Wenn Sie nach Lisp schauen…

Eine Rundreise durch Clojure

Meikel Brandmeyer

JUG Darmstadt
17.05.2011

Plan A

0. Meta
1. Feldherrenhügel
2. Syntax und Reader
3. Datenstrukturen
4. Sequenzen
5. Nebenläufigkeit
6. Java Interop
7. Clojure-flavored OOP
8. Bücher

Ablauf

* Die Folien sind „Plan A“.
  * Gewürzt mit Beispielen
* Wer will kann mitmachen!
* Wieso? Weshalb? Warum? Wer nicht fragt, bleibt dumm!

Mitmachen!

```
user=> (load-file "jugda.clj")
nil
user=>
```

Kurz zu mir

* Diplom-Mathematiker (TU KL) im Dienste von Continental in Frankfurt
* Clojure-Nutzer seit Frühjahr 2008
* Autor von VimClojure und clojuresque (gradle)
* Gelegentlicher Contributor

Plan A

0. Meta
1. **Feldherrenhügel**
2. Syntax und Reader
3. Datenstrukturen
4. Sequenzen
5. Nebenläufigkeit
6. Java Interop
7. Clojure-flavored OOP
8. Bücher

Was ist Clojure?

Clojure ist…

* ein Lisp.

* funktional.

* dynamisch.

* durch Meinungen geprägt.

* neu. (Erste Veröffentlichung im Oktober 2007)

Wieso Clojure?

* kompakter Code: Viel *BOOM* für den €

* REPL (*R* ead *E* val *P* rint *L* oop):
  schneller Entwicklungszyklus

* Konsistenz!

* Einfacher Zugang zur JVM

  * schnell, Infrastruktur, Unmengen an Bibliotheken, etc.

* Community!

Clojures Community

* *Einfach Klasse!*

* Google groups:

  * https://groups.google.com/forum/#!forum/clojure
  * https://groups.google.com/forum/#!forum/clojure-de

* IRC: #clojure und #clojure.de auf freenode

* Bunt zusammengewürfelt: Java, Haskell, Python, …

Editoren und IDEs

* Die Veteranen: Emacs und Vim

* Die Schwergewichte: Eclipse, Netbeans, IntelliJ
* Die Exoten: Textmate, jEdit, Bluefish, Ace, …

Build Tools

* Old school: ant+ivy, maven
* Die Newcomer: gradle, (Apache Buildr)
* Bau-in-der-Sprache: leiningen, cake

Plan A

0. Meta
1. Feldherrenhügel
2. **Syntax und Reader**
3. Datenstrukturen
4. Sequenzen
5. Nebenläufigkeit
6. Java Interop
7. Clojure-flavored OOP
8. Bücher

Kurzer Syntaxguide

* nil ⟶ null
* "Hallo" ⟶ String
* true/false ⟶ Boolean
* 1 ⟶ Long
* 1.0 ⟶ Double
* 1.0M ⟶ BigDecimal
* 1/2 ⟶ Ratio
* :abc ⟶ Keyword
* abc ⟶ Symbol

```
                    * []         ⟶ Vector
                    * {}         ⟶ Map
                    * #{}        ⟶ Set
                    * ()         ⟶ List
```

```
                          # Das war's. Wirklich! #

          * Clojure ist [homoikonisch][1].

          * Beispiel:

            Funktionsdefinition
               │
               │      Funktionsargumente
               │         │
               ▼         ▼
               (defn sum [coll]
                 (reduce + 0 coll))
                 ▲   ▲        ▲
                 │   │        │
                 │ Funktion  Argumente:
                 │           Globals, Locals, Literale
          Funktionsaufruf

          [1]: http://de.wikipedia.org/wiki/Homoikonizität
```

```
                          # Das war's. Wirklich! #

          * Clojure ist [homoikonisch][1].

          * Beispiel:

             List
               │
               │       Vector
               │         │
               ▼         ▼
               (defn sum [coll]
                 (reduce + 0 coll))
                 ▲   ▲        ▲
                 │   │        │
                 │  Symbol   Argumente:
                 │           Symbole, Literale, Listen
                  List

          [1]: http://de.wikipedia.org/wiki/Homoikonizität
```

```
Repl Transcript:
talk.presentation573=> (reduce + 0 [1 2 3])
6
```

```
                              # Der Reader #

          * Übersetzt Quelltext in Clojure-Datenstrukturen

          * Gibt dann die *Datenstruktur* an den Compiler

          * Vorteil: Einfache Schnittstelle zum Compiler
```

