
Testen mit Perl

Qualitätssicherung für Module

Tests schreiben...



Mein Programm ...

```
#!/usr/bin/perl

use strict;
use warnings;

my $tmp;

print 'Zahl eingeben: ';
my $number = <STDIN>;

for my $f(1..$number-1){
    next unless (($number/$f) == int($number/$f));
    $tmp += $f;
}

print "Perfekte Zahl ? ", $tmp == $number ? "Ja" : "Nein";
```

... macht was es soll

■ Programm für perfekte Zahlen

- Wenn alle Faktoren bis zur Zahl addiert die Zahl ist
- 6 ist perfekt: $1 + 2 + 3 = 6$
- 8 ist nicht perfekt: $1 + 2 + 4 = 7$

```
C:\community>test_example.pl
```

```
Zahl eingeben: 6
```

```
Perfekte Zahl ? Ja
```

```
C:\community>test_example.pl
```

```
Zahl eingeben: 8
```

```
Perfekte Zahl ? Nein
```

Eine kleine Änderung ...

```
#!/usr/bin/perl
```

```
use strict;
```

```
use warnings;
```

```
my $tmp;
```

```
while(1){
```

```
    print 'Zahl eingeben: ';
```

```
    my $number = <STDIN>;
```

```
    for my $f(1..$number-1){
```

```
        next unless (($number/$f) == int($number/$f));
```

```
        $tmp += $f;
```

```
    }
```

```
    print "Perfekte Zahl ? ", $tmp == $number ? "Ja" : "Nein";
```

```
    print "\n";
```

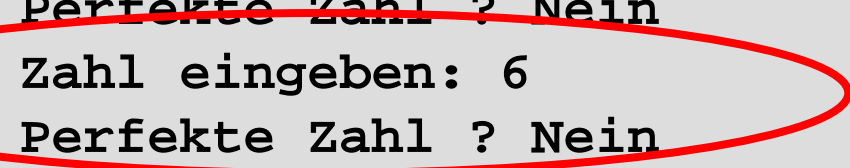
```
}
```

... und es funktioniert noch ...

```
C:\community>test_example.pl  
Zahl eingeben: 6  
Perfekte Zahl ? Ja  
Zahl eingeben: 8  
Perfekte Zahl ? Nein  
Zahl eingeben:
```

... oder auch nicht!

```
C:\community>test_example.pl  
Zahl eingeben: 6  
Perfekte Zahl ? Ja  
Zahl eingeben: 8  
Perfekte Zahl ? Nein  
Zahl eingeben: 6  
Perfekte Zahl ? Nein  
Zahl eingeben:
```





Quelle: www.radgraphics.net



Tests sind wichtig weil...

- Menschen machen Fehler – und wir sind Menschen
- Bugs kosten Zeit & Geld
- Bugs nerven



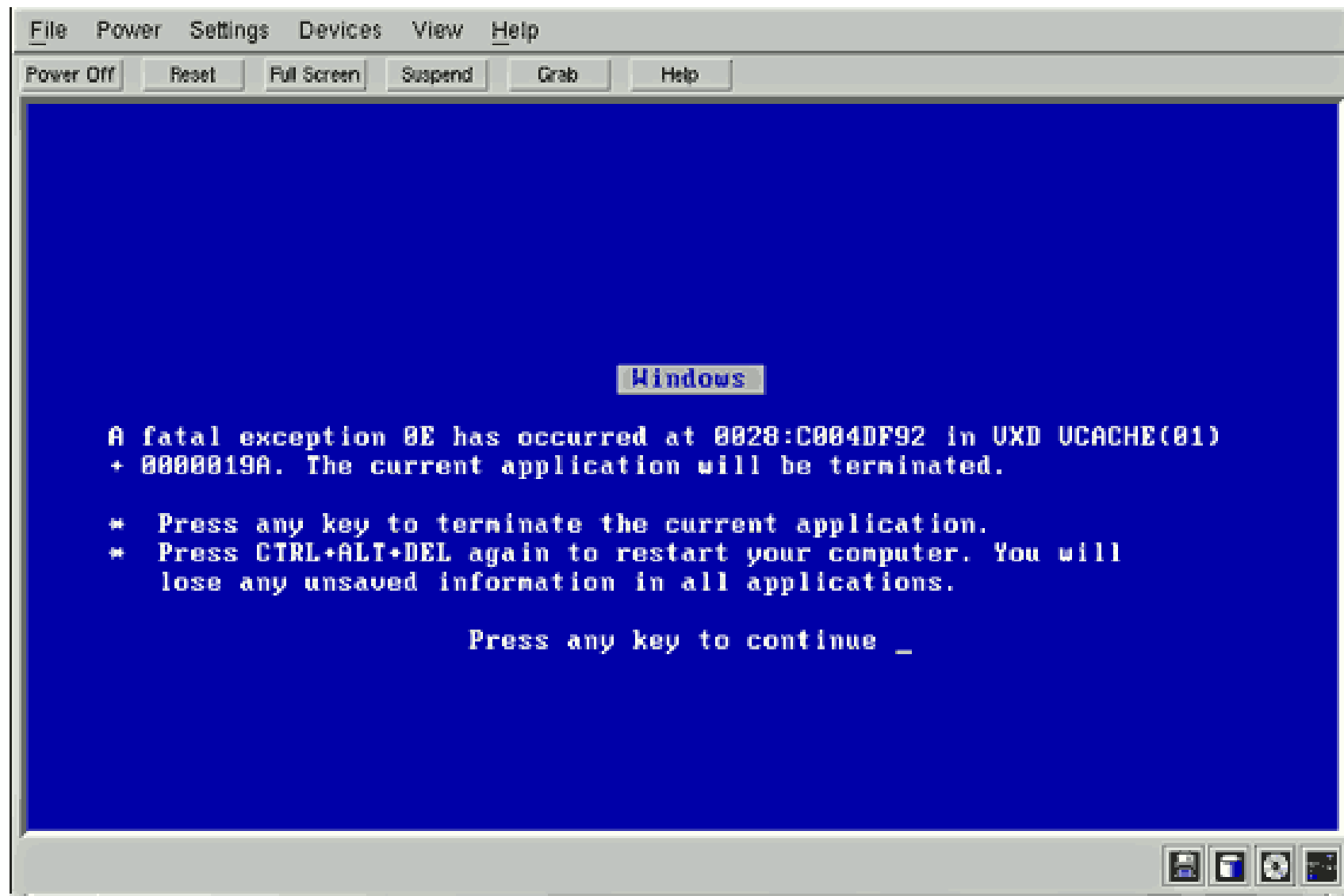
Tests sind wichtig weil...

