LINUX MAGAZIN 12/98

Das einzige deutschsprachige Linux-Magazin

LINUX MAGAZIN

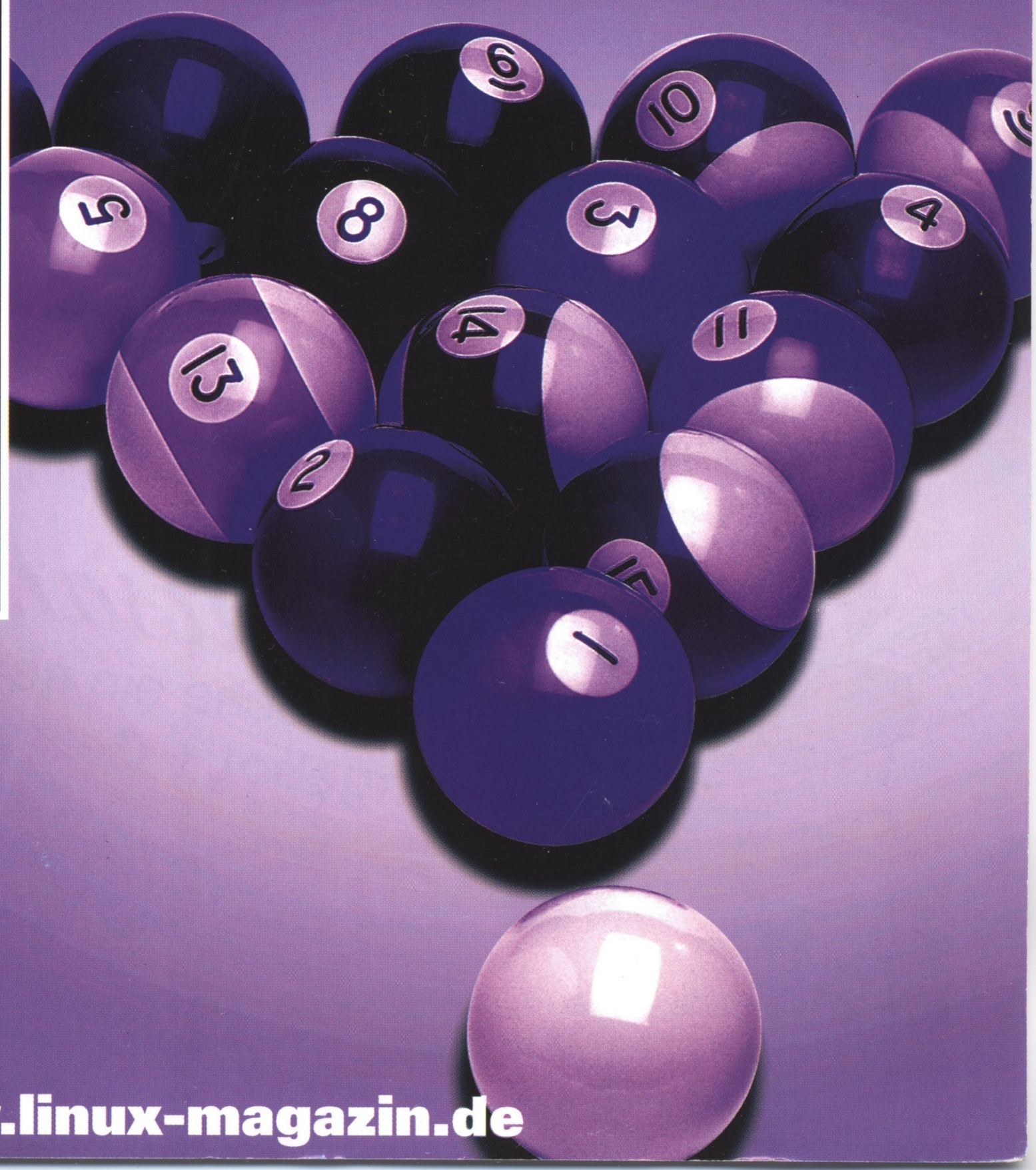
Serie für Einsteiger!

