Europas meistgekaufte Unix-Zeitschrift

öS 62,– sfr 7,50 **DM 7,50** H 10554

4 399148 507509 04

iX MULTIUSER MULTITASKING MAGAZIN

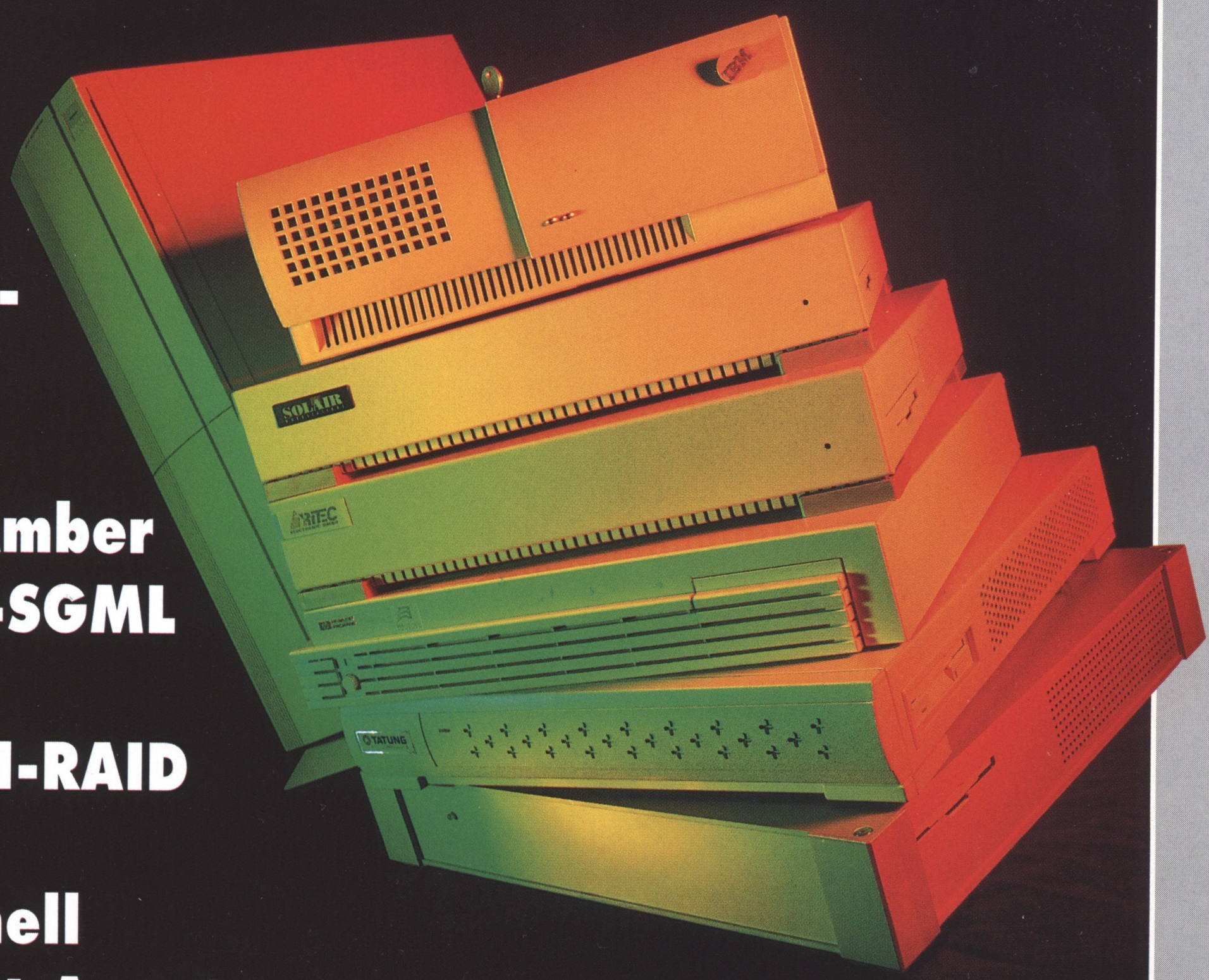
4

April 1996

Vergleichstest Low-Cost-Workstations:

7 unter 7000 Mark

AIX, HP-UX, Linux, Solaris



Erste Betaversion:

NT 4.0

Benutzerservice:

Helpdesk-Systeme

WWW-Publishing:

Acrobat Amber Linuxdoc-SGML

Massenspeicher:

Ultra-SCSI-RAID

Internet:

Secure Shell Intelligent Agents

Softwareentwicklung:

GNU Crosscompiler

Grundlagen, Konzepte, Anbieter

Data Warehouse

Mehrdimensionale Datenanalyse mit OLAP

April 1996 http://www.ix.de

Objektorientierte Anwendungen mit Perl 5

Erben und erben lassen

Michael Schilli

In der letzten Ausgabe ging es um die Grundlagen der objektorientierten Programmierung mit Perl. Nachdem die Klippen bis zur ersten Klassendefinition umschifft sind, kann der nächste Schritt folgen: Klassen dürfen Eigenschaften vererben.

Vererbung ist eine Möglichkeit, Beziehungen zwischen Konzepten herzustellen. Klassen repräsentieren Konzepte: Läßt sich von einer Klasse *Derivedclass* sagen, daß sie 'eine Art' einer Klasse *Baseclass* ist, spricht vieles dafür, die Eigenschaften von *Baseclass* an *Derivedclass* zu vererben.

Offensichtlich sind die Vorteile dieses Verfahrens: Nicht jede Klasse muß von Grund auf implementiert werden, vielmehr reicht es, ein bekanntes Konzept zu übernehmen und zusätzlich erforderliche Funktionalität zu definieren. Um diesen Mechanismus in Perl zu aktivieren, ist – wie gewohnt – etwas Handarbeit notwendig.

Ein Package erhält die Lizenz zum Ausführen fremder Methoden über einen Eintrag in dem Package-eigenen Array *@ISA*. Der Name dieses Array soll wohl die vererbungstypische 'is a'-Beziehung von Klassen symbolisieren, im Sinne von 'ist ein' oder 'ist eine Art von'. Will ein Objekt eine Methode ausführen, die in dem Klassen-Package nicht definiert ist, durchsucht der Interpreter alle in *@ISA* enthaltenen Packages auf die geforderte Methode hin. Dies entspricht einer *depth-first*-Suche in der Klassenhierarchie: Jedes untersuchte Package kann nämlich seinerseits ein *@ISA*-Array definieren und dadurch Methoden anderer Packages nutzen.

Dadurch kann ein Objekt nicht nur auf seine eigenen Methoden zugreifen, sondern auch auf die aller Packages, aus denen seine eigene Klasse direkt oder indirekt abgeleitet ist. Dies erfolgt völlig transparent, also ohne Angabe, in welchem Package sich die gesuchte Methode letztendlich befindet.

So legt

```
package Deripac;
@ISA = ('Basepac1', 'Basepac2');
```

fest, daß das aktuelle Package die Funktionalität von *Basepac1* und *Basepac2* erbt.

Anders als zum Beispiel in C++, wo der Compiler vor die Erzeugung eines abgeleiteten Objekts den Aufruf des Basisklassenkonstruktors setzt, ist in Perl der Konstruktor eine Methode wie jede andere. Wenn für die Erzeugung eines abgeleiteten Objekts zunächst der Konstruktor der Basisklasse aufzurufen ist, muß die Implementierung dies explizit sicherstellen.

Üblicherweise ruft der Konstruktor einer abgeleiteten Klasse einfach den der Basisklasse auf. Falls die abgeleitete Klasse keine zusätzlichen Daten initialisieren muß, gibt es aber auch eine elegantere Lösung: Die abgeleitete Klasse erbt einfach den Konstruktor der Basisklasse.

Konstruktoren einfach erben

Listing 1 zeigt die Vererbung eines Konstruktors: Die Basisklasse *Basepac* enthält den Konstruktor *new*, die abgeleitete Klasse *Deripac* definiert nur eine zusätzliche Methode *derimethod*.