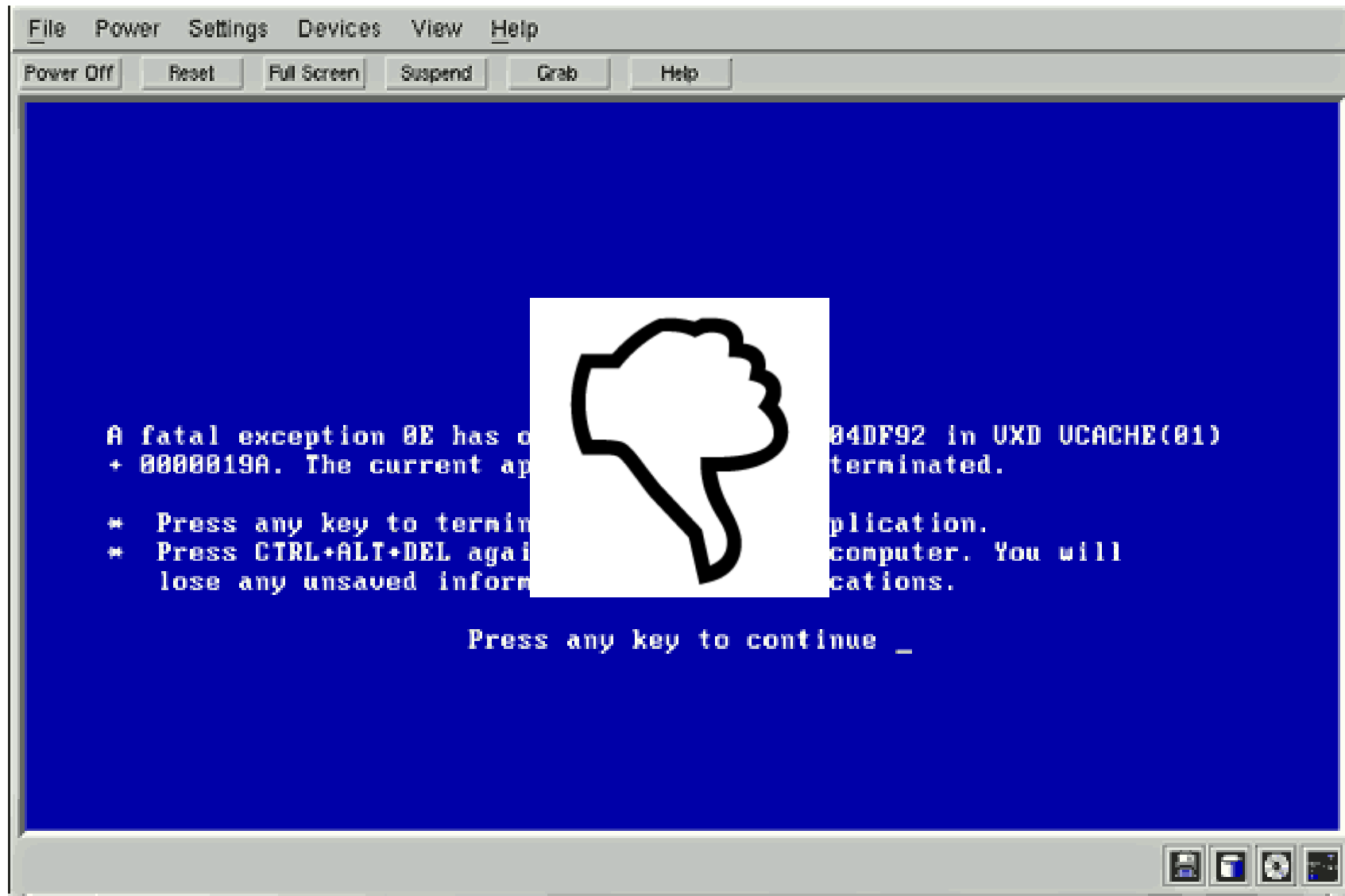
- Erhöhen Qualität des Programms
- Senken die Fehlerwahrscheinlichkeit
- Sichern die Funktionalität des Programms
- ... es das Ranking bei CPANTS erhöht



Tests machen glücklich ;-)







Tests sind einfach weil...

- ... CPAN so riesig ist
- ... es für alles ein Test-Modul gibt
 - Test::More
 - Test::Simple
 - Test::Exception
 - Test::Pod
 - Test::Pod::Coverage
 - ...

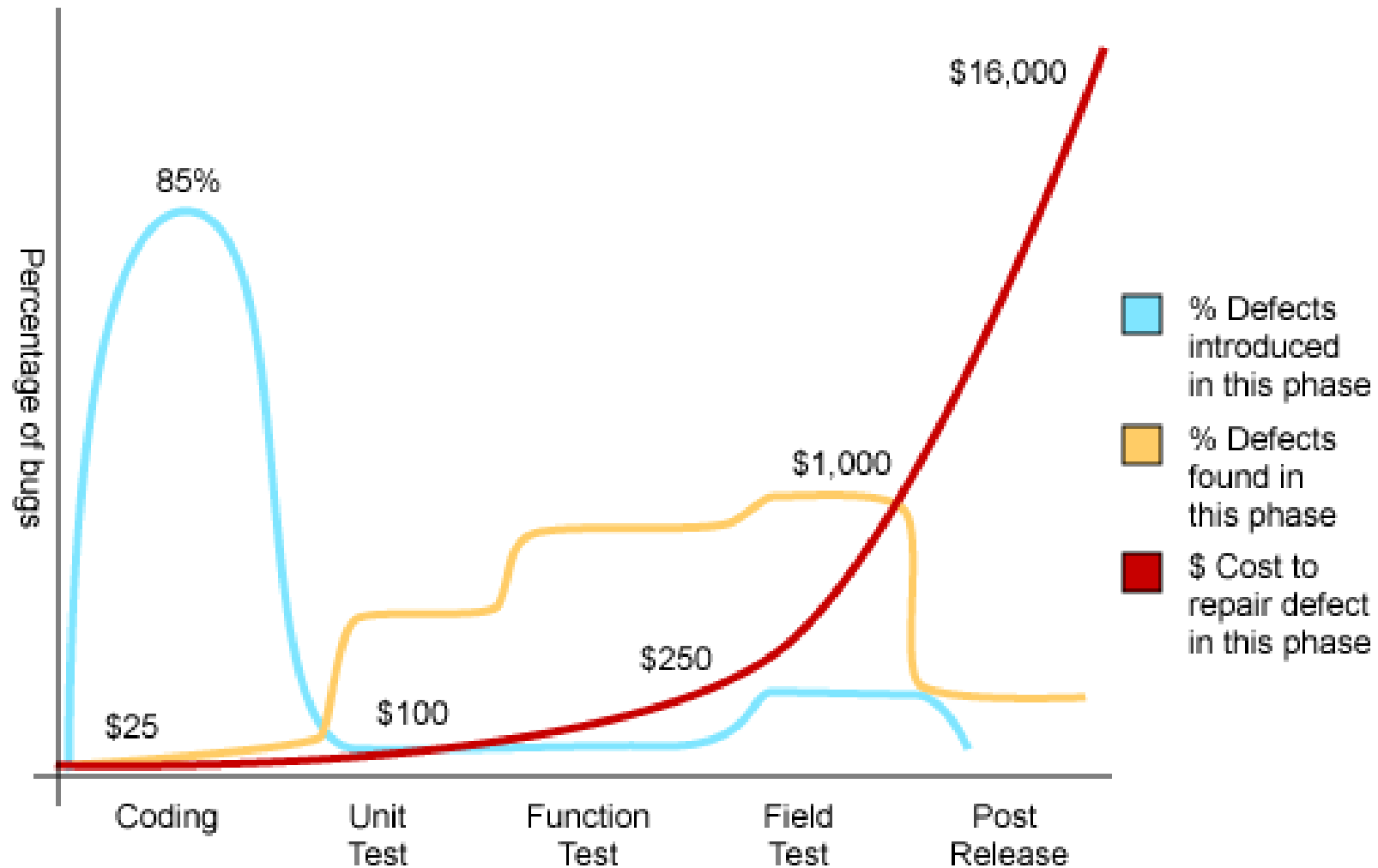


Tests sind einfach weil...

- Tests können automatisiert werden
- Tests den Code verifizieren



Tests sparen Geld weil...



