Programmieren mit Ruby

Armin Roehrl und Stefan Schmiedl
{a,s}@approximity.com
Approximity GmbH, http://www.approximity.com

13. Februar 2002

> For an instant I thought Ruby was reading my mind . . . if matz can do that, it IS a powerful language! –Hal Fulton

Diese Unterlagen basieren auf einem Artikel, den wir für Linux Enterprise (10/2001) geschrieben haben und auf einem Kapitel unseres Buches “Programmieren mit Ruby”, das wir mit Clemens Wyss zusammen verfasst haben und das gerade ausgeliefert wird. Wir danken Nadja Rosmann von Linux Enterprise und René Schönfeldt vom dpunkt.verlag für die Erlaubnis, die Inhalte zu recyclen.

- Ruby ist Open Source und in Japan schon sehr weit verbreitet.
- Ruby ist Web-tauglich.
- Ruby ist ideal für Prototyping und große Systeme
- Ruby ist einfach (und) mächtig.

Ruby: Funkelndes Juwel

Ruby ist eine (bei uns) relativ neue, interpretierte Sprache, in der zum einen (kleine) Skripte erstellt werden können, aber auch größere Projekte objektorientiert wachsen und gedeihen. Der große Vorteil gegenüber Perl ist, dass die Objekte nicht aufgepfropft sind, sondern ihre Wurzeln tief in der Sprachstruktur verankert haben.

Wir wollen in diesem einführenden Tutorial die Stärken von Ruby aufzeigen und in einem Nachfolger dann einige kleine Projekte vollständig entwickeln, um die Flexibilität und Effizienz von Ruby-Skripten zu demonstrieren.

Nicht noch eine Programmiersprache!

Rubys Abstammung ist ziemlich exotisch. Yukihiro “matz” Matsumoto hat sie seit 1993 entwickelt, in ihrer japanischen Heimat ist sie schon lange populär, beliebter jedenfalls als Python. Ruby ist als Open Source frei verfügbar [Ruby], mittlerweile gibt es auch englische Literatur, das erste deutsche Buch sollte dieser Tage ausgeliefert werden. matz wollte seine neue Sprache wie Perl

nach einem Juwel bennenn und er entschied sich für Ruby, nach dem Geburtsstein eines Kollegen. Erst später merkte er, dass Ruby als Nachfolger von Perl prädestiniert ist: im Alphabet, bei Geburtssteinen (Perle = Juni, Rubin = Juli) [RubyFAQ].

Ruby wird zwar hauptsächlich unter Linux entwickelt, läuft aber auch auf vielen anderen Unix Varianten, sowie DOS, Windows-Varianten, Mac, BeOS, OS/2, etc.

Lange Zeit gab es nur japanische Webseiten und Bücher, mittlerweile verbreiten Ruby-Enthusiasten die Sprache aber weltweit in immer stärkerem Maße. Dokumentationen sind mittlerweile auch in nicht-fernöstlichen Sprachen verfügbar, siehe [TH01]. Dieses Buch wurde auch übersetzt. Also gibt es mit unserem Rubybuch [RSW02] schon zwei deutschen Rubybücher.

Von RubyUnit [Rubyunit] über native Windows Calls, COM, CGI scripts, eingebettetem Ruby in HTML Seiten (eRuby, entspricht ASP, JSP oder PHP) bis zu mod_ruby [modruby] (Ruby Interpreter direkt im Apache Server) unterstützt Ruby bereits alles, was das Herz sich wünschen kann. Ruby besitzt eine sehr aktive Entwicklungsgemeinde, auch die Newsgruppe comp.lang.ruby ist sehr aktiv.

Prinzipien

Man nehme das Beste von Smalltalk, Perl, Eiffel, Scheme, CLU, Sather und Lisp, mische mehrmals stark und warte, bis sich ein Rubin herauskristallisiert. Von der Objektstruktur her ist Ruby näher an Smalltalk als an Lisp, auch die mächtigen Lispmakros fehlen, obwohl durch eval wieder Einiges wettgemacht wird.