Wenn *main* über *Deripac->new()* den Konstruktor der Klasse *Deripac* aufruft, greift Perl, da diese Methode

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

#---------------------------------
package Basepac;
#---------------------------------
### Constructor
sub new
{ my $type = shift;
  my $self = {};
  bless $self, $type;
}

### Sample Method
sub basemethod
{ print "basepac method!\n";
}

#---------------------------------
package Deripac;
#---------------------------------
@ISA = qw ( Basepac );

### Sample Method
sub derimethod
{ print "deripac method!\n";
}

#---------------------------------
package main;
#---------------------------------
$derobj = Deripac->new();
$derobj->derimethod(); # own method
$derobj->basemethod(); # inherited
```

In diesem Codestück sind drei Packages definiert: Die Klasse *Basepac*, die aus ihr abgeleitete Klasse *Deripac* sowie das Hauptprogramm *main* (Listing 1).

in *Deripac* nicht existiert, einfach auf die Funktion *new()* der Basisklasse zurück. Damit das entstehende Objekt nun zur Klasse *Deripac* und nicht zu *Basepac* gehört, muß der Konstruktor den Anforderungen der Vererbbarkeit genügen. Die im ersten Teil dieses Artikels [1] verwendete Kurzform

```
sub new
{ bless {};
}
```

reicht dafür nicht mehr, denn ein *bless* des anonymen Hash auf das aktuelle Package *Basepac* würde statt eines *Deripac*-Objekts eines der Klasse *Basepac* erzeugen. Dagegegen wertet die Langform

```
sub new
{ $type = shift;
 $self = {};
 bless $self, $type;
}
```

einfach den standardmäßig mitgelieferten Klassennamen aus und bindet den anonymen Namens-Hash an die richtige Klasse. Dabei ist es gleichgültig, welches Package den Aufruf des Konstruktors ursprünglich initiiert hat. Müssen hingegen sowohl die abgeleitete als auch die Basisklasse eigene Variablen initialisieren, bleibt nichts übrig, als den Konstruktor der Basisklasse explizit aufzurufen (Listing 2).

Im Beispiel legt der *Deripac*-Konstruktor durch den Aufruf seines Pendants in *Basepac* den notwendigen Namens-Hash an und nutzt ihn anschließend für seine eigene Instanzenvariable *derivar* mit.

Statische Konstruktoren und virtuelle Funktionen

Die Frage des Erbens wäre also geklärt – es bleibt das 'Erben lassen'. Während es in C++ Regeln dafür gibt, welche Teile einer Klasse für wen sichtbar sind (Einträge in den Sektionen *public* und *protected*), läßt Perl dem Programmierer freie Hand: Es ist eben keine 'totalitäre' Sprache, wie es in den Manual-Pages so schön heißt. Nur ein wenig gesunder Menschenverstand ist notwendig.

Perl unterscheidet zwischen statischen und virtuellen Funktionen. Konstruktoren sind durchweg statisch: Sie werden durch den Package-Namen spezifiziert, den sie beim Aufruf

```
$objref = Classname->new();
```

implizit als ersten Parameter erhalten. Über die vom Konstruktor zurückgelieferte Objektreferenz ist es anschließend möglich, mittels

```
$objref->method();
```

virtuelle Methoden anzusprechen, die als ersten Parameter eine Objektreferenz erwarten.

Beginnend bei der Klasse, der *objref* angehört, sucht der Interpreter *depth-first* in der Klassenhierarchie die nächsterreichbare Methode mit dem angegebenen Namen. Das OO-typische Überladen von virtuellen Funktionen in einer abgeleiteten Klasse ist jederzeit möglich – eine Methode gleichen Namens muß lediglich dort definiert sein, wo sie der Suchalgorithmus in der Klassenhierarchie zuerst findet. Die in der aktuellen Klasse spezifizierte Methode hat so gegenüber allen anderen den Vorrang.

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

#---------------------------------
package Basepac;
#---------------------------------
### Constructor
sub new
{ my $type = shift;
  my $self = {};

  $self->{'basevar'} = 'BASE';

  bless $self, $type;
}

#---------------------------------
package Deripac;
#---------------------------------
@ISA = qw ( Basepac );

### Constructor
sub new
{ my $type = shift;
  my $self = Basepac->new();

  $self->{'derivar'} = 'DERI';

  bless $self, $type;
}

### Sample Method
sub derimethod
{ my $self = shift;
  print "basevar=",
        $self->{'basevar'}, " ",
        "derivar=",
        $self->{'derivar'}, "\n";
}

#---------------------------------
package main;
#---------------------------------
$derobj = Deripac->new();
$derobj->derimethod();
```