Source: *Applied Software Measurement*, Capers Jones, 1996

Philosophie der Testautomatisierung

- Es müssen nicht immer 1.000 Tests sein
 - Ein Test ist besser als gar keiner
 - Fang' mit einem Test an
 - Tests können nach und nach erweitert werden
 - Erweitern der Tests
 - Wenn Features hinzugefügt werden
 - Wenn ein Bug entdeckt wird
 - Danach muss der Code neuen Test erfüllen
 - Alter Code darf neuen Test nicht erfüllen
-

Test::Harness

- Startet Testskripte
 - Bereitet Ausgabe der Testskripte auf
 - Ausgabe im Test Anything Protocol (TAP)
 - Siehe Test::Harness::TAP
 - Ist sehr einfach
 - Nur zwei Methoden
-

Test::Harness - runtests

- Startet alle Testdateien einer Liste
- Aufruf `runtests(@testfiles)`
 - `runtests('test.t')`



Test::Harness - runtests

```
#!/usr/bin/perl

use Test::Harness;

my $file = 'example.t';

runtests($file);
```

```
#!/usr/bin/perl

use Test::More tests => 1;

ok(1 + 1 == 2);
```

```
C:\community>test_harness.pl
example....ok
All tests successful.
Files=1, Tests=1, 0 wallclock secs ( 0.00 cusr + 0.00 csys = 0.00 CPU)
C:\community>_
```

Test::Harness - runtests

```
#!/usr/bin/perl

use Test::Harness;

my $file = 'example.t';

runtests($file);
```

```
#!/usr/bin/perl

use Test::More tests => 1;

ok(1 + 1 == 3);
```

```
C:\community>test_harness.pl
example....
example....NOK 1# Failed test in example.t at line 5.
# Looks like you failed 1 test of 1.
example....dubious
Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
Failed 1/1 tests, 0.00% okay
Failed Test Stat Wstat Total Fail Failed List of Failed
-----
example.t      1    256      1    1 100.00% 1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
C:\community>_
```

Test::Harness – execute_tests

```
#!/usr/bin/perl

use Test::Harness qw(execute_tests);

my $file = 'example.t';

# ab 2.57 (Dezember 2004)
my ($ok, $failed) = execute_tests(tests => [$file]);
```

Test::Simple

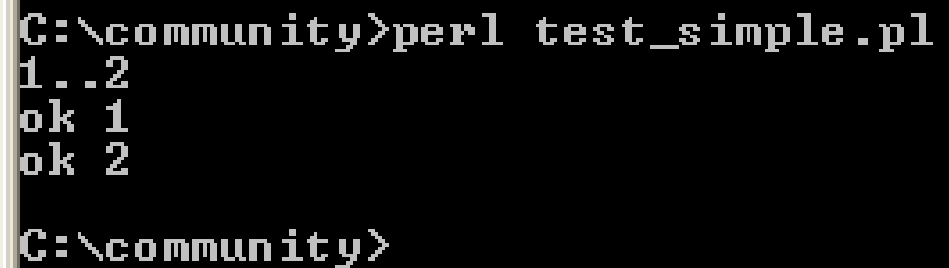
- Einfaches Modul für Tests
 - **ok()** als einzige Methode
 - **tests** als eigener Test, enthält Anzahl der Tests
-

Test::Simple - ok

```
#!/usr/bin/perl

use Test::Simple tests => 2;

ok('example.t' eq 'example.t');
ok((1 + 1) == 2);
```



```
C:\community>perl test_simple.pl
1..2
ok 1
ok 2

C:\community>
```

Test::Simple - ok

```
#!/usr/bin/perl
```

```
use Test::Simple tests => 2;
```

```
ok('example.t' eq 'example.t');
```

```
ok((1 + 2) == 2);
```

```
C:\community>perl test_simple.pl
1..2
ok 1
not ok 2
# Failed test in test_simple.pl at line 6.
# Looks like you failed 1 test of 2.
C:\community>
```

Test::Simple - tests

- Ist ein Plan
 - Anzahl der geplanten Tests
- Selbst ein Test



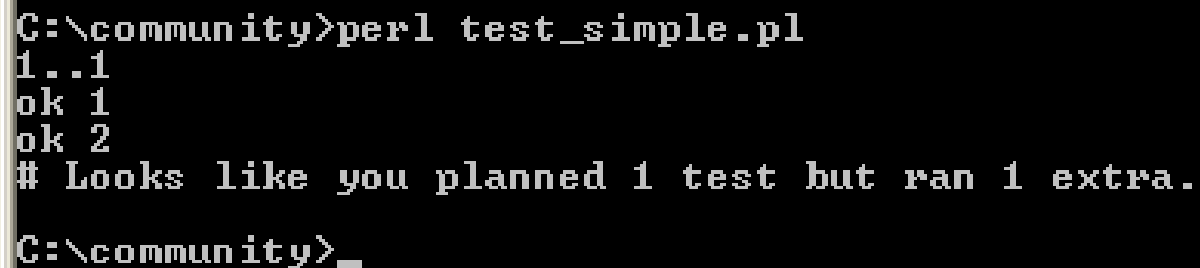
Test::Simple - tests

```
#!/usr/bin/perl
```

```
use Test::Simple tests => 1;
```

```
ok('example.t' eq 'example.t');
```

```
ok((1 + 2) == 2);
```



```
C:\community>perl test_simple.pl
1..1
ok 1
ok 2
# Looks like you planned 1 test but ran 1 extra.
C:\community>
```

Test::More

- Alle notwendigen Funktionen für Tests
 - Zentrale Funktion ist `ok()`
 - Zusätzliche Funktionen sind Wrapper für `ok()`
 - Möglichkeit für Namen der Tests
-

Test::More - ok

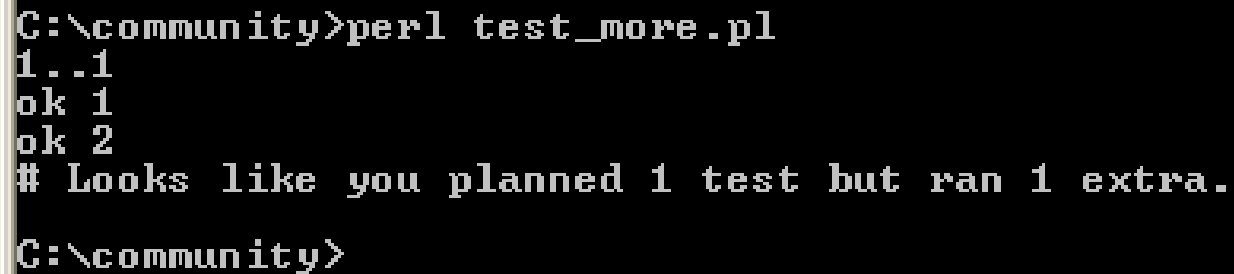
- DIE zentrale Funktion
 - Theoretisch alle Tests mit **ok()**
 - Aufruf: **ok(\$boolean, \$message)**
 - `ok(2 == 3, `2 == 3`)`
 - `ok(„Hallo“ eq „Hello“, „Hello-Vergleich“)`
 - `ok(1)`
-

Test::More - ok

```
#!/usr/bin/perl

use Test::More tests => 1;

ok('example.t' eq 'example.t');
ok((1 + 1) == 2);
```



```
C:\community>perl test_more.pl
1..1
ok 1
ok 2
# Looks like you planned 1 test but ran 1 extra.
C:\community>
```

Test::More - diag

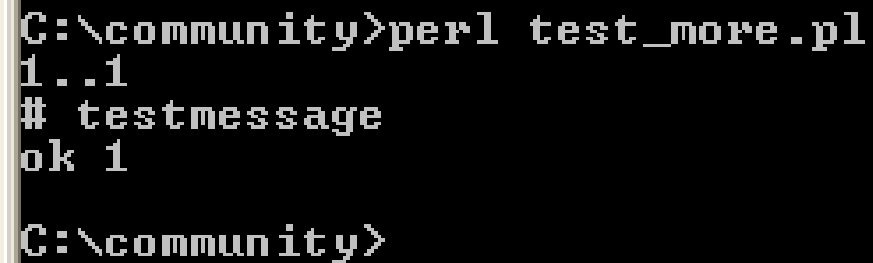
- Kommentarausgabe
 - Erleichtert das Auffinden von Tests
 - Erkennbar am # am Anfang
 - Wird von **Test::Harness** ignoriert
 - Aufruf: **diag(\$message)**
 - `diag(„Testmessage“)`
-

