich oder lauf' ich nicht?

Nun soll die Methode *check()* definiert werden, die den Zustand einer (noch nicht vorhandenen) Instanzvariablen namens *running* überprüft. Bei undefiniertem Wert oder für *running == 0* soll sie 0 zurückliefern, in allen anderen Fällen 1. Das Hauptprogramm ruft diese Methode als *$ret = $myobj->check();* auf, dabei übergibt -> die Referenz des rufenden Objekts als erstes Argument an die Subroutine. Die Definition von *check* schreibt sich

```
sub check
{ my $self = shift;
defined $self->{'running'} &&
$self->{'running'};
}
```

Hierbei holt die Subroutine als erstes die Referenz des rufenden Objekts aus der Argumentliste und legt sie in der lokalen Variablen *self* ab. Anschließend greift *$self->{'running'}* auf den anonymen Hash des Objekts zu, also den Namensraum seiner Instanzvariablen. *check* prüft die entsprechende Variable einerseits auf Existenz und andererseits auf Null und liefert das Ergebnis des Tests zurück.

Noch einmal zur Klarstellung: In der Variablen *$self* liegt die Referenz auf ein Objekt, also auf den im Konstruktor definierten anonymen Hash. Der Ausdruck *$self->{'running'}* bewirkt nichts anderes als den Zugriff darauf.

Perls Referenzmechanismus erledigt selbständig das Zerstören von nicht mehr benötigten Objekten. Im Beispiel verschwindet der Namensraum-Hash des Objektes automatisch, nachdem die Objektreferenz ungültig wird – also erst beim Verlassen des Hauptprogramms. Sobald eine Variable sowie sämtliche ihrer Referenzen

iX-TRACT

- Seit Version 5 unterstützt die Skriptsprache Perl objektorientiertes Programmieren.
- Ähnlich wie in C++ erhalten alle Methoden als erstes Argument einen Zeiger auf die aufrufende Instanzvariable. Es gibt jedoch keine Default-Konstruktoren.
- Ein anonymer Hash stellt für jede Instanzvariable einen eigenen Namensraum zur Verfügung. Die Funktion *bless* bindet diesen Hash an seine Klasse.

```
#!/usr/bin/perl -w
### ----------------------------------
### Module:  mytest
### Purpose: Launch and control a sleep
###          process in background
### Author:  Michael Schilli
### Date:    95/12/28
### ----------------------------------

use Process;

# Create a new process object
$myproc=Process->new();

# Start the background process
$myproc->start("sleep 10");

# Check 3 times if process is running
for($i=1; $i<=3; $i++)
{ sleep(1);
  print "Prozess laeuft ",
    $myproc->poll() ? "\n" : "nicht!\n";
}

# Terminate process
$myproc->kill() || print "Kill failed\n";
```

Kurzes Programm zum Testen der Klasse *Process* (Listing 1)

PC/HOST COMMUNICATION

Lokale Netze, LAN/WAN-Anbindungen, Internet-Zugang

TCP/IP-Netzwerksoftware

OnNet von ftp Software: Mit der neuen Version 2.0 festigt der Marktführer im Bereich TCP/IP-Software für MS Windows seine Position. OnNet ist verfügbar für MS Windows, Windows 95 und Windows NT, enthält viele Windows-Applikationen wie Telnet, ftp, lpr, NFS, Mail, Internet Tools, Document Viewing,...
Marathon von NCD: Auch NCD bietet jetzt ein TCP/IP Paket für MS Windows an. Nachdem PC-Xware schon im Laufe der letzten Versionen mit immer mehr TCP/IP-Funktionalität angereichert wurde, war jetzt der Zeitpunkt gekommen, hieraus ein eigenständiges Produkt zu schnüren.
ENet*NFS von EMATEK: ENet*NFS ist ein NFS-Server für DOS, MS Windows und Windows NT, ist ein stand-alone Produkt und als Server für heterogene Netze einsetzbar.

X-Terminal-Emulation

PC-Xware von NCD: Die 7 Jahre der Entwicklung von X-Servern für PCs spiegeln sich in PC-Xware wieder. PC-Xware ist ohne Zweifel das Top-Produkt in der Kategorie X-Server für MS Windows und Windows NT. PC-Xware enthält die komplette TCP/IP Netzwerksoftware Marathon.
X OnNet von ftp Software: X OnNet heißt das entsprechende Produkt von ftp Software und ist eine interessante Alternative zu PC-Xware für Benutzer des OnNet TCP/IP-Paketes, da es als add-on für diese Netzwerksoftware optimiert wurde.

