

Detour: Automatisiertes Testen

Grundlagen

**Gerne alles unbekannte ignorieren,
wir kommen später drauf zurück**

Ein Test

```
require 'minitest/autorun'

def add(a, b=5)

class SongTest < Minitest::Test
  def test_add_with_one_argument_adds_five
    assert_equal 10, add(5)
  end

  def test_add_with_two_arguments_adds_arguments
    assert_equal 15, add(5, 15)
  end
end
end
```

```
$ ruby example_test.rb  
Run options: --seed 37288
```

```
# Running:
```

```
..
```

```
Finished in 0.001929s, 1036.9158 runs/s, 1036.9158 assertions/s.
```

```
2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

Test Driven Development

1. Red
2. Green
3. Refactor

Red

Schreibe den kleinsten Test für das kleinste Stück funktionalität

Green

Schreibe schnell den einfachsten Code, der den kaputten test Grün macht

Refactor

Räume etwas auf (Entweder im Anwendungs- oder im Testcode), ohne dass die tests kaputt gehen

Gehe zu Schritt 1



Fizz Buzz

- Methode die von 1 bis n (Argument) hochzählt
- Wenn zahl durch 3 teilbar, ersetze Zahl durch "fizz"
- Wenn Zahl durch 5 teilbar, ersetze Zahl durch "buzz"
- Wenn Zahl durch 15 teilbar, ersetze Zahl durch "fizzbuzz"

Alle zusammen ("Swarm!")

- Ein*r fährt
- Alle dürfen reinquatschen
- Kleine, fokussierte Änderungen

Reguläre Ausdrücke

- Formaler Weg um Muster in Zeichenketten zu beschreiben
- Mit einem Muster kann man suchen, validieren, parsen und und und...
- Starker theoretischer Background
- Viel Verwendung in Perl und auf der Shell
- In Ruby Objekte

Einfache Ausdrücke

```
/a/  
/hallo/  
regexp = /test/  
a = "hal"  
/#{a}lo/ # /hallo/
```

Pattern matching

```
"hallo" =~ /a/      # => 1  
"hallo" =~ /l/      # => 2  
"hallo" =~ /test/   # => nil  
"hallo" !~ /p/      # => true
```

```
if "hallo" =~ /allo/  
  puts "Selber Hallo"  
end
```

Objektorientiert

```
regexp = /l/  
match_data = regexp.match("hallo")  
match_data.size      # 1  
match_data.pre_match  # "ha"  
match_data.post_match # "lo"  
/x/.match("hallo")    # nil
```

Zeichengruppen

- Vokale: `[aeiou]`
- Großbuchstaben: `[A-Z]` (Falle: Umlaute!)
- Keine Kleinbuchstaben: `[^a-z]`
- Ziffern: `[0-9]`
- l,m,n,o: `[l-mno]`
- Beliebiges Zeichen: `.`
- `/h.llo/` matcht auf `"hallo"`, `"hello"` usw.

Quantifizierer

- Ein oder keinmal: `?`
- Beliebiger oft (auch keinmal): `*`
- Beliebiger oft, aber mindestens einmal: `+`
- Genau 5 mal: `{5}`
- 5 mal oder öfter: `{5,}`
- 5-10 mal: `{5,10}`

Beispiel

Variablennamen in Ruby, Versuch 1:

```
/[_a-z][a-zA-Z0-9_]*/
```

Alternativen

eine Auswahl zwischen Alternativen wird mit **|** markiert

```
/[_a-z][a-zA-Z0-9_]*(\?|!)?/
```

Bereichsmarkierungen

- Zeilenanfang: `^`
- Zeilenende: `$`
- Stringanfang: `\A`
- Stringende: `\z`

Beispiel

Variablennamen in Ruby, Versuch 2:

```
/^[_a-z][a-zA-Z0-9\_]*$/
```

Variablennamen in Ruby, Versuch 3:

```
/\A[_a-z][a-zA-Z0-9\_]*\z/
```

Zeichenklassen

- `\d` entspricht `[0-9]`, `\D` `[^\d]`
- `\s` Whitespace
- `\w` gültige Zeichen in Bezeichnern (`[a-zA-Z0-9\_]`)
- `\b` Wortgrenze

Posix-Expressions

Mit Unicode-Unterstützung (Auswahl):

- `[[ :alpha: ]]` Buchstabe
- `[[ :digit: ]]` Ziffer
- `[[ :alnum: ]]` alphanumerisches Zeichen

Beispiel

Variablennamen in Ruby, Versuch 4:

```
/\A[_a-z]\w*\z/
```

Modifikatoren

Am Ende eines Ausdrucks
Mehrere Kombinierbar

- **i** steht für case-insensitive
- **m** steht für multiline
- **x** für das entfernen von Whitespace
- **o** dafür, Ersetzungen nur einmal vorzunehmen

Darüberhinaus weitere für Encoding

Gier / Greed

***** wird häufig benötigt, macht aber oft nicht, was man erwartet.

Beispiel HTML-Parsen: `/\<.*>/` findet nicht einzelne Tags, sondern das gesamte Dokument

***** ist greedy (gierig) und sucht den größtmöglichen Match

***?** sucht den kleinsten Match

Arbeiten mit Matches

Für geklammerte Ausdrücke werden gematchte Strings gespeichert ("Capture Groups")

```
"hallo" =~ /h(a)ll(o)/ # => 0  
$1 # => "a"  
$2 # => "o"
```

Objektorientiert

```
date_exp = /(\d{1,2})\. (\w+) (\d{2,4})/  
date_string = "23. Januar 2014"  
match_data = date_exp.match(date_string)  
match_data.size      # => 4  
match_data[0]        # => "23. Januar 2014"  
match_data[1]        # => "23"  
match_data.captures  # ["23", "Januar", "2014"]
```

Benannte Gruppen

```
exp = /(?<day>\d{1,2}). (?<month>\w+) (?<year>\d{2,4})/  
string = "23. Januar 2014"  
match_data = exp.match(string)  
match_data[:day] # "23"  
match_data[:month] # "Januar"
```

Alle Fundstellen finden

```
date = "23.01.2014"  
date.scan(/\d+/)          # => ["23", "01", "2014"]  
date.scan(/(\d{2})\.(\d{2})/) # => [["23", "01"]]  
date.scan(/\d+/) { |match| puts match }
```

Suchen & Ersetzen

```
"hallo".sub(/l/, "test") # => "hatestlo"  
"hallo".gsub(/l/, "test") # => "hatesttesto"  
"Otto Mustermann".sub(/(\w+)\s+(\w+)/, '\2, \1') # "Mustermann, Otto"
```

Strings teilen

```
"1,2,3".split(",") # => ["1", "2", "3"]  
"1 2\t3\n4".split(/\s+/) # ["1", "2", "3", "4"]
```

Dokumentation und Spielwiese

- [API-Docs](#)
- [Rubular.com](#)



Taschenrechner, die Zweite

Verbessert das Taschenrechnerprogramm, indem Ihr die Eingabe des Benutzers in einem einzigen Schritt verlangt und mit Hilfe regulärer Ausdrücke auf ihre Gültigkeit prüft.

Hat der Benutzer nicht, wie gefordert, eine Zahl, bzw. einen der Operatoren **+**, **-**, ***** und **/** eingegeben, soll keine Berechnung erfolgen, sondern eine Fehlermeldung ausgegeben werden.

Umgebungsvariablen

Umgebungsvariablen landen in der Konstanten `ENV` verhält sich wie ein Hash:

```
ENV["HOME"] # "/home/nutzer02"
```

Shellout

```
result = `uname`    # => "Linux\n"  
system("uname -a")  # => true
```

Kommandozeilenargumente

- Alle Kommandozeilenargumente landen in `ARGV`
- `ARGV` ist ein Array
- `ARGV.size` nennt Anzahl der Parameter
- (Anders als in C ist `ARGV[0]` nicht der Programmname)
- `OptionParser` für komplexere Fälle (`--help`, `-v`, `--verbose`)

