Der Tokenizer

```ruby
class Document
  attr_accessor :text, :token_list

  def initialize(doc)
     @text = doc
     @token_list = doc.split
     @bekannte_abkuerzungen = []
     @veraendert = nil          # Diese boolesche Variable wird
     # verwendet um zu überprüfen, ob ein oder mehrere Token
     # noch am Ende einen Punkt enthalten, die durch die
     # Punktdisambiguierung noch zu behandeln sind.

end

def to_s
  puts @text[0..70] + "..."
  puts @token_list[0..9] << "..."
end
```

Der Tokenizer

```
def tokenisiere (token)
  if (token.length > 1) #and (token =~ /[a-zA-Z][a-zA-Z]/)
    if ['(',"\'","\""].include?(token[0])
      [token[0]] + tokenisiere(token[1..-1])
    elsif [")","\'","\"",":",";",",","!","?"].include?(token[-1])
      tokenisiere(token[0...-1]) + [token[-1]]
    else
     @veraendert = true if token[-1] == '.'
      [token]
    end
  else
    [token]
  end
end
```

Der Tokenizer

… weitere Schritte …

Wir benötigen eine Methode, die jedes Token des Dokuments tokenisiert und die alte Tokenliste (@token_list) entsprechend aktualisiert:

```
def tokenisiere_dokument
   …
end
```

Die Tokenisierung eines Dokuments kann als ein Prozeß angesehen werden, bei dem zunächst tokenfinale Punkte analysiert und gegebenenfalls abgetrennt werden (klassifiziere_punkte); anschließend werden die Ausdrücke erneut tokenisiert (tokenisiere_dokument). Diese beiden Schritte werden solange wiederholt, bis keines der Token weiter verändert wird.

Der Tokenizer

… weitere Schritte …

Wir benötigen eine Methode, die jedes Token des Dokuments tokenisiert und die alte Tokenliste (@token_list) entsprechend aktualisiert:

```
def tokenisiere_dokument
   …
end
```

```
def tokenisiere_dokument
  neue_liste = []
  @token_list.each do |x|
    neue_liste += tokenisiere(x)
  end
  @token_list = neue_liste
end
```

Die Tokenisierung eines Dokuments kann als ein Prozeß angesehen werden, bei dem zunächst tokenfinale Punkte analysiert und gegebenenfalls abgetrennt werden (klassifiziere_punkte); anschließend werden die Ausdrücke erneut tokenisiert (tokenisiere_dokument). Diese beiden Schritte werden solange wiederholt, bis keines der Token weiter verändert wird.

Der Tokenizer

… weitere Schritte …

```
def tokenizer
  …
end
```

Der Tokenizer

… weitere Schritte …

```
def tokenizer
  …
end
```

```
def tokenizer
  begin
    @veraendert = nil
    klassifiziere_punkte
    tokenisiere_dokument
  end while @veraendert
  @token_list
end
```

Der Tokenizer

… weitere Schritte …

```
def tokenizer
  …
end
```

```
def tokenizer
  begin
    @veraendert = nil
    klassifiziere_punkte
    tokenisiere_dokument
  end while @veraendert
  @token_list
end
```

Für die Punktklassifikation definieren wir zunächst eine Reihe kleinerer Methoden:

Der Tokenizer

… weitere Schritte …

```
def tokenizer
  …
end
```

```
def tokenizer
  begin
    @veraendert = nil
    klassifiziere_punkte
    tokenisiere_dokument
  end while @veraendert
  @token_list
end
```

Für die Punktklassifikation definieren wir zunächst eine Reihe kleinerer Methoden:

```
def wort_normalerweise_klein?(token)
  tok_up = @token_list.count(token)
  tok_dwn = @token_list.count(token.downcase)
  if tok_up > 0
    tok_dwn / tok_up >= 0.5
  end
end
```

Der Tokenizer

… weitere Schritte …

```
def endet_mit_abkuerzungspunkt? token
  token.count('.') * (1 / (2.7182818 ** token.length)) *
   (1 / token.length ** @token_list.count(token[0...-1]).to_f)
   > 0.03
end

def klassifiziere_punkte
  neue_liste = []
  @token_list.each_with_index { |x,i | if (x[-1] == "."
                                        and (x.length > 1));
                neue_liste += klass_pkt(x, @token_list[i+1])
                else neue_liste << x end}
  @token_list = neue_liste
end
```