ISSN 1432 - 640 X DM 9,80 ÖS 75,- SFR 9,80

12/98

Zahlenspiele

Statistik-Programme unter Linux



Linux News

Neues aus dem Internet

Workshop

Informix SE

Einsteiger

PGP und GNUPG

Kernel

Tasks und Scheduling

4 392587 309802 12

www.linux-magazin.de

Perl-Snapshot

Testen für Faule

von Michael Schilli

Kein „echter" Programmierer testet gern. Andererseits funktioniert kaum etwas, ohne daß man es testet. Schlimmer noch, ändert sich eine Kleinigkeit, brechen oft ganze Systeme wie Kartenhäuser zusammen. Der Ausweg, der Programmierer-Faulheit genial mit Programm-Zuverlässigkeit vereint, lautet: Regressionstests.

Wir wissen nicht, was der freundliche Informatik-Professor rät, doch Regression heißt für mich: Ich schreibe eine Menge Perl-Skripts (oder auch C-Programme oder Shell-Skripts), teste sie zum jeweiligen Entstehungszeitpunkt, fertige daraus einen automatischen Test, und hänge diesen in eine Test-Suite ein, wo er für alle Zeiten bleibt. Um zu testen, ob die Gesamtheit aller Skripts noch funktioniert, stoße ich die Test-Suite an, die geduldig alle bisher eingehängten Tests ablaufen läßt und anzeigt, ob auch nur ein einziger Testfall nicht funktioniert. Dieses Verfahren garantiert, daß nach Hinzufügen einer Kleinigkeit oder der Installation eines neuen Moduls, das viele Skripts verwenden, nicht das Gesamtsystem zusammenkracht - denn schließlich kann man alle naselang die Test-Suite anstoßen, kost' ja nix.

Perls Standard-Test

Perl liegt standardmäßig das Modul *Test::Harness* bei, das den Ablauf der Regressionstests der aktuellen Perl-Installation steuert und - die Manualseite ist sich nicht ganz sicher - entweder von unserem sympathischen österreicherischen Wahl-Berliner *Andreas König* oder aber *Tim Bunce* stammt. Ein Kontrollskript *tests.pl* wie

```
#!/usr/bin/perl -w
use Test::Harness;
runtests(„one.t", „two.t");
```

erwartet von den aufgerufenen Testskripts *one.t* und *two.t*, daß sie jeweils im Erfolgsfall eine Ausgabe wie

```
1..3
ok 1
ok 2
ok 3
```

liefern. Geht alles glatt, liefert *tests.pl* entsprechend

```
one.................ok
two.................ok
All tests successful.
Files=2,  Tests=5,  0 secs ( 0.09 cusr  0.06
csys =  0.15 cpu)
```

Geht aber zum Beispiel der zweite Test von *one.t* schief, sollte es

```
1..3
ok 1
not ok 2
ok 3
```

liefern. *tests.pl* gibt entsprechend Folgendes aus:

```
one.................FAILED test 2
    Failed 1/3 tests, 66.67% okay
two.................ok
Failed Test  Status Wstat Total Fail  Failed
List of failed
-------------------------------------------
one.t                           3    1
33.33%  2
Failed 1/2 test scripts, 50.00% okay. 1/5
subtests failed, 80.00% okay.
```

Die Test-Suite

Statt nun jedem Skript beizubringen, im Erfolgsfall *„ok“* und im Fehlerfall *„not ok“* auszuspucken, dachte ich mir: Jedes Skript liefert eine bestimmte Ausgabe, die ich einmal kontrollieren und dann in eine Referenz-Datei verpacken werde. Ist diese Ausgabe einmal abgesegnet, müssen nachfolgende Aufrufe nur noch die gegenwärtige Ausgabe des Skripts mit dem Inhalt der Referenz-Datei vergleichen und Alarm schlagen, falls die zwei Datensätze nicht miteinander übereinstimmen.

