

SKRIPTSPRACHEN

* RUBY *

VERERBUNG, MODULE
UND

NAUMANN
SOMMERSEMESTER 201

5.1 VERERBUNG

- ◆ Wie auch sonst im Leben, sollte man sich auch bei der Entwicklung von Software darum bemühen, den zur Lösung von Aufgaben erforderlichen Aufwand zu minimieren. Anders formuliert: Das unnötige Duplizieren von Code wie Funktionalität sollte vermieden werden.
- ◆ Die Verwendung von Klassen kann als ein Schritt in diese Richtung interpretiert werden: Alle Methoden der Klasse stehen allen Instanzen, also allen Objekten dieser Klasse zur Verfügung.
- ◆ Wie auch in anderen objektorientierten Sprachen bilden die Systemklassen und die benutzerdefinierten Klassen eine Hierarchie, die die Distribution (Vererbung) von Informationen steuert (Methoden und Instanzvariablen).
- ◆ Ein anderer Mechanismus, der in vielen Fällen flexiblere Lösungen ermöglicht, bilden die so genannten *Mixins*.

5.1 VERERBUNG

Wir wissen, dass `puts` einen String erwartet und wenn für sein Argument eine `to_s`-Methode definiert ist, diese Methode aufruft. Wir wissen außerdem, dass wir für selbstdefinierte Klassen eine geeignete `to_s`-Methode definieren können.

Vererbung / Superklasse / Subklasse

```
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

```
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.

Die **GServer**-Klasse stellt u.a. die Methode `serve` zur Verfügung, die aufgerufen wird, sobald sich ein Client am Server angemeldet hat:

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

Wie man sieht, leistet die Methode selbst nichts. Konkrete Funktionalität muss durch geeignete Subklassen festgelegt 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 13:39:44 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
```

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


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`

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 schließt 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 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 Namen
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 (`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 **Enumerable** 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             # => "abcdefghijklm"
```

```
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 Instanzenvariable `@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 < ...