Test::More – diag

```
#!/usr/bin/perl

use Test::More tests => 1;

diag('testmessage');
ok((1 + 1) == 2);
```



```
C:\community>perl test_more.pl
1..1
# testmessage
ok 1

C:\community>
```

Test::More – cmp_ok

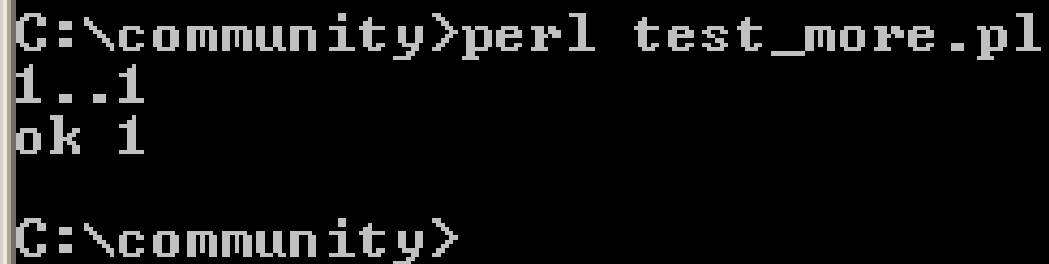
- Vergleicht zwei Werte mit übergebenem Vergleichsoperator
 - Aufruf: `cmp_ok($val, $op, $check)`
 - `cmp_ok('test', 'eq', 'test')`
 - `cmp_ok(1, '==', 1)`
-

Test::More – cmp_ok

```
#!/usr/bin/perl
```

```
use Test::More tests => 1;
```

```
cmp_ok('example.t', 'eq', 'example.t');
```



```
C:\community>perl test_more.pl
1..1
ok 1

C:\community>
```

Test::More - is

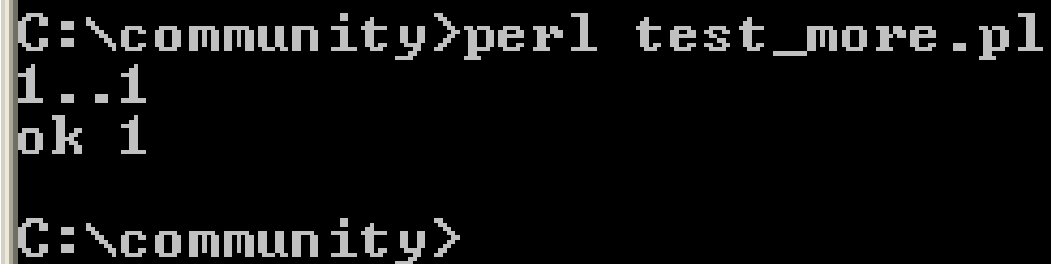
- Vergleich zweier Werte
 - Bei Fehler: Ausgabe von
 - Erwartetem Wert
 - Erhaltenem Wert
 - Aufruf: `is($value, $check[, $message])`
 - `is(2,4)`
 - Gegenteil: `isnt()`
 - Für Arrays etc: `is_deeply()`
-

Test::More – is

```
#!/usr/bin/perl

use Test::More tests => 1;

is('example.t', 'example.t');
```



```
C:\community>perl test_more.pl
1..1
ok 1

C:\community>
```

Test::More – like

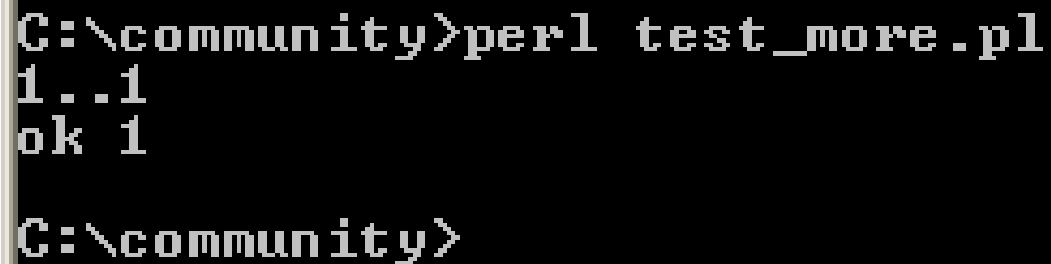
- Entspricht `is()` für RegEx
- Aufruf: `like($value, $regex)`
 - `like(„Larry“,qr/larry/i)`
- Gegenteil: `unlike()`

Test::More – like

```
#!/usr/bin/perl

use Test::More tests => 1;

like('example.t', qr/^ex/);
```



```
C:\community>perl test_more.pl
1..1
ok 1

C:\community>
```

Test::More – is_deeply

- Vergleicht Datenstrukturen (z.B. Arrays)
 - Aufruf: **is_deeply(\$ref_ds, \$ref_check)**
 - `is_deeply(\@array, \@check)`
 - Bessere Alternativen für komplexe DS:
 - **Test::Deep**
 - **Test::Differences**
-

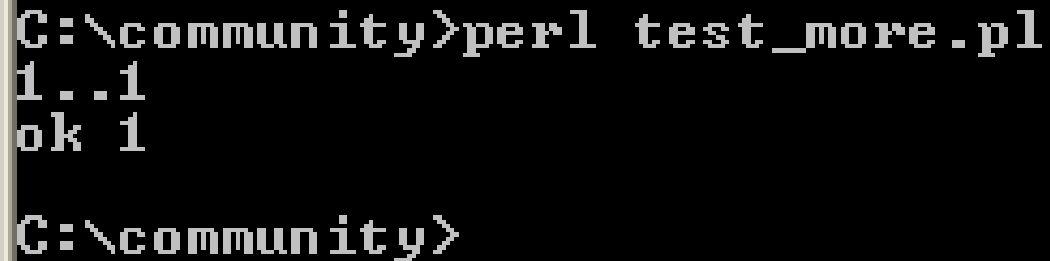
Test::More – is_deeply

```
#!/usr/bin/perl

use Test::More tests => 1;

my @array = qw*12 4 15*;
my @check = qw*12 4 15*;

ok(\@array, \@check);
```



```
C:\community>perl test_more.pl
1..1
ok 1

C:\community>
```

Test::More – use_ok

- Überprüft, ob Modul geladen werden kann
- Aufruf: **use_ok(\$modul)**
 - `use_ok(„Text::Find::Scalar“)`
- Auch für require: **require_ok**