⟹ Macros

Makros

* eigentlich simpel: Funktion, die Code modifiziert
  * aber nicht einfach: symbol capture, etc.
* Kann komplett Clojure für Transformation benutzen

```
(defmacro with-file
  [[local file] & body]
  `(let [~local (FileReader. ~file)]
     (try
       ~@body
       (finally
         (.close ~local)))))
```

```
Repl Transcript:
talk.presentation573=> `(~[1 2 3])
([1 2 3])

talk.presentation573=> `(~@[1 2 3])
(1 2 3)
talk.presentation573=> (defn with-file*
                         [fname thunk]
                         (let [rdr (FileReader. fname)]
                           (try
                             (thunk rdr)
                             (finally
                               (.close rdr)))))
#'talk.presentation573/with-file*
talk.presentation573=> (defmacro with-file2
                         [[local fname] & body]
                         `(with-file* ~fname (fn [~local] ~@body)))
#'talk.presentation573/with-file2
```

Makros

```
(with-file [rdr (File. "abc.txt")]
  (do-something-with rdr)
  (do-more-with rdr))
```

wird zu

```
(let [rdr (FileReader. (File. "abc.txt"))]
  (try
    (do-something-with rdr)
    (do-more-with rdr)
    (finally
      (.close rdr))))
```

Plan A

0. Meta
1. Feldherrenhügel
2. Syntax und Reader
3. **Datenstrukturen**
4. Sequenzen
5. Nebenläufigkeit
6. Java Interop
7. Clojure-flavored OOP
8. Bücher

Persistenz

* Alle Datenstrukturen sind persistent.
* „Modifikation“ gibt neue Kopie.
  * Implementierung ist nicht naiv: Bäume!
* Original bleibt erhalten.

```
Repl Transcript:
talk.presentation573=> v
[1 2 3]
talk.presentation573=> (conj v 4)
[1 2 3 4]
talk.presentation573=> v
[1 2 3]
```

Einheitliche Operationen

* Sind immer das Beste für die Struktur.
* sequentiell: conj, peek, pop, nth
* assoziativ: assoc, dissoc, get, contains?, (conj)
* Set: conj, disj, contains?

Sequentiell

List und Vector sind sequentielle Datenstrukturen.

* `conj`: fügt Element hinzu
* `peek`: schaut „nächstes“ Element an
* `pop`: gibt „restliche“ Elemente zurück
* `nth`: gibt das n-te Element zurück

```
Repl Transcript:
talk.presentation573=> v
[1 2 3]
talk.presentation573=> (conj v 4)
[1 2 3 4]
talk.presentation573=> l
(1 2 3)
talk.presentation573=> (conj l 4)
(4 1 2 3)
talk.presentation573=> (peek v)
3
talk.presentation573=> (peek l)
1
talk.presentation573=> (pop v)
[1 2]
talk.presentation573=> (pop l)
(2 3)
```