Matz hat Ruby als seine “ideale Sprache” von Anfang an nach den Prinzipien des Mensch-Maschinen Interfaces designt [DrDobbs]:

- Konsequent: Eine kleine Menge von Regeln sollte die ganze Sprache definieren. Matz vertritt das Prinzip der kleinsten Überraschung (least surprise)

- Knapp: Um Zeit zu sparen, werden überflüssige Sprachelemente enfernt. Larry Wall, der Autor von Perl meint, dass die Sprache ausdrucksschwächer (weniger “expressiv”) ist, da einige Shortcuts fehlen. Ruby und Python sind im Gegensatz zu Perl mehr das Produkt von Informatikern, die versuchen, minimalistisch zu sein [LarryWall].

- Flexibel: Eine Sprache sollte expressiv sein, so daß man alles mit ihr machen kann. Einfache Dinge sollten einfach sein und schwere Dinge sollten möglich sein. Auf Grund der gut durchdachten Objektstruktur in Ruby müssen schwere Probleme aber nicht unbedingt schwer zu lösen sein.

- Ruby legt den Programmierer nicht durch formale Zwänge an die Leine, in alter Unix-Tradition bekommt er aber auch ausreichend Seil, um sich seine eigene Schlinge zu knüpfen.

Details

Installation und Aufruf

Die Installation ist einfach: `./configure && make install` tut's auf Unix-Systemen, für Windows gibt es einen Installer, der von den "Pragmatic Programmers" bereit gestellt wird.

Wer nur mal reinschnuppern will, kann unter nttp://www.ruby.ch [RubyCH] mit Clemens' Online-Interpreter kleine Skripte ausprobieren, ohne Ruby auf dem eigenen Rechner installieren zu müssen.

Ruby läuft (wie auch Perl und Python) in einer Konsole. Mit ruby skript.rb wird das genannte Skript ausgeführt, Freunde von Einzeilern werden `ruby -e '...'` zu schätzen wissen. Mit `irb` schließlich steht eine interaktive Ruby-Shell zur Verfügung, in die die Beispiele weiter unten ohne Probleme eingetippt werden können.

Codekonventionen und grundlegende Syntax

Die folgenden Informationen sollte man beim Lesen und Schreiben von Ruby-Programmen im Hinterkopf behalten:

- Anweisungen werden durch ein Semikolon (`;`) voneinander getrennt. Wenn eine Zeile nur eine Anweisung enthält, kann das abschließende "`;`" entfallen.

- Wenn Argumente bei Methodenaufrufen in Klammern gesetzt werden, schließt die öffnende Klammer unmittelbar an den Methodennamen an.

```
puts ("hallo")  # nicht gut
puts("hallo")   # ok
puts "hallo"    # ok
```

- Operatoren werden (wegen der besseren Lesbarkeit) in der Regel von Leerzeichen umgeben.

- Der Operator `=` ist eine syntaktische Alternative für den Aufruf von Setter-Methoden, die Zuweisungen im weitesten Sinne vornehmen.

- Das Suffix `?` zeigt an, dass die Methode eine Ja-/Nein-Frage beantwortet und folglich entweder `true` oder `false` zurückliefert.

```
0.zero?                  #-> true
1.zero?                  #-> false
(0.3 - 0.1 - 0.2).zero?  #-> false
0.3 - 0.1 - 0.2          #-> -2.775557562e-17
```

- Ein abschließendes `!` zeigt an, dass die Methode "gefährlich" ist, weil sie zum Beispiel ein bestehendes Array modifiziert, ohne vorher eine Kopie zu erzeugen.

```
x = ’Stefan’, ’Armin’, ’Clemens’]
x.sort    #-> ["Armin", "Clemens", "Stefan"]
x         #-> ["Stefan", "Armin", "Clemens"]
x.sort!   #-> ["Armin", "Clemens", "Stefan"]
x         #-> ["Armin", "Clemens", "Stefan"]
```

Garbage Collection

Ruby besitzt einen echten “mark-and-sweep” Garbage Collector, der auf alle Ruby Objekte wirkt. Sobald ein Objekt nicht mehr von einem anderen Objekt referenziert wird, wird es beseitigt. Deshalb hat Ruby wie Java (und im Gegensatz zu C++) keine Destruktoren. Der Vorteil ist, dass das leidige Referenzzählen entfällt, das bei manchen Perl- und Python-Programmierern schon Haarausfall verursacht haben soll.