Der Tokenizer

… weitere Schritte …

```
def klass_pkt(tk1, tk2)
  if tk2
    if (tk2 =~ /^[a-z]/) || (@bekannte_abkuerzungen.include?(tk1))
      [tk1]
    elsif wort_normalerweise_klein?(tk2)
       tokenisiere(tk1[0...-1]) << tk1[-1]
    elsif endet_mit_abkuerzungspunkt? tk1
      [tk1]
      # Das nächste Token ist ein Sonderzeichen
    elsif ['(',')',',',';',':','-'].include?(tk1)
      [tk1]
    else
      # [tk1[0...-1], tk1[-1]]
      tokenisiere(tk1[0...-1]) << tk1[-1]
    end
  else
    # [tk1[0...-1], tk1[-1]]
    tokenisiere(tk1[0...-1]) << tk1[-1]
  end
end
```

Skriptsprachen
* Ruby *

Vererbung, Module und Mixins

Naumann
Sommersemester 2016

5.1 Vererbung

Programme sollten klar strukturiert sein und es weitestgehend vermeiden, Kode unnötig zu duplizieren. Eine Möglichkeit, diese Ziele zureichen, besteht darin, die in objekt-orientierten Programmiersprachen zur Verfügung stehenden Vererbungsmechanismen zu nutzen. Ruby bietet mit sogenannten ***MIxins*** einen zweiten Mechanismus zur Erreichung dieser Ziele. Auf beide Möglichkeiten werden wir im folgenden ausführlich eingehen.

Vererbung

Wir wissen, das `puts` sein Argument, ein Objekt, in einen String konvertiert und die `to_s`-Methode dieses Objektes aufruft. Dieses Verhalten funktioniert dank der in Ruby realisierten Vererbungsmechanismen auch für Objekte selbstdefinierter Klassen, für die wir nicht explizit eine `to_s`-Methode definiert haben: Diese selbstdefinierten Klassen sind Spezialisierungen (*Subklassen*) anderer selbstdefinierter Klassen oder System-klassen.

5.1 Vererbung

Um diesen Punkt noch einmal zu illustrieren, betrachten wir einen ganz einfachen Fall. Wir definieren zwei Klassen, so dass die zweite Klasse eine *Subklasse* der ersten ist und definieren in der ersten, der *Superklasse* der zweiten, eine Methode say_hello. Diese Methode kann sowohl auf Instanzen der Superklasse wie der Subklasse angewendet werden:

```
class Parent
  def say_hello
    puts "Hello from #{self}"
  end
end

p = Parent.new
p.say_hello                 # => Hello from #<Parent: 0x0a40c4>

# Subclass the parent...
class Child < Parent
end

c = Child.new
c.say_hello                 # => Hello from #<Child: 0x0a3d68>
```

5.1 Vererbung

Die superclass-Methode liefert die Superklasse einer Klasse:

```
class Parent
end
class Child < Parent
end
puts "The superclass of Child is #{Child.superclass}"
# => The superclass of Child is Parent

puts "The superclass of Parent is #{Parent.superclass}"
# => The superclass of Parent is Object

puts "The superclass of Object is #{Object.superclass}"
# => The superclass of Object is BasicObject

puts "The superclass of BasicObject is #{BasicObject.superclass}"
# => The superclass of BasicObject is nil
```

5.1 Vererbung

Die to_s-Methode wird in der Klasse **Object** definiert. Deshalb ist diese Methode für alle Objekte in Ruby verfügbar; denn jede Klasse (mit Ausnahme von **BasicObject**) ist eine Subklasse von **Object**.
Wird in der Klasse eines Objekts keine Spezialisierung von to_s definiert ist, sucht Ruby in der Superklasse der Klasse nach einer to_s-Methode, dann in deren Superklasse, etc.

```
class Person
  def initialize(name)
    @name = name
  end
end

p = Person.new("Michael")
puts p                              # => #<Person:0x0a4efc>
```

5.1 Vererbung

```
class Person
  def initialize(name)
    @name = name
  end
  def to_s
    "Person named #{@name}"
  end
end


p = Person.new("Michael")
puts p                          # => Person named Michael
```

Die in **Person** definierte to_s-Methode sorgt für eine ansprechende Formatierung des Objekts.

5.1 Vererbung

Es ist durchaus üblich, für Anwendungen Systemklassen zu spezialisieren, um eine bestimmte Funktionalität zu realisieren.

Die Systembibliothek **GServer** z.B. implementiert die grundlegende Funktionalität für TCP-Server. Diese Grundfunktionalität kann durch Spezialisierung nach Bedarf erweitert werden.
Wenn man eine Instanz dieser Klasse erzeugt, gibt als Argument die ID des Ports an, auf dem der konkrete Server arbeitet.

Die **GServer**-Klasse stellt u.a. die Methode serve zur Verfügung, die aufgerufen wird, sobald sich ein Client am Server angemeldet. Diese Methode verwaltet dann die Verbindung zwischen dem Server und dem Clienten:

```
def serve(io)
end
```

Wie man sieht, leistet die Methode selbst nichts. Konkrete Funktionalität muss durch geeignete Subklassen festgeleget werden. Unsere Beispielklasse nennen wir aus naheliegenden Gründen **LogServer**.