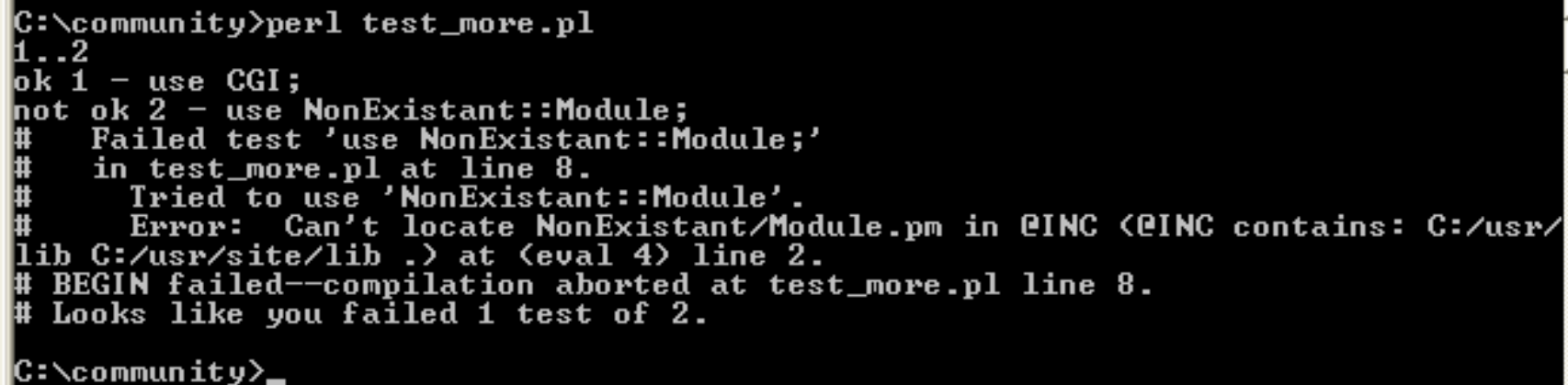
Test::More – use_ok

```
#!/usr/bin/perl
```

```
use Test::More tests => 2;
```

```
use_ok( 'CGI' );
```

```
use_ok( 'NonExistant::Module' );
```



```
C:\community>perl test_more.pl
1..2
ok 1 - use CGI;
not ok 2 - use NonExistant::Module;
# Failed test 'use NonExistant::Module;'
# in test_more.pl at line 8.
# Tried to use 'NonExistant::Module'.
# Error: Can't locate NonExistant/Module.pm in @INC (@INC contains: C:/usr/
lib C:/usr/site/lib .) at <eval 4> line 2.
# BEGIN failed--compilation aborted at test_more.pl line 8.
# Looks like you failed 1 test of 2.
C:\community>_
```

Test::More – can_ok

- Überprüft, ob Klasse/Objekt eine Methode „kennt“
- Nützlich zum Überprüfen, ob alle geplanten Methoden implementiert sind
- Aufruf: `can_ok($obj, @methods)`
 - `can_ok($obj, `new`)`

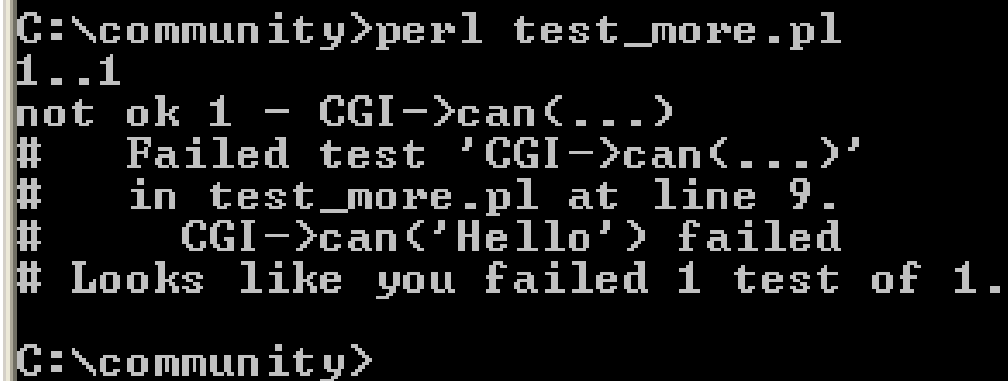


Test::More – can_ok

```
#!/usr/bin/perl

use Test::More tests => 1;
use CGI;

my @array = qw*new Vars Hello*;
can_ok(CGI->new(), @array);
```

A terminal window with a black background and white text. The prompt is 'C:\community>'. The user has entered 'perl test_more.pl'. The output shows '1..1', 'not ok 1 - CGI->can(...)', and a failure message: '# Failed test 'CGI->can(...)' in test_more.pl at line 9. # CGI->can('Hello') failed # Looks like you failed 1 test of 1.' The prompt 'C:\community>' is shown again at the bottom.

```
C:\community>perl test_more.pl
1..1
not ok 1 - CGI->can(...)
# Failed test 'CGI->can(...)'
# in test_more.pl at line 9.
# CGI->can('Hello') failed
# Looks like you failed 1 test of 1.
C:\community>
```

Test::More – isa_ok

- Überprüft Typ eines Skalars
- Überprüft Definiertheit des Skalars
- Wie `ok(ref($scalar) eq „Typ“)`

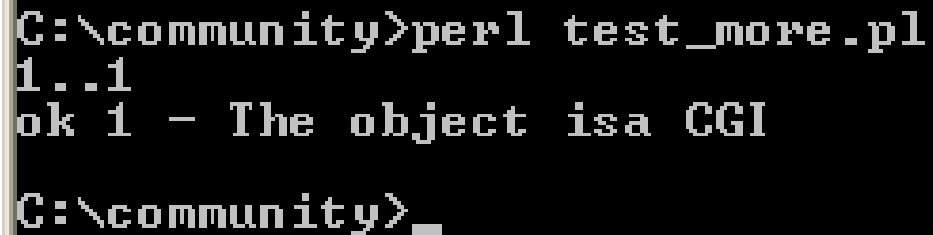


Test::More – isa_ok

```
#!/usr/bin/perl

use Test::More tests => 1;
use CGI;

isa_ok(CGI->new(), 'CGI');
```



```
C:\community>perl test_more.pl
1..1
ok 1 - The object isa CGI
C:\community>_
```

Test::More - SKIP

- Wenn ein Test wegen Fehler nicht laufen kann
 - z.B. Tests die Internet benötigen
 - Tests mit optionalen Modulen



Test::More – SKIP

```
#!/usr/bin/perl
```

```
use Test::More tests => 1;
```

```
SKIP:{  
    eval `require NonExistant::Module`;  
    skip `NonExistant::Module does not exist`, 1 if $@;  
    is(1,1);  
}
```

```
C:\community>perl test_more.pl  
1..1  
ok 1 # skip NonExistant::Module does not exist  
C:\community>
```

Test::More - TODO

- Für „Test-Driven Development“ interessant
 - Wenn Code noch nicht geschrieben
 - Keine Fehler für **Test::Harness**
 - Nachricht bei Erfolg
-

Test::More – TODO

```
#!/usr/bin/perl

use Test::More tests => 1;

TODO:{
    local $TODO = 'subroutine is not implemented yet';
    is(test(),42);
}

sub test{
}
```

```
C:\community>perl test_more.pl
1..1
not ok 1 # TODO subroutine is not implemented yet
#   Failed (TODO) test in test_more.pl at line 10.
#       got: undef
#       expected: '42'

C:\community>_
```

Test::Exception

- Überprüft Code auf Fehler (Abbruch)
 - Z.B. Code mit **or die \$!**
 - Kein Abfangen von Warnungen
- Basiert auf **Test::Builder**



Test::Exception – dies_ok

- Überprüft, ob eine Methode abbricht (mit **die**)
 - Praktisch, um die Verarbeitung von „falschen“ Eingaben zu überprüfen
 - Z.B. nicht existierende Datei
 - Nicht Fehler-spezifisch
 - Aufruf: **dies_ok {BLOCK}**
 - `dies_ok {$obj->method() }`
-

Test::Exception – dies_ok

```
#!/usr/bin/perl

use Test::Coverage;
use Test::More tests => 1;
use ExceptionTest;

my $obj = ExceptionTest->new();
dies_ok { $obj->dies() };
```

```
package ExceptionTest;

sub new{
    bless {}, shift;
}

sub dies{
    die;
}

1;
```

```
C:\community>perl test_exception.pl
1..1
ok 1

C:\community>
```