Auf Grund des Garbage Collectors und des ausgefeilten Extension-APIs kann Ruby sehr einfach durch C-Funktionen erweitert werden. Sinnvoll ist das z. B. bei aufwändigen numerischen Berechnungen. Neben dieser “handwerklichen” Methode gibt es auch noch SWIG (Simplified Wrapper and Interface Generator) [SWIG], der die Einbindung von C-Funktionen fast vollständig automatisiert.

Code-Blöcke

Ruby ermöglicht auch das Arbeiten mit Code-Blöcken. Ein Block wird durch geschweifte Klammern `{ ... }` oder `do ... end` begrenzt und kann als zusätzliches Argument an eine (jede) Methode übergeben werden. Wer sich einmal an das Arbeiten mit solchen anonymen Blöcken gewöhnt hat, wird sie nicht mehr missen wollen.

```
def dreimal
  yield; yield; yield
end
dreimal { puts "Hi" }  #-> Hi Hi Hi
dreimal { puts "Ho" }  #-> Ho Ho Ho
```

`yield` wird mächtiger, wenn man an die Blöcke Argumente übergibt und von ihnen Werte zurück bekommt. Hier dringt ein bisschen Smalltalk-Syntax durch:

```
def findnums(n)
  i = 0
  while i < n
    yield i
    i = i + 3
  end
end

findnums(20) { |p| print p, " " }
#-> 0 3 6 9 12 15 18
```

Blöcke in Verbindung mit Iteratoren ermöglichen schließlich sehr elegante Formulierungen. Analog dem Visitor-Pattern besucht der Block jedes Element der Liste:

```
[1,25,30].each {|i| puts i} #-> 1 25 30
```

Ruby hat auch echte Closures (wie Lisp), die sich die Variablen-Bindungen zum Zeitpunkt ihrer Definition merken.

```
def nTimes(aThing)
  return proc{ |n| aThing * n }
end
p1 = nTimes(23)
p1.call(1)      #-> 23
p1.call(3)      #-> 69
```

Mit `proc` wird aus einem Block eine Funktion gemacht, die als Wert von `nTimes` zurückgeliefert wird. Diese Funktion hat auf den Wert des `aThing`-Parameters zum Zeitpunkt des `nTimes`-Aufrufs Zugriff.

Unabhängige Zähler lassen sich so problemlos erzeugen:

```
def counter(start)
  return proc { start += 1 }
end

f = counter(0)
g = counter(10)
p f.call #-> 1
p f.call #-> 2
p g.call #-> 11
p f.call #-> 3
```

Zahlen zählen

In Ruby sind ganze Zahlen entweder Fixnums (`<2**30`) oder Bignums, die bei Bedarf ineinander umgewandelt werden, wovon der Programmierer in der Regel nichts merkt.

```
def fact(n)
  1 if n == 0; f = 1
  while n>0
    f *= n; n -= 1
  end
  f
end

print fact(ARGV[0].to_i)
#-> 10093326215443944....
```

Reguläre Ausdrücke

Reguläre Ausdrücke werden ähnlich wie in Perl und Python unterstützt, sie können sowohl mit den Perl-Kürzeln als auch mit Konstruktoren erzeugt werden. Das beim Match erzeugte MatchData-Objekt enthält in einem Vektor den kompletten Bereich, auf den das Suchmuster gepasst hat sowie die einzelnen geklammerten Komponenten:

```
aString = "dass das daß jetzt dass ist ..."

/da(ß|ss)/.match(aString)[0]             #-> daß
Regexp.new('da(ß|ss)').match(aString)[0] #-> daß
/da(ß|ss)/.match(aString)[1]             #-> ß
Regexp.new('da(ß|ss)').match(aString)[1] #-> ß
```

Ersetzungen sind ebenso einfach möglich

```
aString.sub('(da)(ß|ss)', '\1ss')
```

Reflexion

Ruby besitzt mächtige Reflexionsmechanismen, welche vor allem beim verteilten Einsatz oder für Marshalling von Interesse sind.

```
x = 43.2
ObjectSpace.each_object(Numeric){|o| p o}
#-> 43.2 2.718281828 3.141592654
```

Die beiden letzten Zahlen sind Näherungswerte für E und PI, die im "Math"-Modul definiert sind und wir alle "lebenden" Objekte zeigen ließen.