Ein Objekt vom zu entwickelnden Typ *Test::Suite* kann dabei unter beliebigen Testnamen (z.B. *Erster Test*) jeweils ein oder mehrere Skripts (z.B. *test.pl*) registrieren. Die *run*-Methode durchläuft alle Skripts, merkt sich deren Ausgaben und vergleicht, ob sie identisch sind mit den gesicherten Referenzdaten, die in Dateien mit der Endung **.ref* im jeweiligen Testverzeichnis liegen (z.B. *test.pl.ref*). Um erst einmal alle Referenzdateien anzulegen, zeichnet die *update*-Methode veranwortlich. *update* soll, wie *run*, alle Skripts ausführen und eventuell schon existierende Referenzfiles nur dann ersetzen (und eine Meldung ausgeben), falls sich etwas geändert hat, wobei es die bisherige Referenzdatei (z.B. nach *test.pl.ref.bak*) sichert, bevor sie sie mit der aktualisierten Ausgabe überschreibt.

Eine *cleanup*-Methode löscht alle Referenz-Dateien und deren Backups. Für jedes Skript wechselt das *Test::Suite*-Objekt dabei in das Verzeichnis, in dem sich das Test-Skript befindet.

Manche Skripts vertragen mehrere Kommandozeilenparameter und müssen mehrfach, mit verschiedensten Parameter-Kombinationen, in die Test-Suite aufgenommen werden. Ein Shell-Skript im Test-Verzeichnis reiht üblicherweise einfach einige Aufrufe hintereinander, die Ausgabe kumuliert die Skript-Ausgaben.

Beispiel-Tests

In den Verzeichnissen *TEST1*, *TEST2* und *TEST3* liegen folgende Skipts:

```
TEST1:
 one.pl
 two.pl

TEST2:
 mult.sh
 one.pl

TEST3:
 one.pl
 two.pl
 three.pl
```

Im Verzeichnis *TEST1* liegen die Skripts *one.pl* und *two.pl*, die zu Test-Zwecken einfach nach dem Muster

```
#!/usr/bin/perl -w
print „This is test one\n“;
```

aufgebaut sind. Listing *suite.pl* zeigt, wie man Tests aufsetzt: Zeile 12 erzeugt ein neues *Test::Suite*-Objekt, die Zeilen 14 und 15 fügen unter dem Testnamen *Erster Test* die zwei Skripts *one.pl* und *two.pl* aus dem Verzeichnis *TEST1* in die Suite ein. Die *register*-Methode nimmt einen Testnamen und eine Liste von Test-Skripts entgegen. Um das Skript *one.pl* im nächsten Verzeichnis, *TEST2*, zu testen, soll ein Shell-Skript *mult.sh* aushelfen, das folgende Aufrufe absetzt:

```
one.pl -v || exit 1
one.pl -f || exit 1
```

Es testet verschiedene Parameter-Kombinationen von *one.pl*. Falls ein Aufruf das Perl-Skript *one.pl* zum Abbruch zwingt, liefert dieses einen Exit-Code ungleich *0* zurück, was, wegen der || *exit 1*-Konstrukte wiederum das Shell-Skript *mult.sh* zum Abbruch mit einem Exit-Code von *1* veranlaßt (die Shell bewertet im Gegensatz zu Perl einen Exit-Code von *0* als Erfolgsmeldung, andernfalls liegt ein Fehler vor). Ohne die Exit-Konstrukte liefe *mult.sh* bei einem auftretenden Fehler einfach weiter und *suite.pl* bekäme davon nichts mit.

Statt *one.pl* registriert *suite.pl* in Zeile 16 entsprechend *mult.sh*. Die Ausgaben beider Aufrufe von *one.pl* landen, hintereinandergehängt, über die *update*-Methode von *Test::Suite* in *mult.sh.ref*, von wo sie zu einem späteren Zeitpunkt wieder ausgelesen und mit der Ausgabe von *mult.sh* verglichen werden. Da ein *Test::Suite*-Objekt vor dem Ausführen eines registrierten Tests in dessen Verzeichnis wechselt, darf *mult.sh* die Tests auch ruhig über ihren Skriptnamen ohne Verzeichnisangabe referenzieren.