```
puts ARGV[0]
```



Kommandozeilen-Rechner

- Baut einen Taschenrechner, der als Argumente Zahl, operator und Zahl verlangt und das Ergebnis wieder ausgibt
- Lest Euch die Dokumentation zu `OptionParser` durch und baut den Taschenrechner so um, dass er nur die Zahlen als freie Argumente bekommt, der Operator aber über ein Flag übergeben wird (`--operator, -o`)

Objektorientierte Programmierung

OO

- Stammt u.a. von Smalltalk, entwickelt im Xerox PARC
- Wird verwendet in C++, Java, Ruby, Python(, Perl, PHP ...)
- Technische Vorteile, aber auch besser an den Menschen angepasst

Was bedeutet Objektorientierung

- Objekte als Funktionseinheiten
- Objekte Bündeln Daten (Attribute) und Funktionen (Methoden)
- Objekte kommunizieren über Nachrichten (Methodenaufrufe)
- Vererbung als Möglichkeit der Abstraktion

Ziele aus Entwicklersicht

Modellierung in Objekten entspricht eher dem menschlichen Denken (citation needed)

Ziele aus technischer Sicht

- Bessere Struktur
- Stärkere Kapselung
- Weniger Redundanz
- Wiederverwendbarkeit
- Oder zusammengefasst: Modularität

Wichtige Begriffe

- Klasse (Bauplan)
- Instanz (konkretes Objekt, das nach Bauplan der Klasse erstellt wurde)
- Instanzenvariable (Eigenschaft des Objekts)
- Instanzenmethode
- Klassenvariablen und Klassenmethoden

Klassen entwerfen

- Welche Objekte soll mein System enthalten?
- Was sind die Daten, wer sind die Akteure?
- Wie sind Zuständigkeiten verteilt?
- Was kann ich möglicherweise wiederverwenden?

Beispiel Musik-Player

- Klassen: Song, Artist, Album
- Song-Eigenschaften: Titel, Länge usw.
- Song-Verhaltensweisen: Abspielen, Pausieren usw.

Beispiel Büromöbel

- Klassen: Möbelstück, Stuhl, Tisch
- Möbelstück-Eigenschaften: Beschreibung, Hersteller usw.
- Stuhl-Eigenschaften: Sitzhöhe, Armlehnen usw.
- Tisch-Eigenschaften: Länge, Breite, Höhe usw.
- Stuhl und Tisch sind beides Möbelstücke und teilen sich die gemeinsamen Eigenschaften

Klassen in Ruby

```
class Song  
  # [...]  
end  
  
s = Song.new
```

Klassen in Ruby

- In einer .rb-Datei darf man beliebig viele Klassen definieren
- Best practice: Bei größeren Programmen eine Klasse pro Datei
- Code zwischen class und end wird einmalig ausgeführt (!)

Methoden

```
class Song
  def play
    puts "playing!"
  end
end
```

```
s = Song.new
s.play
```

this is self

- `self` ist im Instanz-Kontext die **Instanz**
- `self` ist im Klassen-Kontext die **Klasse**

Klassenmethoden

```
class Player  
  def self.play  
    puts "Playing..."  
  end  
end
```

```
Player.play
```

Konstruktor

```
class Song
  def initialize(title)
    end
end
s = Song.new("Piano Man")
```

Instanzvariablen

- `@instanzvariable`
- Repräsentiert internen Zustand, Zugriff von außen möglich, aber mit Absicht doof

Instanzvariablen

```
class Song
  def initialize(title)
    @title = title
  end

  def play
    puts "Now Playing: #{@title}"
  end
end
s = Song.new("Piano Man")
s.play
```

"Attribute"

```
class Song
  attr_accessor :title

  def initialize(title)
    @title = title
  end
end

s = Song.new("Piano Man")
s.title      # => "Piano Man"
s.title = "Piano Man 2"
```

"Attribute"

`attr_accessor`, `attr_reader`, `attr_writer`