Objekte, Klassen und Module

Ruby zeichnet sich durch eine einfache Syntax aus, die teilweise an Eiffel und Ada erinnert. Befehle enden nicht mit Strichpunkten, wenn nur ein Kommando pro Zeile steht. Kommentare verwenden wie in Perl das #. Ebenso wie in Java oder Python sind Exceptions in Ruby integriert, wodurch die Behandlung von Ausnahmesituationen enorm vereinfacht wird.

Rubys Operatoren sind "nur" syntaktische Alternativen für normale Methoden, sie lassen sich bei Bedarf leicht umdefinieren. Wie Smalltalk ist Ruby eine rein objektorientierte Sprache, allerdings sind Kontrollstrukturen als besondere Sprachelemente implementiert. Dafür hat Ruby aber auch eine Syntax die von den meisten Programmierern ohne Schwierigkeiten gelesen werden kann.

Zum Beispiel ist die Zahl 1 ein Ruby eine Instanz der Klasse Fixnum. Funktionsaufrufe sind stets Botschaften, die an ein Objekt geschickt werden. Der Empfaenger steht links, die Funktion rechts vom Punkt:

```
-4.abs() #-> 4      oder auch ohne Klammern:
-4.abs   #-> 4
```

Die komplette Klassenstruktur ist auch während des Programmablaufs offen für dynamische Veränderungen. Es gibt also keine Probleme mit versiegelten Klassen, die nicht mehr geändert werden können. Falls nötig, kann so eine Instanz einer Klasse (Singleton) sich anders verhalten als eine andere Instanz derselben Klasse.

Für die Definition einer Methode wird das Schlüsselwort `def` verwendet, gefolgt vom Funktionsnamen und einer Argumentenliste, für die Parameter können auch Standardwerte gesetzt werden.

```
def eineMethode(x, y=7)
  ...
end
```

Tritt `def` außerhalb einer Klasse auf, bekommt man eine "gewöhnliche" Funktion, ähnlich wie Perls `sub`-Routinen.

Klassendeklarationen beginnen mit `class`, Instanzen werden mit der `new`-Methode erzeugt.

```
class Krokodil
  def essen
    p "Haps"
  end
end
k = Krokodil.new
k.essen            #-> haps
```

Ruby unterstützt bewusst nur einfache Vererbung (single inheritance), um den Komplikationen der Mehrfachvererbung aus dem Weg zu gehen. Vererbung wird durch `<` gefolgt von der Basisklasse angezeigt.

```
class Krokodil < Tier
   ...
```

Ein Modul (Kategorie in Objective-C) gruppiert Methoden und Konstanten. Eine Klasse kann ein Modul importieren und bekommt so dessen Methoden. Dieses Mix-inVerfahren ermöglicht es, die Vorteile der Mehrfachvererbung zu nutzen, ohne sich mit ihrer Komplexität aufzuhalten. Im Unterschied zu reinen Interfaces wie in Java enthalten Module auch die Implementation, so dass in Ruby also "generische" Programmierung von Haus aus eingebaut ist.

Ein Beispiel hierfür ist das Comparable-Modul, das die üblichen Vergleiche basierend auf `<=>` implementiert. Wenn also eine Klasse über eine Implementation von `<=>` verfügt, kann sie das Modul Comparable einbinden und damit alle Vergleiche für ihre Instanzen erhalten.

Variablen und Datentypen

Ruby ist eine der dynamischen Sprachen, bei denen die Variablen nur als Speicher für Objekte dienen, und damit keinen Datentyp benötigen. Die Objekte selbst haben sehr wohl unterschiedliche Typen.