Assoziativ

* `assoc`: assoziert einen Wert mit einem Schlüssel
  * bei einem Vector muss der Schlüssel innerhalb der Länge+1 liegen
* `dissoc`: entfernt einen Schlüssel mitsamt Wert
* `get`: gibt den Wert zu einem Schlüssel zurück
* `contains?`: prüft, ob ein Schlüssel enthalten ist
  * *nicht* der Wert!
* Funktionsaufruf: gibt den Wert zu einem Schlüssel zurück

```
Repl Transcript:
talk.presentation573=> v
[1 2 3]
talk.presentation573=> (assoc v 1 4)
[1 4 3]
talk.presentation573=> v
[1 2 3]
```

```
talk.presentation573=> m
{:a 1, :c 3, :b 2}
talk.presentation573=> (m :a)
1
talk.presentation573=> (m :d)
nil
talk.presentation573=> (map m [:a :b])
(1 2)
talk.presentation573=> (contains? [4 5 6] 1)
true
talk.presentation573=> (contains? [4 5 6] 5)
false
talk.presentation573=> (map (fn [k] (get m k)) [:a :b])
(1 2)
talk.presentation573=> (m [:a :b])
nil
talk.presentation573=> m
{:a 1, :c 3, :b 2}
talk.presentation573=> ({[:a :b] :c} [:a :b])
:c
```

Set

* `conj`: fügt ein Element hinzu
* `disj`: entfernt ein Element
* `contains?`: prüft, ob ein Element enthalten ist
* Funktionsaufruf: gibt Element zurück, wenn enthalten; sonst `nil`

```
Repl Transcript:
talk.presentation573=> s
#{1 2 3}
talk.presentation573=> (conj s 4)
#{1 2 3 4}
talk.presentation573=> (conj s 2)
#{1 2 3}
talk.presentation573=> (disj s 2)
#{1 3}
talk.presentation573=> s
#{1 2 3}
talk.presentation573=> m
{:a 1, :c 3, :b 2}
```

Plan A

0. Meta
1. Feldherrenhügel
2. Syntax und Reader
3. Datenstrukturen
4. **Sequenzen**
5. Nebenläufigkeit
6. Java Interop
7. Clojure-flavored OOP

8. Bücher

Sequenzen

* Sequentielle Sicht auf eine Menge
* Prototyp: einfach verlinkte Liste

  `1` ⟶ `2` ⟶ `3` ⟶ `nil`

* Entweder: ein Element mit einem Rest
  * Clojure Interface: `first` und `next`
* Oder: nichts, kein Objekt – `nil`
  * Insbesondere gibt es keine „leere“ Sequenz!

Sequenzen

* Sequentielle Sicht auf eine Menge
* Prototyp: einfach verlinkte Liste

  `1` ⟶ `2` ⟶ `3` ⟶ `nil`

* Entweder: ein Element mit einem Rest
  * Clojure Interface: `first` und `next`
* Oder: nichts, kein Objekt – `nil`
  * Insbesondere gibt es keine „leere“ Sequenz!
* Wer sagt, dass obiges Beispiel nicht `(seq #{1 2 3})` ist?

Zwei Welten

Wichtig: Sequenzen sind *keine* Datenstrukturen!

* Wer sagt denn, dass ⟶ das nächste Element ist?
* Kann auch eine Vorschrift sein!
* Wird nur auf Bedarf realisiert – „lazy seq“
  * Clojure Interface: `rest`, `seq`

Die natürlichen Zahlen

* Beispiel: `(iterate inc 0)` – die natürlichen Zahlen mit 0

  `0` ⟶ `(inc 0)` ⟶ `(inc (inc 0))` ⟶ …

  `0` ⟶ `1` ⟶ `2` ⟶ …

Die natürlichen Zahlen

* Beispiel: `(iterate inc 0)` – die natürlichen Zahlen mit 0

  `0` ⟶ `(inc 0)` ⟶ `(inc (inc 0))` ⟶ …

  `0` ⟶ `1` ⟶ `2` ⟶ …

* Eine „unendliche“ Sequenz

Sequenzbibliothek

* `map`: Wende Funktion auf jedes Element an
* `filter`/`remove`: Filtere Elemente gemäß eines Prädikates
* `reduce`: Reduzierung auf einen Wert durch eine Funktion
* vieles mehr…

Repl Transcript:

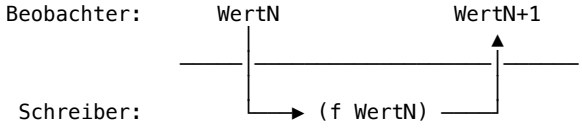
```
talk.presentation573=> (def n (iterate inc 0))
#'talk.presentation573/n
talk.presentation573=> (set! *print-length* 50)
50
talk.presentation573=> (take 10 n)
(0 1 2 3 4 5 6 7 8 9)
talk.presentation573=> (take 10 (filter even? n))
(0 2 4 6 8 10 12 14 16 18)
```

Plan A

0. Meta
1. Feldherrenhügel
2. Syntax und Reader
3. Datenstrukturen
4. Sequenzen
5. **Nebenläufigkeit**
6. Java Interop
7. Clojure-flavored OOP
8. Bücher

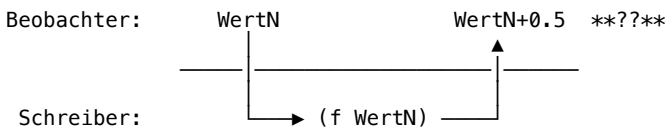
Clojures Verständnis von Zeit

Eine Identität ist eine Reihe von Werten über der Zeit.



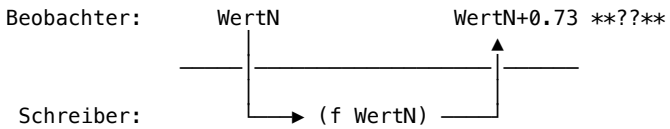
Clojures Verständnis von Zeit

Eine Identität ist eine Reihe von Werten über der Zeit.



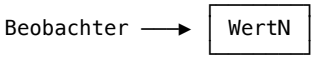
Clojures Verständnis von Zeit

Eine Identität ist eine Reihe von Werten über der Zeit.



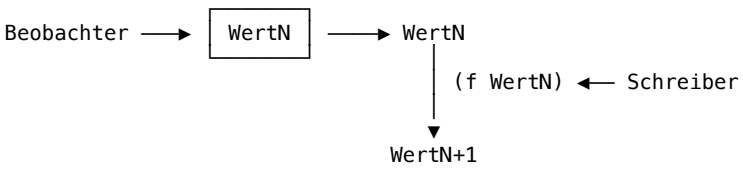
Clojures Antwort: Referenztypen

Einsicht: Man kann die Welt nicht anhalten!



Clojures Antwort: Referenztypen

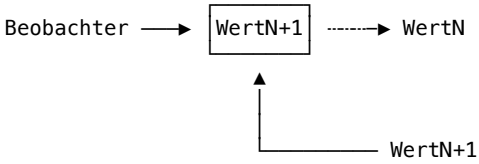
Einsicht: Man kann die Welt nicht anhalten!



* Schreiber erzeugt vermöge f aus WertN den WertN+1.
* Beobachter sieht derweil weiterhin WertN.

Clojures Antwort: Referenztypen

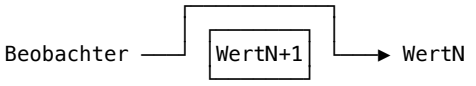
Einsicht: Man kann die Welt nicht anhalten!



* Schreiber erzeugt vermöge f aus WertN den WertN+1.
* Beobachter sieht derweil weiterhin WertN.
* Schreiber macht WertN+1 atomar sichtbar für Beobachter.

Clojures Antwort: Referenztypen

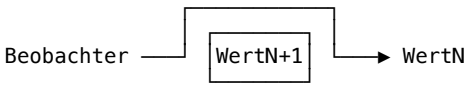
Einsicht: Man kann die Welt nicht anhalten!



* Frage: Was passiert, wenn der Beobachter eine Referenz auf den WertN behält?

Clojures Antwort: Referenztypen

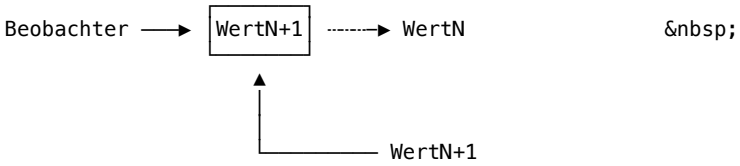
Einsicht: Man kann die Welt nicht anhalten!