```
class Song
  def title=(title)
    @title = title
  end

  def title
    @title
  end
end
```

Klassenvariablen

- `@@songs_played` ist eine Klassenvariable
- Existiert genau einmal pro Klasse (inkl. Unterklassen)
- Übergeordnetes Wissen: Ein einzelner Song kann und muss nicht wissen, welche Songs bisher schon gespielt wurden
- Selten wirklich notwendig



Praxis

Bücher

- Schreibt eine Klasse, die Bücher repräsentiert.
- Ein konkretes Buch soll folgende Eigenschaften haben: Titel, Autor, Seitenzahl, Preis
- Alle Eigenschaften sollen von außen les- und schreibbar sein.
- Außerdem sollen sie bereits beim instanziiieren von BuchObjekten übergeben werden.
- Die Klasse soll nachhalten, wieviele Bücher instanziiert wurden und eine Methode bieten, mit der die Anzahl bisher instanziiierter Bücher abgefragt werden kann.

Vererbung

- Unterschiedliche Objekte können Gemeinsamkeiten aufweisen, sowohl bei Eigenschaften als auch bei Verhaltensweisen
- Sind es gleichartige Objekte, kann man Vererbung nutzen
- Eine Oberklasse enthält alle Gemeinsamkeiten
- Unterklassen erben von dieser und implementieren nur noch die Unterschiede
- Vorsicht vor zu enger Kopplung
- In Ruby gibt es Alternativen

Vererbung in Ruby

```
class Square < Rectangle  
end
```

Vererbung in Ruby

- `Square` (Quadrat) erbt alle Eigenschaften von `Rectangle` (Rechteck) (Ein Quadrat ist ein Spezialfall eines Rechtecks)
- Das Überschreiben und Ergänzen von Methoden ist möglich
- Zugriff auf überschriebene Methoden mit `super`

Vererbung

```
class Rectangle
  def initialize(width, height)
    @width = width
    @height = height
  end
  def area ; @width * @height ; end
end

class Square < Rectangle
  def initialize(width)
    super(width, width)
  end
end
```

Sichtbarkeit

- `public` (implizit): immer aufrufbar
- `protected` : Nur aufrufbar, wenn Empfänger die selbe Klasse hat wie "self"
- `private` (Schlüsselwort) : Nur auf implizitem self aufrufbar (also nur innerhalb der selben Klasse)

Sichtbarkeit

```
class Visibility
  private

  def private_method
    "But I'm private!"
  end
end
```

Sichtbarkeit

```
object.private_method  
#=> visibility.rb:11:in '<main>': private method 'private_method' called for an instance of Visibility (NoMethodError)  
  
object.send(:private_method)  
#=> "But I'm private!"
```



Kaffeemaschine

Exceptions in Ruby

```
raise "Uff, alles kaputt"
```

Exceptions abfangen

```
begin
  things_that_can_go_wrong
rescue ThingsWentWrongError => e
  puts "Things went wrong: #{e}"
ensure
  puts "we'll always get here"
end
```

Eigene Exceptions

```
class ThingsWentWrongError < StandardError
  attr_reader :thing
  def initialize(thing)
    @thing = thing
  end
end

raise ThingsWentWrongError.new("A Thing that went wrong")
```

Sonderfall Methoden-Ebene

```
def cannot_go_wrong
  things_that_can_go_wrong
rescue ThingsWentWrongError => e
  puts "Dammit"
end
```

Detaillierte Fehlerbehandlung

```
begin
  things_that_can_go_wrong
rescue ThingsWentWrongError => e
  puts "Thing went wrong"
rescue IOError => e
  puts "Something different went wrong"
end
```

Retry / Re-Raise

```
begin
  goes_sometimes_wrong
rescue ThingsWentWrongError
  retry
rescue FatalError
  puts "Okay, Mist"
  raise
end
```

Zum Abschluss

Ruby-Golf

Code-Golf

- So wenig Zeichen wie möglich
- Lesbarkeit ist egal
- "Wie sehr kann ich die Syntax kaputt spielen"

FizzBuzz-Golf

```
scp  
nutzer02@192.168.1.202:ruby_schulung/test_driven_fizz_buzz/fizz_buzz.rb  
./
```

- Wie kurz bekommt Ihr das FizzBuzz-Programm
- Leerzeichen zählen
- Von der Basismethode (`def fizz_buzz()`) zählt nur der Methoden-Körper
- Alle Tests müssen grün sein
- Bis 17:45