Test::Exception – throws_ok

- Fehler spezifische Überprüfung
 - Z.B. ob Fehler RegEx matcht
 - Von einer bestimmten Klasse ist
- Aufruf: **throws_ok {BLOCK} \$regex**
 - `throws_ok {$obj->method()} qr/a/`



Test::Exception – throws_ok

```
#!/usr/bin/perl

use Test::Coverage;
use Test::More tests => 1;
use ExceptionTest;

my $obj = ExceptionTest->new();
throws_ok {$obj->throws()}qr/division/;
```

```
package ExceptionTest;

sub new{
    bless {}, shift;
}

sub throws{
    my $zero = 0;
    my $res = 1/$zero;
    return $res;
}
1;
```

```
C:\community>perl test_exception.pl
1..1
ok 1 - threw Regexp (<(?-xism:division)>)

C:\community>_
```

Test::Exception – lives_ok

- Methode muss „normal“ laufen
 - darf nicht abbrechen
- Aufruf: `lives_ok {BLOCK}`
 - `lives_ok {$obj->method() }`



Test::Exception – lives_ok

```
#!/usr/bin/perl

use Test::Coverage;
use Test::More tests => 1;
use ExceptionTest;

my $obj = ExceptionTest->new();
lives_ok {$obj->lives()};
```

```
package ExceptionTest;

sub new{
    bless {}, shift;
}

sub lives{
    return;
}

1;
```

```
C:\community>perl test_exception.pl
1..1
ok 1

C:\community>_
```

Test::Exception – lives_and

- Wenn Methode möglicherweise abbricht
 - Vermeidet unnötige Schritte
 - Kombination aus **lives_ok** und anderem Test von **Test::More**
 - Aufruf: **lives_and {BLOCK}\$msg;**
 - `lives_and {is $obj->method(),42}'test'`
-

Test::Exception – lives_and

```
#!/usr/bin/perl

use Test::Coverage;
use Test::More tests => 1;
use ExceptionTest;

my $obj = ExceptionTest->new();
lives_and {is $obj->lives_and(),
          42} 'test';
```

```
package ExceptionTest;

sub new{
    bless {}, shift;
}

sub lives_ok{
    return 42;
}

1;
```

```
C:\community>perl test_exception.pl
1..1
ok 1 - test

C:\community>_
```

Test::Pod

- Testet die POD-Dokumentation auf Korrektheit
 - gemäß `perldoc perlpod`
 - Kennt nur 3 Methoden
 - `pod_file_ok`
 - `all_pod_files_ok`
 - `all_pod_files`
-

Test::Pod – pod_file_ok

- Überprüft, ob die Dokumentation in einer Datei korrekt ist
- Aufruf: `pod_file_ok($podfile[, $testname])`
 - `pod_file_ok('Module.pm', 'Module.pm-Test');`



Test::Pod – pod_file_ok

```
#!/usr/bin/perl
```

```
use Test::Pod tests => 1;
```

```
pod_file_ok($0);
```

```
=pod
```

```
=head1 Test
```

```
this is a test for Test::Pod
```

```
=cut
```

```
C:\community>test_pod_1.pl
1..1
ok 1 - POD test for C:\community\test_pod_1.pl
C:\community>
```

Test::Pod – pod_file_ok

```
#!/usr/bin/perl
```

```
use Test::Pod tests => 1;
```

```
pod_file_ok($0);
```

```
=pod
```

```
=head1 Test
```

```
this is a test for Test::Pod E<module@renee-  
baecker.de>
```

```
=cut
```

```
C:\community>test_pod_1.pl  
1..1  
not ok 1 - POD test for C:\community\test_pod_1.pl  
# Failed test 'POD test for C:\community\test_pod_1.pl'  
# in C:\community\test_pod_1.pl at line 5.  
# C:\community\test_pod_1.pl (11): Unknown E content in E<module@renee-  
baecker.de>  
# Looks like you failed 1 test of 1.  
C:\community>
```

Test::Pod – all_pod_files_ok

- Prüft Dokumentation aller Dateien einer Liste
- Keine Angabe von `tests`
- Aufruf: **all_pod_files_ok(@files)**
 - `all_pod_files_ok($0);`



Test::Pod – all_pod_files_ok

```
#!/usr/bin/perl
```

```
use Test::Pod tests => 1;
```

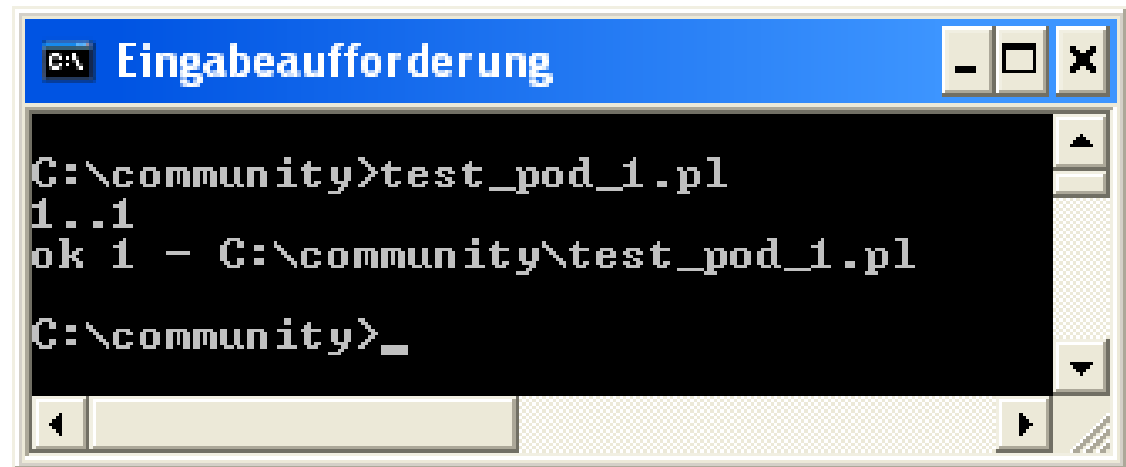
```
all_pod_files_ok($0);
```

```
=pod
```

```
=head1 Test
```

```
this is a test for Test::Pod
```

```
=cut
```



```
C:\community>test_pod_1.pl
1..1
ok 1 - C:\community\test_pod_1.pl

C:\community>_
```

Test::Pod – all_pod_files

- Liefert alle Dateien zurück, die POD enthalten
- Aufruf: **all_pod_files(@verzeichnisse)**
 - `all_pod_files('.')`;