5.1 Vererbung

LogServer ist eine Subklasse von **GServer**.

Durch super wird die gleichnamige Methode der Superklasse aufgerufen; d.g. ein Log-Server lauscht immer auf dem Port 12345.

Die serve-Methode spezialisiert die leere Methode der Superklasse: Es werden immer die letzten 1000 Zeichen der Log-Datei ausgegeben.

```
require 'gserver'
class LogServer < GServer
  def initialize
    super(12345)
  end
  def serve(client)
    client.puts get_end_of_log_file
  end
private
  def get_end_of_log_file
    File.open("/var/log/system.log") do |log|
      log.seek(-1000, IO::SEEK_END)   # back up 1000 characters from end
      log.gets                        # ignore partial line
      log.read                        # and return rest
    end
  end
end

server = LogServer.new
server.start.join
```

5.1 Vererbung

```
$ telnet 127.0.0.1 12345
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Jul 7 13:39:44 dave com.apple.syncservices.SyncServer[54938]:...
Jul 7 13:39:44 dave com.apple.syncservices.SyncServer[54938]:...
Jul 7 13:42:40 dave login[54768]: DEAD_PROCESS: 54768 ttys001
Jul 7 13:45:34 dave mdworker[54977]: fcntl to turn on F_CHECK...
Jul 7 13:48:44 dave mdworker[54977]: fcntl to turn on F_CHECK...
Connection closed by foreign host.
```

5.2 Module

Module ermöglichen es, *Methoden*, *Klassen* und *Konstanten* zusammenzufassen. Ihre größten Vorzüge liegen darin, dass sie
✦ Namenräume bereitstellen und so helfen, Namenskonflikte zu vermeiden und
✦ sogenannte ***Mixins*** unterstützen.

Allerdings gibt es keine Vererbungshierarchie von Modulen.

Namensräume

Bei der Entwicklung komplexer Programme, die oft in Teamarbeit erstellt werden, ist es notwendig über einen Mechanismus der es verhindert, dass zwei gleichnamige Methoden mit unterschiedlicher Funktionalität definiert werden, die dann zu unerwarteten Konsequenzen führen.

Ein Namensraum fasst eine Menge von Bezeichnern/Symbolen zusammen. Um eine Methode A aus dem Namensraum X auszuführen, wird der Namensraumbezeichner X als Präfix verwendet:

X.A

5.2 Module

```
module Trig
  PI = 3.141592654
  def Trig.sin(x)
    # ..
  end
  def Trig.cos(x)
    # ..
  end
end
```

```
module Moral
  VERY_BAD = 0
  BAD      = 1
  def Moral.sin(badness)
    # ...
  end
end
```

Die Referenz auf eine Modul-Methode, -Variable oder -Konstante *X* kann durch *Modulname::X* oder *Modulname.X* erfolgen:

```
require 'trig'
require 'moral'
y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

Wie das Beispiel zeigt, werden Modulmethoden wie Klassenmethoden aufgerufen:

ModulName.MethodenName

5.3 Mixins

Mixins sind ein partieller Ersatz für die Möglichkeiten, die in bestimmten Programmiersprachen durch Mehrfachvererbung (mehr als eine Superklasse für jede Klasse möglich) gegeben sind.

Wenn ein Modul statt Klassenmethoden Instanzenmethoden definiert, dann können diese Instanzenmethoden in andere Klassen eingefügt bzw. ‚*hineingemischt*' werden. Ein solches eingefügtes Modul verhält sich dann ähnlich wie eine Superklasse.

Zum Einfügen eines Modules wird include verwendet: include Modulname
Allerdings: Wenn das Module in einer anderen Datei definiert ist, dann muss zunächst per require diese Datei geladen werden, bevor das Modul in eine oder mehrere Klassen inkorporiert werden kann.

Module lassen sich mehrstufig in andere Module oder Klassen übernehmen.

```
module D
    def initialize(name)
     @name =name
    end
    def to_s
      @name
    end
  end
```

5.3 Mixins

Durch die Übernahme des Debug-Modules können beide Klassen auf die Methode *who_am_i?* zugreifen.

Dieses Modul schließ das Modul D mit ein, dessen Initialisierungsmethode damit auch allen Objekten beider Klassen zur Verfügung steht.