Klassenspezifische Varibablen erfordern den expliziten Aufruf des Basisklassenkonstruktors. Ein Aufruf von *derimethod()* im Hauptprogramm gibt die Werte aller Variablen aus (Listing 2).

Multiple Vererbung verlangt Sorgfalt

Theoretisch spricht nichts dagegen, eine Klasse von mehreren Basisklassen erben zu lassen. Zumindest für die Methoden ist dies kein Problem, denn es genügt, die Namen der beteiligten Packages in das *@ISA*-Array aufzunehmen.

Dies ist aber leider nur die halbe Miete: Falls die Konstruktoren der Basisklassen Daten initialisieren, müssen die Namensräume aller vererbten Klas-

iX-TRACT

- Wie andere objektorientierte Sprachen gestattet auch Perl 5 die Vererbung von Eigenschaften.
- Anders als in C++ kann eine Klasse nicht entscheiden, was sie vererben will, und Konstruktoren der Basisklasse müssen explizit aufgerufen werden.
- Der *ref()*-Operator ermittelt zur Laufzeit den Typ eines Objekts und erlaubt die Implementierung flexibler Algorithmen.

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

#---------------------------------
package Basepac1;
#---------------------------------
### Constructor
sub new
{ my $type = shift;
  my $self = {};

  $self->{'b11'} = "b11";
  $self->{'b12'} = "b12";

  bless $self, $type;
}

#---------------------------------
package Basepac2;
#---------------------------------
### Constructor
sub new
{ my $type = shift;
  my $self = {};

  $self->{'b21'} = "b21";
  $self->{'b22'} = "b22";

  bless $self, $type;
}


#---------------------------------
package Deripac;
#---------------------------------
@ISA = qw ( Basepac1 Basepac2 );

### Constructor
sub new
{ my $type = shift;
  my $self = {};

  # Loop over all base classes
  foreach $pac (@ISA)
  {
    # Call base class constructor
    my $hashref = $pac->new();

    # Merge name spaces of base
    # classes in the class' own hash
    foreach $key (keys %$hashref)
    { $self->{"$key"} =
        $$hashref{"$key"};
    }
  }

  bless $self, $type;
}

### Sample Method
sub derimethod
{ my $self = shift;

  # Print all initialized variables
  foreach $key (keys %{$self})
  { print "\$self->{'$key'}=",
          "$self->{$key}\n";
  }
}


#---------------------------------
package main;
#---------------------------------
$derobj = Deripac->new();
$derobj->derimethod();
```

Der Konstruktor von *Deripac* liest sämtliche Variablen, die über die anonymen Hashes der ererbten Basisklassen *Basepac1* und *Basepac2* definiert sind, und speichert sie im Namensraum von *Deripac* (Listing 3).

sen gemischt werden. Für den Trivialfall, daß die Packages nur eine Reihe von Skalaren als Instanzenvariablen nutzen, läßt sich eine multiple Vererbung nach Listing 3 implementieren. Dieses Verfahren ist allerdings mit Vorsicht zu genießen: Mangelnde Sorgfalt beim Klassenentwurf führt sehr schnell zu Chaos, und Vererbungskonflikte erfordern manuelle Nacharbeit.

Da eine Klasse aber nicht nur skalare Variablen definieren, sondern ihren

```
#!/usr/bin/perl
#---------------------------------
package Persistent;
#---------------------------------
# Your persistent objects should
# inherit from this class.
#---------------------------------

