Skriptsprachen
* Ruby *

... Klassen
&
Unit Testing

Naumann
Sommersemester 2016

3.4 Zugriffskontrolle

- Bei der Definition einer Klasse sollte immer genau überlegt werden, welche Eigenschaften der Objekte dieser Klasse von außen sichtbar oder gar beeinflussbar sein sollen.

- Die Schnittstelle zwischen der Klasse und der Außenwelt sollte so gestaltet werden, dass Benutzer nicht versucht sind, in ihren Programmen Details der spezifischen Implementierung einer Klasse zu nutzen.

- Ruby unterscheidet - was die Zugriffsmöglichkeiten auf die in einer Klasse definierten Methoden betrifft - drei Typen von Methoden:

  1. öffentliche Methoden (*public methods*)
  2. geschützte Methoden (*protected methods*) und
  3. private Methoden (*private methods*).

3.4 Zugriffskontrolle

```
class MyClass
  def method1 # 'public'
    #...
  end
  protected
  def method2 # 'protected'
    #...
  end
  private
  def method3 # 'private'
    #...
  end
  public
  def method4 # 'public'
    #...
  end
end
```

Der Typ einer Methode kann innerhalb der Klasse, in der sie definiert wird, durch Verwendung entsprechender Schlüsselwörter deklariert werden:

```
class MyClass

  def method1
  end

  def method2
  end

  # ... and so on

  public    :method1, :method4
  protected :method2
  private   :method3
end
```

3.4 Zugriffskontrolle

```
class Transaktion
    def initialize(konto_a, konto_b)
      @konto_a = konto_a
      @konto_b = konto_b
    end
  private
    def abbuchung(konto, betrag)
      konto.guthaben -= betrag
    end
    def gutschrift(konto, betrag)
      konto.guthaben += betrag
    end
  public
    def ueberweisung(betrag)
      abbuchung(@konto_a, betrag)
      gutschrift(@konto_b, betrag)
    end
end
```

```
class Konto
    attr_accessor :guthaben
    def initialize(betrag)
      @guthaben = betrag
    end
 end
```

```
konto1 = Konto.new(100)
konto2 = Konto.new(200)

trans = Transaktion.new(konto1, konto2)
trans.ueberweisung(50)
puts konto1.guthaben        # => 50
puts konto2.guthaben        # => 250
```

3.4 Zugriffskontrolle

Geschützte Methoden können anders als *private* Methoden auch auf andere Instanzen derselben Klasse zugreifen.

Angenommen, Konto-Objekte sollen in der Lage sein, ihren Kontostand mit dem eines anderen Konto-Objekts zu vergleichen, muss die entsprechende Methode als *geschützt* und nicht als *privat* deklariert werden.

```
class Konto
  attr_reader  :guthaben      # Zugriffsmethode 'guthaben'
  protected    :guthaben      # als protected markiert

  def initialize(betrag)
    @guthaben = betrag
  end

  def mehr_kohle_als(konto2)
    return guthaben > konto2.guthaben
  end
end
```

3.4 Zugriffskontrolle

```
konto1 = Konto.new(100)
konto2 = Konto.new(200)

puts konto1.mehr_kohle_als(konto2) #=> false

puts "Der Spion sagt: Du hast #{konto1.guthaben} Euro." # => file.rb:20:in
`<main>': protected method `guthaben' called for #<Konto:0x9645f74
@guthaben=100> (NoMethodError)
```

Da die Methode guthaben hier eine Methode vom Typ protected ist, kann sie auch verwendet werden, um auf die Instanzenvariablen anderer Objekte vom Typ Konto zuzugreifen (mehr_kohle_als). Ein expliziter Aufruf außerhalb der Klasse ist nicht möglich.

Wenn guthaben als private deklariert würde, könnte mehr_kohle_als nicht auf den internen Zustand einer anderen Instanz der Klasse Konto (konto2) zugreifen.

Unit Testing

- Unter ***unit testing*** versteht man eine Form der Softwareüberprüfung, bei der nicht komplette Softwaresysteme, sondern kleinere Einheiten, also einzelne Methoden oder Abschnitte von Methoden geprüft werden.

- Komplexe Systeme bestehen oft aus verschiedenen Schichten, wobei in der Regel jede Schicht voraussetzt, dass der Kode in der vorangegangenen Schicht korrekt ausgeführt wurde.

- Auf diese Weise ist es viel einfacher, Fehler zu lokalisieren und zu beheben: Die Details des geschriebenen Kodes sind noch vertraut (zeitliche Nähe zwischen Entwicklung & Testen) und der Bereich, der für den Fehler verantwortlich sein kann, ist eng begrenzt.

- Das Resultat sind besser geschriebene Programme, deren Entwicklung zudem weniger Zeit beansprucht als die konventionelle Form der Programmentwicklung.