Eine einfache Namenskonvention beschreibt den Geltungsbereich von Variablen:

```
a   # lokal in der umgebenden Funktion
$b  # global für das Skript
@c  # lokal für die Instanz einer Klasse
@@d # global für die Klasse
E   # Konstante oder Klassenname
```

Auf eine Klassenvariable kann erst zugegriffen werden, wenn sie mit einem Wert initialisiert worden ist, dagegen haben nicht-initialisierte lokale oder Instanzvariablen den unbestimmten Wert `nil`. Letzteres erlaubt eine elegante lazy initialization":

```
class Krokodil
  def zaehne
    unless @zaehne
      @zaehne = zaehleZaehne
    end
    @zaehne
  end

  # oder noch kompakter
  def schuppen
    @schuppen ||= zaehleSchuppen
  end
end
```

Einfache Benchmarks

Ein nicht sehr repräsentativer Benchmark, aber trotzdem interessant: Fibonnacci Zahlen sind nach der bekannten Formel $f(n) = f(n-1) + f(n-2)$ mit $f(0) = f(1) = 1$ zu berechnen. Wir haben einige Programme von [Fibonacci] übernommen und den Rest selber geschrieben.

```
#!/usr/local/bin/ruby
def fib(n)
  if n<2
    n
  else
   fib(n-2)+fib(n-1)
  end
end
print(fib(30), "\n")
```

Die Benchmarks wurden auf einem Pentium II mit 266MHz Linux(2.2.18) durchgeführt.

Sprache:	Laufzeit	Version
Assembler	0.25 s	GNU assembler 2.10.91
C	0.3 s	GCC 2.95.2
Java	1.4 s	1.3.0_02
Smalltalk	4.5 s	GST 1.95.4
Python	22.6 s	2.0
Ruby	26.5 s	1.6.3
Perl	38.0 s	5.6.0
CLisp	44.6 s	2000-03-06
PHP	52.6 s	4.0.4pl1
AWK	92.0 s	GNU 3.0.6
Scheme	241.1 s	UMB Scheme 3.2
tcl	440.5 s	8.3

Da die Ladezeit der verschieden Sprachen, wie z.B. der JVM erheblich ist, lassen wir noch einmal die gleichen Benchmarks laufen, diesmal jedoch mit einer Schleife, so dass alles 20 mal wiederholt wird. Assembler, AWK, tcl, etc. werden nicht mehr wiederholt.

Rekursiv (20 x)

Sprache:	Laufzeit
C	6.2 s
Java	7.3 s
Smalltalk	1 m 47 s
Python	7 m 54 s
CLisp	12 m 18 s
Perl	13 m 12 s
Ruby	15 m 45 s
PHP	18 m 45 s
Scheme	23 m 40 s
tcl	...

Iterativ (1000 x)

Sprache:	Laufzeit
C	0.01 s
Perl	0.2 s
Smalltalk	0.2 s
Python	0.4 s
Ruby	0.5 s
CLisp	0.7 s
PHP	0.9 s
Java	1.3 s
Scheme	1.7 s
tcl	3.7 s

In einem weiteren Test (Worthäufigkeiten zählen) untersuchen wir die Implementation von Hash-Tabellen in den verwendeten Sprachen. Als Eingabetext wurde das 170KB lange “The Prince” von Nicoló Machiavelli verwendet, welcher 20 mal wiederholt wurde, i.e. eine Datei von 20*176KB=4.2MB Länge.

Sprache	Laufzeit
C	3.7
Perl	8.7
Java	10.8
Python	23.3
Bash	23.3
Ruby	29.3
tcl	64.6

Wir führen das relativ schlechte Abschneiden von Ruby in diesem Test auf eine nicht optimierte Implementation zurück sowie darauf, dass die Perl-Lösung nicht objektorientiert war, und deswegen hier automatisch Laufzeitvorteile hat.

Man wählt Ruby ist sicherlich nicht auf Grund der Geschwindigkeit des ausführbaren Codes, sondern wegen der Einfachheit und Eleganz der Lösungen. Bei Bedarf können rechenintensive "Kleinteile" in C optimiert werden. Aber auch ohne Erweiterungen spielt Ruby in der gleichen Liga wie Python.

Ruby-Schmankerl

Die folgenden Appetithappen sollen einen Eindruck davon vermitteln, wie sich Ruby-Code "anfühlt". Sie stammen aus unserem Buch "Programmieren mit Ruby", das wir mir Clemens Wyss für den dpunkt.verlag geschrieben haben.