Internet-Anschluß aus einer Hand

Das Internet wird verkauft wie eine brandneue Entwicklung, reicht mit seiner Basis des TCP/IP Netzwerkprotokolls jedoch schon in die 70er Jahre zurück. Mit der langjährigen Erfahrung im Bereich TCP/IP Vernetzung sind EMATEK und ihre Partner daher die optimale Basis, um Ihren Internet-Anschluß aus einer Hand zu realisieren.
Explore von ftp Software: Die Internet-Tools von OnNet sind unter dem Namen Explore auch als kostengünstiges Einzelpaket zu erhalten. Explore ermöglicht dabei beides, Arbeiten im LAN und Internet-Zugriff über Dialer und Modem.
Mariner von NCD: Mariner for Windows ist die Antwort von NCD auf die Herausforderung Internet. Der WWW Browser von Mariner ist eine gelungene Mischung von Mosaic und dem Netscape Navigator.

Internetzugriff mit MAZ und IP ISDN Routern von INS

Um unseren Kunden optimale Gesamtlösungen bieten zu können, erweitern wir unsere Basis strategischer Partnerschaften ständig. Schon bewährt ist die Zusammenarbeit mit dem großen deutschen Internet-Provider MAZ. Ab sofort bieten wir den INS Firewall Router an. Dieser TCP/IP-ISDN Router unterstützt alle gängigen ISDN Protokolle, verfügt über Firewall-Sicherheitsmechanismen, beinhaltet SNMP und Datenverschlüsselung und stellt damit eine kostengünstige Software-/Hardwarelösung dar.

X11 PrintManager

Drucken und Plotten aus JEDER X11 Applikation. So einfach wie bei MS Windows!

Unabhängig davon, wie Sie bisher Drucken und Plotten unter X11 realisiert haben: Mit dem *CGI-Standard* X11 PrintManager von EMATEK werden Drucken und Plotten unter X11 so einfach wie unter MS Windows.

Hardwarehersteller können nun ihre X11 Treiber zusammen mit ihren Geräten ausliefern und sind somit in der Lage, den Anteil ihres UNIX Marktes wesentlich zu vergrößern.

Softwarehersteller müssen nun nicht mehr Zeit und Geld investieren, um ihren Applikationen den Zugang zu den aktuellsten Ausgabegeräten zu ermöglichen, da sie den X11 PrintManager über eine API nutzen können.

Endbenutzer von X11 Applikationen sind nun in der Lage, eine Vielzahl von Ausgabegeräten mit ihrem UNIX System zu nutzen, ohne dabei auf die Druckfähigkeiten ihrer X11 Applikation angewiesen zu sein.

X-Terminal- oder PC X-Server-Anbieter und Benutzer können nun ein echtes "print-from-X" Werkzeug einsetzen, anstatt sich wie bisher mit Hardcopies zu begnügen oder mit riesigen PostScript Files zu hantieren.

Denn basierend auf dem Computer Graphics Interface CGI Standard hat EMATEK jetzt eine Druck- und Treiberarchitektur entwickelt. Der X11 PrintManager besteht dabei im wesentlichen aus zwei Teilen, dem Druckserver einschließlich eines Printer Setup Dialogs, und einer Vielzahl von schon jetzt vorhandenen Gerätetreibern, die als separate Module jederzeit ausgetauscht, erweitert und aktualisiert werden können:

● **HPGL/2** ● **Colour PostScript** ● **PCL5** ● **HP Deskjet Drucker Serie** ● **Nadeldrucker** ● **CGM Metafile**

Mit dem X11 PrintManager ist das Problem des Druckens und Plottens unter X11 auf einen Schlag gelöst, und das **auf der Basis eines offenen Standards** und, für Sie bestimmt genau so wichtig, **zu einem wirklich niedrigen Preis.**

Bitte fordern Sie ausführliche Informationen an, wir beraten Sie gerne.

Wir setzen auf Standards, setzen auch Sie auf Standards, um den Werterhalt Ihrer Applikationen langfristig zu sichern.

CeBIT'96 HANNOVER
14.—20. 03. 1996

Sie finden uns in Halle 3, Stand B23!

15 Jahre EMATEK

EMATEK GmbH
Subbelrather Straße 17 · D-50823 Köln
Tel. (0221) 512074 · Fax (0221) 529666
Email: gsscgi@ematek.de

entweder manuell mittels des *undef*-Operators gelöscht wurden oder aber *out of scope* gingen (zum Beispiel Verlassen der Prozedur bei Auto-Variablen), gibt der Interpreter alle zugehörigen Ressourcen frei. Für die bisher definierten Perl-Objekte, die als einzige Ressource einen anonymen Namens-Hash belegten, verschwindet dieser mit der letzten Objektreferenz. Sind hingegen noch zusätzliche Aufräumarbeiten für eine Klasse notwendig, spricht nichts gegen die Implementierung eines Destruktors, der konventionsgemäß *sub delete {...}* heißen sollte und ausdrücklich mit *$myobj->delete();* aufzurufen ist.