Test::Pod – all_pod_files

```
#!/usr/bin/perl

use Test::Pod tests => 1;

my @files = all_pod_files(".");
print $_ for @files;

=pod

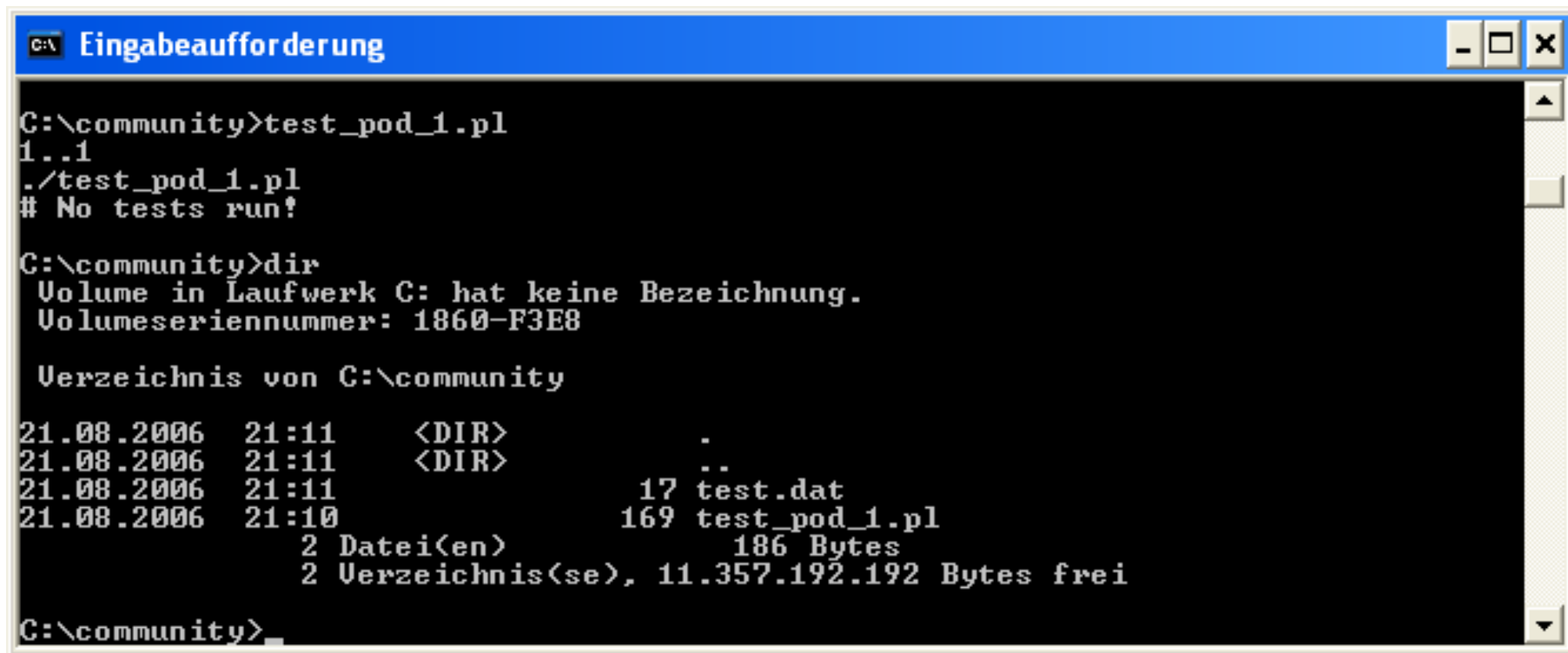
=head1 Test

this is a test for Test::Pod

=cut
```

```
C:\community>more test.dat
Dies ist ein Test
```

Test::Pod – all_pod_files



```
C:\community>test_pod_1.pl
1..1
./test_pod_1.pl
# No tests run!

C:\community>dir
Volume in Laufwerk C: hat keine Bezeichnung.
Volumeseriennummer: 1860-F3E8

Verzeichnis von C:\community

21.08.2006  21:11    <DIR>          .
21.08.2006  21:11    <DIR>          ..
21.08.2006  21:11                17 test.dat
21.08.2006  21:10            169 test_pod_1.pl
                2 Datei(en)           186 Bytes
                2 Verzeichnis(se), 11.357.192.192 Bytes frei

C:\community>
```

Test::Pod - Beispiel

```
# change 'tests => 1' to 'tests => last_test_to_print';
```

```
use Test::More;
```

```
SKIP: {  
    eval "use Test::Pod 1.00";  
    skip "Test::Pod 1.00 required", 1 if $@;  
    my @poddirs = qw(blib);  
    all_pod_files_ok(all_pod_files(@poddirs));  
}
```

Test::Pod::Coverage

- Überprüft, ob alle Methoden im POD dokumentiert sind
 - Stellt drei Methoden zur Verfügung
 - ❑ `all_pod_coverage_ok`
 - ❑ `pod_coverage_ok`
 - ❑ `all_modules`
-

Test::Pod::Coverage – all_pod_coverage_ok

- Überprüft alle Module auf Korrektheit
- Aufruf:

```
all_pod_coverage_ok( [ $params , ] $msg )
```

```
□ all_pod_coverage_ok()
```

Test::Pod::Coverage – all_pod_coverage_ok

```
#!/usr/bin/perl
```

```
use Test::Pod::Coverage;  
use Test::More tests => 1;  
  
all_pod_coverage_ok();
```

```
lib  
|-- FabForce  
|   |-- DBDesigner4  
|   |   |-- Table.pm  
|   |   |-- SQL.pm  
|   |   |-- XML.pm  
|   --- DBDesigner4.pm  
--- t  
    |-- FabForce-DBDesigner4.t  
    --- test_pod_coverage.t
```

```
C:\Dokumente und Einstellungen\Renee\Eigene Dateien\Perl\cpan\FabForce-DBDesigner4>nmake test
```

```
Microsoft (R) Program Maintenance Utility   Version 1.50  
Copyright (c) Microsoft Corp 1988-94. All rights reserved.
```

```
cp lib\FabForce\DBDesigner4\XML.pm blib\lib\FabForce\DBDesigner4\XML.pm  
Skip blib\lib\FabForce\DBDesigner4.pm (unchanged)  
Skip blib\lib\FabForce\DBDesigner4\SQL.pm (unchanged)  
Skip blib\lib\FabForce\DBDesigner4\Table.pm (unchanged)  
C:\usr\bin\perl.exe "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib\lib', 'blib\arch')" t/*.t  
t/FabForce-DBDesigner4....ok  
t/test_pod_coverage.....ok 1/4  
# Failed test 'Pod coverage on FabForce::DBDesigner4::XML'  
# in C:/usr/site/lib/Test/Pod/Coverage.pm at line 126.  
# Coverage for FabForce::DBDesigner4::XML is 0.0%, with 3 naked subroutines:  
# new  
# parsefile  
# writeXML  
# Looks like you failed 1 test of 4.  
t/test_pod_coverage.....dubious  
Test returned status 1 (wstat 256, 0x100)  
DIED. FAILED test 4  
Failed 1/4 tests, 75.00% okay  
Failed Test Stat Wstat Total Fail Failed List of Failed  
-----  
t/test_pod_coverage.t 1 256 4 1 25.00% 4  
Failed 1/2 test scripts, 50.00% okay. 1/9 subtests failed, 88.89% okay.  
NMAKE : fatal error U1077: 'C:\WINDOWS\system32\cmd.exe' : return code '0x1'  
Stop.
```

```
C:\Dokumente und Einstellungen\Renee\Eigene Dateien\Perl\cpan\FabForce-DBDesigner4>
```

```
C:\Dokumente und Einstellungen\Renee\Eigene Dateien\Perl\cpan\FabForce-DBDesigner4>nmake test

Microsoft (R) Program Maintenance Utility   Version 1.50
Copyright (c) Microsoft Corp 1988-94. All rights reserved.

cp lib\FabForce\DBDesigner4\XML.pm blib\lib\FabForce\DBDesigner4\XML.pm
Skip blib\lib\FabForce\DBDesigner4.pm (unchanged)
Skip blib\lib\FabForce\DBDesigner4\SQL.pm (unchanged)
Skip blib\lib\FabForce\DBDesigner4\Table.pm (unchanged)
      C:\usr\bin\perl.exe "-MExtUtils::Command::MM" "-e" "test_harness<@, 'bli
b\lib', 'blib\arch'" t/*.t
t/FabForce-DBDesigner4....ok
t/test_pod_coverage.....ok
All tests successful.
Files=2, Tests=9,  2 wallclock secs < 0.00 cusr + 0.00 csys = 0.00 CPU>

C:\Dokumente und Einstellungen\Renee\Eigene Dateien\Perl\cpan\FabForce-DBDesigner4>
```