Das dritte Testverzeichnis *TEST3* enthält drei Skripts *one.pl*, *two.pl* und *three.pl*. Um sie auf einen Schlag an die Test-Suite anzuhängen, bedient sich *suite.pl* des Globbing-Konstrukts *<TEST3/*.pl>*, das alle **.pl*-Dateien aus *TEST3* als Liste zurückliefert. Startet nun *suite.pl*, sind natürlich in den Test-Verzeichnissen

LISTING 1: *suite.pl*

```
 1 #!/usr/bin/perl -w
 2 ################################################
 3 # suite.pl [-create] [-clean]
 4 # -create: Create reference files
 5 # -clean:  Cleanup ref files and their backups
 6 #
 7 # 1998, schilli@perlmeister.com
 8 ################################################
 9
10 use Test::Suite;
11
12 $suite = Test::Suite->new();
13
14 $suite->register(„Erster Test“,   „TEST1/one.pl“,
15                                    „TEST1/two.pl“);
16 $suite->register(„Zweiter Test“, „TEST2/mult.sh“);
17 $suite->register(„Dritter Test“, <TEST3/*.pl>);
18
19 if(grep { $_ eq „-create“} @ARGV) {
20     $suite->update();
21 } elsif(grep {$_ eq „-clean“} @ARGV) {
22     $suite->clean();
23 } else {
24     $suite->run();
25 }
```

noch keinerlei Referenz-Dateien vorhanden, und das Skript bricht mit

```
No reference file for TEST1/one.pl at
Test/Suite.pm line 129.
```

ab, denn schon zum ersten Test-Skript *TEST1/one.pl* fehlt die Referenz-Ausgabe. Abhilfe schafft hierbei der Aufruf

```
suite.pl -create
```

der *suite.pl* dazu veranlaßt, statt der *run*- die *update*-Methode aufzurufen und so zu jedem registrierten Testskript eine Referenzdatei anzulegen. Die Ausgabe lautet:

```
Created REF TEST1/one.pl.ref
Created REF TEST1/two.pl.ref
Created REF TEST2/mult.sh.ref
Created REF TEST3/one.pl.ref
Created REF TEST3/three.pl.ref
Created REF TEST3/two.pl.ref
```

Ein nachfolgender Aufruf von *suite.pl* bringt

```
Erster Test (2) ..................... ok
Zweiter Test (1) .................... ok
Dritter Test (3) .................... ok
```

zum Vorschein - alles in Butter, die 2 Tests aus *TEST1*, das Shell-Skript aus *TEST2* (zählt als ein Test, ruft aber zwei Unter-Skripts auf) und die drei Perl-Skripts aus *TEST3* liefern genau die Ausgaben, die in den Referenzdateien festgehalten wurden. Ändert sich nun beispielsweise die Ausgabe des Skripts *TEST3/three.pl* gegenüber der Referenzdatei *TEST3/three.pl.ref*, meldet *suite.pl*:

```
Erster Test (2) ..................... ok
Zweiter Test (1) .................... ok
Dritter Test: TEST3/three.pl REF mismatch
Dritter Test (3) .................... 2
ok, 1 not ok
```

Um die Test-Suite wieder auf den neuesten Stand zu bringen, hilft wieder der *-create*-Schalter, der die *update*-Methode auslöst:

```
suite.pl -create
```

Da *suite.pl* in diesem Modus nur Referenzdateien von Skripts verändert, die es nötig haben, lautet die Ausgabe

```
Updated REF TEST3/three.pl.ref
```

Hierbei sichert das *Test::Suite*-Objekt die Datei *TEST3/three.pl.ref* in *TEST3/three.pl.ref.bak*, bevor es erstere überschreibt. Um die Referenzdateien **.ref* samt ihren Backups **.ref.bak* aller registrierten Skripts zu löschen, verarbeitet *suite.pl* den Schalter *-clean*, der die *clean*-Methode in *Test::Suite* auslöst (Vorsicht!):

```
suite.pl -clean
```

bringt so

```
Cleaning up TEST1/one.pl.ref
Cleaning up TEST1/two.pl.ref
Cleaning up TEST2/mult.sh.ref
Cleaning up TEST3/one.pl.ref
Cleaning up TEST3/three.pl.ref
Cleaning up TEST3/three.pl.ref.bak
Cleaning up TEST3/two.pl.ref
```

zum Vorschein und stellt den ursprünglichen Zustand wieder her. Dies sollte freilich nur in den seltensten Fällen vonnöten sein - schließlich basieren Regressions-Tests auf den gespeicherten Referenzdaten.