- Obligatorisches Kultprogramm (Unix-Syntax):

```
ruby -e 'puts "Hallo zukuenftiger Rubyist!"'
```

- Mehrfachzuweisungen für Vertauschungen:

```
x, y = 2, 3        # x == 2 und y == 3
x, y = y, x        # x == 3 und y == 2
```

- Primzahlen finden; der Algorithmus ist langsam, aber kurz.

```
for i in 2..100 do
  puts i unless (2..i-1).find{|j| i % j == 0}
end      #-> 2 3 5 7 11 13 17 ... 83 89 97
```

- Nette Einzeiler

```
ruby -e 'print readlines.length'

ruby -i.bak -pe 'gsub "foo", "bar"' *.[ch]

ruby -pi -e 'printf "%d\t", $.' thgttg.txt
```

- Arrays und Hashtabellen haben in Ruby dynamische Größen und können Elemente verschiedener Typen aufnehmen.

```
array = [1, 'Alligator', 643.001]
pcSpeedHash = { 'Duck2'=>750,
  'Rockaplan'=>25, 'ZX81'=>0.6 }

pcSpeedHash['Ananke']=1700
```

- Iterieren über Aufzählungen aller Art.

```
(5..10).each { |z| puts z**3 }
  #-> 125 216 343 512 729 1000

pcSpeedHash.sort.map{ |key, val|
  "#{key}: #{val} MHz" }
#-> ['Ananke: 1700 Mhz', 'Duck2: 750 MHz',
  'Rockaplan: 25 MHz', 'ZX81: 0.6 MHz']
```

- Worthäufigkeiten zählen.

```
freq = Hash.new(0)
open("dings.rb").read.scan(/\w+/){ |wort|
  freq[wort] += 1 }
freq.sort.each { |wort, zahl|
  puts "#{wort} - #{zahl}" }
```

- Reguläre Ausdrücke in ganzer Pracht. Wir hatten eine Liste von Zitaten in der Form "Autor: Zitat" und erzeugen daraus die LaTeX-Anweisungen, die wir im Buch verwenden.

```
readlines.each{ | line |
  puts line.gsub('(.*?): (.*)',
    '\\begin{rquote}{\1}\2\\end{rquote}')
}
```

- "Die Stringmethode hatte doch ein 'case' im Namen …"

```
"".methods.grep(/case/)
  #-> ["upcase!", "upcase", "swapcase!", ...]
```

…"und wie hieß gleich noch mal die Error-Klasse?"

```
ObjectSpace.each_object(Class) { |c|
  puts c if c.to_s[/error/i]
} #-> SystemStackError, LocalJumpError,
    EOFError, IOError, RegexpError, ...
```

- Ruby ist eine dynamische Sprache!

  Methoden und Variablen können während der Laufzeit hinzugefügt und geändert werden, sogar für einzelne Objekte:

```
class A
  def method_missing(methId)
    print methId.id2name, " gibt es nicht.\n"
  end
end
a=A.new
a.hello          #-> hello gibt es nicht.

# diese Singleton-Methode gibt es nur für a
def a.hello
  p "hier schon"
end
a.hello          #-> hier schon
A.new.hello      #-> hello gibt es nicht.
```

- Quicksort von Tony Hoare in Ruby:

```
def quicksort( xs )
   return xs if xs.size <= 1
   m = xs[0] # Split-Element
   quicksort(xs.select { |i| i < m } ) +
 xs.select { |i| i == m } +
 quicksort(xs.select { |i| i > m } )
end
quicksort([13, 11, 74, 69, 0])
#-> [0, 11, 13, 69, 74]
```

- Netzwerkzugriff über Standardprotokolle, z. B. per ftp:

```
require 'net/ftp'
ftp = Net::FTP.open('ftp.netlab.co.jp')
ftp.login
ftp.chdir('pub/lang/ruby')
puts ftp.dir
ftp.quit
```

- Threads stehen ebenfalls zur Verfügung, um Aufgaben parallel zu erledigen. Dieses Beispiel holt zwei Webseiten gleichzeitig:

```
require 'net/http'
url1="www.approximity.com"
url2="www.ruby.ch"

def getPage(url)
  h = Net::HTTP.new(url, 80)
  puts "Hole: #{url}"
```

```
  resp, data = h.get('/', nil )
  puts "Habe #{url}:  #{resp.message}"
end

thread1=Thread.new { getPage(url1) }
thread2=Thread.new { getPage(url2) }

thread1.join
thread2.join
```