Unit Testing

```
# Dieses Programm enthält Fehler:

class Roman
    MAX_ROMAN = 4999

    def initialize(value)
        if value <= 0 || value > MAX_ROMAN
            fail "Roman values must be > 0 and <= #{MAX_ROMAN}"
        end
        @value = value
    end

    FACTORS = [ ["m", 1000], ["cm", 900], ["d", 500], ["cd", 400],["c", 100], ["xc", 90],
                ["l", 50], ["xl", 40], ["x", 10], ["ix", 9], ["v", 5], ["iv", 4], ["i", 1]]

    def to_s
        value = @value
        roman = ""
        for code, factor in FACTORS
            count, value = value.divmod(factor)
            roman << code unless count.zero?
        end
        roman
    end
end
```

romanbug.rb

Unit Testing

Natürlich lassen sich einfache Tests ohne weiteres direkt formulieren:

```
require_relative 'romanbug'

r = Roman.new(1)
if ! (r.to_s == "i" ); fail "'i' expected" end

r = Roman.new(9)
if !(r.to_s == „ix“); fail "'ix' expected“ end
```

Aber mit zunehmender Komplexität der Programme ist es sinnvoll, den für Ruby mitgelieferten Rahmen für *unit tests* zu nutzen.

Moderne ruby-Versionen (> 1.8) verwenden die MiniTest-Umgebung, die die ältere Test::Unit-Umgebung ersetzt, die aber weiterhin auf Wunsch zur Verfügung steht.

Unit Testing

Die ***Ruby-Testumgebung*** besteht auf folgenden drei Teilen:

- ein Standardformat für individuelle Tests;
- verschiedene Möglichkeiten zur Strukturierung von Tests und
- flexible Möglichkeiten, die formulierten Tests aufzurufen.

Statt eine Folge von IF-Anweisungen zu formulieren, kann man einfache Zusicherungen formulieren. Es gibt verschiedene Typen von Zusicherungen, die aber alle ein ähnliches Format aufweisen. Die Standardzusicherung hat die Form

Annahme == erwarteter Wert

```
require_relative 'romanbug'
require 'minitest/autorun'

class TestRoman < MiniTest::Test
    def test_simple
        assert_equal("i", Roman.new(1).to_s)
        assert_equal("ix", Roman.new(9).to_s)
    end
end
```

```
Loaded suite /tmp/prog Started . Finished in 0.000480 seconds. 1 tests, 2
assertions, 0 failures, 0 errors, 0 skips
```

Unit Testing

Jetzt fügen wir ein paar weitere Tests hinzu:

```
require_relative 'romanbug'
require 'minitest/autorun'

class TestRoman < MiniTest::Test
    def test_simple
        assert_equal("i", Roman.new(1).to_s)
        assert_equal("ii", Roman.new(2).to_s)
        assert_equal("iii", Roman.new(3).to_s)
        assert_equal("iv", Roman.new(4).to_s)
        assert_equal("ix", Roman.new(9).to_s)
    end
end

Loaded suite /tmp/prog
Started
Finished in 0.000703 seconds.
    1) Failure:
    <"ii"> expected but was <"i">.
1 tests, 2 assertions, 1 failures, 0 errors, 0 skips
test_simple(TestRoman) [/tmp/prog.rb:6]:
```

Unit Testing

```
def to_s
    value = @value
    roman = ""
    for code, factor in FACTORS
        count, value = value.divmod(factor)
        roman << (code * count)
    end
    roman
end
```

```
require_relative 'roman3'
require 'minitest/autorun'
class TestRoman < MiniTest::Test
    def test_simple
        assert_equal("i", Roman.new(1).to_s)
        assert_equal("ii", Roman.new(2).to_s)
        assert_equal("iii", Roman.new(3).to_s)
        assert_equal("iv", Roman.new(4).to_s)
        assert_equal("ix", Roman.new(9).to_s)
    end
end

Loaded suite /tmp/prog
Started
.
Finished in 0.000689 seconds.
1 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

Unit Testing

Es ist möglich, die Tests kompakter zu formulieren:

```
require_relative 'roman3'
require 'minitest/autorun'

class TestRoman < MiniTest::Test


    NUMBERS = { 1 => "i", 2 => "ii", 3 => "iii", 4 => "iv", 5 => "v", 9 => "ix" }


    def test_simple
        NUMBERS.each do |arabic, roman|
            r = Roman.new(arabic)
            assert_equal(roman, r.to_s)
        end
    end
end

Loaded suite /tmp/prog
Started
.
Finished in 0.000522 seconds.
1 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