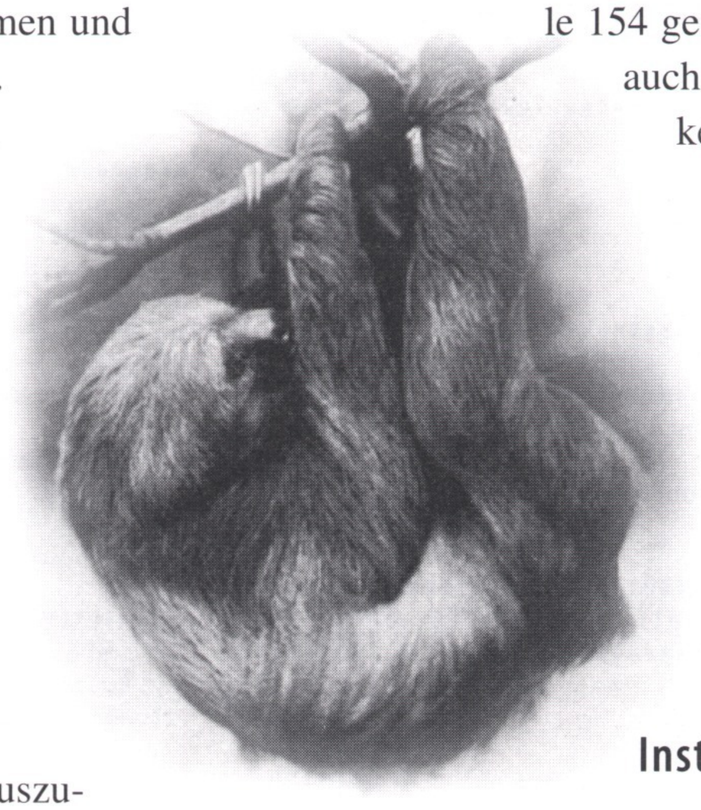
Test::Suite

Die ganze Funktionalität des erzeugten *Test::Suite*-Objekts, das die Tests registriert, Referenzdateien aktualisiert und deren Inhalt mit aktuellen Testskriptausgaben vergleicht, steckt im Modul *Test::Suite*, das in Listing *Suite.pm* zu sehen ist. Der Plain-Vanilla-