### Constructor
sub new
{ my $type = shift;
  my $self = {};
  bless $self, $type;
}

### Store object in file
### Syntax: $objref->store("filename")
sub store
{ my $self = shift;
  my $name = shift;

  # Open the specified file, set the
  # default output filehandle to it
  # and start printing out all
  # variables and their values.
  open(FILE, ">$name") || return 0;
  my $oldfh = select(FILE);
  ref_deep_print("", $self);
  close(FILE);
  select($oldfh);

  1;
}

### Printout all variables in any
### depth of the structure of an
### object.
sub ref_deep_print
{ my $name = shift;
  my $ref  = shift;

  if(!ref($ref))
  { # Printout name and uuencoded
    # value of a scalar value.
    print "$name\n";
    my $uustring = pack('u', "$ref");
    print "$uustring\n";
  }
  elsif(ref($ref) eq "ARRAY")
  { # Enter recursion for every
    # member of the array
    for($i=0; $i<=$#$ref; $i++)
    { ref_deep_print("$name\[$i]",
          $ref->[$i])
            if defined $ref->[$i];
    }
  }
  else
  { # Hash or Object:
    # Enter recursion for every
    # member of the hash/object
    foreach $i (keys %$ref)
    { ref_deep_print("$name\{'$i'}",
                $ref->{$i});
    }
  }
}

### Load object from file
### Syntax: $objref->load("filename")
sub load
{ my $self   = shift;
  my $name   = shift;

  my($uustring, $string, $varname);

  # Open file and read out variable
  # names and values
  open(FILE, "<$name") || return 0;
  while(<FILE>)
  {
    # Read variable name
    chop($varname  = $_);

    # get uuencoded value
    $uustring = "";
    $uustring .= $line
      while ($line=<FILE>) !~ /^$/;

    # uudecode value
    $string = unpack('u', $uustring);

    # ... and set variable
    eval "\$self->$varname = \$string";
  }
  close(FILE);
  1;
}

1;
```

Implementierung der Klasse *Persistent*: Die *store*-Methode liest die gesamte Datenstruktur eines Objekts und speichert sie dauerhaft (Listing 4).

Namens-Hash auch dazu verwenden kann, Referenzen auf neue Hashes und Arrays oder gar fremde Objekte zu speichern, ist dieses Beispiel nur für einfache Fälle geeignet. Eine Erweiterung zeigt Listing 4. Die dort definierte Klasse *Persistent* kann ihre Methoden *store* und *load* an beliebige Klassen vererben und diesen so die Funktionalität verleihen, ihre Objekte dauerhaft als Dateien zu speichern. Somit können Objektdaten die Ausführung eines Prozesses überdauern.

Vererbung von Persistenzmethoden ist in der Softwareentwicklung mit objektorientierten Datenbanken verbreitet. Transiente Objekte wandern so in die Datenbank und auch wieder heraus. Die gezeigte Anwendung spricht jedoch keine Datenbank an, sie legt die Daten lediglich in Dateien ab.

Persistente Objekte – eine Anwendung

Ein Objekt der Anwendungsklasse *Myclass*, die von der Klasse *Persistent* erbt, kann seine Daten über den Vererbungsmechanismus mit

```
$objref->store('filename');
```

in *filename* speichern beziehungsweise mittels

```
$objref->load('filename');
```

restaurieren.

Listing 5 zeigt das zugehörige Testbeispiel. Die vererbten Routinen

```
#!/usr/bin/perl

use Persistent;

#---------------------------------
package Myclass;
#---------------------------------
# A sample class inheriting the
# persistent features
#---------------------------------
@ISA = qw ( Persistent );

### Constructor
sub new
{ my $type = shift;
  my $self = Persistent->new();
  bless $self, $type;
}

### Initialize myobject
sub myinit
{ my $self = shift;

  ### Build up a complex object
  ### structure: Hashes, arrays,
  ### objects.
  $self->{'the_hash'}   = \%the_hash;
  $self->{'the_array'}  = \@the_array;
  $self->{'the_scalar'} =
    "This is a scalarvalue that is " .
    "very long and contains specials: " .
    " @,\$,\\,\",'.";