Unit Testing

```
require_relative 'roman3'
require 'minitest/autorun'

class TestRoman < MiniTest::Test

    NUMBERS = { 1 => "i", 2 => "ii", 3 => "iii", 4 => "iv", 5 => "v", 9 => "ix" }

    def test_simple NUMBERS.each do |arabic, roman|
    ...
    end
    def test_range
        Roman.new(1)
        Roman.new(4999)
        assert_raises(RuntimeError) { Roman.new(0) }
        assert_raises(RuntimeError) { Roman.new(5000) }
    end
end

Loaded suite /tmp/prog
Started
..
Finished in 0.000767 seconds.
2 tests, 8 assertions, 0 failures, 0 errors, 0 skips
```

Unit Testing

Strukturierung von Tests

Anhand der Beispiele auf den vorangegangenen Folien haben wir eine Reihe von Dingen bereits kennengelernt:

1. Die Umgebung zur Durchführung von unit tests wird durch
   require ‘test/unit‘ bzw.
   require ‘mintest/autorun‘
   zur Verfügung gestellt.

2. Es wird unterschieden zwischen *Test-Szenarien* und *Test-Methoden*.

4. Alle Test-Methoden eines Test-Szenarios gehören einer Klasse an, die eine Unterklasse der Klasse MiniTest::Test ist.

5. Die Name der Methoden einer Testklasse müssen alle mit test beginnen.

Oft überprüfen alle Testmethoden einer Testklasse verschiedene Aspekte eines gemeinsamen Szenarios. Grundsätzlich kann es dabei nötig sein, das jede Testmethode bestimmte Aufräumarbeiten umsetzt.

Unit Testing

```
class TestPlaylistBuilder < MiniTest:Test

def test_empty_playlist
    db = DBI.connect('DBI:mysql:playlists')
    pb = PlaylistBuilder.new(db)
    assert_empty(pb.playlist)
    db.disconnect
end

def test_artist_playlist
    db = DBI.connect('DBI:mysql:playlists')
    pb = PlaylistBuilder.new(db)
    pb.include_artist("krauss")
    refute_empty(pb.playlist, "Playlist shouldn't be empty")
    pb.playlist.each do |entry|
        assert_match(/krauss/i, entry.artist)
    end
    db.disconnect
end
# ...
end
```

Unit Testing

In allen Methoden wird am Anfang eine Verbindung zur Datenbank hergestellt, die am Ende wieder getrennt wird. Diese sich wiederholenden Aufgaben können in eine setup- bzw. teardown-Methode ausgelagert werden.
Wenn eine Testklasse Methoden mit diesen Namen enthält, dann wird die setup-Methode vor und die teardown-Methode nach allen Tests ausgeführt.

```
class TestPlaylistBuilder < MiniTest::Test
    def setup
        @db = DBI.connect('DBI:mysql:playlists') @pb =
        PlaylistBuilder.new(@db)
end

def teardown
    @db.disconnect
end

def test_empty_playlist
    assert_empty(@pb.playlist)
end
# ...
```

Unit Testing

Organisieren und Durchführen von Tests

Alle bisher betrachteten Tests waren ausführbare Test::Unit-Programme. Diese Tests können von der Kommandozeile aufgerufen werden:

```
$ ruby test_roman.rb
  Loaded suite test_roman
  Started
  ..
  Finished in 0.000792 seconds.
  2 tests, 8 assertions, 0 failures, 0 errors, 0 skips
```

```
$ ruby test_roman.rb -n test_range
  Loaded suite test_roman
  Started
  .
  Finished in 0.000617 seconds.
  1 tests, 2 assertions, 0 failures, 0 errors, 0 skips--name "test_range"
```

```
$ ruby test_roman.rb -n /range/
  Loaded suite test_roman
  Started
  .
  Finished in 0.000632 seconds.
  1 tests, 2 assertions, 0 failures, 0 errors, 0 skips--name "/range/"
```

Unit Testing

Bei größeren Projekten ist es sinnvoll, mehrere Verzeichnisse anzulegen, in denen die Dateien organisiert werden. So ist es in diesem Fall auch sinnvoll, ein separates Verzeichnis für alle Testdateien anzulegen:

```
roman/
    lib/
        roman.rb
        weitere Dateien...
    test/
        test_roman.rb
        andere Tests...

    weitere Dateien...
```

Unit Testing

```
assert(test, msg = nil)
assert_empty(obj, msg = nil)
assert_equal(exp, act, msg = nil)
assert_includes(collection, obj, msg = nil)
assert_instance_of(cls, obj, msg = nil)
assert_kind_of(cls, obj, msg = nil)
assert_match(matcher, obj, msg = nil)
assert_nil(obj, msg = nil)
assert_raises(*exp) { || ... }
assert_throw(sym, msg = nil) { || ... }
```

Quelle: http://docs.seattlerb.org/minitest/Minitest/Assertions.html