LISTING 2: *Suite.pm*

```
 1 ######################################################
 2 # Test::Suite
 3 #
 4 # 1998, schilli@perlmeister.com
 5 ######################################################
 6
 7 use File::Copy;
 8 use File::Basename;
 9 use Cwd;
10
11 package Test::Suite;
12
13 ######################################################
14 sub new {                                  #
Constructor
15 ######################################################
16     my $class = shift;
17
18     my $self  = {};
19     bless($self, $class);
20 }
21
22 ######################################################
23 sub register {                    # Register scripts
24 ######################################################
25     my ($self, $testname, @scripts) = @_;
26
27     push(@{$self->{tests}}, [$testname, @scripts]);
28 }
29
30 ######################################################
31 sub update {           # Update all reference files
32 ######################################################
33     my ($self) = @_;
34
35     foreach $test (@{$self->{tests}}) {
36         my ($testname, @scripts) = @$test;
37
38         foreach $script (@scripts) {
39             test_script($script, 1);
40         }
41     }
42 }
43
44 ######################################################
45 sub clean {          # Clean up all reference files
46 ######################################################
47     my ($self) = @_;
48     my $file;
49
50     foreach $test (@{$self->{tests}}) {
51         my ($testname, @scripts) = @$test;
52
53         foreach $script (@scripts) {
54             foreach $file („$script.ref",
55                            „$script.ref.bak") {
56                 if(-e $file) {
57                     print „Cleaning up $file\n";
58                     unlink($file) ||
59                         die „Cannot unlink
$file";
60                 }
61             }
62         }
63     }
64 }
65
66 ######################################################
67 sub run {
68 ######################################################
69     my $self = shift;
70
71     foreach $test (@{$self->{tests}}) {
72         my ($testname, @scripts) = @$test;
73         my $total = @scripts;
74         my ($failed, $success) = (0,0);
75
76         foreach $script (@scripts) {
77             if(test_script($script)) {
78                 $success++;
79             } else {
80                 print „$testname: „ .
81                       „$script REF mismatch\n";
82                 $failed++;
83             }
84         }
85
86         my $dots = „";
87         if(length(„$testname$total") < 40) {
88             $dots = „." x
89               (40 - 2 - length(„$testname$total"));
90         }
91
92         if($total eq $success) {
93             print „$testname ($total) $dots ok\n";
94         } else {
95             print „$testname ($total) $dots „ .
96                   „$success ok, $failed not ok\n";
97         }
98     }
99 }
100
101 ######################################################
102 sub test_script {
103 ######################################################
104     my ($script, $create_ref) = @_;
105     my $shouldbe;
106     my $retval = 0;
107
108     my $cwd = Cwd::cwd();       # Get current dir
109
110         # Change to dir where script resides in
111     my($base, $path) =
112               File::Basename::fileparse($script);
113     chdir($path) || die „Chdir $path failed";
114
115     open(PIPE, „$base |") ||
116         die „Cannot open $script";
117     my $output = join('', <PIPE>);
118     close(PIPE) || die „Script $script fails";
119
120     if(! -f „$base.ref") {
121         # REF file doesn't exist
122         if($create_ref) {
123             open(FILE, „>$base.ref") ||
124                 die „Cannot create $script.ref";
125             print FILE $output;
126             close(FILE);
127             print „Created REF $script.ref\n";
128         } else {
129             die „No reference file for $script";
130         }
131     } else {
132         # REF file does exist, read it
133         open(FILE, „<$base.ref") ||
134             die „Cannot open $script.ref";
135         $shouldbe = join('', <FILE>);
136         close(FILE);
137
138         if($create_ref) {
139             if($output ne $shouldbe) {
140
141                 # Create backup
142                 File::Copy::copy(„$base.ref",
143                                  „$base.ref.bak")
||
144                     die „Backup $base.ref fai-
led";
145
146                 # Write new reference file
147                 open(FILE, „>$base.ref") ||
148                     die „Cannot write
$script.ref";
149                 print FILE $shouldbe;
150                 close(FILE);
151                 print „Updated REF
$script.ref\n";
152             }
153         } else {
154             $retval = ($output eq $shouldbe);
155         }
156     }
157
158     chdir($cwd) || die „Cannot chdir to $cwd";
159
160     return($retval);
161 }
162
163 1;
```