Kontrolle einmal pro Sekunde

Hiermit wäre der Einstieg geschafft. Als ein nützliches Anwendungsbeispiel soll die Klasse *Process* die Steuerung eines Unix-Prozesses modellieren. Sie verfügt über Methoden, um den Prozeß im Hintergrund zu starten, zu beenden und seinen Zustand abzufragen ('läuft', 'läuft nicht'). Ein Objekt dieser Klasse repräsentiert so einen Prozeß, der Auskunft über seinen Zustand geben kann, selbst wenn er nicht aktiv ist. Einzige Instanzvariable der Klasse ist die Prozeßnummer, die beim Starten gesetzt und beim Pollen beziehungsweise Anhalten an die entsprechenden Systembefehle übergeben wird.

Das Testprogramm von Listing 1 startet einen zehn Sekunden dauernden *sleep*-Prozeß im Hintergrund, pollt ihn anschließend dreimal im Sekundentakt und bricht ihn schließlich ab. Vertauscht man die Zahlenwerte dergestalt, daß ein dreisekündiger *sleep*-Prozeß gestartet und zehn Sekunden lang gepollt wird, stellt das Skript nach drei Sekunden fest, daß der Prozeß nicht mehr existiert, und meldet dies.

Zu Anfang bindet *use Process* die Implementierung der Klasse *Process* ein. Hierbei sucht der Perl-Interpreter in allen für das Skript definierten *include*-Verzeichnissen nach einem Modul des Namens *Process.pm*, das das *package Process* enthält.

Listing 2 zeigt *Process.pm*. Neben dem Konstruktor *new()* sind die Methoden *start()*, *poll()* und *kill()* implementiert. Der Rest ist Unix: Die *start*-Methode erzeugt mittels des System-Call *fork()* einen Child-Prozeß, der wiederum das angegebene Programm startet. Aufmerksamkeit verdient noch der Signal-Handler, der beim Terminieren des Child die Entstehung eines Zombie-Prozesses verhindert. Eine Instanzvariable speichert die Prozeßnummer *pid*, so daß nachfolgende Aufrufe der Methoden *poll* beziehungsweise *kill* sie verwenden können. *poll* beruht darauf, daß man einem laufenden Prozeß fehlerfrei ein Zero-Signal senden kann, die *kill*-Methode hingegen schickt ein SIGTERM oder ein anderes spezifiziertes Signal.

Leider haben die praktischen neuen Features noch nicht Einzug in das legendäre Camel-Buch, die Bibel der Perl-Programmierer, gehalten [2]. Bleibt zu hoffen, daß dies in der folgenden Auflage geschieht. Die nächste Folge behandelt Aspekte der Vererbung und Perl 5. (ck)

MICHAEL SCHILLI

arbeitet als Softwareentwickler bei softlab München (EMail: scx@softlab.de).

Literatur

[1] Perl Manual pages, insbesondere *perlref, perlobj, perlbot*

[2] Larry Wall und Randal L. Schwartz; Programming Perl; O'Reilly and Associates, Sebastopol, CA, 1991

```
### ----------------------------------------
### Module:  Process.pm
### Purpose: Launch and control programs
###          in background mode
### Author:  Michael Schilli
### Date:    95/12/28
### ----------------------------------------
package Process;

###----------------------------------------
### Function: Constructor for process object
### Syntax:   $proc_obj=Process->new();
###----------------------------------------
sub new
{
  # Bless an anonymous hash to the package
  # for holding the instance variables
  bless {};
}

###----------------------------------------
### Function: Launch a process in background
### Syntax:   $ret = $proc_obj->start("prg");
###----------------------------------------
sub start
{
  my $self  = shift;
  my $func  = shift;

  # Reap zombies
  $SIG{'CHLD'} = sub { wait };

  # Fork
  # Child:  Execute function specified in bg
  # Parent: Save child pid in instance
  #         variable and return
  exec "$func" unless $self->{'pid'}=fork();
}

###----------------------------------------
### Function: Check if the process is still
###           running
### Syntax:   $ret = $proc_obj->poll();
###----------------------------------------
sub poll

{
  my $self      = shift;

  # Check if pid is defined and, if so,
  # send zero signal to process to check if
  # it's running
  defined $self->{'pid'} &&
    kill(0, $self->{'pid'});
}

###----------------------------------------
### Function: Terminate a launched process
###           and 'wait' for it
### Syntax:   $ret = $proc_obj->kill();
###        or $ret = $proc_obj->kill(SIG...);
###----------------------------------------
sub kill

{
  my $self = shift;
  my $sig  = shift;

  # If no signal is provided, take SIGTERM
  # as default
  $sig = SIGTERM unless defined $sig;

  # Function fails if no pid is set
  return undef if !defined $self->{'pid'};

  # Send kill signal
  kill($sig, $self->{'pid'}) || return undef;

  # Reap zombies
  waitpid($self->{'pid'}, 0) == $self->{'pid'}
    || return undef;

  undef $self->{'pid'};

  1;
}

1;
```

Die Klasse *Process* definiert dieses Package (Listing 2).