- Mit dem so genannten Marshalling können in Ruby fast alle Objekte in Binärstreams/Strings transformiert werden. Diese werden dann in Instanzen der Klasse String gespeichert.

```
include Marshal
a = 25.6;
pt = Struct.new('Point', :x,:y);
x = pt.new(10, 10)
y = pt.new(20, 20)
rt = Struct.new('Rectangle', :origin,:corner);
z = rt.new(x, y)
c = Object.new
thing = [a, x, z, c, c, "fff"];
representation = dump(thing);
p thing
p representation
p load(representation)
```

- Wenn man sich wirklich, wirklich Mühe gibt, kann man auch kryptisches Ruby erzeugen. Das folgende Beispiel sollte man in einer Zeile eingeben:

```
# ruby.rb
(n=$*[0].to_i).times{|i|s=(n/2-i).abs;
puts" "*s+"#"*(n-2*s)}

# Konsole:
> ruby ruby.rb 10
```

- Client/Server-Kommunikation ist mit Distributed Ruby kein Problem. Ein einfacher Server ist schnell geschrieben:

```
# server.rb
require 'drb'
class TestServer
  def doit; "Hallo, verteilte Welt"; end
end
```

```
ts = TestServer.new
DRb.start_service(
  'druby://approximity.com:9000', ts)
DRb.thread.join

# client.rb
require 'drb'
DRb.start_service
cl = DRbObject.new(nil,
  'druby://approximity.com:9000')
p cl.doit
```

Wir haben den TestServer auf approximity.com eingerichtet, so dass nach dem Start des Clients der Text "Hallo, verteilte Welt" erscheinen sollte.

- So genannte Closures binden die Umgebung, in der ihre Definition ausgeführt wird.

```
def addierer(incr); proc { |n| n + incr }; end
ad = addierer(3)
p ad.call(5)      # => 8
```

- Außer "Hallo Welt" scheint es noch ein Programm [Bottles] zu geben, das anscheinend in jeder Sprache programmiert worden ist:

```
99.downto(0) { |x|
  u = "#{x > 0 ? (x) : 'No more'}" +
" bottle#{x != 1 ? 's' : ''} of beer"
  w = u + " on the wall, "
  puts w + u +
". Take one down, pass it around. " + w
}
```

Man kann die Flaschen auch objektorientiert leeren:

```
class Wall
  def initialize(num); @beers = num; end
  def beer?; @beers > 0; end
  def takeBottle; @beers -= 1; end
  def bottles
    @beers == 1 ? "bottle" : "bottles"
  end

  def count
    @beers > 0 ? @beers.to_s : "No more"
  end
```

```
  def bob
    "#{count} #{bottles} of beer"
  end
end

wall = Wall.new 99
while wall.beer?
  print "#{wall.bob} on the wall, " <<
    "#{wall.bob}. Take one down, "
  wall.takeBottle
  puts "pass it around. #{wall.bob} on the wall."
end
```

Literatur

[M01] Yukihiro Matsumoto, Ruby in a Nutshell, O'Reilly 2001

[Ruby] Ruby Homepage, http://www.ruby-lang.org

[RubyFAQ] Ruby FAQ, http://www.rubycentral.com/faq/

[DrDobbs] Andrew Hunt, Dave Thomas, Programming in Ruby, Dr. Dobb's Journal, January 2001

[LarryWall] Larry Wall interview http://lwn.net/2001/features/LarryWall/

[RubyCH] Online Ruby Interpreter, http://www.ruby.ch

[Fibonacci] http://www.lionking.org/ cubbi/serious/fibonacci.html

[Register] http://www.theregister.co.uk/content/4/20703.html

[Bottles] 99 Bottles of Beer, http://internet.ls-la.net/mirrors/99bottles/

[SWIG] Simplified Wrapper and Interface Generator (SWIG), http://www.swig.org

[modruby] modruby, http://www.modruby.net/

[TH01] David Thomas, Andrew Hunt, Programming Ruby, Addison-Wesley, 2001

[RSW02] Armin Röhrl, Stefan Schmiedl, Clemens Wyss, Programmieren mit Ruby, dpunkt-Verlag, 2002

[Rubyunit] http://homepage1.nifty.com /markey/ruby/rubyunit/index_e.html