* Frage: Was passiert, wenn der Beobachter eine Referenz auf den WertN behält?
* Antwort: Nichts schlimmes! Persistenz!

Clojures Antwort: Referenztypen

Einsicht: Man kann die Welt nicht anhalten!



* Schreiber erzeugt vermöge f aus WertN den WertN+1.
* Beobachter sieht derweil weiterhin WertN.
* Schreiber macht WertN+1 atomar sichtbar für Beobachter.

Clojures Antwort: Referenztypen

Einsicht: Es gibt kein One-size-fits-all!

* Atom: schnell, simple, aber unkoordiniert
* Ref: koordiniert, Transaktionen, aber STM Overhead
* Agent: asynchron, seriell, aber unkoordiniert
* (Var): thread-lokal, selten (direkt) benutzt
* (Pods): noch nichts genaues bekannt

```
Repl Transcript:
talk.presentation573=> (norad atom)
#'talk.presentation573/rockets
talk.presentation573=> defcon
#<Atom@320399: 1>
talk.presentation573=> rockets
#<Atom@a1c582: :hold>
talk.presentation573=> (code start-rockets-example)
(defn start-rockets-example
  [starter-fn stopper-fn]
  (let [starter (future-call starter-fn)]
    (wait 4)
    (println @defcon @rockets)
    (stopper-fn)
    (println @defcon @rockets)
    @starter
    (println @defcon @rockets)))
nil
talk.presentation573=> (code start-rockets-a)
(defn start-rockets-a
  []
  (when (and (= @defcon 1) (start-codes-valid?))
    (swap! defcon dec)
    (reset! rockets :fire)))
nil
talk.presentation573=> (code stop-doomsday-a)
(defn stop-doomsday-a
  []
  (reset! defcon 2))
nil
talk.presentation573=> (start-rockets-example start-rockets-a stop-doomsday-a)
```

```
1 :hold
2 :hold
1 :fire
nil
talk.presentation573=> (code start-rockets-r)
(defn start-rockets-r
  []
  (dosync
    (when (and (= @defcon 1) (start-codes-valid?))
      (alter defcon dec)
      (ref-set rockets :fire))))
nil
talk.presentation573=> (norad ref)
#'talk.presentation573/rockets
talk.presentation573=> (start-rockets-example start-rockets-r stop-doomsday-r)
1 :hold
2 :hold
2 :hold
nil
talk.presentation573=> (let [a (atom 0)] (swap! a + 2))
2
talk.presentation573=> (let [a (ref 0)] (dosync (alter a + 2)))
2
```

Uneiheitliche Interfaces

* `swap!` vs. `alter` vs. `send`
* Gewollt! Kein One-size-fits-all!
* Aber immer das gleiche Prinzip!

    (swap! an-atom f arg1 arg2) ⟶ (f valueN arg1 arg2)

* Konsistenz!

Sonstiges

* `future`: Verarbeitung im Hintergrund
* `promise`: blockiert, bis Ergebnis geliefert wird
* `pmap`: paralleles `map`
* Java ist nicht weit weg: java.util.concurrent
* `locking`, wenn's denn sein muss

Plan A

0. Meta
1. Feldherrenhügel
2. Syntax und Reader
3. Datenstrukturen
4. Sequenzen
5. Nebenläufigkeit
6. **Java Interop**
7. Clojure-flavored OOP
8. Bücher

Der Gastgeber: die JVM

* Klar erkennbar, zB. `"Hallo"` ist ein Java String
* Einfach erreichbar:
  * (.startsWith "Hallo" "Ha") ⟹ true
  * (java.util.Date.) ⟹ #<Date Mon May 16 21:49:54 CEST 2011>