  $the_hash{'hash_key'} = 'hash_value';
  $the_array[1]         = 'array_value';

  ### Create a new object that
  ### is a part of my object
  my $objref = Myclass->new();
  $objref->{'myobjvar'} = 'myobjvarval';
  $self->{'the_object'} = $objref;
}


#---------------------------------
package main;
#---------------------------------

### Create an object, initialize it
### and call the save routine.
$myobj = Myclass->new();
$myobj->myinit();
$myobj->store("myobj.sav") ||
  print "Cannot save\n";

### Create a new object and call the
### load routine to initialize it.
$obj2 = Myclass->new();
$obj2->load("myobj.sav") ||
  print "Cannot load\n";

### Printout object attributes
print
  "\$obj2->{'the_object'}->{'myobjvar'} = ",
  "$obj2->{'the_object'}->{'myobjvar'}\n";
print
  "\$obj2->{'the_array'}->[1]           = ",
  "$obj2->{'the_array'}->[1]\n";
print
  "\$obj2->{'the_hash'}->{'hash_key'}   = ",
  "$obj2->{'the_hash'}->{'hash_key'}\n";
print
  "\$obj2->{'the_scalar'}               = ",
  "$obj2->{'the_scalar'}\n";
```

Dieses Skript initialisiert das komplexe Objekt *myobj* und speichert es in einer Datei. Anschließend belegt es ein neu erzeugtes Objekt *obj2* über die *load()*-Methode mit den eben gespeicherten Daten (Listing 5).

store() und *load()* bedienen sich dabei des Objekt-Namens-Hash und speichern beziehungsweise laden die Daten. Hierbei müssen sie bis in beliebige Verschachtelungstiefen der Datenstruktur vordringen, falls die Instanzenvariablen des Objektes ihrerseits Referenzen auf weitere Hashes, Arrays oder andere Objekte sind.

In diesem Zusammenhang verwendet das Programm Perls *ref()*-Operator, der den Typ einer Referenz liefert. Folgende Rückgabewerte sind möglich:

– Skalar	undef
– Skalarreferenz	'SCALAR'
– Hashreferenz	'HASH'
– Arrayreferenz	'ARRAY'
– Objekt	Name der Klasse

Entdeckt die Prozedur *ref_deep_print()* im Namens-Hash des Objekts eine Referenz auf eine weitere Datenstruktur, verfolgt sie diese solange, bis sie nur noch Skalare sichtet. Diese schreibt sie dann inklusive der Zugriffspfade innerhalb der Struktur und den zugehörigen Werten in die angegebene Datei. Um Sonderzeichen, mehrzeilige Strings und Nullbytes problemlos ablegen zu können, speichert *ref_deep_print()* die Werte UU-encoded, was sehr kompakt über die Perl-eigene *pack()*-Funktion implementiert ist.

Im Bedarfsfall liest *load()* die Daten wieder ein und setzt die Variablen dementsprechend – was dank der dynamischen Strukturerweiterung von Perl sehr einfach ist. Eine Zuweisung wie etwa

```
$hashref->{'key'}[17][42]{'key2'} = 17;
```

ist möglich, ohne vorher die notwendigen Hashes und Arrays explizit anzulegen – Perl erledigt das automatisch.

Sicher sind einige Aspekte der objektorientierten Programmierung mit Perl stark gewöhnungsbedürftig. Die Schreibweise der Instanzenvariablen oder nötige Handarbeit bei Konstruktoren, Destruktoren und Vererbung sind Beispiele dafür. Trotzdem ist es erstaunlich, wie kompakt sich mit den Neuerungen recht professionelle Lösungen entwickeln lassen. Zudem werden Freiheitsgrade geboten wie sonst in kaum einer anderen Sprache – denn auch in Version 5 gilt das Perl-Motto 'There's more than one way to do it.' (ck)

MICHAEL SCHILLI

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

Literatur

[1] Michael Schilli; Gesegnet; Objektorientierte Programmierung mit Perl 5 *iX* 3/96, S.162 ff.