```
module Debug
  include D
    def who_am_i?
      "#{self.class.name} (\##{self.object_id}): #{self.to_s}"
    end
end

class Phonograph
    include Debug
    # ...
end

class EightTrack
    include Debug
    # ...
end

ph = Phonograph.new("West End Blues")
et = EightTrack.new("Surrealistic Pillow")

ph.who_am_i?
# => “Phonograph (#330450): West End Blues“
et.who_am_i?
# => “EightTrack (#330420): Surrealistic Pillow “
```

5.3 Mixins

Durch *Mixins* lässt sich auf sehr einfache Weise die Funktionalität von Klassen erweitern:
Das System-Mixin **Comparable** definiert die Vergleichsoperatoren <, <=, ==, >= und > sowie die Methode between?. Sie alle werden auf Grundlage des <=>-Operators definiert. Wenn man für eine eigene Klasse diesen Operator definiert und anschließend mit include um das Mixin **Comparable** erweitert, erhält man für seine Klasse diese sechs Methoden *umsonst*.

```
class Person
  include Comparable
  attr_reader :name
  def initialize(name)
    @name = name
  end
  def to_s
    "#{@name}"
  end
  def <=>(other)
    self.name <=> other.name
  end
end
```

5.3 Mixins

```
p1 = Person.new("Matz")
p2 = Person.new("Guido")
p3 = Person.new("Larry")

# Vergleiche eine Reihe von Namem
if p1 > p2
  puts "#{p1.name}'s name > #{p2.name}'s name"
end

# Sortiere einen Array von Personen-Objekten

puts "Sorted list:"
puts [ p1, p2, p3].sort
```

Sorted list:
Guido
Larry
Matz

5.4 Putting it all together

Die Klassen **Array** und **Hash** liefern viele nützliche Methoden, um Gruppen von Objekten effizient zu verwalten (inject, map, include?, ...). Um diese Methoden auch in einer selbstdefinierten Klasse nützen zu können, reicht es aus, für diese Klasse eine geeignete each-Methode zu definieren und das Modul **Emumerable** zu inkorporieren.

```
[ 1, 2, 3, 4, 5 ].inject(:+)        # => 15
( 'a'..'m').inject(:+)              # => "abcdefghijklm"

class VowelFinder
    include Enumerable
    def initialize(string)
        @string = string
    end
    def each
        @string.scan(/[aeiou]/) do |vowel|
            yield vowel
        end
    end
end

vf = VowelFinder.new("the quick brown fox jumped")
vf.inject(:+)                       # => "euiooue"
```

5.4 Putting it all together

```
class VowelFinder
  include Enumerable

  def initialize(string)
    @string = string
  end

  def each
    @string.scan(/[aeiou]/) do |vowel|
      yield vowel
    end
  end
end
```

```
module Summable
  def sum
    inject(:+)
  end
end

class Array
  include Summable
end

class Range
  include Summable
end

class VowelFinder
  include Summable
end
```

```
[ 1, 2, 3, 4, 5 ].sum                  # => 15
('a'..'m').sum                         # => “abcdefghiklm“

vf = VowelFinder.new("the quick brown fox jumped")
vf.sum                                 # => “euiooue“
```

5.4 Putting it all together

Instanzenvariablen in *Mixins*

Inkorporierte Module können, wie wir bereits gesehen haben, neue Instanzenvariablen für die Objekte der Klasse anlegen, in die sie eingebettet werden.

In dem Beispiel rechts wird eine Instanzen-variable @observer_list erzeugt (‚||=‘ ist eine bedingte Zuweisung; d.h. die Variable erhält nur den rechts stehenden Wert, wenn sie nicht bereits mit einem anderen Wert initialisiert wurde).

Dieses Vorgehen birgt natürlich auch Risiken:

```
module Observable

    def observers
        @observer_list ||= []
    end

    def add_observer(obj)
        observers << obj
    end

    def notify_observers
        observers.each {|o| o.update }
    end
end
```

5.4 Putting it all together

Instanzenvariablen in *Mixins*

So kann es z.B. zu Namenskonflikten mit in der Klasse selbst definierten Instanzenvariablen kommen:

```
class TelescopeScheduler
    # other classes can register to get notifications
    # when the schedule changes
    include Observable


    def initialize
        @observer_list = [] # folks with telescope time
    end
    def add_viewer(viewer)
        @observer_list << viewer
    end
    # ...
end
```

5.4 Putting it all together

Normalerweise verwenden die *Mixin*-Module keine eigenen Instanzenvariablen. Anderenfalls ist darauf zu achten, dass es nicht zu Namenskonflikten der beschriebenen Art kommt. Statt der Verwendung von Instanzenvariablen ist es oft auch möglich, die notwendigen Informationen in einem Hash im Modul zu speichern:

```
module Test
  State = {}
  def state=(value)
    State[object_id] = value
  end
  def state
    State[object_id]
  end
end
```

```
class Client
  include Test
end

c1 = Client.new
c2 = Client.new
c1.state = 'cat'
c2.state = 'dog'

c1.state
c2.state
```

Rubys Strategie zur Lösung von Namenskonflikten bei Methodenaufrufen

Klasse des Objekts < Mixins der Klasse < Superklasse < Mixins der Superklasse < ...