Konstruktor ab Zeile 14 erzeugt den in der objektorientierten Programmierung mit Perl üblichen Namens-Hash für Instanzvariablen von Objekten und verabschiedet sich danach sofort wieder.
Die *register*-Methode nimmt außer der standardmäßig hereingereichten Objektreferenz einen Testnamen und eine Reihe von Skriptnamen entgegen und hängt sie an die Instanzvariable *tests* an, eine Referenz auf eine Liste, die für jeden Test eine Referenz auf eine Liste enthält, die wiederum den Testnamen und eine Reihe von Skriptnamen führt.
Die *update*-Methode ab Zeile 31 durchläuft alle registrierten Tests und ruft für jedes Skript einer jeden Testreihe die weiter unten besprochene Funktion *test_script* mit dem Skriptnamen und einer *1* (für Update!) auf. *clean* ab Zeile 45 durchläuft ebenfalls alle registrierten Testnamen, und löscht, falls vorhanden, die Referenzdateien und deren Backups.
run ab Zeile 67 ruft für jeden auszuführenden Test die Funktion *test_script* mit dem Scriptnamen als Parameter auf, und läßt, im Gegensatz zu *update*, bewußt den zweiten Parameter aus, um *test_script* im Falle einer fehlenden Referenzdatei zu einer Fehlermeldung zu veranlassen. Falls der Inhalt des Referenzfiles nicht mit der Ausgabe des Skripts übereinstimmt, liefert *test_script* im Gegensatz zum sonst zurückgegebenen *1* den Wert *0* , also einen falschen Wert, zurück.
run zählt die erfolgreichen und die gescheiterten Tests und gibt diese Zahlen schön formatiert aus, indem es in *$dots* eine Anzahl von Punkten erzeugt, die genau zwischen die angezeigten Testnamen und deren Statusanzeige paßt.
test_script ab Zeile 102 speichert in Zeile 108 zunächst das aktuelle Verzeichnis, das es mittels der Funktion *cwd* aus dem Standard-Modul *Cwd* ermittelt. Daraufhin extrahiert es das Verzeichnis aus dem Skript-Pfad mittels der Funktion *fileparse* aus dem *File::Basename*-Modul (Standard-Distribution) - und springt dorthin.
Anschließend ruft es das angegebene Test-Skript auf und speichert dessen Ausgabe in *$output*. Existiert noch keine Referenz-Datei, erzeugen die Zeilen 123 bis 126 eine - aber nur, falls *test_scripts* zweiter Parameter (*$create_ref*) gesetzt ist, andernfalls liegt ein Fehler vor und Zeile 129 bricht alle Tests ab.
Existiert eine Referenzdatei und deren Inhalt stimmt mit der aktuellen Skriptausgabe überein, gibt's nichts zu meckern. Unterscheiden sich beide, wird bei gesetztem *$create_ref*-Parameter eine neue Referenzdatei erzeugt, aber nicht bevor die alte mit *File::Copy::copy* (auch aus der Standarddistribution) gesichert wurde. Ist *$create_ref* nicht gesetzt, ist also die *run*-Methode aktiv, muß *test_script*, falls die Skriptausgabe nicht dem Inhalt der Referenzdatei entspricht, einen Fehler melden, was es über den in Zeile 154 gesetzten Rückgabewert schließlich auch tut. Bevor *test_script* zurückkehrt, setzt es noch schnell das aktuelle Verzeichnis auf den Wert zurück, der eingestellt war, als *test_script* betreten wurde. Falls sich ein Skript nicht starten läßt, oder mit einem Exit-Status ungleich *0* zurückkehrt, meldet *test_script* ebenfalls einen Fehler.



Installation

Das Modul *Test::Suite* wird installiert, indem die Datei *Suite.pm* im Verzeichnis *Test* landet, welches *perl* finden muß. */usr/lib/perl5/Test* bietet sich an, oder einfach ein neu angelegtes Subdirectory *Test* in einem Verzeichnis, aus dem ein Suite-Steuerungs-Skript a la *suite.pl* startet. *suite.pl* muß alle zu testenden Skripts registrieren, bevor es sie ausführt, Globbing-Konstrukte wie das im Text angegebene erleichtern die Arbeit, wenn viele Dateien mit bestimmten Endungen (z.B. **.pl*) in Unterverzeichnissen vorliegen.
Wer seine Test-Suite kontinuierlich pflegt und, während Programme nach und nach entstehen, minimalen Aufwand spendiert, um kleine Testskripts einzuhängen, darf sich, wenn es heißt: Systemtest! getrost zurücklehnen, die Suite aufrufen und die vorbereitete gekühlte Flasche öffnen. *Mm-mm-mmhh!* Fröhliches Testen!

DER AUTOR

Michael Schilli arbeitet als Web-Engineer für America Online Inc., San Mateo. Er ist Autor des 1998 bei Addison-Wesley erschienenen Buches *„GoTo Perl 5“* und unter *michael@perlmeister.com* oder *http://perlmeister.com* zu erreichen.