Konterbande

* `reify`: implementiert anonyme Klasse mit Interfaces
  + schnell
  - kann nur Object als Superklasse haben
  - kann keine Methoden hinzufügen
* `proxy`: implementiert anonyme Klasse mit Interfaces
  + kann beliebige Superklasse haben
  + kann per Instanz Methoden ändern
  - langsamer als `reify`
  - kann keine Methoden hinzufügen

Moonshine

* `gen-class`: implementiert Klasse

    + kann beliebige Superklasse haben
    + kann beliebige Methoden hinzufügen
    - benötigt AOT-Compilierung

* `gen-interface`: implementiert Interface

    - benötigt AOT-Compilierung

```
Repl Transcript:
talk.presentation573=> (defprotocol Flu (cough [this]))
Flu
talk.presentation573=> (reify Flu (cough [this] "*cough*"))
#<presentation573$eval740$reify__741 talk.presentation573$eval740$reify__741@4f8358>
talk.presentation573=> (cough *1)
"*cough*"
```

Plan A

0. Meta
1. Feldherrenhügel
2. Syntax und Reader
3. Datenstrukturen
4. Sequenzen
5. Nebenläufigkeit
6. Java Interop
7. **Clojure-flavored OOP**
8. Bücher

„Klassen“

* `defrecord`: definiert eine „Datenklasse“

    * bringt Map verhalten mit
    * Zugriff für Felder schneller als eine Map
    * zur Implementierung von Domainobjekten
    * kann Interfaces implementieren
    * es gibt aber keine Vererbung

* `deftype`: definiert eine „Hintergrundklasse“

    * Barebone
    * zur Implementierung von Datenstrukturen
    * kann veränderliche/primitive Felder haben
    * kann Interfaces implementieren
    * es gibt aber keine Vererbung

Protokolle

* Interface on steroids!

* sehen aus wie Funktionen

* Können für bestehende Klassen nachgerüstet werden

  * lösen das „Expression“-Problem

* Bieten ein Interface für Java-Seite

```
Repl Transcript:
talk.presentation573=> (extend-protocol Flu String (cough [this] (str "*cough* " this "
*cough*")))
nil
talk.presentation573=> (cough 1)
talk.presentation573=> (cough "Hallo")
"*cough* Hallo *cough*"
talk.presentation573=> (extend-protocol Flu Long (cough [this] (apply str (repeat this
"*cough*"))))
nil
talk.presentation573=> (cough 5)
"*cough**cough**cough**cough**cough*"
talk.presentation573=> (defmulti foo (fn [_x y] (type y)))
#'talk.presentation573/foo
talk.presentation573=> (defmethod foo String [x y] (str x "->" y))
#<MultiFn clojure.lang.MultiFn@ad97f5>
talk.presentation573=> (defmethod foo Number [x y] (+ x y))
#<MultiFn clojure.lang.MultiFn@ad97f5>
talk.presentation573=> (foo 1 "Hallo")
"1->Hallo"
talk.presentation573=> (foo 1 2)
3
talk.presentation573=> (derive String ::foo)
nil
talk.presentation573=> (isa? String ::foo)
true
talk.presentation573=> (derive ::foo ::bar)
nil
talk.presentation573=> (isa? String ::bar)
true
talk.presentation573=> (defmethod foo ::baz [x y] nil)
#<MultiFn clojure.lang.MultiFn@ad97f5>
talk.presentation573=> (derive clojure.lang.Keyword ::baz)
nil
talk.presentation573=> (foo 3 :a)
nil

-----------
```

Plan A

0. Meta
1. Feldherrenhügel
2. Syntax und Reader
3. Datenstrukturen
4. Sequenzen
5. Nebenläufigkeit
6. Java Interop
7. Clojure-flavored OOP
8. **Bücher**

Bücher

* Stefan Kamphausen, Tim Oliver Kaiser:
  „Clojure“, dpunkt.verlag

* Stuart Halloway:
  „Programming Clojure“

* Amit Rathore:
  „Clojure in Action“, Manning

* Chris Houser, Michael Fogus:
  „Joy of Clojure“, Manning

* Luke van der Hart, Stuart Sierra:
  „Practical Clojure“

Fin