Test::Pod::Coverage – pod_coverage_ok

- Überprüft für ein Module die Korrektheit
- Aufruf:

```
pod_coverage_ok($module, [$params, ]$msg)
```

```
□ pod_coverage_ok("My::Module")
```

Test::Pod::Coverage – pod_coverage_ok

```
#!/usr/bin/perl
```

```
use Test::Pod::Coverage;  
use Test::More tests => 1;
```

```
pod_coverage_ok("FabForce::DBDesigner4");
```

```
C:\Dokumente und Einstellungen\Renee\Eigene Dateien\Perl\cpan\FabForce-DBDesigner4>nmake test  
  
Microsoft (R) Program Maintenance Utility   Version 1.50  
Copyright (c) Microsoft Corp 1988-94. All rights reserved.  
  
      C:\usr\bin\perl.exe "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib\lib', 'blib\arch')" t/*.t  
t/FabForce-DBDesigner4....ok  
t/test_pod_coverage.....ok  
All tests successful.  
Files=2, Tests=6,  2 wallclock secs ( 0.00 cusr +  0.00 csys =  0.00 CPU)  
  
C:\Dokumente und Einstellungen\Renee\Eigene Dateien\Perl\cpan\FabForce-DBDesigner4>
```

Test::Pod::Coverage – all_modules

- Gibt alle Module in Verzeichnissen zurück
- Default: blib bzw. lib
- Aufruf: **all_modules([@dirs])**
 - `all_modules()`



Test::Pod::Coverage – all_modules

```
#!/usr/bin/perl

use Test::Pod::Coverage;
use Test::More;

my @mods = all_modules();
print $_, "\n" for @mods;
```

```
C:\Dokumente und Einstellungen\Renee\Eigene Dateien\Perl\cpan\FabForce-DBDesigner4>perl t\test_pod_coverage.t
FabForce::DBDesigner4
FabForce::DBDesigner4::SQL
FabForce::DBDesigner4::Table
FabForce::DBDesigner4::XML

C:\Dokumente und Einstellungen\Renee\Eigene Dateien\Perl\cpan\FabForce-DBDesigner4>
```

Test::Pod::Coverage - Beispiel

```
# change 'tests => 1' to 'tests => last_test_to_print';
```

```
use Test::More;
```

```
SKIP: {  
    eval "use Test::Pod::Coverage";  
    skip "Test::Pod::Coverage required", 1 if $@;  
    pod_coverage_ok("Text::Find::Scalar");  
}
```

Test::TestCoverage

- Werden alle „public“-Methoden aufgerufen?
 - Lädt Modul, liest Methodennamen aus
 - „privat“ beginnt mit _
 - Überprüft Methodenaufrufe
- Ähnlich wie **Devel::Cover**



Test::TestCoverage – test_coverage

- „registrieren“ des Moduls
- Aufruf: **test_coverage(\$module)**
 - `test_coverage('My::Module');`



Test::TestCoverage – test_coverage

```
#!/usr/bin/perl

use Test::Coverage;

test_coverage( "Testmodule" );
```

```
package Testmodule;

sub methode{
    return 1;
}

1;
```

Test::TestCoverage – ok_test_coverage

- Test, ob alle “public”-Methoden aufgerufen wurden.
- Verwendet **Test::Builder**-Methoden
- Aufruf: **ok_test_coverage(\$module)**
 - `ok_test_coverage('My::Module');`



Test::TestCoverage – ok_test_coverage

```
#!/usr/bin/perl

use Test::Coverage;

test_coverage("Testmodule");
Testmodule::methode();
ok_test_coverage("Testmodule");
```

```
package Testmodule;

sub methode{
    return 1;
}

1;
```

```
C:\community>perl test_testcoverage.pl
1..1
ok 1 - Test test-coverage
C:\community>_
```

Test::TestCoverage – ok_test_coverage

```
#!/usr/bin/perl

use Test::Coverage;

test_coverage("Testmodule");
#Testmodule::methode()
ok_test_coverage("Testmodule");
```

```
package Testmodule;

sub methode{
    return 1;
}

1;
```

```
C:\community>perl test_testcoverage.pl
1..1
not ok 1 - Test test-coverage methode are missing
# Failed test 'Test test-coverage methode are missing'
# in test_testcoverage.pl at line 8.
# got: 0
# expected: 1
# Looks like you failed 1 test of 1.

C:\community>
```

Test::TestCoverage – test_coverage_except

- Bestimmt Ausnahmen der Überprüfung
- Aufruf:

```
test_coverage_except($module,@methods)
```

```
□ test_coverage_except('My::Module','foo','bar')
```

Test::TestCoverage – test_coverage_except

```
#!/usr/bin/perl
```

```
use Test::Coverage;
```

```
test_coverage("Testmodule");
```

```
test_coverage_except("Testmodule", "bar");
```

```
Testmodule::methode();
```

```
ok_test_coverage("Testmodule");
```

```
package Testmodule;
```

```
sub methode{  
    return 1;  
}
```

```
sub bar{  
    return 3;  
}
```

```
1;
```

Test::TestCoverage – reset_test_coverage

- Setzt den Zähler für ein Modul zurück
 - Ermöglicht blockweises Vorgehen
- Aufruf: **reset_test_coverage(\$module)**
 - `reset_test_coverage ('My::Module')`



Test::TestCoverage – ok_test_coverage

```
#!/usr/bin/perl

use Test::Coverage;

test_coverage("Testmodule");
#Testmodule::methode()
ok_test_coverage("Testmodule");
```

```
package Testmodule;

sub methode{
    return 1;
}

1;
```

```
C:\community>perl test_testcoverage.pl
1..1
not ok 1 - Test test-coverage methode are missing
# Failed test 'Test test-coverage methode are missing'
# in test_testcoverage.pl at line 8.
# got: 0
# expected: 1
# Looks like you failed 1 test of 1.

C:\community>
```

Test::TestCoverage – reset_all_test_coverage

- Setzt die Zähler aller Module zurück
- ruft intern **reset_test_coverage** auf
- Aufruf: **reset_all_test_coverage()**



Test::TestCoverage – ok_test_coverage

```
#!/usr/bin/perl

use Test::Coverage;

test_coverage("Testmodule");
#Testmodule::methode()
ok_test_coverage("Testmodule");
```

```
package Testmodule;

sub methode{
    return 1;
}

1;
```

```
C:\community>perl test_testcoverage.pl
1..1
not ok 1 - Test test-coverage methode are missing
# Failed test 'Test test-coverage methode are missing'
# in test_testcoverage.pl at line 8.
# got: 0
# expected: 1
# Looks like you failed 1 test of 1.

C:\community>
```

Vorsicht!

- False Dilemma:
 - Falscher Test + Falscher Code = richtiges Ergebnis



Was bringt mir das jetzt?

- Qualität eines Moduls steigern
 - Aufbau einer Test-Suite
 - Sicherheit bei zukünftigen Änderungen
-

Weiterführende Links

- <http://petdance.com/perl/automated-testing/>
 - http://www.perl.com/pub/a/2005/12/08/test_files.html
 - http://langworth.com/pub/perl_test_refcard.pdf
 - <http://qa.perl.org/>
 - <http://www.ora.de/catalog/perltestingadn/chapter/>
 - <http://www.qapodcast.com/news/2006/02/23/qa-podcast-10-testing-with-perl-with-ian-langworth-and-chromatic>
 - <http://www.wgz.org/chromatic/perl/IntroTestMore.pdf>
 - <http://www.linux-magazin.de/Artikel/ausgabe/2005/11/perl/perl.html>
 - <http://twoalpha.blogspot.com/2005/11/unit-testing-in-perl-with-testunit.html>
 - http://perl.net.au/wiki/Perl_testing_tools
-